Sublinear Algorithms for Processing Massive Datasets

Hamed Saleh

Acknowledgements

Many results in this proposal are part of joint works with multiple co-authors, namely Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi HajiAghayi, Marina Knittel, Saeed Seddighin, and Xiaouri Sun.

Abstract

Designing algorithms which are sublinear in space is an inevitable scenario when the input data does not fit in the memory of available systems. In recent years, with the abundance of data and the increasing demand for large scale data processing, the size of datasets easily surpasses that of the typical machine's memory. Examples of the problem domain vary from social network graphs to DNA sequences. However, to address any problem subject to this restriction efficiently, we need alternative models of computation as the traditional RAM model is no longer an option. In this manuscript, we study sublinear space algorithms in two computation models, *Massively Parallel Computation* and *Streaming*, that enable us to overcome this challenge.

During the last decade, the *Massively Parallel computation* (MPC) model attracted a considerable amount of attention. The MPC model was originally proposed to provide a theoretical foundation to algorithms implemented on modern large scale data processing frameworks such as MapReduce, Hadoop, and Spark. The key idea behind this model, and also aforementioned frameworks, is to use many machines to compensate for the shortage of space in individual machines. In this model, the data is distributed among a set of machines each with a sublinear memory, and the process is consisted of several rounds. In each round, machines perform an arbitrary amount of computation on their local data independently. At the end of each round, machines can communicate with each other. We study the *Edge Coloring* problem in this model, and we show a constant-round MPC algorithm can produce a proper coloring with almost optimal number of colors. In addition, we come up with truly-sublinear MPC algorithms for multiple variants of the *String Matching* problem in the presence of different wildcards {'?', '+', '*'}.

The *Streaming* model is yet another approach that allows us to process massive data in a sublinear space. The streaming model was introduced earlier in comparison to MPC, and it is a well-known model with a plethora of results already known about it. In contrast to MPC, there is only one machine in the streaming model. The process is consisted of several passes, and the input entries arrive sequentially and one by one in each pass. The machine needs to manage its sublinear space while processing the entries upon arrival. The order by which the entries arrive could be either random or adversarial. We study the *Edge Coloring* problem in both random and adversarial order, and we show there are 1-pass algorithms in the special "W-streaming" model, with different guarantees for the number of colors used in each case. In the W-streaming model, the output is also returned in a streaming fashion since the output of the edge coloring problem has the same size as its input.

Contents

Chapter 1

Introduction

Massively Parallel Computation. The MPC model [15, 43, 53] is a popular abstraction of modern large scale data processing frameworks such as MapReduce, Hadoop, Spark, etc. In this model, there are $\mathcal{M}$ machines, each with a sublinear memory of $\mathcal{S}$ words that all run in parallel. The input, a total of N words, is initially distributed among the machines arbitrarily, which means that $N = O(\mathcal{M} \cdot \mathcal{S})$. In an ideal situation, both the number of machines and the local memory size of each machine is asymptotically smaller than the input size, i.e., $\mathcal{M} = \widetilde{O}(N^x)$ and $\mathcal{S} = \widetilde{O}(N^{1-x})$ for some $0 < x < 1$. However, in many graph problems we cannot always afford strictly sublinear local space, especially because of sparse instances. An interesting choice of space for graph problems is $\mathcal{S} = \widetilde{O}(n)$ as the problem is neither too trivial nor impossible.

Following the initial step, the system proceeds in synchronous rounds wherein the machines can perform any arbitrary local computation on their data and can also send messages to other machines. The messages are then delivered at the start of the next round so long as the total messages sent and received by each machine is $O(\mathcal{S})$ for local machine space $\mathcal{S}$. The main parameters of interest are $\mathcal{S}$ and the round-complexity of the algorithm, i.e., the number of rounds it takes until the algorithm stops. Furthermore, the total available space over all machines should ideally be linear in the input size, i.e., $\mathcal{M} \cdot \mathcal{S} = O(N)$.

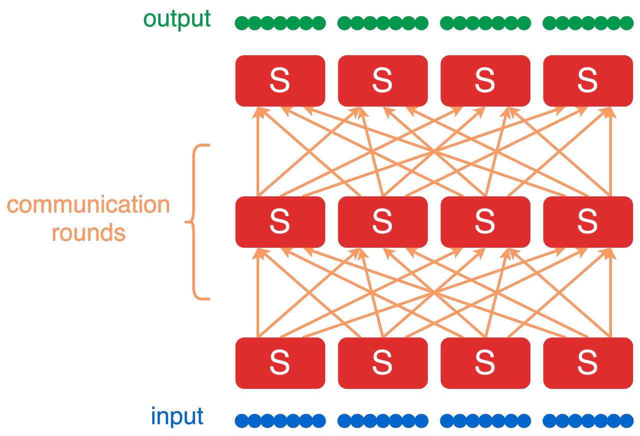
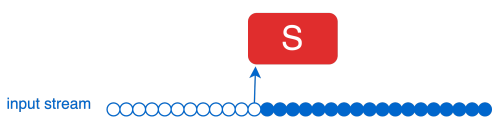
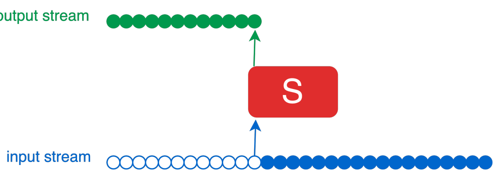


Figure 1.1: An illustration of the data flow in the MPC model.

Streaming. In the standard graph streaming model, the edges of a graph arrive one by one and the algorithm has a space that is much smaller than the total number of edges. A particularly important choice of space is $\widetilde{O}(n)$—which is also known as the *semi-streaming* model [35]—so that the algorithm has enough space to store the vertices but not the edges. For edge coloring, the output is as large as the input, thus, we cannot hope to be able to store the output and report it in bulk at the end. For this, we consider a standard twist on the streaming model where the output is also reported in a streaming fashion. This model is referred to in the literature as the "W-streaming" model [33, 40]. We particularly focus on one-pass algorithms.



(a) Streaming model



(b) W-streaming model

Figure 1.2: A comparison of a regular streaming model, and a W-streaming model. In (a), the output fits inside the memory. Therefore, the machine might store the output and report it in the end. However, the size of output in (b) is much larger than the size of memory. Thus, the machine needs to output respectively as soon as an input entry arrives (or at least output in small buffers).

Edge Coloring. A valid *edge-coloring* of a graph is an assignment of "colors" to its edges such that no two incident edges receive the same color. The goal is to find a proper coloring that uses few colors (Note that the maximum degree, Δ, is a trivial lower bound). In Chapter 2, we revisit this fundamental problem in both models we defined earlier:

- *Massively Parallel Computation.* We give a randomized MPC algorithm that with high probability returns a $\Delta + \widetilde{O}(\Delta^{3/4})$ edge coloring in $O(1)$ rounds using $O(n)$ space per machine and $O(m)$ total space. The space per machine can also be further improved to $n^{1-\Omega(1)}$ if $\Delta = n^{\Omega(1)}$. Our algorithm improves upon a previous result of Harvey *et al.* [45].
- *Graph Streaming.* Since the output of edge-coloring is as large as its input, we consider a standard variant of the streaming model where the output is also reported in a streaming fashion. The main challenge is that the algorithm cannot "remember" all the reported edge colors, yet has to output a proper edge coloring using few colors.

  We give a one-pass $\widetilde{O}(n)$-space streaming algorithm that always returns a valid coloring and uses 5.44Δ colors with high probability if the edges arrive in a random order. For adversarial order streams, we give another one-pass $\widetilde{O}(n)$-space algorithm that requires $O(\Delta^2)$ colors.

String Matching. We study distributed algorithms for string matching problem in presence of wildcard characters. Given a string T (a text), we look for all occurrences of another string P (a pattern) as a substring of string T. Each wildcard character in the pattern matches a specific class of strings based on its type. String matching is one of the most fundamental problems in computer science, especially in the fields of bioinformatics and machine learning. Persistent effort has led to a variety of algorithms for the problem since 1960s.

With rise of big data and the inevitable demand to solve problems on huge data sets, there have been many attempts to adapt classic algorithms into the MPC framework to obtain further efficiency. In Chapter 3, we study the string matching problem using a set of tools translated to MPC model. We consider three types of wildcards in string matching:

- ‘?’ *wildcard*: In this setting, the pattern is allowed to contain special ‘?’ characters or don’t cares that match any character of the text. String matching with don’t cares could be solved by fast convolutions, and we give a constant round MPC algorithm for which by utilizing FFT in a constant number of MPC rounds.
- ‘+’ *wildcard*: ‘+’ wildcard is a special character that allows for arbitrary repetitions of a character. When the pattern contains ‘+’ wildcard characters, our algorithm runs in a constant number of MPC rounds by a reduction from subset matching problem.
- ‘*’ *wildcard*: ‘*’ is a special character that matches with any substring of the text. When ‘*’ is allowed in the pattern, we solve two special cases of the problem in logarithmic rounds.

Chapter 2

Edge Coloring

2.1 Introduction

Given a graph $G(V, E)$, an edge coloring of G is an assignment of "colors" to the edges in E such that no two incident edges receive the same color. The goal is to find an edge coloring that uses few colors. Edge coloring is among the most fundamental graph problems and has been studied in various models of computation, especially in distributed and parallel settings.

Denoting the maximum degree in the graph by Δ, it is easy to see that Δ colors are necessary in any proper edge coloring. On the other hand, Vizing's celebrated theorem asserts that $\Delta + 1$ colors are always sufficient [76]. While determining whether a graph can be Δ colored is NP-hard, a $\Delta + 1$ coloring can be found in polynomial time [7, 38]. These algorithms are, however, highly sequential. As a result, in restricted settings, it is standard to consider more relaxed variants of the problem where more colors are allowed [4, 11, 37, 41, 42, 44, 50, 57, 61, 63, 67].

In this paper, we study edge coloring in large-scale graph settings. Specifically, we focus on the *Massively Parallel Computations* (MPC) model and the *Graph Streaming* model.

2.1.1 Massively Parallel Computation

The Model. The MPC model [15, 43, 53] is a popular abstraction of modern parallel frameworks such as MapReduce, Hadoop, Spark, etc. In this model, there are N machines, each with a space of S words[1] that all run in parallel. The input, which in our case is the edge-set of graph $G(V, E)$, is initially distributed among the machines arbitrarily. Afterwards, the system proceeds in synchronous rounds wherein the machines can perform any arbitrary local computation on their data and can also send messages to other machines. The messages are then delivered at the start of the next round so long as the total messages sent and received by each machine is $O(S)$ for local machine space S. The main parameters of interest are S and the round-complexity of the algorithm, i.e., the number of rounds it takes until the algorithm stops. Furthermore, the total available space over all machines should ideally be linear in the input size, i.e., $S \cdot N = O(|E|)$.

Related Work in MPC. We have seen a plethora of results on graph problems ever since the formalization of MPC. The studied problems include matching and vertex cover [3, 8, 19, 31, 39, 58, 17, 21], maximal independent set [39, 45, 17, 21], vertex coloring [9, 26, 45, 68, 69], as well as graph connectivity and related problems [5, 6, 18, 51, 13]. (This is by no means a complete list of the prior works.)

[1]Throughout the paper, the stated space bounds are in the number of words that each denotes $O(\log n)$ bits.

We have a good understanding of the complexity of vertex coloring in the MPC model, especially if the local space is near linear in n: Assadi et al. [9] gave a remarkable algorithm that using $\widetilde{O}(n)$ space per machine, finds a $(\Delta+1)$ vertex coloring in a constant number of rounds. The algorithm is based on a sparsification idea that reduces the number of edges from m to $O(n \log^2 n)$. But this algorithm alone cannot be used for coloring the edges, even if we consider the more relaxed $(2\Delta-1)$ edge coloring problem which is equivalent to $(\Delta+1)$ vertex coloring on the line graph. The reason is that the line-graph has $O(m)$ vertices where here m is the number of edges in the original graph. Therefore even after the sparisification step, we have $\widetilde{O}(m)$ vertices in the graph which is much larger than the local space available in the machines.

Not much work has been done on the edge coloring problem in the MPC model. The only exception is the algorithm of Harvey *et al.* [45] which roughly works by random partitioning the *edges*, and then coloring each partition in a different machine using a sequential $(\Delta+1)$ edge coloring algorithm. The choice of the number of partitions leads to a trade-off between the number of colors used and the space per machine required. The main shortcoming of this idea, however, is that if one desires a $\Delta + \widetilde{O}(\Delta^{1-\Omega(1)})$ edge coloring, then a strongly super linear local space of $n\Delta^{\Omega(1)}$ is required.

Our main MPC result is the following algorithm which uses a more efficient partitioning. The key difference is that we use a *vertex* partitioning as opposed to the algorithm of Harvey *et al.* which partitions the edges.

Result 1 (Theorem 2.2.1)**.** There exists an MPC algorithm that using $O(n)$ space per machine and $O(m)$ total space, returns a $\Delta + \widetilde{O}(\Delta^{3/4})$ edge coloring in $O(1)$ rounds.

The algorithm exhibits a tradeoff between the space and the number of colors (see Theorem 2.2.1) and can be made more space-efficient as the maximum degree gets larger. For instance, if $\Delta > n^{\epsilon}$ for any constant $\epsilon > 0$, it requires a strictly sublinear space of $n^{1-\Omega(1)}$ to return a $\Delta + o(\Delta)$ edge coloring in $O(1)$ rounds. This is somewhat surprising since all previous non-trivial algorithms in the strictly sublinear regime of MPC require $\omega(1)$ rounds.

Our algorithm can also be implemented in $O(1)$ rounds of *Congested Clique*, leading to a $\Delta + \widetilde{O}(\Delta^{3/4})$ edge coloring there. Prior to our work, no sublogarithmic round Congested Clique algorithm was known even for $(2\Delta-1)$ edge coloring.

2.1.2 Streaming

The Model. In the standard graph streaming model, the edges of a graph arrive one by one and the algorithm has a space that is much smaller than the total number of edges. A particularly important choice of space is $\widetilde{O}(n)$—which is also known as the *semi-streaming* model [35]—so that the algorithm has enough space to store the vertices but not the edges. For edge coloring, the output is as large as the input, thus, we cannot hope to be able to store the output and report it in bulk at the end. For this, we consider a standard twist on the streaming model where the output is also reported in a streaming fashion. This model is referred to in the literature as the "W-streaming" model [33, 40]. We particularly focus on one-pass algorithms.

Designing one-pass W-streaming algorithms is particularly challenging since the algorithm cannot "remember" all the choices made so far (e.g., the reported edge colors). Therefore, even the sequential greedy algorithm for $(2\Delta-1)$ edge coloring, which iterates over the edges in an arbitrary order an assigns an available to each color upon visiting it, cannot be implemented since we are not aware of the colors used incident to an edge.

Our first result is to show that a natural algorithm w.h.p.[2] provides an $O(\Delta)$ edge coloring if the edges arrive in a random-order.

Result 2 (Theorem 2.3.2)**.** If the edges arrive in a random-order, there is a one-pass $\widetilde{O}(n)$ space W-streaming edge coloring algorithm that always returns a valid edge coloring and w.h.p. uses $(2e + o(1))\Delta \approx 5.44\Delta$ colors.

If the edges arrive in an arbitrary order, we give another algorithm that requires more colors.

Result 3 (Theorem 2.3.3)**.** For any arbitrary arrival of edges, there is a one-pass $\widetilde{O}(n)$ space W-streaming edge coloring algorithm that succeeds w.h.p. and uses $O(\Delta^2)$ colors.

These are, to our knowledge, the first streaming algorithms for edge coloring.

2.2 The MPC Algorithm

In this section, we consider the edge coloring problem in the MPC model. Our main result in this section is an algorithm that achieves the following:

Theorem 2.2.1. *For any parameter k (possibly dependent on Δ) such that $n/k \gg \log n$, there exists an* MPC *algorithm with $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ space per machine and $O(m)$ total space that w.h.p. returns a $\Delta + O(\sqrt{k\Delta \log n})$ edge coloring in $O(1)$ rounds.*

By setting $k = \sqrt{\Delta} + \log n$, the space required per machine will be $O(n)$ and the number of colors would be $\Delta + \widetilde{O}(\Delta^{3/4})$. Using a reduction from [16], this also leads to an $O(1)$ round Congested Clique algorithm using the same number of colors.

