A Massively Parallel Algorithm for Minimum Weight Vertex Cover

Master Thesis

Daan Nilis

September 30, 2019

Advisor: Prof. Dr. M. Ghaffari
Department of Computer Science, ETH Zürich
Abstract

In this thesis we study the problem of finding a minimum weight vertex cover of an $n$-vertex graph in the Massively Parallel Computation (MPC) model with $\tilde{O}(n)$ memory per machine. We present an algorithm that returns a $(2 + \varepsilon)$-approximation in $O(\log \log d)$ rounds, where $d$ is the average degree of the input graph. This result fills a gap in the state-of-the-art algorithms for central symmetry breaking problems in MPC. Czumaj et al. [8] showed the first sub-logarithmic time algorithm for approximate maximum matching. Since then, there are algorithms for $(1 + \varepsilon)$-approximate maximum matching, $(2 + \varepsilon)$-approximate minimum vertex cover and maximal independent set that all run in $O(\log \log n)$ rounds [12]. The result of this thesis shows that also the weighted minimum vertex cover can be solved in the same running time.
Acknowledgements

First and foremost I want to thank my advisor Prof. Mohsen Ghaffari. Not only because most of the ideas in this thesis come from the discussions we had, but also for the guidance and answers to all my other questions. It was an incredible privilege to work with such a driven and skilled researcher. Thank you!

Secondly, I want to thank all the friends and colleagues that provided insights into the problem, proofread the drafts, or even just made the time to have a completely off-topic but nonetheless motivating discussion! Thank you Varun, Nuriya, Christoff, Zeyong, Giorgio, Roko, Kevin, Edwin and so many others. Finally I would like to thank my parents for providing me with the opportunity to study abroad, for their continuous support in this amazing adventure and for always believing in a positive result.
# Contents

Acknowledgements iii

Contents v

1 Introduction 1
  1.1 Our Results in a Nutshell 2

2 The MPC Model 3
  2.1 Background 3
  2.2 Model Definition 5
  2.3 State-of-the-art 6
  2.4 Notation and Preliminaries 7

3 Minimum Weight Vertex Cover 9
  3.1 Problem description 9
  3.2 Background 10
  3.3 State-of-the-art 12

4 The Algorithm 15
  4.1 $O(\log \Delta)$ Centralized Algorithm 15
  4.2 Round Compression 18
     4.2.1 Why not run even more iterations in one phase? 19
     4.2.2 Round compression requirements 19
  4.3 MPC Simulation 21

5 Analysis 25
  5.1 Degree Properties 25
  5.2 Memory Constraint 26
  5.3 Round Complexity 28
  5.4 Approximation Ratio 29
6 Conclusion and Outlook 37
Bibliography 39
Recent years have seen an impressive increase in computational problems on massive data sets. Image recognition, DNA analysis, financial market forecasting are just a few examples of the application of big data techniques on increasingly larger data sets. These new problem sizes demand also more powerful machines to avoid longer running times. However due to physical limitations, there is a stagnation of the progress made in integrated circuit design. This means that it is not possible anymore to rely on the chip sets to become ever more powerful as the problem sizes grow. One way of tackling this new challenge is by the use of a cluster of machines. In fact this is what is already being done for certain problems in practice. Popular platforms that provide such functionality include MapReduce [9], Hadoop [30], Dryad [18] and Spark [32]. The idea behind these platforms is to have a cluster of powerful machines, each with some local memory, that are all connected via a communication channel. Then the input is divided among the machines and a parallel algorithm is run on the cluster to compute the solution.

After their initial success, the question arose: 'how fast can these platforms be?'. In order to answer such a question, there are two necessary ingredients. First of all it is necessary to establish a common understanding of how to quantify the performance of different algorithms. This is achieved by defining a suitable model that captures the type of computation. The earliest mentions of a specific model to describe these computational platforms appear in [19, 11]. In consequent years this model evolved thanks to the work of several authors [14, 5, 1]. For the first time the abbreviation MPC was used, which stands in those works for ”Massively Parallel Communication”.

Secondly, it would be overambitious to try to answer the question of efficiency immediately for all computational problems. Therefore it is important to select a class of problems to study. A mainstay of problems in the realm of distributed computing has been the LP-type problems that arise...
1. Introduction

for example from graph optimization problems, such as maximum matching, maximal independent set, minimum vertex cover, maximum flow etc. Moreover, these problems are becoming more relevant due to practical applicability as well. Often large data is presented as graphs and hence graph theory techniques are necessary to analyze the data.

The problem we will study in this thesis is that of finding a minimum weight vertex cover (MWVC). We are given a graph as input with a weight function that assigns positive weights to the vertices. The objective is to find a subset of the vertices with minimal weight such that every edge in the graph has at least one endpoint in this set.

In the next chapter we will start out with introducing the MPC model and compare it with other models for parallel computation. Also we discuss the state-of-the-art for closely related problems such as (unweighted) vertex cover and maximum matching. Then in the next chapter follows a discussion of the minimum weight vertex cover and its history. To get a feeling for the problem we analyze a very simple centralized algorithm. After these chapters the stage is set to introduce our algorithm in chapter 4. The main technique we use is round compression, where many rounds of a centralized algorithm are compressed into one MPC round. This approach requires some care however, and to make that clear, we list some necessary conditions and show how to handle these one by one. Then in chapter 5, we carefully analyse the memory requirements, round complexity and approximation guarantees of our proposed algorithm. In the final chapter we briefly recapitalize and suggest a possible direction for future work. The results are summarized next.

1.1 Our Results in a Nutshell

Given a graph $G$ on $n$ vertices with a weight assigned to each vertex, we present an algorithm that computes a $(2 + \mathcal{O}(\epsilon))$-approximate minimum weight vertex cover in $\mathcal{O}(\log \log d)$ MPC rounds, where $d$ is the average degree of the graph. Moreover, this algorithm uses $\mathcal{O}(n)$ memory per machine. As we will see later more formally, this result carries over immediately to the congested clique model with a result from [6].
In this chapter we first discuss the origin of the Massively Parallel Computation (MPC) model and contrast it with other distributed computing models. Subsequently we precisely define the model that is used in this thesis and discuss the state-of-the-art in algorithms design for MPC. The chapter ends with a section on preliminary results that are used throughout the thesis.

2.1 Background

Since Gordon Moore postulated his famous prediction in 1965 that the number of components per integrated circuit would roughly double every year \[26\] (later revised to double every two years), the industry has been closely following this de facto law and even using it as a guideline for future releases of hardware. However, this progression has slowed down recently due to fundamental physical limitations on the size of these components. Therefore to be able to further increase computing power and speed up large scale computations, a natural step is to consider distributed systems. The idea of parallelization is not a very new one. Throughout the history of computing, several models have been proposed and studied. In the following we discuss some prominent examples of distributed computing models and argue about their relevance with respect to the Massively Parallel Computation model that we use in this thesis.

One of the oldest and most studied models for parallel computation is the Parallel Random Access Machine (PRAM) \[31\]. In this model there are a number of processors each with some local registers. All the processors have access to a global memory. Computation proceeds in synchronized rounds, where in one round each processor can do one of the following: either load from a global memory cell into a local register, or execute a basic operation on the data in its registers or write in a global memory cell. By a basic operation we mean for example computing the sum of two numbers,
a comparison of two numbers, etc. There is a differentiation on how concurrency is handled with respect to reading and writing of different processors to the same global memory cell in one round. Four variants are distinguished: exclusive read exclusive write (EREW), exclusive read concurrent write (ERCW), concurrent read exclusive write (CREW) and concurrent read concurrent write (CRCW). Two common measures of efficiency in the PRAM model are the number of synchronized rounds and the total number of processors used. From the very start [31] a polynomial number of processor units with respect to the input size and a poly-logarithmic number of rounds are considered efficient. On the other hand, sharing or passing of data from one processor to another is abstracted away and is almost trivial through the use of the global memory. It is exactly this assumption of a shared memory that is not adequate to capture the kind of computation in the MPC setting, where the machines have only local memory and have to explicitly send data over a communication channel.

In the early 90s, Linial [24] introduced a new kind of distributed computing model: the message passing model. In this model, the computation is executed on a graph with a processor on each node and nodes can communicate over the edges. Computation proceeds in synchronized rounds, where each node can execute some computation on its local data and send a message to every neighbor. The local memory and computing power of a processor are considered unbounded. There are two flavours with regards to the size of the messages that can be sent over edges: in the LOCAL model, messages can be of unbounded size, in the CONGEST model, messages are restricted to be of size $O(\log n)$ where $n$ is the number of processors/nodes in the graph. This model is mostly used for the analysis of graph problems such as calculating a minimum s-t cut, finding a valid vertex coloring of the nodes, etc where the network graph is equal to the input graph. Then at the start of an algorithm each processor has an ID and only knows the ID’s of its neighbors. One key aspect of this model is that it captures the locality of a graph problem. Namely consider a specific node $v$ and $T$ rounds of computation/communication; $v$ can only receive messages from nodes that are at most $T$ hops away in the graph. Therefore finding an algorithm to solve a problem in $T$ rounds is equivalent to finding a function that calculates the solution for a node given its $T$-hop neighborhood. The study of such locality turns out to be very valuable in designing fast MPC algorithms. However, this message passing model also does not capture MPC completely.

Next, the Bulk Synchronous Parallel (BSP) model is worth mentioning. The BSP model was introduced by Valiant [29] in an effort to find a bridging model between hardware and software for parallel computing similar to the Von Neumann architecture for sequential computing. As Valiant describes it:

Thus, the von Neumann model is the connecting bridge that en-
ables programs from the diverse and chaotic world of software to run efficiently on machines from the diverse and chaotic world of hardware. Our claim is that what is required before general purpose parallel computation can succeed is the adoption of an analogous unifying bridging model for parallel computation.

The resulting BSP model can in fact completely capture the type of computation executed in the MPC model. Unfortunately, as pointed out by Andoni et al. in [1], the many parameters made it unwieldy to use for the design of algorithms and therefore even the most fundamental results are lacking.

Lastly there is the congested clique model, initially proposed to capture computing on overlay networks [25]. The network is presented as a fully connected graph, where on each node there is a machine and the machines can communicate in an all-to-all fashion. The precise method of communication is abstracted away – there could be a direct link between all machines or a routing protocol to make communication possible. Therefore, the communication is considered to be the bottleneck and only small messages can be sent (of size $O(\log n)$, with $n$ the number of nodes in the graph). Computation proceeds again in synchronized rounds where each machine executes some local computation and then sends messages to other machines. The local memory and computation power are unlimited. Similar to the message passing model (LOCAL and CONGEST) congested clique is often used to analyse algorithms for graph problems. The initial distribution of data is the same: initially each machine receives the ID of one vertex of the input graph and the IDs of its neighbors. However in congested clique a machine can communicate directly with every other machine while in the LOCAL/-CONGEST model a node can only communicate with its neighbors from the input graph. As we will see, the congested clique model is very closely related to one specific setting of MPC.

