applied sciences

Article

Order-Preserving Multiple Pattern Matching in Parallel

Somin Park [1], Jinhyeok Park [2], Youngho Kim [1] and Jeong Seop Sim [1,*]

[1] Department of Computer Engineering, Inha University, Incheon 22212, Republic of Korea; smpark95@inha.edu (S.P.); yhkim85@inha.ac.kr (Y.K.)
[2] System Team, Mobile Experience, Samsung Electronics Co., Ltd., Suwon 16677, Republic of Korea; jh1015.park@samsung.com
* Correspondence: jssim@inha.ac.kr; Tel.: +82-32-860-7455

Abstract: The order-preserving multiple pattern matching problem is to find all substrings of T whose relative orders are the same for any pattern in a set of patterns. Various sequential algorithms have been studied for the order-preserving multiple pattern matching problems. In this paper, we propose two parallel algorithms, each of which uses Aho–Corasick automata and fingerprint tables, respectively. We also present experimental results of comparing the execution times of each parallel algorithm on various types of time-series data.

Keywords: string matching; order-preserving multiple pattern matching; parallel algorithms; time-series data analysis

Citation: Park, S.; Park, J.; Kim, Y.; Sim, J.S. Order-Preserving Multiple Pattern Matching in Parallel. *Appl. Sci.* **2023**, *13*, 5142. https://doi.org/10.3390/app13085142

Academic Editor: Richard (Chunhui) Yang

Received: 9 March 2023
Revised: 30 March 2023
Accepted: 18 April 2023
Published: 20 April 2023

1. Introduction

Two strings x, y $(|x| = |y|)$ from an integer alphabet are order-isomorphic if the relative orders of characters are the same. For example, given two strings $x = (17, 30, 10)$ and $y = (4, 9, 1)$, they are order-isomorphic because their relative orders of characters are the same as $(2, 3, 1)$. Given text $T(|T| = n)$ and pattern $P(|P| = m)$, finding all substrings of T, which are order-isomorphic to P, is called the order-preserving pattern matching (OPPM for short) problem [1–5]. Given text $T(|T| = n)$ and a set of patterns $\mathbb{P} = \{P_1, \ldots, P_k\}$, the order-preserving multiple pattern matching (OPMPM for short) problem is to find all substrings of T that are order-isomorphic to any $P_a (1 \leq a \leq k)$ in $\mathbb{P}$ [1–5]. The order-preserving pattern matching problem and order-preserving multiple pattern matching problem can be used to analyze various time-series data such as stock price indices, music melodies, and biomedical data [1–5].

Most existing OPMPM algorithms are performed in two phases: a preprocessing phase and a search phase. Let m, $\overline{m}$, and M denote the shortest pattern length, the longest pattern length, and the sum of the lengths of all patterns in $\mathbb{P}$, respectively. Kim et al. [1] proposed an algorithm that performs the preprocessing phase in $O(M \log \overline{m})$ time and the search phase in $O(n \log \overline{m})$ time using Aho–Corasick automata [6]. Han et al. [7] proposed two algorithms for the order-preserving multiple pattern matching problem. In the first algorithm, the preprocessing and search phases run in $O(kmq + M \log \overline{m})$ time and $O((n/m) \log M)$ time, on average, and in the second algorithm, the preprocessing and search phases run in $O(M \log \overline{m})$ time and $O(n)$ time, on average. Park et al. [8] proposed an algorithm that improved the space complexity of the first algorithm proposed in [7] from $O(q! + k + M)$ to $O(k + M)$ using fingerprint tables. Park et al. [9] parallelized the second algorithm proposed in [7]. The preprocessing phase runs in $O(\overline{m})$ time using $O(M)$ threads and the search phase runs in $O(m)$ time, on average, using $O(n)$ threads in [9].