Corollary 2.2.2. *There exists a randomized* MPC *algorithm with $O(n)$ local space, as well as a Congested Clique algorithm, that both w.h.p. find a $\Delta + \widetilde{O}(\Delta^{3/4})$ edge coloring in $O(1)$ rounds.*

Moreover, assuming that $\Delta = n^{\Omega(1)}$, by setting $k = \Delta^{0.5+\epsilon}$ for a small enough constant $\epsilon \in (0, 1)$, we get the following $O(1)$ round algorithm which requires $n^{1-\Omega(1)}$ machine space, which is notably strictly sublinear in n:

Corollary 2.2.3. *If $\Delta = n^{\Omega(1)}$, there exists a randomized* MPC *algorithm with $O(n/\Delta^{2\epsilon}) = n^{1-\Omega(1)}$ space per machine and $O(m)$ total space that w.h.p. returns a $\Delta + \widetilde{O}(\Delta^{0.75+\epsilon/2})$ edge coloring in $O(1)$ rounds.*

The Idea Behind the Algorithm. The first step in the algorithm is a random partitioning of the *vertex set* into k groups, $V_1, \ldots, V_k$. We then introduce one subgraph for each vertex subset, called $G_1, \ldots, G_k$, and one subgraph for every pair of groups which we denote as $G_{1,2}, \ldots, G_{1,k}, \ldots, G_{k-1,k}$. Any such G_i is simply the induced subgraph of G on V_i. Moreover, any such $G_{i,j}$ is the subgraph on vertices $V_i \cup V_j$, with edges with one point in V_i and the other in V_j.

The general idea is to assign different *palettes*, i.e., subsets of colors, to different subgraphs so that the palettes assigned to any two neighboring subgraphs (i.e., those that share a vertex) are completely disjoint. A key insight to prevent this from blowing up the number of colors, is that since any two edges from $G_{i,j}$ and $G_{i',j'}$ with $i \neq i'$ and $j \neq j'$ cannot share endpoints by definition, it is safe to use the same color palette for them.

[2]Throughout, we use "w.h.p." to abbreviate "with high probability" implying probability at least $1 - 1/\operatorname{poly}(n)$.

To assign these color palettes, we consider a complete k-vertex graph with each vertex v_i in it corresponding to partition V_i and each edge (v_i, v_j) in it corresponding to the subgraph $G_{i,j}$. We then find a k edge coloring of this complete graph, which exists by Vizing's theorem since maximum degree in it is $k-1$. This edge coloring can actually be constructed extremely efficiently using merely the edges' endpoint IDs. Thereafter, we map each of these k colors to a color palette. By carefully choosing k and the number of colors in each palette, we ensure that: (1) The total number of colors required is close to Δ. (2) Each subgraph $G_{i,j}$ can be properly edge-colored with those colors in its palette. (3) Each subgraph fits the memory of a single machine so that we can put it in whole there and run the sequential edge coloring algorithm on it.

Algorithm 1: An MPC algorithm for edge coloring.

```
1  Parameter: $k$.;
   Output: An edge coloring of a given graph $G = (V, E)$ with maximum degree $\Delta$ using $\Psi := \Delta + d\sqrt{k\Delta \log n}$ colors
           for some large enough constant $d$.
2  Independently and u.a.r. partition $V$ into $k$ subsets $V_1, \dots, V_k$.;
3  For every $i \in [k]$, let $G_i$ be the induced subgraph of $G$ on $V_i$.;
4  For every $i, j \in [k]$ with $i \neq j$, let $G_{i,j}$ be the subgraph of $G$ including an edge $e \in E$ iff one end-point of $e$ is in $V_i$
     and the other is in $V_j$.;
5  Partition $[\Psi]$ into $k+1$ disjoint subsets $C_1, \dots, C_k, C'$, which we call color palettes, in an arbitrarily way such that
     each palette has exactly $\frac{\Psi}{k+1}$ colors.;
6  for each graph $G_i$ in parallel do
7      Color $G_i$ sequentially in a single machine with palette $C'$.;
8  end
   // In what follows, we implicitly construct a $k$ edge coloring of a complete $k$-vertex graph $K_k$ and assign palette $C_\alpha$
      to subgraph $G_{i,j}$ where $\alpha$ is the color of edge $(i, j)$ in $K_k$.
9  for each graph $G_{i,j}$ in parallel do
10     Color $G_{i,j}$ sequentially in a machine with palette $C_\alpha$ where $\alpha = ((i + j) \bmod k) + 1$.;
11 end
```

The algorithm outlined above is formalized as Algorithm 1. We start by proving certain bounds on subgraphs' size and degrees.

Claim 2.2.4. *W.h.p., every subgraph of type G_i or $G_{i,j}$ has maximum degree $\frac{\Delta}{k} + O(\sqrt{\Delta \log \frac{n}{k}})$ and has at most $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ edges.*

Proof. Let us start with bounding the degree of an arbitrary vertex $v \in V_i$ in subgraph G_i. The degree of vertex v in G_i is precisely the number of its neighbors that are assigned to partition V_i. Since there are k partitions, the expected degree of v in G_i is $\deg_G(v)/k \leq \Delta/k$. Furthermore, since the assignment of vertices to the partitions is done independently and uniformly at random, by a simple application of Chernoff bound, v's degree in G_i should be highly concentrated around its mean. Namely, with probability at least $1 - n^{-2}$, it holds that $\deg_{G_i}(v) \leq \frac{\Delta}{k} + O(\sqrt{\Delta \log n/k})$. Now, a union bound over the n vertices in the graph, proves that the degree of all vertices in their partitions should be at most $\frac{\Delta}{k} + O(\sqrt{\Delta \log n/k})$ with probability $1 - 1/n$.

Bounding vertex degrees in subgraphs of type $G_{i,j}$ also follows from essentially the same argument. The only difference is that we have to union bound over $n \cdot k$ choices, as we would like to bound the degree of any vertex v with say $v \in V_i$ in k subgraphs $G_{i,1}, \dots, G_{i,k}$. Nonetheless, since $k \leq n$, there are still poly(n) many choices to union bound over. Thus, by changing the constants in the lower terms of the concentration bound, we can achieve the same high probability result.

Finally, we focus on the number of edges in each of the subgraphs. Each partition V_i has n/k vertices in expectation since the n vertices are partitioned into k groups independently and uniformly at random. A simple application of Chernoff and union bounds, implies that the number of vertices in each partition V_i is at most $O(\frac{n}{k})$ w.h.p., so long as $n/k \gg \log n$, which is the case. Since the number of edges in each partition is less than the number of vertices times max degree, combined with the aforementioned bounds on the max degree, we can bound the number of edges in G_i and $G_{i,j}$ for any i and j by

$$O\Big(\frac{n}{k}\Big) \cdot O\left(\frac{\Delta}{k} + \sqrt{\frac{\Delta}{k}\log n}\right) = O\left(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\frac{\Delta}{k}\log n}\right),$$

which is the claimed bound. □

Next, observe that we use palettes $C_1, \ldots, C_{k+1}, C'$, each of size $\frac{\Psi}{k+1}$ to color the subgraphs. We need to argue that the maximum degree in each subgraph is at most $\frac{\Psi}{k+1} - 1$ to be able to argue that using Vizing's theorem in one machine, we can color any of the subgraphs with the assigned palettes. This can indeed be easily guaranteed if the constant d is large enough:

Observation 2.2.1. *If constant d in Algorithm 1 is large enough, then maximum degree of every graph is at most $\frac{\Psi}{k+1} - 1$, w.h.p.*

Proof. We have $\Psi = \Delta + d\sqrt{k\Delta \log n}$ in Algorithm 1, therefore:

$$\frac{\Psi}{k+1} = \frac{\Delta}{k+1} + \frac{d\sqrt{k\Delta\log n}}{k+1} = \frac{\Delta}{k} + \Theta(\sqrt{\Delta \log n/k}),$$

where the hidden constants in the second term of the last equation can be made arbitrarily large depending on the choice of constant d. On the other hand, recall from Claim 2.2.4 that the maximum degree in any of the subgraphs is also at most $\frac{\Delta}{k} + O(\sqrt{\Delta \log n/k})$. Thus, the palette sizes are sufficient to color the subgraphs if d is a large enough constant. □

We are now ready to prove the algorithm's correctness.

Lemma 2.2.5. *Algorithm 1 returns a proper edge coloring of G using $\Delta + O(\sqrt{k\Delta\log n})$ colors.*

Proof. The algorithm clearly uses $\Psi = \Delta + O(\sqrt{k\Delta \log n})$ colors, it remains to argue that the returned edge coloring is proper. Each subgraph (of type G_i or $G_{i,j}$) is sent to a single machine and edge-colored there using the palette that it is assigned to. Since by Observation 2.2.1, each palette has at least $\Delta' + 1$ colors for Δ' being the max degree in the subgraphs, there will be no conflicts in the colors associated to the edges within a partition. We only need to argue that two edges e and f sharing a vertex v that belong to two different subgraphs are not assigned the same color. Note that all subgraphs of type G_i are vertex disjoint and all receive the special color palette C', thus there cannot be any conflict there. To complete the proof, it suffices to prove that any two subgraphs $G_{i,j}$ and $G_{i',j'}$ that share a vertex receive different palettes. Note that in this case, either $i = i'$ or $j = j'$ by the partitioning. Assume w.l.o.g. that $i = i'$ and thus $j \neq j'$. Based on Algorithm 1 for $G_{i,j}$ and $G_{i',j'}$ to be assigned the same color palette, it should hold that

$$((i + j) \bmod k) + 1 = ((i' + j') \bmod k) + 1.$$

Since $i = i'$, this would imply that $(j \bmod k) = (j' \bmod k)$, though this would not be possible given that both j and j' are in $[k]$ and that $j \neq j'$. Therefore, any two subgraphs that share a vertex receive different palettes and thus there cannot be any conflicts, completing the proof. □

Next, we turn to prove the space bounds.

Lemma 2.2.6 (Implementation and Space Complexity)**.** *Algorithm 1 can be implemented with total space $O(m)$ and space per machine of $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ w.h.p.*

Proof. We start with an implementation that uses the specified space per machine but can be wasteful in terms of the total space, then describe how we can overcome this problem and also achieve an optimal total space of $O(m)$.

We can use $k + \binom{k}{2}$ machines, each with a space of size $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ to assign colors to the edges in parallel. The first $m_1, \dots, m_k$ machines will be used for edge coloring on $G_1, G_2, \dots, G_k$ respectively. The other $m_{k+1}, \dots, m_{k+\binom{k}{2}}$ machines will be used for edge coloring on the $G_{i,j}$ graphs. Lemma 2.2.4 already guarantees that each subgraph has size $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ w.h.p., and thus fits the memory of a single machine.

In the implementation discussed above, since the machines use $\widetilde{O}(n\Delta/k^2)$ space and there are $O(k^2)$ machines, the total memory can be $\widetilde{O}(n\Delta)$ which may be much larger than $O(m)$. This is because we allocate $O(n\Delta/k^2)$ space to each machine regardless of how much data it actually received. Though, observe that each edge of the graph belongs to exactly one of the subgraphs, i.e., the machines together only handle a total of $O(m)$ data. So we must consolidate into fewer machines. We do this by putting multiple subgraphs in each machine.

We start by recalling a sorting primitive in the MPC model which was proved in [43]. Basically, if there are N items to be sorted and the space per machine is $N^{\Omega(1)}$, then the algorithm of [43] sorts these items into the machines within $O(1)$ rounds. To use this primitive, we first label each edge $e = (u, v)$ of the graph by its subgraph name (e.g. G_i or $G_{i,j}$) which can be determined solely based on the end-points of the edge. After that, we sort the edges based on these labels. This way, all the edges inside each subgraph can be sent to the same machine within $O(1)$ rounds while also ensuring that the total required space remains $O(m)$. $\square$

The algorithm for Theorem 2.2.1 was formalized as Algorithm 1. We showed in Lemma 2.2.5 that the algorithm correctly finds an edge coloring of the graph with the claimed number of colors. We also showed in Lemma 2.2.6 that the algorithm can be implemented with $O(m)$ total space and $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ space per machine. This completes the proof of Theorem 2.2.1.

2.3 Streaming Algorithms

We start in Section 2.3.1 by describing our streaming algorithm and its analysis when the arrival order is random. Then in Section 2.3.2, we give another algorithm for adversarial order streams.

2.3.1 Random Edge Arrival Setting

In this section, we give a streaming algorithm for $O(\Delta)$ edge coloring using $\widetilde{O}(n)$ space where the edges come in a random stream. That is, a permutation over the edges is chosen uniformly at random and then the edges arrive according to this permutation.

We first note that if $\Delta = O(\log n)$ then the problem is trivial as we can store the whole graph and then report a $\Delta+1$ edge coloring (even without knowledge of Δ). As such, we assume $\Delta = \omega(\log n)$.

The algorithm — formalized as Algorithm 2 — maintains a counter c_v for each vertex v. At any point during the algorithm, this counter c_v basically denotes the highest color number used for the edges incident to v so far, plus 1. Therefore, upon arrival of an edge (u, v), it is safe to color this edge with $\max(c_u, c_v)$ as all edges incident to u and v have a color that is strictly smaller than this. Then, we increase the counters of both v and u to $\max(c_u, c_v) + 1$. It is not hard to see that the solution is always a valid coloring, in the remainder of this section, we mainly focus on the number of colors required by this algorithm and show that w.h.p., it is only $O(\Delta)$ for random arrivals.

We start by noting that this algorithm can actually be extremely bad if the order is adversarial. To see this, consider a path of size n. In an adversarial stream where the edges arrive in the order

Algorithm 2: Edge coloring for random streams.

```
Result: A feasible coloring $\mathcal{C} : E \to [\Psi]$ for a given graph $G = (V, E)$ with maximum degree $\Delta$ in a random stream
1 $c_v \leftarrow 0 \quad \forall v \in V$;
2 while $(u, v)$ is read from stream do
3     $\mathcal{C}(u, v) \leftarrow \max(c_u, c_v)$;
4     $c_u, c_v \leftarrow \mathcal{C}(u, v) + 1$;
5 end
```

of the path, Algorithm 2 uses as many as $n-1$ colors while the maximum degree is only 2! It is easy to see why this example is very unlikely to occur in random order streams: For a fixed path, it is very unlikely that the edges are randomly ordered in this very specific way.

To make this intuition rigorous for general graphs, we first prove the following crucial lemma which gives us the correct parameter to bound.

Lemma 2.3.1. *Let Ψ be the size of the longest monotone (in the order of arrival) path in the line-graph of G. Then Algorithm 2 uses exactly Ψ colors.*

Proof. Take a monotone path $v_1, v_2, \ldots, v_\Psi$ in the line-graph of G and let $e_1, e_2, \ldots, e_\Psi$ be the edges of the original graph that correspond to these vertices respectively, i.e., e_1 arrives before e_2 which arrives before e_3 and so on. Since for any i, v_i and v_{i+1} are neighbors in the line-graph, then e_i and e_{i+1} should share an end-point v. This means that at the time of arrival of e_{i+1}, we have $c_v \geq \mathcal{C}(e_i) + 1$ which in turn, implies $\mathcal{C}(e_\Psi) > \mathcal{C}(e_{\Psi-1}) > \ldots > \mathcal{C}(e_1)$. Therefore, $\mathcal{C}(e_\Psi) \geq \Psi$.

On the other hand, suppose that there is an edge $e_1 = (u, v)$ for which $\mathcal{C}(e_1) = \Psi$ in Algorithm 2. This means that at least one of c_u or c_v equals Ψ when e_1 arrives, say c_u w.l.o.g. Let e_2 be the last edge incident to u that has arrived before e_1. It should hold that $\mathcal{C}(e_2) = \Psi - 1$. Using the same argument, for each $1 < i \leq \Psi$, we can find a neighboring edge e_i such that $\mathcal{C}(e_i) = \mathcal{C}(e_{i-1}) - 1$. This way, we end up with a sequence $e_1, \ldots, e_\Psi$ of edges, the path corresponding to this sequence in the line graph will be a monotone path of length Ψ, completing the proof. $\square$

Theorem 2.3.2. *There is a streaming edge coloring algorithm that for any graph $G = (V, E)$ uses at most $(2e + \epsilon)\Delta \approx 5.44\Delta$ colors w.h.p. for any constant $\epsilon > 0$ given that the edges in E arrive in a random order.*

Proof. We first prove that Algorithm 2 gives us a feasible coloring of graph G. Consider two edges $e_1 = (u, v)$ and $e_2 = (u, v')$ incident to vertex u such that e_1 appears earlier than e_2 in the stream. For any edge e we represent by $\mathcal{C}(e)$ the color assigned to that by the algorithm. After the algorithm colors e_1 with $\mathcal{C}(e_1)$, it sets c_u to $\mathcal{C}(e_1) + 1$. Thus, c_u is at least $\mathcal{C}(e_1) + 1$ when e_2 arrives and $\mathcal{C}(e_2) \geq \mathcal{C}(e_1) + 1$ consequently. Therefore, $\mathcal{C}(e_2) > \mathcal{C}(e_1)$ for any pair of edges incident to a common vertex, and $\mathcal{C}$ is a feasible coloring.

