String Matching with Wildcards in the Massively Parallel Computation Model

MohammadTaghi Hajiaghayi
University of Maryland
College Park, Maryland, USA
hajiagha@cs.umd.edu

Hamed Saleh
University of Maryland
College Park, Maryland, USA
hamed@cs.umd.edu

Saeed Seddighin
Toyota Technological Institute
Chicago, Illinois, USA
saeedreza.seddighin@gmail.com

Xiaorui Sun
University of Illinois
Chicago, Illinois, USA
xiaorui@uic.edu

ABSTRACT

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. MPC is a recent framework for parallel computation of big data, which is designed to capture the MapReduce-like algorithms. In this paper, 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.

SPAA '21, July 6–8, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8070-6/21/07…$15.00
https://doi.org/10.1145/3409964.3461793

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**.

KEYWORDS

MPC, String Matching, Wildcard, Pattern Matching

ACM Reference Format:
MohammadTaghi Hajiaghayi, Hamed Saleh, Saeed Seddighin, and Xiaorui Sun. 2021. String Matching with Wildcards in the Massively Parallel Computation Model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3409964.3461793

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 [31] (KMP) deterministically solves the problem in linear time. Since then, attention has been given to many variants and generalizations of pattern matching [1, 2, 5, 8, 10, 11, 13, 15, 16, 20, 22, 25–27, 31, 34–36, 38–41]. Natural generalizations of string matching are when either the text or the pattern is a tree instead of a string [20, 25, 39, 40] or when the pattern has a more sophisticated structure that allows for '?', '+', '*', or in general any regular expression [5, 13, 27, 34]. Also, different computational systems have been considered in the literature: from sequential algorithms [5, 13, 20, 25, 27, 31, 34, 39, 40], to quantum algorithms [8, 35, 36, 38], to distributed settings [2, 16, 26], to the streaming setting [1, 15, 37], to PRAM [10, 11, 22, 41], 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 [3, 7, 23, 30] has been introduced to understand the power and limitations of parallel algorithms. It first proposed by Karloff et al. [30] 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 [19, 30, 33]. Karloff et al. [30] 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 [3, 7, 23, 30], 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.

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 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.

Theorem 3.1 (restated). *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 [21] 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) [12].

Theorem 4.2 (restated). *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 4.3 gives us an efficient MPC algorithm for the wildcard setting. While the reduction from wildcard matching to FFT is a known technique [18] 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 [6], 3-sum [14], subset-sum [32], tree-sparsity [4], tree-separability [6], necklace-alignment [9], 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 $a^x.b^y$ 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 [18]. They showed the subset matching problem could be solved in near-linear time. The definition of the subset matching problem is as follows.

Problem 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 [18], 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.

Theorem 5.5 (restated). *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 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.

(1) 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 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.

Theorem 6.1 (restated). *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.*

(2) 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 [30].

Theorem 6.2 (restated). *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.*

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 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 2.1 has been vastly studied in the literature, and there are a number of solutions that solve the problem in linear time. [31] 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 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:

(1) Character replacement wildcard '?': Any character can match character replacement wildcard.
(2) 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.
(3) String replacement wildcard '*': A string replacement wildcard can be matched with any string of arbitrary length.

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.

Further details regarding a hash function with partially decomposability property are provided in Appendix A of the full version of the paper.

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 3.1. Therefore, the main catch of the algorithm is to deal with the matching when string P spans multiple blocks.

Observation 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 1 for more details. □

Theorem 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

Problem		Theorem	Rounds	Total Runtime	Nodes
$P \in \Sigma^m$		3.1	$O(1)$	$O(n)$	$O(n^x)$
$P \in \{\Sigma \cup \text{'?'}\}^m$		4.3	$O(1)$	$\widetilde{O}(n)$	$O(n^x)$
$P \in \{\Sigma \cup \text{'+'}\}^m$		5.5	$O(1)$	$\widetilde{O}(n)$	$O(n^x)$
$P \in \{\Sigma \cup \text{'*'}\}^m$	$m = O(n^{1-x})$	6.1	$O(\log(n))$	$O(n)$	$O(n^x)$
	no prefix	6.2	$O(\log(n))$	$O(n)$	$O(n^x)$

Table 1: Overview of results

Algorithm 1: StringMatching(a)

Data: two array T and P, where $m = O(\mathcal{S})$
Result: function $f : \{1, 2, \ldots, n - m + 1\} \to \{0, 1\}$ such that $f(i) = 1$ iff $T[i : i + m - 1] = P$.

