Karlsruher Institut für Technologie

Bachelor thesis

Distributed Kernelization for Independent Sets

Tom George

Date: 29.11.2018

Supervisors: Prof. Dr. Peter Sanders
M.Sc. Demian Hespe
M.Sc. Sebastian Lamm

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Graphs are mathematical structures used to model networks. Entities in the network are associated with vertices in a graph. Relationships between entities are described by edges connecting vertices. A lot of different problems from various domains can be transformed to problems on graphs. They can then be solved using graph theory. In this thesis we tackle the NP-hard *maximum independent set* problem.

An independent set is a set of vertices such that pairwise vertices in it are not connected by an edge. The maximum independent set for a graph looks for the largest cardinality independent set. It is used to solve problems from various domains, like biology or computer vision.

Kernelization is a technique to reduce the complexity of a problem by computing a *kernel* on it. A kernel is *exact* if solving the problem on the original instance yields the same result as solving it on the kernel. Exact algorithms for the maximum independent set problem use the *branch-and-reduce paradigm*. They shrink the problem instance by computing its kernel and branch on vertices, if the kernel is minimal. Successive branching and reducing eventually leads to a base case and the independent set size can be determined. Branching-and-reducing an instance can be represented as a rooted tree, where a leaf is the base case. There exists one path from leaf to root, that results in a independent set with the maximum cardinality.

Related is the *maximal independent set* problem. A maximal independent set for a graph is an independent set, whose cardinality can not be increased by adding further vertices to it. Due to the complexity of the maximum independent set problem it is intractable for most instances. The computation of a maximal independent set can be used as an approximation for the maximum independent set. Different approaches have been developed to improve that approximation. A technique called *local search* transforms a maximal independent set into a different maximal independent set with larger cardinality.

Growing problem instances add another problem: They might be too big to fit onto one machine. Parallelism and distributed computing can be used to tackle this problem.

In this thesis we propose a distributed algorithm using exact kernelization before computing a high quality maximal independent set. Our kernelization removes degree one and two vertices to compute a kernel. We prove the exactness of the kernelization and evaluate the impact on the quality of the maximal independent set. Our results indicate that it is a promising technique increasing the cardinality of the maximal independent sets up to 12 percent. We achieve relative speedup up to 30 for the kernelization and maximal independent set algorithm on 64 processor elements.

Zusammenfassung

Graphen sind mathematische Strukturen zur Modellierung von Netzwerken. Objekte in Netzwerken werden mit Knoten in einem Graph assoziiert. Beziehungen zwischen Objekten im Netzwerk werden durch Kanten zwischen den Knoten beschrieben. Viele Probleme aus diversen Domänen lassen sich auf Probleme auf Graphen transformieren. Graphentheorie kann dann benutzt werden, um diese zu lösen. In dieser Arbeit behandeln wird das NP-harte Problem der *größten unabhängigen Menge*.

Eine unabhängige Menge ist eine Menge von Knoten, sodass paarweise Elemente daraus nicht mit einer Kante verbunden sind. Die größte unabhängige Menge für einen Graphen ist die unabhängige Menge mit der größten Mächtigkeit. Das Problem löst diverse Probleme unterschiedlicher Domänen, beispielsweise in der Biologie oder Computer Vision.

Kernfindung ist eine Technik zur Komplexitätsreduktion einer Probleminstanz. Dazu wird der *Kern* der Probleminstanz berechnet. Ein Kern ist *exakt*, falls die Lösung des Problem auf Kern und ursprünglicher Probleminstanz dasselbe Resultat ergeben. Exakte Algorithmen zur Berechnung der größten unabhängigen Menge benutzen das *branch-and-reduce* Paradigma. Sie verkleinern die Probleminstanz durch Kernfindung und verzweigen an Knoten wenn der Kern minimal ist. Wiederholtes Verkleinern und Verzweigen führt zu einem Basisfall, der es ermöglicht die Mächtigkeit der unabhängigen Menge zu ermitteln. Das Verzweigen-und-Verkleinern der Probleminstanz kann als gewurzelter Baum repräsentiert werden. Ein Blatt repräsentiert den Basisfall. Es existiert ein Pfad von Blatt zu Wurzel, der in einer unabhängigen Menge mit größter Mächtigkeit resultiert.

Verwandt ist das Problem der *maximalen unabhängigen Menge*. Die Mächtigkeit einer maximalen unabhängigen Menge kann nicht durch das Hinzufügen weiterer Knoten vergrößert werden. Aufgrund der Komplexität des größte unabhängige Menge Problem ist dieses unlösbar auf den meisten Instanzen. Die Berechnung einer maximalen unabhängigen Menge kann als Approximation für die größte unabhängige Menge verwendet werden. Verschiedene Ansätze verbessern diese Approximation. Eine Technik namens *lokale Suche* transformiert eine maximal unabhängige Menge in eine andere maximal unabhängige Menge mit größer Mächtigkeit.

Wachsende Probleminstanzen erzeugen ein neues Problem: Sie passen eventuell nicht in den Speicher einer Maschine. Parallelismus und verteiltes Rechnen kann zur Lösung dieses Problems verwendet werden.

In dieser Thesis stellen wir einen verteilten Algorithmus vor, der exakte Kernfindung benutzt um hochqualitative, maximale unabhängige Mengen zu berechnen. Unsere Kernfindung entfernt Knoten mit Grad eins und zwei, um einen Kern zu finden. Wir zeigen, dass unsere Kernfindung exakt ist und evaluieren ihren Einfluss auf die Qualität unserer maximalen unabhängigen Menge. Unsere Ergebnisse zeigen, dass Kernfindung eine vielversprechende Technik ist, welche die Größe der maximalen unabhängigen Menge um bis zu 10 % erhöht. Wir erreichen relativen Speedup von bis zu 30 für die Kernfindung und den Algorithmus zur Berechnung der maximalen unabhängigen Menge auf 64 Prozessorelementen.

Acknowledgments

I'd like to thank my supervisors Demian Hespe and Sebastian Lamm for their guidance during my thesis. Also I'd like to thank Prof. Sanders for the opportunity to work on this thesis. Last, I'd like to thank my friends and family for their love and support.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

Contents

1 Introduction

1.1 Motivation

The *maximum independent set* problem is a well studied NP-hard problem [19]. For a given graph the maximum independent set problem looks for the largest cardinality subset of vertices such that elements are pairwise non-adjacent.

A related problem is the *maximal independent set* (MIS) problem, which looks for an independent set whose cardinality can not be enhanced by adding further vertices to it. The maximal independent set problem can be solved in polynomial time.

Its applications cover biology [10], coding theory [11], computer vision [14], route planing [27] and analysis of social networks [31]. To give a concrete example we look at the *protein docking problem*. For two proteins it finds out whether they interact to form a stable complex and if so how. Graph theory is used to solve this problem [18]. The proteins can be represented as a set of potential hydrogen bond donors and acceptors. A clique-detection (equivalent to the independent set problem) algorithm is used to find maximally complementary sets of donor/ acceptor pairs.

Due to the complexity of the maximum independent set problem, most real world and synthetic instances are intractable. For almost all of them we can only manage to find comparably large independent sets with close to maximum cardinality.

Extensive study has been invested to deal with the complexity of the problem. A promising technique is called *kernelization*. Kernelization looks to reduce the size of the problem instance while maintaining optimality. The initial graph is shrinked by removing vertices with a set of reduction rules. These reduction rules exploit properties of vertices to decide whether or not they are part of the solution. A kernel is the result of exhaustively application of the reduction rules. For a vertex in the kernel it is not trivial to decide whether or not it belongs to the solution. We can say that the kernel contains the part of the problem instance that makes it hard, thus a small kernel is desirable.

To fully utilize modern CPUs and their multiple cores, face the challenge of growing real world instances and improve upon existing (sequential) results we combine kernelization and parallelization to find high quality independent sets. A distributed version further provides an opportunity to deal with instances that do not fit onto one machine.

1.2 Contribution

We propose a distributed algorithm that computes a MIS. As preprocessing step we perform exact kernelization. Our kernel is computed by reducing degree one and two vertices. These reduction rules were original proposed by Chang et al. [12]. Afterwards we execute a state-of-the-art parallel MIS algorithm on the kernel.

Our algorithm is designed to exhaustively perform local work, minimize the rounds of communication and communication overhead.

1.3 Structure of Thesis

In chapter 2 we take a look at the mathematical fundamentals. Also, we shortly survey related work. Chapter 3 describes the reduction rules we use to compute a kernel. We cover the degree one and degree two path reduction. Chapter 4 explains the details of our distributed algorithm. We first describe the kernelization and afterwards the maximal independent set algorithm. Chapter 5 is devoted to the experimental evaluation of our algorithm. In this chapter we examine the speed and scalability of our algorithm as well as the quality of our result. We conclude this thesis with chapter 6, which reviews our result and gives an outlook on further work.

2 Fundamentals and Related Work

In this chapter we take a look at the required preliminaries and precisely formulate our problem. We further define our computation model and how we deal with graphs in a parallel manner. Closing the chapter we shortly survey related work.

2.1 Basic Definitions

A graph $G = (V, E)$ consists of sets V *vertices* and $E \subseteq V \times V$ *edges*. Vertices $v, w \in V$ are called *adjacent*, if there is an edge between v and w and thus $(v, w), (w, v) \in E$. We therefore use unordered pairs $\{u, v\}$ to represent an edge. For a given G, I is called *independent set* if $I \subset V$ and $\forall i_1, i_2 \in I : \{i_1, i_2\} \notin E$.

An independent set I for graph G is called *maximal* if there is no independent set I' of G with $|I'| > |I|$ and $I' \supset I$. It is called *maximum*, if it has maximal cardinality among all independent sets for G.

There might be more than one maximum independent set for a graph G. The cardinality of a maximum independent set for G is called *independence number* $\alpha(G)$.

For $v \in V$, we define the *neighborhood* of v as $\mathcal{N}(v) = \{w \in V : \{v, w\} \in E\}$. Furthermore $\mathcal{N}[v] := \mathcal{N}(v) \cup v$ is called the *closed* neighborhood of v. The cardinality of the neighborhood for a vertex v, or the number of its edges, $d(v) := |\mathcal{N}(v)|$ is called the *degree* of v.

A *path* P is a sequence of distinct vertices $(p_1, \ldots, p_n)$ with p_i and p_{i+1} adjacent for $i \in \{1, \ldots, |P| - 1\}$. Also, if p_1 and $p_{|P|}$ are adjacent and $|P| > 2$ we call P a *cycle*. We call a path (or cycle) P *degree two path* (*degree two cycle*) if $\forall i \in \{1, \ldots, |P|\} : d(p_i) = 2$.

For a degree two path (or cycle) P we define

$$\mathcal{T}(P) := \{u \in V : d(u) \neq 2 \wedge (\{u, p_1\} \in E \vee \{u, p_{|P|}\} \in E)\}$$

as *termination* of P. The following equivalence exists: P is a degree two cycle $\Leftrightarrow \mathcal{T} = \emptyset$.

Further, we define for a degree two *path* P, $\mathcal{E}(P) := \{p_1, p_{|P|}\}$ as *endpoints* of P. The concept of endpoints does not make sense when looking at a degree two cycle C, so we define $\mathcal{E}(C) = \emptyset$. A simple degree two path and its termination as well as endpoints are shown in Figure 2.1.

Our reduction rules remove vertices and insert new edges. For a graph $G = (V, E)$ and a subset V' of V we define $G[V'] = (V', E')$ with $E' = E \cap (V' \times V')$ as the *induced*

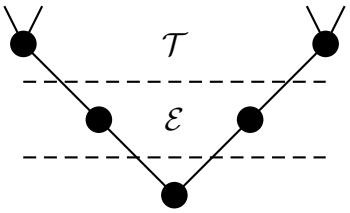


Figure 2.1: Simple degree two path with length three. Termination $\mathcal{T}$ and endpoints $\mathcal{E}$ are marked. Vertices with degree greater than two are represented with two outgoing edges throughout the thesis.

subgraph of V'. Moreover, we define an operation on graphs, called *edge insertion*, as

$$G \oplus \{v, w\} := (V, E \cup \{\{v, w\}\})$$

2.1.1 Related Problems

Closely related to the maximum independent set problem are the *minimum vertex cover* and the *maximum clique* problem. A vertex cover V_C is a subset of vertices, such that every edge of the graph has an endpoint in V_C. The minimum vertex cover problem looks for the smallest cardinality vertex cover. A clique is a subset C of V, such that distinct pairwise elements from C are adjacent. The maximum clique problem looks for the largest cardinality clique.

The minimum vertex cover and maximum independent set problem are equivalent and they are related in the following way. For a maximum independent set I in G, $S = V \backslash I$ is a minimum vertex cover in G.

We can also transform the maximum clique problem into the maximum independent set problem. A maximum clique in the complementary graph $\overline{G} = (V, \overline{E})$ where $\overline{E} = \{\{u, v\} : u, v \in V \wedge \{u, v\} \notin E \wedge u \neq v\}$ is a maximum independent set in G.

2.2 Distributed Memory and Graphs

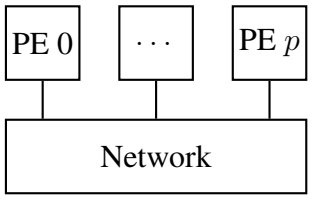


Figure 2.2: Distributed memory computation model

Our algorithm is a parallel, distributed memory implementation. We use a fixed number p of processing elements (PEs) to perform our algorithm. The algorithm is executed p times. Each PE has its own memory. There is no shared memory. Communication is achieved over a network connecting the different PEs. To identify PEs among each other we use an id called *rank*. Figure 2.2 shows a visualization of our computation model.