Next, for some constant α that we fix later, we show that the probability that an edge is assigned a color number at least $\alpha\Delta$ is at most n^{-c} for some constant $c \geq 2$, implying via a union bound over all the edges that indeed w.h.p., $\Psi \leq \alpha\Delta$.

We showed in Lemma 2.3.1 that if the number of colors Ψ used is $\alpha\Delta$, then there should exist a monotone path in the line-graph with size at least $\alpha\Delta$. Let $e_0, e_2, \ldots, e_{\alpha\Delta}$ be the corresponding edges to this path. Thus, it suffices to bound the probability of this event. Let Π denote the set of all such paths in the line graph. For a specific path $\pi \in \Pi$, the probability that it is monotone is $1/(\alpha\Delta)!$. Call this event X_π. On the other hand, we can upper bound the number of such paths by $(2\Delta)^{\alpha\Delta}$, i.e., $|\Pi| \leq (2\Delta)^{\alpha\Delta}$. This follows from the fact that each path should start from the

Clique Size	100	200	300	400	500	600	700	800	900	1000
Colors Used	3.363Δ	3.563Δ	3.665Δ	3.717Δ	3.756Δ	3.787Δ	3.815Δ	3.838Δ	3.849Δ	3.863Δ

Table 2.1: The number of colors used by Algorithm 2 on cliques averaged over 100 trials.

corresponding vertex to e_0 in the line-graph, and that maximum degree in the line graph is $2\Delta - 2$ (which is the upper bound on the number of neighboring edges to each edge). Thus:

$$\Pr[\mathcal{C}(e_0) \geq \alpha\Delta] = \Pr[\bigvee_{\pi \in \Pi} X_\pi] \leq \sum_{\pi \in \Pi} \Pr[X_\pi = 1] \leq \frac{(2\Delta)^{\alpha\Delta}}{(\alpha\Delta)!},$$

where the last inequality is obtained by replacing $\Pr[X_\pi = 1]$ and $|\Pi|$ by the aforementioned bounds. Taking the logarithm of each side of the inequality, we get

$$\begin{aligned} \ln(\Pr[\mathcal{C}(e_0) \geq \alpha\Delta]) &\leq \alpha\Delta \ln(2\Delta) - \ln((\alpha\Delta)!) \\ &\leq \alpha\Delta \ln(2\Delta) - ((\alpha\Delta + 1/2)\ln(\alpha\Delta) - \alpha\Delta) && (2.1) \\ &= \alpha\Delta \ln(2e/\alpha) - 1/2 \ln(\alpha\Delta) && (2.2) \\ &\leq \alpha\Delta \ln(2e/\alpha). && (2.3) \end{aligned}$$

To obtain (2.1), we use Stirling's approximation of factorials to lower-bound $\ln((\alpha\Delta)!)$. Finally, we rearranged terms to imply (2.2). By plugging in $\alpha = 2e(1+\epsilon)$, we get

$$\begin{aligned} \ln(\Pr[\mathcal{C}(e_0) \geq 2e(1+\epsilon)\Delta]) &\leq 2e(1+\epsilon)\Delta \ln\left(\frac{1}{1+\epsilon}\right) \\ &= -2e(1+\epsilon)\ln(1+\epsilon)\Delta \\ &\leq -2e(1+\epsilon)\ln(1+\epsilon)\frac{c}{2e(1+\epsilon)\ln(1+\epsilon)}\ln(n) && (2.4) \\ &= -c\ln(n) \end{aligned}$$

Since $\Delta = \omega(\log(n))$, we have $\Delta > c' \ln(n)$ for any constant c'. Inequality (2.4) follows from setting $c' = c/(2e(1+\epsilon)\ln(1+\epsilon))$ in $\Delta > c'\ln(n)$, where c is the constant for which we want to show the probability is upper-bounded by n^{-c}. Hence,

$$\Pr[\mathcal{C}(e_0) \geq 2e(1+\epsilon)\Delta] \leq n^{-c}.$$

Thus, Algorithm 2 returns a feasible coloring of the input graph G using at most $2e(1+\epsilon)\Delta$ colors, for any constant $\epsilon > 0$ w.h.p. if the edges arrive in a random order. $\square$

To further evaluate the performance of Algorithm 2, we implemented and ran it for cliques of different size. The result of this experiment is provided in Table 2.1. The numbers are obtained by running the experiment 100 times and taking the average number of colors used. As it can be observed from Table 2.1, for cliques of size 100 to 1000, the number of colors used by the algorithm is in range $[3.3\Delta, 3.9\Delta]$ and it slightly increases by the size of the graph. Our analysis, however, shows that it should never exceed 5.44Δ.

2.3.2 Adversarial Edge Arrival Setting

In this section, we turn to arbitrary (i.e., adversarial) arrivals of the edges. We assume that the adversary is *oblivious*, i.e., the order of the edges is determined before the algorithm starts to operate so that the adversary cannot abuse the random bits used by the algorithm. Having this assumption, we give a randomized algorithm that w.h.p., outputs a valid edge coloring of the graph using $O(\Delta^2)$ colors while using $\tilde{O}(n)$ space. The algorithm is formalized as Algorithm 3. We note that this algorithm, as stated, requires knowledge of Δ. However we later show that we can get rid of this assumption. Overall, we get the following result:

Algorithm 3: Edge coloring in the adversarial order

```
Result: A feasible coloring for a given graph $G = (V, E)$ with maximum degree $\Delta$
for any vertex $v \in V$ do
    $r_v \leftarrow$ a sequence of $\log(n)$ independent random bits.
    for any $i \in [\log n]$ do
        $c_{v,i} \leftarrow 0$
    end
end
for any edge $e = (u, v)$ in the stream do
    Let $i$ be the smallest index for which $r_{v,i} \neq r_{u,i}$.
    if $\Delta 2^{-i} > \log n$ then
        if $r_{u,i} = 1$ then
            Assign color $(c_{u,i}, c_{v,i}, i)$ to $e$.
        else
            Assign color $(c_{v,i}, c_{u,i}, i)$ to $e$.
        end
        Increase both $c_{v,i}$ and $c_{u,i}$ by one.
    else
        Store edge $e$.
    end
end
Color the stored edges using a new set of colors.
```

Theorem 2.3.3. *Given a graph G with maximum degree Δ, there exists a one pass streaming algorithm, that outputs a valid edge coloring of the G using $O(\Delta^2)$ colors w.h.p., using $\widetilde{O}(n)$ memory.*

Consider two vertices v and u and their string of random bits r_v and r_u defined in the algorihtm. Let $d_{u,v}$ be the smallest index i where $r_{u,i} \neq r_{v,i}$. Upon arrival of an edge $e = (u, v)$, we first find $i := d_{u,v}$. If $\Delta 2^{-i} > \log n$, we color the edge immediately. Otherwise, we store it. We will show that all the stored edges fit in the memory thus after reading all the stream we can color them with a palette of at most $\Delta + 1$ new colors. In the algorithm, for any vertex v and any $i \in [\log n]$, we define a counter $c_{u,i}$. If $\Delta 2^{-i} > \log n$ for any edge e, then we immediately assign e a color which is represented by a tuple $(c_{u,i}, c_{v,i}, i)$. Then, we increase counters $c_{u,i}$ and $c_{v,i}$. Note that we say two colors are the same if all three elements of them are equal. We first show that this gives us a valid coloring, which means it does not assign the same color to two edges adjacent to the same vertex. We use proof by contradiction. Assume that our algorithm assigns the same color to edges $e_1 = (u, v_1)$ and $e_2 = (u, v_2)$ adjacent to vertex u. None of them can be from the stored edges since we color them using a new palette. This means that $d_{u,v_1} = d_{u,v_2}$. Let us denote it by i. Without loss of generality, we assume that $r_{u,i} = 1$ and that in the input stream e_1 arrives before e_2. Note that the first element of the colors (which are tuples) assigned to these edges is the value of counter $c_{u,i}$ when they arrive. However, the algorithm increases $c_{u,i}$ by one after arrival of e_1 thus the colors assigned to e_1 and e_2 cannot be the same.

Now, it suffices to show that the total number of colors used by the algorithm is $O(\Delta^2)$. Given a vertex v, and a number $l \in [\log n]$ let us compute an upper-bound for counter $c_{v,i}$. Let N_v be the set of neighbors of this vertex and let $N_{v,i}$ be the set of neighbors like u where $d_{v,u} = i$. We know that $c_{v,i} = |N_{v,i}|$, thus given any vertex v and $i \in [\log(n)]$, we need to find a bound for $|N_{v,i}|$. Given any edge $e = (v, u)$ the probability of e being in set $N_{v,i}$ is 2^{-i} which means $\mathbb{E}[|N_{v,i}|] = \deg(v)2^{-i}$ where $\deg(v)$ is the degree of vertex v in the input graph.

Using a simple application of the Chernoff bound, for any vertex v, we get:

$$Pr\Big[|N_{v,i}| \geq \deg(v)2^{-i} + O\big(\sqrt{\deg(v)2^{-i}\log n}\big)\Big] \leq \frac{1}{n^c}.$$

Setting c to be a large enough constant, one can use union bound and show that w.h.p., for any vertex v and $i \in [\log n]$ where $\deg(v)2^{-i} \geq \log n$, we have $|N_{v,i}| \leq O(\deg(v)2^{-i})$.

Having this, we conclude that for any $i \in [\log n]$, where $\Delta 2^{-i} > \log n$, the number of colors used by the algorithm whose third element is i is at most $O(\Delta^2 2^{-2i})$ since the first and the second element of the color can get at most $O(\Delta 2^{-i})$ different values. Therefore, the total number of colors used for any such i is at most $O\big(\sum_{i\in[\log n]}\Delta^2 2^{-2i}\big) = O(\Delta^2)$. We will also show that the stored edges fit in the memory and thus we can color them using $O(\Delta)$ new colors. As a result the total number of colors used is $O(\Delta^2)$.

To give an upper-bound for the number of stored edges we first show that the expected number of stored edges for each vertex is $O(\log n)$. Let $j := \log(\frac{\Delta}{\log n})$. Recall that we store an edge (u, v) when $\Delta 2^{-d_{u,v}} < \log n$. Thus the expected number of stored edges adjacent to a single vertex v is at most

$$\sum_{j\leq i\leq \log n} d_v 2^{-i} \leq \sum_{j\leq i\leq \log n} \Delta 2^{-i} \leq \sum_{j\leq i\leq \log n} \log(n)2^{-i+j} = O(\log n).$$

To get the last equation we use the fact that $\Delta 2^{-j} \leq \log n$. By a similar argument that we used above (using Chernoff and Union bounds), with a high probability the total number of stored edges is $O(n\log n)$ which can be stored in the memory. Therefore the proof of this theorem is completed.

Knwoledge of Δ. As written, our algorithm depends on the knowledge of Δ because we must check $\Delta 2^{-i} > \log n$. We can get rid of this condition by keeping track of the degree $\deg_v^H$ of a vertex in the subgraph H we have seen so far, and then computing the max degree $\deg_{max}^H$. This only requires an additional $O(n)$ space. Thereafter, instead of checking if $\Delta 2^{-i} > \log n$, we check if $\deg_{max}^H 2^{-i} > \log n$. Whenever $\deg_{max}^H$ increases, we iterate over all stored edges and recompute whether or not $\deg_{max}^H 2^{-i} > \log n$. If so, we color the edge and remove it from the buffer, else we keep it. It is easy to see that this will not exceed the space bounds because at any timestep, we can assume the input graph was H in the first place. Then its max degree is $\Delta_H = \deg_{max}^H$, and we can apply the same argument for the space bounds as before, but using Δ_H instead of Δ. All other parts of the proof still hold. Therefore our algorithm does not require knowledge of Δ.

Finally, we remark that if one allows more space, then one can modify Algorithm 3 to use fewer number of colors. Though we focused only on the $\widetilde{O}(n)$ memory regime.

2.4 Open Problems

We believe the most notable future direction is to improve the number of colors used in our streaming algorithms. Specifically, our streaming algorithm for adversarial arrivals requires $O(\Delta^2)$ colors. A major open question is whether this can be improved to $O(\Delta)$ while also keeping the memory near-linear in n. Also for random arrival streams, we showed that Algorithm 2 achieves a 5.44Δ coloring and showed, experimentally, that it uses at least 3.86Δ colors. A particularly interesting open question is whether there is an algorithm that uses arbitrarily close to 2Δ colors using $\widetilde{O}(n)$ space in random arrival streams.

Chapter 3

String Matcing

3.1 Introduction

The string matching problem with wildcards, or *pattern matching*, seeks to identify pieces of a text that adhere to a certain structure called the *pattern*. Pattern matching is one of the most applied problems in computer science. Examples range from simple batch applications such as Awk, Sed, and Diff to very sophisticated applications such as anti-virus tools, database queries, web browsers, personal firewalls, search engines, social networks, etc. The astonishing growth of data on the internet as well as personal computers emboldens the need for fast and scalable pattern matching algorithms.

In theory too, pattern matching is a well-studied and central problem. The simplest variant of pattern matching, namely *string matching*, dates back to 1960s. In this problem, two strings T and P are given as input and the goal is to find all substrings of T that are identical to P. The celebrated algorithm of Knuth, Morris, and Pratt [55] (KMP) deterministically solves the problem in linear time. Since then, attention has been given to many variants and generalizations of pattern matching [46, 74, 75, 34, 55, 60, 12, 24, 48, 64, 65, 20, 73, 2, 47, 28, 27, 1, 70, 72, 62, 71, 66]. Natural generalizations of string matching are when either the text or the pattern is a tree instead of a string [46, 74, 75, 34] or when the pattern has a more sophisticated structure that allows for '?', '+', '*', or in general any regular expression [60, 12, 24, 48]. Also, different computational systems have been considered in the literature: from sequential algorithms [46, 74, 75, 34, 55, 60, 12, 24, 48], to quantum algorithms [64, 65, 20, 73], to distributed settings [2, 47, 28], to the streaming setting [70, 27, 1], to PRAM [72, 62, 71, 66], etc.

An obvious application of pattern matching is in anti-virus softwares. In this case, a malware is represented with a pattern and a code or data is assumed to be infected if it contains the pattern. In the simplest case, the pattern only consists of ascii characters. However, it happens in practice that malwares allow for slight modifications. That is, parts of the pattern code are subject to change. This can be captured by introducing wildcards to pattern matching. More precisely, each element of the pattern is either an ascii code or a special character '?' which stands for a wildcard. The special character is allowed to match with any character of the text.

Indeed the desirable property of the '?' case is that the length of the pattern is always fixed. However, one may even go beyond this setting and consider the cases where the pattern may match to pieces of the text with variant lengths. Two classic ways to incorporate this into the model is to consider two special characters '+' and '*'. The former allows for arbitrary repetitions of a single character and the latter allows for arbitrary repetitions of any combination of characters. For example, as an application of pattern matching in bioinformatics, we might be looking for a set of gene patterns in a DNA sequence. Obviously, these pattern are not necessarily located consecutively in the DNA sequence, and one might utilize '*' wildcard to address this problem.

In practice, these problems are formulated around huge data sets. For instance, a human DNA encompasses roughly a Gigabyte of information, and an anti-virus scans Gigabytes (if not Petabyets) of data on a daily basis. Thus, the underlying algorithm has to be scalable, fast, and memory efficient. A natural approach to obtain such algorithms is parallel computation. Motivated by such needs, the massively parallel computation (MPC) model [53, 5, 43, 15] has been introduced to understand the power and limitations of parallel algorithms. It first proposed by Karloff et al. [53] as a theoretical model to embrace Map Reduce algorithms, a class of powerful parallel algorithms not compatible with previously defined models for parallel computation. Recent developments in the MPC model have made it a cornerstone for obtaining massively parallel algorithms.

While in the previous parallel settings such as the PRAM model, usually an $O(\log n)$ factor in the round complexity is inevitable, MPC allows for sublogarithmic round complexity [53, 31, 59]. Karloff et al. [53] also compared this model to PRAM, and showed that for a large portion of PRAM algorithms, there exists an MPC algorithm with the same number of rounds. In this model, each machine has unlimited access to its memory, however, two machines can only interact in between two rounds. Thus, a central parameter in this setting is the round complexity of algorithm since network communication is the typical main bottleneck in practice. The ultimate goal is developing constant-round algorithms, which are highly desirable in practice.