$f(i) \leftarrow 0 \quad \forall\, 1 \le i \le n - m + 1$;
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.
Run in parallel: **for** $1 \le i \le n/\mathcal{S}$ **do**
  append $T[i\mathcal{S} + 1 : i\mathcal{S} + m]$ to T^i;
  for $1 \le j \le \mathcal{S}$ **do**
    if $h(P) = h(T^i[j : j + m - 1])$ **then**
      $f((i - 1)\mathcal{S} + j) \leftarrow 1$;

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 2.

Proof. Assume there are $(n + m)/\mathcal{S} + 2$ machines as following:

- machines $a_1, a_2, \ldots, a_{n/\mathcal{S}}$ which T^i is stored in a_i.
- machines $b_1, b_2, \ldots, b_{m/\mathcal{S}}$ 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 $1 \le i \le n/\mathcal{S}$. 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 $1 \le i \le n/\mathcal{S}$. 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 2: 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 $T[i : i + m - 1] = P$.

$f(i) \leftarrow 0 \quad \forall\, 1 \le i \le n - m + 1$;
Run in parallel: **for** $1 \le i \le m/\mathcal{S}$ **do**
  send $h(P^i)$ to machine d;
Run in parallel: **for** $1 \le i \le n/\mathcal{S}$ **do**
  send $h(T^i)$ to machine c;
  for $1 \le j \le \mathcal{S}$ **do**
    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;
send $h(P) \leftarrow \text{merge}(h(P^1), h(P^2), \ldots, h(P^{m/\mathcal{S}}))$ to each a_i for $1 \le i \le n/\mathcal{S}$;
for $1 \le i \le n/\mathcal{S}$ **do**
  $h(T^1 + T^2 + \ldots + T^i) \leftarrow \text{merge}(h(T^1 + T^2 + \ldots + T^{i-1}), h(T^i))$;
  send $h(T^1 + T^2 + \ldots + T^i)$ to each machine a_j that needs it in the next round;
Run in parallel: **for** $1 \le i \le n/\mathcal{S}$ **do**
  for $1 \le l \le \mathcal{S}$ **do**
    $s \leftarrow (i - 1)\mathcal{S} + l$;
    let k be the index of the machine where $s + m - 1$ lies in T^k;
    $r \leftarrow s + m - 1 - (k - 1)\mathcal{S}$;
    $\eta = h(T^{i+1} + T^{i+2} + \ldots + T^{k-1})$;
    $h(T[s, s + m - 1]) \leftarrow \text{merge}(h(T^i[l, \mathcal{S}]), \eta, h(T^k[1, r]))$;
    if $h(T[s, s + m - 1]) = h(P)$ **then**
      $f(s) \leftarrow 1$;

4 CHARACTER REPLACE WILDCARD '?'

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

Theorem 4.1 (Proven in [21]). *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 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 [21] 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. [17, 28, 29]

As stated in Theorem 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:

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

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 1 becomes

$$a_k^* = \sum_{j=0}^{n-1} a_j \cdot W_n^{jk} \tag{2}$$

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

Theorem 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 Appendix B of the full version of the paper, we justify Theorem 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 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 $C = T^\dagger \circledast \mathsf{rev}(P^\dagger)$, 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

$$C_{2i+2m-1} = \sum_{j=1}^{2m} T^\dagger_{2(i-1)+j} \cdot P^\dagger_j \tag{3}$$

It is clear that if $T[i, i+m-1]$ matches pattern P, then $C_{2i+2m-1} = 2\mathsf{nz}_P$ 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 [21], the other side of this expression also holds, i.e., $T[i, i+m-1]$ matches pattern P if $C_{2i+2m-1} = 2\mathsf{nz}_P$. Therefore, to find that whether a substring $T[i, i+m-1]$ matches pattern P or not, it suffices to

(1) Compute $C = T^\dagger \circledast \mathsf{rev}(P^\dagger)$ utilizing FFT, which can be implemented in $O(1)$ rounds in MPC model as stated in Theorem 4.2.
(2) For all $1 \le i \le n-m+1$, substring $T[i, i+m-1]$ matches pattern P iff $C_{2i+2m-1} = 2\mathsf{nz}_P$.