In this paper, we present two parallel algorithms for the OPMPM problem. First, we parallelize the algorithm proposed in [1]. The preprocessing phase of the first algorithm runs in $O(\overline{m})$ time using $O(M)$ threads and the search phase runs in $O(\overline{m} \log \overline{m})$ time using $O(n/\overline{m})$ threads. Second, we parallelize the algorithm proposed in [8]. The preprocessing phase of the second algorithm runs in $O(\overline{m})$ time using $O(M)$ threads and the search phase runs in $O(M)$ time using $O(n)$ threads. Then, we present experimental results that compare the execution times of each parallel algorithm for various types of time-series data. The experimental results showed that for randomly generated strings, when $n = 100{,}000$, $m = 9$, and $k = 100$, the execution time of the first parallel algorithm was approximately 2.12 times faster than the sequential algorithm, and the second parallel algorithm was approximately 1.6 times faster than the sequential algorithm. For all time-series data used in our experiments, the execution time of our second parallel algorithm was comparable to that of a fast parallel algorithm proposed in [9].

This paper is organized as follows. In Section 2, we introduce the definitions of terms and related studies. In Section 3, we explain our parallel algorithms for the OPMPM problem. In Section 4, the execution times of parallel algorithms are compared through experiments. In Section 5, we conclude.

2. Related Works

Given string x, the length of x is denoted by $|x|$ and the i-th character of x $(1 \leq i \leq |x|)$ is denoted by $x[i]$. The substring from $x[i]$ to $x[j]$ is denoted by $x[i..j]$ $(1 \leq i \leq j \leq |x|)$. $x[1..i]$ is called a prefix of x and $x[i..|x|]$ is called a suffix of x for $1 \leq i \leq |x|$. For convenience, we assume that all the characters in the string are different. Given two strings x, y $(|x| = |y|)$, if $x[i] < x[j] \Leftrightarrow y[i] < y[j] (1 \leq i, j \leq |x|)$ is satisfied, and x and y are order-isomorphic and denoted by $x \approx y$ [1,4,10].

Let x' be the string constructed by sorting all the characters of x in ascending order. Then, the position table POS_x of x is defined as follows:

$$POS_x[i] = j(x'[i] = x[j], 1 \leq i \leq |x|).$$

That is, $POS_x[i]$ is the index of the i-th smallest character in x. Given two strings x, y $(|x| = |y|)$, for all $i(1 \leq i < n)$, if $y[POS_x[i]] \leq y[POS_x[i+1]]$ is satisfied, $x \approx y$ [8]. A prefix table μ_x of string x is defined as follows:

$$\mu_x[i] = |\{j : x[j] < x[i], 1 \leq j < i\}| + 1.$$

That is, $\mu_x[i]$ represents the number of characters that are smaller than $x[i]$ in $x[1..i]$. If the prefix tables of two strings x and y are the same, that is, $\mu_x = \mu_y$, $x \approx y$ [1]. Note that when there exist the same characters in the given string, order-isomorphism can still be determined using the extended prefix representation proposed in [2]. Table 1 presents the prefix table μ_x and position table POS_x when $x = (23, 29, 20, 57, 59)$.

Table 1. Prefix table and position table of $x = (23, 29, 20, 57, 59)$.

x	23	29	20	57	59
$\mu_x[i]$	1	2	1	4	5
$POS_x[i]$	3	1	2	4	5

In [7], an OPMPM algorithm was proposed using fingerprints of q-grams. The fingerprint converts a q-gram (a string of length q) into an integer within the range of $[1, q!]$ using the factorial number system [11,12]. The fingerprints are used to find candidate substrings of text T that may be order-isomorphic to pattern P. Given a q-gram x, fingerprint $f(x)$ is defined as follows [3,7]:

$$f(x) = \textstyle\sum_{k=1}^{q} [(\mu_x[k] - 1) \times (k-1)!] + 1.$$

For example, if q-gram $x = (8, 10, 6)$, $f(x) = 2$.

In [1], the Aho–Corasick automaton [6] is used to solve the order-preserving multiple pattern matching problem. In the preprocessing phase, the Aho–Corasick automaton is created in $O(M \log \overline{m})$ time using a set of prefix tables of patterns. In each Step $i(1 \leq i \leq n)$ of the search phase, the state for $T[i]$ is computed and $P_a(1 \leq a \leq k)$ is searched in the automaton using the transition function, the failure function, and the output function of the automaton. Figure 1 shows the Aho–Corasick automaton when $P_1 = (17, 25, 15, 30)$, $P_2 = (30, 44, 25, 40)$, and $P_3 = (40, 50, 61)$.