### 2.2 Model Definition

The model we use in this work can be traced back to descriptions given by Karloff et al. [19] and Feldman et al [11] which was later refined by several works [14, 5, 1]. Given a problem with input size $N$, there are $M$ machines, each with $S$ words of memory. The size of each machine’s memory is limited with respect to the total problem size: $S \leq N^{1-a}$ for a constant $a > 0$. On the other hand the cluster of machines has to be able to store the input, which implies a natural lower bound for the number of machines: $M \geq \frac{N}{S}$. This is typically tight up to a logarithmic factor: $M = \tilde{O}(\frac{N}{S})$.

Initially the input is divided arbitrarily among all machines. Computation proceeds in synchronous rounds, where in each round every machine can execute some computation on the data it holds. This local computation is restricted to be of polynomial running time with respect to the local mem-
The MPC Model

ory size. After a round of computation there is a round of communication where each machine can send to every other machine some data – thus the network graph is the complete graph. The only restriction on the communication is that the total amount of data that one machine sends or receives cannot exceed its memory capacity $S$. The bottleneck in this model is the communication. Therefore the analysis of an algorithm running in the MPC model is focused on the number of rounds of computation.

When considering graph optimization problems there is a further distinction based on how much memory a machine has relative to the number of vertices, $n$, in the graph.

- Strictly super-linear memory regime: $S \geq n^{1+\beta}$, for a constant $\beta \in (0, 1)$
- Near linear memory regime: $S \in \Theta(n)$
- Strictly sub-linear memory regime: $S \leq n^{1-\beta}$, for a constant $\beta \in (0, 1)$

In this work we consider the near linear memory regime. It should be noted that using this notion for the memory constraint, the model deviates slightly from the description of MPC given above when the input is a sparse graph. Namely consider a sparse graph on $n$ vertices with only $O(n)$ edges. Now in the near linear memory regime, the whole graph fits in one machine, which is prohibited in the original statement of MPC. Therefore we will be focusing in this thesis on dense graphs.

Now we can also formalize the relation between the congested clique model and the MPC model. In [6] the authors show a two way simulation to show that the near linear memory MPC setting, in their words semi-MapReduce, is equivalent to congested clique. The following theorem formalizes the notion of simulating an MPC algorithm in the congested clique model.

**Theorem 2.1 (Theorem 3.2 in [6])** Let $A_M$ be an algorithm that for an input graph $G$, with $n$ vertices, runs in $T$ rounds of semi-MapReduce. One can simulate $A_M$ in $O(T)$ rounds of congested clique.

### 2.3 State-of-the-art

Let us briefly discuss the state-of-the-art of algorithm design in MPC with a focus on problems related to minimum weight vertex cover. First of all there is a positive simulation result by Karloff et al. [19] that proves that a large class of PRAM algorithms can be simulated in MPC. More precisely, any CREW PRAM algorithm using $O(n^{2-2\epsilon})$ total memory, $O(n^{2-2\epsilon})$ processors and $t = t(n)$ time can be run in $O(t)$ rounds of MPC. Later Goodrich et al. [14] expanded this result to the most general class of PRAM algorithms: CRCW.

Therefore a new goal would be to find algorithms that run strictly faster
than their PRAM counter parts. When considering the strictly super linear memory regime, many problems are solved in a constant number of rounds. Already in the work by Karloff et al. [19] a constant round algorithm is presented to compute a minimum spanning tree. Then Lattanzi et al. introduced a powerful technique called filtering [23]. The main idea is to peel of vertices or edges and thereby sparsifying the graph until it fits in one machine. Then the result is computed on this smaller graph. Using this approach they provide constant round algorithms in the super linear memory regime for minimum spanning tree, maximal matching, nearly maximum weighted matching, vertex and edge cover.

Also on the other end of the memory spectrum, the strictly sub-linear memory regime, it is possible to design strictly faster algorithms as demonstrated by the recent results of Ghaffari et al. [13]. They provide algorithms for maximal independent set, maximal matching, 2-approximation of minimum vertex cover, and $(1 + \varepsilon)$-approximation of maximum matching that all run in $O\left(\sqrt{\log n}\right)$ rounds.

Finally in the near linear memory regime an important breakthrough came from Czumaj et al. that showed the first sub logarithmic time algorithm for maximum matching [8]. The technique of round compression is introduced in that work. The main idea is fairly simple: we compress a large amount of iterations of a LOCAL/centralized algorithm into a constant number of MPC rounds. Roughly speaking, this is done as follows: at the start of an MPC round we partition the vertices randomly among a set of machines and gather the induced sub-graphs on each machine. Next we simulate the centralized algorithm for many iterations using only local information. Finally the result is sent to the other machines and the process can be repeated for the next phase of iterations. When introducing our algorithm in chapter 4 we will discuss the mechanics of round compression in more detail. The running time of the algorithm presented by by Czumaj et al. [8] is $O\left((\log \log n)\right)^2$ to compute a $(2 + \varepsilon)$-approximate maximum matching. Later this result has been improved and simplified by Assadi et al. [2] and Ghaffari et al. [12] to reach an $O\left(\log \log n\right)$ round algorithm that computes a $(1 + \varepsilon)$ approximate maximum matching and $(2 + \varepsilon)$ approximate minimum vertex cover.

2.4 Notation and Preliminaries

In this section we provide some tools and notation we will frequently use throughout this thesis. We denote an undirected graph $G = (V, E)$ by its vertex set $V$ and its edge set $E$, where an edge $e \in E$ is an unordered pair of vertices, e.g. $e = (u, v)$ represents an undirected edge between the vertices $u, v \in V$. We use the words vertices and nodes interchangeably. Furthermore we denote the degree of a vertex with $d(v)$ and the maximum degree
of any vertex present in the graph is $\Delta$. To indicate that a certain variable $a$ takes values in the interval $[b - c, b + c]$ we slightly abuse notation and write this as $a = b \pm c$. For further reference on graph theory we direct the interested reader to any standard graph theory textbook.

When talking about running times and round complexity we use the usual notions $O, \Theta, \Omega$. When written with an additional tilde, e.g. $\tilde{O}$, it indicates that additional polylogarithmic factors are hidden. For example, the following relations hold: $n^2 \log^3 n \in \tilde{O}(n^2)$ and $23 \log n \log \log n \in \tilde{O}(\log n)$. We will frequently use the following form of Chernoff bounds to bound the tails of a sum of independent random variables.

**Theorem 2.2 (Chernoff bounds)** Let $X = \sum_{i=1}^n X_i$, where $X_i = 1$ with probability $p_i$ and $X_i = 0$ with probability $1 - p_i$, and all $X_i$ are independent. Let $\mu = \mathbb{E}(X) = \sum_{i=1}^n p_i$. Then

- $P(|X - \mu| \geq \delta \mu) \leq 2 \exp \left( -\frac{\delta^2 \mu}{2} \right)$ for $0 \leq \delta \leq 1$
- $P(|X - \mu| \geq \delta \mu) \leq 2 \exp \left( -\frac{\delta \mu}{2} \right)$ for $\delta > 1$

More information about the background and applications of these Chernoff bounds can be found in [10]. When talking about linear programs (LPs), we use the strong and weak duality theorems:

**Theorem 2.3 (Strong duality)** Given a primal problem: maximize $c^T x$ subject to $Ax \leq b$ and $x \geq 0$ and its dual: minimize $b^T y$ subject to $A^T y \geq c$ and $y \geq 0$, an optimal primal solution $x^*$ and optimal dual solution $y^*$ satisfy: $c^T x^* = b^T y^*$.

**Theorem 2.4 (Weak duality)** Given a primal problem: maximize $c^T x$ subject to $Ax \leq b$ and $x \geq 0$ and its dual: minimize $b^T y$ subject to $A^T y \geq c$ and $y \geq 0$, an optimal primal solution $x^*$ and optimal dual solution $y^*$ satisfy: $c^T x^* \leq b^T y^*$.

More information about the theory of linear programming can be found in [7]. Finally, we use the phrase with high probability, or the abbreviation w.h.p., to indicate that an event happens with probability at least $1 - \frac{1}{n^c}$ for any desirable constant $c > 0$. 


Chapter 3

Minimum Weight Vertex Cover

The problem of finding a minimum weight vertex cover (MWVC) in a graph is considered as one of the central symmetry breaking problems in distributed computing and hence has a long history. A symmetry breaking problem in distributed computing is any problem where initially all entities (processors/machines in a network or vertices in a graph) are in the same state, however during the course of the algorithm different entities have to take on different roles to complete the task. Leader election among a set of machines is the most basic example. Other examples include maximum matching, maximal independent set, vertex coloring, etc. In this chapter we state the exact problem of minimum weight vertex cover and discuss some algorithms in closely related models.

3.1 Problem description

Informally speaking, a vertex cover for a graph is a subset of its vertices such that every edge of the graph has at least one endpoint in this subset. More formally, we have the following definition.

Definition 3.1 Let \( G = (V, E) \) be a graph and \( S \subseteq V \). \( S \) is called a vertex cover for \( G \) if \( \forall e \in E : e \cap S \neq \emptyset \).

For a graph \( G = (V, E) \) with non-negative weights assigned to its vertices, we represent these weights as a function

\[
  w : V \rightarrow \mathbb{R}^+,
\]

so that the weight of a specific vertex \( v \in V \) is denoted by \( w(v) \). For a subset of vertices \( S \subseteq V \), we denote with \( w(S) \) the sum of the weights of vertices in \( S \):

\[
  w(S) = \sum_{v \in S} w(v).
\]
With this notation we are ready to introduce the problem.

**Problem 3.2 (Minimum weight vertex cover (MWVC))** Given an undirected graph $G = (V, E)$ with non-negative vertex weights $w : V \rightarrow \mathbb{R}^+$, find the vertex cover that has the minimum weight with respect to $w$.

### 3.2 Background

The vertex cover problem has a long history in both distributed and centralized computing. The unweighted vertex cover problem is one of Karp’s 21 NP-complete problems [20]. Since the weighted variant is a generalization, it is also NP-complete and we can only hope for good approximation algorithms unless $P = NP$. In the theory of approximation algorithms, the vertex cover problem occupies an interesting place. There are NP-hard problems which admit fast approximation algorithms to get a solution within a $(1 + \epsilon)$ factor of the optimal for any constant $\epsilon$; the knapsack problem is an example of such a problem [17]. On the other end of the spectrum are problems that don’t admit any polynomial time constant factor approximation algorithm; the general traveling salesman problem is an example [28]. In the middle of these two extremes are the vertex cover and weighted vertex cover problems. While it is possible to achieve a 2-approximation, it is not possible to get a strictly better approximation based on the unique games conjecture [21].