□

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 none of its substrings match pattern "oo+kee+". In Subsection 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 5.1. Afterwards, we show a reduction

of greater-than matching from subset matching problem, explained in 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 5.5.

5.1 Relation to greater-than matching

Problem 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., $T_{i+j-1} \ge P_j \quad \forall 1 \le j \le m$.*

To reduce pattern matching with wildcard '+' to greater-than matching, we perform run-length encoding 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 5.1.

Definition 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 $\langle c_i, cnt_i\rangle$ 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 $\langle c_i, cnt_i+\rangle$, where cnt_i is still the number of the occurrences of character c_i in the block.*
- *s° equals the list of these pairs $\langle c_i, cnt_i\rangle$ or $\langle c_i, cnt_i+\rangle$, concatenated in a way that preserves the original ordering of the string.*

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

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

$$P^\circ = \langle\langle \mathrm{o}, 2+\rangle, \langle \mathrm{k}, 1+\rangle, \langle \mathrm{e}, 2+\rangle, \langle \mathrm{p}, 1\rangle\rangle$$

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

- T° = "b[1]o[2]k[2]e[2]p[1]e[1]r[1]".
- P° = "o[2+]k[1+]e[2+]p[1]".

Observation 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. □

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:

(1) $T^\circ_{i+j-1} = \langle c_j, x\rangle$ for some $x \ge \mathrm{cnt}_j$ if $P^\circ_j = \langle c_j, \mathrm{cnt}_j+\rangle$.
(2) $T^\circ_{i+j-1} = \langle c_j, \mathrm{cnt}_j\rangle$ if $P^\circ_j = \langle c_j, \mathrm{cnt}_j\rangle$ and $2 \le j \le m-1$.
(3) $T^\circ_{i+j-1} = \langle c_j, x\rangle$ for some $x \ge \mathrm{cnt}_j$ if $P^\circ_j = \langle c_j, \mathrm{cnt}_j\rangle$ and $j \in \{1, m\}$.

Constraint 3 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 2, 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 2 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 1 and 3, and we can replace numbers we already checked for constraint 2 by some 0's.

Observation 5.2. *We can reduce pattern matching with wildcard '+' from greater-than matching in $O(1)$ MPC rounds.*

Using Observation 5.1, it only remains to perform $O(1)$ wildcard '?' matchings to obtain an instance of greater-than matching that already satisfies constraint 2, as well as letter constraints. Thus, Observation 5.2 is implied.

5.2 Reduction from subset matching

Problem 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 [18].

Observation 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 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 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 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 $T^{i,1}, T^{i,2}, \ldots, T^{i,\Sigma}$ 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 [24].

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 [18], 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 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 [18] showed it is analogous to sparse wildcard matching. □

Theorem 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 2 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:

$$T'_i = \bigcup_{j=1}^{cnt_i} \{\langle c_i, j+\rangle\} \cup \{\langle c_i, cnt_i\rangle\} \qquad \text{if } T_i^\circ = \langle c_i, cnt_i\rangle \quad (4)$$

$$P'_i = \{\langle c_i, cnt_i\rangle\} \qquad \text{if } P_i^\circ = \langle c_i, cnt_i\rangle \quad (5)$$

$$P'_i = \{\langle c_i, cnt_i+\rangle\} \qquad \text{if } P_i^\circ = \langle c_i, cnt_i+\rangle \quad (6)$$

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. □

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:

(1) A sequential algorithm for string matching with wildcard '*' in time $O(n + m)$.
(2) 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 $O(n^{1-x})$.
(3) 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.