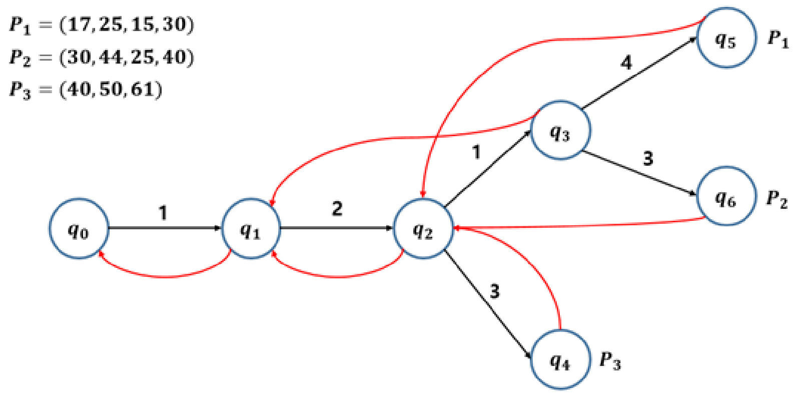

Figure 1. Example of the Aho–Corasick automaton for OPMPM.

Let $\tilde{P}_a(1 \le a \le k)$ be the prefix of length m for each pattern P_a and let $\tilde{\mathbb{P}} = \{\tilde{P}_1, \tilde{P}_2, \ldots, \tilde{P}_k\}$. Park et al. [8] proposed an OPMPM algorithm that generates the position table POS_{P_a} for each $P_a(1 \le a \le k)$ and the fingerprint table FP_a that stores the rightmost q-gram's fingerprint of $\tilde{P}_a(1 \le a \le k)$ in the preprocessing phase, to find candidate patterns that may be order-isomorphic to substrings of T. In the search phase, all substrings of T, which are order-isomorphic to any $P_a(1 \le a \le k)$, are searched using FP_a and POS_{P_a}.

3. Parallel Algorithms for the OPMPM Problem

In this section, we propose two parallel algorithms for the OPMPM problem. The first algorithm uses the Aho–Corasick automaton, while the second algorithm utilizes the position tables and fingerprint tables.

3.1. Parallel OPMPM Algorithm Using the Aho–Corasick Automaton

Our parallel algorithm for the OPMPM problem using the Aho–Corasick automaton consists of the following steps: In the preprocessing phase, we create prefix tables using $O(M)$ threads in $O(\overline{m})$ time. That is, the prefix table of each pattern is calculated in parallel by assigning $|P_a|$ threads for each pattern $P_a(1 \le a \le k)$. In the prefix table μ_{P_a} of pattern P_a, each thread $t(1 \le t \le |P_a|)$ linearly searches $P_a[1..t-1]$ to calculate $\mu_{P_a}[t]$. See Algorithm 1. Thus, a set of prefix tables for all patterns can be calculated in $O(\overline{m})$ time using $O(M)$ threads. To create an automaton using μ_{P_a}, the existing algorithm [1] method is employed.

Algorithm 1 Preprocessing phase (parallel calculation of prefix tables for a pattern set).

```
Input: A set of strings $\{P_1, P_2, \ldots, P_k\}$
Output: A set of prefix tables $\{\mu_{P_1}, \mu_{P_2}, \ldots, \mu_{P_k}\}$
1 parallel for $a \leftarrow 1$ to $k$ do
2     parallel for $t \leftarrow 1$ to $|P_a|$ do
3         $\mu_{P_a}[t] \leftarrow 1$
4         for $i \leftarrow 1$ to $t-1$ do
5             if $P_a[i] < P_a[t]$ then
6                 $\mu_{P_a}[t] \leftarrow \mu_{P_a[t]+1}$
```

In the search phase, T is divided into b blocks and each block is searched in parallel using b threads (Figure 2). Thread $t(1 \le t \le b)$ searches each block of T for the location of the substring that is order-isomorphic to P_a using the automaton created in the preprocessing phase. See Algorithm 2. Since the substring of T that is order-isomorphic to P_a can occur across two adjacent blocks, all threads except the last one set the block size to

$\lceil n/b \rceil + \overline{m} - 1$ (Figure 2). Since the insertion, deletion, and rank calculation operations in the order-statistics tree require $O(\log \overline{m})$ time during the search phase, they are conducted in $O((n/b + \overline{m}) \log \overline{m})$ time. If we set $b = \lceil n/\overline{m} \rceil$, the search phase can be conducted in $O(\overline{m} \log \overline{m})$ time. Thus, this parallel algorithm can be solved as an OPMPM problem in $O(M + \overline{m} \log \overline{m})$ time using $O(max(M, \lceil n/\overline{m} \rceil))$ threads.