Each PE holds a part of the graph data. We have to find a way to distribute the graph. For now we use

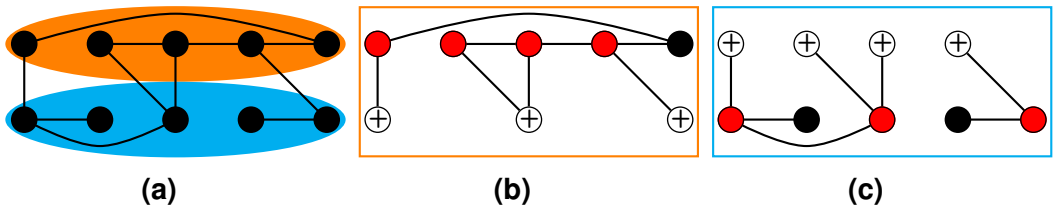


Figure 2.3: A graph with ten vertices is distributed across two PEs. Figure 2.3a shows the graph in its entirety. Throughout this thesis we are going to illustrate PEs with ellipses. Vertices within that ellipse are local vertices to the PE. Figures 2.3b and 2.3c show relevant vertices for the orange and blue PE respectively. Ghost vertices are marked with a plus, while local vertices are bold. Further a interface vertex is marked red.

this simple approach: a graph G is distributed among p PEs such that every PE holds $k = \frac{n}{p}$ vertices. The first PE gets the first k vertices, the second the next k and so forth. If $|V|$ is not divisible by p we have to divide the remainder. The k vertices assigned to a PE are called *local* vertices. For the PE with rank i the set V^i_{local} consists of these vertices. Its incident edges are defined as $E^i = \{\{u, v\} \in E : u \in V^i_{\text{local}}\}$. The set $V^i_{\text{ghost}} := \{v \in V \backslash V^i_{local} : \{u, v\} \in E^i\}$ contains the vertices that are adjacent to local vertices of PE i and are local vertices stored by another PE. These vertices are called *ghost* vertices. A final set $V^i_{\text{interface}} := \{v \in V^i_{\text{local}} : \{u, v\} \in E^i \wedge u \in V^i_{\text{ghost}}\}$ consists of the local vertices that are adjacent to ghost vertices. The elements of it are called *interface* vertices. Overall, the vertices V^i of the graph that are relevant for PE i are a subset of V. They consist of local and ghost vertices and are defined as $V^i := V^i_{\text{local}} \cup V^i_{\text{ghost}}$.

To facilitate talking about distributed graphs, we further define a function $\texttt{PE} : V \to \mathbb{N}_0$, that maps $v \in V$ to the PE having v as a local vertex and indicator functions for the defined sets:

$$\texttt{IsGhost} : (\mathbb{N}_0, V) \to \{0, 1\} \quad \texttt{IsGhost}(i, v) := \begin{cases} 1 & \text{if } v \in V^i_{\text{ghost}} \\ 0 & \text{if } v \notin V^i_{\text{ghost}} \end{cases}$$

$$\texttt{IsInterface} : (\mathbb{N}_0, V) \to \{0, 1\} \quad \texttt{IsInterface}(i, v) := \begin{cases} 1 & \text{if } v \in V^i_{\text{interface}} \\ 0 & \text{if } v \notin V^i_{\text{interface}} \end{cases}$$

$$\texttt{IsLocal} : (\mathbb{N}_0, V) \to \{0, 1\} \quad \texttt{IsLocal}(i, v) := \begin{cases} 1 & \text{if } v \in V^i_{\text{local}} \\ 0 & \text{if } v \notin V^i_{\text{local}} \end{cases}$$

An example of a distributed graph is given in Figure 2.3.

2.2.1 Degree two paths spreading multiple PEs

We call an edge $E = \{u, v\}$ *cut edge* if $\texttt{PE}(u) \neq \texttt{PE}(v)$. A path P is called *distributed*, if between two vertices on it exists a cut edge. Otherwise it is called *local*. We define $\texttt{NCE}(P)$ as the number of cut edges in P.

If P is distributed, it is partitioned into subsequences, that reside on different PEs. To formalize this we write P as a concatenation of subsequences

$$P = \hat{P}^1 \cdot \hat{P}^2 \cdot \ldots \cdot \hat{P}^n$$

We call $\hat{P}^k$ *path segment* of P. Path P is partitioned into $\texttt{NCE}(P) + 1$ path segments. Between successive path segments there exists a cut edge. A path segment either has one or two cut edges.

Figure 2.4a shows an example of a distributed degree two path.

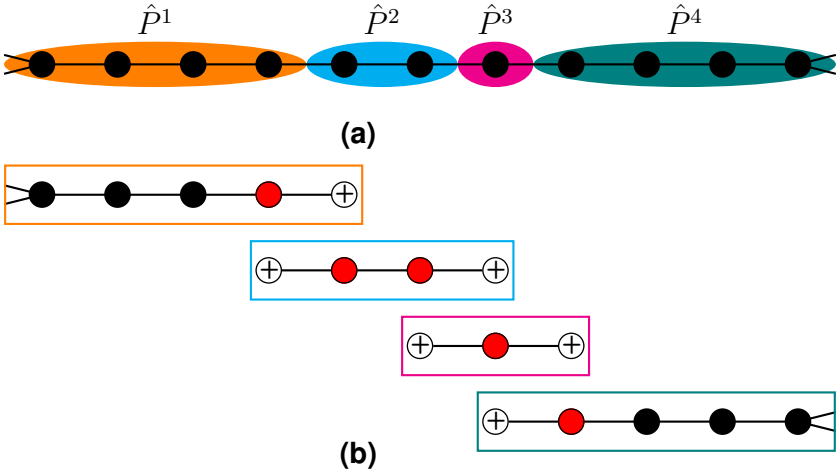


Figure 2.4: (a) Shows the global degree two path with nine vertices distributed over four different PEs. (b) The relevant vertices for each PE. Interface vertices of the path are red while ghost vertices are marked with a plus. The NCE of the path is three.

2.2.2 Graph Partitioning

A graph partition G is the division of V into k roughly equal sized disjoint subsets, such that a cost function is optimized. Further each $v \in V$ needs to be in one subset and therefore:

$$\bigcup_k V_k = V$$

A common example is a partitioning that minimizes the number of cut edges for G.

2.3 Related Work

This section covers related work. We take a look at exact algorithms, heuristic approaches and parallel algorithms for the maximum independent set problem.

2.3.1 Exact Algorithms

A trivial exact algorithm runs in $O(p(x)2^n)$ (where $p(x)$ is some polynomial) by simply checking all 2^n possible subsets for independence. More sophisticated algorithms use the branch-and-reduce paradigm to achieve faster running times [3, 36]. The problem instance is made smaller by applying a set of rules (reducing), before eventually no more rules can be applied and a branch is performed creating two problem instances. Branching is repeatedly done until a base case is reached.

Lots of studies have been invested to improve the base of exact algorithms over the years. One of the first results was proposed by Tarjan and Trojanowski [35], who proved complexity $O(2^{\frac{1}{3}n})$. The currently best known exact algorithm was proposed by Xiao and Nagamochi [36] and has a time complexity of $O(1.1996^n)$. It was analyzed using a technique named *measure and conquer* [16], which uses advanced measure to tighten the worst case complexity bound.

Akiba and Iwata [3] developed an exact branch-and-reduce algorithm for Vertex Cover using a variety of reduction rules and showed their practicality. Their result indicates that kernelization is a useful technique for developing fast algorithms.

2.3.2 Approximation for Maximum Independent Set

Due to the hardness of the maximum independent set problem other approaches have been developed to calculate high quality maximal independent sets. One heuristic technique is called local search [32, 22, 6, 13]. Local search iteratively improves a result using simple operations like vertex insertion, deletion or swap. A technique called *plateau search* only allows operations that do not change the value of an objective function.

Andrade et al. [32] proposed an algorithm (ARW) with linear time complexity to determine whether a solution can be improved performing (1,2)-swaps. A (1,2)-swap replaces one vertex in the solution with two others. By iteratively applying this algorithm to an initial solution and the use of suitable data structures, they achieved linear time complexity for their iterated local search algorithm.

Dahlum et al. [13] developed an algorithm combining kernalization and local search. They compute an exact kernel by applying reduction rules in an online fashion. Additionally they show that cutting a part of high degree vertices accelerates local search while maintaining high quality solutions.

Chang et al. [12] proposed algorithms with linear and near-linear time complexity. Their linear time algorithm uses exact reductions to remove degree one and two vertices from the graph. The near-linear time algorithm additionally applies the dominance rule [16]. By applying the dominance rule $u \in V$ can be reduced, if an adjacent vertex v exists, such that every neighbor from v other than u is also connected to u. They showed that ARW can be accelerated by finding a large initial solution using their algorithm. The initial solution is

found by iteratively applying exact reduction rules and cutting high degree vertices when no more reductions can be applied.

2.3.3 Parallel Algorithms for Maximal Independent Set

Maximal independent set algorithms have been proposed for shared [20, 26, 21, 4, 7] as well as distributed memory [25, 15, 24, 34].

One of the most famous parallel algorithm for maximal independent set was proposed by Luby [29]. His algorithm assigns random values to vertices and includes the local minima v into the solution while removing $\mathcal{N}[v]$ from the graph G. In each iteration an independent set is removed from G. The algorithm terminates when all vertices are removed, the solution is the union of the removed independent sets.

Blelloch et al. [7] showed that the dependency for a greedy MIS algorithm has polylogarithmic height for random graphs with high probability. They proposed a parallel algorithm mimicking the sequential greedy algorithm. Their algorithm provides a trade-off between parallelism and work by processing a prefix of the vertices in parallel.

Hespe et al. [23] proposed a parallel shared memory algorithm accelerating kernelization. They use a variety of reduction rules. Graph partitioning and parallel maximum bipartite matching are used to apply these in parallel. Furthermore they use a technique called dependency checking to speed up the check if a reduction rule is applicable in the later stages of the algorithm. They use reduction tracking to stop reducing once it becomes inefficient. Their result showed that a fast kernelization is a key ingredient for fast independent set algorithms.

3 Reduction Rules

The reduction rules defined in this section and their proofs are proposed in Chang et al. [12].

To extract a kernel we need to remove vertices from the graph to shrink its size and either include or exclude them from the independent set. We can do that under certain circumstances without jeopardizing the correctness of our result. In this thesis we apply reduction rules on degree one and two vertices. We present the reduction rules in this chapter.

For a given degree one vertex $v \in V$ there exists a maximum independent set that includes v and excludes its neighbor.

Theorem 1 (Degree one reduction)**.** *Given a graph $G = (V, E)$, $v, w \in V$, $d(v) = 1$ and $\{v, w\} \in E$. Then*

$$\alpha(G) = \alpha(G[V \backslash \{w\}])$$

Proof. To show this we construct a maximum independent set that excludes w.

Consider a maximum independent set I that includes w. Since I includes w it excludes v. So by exchanging v and w we end up with a different maximum independent set that excludes w. □

With the degree one reduction the following lemma can be proven:

Lemma 1. *For a path P, $|P| > 2$, $d(p_1) = d(p_{|P|}) = 1$, $d(p_i) = 2$ for $1 < i < |P|$ exhaustive application of degree one reduction leads to $\lceil \frac{|P|}{2} \rceil$ vertices becoming degree zero. We call such path a chain.*

Proof. The Lemma holds for chains of length two and three. Now P is a chain of length > 3. Degree one reduction of p_1 leads to the removal of p_2. Vertex p_1 becomes degree zero. Chain P becomes a chain P' with $|P'| = |P| - 2$. With the induction hypothesis this leads to $\lceil \frac{|P'|}{2} \rceil + 1 = \lceil \frac{|P|-2}{2} \rceil + 1 = \lceil \frac{|P|}{2} \rceil$ vertices becoming degree zero. □

Lemma 2. *Consider a path P, $|P| > 2$, $d(p_1) = 1$, $d(p_{|P|}) \neq 1$, $d(p_i) = 2$ for $1 < i < |P|$. We call such a path one sided chain. Exhaustive application of the degree one reduction leads to decrementing $d(p_{|P|})$ if $|P|$ is odd and the removal of $p_{|P|}$ if $|P|$ is even. It leads to $\lfloor \frac{|P|}{2} \rfloor$ vertices becoming degree zero.*

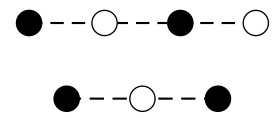
Proof. The base cases are one sided chains P with length two and three. The lemma holds for both cases. A one sided chain with length two degree one reduces p_1, therefore removes

p_2. If it has length three, degree one reduction of p_1 removes p_2 and its edges. Since its adjacent to p_3, $d(p_3)$ is reduced.

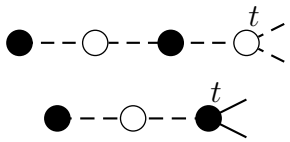
For $|P| > 2$ the degree one reduction of p_1 transforms the remainder of P into a onesided chain P' with length $|P'| = |P| - 2$. Vertex p_1 becomes degree zero. A degree one reduction does not change the length of the emerging one sided chain is odd or even. With the induction hypothesis the first part of the lemma holds. For the number of vertices turning degree zero holds: $1 + \lfloor \frac{|P'|}{2} \rfloor = 1 + \lfloor \frac{|P|-2}{2} \rfloor = \lfloor \frac{|P|}{2} \rfloor$. □

A visualization for the lemmas is shown in Figure 3.1. While the black vertices are part of the independent set the white vertices and dashed edges have been removed by the degree one reduction.