6.1 Linear time sequential algorithm

Observation 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 3: StringMatchingWithStar(a)

Data: $s \in \Sigma^n$ and $p \in \{\Sigma \cup \text{'*'}\}^m$.
Result: Yes or No.

```
Set i ← 1;
for j = 1, 2, ..., w and i ≤ n do
    Run KMP for the string s[i, n] and pattern P_j;
    If KMP fails, then Return No;
    Let i' be the position satisfying s[i', i' + |P_j| − 1] = P_j
      obtained by KMP;
    Set i ← i' + |P_j|;
Return Yes.
```

6.2 MPC algorithm for small subpattern

Theorem 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:

(1) $g(k)$ for position k of p: the largest position which is smaller than or equal to k such that $p[k]$ is '*'.
(2) $h(k)$ 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 $p[\beta] \neq$ '*':

$$f(i,j,k) = \max \begin{cases} f(i'+1, j, g(\beta)), \\ \max_{0 \le \ell \le \lfloor \frac{\beta - h(\beta)}{g(\beta)} \rfloor} \{f(i'+1, j, \beta - \ell \cdot h(\beta))\} \end{cases} \tag{7}$$

Our algorithm is StringMatchingWithStar(b).

OF 6.1. We show that Eq. 7 correctly computes $f(i,j,k)$ for an arbitrary $i \le i' < j$. Then the theorem follows from the description of StringMatchingWithStar(b), Eq. 7 and Observation 6.1.

Let α be the largest position of p such that there is an exact matching of the concatenation of $s_i, \ldots, s_j$ 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. 7, then $f(i,j,k) \le \alpha$, since Eq. 7 implies a feasible exact matching.

Now we show that $f(i,j,k) \ge \alpha$ if $f(i,j,k)$ is computed by Eq. 7. There is an exact matching of the concatenation of $s_i, \ldots, s_j$ 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 $p[\gamma] \neq$'*' 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 $p[\gamma] \neq$'*' and $p[\beta] \neq$'*'. 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 4: StringMatchingWithStar(b)

Data: two array s and p.
Result: Yes or No.

1 Distribute $s_1, s_2, \ldots, s_t$ 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** $m = 1, 2, \ldots, \lceil \log_2 t \rceil$ **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 7 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.

6.3 MPC algorithm for non-prefix subpatterns

THEOREM 6.2. *Given strings $s \in \Sigma^n$ and $p \in \{\Sigma \cup$ '*'$\}^m$ 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 $a_1, a_2, \ldots, a_u$ with the following two conditions

(1) P_i is a prefix of $s[a_i, n]$ for every $i \in [w]$.
(2) $a_i + |P_i| \le a_{i+1}$ 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 3, 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 $j \ge i + |P_{f(i)}|$ and $f(j) = f(i)+1$ for all the pairs $(i, f(i))$. The graph connectivity needs $O(\log n)$ rounds. □

Algorithm 5: 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 $j \ge i + |P_{f(i)}|$ 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 ;

ACKNOWLEDGMENTS

MohammadTaghi Hajiaghayi and Hamed Saleh are supported in part by NSF BIGDATA grant IIS-1546108, and NSF SPX grant CCF-1822738. Saeed Seddighin is partially funded by an unrestricted gift from Google and a Research Award from Adobe.

REFERENCES

[1] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 147–160.

[2] Reaz Ahmed and Raouf Boutaba. 2007. Distributed pattern matching: a key to flexible and efficient P2P search. *IEEE Journal on selected areas in communications* 25, 1 (2007).

[3] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. 2014. Parallel algorithms for geometric graph problems. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*. 574–583.

[4] Arturs Backurs, Piotr Indyk, and Ludwig Schmidt. 2017. Better approximations for tree sparsity in nearly-linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2215–2229.

[5] Joshua Baron, Karim El Defrawy, Kirill Minkovich, Rafail Ostrovsky, and Eric Tressler. 2012. 5pm: Secure pattern matching. In *International Conference on Security and Cryptography for Networks*. Springer, 222–240.