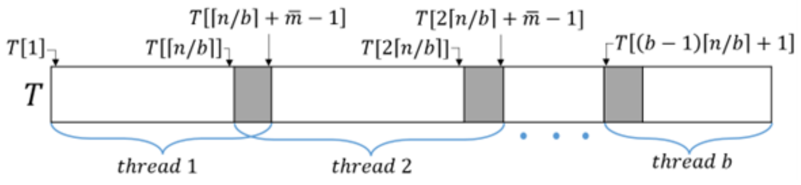
$T[1]$
$T[\lceil n/b \rceil]$
$T[\lceil n/b \rceil + \overline{m} - 1]$
$T[2\lceil n/b \rceil]$
$T[2\lceil n/b \rceil + \overline{m} - 1]$
$T[(b-1)\lceil n/b \rceil + 1]$
T
thread 1
thread 2
thread b

Figure 2. Search phase of the parallel OPMPM algorithm using the Aho–Corasick automaton.

Algorithm 2 Search phase.

Input: Aho–Corasick automaton, string T
Output: Positions i of substrings of T which is order-isomorphic to P_a

```
1  $q_s \leftarrow q_0$, **OST** $\tau$, **int** $r$
2  **parallel for** $t \leftarrow 1$ **to** $b$ **do**
3      **for** $i \leftarrow 1$ **to** $\lceil n/b \rceil + \overline{m} - 1$ **do**
4          **if** $(t-1)\lceil n/b \rceil + i > n$ **then**
5              **break**
6          $\tau$.insert($T[(t-1)\lceil n/b \rceil + i)]$)
7          $r \leftarrow \tau$.rank($T[(t-1)\lceil n/b \rceil + i]$)
8          **while** $g(q_s, r) = fail$
9              $\tau$.delete($T[i - d(q_s)..i - d(\pi(q_s)) - 1]$)
10             $q_s \leftarrow \pi(q_s)$
11             $r \leftarrow \tau$.rank($T[(t-1)\lceil n/b \rceil + i]$)
12         $q_s \leftarrow g(q_s, r)$
13         **if** $P_a \in out(q_s)$ **and** $i - |P_a| + 1 \leq t\lceil n/b \rceil$ **then**
14             **print** $(i - |P_a| + 1, a)$
```

3.2. Parallel OPMPM Algorithm Using the Fingerprint Table

The algorithm that solves the OPMPM problem using a fingerprint table in parallel is as follows: In the preprocessing phase, position table POS_{P_a} and fingerprint table FP_a are created in parallel for each pattern of $P_a (1 \leq a \leq k)$. The POS_{P_a} calculation is performed in parallel using $|P_a|$ threads. Each thread $t(1 \leq t \leq |P_a|)$ linearly searches P_a to calculate the order r of $P_a[t]$. By the definition of POS_{P_a}, it satisfies $POS_{P_a}[r] = t$, and the position tables for all patterns can be calculated in parallel in $O(\overline{m})$ time using $O(M)$ threads. FP_a is created in parallel using q threads (Algorithm 3). Each thread $t(1 \leq t \leq q)$ calculates the fingerprint of $\tilde{P}_a[m - q + 1..m]$ in parallel. Here, atomic operations are used to prevent multiple threads from accessing FP_a concurrently (Line 8 in Algorithm 3). Thus, the fingerprint table can be calculated in $O(q)$ time using $O(kq)$ threads, and the preprocessing phase is conducted in $O(\overline{m})$ time using $O(M)$ threads.