To further shrink the size of our kernel graph we use reduction rules applicable to degree two paths and cycles. The following theorem applies to degree two cycles:



(a) Degree one reduction on chains



(b) Degree one reduction on one sided chains

Figure 3.1: Visualization for Lemma 1 (a) and Lemma 2 (b)

Theorem 2 (Degree Two Cycle Reduction)**.** *Let C be a degree two cycle in G. Then*

$$\alpha(G) = \alpha(G[V \backslash C]) + \left\lfloor \frac{|C|}{2} \right\rfloor$$

Proof. A degree two cycle C is a connected component of graph G, thus $\alpha(G) = \alpha(G \backslash C) + \alpha(C)$.

For $|C| = 3$ an arbitrary $v \in C$ is inserted into I. The removal of $\mathcal{N}[v]$ removes C entirely. For $|C| = 4$, a degree zero vertex remains which is included into I. If $|C| > 4$ the remainder with length $|C - 3|$ forms a chain. With Lemma 1 follows

$$\alpha(C) = 1 + \left\lceil \frac{|C| - 3}{2} \right\rceil = \left\lceil \frac{|C| - 1}{2} \right\rceil = \left\lfloor \frac{|C|}{2} \right\rfloor$$

□

Figure 3.2 shows a visualization for the proof.

We now take a look at degree two paths. The following theorem classifies them based on the relationship of termination vertices and shows how to reduce them. A visualization for these reduction rules is displayed in Figure 3.3.

Theorem 3 (Degree Two Path Reduction)**.** *Given a graph G and a degree two path P. Following cases hold*

(a) $\mathcal{T}(P) = \{v\}$

$$\alpha(G) = \alpha(G[V \backslash \{v\}])$$

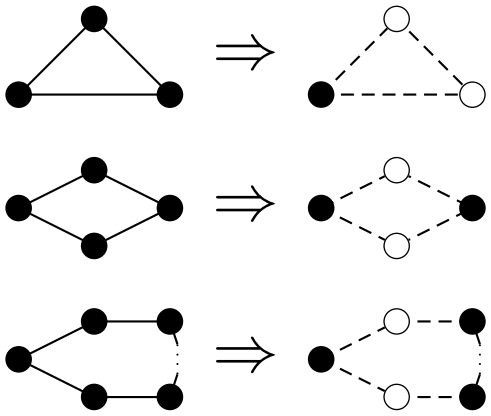


Figure 3.2: Basic cases of degree two cycles.

Proof. Consider an independent set I that includes v. We add vertices from P. Since $v \in I, \mathcal{N}[v]$ is removed from the graph. For $|P| = 2$ no further vertices can be included. In case $|P| = 3$ one vertex in P becomes degree zero and can be added to I. Is $|P| > 3$ the remainder of the path forms a chain of length $|P| - 2$ (two vertices of P are in $\mathcal{N}(v)$). With Lemma 1 we see that $\lceil \frac{|P|-2}{2} \rceil = \lceil \frac{|P|}{2} \rceil - 1$ of it can be added to I.

We show that an independent set I' that excludes v has at least the same size as I. Since $v \notin I'$ we can remove it from the graph. Path P becomes a chain of length $|P|$. We can apply Lemma 1 to see that $\lceil \frac{|P|}{2} \rceil$ vertices from P can be added to I'.

We see that I contains v and $\lceil \frac{|P|}{2} \rceil - 1$ vertices from the path where I' contains $\lceil \frac{|P|}{2} \rceil$ from the path and therefore $|I| = |I'|$. □

(b) $|P|$ *is odd,* $\mathcal{T} = \{u, v\}$ *and* u, v *adjacent*

$$\alpha(G) = \alpha(G[V \backslash \{u, v\}])$$

Proof. Let I be independent set and without loss of generality $u \in I$. Again we add vertices from P. Since $u \in I$ we remove $\mathcal{N}[u]$ from the graph. For $|P| = 1$ there is no remaining path. If $|P| > 1$ the remainder forms a chain with length $|P| - 1$ since one endpoint of P is in $\mathcal{N}(u)$. With Lemma 1 we see that we can add $\lceil \frac{|P|-1}{2} \rceil \overset{|P| \text{ odd}}{=} \frac{|P|-1}{2}$ vertices from P to I.

Now we construct an independent set I' that excludes u and v. We therefore remove $\{u, v\}$ from the graph. Path P becomes a chain of length $|P|$. With Lemma 1 we see that we can include $\lceil \frac{|P|}{2} \rceil \overset{|P| \text{ odd}}{=} \frac{|P|+1}{2}$ vertices into I'.

This results in I containing u and $\frac{|P|-1}{2}$ from the path while I' contains $\frac{|P|+1}{2}$ vertices from the path. Therefore they have the same cardinality. □

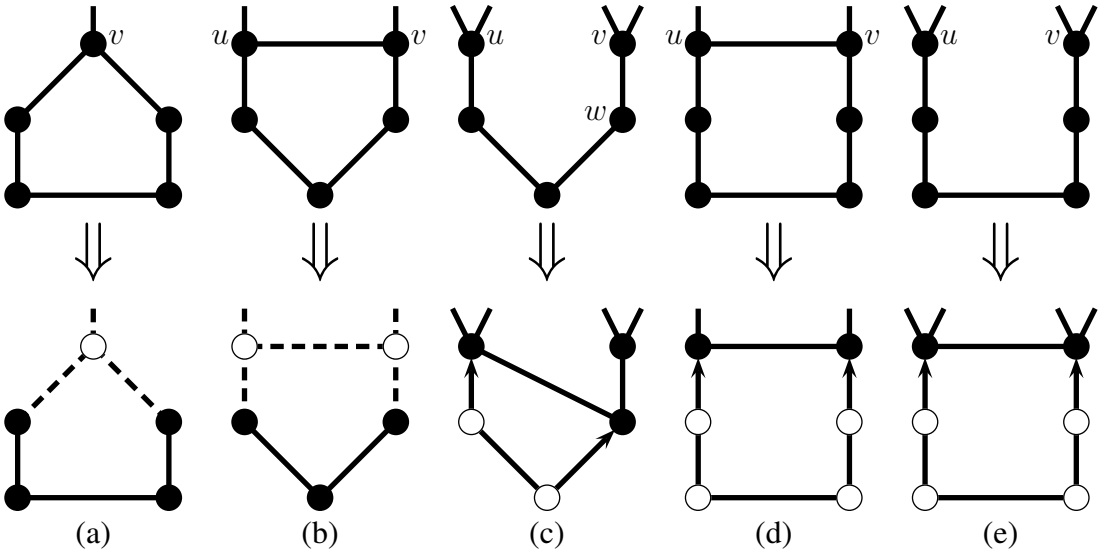


Figure 3.3: Visualization for the degree two reduction rules

(c) $|P| > 1$ *is odd,* $\mathcal{T} = \{u, v\}$ *and* u, v *not adjacent. With* $w \in \mathcal{E}(P)$ *such that* $\{u, w\} \notin E$

$$\alpha(G) = \alpha(G[V \backslash (P \backslash \{w\})] \oplus \{u, w\}) + \frac{|P| - 1}{2}$$

Proof. We show that there exists a maximum independent set that excludes w or u. Without loss of generality $w = p_1$.

Consider a maximum independent set I with $u, w \in I$. Since $u, w \in I$, $\mathcal{N}[u] \cap \mathcal{N}[v] \notin I$ and can be removed from the graph. The remaining path has length $\frac{|P|-3}{2}$. Because I is maximum Lemma 1 states that it contains $\frac{|P|-3}{2}$ vertices from it aswell. Let $I' = I \backslash (\{u\} \cup P) \cup \{w\}$ (includes w) and $I'' = I \backslash P$ (includes u). For their cardinality holds $|I'| = |I| - 1 - \frac{|P|-3}{2} = |I| - \frac{|P|-1}{2} = |I''|$.

Since I' includes w we can remove $\mathcal{N}[w]$ from the graph. The remaining path has length $|P| - 2$. It forms a one sided chain. Reducing it produces $\lfloor \frac{|P|-2}{2} \rfloor \overset{|P| \text{ odd}}{=} \frac{|P|-1}{2}$ degree zero vertices that can be added to I'. Further u is removed because the one sided chain has odd length (Lemma 2). Now I' has same cardinality as I and is therefore maximum.

Next, we extend I'' to be maximum. We remove $\mathcal{N}[u]$ from the graph. The remaining path has length $|P| - 1$. It again forms a one sided chain. We apply Lemma 2, $\lfloor \frac{|P|-1}{2} \rfloor \overset{|P| \text{ odd}}{=} \frac{|P|-1}{2}$ vertices become degree zero and can be added to I''. Since $|P|$ is odd, the one sided chain is even. It does not remove the termination vertex v. Instead it removes w (v and w adjacent). The independent set I'' has same cardinality as I and is therefore maximum.

We saw that a maximum independent set exists that includes either u or w. Such a maximum independent set definitely contains $\frac{|P|-1}{2}$ vertices from the path. We therefore can remove $P \backslash \{w\}$ from the graph. To assure we end up with a maximum independent set that excludes either u or w we insert the edge (u, w). □

(d) $|P|$ *is even and* $\mathcal{T} = \{u, v\} \in E$

$$\alpha(G) = \alpha(G[V \backslash P]) + \frac{|P|}{2}$$

Proof. Since $u, v \in \mathcal{T}(P)$ adjacent, a maximum independent set I never includes both. If $u, v \notin I$ then $\frac{|P|}{2}$ vertices from P are in I (I maximum and Lemma 1). Otherwise without loss of generality $u \in I$. Then $\mathcal{N}[u]$ is removed from graph. The remaining path has length $|P|-1$ and is a chain. Lemma 1 says that $\lceil \frac{|P|-1}{2} \rceil \overset{|P| \text{ even}}{=} \frac{|P|}{2}$ can be included. In both cases the same number of vertices get included into a maximum independent set. The path can therefore be removed from the graph. □

(e) $|P|$ *is even and* $\mathcal{T} = \{u, v\} \notin E$

$$\alpha(G) = \alpha(G[V \backslash P] \oplus \{u, v\}) + \frac{|P|}{2}$$

Proof. We prove this similiar to the (c) case by showing that there is a maximum independent set that excludes one element from the termination and includes the other.

Consider a maximum independent set I and $\mathcal{T}(P) \subset I$. The removal of $\mathcal{N}[u] \cup \mathcal{N}[v]$ shortens the length of P by two and makes P a chain. Once again Lemma 1, says that $\lceil \frac{|P|-2}{2} \rceil = \frac{|P|-2}{2} = \frac{|P|}{2} - 1$ are included into I aswell.

Next, consider (without loss of generality) $I' = I \backslash (\{v\} \cup P)$. Because it does not include v and the vertices from the path $|I'| = |I| - \frac{|P|}{2}$. We remove $\mathcal{N}[u]$. Path P now has length $|P| - 1$. It forms a one sided chain of length $|P|$. With Lemma 2 we see that $\lfloor \frac{|P|}{2} \rfloor \overset{|P| \text{ even}}{=} \frac{|P|}{2}$ become degree zero and can be included into I'. Further since the one sided chain has even length v is removed and not in I'.

Adding $\frac{|P|}{2}$ vertices from the path to I' makes it maximum, because it has the same cardinality as I.

Similiar to (c), there exists a maximum independent set that excludes one of the termination vertices of the path and adds $\frac{|P|}{2}$ vertices from it. We can remove P from the graph and insert the edge (u, v). □

4 The Algorithm

In this chapter we describe our distributed algorithm. The algorithm exhaustively applies degree one and degree two path reductions until the kernel graph is determined. Afterwards we compute a MIS on it. Pseudocode is given in Algorithm 1.

At first, we describe the process of the kernelization and afterwards the MIS algorithm.

Algorithm 1: Top-level algorithm

`Input` : unidirected graph G
`Output`: Independence number α of G

1 $(K, \alpha) \leftarrow$`ComputeKernel(`G`)`
2 $\alpha' \leftarrow$`ComputeMIS(`K`)`
3 `return` $\alpha + \alpha'$

4.1 Computing the Kernel

Like most parallel algorithms our algorithm consists of a local computation part performed on each PE and a communication phase in which ghost vertices are updated. Simply applying reductions locally without communicating does not necessarily compute a minimal kernel. That is because reductions may affect ghost vertices and eventually allow further reduction of vertices on another PE. The very simple example in Figure 4.1 shows such a case.

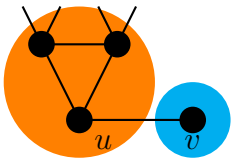


Figure 4.1: Degree one reduction of v leads to degree two reduction of vertex u

We first describe the local part of the kernelization. Afterwards we explain the communication phase. Unless otherwise mentioned we describe our algorithm from the viewpoint of PE i.

4.1.1 Local Work

When initialized, our algorithm constructs three stacks V_o, V_1 and V_2. Each of them holds the degree zero, one and two vertices respectively. We only push local vertices onto these stacks.

Degree zero vertices are by definition part of the maximum independent set. Due to them being local there is also no need for communication. For degree one and two vertices there is a little more work to do. The following subsections cover the details of their reduction.

Degree One Reduction

To degree one reduce a vertex v we locate its single neighbor w. The neighbor is removed. We decrease the degree of adjacent vertices of w and push them onto one of the stacks if necessary.

The operation is local if w is a local vertex and not an interface vertex. Otherwise we have to communicate with other PEs that are affected by the removal of the vertex. In this case a message containing the id of w is inserted into a buffer and is later communicated.

Pseudocode is given in Algorithm 2.