The MPC model. In this paper, we assume that the input size is bounded by $O(n)$, and we have $\mathcal{M}$ machines of each with a memory of $\mathcal{S}$. In the MPC model [53, 5, 43, 15], we assume the number of machines and the local memory size on each machine is asymptotically smaller than the input size. Therefore, we fix an $0 < x < 1$ and bound the memory of each machine by $\widetilde{O}(n^{1-x})$. Also, our goal is to have near linear total memory and therefore we bound the number of machines by $\widetilde{O}(n^x)$. An MPC algorithm runs in a number of rounds. In every round, every machine makes some local computation on its data. No communication between machines is allowed during a round. Between two rounds, machines are allowed to communicate so long as each machine receives no more communication than its memory. Any data that is outputted from a machine must be computed locally from the data residing on the machine and initially the input data is distributed across the machines.

In this work we give MPC algorithms for different variants of the pattern matching problem. For the regular string matching and also '?' and '+' wildcard problems, our algorithms are tight in terms of running time, memory per machine, and round complexity. Both '?' and '+' wildcard problems are reduced to fast convolution at the end, and make use of the fact that FFT could computed in $O(1)$ MPC rounds with near-linear total running time and total memory. Also, for the case of '*' wildcard we present nontrivial MPC algorithms for two special cases that mostly tend to happen in practice. However, the round complexity of these two cases is $O(\log(n))$, and the general case problem is not addressed in this paper.

3.1.1 Our Results and Techniques

Throughout this paper, we denote the text by T and the pattern by P. Also, we denote the set of characters by Σ.

We begin, as a warm-up, in Section 3.3 by giving a simple MPC algorithm that solves string matching in 2 rounds. The basic idea behind our algorithm is to cleverly construct hash values for the substrings of the text and the pattern. In other words, we construct an MPC data structure that enables us to answer the following query in a single MPC round:

Given indices i and j of the text, what is the hash value for the substring of the text starting from position i and ending at position j?.

Indeed, after the construction of such a data structure, one can solve the problem in a single round by making a single query for every position of the text. This gives us a linear time MPC algorithm that solves string matching in constant rounds.

Result 4 (Theorem 3.3.1)**.** There exists an MPC algorithm that solves string matching in constant rounds. The total memory and the total running time of the algorithm are linear.

For the case of wildcard '?', the hashing algorithm is no longer useful. It is easy to see that since special '?' characters can be matched with any character of the alphabet, no hashing strategy can identify the matches. However, a more sophisticated coding strategy enables us to find the occurrences of the pattern in the text. Assume for simplicity that $m = |\Sigma|$ is the size of the alphabet and we randomly assign a number $1 \leq \mathsf{mp}_c \leq m$ to each character c of the alphabet. Moreover, we assume that all the numbers are unique that is for two characters c and c' we have $\mathsf{mp}_c = \mathsf{mp}_{c'}$ if and only if $c = c'$. Now, construct a vector $T^\dagger$ of size $2|T|$ such that $T^\dagger_{2i-1} = \mathsf{mp}_{T_i}$ and $T^\dagger_{2i} = 1/\mathsf{mp}_{T_i}$ for any $1 \leq i \leq |T|$. Also, we construct a vector $P^\dagger$ of size $2|P|$ similarly, expect that $P^\dagger_{2i-1} = P^\dagger_{2i} = 0$ if the i'th character of P is '?'. Let nz_P be the number of the normal characters ('?' excluded) of the pattern. It follows from the construction of $T^\dagger$ and $P^\dagger$ that if P matches with a position i of the text, then we have:

$$T^\dagger[2i-1, 2i+2(|P|-1)].P^\dagger = \mathsf{nz}_P$$

where $T^\dagger[2i-1, 2i+2(|P|-1)]$ is a sub-vector of A only containing indices $2i-1$ through $2i+2(|P|-1)$. Moreover, it is showed in [36] that the vice versa also holds. That is if $T^\dagger[2i-1, 2i+2(|P|-1)].P^\dagger = \mathsf{nz}_P$ for some i then pattern P matches position i. This reduces the problem of pattern matching to the computation of dot products which is known to admit a linear time solution using fast Fourier transform (FFT) [23].

Result 5 (Theorem 3.4.2)**.** There exists an MPC algorithm that computes FFT in constant rounds. The total memory and the total running time of the algorithm are $\widetilde{O}(n)$.

Corollary 3.4.3 gives us an efficient MPC algorithm for the wildcard setting. While the reduction from wildcard matching to FFT is a known technique [30] the fact that FFT is computable in $O(1)$ MPC rounds leads to efficient MPC algorithm for a plethora of problems. FFT is used in various combinatorial problems such as knapsack [14], 3-sum [25], subset-sum [56], tree-sparsity [10], tree-separability [14], necklace-alignment [22], etc.

The case of '+' and '*' wildcards are technically more involved. In the case of repetition, special characters '+' may appear in the pattern. These special characters allow for arbitrary repetitions of a single character. For instance, a pattern "a+bb+" matches all strings of the form $\mathsf{a}^x.\mathsf{b}^y$ such that $x \geq 1$ and $y \geq 2$. Indeed this case is more challenging as the pattern may match with substrings of the text with different lengths.

To tackle this problem, we first compress both the text and the pattern into two strings T° and P° using the run-length encoding method. In the compressed versions of the strings, we essentially avoid repetitions and simply write the numbers of repetitions after each character. For instance, a text "aabcccddad" is compressed into "a[2]b[1]c[3]d[2]a[1]d[1]" and a pattern "ab+ccc+" is compressed into "a[1]b[1+]c[3+]". With this technique, we are able to break the problem into two parts. The first subproblem only incorporates the characters which is basically the conventional string matching. The second subproblem only incorporates repetitions. More precisely, in the second subproblem, we are given a vector A of n integer numbers and a vector B of m entries in the form of either i or $i+$. An entry of i matches only with indices of A with value i but an entry of $i+$ matches with any index of A with a value at least i. Since we already know how to solve string matching efficiently, in order to solve the repetition case, we need to find a solution for the latter subproblem.

To solve this subproblem, we use an algorithm due to Cole and Hariharan [30]. They showed the subset matching problem could be solved in near-linear time. The definition of the subset matching problem is as follows.

Problem 3.5.3 (restated). *Given T, a vector of n subsets of the alphabet Σ, and P, a vector of m subsets in the same format as T, find all occurrences of P in T. P is occurred at position i in T if for every $1 \leq j \leq |P|$, $P_j \subseteq T_{i+j-1}$.*

We can reduce our problem to an instance of the subset matching problem by replacing every T_i with $\{1+, 2+, \ldots, T_i+\} \cup \{T_i\}$, and keep S intact, i.e., replacing each S_i with $\{S_i\}$. This way, if there is a match, each S_i has to be included in the respective T_j. There is an algorithm for this problem with $\widetilde{O}(s)$ running time [30], where s shows the total size of all subsets in T and P. The running time is good enough for our algorithm to be near linear since s is at most twice the number of characters in the original text and pattern. Each T_i is the compressed version of T_i consecutive repetitions of a same character in T. We also show that the subset matching problem could be implemented in a constant number of MPC rounds, and thus a constant-round MPC algorithm for string matching with '+' wildcard is implied.

Result 6 (Theorem 3.5.5)**.** There exists an MPC algorithm that solves string matching with '+' wildcard in constant rounds. The total memory and the total running time of the algorithm are $\widetilde{O}(n)$.

Despite positive results for '?' and '+' wildcards, we do not know whether a poly-logarithmic round MPC algorithm exists for '*' wildcard in the most general case or not. This wildcard character matches any substring of arbitrary size in the text. We can consider a pattern consisted of '*' and Σ characters as a sequence of subpatterns, maximal substrings not having any '*', with a '*' between each consecutive pair. In the sequential settings, we can solve the problem by iterating over the subpatterns, and find the next matching position in the text for each one. If we successfully find a match for every subpattern in order, we end up with a substring of T matching P. Otherwise, it is easy to observe that T does not match the pattern P. To find the next matching position, we can perform a KMP algorithm on T and all the subpatterns one at a time, and make a transition to the next subpattern whenever we find a match, which is still linear in $n + m$, as the total size of the subpatterns is limited by m. The algorithm is provided with more details in Observation 3.6.1.

However, things are not as easy in the MPC model, because each subpattern can happen virtually anywhere in the text, and furthermore the number of subpatterns could be as large as $O(m)$. Thus, intuitively, it is impossible to know which subpatterns match each location of T with a linear total memory since the total size of this data could be $\Omega(nm)$, and also we cannot transfer this data in poly-logarithmic number of MPC rounds. Nonetheless, we provide two examples of how we can overcome this restriction by adding a constraint on the input. In both the cases, a near-linear $O(\log(n))$-round MPC algorithm exists for solving '*' wildcard problem.

(i). When the whole pattern fits in a single machine, $m = O(n^{1-x})$, then the number of subpatterns is limited by $O(n^{1-x})$. Thus, each machine could access each subpattern, and finds out

Problem		Theorem	Rounds	Total Runtime	Nodes
$P \in \Sigma^m$		3.3.1	$O(1)$	$O(n)$	$O(n^x)$
$P \in \{\Sigma \cup \text{‘?’}\}^m$		3.4.3	$O(1)$	$\widetilde{O}(n)$	$O(n^x)$
$P \in \{\Sigma \cup \text{‘+’}\}^m$		3.5.5	$O(1)$	$\widetilde{O}(n)$	$O(n^x)$
$P \in \{\Sigma \cup \text{‘*’}\}^m$	$m = O(n^{1-x})$	3.6.1	$O(\log(n))$	$O(n)$	$O(n^x)$
	no prefix	3.6.2	$O(\log(n))$	$O(n)$	$O(n^x)$

Table 3.1: Overview of results

if an interval of subpatterns happens in its part of input. At the end, we merge these pieces of information using dynamic programming to obtain the result.

Result 7 (theorem 3.6.1)**.** Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{‘*’}\}^m$ for $m = O(n^{1-x})$, there is an MPC algorithm to find the solve the string matching problem in $O(\log n)$ rounds using $O(n^x)$ machines.

(ii). When no subpattern is a prefix of another, then the number of different subpatterns matching each starting position is limited by 1. Exploiting this property, we can find out which subpattern matches each starting position, if any. Next, we create a graph with positions in T as vertices, and we put an edge from a position i which matches a subpattern P_k, to the minimum position j matching subpattern P_{k+1}, which is also located after $i + |P_k| - 1$. This way, ‘*’ wildcard string matching reduces to a graph connectivity problem; finding whether there is path from a position matching the first subpattern to a position matching the last subpattern. Therefore, we will have a $O(\log(n))$-round MPC algorithm for ‘*’ wildcard problem using the standard graph connectivity results in MPC [53].

Result 8 (Theorem 3.6.2)**.** Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{‘*’}\}^m$ such that subpatterns are not a prefix of each other, there is an MPC algorithm to find the solve the string matching problem in $O(\log n)$ rounds using $O(n^x)$ machines.

3.2 Preliminaries

In the pattern matching problem, we have two strings T and P of length n and m over an alphabet Σ. In the string matching problem, the first string T is a text, and the second string P is the pattern we are looking for in T. For a string s, we denote by $s[l, r]$ the substring of s from l to r, i.e., $s[l, r] = \langle s_l, s_{l+1}, \ldots, s_r \rangle$. Given two strings T and P, we are looking for all occurrences of the pattern P as a substring of T. In other words, we are looking for all $i \in [1, n - m + 1]$ such that $T[i, i + m - 1] = P$. In this case, we say substring $T[i, i + m - 1]$ matches pattern P.

Problem 3.2.1. Given two strings $T \in \Sigma^n$ and $P \in \Sigma^m$, we want to find all occurrences of the string P (pattern) as a substring of string T (text).

Problem 3.2.1 has been vastly studied in the literature, and there are a number of solutions that solve the problem in linear time. [55] However, the main focus of this paper is to design massively parallel algorithms for the pattern matching problem. We consider MPC as our parallel computation model, as it is a general framework capturing state-of-the-art parallel computing frameworks such as Hadoop MapReduce and Apache Spark. We will precisely define the model in Section **??**.

We extend the problem of string matching adding wildcard characters to the pattern. A wildcard character is a special character $\phi \notin \Sigma$ in pattern that is not required to match by the same character in T. For example, wildcard character '?' can be matched by any arbitrary character in T. For instance, let T and P be "abracadabra" and "a?a" respectively. Then, pattern P occurs at $i = 4$ and $i = 6$ since $P \overset{?}{\equiv} T[4,6]$ and $P \overset{?}{\equiv} T[6,8]$. Notation $\overset{\phi}{\equiv}$ denoted the equality of two strings regarding the wildcard character ϕ.

We consider three kinds of wildcard characters in this paper:

(i). Character replacement wildcard '?': Any character can match character replacement wildcard.

(ii). Character repetition wildcard '+': If character repeat wildcard appears immediately after a character c in the pattern, then any number of repetition for character c is accepted.

(iii). String replacement wildcard '*': A string replacement wildcard can be matched with any string of arbitrary length.

3.2.1 Hashing

To boost comparing the pattern p with the substrings of string s, we can compare the fingerprints or hashes instead. A hash of string s, denoted by $h(s)$, is a single number such that $h(s) \neq h(s')$ implies $s \neq s'$ always, and $h(s) = h(s')$ implies $s = s'$ with a high probability. Consider a naive hash function h_r such that $h_r(s) = \sum_{i=1}^{|s|} s_i \mod r$, where r is an arbitrary number. Note that we can label the characters in Σ with a permutation of size $|\Sigma|$, thereby, considering a numerical value for each character to simplify the formulas. This hash function does not guarantee a high probability of $s = s'$ in the case of $h_r(s) = h_r(s')$, because this hash function is not locally sensitive, i.e., modifying two consecutive characters in s in a way that the first one reduced by one and the other one increased by one gives us the same hash.

However, simple hash function h_r has a desirable property that is rolling. A rolling hash function h allows us to achieve $h(s[l+1, r+1])$ from $h(s[l,r])$ in $O(1)$ time, i.e., $h_r(s[l+1, r+1]) = h_r(s[l,r]) - s_l + s_{r+1} \mod r$. Using a rolling hash function, one can imagine a Monte-Carlo randomized algorithm for string matching problem by first computing $h(p)$ and $h(s_{1,m})$ in $O(m)$ time, and then building $h(s[i+1, i+m])$ from $h(s[i, i+m-1])$ for i from 1 to $n-m$ in $O(n)$ time. Afterwards, we can compare the hash of each substring with $h(p)$ in $O(1)$ time. Therefore, we desire a locally sensitive rolling hash function. Karp and Robbin [54] provided such rolling hash functions that guarantee a bounded probability of hash collision. An example of such hash function is

$$h_{r,b}(s) = \sum_{i=1}^{|s|} s_i b^{|s|-i} \mod r$$

The difference between this hash function and the previous one is local sensitivity. It's not likely anymore to end up with the same hash applying few modifications if we chose r to be a prime number or simply chose such r and b to be relatively prime numbers. Besides, the hash collision probability of an ideal hash function is r^{-1}, and no matter how we minimize the correlation of the hashes of similar strings, we cannot achieve a smaller probability. By the way, $h_{r,b}$ with relatively prime r and b achieve a probability in that this correlation is almost negligible.

Therefore, it suffices to pick a large enough r to guarantee high probability. More precisely, we want $\bigcup_{i=1}^{n-m+1} \Pr[P \neq T[i, i+m-1] \wedge h(P) = h(T[i, i+m-1])]$ be at most $1/n$. Thus,

$$\bigcup_{i=1}^{n-m+1} \Pr[P \neq T[i, i+m-1] \wedge h(P) = h(T[i, i+m-1])] \leq \frac{1}{n} \Rightarrow$$
$$\sum_{i=1}^{n-m+1} \Pr[P \neq T[i, i+m-1] \wedge h(P) = h(T[i, i+m-1])] \leq \frac{1}{n} \Rightarrow$$
$$O(n)\frac{1}{r} \leq \frac{1}{n} \Rightarrow r \geq O(n^2) \Rightarrow \log(r) \geq O(\log(n))$$

Therefore, the number of bits required to store hashes should be logarithmic in the size of the text which is a fairly reasonable assumption. Alternatively, we can reduce the probability substantially by comparing hashes based on multiple values of r. Hence, we can safely assume that using locally such sensitive hash functions, we can guarantee a high probability of avoiding hash collisions.

Fact 3.2.1. Given two strings $T \in \Sigma^n$ and $P \in \Sigma^m$, using locally sensitive rolling hash functions one can find all occurrences of the string P (pattern) as a substring of string S (text) in $O(n+m)$ time with a high probability.

In addition to rolling property, we can consider a more general property for $h_{r,b}$ namely partially decomposability. A hash function h is partially decomposable if it is possible to calculate in $O(1)$ time $h(s[l, r])$ from the hash of the prefixes of s, that is $h(s[1, i])$ for all $i \in [1, |s|]$. In addition, it should be possible to calculate the hash of the concatenation of two strings in $O(1)$ time. For example, $h_{r,b}(s[l, r]) = (h_{r,b}(s[1, r]) - h_{r,b}(s[1, l-1])b^{r-l+1}) \mod r$, and $h_{r,b}(s + s') = (h_{r,b}(s)b^{|s'|} + h_{r,b}(s')) \mod r$, where $s + s'$ means the concatenation of s and s'.