In the search phase, all substrings of T are checked in parallel using $n - m + 1$ threads (Algorithm 4). Each thread $i(1 \leq i \leq n - m + 1)$ first calculates $f(T[i + m - q..i + m - 1])$. $f(T[i + m - q..i + m - 1])$ and each of $FP_a(1 \leq a \leq k)$ are sequentially compared, and if they are the same, it is verified whether P_a and $T[i..i + |P_a| - 1]$ are order-isomorphic using POS_{P_a} (from Lines 2 to 5 in Algorithm 4) [13]. If $P_a \approx T[i..i + |P_a| - 1]$, then (i, a) is printed. In the worst case, it verifies whether all substrings of T are order-isomorphic to all patterns. Thus, the search phase can be performed in $O(M)$ time using $O(n)$ threads. This parallel algorithm can be solved as an OPMPM problem in $O(\overline{m} + M)$ time using $O(max(M, n))$ threads.

Algorithm 3 Parallel calculation of the fingerprint table. FP

```
Input: A set of strings $\{P_1, P_2, \ldots, P_k\}$,, int $m$, int $q$
Output: $FP_1, FP_2, \ldots, FP_k$
1 parallel for $a \leftarrow 1$ to $k$ do
2     parallel for $t \leftarrow 1$ to $q$ do
3         $FP_a \leftarrow 1$
4         $c \leftarrow 0$
5         for $i \leftarrow 1$ to $t - 1$ do
6             if $P_a[m - q + i] < P_a[m - q + t]$ then
7                 $c \leftarrow c + 1$
8         atomicAdd$(FP_a \leftarrow FP_a + c \times (t - 1)!)$
```

Algorithm 4 The search phase of the algorithm using a fingerprint table.

```
Input: A string $T, FP_1, FP_2, \ldots, FP_k$, int $m$
Output: Positions $i$ of substrings of $T$ which is order-isomorphic to $P_a$
1 parallel for $i \leftarrow 1$ to $n - m + 1$ do
2     for $a \leftarrow 1$ to $k$ do
3         if $f(T[i + m - q..i + m - 1]) = FP_a$ then
4             if $P_a \approx T[i..i + |P_a| - 1]$ then
5                 print $(i, a)$
```

4. Experimental Results

The experiment was conducted on the following environment: Windows 10 (64-bit) operating system, AMD Ryzen9 3950X CPU, 64 GB RAM, NVIDIA GeForce RTX 3080 Ti GPU, C++ and CUDA programming language, and Visual Studio 2019 (CUDA SDK 11.0).

The algorithms experimented with in the present paper are denoted as follows: The OPMPM algorithm using the Aho–Corasick automaton proposed in [1] is denoted by AC, and the OPMPM algorithm using the fingerprint table proposed in [8] is denoted by FT. The parallel OPMPM algorithm proposed in [9] is denoted by pKR, the parallel OPMPM algorithm using the Aho–Corasick automaton proposed in this paper is denoted by pAC, and the parallel OPMPM algorithm using the fingerprint table is denoted by pFT, respectively.

The data used in the experiment are randomly generated strings and two types of time-series data: the Dow Jones Index and electrocardiogram data.

- Randomly generated strings: Texts and patterns consisting of $\sum = \{1, 2, \ldots, 2^{30}\}$ were randomly generated. Text T was generated by increasing length n from 10,000 by 10,000 to 100,000. A set of patterns P was generated by increasing the number of patterns k from 100 by 100 to 1000 and increasing the pattern length m from 5 by 1 to 15.
- Dow Jones Index: Texts and patterns were extracted at random for each experiment from the daily closing price of the Dow Jones Industrial Average between 2 May 1885 and 12 April 2019 [14]. The text length n was increased from 1000 by 1000 to 10,000. The number of patterns k and their length m were set to the same values used for the randomly generated strings.
- Electrocardiogram data: The electrocardiogram (ECG) data used in the experiment were obtained from the MIT-BIH ECG biosignal database provided by Physionet [15]. An electrocardiogram records the electrical impulses from the heart. Texts were randomly extracted from the total records, while patterns were extracted evenly from abnormal symptom data and normal data. It should be noted that the texts and patterns were extracted from electrocardiogram data of different individuals. The text length n and the number of patterns k were set equal to those of the randomly generated strings, and the pattern length m increased from 10 by 1 to 15 during the generation process.

