Distributed Kernelization Techniques for the Maximum Weight Independent Set Problem

Master's Thesis of

Jannick Borowitz

At the KIT Department of Informatics
ITI – Institute of Theoretic Infortmatics, Algorithm Engineering

First examiner: Prof. Dr. Peter Sanders
First advisor: M.Sc. Matthias Schimek
Second advisor: M.Sc. Ernestine Großmann

01. August 2024 – 31. January 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Acknowledgement

I want to use this opportunity to thank Ernestine Großmann and Matthias Schimek for their support and ideas in the weekly meetings. Moreover, I would like to thank Prof. Dr. Sanders for the chance to work on this thesis. Finally, I would like to thank my friends and family for their support.

Abstract

This work contributes novel distributed kernelization algorithms for the NP-complete Maximum Weight Independent Set problem (MWISP). There are many fields of interest such as the map-labeling and vehicle routing problem where the problem corresponds to the MWISP. Problem instances of these applications are often modeled as vertex-weighted, undirected graphs. The considered graphs can be very large with millions of vertices and edges. To obtain (near)-optimal results in practice, a key technique called *kernelization* is used by heuristic and exact solvers. Kernelization algorithms use data reduction rules to reduce the problem size while maintaining optimality. Thus, they can be used as preprocessing or as a subroutine in a more complex solver. However, the graphs can require more memory than a single machine can possibly offer. Moreover, testing and applying many data reduction rules can make up a significant portion of the overall running time. Therefore, one is interested in representing a graph in a distributed memory machine where each processing element (PE) stores only a subgraph. Moreover, a graph may already be stored in distributed memory. Many of the data reductions act locally because they are tested and applied only on a small subgraph. This motivates their use in a distributed manner. A key challenge is to apply reductions so that no conflicts arise between the PEs. To the best of our knowledge, we are the first to develop distributed data reduction rules and kernelization algorithms for the MWISP. We propose two communication variants for synchronizing the reduction progress: A synchronous algorithm called KaDisReduS and asynchronous algorithm called KaDisReduA. They are used for preprocessing to then apply novel distributed greedy algorithms to the reduced graph. We conduct strong and weak scaling experiments for up to 1 024 cores. For the strong scaling experiments, we consider artificial and real world graphs with up to 50 Mio. vertices and 54 Mio. edges. The median number of vertices in the reduced graph increases from 5 % to 25 % compared to sequential kernelization. This increase can be mitigated by partitioning the graph with dKaMinPar. At this scaling, KaDisReduA outperforms KaDisReduS by a factor of 1.42 in terms of running time and has a relative speed-up of 90 on average. In comparison with the reduce-and-peel solver HtWIS, KaDisReduA combined with the greedy algorithm reports a speedup of 15 on average with a maximum of 170 on a grid graph. While HtWIS reaches a solution quality of 99.6 %, our approach reaches a solution quality of 96.85 %, when compared against the best found solutions.

Zusammenfassung

Diese Arbeit stellt neue verteilte Kernelisierungsalgorithmen für das NP-vollständige Maximum Weight Independent Set Problem (MWISP) vor. Es gibt viele Anwendungsbereiche wie etwa das Map-Labeling (deutsch: Kartenbeschriftungsproblem) und das Vehicle Routing Problem (deutsch: Routenplanungsproblem), wo das Problem dem MWISP entspricht. Problem Instanzen dieser Anwendungen werden oft als knotengewichtete, ungerichtete Graphen modelliert. Diese Graphen können sehr großsein mit Millionen Knoten und Kanten. Um in der Praxis (fast) optimale Lösungen zu erhalten, wird eine wichtige Technik namens Kernelisierung von heuristischen und exakten Lösern benutzt. Kernelisierungsalgorithmen benutzen Datenreduktionsregeln um die Problemgröße zu reduzieren, sodass die Optimalität nicht verletzt wird. Dadurch können sie in einem Vorverarbeitungschritt oder als eine Subroutine in einem komplexeren Löser. Jedoch, kann das Testen und Anwenden der Regeln einen großen Anteil der Gesamtlaufzeit ausmachen. Daher ist man daran interessiert, den Graphen auf Maschinen mit verteiltem Speicher zu repräsentieren, wo jedes Prozesselement (PE) nur einen Teilgraphen speichert. Darüberhinaus kann ein Graph bereits verteilten Speicher vorliegen. Viele der Datenreduktionen agieren lokal, indem sie auf nur auf einem kleineren Teilgraphen getestet und angewendet werden. Dies motiviert die Datenreduktionen im verteilten Kontext anzuwenden. Die Hauptschwierigkeit besteht darin die Datenreduktionen so anzuwenden, dass keine Konflikte entstehen zwischen den PEs. Soweit uns bekannt ist, sind wir die Ersten, die verteilte Datenreduktionen und Kernelisierungsalgorithmen für MWISP entwickeln. Wir schlagen zwei Varianten für die Kommunikation des Reduktionsfortschritts: einen synchronen Algorithmus namens KaDisReduS und einen asynchronen Algorithmus namens KaDisReduA. Sie werden als Vorverarbeitungschritt genutzt, um dann neue verteilte Greedyalgorithmen auf dem reduzierten Graphen anzuwenden. Wir führen starke und schwache Skalierungsexperimente mit bis zu 1 024 Kernen durch. Für die starken Skalierungsexperimente betrachten wir künstliche und reale Graphen, die bis zu 50 Mio. Knoten und 54 Mio. Kanten besitzen. Die mediane Anzahl an Knoten in den reduzierten Graphen steigt von 5% auf 25 % verglichen mit der sequenziellen Kernelisierung. Dieser Anstieg kann verringert werden, indem der Graph mit dKaMinPar partitioniert wird. Auf dieser Skalierung ist KaDisReduA um einen Faktor 1.42-mal schneller als KaDisReduS und hat einen relativen Speedup von 90 im Mittel. Im Vergleich mit dem Reduce-And-Peel Löser HtWIS erhalten wir mit KaDisReduA, kombiniert mit dem Greedyalgorithmus, einen Speedup von 15 im Mittel und 170 im Maximum auf einem Gittergraphen. Während HtWIS eine Lösungsqualität von 99.6 % erzielt, erreicht unser Ansatz 96.85 % verglichen mit den besten gefundenen Lösungen.

Contents

1. Introduction

We start this thesis by motivating the MWISP problem and the benefits of a distributed kernelization algorithm. Afterward, we give an overview of our contributions and the structure of this work.

1.1. Motivation

The Maximum Weight Independent Set problem (MWISP) applies to many fields of interest such as vehicle routing [11] or map-labeling [6, 18]. In the vehicle routing problem, we consider *routes* which are defined over properties such as the assigned driver, and the loads of the vehicle. The proposed routes can conflict with each because the same driver was assigned for multiple routes but can operate one route. In addition, the routes are rated. Now, we are interested in a subset of routes which are not in a conflict where the sum of their ratings is as large as possible. This discrete optimization problem can be modeled as a vertex-weighted, undirected graph. The vertices correspond to the routes with the ratings as weights and the pairwise conflicts can be represented by an edge. Then, a *Maximum Weight Independent Set* (MWIS) is a solution for the vehicle routing problem. An MWIS is a subset of vertices of maximum weight while no two verties are adjacent.

Finding an MWIS can be challenging because MWISP is NP-complete. Depending on the application, a maximal independent set of large weight, i.e., where no vertex can be added anymore, is good enough. Most heuristic and exact solvers use *kernelization* algorithms as a prepocessing or in-between as a subroutine. Kernelization algorithms test and apply data reduction rules exhaustively to reduce the problem size while they maintain optimality. To be more specific, the MWIS can be reconstructed given an MWIS for the reduced graph, the so-called *kernel*.

Solving the MWISP can also be difficult because the graph is too large to fit into the memory of one machine. Then the graph may be represented in a distributed memory machine where each process stores a part of the graph, i.e., a subgraph. It is also conceiveable that the graph is already stored in distributed memory. To find (near)-optimal solutions for this case, we develop distributed kernelization algorithms where data reduction rules are applied in a distributed fashion. Many existing data reduction rules are tested and modify only a small subgraph. This property is convenient to transfer and apply the rules in the distributed memory model.

1.2. Contribution

To the best of our knowledge, we propose the first distributed kernelization algorithms for the MWISP. We transfer data reduction rules to the distributed memory model. We devise two variants which differ only in their communication approach. KaDisReduS uses synchronous communication to synchronize and exchange the reduction progress between (processing elements) PEs using blocking, irregular all-to-all coomunication. The reduction progress is exchanged if all processes are locally out of work. KaDisReduA follows an asynchronous approach with the help of the message buffer queue library message-queue which was intrduced by Sanders and Uhl for distributed algorithms for the TriangleCounting problem [36]. It allows to control the message sizes with an additional parameter δ. Both can be used in combination with the graph partitioner dKaMinPar by Sanders et al. [35] before or in-between reducing the graph because a good partitioning can be crucial for the reduction impact for large number of PEs. Moreover, we devise a separate greedy algorithm for each communication scheme, called GreedyS and GreedyA. Finally, we present new *reduce-and-greedy* algorithms for the MWISP by applying them to the reduced graph after kernelization.

1.3. Overview

The remainder of this thesis is structured as follows. In Chapter 2 we introduce important notation, particularly the MWISP and the distributed memory model which form the basis of this work. Moreover, we give an overview of the related work in Chapter 3. In Chapter 4 we present our distributed data reduction rules and distributed kernelization algorithms. Afterward, we introduce the new maximal independent set solvers for in Chapter 5. In experiments in Chapter 7, we evaluate the impact and performance of our reductions in strong and weak scaling experiments. We given the implementation details in Chapter 6. Furthermore, we compare our distributed independent set solvers against the reduce-and-peel solver HtWIS by Gu et al.[23]. Finally, we conclude our work and outline future work in Chapter 8.

2. Fundamentals

In this chapter we introduce some definitions including a formal definition of the Maximum Weight Independent Set problem. Moreover, we outline the machine model and how we represent the input in it.

2.1. Preliminaries

Consider a simple, undirected, weighted graph $G = (V, E, \omega)$ with vertices V, edges $E \subseteq 2^V$, and $w : V \to \mathbb{N}_{>0}$ which assigns every vertex of V a weight. We extend ω to sets, i.e., $w(U) \coloneqq \sum_{u \in U} \omega(u)$ for $U \subseteq V$.

Let $N(v) \coloneqq \{u \in V : \{v, u\} \in E\}$ be the *open neighborhood* of a vertex $v \in V$ which we simply refer to as the *neighborhood* of v. Further, the *closed neighborhood* of v is defined as $N[v] \coloneqq N(v) \cup \{v\}$. For a subset of vertices $U \subseteq V$ the definitions generalize to the *neighborhood* $N(U) \coloneqq \bigcup_{u \in U} N(u) \setminus U$ and the *closed neighborhood* $N[U] \coloneqq N(u) \cup U$ of U. We define the *closed distance d-neighborhood* inductively as $N^d[U] \coloneqq N[N^{d-1}[U]]$ with $N^0[U] \coloneqq N[U]$. The open *distance d-neighborhood* is defined as $N^d(U) \coloneqq N(N^{d-1}(U))$. The degree of a vertex $v \in V$ is defined as $\deg(v) \coloneqq |N(v)|$.. Morover, $\Delta(G) \coloneqq \max \deg(v) : v \in V$ is the maximum degree in G.

Often we consider subgraphs that are *induced* on a subset of vertices $U \subseteq V$. Given G and U the *induced subgraph* is defined as $G[U] \coloneqq (U, E \cap (2^U))$. Induced subgraphs can be the result of modifying the graph G where we remove a vertex $v \in V$ or a set of vertices $U \subseteq V$. For simplicity, we denote these operations as $G - v \coloneqq G[V \setminus \{v\}]$ and $G - U \coloneqq G[V \setminus V]$, respectively.

We call $C \subseteq V$ a *clique* in G if all vertices of C are pairwise adjacent. Moreover, $v \in V$ is *simplicial* if $N[v]$ forms a clique in G.

An *independent set* $\mathcal{I}$ of G is a subset of V such that every pair of vertices $(u, v) \in \mathcal{I} \times \mathcal{I}$ is not adjacent in G, i.e., it holds $\{u, v\} \notin E$. Further, $\mathcal{I}$ is *maximal* if there is no vertex in $V \setminus \mathcal{I}$ that can be added to obtain an independent set that contains $\mathcal{I}$. We denote $\mathcal{I}$ a *maximum independent set* (MIS) if there is no independent set of G that contains more vertices. An independent set $\mathcal{I}$ of G is called a *maximum weight independent set* (MWIS) if there exists no independent set of G with larger weight in G, i.e., there exists no independent set $\mathcal{I}$ with $\omega(\mathcal{I}') > w(\mathcal{I})$. We denote the weight of a maximum weight independent set of G as $\alpha(G)$. Given G, the Maximum Weight Independent Set problem (MWISP) asks for an MWIS of G. Analogously, the

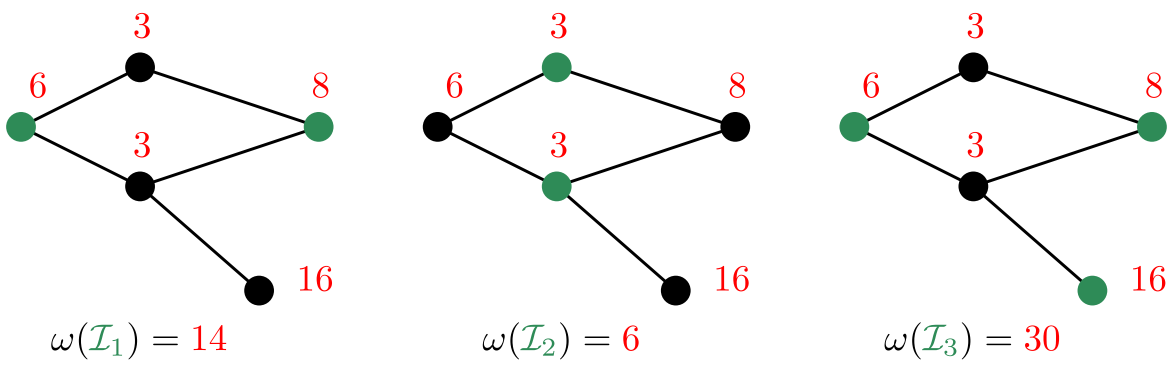
3
6
8
3
16
$\omega(\mathcal{I}_1) = 14$
3
6
8
3
16
$\omega(\mathcal{I}_2) = 6$
3
6
8
3
16
$\omega(\mathcal{I}_3) = 30$

Figure 2.1.: This figure shows three different independent sets (highlighted in green) for the same weighted undirected graph. Weights are highlighted in red. Independent set $\mathcal{I}_1$ is not maximal since the vertex with weight 16 can be added to the independent set. Therefore, it is also not an MWIS. Independent set $\mathcal{I}_2$ is an example for a maximal independent set that is not an MWIS (e.g., $\mathcal{I}_1$ has larger weight than $\mathcal{I}_2$). On the right side, an MWIS for the graph is shown.

Maximum Independent Set problem asks for an MIS. In the unweighted case, $\alpha(G)$ is also denoted as the *independence number of* G. Note that an MWIS is always maximal. Figure 2.1 gives an example for a non-maximal independent set, a maximal independent set, and an MWIS for the same graph.

A maximal independent set can be found in polynomial time in the number of vertices with simple greedy algorithms. However, deciding whether a (maximal) independent set is an MWIS is a well-known NP-complete problem [16]. The MWISP is closely related to other NP-complete problems. The weighted problem generalizes the unweighted problem because we can simply assign one as the weight for every vertex. If $\mathcal{I}$ is an MWIS of G, then $V \setminus \mathcal{I}$ is a *minimum weight vertex cover* of G. Moreover, in the complementary graph $G_c = (V, (2^V) \setminus E)$ of G, an MWIS of G is a *maximum weight clique*.

In addition to $G = (V, E, \omega)$, we often consider *blocks* of vertices $\Pi := (V_1, \dots, V_k)$, for $k \in \mathbb{N}_{>0}$, that partition V. More precisely, the blocks of Π *partition* V if $V_1 \cup \dots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for all $i \neq j$. The edges of $E_{ij} := \{\{u, v\} \in E : u \in V_i,\ v \in V_j\}$ are denoted *cut edges* because they connect two blocks V_i and V_j, for $i \neq j$. Incident vertices of the cut-edges are denoted *border vertices*.

Sometimes we are interested in the *balanced graph partitioning problem*. In this case, we are given $G = (V, E)$, $k \in \mathbb{N}_{>0}$, a cost function $c : V \to \mathbb{N}_{>0}$ that analogously generalizes to sets as ω does, and some $\varepsilon > 0$. The objective is to find a partitioning Π that minimizes the edge-cut $\text{cut}(\Pi) := \sum_{i<j} |E_{ij}|$ while it fulfills the *balance constraint* $c(V_i) \leq (1 + \varepsilon)c(V)/k$, for every $i \leq k$, $i \in \mathbb{N}$.

2.2. Machine Model and Distributed Memory Graph

Throughout this work, we consider the *distributed memory model* as described in [37, pages 265-267]. It consists of p indexed processing elements (PEs) which are able to communicate with each other in a full-duplex, single-ported network. Moreover, each PE has its own memory while no memory is shared.

We consider a simple, undirected, weighted graph. It is represented as a weighted, directed graph in the *adjacency array format* with a $1D$ partitioning of the vertices. This balances the vertices or edges. More precisely, each undirected edge $\{u, v\}$ is represented by two directed edges (u, v) and (v, u). Each PE is given a weighted subgraph (one block of the partitioning $(V_1, \ldots, V_k)$) with consecutive vertices. PE i *owns* the vertices V_i. They are denoted the *local vertices* of PE i. For a vertex $v \in V$, $\mathrm{rank}(v)$ maps v to its *owner*, i.e., $\mathrm{rank}(v) = i \iff v \in V_i$. Further, we assume the vertices have consecutive vertex identifiers for consecutive blocks, i.e., it holds for all $v, u \in V$ with $v < u$ that $\mathrm{rank}(v) \leq \mathrm{rank}(u)$. The subgraph contains all the direct edges (v, u) where the respective PE owns the tail vertex. The border vertices are duplicated at the respective PEs because an undirected edge is represented by two directed edges. These replicates are denoted *ghost vertices* (or simply *ghosts*) at the according PEs. In the perspective of PE i, we refer by border vertices to the local border vertices and ghosts. The local border vertices are also referred to as the *interface vertices of* PE i. The set of ghosts at PE i is denoted $V_i^g \subseteq V$. Further, the weight $\omega(v)$ of a vertex $v \in V$ is stored at its PE $\mathrm{rank}(v)$ and is denoted $\omega_{\mathrm{rank}(v)}(v) := \omega(v)$. To facilitate taking about the PEs which own ghosts in the neighborhood of $v \in V_i$, we define the set of *adjacent PEs* of v as $\mathcal{R}_i(v) := \{\mathrm{rank}(u) : u \in N_i(v) \cap V_i^g\}$.

We often need to know weights of ghosts. Therefore, we assume the weights are replicated with the interface vertices for the ghosts, i.e., $w_i(u)$ denotes the replicated weight for a ghost $u \in V_i^g$. Thus, to represent G in distributed memory, we store at PE i the undirected weighted subgraph $G_i := (V_i, V_i^g, E_i, \omega_i)$ with vertices $\overline{V}_i := V_i \cup V_i^g$, edges $E_i := \{\{u, v\} \in E : u \in \overline{V}_i, v \in V_i\}$ and weights $\omega_i(v)$ for $v \in \overline{V}_i$. We give an example in Figure 2.2 for a subgraph G_i stored at PE i.

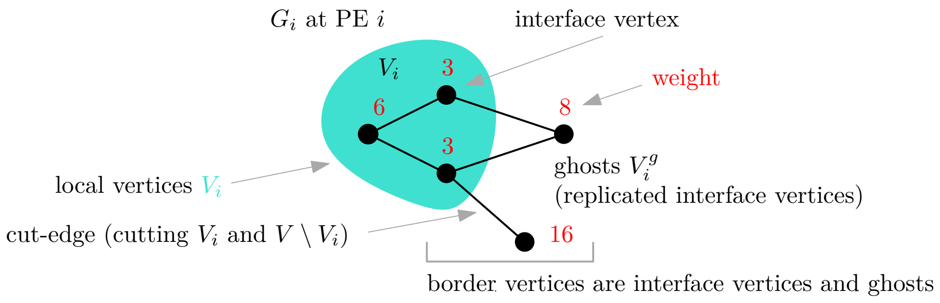
G_i at PE i
interface vertex
V_i
3
weight
6
8
3
local vertices V_i
ghosts V_i^g
(replicated interface vertices)
cut-edge (cutting V_i and $V \setminus V_i$)
16
border vertices are interface vertices and ghosts

Figure 2.2.: An example for an undirected weighted subgraph $G_i = (V_i, V_i^g, E_i, \omega_i)$ of G stored at PE i to represent G in the distributed memory.

2.3. Notation of Data Reduction Rules

For the data reduction rules, we follow the new standardized scheme and formulations of Großman et al. [20]. Each rule is identified by a name and followed by a condition under which the rule can be applied for the currently considered graph G. Afterward, it is stated how to modify the graph in order to obtain the *reduced graph* G'. Moreover, the *offset* states the difference of the optimal solution weight $\alpha(G) - \alpha(G')$ between both problems. Last, it is explained how to *reconstruct* an MWIS I for G once an MWIS I' for G' is obtained.

Distributed Data Reduction Rules. The notation slightly changes compared to sequential data reduction rules. In the distributed memory model we apply the reductions to the locally stored subgraph G_i. As reduced graph, we state the modified local graph G_i' which translates to a modification in G. The reduction offset is stated with respect to the global reduced graph G'. If the reduction rule supports to include interface vertices, it is possible that conflicting vertices are proposed for the solution. In this case, we state an *include proposal* which adds the proposed vertex to a set of include proposals $\tilde{\mathcal{I}}_i$. When the PEs receive the include proposal by another PE, they can check locally for a conflict. The data reduction rules state how to resolve the conflict.

3. Related Work

In this chapter we give an overview of the related work. It is structured as follows: We start with sequential approaches in Section 3.1 because the first kernelization techniques for MWISP have been developed in combination with a sequential solver. Then, we describe parallel and distributed algorithms and kernelization techniques in Section 3.2 from the literature.

3.1. Sequential Approaches

This work focuses on existing data reduction rules and kernelization techniques. Most of the data reduction rules were published as part of a novel solver. We outline the different types and ideas of data reduction rules together with the solvers. Many of them will be introduced formally in the next chapter where we discuss them in detail and use them in our distributed memory model. For a full overview of the existing solvers and data reduction rules, we refer the reader to a recent survey by Großman et al. [20]. They also cover solvers of the related problems such as the Minimum Weight Vertex Cover and Maximum Weight Clique. For a better overview, we differentiate exact algorithms in Section 3.1.1 and heuristic algorithms in Section 3.1.2.

3.1.1. Exact Algorithms

There exist multiple ways to find an MWIS. For example, one can state the problem simply as an *Integer Linear Program* (ILP) [10]. A more common approach for solving a combinatorial optimization problem is to use a sophisticated *branch-and-bound* solver. Several branch-and-bound solvers have been proposed for the weighted problem [39, 5]. The idea is to search for an exact solution by recursively splitting the search spaces. The search space is split by branching on the solution state of a vertex v. Then, one can consider the search space with all the solutions including the branching vertex v and then the solution space which excludes v. A Branching rule picks the next branching vertex. This rules follows a heuristic which aims to reduce the number of branching steps. In practice, branch-and-bound improves upon an exponential running time and a naive brute-force approach in practice by pruning the search space. Pruning is applied if no better solution can be found in a branch, i.e., the best known solution is equal to an upper bound for the considered branch. An upper bound for the weighted problem is given by weighted clique covers [39] which can be efficiently computed.

Kernelization. To tackle (sparse) graphs with up to millions of vertices under reasonable resource limitations, kernelization has proven to be a successful algorithmic technique [2, 3, 29]. Originally, kernelization was used for parameterized decision problems which are problems that have an additional parameter as input, e.g., the solution size. With the help of kernelization, one can show for some parameterized problems that they can be decided in polynomial time in the input size while the parameter is fixed. Intuitively speaking, such problems can be efficiently solved as long as the parameter is small. Such parameterized problems belong to the complexity class *fixed parameter tractable* (FPT).

Roughly speaking, the idea of kernelization is to try to apply so-called data reduction rules to the current problem instance to obtain a smaller equivalent problem. These rules aim to reduce vertices and/or edges from the graph or reduce the solution weight of the instance. The resulting problem instance, the *reduced graph*, is equivalent in the sense that an exact solution can be reconstructed for the graph given an exact solution for the reduced graph and vice versa. If no reductions are applicable anymore, the *reduced graph* is called a *kernel*. In the context of fixed-parameter-tractability, an FPT problem always yields a kernel that is bounded polynomially only in size by the parameter while the kernelization maintains an FPT running time.

While MVCP with the solution size as parameter is an FPT problem, MISP and MWISP are probably not [12]. Nonetheless, in practice they benefit from data reduction rules because they efficiently remove many vertices and edges from the graph or decrease the solution weight. However, from a practical perspective kernelization as an algorithmic toolbox just started to receive more attention in the recent years as pointed out by Abu-Khzam et al. [2]. They survey recent advances of transferring data reduction rules to practical algorithms including problems which are unlikely to be fixed-parameter-tractable.

Branch-and-Reduce. A very successful paradigm using kernelization is the *branch-and-reduce* scheme which builds on top of the concept of branch-and-bound. It exhaustively applies data reduction rules to shrink the problem size before doing a branching step to improve upon the exponential running time. As a result, branching is only necessary if no reductions are applicable anymore. A widely known and used branch-and-reduce solver for the weighted problem is KaMIS_wB&R by Lamm et al. [29].

The *Critical Weighted Independent Set* reduction by Butenko and Trukhanov [7] was the very first rule for the weighted problem. It is an effective data reduction rule used by KaMIS_wB&R. This rule tries to find a subset of an MWIS by solving a minimum cut problem for a bipartite graph [7]. While this rule acts more in a global fashion, i.e., takes the whole graph in account, the rules introduced by Lamm et al. [29] act more locally. They derive rules from so-called novel *meta reduction rules* which only read and modify neighborhoods up to a certain distance of a considered vertex. They are called meta reduction rules because they require to find an MWIS for a sub problem, e.g., for an induced subgraph of the neighborhood. In practice this is not very efficient because the resulting sub problem can be as hard as the original problem. To that end, they derive rules, which make use of upper bounds with small computational cost for the sub problems. In practice, these upper bounds are good enough to successfully reduce parts of the graphs. KaMIS_wB&R only uses the meta reduction rules if efficient rules do not apply anymore while it is not in a recursion of solving a sub problem. Additionally, the running time for solving the sub problem is bound by a time limit.

To obtain a kernel, the reduction rules are applied iteratively in certain order to prefer more efficient and effective rules. In each step, a rule is tested for all candidate vertices. At the end of each step, one starts over with the first rule if progress was made, i.e.,. a reduction applied for some candidate. Otherwise, the next rule is considered if one is left. To prevent testing a reduction rule for all vertices all over again, a common technique is *dependency checking* where vertices are only considered if their neighborhood has changed. This technique is especially helpful for data reductions which apply locally.