There is a plethora of algorithms that provide a 2-approximation, but after a closer examination, most of these algorithms fall in either one of two categories: LP-based algorithms or combinatorial algorithms. To see the applicability of linear programming techniques, we can formulate problem 3.2 as an integer program.

$$\begin{align*}
\min & \sum_{v \in V} y_v \cdot w(v) \\
\text{s.t.} & \quad y_u + y_v \geq 1 & \forall (u, v) \in E \\
& \quad y_v \in \{0, 1\} & \forall v \in V
\end{align*}$$

Now when relaxing the integer constraints for $y_v$, this becomes a linear program. Moreover, the extreme points of this LP have a very interesting property: they are half integral. This means that all the coordinates of any extreme point have values in $\{0, \frac{1}{2}, 1\}$. Then it is easy to see that once an optimal solution for the linear program is found, it can be turned into a 2-approximate integral solution by simply picking all the vertices with non zero $y_v$ value. This is essentially the very first 2-approximation algorithm for MWVC, due to Nemhauser et al. [27] and made explicit by Hochbaum [16].

Alternatively to such LP based approaches, there are algorithms that exploit
3.2. Background

some combinatorial property of the problem. Often such algorithms give some more insight into the problem at hand. We present one example of such an algorithm here and use a result by Bar-Yehuda et al. [4] to analyse its accuracy. This result is also of particular interest since it captures many of the combinatorial algorithms that exist today for MWVC.

At first glance Algorithm 1 might seem counter-intuitive. After all, why are the weights reduced for vertices that are not included in the cover? One might think of a more obvious greedy algorithm that continuously picks the vertex that covers the most edges with the lowest cost, but this in fact gives no constant approximation at all. Algorithm 1 in contrast achieves a 2-approximation. For the analysis we will use the following local ratio lemma by Bar-Yehuda et al.

**Algorithm 1:** simple MWVC algorithm

1. Input: graph \( G = (V, E) \) with weights \( w : V \rightarrow \mathbb{R}^+ \)
2. Output: 2-approximate minimum vertex cover
3. initialize \( \forall v \in V : w_0(v) \leftarrow w(v) \)
4. iterate over all \( e = (u, v) \in E \) in any order:
   a) set \( \delta = \min\{w_0(v), w_0(u)\} \)
   b) update \( w_0(v) \leftarrow w_0(v) - \delta \) and \( w_0(u) \leftarrow w_0(u) - \delta \)
5. Return set \( C = \{v \in V | w_0(v) = 0\} \)

**Lemma 3.3 (cf [4])** Let \( G(V, E) \) be a graph and \( w, w_1, w_2 \) be weight functions on \( V \), s.t. for every \( v \in V : w(v) \geq w_1(v) + w_2(v) \). Let \( C^*, C_1^* \) and \( C_2^* \) be optimum covers of \( G \) with respect to \( w, w_1 \) and \( w_2 \). It follows that

\[
w(C^*) \geq w_1(C_1^*) + w_2(C_2^*)
\]

Given this lemma, one can start to understand why Algorithm 1 finds a 2-approximation. Consider the update step in line 4b where the weight is reduced from both endpoints of an edge. This can also be viewed as a split of the weight function in two parts for these two vertices. We can combine all of these splits into one weight function, say \( w_1 \), such that at the end of the algorithm we have \( w = w_0 + w_1 \). Now what remains to be shown is that the solution found at the end of the algorithm is almost optimal with respect to these weight functions constructed during the algorithm. Then we can apply the local ratio lemma to give the desired result. This idea is formalized in the proof of the following theorem.
3. Minimum Weight Vertex Cover

**Theorem 3.4** Denote by OPT the weight of an optimal vertex cover for \( G = (V,E) \) and \( w : V \to \mathbb{R}^+ \). Then running algorithm 1 on \( (G,w) \) returns a cover \( C \) that satisfies \( w(C) \leq 2 \cdot OPT \).

**Proof** A \( \delta \) is computed for every edge. Store this value for edge \( e \) in \( x_e \). Define the following weight function: \( w_1(v) = \sum_{e \ni v} x_e \). Then by the end of the algorithm, we have for each vertex \( v \) : \( w(v) = w_0(v) + w_1(v) \). The set \( C \) is obviously an optimal cover with respect to the weight function \( w_0 \).

First of all notice that \( C \) is indeed a cover: fix any edge \( e = (u,v) \); when it is processed in the main loop on line 4 we have that either \( w_0(v) = 0 \) or \( w_0(u) = 0 \). Therefore, every edge has at least one endpoint in \( C \). Clearly \( C \) is optimal since it has zero weight. Next we claim that \( \sum_{v \in V} w_1(v) \leq 2 \cdot OPT_1 \), where \( OPT_1 \) is the weight of an optimal cover with respect to weight function \( w_1 \). Using this claim, the theorem follows immediately:

\[
w(C) = w_0(C) + w_1(C) \leq OPT_0 + 2 \cdot OPT_1 \leq 2(OPT_0 + OPT_1) \leq 2 \cdot OPT
\]

where the last inequality follows from Lemma 3.3. What remains to be proven is the claim: \( \sum_{v \in V} w_1(v) \leq 2 \cdot OPT_1 \). For this, we use a charging argument. We charge the edges of the graph in two different ways with money and compare the amounts each edge receives in the two situations. In the first situation, we put on every vertex \( v \in V \) an amount of \( w_1(v) \) dollars. Then every vertex distributes their money over their incident edges where edge \( e \ni v \) gets exactly \( x_e \) dollars (notice that this is indeed a partitioning of the money due to the definition of the weight function \( w_1 \)). So after the first charging, each edge gets precisely \( 2 \cdot x_e \) dollars. Now for the second setting, consider an optimal vertex cover \( C_1 \) with respect to weights \( w_1 \). We put on every vertex in the optimal cover, \( v \in C_1 \), an amount of \( w_1(v) \) dollars. We distribute this amount again over the edges using the same rule as before: a vertex \( v \in C_1 \) gives \( x_e \) dollars to each incident edge \( e \ni v \). Now we have that each edge receives at least \( x_e \) dollars, since at least one of its endpoints is in the cover \( C_1 \). Therefore we have that every edge receives at most twice as much in the first setting as in the second. Moreover, the total sum of the money on the edges in the first case is exactly \( \sum_{v \in V} w_1(v) \), and in the second case it is precisely \( OPT_1 \) and hence the claim follows.

\( \square \)

### 3.3 State-of-the-art

To the best of our knowledge, there is no specific result for the minimum weight vertex cover problem in the near linear memory regime of MPC. However in the strictly super-linear memory regime, Harvey et al. [15] give a constant round \( f \)-approximation algorithm for set cover. Here \( f \) denotes the maximum frequency of any element, i.e. in how many sets it appears. The vertex cover problem is an instance of the set cover problem with \( f = 2 \),
therefore this algorithm translates to a constant round 2-approximation for minimum weight vertex cover. Unfortunately, when considering the same approach for the near linear memory setting, the running time collapses to $O(\log n)$ rounds. In the LOCAL model, Bar-Yehuda et al. \cite{3} provide an algorithm that finds a $(2 + \epsilon)$-approximation for weighted vertex cover in $O(\log(\Delta) / \epsilon \log \log(\Delta))$ rounds. It is important to note that this is tight according to a lower bound of Kuhn et al. \cite{22}. That algorithm also gives an MPC algorithm of the same running time by doing a straightforward simulation, where one machine executes the steps for one vertex.
In this chapter we give an algorithm in the MPC model for computing a minimum weight vertex cover of a graph as defined in problem 3.2. As discussed earlier in chapter 2, one major advancement in the design of MPC algorithms is the technique of round compression, first introduced in [8]. Our MPC algorithm leverages this idea to compress the rounds of a very elegant centralized algorithm. The centralized algorithm we start with computes a minimum weight vertex cover in $O(\log \Delta)$ rounds. Then using techniques from [12] and an observation about the local degree, the round complexity in MPC can be shown to be $O(\log \log d)$, where $d$ is the average degree of the graph. In the first section we outline the centralized algorithm together with an analysis of its running time and accuracy. Then follows a section about how to apply the round compression technique successfully. Finally we present the full MPC algorithm.

### 4.1 $O(\log \Delta)$ Centralized Algorithm

Let us start by writing down the primal and dual program that arise after relaxing the integer program (see Figure 4.1). Algorithm 2 can be seen as a primal-dual scheme for this formulation. The variables $\{x_e\}_{e \in E}$ correspond to the dual variables which maintain a fractional matching. On the other hand the variables $\{y_v\}_{v \in V}$ keep track of which dual constraints become $\varepsilon$-tight. Notice that the variables $y_v$ do not correspond to the primal variables

\begin{align*}
\text{Primal} & \quad \text{Dual} \\
\min & \quad \sum_{v \in V} z_v \cdot w(v) & \max & \quad \sum_{e \in E} x_e \\
\text{s.t.} & \quad z_u + z_v \geq 1 & \forall (u,v) \in E & \text{s.t.} \quad \sum_{e \ni v} x_e \leq w(v) & \forall v \in V \\
& \quad z_v \geq 0 & \forall v \in V & \quad x_e \geq 0 & \forall e \in E
\end{align*}

\begin{figure}[h]
\centering
\begin{align*}
\text{Figure 4.1: Linear programming relaxation for MWVC}
\end{align*}
\end{figure}
4. The Algorithm

Algorithm 2: Central MWVC algorithm

1. Input: graph \( G = (V, E) \), weight function \( w : V \rightarrow \mathbb{R}^+ \)
2. Initialization:
   - \( \forall e = (u, v) \in E : x_e = (1 - 2\varepsilon) \cdot \min \{ \frac{w(v)}{d(v)}, \frac{w(u)}{d(u)} \} \)
   - \( C = \emptyset \)
3. While not all edges frozen, in iteration \( t \):
   - If \( y_v = \sum_{e \ni v} x_e \geq (1 - 2\varepsilon)w(v) \): add \( v \) to \( C \) and freeze its incident edges
   - for all unfrozen edges: \( x_e \leftarrow x_e / (1 - \varepsilon) \)
4. Return set \( C \) as a vertex cover

\( z_0 \) from the linear program. The idea is now to slowly increase the dual variables while not violating the dual constraints and hence maintaining a valid fractional matching. Then we can argue that the set of \( \varepsilon \)-tight vertices corresponds to a \((2 + \varepsilon)\)-approximate minimum vertex cover using weak LP-duality. This is formalized in the following lemma.

Lemma 4.1 Algorithm 2 returns a vertex cover \( C \) which satisfies:

\[
w(C) \leq (2 + \varepsilon)OPT,
\]

where \( OPT \) is the weight of a minimum weight vertex cover. Moreover algorithm 2 terminates in \( O(\log \Delta) \) rounds.