The parameter setup of the algorithm and measurement of execution time were conducted as follows: In FT and pFT, q was set to 5. In each of the parallel algorithms, pAC employed 1000 threads, and pKR and pFT employed n threads. The execution time of each algorithm was the mean of 100 execution times, which was measured in milliseconds (ms) and rounded to three decimal places. The execution time of parallel algorithms includes the execution time of the cudaMemcpy() function that copies data between host memory and device (GPU) memory.

Experiment (1). Comparison of execution times of AC and pAC, and FT and pFT for randomly generated strings: Figure 3 shows the execution times of AC and pAC for randomly generated strings according to n when $m = 9$ and $k = 100$. When n = 10,000, pAC was slower than AC due to the additional execution time required by the cudaMemcpy() function in pAC. Most of the execution time for pAC was spent on the cudaMemcpy() function. However, for $n \geq$ 20,000, pAC performed faster than AC. When $n =$ 100,000, $m = 9$, and $k = 100$, the execution time of AC was around 41.22 ms and that of pAC was around 19.49 ms, indicating that pAC was approximately 2.12 times faster than AC. The execution time of pAC increased as n increased due to the limitation of the number of available GPU cores. Figure 4 shows the execution times of AC and pAC for randomly generated strings according to k when $n =$ 100,000 and $m = 9$. In this case, pAC was faster than AC in all cases.

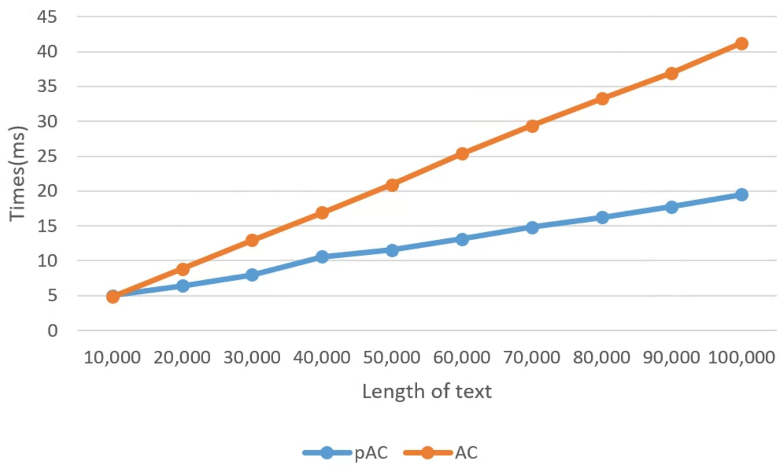

Figure 3. Comparison of execution times of AC and pAC varying n for randomly generated strings when $m = 9$ and $k = 100$.

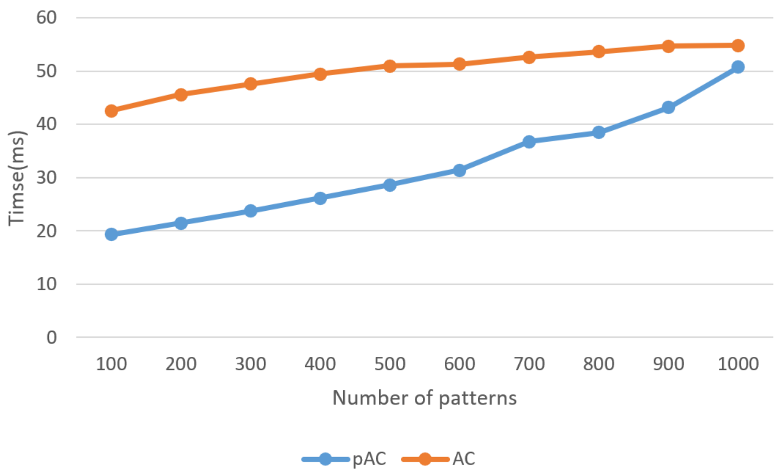

Figure 4. Comparison of execution times of AC and pAC varying k for randomly generated strings when $n =$ 100,000 and $m = 9$.