Gellner et al. [17] extended the data reduction portfolio of KaMIS_wB&R with a weighted version of the *struction rule* by Ebenegger et al. [13]. This rule transforms the problem into an equivalent problem of decreased solution weight. However, this comes at the cost of a possibly increased graph size. Therefore, it is referred to as a *transformation rule* instead of a data reduction rule. Once this rule is applied, data reduction rules become ideally applicable again so that the reduced problem size is overall smaller. Their experiments with their solver Struction underline the effectiveness of their approach. Struction solves significantly more graphs than KaMIS_wB&R and is often multiple orders of magnitude faster.

Figiel et al. [14] investigated the reduction order for MVCP and point out that different orders of reduction applications can lead to different kernels. Since one is particularly interested in a small kernel one may try to undo reduction applications and try to apply a (different) rule to (different) vertices. They go a step further and propose so-called *backward rules* which yield an (increased) equivalent problem instance. Applying these rules can be thought of as undoing the corresponding *forward rule* (data reduction rules decreasing the problem size). These backward rules are then used in new kernelization techniques that aim to find overall smaller kernels. They propose two new kernelization algorithms which are called Find-And-Reduce and Inflate-Deflate. Both algorithms apply forward rules to obtain a kernel and then try to improve upon the kernel size using backward rules. Find-And-Reduce tries to search the search space for sequences of reduction applications to find an overall smaller kernel. Whereas Inflate-Deflate is more of a random walk trying to apply backward rules randomly to ideally find a smaller kernel. A key challenge is to control the huge search space of backward rule applications. The first kernelization algorithm restricts the search by allowing only a maximum number of reduction steps, applies rules in a subgraph, and stops once some reduction order is found that yields a smaller reduced graph. Their second algorithm, (Local) Inflate-Deflate works in two phases. First, they *inflate* the graph by randomly (locally) applying backward rules in the current kernel until the resulting graph size reaches a threshold size. Afterward, the kernelization algorithm tries to *deflate* the problem by exhaustively applying forward rules in a randomized fashion. Their experiments underline that both approaches are capable to find even smaller reduced graphs for various graph families compared to reduction rules by Akiba and Iwata [3].

Utilizing kernelization also gives rise to new branching vertex selection strategies. Hespe et al. [24] proposed a novel strategy for the MISP problem that prefers branching on vertices which entail further reduction applications in the reduced graph. Their conducted experiments show promising results: The new strategy is at least as good in terms of running time as the heuristic that branches on maximum degree vertices.

There exists also other branch and reduce solvers [41, 31]. They use data reduction rules which act more globally. For example *One Vertex Cut* by Xiao et al. [41] reduces connected components by searching articulation points. If an articulation point $v \in V$ is removed from the graph decays into multiple connected components. One can then build a case distinction for one of the resulting connected components C by computing an MWIS for it and for $C \setminus N(v)$.

Recently, another branch-and-reduce reduce solver for MWIS was proposed by Xiao et al. [40] for a theoretical analysis. They use novel data reduction rules in their solver which reduce short paths and cycles. With the help of the measure-and-conquer method they show an improved worst-case running time for sparse graphs. More precisely, their algorithm runs in time $\mathcal{O}^*(1.1443^{(0.624x-0.872)n'})$ time where n' is the number of vertices of degree at least two and x the average degree of these vertices while using polynomial space.

3.1.2. Heuristic Algorithms

Greedy algorithms for finding a maximal independent set appear throughout the literature and are often used for initial solutions in more sophisticated local-search solvers [10, 22, 29]. For the weighted case well-known greedy heuristics are weight and weight_diff which rate vertices with $w(v)$ and $\omega(N(v)) - w(v)$, respectively [23]. Then, the vertices are included if free, i.e., adjacent vertices are not in the solution yet, in descending order of their rating. For example, KaMIS_wB&R by Lamm et al. [29] adds free vertices to the solution ordered by their weight if a time limit is reached. Sometimes, better results are achieved if the rating is adaptive, i.e., is re-evaluated whenever a vertex is assigned a solution status [10].

In the following we given an overview of heuristic algorithms for the unweighted and weighted problem. They combine and follow different schemes such as *iterated-local-search* (ILS), *memetic*, and *reduce-and-peel*. We will point out that most state-of-the-art in-exact solvers make heavy use of data reduction rules in order to obtain high-quality solutions.

Various local-search techniques exist for the weighted and unweighted problem which often find high quality solutions. Starting with ARW by Andrade et al. [4] which is an iterated local search (ILS) solver for the unweighted problem. They introduced a successful technique called $(1, 2)$-swaps that improves a solution by *swapping* two vertices into the solution by removing one from it.

Dahlum et al. [9] build on top of this technique by utilizing kernelization. Their first approach, called KerMIS, first reduces the graph and then applies local-search to find a good solution for the kernel. The second algorithm, called OnlineMIS, reduces simplicial vertices in an online fashion while scanning the graph to perform local-search. Further, both approaches prune high degree vertices which can be thought of as an in-exact reduction to speed up local-search by removing them from the graph and excluding them from a solution because they are unlikely to be part of it. Pruning high-degree vertices is helpful for efficiently obtaining high-quality solutions for scale-free graphs.

Nogueira et al. [34] generalize the vertex swapping technique to the weighted case, to so-called $(w, 1)$-swaps and $(1, 2)$-swaps, and propose the iterated local-search solver HILS. Here, the idea is to swap a vertex into the solution by removing its w neighbors the solution if it improves the solution. For both kind of swaps an improving swap can be found in linear time or deduced that non exists anymore. These general local-search techniques are often combined with tabu-mechanisms and perturbations steps to overcome local maxima.

For some graph applications, e.g., the vehicle routing (VR) instances by Dong et al. [11], data reductions are so far not very effective due to their structure [10]. To that end, Dong et al. [10] proposed a novel iterated local search algorithm called METAMIS that utilize a metaheuristic called GRASP along with the $(w, 1)$-, $(1, w)$- and $(2, w)$-swaps, and AAP-moves. METAMIS outperforms HILS on the VR instances.

Beside these local-search techniques, a common heuristic solver is the *reduce-and-peel* solver HtWIS by Gu et al. [23]. In contrast to local-search solvers it does not improve upon an initial solution but aims to find a single high-quality solution incrementally. The idea of reduce-and-peeling is to apply a kernelization algorithm whenever possible until no reduction rule is applicable anymore; and only then an in-exact decision regarding the solution is done. For the in-exact step, a vertex is *peeled* which ideally is unlikely to be (not) part of a solution. This vertex selected with a heuristic and excluded from (included into) the solution. A solution is obtained once the graph is empty. HtWIS outperforms the iterated local-search solver HILS in the experiments conducted by Gu et al. [23] while often computing solutions in milliseconds. Their solver especially benefits from their newly proposed data reduction rules. They present so-called *low-degree* reductions which target vertices of degree one and two which all can be fully reduced. For the remainder of the graph they propose the *Basic Single Edge* reduction and *Extended Single Edge* reduction. These rules reduce vertices which are not part of at least one optimal solution. The *Basic Single Edge* reduction generalizes the *Weighted Domination* rule by Lamm et al. [29].

For the unweighted problem exist multiple evolutionary algorithms, e.g., EvoMIS by Lamm et al. [28] and ReduMIS by Lamm et al. [30]. The latter uses data reduction rules in combination with EvoMIS. More recently, Großman et al. [22] proposed an advanced memetic algorithm for the weighted problem, called m^{2}wis+s. It repeatedly applies a kernelization algorithm, determines a solution for the kernel with an evolutionary algorithm, and then uses this (high-quality) solution to prune vertices which are unlikely to be in an optimal solution. The last step re-opens up the search space so that further parts of the graph can be reduced. Their algorithm shows its full potential especially when run for a long time focusing on high quality solutions. To reduce the graph, they make use of data reductions from KaMIS_wB&R et al. [29], HtWIS et al. [23]. They also use Struction [17] within the evolutionary approach.

Very recently, Großman et al. [21] contributed a new metaheuristic which they implemented in a new iterated local-search solver called CHILS. CHILS is comprised of two phases. In the first phase, k best solutions are determined or improved with a baseline local-search algorithm. Then they apply the metaheuristic which is called *Concurrent Difference-Core Heuristic*. Therefore, a subgraph induced on all the vertices is built where the k best solutions disagree, the so-called *Difference-Core*. The local-search is then applied on this difference-core. Afterward, the best solution for the difference-core can be embedded back into the maintained best solutions which yields new (best) solutions. Some of the new solutions replace the old ones. Both steps can be repeated for a given time limit. Moreover, the authors apply CHILS to the kernel. For the kernelization step they use novel data-reduction rules which are not yet covered in the survey by Großmann et al. [20]. Roughly speaking, if one vertex contains the neighborhood of another vertex, then edges can somtimes be added or removed between them. A key challenge regarding kernelization algorithm is to decide whether more involved data reduction rules of higher computational cost are effective for a given vertex. So far, the set of reduction rules and their sequence in the iterative reduction process were heuristically chosen, mostly following the computational cost and their effectiveness in experiments. Therefore, the authors propose a neural network architecture based on graph neural networks to obtain a so-called screening algorithm that is capable of efficiently deciding whether a reduction rule is likely to be effective for a vertex. Overall, the authors outperform state-of-the-art algorithms,e.g., obtain new best solution for many VR instances, and perform well especially on large graphs. In the context of parallel algorithms, CHILS is interesting as it is also a simple shared-memory algorithm. All threads only read from the graph, but do not modify it. Further, each thread of the p threads maintains a solution ($p = k$), solves the same difference core while diversifying the local-search algorithm among the threads.

3.2. Distributed and Parallel Approaches

3.2.1. Greedy Algorithms

A simple parallel randomized algorithm for finding a maximal independent set is Luby's algorithm [32]. It was originally proposed for an *exclusive read exclusive write parallel random access machine* (EREW PRAM). To obtain a maximal independent set, the idea is to randomly choose a subset of vertices X into the solution in parallel. This set might not be an independent set yet because vertices of X can be adjacent. Therefore, each adjacent pair of vertices of X is considered in parallel. Conflicts are resolved by always removing the neighbor of larger degree from X. Afterward, the independent set and the neighbors are removed from the graph. Repeating theses steps exhaustively, until the graph becomes empty, outputs a maximal independent set.

There exist multiple distributed greedy algorithms proposed for the LOCAL model. Roughly speaking, this model follows a vertex-centric approach where each process is assigned one vertex and can communicate processes of the neighbors. Such an algorithm then works in rounds, so-called super steps, with communication in between to globally synchronize their state. This model is often used for theoretical analysis regarding the communication volume and needed synchronization steps.

Peleg [1] proposed such an algorithm to find a maximal independent set. Every vertex is assigned a global unique identifier. The algorithm is based on the observation that a vertex can be added to a solution without any conflict if it has the largest identifier in its neighborhood. In a first step, a vertex joins the solution if it has the largest identifier among the undecided neighbors. Then, the included vertices notify their neighbors that they will not be part of a solution (excluded). Another synchronization is needed so that the excluded vertices notify their neighbors about their exclusion. These three steps are now repeated until all vertices are decided.

Wang et al. [38] used this distributed greedy algorithm, denoted as DisMIS. They rank the vertices globally by their degree and break ties with the rank of the process. The intuition is that vertices of smaller degree may be more likely to be part of an optimal solution. They propose another distributed greedy algorithm called OIMIS which computes the same solution but aims to improve running time by mitigating the order dependency of the vertices. Therefore, OIMIS initially takes all vertices into a solution and at every further super step their solution status is re-evaluated. The re-evaluation works as follows: A vertex is only removed from the solution if it has a higher ranking neighbor that is (still) part of the solution; otherwise it (re-)joins the solution. If the solution status changes the neighbors are notified so that they can re-evaluate their solution status in the next super step. If no vertex is notified for re-evaluation a maximal independent set is found. Their experiments indicate that OIMIS finds a solution by up to a factor 2 faster than DisMIS.

For the weighted case, a distributed greedy algorithm was proposed by Joo et al. [27] which was specifically designed for the application of wireless scheduling. Their idea is to build a maximal independent set incrementally by selecting a subset of vertices X in each iteration that have a sufficiently large weight compared to their neighbors. Moreover, the vertices must be eligible to join the solution, i.e., they must not be adjacent to vertices of the partial solution. For X, they compute a maximal independent set and the solution vertices to the solution.

3.2.2. Kernelization in Parallel

Although we are not aware of any distributed-memory or parallel kernelization-based algorithms for the MWISP, a shared-memory and a distributed-memory approach for the unweighted problem exists. They are in their idea most closest to our work.

A shared-memory approach. Hespe et al. [25] were the first to propose a parallel kernelization approach. They present a lock-free algorithm for shared-memory machines for the unweighted problem. The high-level idea of their approach is to partition the graph using a graph partitioner so that each thread can afterwards apply reductions at its own vertex-disjoint block simultaneously. This idea is based on the observation that the data reduction rules by Akiba and Iwata [3] and Butenko et al. [8] act very locally. Thus, the data reductions can be applied blockwise, i.e., each thread applies reductions within its own block, given a shared graph representation. Close to the border, reductions can only be applied with restrictions since they cannot modify the neighborhood or vertices at other blocks without risking race-conditions. Therefore, the graph is partitioned, minimizing cut-edges, with the shared-memory parallel graph partitioner ParHIP Meyerhenke et al. [33] before applying reductions. Further, they only apply reduction where they do not include border vertices to prevent conflicts regarding the solution. Nonetheless, border vertices can be excluded (and eventually removed from the graph) by marking them as removed. As a result, adjacency lists at other blocks are not touched. Beside the blockwise reductions, they propose a shared-memory parallelize the *Critical Independent Set* reduction which acts globally taking the full (reduced) graph into account. It requires to find a maximum bipartite matching for a bipartite graph to solve a linear program. They apply this reduction after the blockwise reduction were applied exhaustively. The resulting reduced graph is called a *quasi-kernel* since reductions rule may still apply at the border.

A distributed approach. George et al. [19] proposed a distributed kernelization algorithm for the unweighted problem. The input graph is partitioned into p vertex-disjoint blocks where each of the p processes owns exactly one block. The key idea of their kernelization algorithm is to reduce all degree one and two vertices exhaustively with the help of the reduction rules by Chang et al.ComputingANeaChang2017. Roughly speaking, these rules require to find degree-one vertices, degree-two paths, and degree-two cycles in the graph. If such structures lay local within a block they can be reduced without further communication. In the event that border vertices are reduced vertex removal messages are written the respective processes can update their stored subgraph. To identify and find such paths and cycles across multiple PEs they use a label propagation approach. They propagate properties of the path segments with cut-edges to the other PEs to reduce them. All these messages are buffered, as path segments are encountered and border vertices are reduced, and eventually communicated with an collective MPI_Alltoall. This synchronization step is started once all processes are locally out work.

They propose a heuristic solver to find a maximal independent set by applying a randomized greedy algorithm, similar to Luby's algorithm, to the computed kernel to find a maximal independent set. Their experiments, conducted with up to 64 cores on real world instances and artificial graphs covering many graph families, indicate speed-ups of a factor up to 30 on 64 cores compared with a linear reduce-and-peel algorithm. While their approach works well in terms of running time for street networks, their approach does not scale for many others. More involved experiments show that these issues possibly arise due to slow look-ups of ghost vertices from received update messages. Moreover, bad workload distributions might cause long idle times on many processes, e.g., if many processes have only a few ghost vertices while some processes have to deal with updates messages for many ghosts.

4. Distributed Kernelization

The following chapter presents the details of our distributed kernelization techniques for the MWISP. Therefore, we repeat important data reductions rules [23, 29] in Section 4.1 which we later transfer to our distributed kernelization approach. In an example in Section 4.2, we demonstrate how sequential data reduction rules can be applied in the distributed memory model. In Section 4.3, we introduce the distributed data reductions rules. These are rules that each process can apply to its locally stored subgraph in the distributed memory model. All these rules are part of a local kernelization algorithm in Section 4.4. In Section 4.5, we give the details of our dynamic graph data structure representing the reduced graph in distributed memory. Applying the local kernelization algorithm requires communication to update the reduction progress at the border. Therefore, Section 4.6 gives the details of the message types holding the reduction progress. To exchange and update the sent and received reduction progress we propose a synchronous and an asynchronous communication approach. By using the local kernelization algorithm with each of them, we obtain two distributed kernelization algorithms KaDisReduS and KaDisReduA, respectively.

4.1. Reduction Rules

As mentioned in Chapter 3, many data reduction rules for sequential kernelization of the MWISP exist. In a recent survey, Großman et al. [20] give a full overview over existing rules. We cover a variety of effective data reduction rules that exploit locality and are crucial for a strong reduction impact for many real world instances.

We start by repeating the data reduction rules of our interest. The proofs of correctness can be found in the cited works. For the data reduction rules, we follow the new standardized scheme and formulations of Großman et al. [20]. We introduce the scheme in Section 2.3.

All the following rules make use of the following operations to reduce the graph and re-construct a solution. A vertex $v \in V$ can be *excluded* which says that is not part of $\mathcal{I}$ and that it can be removed from G. There are also cases, where a vertex $v \in V$ is *included* which says it is added $\mathcal{I}$. If v is added to $\mathcal{I}$, its neighbors are excluded and $N[v]$ can be removed from the graph. Moreover, sometimes vertices are *folded*. This operation changes the weight of vertices, removes vertices from G, or replaces vertices by a new vertex. In this case, the solution reconstruction depends on an MWIS for the reduced graph in order to decide whether or which folded vertices join the solution.

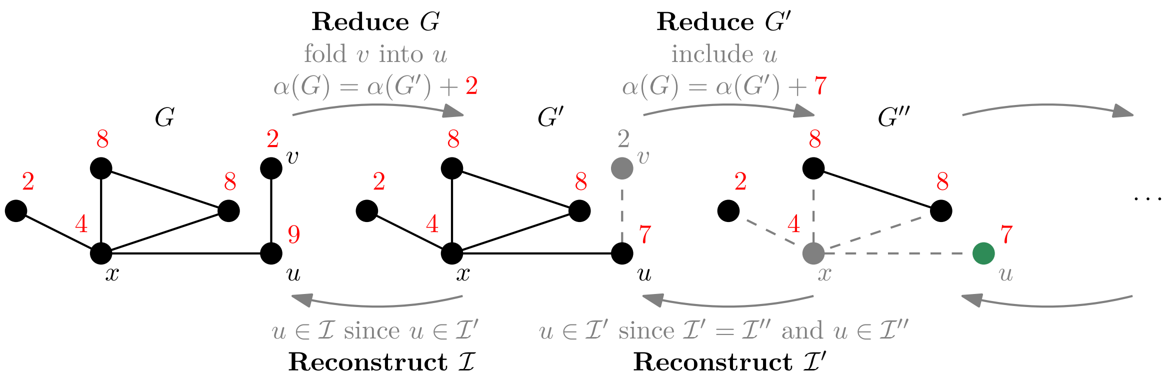
Reduce G
fold v into u
$\alpha(G) = \alpha(G') + 2$
Reduce G'
include u
$\alpha(G) = \alpha(G') + 7$
G
G'
G''
2
8
8
4
2
v
9
u
x
2
8
8
4
2
v
7
u
x
2
8
8
4
7
u
x
. . .
$u \in \mathcal{I}$ since $u \in \mathcal{I}'$
Reconstruct $\mathcal{I}$
$u \in \mathcal{I}'$ since $\mathcal{I}' = \mathcal{I}''$ and $u \in \mathcal{I}''$
Reconstruct $\mathcal{I}'$

Figure 4.1.: This figure shows an example where Degree One by Gu et al. [23] is applied two times to reduce vertices in G. Weights are highlighted in red, reduced vertices and edges are darkgray in the reduced graph, and included vertices are green.

First, we consider rules that target vertices of small degree because they can be efficiently tested due to their small neighborhood. The first rule allows one to reduce all vertices of degree one. To give some intuition for the rule, we show an example in Figure 4.1.

Reduction 4.1.1 (Degree One by Gu et al. [23]).
Let $v, u \in V$ with $N(v) = \{u\}$.

- *If $\omega(v) \geq \omega(u)$: include v.*

Reduced Graph	$G' = G - N[v]$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	$\mathcal{I} = \mathcal{I}' \cup \{v\}$

- *If $\omega(v) < \omega(u)$: fold u and v into a new vertex v'.*

Reduced Graph	$G' = G - N[v] + v'$ *with* $N(v') = N(u)$ *and* $\omega(v') = \omega(v) - \omega(u)$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	*If* $v' \in \mathcal{I}'$, *then* $\mathcal{I} = \mathcal{I}' \setminus \{v'\} \cup u$, *else* $\mathcal{I} = \mathcal{I}' \cup \{v\}$

Further, there exist rules to completely reduce vertices of degree two. To reduce them, one distinguishes between the following two cases: A degree-two vertex $v \in V$ either forms a *triangle* with its two neighbors so that all three are pairwise adjacent or $N[v]$ has a *V-shape*, i.e., both neighbors are not adjacent. A triangle is merely a special case of a clique where v is a simplicial vertex. We will repeat data reduction rules that generalizes to cliques and reduce triangles. To that end, we repeat only the V-Shape by Gu et al. [23].

Reduction 4.1.2 (V-Shape by Gu et al. [23]).
Let $v \in V$ be a degree-two vertex with two non-adjacent neighbors $x, y \in V$. Without loss of generality, assume $\omega(x) \leq \omega(y)$.

- *If $\omega(v) < \omega(x)$: fold v into a new vertex v'.*

Reduced Graph	$G' = G[(V \cup \{v'\}) \setminus \{v\}]$ *with* $N(v') = N(x) \cup N(y)$ *and set* $\omega(x) = \omega(x) - \omega(v), \omega(y) = \omega(y) - \omega(v), \omega(v') = \omega(v)$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	*If* $x \in \mathcal{I}'$ *or* $y \in \mathcal{I}'$, *then* $\mathcal{I} = \mathcal{I}' \cup \{v\}$, *else* $\mathcal{I} = \mathcal{I}'$

- *If $\omega(x) \leq \omega(v) < \omega(y)$: fold v into x and y.*

Reduced Graph	$G' = G - v$ *with* $N(x) = N(x) \cup N(y)$ *and set* $\omega(y) = \omega(y) - \omega(v)$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	*If* $x, y \in \mathcal{I}'$, *then* $\mathcal{I} = \mathcal{I}' \cup \{v\}$, *else* $\mathcal{I} = \mathcal{I}'$

- *If $\omega(y) \leq \omega(v)$ and $\omega(x) + \omega(y) \leq \omega(v)$: include v.*

Reduced Graph	$G' = G - N[v]$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	$\mathcal{I} = \mathcal{I}' \cup \{v\}$

- *If $\omega(y) \leq \omega(v)$ and $\omega(x) + \omega(y) > \omega(v)$: fold v, x, y into a new vertex v'.*

Reduced Graph	$G' = G[(V \cup \{v'\} \setminus \{v, x, y\}$ *with* $N(v') = (N(x) \cup N(y))$ *and* $\omega(v') = \omega(x) + \omega(y) - \omega(v)$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	*If* $v' \in \mathcal{I}'$, *then* $\mathcal{I} = \mathcal{I}' \cup \{v\} \setminus \{v'\}$, *else* $\mathcal{I} = \mathcal{I}' \cup \{v\} \setminus \{v'\}$

Note that in the first case no vertices are reduced but the solution weight decreases for the cost of new edges. In the second case the reduced graph also might have more edges but at least the number of vertices decreases.

The next rules applies to vertices of arbitrary degree. They include a vertex if its weight is at least as large as the weight of an MWIS in its neighborhood.

Reduction 4.1.3 (Heavy Vertex by Lamm et al. [29]).
Let $v \in V$ with $\omega(v) \geq \alpha(G[N(v)])$, then include v.

Reduced Graph	$G' = G - N[v]$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	$\mathcal{I} = \mathcal{I}' \cup \{v\}$

Note that the latter rule requires to find an MWIS for an induced subgraph. In practice, easier special cases are tested first, e.g., the next rule called Distributed Neighborhood Removal. The rule approximates an MWIS for the induced neighborhood graph by summing up all weights in the neighborhood.

Reduction 4.1.4 (Neighborhood Removal by Lamm et al. [29]).
Let $v \in V$ with $\omega(v) \geq \omega(N(v))$, then include v.

Reduced Graph	$G' = G - N[v]$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	$\mathcal{I} = \mathcal{I}' \cup \{v\}$

If $v \in V$ is simplicial, i.e., its closed neighborhood forms a clique in G, then an MWIS of the induced neighborhood graph contains only a single vertex. This observation gives rise to the following data reduction rule.

Reduction 4.1.5 (Simplicial Vertex by Lamm et al. [29]).
Let $v \in V$ be simplicial with maximum weight in its neighborhood, i.e., $\omega(v) \geq \max\{\omega(u) : u \in N(v)\}$, the include v.

Reduced Graph	$G' = G - N[v]$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	$\mathcal{I} = \mathcal{I}' \cup \{v\}$

If a simplicial vertex $v \in V$ does not have maximum weight in its neighborhood, we can still reduce some vertices of the clique and decrease the weight of the remaining vertices.

Reduction 4.1.6 (Simplicial Weight Transfer by Lamm et al. [29]).
Let $v \in V$ be simplicial, let $S(v) \subseteq N(v)$ be the set of all simplicial vertices. Further, let $w(v) \geq w(u)$ for all $u \in S(v)$.

- *If $\omega(v) \geq \max\{\omega(u) : u \in N(v)\}$, then use Simplicial Vertex by Lamm et al. [29].*
- *Else, fold v into $N(v)$.*

Reduced Graph	$G' = G - X$ *with* $X := \{u \in Nv : \omega(u) \leq \omega(v)\}$ *and* *set* $w(u) \leftarrow w(u) - w(v)$ *for all* $x \in N(v) \setminus X$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	*If* $\mathcal{I}' \cap N(v) = \emptyset$, *then* $\mathcal{I} = \mathcal{I}' \cup \{v\}$, *else* $\mathcal{I} = \mathcal{I}'$

So far, most rules reduce vertices by including a vertex into the solution. The following rules are often used in sequential kernelization algorithms to exclude vertices once the previous rules were tested exhaustively. Intuitively speaking, these rules can exclude a vertex if there are vertices in the neighborhood which always can be swapped into the solution to obtain a solution which is at least as good.

Reduction 4.1.7 (Domination by Lamm et al. [29]).
Let $u, v \in V$ be adjacent vertices with $N[u] \subseteq N[v]$. If $\omega(v) \leq w(u)$, then exclude v.

Reduced Graph	$G' = G - v$
Offset	$\alpha(G) = \alpha(G')$
Reconstruction	$\mathcal{I} = \mathcal{I}'$

Later, Basic Single-Edge by Gu et al. [23] was introduced. They show that Domination by Lamm et al. [29] is generalized by Basic Single-Edge by Gu et al. [23]. Basically, they allow u to have neighbors that are not necessarily neighbors of v, as long as they have sufficiently small weights.

Reduction 4.1.8 (Basic Single-Edge by Gu et al. [23]).
Let $u, v \in V$ be adjacent vertices with $\omega(N(u) \setminus N(v)) \leq w(u)$, then exclude v.

Reduced Graph	$G' = G - v$
Offset	$\alpha(G) = \alpha(G')$
Reconstruction	$\mathcal{I} = \mathcal{I}'$

Further, Gu et al. [23] introduce a rule which is called Extended Single-Edge by Gu et al. [23]. It allows to reduce common neighbors of u and v.

Reduction 4.1.9 (Extended Single-Edge by Gu et al. [23]).
Let $u, v \in V$ be adjacent vertices with $\omega(v) \geq \omega(N(v)) - \omega(u)$, then exclude $N(v) \cap N(u)$.