Throughout this paper, we refer to the suitable hash function for an instance of string matching problem by h and ignore the internal complications of the hash functions.

3.3 String matching without wildcard

In this section, we provide an algorithm for the string matching problem with no wildcards; Given strings T and P, we wish to find all the starting points in T that match P. The round complexity of our algorithm, which is the most important factor in MPC model, is constant. Furthermore, the algorithm is tight in a sense that the total running time and memory is linear. For the more restricted cases, we enhance our algorithm to work in even less rounds.

The algorithm consists of three stages. We discuss the different stages of the algorithm with details in the following. The main goal of the algorithm is to compare the hash of each length m substring of string T with the hash of string P, and therefore, find an occurrence if they are equal. We consider the following properties, namely partially decomposability, for our hash function h,

- Merging the hashes of two strings s and s', to find the hash of their concatenation $s + s'$, can be done in $O(1)$.
- Querying the hash of a substring $s[l, r]$ can be done in $O(1)$, with $O(|s|)$ preprocessing time.

Consider strings T and P are partitioned into several blocks of size $\mathcal{S}$, each fit into a single machine. We denote these blocks by $T^1, T^2, \ldots, T^{n/\mathcal{S}}$ and $P^1, P^2, \ldots, P^{m/\mathcal{S}}$. It is easy to solve the problem when P is small relative to $\mathcal{S}$. We can search for P in each block independently by maintaining the partial hash for each prefix of the block. The caveat of this solution is that some substrings lie in two machines. We can fix this issue by feeding the initial string chunks with overlap. If the length of the overlaps is greater than m, it is guaranteed than each candidate substring is contained as a whole in one of the initial string chunks. We call this method "Double Covering". Utilizing this method, we can propose an algorithm which leads to Observation 3.3.1. Therefore, the main catch of the algorithm is to deal with the matching when string P spans multiple blocks.

Observation 3.3.1. *Given $T \in \Sigma^n$ and $P \in \Sigma^m$ where $m = O(n^{1-x})$, there is an MPC algorithm for string matching problem with no wildcards in 1 MPC round using $O(n^x)$ machines in linear total running time.*

Proof. We provide for each machine, which holds T^i, the next m characters in the T, i.e., $T[i\mathcal{S}+1 : i\mathcal{S}+m]$, which is viable since $m = O(\mathcal{S})$. Therefore, we can compute the hash of each substring starting in T^i as well as P in each machine. See Algorithm 4 for more details. ☐

Algorithm 4: StringMatching(a)

```
Data: two array $T$ and $P$, where $m = O(\mathcal{S})$
Result: function $f : \{1, 2, \dots, n-m+1\} \to \{0, 1\}$ such that $f(i) = 1$ iff $T[i : i+m-1] = P$.
1 $f(i) \leftarrow 0 \quad \forall 1 \le i \le n-m+1$;
2 send a copy of $P$ as well as $T[i\mathcal{S}+1 : i\mathcal{S}+m]$ to the $i$-th machine, which holds $T^i$, truncate if $i\mathcal{S}+m$ surpasses $n$.
3 Run in parallel: for $1 \le i \le n/\mathcal{S}$ do
4     append $T[i\mathcal{S}+1 : i\mathcal{S}+m]$ to $T^i$;
5     for $1 \le j \le \mathcal{S}$ do
6         if $h(P) = h(T^i[j : j+m-1])$ then
7             $f((i-1)\mathcal{S}+j) \leftarrow 1$;
```

Theorem 3.3.1. *For any $x < 1/2$, given $T \in \Sigma^n$ and $P \in \Sigma^m$, there is an MPC algorithm for string matching problem with no wildcards in $O(1)$ MPC rounds using $O(n^x)$ machines in linear total running time.*

Intuitively, we can handle larger patterns by transferring the partial hash of each ending point to the machine which the respective starting point lies in. In addition, we compute the hash of each k first blocks, and by which, we can find the hash of each interval of blocks. Note that we need to assume $x < 1/2$, so that memory of a single machine be capable of storing $O(1)$ information regarding each machine. The algorithm is outlined in Algorithm 5.
Proof. Assume there are $(n+m)/\mathcal{S} + 2$ machines as following:

- machines $a_1, a_2, \dots, a_{n/\mathcal{S}}$ which T^i is stored in a_i.
- machines $b_1, b_2, \dots, b_{m/\mathcal{S}}$ which P^i is stored in b_i.
- machines c and d which are used for aggregation purposes.

At the first round, we compute the hash of each block T^i and P^i and send them to machines c and d respectively. Furthermore, we compute the partial hashes of each prefix of each block T^i, that is $h(T^i[1, j])$ for $1 \le j \le \mathcal{S}$. The calculated hash of each prefix should be sent to the machine containing the respective starting point. Therefore, $T^i[1, j]$ should be sent to the node containing starting point $(i-1)\mathcal{S} + j - m + 1$. The total communication overhead of this round is $O(n+m)$.

In the next round, we aggregate the calculated hashes in the previous round. In node d, we calculate the hash of string P by merging the hashes of $T^1, T^2, \ldots, T^{m/\mathcal{S}}$. The value of $h(P)$ should be sent to all a_i for $1 \leq i \leq n/\mathcal{S}$. Those nodes are supposed to compare $h(P)$ with the hash of each starting point lying in their block to find the matches in the last round. Furthermore, in node c, the hash of each first k blocks of T should be calculated, i.e., $h(T^1 + T^2 + \ldots + T^i)$ for all $1 \leq i \leq n/\mathcal{S}$. Each machine a_i needs to compute the hash of a sequence of consecutive blocks of T that lie between each starting point in T^i and its respective end point. Therefore, each machine should receive up to $O(1)$ of these hashes, because the set of respective ending points spans at most 2 blocks.

In the final round, we have all the data required for finding the matches with starting point inside block T^i at node a_i. It suffices to find the hash of each suffix of T^i, i.e., $h(T^i[l, \mathcal{S}])$, in which l is the candidate starting point. Consider T^k is the block where the ending point respective to l lies inside, and r is the index of the ending point inside T^k. Then, using the hash of the sequence of block between T^i and T^k, and $h(T^k[1, r])$ (both the hashes has been sent to a_i in the previous steps) one can decide whether the starting point l in block T^i is a match or not. □

Algorithm 5: StringMatching(b)

```
Data: two array $T$ and $P$
Result: function $f : \{1, 2, \ldots, n-m+1\} \to \{0, 1\}$ such that $f(i) = 1$ iff $T[i : i+m-1] = P$.
1  $f(i) \leftarrow 0 \quad \forall 1 \leq i \leq n-m+1$;
2  Run in parallel: for $1 \leq i \leq m/\mathcal{S}$ do
3      send $h(P^i)$ to machine $d$;
4  Run in parallel: for $1 \leq i \leq n/\mathcal{S}$ do
5      send $h(T^i)$ to machine $c$;
6      for $1 \leq j \leq \mathcal{S}$ do
7          send $h(T^i[1, j])$ to the machine $a_k$, where the corresponding starting point, i.e., $(i-1)\mathcal{S} + j - m + 1$, lies in
             $T^k$;
8  send $h(P) \leftarrow$ merge$(h(P^1), h(P^2), \ldots, h(P^{m/\mathcal{S}}))$ to each $a_i$ for $1 \leq i \leq n/\mathcal{S}$;
9  for $1 \leq i \leq n/\mathcal{S}$ do
10     $h(T^1 + T^2 + \ldots + T^i) \leftarrow$ merge$(h(T^1 + T^2 + \ldots + T^{i-1}), h(T^i))$;
11     send $h(T^1 + T^2 + \ldots + T^i)$ to each machine $a_j$ that needs it in the next round;
12 Run in parallel: for $1 \leq i \leq n/\mathcal{S}$ do
13     for $1 \leq l \leq \mathcal{S}$ do
14         $s \leftarrow (i-1)\mathcal{S} + l$;
15         let $k$ be the index of the machine where $s + m - 1$ lies in $T^k$;
16         $r \leftarrow s + m - 1 - (k-1)\mathcal{S}$;
17         $\eta = h(T^{i+1} + T^{i+2} + \ldots + T^{k-1})$;
18         $h(T[s, s+m-1]) \leftarrow$ merge$(h(T^i[l, \mathcal{S}]), \eta, h(T^k[1, r]))$;
19         if $h(T[s, s+m-1]) = h(P)$ then
20             $f(s) \leftarrow 1$;
```

3.4 Character Replace Wildcard '?'

We start this section summarizing the algorithm for string matching with '?' in sequential settings, which require Fast Fourier Transform.

Theorem 3.4.1 (Proven in [36]). *Given strings $T \in \Sigma^n$ and $P \in (\Sigma \cup \{?\})^m$, it is possible to find all the substrings of T that match with pattern P in $\widetilde{O}(n+m)$.*

The idea behind Theorem 3.4.1 is taking advantage of fast multiplication algorithms, e.g. using Fast Fourier Transform, to find the occurrences of P which contains '?' wildcard characters. Fischer and Paterson [36] proposed the first algorithm for string matching with '?' wildcard. The main idea is to replace each character $c \in \Sigma$ in string T or P with two consecutive numbers mp_c and $1/\mathsf{mp}_c$ and each '?' with two consecutive zeros. If we compute the convolution of T and the reverse of P, we can ensure a match if the convoluted value with respect to a substring of T equals the number non-wildcard characters in P, aka nz_P. This procedure requires a non-negligible precision in float arithmetic operations that adds a $\log(|\Sigma|)$ factor to the order of the algorithm. In the subsequent works, the dependencies on the size of alphabet has been eliminated. [52, 49, 29]

As stated in Theorem 3.4.1, we can solve string matching with '?' wildcards in $O(n\log(n))$ using convolution. Convolution can be computed in $O(n\log(n))$ by applying Fast Fourier Transform on both the arrays, performing a point-wise product, and then applying inverse FFT on the result. Therefore, we should implement FFT in a constant round MPC algorithm in order to solve pattern matching with wildcard '?'. Inverse FFT is also possible with the similar approach.

Given an array $A = \langle a_0, a_1, \ldots, a_{n-1}\rangle$, we want to find the Discrete Fourier Transform of array A. Without loss of generality, we can assume $n = 2^k$, for some k, to avoid the unnecessary complication of prime factor FFT algorithms; it is possible to right-pad by zeroes otherwise. By Fast Fourier Transform, applying radix-2 Cooley-Tukey algorithm for example, one can compute the Discrete Fourier Transform of A in $O(n\log(n))$ time, which is defined as:

$$a_k^* = \sum_{j=0}^{n-1} a_j \cdot e^{-2\pi ijk/n} \tag{3.1}$$

Where $A^* = \langle a_0^*, a_1^*, \ldots, a_{n-1}^*\rangle$ is the DFT of A. We interchangeably use the alternative notation $W_n = e^{-2\pi i/n}$, by which Equality 3.1 becomes

$$a_k^* = \sum_{j=0}^{n-1} a_j \cdot W_n^{jk} \tag{3.2}$$

We state the following theorem regarding the FFT in the MPC model.

Theorem 3.4.2. *For any $x \le 1/2$, a collection of $O(n^x)$ machines each with a memory of size $O(n^{1-x})$ can solve the problem of finding the Discrete Fourier Transform of A with $O(1)$ number of rounds in MPC model. The total running time equals $O(n\log(n))$, which is tight.*

Roughly speaking, our algorithm is an adaptation of Cooley-Tukey algorithm in the MPC model. In Subsection 3.4.1, we justify Theorem 3.4.2 by giving an overview of the algorithm, and then we show how it results in a $O(1)$-round algorithm for computing the DFT of an array in the MPC model.

Corollary 3.4.3. *Given strings $T \in \Sigma^n$ and $P \in (\Sigma \cup \{?\})^m$, it is possible to find all the substrings of T that match with pattern P with a $O(1)$-round MPC algorithm with total runtime and memory of $\widetilde{O}(n+m)$.*

Proof. Build vectors $T^\dagger$ and $P^\dagger$ as following:

$$T^\dagger = \langle \mathsf{mp}_{T_1}, \mathsf{mp}_{T_1}^{-1}, \mathsf{mp}_{T_2}, \mathsf{mp}_{T_2}^{-1}, \ldots, \mathsf{mp}_{T_n}, \mathsf{mp}_{T_n}^{-1}\rangle$$
$$P^\dagger = \langle \mathsf{mp}_{P_1}, \mathsf{mp}_{P_1}^{-1}, \mathsf{mp}_{P_2}, \mathsf{mp}_{P_2}^{-1}, \ldots, \mathsf{mp}_{P_n}, \mathsf{mp}_{P_n}^{-1}\rangle$$

Note that mp_c for all characters $c \in \Sigma$ has a positive integer value, and mp_c^{-1} equals $1/\mathsf{mp}_c$. However, let $\mathsf{mp}_? = \mathsf{mp}_?^{-1} = 0$ for wildcard character '?'. For example, for text "abracadabra" and pattern "a?a", considering mp_c equals the index of each letter in the English alphabet, we will have

$$T^\dagger = \langle 1, \frac{1}{1}, 2, \frac{1}{2}, 18, \frac{1}{18}, 1, \frac{1}{1}, 3, \frac{1}{3}, 1, \frac{1}{1}, 4, \frac{1}{4}, 1, \frac{1}{1}, 2, \frac{1}{2}, 18, \frac{1}{18}, 1, \frac{1}{1} \rangle$$
$$P^\dagger = \langle 1, \frac{1}{1}, 0, 0, 1, \frac{1}{1} \rangle$$

Let $\mathcal{C} = T^\dagger \circledast \mathsf{rev}(P^\dagger)$, where operator $\circledast$ shows the convolution of its two operands. We can observe that for all $1 \le i \le n - m + 1$ we have

$$\mathcal{C}_{2i+2m-1} = \sum_{j=1}^{2m} T^\dagger_{2(i-1)+j} \cdot P^\dagger_j \tag{3.3}$$

It is clear that if $T[i, i + m - 1]$ matches pattern P, then $\mathcal{C}_{2i+2m-1} = 2\mathsf{nz}_P$ since each the wildcard characters add up to 0, and for any other character c, $\mathsf{mp}_c \times 1/\mathsf{mp}_c + \mathsf{mp}_c \times 1/\mathsf{mp}_c = 2$. According to [36], the other side of this expression also holds, i.e., $T[i, i + m - 1]$ matches pattern P if $\mathcal{C}_{2i+2m-1} = 2\mathsf{nz}_P$. Therefore, to find that whether a substring $T[i, i + m - 1]$ matches pattern P or not, it suffices to

(i). Compute $\mathcal{C} = T^\dagger \circledast \mathsf{rev}(P^\dagger)$ utilizing FFT, which can be implemented in $O(1)$ rounds in MPC model as stated in Theorem 3.4.2.

(ii). For all $1 \le i \le n - m + 1$, substring $T[i, i + m - 1]$ matches pattern P iff $\mathcal{C}_{2i+2m-1} = 2\mathsf{nz}_P$.