Figure 5 shows the execution times of FT and pFT for randomly generated strings according to n when $m = 9$ and $k = 100$. pFT was faster than FT in all cases. For instance, when $n = 100{,}000$, $m = 9$, and $k = 100$, the execution time of FT was around 8.6 ms, while that of pFT was around 5.39 ms, indicating that pFT was approximately 1.6 times faster than FT. However, as with pAC, the execution time of pFT also increased as n increased due to the limitation of the number of available GPU cores. Figure 6 shows the execution times of FT and pFT for randomly generated strings according to k when $n = 100{,}000$ and $m = 9$. In this case, as in the previous case, pFT was faster than FT in all cases.

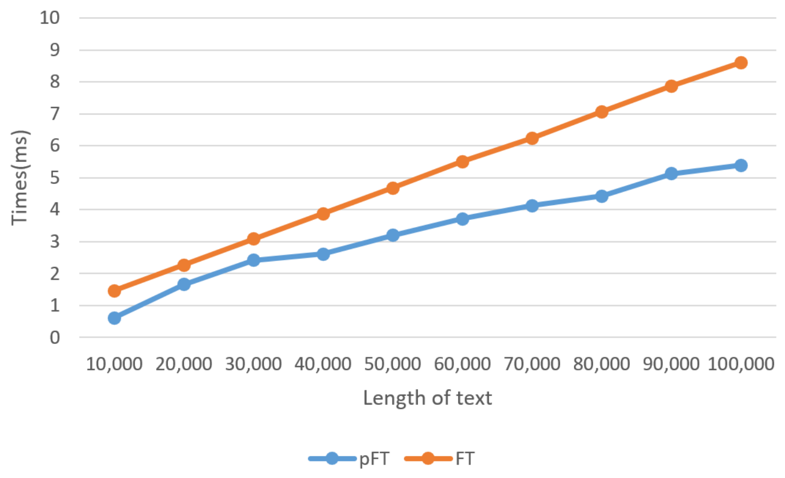

Figure 5. Comparison of execution times of FT and pFT according to n for randomly generated strings when $m = 9$ and $k = 100$.

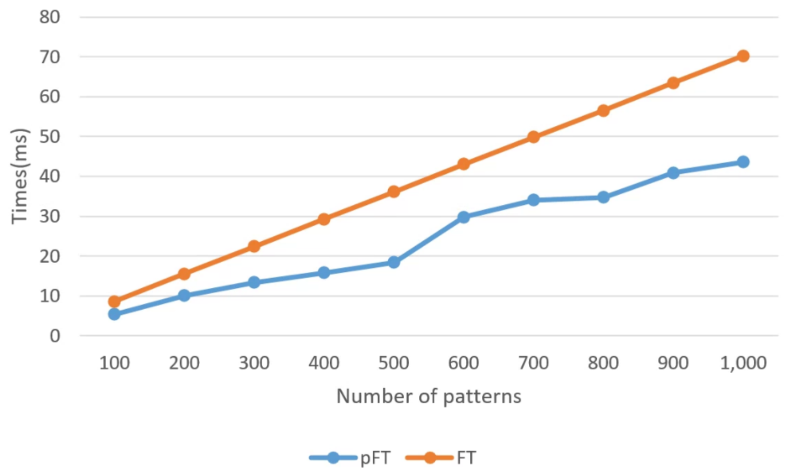

Figure 6. Comparison of execution times of FT and pFT according to k for randomly generated strings when $n = 100{,}000$ and $m = 9$.

Experiment (2). Comparison of execution times of pKR, pAC, and pFT for randomly generated strings: Table 2 presents the execution times of pKR, pAC, and pFT for randomly generated strings with varying parameter settings. In all cases, pKR and pFT performed faster than pAC, and pKR and pFT exhibited comparable execution times. When $n = 100{,}000$, $m = 9$, and $k = 1000$, the execution times of pKR and pFT were approximately 1.98 and 2.03 times faster, respectively, than those of pAC.