Reduced Graph	$G' = G - (N(v) \cap N(u))$
Offset	$\alpha(G) = \alpha(G')$
Reconstruction	$\mathcal{I} = \mathcal{I}'$

The last two rules fold a vertex $v \in V$ with an MWIS of the neighborhood. Either v or the vertices of the MWIS in the neighborhood are part of $\mathcal{I}$. This is the case, if there is only one MWIS in the neighborhood of larger weight than $\omega(v)$. A special case of it is Neighborhood Folding by Lamm et al. [29] because it only applies if the neighborhood is independent,i.e., no two vertices are adjacent. This case can be simply tested. The generalized rule, Generalized Neighborhood Folding by Lamm et al. [29], is given afterward. It needs to solve multiple MWISP for the neighborhood of the considered vertex v.

Reduction 4.1.10 (Neighborhood Folding by Lamm et al. [29]).
Let $v \in V$, and suppose that $N(v)$ is independent. If $\omega(N(v)) > \omega(v)$, but $\omega(N(v)) - \min\{\omega(u) : u \in N(v)\} < \omega(v)$, then fold v and $N(v)$ into a new vertex v'.

Reduced Graph	$G' = G[(V \cup \{v'\}) \setminus N[v]]$ *with* $N(v') = N(N(v))$ *and* $\omega(v') = \omega(N(v)) - \omega(v)$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	*If* $v \in \mathcal{I}'$, *then* $\mathcal{I} = (\mathcal{I}' \setminus \{v'\}) \cup N(v)$, *else* $\mathcal{I} = \mathcal{I}' \cup \{v\}$

Reduction 4.1.11 (Generalized Neighborhood Folding by Lamm et al. [29]).
Let $v \in V$, then

- *if $G[N(v)]$ contains only one independent set $\tilde{\mathcal{I}}$ with $\omega(\tilde{\mathcal{I}}) > \omega(v)$, fold v and $N(v)$ into a new vertex v'.*

Reduced Graph	$G' = G[(V \cup \{v'\}) \setminus N[v]]$ *with* $N(v') = N(N(v))$ *and* $\omega(v') = \omega(\tilde{\mathcal{I}}) - \omega(v)$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	*If* $v \in \mathcal{I}'$, *then* $\mathcal{I} = (\mathcal{I}' \setminus \{v'\}) \cup N(v)$, *else* $\mathcal{I} = \mathcal{I}' \cup \{v\}$

- *if for $u \in N(v)$ all independent sets in $G[N(v)]$ including u have less weight than $\omega(v)$, exclude u.*

Reduced Graph	$G' = G - u$
Offset	$\alpha(G) = \alpha(G')$
Reconstruction	$\mathcal{I} = \mathcal{I}'$

4.2. A first Example

All the reductions introduced in Section 4.1 have in common that they act locally. More precisely, they require and modify only the neighborhoods up to a small distance of the currently considered vertex. We can exploit this locality for our distributed machine model. If each process stores a vertex disjoint block, we can safely apply reductions blockwise. Each process can apply reduction simultaneously to its stored block as long as they are careful at the border. This property of locality was also exploit by parallel kernelization approaches for the unweighted problem [19, 25].

A process can also try to reduce vertices where it has border vertices in the neighborhood or is a interface vertex. However, when we reduce a vertex at the border, each process needs to ensure that this reduction can be applied independent of possible blockwise border modifications at other processes.

The example in Figure 4.2 illustrates this idea. In this example, the graph is stored distributed across three PEs. Each PE tries to apply reductions locally. The PE i can apply Neighborhood Removal by Lamm et al. [29] to v_i because $6 = \omega(v_i) \geq \omega(N(v_i)) = 6$. When it applies, it includes v_i and removes $N[v_i]$ from its locally stored subgraph. Even if the interface neighbors of v_i, x_i and x_j, were already excluded at another process, we can still include v_i because v_i still has a sufficiently large weight. From a global point of view, we do not have to remove any vertex at another process because all removed vertices are local vertices. However, we still removed cut-edges locally. Not communicating the removal of x_i and x_j is no issue if the adjacent PEs, e.g., PE i, do not reduce any local vertices demanding the existence of x_i or x_j. Nonetheless, we might want to notify the adjacent PEs of the excluded interface vertices, x_i and y_i, at some point so that the application of further reductions is possible.

At PE j we can apply Simplicial Weight Transfer by Lamm et al. [29] for the local vertex v_j because it forms a triangle (clique) with its neighbors. The weight of the interface neighbor x_j is decreased by $\omega(v_j) = 6$, from 8 to 2, and the interface neighbor

u_j is excluded because $5 = w(u_j) \geq \omega(v_j) = 6$. The folded vertex v_j joins the solution only if its remaining neighbors are not part of a solution. If, in the mean time, its interface neighbors are already excluded at other PEs, folding v_j might be obsolete but is still correct. In this case, we merely decreased the weight of an interface vertex instead of excluding it immediately. Decreasing the weight does not cause a vertex to be included into the solution if an include is not possible without the weight shift. PE j locally excludes x_j once it receives a removal message for x_j from another PE. When excludes were communicated by then, we can unfold the vertex later and include v_j when we reconstruct the solution.

We can also apply the data reduction rules at the border and reduce ghost neighbors. At PE h we can include an interface vertex v_h using Neighborhood Removal by Lamm et al. [29]. For the ghost neighbors, updates may not bet received yet: they might be excluded or their weight was decreased. Moreover, v_h did not receive any new neighbors in the mean time. Therefore, the weights of ghost neighbors can be used as an upper bound for the weights of the respective interface vertices. Finally, PE h can communicate the include of v_h with the PEs owning the ghost neighbors so that their exclude is propagated accordingly.

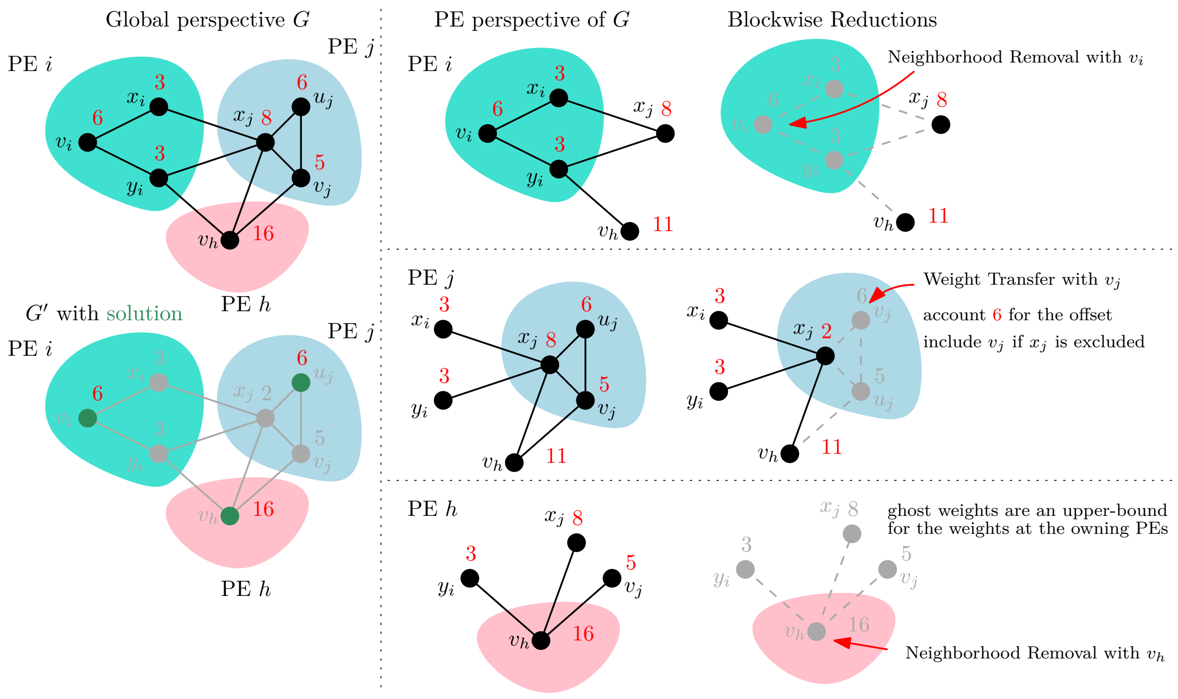
Global perspective G
PE i
PE j
PE h
G' with solution
PE perspective of G
Blockwise Reductions
Neighborhood Removal with v_i
Weight Transfer with v_j
account 6 for the offset
include v_j if x_j is excluded
ghost weights are an upper-bound for the weights at the owning PEs
Neighborhood Removal with v_h
x_i 3
v_i 6
y_i 3
x_j 8
u_j 6
v_j 5
v_h 16
11
x_j 2

Figure 4.2.: The figure shows an example were we apply reductions blockwise to reduce the graph G'. In the middle, the perspective of each PEs is shown. On the right, the reduced subgraph is highlighted in darkgray. The blockwise reduction applications yield the global reduced graph G' on the bottom-left side. We marked the solution vertices in green and their weight in red.

In general, including an interface vertex can lead to a solution conflict, because in the mean time a neighbor at another PE can be included as well. However, we show that these kind of conflicts do not require the reversal of modifications in the reduced graph and can be resolved by tie-breaking two conflicting vertices.

4.3. Distributed Reductions

We now transfer the data reductions rules to the distributed memory model so that they can be applied blockwise without any conflicts at the border. The input graph $G = (V, E, \omega)$ is stored in distributed memory among the PEs 1 to p as described in Section 2.2. The idea is as follows: Each PE can apply reductions locally in G_i. We state with a Border Argeement to what extent reductions are allowed to reduce the border. As a result it is ensured that the global reduced graph G' is always given via the reduced local subgraphs G_1 up to G_p if the reduction progress at the border is synchronized. Under the Border Argeement and given G_i, we can make certain assumptions to what extent the border is modified by other PEs. In some cases, these assumption allow us to reduce vertices close to the border. We give the details without loss of generality in the perspective of PE i.

Reduction Operations. A reduction application at PE i modifies G_i, e.g., removes vertices, folds multiple vertices into a new vertex or changes the weight of a vertex. It is also possible that the weight of a local vertex is modified. Moreover, a vertex $v \in V_i$ can be *moved* to another process. While this removes v from G_i, it does not change G. We denote the resulting reduced subgraph G_i' with vertices $\overline{V}_i' := V_i' \cup V_i^{g\prime}$ where V_i' are the modified owned vertices and $V_i^{g\prime}$ are the modified ghosts.

The following Border Argeement describes to what extent border vertices may be modified and reduced by distributed reductions. In general, arbitrary blockwise reduction applications at the border can result in a diverged reduced border because updates were not communicated yet. As a consequence, reductions might assume a wrong state of the border and propagate an error through the reduction process. To that end, the following reduction rules ensure that they can be safely applied blockwise, i.e., independent of the reduction applications at other PEs. They ensure that a modification at the border does not require a reversal of graph modifications at other PEs. It allows us to use weights of ghosts as upper bounds for the weight of the corresponding interface vertices at their PEs. Moreover, it ensures that G_i sees at the border at worst redundant information. Note that the Border Argeement initially holds for the input graph since we replicated the interface vertices with their weights as ghosts at the respective PEs.

Model Assumption 4.3.1 (Border Argeement)**.**
Let $v \in V_i$ be an interface vertex, i.e., $v \in N_i(V_i^g)$. PE i can only decrease the weight of v, modify its local neighbors, exclude v or propose to include v. A ghost $u \in V_i^g$ is at most excluded by PE i if a neighbor is included; otherwise it remains unmodified. If an interface vertex $v \in V_i$ is included, it holds $\omega_i(v) \geq \alpha(G_i[N_i(v)])$.

Theorem 4.3.1 (Distributed Reductions ensure the Border Agreement)**.**
All distributed data reduction rules reduce border vertices only as stated in the Border Argeement.

Proof. The proofs of data reduction rules verify that the Border Argeement is ensured when a reduction is applied. □

Roughly speaking, the (reduced) graph $G = (V, E, \omega)$ is given at any time via $G_1, \ldots, G_p$. A modification in G_i translates to a modification in G. For example, if a vertex $v \in \overline{V}_i$ is removed, it is also removed from V. If the weight $\omega_i(v)$ of a local vertex $v \in V_i$ is modified and v is still part of V, so it is modified at G, i.e., $\omega(v)$ is assigned $\omega_i(v)$. The reduced graph G can be obtained (e.g., with collective communication) from all the G_1 up to G_p with the weights of the local vertices. An outdated interface vertex $v \in V_i$ can easily be identified. The interface vertex v is outdated if it was removed as ghost at an adjacent PE, i.e., $v \notin V^g_{\mathrm{rank}(u)}$ for some ghost $u \in V^g_i \cap N_i(v)$.

Including Interface Vertices. Under the Border Argeement, conflicts might still occur when adjacent interface vertices are included blockwise as in Figure 4.3. However, we show in Lemma 4.3.1 that we can resolve such solution conflicts without loss of optimality after reducing the graph without undoing the reductions. Figure 4.3 illustrates the reason why this holds. The vertex x_j at PE j can only be included if it is merely adjacent to a single neighbor because x_i has equal weight. This is similar for x_i at PE i except for the fact that it is simplicial with more than one neighbor. A conflict arises if a simplicial vertex (here: x_i) is included while all neighbors were excluded by another PE except for one of equal weight (here: x_j). Such a conflict can only arise between two processes. If a third is involved no vertex sees a clique of at least three vertices spread acres three PEs but only non-adjacent ghost neighbors. If these ghost neighbors have equal weights, then an interface vertex v is not included since $\omega(v) < N(v)$.

Lemma 4.3.1 (Including Interface Vertices)**.**
Assume the Border Argeement is ensured through the reduction process. Let $v \in V_i$ be an interface vertex that is included at PE i. Further, let $u \in V_j$ be a ghost neighbor of v which is included at PE j, where $j = \mathrm{rank}(u)$. Then, $N_i[v]$ is excluded by an include of u and both vertices have equal weights, i.e., $\omega_i(v) = \omega_j(u)$. Further, u is the only ghost neighbor of v that is proposed for include.

Proof. We proof it via induction over the number of local included interface vertices l at PE i.

Induction beginning. Let $G^0_i = (\overline{V}^0_i, E^0_i, \omega^0_i)$ be the reduced subgraph at PE i before the first ($l = 1$) include of a vertex $x \in V^0_i$ which is excluded at PE j. Then, x is excluded due to the include of a neighbor at PE j which we denote y (Border Argeement). Further, let $G^*_j = (\overline{V}^*_j, E^*_j, \omega^*_j)$ denote the reduced subgraph at PE j before y is included. Since y is an interface vertex at PE j in G^*_j, y is included with a weight

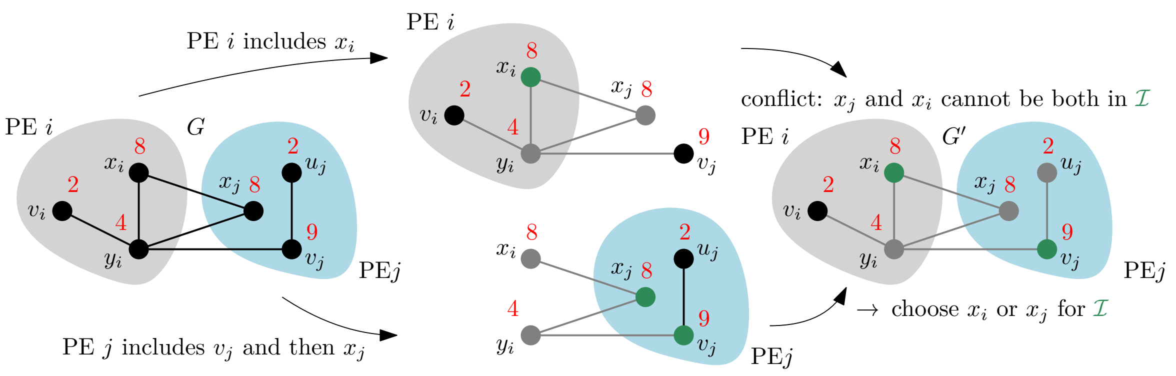
PE i includes x_i
PE i
G
PEj
PE j includes v_j and then x_j
conflict: x_j and x_i cannot be both in $\mathcal{I}$
G'
$\rightarrow$ choose x_i or x_j for $\mathcal{I}$
v_i
x_i
y_i
x_j
u_j
v_j
2
8
4
9

Figure 4.3.: The Figure shows two PEs trying to include (green) vertices at the border. All includes satisfy the Border Argeement, i.e., some interface vertex $v \in V$ is only included if $\omega_i(v) \geq \alpha(G_i[N_i(v)])$. Still, a conflict between x_i and x_j arises. Both have equal weights (red) and force the same vertices out of a solution (darkgray) except for themselves. According to Lemma 4.3.1 we can choose either x_i or x_j for $\mathcal{I}$ while maintaining optimality without undoing reductions.

which is at least as large as the weight of x at PE j, i.e., $\omega_j^*(y) \geq \alpha(G_j^*[N_j^*(y)]) \geq \omega_j^*(x)$ due to the Border Argeement On the other hand, y is not yet reduced at G_i^0 because an include of a local interface vertex (before including x) at PE i can exclude y. Thus, we know y is a neighbor of x in G_i^0, i.e., $y \in N_i^0(x)$. Further, it must hold $\omega_i^0(x) \geq \alpha(G_i^0[N_i^0(x)]) \geq w_i^0(y) \geq \omega_j^*(y)$ under the Border Argeement. Thus, both conflicting included vertices have equal weights $w_i(x) = w_i^0(x) = w_j^*(y) = w_j(y)$.

Moreover, we show that including x at PE i at most removes vertices which the include of y at PE j removes anyway, i.e., $U := N_i^0[x] \setminus N_j^*[y] = \emptyset$. Therefore, assume it exists $x' \in U \cap N_i^0(y)$. Then x' is already excluded at PE j anyway since $x' \notin N_j^*(y)$. Therefore, assume it exists a vertex $z \in U \setminus N_i^0[y]$, then $\{y, z\}$ is an independent set in the neighborhood of x in G_i^0. In this case we obtain $\omega(x) \geq \alpha(G_i^0[N_i^0(x)]) \geq \omega_i^0(\{z, y\}) > \omega_i^0(y) \geq \omega_j^0(y)$. This contradicts $\omega_i^0(x) = \omega_j^*(y)$. Therefore, U is the empty set. Analogously we obtain that $N_j^*(y) \setminus N_i^0[x] = \emptyset$. Thus, by including x at PE i and y at PE j we remove the same vertices from the graph. Both have the same weight, and one can decide for either of them while their neighbors remain excluded.

Note, y is the only such conflicting ghost neighbor of v. Assume there is another conflicting ghost neighbor $z \in N_i^0(v)$, $z \neq y$. It is sufficient to assume that it is not located at PE i or j; otherwise there is a conflict of two local vertices. Then $\{y, z\}$ is not an edge at G_i^0 since both are ghosts. Therefore, both vertices are an independent set in G_i^0. We obtain $\omega(x) \geq \alpha(G_i^0[N_i^0(x)]) \geq \omega_i^0(\{z, y\}) > \omega_i^0(y) \geq \omega_j^0(y)$ which contradicts that x and y are conflicting vertices which have equal weights.

Induction precondition. Now assume, this until the l-th include of an interface vertex at PE i in the case of conflicting includes between PE i and PE j.

Induction step. We show that this still holds for the $l + 1$-th include at PE i. Let x be the next included vertex and assume y is a (former) ghost neighbor that is included at PE j leading to the exclude of x at PE j. Let $G_i^l = (V_i^l, E_i^l, \omega_i^l)$ denote the induced subgraph at P_i before x is included at $G_j^* = (V_j^*, E_j^*, \omega_j^*)$ the subgraph at PE j before y is included. If y is already excluded at PE i, then it exists a former interface vertex $x' \neq x, x' \in V_i^l$ at PE i which was already included at some step $l' < l + 1$. Then, the induction hypothesis holds for the l'-th include where x' was included in conflict with the include of y at j. Since x is a neighbor of y and both, x' and y, exclude the same vertices, we know x is a neighbor of x' which was already excluded at PE i. This contradicts that x is not reduced yet at PE i.

Therefore, y must be a remaining ghost neighbor (and not an excluded ghost) which is included at PE j. Then it holds according to the Border Agreement that $w_i^l(x) \geq \alpha(G_i^l[N_i^l(x)) \geq w_i^l(y) \geq w_j^l(y)$. Similarly, it follows $\omega_j^l(y) \geq \omega_i^l(x)$. Thus, we obtain equality of their weights. Analogously to the induction beginning, we conclude that they remove the same remaining vertices and that there is at most one conflict with x. Ties only need to be broken between x and y while leaving the other vertices in their neighborhood excluded. □

We now give the details of our first distributed data reduction rule which allows us to include vertices of V_i which transfers Heavy Vertex by Lamm et al. [29]. The details of the kernelization algorithm and especially the reduction order are given in Section 4.4. The notation for data reduction rules is introduced in Section 2.3. Note that PE i proposes only to include an interface vertex by adding it to $\tilde{I}_i$. Distributed Heavy Vertex provides a strategy how to reconstruct the solution given $\tilde{I}_i$.

Distributed Reduction 4.3.1 (Distributed Heavy Vertex).
Let $v \in V_i$ with $\omega_i(v) \geq \alpha(G_i[N_i(v)])$, we can reduce $N_i[v]$ and propose to include v.

Reduced Graph	$G_i' = G_i - N_i[v]$
Include Proposal	$\tilde{\mathcal{I}}_i' = \tilde{\mathcal{I}}_i \cup \{v\}$

Reconstruct the solution as follows:

- *If* $\{u \in \tilde{\mathcal{I}}_{\text{rank}(u)} \cap N_i(v) : v < u\} = \emptyset$:

Offset	$\alpha(G) = \alpha(G') + \omega_i(v)$
Reconstruction	$\mathcal{I} = \mathcal{I}' \cup \{v\}$

- *Else:*

Offset	$\alpha(G) = \alpha(G')$
Reconstruction	$\mathcal{I} = \mathcal{I}'$

Proof. First, observe that all modifications satisfy the Border Argeement. Let $v \in V_i$ with $\omega_i(v) \geq \alpha(G_i[N_i(v)])$

There are two cases two consider. If v is not reduced yet by another process, we must show that there is an MWIS of G that contains v. If v is already reduced, i.e., v is not a vertex in G anymore, we must show that by removing $N_i[v]$ in G_i, we remove only already reduced vertices,i.e., we do not change G. Then, we need still to show that v can join the solution instead of its neighbors that are proposed for include.

First, suppose v is not reduced yet by another process. Every neighbor of v in G are neighbors in G_i and $\omega_i(v) \geq w(v)$, i.e., $N(v) \subseteq N_i(v)$. And every edge in $G_i[N_i(v) \cap V]$ is an edge in $G[N(v)]$. Thus, an MWIS of the induced neighborhood graph in G_i is an upper bound for an MWIS in G $\alpha(G_i[N_i(v)]) \geq \alpha(G[N(v)])$. Thus, we can apply Heavy Vertex by Lamm et al. [29] in G to include v, remove $N(v)$. By removing $N_i[v]$ in G_i, we remove already reduced vertices or vertices that can be reduced.

Assume now v is reduced by another process. Then v must be an interface vertex. With Lemma 4.3.1, we know there is exactly one neighbor that is a ghost which is proposed for include $uN_i(v) \cap V_i^g$. Furthermore, both have equal weights and u forces the vertices of $N_i[v] \setminus \{u\}$ out of the solution while v forces at most $N_i(v)$ these vertices out of the solution. Thus, we can include v instead of u into $\mathcal{I}$ while excluding no more vertices than u does.

Therefore, we can add v to P_i in both cases. The tie breaking ensures that only of multiple candidates is chosen into the solution. The offset and the solution can be reconstructed When P was communicated, ties can be broken between pairwise conflicting vertices. If v is included, it joins the solution, and $\omega(v)$ is added to the offset. □

Analogously to the sequential data reduction rules, Distributed Heavy Vertex gives rise to multiple special cases that are computationally more efficient.

Distributed Reduction 4.3.2 (Distributed Neighborhood Removal)**.**
Let $v \in V_i$ with $\omega_i(v) \geq \omega_i(N_i(v))$, then include v as in Distributed Heavy Vertex.

Proof. To proof correctness, we show that this reduction is a special case of Distributed Heavy Vertex. Let $v \in V_i$ with $\omega_i(v) \geq \omega_i(N_i(v))$. Then the weight of the neighborhood is an upper bound of an MWIS in the induced neighborhood graph, i.e., $\omega_i(N_i(v)) \geq \alpha(G_i[N_i(v)])$. Thus, we can apply the Distributed Heavy Vertex. This yields the described reduced graph, offset and proposed solution reconstruction. □

Next, we transfer the Simplicial Vertex Reduction 4.3.3 to reduce cliques. Note that a clique in G_i contains at most one ghost because ghosts cannot be adjacent. Moreover, we cannot conclude for a ghost that it is a simplicial vertex in G because we do not know its complete neighborhood.

Distributed Reduction 4.3.3 (Distributed Simplicial Vertex)**.**
Let $v \in V_i$ be a simplicial vertex with maximum weight in its neighborhood in G_i, i.e., it holds $\omega_i(v) \geq \max\{\omega_i(u) : u \in N_i(v)\}$, then include v as in Distributed Heavy Vertex.

Proof. To proof correctness, we show that this reduction is a special case of Distributed Heavy Vertex. Let $v \in V_i$ be a simplicial vertex with $\omega_i(v) \geq \max\{\omega_i(u) : u \in N_i(v)\}$. Then, $N_i(v)$ forms a clique in G_i. Thus, an MWIS of the induced neighborhood graph of $G_i[N_i(v)]$ can consist of a most one vertex. Therefore it holds, i.e., $\omega_i(v) \geq \max\{\omega_i(u) : u \in N_i(v)\} \geq \alpha(G_i[N_i(v)])$. Thus, we can apply Distributed Heavy Vertex. This yields the described reduced graph, offset and proposed solution reconstruction. □

Distributed Reduction 4.3.4 (Distributed Simplicial Weight Transfer).
Let $v \in V_i$ be a simplicial vertex, let $S(v) \subseteq N_i(v)$ be the set of all simplicial vertices. Further, let $w_i(v) \geq w_i(u)$ for all $u \in S(v)$.

- *If $\omega_i(v) = \max\{w_i(u) : u \in N_i(v)\}$, then use Distributed Simplicial Vertex.*
- *Else if v is not an interface vertex, i.e., $N_i(v) \cap V_i^g = \emptyset$, fold v into $N_i(v)$.*

Reduced Graph	*$G'_i = G_i - X$ with $X := \{u \in N_i[v] : \omega_i(u) \leq \omega_i(v)\}$ and set $w'_i(u) = w_i(u) - w_i(v)$ for all $x \in N_i(v) \setminus X$*
Offset	*$\alpha(G) = \alpha(G') + \omega(v)$*
Reconstruction	*If $\mathcal{I}' \cap N_i(v) = \emptyset$, then $\mathcal{I} = \mathcal{I}' \cup \{v\}$, else $\mathcal{I} = \mathcal{I}'$*

Proof. Let $v \in V_i$ be a simplicial vertex, let $S(v) \subseteq N_i(v)$. In the first case, i.e., $\omega_i(v) = \max\{w_i(u) : u \in N_i(v)\}$, we can apply Distributed Simplicial Vertex. For the second case, we assume v is not an interface vertex. We first observe that the proposed modification ensures the Border Argeement. Interface vertices are assigned at most a decreased weight or are excluded. Ghost vertices remain part of the graph and their weight is not modified. We show now that we can apply Simplicial Weight Transfer by Lamm et al. [29] under all intermediate modifications at the border. In the neighborhood of v are only local neighbors which are possibly interface vertices. These interface vertices may be excluded by another process. Other neighbors are not reduced yet. Let $U \subseteq N_i(v)$ be set of excluded interface neighbors of v. Independent of whether vertices of U are excluded, v remains simplicial in G_i. In any case, the second case of Simplicial Weight Transfer by Lamm et al. [29] applies in G_i. From a global perspective, we decreased the weight of vertices of U at worst instead excluding them in G_i as well when we apply the fold. It is possible that v may have the largest weight in its neighborhood G since the vertices of U were excluded. Then v is included when it is unfolded in the reconstruction. □

In the following, we transfer data reduction rules to the distributed memory model which exclude vertices. They also exclude interface vertices because in these cases they are not part of some MWIS independent of their ghost neighbors.