Algorithm 2: Reduction of degree one vertices

```
1  Procedure DegreeOneReduction()
2      M ← ∅
3      foreach v ∈ V_1 do
4          if d(v) ≠ 1 then
5              continue
6          w ← adjacent vertex to v
7          RemoveVertexAndAdjacentEdges (w)
8          V_1 ← V_1\v
9  Procedure RemoveVertexAndAdjacentEdges (v)
10     if IsGhost(v) then
           // first element in a n-tupel is receiver id
11         M_Rem ← M_Rem ∪ (PE(v), v)
12     foreach neighbor u of v in G do
13         Decrease d(u)
14         if IsLocal(rank,v) then
15             Push onto one of the stacks if necessary
16         if IsGhost(u) then
17             M_Rem ← M_Rem ∪ PE(v), v)
18     G ← G\v
```

Degree Two Path Reductions

When we deal with degree two vertices our goal is to find maximum length degree two paths and reduce them. When processing the graph in a distributed way the situation may occur that a path continues on an adjacent PE. Firstly we look at how to deal with local paths, then we describe our approach on handling distributed paths.

We noticed during implementation that removing the vertices while expanding the path is faster than doing it during reduction. Thus we interleave the actual path creation with the reduction. Everytime a vertex is added to a path it is marked as removed. The finding of a path is not difficult. A degree two vertex v is added to a new path P. We then check its active neighbors and add them to the path as well if their degree is two. Otherwise it has degree greater two and is a termination vertex for P. In that case its degree is decremented. We also keep count of the number of paths that are attached to a vertex and removed, but not yet reduced. Such a path is called *pending*. When we find a termination vertex the pending counter for it is incremented. One important fact is that vertices are not eligible for a degree one or degree two reduction if they have pending paths. That is because we cannot predict its degree before the reduction is performed. Therefore a vertex whose pending counter is greater zero can not be part of a degree two path.

We can reduce a local path instantly by applying the reduction rules presented in the previous chapter. A reduction of a path decrements the pending counter for the vertices in the termination. Also, if a termination vertex becomes eligible for degree one or two reduction it is pushed on the respective stack. Notice that the reinserted endpoint when reducing a path with the (c) reduction rule, although being degree two, is not pushed onto V_2. The reason is that it forms an odd path again and is therefore removed, reinserted and pushed on the stack again. Our algorithm does not make progress and never terminates. This leads to a special case which we discuss in the next paragraph.

The Order of Path Reductions Might Lead to Different Kernels Consider the following case. Two non-adjacent vertices u, v are the termination vertices of two paths P, P' with $|P|$ odd and $|P'|$ even. Lets assume that our kernel is minimal besides the reduction of these two paths. In the beginning we look at the case in which P is reduced first. The path is reduced by applying (c) reduction rule. Afterwards there is a degree two path with length one between u and v. After performing the reduction of P' by applying the (e) reduction rule, the kernel is determined. It contains u, v and the reinserted endpoint w of P. Since w is not pushed onto the stack our algorithm terminates and the kernel is determined. Next, we look at the case in which P' is reduced first. The reduction inserts an edge between u and v. When P is reduced, reduction rule (b) instead of (c) is applied. This leads to the removal of u and v. The kernel has three vertices less.

We see that this far our algorithm might miss a possible reduction. To fix this we need to put termination vertices $\{u, v\}$ of a path that is reduced with (c) reduction into a set. Before we apply the (e) reduction rule we check if the termination vertices of the path are in that

set. If they are, we remove both termination vertices from the graph.
The algorithm proposed by Chang et. al [12] does not consider this case and therefore possibly misses some degree two reductions.
Figure 4.2 illustrates this case.

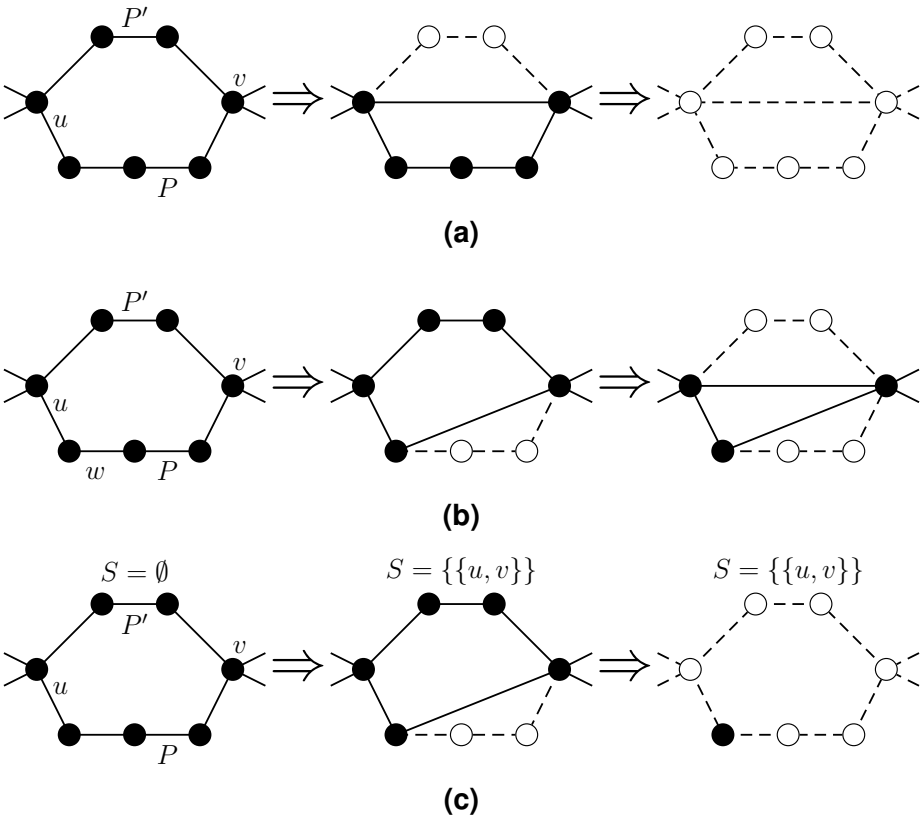


Figure 4.2: This example shows the need to store the termination vertices after a path reduction with rule (c). Our algorithm performs reduction of an odd path P and an even path P' sharing the same termination vertices. In (a), P' is removed first with the (e) reduction rule. Afterwards P is reduced with the (b) reduction rule. In (b), P' is reduced first with the (c) reduction rule. It reinserts endpoint w. Afterwards P is reduced. Although another degree two reduction is possible, w is not pushed on the stack and therefore never reduced. Figure (c) shows the way to handle this. A reduction with the (c) rule puts the termination vertices in a set. Before a reduction with rule (e) is performed we check whether the termination vertices are mapped. In that case we do not need to insert an edge between them, because they are removed by reduction rule (b) that now becomes possible. Therefore we simply remove the termination vertices of the path aswell.

When we expand a path and add an interface vertex to it, we deal with a distributed path. We identified a path segment of it. For now we are not yet able to reduce this. To do so we need to acquire knowledge about the paths properties. We need to know the termination, endpoints and length. The way we gain these information is described in the next section.

4.1.2 Communication Between PEs

Communication is necessary when a PE can not make further local progress. The receival of information from other PEs might enable local progress again. Our algorithm sends different messages between PEs. This section discusses these messages.

We already realized that the removal of a vertex might need to be communicated to other PEs. For this situation we introduce the *vertex removal message*.

Removal of Ghost and Interface Vertices

Every time an interface vertex u or ghost vertex v is removed during degree one or degree two reduction, communication is necessary. Other PEs need to be informed about the removal to update their part of the graph. Sample situations that require communication are given in Figure 4.3.

We have to tiebreak the situation where two degree one vertices are connected and reside on different PEs. In such a case the independence number gets incremented twice, since each PE performs a local degree one reduction. The reception of a vertex removal message on PE i for a vertex u indicates that a neighbor of it got into the independent set. If v was degree one reduced ealier and i is smaller than the rank of the sender, it has to remove u from the independent independent set. This case is displayed in Figure 4.3d.

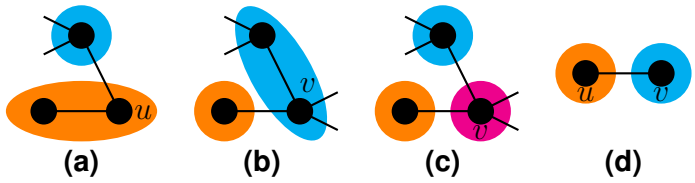


Figure 4.3: The removal of an interface (Figure a) or ghost vertex (Figure b-d) requires communication. Figure (c) shows a case that requires two communication rounds to fully update. A special case is shown in Figure (d). Here we need to tiebreak which vertex gets into the independent set.

Next, we look at how we can reduce distributed degree two paths. We do this with a technique called label propagation (LP). It is described in the next section.

Label propagation

Our situation is the following: a global path P is distributed into path segments on different PEs. We need to connect them and determine the length, endpoints and termination of P in order to reduce it. Length and termination are necessary to decide which rule we need to apply. In case reduction rule (c) is performed, we also need to know the endpoints of the

path. The label propagation process consists of passing a *label propagation* message along cut edges of path segments until the properties of the path are determined.

A path segment $\hat{P}^i$ that has one cut edge $\{u, v\}$ ($\texttt{PE}(u) = i$) is responsible for starting the LP process. A LP message consists of information about the path (length, endpoint, termination) and the cut edge $\{u, v\}$ and is sent to PE $\texttt{PE}(v)$. We mark cut edge $\{u, v\}$ as sent.

When PE j receives a LP message different cases can occur; We are able to extend the path if the PE has a path segment $\hat{P}^j$ that has a cut edge $\{u, v\}$. In that case we mark cut edge $\{u, v\}$ of $\hat{P}^j$ as received. We can stop the label propagation if all cut edges of $\hat{P}^j$ are marked received, since all path properties are known to PE j. Otherwise we need to propagate the message further along the cut edge that is not marked as received. If no such $\hat{P}^j$ exists then v is a termination vertex of the path. In the special case, $d(v) = 0$, u becomes termination vertex of the path. That case occurs when v was degree one reduced earlier. Again the properties are known and the path can be reduced.

If LP is only started by path segments that have one cut edge certain paths are never reduced. An example is given in Figure 4.4c. Here both termination vertices are not aware that their neighbors have degree two, because they are ghost vertices. Hence the LP process is never started. To catch such cases, path segments with two cut edges need to check whether the cut edge is incident to a termination vertex. They do so by sending a *request termination* message for every cut edge to the incident PE. For cut edge $e = \{u, v\}$ with $\texttt{PE}(u) = i$ and $\texttt{PE}(v) = j$ the request termination message sent by PE i is answered by PE j with a *send termination* message if $d(v) \neq 2$ or v has pending paths. The send termination message contains the cut edge and whether v is active or not. In the special case in which v was degree one reduced earlier, it contains the tripel $(u, u, 0)$. Then u becomes a termination vertex of the path and the LP process is started by PE i.

The algorithm could be further improved at this point. A more sophisticated approach could use the fact that the removal of vertices is already communicated via vertex removal messages. If the removal of a vertex affects a path segment with two cut edges, it would need to extract the information from that message and perform further steps. The advantage is that send termination messages do not need to be sent for inactive vertices. This would decrease communication volume and rounds. However such an approach is quite error prone since a lot of special cases have to be considered. Eventually we switched back to the easier approach of sending the status of each vertex.

With these two new introduced messages, we can be sure that every distributed degree two path is found (except for degree two cycles, later more on that). The next chapter is devoted to the actual reduction of distributed paths.

Reducing Distributed Paths

The LP process terminates if a PE is aware of the properties of a path. That is if for a path segment $\hat{P}^j$ each cut edge is marked as received. Further, if a cut edge is marked as sent

Algorithm 3: Reduction of degree two vertices

```
1  Procedure DegreeTwoReduction()
2      M_LP ← ∅
3      M_ReqT ← ∅
4      foreach v ∈ V_2 do
5          if d(v) ≠ 2 ∨ already in a path then
6              continue
7          Extend v to be a maximal degree two path P
8          if P is local path then
9              ReducePath (P)
10         else
11             if P has one cut edge {u, v} then
12                 M_LP ← M_LP ∪ (PE(v), u, v, E(P), T(P), |P|)
13             else
14                 foreach cut edge {u, v} of P do
15                     M_ReqT ← M_ReqT ∪ (PE(v), u, v)
16         V_2 ← V_2\{v}
```

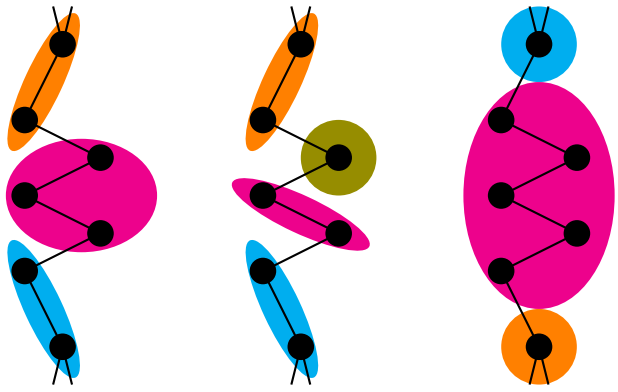


Figure 4.4: Distributed degree two paths. Properties of the path in case (a) are known after one round of label propagation, the magenta PE needs to initiate the reduction. Case (b) requires a tiebreak since both magenta and olive are aware of the path properties after two rounds of label propagation. The lower ranked PE needs to initiate the reduction. Case (c) shows the need for request termination messages, otherwise the path would never be reduced.

then two PEs are aware of the path properties and in position to initiate the reduction. Then the lower ranked PE is responsible for initiating the reduction. These two situations are illustrated in Figure 4.4a and Figure 4.4b.

To inform a PE about the reduction of a path we introduce the *path reduction* message. We need to send between zero and two path reduction messages containing the path properties to perfom the reduction. There is no need for communication if the termination vertices of P lie in the PE that is responsible for initiating the reduction. We need to send one message if we deal with cases (a) from Theorem 3, the termination vertices lie in the same PE or one termination vertex lies in the PE that is responsible for initiating the reduction. Otherwise the termination vertices lie in different PEs and we need to send two messages. Figure 4.5 gives an example case for each possible situation.