Proof First we argue that the algorithm maintains a valid fractional matching. Denote with \( x_{e,t} \) the weight of an edge \( e = (u, v) \) after \( t \) iterations. Consider any vertex \( v \), after initialization no constraint can be violated:

\[
\sum_{e \ni v} x_{e,0} \leq d(v) \cdot \frac{(1 - 2\varepsilon)w(v)}{d(v)} < w(v)
\]

Now assume that we have a valid matching after round \( t \) then we have for any active vertex \( v \) in round \( t + 1 \):

\[
y_{v,t+1} = \sum_{e \ni v} x_{e,t+1} \leq \sum_{e \ni v} \frac{x_{e,t}}{1 - \varepsilon} \leq \frac{(1 - 2\varepsilon)w(v)}{(1 - \varepsilon)} < w(v)
\]

Where the one but last inequality follows from the fact that vertex \( v \) was active after round \( t \).

Next we claim that the set of frozen vertices forms a valid vertex cover at
the end of the algorithm. This follows from the fact that the algorithm only terminates when all edges have been frozen, i.e. when they contain at least one vertex that is frozen. Therefore the set of frozen vertices covers all the edges.

Finally we will relate the weight of the fractional matching to the size of the vertex cover and use LP-duality to prove the claimed approximation ratio. Denote the value of the final fractional matching by $W_M = \sum_{e \in E} x_e$. We have the following:

$$\forall v \in C : y_v = \sum_{e \in \delta v} x_e \geq (1 - 2\epsilon)w(v).$$

Next, sum over all vertices in the cover:

$$\sum_{v \in C} \sum_{e \in \delta v} x_e \geq (1 - 2\epsilon)w(C).$$

Each edge can be covered at most twice, once per endpoint:

$$2W_M = 2 \sum_{e \in E} x_e \geq \sum_{v \in C} \sum_{e \in \delta v} x_e \geq (1 - 2\epsilon)w(C).$$

Now we can apply LP-duality since we have shown that the set $C$ is indeed a vertex cover and the fractional matching maintained is feasible and therefore is a lower bound for the optimal minimum vertex cover:

$$w(C) \leq \frac{2}{1 - 2\epsilon} W_M \leq \frac{2}{1 - 2\epsilon} OPT.$$ 

Next for the running time, look at any edge $e = (u,v)$ and w.l.o.g. assume its initial weight is $x_e = (1 - 2\epsilon)\frac{w(u)}{\delta(u)}$, then after at most $\log_{\frac{1}{1 - (1 - \epsilon)}}(\Delta)$ iterations we have that $x_e \geq (1 - 2\epsilon)w(u)$, at which point it will definitely become frozen. Since this is true for any edge, we have that the running time is bounded by

$$\log_{\frac{1}{1 - (1 - \epsilon)}}(\Delta) \in O(\log \Delta),$$

since $\epsilon$ is a constant.

This analysis only shows that after enough rounds every edge will be incident to a vertex that is in the vertex cover, but it does not reveal how progress is actually made. As we will see later it is useful to find some parameters that characterize the progress made by the algorithm after a number of iterations. An obvious first idea is to look at the maximum degree (with respect to active edges) present in the graph. However it is not clear that in this algorithm the maximum degree will steadily decrease (the maximum degree will of course decrease monotonically, but there may be very irregular jumps in these drops). Therefore let us orient the edges from the endpoint which has the lower weight over degree ratio to the higher weight over degree.
4. The Algorithm

ratio. That is, we orient \( e = (u, v) \) from \( u \) to \( v \) if \( \frac{w(u)}{d(u)} < \frac{w(v)}{d(v)} \) and reverse otherwise, breaking ties arbitrarily. Now the degree of a vertex splits into an in-degree \( d^{in} \) and an out-degree \( d^{out} \). Also denote the neighbors of \( v \) from in-edges and out-edges, with \( N^{in}(v) \), respectively \( N^{out}(v) \). When we inspect now the maximum degrees present in the graph, we can show that the maximum out-degree is steadily decreasing. This is because of the condition for a vertex to stay active after \( t \) iterations:

\[
y_v = \sum_{u \in N^{in}(v)} \frac{(1 - 2\varepsilon)w(u)}{(1 - \varepsilon)^t d(u)} + \sum_{u \in N^{out}(v)} \frac{(1 - 2\varepsilon)w(v)}{(1 - \varepsilon)^t d(v)} < (1 - 2\varepsilon)w(v).
\]

Rewriting inequality 4.1 delivers the following condition on the out-degree of any active vertex \( v \):

\[
d^{out}(v) < \frac{(1 - 2\varepsilon)w(v)d(v)(1 - \varepsilon)^t}{(1 - 2\varepsilon)w(v)} = d(v)(1 - \varepsilon)^t
\]

Also looking at this condition, we see that it is valid for every vertex. This already hints at the possibility for the total running time to depend on the average degree rather than the maximum degree.

4.2 Round Compression

Now that we have established a clean centralized algorithm the idea is to apply a form of round compression as introduced in [8]. The main idea is fairly straightforward: partition the vertices of the graph randomly among a set of machines, gather the induced sub-graphs on the machines and locally simulate a number of rounds of the centralized algorithm. By simulating we mean that each vertex only looks at its local neighborhood in the induced graph and infers from that information its global properties and makes a decision. If almost all vertices can accurately be simulated over the number of iterations executed in one MPC round, there is a hope that this procedure will produce a very similar result as the centralized algorithm. Let us now make this more precise and look at the requirements for this simulation to work well.

As seen in the previous section, a vertex \( v \) needs to check in every round whether it has to become frozen or not based on the sum of the incident edges.

\[
y_v = \sum_{\ell \in v} x_\ell \geq (1 - 2\varepsilon)w(v)
\]

In the MPC simulation the vertices are partitioned among \( m = \sqrt{d} \) machines. We look again at the vertex \( v \), but now on the machine it landed. There \( v \) only sees a fraction of its neighbors. We denote by \( N_{local}(v) \) the neighbors
of $v$ that landed on the same machine. Then instead of calculating $y_v$ we estimate this value by $\tilde{y}_v$ as follows:

$$ \tilde{y}_v = m \cdot \sum_{u \in N^{\text{out}}(v)} x(u,v) $$

An obvious first requirement is that it has to be possible to estimate this value $\tilde{y}_v$ well. Secondly we need the property that if a vertex is indeed able to estimate this sum well, meaning $|\tilde{y}_v - y_v| \ll \epsilon$, it makes the same decision in the MPC simulation as in the centralized algorithm with high probability. Finally, for the ease of the analysis, we would like that the failure probability for a vertex is independent between iterations in one phase. If we can achieve these requirements it is clear that in at least a few simulated iterations the two algorithms make for all vertices the same decision and hence produce the same vertex cover.

However, to achieve a considerable speedup, it is important to run as many iterations as possible in one phase. If it is possible to simulate $I = \log_{1/(1-\epsilon)}(d) / 20$ iterations in one phase, we have for every vertex $v$ that its active out-degree after the phase ends, is at most $d(v) \cdot d^{-0.05}$ following inequality 4.2. Summing up over all vertices we can then conclude that the average degree is reduced from $d$ to $d^{0.95}$. Such a progress in one step means that after only $O(\log \log d)$ phases the graph is small enough to fit into one machine and the algorithm can terminate.

### 4.2.1 Why not run even more iterations in one phase?

It would be nice to simulate even more iterations so that we only need $O(\log \log \log d)$ or even a constant number of phases. There are two main limiting factors that prevent this. The first issue is that we are dealing with an estimate and thus there is a possibility that a vertex makes a different decision in the two algorithms. As soon as one vertex behaves differently in the two algorithms, we call such a vertex bad, this might influence the estimation of the neighbors of that vertex in the next round. Then these skewed estimations might influence the estimation of their neighbors etc. This means that we expect $|\tilde{y}_v - y_v|$ to grow slowly the more iterations are simulated. Until after too many iterations this difference becomes so large that there is a large fraction of the vertices that can become bad in one iteration. The second issue has to do with the ability to estimate $y_v$ well as stated in the first requirement above. Therefore let us go over these requirements in more detail.

### 4.2.2 Round compression requirements

The first requirement states that we have to be able to estimate $y_v$ well from looking at a random sample of the neighbors. From probability theory we
4. The Algorithm

know that when we sample independently from a large enough set, the obtained sample will be close to the expected value. Informally speaking this translates in our situation to having a large enough degree, such that after sampling there is a considerable number of neighbors present on the same machine. Later in the analysis we will formalize these notions of “large enough degree” and “considerable number”. This explains also the second limiting factor for running an even larger number of iterations in one phase, we simply can not expect to see any neighbors any more in the local machine. To achieve that there are enough local neighbors we can restrict the vertices that are being simulated to the vertices with a large degree. Vertices with a low degree will not be simulated and therefore their incident edges will not increase in weight. Otherwise the constraint of such a vertex might be violated. For further reference we call these vertices and edges sleeping, they are not frozen nor active. An important observation here is that by restricting the lowest degree vertices to be sleeping, only a small fraction of the total amount of edges is sleeping during one phase. The second and third requirements can actually be addressed at the same time with the technique of using random thresholds (introduced in [12]). First notice that there is a problem by using a fixed threshold to decide if a vertex becomes frozen (as in algorithm 2). Because even when the estimate $\tilde{y}_v$ is close to the true value $y_v$, they might still fall on different sides of this threshold. Moreover this might happen for a large fraction of the vertices at the same time. This issue can be alleviated by introducing a random threshold as in Algorithm 3 below. To see the difference, consider again a vertex for which $\tilde{y}_v$ is very close to $y_v$. To end up on different sides of the threshold means that this threshold has to fall in the interval between $\tilde{y}_v$ and $y_v$. Since the threshold is chosen uniformly at random, this probability is proportional to the closeness of the estimate and the true value. Moreover by choosing these thresholds independently for every vertex and every iteration, we achieve that the probability of a vertex turning bad is independent between vertices and iterations.