Distributed Reduction 4.3.5 (Distributed Basic Single-Edge).
Let $u, v \in V_i$ be adjacent vertices with $\omega_i(N_i(u) \setminus N_i(v)) \leq \omega_i(u)$, then exclude v.

Reduced Graph	$G'_i = G_i - v$
Offset	$\alpha(G) = \alpha(G')$
Reconstruction	$\mathcal{I} = \mathcal{I}'$

Proof. Let $u, v \in V_i$ be adjacent vertices with $\omega_i(N_i(u) \setminus N_i(v)) \leq \omega_i(u)$. First, we observe that the application of this reduction satisfies the Border Argeement because we exclude only local vertices. To show that v can be excluded, we show that we can apply Basic Single-Edge by Gu et al. [23] under every possible border modifications in G. If v is already reduced by another process, i.e., $v \notin V$, then v is redundant at G_i and can be removed from G_i anyway. Therefore, assume that v is not reduced yet. Note that vertices of $N_i(u) \setminus N_i(v)$ can be assigned smaller weights or may be excluded by other processes. If u is not excluded yet, then we can still apply Basic Single-Edge by Gu et al. [23] in G.

We show now that u is either not excluded by another process or v is excluded too. For a proof by contradiction, assume u is excluded while v is not excluded. Then u was excluded due to the include of an interface vertex x at an adjacent process. Furthermore, x cannot be adjacent to v; otherwise v is also excluded. We also know that $x \in N_i(u)$; otherwise x had another neighbor y at PE i which was included and further, excluded x. According to Lemma 4.3.1, y and x forced the same vertices out of the solution, i.e., y is adjacent to u and u is reduced at PE i. However, this contradicts $u \in V_i$. Consequently, it holds $x \in N_i(u)$, and therefore $\omega_i(u) \geq \omega_i(N_i(u) \setminus N_i(v)) \geq \omega_i(\{x, u\})$ at PE i because x and u are not adjacent. Under the Border Argeement, it holds $\omega_i(x) \geq \omega_{\text{rank}(x)}(x) = \omega(x)$, and in conclusion, $\omega(u) = \omega_i(u) \geq \omega_i(\{x, u\}) > \omega_{\text{rank}(x)}(x) = \omega(x)$. However, this contradicts the Border Argeement because x can only be included as an interface vertex if $\omega(x) \geq \alpha(G_{\text{rank}(x)}[N_{\text{rank}(x)}(x))]$. Thus, u is not excluded by another process if v is not excluded yet.

In conclusion, we can either apply Basic Single-Edge by Gu et al. [23] to v in G or v is already reduced by another process. In the latter case we remove only a redundant vertex from G_i. □

A special case of the Distributed Basic Single-Edge is Distributed Domination.

Distributed Reduction 4.3.6 (Distributed Domination).
Let $u, v \in V_i$ be adjacent vertices with $N_i(u) \subseteq N_i(v)$. If $\omega_i(v) \leq \omega_i(u)$, then exclude v.

Reduced Graph	$G'_i = G_i - v$
Offset	$\alpha(G) = \alpha(G')$
Reconstruction	$\mathcal{I} = \mathcal{I}'$

Proof. Let $u, v \in V_i$ be adjacent vertices with $N_i(u) \subseteq N_i(v)$ and $\omega_i(v) \leq \omega_i(u)$. Since $N_i(u) \subseteq N_i(v)$, it holds $N_(u) \setminus N_i(v) = \{v\}$. We obtain $\omega_i(N_i(u) \setminus N_i(v)) = \omega_i(v) \leq w_i(u)$. Therefore, we can apply Distributed Basic Single-Edge. This yields the reduced graph, offset, and solution reconstruction while ensuring the Border Argeement. □

We also transfer Extended Single-Edge by Gu et al. [23] to the distributed memory model.

Distributed Reduction 4.3.7 (Distributed Extended Single-Edge)**.**
Let $u, v \in V_i$ be adjacent vertices with $\omega_i(N_i(v)) - \omega_i(u) \leq \omega_i(v)$, then exclude $X := N_i(v) \cap N_i(u)) \setminus V_i^g$.

Reduced Graph	$G_i' = G_i - X$
Offset	$\alpha(G) = \alpha(G')$
Reconstruction	$\mathcal{I} = \mathcal{I}'$

Proof. Let $u, v \in V$ be adjacent vertices with $\omega_i(N_i(v)) - \omega_i(u) \leq \omega_i(v)$. Furthermore, let $X := N_i(v) \cap N_i(u)) \setminus V_i^g$ be non-empty. First of all, we observe that applying Distributed Extended Single-Edge satisfies Border Argeement because it only excludes local vertices. To proof correctness, we show for every possible modification under the Border Argeement that we can apply Extended Single-Edge by Gu et al. [23] in G for v which removes X.

First, consider the case where v and u are not reduced yet by other processes. Independent of whether neighbors of v are reduced or were assigned smaller weights, the inequality still holds in G, i.e., $\omega(v) = \omega_i(v) \geq N_i(v) - \omega_i(u) \geq N(u) - \omega(u)$ and we can remove the remaining vertices of X due to Extended Single-Edge by Gu et al. [23]. Now, consider the case where u is reduced but v is not. Then, we obtain in G that $\omega(v) \geq N(v)$. Thus, the vertices of X can be excluded, because we can replace in every independent set of G, the vertices of X by v to obtain a solution which is at least the same weight.

We show that those are all the cases need to be considered by showing that v is not reduced yet and removing X in G_i removes vertices in G,i.e., $X \cap V \neq \emptyset$. For a proof by contradiction, assume v is an interface vertex which is reduced by another process due to including one of its the neighbors, i.e., some $x \in N_i(v)$. Observe that $u \notin X \cup \{x\}$ and that $x \notin X$. Due to the Border Argeement, it holds $\omega_i(x) \geq \omega(x)$. Overall, we obtain $\omega_i(v) \geq \omega_i(N_i(v) \setminus \{u\}) \geq \omega_i(X) + \omega_i(x) > \omega(x)$. This contradicts that x is an included interface vertex since it can only be included if $\omega(x) \geq \alpha(G_{\text{rank}(x)}[N_{\text{rank}(x)}(x)]) \geq \omega(v)$. In conclusion, we can apply always Extended Single-Edge by Gu et al. [23] in G for v and reduce the remaining vertices of X since we know that x is not reduced yet. This yields the reduced graph, offset, and solution reconstruction. □

Many graphs such as scale-free graphs have a large proportion of degree one vertices. Often, even more degree one vertices arise when reductions are applied. Therefore, we want to reduce all degree one vertices similar to Degree One by Gu et al. [23], especially if these are interface vertices. However, folding an interface vertex into its ghost neighbor, i.e., decreasing the weight of a ghost, contradicts the Border Argeement. Therefore, we move v to the adjacent PE j which can fold v into its neighbor if

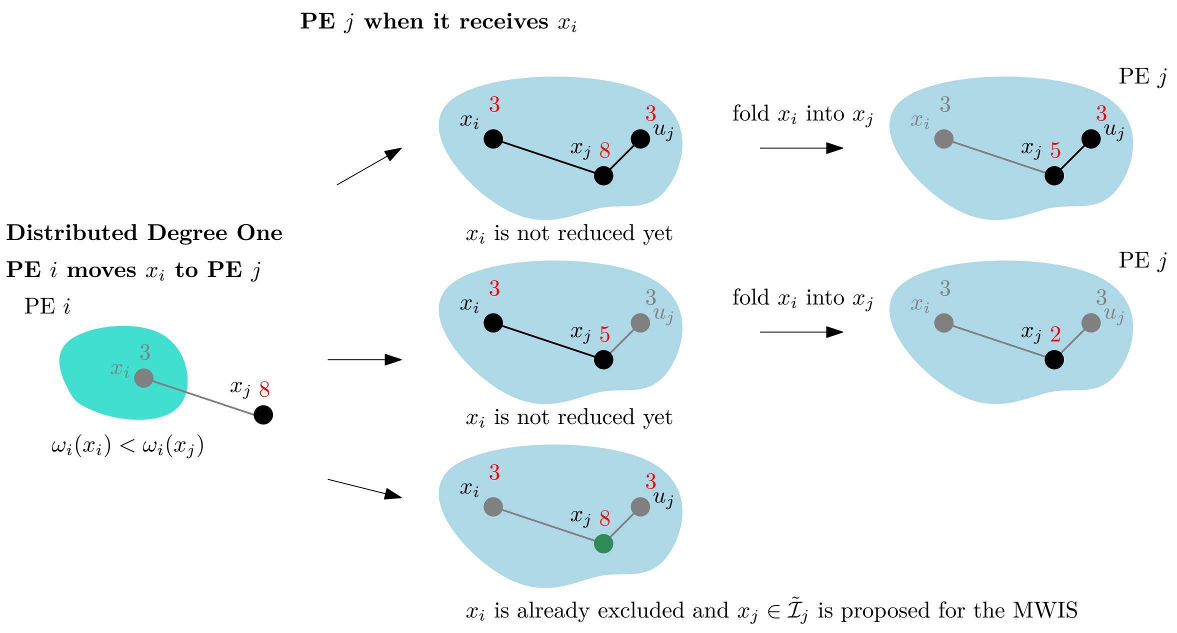
PE j when it receives x_i
Distributed Degree One
PE i moves x_i to PE j
PE i
$\omega_i(x_i) < \omega_i(x_j)$
x_i is not reduced yet
x_i is not reduced yet
x_i is already excluded and $x_j \in \tilde{\mathcal{I}}_j$ is proposed for the MWIS
fold x_i into x_j
fold x_i into x_j
PE j
PE j

Figure 4.4.: The Figure shows an example for the *move* case of Degree One. PE i moves x_i to PE j because x_i where it can be reduced or is already reduced but PE i is not aware of it yet. At PE j is can be folded (top and middle). In the middle, x_i can still be folded although the weight of x_j decreased. In the last case x_j is already proposed for include (green) with Distributed Neighborhood Removal.

v or its neighbor is not reduced yet. If its neighbor is already reduced, PE j can include v for PE i. Figure 4.4 gives an example.

Distributed Reduction 4.3.8 (Degree One)**.**
Let $v \in V_i, u \in \overline{V}_i$ with $N_i(v) = \{u\}$.

- *If $\omega_i(v) \geq \omega_i(u)$, then include v with Distributed Neighborhood Removal*
- *Else if $\omega_i(v) < \omega_i(u)$ and $u \in V_i^g$, move v from PE i to PE* rank(u)

  Reduced Graph $G_i' = G_i - v$

  In the case u is moved to PE i with weight w_u, reconstruct the solution as follows. If $w_i(v) \geq w_u$ or $w_i(v) = w_u$ and $v < u$, then include v.

Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Reconstruction	$\mathcal{I} = \mathcal{I}' \cup \{v\}$

- *Else if $\omega_i(v) < \omega_i(u)$ and $u \in V_i$, and fold u into v and decrease the weight of v by $\omega_i(u)$.*

Reduced Graph	$G_i' = G_i - u$ *with* $\omega_i'(v) = \omega_i(v) - \omega_i(u)$
Offset	$\alpha(G) = \alpha(G') + \omega(v)$
Proposed Reconstruction	$\mathcal{I} = \mathcal{I}' \cup \{v\}$
Accept if	$\{u \in \mathcal{I} \cap N_i(v) : v < u\} = \emptyset$

Proof. First of all, we observe that all proposed modifications at the border satisfy the Border Argeement. Now let $v \in V_i, u \in \overline{V}_i$ with $N_i(v) = \{u\}$. Note that the first case is a special case of Distributed Neighborhood Removal. Further, observe that the third case is a special case of Distributed Simplicial Weight Transfer where v is the considered simplicial vertex and u has larger weight. Therefore, we only need to consider the second case of detaching v, i.e., $\omega_i(v) < \omega_(u)$ where u is a ghost. We show that v is either successfully detached to PE rank v, already reduced and we only remove a redundant vertex, or if u is detached as well, we reconstruct the solution correctly. First, it is possible that v is already reduced. Then v is excluded due to including u at PE rank(u). By detaching v to PE rank(u), we remove only a redundant vertex at G_i and do not change G. Secondly, if v is not yet reduced and u is not detached, detaching v leaves G unmodified. In this case PE rank(u) owns v and if u is not excluded yet, v still has u as neighbor. Last, it is possible that v is not reduced yet and u is detached as well. At PE rank u, u had as last neighbor v before it was detached to PE i. Both ends consider now an isolated cut-edge in G were both incident vertices were detached. The vertex with the larger weight must join the solution. In the case of a tie, the vertex wins which has the smaller global vertex identifier. □

We now transfer more data reduction which fold vertices. They modify the neighborhood only of local vertices so that other processes are not affected by the changes.

Distributed Reduction 4.3.9 (Distributed Generalized Neighborhood Folding)**.** *Let $v \in V_i \setminus N_i^2[V_i^g]$. Further, suppose that $G_i[N_i(v)]$ contains only one independent set $\tilde{\mathcal{I}}$ with $\omega_i(\tilde{\mathcal{I}}) > \omega_i(v)$, fold v and $N_i(v)$ into a new vertex v'.*

Reduced Graph	$G_i' = G_i[(V_i \cup \{v'\}) \setminus N_i[v]]$ *with* $N_i(v') = N_i(N_i(v))$ *and* $\omega_i(v') = \omega_i(\tilde{\mathcal{I}}) - \omega_i(v)$
Offset	$\alpha(G) = \alpha(G') + \omega_i(v)$
Reconstruction	*If* $v \in \mathcal{I}'$, *then* $\mathcal{I} = (\mathcal{I}' \setminus \{v'\}) \cup N_i(v)$, *else* $\mathcal{I} = \mathcal{I}' \cup \{v\}$

Proof. First, we observe that the proposed modification satisfies the Border Argeement. We show now that we can apply in G Neighborhood Folding by Lamm et al. [29] under every possible border modification. Therefore, let $v \in V_i \setminus N_i^2[V_i^g]$. Further, suppose that $G_i[N_i(v)]$ contains only one independent set $\tilde{\mathcal{I}}$ with $\omega_i(\tilde{\mathcal{I}}) > \omega_i(v)$, fold v and $N_i(v)$ into a new vertex v'. We note that $N_i(v)$ contains no border vertex since v is chosen with minimum distance 3 apart from any ghost. Thus, no vertex in $N_i(v)$ is reduced or received a smaller weight. Therefore, $G_i[N_i(v)] = G[N(v)]$, and further, $\tilde{\mathcal{I}}$ is a unique independent set with $\omega(\mathcal{I}) > \omega(v)$ in G. Thus, we can apply Distributed Neighborhood Folding in G for v. This yields also the reduced graph, offset and reconstruction. Regarding G_i, it is possible that the reduction causes the new vertex v' to be adjacent to a vertex $x \in N_i(N_i[v])$ that is already excluded by another process. However, the modification in G_i is still correct, because when v' is unfolded in the reconstruction, x is known to be excluded. □

Distributed Reduction 4.3.10 (Distributed Neighborhood Folding)**.**
Let $v \in V_i \setminus N_i^2[V_i^g]$, and suppose that $N_i(v)$ is independent. If $\omega_i(N_i(v)) > \omega_i(v)$, but $\omega_i(N_i(v)) - \min\{\omega_i(u) : u \in N_i(v)\} < \omega_i(v)$, then fold v and $N_i(v)$ into a new vertex v'.

Reduced Graph	$G_i' = G_i[(V \cup \{v'\}) \setminus N_i[v]]$ *with* $N_i(v') = N_i(N_i(v))$ *and* $\omega_i(v') = \omega_i(N_i(v)) - \omega_i(v)$
Offset	$\alpha(G) = \alpha(G') + \omega_i(v)$
Reconstruction	*If* $v \in \mathcal{I}'$, *then* $\mathcal{I} = (\mathcal{I}' \setminus \{v'\}) \cup N(v)$, *else* $\mathcal{I} = \mathcal{I}' \cup \{v\}$

Proof. This data reduction is only a special case of Distributed Generalized Neighborhood Folding. This yields the reduced graph, offset, and reconstruction. □

Distributed Reduction 4.3.11 (Distributed Partial V-Shape)**.**
Let $v \in V_i \setminus N_i^2[V_i^g]$, $x, y \in V_i \setminus N_i(V_i^g)$ with $N(v) = \{x, y\}$ so that x and y are not adjacent. Without loss of generality, assume $\omega_i(x) \leq \omega_i(y)$ and $\omega_i(v) \geq \omega_i(x)$.

- *If $\omega_i(v) < \omega_i(y)$, then fold v into x and y.*

Reduced Graph	$G_i' = G_i - v$ *with* $N_i(x) = N_i(x) \cup N_i(y)$ *and set* $\omega_i(y) = \omega_i(y) - \omega_i(v)$
Offset	$\alpha(G) = \alpha(G') + \omega_i(v)$
Reconstruction	*If* $x, y \in \mathcal{I}'$, *then* $\mathcal{I} = \mathcal{I}' \cup \{v\}$, *else* $\mathcal{I} = \mathcal{I}'$

- *If $\omega_i(y) \leq \omega_i(v) < \omega_i(x) + \omega_i(y)$, fold v, x, y into a new vertex v'.*

Reduced Graph	$G_i' = G_i[(V_i \cup \{v'\}) \setminus \{v, x, y\}$ *with* $N_i(v') = (N_i(x) \cup N_i(y)) \setminus \{v\}$ *and* $\omega_i(v') = \omega_i(x) + \omega_i(y) - \omega_i(v)$
Offset	$\alpha(G) = \alpha(G') + \omega_i(v)$
Reconstruction	*If* $v' \in \mathcal{I}'$, *then* $\mathcal{I} = \mathcal{I}' \cup \{v\} \setminus \{v'\}$, *else* $\mathcal{I} = \mathcal{I}' \cup \{v\} \cup \setminus \{v\}$

- *If $\omega_i(x) + \omega_i(y) \leq \omega_i(v)$, include v with Distributed Neighborhood Removal.*

Proof. First, we observe that the proposed modifications satisfy the Border Argeement. The third case is just a special case of Distributed Neighborhood Removal. For the other case we can make a similar argument as in the proof of Distributed Generalized Neighborhood Folding. In each case we can apply the respective case of V-Shape by Gu et al. [23] in G because $G_i[N_i(v)] = G[N(v)]$ for $v \in N_i^2(V_i^g)$. Regarding G_i, the at most add neighbors of to already excluded interface vertices. However, this does not affect the reconstruction of the solution. □

4.4. Local Kernelization

We now give the details of our local kernelization algorithm.

High-Level Idea. The idea is very similar to many sequential kernelization algorithms. The reduction rules are applied iteratively in an exhaustive fashion. More precisely, we process the reduction rules in a fixed order and each rule is tested for all vertices. Once a rule has been tested (and applied) for all vertices, the iteration restarts with the first reduction rule if there has been any progress. Otherwise, the next reduction rule is considered if there is one left. If no reduction rule is left, we are finished. Then, no vertices can be reduced anymore by any reduction, i.e., we applied all reduction rules exhaustively. Note, that new reductions might become applicable if reduction progress at the border is communicated. Therefore, if the border is not fully synchronized yet but the PEs applied reductions exhaustively, we call the local reduced subgraph a *local quasi-kernel*.

Reduction Order. Our reduction order follows the intuition to start with reduction rules that can reduce interface vertices as well as other local vertices. If interface vertices are reduced, there is a chance that the local reduced subgraph becomes less connected to the remaining graph stored at other PEs. Ideally, a good reduction impact of the following reductions entails more reduction applications with reduction rules that cannot handle interface vertices well. Therefore, we put Degree One first and Distributed Neighborhood Removal second. Likewise, both rules can also be efficiently tested in $\mathcal{O}(1)$ and $\mathcal{O}(\Delta)$ given a vertex $v \in V_i$. The first two data reduction rules are followed by Distributed Simplicial Weight Transfer. The implementation also covers the Distributed Simplicial Vertex. Afterward, we test for the Distributed Partial V-Shape to target vertices of degree two, and the excluding data reduction rules: Distributed Domination, Distributed Basic Single-Edge, Distributed Extended Single-Edge. Then, we try Distributed Neighborhood Folding, followed by two rules of higher computational cost, Distributed Generalized Neighborhood Folding and Distributed Heavy Vertex. Both need to find a MWIS for at least one subproblem. The size of the subproblem size can be controlled with an extra parameter to circumvent too long running times for single subproblems. For our experiments we choose 100 as the maximum number of vertices in the subproblem. The MWIS is determined with the weighted branch-and-reduce solver KaMIS_wB&R [29].

Dependency Checking is a common technique to prune reduction tests. The idea is to skip the tests for vertices which have not change locally. By locally, we refer to the subgraph which is *read* in order to evaluate for a vertex whether a reduction rule applies to it. For example, Distributed Heavy Vertex reads all weights in its direct neighborhood of a vertex $v \in V_i$. If the weight of a neighbor changes, Distributed Heavy Vertex should be tested again for v because v may have larger weight than $\omega(N_i[v])$. On the other hand if $G_i[N_i(v)]$ does not change, and we tested the data reduction rule once, it will still not succeed. In these cases, a reduction rule would read the same information again to evaluate whether a vertex can be reduced.

We implement this idea using vertex markers where each vertex of V_i can be marked for each reduction rule. Initially we mark all vertices of V_i for each reduction rule. Once a reduction was tested for a vertex $v \in V_i$, the respective mark for v is removed. Whenever a vertex is modified but not reduced yet, e.g., due to a weight shift or a changed neighborhood up to distance d, we mark the vertex for all reductions.

For the sequential data reduction rules in Section 4.1, it is sufficient to choose $d = 1$ because only the direct neighborhood is read. However, for our distributed data reduction rules dependency checking is a bit more expensive. Some of our distributed data reduction rules, e.g., Distributed Generalized Neighborhood Folding, rely on the fact that they have no interface vertices in the direct neighborhood. In general, this requires marking neighbors up to distance $d = 2$ if a vertex changed. We can prune the computational cost for local vertices to $\mathcal{O}(\Delta)$ since we only need to do so if a ghost is excluded. Only then, interface vertices might have lost all ghost neighbors. When a ghost is excluded, we search its neighborhood for vertices that are no interface vertices anymore. These can be found in $\mathcal{O}(\Delta)$ since we maintain for an interface vertex the number of ghosts. Only for these former interface vertices we mark the neighborhood for further reduction tests.

4.5. Dynamic Graph Data Structure

Our internal graph representation stays close to the input format of the graph as described in Section 2.2. Each of the p PEs only stores a subgraph of the input graph which is partitioned into p blocks $(V_1, \ldots, V_p)$. PE i is assigned the local vertices V_i. The neighborhoods are represented using adjacency lists where each neighborhood is stored in a separate array. Moreover, we also store the (partial) neighborhoods of ghost vertices, i.e., a ghost adjacency array stores the neighbors of the local block V_i. Note that each undirected cut-edge is represented by four directed edges in the distributed memory. We use a hash-map to map the global vertex identifiers of the ghosts to local vertex identifiers. Since we consider weighted graphs, we maintain the weight of each local vertex and ghost.

We follow the paradigm as in other kernelization algorithms by using a dynamic graph data structure [29]. A dynamic graph data structure hides a vertex v by hiding v in the adjacency arrays of its neighbors. Hiding an edge requires to scan an adjacency array to find the edge, swapping it to the end of the visible part, and to decrease the visible part of the adjacency array by one. As a result, this allows one to efficiently iterate over the remaining visible part of a neighborhood. Moreover, the hide operations can be undone in reversed order where restoring a vertex can be done in $\mathcal{O}(\Delta)$.

Some of our distributed data reduction can only act entirely locally, i.e., there must be no interface vertex in the closed neighborhood of a considered vertex $v \in V_i$. To circumvent searching $N_2[v]$ for ghosts every time, we maintain for a local vertex $u \in V_u$ the number of ghost neighbors in G_i. If this count reaches zero for u, the vertex u is not an interface vertex anymore. We update the number of ghosts neighbors in $\mathcal{O}(\Delta(G_i))$, whenever a ghost is hidden.

4.6. Communication

So far we transferred the data reduction rules to the distributed memory model and discussed how each process can apply them. Applying reductions without any communication between the PEs, in an exhaustive fashion results in a local quasi-kernel. There are two main reasons for communicating changes at the border. First of all, PEs may propose to include vertices that were adjacent. They need to communicate the included interface vertices to tie-break which proposed solution is accepted. Secondly, if we reduce vertices at the border, it may re-open the search space for reduction applications. To to that end, we introduce different message types in Section 4.6.1 that exchange the reduction progress. Afterward, we propose and present two distributed kernelization algorithms by utilizing two different communication approaches. The synchronous approach is introduced in Section 4.6.2 and in the asynchronous approaches is presented in Section 4.6.3.

4.6.1. Message Types

We now give the details of the different message types and outline how our distributed data reductions from Section 4.3 make use of them. The idea is to inform PEs about the reduction progress at the border that own or replicated the corresponding modified border vertices. Both communication approaches implement the message types and their communication differently but the idea is the same.

Weight Shift Message. When the weight of an interface vertex is decreased, e.g., by Distributed Simplicial Weight Transfer, the PEs of the adjacent ghost vertices are not aware of the weight shift yet. However, notifying the according PEs may induce further reduction applications in return. A *weight shift message* for a vertex $v \in V_i$ is a tuple $(v, w_i(v)) \in V_i \times \mathbb{N}_{>0}$ where v is represented by its global vertex identifier.

The next two message types fulfill two roles. First, they inform adjacent PEs of the respective reduced border vertices. As a result, the receiving process is aware that they are no long part of the global reduced graph G and can remove the redundant vertices locally. Secondly, they notify about the assigned solution status. In data reductions such as Distributed Heavy Vertex, interface vertices can be proposed to be included. This may lead to a situation where a tie-breaking is needed. By sending the proposed solution status, each process can apply the defined tie breaking in Distributed Heavy Vertex locally if necessary.

Exclude Vertex Message. When an interface vertex $v \in V_i$ is excluded at PE i, it notifies the adjacent PEs $\mathcal{R}(v)$ so that they can remove the replicated ghost vertex.

The *exclude vertex message* consists of the global vertex identifier of v. Furthermore, the message informs them that PE i excludes v from the solution.

Include Vertex Message. Now, suppose PE i proposes to include an interface vertex $v \in V_i$. Similarly to an *exclude vertex message*, an *include vertex message* consists only of the global vertex identifier of v which is sent to the PEs $\mathcal{R}(v)$. Note that if a v is proposed for include, $N_i[v]$ is reduced. Thus, ghost neighbors are removed as well. Removing $N_i[v]$ can take up to two communication steps until all affected PEs are informed. Once the adjacent PEs receive the include message, they reduce the ghost v and its neighbors if they were not reduced yet. Reducing the interface vertices results in new exclude messages because these interface vertices may have other ghost neighbors.