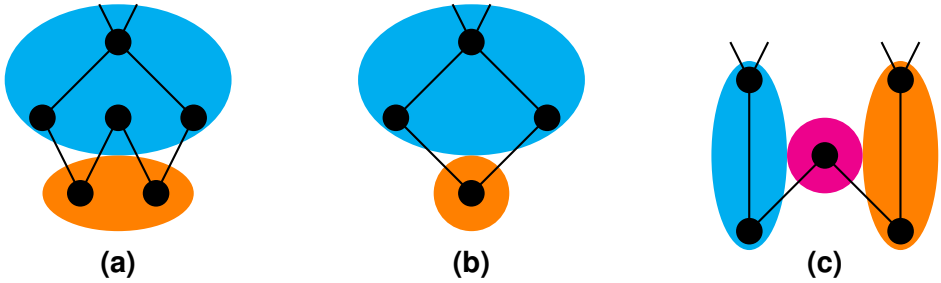


Figure 4.5: After completing the LP process zero (Figure a), one (Figure b) or two (Figure c) messages need to be sent to inform relevant PEs about the reduction of a path.

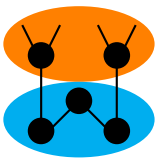


Figure 4.6: We need to communicate to reinsert the edge to an endpoint when reducing this path.

In case the termination lies on a single PE the reduction is quite similiar as when we reduce a local path. The difference is that it is not guaranteed that the vertices in the termination are still active. An earlier degree one or two reduction might removed a termination vertex. This does not matter for cases (a) or (b) from Theorem 3 because these cases remove the termination vertices anyway. Otherwise, the path dissolves by repeated degree one reduction. Since it has already been removed these reductions are not performed. We have to apply Lemma 1 or Lemma 2 depending on whether both or just one vertex in the termination has been removed. Lemma 2 states that the degree of the termination vertex has to be reduced if the one sided chain has odd length. However this is not necessary since we already removed the path and decremented the degree of termination vertices. We again have to decrement the pending counter for active termination vertices and increment the independence number.

Additional communication is necessary in case we reduce a path with reduction (c) where the endpoint that is reinserted is not local. Such a case in show in Figure 4.6. Notice that the insertion of edges might lead to the creation of new ghost vertices. Such a case is displayed in Figure 4.7.

Next we look at the case that the termination vertices of a path lie on different PEs. Each PE

performs the necessary actions to their local vertex. When the (c) reduction is performed the endpoint w with the lower id is reinserted. This reinsertion can be done without communication if it lies in the same PE as one of the termination vertices. Otherwise, the PE holding the lower ranked termination vertex sends a message to inform the PE $\texttt{PE}(w)$ about the reinsertion. The PE holding the termination vertex with the lower rank is responsible for incrementing the independence number.

Again, the case might occur in which a termination vertex v is already removed. Additional work is necessary if the path would have been reduced with the (c) or (e) reduction. In some cases v becomes a ghost vertex on another PE j. The prior removal of v did not inform PE j if an edge from v to u with $\texttt{PE}(u) = j$ has not already existed. We therefore have to send a vertex removal message for v to the PE that inserts an edge to v when performing the reduction. Figure 4.8 shows an example case.

The other cases are covered. When reducing with case (a) there is no other termination vertex. Reduction (b) removes both termination vertices anyway. In case (d) an edge between the termination vertices already exists, so the removal of v is definitely communicated to the other PE.

Distributed Cycles

During the label propagation section we said that a path segment with one cut edge starts the LP process. Further, we described the necessity for request termination messages to detect cases in which no path segment of a path has one cut edge. We said that request termination messages are ignored if the relevant vertex is part of path segment aswell. Considering these circumstances, a degree two path that is spread across multiple PEs is never found and therefore never reduced. To find them we do the following.

A path segment $\hat{P}^i$ has two cut edges, $\{u, v\}, \{w, x\}$ with u, w local. It sets its id to minimum id of u and w. Lets assume u is the vertex with the smaller id. A *cycle propagation* message is propagated along the cut edge $\{u, v\}$. The message contains the cut edge $\{u, v\}$, the length and the id of the path segment. It does not matter on which cut edge we sent the cycle propagation message in case $u = w$.

When PE j receives a cycle propagation message concerning path segment $\hat{P}^j$ it only propagates it further along the other cut edge of $\hat{P}^j$ if the id is greater then the id received in the message. Eventually the PE that originally sended the cycle propagation message

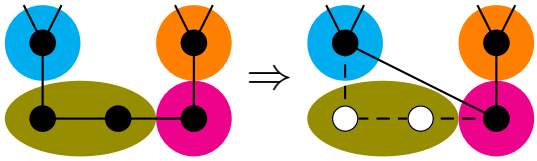


Figure 4.7: The reduction of this path creates a new ghost vertex.

Figure 4.8: The reduction of this distributed path shows the case in which a PE creates a new ghost that is already deactivated. It shows the necessity to resend a removal message if the reduction of a path inserts an edge to an already deactivated termination vertex.

receives it with the accumulated length of the cycle. It increases α according to Theorem 2.

4.1.3 Putting it All Together

We spent quite some time discussing the necessary ingredients to distributed kernelization. Different message types have been introduced and we talked about their handling. This section is devoted to the assembly of the presented parts into a working kernelization algorithm.

Let's quickly repeat what message types have been introduced. The vertex removal message informs a PE about the removal of a vertex. For distributed path finding, we need a label propagation, request and send termination messages. To inform a PE about the details for a path reduction we need a path reduction message. In case we need to communicate when reinserting the endpoint of a distributed path we need the *reinsert endpoint* message. Although we did not explicitly mention this, we need an additional message. When we apply Lemma 2 we might need to decrease the number of pending paths for a vertex. For this case we introduce the *decrease pending path* message. With these eight messages we are able to perform the distributed kernelization.

Our main theme is to exhaustively apply local work. Local work consists of degree one and degree two reductions. Algorithm 4 shows pseudocode.

Algorithm 4: The local work part of our algorithm

```
1 Procedure LocalWork()
2     while V_1 ≠ ∅ ∨ V_2 ≠ ∅ do
3         if V_1 ≠ ∅ then
4             DegreeOneReduction()
5         else
6             DegreeTwoReduction()
```

After a PE can no longer perform local work, it is ready to communicate. If every PE is ready to communicate, messages are exchanged. We handle one message and check

if we are able to perform local work again. We do this until all the received messages are handled at which point we are ready to communicate again. The algorithm stops if no more messages are sent. Pseudocode is given in Algorithm 5.

Algorithm 5: COMPUTEKERNEL

```
Input : Unidirected graph G
Output: Kernel graph K and independence number α
1  α ← 0
2  Initialize V_0, V_1 and V_2
3  LocalWork()
4  while atleast one of the PEs has data to send do
5  |  Send messages
6  |  M ← received messages
7  |  foreach m ∈ M do
8  |  |  HandleMessage (m)
9  |  |  LocalWork ()
10 Find distributed cycles
11 return (G,α + |V_0|)
```

We did not provide pseudocode for the ReducePath and HandleMessage method. They perform the actions described in this chapter and the previous one.

4.2 Computing the maximal independent set

After we aquire the kernel graph, our next goal is to calculate a maximal independent set on it. We do this by using a randomized approach, similiar to LUBYS algorithm [29]. Algorithm 6 shows the pseudocode.

Each PE processes their local vertices and assigns random numbers to them. The random number for a vertex v has to be shared to other PEs having v as a ghost vertex. Every vertex v that is a local minimum will be taken into the independent set and thus v and its neighborhood are removed from the graph. In case v is an interface vertex, we again have to inform the PEs that have v as a ghost about its removal. They need to remove $\mathcal{N}[v]$ aswell.

It is not necessary to inform other PEs about the removal of interface vertices in $\mathcal{N}[v]$. Even if on PE i there is a vertex $v \in V^i_{ghost}$ that has been previously removed by PE $\texttt{PE}(v)$, since we initialize the random values for every active vertex with the maximum value and only receive actual random numbers from active vertices, the correctness of our result is not threatened by such dangling vertices.

Algorithm 6: COMPUTEMIS

```
Input  : Unidirected graph G
Output : Size of maximal independent set for G
1  α ← 0
2  M ← ∅
3  rvals ← array of size |V| initialized with maximum value
4  while |V_l| ≠ ∅ in parallel do
5      foreach v ∈ V_local do
6          rvals[v] ← random value between 0 and maximum value
7          if IsInterface(v) then
8              foreach neighbor u of v do
9                  if IsGhost(u) then
10                     M ← M ∪ (PE(u), v, rvals[v])
11     Send messages in M to their destination
12     Receive buffer R containing messages from other pes
13     Set rvals for the received vertices
14     M ← ∅
15     foreach v ∈ V_l do
16         if rvals[v] is smallest among N(v) then
17             Increment α
18             if IsInterface(v) then
19                 foreach neighbor u of v do
20                     if IsGhost(u) then
21                         M ← M ∪ (PE(u), v)
22             G ← G\N̄(v)
23     Send messages in M to their destination
24     Receive buffer R containing messages from other pes
25     Remove every vertex in R from graph
26 return α
```

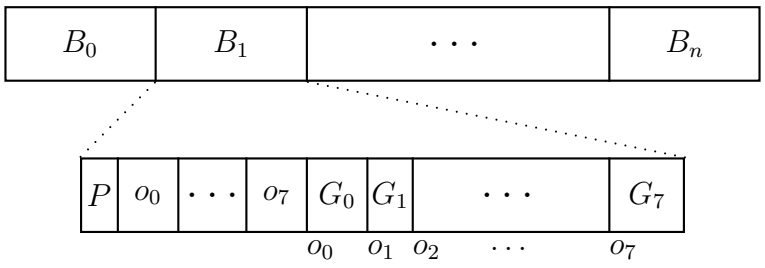


Figure 4.10: A message M contains all the individual messages from other PEs. The messages from a PE are organized in blocks. A block starts with a preamble containing nine elements. The first element indicates progress, the next eight elements the offset for the group. A group is all individual messages of a certain type.

We terminate as soon there are no more local vertices to process and return the independent set size.

4.3 Implementation Details

Our implementation is written in C++ and uses MPI for the communication between processes. We use the BSP communication model and implement it with the MPI Alltoall operation. Alltoall is a collective operation in which every process sends data to all other processes. Our experiments show that for most of the unpartitioned instances we tested, there is at least one cut edge between almost every pair of PEs.

A message M received by a PE contains all the individual messages from other PEs (for it). Message M is structured in the following way: For each PE a block within M exists, containing all the individual messages from it. Since there are different message types, we avoid the necessity of a message identifier by grouping them into groups of the same type. A preamble contains information about a block. It contains nine elements. The first element is the progress bit. It contains whether or not at least one message has been sent by that PE this communication round. The next eight elements contain the offset of the starting position for each group of messages. After the preamble, the individual messages grouped by type follow. Figure 4.10 shows an overview. Communication is stopped if all progress bits are zero.

We use adjacency lists to save edges. For each vertex we store the degree and number of pending paths. To speed up our algorithm, we mark vertices as deactivated rather than updating the adjacency lists. A vertex is marked as deactivated when its degree and the number of pending paths are zero. For each vertex we save pointers to path objects storing information about the path it is part of. A null pointer is stored if it is not part of a path.

The Alltoall operation requires the message to be stored in continous memory. We therefore buffer messages and copy them before sending. This gives us the possibility to check if the

message still needs to be sent. For example, a label propagtaion message is buffered when a path segment with two cut edges receives a LP message. When we prepare the send operation, we can check if we received another LP message concerning that path. In that case we do not need to send the LP message. This way we can reduce the communication volume.

We insert edges when reducing a path with the (c) or (e) reduction rule. In both cases a neighbor of the vertex that gets a new edge is removed. We therefore do not append a new vertex to the adjacency list but rather take the spot of the removed neighbor. This way we do not need to reallocate memory. It is important to insert a backward edge for a ghost vertex if the target of the edge insertion is not a local vertex.

We use the xxHash[1] hash algorithm to store std::pair in an unordered set.

[1] `https://github.com/Cyan4973/xxHash`

5 Experimental Evaluation

This section describes the experimental evaluation of our algorithm. We evaluate both running time and quality of the result of our algorithm and compare it to previous results. It is structured as follows: First we describe our experimental setup. Next we evaluate running time and scalability of our algorithm. Both parts of our algorithm, the kernelization and the computation of the MIS, are evaluated. We test strong and weak scaling behavior. Afterwards, we take a look at the impact of partitioning on our algorithm. Finally, we evaluate the effect of kernelization.

5.1 Experimental setup

Our program was compiled with gcc 7 with full optimization turned on (-O3) and uses Openmpi 1.10. We ran our tests on the Forschungshochleistungsrechner II. A node on this cluster consists of two deca-core Intel Xeon processors E5-2660 v3 (Haswell) with 2.6 GHz clock rate, 10 times 256 KB of level 2 cache and 25 MB level 3 cache. A node has 64 GB of main memory and a 480 GB SSD. It is connected to the other nodes of the cluster by an InfiniBand 4X EDR interconnect.

We tested both real world and synthetic instances. While a full overview of our experiments can be found in the appendix, we restrict ourselves to a few representative candidates in this section. Table 5.1 shows the real world instances used in this section. We obtained these data sets from the Laboratory for Web Algorithmics (LAW) [9, 8], the Stanford Large Network Dataset Collection (SNAP) [28], the 9th and 10th Dimacs implementation challenge [1, 5] and the Koblenz Network Collection (KONECT) [30].