So far we have introduced a few states a vertex can be in. We briefly repeat what each of these states mean. A vertex that joins the cover is called frozen, its edges are not increasing in weight anymore. Vertices that are not yet frozen are called alive. Between the alive vertices there is a further distinction based on the degree with respect to alive vertices. Vertices with a low alive degree are called sleeping, they are not being simulated in this phase. Vertices with a high alive degree are being simulated and are called active. Then finally we call a vertex bad if it is frozen in the centralized algorithm and alive in the MPC simulation or vice versa. A good vertex is not bad. Table 4.1 below summarizes these states.
Algorithm 3: Randomized central MWVC algorithm

1. Initialization:
   - Each vertex $v$ chooses a list of thresholds $T_{v,t}$ such that: the thresholds are chosen independently; each threshold is chosen uniformly at random from $[1 - 4\epsilon, 1 - 2\epsilon]$
   - $\forall e = (u, v) \in E : x_e = (1 - 2\epsilon) \cdot \min\{\frac{w(u)}{d(v)}, \frac{w(u)}{d(u)}\}$
   - $C = \emptyset$

2. While not all edges frozen, in iteration $t$:
   - If $y_v = \sum_{e \ni v} x_e \geq T_{v,t}w(v)$: add $v$ to $C$ and freeze its incident edges
   - for all unfrozen edges: $x_e \leftarrow x_e / (1 - \epsilon)$

3. Return set $C$ as a vertex cover

| State   | Meaning                                                      |
|---------|--------------------------------------------------------------|
| Frozen  | $y_v \geq (1 - 2\epsilon)w(v)$, and $v$ belongs to the cover |
| Alive   | Not frozen                                                   |
| Sleeping| Alive with a small degree $d(v) \leq d^{0.95}$, incident edges do not increase weight |
| Active  | Alive and high degree $d(v) \geq d^{0.95}$, incident edges are increasing in weight |
| Bad     | Frozen in simulation and alive in centralized or vice versa |
| Good    | Not bad, in the same state in both algorithms                |

Table 4.1: Vertex states

4.3 MPC Simulation

The simulation in the MPC model is given in algorithm 4. Initially every vertex picks a list of random thresholds. In practice, these thresholds can be picked on the fly to reduce the memory requirements. The average degree is calculated and stored in $d$. Through the remainder of the algorithm the variable $d$ does not exactly correspond to the average degree but is a valid upper bound, as we will show in the analysis. Also a global iteration counter is set $t = 0$. This counter is used to compare a specific iteration of the MPC algorithm with the corresponding iteration in the centralized algorithm. Moreover when a vertex gets frozen it stores this iteration, so it can use this information at the end of the phase to update incident edge...
weights (see line 3j). The set $V'$ will maintain the vertices for which the dual constraint is not violated. This is important because in doing the simulation it might happen that at some point a dual constraint gets violated because $y_v$ was not estimated well. At that point the fractional matching would not be valid anymore and this prohibits the application of LP-duality later on. To avoid this situation we simply remove vertices that became too heavy, see also line 3l. We call one execution of line 3 a phase.

In the beginning of a phase the residual weight is determined, $w'(v) = (1-2\varepsilon)w(v) - \sum_{e \in V_0, \text{frozen}} x_e$, and the weights of all non frozen edges, we call this set $E^{\text{alive}}$, is redistributed as follows: $\forall (u,v) = e \in E^{\text{alive}}, x^{\text{MPC}}_e = \min\{w'(u), w(v)\}$. This redistribution of weights makes the analysis cleaner, because otherwise we would have to keep track of the edges that have been sleeping in a previous phase. On the following line 3c only the vertices with a high enough degree are selected in $V^{\text{active}}$. The graph $G'$ represents the induced sub-graph on $V^{\text{active}}$. Then the value $y^{\text{fixed}}_v$ is calculated, which represents the sum of incident edges that remain constant during the current phase. Those are the frozen and sleeping incident edges. On line 3f the number of machines $m$ that will be used in the simulation of this phase is calculated together with the number of iterations that will be simulated $I$. Then the vertices get partitioned uniformly at random between $m$ machines and the respective sub-graphs are gathered on each machine. Subsequently, on line 3h, the centralized algorithm is simulated in each of the $m$ machines locally. Afterwards, the upper bound for the average degree is updated and the active edges that were not simulated are updated. Notice that an edge is not simulated when its endpoints land in a different machine. At the very end of the phase the values $y^{\text{MPC}}_v$ are calculated, using the updated edge weights. These can be seen as an implicit estimate for the variable $y_v$. This value is then used to check the feasibility of the current fractional matching. Once the average degree is below $\log^{20} n$, there are at most $n \log^{20} n \in \tilde{O}(n)$ edges left. These can be gathered in one machine to execute directly the last iterations of the centralized algorithm.
4.3. MPC Simulation

**Algorithm 4: MPC-Simulation for MWVC**

1. Each vertex $v$ chooses a list of thresholds $\mathcal{T}_v$ such that: the thresholds are chosen independently; each threshold is chosen uniformly at random from $[1 - 4\epsilon, 1 - 2\epsilon]$
2. Init: $V' = V$; $d = \sum_{v \in V} d(v) / n$; $t = 0$
3. While $d > \log_{20} n$
   a) Update residual weights: $w'_v(v) = (1 - 2\epsilon)w(v) - \sum_{e \ni v, \text{frozen}} x_e$
   b) Assign edge weights $\nu(u, v) = e \in E^{\text{alive}}$, set $x^\text{MPC}_{e,0} = \min \{ w'_v(u), w'_v(v) \}$
   c) Let $V^{\text{active}} = \{ v \in V' | d(v) \geq d^{0.95} \}$
   d) Let $G'$ be the induced sub-graph on $V^{\text{active}}$
   e) For each $v \in V^{\text{active}}$, define $y^\text{fixed}_v = \sum_{e \ni v \in G[V'] \cap G'} x^\text{MPC}_e$
   f) Set number machines $m = \sqrt{d}$; number iterations $I = \frac{\log m}{10 \log(5)}$
   g) Partition $V^{\text{active}}$ into $m$ sets $V_1, \ldots, V_m$ by assigning each vertex to a machine independently and uniformly at random
   h) For each $i \in \{1, \ldots, m\}$ in parallel execute $I$ iterations
      i. For each $v \in V_i$ such that
         \[ g_{v,t} = m \cdot \sum_{e \ni v \in G[V_i]} x^\text{MPC}_e + y^\text{fixed}_v \geq \mathcal{T}_v \]
         set $x^\text{MPC}_e = \frac{y^\text{fixed}_v}{1 - \epsilon}$, freeze $v$ and freeze all its incident edges.
      ii. For each active edge of $G'[V_i]$, set $x^\text{MPC}_e = \frac{x^\text{MPC}_e}{1 - \epsilon}$
      iii. Increment the global iteration count: $t = t + 1$
   i) Update $d = 2d(1 - \epsilon)^I$
   j) For every non sleeping edge $e = u, v$: set $x^\text{MPC}_e = \frac{x^\text{MPC}_e}{1 - \epsilon}$, where $I'$ is the last iteration in which both $u$ and $v$ were active in this phase.
   k) For each $v \in V'$ let $y^\text{MPC}_v = \sum_{e \ni v \in G[V']} x^\text{MPC}_e$
   l) For each $v \in V'$ such that $y^\text{MPC}_v > 1$: remove $v$ from $V'$
   m) For each $v \in V'$ such that $y^\text{MPC}_v > (1 - 2\epsilon)w(v)$: freeze $v$ and freeze all its edges
4. Directly simulate $\log_{1/\epsilon} (\log^{20}(n))$ iterations of centralized algorithm in one machine
5. Output the set of frozen vertices as a vertex cover
Chapter 5

Analysis

In this chapter we provide an analysis of Algorithm 4, the MPC simulation. The analysis is split into four major parts. First we derive some useful properties for the active degrees, then in the next section we address the memory constraints for the machines. Section 5.3 is devoted to deriving the round complexity. Finally we turn the attention to the approximation ratio in section 5.4.

5.1 Degree Properties

As described earlier, we direct the edges at the beginning of a phase. Recall that the edge \( e = (u, v) \) is directed outward from the endpoint with the lower residual weight over degree ratio, that is from \( \arg \min_{u,v} \left\{ \frac{w'(u)}{d(u)}, \frac{w'(v)}{d(v)} \right\} \).

Where for each vertex \( w'(v) = (1 - 2\epsilon)w(v) - \sum_{e \in v, \text{frozen}} x_e \) and \( d(v) \) is the degree with respect to alive (not frozen) edges. We denote with \( d^\text{out}_A(v) \) the number of active edges that are directed outward from \( v \). A first observation to make is that the active out-degree decreases significantly over a phase.

\[ \text{Observation 5.1 (active out degree)} \quad \text{After a phase ends, we have that for any vertex } v \in V^{\text{active}}: \]

\[ d^\text{out}_A(v) \leq d_{\text{init}}(v)(1 - \epsilon)^I, \]

where \( d_{\text{init}}(v) \) is the degree of alive edges of \( v \) at the start of that phase.

\[ \text{Proof} \quad \text{Assume towards a contradiction that there exists a phase and a vertex } v \text{ such that after the phase ends } d^\text{out}_A(v) > d_{\text{init}}(v)(1 - \epsilon)^I. \quad \text{The weight of} \]

25
5. Analysis

active out-edges is at that point $\frac{w'(v)}{d_{\text{init}}(v)(1 - \epsilon)}$. Therefore we have that

$$y_0^{\text{MPC}} \geq \sum_{e \ni v, \text{frozen}} x_e + d_{\text{init}}^\text{out}(v) \cdot \frac{w'(v)}{d_{\text{init}}(v)(1 - \epsilon)^l}$$

$$> \sum_{e \ni v, \text{frozen}} x_e + d_{\text{init}}(v)(1 - \epsilon)^l \frac{w'(v)}{d_{\text{init}}(v)(1 - \epsilon)^l}$$

$$\geq \sum_{e \ni v, \text{frozen}} x_e + w'(v)$$

$$\geq (1 - 2\epsilon)w(v),$$

meaning that $v$ would have been frozen on line 3m, hence there is no such vertex.

Using Observation 5.1 we show that the average degree is decreasing over the course of a phase. This result will be useful to prove bounds on the memory consumption and round complexity.

Lemma 5.2 Consider one MPC phase with an initial average degree $d$. After the phase ends the average degree is at most $2d(1 - \epsilon)^l$, which is a polynomial decrease in $d$, with $l = \log \frac{m}{\log(5)}$.

Proof As before, call the degree with respect to alive edges of a vertex $v$ at the start of the phase $d_{\text{init}}(v)$. Recall that at the start of the MPC phase the vertices with a low degree are put on hold, meaning that their edges are not increasing in weight and cannot become frozen. This number of sleeping edges is upper bounded by $nd^{0.95}$. For the active vertices we use observation 5.1. Then the upper bound for the total amount of alive edges (active and sleeping) after the phase ends is:

$$\sum_{v \in V'} d_{\text{init}}^\text{out}(v) + n \cdot d^{0.95} \leq \sum_{v \in V'} d_{\text{init}}(v)(1 - \epsilon)^l + nd^{0.95}$$

$$\leq nd(1 - \epsilon)^l + nd^{0.95} \leq 2nd(1 - \epsilon)^l$$

Where the last inequality follows from the fact that $m = \sqrt{d}$ and thus $l = \frac{\log d}{20\log 5}$. Therefore we have $(1 - \epsilon)^l \geq d^{-0.05}$. This implies that the new average degree is at most $2d(1 - \epsilon)^l$. □

5.2 Memory Constraint

To simulate one phase, one machine needs to store the induced sub-graph it receives together with one weight per edge, one weight per vertex and the random thresholds for each vertex. Since these thresholds can be sampled on the fly, this means that there is only one threshold per vertex and thus
the necessary memory is proportional to the size of the induced sub-graph. In the following lemma we show that one machine receives at most $O(n)$ edges with high probability.

**Lemma 5.3** Look at one phase and a machine $i$ and let $G$ and $V_i$ be as defined on lines 3d and 3g of algorithm 4. Then we have with high probability $|E(G'[V_i])| \in O(n)$.