□

3.4.1 $O(1)$-round algorithm for FFT

To find the $O(1)$ round algorithm in MPC model, we first review how one can find the Discrete Fourier Transform in $O(n \log(n))$ time. The following recurrence helps us to solve the problem using a divide and conquer algorithm. [32] Let $\Psi[2k] = \langle a_0, a_2, \ldots, a_{n-2} \rangle$, i.e., an array of the even-indexed elements of A, and $\Psi[2k + 1] = \langle a_1, a_3, \ldots, a_{n-1} \rangle$, an array of the odd-indexed elements of A likewise. If $\Psi[2k]^* = \langle \psi[2k]^*_0, \psi[2k]^*_1, \ldots, \psi[2k]^*_{n/2-1} \rangle$ and $\Psi[2k + 1]^* = \langle \psi[2k + 1]^*_0, \psi[2k + 1]^*_1, \ldots, \psi[2k + 1]^*_{n/2-1} \rangle$ show the DFT of $\Psi[2k]$ and $\Psi[2k + 1]$ respectively, then for all $0 \le j < n/2$

$$\begin{cases} a^*_j & = \psi[2k]^*_j + W^j_n \psi[2k + 1]^*_j \\ a^*_{j+n/2} & = \psi[2k]^*_j - W^j_n \psi[2k + 1]^*_j \end{cases} \tag{3.4}$$

Intuitively, the recurrence decomposes A into two smaller arrays of size $n/2$, $\Psi[2k]$ and $\Psi[2k+1]$, and represents the DFT of A based on $\Psi[2k]^*$ and $\Psi[2k+1]^*$. Accordingly, we can solve two smaller DFT problems, each with size $n/2$, and then merge them in $O(n)$ time to find A^*. Therefore, we can find A^* in $O(n \log(n))$ time since the depth of the recurrence is $O(\log(n))$. We cannot find a constant round MPC algorithm exclusively using this recurrence. To extend the recursion, we can define $\Psi[pk + q]$ as the following,

$$\Psi[pk+q] = \langle a_q, a_{p+q}, \dots, a_{(l-1)p+q} \rangle \quad \forall\ 0 \le q < p \le n$$

Where $l = \lfloor \frac{n-1-q}{p} \rfloor + 1$ is the number of the indices whose reminder is q in division by p. $\Psi[pk+q]$ contains all the elements of A whose index is q modulo p. Note that the notation of $\Psi[pk+q]$ assumes a universal quantification on all $0 \le k \le n/p$. We also show the DFT of $\Psi[pk+q]$ by

$$\Psi[pk+q]^* = \langle \psi[pk+q]^*_0, \psi[pk+q]^*_1, \dots, \psi[pk+q]^*_{l-1} \rangle$$

We can observe that for all $0 \le b \le \log_2(n)$ and $0 \le q < 2^{b+1}$, assuming $p = 2^b$, the following counterpart of Recurrence 3.4 holds for all $0 \le j < l$, where l is the length of $\Psi[2^b k + q]$ which equals $\log(n) - b$:

$$\begin{cases} \psi[2^b k+q]^*_j & = \psi[2^{b+1}k+q]^*_j + W_l^j \psi[2^{b+1}k+2^b+q]^*_j \\ \psi[2^b k+q]^*_{j+l/2} & = \psi[2^{b+1}k+q]^*_j - W_l^j \psi[2^{b+1}k+2^b+q]^*_j \end{cases} \tag{3.5}$$

Since $\Psi[2^{b+1}k+q]$ and $\Psi[2^{b+1}k+2^b+q]$ decompose $\Psi[2^b k+q]$ into the even-indexed and odd-indexed elements, Recurrence 3.5 is implied by plugging in $\Psi[2^b k+q]$ as A in Recurrence 3.4. The following Lemmas (3.4.4 and 3.4.5) point out two properties regarding FFT in the MPC model. These properties immediately give us an $O(\log(n))$-round algorithm for the FFT problem.

Lemma 3.4.4. *For all* $\log(\mathcal{M}) \le b \le \log(n)$ *and* $0 \le q < 2^b$*, the value of* $\Psi[2^b k+q]^*$ *could be computed in a single machine with memory of* $\mathcal{S}$*.*

Proof. If all of the entries of $\Psi[2^b k+q]$ exist in the memory of a single machine, which is viable since $b \ge \log(\mathcal{M})$, then we can compute $\Psi[2^b k+q]^*$ as following:

$$\Psi[2^b k+q]^* = \mathsf{fft}(\Psi[2^b k+q])$$

□

In many implementations of Fast Fourier Transform, bit-reversal technique is used to facilitate the algorithm from various perspectives, to achieve an in-place FFT algorithm for example. Bit-reversal technique reorders the initial array based on the reverse binary representation of the indices; Considering $(10100)_2$ as the sort key for a_5, $5 = (00101)_2$, when $n = 32$ for example. In other words, bit reversal technique permutes the elements of A by a permutation π such that $\pi(i) = \mathsf{rev}(i)$, where $\mathsf{rev}(i)$ is the reverse binary presentation of i. Applying bit reversal technique guarantees the locality of data, especially in the first $\log(\mathcal{S})$ stages.

Lemma 3.4.5. *The bit-reversal operation makes a permutation of input entries which if we it split into* 2^b *continuous chunks of equal size, then for all* $0 \le b \le \log(n)$*,* $\Psi[2^b k + \mathsf{rev}(q)]$ *would the q-th chunk.*

Proof. Notice that the bit-reversal permutation is actually sorting the entries based on the reverse binary representation of the index of each entry. Therefore, all indices with the same low b bits form a continuous interval of size $2^{\log(n)-b}$ in the bit-reversal permutation. Thus, $\Psi[2^b k + \mathsf{rev}(q)]$ which all of its members have the same low b bits, $\mathsf{rev}(q)$. Moreover, it is the q-th chunk in the bit-reversal permutation, because $\mathsf{rev}(q)$ ranks q-th in the bit-reversal ordering of all non-negative integers less than 2^b. □

Definition 3.4.6. For all $0 \leq q < \mathcal{M}$, Let $\Phi_q = \Psi[\mathcal{M}k + \mathsf{rev}(q)]$. ($\Phi_q$ shows the input of the q-th machine after performing the bit-reversal permutation, according to Lemma 3.4.5) We also show the DFT of Φ_q by Φ_q^+, instead of Φ_q^*. The j-th element of Φ_q (Φ_q^+) is denoted by $\phi_{q,j}$. ($\phi_{q,j}^+$)

Utilizing Lemma 3.4.4 and Lemma 3.4.5, we can achieve a $O(\log(n))$-round MPC algorithm which merges all Φ_q^+s in $\log(\mathcal{M})$ steps. At the first step, we apply bit-reversal permutation in order to gather each Φ_q in a single machine. Applying bit-reversal permutation does not have any special communication overhead, as we are just transferring the elements. Then we compute Φ_q^+ for all $q \in \mathbb{Z}_m$. Afterwards, in the l-th step, for $0 \leq l < \log(m)$, we can merge $2^{\log(m)-l}$ blocks of size $2^{\log(\mathcal{S})+l}$ using equation 3.5 in pairs.

It seems that $\log(n)$ MPC rounds is required because we need to merge Φ_q^+s which are distributed in various machines, and according to Equality 3.1, each a_k^* depends on every Φ_q^+ for $q \in \mathbb{Z}_m$. However, we can exploit the nice properties of the graph of dependencies of a_i^*s to Φ_q^+s, which is also called the *Butterfly Graph*, to decompose these dependencies efficiently. An example of this dependencies graph is demonstrated in Figure 3.1. We can concentrate these dependencies in single machines by gathering $\phi_{q,j}^+$s with the same j. Figure 3.1 demonstrates why it suffices to have $\phi_{q,j}^+$s with the same j stored in a single machine. In this figure, $\mathcal{S}$ and $\mathcal{M}$ are equal to 4, and the entries that should be stored in the same machine are colored with the same color. At the first step, the entries are distributed in bit-reversal permutation, and continuous chunks are stored in each machine. However, in the next step, we can see for example a_5^*, which is colored green, depends only on the green entries, which all have the same $j = 1$.

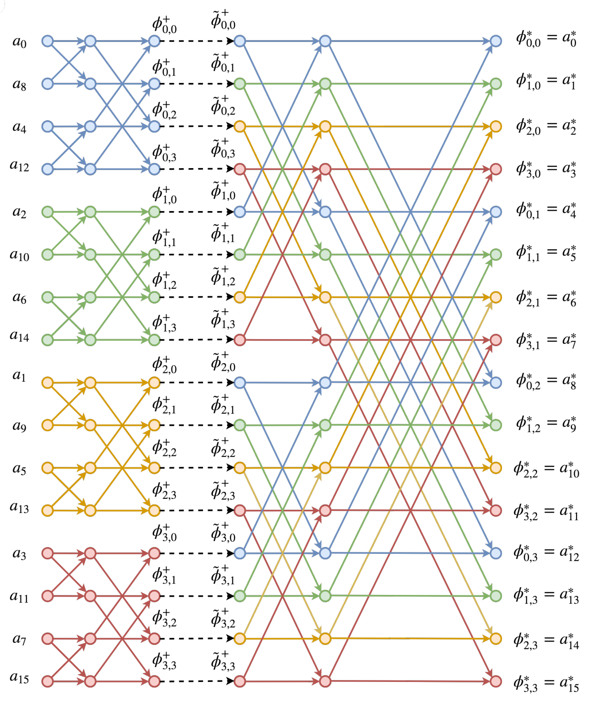
Definition 3.4.7. Let $\widetilde{\phi}_{q,j}^+ = W_n^{qj}\phi_{q,j}^+$. These coefficients are called twiddle factors, which allows us to express a_k^* based on the FFT of $\widetilde{\phi}_{q,j}^+$s. Let Φ_j^* be the DFT of vector $\langle \widetilde{\phi}_{0,j}^+, \widetilde{\phi}_{1,j}^+, \ldots, \widetilde{\phi}_{\mathcal{M}-1,j}^+ \rangle$. We will show in Lemma 3.4.8 that $a_{q\mathcal{S}+j}^*$ equals $\phi_{j,q}^*$, which is the j-th element of Φ_j^*.

Lemma 3.4.8. $a_{q\mathcal{S}+j}^* = \phi_{j,q}^*$.

Proof. We know that a_k^* equals a weighted sum of all a_j where weights are the n-th roots of unity to the power of k. i.e.,

$$a_k^* = \sum_{j=0}^{n-1} W_n^{jk} a_j$$

We break down the summation into $\mathcal{M}$ smaller summations of size $\mathcal{S}$ such that each smaller summation represent one Φ_i for some $0 \leq i \leq \mathcal{M} - 1$. Moreover, we show k by $q\mathcal{S} + j$, where $0 \leq j \leq \mathcal{S} - 1$. Hence, we can show the following.



a_0
a_8
a_4
a_{12}
a_2
a_{10}
a_6
a_{14}
a_1
a_9
a_5
a_{13}
a_3
a_{11}
a_7
a_{15}
$\phi^+_{0,0}$ $\tilde{\phi}^+_{0,0}$
$\phi^+_{0,1}$ $\tilde{\phi}^+_{0,1}$
$\phi^+_{0,2}$ $\tilde{\phi}^+_{0,2}$
$\phi^+_{0,3}$ $\tilde{\phi}^+_{0,3}$
$\phi^+_{1,0}$ $\tilde{\phi}^+_{1,0}$
$\phi^+_{1,1}$ $\tilde{\phi}^+_{1,1}$
$\phi^+_{1,2}$ $\tilde{\phi}^+_{1,2}$
$\phi^+_{1,3}$ $\tilde{\phi}^+_{1,3}$
$\phi^+_{2,0}$ $\tilde{\phi}^+_{2,0}$
$\phi^+_{2,1}$ $\tilde{\phi}^+_{2,1}$
$\phi^+_{2,2}$ $\tilde{\phi}^+_{2,2}$
$\phi^+_{2,3}$ $\tilde{\phi}^+_{2,3}$
$\phi^+_{3,0}$ $\tilde{\phi}^+_{3,0}$
$\phi^+_{3,1}$ $\tilde{\phi}^+_{3,1}$
$\phi^+_{3,2}$ $\tilde{\phi}^+_{3,2}$
$\phi^+_{3,3}$ $\tilde{\phi}^+_{3,3}$
$\phi^*_{0,0} = a^*_0$
$\phi^*_{1,0} = a^*_1$
$\phi^*_{2,0} = a^*_2$
$\phi^*_{3,0} = a^*_3$
$\phi^*_{0,1} = a^*_4$
$\phi^*_{1,1} = a^*_5$
$\phi^*_{2,1} = a^*_6$
$\phi^*_{3,1} = a^*_7$
$\phi^*_{0,2} = a^*_8$
$\phi^*_{1,2} = a^*_9$
$\phi^*_{2,2} = a^*_{10}$
$\phi^*_{3,2} = a^*_{11}$
$\phi^*_{0,3} = a^*_{12}$
$\phi^*_{1,3} = a^*_{13}$
$\phi^*_{2,3} = a^*_{14}$
$\phi^*_{3,3} = a^*_{15}$

Figure 3.1: The dependency graph of the FFT recurrence in MPC model. Different colors represent different machines, in a setting with 4 machines with memory of 4 each. It could be observed that the dependency graph in each machine at the first stage is isomorphic to the one of the second stage.

$$
\begin{aligned}
a^*_{q\mathcal{S}+j} &= \sum_{z=0}^{\mathcal{M}-1}\sum_{l=0}^{\mathcal{S}-1} W_n^{(l\mathcal{M}+z)(q\mathcal{S}+j)} a_{l\mathcal{M}+z} && (3.6)\\
&= \sum_{z=0}^{\mathcal{M}-1}\sum_{l=0}^{\mathcal{S}-1} W_n^{l\mathcal{M}q\mathcal{S}} W_n^{l\mathcal{M}j} W_n^{zq\mathcal{S}} W_n^{zj} a_{l\mathcal{M}+z} && (3.7)\\
&= \sum_{z=0}^{\mathcal{M}-1}\sum_{l=0}^{\mathcal{S}-1} W_{\mathcal{S}}^{lj} W_{\mathcal{M}}^{zq} W_n^{zj} a_{l\mathcal{M}+z} && (3.8)\\
&= \sum_{z=0}^{\mathcal{M}-1} W_{\mathcal{M}}^{zq} W_n^{zj} \sum_{l=0}^{\mathcal{S}-1} W_{\mathcal{S}}^{lj} a_{l\mathcal{M}+z} && (3.9)\\
&= \sum_{z=0}^{\mathcal{M}-1} W_{\mathcal{M}}^{zq} W_n^{zj} \phi^+_{z,j} && (3.10)\\
&= \sum_{z=0}^{\mathcal{M}-1} W_{\mathcal{M}}^{zq} \widetilde{\phi}^+_{z,j} = \phi^*_{j,q} && (3.11)
\end{aligned}
$$

Equality 3.8 is implied from the fact that $W^k_{kn} = W_n$, and therefore, $W_n^{l\mathcal{M}p\mathcal{S}} = W_n^{lpn} = W_1^{lp} = 1$, $W_n^{l\mathcal{M}j} = W_{\mathcal{S}}^{lj}$, and $W_n^{zp\mathcal{S}} = W_{\mathcal{M}}^{zp}$. $\square$

Utilizing Lemma 3.4.8, we can provide an algorithm which solves the FFT problem in $O(1)$ MPC rounds, because we already know how to compute $\phi^*_{j,q}$s. Algorithm 6 shows the algorithm in details. It can be observed that our algorithm is analogous to Cooley-Tukey algorithm with radix n^x.

Algorithm 6: mpc-fft(b)

```
Data: an array $a$.
Result: array $a^*$ containing the DFT of $a$.
1 permute $a$ such that members of $\Phi_q$ be together in a single machine for all $q \in \mathbb{Z}_{\mathcal{M}}$.
2 Run in parallel: for $q \in \mathbb{Z}_{\mathcal{M}}$ do
3     $\Phi^+_q \leftarrow \mathsf{fft}(\Phi_q)$;
4     $\widetilde{\phi}^+_{q,j} \leftarrow W^{qj}_n \phi^+_{q,j} \quad \forall j \in \mathbb{Z}_{\mathcal{S}}$;
5 distribute $\widetilde{\phi}^+_{q,j}$ into different machines such that entries with same $j$ be in a same machine.
6 Run in parallel: for $j \in \mathbb{Z}_{\mathcal{S}}$ do
7     $\Phi^*_j \leftarrow \mathsf{fft}(\langle \widetilde{\phi}^+_{0,j}, \widetilde{\phi}^+_{1,j}, \dots, \widetilde{\phi}^+_{\mathcal{M}-1,j} \rangle)$;
8 permute all $\phi^*_{j,q}$ $(= a^*_{q\mathcal{S}+j})$ in a way that each of them retain its correct position.
```

Proof. [Theorem 3.4.2] At the first step of the algorithm, we apply the bit-reversal permutation in the input, thereby grouping the members of Φ_q in each of $\mathcal{M}$ machines. Applying a permutation imposes a communication overhead of $O(\mathcal{S})$ for each machine. Now, the q-th machine can compute Φ^+_q and $\widetilde{\Phi}^+_j$ standalone in $\widetilde{O}(n)$ time as following:

- $\Phi^+_q = \mathsf{fft}(\Phi_q)$
- $\widetilde{\Phi}^+_{q,j} = W^{qj}_n \Phi^+_{q,j}$

In the next round, we distribute $\widetilde{\Phi}^+_{q,j}$s with the same j in the same machines. Note that $\widetilde{\Phi}^+_{q,j}$s with the same j fit into a single machine since $x \leq 1/2$. We even might need to fit multiple groups of $\widetilde{\Phi}^+_{q,j}$s into a single machine, which causes no problem. By the way, we can compute Φ^*_js in this stage in $\widetilde{O}(n)$ time. Afterwards it suffices restore appropriate a^*_ks to their original position, where for $k = qS + j$, $a^*_k = \phi^*_{j,q}$. □

Observation 3.4.1. *Although Algorithm 6 requires* 3 *rounds of communication for computing DFT in the MPC model, the convolution of two arrays can be computed in* 4 *communication rounds.*

Proof. Since we need to apply FFT twice for computing the convolution, we can find the convolution in 6 communication rounds. However, since the first round applies a permutation on the elements, and the last round applies the inverse permutation, we can ignore the last round of computing the FFT, along with the first round of computing the inverse FFT. The point-wise product operation which is needed two be done between two FFTs allows us to do so, as it is independent of the order of the arrays. □

3.5 Character Repetition Wildcard '+'

The character repetition wildcard, shown by '+', allows expressing "an arbitrary number of a single character" within the pattern. An occurrence of '+' which is immediately after a character c, matches arbitrary repetition of character c. For example, text "bookkeeper" has a substring that matches pattern "oo+k+ee+", but neither of its substrings won't match pattern "oo+kee+". In Subsection 3.5.1, we reduce pattern matching with wildcard '+' to greater-than matching in

$O(1)$ MPC rounds. The greater-than matching problem is defined with further details in Problem 3.5.1. Afterwards, we show a reduction of greater-than matching from subset matching problem, explained in 3.5.3. Then, by showing subset matching could be implemented in $O(1)$ MPC rounds, we propose an $O(1)$-round MPC algorithm for string matching with wildcard '+' in Theorem 3.5.5.