The LINEARTIME algorithm has a running time of at least 0.5 seconds on each of the instances from Table 5.1. While our test suite also contains smaller instances, we achieve no speedup on most of them. That is somehow comprehensible, since the overhead of communication exceeds the running time of a sequential algorithm. Further small instances do not require the use of parallelism. It makes sense to include them in our test suite anyway. We can still use them to examine the quality of our solution.

For the generation of synthetic instances, we use the KaGen graph generator [17]. Table 5.2 displays the synthetic instances we use in this section and the parameters used for their creation. For weak scaling experiments, we use instances with 2^{20} nodes per core. The n parameter in KaGen determines the number of vertices as a power of two. Since we create instances for up to 64 PEs, n is in $\{20, 21, 22, 23, 24, 25, 26\}$.

name	type	$\|n\|$	$\|m\|$	source
enwiki-2013	web	4206785	91939728	[9, 8]
europe	road	18029721	22217686	[5]
indochina-2004	web	7414866	150984819	[9, 8]
orkut	social	3072441	117185082	[28]
soc-LiveJournal1	social	4847571	42851237	[28]
uk-2002	web	18520486	261787258	[9, 8]
USA	road	23947347	28854312	[1]
youtube-u-growth	web	3223643	9376594	[30, 2]

Table 5.1: Real world instances

name	graph model	KaGen parameters
gnm	Erdos-Renyi	gen = gnm_undirected, m = n+2 (number of edges as power of two)
rhgd8	random hyperbolic	d = 8 (average degree), gamma = 2.8 (power-law exponent)
rhgd4	random hyperbolic	d = 4, gamma = 2.8
rgg	random geometric	gen = rgg_2d, r = $0.55 \cdot \sqrt{\frac{\log 2^n}{2^n}} \Rightarrow$ 0.00132; 0.00095; 0.0007; 0.0005; 0.00036; 0.00026; 0.00019 (radius)

Table 5.2: Synthetic test instances

For the partitioning of our real world instances, we use the KaHIP framework [33]. We use fast as preconfiguration.

An experiment for an instance consists of ten runs of our algorithm. Half of the runs perform kernelization before computing a MIS. We keep track of the time to compute a kernel and the time until our algorithm terminates. Further we are interested in the kernel size and the size of the MIS. Since our algorithm is random, we use the arithmic mean of the result from the different runs.

5.2 Running Time and Scalability

We test the scaling behavior of our algorithm on 2^i PEs with $i \in \{0, 1, 2, 3, 4, 5, 6\}$. The speedup is defined as the execution time of a sequential algorithm divided by the execution time of a parallel algorithm on i PEs.

We evaluate the speedup of our kernelization compared to the LINEARTIME algorithm by Chang et al. [12]. Also we examine the relative speedup of our kernelization. Finally, we

take a look at the relative speedup of our MIS algorithm. For the relative speedup we divide the running time on one PE through the running time on i PEs.

Let us start with the speedup of the kernelization. Figure 5.1 shows the speedup of the kernelization over LINEARTIME. The figure contains two plots. When ordering the instances by its speedup, Figure 5.1a contains the top four, while Figure 5.1b contains the bottom four instances. Likewise, Figure 5.2 shows the relative speedup of the kernelization. Figure 5.2a shows the speedup of the top four instances, Figure 5.2b shows it for the bottom four.

We would like to examine why certain instances perform better than others. To do so, we need some advanced measures for our instances. We look at the density d of a graph. It is defined as $d = \frac{2m}{n(n-1)}$, with n and m being number of vertices and edges respectively. Further, we want to know how many ghost vertices exist on a PE on average. For a run on i PEs, we sum up the number of ghost vertices for every PE and divide it by i. This gives us the average number of ghosts per PE. Since absolute numbers are not meaningful, we divide it by n. We call it 'average percentage of n hold as ghost' g, and determine it for $i \in \{2, 4, 8, 16, 32, 64\}$ PEs. Interesting are maxima and minima. These measures are displayed in Table 5.3.

instance	d	g_{max}	g_{min}
enwiki-2013	$1 \cdot 10^{-5}$	0.5	0.11
europe	$1 \cdot 10^{-7}$	0.0004	0.0002
indochina-2004	$5 \cdot 10^{-6}$	0.04	0.005
orkut	$2 \cdot 10^{-4}$	0.66	0.3
soc-Livejournal1	$4 \cdot 10^{-6}$	0.38	0.09
uk-2002	$1.5 \cdot 10^{-6}$	0.012	0.0066
USA-road	$1 \cdot 10^{-7}$	0.04	0.017
youtube-u-growth	$2 \cdot 10^{-6}$	0.21	0.04

Table 5.3: Advanced measures for our test instances: d is the density of the graph, g is the average number of ghosts per PE divided by the number of vertices n. The minima and maxima for g on the number of tested PEs is displayed.

The best speedup is achieved by the europe instance. It has a speedup of 15 on 64 PEs. Generally, road graphs work pretty well with our algorithm (USA-road is the exception, later more). Even the instances where LINEARTIME has a running time smaller than 0.5 seconds achieve speedup with our algorithm (between 5.8 and 6.2). The relative speedup of the kernelization is 30, the relative speedup of the MIS algorithm is 30 aswell (both achieved on 64 PEs). Noticable is its low density and small number of ghosts.

Second and third best speedup are achieved by web graphs. The uk-2002 instance has a speedup against LINEARTIME of 12, while for indochina-2014 it is 7.7. Their relative speedup for kernelization are 14 and 7.8 respectively. The MIS algorith achieves a relative

speedup of 12 and 10. Noticable here is the slightly lower density and average number of ghosts for the uk-2002 instance.

The rest of the section we examine why the other instances do not scale. Two of the remaining instances are web graphs (enwiki-2013 and youtube-u-growth), two are social graphs (orkut and soc-Livejournal1) and the last one is the already mentioned road graph USA-road. All of these instances do not scale well or at all. The maximal speedup achieved by any of these instances is the relative speedup of the kernelization by enwiki-2013 (5.3). The worst instance is orkut.

We try to examine whats the problem with these instances. To do that we take a closer look at orkut. It is pretty dense and has many ghosts on average. Ghost vertices are redundant information that require communication to update.

One bottleneck of our implementation is the lookup to check, if a vertex is a ghost vertex. When a PE receives a vertex removal message, it checks whether that vertex is a ghost. The case might occur in which it receives a vertex remove message that is not yet a ghost vertex. That is because our algorithm handles vertex removal messages before reduce path messages. Therefore the check is necessary. If it is not a ghost, it becomes one and is immediately deactivated. The lookup takes longer the more ghost vertices exist.

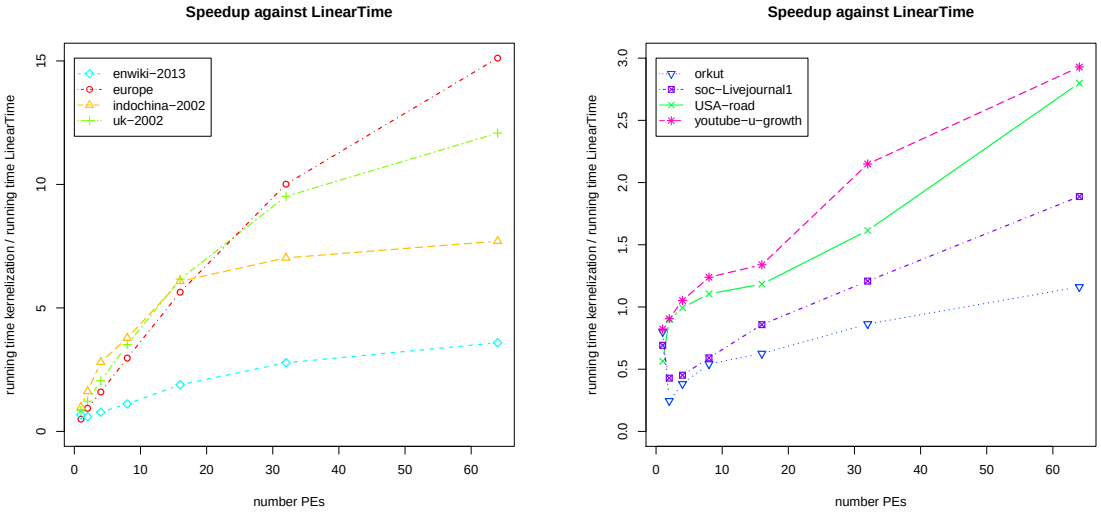
This is a major flaw in our implementation. The argument during implementation for handling removal messages before reduction messages was, that you do not have to perform the work of the reduction if the vertex is going to be removed this communication round anyway. Since this requires a lookup and these are expensive on certain instances we should have went with the other order. A vertex removal message is probably the most frequently send message. So everytime such a message is handled, we perform a slow lookup in the map.

The other instances, except USA-road, suffer from the same problem. They do not perform as bad as orkut, because they do not have as many ghosts. A dense instance is more likely to have a large number of ghosts.

The MIS algorithm performs a lookup everytime it receives a random number for a ghost vertex. Therefore, the same problem arises here. Although it is not quite clear how to fix this. The algorithm needs to perform this lookup to save the receives random value for its ghost.

We expect that the USA-road instance performs well, because similiar instances achieve quite good speedup with our algorithm. But it does not. Despite beeing as dense as europe its speedup is only 2.8. $g_{\min}$ and $g_{\max}$ are a magnitude larger than on the europe instance. The values are similiar than those from indochina-2004, which has atleast some speedup.

In a BSP modell, poorly distributed workload might cause long idle times on some PEs. Just looking at the average number of ghosts over all PEs is probably not the best idea. There might be many PEs with few ghosts and one PE with a large number of ghost. This PE might slow down all the others, and cause poor performance overall. We expect this for the USA-road instance. In the next chapter we test partitioned instances and find out,



(a) **(b)**

Figure 5.1: These figures show the speedup against the LinearTime algorithm on real world instances. Figure 5.1a shows those instances on which our algorithm performs reasonable well, Figure 5.1b those on which it performs poor.

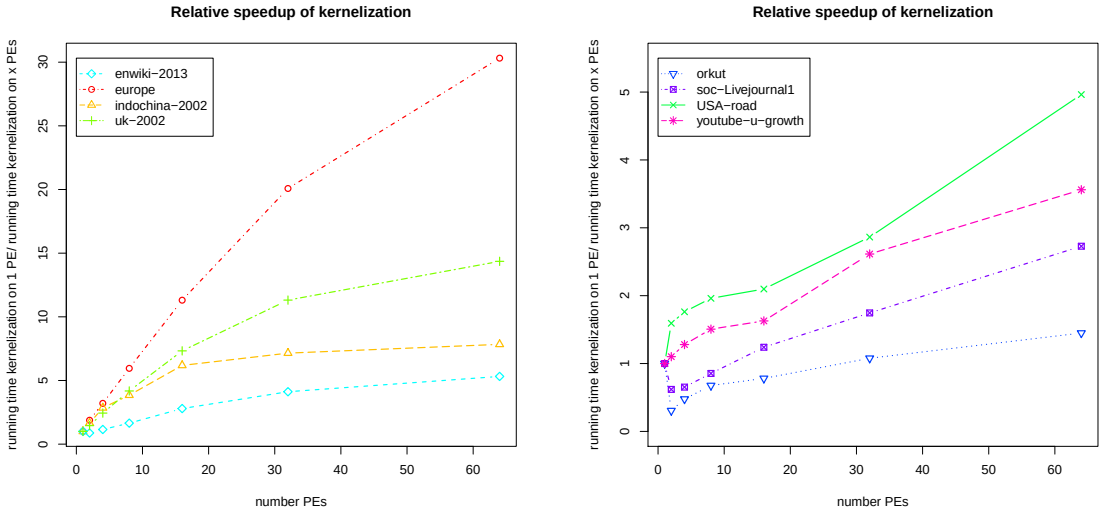
whether that is the case for the USA-road instance.

Before we do this, we take a look at the weak scaling of our algorithm on synthetic instances. Figure 5.4 shows the plot for the different synthetic instances. Figure 5.4a shows the gnm instance, Figure 5.4b the rhgd8 instance, Figure 5.4c the rhgd4 instance and Figure 5.4d the rgg instance.

For the gnm instance the kernelization does nothing. The rhg instance shows that using kernelization as a preprocessing step leads to a greater running time. For none of these instances our algorithm scales. Solely the rgg instance shows some scalability. We are able to calculate a MIS on an instance with 2^{26} vertices in 0.16 seconds (with kernelization) and 0.14 seconds (without kernelization). An instance with 2^{20} vertices requirest 0.01 and 0.007 seconds.

The reason for the poor performance is the same as for the real world instances. The average number of ghosts is just too high. It is between 0.11 (on 64 PEs) and 0.64 (on 4 PEs) for gnm, between 0.03 (on 64 PEs) and 0.37 (on 2 PEs) for rhg4 and between 0.04 (on 64 PEs) and 0.45 (on 2 PEs) for rhg8. For rgg this values has magnitude 10^{-4}.

Still, for all instances we manage to find a MIS for graphs with 2^{26} vertices in less than 10 seconds.



(a) **(b)**

Figure 5.2: These figure show the relative speedup of our kernelization algorithm on real world instances. Again figures on the left show instances on which our algorithm performs good, figures on the right those on which it performs poor.