Note that this approach differs from George et al. [19] as they write messages for all the removed ghosts and send them to the respective adjacent PEs. If there are multiple ghosts removed which are owned by the same PE, this can increase the communication volume,i.e., if $\deg_i(v) > |\mathcal{R}_i(v)|$. Their approach might be sufficient as they mainly target to reduce vertices of small degree. However, in our case interface vertices of large degree can be reduced as well, e.g., Distributed Heavy Vertex reduces neighborhoods of arbitrary large degree.

Vertex Move Message. Lastly, we introduce the *vertex move message* that is used for reducing degree-one interface vertices at adjacent PEs. It contains the the global vertex identifier of v and is sent to the adjacent PE $j \neq i$. Intuitively speaking, we transfer the ownership of v to some PE $j \neq i$ For example, Degree One removes v if $\omega_i(v) < \omega_i(u)$ and writes a vertex move message to PE j. Then, it can be folded at PE j into u or included. PE i expects to receive the solution status for v from PE j at latest if the solution is reconstructed. The reason for this is that prior reductions, e.g., folded vertices, may depend on its solution status.

4.6.2. Synchronous Approach

The first distributed kernelization approach follows a synchronous communication scheme. The reduction progress at the border is exchanged using blocking, irregular all-to-all collective communication (MPI_Alltoallv). We give the pseudo code of our synchronous distributed kernelization algorithm, called KaDisReduS, in Algorithm 1 and Algorithm 2. Roughly speaking, KaDisReduS exhaustively tries to reduce the local subgraph G_i in LocalReduce until a local quasi-kernel is obtained. Then it blocks to communicate and update the border and solution in UpdateBorderAndSolution. Both steps are repeated until no PE has made any new reduction progress. We use a (MPI_Allreduce) to determine whether is some new global reduction progress. When no global reduction progress is possible anymore, each process stores the reduced graph G_i locally and the local reduction offset o_i.

Status Messages. Our synchronous approach unifies three of the four message types: excluded, included, and move vertex messages by adding the respective vertex status to the message. Therefore, a message send by PE i has the shape $\overline{V}_i \times \{\textsf{included}, \textsf{excluded}, \textsf{move}\}$.

Algorithm 1 KaDisReduS

```
Input: subgraph $G_i$
Output: reduced subgraph $G_i$, offset $o_i$
procedure KaDisReduS($G_i$)
    $\tilde{\mathcal{I}}_i \leftarrow \emptyset$                                  ▷ proposed solution vertices $\tilde{\mathcal{I}}_i \subseteq V_i$
    $o_i \leftarrow 0$                                  ▷ reduction offset $o_i \in \mathbb{N}_0$
    while not exhaustively reduced do                   ▷ MPI_Allreduce
        $W_i \leftarrow \emptyset$                      ▷ weight shifted interfaces vertices $W_i \subseteq V_i$
        $M_i \leftarrow \emptyset$        ▷ status updates $M_i \subseteq V_i \times \{\text{included}, \text{excluded}, \text{moved}\}$
        LocalReduce($G_i$, $\tilde{\mathcal{I}}_i$, $o_i$, $W_i$, $M_i$)          ▷ as in Section 4.4

        $\tilde{W}_i \leftarrow$ CreateWeightUpdateMessages($W_i$)   ▷ to those with ghost neighbors
        ExchangeAndUpdateWeights($\tilde{W}_i$)                ▷ MPI_Alltoallv
        $W_i \leftarrow \emptyset$
        $\tilde{M}_i \leftarrow$ CreateStatusMessages($G_i$, $M_i$)     ▷ filter out redundant moves
        $U_i \leftarrow$ ExchangeStatusUpdates($\tilde{M}_i$)          ▷ MPI_Alltoallv
        $M_i \leftarrow \emptyset$
        UpdateBorderAndSolution($G_i$, $\tilde{\mathcal{I}}_i$, $o_i$, $U_i$, $M_i$)    ▷ described in Algorithm 2
    end while
end procedure
```

Figure 4.5.: This pseudo code give the high level idea of the synchronous distributed kernelization algorithm KaDisReduS in the perspective of PE i.

We denote this message in the following a *Status Message*. When reductions in LocalReduce propose to include, exclude or move an interface vertex v, we first write v into a buffer. The actual messages are written and sent after the local quasi-kernel is obtained. This strategy allows us to filter out vertex moves in CreateStatusMessages if the ghost neighbor was assigned a solution status later on in LocalReduce. Such a move has become redundant because its neighbors are reduced and thus it is a degree zero vertex that can be proposed for include.

Weight Shifts. The weight shift messages are handled separately because we send an additional weight. A weight of a vertex might be modified several times in LocalReduce, e.g., if an interface vertex participates in multiple cliques. It is also possible that its all its neighbors are eventually removed from the graph before the local quasi-kernel G'_i is determined. In both cases (multiple) weight messages are redundant since other PEs are at most interested in the final weights in G'_i. Therefore, we do not write the messages immediately, but mark the modified vertex and postpone the message creation. Vertices are marked by adding them to a set W_i if their weight were shifted in LocalReduce. Messages are only addressed to the adjacent PEs of marked vertices in W_i which remain in G'_i.

Algorithm 2 UpdateBorderAndSolution in KaDisReduS

```
Input: reduced subgraph $G_i$, proposed solution vertices $\tilde{\mathcal{I}}_i$, reduction offset $o_i$,
updates messages $U_i$, status updates $M_i$
Output: reduced subgraph $G_i$, proposed solution vertices $\tilde{\mathcal{I}}_i$, reduction offset $o_i$,
status updates $M_i$
procedure UpdateBorderAndSolution($G_i$, $\tilde{\mathcal{I}}_i$, $o_i$, $U_i$, $M_i$)
    for $(v, s) \in U_i$ do                                    ▷ received ghost and status
        if $v \in V_i^g$ then                                  ▷ locally not reduced yet
            if $s =$ moved and $v$ has hidden moved neighbor $u$ then
                decide moved cut-edge {v,u}                    ▷ tie breaking of 4.3.8
            else
                include or exclude $v$ according to $s$
                update $M_i$
            end if
        else if $s =$ included then                            ▷ $v$ is excluded locally
            $u \leftarrow$ hidden included neighbor of $v$
            follow tie breaking of 4.3.1
            if if $u$ loses tie-breaking then
                $\mathcal{I}_i \leftarrow \mathcal{I}_i \setminus \{u\}$ and $o_i \leftarrow o_i - \omega_i(u)$
            end if
        end if
    end for
end procedure
```

Figure 4.6.: This is the pseudo code processing received border updates in KaDisReduS.

The weight shift messages are synchronized first. Then another MPI_ Alltoallv exchanges the solution status messages. Thus, all PEs receive the correct weight of a ghost before it is possibly reduced when status updates have been received.

Receiving Border Updates. Once the updates were received, they are processed in UpdateBorderAndSolution in Algorithm 2. The most simple case is that a ghost (and its neighbors) should be reduced and assigned a solution status. If the ghost has already a solution status that conflicts with the received one, we can tie break the solution by the vertex identifier as described in Distributed Reduction 4.3.1. If a moved vertex u is received which has not been reduced yet, we need to reduce it. By now u has at most degree one; otherwise it was not moved. If it has degree zero, it is possible that a neighbor is moved to PE from which it received u. We check this by scanning its hidden neighborhood and search for the last moved neighbor which was moved in the current synchronization step. If there is such a moved neighbor v, we consider a cut-edge where both incident vertices were moved. We can include the vertex with the larger weight and break ties with the global vertex identifier as described in Distributed Heavy Vertex.

Reconstruction. We write the reduced vertices onto a stack so that we can handle the reconstruction of the solution according to each reduction rule in reversed order. In general, we cannot fully reconstruct the solution locally. The reason for that is that sent moved messages create dependencies in the reconstruction due to vertex folding and the moved degree one vertices.

For example, consider an interface vertex v that is moved to an adjacent PE $j \neq i$ to fold it into a ghost. If the fold succeeds at PE j, then it needs to notify PE i after unfolding v about its solution status. This requires sending a message with v and its solution status back to PE i when the solution is reconstructed. This must be early enough before PE j requires to know the solution status of v. Therefore, we push a *barrier* token on top of the stack whenever at least one PE sends a vertex move message in the synchronization step. Later, when we reconstruct the solution and encounter a barrier token on top of the stack, we need to block and communicate the solution status of unfolded ghosts.

4.6.3. Asynchronous Approach

Besides the synchronous approach KaDisReduS, we propose another distributed kernelization approach following an asynchronous communication scheme by using so-called *message buffer queues*. For this approach, we use the message-queue[1] library by Sanders and Uhl et al. [36] that was developed for asynchronous distributed algorithms solving the Triangle Counting problem.

Compared to KaDisReduS, we do not compute a local quasi-kernel with LocalReduce at each PE first, before any updates are exchanged in a synchronization step. Instead, we allow communication in-between in LocalReduce in an asynchronous fashion. Ideally, this approach mitigates idle times of PEs waiting for the other that need to finish computing their local quasi-kernel.

The high level idea is that each PE buffers the messages in dynamic arrays, for each receiving PE. The messages are written as we reduce interface vertices in LocalReduce. If now a buffer surpasses a threshold of $\delta > 0$ bytes, the buffer is flushed while the filled buffer is sent with MPI in a non-blocking manner. Each PE polls regularly for incoming updates from other PEs in a non-blocking fashion. In the mean time the LocalReduce routine can proceed and write message into the buffers if interface vertices are reduced. Although it is unlikely, PEs block, if the replaced buffer becomes full while the other ones are not completely sent yet. We denote this distributed kernelization approach KaDisReduA. We give the pseudo code in Algorithm 3.

Buffer Threshold δ. The buffer size threshold $\delta > 0$ is a hyper parameter in this approach addressing the running time cost of sending a single message of length ℓ in the network and local work. The communication cost can be modeled as $\alpha + \beta\ell$ where α is the start-up overhead of sending a message and β accounts for the message length ℓ.

[1] `https://github.com/niklas-uhl/message-queue`

Algorithm 3 KaDisReduA

```
Input: subgraph $G_i$
Output: reduced subgraph $G_i$, offset $o_i$
procedure KaDisReduA($G_i$, $\delta$)
    $\tilde{\mathcal{I}}_i \leftarrow \emptyset$                              ▷ proposed solution vertices $\tilde{\mathcal{I}}_i \subseteq V_i$
    $o_i \leftarrow 0$                                  ▷ reduction offset $o_i \in \mathbb{N}_0$
    $Q_i \leftarrow$ InitMessageBufferQueue($\delta$)        ▷ message buffer queue
    while true do
        LocalReduce($G_i$, $\tilde{\mathcal{I}}_i$, $o_i$)                  ▷ as in Section 4.4
        if Terminate($Q_i$) then        ▷ terminate unless global progress is made
            break
        end if
    end while
end procedure

Input: reduced subgraph $G_i$, proposed solution vertices $\tilde{\mathcal{I}}_i$, reduction offset $o_i$,
updates messages $U_i$, queue $Q_i$
Remark: Called if messages are received
procedure HandleMessages($G_i$, $\tilde{\mathcal{I}}_i$, $o_i$, $U_i$, $Q_i$)
    for ($v$, $s$, $x$) $\in U_i$ do        ▷ received ghost, status, and weight or last neighbor
        if $v \in V_i^g$ then                            ▷ locally not reduced yet
            if $s =$ moved and $v$ and $x$ is moved then
                decide moved cut-edge {v,u}                  ▷ tie breaking of 4.3.8
            else if $s =$ unset then                     ▷ $x$ is the new weight of $v$
                $w_i(v) \leftarrow x$
                send weight updates with $Q_i$
            else
                include or exclude $v$ according to $s$
                send status updates with $Q_i$
            end if
        else if $s =$ included then                      ▷ $v$ is excluded locally
            $u \leftarrow$ hidden included neighbor of $v$
            follow tie breaking of 4.3.1
            if if $u$ loses tie-breaking then
                $\mathcal{I}_i \leftarrow \mathcal{I}_i \setminus \{u\}$ and $o_i \leftarrow o_i - \omega_i(u)$
            end if
        end if
    end for
end procedure
```

Figure 4.7.: This is the pseudo code for the asynchronous distributed kernelization algorithm KaDisReduA in the perspective of PE P_i.

The buffer threshold δ controls now the message length ℓ. In addition to the time complexity of the communication, δ has an impact on the idle times and the running time of the local reduction phase. If δ is too large until a message buffer is sent, the receiving PE might become idle and is waiting for updates to proceed. On the other hand, if δ is small, receiving updates *instantly* may improve the running time of the local work. If it receives updates, it make a reduction progress and jumps back earlier to the first reduction rules in the order. These are the more efficient rules, e.g., Degree One. However, a small δ might increase the amount of sent messages drastically so that the start-up overhead might become the bottleneck of this approach.

Message Shape. The different message types are unified to a single type of shape $\overline{V}_i \times \{\mathsf{included}, \mathsf{excluded}, \mathsf{moved}, \mathsf{not\_set}\} \times \mathbb{N}_0$ at P_i. The status not_set indicates that the weight of the vertex changed. Compared to KaDisReduS, a move message for a moved interface vertex $v \in V_i$ also sends the remaining ghost neighbor. The receiving PE needs to know the cut-edge in order to check whether both vertices were each moved to the other process. In contrast to KaDisReduS, we do not know whether hidden moved neighbors of v were already processed already before v was moved. We use the slot for the weight to send the ghost neighbor in the move message.

Polling and Handling Messages. Messages can be received when the PE polls for messages; otherwise it asks the queue to terminate because no reduction progress was made and no messages were received so far. The queue terminates, if indeed no process has to send messages anymore and wants to terminate; otherwise if new messages are still received, the reduction process proceeds. LocalReduce polls for updates whenever it runs out of local work or reduction rules proposed to include or excluded vertices. The message handling works similar to the synchronous case because the same message types are sent.

The only difference is the handling of received moved ghosts. If a move message is received by PE i for a ghost v which already is excluded, then the process of v was not aware of the exclude before moving it. In this case, PE i sends an exclude message for v back.[2]

Filtering Messages. A key observation for this approach is that weight shift messages are written immediately when the weight of an interface vertex is decreased. Since the weight of a vertex can be decreased multiple times, outdated weight shift messages are filtered when the buffer should be sent. Similar to the synchronous approach, the filter removes vertex move messages if the ghost neighbor was already reduced. If this is the case, we remove also the potential weight shift messages from the buffer. Note that δ also has an impact on the message filtering since it controls the size of the filtered message.

Reconstructing the Solution. For KaDisReduS, we already discussed that a solution cannot be reconstructed without communication due to the dependency introduced

[2]This is how we have done it in the final version for the experiments. However, sending this exclude message is not necessary. Since v is already excluded, the respective PE will receive an include message which excludes v implicitly.

by vertex folding and moving. Here, we can again make use of message buffer queues. When a ghost is unfolded and receives a solution status, an update is written into the according message buffer. The status of a moved vertex is first needed when the reconstruction reaches a local folded vertex which depends on the status of a moved vertex. If such a moved vertex is encountered, we poll for messages until the required message is received. In the mean time the solution status is set for all those, where a solution status is received in the mean time.

4.7. Distributed Partition-And-Reduce

So far, we proposed two distributed kernelization algorithms using different communication approaches with the same set of distributed reduction rules. In this section we utilize the graph partitioner dKaMinPar by Sanders and Seemaier[35] with its fast configuration for our kernelization algorithms. Some of the distributed kernelization rules do not support to reduce border vertices, e.g., Distributed Generalized Neighborhood Folding. Consequently, as the number of process increases, the the reduction impact may diminish. Therefore, it may be of interest for certain graphs to find a partitioning that minimizes the cut, so that a good reduction impact is maintained for large numbers of cores. Moreover, a good partitioning can also reduce the cost of communication if there are overall fewer border vertices.

To that end, our kernelization algorithms support in addition to the plain kernelization two further variants utilizing dKaMinPar. The first variant *partitions* and then *reduces* the partitioned input graph, denoted PR. The second variant *reduces* the graph, partitions the quasi-kernel, and then *reduces* the partitioned quasi-kernel RPR. The intuition for RPR is that the first reduce can possibly shrink the graph size a lot which ideally yields a simpler partitioning problem. Moreover, the enclosing partitioning phase ideally rebalances the load between the processes. Note that the balanced graph partitioning problem is NP-hard. dKaMinPar is a heuristic solver. The graph partitioner is used with an imbalance of ε to compute a partitioning with p blocks if the first quasi-kernel has more than $C \times p$ vertices, where $C > 0$ is a constant; otherwise the quasi-kernel is gathered on a single process and sequentially reduced. We add PR or RPR in the name of KaDisReduA and KaDisReduS whenever used. Pseudo codes are show for PR in and for RPR in Figure 4.8.

Algorithm 4 PR

```
Input: subgraph $G_i$,
imbalance $\varepsilon$, #PEs $p$
Output: reduced subgraph $K_i$, offset $o_i$
procedure PR($G_i$, $\varepsilon$, $p$)
    if $p > 1$ then
        $\Pi_i \leftarrow$ partitionGraph($G_i$, $\varepsilon$, $p$)
        $G_i \leftarrow$ redistributeGraph($G_i$, $\Pi_i$)
    end if
    $K_i$, $o_i \leftarrow$ reduce($G_i$)
end procedure
```

Algorithm 5 RPR

```
Input: subgraph $G_i$ and imbalance $\varepsilon$,
partition threshold $C$, #PEs $p$
Output: reduced subgraph $K_i$, offset $o_i$
procedure PR($G_i$, $\varepsilon$, $C$, $p$)
    $K_i$, $o_i \leftarrow$ reduce($G_i$)
    if $|V|>0$ then                ▷ MPI_Allreduce
        $\Pi_i : V_i \to \{1, \ldots, p\}$
        if $|V| > Cp$ or $p > 1$ then
            $\Pi_i \leftarrow$ partitionGraph($G_i$, $\varepsilon$, $p$)
        else
            $\Pi_i \equiv 0$
        end if
        $K_i \leftarrow$ redistributeGraph($G_i$, $\Pi_i$)
        $K_i$, $o'_i \leftarrow$ reduce($K_i$)
        $o_i \leftarrow o_i + o'_i$
    end if
end procedure
```

Figure 4.8.: This is the pseudo code for *reduce-partition-reduce* PR and *reduce-partition-reduce* RPR in the perspective of PE P_i.

5. Distributed Independent Set

In the following, we propose different maximal independent set solvers. The goal of this chapter is to utilize our distributed kernelization algorithms as a preprocessing before applying a simple greedy algorithm. First, we introduce the distributed greedy algorithms in Section 5.1. Afterward, we combine them with our distributed kernelization algorithms in Section 5.2.

5.1. Distributed Greedy Algorithms

There are greedy algorithms for the unweighted problem following as discussed in Section 3.2. A solution provided by these algorithms is always a feasible solution for the weighted problem. However, they do not take into account to maximize the weight of an independent set.

We generalize greedy the algorithm DisMIS [1, 38] so that we have multiple vertices at each PE and use an extra greedy heuristic for weighted graphs. The greedy algorithm works in three steps. First, all vertices are rated using a greedy heuristic which is exchanged for the interface vertices with the other processes. Afterwards, vertices are greedily added to a local solution $\mathcal{I}_i$. At the border ties are broken between a considered interface vertex and a ghost with the global vertex identifier to prevent solution conflicts. This global ranking of the vertices introduces a dependency between the processes because local vertices cannot be decided before ghosts at other processes were decided first. Therefore, the solution status of the interface vertices must be communicated regularly.

We attempt to add the local vertices greedily to a local solution $\mathcal{I}_i$ using the heuristic greedy rating weight_diff, i.e., $r(v) := w_i(v) - \omega(N_i(v)$, which takes the weights into account. For interface vertices, we ensure that not conflicts arise by tie-breaking with ghost via the global vertex identifier. More precisely, if an interface vertex $v \in V_i$ has the same rating as one of its ghost neighbors which has a smaller vertex identifier, we refrain from including v into the independent set $\mathcal{I}$.

Similar to the distributed kernelization algorithms in Section 4.6, the solution status of the interface vertices is communicated with the adjacent PEs. Once, the updates are received, the PEs can remove the decided border vertices from their local subgraphs. Then, they try to add the remaining free vertices, to a solution. In the following, we propose a synchronous and an asynchronous greedy algorithm similar to the distributed kernelization algorithms in Section 4.6. We assume the graph is stored our distributed dynamic graph data structure as described in Section 4.5.

5.1.1. Synchronous Greedy Algorithm

The synchronous distributed greedy algorithm works similar to DisMIS in supersteps. Each superstep consists of local work followed by communication step to synchronize the state between the processes. However, in contrast to DisMIS, our algorithm does not follow a vertex-centric approach.

The pseudo code is given in Algorithm 6. The idea is to determine the rating once and exchange it for the ghosts. Then, the vertices are processed and included whenever possible if they are minimal according to the the strict order in their neighborhood. We hide the neighbors once they are included or excluded. If a vertex is excluded, the neighbors are marked so that they are reconsidered for next scan over the remaining vertices. A synchronization step is performed if all processes are idle,i.e., cannot decide anymore vertices. Then include and exclude messages for the interface vertices are sent to the adjacent PEs with an MPI_Alltoallv. They are used to hide the according border vertices. Interface vertices are marked if their neighborhood changes. Finally, once all PEs decided all vertices, each PE returns its local solution $\mathcal{I}_i \subseteq V_i$.

5.1.2. Asynchronous Greedy Algorithm

The asynchronous greedy algorithm, called GreedyA, works works very similar to the synchronous greedy algorithm. The key difference is that we use a message buffer queue as for KaDisReduA to exchange the message in an asynchronous fashion. We give the pseudo code in Algorithm 7. We write the include and exclude messages immediately into the message buffers. We poll regularly for incoming messages.

5.2. Distributed Reduce-And-Greedy

The idea is now to utilize our distributed kernelization algorithms and combine them with the presented greedy algorithms. First, we reduce the input graph to determine a quasi-kernel K, and then apply a distributed greedy algorithm to K. We call this algorithmic scheme ReduceAndGreedyMaxIS in the following. Algorithm 8 gives the high level pseudo code. Although the distributed greedy algorithm computes a maximal independent set for K, it is not guaranteed that the built solution for G is maximal. The reason for this issue are reductions which exclude vertices under the assumption that an optimal solution for the reduced graph is found. To that end, we maximize the solution with a variant of the greedy algorithm which uses weight, i.e., the vertex weight, for the heuristic greedy rating. It takes the independent set as input and greedily adds the remaining free vertices into the solution. We propose an synchronous and an

Algorithm 6 GreedyS

```
Input: subgraph $G_i$
Output: local maximal independent set vertices $\mathcal{I}_i$
procedure GreedyS($G_i$)
    $r(v) \leftarrow$ weight_diff($v$) for every $v \in V_i$          ▷ compute rating locally
    ReplicateRatingForGhosts($G_i$, $r$)                              ▷ MPI_Alltoallv
    $C \leftarrow V_i$                                                ▷ candidate Queue
    $M_i \leftarrow \emptyset$                                        ▷ decided interface vertices
    $\mathcal{I}_i \leftarrow \emptyset$                              ▷ local greedy solution, $\mathcal{I}_i \subseteq V_i$
    while $|V| > 0$ do                                                ▷ MPI_Allreduce
        while $C \neq \emptyset$ do                                   ▷ while not idle
            $c \leftarrow$ pop($Q$)
            if $u \notin V_i$ then
                continue                                              ▷ $u$ received solution status
            end if
            if $r(c) \geq r(u)$ for each $u \in N_i(u)$ then
                if $c$ is has ghost neighbor $u$ with $r(c) = r(u)$ and $u < c$ then
                    continue
                end if
                if $c$ is interface vertex then
                    $M_i \leftarrow M_i \cup \{c\}$
                end if
                hide($G_i$, $c$); $\mathcal{I}_i \leftarrow \mathcal{I}_i \cup \{c\}$        ▷ include $c$
                for $u \in N_i(v)$ do
                    hide($G_i$, $u$);                                 ▷ exclude neighbor $u$
                    if $u$ has ghost neighbors then
                        $M_i \leftarrow M_i \cup \{u\}$
                    end if
                    $C \leftarrow C \cup N_i(u)$
                end for
            end if
        end while
        ▷ MPI_Alltoallv for include/exclude messages from/to adjacent PEs
        $C \leftarrow$UpdateBorder($M_i$)
        $M_i \leftarrow \emptyset$
    end while
    return $\mathcal{I}_i$
end procedure
```

Figure 5.1.: This is the pseudo code for the synchronous distributed kernelization algorithm GreedyS in the perspective of PE P_i.

asynchronous variant. KaDisReduS-RG uses KaDisReduS with the synchronous greedy

Algorithm 7 GreedyA

```
Input: subgraph $G_i$, message buffer queue threshold $\delta$
Output: local maximal independent set vertices $\mathcal{I}_i$
procedure GreedyA($G_i$, $\delta$)
    $r(v) \leftarrow$ weight_diff($v$) for every $v \in V_i$          ▷ compute rating locally
    ReplicateRatingForGhosts($G_i$, $r$)                               ▷ MPI_Alltoallv
    $C_i \leftarrow V_i$                                               ▷ candidate queue
    $Q_i \leftarrow$ InitMessageBufferQueue($\delta$)                  ▷ message buffer queue
    $\mathcal{I}_i \leftarrow \emptyset$                               ▷ local greedy solution, $\mathcal{I}_i \subseteq V_i$
    while true do
        while $C_i \neq \emptyset$ do
            $c \leftarrow$ pop($Q$)
            if $u \notin V_i$ then
                continue                                               ▷ $u$ received solution status
            end if
            if $r(c) \geq r(u)$ for each $u \in N_i(u)$ then
                if $c$ is has ghost neighbor $u$ with $r(c) = r(u)$ and $u < c$ then
                    continue
                end if
                if $c$ is interface vertex then
                    send($Q_i$, ($c$, included)) to all PEs of $\mathcal{R}_i(c)$
                end if
                hide($G_i$, $c$); $\mathcal{I}_i \leftarrow \mathcal{I}_i \cup \{c\}$          ▷ include $c$
                for $u \in N_i(v)$ do
                    hide($G_i$, $u$);                                  ▷ exclude neighbor $u$
                    if $u$ has ghost neighbors then
                        send($Q_i$, ($u$, excluded)) to all PEs of $\mathcal{R}_i(u)$
                    end if$C \leftarrow C \cup N_i(u)$
                end for
            end if
            ▷ handling: hide vertices, (write new updates), mark modified neighbors
            poll($Q_i$)                                                ▷ poll and handle incoming messages
        end while
        if terminate($Q_i$) then                                       ▷ send buffers/receive messages
            break                                    ▷ all processes finished; all vertices decided
        end if
    end while
    return $\mathcal{I}_i$
end procedure
```

Figure 5.2.: This is the pseudo code for the synchronous distributed kernelization algorithm GreedyS in the perspective of PE P_i.

Algorithm 8 ReduceAndGreedyMaxIS

```
Input: subgraph $G_i$
Output: local solution vertices $\mathcal{I}_i \subseteq V_i$
procedure ReduceAndGreedyMaxIS($G_i$)
    $K_i$, $o_i \leftarrow$ reduce($G_i$)
    $\mathcal{I}_K \leftarrow$ greedy($K_i$)
    $\mathcal{I}_i \leftarrow$ applyReductions($K_i$, $\mathcal{I}_K$)
    $\mathcal{I}_i \leftarrow$ greedyMaximize($G_i$, $\mathcal{I}_i$)
    return $\mathcal{I}_i \subseteq V_i$
end procedure
```

Figure 5.3.: This is the pseudo code of the scheme ReduceAndGreedyMaxIS in the perspective of PE P_i.

algorithm GreedyS and KaDisReduA-RG uses KaDisReduA with the asynchronous greedy algorithm GreedyA.