3.5.1 Relation to greater-than matching

Problem 3.5.1. *Greater-than matching:* Given two arrays T and P, of length n and m respectively, find all indices $1 \le i \le n - m + 1$ such that the continuous subarray of T starting from i and of length m is element-wise greater than P, i.e., $T_{i+j-1} \ge P_j \ \ \forall 1 \le j \le m$.

To reduce pattern matching with wildcard '+' to greater-than matching, we perform run-length encoding 3.5.2 on both the text and the pattern. This way, we have a sequence of letters each along with a number showing the number of its repetitions, or a lower-bound restricting the number of repetitions in case we have a '+' wildcard. Subsequently, we can solve the problem for letters and numbers separately, and merge the result afterwards. Note that pattern P matches a substring of T if and only if the both the respective letters and the respective repetition restrictions match. We also show we can find the run-length encoding of a string in $O(1)$ MPC rounds in Observation 3.5.1.

Definition 3.5.2. For an arbitrary string s, let s° be the run-length encoding of s, computed in the following way:

- Ignoring '+' characters, decompose string s into maximal blocks consisting of the same character representing by pairs $\langle c_i, \mathsf{cnt}_i \rangle$ which show a block of cnt_i repetitions of character c_i.
- If a '+' character is located immediately after a block or within a block, that block becomes a wildcard block, and represented as $\langle c_i, \mathsf{cnt}_i + \rangle$, where cnt_i is still the number of the occurrences of character c_i in the block.
- s° equals the list of these pairs $\langle c_i, \mathsf{cnt}_i \rangle$ or $\langle c_i, \mathsf{cnt}_i + \rangle$, concatenated in a way that preserves the original ordering of the string.

For example, for $T =$ "bookkeeper" and $P =$ "o+o+k+ee+p",

$$T^\circ = \langle \langle \mathsf{b}, 1 \rangle, \langle \mathsf{o}, 2 \rangle, \langle \mathsf{k}, 2 \rangle, \langle \mathsf{e}, 2 \rangle, \langle \mathsf{p}, 1 \rangle, \langle \mathsf{e}, 1 \rangle, \langle \mathsf{r}, 1 \rangle \rangle$$
$$P^\circ = \langle \langle \mathsf{o}, 2+ \rangle, \langle \mathsf{k}, 1+ \rangle, \langle \mathsf{e}, 2+ \rangle, \langle \mathsf{p}, 1 \rangle \rangle$$

We alternatively show the compressed string as a string, for example,

- $T^\circ =$ "b[1]o[2]k[2]e[2]p[1]e[1]r[1]".
- $P^\circ =$ "o[2+]k[1+]e[2+]p[1]".

Observation 3.5.1. *We can perform run-length encoding in $O(1)$ MPC rounds.*

Proof. Suppose string s is partitioned among $\mathcal{M}$ machines such that i'th machine contains s_i for $1 \le i \le \mathcal{M}$. Run-length encoding of each s_i could be computed separately inside each machine. Let

$$s_i^\circ = \langle \langle c_{i,1}, cnt_{i,1} \rangle, \langle c_{i,2}, cnt_{i,2} \rangle, \ldots, \langle c_{i,l_i}, cnt_{i,l_i} \rangle \rangle$$

be the run-length encoding of s_i, where l_i is its length. Then in the next round, we can merge $\langle c_{i,l_i}, cnt_{i,l_i}\rangle$ and $\langle c_{i+1,1}, cnt_{i+1,1}\rangle$ if $c_{i,l_i} = c_{i+1,1}$ for all $1 \le i \le \mathcal{M} - 1$, and we will end up with s° if we concatenate all s_i°s. $\square$

As we mentioned before, in order to reduce from greater-than matching, it is possible to divide the problem into two parts: matching the letters, and ensuring whether the repetition constraints are hold, and solve each part separately. The former is a simple string matching problem, but the latter requires could be solved with greater-than matching. Formally, we need to find all indices $1 \le i \le n - m + 1$ such that for every $1 \le j \le m$, the following constraints holds:

(i). $T^\circ_{i+j-1} = \langle c_j, x\rangle$ for some $x \ge \mathsf{cnt}_j$ if $P^\circ_j = \langle c_j, \mathsf{cnt}_j+\rangle$.

(ii). $T^\circ_{i+j-1} = \langle c_j, \mathsf{cnt}_j\rangle$ if $P^\circ_j = \langle c_j, \mathsf{cnt}_j\rangle$ and $2 \le j \le m-1$.

(iii). $T^\circ_{i+j-1} = \langle c_j, x\rangle$ for some $x \ge \mathsf{cnt}_j$ if $P^\circ_j = \langle c_j, \mathsf{cnt}_j\rangle$ and $j \in \{1, m\}$.

Constraint (iii) needs to be considered to allow the substring in the original text T starts and ends in the middle of a block of c_1 or c_m letters. By considering only those substrings which their letters match, we can get rid of letter constraints. Also to ensure constraint (ii), we can perform a wildcard '?' pattern matching by replacing each cnt_j+ and also cnt_1 and cnt_m by a '?' wildcard, and keep only the numbers that constraint (ii) checks. Thus, if a substring $T^\circ_{i,i+m-1}$ matches according to this wildcard '?' pattern matching, what remains is a greater-than matching, as we only need to check constraints (i) and (iii), and we can replace numbers we already checked for constraint (ii) by some 0's.

Observation 3.5.2. *We can reduce pattern matching with wildcard '+' from greater-than matching in $O(1)$ MPC rounds.*

Using Observation 3.5.1, it only remains to perform $O(1)$ wildcard '?' matchings to obtain an instance of greater-than matching that already satisfies constraint (ii), as well as letter constraints. Thus, Observation 3.5.2 is implied.

3.5.2 Reduction from subset matching

Problem 3.5.3. *Subset Matching:* Given T, a vector of n subsets of the alphabet Σ, and P, a vector of m subsets in the same format as T, find all occurrences of P in T. P is occurred at position i in T if for every $1 \le j \le |P|$, $P_j \subseteq T_{i+j-1}$.

The subset matching problem is a variation of pattern matching where pattern matches text if each of the pattern entries is a subset of the respective entries in text. Keeping this in mind, we use subset matching to solve greater-than matching in $O(1)$ MPC rounds, and thereby achieving a $O(1)$-round solution for the problem of pattern matching with '+' wildcard character using the subset matching algorithm proposed by Cole and Hariharan [30].

Observation 3.5.3. *We can reduce greater-than matching to subset matching in $O(1)$ MPC rounds with total runtime and total memory of $O(Q)$, where Q is the sum of all T_t's, i.e. $Q = \sum_{i=1}^{m} T_i$.*

The idea behind Observation 3.5.3 is to have a subset $\{0, 1, 2, \ldots, T_i\}$ instead of each entry of T_i, and a subset $\{P_i\}$ instead of each P_i. This way if T matches P in a position $1 \le i \le n - m + 1$, then we have $P_j \in \{0, 1, 2, \ldots, T_{i+j-1}\}$ since $P_j \le T_{i+j-1}$ for all $1 \le j \le m$. This way, we can solve pattern matching with wildcard '+' in $O(1)$ MPC rounds if subset matching could be solved in $O(1)$ MPC rounds. In the following, Lemma 3.5.4 shows it is possible to solve subset matching w.h.p in a constant number of MPC rounds. The algorithm utilizes $O(log(n))$ invocations of wildcard '?' matching.

Lemma 3.5.4. *We can solve subset matching in $O(1)$ MPC rounds with total runtime and total memory of $\widetilde{O}(Q)$, where Q is the sum of the size all T_i's, i.e. $Q = \sum_{i=1}^{n} |T_i|$.*

Proof. First, consider solving an instance of the greater than matching problem in $O(1)$ MPC rounds. We partition the entries inside all T^i's into Σ sequences $T^{i,1}, T^{i,2}, \dots, T^{i,\Sigma}$ inside each machine, where $T^{i,j} = \{k \mid j \in T_{iS+k}\}$. We just create a subset of these sets which are not empty, i.e., $\mathcal{T}_i = \{T^{i,j} \mid |T^{i,j}| > 0\}$, so that $\mathcal{T}_i$ fits inside the memory of each machine. $\mathcal{P}_i$ also can be defined similarly. We can sort all the union of all $\mathcal{T}_i$'s and $\mathcal{P}_i$'s in a non-decreasing order of j and breaking ties using i in $O(1)$ MPC rounds [43].

This way, we end up with a sparse wildcard matching for each character in Σ, because we want to check in which substrings of T each occurrence of character $j \in \Sigma$ in P is contained in the corresponding T_i. We can put a '1' instead of each occurrence of j in P, a '?' instead of each j in T, and a '0' in all other places since we are considering a greater than matching instance. Using the similar algorithm as in [30], we can easily reduce each instance of sparse wildcard matching to $O(\log(k))$ normal wildcard matching with size of $O(k)$ in $O(1)$ MPC rounds, where k is the number of non-zero entries. Using Corollary 3.4.3, we can give a $O(1)$ round MPC algorithm for subset matching when the input is an instance of greater than matching. We can easily extend these techniques to solve subset matching algorithm in $O(1)$ MPC rounds, as Cole and Hariharan [30] showed it is analogous to sparse wildcard matching. □

Theorem 3.5.5. *There exists an MPC algorithm that solves string matching with '+' wildcard in constant rounds. The total memory and the total running time of the algorithm are $\widetilde{O}(n)$.*

Proof. We can also simplify the algorithm for pattern matching with wildcard '+', by exploiting subset matching flexibility. We can also use subset matching to verify constraint (ii) of greater-than matching reduction (instead of wildcard '?' pattern matching), as well as letter constraint (instead of regular string matching with no wildcard). We define T' and P' as follows:

$$T'_i = \bigcup_{j=1}^{cnt_i} \{\langle c_i, j+\rangle\} \cup \{\langle c_i, cnt_i\rangle\} \qquad \text{if } T^\circ_i = \langle c_i, cnt_i\rangle \tag{3.12}$$

$$P'_i = \{\langle c_i, cnt_i\rangle\} \qquad \text{if } P^\circ_i = \langle c_i, cnt_i\rangle \tag{3.13}$$

$$P'_i = \{\langle c_i, cnt_i+\rangle\} \qquad \text{if } P^\circ_i = \langle c_i, cnt_i+\rangle \tag{3.14}$$

It could easily be observed that subset matching of T' and P' is equivalent to pattern matching with wildcard '+' of T and P, and also this simpler reduction is straight-forward to implement in $O(1)$ MPC rounds. In addition, the total runtime and memory of this subset matching which is $\widetilde{O}(Q)$ is equal to $\widetilde{O}(n)$ in the original input. Note that T' is resulted form T° whose sum of its numbers, that each of them shows the repetitions of the respective letter, equals n. □

3.6 String Replace Wildcard '*'

In this section we consider the string matching problem with wildcard '*'.

Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{'*'}\}^m$, we say a substring of p is a subpattern if it is a maximal substring not containing '*'. We present the following results in this section:

(i). A sequential algorithm for string matching with wildcard '*' in time $O(n + m)$.

(ii). An MPC algorithm for string matching with wildcard '*' in $O(\log n)$ rounds using $O(n^x)$ machines if the length of pattern p is at most $O(n^{1-x})$.

(iii). An MPC algorithm for string matching with wildcard '*' in $O(\log n)$ rounds using $O(n^x)$ machines if all the subpatterns of p are not prefix of each other.

3.6.1 Linear time sequential algorithm

Observation 3.6.1. *Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{'*'}\}^m$, there is a sequential algorithm to decide if s matches with pattern p in time $\tilde{O}(n+m)$.*

Let the subpatterns of p to be $P_1, P_2, \ldots, P_w$. Our sequential algorithm is StringMatchingWithStar(a).

Algorithm 7: StringMatchingWithStar(a)

```
Data: $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{'*'}\}^m$.
Result: Yes or No.
1 Set $i \leftarrow 1$;
2 for $j = 1, 2, \ldots, w$ and $i \leq n$ do
3     Run KMP for the string $s[i, n]$ and pattern $P_j$;
4     If KMP fails, then Return No;
5     Let $i'$ be the position satisfying $s[i', i' + |P_j| - 1] = P_j$ obtained by KMP;
6     Set $i \leftarrow i' + |P_j|$;
7 Return Yes.
```

3.6.2 MPC algorithm for small subpattern

Theorem 3.6.1. *Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{'*'}\}^m$ for $m = O(n^{1-x})$, there is an MPC algorithm to find the solve the string matching problem in $O(\log n)$ rounds using $O(n^x)$ machines.*

We assume string s is partitioned into $s_1, s_2, \ldots, s_t$ for some $t = O(n^x)$ such that every s_i has length at most $O(n^{1-x})$. We say string s is an exact matching of p if there is a partition of s into $|p|$ (possibly empty) substrings such that if $p[i]$ is not '*', then i-th substring is same to $p[i]$.

Given indices $i, j \in [t]$ of string s and position k of pattern p, let $f(i, j, k)$ be the largest position of p such that the concatenation of $s_i, s_{i+1}, \ldots, s_j$ matches pattern $p[k, f(i, j, k)]$.

To illustrate our idea, we need the following definitions:

(i). $g(k)$ for position k of p: the largest position which is smaller than or equal to k such that $p[k]$ is '*'.

(ii). $h(k)$ for position k of p such that $p[k] \neq$ '*': the smallest integer r such that $p[g(k)+1, k-r]$ is a prefix of $p[g(k)+1, k]$.

Consider the following equation for an arbitrary $i \leq i' < j$ (let $\beta = f(i, i', k)$)

- if $p[\beta] =$ '*':
$$f(i, j, k) = f(i'+1, j, \beta)$$
- if $p[\beta] \neq$ '*':

$$f(i,j,k) = \max\begin{cases} f(i'+1,j,g(\beta)), \\ \max_{0\le \ell \le \lfloor \frac{\beta-h(\beta)}{g(\beta)} \rfloor}\{f(i'+1,j,\beta-\ell\cdot h(\beta))\} \end{cases} \tag{3.15}$$

Our algorithm is StringMatchingWithStar(b).

Proof. [of 3.6.1] We show that Eq. 3.15 correctly computes $f(i,j,k)$ for an arbitrary $i \le i' < j$. Then the theorem follows from the description of StringMatchingWithStar(b), Eq. 3.15 and Observation 3.6.1.

Let α be the largest position of p such that there is an exact matching of the concatenation of $s_i,\dots,s_j$ and $p[k,\alpha]$. If $f(i,i',\cdot)$ and $f(i'+1,j,\cdot)$ are correct, and $f(i,j,k)$ is computed by Eq. 3.15, then $f(i,j,k) \le \alpha$, since Eq. 3.15 implies a feasible exact matching.

Now we show that $f(i,j,k) \ge \alpha$ if $f(i,j,k)$ is computed by Eq. 3.15. There is an exact matching of the concatenation of $s_i,\dots,s_j$ and $p[k,\alpha]$. Let γ be the position of pattern p such that the last symbol of $s_{i'}$ is matched to $p[\gamma]$. By the definition of function f, $\gamma \le \beta$.

Consider the case of $p[\gamma] =$'*' or $p[\gamma] \ne$'*' but $p[\gamma] =$'*'. We have $g(\beta) \ge \gamma$ and $f(i'+1,j,\gamma) = \alpha$. By the monotone property of f, we have $f(i,j,k) \ge f(i'+1,j,g(\beta)) \ge f(i'+1,j,\gamma) = \alpha$.