5.3 Impact of Partitioning

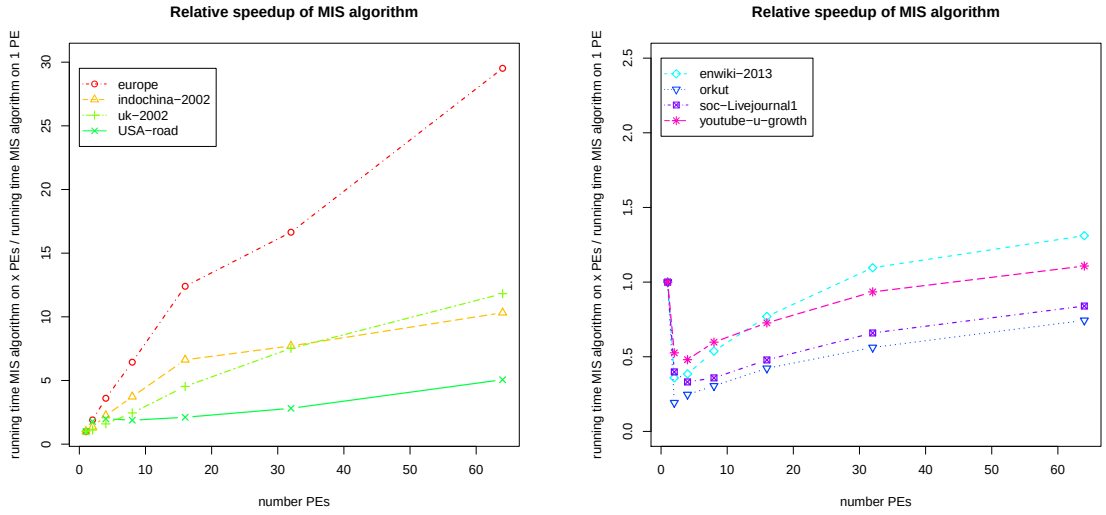
Since we have a problem with instances that have a large number of ghost vertices, we could try to reduce this number by partitioning our instances. We evaluate, how the running time for a partitioned instance compares to the unpartitioned instance. We compare the relative speedup of the kernelization and the relative speedup of the MIS algorithm of unpartitioned and partitioned instances. Figure 5.5 shows this for 4 and 64 PEs. Further Table 5.4 summarizes the speedups on different PEs.

Partiioning speeds up most of the instances. On some significant speedup is achieved.

The USA-road instance has partitioned a speedup of 27 over LINEARTIME, 9.6 relative speedup of kernelization and 7.2 relative speedup of the MIS algorithm. Also the number of average ghost vertices dropped significantly.

Orkut achieves a speedup of 2.7 over LINEARTIME, 2.4 relative speedup of the kernelization and no speedup for the MIS algorithm. This shows, that even partitioning reduces ghost vertices ($g_{\min} = 0.1$, $g_{max} = 0.35$ for partioned orkut), on particular dense instances the number might still be large. So, partitioning definitely improves here but still has its limitations.

Partitioning might also slow some instances down. It might be possible, that on some instances the partitioning algorithm increases the amount of distributed paths, because it tries to minimize the number of cutting edges overall.



(a) **(b)**

Figure 5.3: Relative speedup of our MIS algorithm on real world instances. Like previously figures on the left show instances on which our algorithm performs well, figures on the right those on which it performs poor.

5.4 The Effect of Kernelization

As mentioned earlier our algorithm finds smaller kernels than the LINEARTIME algorithm. This difference is quite insignificant. Our experiments showed that our kernels are up to 0.04 percent smaller compared to the ones LINEARTIME computes. What is more interesting is the effect of kernelization on the size of the MIS. We therefore compare the size of a solution from our algorithm that uses kernelization as a preprocessing step with a solution that does not. Figure 5.6 shows a visualization for our test instances. We can see that kernelization improves the size of the maximal independent set up to 13 percent. This shows that kernelization is a useful technique to obtain high quality solutions.

The downside is, that kernelization generally slows down our MIS algorithm. Figure 5.7 shows this for 1, 4 and 64 PEs. For 1 PE the slowdown is between 0.2 and 0.81, for 4 PEs we achieve speedup between 0.11 and 2.2 and for 64 PEs it is between 0.03 and 1.91.

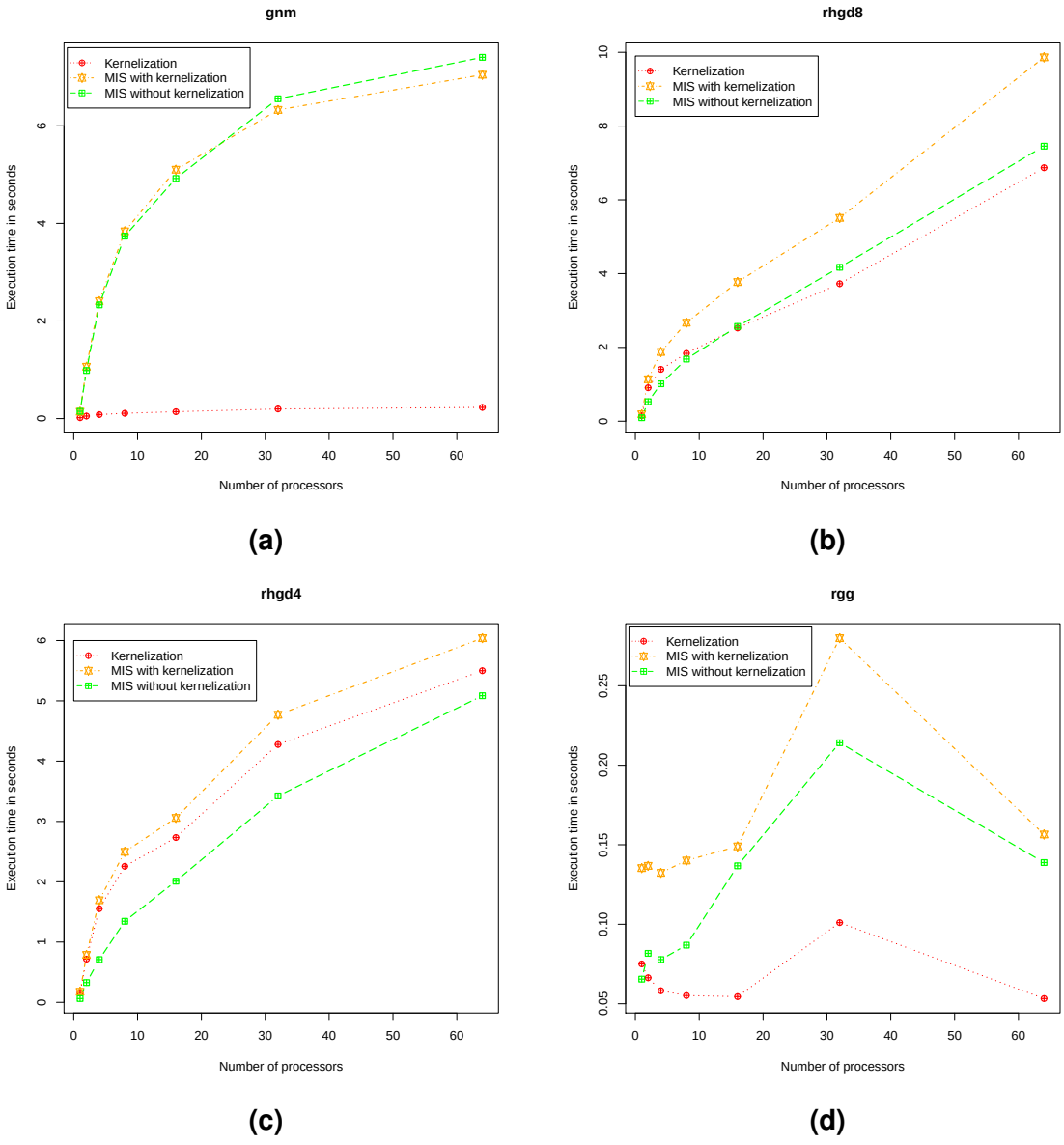


Figure 5.4: Weak scaling experiments on synthetic instances. Figure (a) shows the gnm instace, Figure (b) the rhdg8 instance, Figure (c) the rhgd4 instance and Figure (d) the rgg instance.

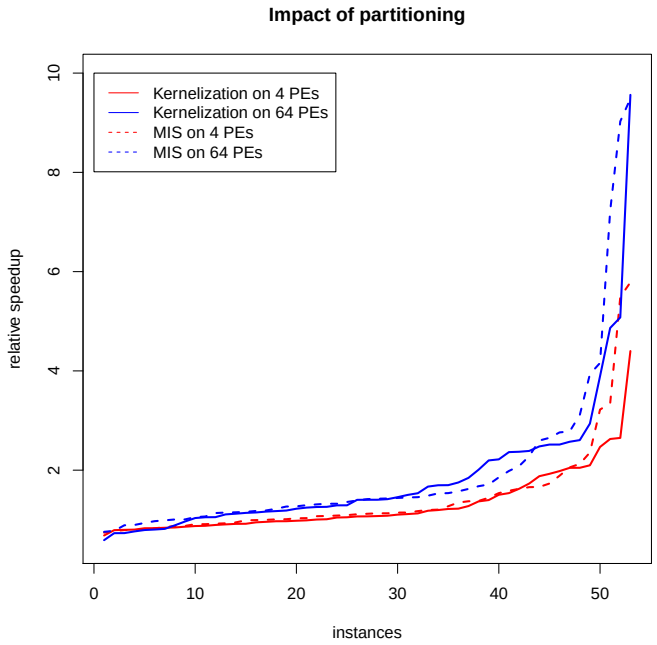


Figure 5.5: Speedup from partitioned over unpartitioned instances for kernelization and MIS algorithm on 4 and 64 PEs.

	Kernelization			MIS algorithm		
i	$\min S$	$\max S$	% with $S > 1$	$\min S$	$\max S$	% with $S > 1$
2	0.59	2.71	77	0.75	4.25	72
4	0.68	4.4	60	0.74	5.78	68
8	0.69	6.15	64	0.68	7.5	64
16	0.25	5.94	66	0.75	10.2	68
32	0.14	8.54	74	0.76	12	79
64	0.59	9.6	83	0.76	9.49	87

Table 5.4: This table summarizes the effect of partitioning. Speedup S is the running time on the partitioned instance over the running time of the unpartitioned instance. We examine it for kernelization and MIS algorithm without kernelization. The table shows for i PEs the maximum and minimum relative speedup of kernelization and MIS algorithm aswell as the percentage of instances which achieve speedup greater one.

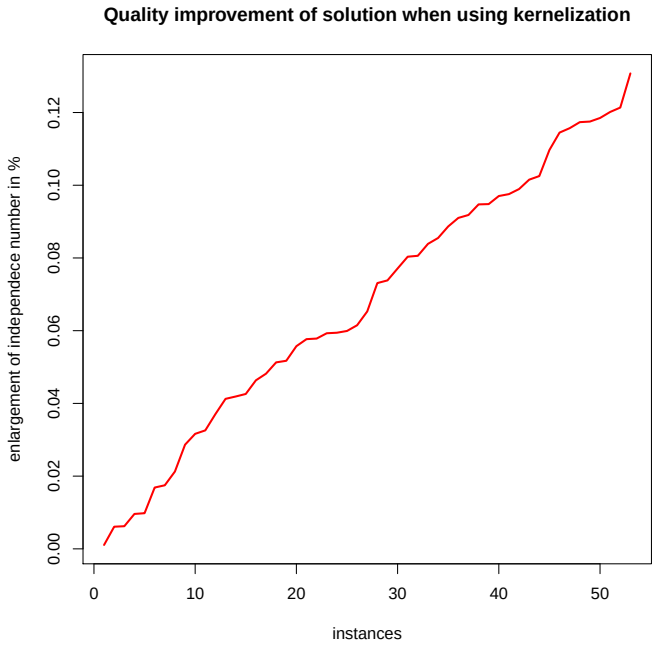


Figure 5.6: This plot displays the enlargmente of the maximal independent set when using kernelization as a preprocessing step.

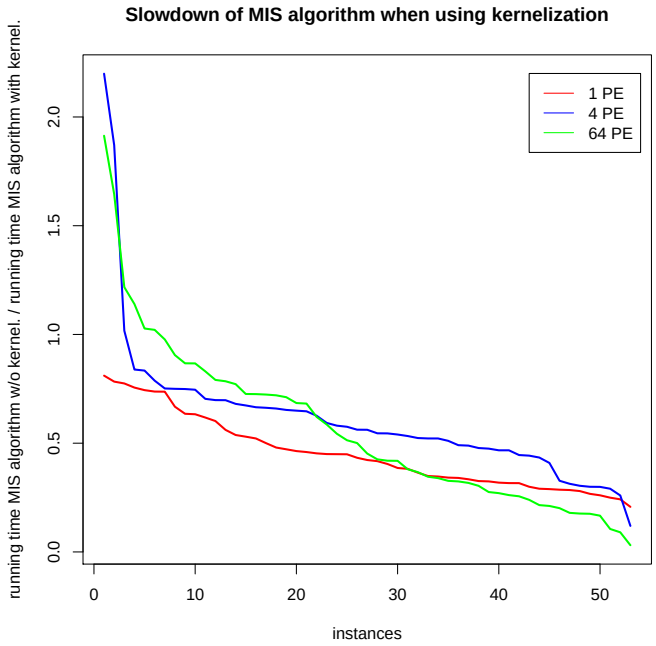


Figure 5.7: This plot shows the slowdown of the MIS algorithm with kernelization compared to the MIS algorithm without kernelization on 1, 4 and 64 PEs.

6 Discussion

6.1 Conclusion

We proposed a distributed algorithm that computes maximal independent sets using exact kernelization as a preprocessing step. Our kernelization removes degree one and two vertices. We showed, that the exact kernelization achieves speedups up to 30 on unpartitioned and up to 27 on partitioned instances. Our experiments showed, that kernelization improves the quality of the maximal independent set. Computing a maximal independent set on the exact kernel, rather than on the original instances leads to solutions that are up to 13 percent larger.