**Proof** From lemma 5.2 we know that at the start of a phase the average degree is upper bounded by $d$. First we provide an upper bound on the local degree of any vertex $v$. The local degree of $v$ is denoted by $d_{\text{local}}(v)$ and indicates the amount of neighbors of $v$ that landed on the same machine. Recall that the vertices that are partitioned all satisfy $d(v) \geq d^{0.95}$. In the following we assume that all the neighbours of $v$ are also in $V_{\text{active}}$, because if this were not the case, the local degree of $v$ would only be smaller and hence lowering the amount of induced edges. Since the vertices are partitioned uniformly at random, we have for the expected local degree of $v$:

$$
\mu_v := \mathbb{E}[d_{\text{local}}(v)] = \frac{d(v)}{\sqrt{d}} \geq \frac{d^{0.95}}{d^{0.5}} = d^{0.45}
$$

Observe that $d_{\text{local}}(v)$ is essentially a sum of independent indicator random variables, that indicate whether a neighbour of $v$ lands in the same machine as $v$. Therefore we can apply Chernoff bounds to achieve that

$$
P[d_{\text{local}}(v) - \mu_v \geq 10\mu_v] \leq \exp\left(-\frac{10\mu_v}{3}\right) \leq \exp\left(-\frac{10 \cdot d^{0.45}}{3}\right).
$$

By the definition of the algorithm we have in each phase that $d > \log^{20} n$, which implies that

$$
d_{\text{local}}(v) \leq O\left(\frac{d(v)}{\sqrt{d}}\right) \quad (5.1)
$$

with high probability.

So far we have established an upper-bound on the local degree for each vertex, that holds with high probability.

Next we look at one machine $i$ and argue that the amount of vertices with a certain degree that lands in this machine is tightly concentrated. To that extend we split the vertices of $V_{\text{active}}$ in degree classes as follows. For $j \in [1, \cdots, O(\log n)]$, let $V_j$ be the vertices with a degree in the interval $[2^{j-1}d^{0.95}, 2^jd^{0.95}]$. With $V_j^i$ we indicate the vertices of class $j$ that land in machine $i$. Similarly as before we can easily calculate the expected size of a degree class on machine $i$ because the vertices are partitioned uniformly at random: $\mu_j := \mathbb{E}[|V_j^i|] = \frac{|V_j|}{\sqrt{d}}$. Based on the size of this expectation we distinguish between two types of degree classes.
5. Analysis

Case 1: \( \mu_j < \log n \)
We can apply Chernoff bounds to achieve the following:

\[
P[|V_i| - \mu_j \geq 10 \log n \mu_j] \leq \exp \left(- \frac{10 \log n}{3 \mu_j} \mu_j \right) \leq \exp \left(- 10 \log n / 3 \right),
\]

which is high probability. This means that when combining all the small degree classes together there will be at most \( O(\log n \cdot \log n) \) vertices in them w.h.p. Moreover, for each of these vertices we have that their degree is upper bounded by \( O\left(\frac{n}{\sqrt{d}}\right) \leq O\left(\frac{n}{\log^2 n}\right) \) w.h.p. Therefore using a union bound over all these vertices, there are at most \( O(\log^2 n \cdot \frac{n}{\log^2 n}) = O\left(\frac{n}{\log^2 n}\right) \) local edges from the small degree classes with high probability.

Case 2: \( \mu_j \geq \log n \)
We can again apply Chernoff bounds to achieve:

\[
P[|V_i| - \mu_j \geq 10 \mu_j / 3] \leq \exp \left(- 10 \mu_j / 3 \right)
\]

which is high probability. Using a union bound over all vertices from degree class \( j \) and with the degree bound 5.1 we have that the number of local edges incident to class \( j \) is w.h.p. at most \( O\left(2 \frac{d^{0.95}}{\sqrt{d}} |V_i|\right) \). Combining all the large weight classes we have that the total amount of local edges is with high probability at most

\[
O\left(\sum_j 2 \frac{d^{0.95}}{\sqrt{d}} |V_j|\right) = O\left(2 \sum_{v \in V} \frac{d(v)}{d} \right) = O\left(\frac{2nd}{d} \right) = O(n)
\]

Finally we can sum up the edges incident to the small degree classes and the edges incident to the large degree classes to conclude that with high probability

\[
|E(G'[V_i]| \in O(n), \quad \square
\]

Using this lemma and the observation that there are only \( O(\sqrt{d}) \in O(n) \) machines per phase and \( O(\log \log d) \in O(\log \log n) \) phases in total, we have that during the course of the whole algorithm with high probability no machine will use more than \( O(v) \) memory.

5.3 Round Complexity

Lemma 5.4 MPC simulation takes at most \( O(\log \log d) \) rounds, where \( d \) is the initial average degree.

Proof Call the average degree after \( i \) phases \( d_i \). Using lemma 5.2 and \( I = \frac{\log d_i}{20 \log 5} \) we have that

\[
d_{i+1} = 2d_i(1 - \varepsilon)^I = d_i^{1 + \frac{1}{\log 5} \frac{\log((1/(1-\varepsilon)))}{20 \log 5}}.
\]
For convenience we define \( g = \frac{\log(1/(1-\epsilon))}{20 \log 5} - \frac{1}{\log d_i} \). Note that we have the conditions: \( \frac{1}{\log n} < \epsilon < 1/2 \) and \( d > \log^{20} n \) which imply that \( \gamma \in (0,1) \).

Now solving the recursion relation yields

\[
d_i = d_{(i-1)}^{1-\gamma} = \cdots = d^{(1-\gamma)^i}
\]

The simulation is executed for \( i \) iterations where \( i \) is the smallest integer such that \( d_i \leq \log^{20} n \). This implies \( d^{(1-\gamma)^i} \leq \log^{20} n \) or after some calculations:

\[
i \in \mathcal{O}\left( \frac{\log \log d}{\log(1/(1-\gamma))} \right) \in \mathcal{O}(\log \log d) \text{ for } \epsilon \in \left( \frac{1}{\log n}, \frac{1}{2} \right) \text{ constant.}
\]

### 5.4 Approximation Ratio

Finally we turn our attention to the approximation guarantee. The goal is to show that the fractional matching and vertex cover at the end of the MPC simulation are feasible and close together. Then we can apply LP-duality to derive a bound with respect to the optimal solution. In the analysis of the centralized algorithm we could directly link the weight of incident edges to the weight of a vertex in the cover. Unfortunately this is not possible for the MPC simulation. Consider, for example, a vertex \( u \) that gets frozen in the middle of an MPC phase. At that point we have that \( \tilde{y}_v \geq T_v w(v) \). Since \( \tilde{y}_v \) is an estimate, we might have that the sum of the actual incident edges is lower than the threshold: \( y_v < T_v w(v) \). Or even more problematically, we might have that \( y_v > w(v) \), which means that the vertex \( v \), together with its incident edges, is taken out of the consideration and added directly to the resulting cover. Both these situations occur when a vertex turns bad, i.e. the vertex is frozen too soon or too late in the MPC simulation when compared to its behaviour in the centralized algorithm. Luckily not all is lost, because we can show that there is only a limited amount of vertices that become bad during the execution of the whole algorithm. Therefore the main goal in this section will be to show that there is only a small fraction of vertices that turn bad. Once this is done, we can account for these vertices when comparing the weight of the fractional matching with the weight of the vertex cover and still use LP duality.

The main tool we will use to bound the number of bad vertices is that the estimated values \( \tilde{y}_v \) and \( y_v^{\text{MPC}} \) stay close to the actual values \( y_v \) during one phase of MPC simulation. Once we establish that we can use Lemma 5.5 below to bound the number of bad vertices in a particular iteration.

**Lemma 5.5** Consider iteration \( t \) of a phase. Let \( |y_{v,t} - \tilde{y}_{v,t}| \leq \sigma w(v) \) for all vertices that are active in both the centralized algorithm and MPC-simulation. Then, \( v \) becomes bad in iteration \( t \) with probability \( \sigma / \epsilon \) and independently of other vertices.
5. Analysis

Proof For a vertex to become bad, the estimate $\tilde{y}_{v,t}$ has to be on the other side of the threshold as $y_{v,t}$. Call the effective threshold $T_{v,t} = y_{v,t}w(v)$. Notice that when $|\tilde{y}_{v,t} - T_{v,t}| > \sigma w(v)$ it is not possible for vertex $v$ to become bad. Therefore, only if $T_{v,t}$ falls in the interval of size $2\sigma w(v)$ around $\tilde{y}_{v,t}$, $v$ might become bad. $T_{v,t}$ is chosen uniformly at random from an interval with size $2\varepsilon$, this is equivalent to pick $T_{v,t}$ uniformly at random from the scaled interval with size $2\varepsilon w(v)$. Therefore the possibility of $v$ becoming bad in iteration $t$ is upper-bounded by $2\sigma w(v)/(2\varepsilon w(v)) = \sigma/\varepsilon$.

Instead of directly calculating a bound on $|y_v - y_{v}^{MPC}|$, we use a slightly different notion introduced below.

Definition 5.6 weight-difference Let $x_{e,t}$ and $x_{e,t}^{MPC}$ be the weight of edge $e$ in iteration $t$ of algorithm 3 and 4 respectively.

$$\text{diff}(v, t) := \sum_{e \in N(v)} |x_{e,t} - x_{e,t}^{MPC}|$$

Note that $|y_v - y_v^{MPC}| \leq \text{diff}(v, t)$. Before going into the rest of the proofs we would like to spend some time on the similarities and differences to the analysis from Ghaffari et al. [12]. The analysis presented here is heavily based on their work. Nevertheless there are some notable differences. First we explicitly restrict the vertices that are partitioned to the high degree vertices. In the original work of Ghaffari et al. this restriction is not necessary because vertices that have a degree below approximately $D_{0.9}$, where $D$ is the maximum active degree at the start of a phase, can not become frozen in the current phase. Therefore it is not a problem when the estimate $\tilde{y}_v$ can’t be calculated well for such vertices. However in the algorithm presented here such low degree vertices could become frozen in the current iteration, because the weight of their incident edges might be high. Therefore since we can not estimate a small number of incident edges well after the partitioning, these vertices are sleeping during this phase. Secondly, the estimate for high degree vertices might be influenced by these sleeping edges. Thankfully this is a change in the positive direction, because the sleeping edges are counted in the fixed (and hence deterministic) part of the estimate $y_v^{fixed}$. Concretely this means that the variance of the estimate $\tilde{y}_v$ becomes smaller since a larger fraction is deterministic. Finally since there is no restriction on the range of the weights one vertex being good or bad can have a dramatic difference in the final approximation guarantee. Consequently we can only provide an expected approximation ratio for running this algorithm once. As we will see later this poses no problem because the confidence can be boosted by running multiple versions in parallel.