[6] MohammadHossein Bateni, MohammadTaghi Hajiaghayi, Saeed Seddighin, and Cliff Stein. 2018. Fast algorithms for knapsack via convolution and prediction. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 1269–1282.

[7] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*. 273–284.

[8] Charles H Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. 1997. Strengths and weaknesses of quantum computing. *SIAM journal on Computing* 26, 5 (1997), 1510–1523.

[9] David Bremner, Timothy M Chan, Erik D Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, Mihai Patraşcu, and Perouz Taslakian. 2014. Necklaces, Convolutions, and X+Y. *Algorithmica* 69, 2 (2014), 294–314.

[10] Dany Breslauer and Zvi Galil. 1990. An optimal O(\log\logn) time parallel string matching algorithm. *SIAM J. Comput.* 19, 6 (1990), 1051–1058.

[11] Dany Breslauer and Zvi Galil. 1991. A lower bound for parallel string matching. In *Proceedings of the twenty-third annual ACM symposium on Theory of Computing*. 439–443.

[12] E Oran Brigham and E Oran Brigham. 1988. *The fast Fourier transform and its applications*. Vol. 448. prentice Hall Englewood Cliffs, NJ.

[13] Benjamin C Brodie, David E Taylor, and Ron K Cytron. 2006. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH computer architecture news* 34, 2 (2006), 191–202.

[14] Timothy M Chan and Moshe Lewenstein. 2015. Clustered integer 3SUM via additive combinatorics. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. ACM, 31–40.

[15] Ting Chen, Jiaheng Lu, and Tok Wang Ling. 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*. ACM, 455–466.

[16] Christopher R Clark and David E Schimmel. 2004. Scalable pattern matching for high speed networks. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*. IEEE, 249–257.

[17] Peter Clifford and Raphaël Clifford. 2007. Simple deterministic wildcard matching. *Inform. Process. Lett.* 101, 2 (2007), 53–54.

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

[19] Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. 2018. Round Compression for Parallel Matching Algorithms. In *STOC*.

[20] Jean-François Dufayard, Laurent Duret, Simon Penel, Manolo Gouy, François Rechenmann, and Guy Perrière. 2005. Tree pattern matching in phylogenetic trees: automatic search for orthologs or paralogs in homologous gene sequence databases. *Bioinformatics* 21, 11 (2005), 2596–2603.

[21] Michael J Fischer and Michael S Paterson. 1974. *String-matching and other products*. Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC.

[22] Zvi Galil. 1985. Optimal parallel algorithms for string matching. *Information and Control* 67, 1-3 (1985), 144–157.

[23] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. 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*. 374–383.

[24] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*. Springer, 374–383.

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

[26] Jan Holub, Costas S Iliopoulos, Bořivoj Melichar, and Laurent Mouchard. 2001. Distributed pattern matching using finite automata. *Journal of Automata, Languages and Combinatorics* 6, 2 (2001), 191–204.

[27] Haruo Hosoya and Benjamin Pierce. 2001. Regular expression pattern matching for XML. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 67–80.

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

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

[30] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A model of computation for MapReduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 938–948.

[31] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. 1977. Fast pattern matching in strings. *SIAM journal on computing* 6, 2 (1977), 323–350.

[32] Konstantinos Koiliaris and Chao Xu. 2017. A faster pseudopolynomial time algorithm for subset sum. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1062–1072.

[33] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. 2011. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 85–94.

[34] Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. 2014. Space-efficient string indexing for wildcard pattern matching. *arXiv preprint arXiv:1401.0625* (2014).

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

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

[37] Benny Porat and Ely Porat. 2009. Exact and approximate pattern matching in the streaming model. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*. IEEE, 315–323.

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

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

[40] Jean-Marc Steyaert and Philippe Flajolet. 1983. Patterns and pattern-matching in trees: an analysis. *Information and Control* 58, 1-3 (1983), 19–58.

[41] Uzi Vishkin. 1985. Optimal parallel pattern matching in strings. In *International Colloquium on Automata, Languages, and Programming*. Springer, 497–508.