However, our experiments exposed some major problems with our implementation. It does not scale on dense graphs. Dense graphs tend to have a large number of ghost vertices. We hold a map containing the ghost vertices. If this map gets large, a lookup gets slow. This is the bottleneck of our algorithm. It slows down kernelization and MIS computation.

Partitioning reduces the number of ghost vertices and can therefore be used to improve the performance of some instances.

6.2 Future Work

Our implementation can be improved. We need to minimize the lookups of ghost vertices.

One way is to change the order in which messages are handled. Currently, the vertex removal messages get handled before the path reduction messages. Therefore, the case can occur in which we reiceive a vertex removal message for a vertex that is neither local nor ghost. A lookup is required.

By simply turning the order around, we would avoid the lookup and improve the performance.

Further, the communication volume and rounds can be reduced. We would not need to send termination messages for already deactivated vertices, if we extract the information from the already send vertex removal message. A technique called *shortcutting* can be used to decrease the number of communication rounds it takes to find a path from $O(n)$ to $O(\log n)$.

It might be a good idea to try a point-to-point communication model and compare it with the BSP model used in this implementation. An algorithm using such a communication

model could scale to a higher number of PEs.

A Results of Experiments

The next page shows a table containing our experimental results. For each graph it displays number of vertices ($|n|$) and number of edges ($|m|$), number of vertices and edges of the kernel graph (K_n and K_m), size of the MIS with ($|I|_K$) and without kernelization ($|I|$). Next, it shows the running time of our algorithm on one core (t_{seq}) and the LINEARTIME algorithm (t_{LT}). Further, it displays the speedup over LINEARTIME (S_{LT}), aswell as the relative speedup of the kernelization (S_{Rel}) and MIS algorithm (S_{MIS}). These speedups are the maximum among the speedups on $2^i, i \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ cores for that test instance. In comparision the table also shows the same speedups on the partitioned instance. The speedup S_{LT_p} is the speedup against LINEARTIME on the number of cores that achieved the highest speedup on the unpartitioned instace (S_{LT}). Likewise S_{Rel_p} and S_{MIS_p} show this for relative speedup of kernelization and MIS algorithm on partiioned instances. Notice that a 1 in S_{Rel} or S_{MIS} indicate that our algorithm does not scale.

name	$\|n\|$	$\|m\|$	K_n	K_m	$\|I\|_K$	$\|I\|$	t_{seq} in s	t_{LT} in s	S_{LT}	S_{Rel}	S_{MIS}	S_{LT_p}	S_{Rel_p}	S_{MIS_p}
amazon-2008	735323	3523472	392224	1839029	280140	252959	0.12	0.077	3.9	6	4.1	4.7	1.2	1.6
asia.osm	11950757	12711603	598919	873542	5953823	5175365	1.2	0.85	5.5	7.9	12	21	3.9	2.6
as-skitter	1696415	11095298	235645	792036	1150201	1057504	0.45	0.31	1.4	2.1	1.1	3.1	2.2	2
ca-AstroPh	18772	198050	7843	66764	6438	5867	0.0035	0.0031	0.89	1	1	0.89	1	1
ca-CondMat	23133	93439	6233	22017	9397	8366	0.0029	0.0031	1.1	1	1.3	1.1	1	1.9
ca-GrQc	5242	14484	681	7412	2446	2195	0.00049	0.001	2.1	1	1	2.1	1	1
ca-HepPh	12008	118489	2749	30838	4941	4440	0.002	0.0023	1.2	1	1	1.2	1	1
ca-HepTh	9877	25973	613	1937	4886	4326	0.0012	0.0016	1.3	1	1	1.3	1	1
cnr-2000	325557	2738969	153503	2195907	216785	207703	0.016	0.015	4	4.4	4.9	4	0.98	0.9
dblp-2010	986324	2738969	62567	196933	158972	146714	0.039	0.024	2.9	4.8	5	3.7	1.3	1.2
dblp-2011	986324	3353618	152923	443516	484638	443206	0.18	0.11	4.3	7.2	3.9	3.2	0.73	0.78
dewiki-2013	1532354	33093029	1330207	19026093	595005	584604	0.36	0.29	2	2.5	1	2.8	1.4	0.89
email-Enron	36692	183831	4100	8634	22150	21004	0.0043	0.0045	1	1	1	1	1	1
email-EuAll	265214	364481	190	348	246893	244472	0.013	0.012	1.4	1.6	1	3.6	2.5	1
enwiki-2013	4206785	91939728	1584888	10983363	2057379	1884823	3.2	2.2	3.6	5.3	1.3	3.7	1	1
eu-2005	862664	16138468	634829	12564597	412384	399340	0.062	0.054	1.9	2.2	3.5	3	1.5	1.7
europe	18029721	22217686	1927272	2829524	9134844	8062885	3	1.5	15	30	30	21	1.4	1.1
flickr-growth	2302925	15555041	16156	52627	1660981	1539579	0.79	0.63	1	1.3	1	1.2	1.1	1
flickr-links	1715255	22838276	12909	39532	1239833	1148299	0.51	0.4	0.96	1.2	1	1	1.1	1
hollywood-2011	2180759	114492816	1860190	95877463	495031	491949	0.26	0.25	1.5	1.5	1	1.7	1.2	1
in-2004	1382908	13591473	758915	10654024	844696	809849	0.11	0.096	2.7	3	12	2.9	1.1	1.3
indochina-2004	7414866	150984819	4632953	126255306	4402108	4258685	0.58	0.57	7.7	7.8	10	8.8	1.1	0.97
libimseti	220970	17233144	150051	7755904	109448	109330	0.12	0.11	1.9	2.2	1	1.1	0.59	1
ljournal-2008	5363260	49514271	280787	4990704	1475842	2680669	2.3	1.6	2.9	4	1.4	3.7	1.3	1.2
orkut	3072441	117185082	2824833	99724176	681133	674594	0.63	0.5	1.2	1.4	1	2.7	2.4	1
p2p-Gnutella24	26518	65369	9	14	19308	17020	0.0019	0.0031	1.6	1	1.4	1.6	1	1.4
p2p-Gnutella25	22687	54705	32	50	16665	14642	0.0019	0.0027	1.6	1.1	1.7	2.1	1.3	1.3
p2p-Gnutella30	36682	88328	33	49	27410	24189	0.0027	0.0039	1.5	1	1.7	2	1.3	0.9
p2p-Gnutella31	62586	147892	26	40	46891	41466	0.0048	0.0064	1.8	1.4	1.8	2.2	1.2	1.3
petster-carnivore	623766	15695166	353827	6093392	240257	233373	0.18	0.14	2	2.6	1	1.5	0.73	1
petster-friendships-cat-uniq	149700	5448197	102587	2518012	101844	100127	0.064	0.046	0.73	1	1	0.73	1	1
petster-friendships-dog-uniq	426820	8543549	292885	3762307	256572	251116	0.16	0.11	0.68	1	1	0.68	1	1
roadNet-CA	1965206	2766607	672936	1057935	913171	826667	0.37	0.13	5.8	16	17	15	2.5	2.3
roadNet-PA	1088092	1541898	348380	554299	508124	458553	0.19	0.069	6.2	17	23	7.8	1.3	1.5
roadNet-TX	1379917	1921660	403890	638503	648917	584708	0.24	0.088	6	16	27	13	2.2	1.5
soc-Epinions1	75879	405740	333	1358	26792	50399	0.01	0.0089	0.85	1	1	0.85	1	1
soc-LiveJournal1	4847571	42851237	271570	3497813	2613337	2365560	2.4	1.6	1.9	2.7	1	2.6	1.4	1
soc-pokec-relationships	1632803	22301964	952915	10250906	681802	643801	0.57	0.42	2.1	2.8	1	3.1	1.5	1
soc-Slashdot0902	82168	504230	759	2472	28166	53322	0.012	0.011	0.89	1	1	0.89	1	1
uk-2002	18520486	261787258	11745996	203424390	11015743	10607533	1.9	1.6	12	14	12	9.6	0.79	0.76
USA-road	23947347	28854312	2443552	3812247	12237030	10766756	4.4	2.5	2.8	5	5.1	27	9.6	7.2
web-BerkStan	685230	6649470	441248	4882719	374111	358176	0.052	0.033	0.71	1.1	4.7	0.099	0.14	1.4
web-Google	875713	4322051	321016	1848291	499983	459812	0.15	0.11	1.9	2.8	1	9.8	5.1	1
web-NotreDame	325729	1090108	82093	575251	246697	235263	0.027	0.03	1.7	1.5	2.5	4.9	2.9	2.8
web-Stanford	281903	1992636	170019	1361696	150388	142675	0.037	0.022	0.65	1.1	1	1.7	2.7	4.2
wiki-Talk	2394385	4659565	76	133	1169110	2323980	0.17	0.17	1.1	1.1	1	1.6	1.5	1
wiki-Vote	7115	100762	0	0	4866	4423	0.0015	0.0017	1.1	1	1	1.1	1	1
youtube	1134890	2987623	1182	2270	857897	808421	0.18	0.13	1.5	2.1	1	1.9	1.3	1.2
youtube-links	1138499	9376594	1193	2290	860324	810572	0.18	0.13	1.8	2.5	1.1	2	1.1	1.3
youtube-u-growth	3223643	9376594	4335	8577	2383408	2242095	0.66	0.54	2.9	3.6	1.1	2.2	0.77	0.94
zhishi-baidu-internallink	2141300	17014946	13730	30957	1503342	1410894	0.76	0.6	0.88	1.1	1	1.6	1.8	1
zhishi-baidu-relatedpages	415641	2374044	62919	422072	268581	255643	0.069	0.042	0.62	1	1.1	0.62	1	2.7
zhishi-hudong-internallink	1984484	14427382	31054	245515	1479193	1390558	0.62	0.56	0.89	1	1	0.89	1	1

Bibliography

[1] 9th dimacs implementation challenge - shortest paths.

[2] Youtube network dataset – KONECT, April 2017.

[3] Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, ALENEX '15, pages 70–81, Philadelphia, PA, USA, 2015. Society for Industrial and Applied Mathematics.

[4] Noga Alon, László Babai, and Alon Itai. A fast randomized parallel algorithm for the maximal independent set problem. 7:567–583, 12 1986.

[5] David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. 2014.

[6] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem1. *Algorithmica*, 29(4):610–637, Apr 2001.

[7] 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.

[8] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[9] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[10] S. Butenko and W.E. Wilhelm. Clique-detection models in computational biochemistry and genomics. *European Journal of Operational Research*, 173(1):1 – 17, 2006.

[11] Sergiy Butenko, Panos Pardalos, Ivan Sergienko, Vladimir Shylo, and Petro Stetsyuk. Finding maximum independent sets in graphs arising from coding theory. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, SAC '02, pages 542–546, New York, NY, USA, 2002. ACM.

[12] Lijun Chang, Wei Li, and Wenjie Zhang. Computing a near-maximum independent set in linear time by reducing-peeling. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1181–1196, New York, NY, USA, 2017. ACM.

[13] J. Dahlum, S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. Accelerating Local Search for the Maximum Independent Set Problem, February 2016.

[14] Thomas A. Feo, Mauricio G. C. Resende, and Stuart H. Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42(5):860–878, 1994.

[15] Jeremy T. Fineman, Calvin Newport, Micah Sherr, and Tonghe Wang. Fair maximal independent sets. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 712–721, Washington, DC, USA, 2014. IEEE Computer Society.

[16] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *J. ACM*, 56(5):25:1–25:32, August 2009.

[17] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*, 2018.

[18] Peter Artymiuk P.J. Gardiner, Eleanor Willett. Graph-theoretic techniques for macromolecular docking. In *Journal of Chemical Information and Computer Science*, 2000.

[19] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[20] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 315–324, New York, NY, USA, 1987. ACM.

[21] Mark Goldberg and Thomas Spencer. Constructing a maximal independent set in parallel. *SIAM J. DISC. MATH*, 2:322–328, 1989.

[22] Andrea Grosso, Marco Locatelli, and Wayne Pullan. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *Journal of Heuristics*, 14(6):587–612, December 2008.

[23] Demian Hespe, Christian Schulz, and Darren Strash. *Scalable Kernelization for Maximum Independent Sets*, pages 223–237.

[24] Hasan Heydari, S. Mahmoud Taheri, and Kaveh Kavousi. Distributed maximal independent set on scale-free networks. *CoRR*, abs/1804.02513, 2018.

[25] T. Kanewala, M. Zalewski, and A. Lumsdaine. Parallel asynchronous distributed-memory maximal independent set algorithm with work ordering. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 52–61, Dec 2017.

[26] Richard M. Karp and Avi Wigderson. A fast parallel algorithm for the maximal independent set problem. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 266–272, New York, NY, USA, 1984. ACM.

[27] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed time-dependent contraction hierarchies. In Paola Festa, editor, *Experimental Algorithms*, pages 83–93, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[28] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[29] M Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 1–10, New York, NY, USA, 1985. ACM.

[30] Alan Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, 2009.

[31] D. Puthal, S. Nepal, C. Paris, R. Ranjan, and J. Chen. Efficient algorithms for social network coverage and reach. In *2015 IEEE International Congress on Big Data*, pages 467–474, June 2015.

[32] Mauricio G. C. Resende, Diogo V. Andrade, and Renato Werneck. Fast local search for the maximum independent set problem. In *International Workshop on Experimental Algorithms (WEA)*, volume 5038, pages 220–234, Provincetown, MA, May 2008. Springer.

[33] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[34] Johannes Schneider and Roger Wattenhofer. A log-star distributed maximal independent set algorithm for growth-bounded graphs. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 35–44, New York, NY, USA, 2008. ACM.

[35] R. Tarjan and A. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[36] Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. *CoRR*, abs/1312.6260, 2013.