Assumptions In the following we assume that at the beginning of each phase $x$ and $x^{MPC}$ coincide. Since $x^{MPC}$ is always a valid edge packing at the begin-
5.4. Approximation Ratio

ning of a phase also the Centralized algorithm maintains a valid edge packing. Furthermore, the thresholds chosen by the vertices $T_{v,p}$ are assumed to be the same in algorithm 3 and 4. For the constant $\epsilon$ we assume it lies in the interval $[\frac{1}{\log \gamma}, \frac{1}{\gamma}]$.

The following lemma will be useful in proving that $|y_v - \tilde{y}_v|$ and $\text{diff}(v)$ stay small during one phase.

**Lemma 5.7** (maximum active edge weight) Consider any active edge $e = (u, v)$ and assume w.l.o.g. that the edge weight at the start of a phase is $x_v = \frac{w'(u)}{d(u)}$. Then we have that $x_v^{\text{MPC}} \leq \frac{w(u)}{m^{1.8}}$ throughout the whole phase.

**Proof** Since $e$ is active we have that $u, v \in V_{\text{active}}$ and hence $d(u) \geq d^{0.95}$. Using this, we have that

$$x_v^{\text{MPC}} \leq \frac{w'(u)}{d(u)} \left(\frac{1}{1 - \epsilon}\right)^l \leq \frac{w'(u)}{d^{0.95}} m^{0.1} \leq \frac{w(u)}{m^{1.8}}$$

Where we used that $(1/(1 - \epsilon))^l \leq m^{0.1}$ and $m^2 = d$.

To prove that $|y_v - \tilde{y}_v|$ and $\text{diff}(v)$ stay small during one phase we will use and induction argument. Lemma 5.8 provides the base case for the very first iteration of a phase and Lemma 5.9 handles the induction step. Then in Lemma 5.10 these pieces are put together.

**Lemma 5.8** At the first iteration $t^*$ of a phase we have that for an active vertex $v$ w.h.p.:

$$|y_{v,t^*} - \tilde{y}_{v,t^*}| < m^{-0.2} \cdot w(v)$$

Moreover it holds that $\text{diff}(v, t^*) = 0$ with certainty.

**Proof** First note that we have $\text{diff}(v, t^*) = 0$ trivially due to the assumptions we made earlier.

At the start of a phase we have that $\tilde{y}_{v,t^*} = y_{v,t^*}^{\text{fixed}} + m \cdot \sum_{u \in N_A^{\text{local}}(v, t^*)} X_{(u,v),t^*}$ and $y_{v,t^*} = \tilde{y}_{v,t^*} + \sum_{u \in N_A^{\text{central}}(v, t^*)} X_{(u,v),t^*}$. Where $N_A^{\text{local}}(v, t^*)$ denotes the active neighbors of $v$ that occur in the same machine in iteration $t^*$, and $N_A^{\text{central}}(v, t^*)$ are the active neighbors in the complete graph in iteration $t^*$. Since it is the start of the phase the only difference between $y_v$ and $\tilde{y}_v$ comes from the partitioning of the vertices. We denote the current weight of active out-edges of a vertex $v$ by $w_{out}(v)$ and rewrite the expression for $\tilde{y}_{v,t^*}$:

$$\tilde{y}_{v,t^*} = y_{v,t^*}^{\text{fixed}} + w_{out}(v) \cdot m \sum_{u \in N_A^{\text{local}}(v, t^*)} \frac{X_{(u,v),t^*}}{w_{out}(v)}.$$

Denote the random variable $X := m \cdot \sum_{u \in N_A^{\text{central}}(v, t^*)} \frac{X_{(u,v),t^*}}{w_{out}(v)}$. Since the vertices are partitioned uniformly at random we have that

$$w_{out}(v) \cdot \mathbb{E}[X] = \sum_{u \in N_A^{\text{central}}(v, t^*)} X_{(u,v),t^*}.$$
5. Analysis

Now we will upper-bound this random variable $X$. Notice that $X$ is a scaled sum of independent random variables, all of which fall in the interval $[0, 1]$, therefore we can apply Chernoff bounds on $X$.

Case 1: $\mathbb{E}[X] \leq m^{1.6}$.

$$P[|X - \mathbb{E}[X]| \geq \frac{m^{1.6}}{\mathbb{E}[X]} \mathbb{E}[X]] \leq 2 \exp\left(-\frac{m^{3.2}}{3\mathbb{E}[X]m}\right) \leq 2 \exp\left(-m^{0.6}/3\right)$$

Which is high probability given that $m \geq \log^{10} n$ during every phase.

Case 2: $\mathbb{E}[X] > m^{1.6}$.

$$P[|X - \mathbb{E}[X]| \geq m^{-0.2}\mathbb{E}[X]] \leq 2 \exp\left(-m^{-0.4}\mathbb{E}[X]/3m\right) \leq 2 \exp\left(-m^{0.2}/3\right)$$

Which is again high probability.

Combining these cases we have now the following:

$$X = \mathbb{E}[X] \pm \max\{m^{-0.2}\sum_{u \in N_{central}(v,t')} \frac{X(u,v),t'}{w_{out}(v)} \cdot m^{1.6}\}$$

Multiplying this whole expression with $w_{out}(v)$ and using lemma 5.7 delivers w.h.p.:

$$w_{out}(v) \cdot X = \sum_{u \in N_{central}(v,t')} x(u,v),t' \pm m^{-0.2}w(v)$$

Therefore we have also that:

$$|y_{v,t} - \tilde{y}_{v,t}| < m^{-0.2} \cdot w(v)$$

with high probability.

Lemma 5.9 If we have in iteration $t - 1$ for a vertex $v$, active in both the centralized algorithm and MPC simulation, that $|y_{v,t-1} - \tilde{y}_{v,t-1}| < \sigma \cdot w(v)$ and $\text{diff}(v, t - 1) < \sigma \cdot w(v)$, then in iteration $t$ we have w.h.p.:

- $|y_{v,t} - \tilde{y}_{v,t}| < 4(\sigma + \epsilon m^{-0.2})w(v)$
- $\text{diff}(v, t) < 4(\sigma + \epsilon m^{-0.2})w(v)$

Proof The change in the difference between the estimated values for a vertex and the true value can be attributed to three factors. First there is the contribution of the neighbors that were already bad in the previous iteration. Then there are neighbors that might turn bad in the current iteration and finally there is the influence of the random partitioning of the vertices. Let us discuss these three factors separately.

Old bad vertices: There might be neighbors that turned bad before iteration $t$. However, we know that their weight is bounded by $\sigma w(v)$ in iteration $t - 1$. Therefore their weight is bounded by $(1 + \epsilon)\sigma w(v)$ in iteration $t$.

New bad vertices: Additionally, there might be neighbors that turn bad in
5.4. Approximation Ratio

the current iteration. Therefore in this part we bound the weight of incident edges to neighbors that turn bad in the current iteration. For a neighbor to turn bad in iteration $t$, it had to be active in iteration $t - 1$ in both the centralized algorithm 3 and the MPC simulation 4. More formally this indicates that such a neighbor belongs to the set $N_{A}^{\text{central}}(v, t - 1) \cap N_{A}^{\text{local}}(v, t - 1)$, therefore let us first upper-bound the weight of the edges incident to this set. For convenience, define:

$$n_{v,t-1}^{\text{local}} := \sum_{u \in N_{A}^{\text{central}}(v,t-1) \cap N_{A}^{\text{local}}(v,t-1)} \frac{x_{(u,v),t-1}}{w_{\text{out}}(v,t-1)}$$

Where $N_{A}^{\text{local}}(v,t-1)$ is the set of all local neighbors including the ones that are frozen by iteration $t - 1$, and as before $w_{\text{out}}(v,t-1)$ is the weight of outgoing edges from $v$ in iteration $t - 1$. The value $n_{v,t-1}^{\text{local}}$ can be thought of as a sort of normalized incident weight, because multiplying it by the active out weight gives the actual incident weight. Now by lemma 5.5 we have that the weight of new bad incident edges is in expectation at most $n_{v,t-1}^{\text{local}}w_{\text{out}}(v,t-1)\sigma/\varepsilon$. Notice that this is only an upper-bound because in the calculation of $n_{v,t-1}^{\text{local}}$ we consider all vertices in the set $N_{A}^{\text{central}}(v,t-1) \cap N_{A}^{\text{local}}(v,t-1)$. First we will show an upper-bound on $n_{v,t-1}^{\text{local}}$ that holds with high probability. Note that the set $N_{A}^{\text{central}}(v,t-1)$ is deterministic and is defined independently of the MPC algorithm.

We proceed in a similar fashion as in the proof of lemma 5.8 and omit some of the calculations. When

$$\sum_{u \in N_{A}^{\text{central}}(v,t-1) \cap N_{A}^{\text{local}}(v,t-1)} \frac{x_{(u,v),t-1}}{w_{\text{out}}(v,t-1)} \geq m^{1.6},$$

we have w.h.p. that

$$n_{v,t-1}^{\text{local}} \leq (1 + m^{-0.2}) \sum_{u \in N_{A}^{\text{central}}(v,t-1) \cap N_{A}^{\text{local}}(v,t-1)} \frac{x_{(u,v),t-1}}{w_{\text{out}}(v,t-1)}/m.$$  

On the other hand, if

$$\sum_{u \in N_{A}^{\text{central}}(v,t-1) \cap N_{A}^{\text{local}}(v,t-1)} \frac{x_{(u,v),t-1}}{w_{\text{out}}(v,t-1)} < m^{1.6},$$

then w.h.p. $n_{v,t-1}^{\text{local}} \leq 2m^{0.6}$. Therefore we have that w.h.p.

$$m \cdot n_{v,t-1}^{\text{local}} \leq \max \left\{ (1 + m^{-0.2}) \sum_{u \in N_{A}^{\text{central}}(v,t-1) \cap N_{A}^{\text{local}}(v,t-1)} \frac{x_{(u,v),t-1}}{w_{\text{out}}(v,t-1)} \cdot 2m^{1.6} \right\} =: \gamma$$

We can apply similar reasoning to $n_{v,t-1}^{\text{local}}\sigma/\varepsilon$, i.e. considering the case $n_{v,t-1}^{\text{local}}\sigma/\varepsilon \geq m^{0.6}$ and $n_{v,t-1}^{\text{local}}\sigma/\varepsilon \leq m^{0.6}$, to obtain that the normalized weight of new bad incident edges is upper-bounded by $\max\{1 + \ldots\}$.
5. Analysis

\(m^{-0.2})^{10} \sigma \leq \sigma/\epsilon, 2m^{1.6}\}. Now putting it all together we have that the weight coming from new bad vertices is at most:

\[\sigma_2 := \epsilon w_{out}(v, t - 1) \cdot \max \{(1 + m^{-0.2})\sigma/\epsilon, 2m^{1.6}\}.