6. Implementation Details

Before we discuss our experiments, we want to give an overview over the core libraries that are used in our implementation and features of our software architecture.

Our algorithm is designed for `C++20`. Communication between the PEs is realized with IntelMPI/2021.11. To circumvent using old MPI interface, we use a wrapper library for MPI, called KaMPIng by Hespe et al. [26]. We use KaGen [15] to read, write, and generate graphs. All our algorithms support the static graph data structure of KaGen as input. Furthermore, we use dKaMinPar by Sanders and Seemaier [35] with the fast configuration as distributed graph partitioner. Some of our distributed data reduction rules solve an MWISP as subproblem. The subproblem size can be controlled with an extra parameter. In this case, we use the branch-and-reduce solver KaMIS_wB&Rby Lamm et al. [29] to find exact solution. For mapping the global vertex identifiers of ghosts to their local vertex identifier, we use the `abseil:flat_hash_map`[1].

Our software architecture supports to add distributed data reductions once and use them with both communication approaches. Although we provide a distributed algorithm, all out distributed data reduction rule implementations stay very close to the implementations in KaMIS_wB&R [29] and m^{2}wis [22]. We hope this simplifies to add more data reduction rules in the future. Our provided algorithms are available in the repository distributed-kernelization[2]

[1]URL to abseil: `https://abseil.io/docs/cpp/guides/container`

[2]URL to the distributed-kernelization repository: `https://github.com/jabo17/distributed-kernelization/`

7. Experiments

We conducted strong scaling and weak scaling experiments to compare our distributed kernelization algorithms. Therefore, we investigate the impact of our reductions regarding the kernel size in Section 7.2. In Section 7.3 we investigate the impact of a good partitioning regarding our kernelization algorithms. Afterward, we compare our distributed maximal independent set solvers against the state-of-the-art reduce-and-peel solver HtWIS in Section 7.4. Finally, we present our weak-scaling experiments in Section 7.5.

Each analysis is summarized in a short observation. Observation 1 summarizes the results of the impact of our reductions in the strong scaling experiments. In Observation 2 we compare the differences that arise when the graph is partitioned so that the cut is minimized. Observation 7.4 concludes our experiments with the different distributed independent set solvers and HtWIS. Last, Observation 4 recapitulates our weak-scaling experiments.

7.1. Methodology

Machine and Setup. We conducted our experiments on the cluster *SuperMUC-NG* interconnected by Intel OmniPath. Each compute node provides two Intel Skylake Xeon Platinum 8174 sockets, with 24 cores each, and features 96 GB of RAM. Each configuration of the compared algorithms was run three times. For each run, we set a different random seed if supported by the algorithm. We map each process to one core. Throughout the experiments, we represent local (global) node identifiers with 32 (64) bit integers, local edges with 32 bits, local (global) node weights 32 (64) bits. All algorithms were compiled with `g++/12.2.0` and `intel-mpi/2021.11` with full optimizations (`-O3`) turned on. Our algorithms are implemented in `C++20`. We set a maximum subproblem size of 100 vertices for our distributed data reduction rules and a message buffer queue threshold of 16 000 Bytes.

Strong Scaling. For our benchmarks, we use a data set, denoted STRONG, consisting of 18 graphs. It contains nine SNAP [**SnapDatasetsLeskov2014**] graphs and one mesh graph [**EfficientTraveSander2008**] which are also used in other MWIS benchmarks [29, 17], two Open Street Map[1] (OSM) instances from the 10-th DiMACS challenge [**GraphPartition2013**, **BenchmarkingFoBader2014**] which we assigned

[1]Open Street Map: `https://www.openstreetmap.org/`.

uniform distributed weights from one to 50, six graphs generated with KaGen [15] with uniform distributed weights one to either 100 or 50. The number of vertices and edges ranges from one up to 50 Mio. vertices and 1.9 up to 54 Mio. edges. Overall, we cover random grid, random geometric (rgg), random hyperbolic (rhg), mesh, road, graph and social networks. For detailed meta information and their full names, see Table A.1. For data set STRONG, we set a time limit of 320 s.

Weak Scaling. For our weak scaling experiments, we use the graph generator KaGen by Funke et al. [15] which allows us to generate graphs of different graphs families distributed and assign the vertices. Here, we fix the number of vertices 2^N and edges 2^M per process and linearly increase the number of vertices in the number of cores p, i.e., for p cores, we consider a graph with $n := 2^N p$ vertices. We consider 2D and 3D *random geometric* (rgg_{2D} and rgg_{3D}) and *random hyperbolic* (rhg). We choose $2^N = 2^{20}$ vertices with uniform weights from 1 to 100 and $2^M = 2^{23}$ edges which results in an average degree of 16. For rhg we set a power-law exponent of 2.6. For the weak scaling experiments we set a time limit of 300 s. Our distributed independent set solver use the output partitioning of the generated graph.

Metrics And Plots. We often consider the *relative speedup* of an algorithm for p cores, i.e., t_1/t_p where t_1 is the sequential geometric mean running time and t_p the geometric mean running for p cores. In the weak scaling experiments we investigate the *throughput*, i.e., edges per second processed. Moreover, we investigate the *quality solution* over all instances by comparing the geometric mean solution quality to the geometric mean of the best solutions found by any solver. Boxplots give insights into an empirical distribution. From top to bottom, a boxplot marks the upper-whisker, the third quartile (75-th percentile), the median (second quartile or 50-th percentile), the first quartile (25-th percentile), and the lower-whister of a distribution. The first (third) quartile is marked at the bottom (top) of the bar, 25 % (75 %) of the data fall below this mark. The upper whisker it the largest first data point that lays below the third quartile plus 1.5 times the interquartile range. Analogously, the lower quartile is defined as the smallest data point that is larger than the first quartile minus 1.5 times the interquartile range. The range between the first and the third quartile (the bar) defines the so-called *interquartile range* where 50 % of the data lay in the *middle*. In addition, we put an overlay above the boxplots with the data points which are jittered on the x-axis to better visualize the empirical distribution.

7.2. Reduction Impact

We start this evaluation with a strong scaling experiment investigating the reduction impact on data set STRONG (18 instances). In particular we are interested in the strong scaling behavior for both communications approaches (KaDisReduS and KaDisReduA) for up to 1 024 cores. This concerns the reduction impact in terms of reduced vertices and edges, and further the strong scaling of the running time.

Kernel Size. Some of the data reduction rules depend on a sufficient locality because they do not apply to the border. This locality might not be given among some graphs when they are distributed among a large number of processes. Moreover, to this assumption contributes that in general graphs are not partitioned before they are assigned to processes. On the contrary, it still holds that these rules might become applicable once other rules reduced border vertices. Nonetheless, we suspect to observe a decline in the reduction impact for larger numbers of cores.

Figure 7.1 shows boxplots for the number of vertices and edges in the reduced graphs, relative to to the input graph, for both kernelization approaches, KaDisReduS and KaDisReduA.In general, both approaches communicate border updates at different time steps due to their design and the parallel execution which introduces non-determinism. Although both approaches use the same reduction rules, these two factors can lead to different reduction applications which eventually might lead to different reduced graphs. However, in this effect appears to be rather small as we observe in Figure 7.1 and Table 7.1 that the **reduction impact between KaDisReduS and KaDisReduA is almost the same** for any number of processes. When running either of both approaches sequentially, the number of vertices (edges) can be reduced down to at least 19 % (13 %) for 75 % of the instances with a median of 4.8 % (6%). Whereas the reduction impact for the other 25 % seems to have only moderate impact with respect to our set of reduction rules.

For 64 processes we can observe a decline in the reduction impact, as mentioned above, on some instances. The median fraction of vertices remaining in the reduced graph grows to 24 % while the third quartile reaches 47 % for both approaches.

For $p \geq 64$ processes that the third quartile for the fraction of vertices in the reduced graph, grows sub-linear in the number of processes. Furthermore, the **median relative number of vertices in the quasi-kernel remains steady at around** 24 %. Similar observation can be made for edges. We conclude from these observations regarding the empirical distribution of the kernel sizes that for half of the instances, we benefit from data reduction rules which are not restricted to non-border vertices. Still, for the other half, the decline in the reduction impact becomes more noticeable in terms of the relative kernel size (vertices and edges).

Running Time. Now we investigate the running time and relative speed-ups of the reduction phase, i.e., the running time from initialization of kernelization algorithm until the kernel K is determined. Figure 7.2 shows the running times in seconds s for both kernelization approaches using boxplots. Note the logarithmic scaling of the y-axis (running time). For $p = 1$, the running times of KaDisReduS and KaDisReduA range from 0.86 s to 84.9 s and from (1 s to 85.3 s), respectively.

Table 7.1 summarizes the geometric mean reduction times of both approaches for 1, 64 and 1 024 cores. For 64 and 1024 processes, KaDisReduS decreases the geometric mean reduction time from 5.47 s down to 0.37 s and 0.10, respectively. KaDisReduA performs better for $p > 1$. For 64 and 1024 processes, KaDisReduA decreases the reduction time from 6.09 s down to 0.31 s and 0.07 s on average, respectively. Thus, **KaDisReduA is**

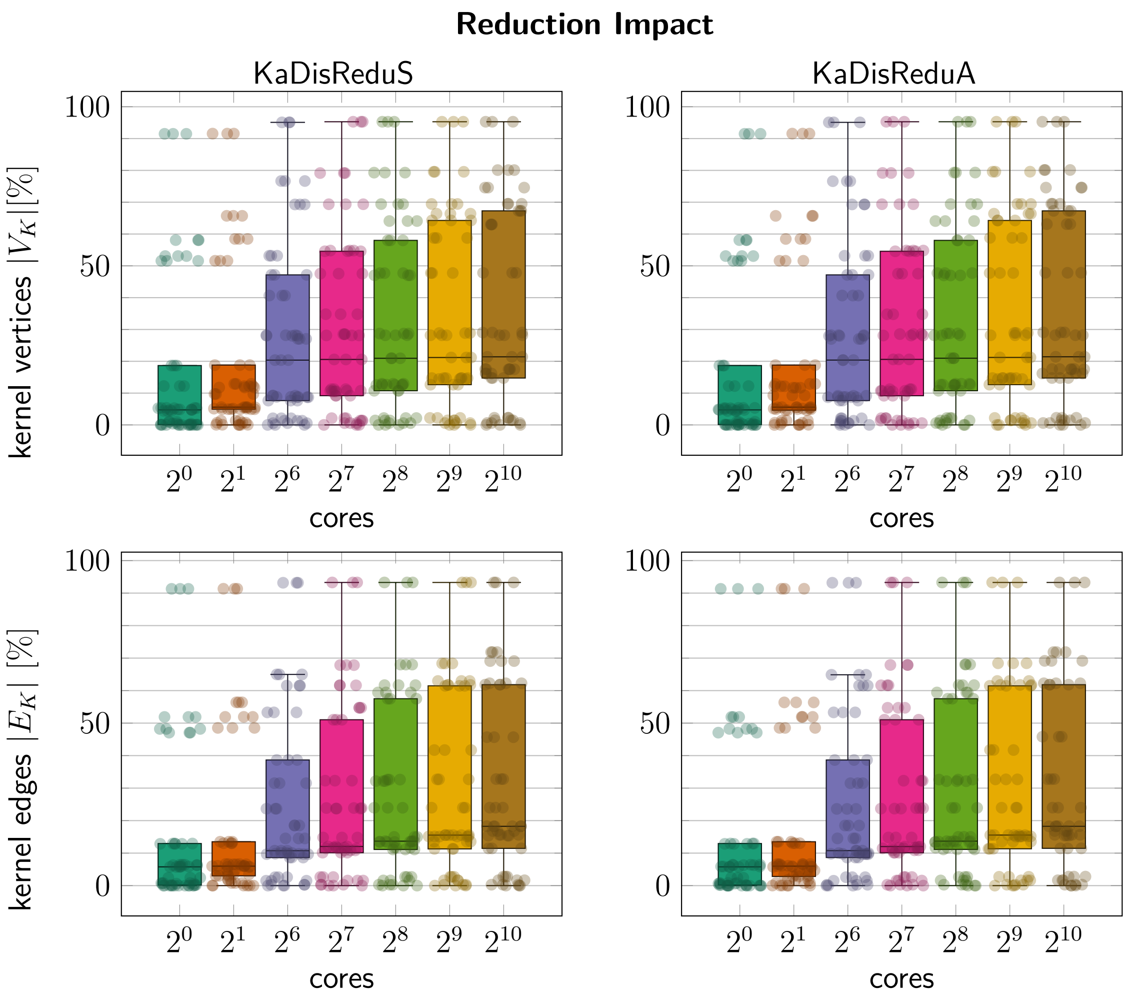

Figure 7.1.: The Figure shows the relative quasi-kernel sizes for both kernelization approaches, (KaDisReduS and KaDisReduA), and different choice of cores (here 1, 2, 64, 128, 256, 512, and 1 024 cores) with the help of boxplots. The first row shows the number of vertices of the quasi-kernel K, relative to the number of vertices in input graph G, for data set STRONG (18 graphs). Analogously, the second row provides boxplots for the relative number of edges in K.

19 % and 43 % faster than KaDisReduS on average for 64 and 1 024 processes, respectively.

Regarding the empirical distribution of the running times in Figure 7.2, we especially notice for $p > 128$ that KaDisReduS only hardly improves upon the best running times obtained for $p = 64$. In contrast, KaDisReduA yields on some instances for 1 024 processes running times that are at least a factor 5 faster than on 64 s.

Figure 7.3 shows the empirical distributions of the speed-ups of the reduction times for data set STRONG. The median relative speed-up of KaDisReduS stagnates for $p > 128$ at 20, indicating that KaDisReduS can only hardly improve upon the running

	median $\lvert V_K \rvert$ [%]			geo. mean t [s]		
	cores p			cores p		
Algorithm	1	64	1 024	1	64	1 024
KaDisReduS	**4.80**	23.70	24.80	**5.47**	0.37	0.10
KaDisReduA	**4.80**	23.70	24.81	6.09	**0.31**	**0.07**
KaDisReduA-PR	**4.80**	5.44	6.28	6.10	2.17	2.54
KaDisReduA-RPR	**4.80**	**5.03**	**5.63**	6.09	1.26	1.44

Table 7.1.: The table summarizes the reduction impact and running time of our different distributed kernelization algorithms on data set STRONG (18 graphs). It shows the median relative number of vertices in the quasi-kernel and the geometric mean running time to reduce a graph with 1, 64, and 1 024 cores. A value is bold if it is the best among both algorithms for the according p. Note that for $p = 1$ KaDisReduA-PR and KaDisReduA-RPR do no graph partitioning at all. Therefore, they are identical to KaDisReduA.

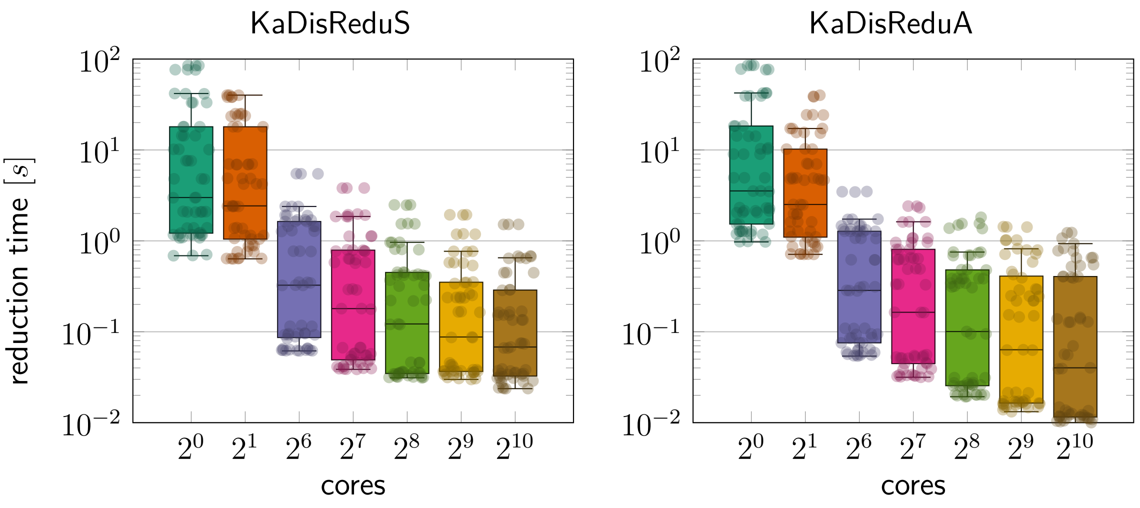

Figure 7.2.: The boxplots show for both kernelization approaches, KaDisReduS and KaDisReduA, and each choice of cores (here 1, 2, 64, 128, 256, 512, and 1 024 cores) the running times for reducing the 18 graphs of data set STRONG. Note the logarithmic scaling of the y-axis (running time).

time for larger p. Whereas KaDisReduA still observes moderately increasing relative speed-ups, e.g., median relative speed-ups of 27, 69 and 148 for 64, 256 and 1 024 processes is achieved, respectively.

We conclude our first experiments regarding the reduction impact in Observation 1.

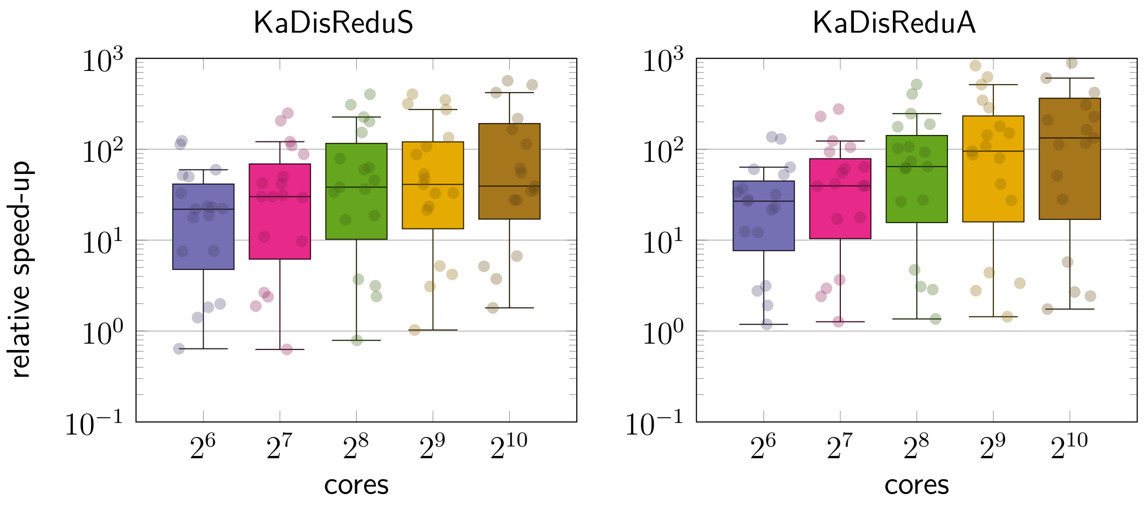

Figure 7.3.: The boxplots show for both kernelization approaches, KaDisReduS and KaDisReduA, and each choice of cores (here 1, 2, 64, 128, 256, 512, and 1 024 cores) the relative speed-ups (relative to $p = 1$) of the reduction time for the 18 graphs of data set Strong. Note the logarithmic scaling of the y-axis (relative geometric mean speed-up).

Observation 1: Reduction Impact of KaDisReduS and KaDisReduA Both approaches yield similar sized kernels while also both maintain a good reduction impact for larger numbers of cores, i.e., at most 20% of the vertices remain in the kernel for any $p \leq 1\,024$ for half of the instances. KaDisReduA outperforms KaDisReduS in terms of running time, especially for larger numbers of cores. Whereas the relative speed-ups of KaDisReduA increase moderately up to 148 at median for 1 024 processes while the relative speed-up of KaDisReduS seems to grow at least slower.

7.3. Reduction Impact on Partitioned Graphs

In the following, we investigate whether we can maintain the reduction impact where the instances of data set Strong are partitioned so that the cut is minimized cut. Moreover, we compare the running times of the reduction phase (with and without the partitioning time) with the reduction times for the unpartitioned instances.

We allow an imbalance of $\varepsilon = 3\%$ using unit vertex weights for the cost function. The partitioning is determined with dKaMinPar [35] using its fast configuration. We compute the partitioning every time before we run our kernelization algorithm using a different seed. For this analysis, we only use KaDisReduA as it performs better in terms of running than KaDisReduS when scaling to a large number of cores.

Table 7.2 shows the changed number of vertices in the kernel, the difference in the cut, and compares the geometric mean speed ups over the sequential running time of KaDisReduA for 1 024 cores. First, we note for both, unpartitioned and partitioned instances that graphs experience a larger speed-up if reduction impact smaller. The cut is improved by 35.79% on median. As a result, more vertices can be reduced on every instance. For unpartitioned instances, the median the kernel size increases by 14.36% and on maximum by 55.07% compared to reducing sequentially.. When the instances are partitioned, we obtain almost the same kernel size: On median the kernel increases only by 0.97% and on maximum by 10.52%. Regarding running time, we note that KaDisReduA is 19% slower on the partitioned instances compared to the unpartitioned instances with a geometric mean speedup of 89.28 and 72.34, respectively. This observation may be a result of more local work on partitioned instances. Multiple data reduction rules fail very early because they cannot be applied at the border. If now a better partitioning is given, the reduction rule is tested beyond this first obstacle and can possibly be applied.

Table 7.2 shows that road networks can be reduced with speed ups up to 164 while number of vertices in the kernel increases by at most 12 %. If the graph is now partitioned, speed-ups range in the same order of magnitude while the kernel size by at most 1 %.

Figure 7.5 shows the empirical distributions of the reduction time and speed-ups for partitioned and unpartitioned instances. The outliers with steady large running times indicate that there are instances where partitioning by minimizing cut-edges makes a good scaling of the reduction time impossible. We suspect that reductions such as Distributed Generalized Neighborhood Folding where solving the subproblem is sometimes still a bottleneck in the overall running time. A more restrictive choice of the maximum subproblem size might help.

We now compare the reduction impact of the partition-and-reduce (PR) and reduce-partition-reduce (RPR) scheme which we conducted with the asynchronous approach. In Figure 7.4, boxplots show the relative kernel sizes in terms of vertices. We note that the third quartile is sometimes a bit smaller for KaDisReduA-RPR than for KaDisReduA-PR. Overall they our reductions have a very similar impact and are able maintain the reduction if the number of cores is increased. Regarding running, Figure 7.6 indicates that it helps sometimes to reduce first and to partition in-between. We conclude our observations in Observation 2.

Observation 2: Reduction Impact on Partitioned Graphs KaDisReduA maintains the reduction impact for most graphs on a large number of cores if a partitioning is chosen that minimizes the cut. Speed-ups may decrease slightly from 90 to 72 on average which is possibly caused by more local work.

Graph		Sequential		Unpartitioned		Partitioned		
		$\lvert V_K\rvert$ [%] ▼	t [s]	$\Delta\lvert V_K\rvert$ [%]	su	$\Delta\lvert V_K\rvert$ [%]	su	Δcut [%]
`snap`	wiki-Talk	0.00	2.05	0.01	1.74	0.00	4.80	−44.90
`snap`	com-youtube	0.00	0.98	0.60	2.42	0.02	13.88	−44.42
`kagen`	rhg-N20	0.00	1.18	47.82	2.69	0.00	9.98	−81.44
`kagen`	rhg-N22	0.00	3.52	28.20	5.75	0.00	18.70	−84.26
`osm`	europe	0.12	39.28	1.34	307.45	0.00	161.20	−11.33
`osm`	asia	0.40	9.00	1.84	227.74	0.00	234.33	−8.53
`snap`	LiveJour.	0.98	42.29	28.10	51.26	6.23	11.28	−45.07
`snap`	as-skitter	4.40	18.35	17.02	28.13	3.47	1.99	−56.90
`snap`	roadNet-TX	4.72	1.53	10.02	133.17	0.72	115.01	−17.01
`snap`	roadNet-CA	4.88	2.30	11.29	164.34	0.77	146.91	−16.53
`snap`	roadNet-PA	5.26	1.28	12.29	112.40	0.96	100.11	−19.68
`snap`	roadNet-PA-uf	5.36	1.27	12.13	117.30	0.97	104.05	−19.50
`mesh`	buddha	12.21	2.14	55.07	209.29	5.93	170.33	−38.71
`kagen`	rgg2d-N20M22	18.66	4.94	44.45	420.63	3.37	267.24	−41.53
`kagen`	grid2d-XY12	51.56	76.87	23.02	1 273.87	1.69	739.66	−11.69
`snap`	pokec-rel.	53.07	85.09	16.43	606.55	10.52	140.17	−35.67
`kagen`	rgg3d-N20M22	58.09	10.35	22.04	891.88	6.92	503.50	−35.90
`kagen`	grid3d-XYZ7	91.49	14.19	3.80	1 123.83	1.92	670.30	−26.45
	overall	4.80	6.09	14.36	89.28	0.97	72.34	−35.79

Table 7.2.: This table shows for KaDisReduA the impact of a partitioning for data set STRONG regarding the reduction impact on 1 024 cores. For each instance the median number of vertices in the kernel, relative to the number of vertices in the graph, and the running time for reducing them sequentially is given. Furthermore, $\Delta|V_k|$ shows the difference of the number of vertices in the kernel compared to the sequential kernel, relative to the number of vertices in the graph. For both, partitioned and unpartitioned instances, the geometric mean speedup is given. Last, the difference in the cut compared to the unpartitioned instances is show. It is given relative to the number of edges in the input graph. The instances are sorted ascending regarding the relative kernel size.

7.4. Comparison of Distributed Independent Set Solvers

In the following, we compare our distributed maximal independent set solvers on data set STRONG. We consider GreedyS, GreedyA, KaDisReduA-RG, KaDisReduS-RG, and KaDisReduA-RPRG. Note that KaDisReduS-RG uses the synchronous greedy algorithm while the others use the asynchronous variant. To investigate their solution quality and speed-ups, we compare them with the state-of-the-art reduce-and-peel solver HtWIS by Gu et al. [23]. Note that we cannot compare us with the distributed kernelization algorithm by George et al. [19] for MISP because the implementation is no longer available. The summarized results are given in Table 7.3.