Consider the case of $p[\gamma] \ne$'*' and $p[\beta] \ne$'*'. If γ and β are in different subpatterns, then using above argument, we have $f(i,j,k) \ge \alpha$. Otherwise, $p[h(\beta)+1,\gamma]$ is a suffix of $p[h(\beta)+1,\beta]$, which implies that there is a non-negative integer ℓ such that $\gamma = \beta - \ell \cdot h(\beta)$. Hence, $f(i,j,k) \ge \alpha$. □

Algorithm 8: StringMatchingWithStar(b)

```
Data: two array $s$ and $p$.
Result: Yes or No.
1 Distribute $s_1, s_2, \dots, s_t$ to distinct machines, and distribute $p$ to every machine;
2 Compute $f(i,i,k)$ for all the $k$ on the machine containing $s_i$ by algorithm StringMatchingWithStar($a$) in parallel;
3 for $m = 1, 2, \dots, \lceil \log_2 t \rceil$ do
4     For every $i \ge 1$, put $f(i, i+2^{m-1}-1, k)$ and $f(i+2^{m-1}, i+2^m-1, k)$ into same machine for all the $k$ in parallel;
5     For every $i \ge 1$, compute $f(i, i+2^m-1, k)$ for all the $k$ by Equation 3.15 with $j = i+2^m-1$ and $i' = i+2^{m-1}-1$ in parallel;
6 Return Yes if $f(1,t,1) = w$, otherwise return No.
```

3.6.3 MPC algorithm for non-prefix subpatterns

Theorem 3.6.2. *Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{'*'}\}^m$ such that subpatterns are not a prefix of each other, there is an MPC algorithm to find the solve the string matching problem in $O(\log n)$ rounds using $O(n^x)$ machines.*

Proof. We show that algorithm StringMatchingWithStar(c) solves the problem.

We first prove the correctness of the algorithm. Since all the subpatterns are not a prefix of each other, for every position i of string s, there is at most one subpattern P_u such that P_u is a prefix of $s[i,n]$. On the other hand, if $s[i,j]$ is not a prefix of any subpattern, then $h(s[i,j])$ does not equal to any of the hash value obtained in Step 1, otherwise, $h(s[i,j])$ is equal to some hash value obtained in Step 1. Hence, for every $i \in [n]$, the "for" loop of Step 2 finds the subpattern P_u such that P_u is a prefix of $s[i,n]$ by binary search if P_u exists.

If string s matches pattern p, then any set of positions $a_1, a_2, \dots, a_u$ with the following two conditions

(i). P_i is a prefix of $s[a_i,n]$ for every $i \in [w]$.

(ii). $a_i + |P_i| \le a_{i+1}$ for every $i \in [w-1]$.

corresponds to a matching between s and p. Hence, string s matches pattern p if and only if v_0 and v_{m+1} are connected in the constructed graph of Step 15 and 16.

Now we consider the number of MPC rounds required. Using the argument of Section **??**, computing hash of all the prefixes of every subpattern or a set of n substrings of s needs constant MPC rounds. Hence Step 1 needs constant rounds, and the "for" loop of Step 2 needs $O(\log n)$ rounds. Step 15 naturally needs a single round. Step 16 can be done in $O(\log n)$ rounds by sorting all the $(i, f(i))$ pairs according to the f function values and selecting j such that $j \geq i + |P_{f(i)}|$ and $f(j) = f(i) + 1$ for all the pairs $(i, f(i))$. The graph connectivity needs $O(\log n)$ rounds. ☐

Algorithm 9: StringMatchingWithStar(c)

```
   Data: two array $s$ and $p$.
   Result: Yes or No.
 1 For each subpattern $P_i$ in parallel compute the hash value of every prefix of $p_i$;
 2 for each position $i$ of string $s$ in parallel do
 3     Set $j = 0$ and $k = n$ initially;
 4     while $j < k$ do
 5         Let $\ell = \lceil (j + k)/2 \rceil$;
 6         Compute the hash $h(s[i, i + \ell])$;
 7         if there is a hash obtained in step 2 same to $h(s[i, i + \ell])$ then
 8             Set $j \leftarrow \ell$;
 9         else
10             Set $k \leftarrow \ell - 1$;
11     if there is a subpattern $p_u$ same to $s[i, i + j]$ then
12         Set $f(i) \leftarrow u$;
13     else
14         Set $f(i) \rightarrow 0$;
15 Construct an empty graph in parallel with vertices $v_0, v_1, \ldots, v_{n+1}$. Add edge $(v_0, v_s)$ and $(v_t, v_{n+1})$ where $s$ is the
     smallest integer such that $f(s) = 1$, $t$ is the largest integer such that $f(t) = w$;
16 For every $i \in [n]$ such that $f(i) > 0$, in parallel add edge $(v_i, v_j)$ to the graph where $j$ is the smallest integer such
     that $j \geq i + |P_{f(i)}|$ and $f(j) = f(i) + 1$;
17 Run graph connectivity algorithm on the graph constructed. return Yes if $v_0$ and $v_{n+1}$ are in the same connected
     component of the graph, otherwise return No ;
```

Bibliography

[1] Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. (2008). Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 147–160. ACM.

[2] Ahmed, R. and Boutaba, R. (2007). Distributed pattern matching: a key to flexible and efficient p2p search. *IEEE Journal on selected areas in communications*, 25(1).

[3] Ahn, K. J. and Guha, S. (2015). Access to Data and Number of Iterations: Dual Primal Algorithms for Maximum Matching under Resource Constraints. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 202–211.

[4] Alon, N., Babai, L., and Itai, A. (1986). A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms*, 7(4):567–583.

[5] Andoni, A., Nikolov, A., Onak, K., and Yaroslavtsev, G. (2014). Parallel algorithms for geometric graph problems. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 574–583.

[6] Andoni, A., Song, Z., Stein, C., Wang, Z., and Zhong, P. (2018). Parallel graph connectivity in log diameter rounds. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 674–685.

[7] Arjomandi, E. (1982). An efficient algorithm for colouring the edges of a graph with $\Delta+$ 1 colours. *INFOR: Information Systems and Operational Research*, 20(2):82–101.

[8] Assadi, S., Bateni, M., Bernstein, A., Mirrokni, V. S., and Stein, C. (2019a). Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. *Proceedings of the 30th annual ACM-SIAM Symposium on Discrete Algorithms (SODA), to appear.*

[9] Assadi, S., Chen, Y., and Khanna, S. (2019b). Sublinear Algorithms for $(\Delta+1)$ Vertex Coloring. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 767–786.

[10] Backurs, A., Indyk, P., and Schmidt, L. (2017). Better approximations for tree sparsity in nearly-linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2215–2229. SIAM.

[11] Barenboim, L., Elkin, M., Pettie, S., and Schneider, J. (2016). The Locality of Distributed Symmetry Breaking. *J. ACM*, 63(3):20:1–20:45.

[12] Baron, J., El Defrawy, K., Minkovich, K., Ostrovsky, R., and Tressler, E. (2012). 5pm: Secure pattern matching. In *International Conference on Security and Cryptography for Networks*, pages 222–240. Springer.

[13] Bateni, M., Behnezhad, S., Derakhshan, M., Hajiaghayi, M., Kiveris, R., Lattanzi, S., and Mirrokni, V. S. (2017). Affinity Clustering: Hierarchical Clustering at Scale. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 6867–6877.

[14] Bateni, M., Hajiaghayi, M., Seddighin, S., and Stein, C. (2018). Fast algorithms for knapsack via convolution and prediction. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1269–1282. ACM.

[15] Beame, P., Koutris, P., and Suciu, D. (2017). Communication Steps for Parallel Query Processing. *J. ACM*, 64(6):40:1–40:58.

[16] Behnezhad, S., Derakhshan, M., and Hajiaghayi, M. (2018a). Brief Announcement: Semi-MapReduce Meets Congested Clique. *CoRR*, abs/1802.10297.

[17] Behnezhad, S., Derakhshan, M., Hajiaghayi, M., and Karp, R. M. (2018b). Massively parallel symmetry breaking on sparse graphs: MIS and maximal matching. *CoRR*, abs/1807.06701.

[18] Behnezhad, S., Dhulipala, L., Esfandiari, H., Lacki, J., and Mirrokni, V. (2019a). Near-optimal massively parallel graph connectivity. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, to appear.*

[19] Behnezhad, S., Hajiaghayi, M., and Harris, D. G. (2019b). Exponentially faster massively parallel maximal matching. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, to appear.*

[20] Bennett, C. H., Bernstein, E., Brassard, G., and Vazirani, U. (1997). Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, 26(5):1510–1523.

[21] Brandt, S., Fischer, M., and Uitto, J. (2018). Matching and MIS for uniformly sparse graphs in the low-memory MPC model. *CoRR*, abs/1807.05374.

[22] Bremner, D., Chan, T. M., Demaine, E. D., Erickson, J., Hurtado, F., Iacono, J., Langerman, S., Patraşcu, M., and Taslakian, P. (2014). Necklaces, convolutions, and x+y. *Algorithmica*, 69(2):294–314.

[23] Brigham, E. O. and Brigham, E. O. (1988). *The fast Fourier transform and its applications*, volume 448. prentice Hall Englewood Cliffs, NJ.

[24] Brodie, B. C., Taylor, D. E., and Cytron, R. K. (2006). A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH computer architecture news*, 34(2):191–202.

[25] Chan, T. M. and Lewenstein, M. (2015). Clustered integer 3sum via additive combinatorics. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 31–40. ACM.

[26] Chang, Y.-J., Fischer, M., Ghaffari, M., Uitto, J., and Zheng, Y. (2018). The Complexity of $(\Delta + 1)$ Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation. *CoRR*, abs/1808.08419.

[27] Chen, T., Lu, J., and Ling, T. W. (2005). On boosting holism in xml twig pattern matching using structural indexing techniques. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 455–466. ACM.

[28] Clark, C. R. and Schimmel, D. E. (2004). Scalable pattern matching for high speed networks. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 249–257. IEEE.

[29] Clifford, P. and Clifford, R. (2007). Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54.

[30] Cole, R. and Hariharan, R. (2002). Verifying candidate matches in sparse and wildcard matching. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 592–601. ACM.

[31] Czumaj, A., Lacki, J., Madry, A., Mitrovic, S., Onak, K., and Sankowski, P. (2018). Round Compression for Parallel Matching Algorithms. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 471–484.

[32] Danielson, G. C. and Lanczos, C. (1942). Some improvements in practical fourier analysis and their application to x-ray scattering from liquids. *Journal of the Franklin Institute*, 233(4):365–380.

[33] Demetrescu, C., Finocchi, I., and Ribichini, A. (2006). Trading off space for passes in graph streaming problems. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 714–723.

[34] Dufayard, J.-F., Duret, L., Penel, S., Gouy, M., Rechenmann, F., and Perrière, G. (2005). Tree pattern matching in phylogenetic trees: automatic search for orthologs or paralogs in homologous gene sequence databases. *Bioinformatics*, 21(11):2596–2603.

[35] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J. (2004). On graph problems in a semi-streaming model. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 531–543.

[36] Fischer, M. J. and Paterson, M. S. (1974). String-matching and other products. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC.

[37] Fraigniaud, P., Heinrich, M., and Kosowski, A. (2016). Local conflict coloring. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 625–634.

[38] Gabow, H. N., Nishizeki, T., Kariv, O., Leven, D., , and Terada, O. (1985). Algorithms for edgecoloring graphs. Technical report, Tohoku University.

[39] Ghaffari, M., Gouleakis, T., Konrad, C., Mitrovic, S., and Rubinfeld, R. (2018). Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 129–138.

[40] Glazik, C., Schiemann, J., and Srivastav, A. (2017). Finding euler tours in one pass in the w-streaming model with o(n log(n)) RAM. *CoRR*, abs/1710.04091.

[41] Goldberg, A., Plotkin, S., and Shannon, G. (1987). Parallel Symmetry-breaking in Sparse Graphs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 315–324, New York, NY, USA. ACM.

[42] Goldberg, A. V. and Plotkin, S. A. (1987). Parallel $(\Delta+1)$-coloring of Constant-degree Graphs. *Inf. Process. Lett.*, 25(4):241–245.

[43] Goodrich, M. T., Sitchinava, N., and Zhang, Q. (2011). Sorting, Searching, and Simulation in the MapReduce Framework. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, pages 374–383.

[44] Harris, D. G., Schneider, J., and Su, H.-H. (2016). Distributed $(\Delta + 1)$-coloring in sublogarithmic rounds. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 465–478. ACM.

[45] Harvey, N. J. A., Liaw, C., and Liu, P. (2018). Greedy and Local Ratio Algorithms in the MapReduce Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 43–52, New York, NY, USA. ACM.

[46] Hoffmann, C. M. and O'Donnell, M. J. (1982). Pattern matching in trees. *Journal of the ACM (JACM)*, 29(1):68–95.

[47] Holub, J., Iliopoulos, C. S., Melichar, B., and Mouchard, L. (2001). Distributed pattern matching using finite automata. *Journal of Automata, Languages and Combinatorics*, 6(2):191–204.

[48] Hosoya, H. and Pierce, B. (2001). Regular expression pattern matching for xml. In *ACM SIGPLAN Notices*, volume 36, pages 67–80. ACM.

[49] Indyk, P. (1998). Faster algorithms for string matching problems: Matching the convolution bound. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 166–173. IEEE.

[50] Johansson, Ö. (1999). Simple distributed $Delta$+1-coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232.

[51] Jurdzinski, T. and Nowicki, K. (2018). MST in $O(1)$ rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2620–2632.

[52] Kalai, A. (2002). Efficient pattern-matching with don't cares. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 655–656. Society for Industrial and Applied Mathematics.

[53] Karloff, H. J., Suri, S., and Vassilvitskii, S. (2010). A Model of Computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948.

[54] Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260.

[55] Knuth, D. E., Morris, Jr, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350.

[56] Koiliaris, K. and Xu, C. (2017). A faster pseudopolynomial time algorithm for subset sum. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1062–1072. SIAM.

[57] Kuhn, F. and Wattenhofer, R. (2006). On the Complexity of Distributed Graph Coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 7–15, New York, NY, USA. ACM.

[58] Lattanzi, S., Moseley, B., Suri, S., and Vassilvitskii, S. (2011a). Filtering: a method for solving graph problems in mapreduce. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 85–94.

[59] Lattanzi, S., Moseley, B., Suri, S., and Vassilvitskii, S. (2011b). Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM.

[60] Lewenstein, M., Nekrich, Y., and Vitter, J. S. (2014). Space-efficient string indexing for wildcard pattern matching. *arXiv preprint arXiv:1401.0625*.

[61] Linial, N. (1992). Locality in Distributed Graph Algorithms. *SIAM J. Comput.*, 21(1):193–201.

[62] Liu, Y., Guo, L., Li, J., Ren, M., and Li, K. (2012). Parallel algorithms for approximate string matching with k mismatches on cuda. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2414–2422. IEEE.

[63] Luby, M. (1985). A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 1–10, New York, NY, USA. ACM.

[64] Mateus, P. and Omar, Y. (2005). Quantum pattern matching. *arXiv preprint quant-ph/0508237*.

[65] Montanaro, A. (2017). Quantum pattern matching fast on average. *Algorithmica*, 77(1):16–39.

[66] Nakano, K. (2012). Efficient implementations of the approximate string matching on the memory machine models. In *Networking and Computing (ICNC), 2012 Third International Conference on*, pages 233–239. IEEE.

[67] Panconesi, A. and Srinivasan, A. (1992). Improved Distributed Algorithms for Coloring and Network Decomposition Problems. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 581–592, New York, NY, USA. ACM.

[68] Parter, M. (2018). $(\Delta + 1)$ Coloring in the Congested Clique Model. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 160:1–160:14.

[69] Parter, M. and Su, H. (2018). Randomized $(\Delta + 1)$-Coloring in $O(\log^* \Delta)$ Congested Clique Rounds. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 39:1–39:18.

[70] Porat, B. and Porat, E. (2009). Exact and approximate pattern matching in the streaming model. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 315–323. IEEE.

[71] Raju, S. V. and Vinayababu, A. (2006). Optimal parallel algorithm for string matching on mesh network structure. *International Journal of Applied Mathematical Sciences*, 3(2):167–175.

[72] Raju, V. and Vinayababu, A. (2007). Parallel algorithms for string matching problem on single and two-dimensional reconfigurable pipelined bus systems. *Journal of Computer Science*, 3(9):754–759.

[73] Ramesh, H. and Vinay, V. (2003). String matching in o (n+ m) quantum time. *Journal of Discrete Algorithms*, 1(1):103–110.

[74] Ramesh, R. and Ramakrishnan, I. (1992). Nonlinear pattern matching in trees. *Journal of the ACM (JACM)*, 39(2):295–316.

[75] Steyaert, J.-M. and Flajolet, P. (1983). Patterns and pattern-matching in trees: an analysis. *Information and Control*, 58(1-3):19–58.

[76] Vizing, V. G. (1964). On an estimate of the chromatic class of a p-graph. *Discret Analiz*, 3:25–30.