Applying lemma 5.7 and the fact that

\[w_{out}(v, t - 1) \sum_{u \in N^c_A(v, t - 1)} \frac{x_{(u, v), t - 1}}{w_{out}(v, t - 1)} \leq w(v)\]

delivers \(\sigma_2 \leq 2(\sigma + \epsilon m^{-0.2})w(v)\). Note that the same arguments apply to \(\text{diff}(v, t)\) because the bad vertices that can influence \(\text{diff}(v, t)\) also come from the set \(N^c_A(v, t - 1)\). Therefore the weight of new bad vertices affects \(\text{diff}(v, t)\) by at most \(\sigma_2\) w.h.p.

**Effect of random partitioning:** Finally there is an effect of random partitioning the vertices of \(\tilde{y}_{v, t}\). Similarly as before, the normalized weight of local incident edges deviates from its expectation

\[\sum_{u \in N^c_A(v, t - 1)} \frac{x_{(u, v), t - 1}}{w_{out}(v, t - 1)} / m\]

by at most

\[\eta := \max \{m^{-0.2} \sum_{u \in N^c_A(v, t - 1)} \frac{x_{(u, v), t - 1}}{w_{out}(v, t - 1)} / m^{1.6}) / m\]

w.h.p. The total weight of these edges scaled by \(m\) is at most \(\epsilon w_{t - 1} m \eta \leq \epsilon m^{-0.2} w(v)\).

**Finalizing** Now we can put the contributions of these influences together to derive that if

\[|y_{v, t - 1} - \tilde{y}_{v, t - 1}| < \sigma w(v)\]

and

\[\text{diff}(v, t - 1) < \sigma w(v),\]

we have that

\[|y_{v, t} - \tilde{y}_{v, t}| \leq (1 + \epsilon)w(v)\sigma + 2(\sigma + \epsilon m^{-0.2})w(v) + \epsilon m^{-0.2}w(v) \leq 4(\sigma + \epsilon m^{-0.2})w(v)\]

and

\[\text{diff}(v, t) \leq (1 + \epsilon)w(v)\sigma + 2(\sigma + \epsilon m^{-0.2})w(v) \leq 4(\sigma + \epsilon m^{-0.2})w(v)\]

\[\square\]

**Lemma 5.10** Let \(v\) be an active vertex in iteration \(t - 1\) in both the centralized algorithm and MPC-simulation. If a phase consists of at most \(I = \log m / 10 \log 5\) iterations, then it holds that \(|y_{v, t} - \tilde{y}_{v, t}| \leq m^{-0.1}\) and \(\text{diff}(v, t) \leq m^{-0.1}\) w.h.p.
5.4. Approximation Ratio

Proof Let iteration $t^*$ be the first iteration of the respective phase. Then combining lemma 5.8 and lemma 5.9 we have for any iteration $t^* \leq t \leq t^* + I$ in which $v$ is not bad:

$$|y_{v,t} - \tilde{y}_{v,t}| \leq 5^I m^{-0.2} \leq m^{-0.1},$$

and

$$\text{diff}(v,t) \leq 5^I m^{-0.2} \leq m^{-0.1}. \quad (5.2)$$

Now everything is in place to state and proof the final result. Note that the following theorem only states that in expectation a desired approximation ratio is achieved. The success probability can be boosted by running $O(polylog(n))$ versions of algorithm 4 in parallel. This is possible since Algorithm 4 uses with high probability only $O(n)$ memory per machine while we have $\tilde{O}(n)$ memory available.

Lemma 5.11 Algorithm 4 outputs in expectation a $(2 + 50\varepsilon)$ approximate minimum weight vertex cover.

Proof Call $\tilde{C}$ the vertex cover returned by algorithm 4. Now we will relate the weight of this vertex cover, $w(\tilde{C})$ to the weight of the fractional matching $x^{\text{MPC}}$. Instead of doing this directly we will compare the result with the result of the centralized algorithm. Over the course of the algorithm there are three kinds of vertices: vertices that always make the same decision in both the centralized algorithm and MPC-simulation (1), vertices that are frozen first in MPC simulation (2) and vertices that are frozen first in the centralized algorithm (3).

Vertices from categories (1) and (2): we have from lemma 5.10 that in the iteration $v$ became frozen: $y_{v}^{\text{MPC}} \geq (1 - 5\varepsilon)w(v)$. However, some additional care has to be taken, since a neighbour $u$ might later become too heavy ($y_{u}^{\text{MPC}} \geq w(u)$) which means it is removed together with its edges from the graph. We will later take this effect into account.

Vertices from category (3): denote the set of these vertices by $\tilde{C}^{\text{late}}$. We can bound their total expected weight as a function of the weight of the vertex cover returned by the centralized algorithm. First we will bound the total weight of all bad vertices and then the weight of category 3 vertices is trivially also upper-bounded by this. To achieve this, note that only vertices that are close to the threshold of joining the cover can turn bad. Indicate with $C_t$ the vertices that join the cover $C$ in iteration $t$. Recall that this is a deterministic set. From lemma 5.5 we know that every vertex in $C_t$ turns bad independently with probability at most $m^{-0.1}/\varepsilon$. Hence the expected weight of vertices that turn bad in iteration $t$ is $w(C_t)m^{-0.1}/\varepsilon$. So the weight of all bad vertices is in expectation $w(C)m^{-0.1}/\varepsilon$. With the assumption that
5. Analysis

$m^{-0.1} \leq \epsilon^2$, we have now the following:

$$w(\tilde{C}) \geq \left(1 - \frac{m^{-0.1}}{\epsilon}\right)w(C) \geq (1 - \epsilon)w(C),$$

and also

$$w(\tilde{C}_{late}) \leq \epsilon w(C) \leq \frac{\epsilon}{1 - \epsilon} w(\tilde{C}).$$

Finally we have to bound the weight of the vertices that became too heavy $w(\tilde{C}_{heavy})$. An important observation to make is that these vertices were with high probability already bad. Because as long as $v$ is not bad, we have by lemma 5.10 that $y_v^{MPC} \leq y_v + m^{-0.1}w(v) \leq (1 - \epsilon)w(v) + m^{-0.1}w(v) < w(v)$. This implies that also the weight of too heavy vertices is upper bounded by $w(\tilde{C}_{late})$. Now with $W_M$ the weight of the fractional matching we have:

$$\frac{W_M + w(\tilde{C}_{heavy})}{\tilde{C} - w(\tilde{C}_{late})} \geq \frac{1 - 5\epsilon}{2}.$$  

Rewriting this delivers

$$w(\tilde{C}) \leq 2(1 + 50\epsilon)W_M.$$  

Together with LP-duality, this completes the proof. \qed
Let us go back to the initial goal of this thesis: finding an algorithm to solve the weighted minimum vertex cover that runs as fast as the state-of-the-art algorithms for unweighted vertex cover in MPC. Moreover this algorithm should achieve a similar approximation ratio. After a study of the MPC model with the current fastest algorithms for related problems and an examination of the weighted vertex cover problem, we set out to find such an algorithm. The first step was to find a suitable centralized/LOCAL algorithm that allows to apply round compression. It is helpful in that situation to find out what the necessary requirements are to apply this technique. We hope that the discussion in chapter 4 can help in understanding these requirements. Then once a centralized algorithm is established, the bulk of the work lies in thoroughly analysing the MPC simulation.

A natural next step would be to consider the hypergraph case. However, there is one notable difficulty with the round compression technique as used in this thesis. We can think of doing the same procedure: partitioning the set of vertices among some machines and then simulating an appropriate centralized algorithm. One key property to allow for round compression is that we have to be able to estimate the weight of incident edges well. For regular graphs this is a sum of independent random variables and hence Chernoff bounds can be used to give tight concentration bounds. For general hypergraphs this is not the case however. Since two incident edges might share another vertex, these random variables are no longer independent. When restricting to linear hypergraphs, where every pair of edges has at most one vertex in common, this issue does not arise and we believe it is possible to extend our approach to handle that case. When considering general hypergraphs, new tools are needed to analyse what happens after partitioning the vertices. Perhaps it is not even possible at all to adapt our algorithm successfully and a completely new approach might be necessary.
Bibliography

[1] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. *ArXiv*, abs/1401.0042, 2013.

[2] Sepehr Assadi, Mohammad Hossein Bateni, Alexander Bernstein, Vahab S. Mirrokni, and Clifford Stein. Coresets meet edcs: Algorithms for matching and vertex cover on massive graphs. In *SODA*, 2017.

[3] Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A distributed $(2 + \varepsilon)$-approximation for vertex cover in $O(\log(\delta)/\varepsilon \log \log(\delta))$ rounds. 2016.

[4] Reuven Bar-Yehuda and Shimon Even. A local-ratio theorem for approximating the weighted vertex cover problem. In *WG*, 1983.

[5] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *ArXiv*, abs/1306.5972, 2013.

[6] Soheil Behnezhad, Mahsa Derakhshan, and Mohammad Taghi Hajiaghayi. Brief announcement: Semi-mapreduce meets congested clique. *ArXiv*, abs/1802.10297, 2018.

[7] Dimitris Bertsimas and John N. Tsitsiklis. Introduction to linear optimization. 1997.

[8] Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. *ArXiv*, abs/1707.03478, 2018.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51:107–113, 2004.
[10] Devdatt P. Dubhashi and Alessandro Panconesi. Concentration of measure for the analysis of randomized algorithms. 2009.

[11] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On distributing symmetric streaming computations. In *ACM Trans. Algorithms*, 2008.

[12] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *PODC*, 2018.

[13] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. *ArXiv*, abs/1807.06251, 2018.

[14] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *ISAAC*, 2011.

[15] Nicholas J. A. Harvey, Christopher Liaw, and Paul Liu. Greedy and local ratio algorithms in the mapreduce model. *ArXiv*, abs/1806.06421, 2018.

[16] Dorit S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Mathematics*, 6:243–254, 1983.

[17] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22:463–468, 1975.

[18] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[19] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, 2010.

[20] Richard M. Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming*, 1972.

[21] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-epsilon. *J. Comput. Syst. Sci.*, 74:335–349, 2003.

[22] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In *PODC*, 2004.
[23] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In SPAA, 2011.

[24] Nathan Linial. Locality in distributed graph algorithms. SIAM J. Comput., 21:193–201, 1992.

[25] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $o(\log \log n)$ communication rounds. SIAM J. Comput., 35:120–131, 2005.

[26] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[27] George L. Nemhauser and Leslie E. Trotter. Vertex packings: Structural properties and algorithms. Mathematical Programming, 8:232–248, 1975.

[28] Sartaj Sahni and Teofilo F. Gonzalez. P-complete approximation problems. J. ACM, 23:555–565, 1976.

[29] Leslie G. Valiant. A bridging model for parallel computation. Commun. ACM, 33:103–111, 1990.

[30] Tom White. Hadoop: The definitive guide. 2009.

[31] James C. Wyllie. The complexity of parallel computations. 1979.

[32] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In HotCloud, 2010.
Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A MASSIVELY PARALLEL ALGORITHM FOR MINIMUM WEIGHT VERTEX COVER

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):
NILIS

First name(s):
DAAN

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zürich, 30/09/2019

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.