Solution Quality. First, we investigate the geometric mean solution quality where we compare the solutions of a solver against the best solutions found by any of the compared solvers. GreedyA performs worst with a solution quality of 94.522 % on average. HtWIS yields the best solution quality on average with 99.609 %. Algorithms

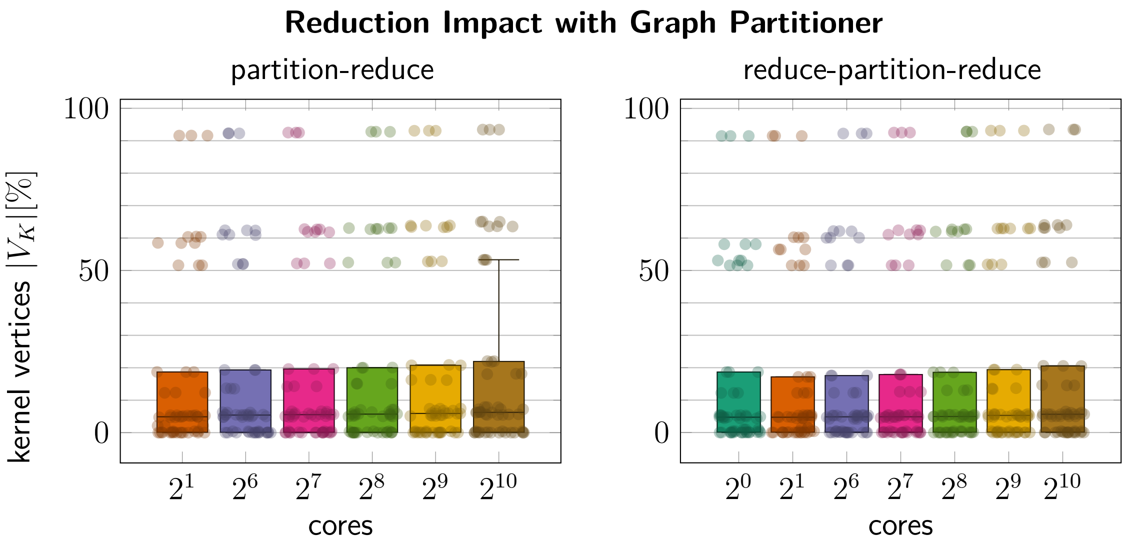

Figure 7.4.: The Figure shows the empirical distributions of the relative number of vertices in the kernel for KaDisReduA-PR and KaDisReduA-RPR for the partitioned and unpartitioned graphs of Strong.

that make use of our distributed kernelization algorithms yield a solution quality that ranges in-between on any number of cores. For $p = 1$, they all reach a solution quality of 98.168 % on average. For KaDisReduA-G and KaDisReduS-G, we observe a decline in the solution quality which goes down to 96.85 % on average. Nonetheless, the solution is still better than the those found by the greedy algorithms, e.g.,GreedyA is 2.33 % worse on average for 1 024 cores. Both approaches that utilize the graph partitioner, can maintain the solution quality on average. KaDisReduA-RPRG performs slightly better than KaDisReduA-PRG on 1 024.

Running Time. Beside the geometric mean running times and speed-ups over all instances, we show the empirical distributions of the speed-ups in Figure 7.7. HtWIS and GreedyA have a similar running times on geometric mean, i.e., 1.689 s and 1.691 s, respectively. For 64 and 1 024 cores we observe geometric mean speed-ups of 11 and 37 over HtWIS for GreedyA. In the same order of magnitude, we observe for KaDisReduA-G and KaDisReduS-G a geometric mean speed-up of 15 and 12 for 1 024, respectively. For a the grid graph `grid2d-XY12` they obtain the largest speed-up of 173 and 149, respectively. For KaDisReduA-PRG and KaDisReduA-RPRG, the running times start to increase when the number of processes is increased. For 1 024 cores, KaDisReduA-RPRG is a factor 1.7 faster than KaDisReduA-PRG while it is a factor 10 slower than KaDisReduA-RG on average. In a more detailed analysis, we noted that the graph partitioner becomes a bottleneck in the overall running time.

We summarize our observations in Observation 7.4.

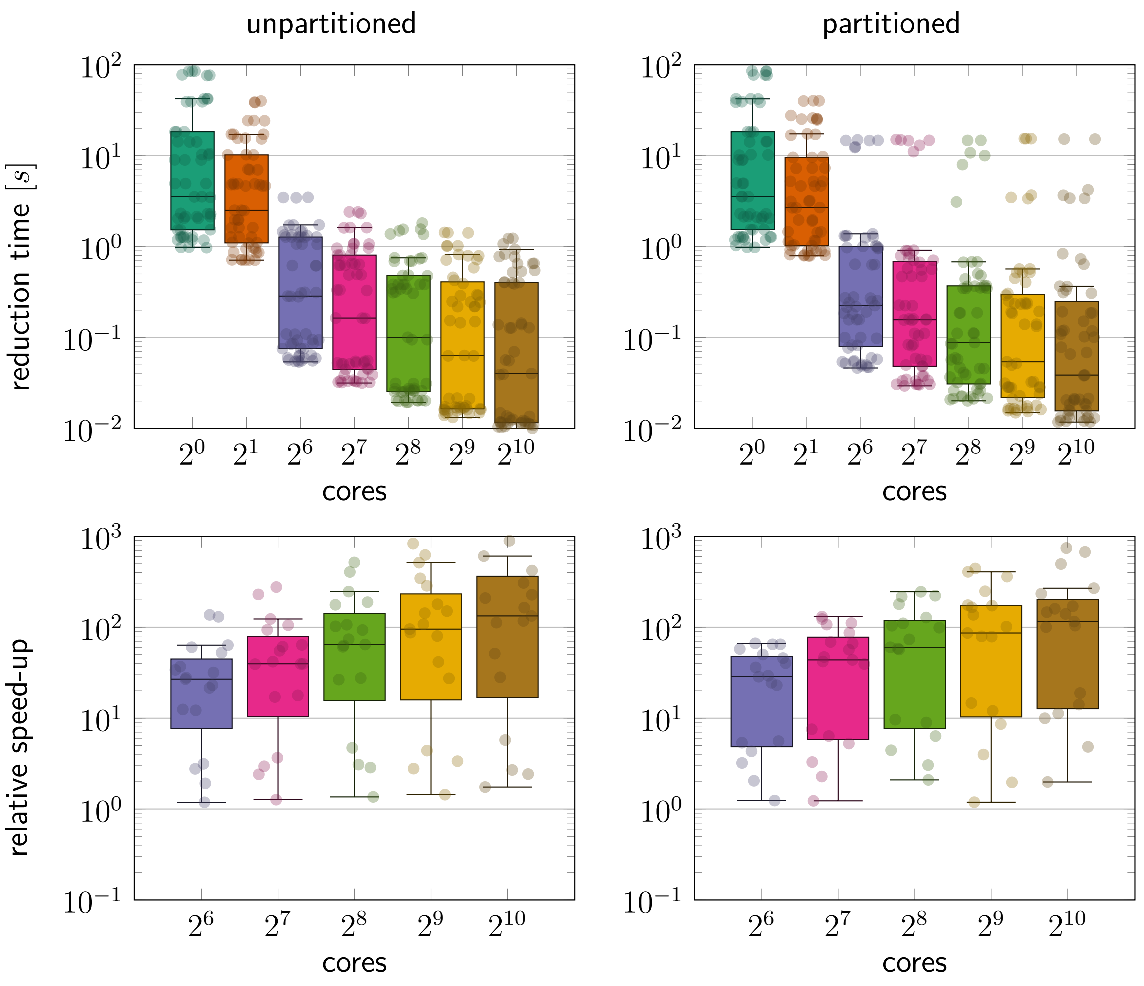

Figure 7.5.: These boxplots show for the unpartitioned and partitioned instances the empirical distributions of the reductions times. Moreover, in the row below the speed-ups are shown. These are the results obtained with KaDisReduA.

Observation 3: State-of-the-Art Experiments Our experiments indicate that using our distributed kernelization algorithms significantly improves on the solution quality of a greedy algorithm with and without utilizing a graph partitioner. Utilizing graph partitioners allows us to almost fully maintain the solution quality for up to 1 024 cores but does not scale not in terms of running time for this set of instances. In a comparison with the reduce-and-peel solver HtWIS, we show that combining our set of reduction rules with a simple greedy algorithm reaches a solution quality close to the one of HtWIS. Moreover, KaDisReduA achieves speed-ups of 15 on geometric mean up to 170 for 1 024 cores over HtWIS.

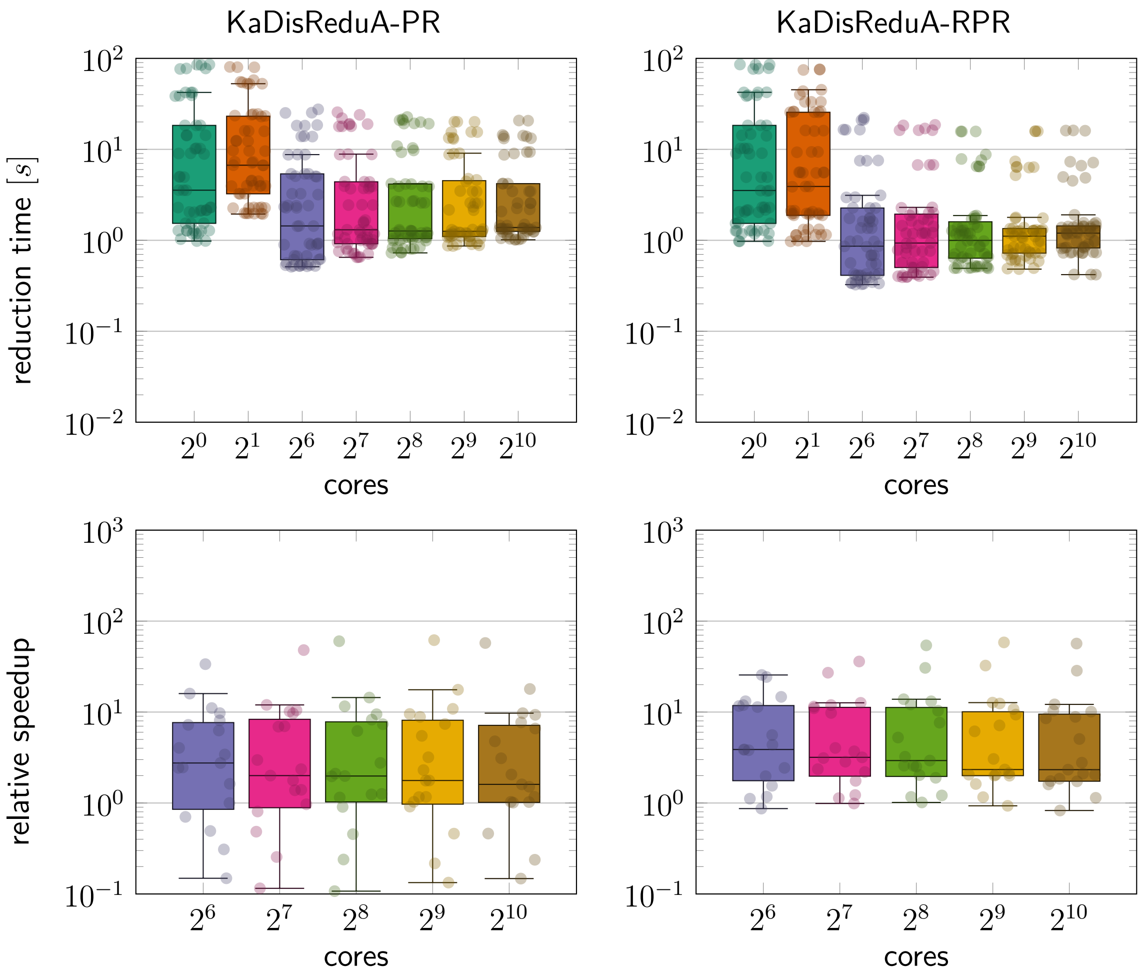

Figure 7.6.: The boxplots show for KaDisReduA-PR (partition-and-reduce) and for KaDisReduA-RPR (reduce-partition-reduce) the running times (reduction and graph partitioning) as well as the relative speed-ups for data set Strong (18 graphs).

7.5. Weak Scaling Experiments

In the weak scaling experiments, we investigate the throughput of edges per seconds as we increase the graph size with the number of cores. We compare all asynchronous approaches and the synchronous approach KaDisReduS-RG. We run these algorithms for 1, 16, 128 and 1 024 cores. The scaling of the throughput is shown in Figure 7.8.

In general, we note that both communication approaches achieve very similar thresholds. Note that they overlap in the graph in the plots. The reason for this observation is that all the considered graphs have a very small cut when they are generated with KaGen.

For rhg graphs, we note that KaDisReduS-RG scales better than KaDisReduA-RG and even better than the greedy algorithms. They can be almost completely reduced. In a more detailed analysis, we were able to explain this as follows: The running time of KaDisReduA-RG strongly depends on the choice of the message buffer threshold δ. For the strong-scaling experiments, we simply chose $\delta = 16\,000$ bytes without making an extra parameter tuning. However, this δ does not scale for this graph in the weak scaling experiment because it is too large since the cut is very small.

For the rgg graphs, we observe that they have a larger throughput than the greedy algorithms although all approaches scale linearly. We conclude our observations in Observation 4.

Observation 4: Weak Scaling Experiments The weak experiments show that our asynchronous approach depends on a good choice of the message buffer queue threshold δ. KaDisReduS-RG yields for our rhg graphs a better throughput than the greedy algorithm does. For our choice of graphs and due to the good input partitioning, communication was only needed seldom. For a better differentiation of both communication approaches we need to consider either other types of graphs or need a worse partitioning in terms of cut edges so that the communication volume is larger.

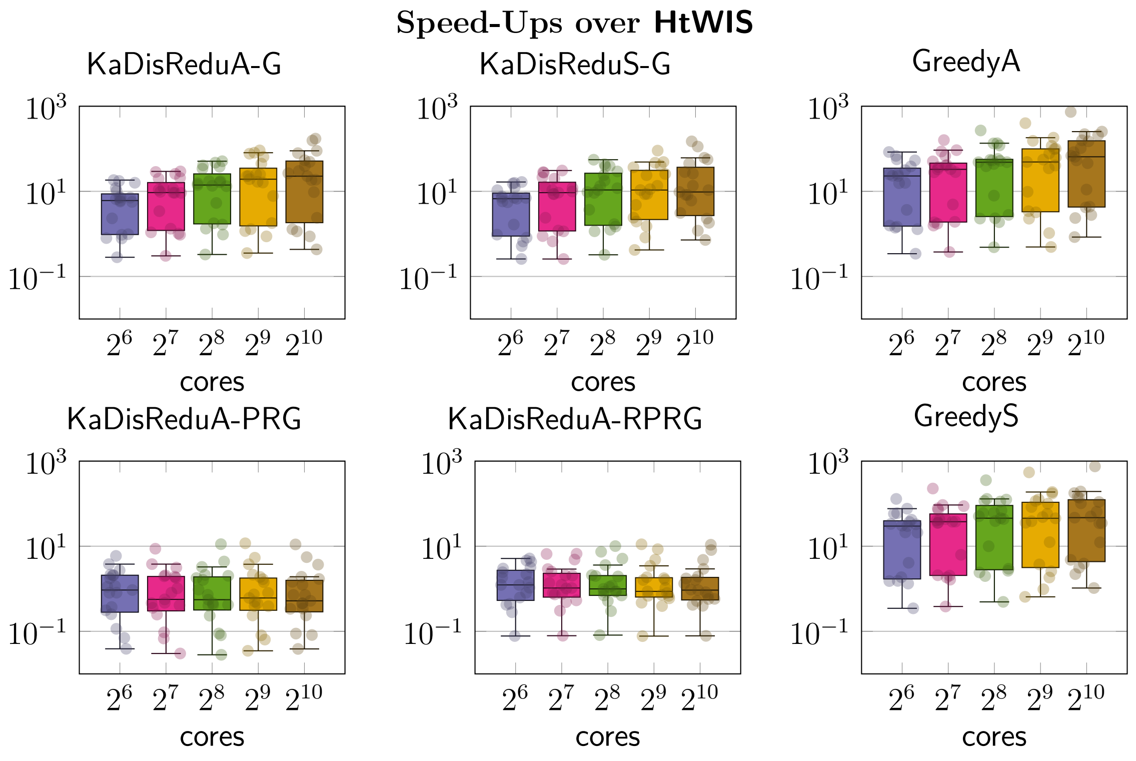

Figure 7.7.: The figure show the speedups of HtWIS for the compared distributed maximal independent set solvers.

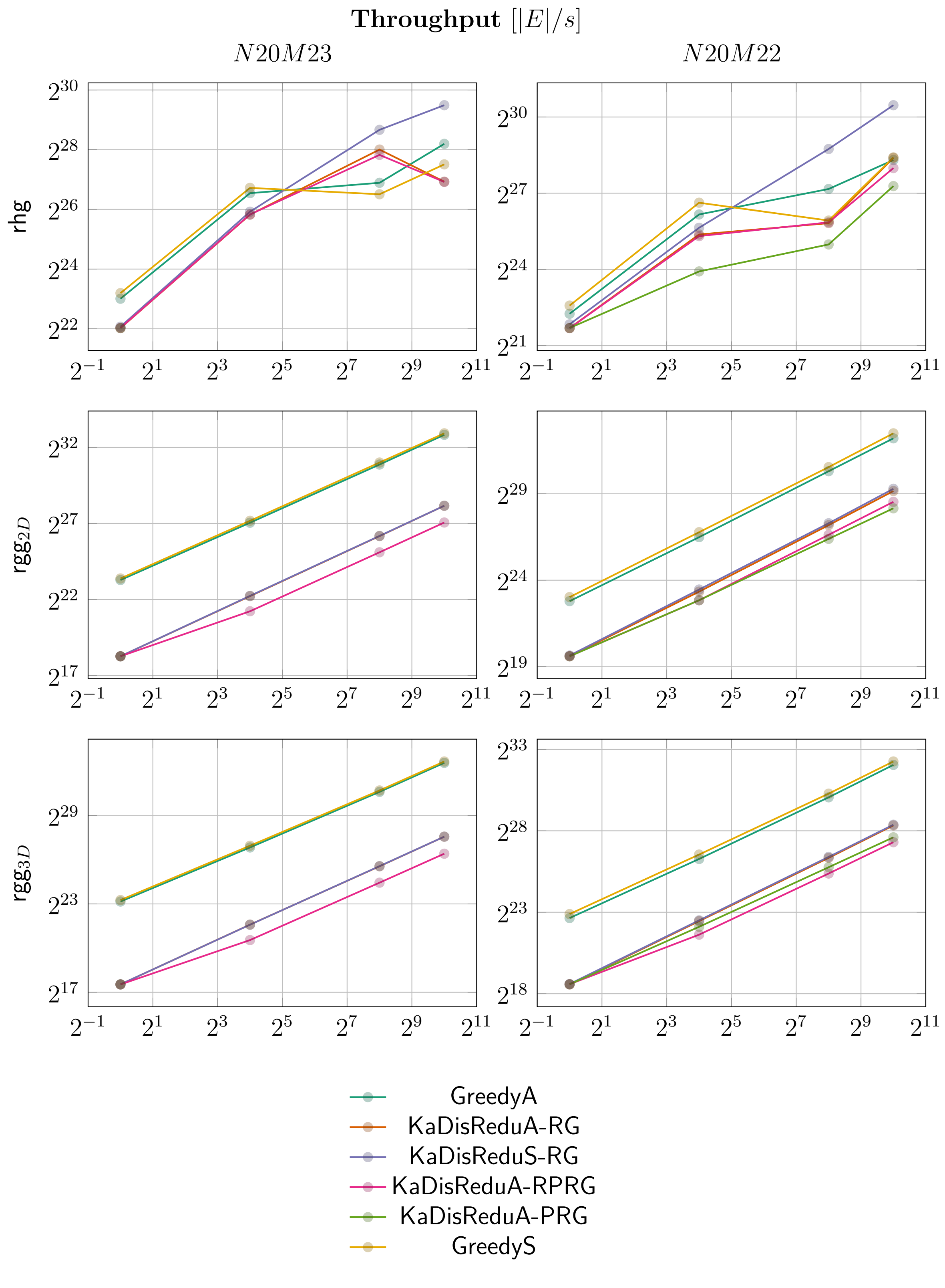

Figure 7.8.: The plots show the throughput of the different distributed independent set solvers for different graphs. Note that the different synchronization approaches often overlap because the are only few cut edges.

	$\omega(\mathcal{I})/\,\omega(\mathcal{I}_{\texttt{best}})$ [%]			t [s]			speed-up	
	cores p			cores p			cores p	
Algorithm	1	64	1 024	1	64	1 024	64	1 024
GreedyS	94.522	94.521	94.521	1.193	0.126	0.052	13.39	32.64
GreedyA	94.522	94.521	94.522	1.691	0.149	0.045	11.32	37.46
KaDisReduS-G	98.168	97.415	96.852	6.340	0.439	0.147	3.85	11.50
KaDisReduA-G	98.168	97.415	96.851	6.950	0.411	0.112	4.11	15.06
KaDisReduA-PRG	98.168	98.066	97.971	6.956	2.221	2.558	0.76	0.66
KaDisReduA-RPRG	98.168	98.097	98.016	6.949	1.366	1.483	1.24	1.14
HtWIS	99.609	-	-	1.689	-	-	-	-

Table 7.3.: This table summarizes the results of our distributed maximal independent set algorithms and HtWIS. We give the geometric mean solution quality $\omega(\mathcal{I})/\,\omega(\mathcal{I}_{\texttt{best}})$, relative to the best solutions found by any solver. Moreover, the geometric mean running time t, and speed-up over HtWIS on all instances of data set Strong are given for $p \in \{1, 64, 1\,024\}$ cores. The detailed results are given in Table A.3 and Table A.4.

8. Discussion

This last chapter conludes our proposed distributed kernelization algorithms and solvers. Moreover, we give an outlook on future work.

8.1. Conclusion

In this work, we transfered well-known exact data reduction rules for the MWISP to the distributed memory model. In the distributed memory model, each PE has its own memory. The graph is partitioned into blocks where each PE stores a block. To apply reduction rules in this representation, the key idea was to consider rules that exploit locallity. Similar approaches exists for the MISP. To the best of our knowledge, we proposed the first distributed data reduction rules for the MWISP. Furthermore, these new rules gave rise to new distributed kernelization algorithms which apply thes rules exhaustively to obtain a quasi-kernel in distributed memory. We designed two different distributed kernelization algorithms, called KaDisReduS and KaDisReduA, which both rely on communication to update the reduction progress at the border and construct a solution. KaDisReduS employs blocking, irregular all-to-all communication to exchange the reduction progress whenever each process applied reductions locally exhaustively. KaDisReduA follows an asynchronous approach by using message buffer queues and provide a parameter δ to control the message sizes. Both support to partitioning the graph before one of the kernelization algorithm (PR) or once in-between (RPR) using dKaMinPar (fast). Furthermore we developed greedy algorithms for both communication schemes. This gave rise to new reduce-and-greedy algorithms KaDisReduA-G and KaDisReduS-G.

Our strong scaling experiments have shown that KaDisReduA outperforms KaDisReduS by 1.42 in terms of the running on artificial and real-world graphs with millions of vertices and edges on 1 024 cores. At this scale, the number of vertices in the quasi-kernel increase from 4.8 % to 24.80 % at median. A good reduction impact is especially maintained for the road networks where KaDisReduA reports relative speed-ups of up to 164. Partitioning the graphs with dKaMinPar, by minimizing cut edges, can mitigate a decreasing impact of our reductions at the border. This sometimes decreases the running times of the reduction phase which is possibly a reason of more local work as more (expensive) reductions become applicable.

Furthermore, we compared our (reduce-)and-greedy independent set solvers with the state-of-the-art reduce-and-peel solver HtWIS in terms of solution quality and running time. Although HtWIS is the overall winner in solution quality, our distributed kernelization algorithms improves the solution quality of the greedy algorithm significantly and is still close to the solution quality of HtWIS. KaDisReduA-G achieves a speed-up of 15.05 on average and 170 on maximum over HtWIS while KaDisReduS-G has a speed-up of 11.5 on average for 1 024. The variants that partition the (reduced) graph only seldomly report a speed-up possibly because partitioning become a significant bottelneck. Nonetheless, KaDisReduA-RPRG maintains the solution quality for larger numbers of cores and is on average slightly faster than KaDisReduA-PRG.

In the weak-scaling experiments we also noted that KaDisReduS can sometimes perform better than KaDisReduA. We suspect that a better fine tuning of the message buffer size threshold δ is needed here. Overall, we noted that KaDisReduS and KaDisReduA have a similar throughput in the weak scaling experiments. This is possibly due to a good input partitioning from the graph generation in KaGen. Nonetheless, the weak scaling experiments show that reduce-and-greedy can be even faster than the plain greedy algorithm for random hyperbolic graphs.

In conclusion, we think that the asynchronous variant KaDisReduA are a good choice to reduce the graph on a distributed memory machine. The reduce-and-greedy algorithm KaDisReduA-G allows to maximal independent sets that significantly improve greedy solutions.

8.2. Future Work

Regarding our kernelization algorithms, there is still potential for improvements.

Data Reduction Rules. In general, our kernelization algorithms can be extended with further data reduction rules. It would also be interesting to transfer more globally acting rules, e.g.,Critical Weight Independent Set rule by Butenko and Trukhanov[7]. However, the rule requires to solve a minimum cut problem for bipartite graph by taking the whole (reduced) graph into acount. We also noted that we often fail to reduce all degree two vertices. To overcome this issue, we need to adapt reduction rules such as Distributed Partial V-Shape to reduce degree-two border vertices. This might work similar for degree-one vertices where we move an interface vertex to another process. One just has to be careful if the adjacent ghost vertices are moved as well.

Inexact Data Reductions. Similar to a reduce-and-peel solver, it would be interesting to support in-exact reductions which peel vertices by forcing them out of a solution. Pelling follows typically a simple greedy heuristic or a more sophisticated approach. This in-exact data reduction can be applied by any process if no reduction progress is made.

More Distributed Independent Set Solvers. In general, our distributed kernelization algorithms can be used with many solver schemes and we hope to see progress here. One idea is to use this in-exact approach for a distributed portfolio local-search solver.

Therefore, we can introduce a threshold for the minimum in-exact reduced graph size. If the in-exact reduced graph size falls below this threshold, we can replicate the in-exact reduced graph at each PE and apply a different local-search algorithm.

Memory. In further experiments, we noted for large web graphs ($\approx$ 30 Mio. vertices and $\approx$ 300 Mio. edges) that we run out of memory for $p < 512$ although we can represent the graph for $p = 1$ on a single compute node. We basically copy the input graph (KaGen) by building the dynamic graph data structure. In addition we insert backward edges from the ghosts to their adjacent interface vertices. Moreover, the global ghost vertex identifier is mapped to a local vertex identifier with a hash-map. Before we initialize the kernelization algorithm, we free the input graph. As far as our analysis has taken us, we fail to build the dynamic graph data structure before the kernelization algorithm can even start. We want to further investigate the memory performance.

Bibliography

[1] "8. Maximal Independent Sets (MIS)". en. In: Society for Industrial and Applied Mathematics, Jan. 2000, pp. 91–102. ISBN: ['9780898714647', '9780898719772']. DOI: 10.1137/1.9780898719772.ch8. URL: https://doi.org/10.1137/1.9780898719772.ch8.

[2] Faisal N. Abu-Khzam et al. "Recent Advances in Practical Data Reduction". en. In: Springer Nature Switzerland, Jan. 2022, pp. 97–133. ISBN: ['9783031215339', '9783031215346']. DOI: 10.1007/978-3-031-21534-6_6. URL: https://doi.org/10.1007/978-3-031-21534-6%5C_6.

[3] Takuya Akiba and Yoichi Iwata. "Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover". en. In: *Theoretical Computer Science* 609 (Jan. 2016), pp. 211–225. ISSN: 03043975. DOI: 10.1016/j.tcs.2015.09.023. URL: https://linkinghub.elsevier.com/retrieve/pii/S030439751500852X (visited on 06/03/2024).

[4] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. "Fast local search for the maximum independent set problem". en. In: *Journal of Heuristics* 18.4 (Aug. 2012), pp. 525–547. ISSN: 1381-1231, 1572-9397. DOI: 10.1007/s10732-012-9196-4. URL: http://link.springer.com/10.1007/s10732-012-9196-4 (visited on 06/03/2024).

[5] Luitpold Babel. "A fast algorithm for the maximum weight clique problem". In: *Computing* 52 (1994), pp. 31–38.

[6] Lukas Barth et al. "Temporal map labeling: a new unified framework with experiments". en. In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. Burlingame California: ACM, Oct. 2016, pp. 1–10. ISBN: 978-1-4503-4589-7. DOI: 10.1145/2996913.2996957. URL: https://dl.acm.org/doi/10.1145/2996913.2996957 (visited on 05/28/2024).

[7] Sergiy Butenko and Svyatoslav Trukhanov. "Using critical sets to solve the maximum independent set problem". en. In: *Operations Research Letters* 35.4 (July 2007), pp. 519–524. ISSN: 01676377. DOI: 10.1016/j.orl.2006.07.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167637706000952 (visited on 06/03/2024).

[8] Sergiy Butenko et al. "Finding maximum independent sets in graphs arising from coding theory". In: *SAC02: 2002 ACM Symposium on Applied Computing* (Madrid Spain). ACM, Mar. 2002, pp. 542–546. DOI: 10.1145/508791.508897. URL: https://doi.org/10.1145/508791.508897.

[9] Jakob Dahlum et al. “Accelerating Local Search for the Maximum Independent Set Problem”. en. In: Springer International Publishing, June 2016, pp. 118–133. ISBN: ['9783319388502', '9783319388519']. DOI: `10.1007/978-3-319-38851-9\_9`. URL: `https://doi.org/10.1007/978-3-319-38851-9%5C_9`.

[10] Yuanyuan Dong et al. *A Metaheuristic Algorithm for Large Maximum Weight Independent Set Problems.* Version Number: 1. 2022. DOI: `10.48550/ARXIV.2203.15805`. URL: `https://arxiv.org/abs/2203.15805` (visited on 06/03/2024).

[11] Yuanyuan Dong et al. “New Instances for Maximum Weight Independent Set From a Vehicle Routing Application”. en. In: *Operations Research Forum* 2.4 (Dec. 2021), p. 48. ISSN: 2662-2556. DOI: `10.1007/s43069-021-00084-x`. URL: `https://link.springer.com/10.1007/s43069-021-00084-x` (visited on 05/28/2024).

[12] R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer New York, 1999. ISBN: ['9781461267980', '9781461205159']. DOI: `10.1007/978-1-4612-0515-9`. URL: `https://doi.org/10.1007/978-1-4612-0515-9`.

[13] Ch. Ebenegger, P.L. Hammer, and D. de Werra. “Pseudo-Boolean Functions and Stability of Graphs”. In: *Algebraic and Combinatorial Methods in Operations Research.* Ed. by R.E. Burkard, R.A. Cuninghame-Green, and U. Zimmermann. Vol. 95. North-Holland Mathematics Studies. North-Holland, 1984, pp. 83–97. DOI: `https://doi.org/10.1016/S0304-0208(08)72955-4`. URL: `https://www.sciencedirect.com/science/article/pii/S0304020808729554`.

[14] Aleksander Figiel et al. “There and Back Again: On Applying Data Reduction Rules by Undoing Others”. In: *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany.* Ed. by Shiri Chechik et al. Vol. 244. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 53:1–53:15. DOI: `10.4230/LIPICS.ESA.2022.53`. URL: `https://doi.org/10.4230/LIPIcs.ESA.2022.53`.

[15] Daniel Funke et al. “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.

[16] M. R. Garey, D. S. Johnson, and L. Stockmeyer. “Some simplified NP-complete problems”. en. In: *Proceedings of the sixth annual ACM symposium on Theory of computing - STOC '74.* Seattle, Washington, United States: ACM Press, 1974, pp. 47–63. DOI: `10.1145/800119.803884`. URL: `http://portal.acm.org/citation.cfm?doid=800119.803884` (visited on 05/28/2024).

[17] Alexander Gellner et al. “Boosting Data Reduction for the Maximum Weight Independent Set Problem Using Increasing Transformations”. In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021.* Ed. by Martin Farach-Colton and Sabine Storandt. SIAM, 2021, pp. 128–142. DOI: `10.1137/1.9781611976472.10`. URL: `https://doi.org/10.1137/1.9781611976472.10`.

[18] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. "Evaluation of Labeling Strategies for Rotating Maps". In: *Experimental Algorithms*. Ed. by David Hutchison et al. Vol. 8504. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 235–246. ISBN: 978-3-319-07958-5 978-3-319-07959-2. DOI: 10.1007/978-3-319-07959-2_20. URL: http://link.springer.com/10.1007/978-3-319-07959-2%5C_20 (visited on 05/28/2024).

[19] Tom George and Demian Hespe. *Distributed Kernelization for Independent Sets*. Nov. 2018.

[20] Ernestine Großmann, Kenneth Langedal, and Christian Schulz. "A Comprehensive Survey of Data Reduction Rules for the Maximum Weighted Independent Set Problem". In: (Dec. 2024). arXiv: 2412.09303v1 [cs.DS]. URL: http://arxiv.org/abs/2412.09303v1.

[21] Ernestine Großmann, Kenneth Langedal, and Christian Schulz. "Accelerating Reductions Using Graph Neural Networks and a New Concurrent Local Search for the Maximum Weight Independent Set Problem". In: (Dec. 2024). arXiv: 2412.14198v1 [math.OC]. URL: http://arxiv.org/abs/2412.14198v1.

[22] Ernestine Großmann et al. "Finding Near-Optimal Weight Independent Sets at Scale". en. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Lisbon Portugal: ACM, July 2023, pp. 293–302. ISBN: 9798400701191. DOI: 10.1145/3583131.3590353. URL: https://dl.acm.org/doi/10.1145/3583131.3590353 (visited on 05/28/2024).

[23] Jiewei Gu et al. "Towards Computing a Near-Maximum Weighted Independent Set on Massive Graphs". en. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery
Data Mining*. Virtual Event Singapore: ACM, Aug. 2021, pp. 467–477. ISBN: 978-1-4503-8332-5. DOI: 10.1145/3447548.3467232. URL: https://dl.acm.org/doi/10.1145/3447548.3467232 (visited on 06/03/2024).

[24] Demian Hespe, Sebastian Lamm, and Christian Schorr. "Targeted Branching for the Maximum Independent Set Problem". In: *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*. Ed. by David Coudert and Emanuele Natale. Vol. 190. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 17:1–17:21. DOI: 10.4230/LIPICS.SEA.2021.17. URL: https://doi.org/10.4230/LIPIcs.SEA.2021.17.

[25] Demian Hespe, Christian Schulz, and Darren Strash. "Scalable Kernelization for Maximum Independent Sets". en. In: *ACM Journal of Experimental Algorithmics* 24 (Dec. 2019), pp. 1–22. ISSN: 1084-6654, 1084-6654. DOI: 10.1145/3355502. URL: https://dl.acm.org/doi/10.1145/3355502 (visited on 05/15/2024).

[26] Demian Hespe et al. *KaMPIng: Flexible and (Near) Zero-overhead C++ Bindings for MPI*. 2024. arXiv: 2404.05610 [cs.DC].

[27] Changhee Joo et al. “Distributed Greedy Approximation to Maximum Weighted Independent Set for Scheduling With Fading Channels”. In: *IEEE/ACM Transactions on Networking* 24 (3 June 2016), pp. 1476–1488. DOI: 10.1109/tnet.2015.2417861. URL: https://doi.org/10.1109/tnet.2015.2417861.

[28] Sebastian Lamm, Peter Sanders, and Christian Schulz. “Graph Partitioning for Independent Sets”. In: Springer International Publishing, June 2015, pp. 68–81. ISBN: [’9783319200859’, ’9783319200866’]. DOI: 10.1007/978-3-319-20086-6_6. URL: https://doi.org/10.1007/978-3-319-20086-6%5C_6.

[29] Sebastian Lamm et al. “Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs”. In: *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019.* Ed. by Stephen G. Kobourov and Henning Meyerhenke. SIAM, 2019, pp. 144–158. DOI: 10.1137/1.9781611975499.12. URL: https://doi.org/10.1137/1.9781611975499.12.

[30] Sebastian Lamm et al. “Finding near-optimal independent sets at scale”. en. In: *Journal of Heuristics* 23.4 (Aug. 2017), pp. 207–229. ISSN: 1381-1231, 1572-9397. DOI: 10.1007/s10732-017-9337-x. URL: http://link.springer.com/10.1007/s10732-017-9337-x (visited on 06/03/2024).

[31] Jianfeng Liu, Sihong Shao, and Chaorui Zhang. “Application of Causal Inference Techniques to the Maximum Weight Independent Set Problem”. In: (Jan. 2023). arXiv: 2301.05510v1 [math.OC]. URL: http://arxiv.org/abs/2301.05510v1.

[32] M Luby. “A simple parallel algorithm for the maximal independent set problem”. en. In: *Proceedings of the seventeenth annual ACM symposium on Theory of computing - STOC ’85.* Providence, Rhode Island, United States: ACM Press, 1985, pp. 1–10. ISBN: 978-0-89791-151-1. DOI: 10.1145/22145.22146. URL: http://portal.acm.org/citation.cfm?doid=22145.22146 (visited on 05/20/2024).

[33] Henning Meyerhenke, Peter Sanders, and Christian Schulz. “Parallel Graph Partitioning for Complex Networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.9 (2017), pp. 2625–2638. DOI: 10.1109/TPDS.2017.2671868.

[34] Bruno Nogueira, Rian G. S. Pinheiro, and Anand Subramanian. “A hybrid iterated local search heuristic for the maximum weight independent set problem”. en. In: *Optimization Letters* 12.3 (May 2018), pp. 567–583. ISSN: 1862-4472, 1862-4480. DOI: 10.1007/s11590-017-1128-7. URL: http://link.springer.com/10.1007/s11590-017-1128-7 (visited on 06/03/2024).

[35] Peter Sanders and Daniel Seemaier. “Distributed Deep Multilevel Graph Partitioning”. en. In: *Euro-Par 2023: Parallel Processing.* Ed. by José Cano et al. Vol. 14100. Series Title: Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 443–457. ISBN: 978-3-031-39697-7 978-3-031-39698-4. DOI: 10.1007/978-3-031-39698-4_30. URL: https://link.springer.com/10.1007/978-3-031-39698-4%5C_30 (visited on 05/21/2024).

[36] Peter Sanders and Tim Niklas Uhl. “Engineering a Distributed-Memory Triangle Counting Algorithm”. In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (St. Petersburg, FL, USA). IEEE, May 2023, pp. 702–712. DOI: 10.1109/ipdps54959.2023.00076. URL: https://doi.org/10.1109/ipdps54959.2023.00076.

[37] Peter Sanders et al. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox.* Springer, 2019, pp. 265–267. ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0. URL: https://doi.org/10.1007/978-3-030-25209-0.

[38] Xubo Wang et al. “Distributed Near-Maximum Independent Set Maintenance over Large-scale Dynamic Graphs”. In: *2023 IEEE 39th International Conference on Data Engineering (ICDE).* Anaheim, CA, USA: IEEE, Apr. 2023, pp. 2538–2550. ISBN: 9798350322279. DOI: 10.1109/ICDE55515.2023.00195. URL: https://ieeexplore.ieee.org/document/10184836/ (visited on 05/15/2024).

[39] Jeffrey S Warren and Illya V Hicks. “Combinatorial branch-and-bound for the maximum weight independent set problem. 2006”. In: *URL https://www. caam. rice. edu/∼ ivhicks/jeff.rev.pdf* (2006).

[40] Mingyu Xiao, Sen Huang, and Xiaoyu Chen. “Maximum Weighted Independent Set: Effective Reductions and Fast Algorithms on Sparse Graphs”. en. In: *Algorithmica* 86 (5 May 2024), pp. 1293–1334. DOI: 10.1007/s00453-023-01197-x. URL: https://doi.org/10.1007/s00453-023-01197-x.

[41] Mingyu Xiao et al. “Efficient Reductions and a Fast Algorithm of Maximum Weighted Independent Set”. en. In: *Proceedings of the Web Conference 2021.* Ljubljana Slovenia: ACM, Apr. 2021, pp. 3930–3940. ISBN: 978-1-4503-8312-7. DOI: 10.1145/3442381.3450130. URL: https://dl.acm.org/doi/10.1145/3442381.3450130 (visited on 11/02/2024).

A. Appendix

Graph	Original Name	$\lvert V \rvert$	$\lvert E \rvert$	avg. deg.
	`kagen`			
grid2d-XY12	kagen-grid2d-X12-Y12-p0.9-s12-wuniform_random-wmin1-wmax100	16 777 216.0	30 191 845.0	3.6
grid3d-XYZ7	kagen-grid3d-X7-Y7-Z7-p0.8-s20-wuniform_random-wmin1-wmax100	2 097 152.0	4 992 844.0	4.8
rgg2d-N20M22	kagen-rgg2d-N20-M22-s5-wuniform_random-wmin1-wmax50	1 048 576.0	4 195 172.0	8.0
rgg3d-N20M22	kagen-rgg3d-N20-M22-s7-wuniform_random-wmin1-wmax100	1 048 576.0	4 191 989.0	8.0
rhg-N20	kagen-rhg-N20-d10-g2.6-s25-wuniform_random-wmin1-wmax100	1 048 576.0	5 030 376.0	9.6
rhg-N22	kagen-rhg-N22-d5-g2.9-s29-wuniform_random-wmin1-wmax100	4 194 304.0	10 350 108.0	4.9
	`mesh`			
buddha	mesh_buddha-uniform	1 087 716.0	1 631 574.0	3.0
	`osm`			
asia	uniform-asia.osm	11 950 757.0	12 711 603.0	2.1
europe	uniform-europe.osm	50 912 018.0	54 054 660.0	2.1
	`snap`			
LiveJour.	snap_soc-LiveJournal1-uniform	4 847 571.0	42 851 237.0	17.7
as-skitter	snap_as-skitter-uniform	1 696 415.0	11 095 298.0	13.1
com-youtube	snap_com-youtube	1 134 890.0	2 987 624.0	5.3
pokec-rel.	snap_soc-pokec-relationships-uniform	1 632 803.0	22 301 964.0	27.3
roadNet-CA	snap_roadNet-CA-uniform	1 965 206.0	2 766 607.0	2.8
roadNet-PA	snap_roadNet-PA	1 088 092.0	1 541 898.0	2.8
roadNet-PA	snap_roadNet-PA-uniform	1 088 092.0	1 541 898.0	2.8
roadNet-TX	snap_roadNet-TX-uniform	1 379 917.0	1 921 660.0	2.8
wiki-Talk	snap_wiki-Talk-uniform	2 394 385.0	4 659 565.0	3.9

Table A.1.: The table lists the graphs of data set Strong (18 graphs) with additional meta information.

		KaDisReduS						KaDisReduA					
		median n [%]			geo. mean t [s]			median n [%]			geo. mean t [s]		
		cores p			cores p			cores p			cores p		
	Instance	1	64	1 024	1	64	1 024	1	64	1 024	1	64	1 024
kagen	grid2d-X12-Y12-p0.9-s12-uf100	**51.56**	**53.18**	**74.59**	**76.23**	**1.28**	0.07	**51.56**	**53.18**	**74.59**	76.87	**1.28**	**0.06**
kagen	grid3d-X7-Y7-Z7-p0.8-s20-uf100	**91.49**	**95.09**	**95.29**	14.35	0.12	0.03	**91.49**	**95.09**	**95.29**	**14.19**	**0.11**	**0.01**
kagen	rgg2d-N20-M22-s5-uf50	**18.66**	27.04	**63.00**	**4.80**	**0.09**	0.03	**18.66**	**27.02**	63.11	4.94	**0.09**	**0.01**
kagen	rgg3d-N20-M22-s7-uf100	**58.09**	76.66	**80.09**	**10.13**	0.09	0.02	**58.09**	**76.60**	80.13	10.35	**0.08**	**0.01**
kagen	rhg-N20-d10-g2.6-s25-uf100	**0.00**	**47.16**	**47.81**	**1.08**	0.77	**0.29**	**0.00**	**47.16**	47.82	1.18	**0.62**	0.44
kagen	rhg-N22-d5-g2.9-s29-uf100	**0.00**	28.02	**28.19**	**2.99**	1.63	**0.45**	**0.00**	**27.97**	28.20	3.52	**1.27**	0.61
mesh	buddha-uf	**12.21**	**40.66**	67.29	**2.08**	**0.06**	0.03	**12.21**	**40.66**	**67.28**	2.14	**0.06**	**0.01**
osm	uf-asia	**0.40**	**2.08**	**2.24**	**7.61**	0.32	0.07	**0.40**	**2.08**	**2.24**	9.00	**0.28**	**0.04**
osm	uf-europe	**0.12**	1.32	**1.45**	**33.17**	1.45	0.15	**0.12**	**1.31**	**1.45**	39.28	**1.06**	**0.13**
snap	as-skitter-uf	**4.40**	**20.37**	**21.41**	**18.00**	2.39	**0.64**	**4.40**	20.38	21.42	18.35	**1.47**	0.65
snap	com-youtube	**0.00**	**0.55**	**0.60**	**0.69**	0.35	**0.13**	**0.00**	0.56	0.61	0.98	**0.31**	0.40
snap	roadNet-CA-uf	**4.88**	9.20	**16.16**	**2.10**	0.10	0.04	**4.88**	**9.17**	16.17	2.30	**0.09**	**0.01**
snap	roadNet-PA	**5.26**	8.72	**17.55**	**1.16**	**0.06**	0.03	**5.26**	**8.71**	**17.55**	1.28	**0.06**	**0.01**
snap	roadNet-PA-uf	**5.36**	8.82	**17.48**	**1.16**	0.07	0.03	**5.36**	**8.80**	**17.48**	1.27	**0.06**	**0.01**
snap	roadNet-TX-uf	**4.72**	**7.65**	**14.74**	**1.39**	**0.06**	0.04	**4.72**	**7.65**	14.75	1.53	**0.06**	**0.01**
snap	soc-LiveJournal1-uf	**0.98**	**28.20**	**29.03**	**41.58**	5.47	1.52	**0.98**	28.23	29.08	42.29	**3.47**	**0.82**
snap	soc-pokec-relationships-uf	**53.07**	**69.29**	**69.50**	**84.81**	1.70	0.17	**53.07**	69.30	**69.50**	85.09	**1.34**	**0.14**
snap	wiki-Talk-uf	**0.00**	**0.01**	**0.01**	**1.22**	1.91	**0.68**	**0.00**	**0.01**	**0.01**	2.05	**1.73**	1.17

Table A.2.: Detailed overview of the reduction impact and the required running time by KaDisReduS and KaDisReduA to reduce the graphs of data set Strong (18 graphs). It shows the median number of vertices of the kernel, relative to the number of vertices in the input graph, and the geometric mean running time for 1, 64, and 1 024 processes.

		HtWIS		KaDisReduA-RG			PR	RPR	KaDisReduS-RG			GreedyA			GreedyS		
	Graph	$\omega(\mathcal{I})$	t [s]	$\omega(\mathcal{I})$	t [s]	su.	su.	su.	$\omega(\mathcal{I})$	t [s]	su.	$\omega(\mathcal{I})$	t [s]	su.	$\omega(\mathcal{I})$	t [s]	su.
`kagen`	grid2d-XY12	**444 367 271**	15.07	426 374 036	84.89	173.11	11.73	11.10	426 374 036	83.09	149.38	410 832 482	7.01	733.96	410 832 482	4.53	752.36
`kagen`	grid3d-XYZ7	**51 397 333**	2.51	46 399 972	15.39	89.61	2.71	2.45	46 399 972	15.30	55.92	46 052 941	1.03	253.05	46 052 941	0.75	193.49
`kagen`	rgg2d-N20M22	**8 289 274**	1.31	8 102 402	5.34	53.65	2.09	2.99	8 102 402	5.18	29.34	7 513 200	0.61	149.91	7 513 200	0.53	112.23
`kagen`	rgg3d-N20M22	**17 711 545**	1.95	16 532 699	10.97	78.99	2.08	2.20	16 532 699	10.69	45.13	15 968 885	0.67	208.32	15 968 885	0.57	144.60
`kagen`	rhg-N20	23 903 728	0.61	**24 123 473**	1.39	1.33	0.25	0.52	**24 123 473**	1.30	1.82	22 917 135	0.71	2.78	22 917 135	0.56	3.13
`kagen`	rhg-N22	104 705 319	1.58	**111 332 870**	4.27	2.39	0.46	0.93	**111 332 870**	3.78	3.11	108 285 899	2.27	4.37	108 285 899	1.57	4.79
`mesh`	buddha	**57 508 556**	0.61	57 175 207	2.54	26.65	1.11	1.06	57 175 207	2.46	11.84	54 724 471	0.47	68.56	54 724 471	0.32	46.89
`osm`	asia	**177 944 319**	2.63	177 909 022	10.96	39.45	2.15	3.61	177 909 022	9.61	28.39	172 407 097	5.12	103.36	172 407 097	2.99	109.46
`osm`	europe	**759 787 635**	12.15	759 755 354	48.33	49.35	5.54	8.51	759 755 354	42.56	61.29	736 379 981	22.92	158.07	736 379 981	13.57	177.35
`snap`	as-skitter	**124 141 373**	1.29	123 895 991	18.78	1.66	0.08	0.08	123 895 991	18.43	1.25	118 077 022	3.07	2.79	118 077 022	2.57	2.96
`snap`	com-youtube	**90 295 285**	0.37	**90 295 285**	1.15	1.16	0.11	0.83	**90 295 285**	0.87	2.29	88 274 720	0.96	4.19	88 274 720	0.62	3.92
`snap`	roadNet-CA	**111 325 524**	0.80	111 040 444	2.79	30.02	1.16	1.84	111 040 444	2.58	13.54	106 123 348	0.95	83.40	106 123 348	0.62	64.82
`snap`	roadNet-PA	**61 688 549**	0.43	61 510 241	1.54	18.36	0.81	1.23	61 510 241	1.42	9.19	58 767 687	0.53	45.85	58 767 687	0.35	39.00
`snap`	roadNet-PA-uf	**61 710 606**	0.43	61 535 801	1.53	18.73	0.80	1.23	61 535 801	1.42	9.65	58 828 685	0.53	53.41	58 828 685	0.34	43.40
`snap`	roadNet-TX	**78 575 460**	0.54	78 372 685	1.86	22.73	0.94	1.43	78 372 685	1.71	11.23	74 875 147	0.66	64.86	74 875 147	0.43	50.61
`snap`	LiveJour.	**283 922 214**	9.81	283 855 645	44.25	9.58	0.74	1.95	283 855 645	43.54	6.07	269 452 344	9.38	11.00	269 452 344	8.30	12.46
`snap`	pokec-rel.	**83 920 370**	33.86	77 595 685	88.71	156.11	3.80	5.11	77 595 685	88.28	113.08	74 968 957	5.09	220.85	74 968 957	4.76	187.88
`snap`	wiki-Talk	**235 837 346**	0.54	**235 837 346**	2.27	0.43	0.07	0.40	**235 837 346**	1.44	0.72	235 317 688	1.94	0.84	235 317 688	1.05	1.04

Table A.3.: Detailed results of the compared (distributed) independent set solvers for data set Strong (18 graphs). PR and RPR refer to KaDisReduA-PRG and KaDisReduA-PRG, respectively. We show for each algorithm the geometric mean solution and running time for $p = 1$. PR and RPR are identical for $p = 1$, therfore only the speed-up is shown. Moreover, we show the maximum speed-up (su.) over HtWIS for each algorithm over all $p \in \{2, 64, 128, 256, 1\,024\}$. Note that a speed-up less than one, indicates that there is no speed-up at all.

		HtWIS		KaDisReduA-RG			KaDisReduA-PRG			KaDisReduA-RPRG			KaDisReduS-RG		
	Graph	$\omega(\mathcal{I})$	t $[s]$	$\omega(\mathcal{I})$	t $[s]$	su.	$\omega(\mathcal{I})$	t $[s]$	su.	$\omega(\mathcal{I})$	t $[s]$	su.	$\omega(\mathcal{I})$	t $[s]$	su.
`kagen`	grid2d-XY12	**444 367 271**	15.07	417 012 867	0.09	173.11	425 681 071	1.37	11.03	426 013 133	1.39	10.83	417 011 461	0.10	149.38
`kagen`	grid3d-XYZ7	**51 397 333**	2.51	46 145 432	0.03	89.61	46 273 128	1.48	1.70	46 265 332	1.43	1.75	46 145 685	0.04	55.92
`kagen`	rgg2d-N20M22	**8 289 274**	1.31	7 683 781	0.02	53.65	8 070 352	1.05	1.25	8 082 363	0.99	1.33	7 684 023	0.05	26.11
`kagen`	rgg3d-N20M22	**17 711 545**	1.95	16 163 148	0.02	78.99	16 422 714	1.34	1.46	16 434 291	1.21	1.61	16 163 286	0.04	45.13
`kagen`	rhg-N20	23 903 728	0.61	23 486 511	0.47	1.29	24 123 459	2.57	0.24	**24 123 473**	1.45	0.42	23 486 120	0.33	1.82
`kagen`	rhg-N22	104 705 319	1.58	110 314 759	0.66	2.39	**111 332 870**	3.46	0.46	**111 332 870**	1.70	0.93	110 314 131	0.51	3.11
`mesh`	buddha	**57 508 556**	0.61	55 239 913	0.02	26.65	56 975 075	1.35	0.45	57 096 043	1.18	0.52	55 239 248	0.06	10.67
`osm`	asia	**177 944 319**	2.63	177 727 363	0.07	39.45	177 908 879	1.37	1.92	177 908 506	0.90	2.93	177 727 418	0.09	28.39
`osm`	europe	**759 787 635**	12.15	759 238 497	0.25	49.35	759 755 275	2.19	5.54	759 755 474	1.49	8.14	759 238 380	0.20	61.29
`snap`	as-skitter	**124 141 373**	1.29	123 082 249	1.13	1.14	123 748 677	15.71	0.08	123 810 357	16.45	0.08	123 082 534	1.08	1.19
`snap`	com-youtube	**90 295 285**	0.37	90 289 892	0.43	0.87	90 295 004	4.21	0.09	**90 295 285**	0.45	0.83	90 290 049	0.16	2.29
`snap`	roadNet-CA	**111 325 524**	0.80	110 309 636	0.03	30.02	110 989 853	1.12	0.72	111 032 253	0.84	0.95	110 309 780	0.06	13.54
`snap`	roadNet-PA	**61 688 549**	0.43	61 080 405	0.02	18.36	61 473 413	1.28	0.34	61 496 654	0.75	0.58	61 081 324	0.05	8.17
`snap`	roadNet-PA-uf	**61 710 606**	0.43	61 099 308	0.02	18.73	61 499 643	1.26	0.34	61 524 404	0.75	0.58	61 099 698	0.05	8.32
`snap`	roadNet-TX	**78 575 460**	0.54	77 914 830	0.02	22.73	78 340 497	1.03	0.52	78 360 447	0.77	0.70	77 915 294	0.06	9.66
`snap`	LiveJour.	**283 922 214**	9.81	278 565 085	1.02	9.58	282 722 066	13.66	0.72	283 018 326	5.02	1.95	278 574 015	1.62	6.07
`snap`	pokec-rel.	**83 920 370**	33.86	76 397 775	0.22	156.11	76 784 742	9.16	3.70	76 820 585	7.09	4.78	76 398 134	0.30	113.08
`snap`	wiki-Talk	**235 837 346**	0.54	235 837 085	1.25	0.43	**235 837 346**	14.00	0.04	**235 837 346**	1.36	0.40	235 837 115	0.75	0.72

Table A.4.: Detailed results of the compared (distributed) independent set solvers which use kernelization for data set Strong (18 graphs) for $p = 1024$. We show for each algorithm the geometric mean solution, running time and speed-up (su.) over HtWIS. Note that a speed-up less than one, indicates that there is no speed-up at all.