Table 2a presents the execution times of pKR, pAC, and pFT according to m when $n = 100{,}000$ and $k = 1000$. As the length of patterns m increased, the creation time of the Aho–Corasick automaton increased, leading to an increase in the total execution time of pAC. The length of patterns m did not significantly affect the execution times of pFT and

pKR. Table 2b presents the execution times of pKR, pAC, and pFT according to k when $n = 100{,}000$ and $m = 9$. The execution times of all algorithms increased as k increased.

Table 2. Comparison of execution times of pAC, pKR, and pFT for random strings.

(a) Comparison of execution times varying m when $n = 100{,}000$, $k = 1000$			
	Execution times of the algorithms (unit: ms)		
m	pAC	pKR	pFT
6	41.31	21.58	22.62
9	44.83	22.64	21.98
12	49.07	22.18	22.51
15	52.49	22.16	22.52
(b) Comparison of execution times varying k when $n = 100{,}000$, $m = 9$			
	Execution times of the algorithms (unit: ms)		
k	pAC	pKR	pFT
100	19.35	2.22	2.24
500	31.07	11.25	11.47
1000	44.83	22.64	21.98

Experiment (3). Comparison of execution times of pKR, pAC, and pFT for Dow Jones Index data: Table 3 presents the execution times of pKR, pAC, and pFT according to the parameters for Dow Jones Index data. In all conditions, pKR and pFT performed faster than pAC, and pKR and pFT exhibited comparable execution times. When $n = 10{,}000$, $m = 9$, and $k = 1000$, the execution times of pKR and pFT were around 5.61 times and 5.71 times faster than that of pAC. Overall, it showed a similar trend to that of Experiment 2.

Table 3. Comparison of execution times of pAC, pKR, and pFT for Dow Jones Index data.

(a) Comparison of execution times varying m when $n = 10{,}000$, $k = 1000$			
	Execution times of the algorithms (unit: ms)		
m	pAC	pKR	pFT
6	9.58	2.29	2.55
9	12.84	2.29	2.25
12	16.35	2.22	2.25
15	20.33	2.34	2.45
(b) Comparison of execution times varying k when $n = 10{,}000$, $m = 9$			
	Execution times of the algorithms (unit: ms)		
k	pAC	pKR	pFT
100	5.03	0.49	0.47
500	8.52	1.35	1.38
1000	12.84	2.29	2.25

Experiment (4). Comparison of execution times of pKR, pAC, and pFT for ECG data: Table 4 presents the execution times of pKR, pAC, and pFT according to the parameters for electrocardiogram data. In all cases, pKR and pFT performed faster than pAC, and pKR and pFT exhibited comparable execution times. When $n = 100{,}000$, $m = 10$, and $k = 1000$, the execution times of pKR and pFT were around 2.13 times and 2.16 times faster than that of pAC. Overall, it showed a similar result to that of Experiments 2 and 3.

Table 4. Comparison of execution times of pAC, pKR, and pFT for ECG data.

(a) Comparison of execution times varying m when $n = 100{,}000$, $k = 1000$			
	Execution times of the algorithms (unit: ms)		
m	pAC	pKR	pFT
10	41.18	19.32	19.08
12	42.6	18.68	19.03
15	44.83	18.82	18.31
(b) Comparison of execution times varying k when $n = 100{,}000$, $m = 10$			
	Execution times of the algorithms (unit: ms)		
k	pAC	pKR	pFT
100	18.57	2.32	2.29
500	28.23	9.89	9.46
1000	41.18	19.32	19.08

5. Conclusions

In this paper, a parallel OPMPM algorithm using the Aho–Corasick automaton and a parallel OPMPM algorithm using a fingerprint table were proposed. In addition, comparison experiments were conducted for randomly generated strings, Dow Jones Index data, and electrocardiogram data to evaluate the performance of the proposed parallel algorithms. The experimental results showed that the execution times of the algorithms had a similar trend regardless of the data type when data-related parameters such as the length of the text, lengths of the patterns, and the number of patterns, as well as algorithm-related parameters such as the q-gram length, were kept the same. This suggests that the performance of the algorithms is dependent on these parameters rather than the specific characteristics of the data type. The performance of the verification of order-isomorphism may vary depending on the order representation method of strings used [16]. Therefore, further studies are necessary to compare the execution times and number of verifications required for different-order representation methods.

Author Contributions: S.P. and J.P. designed and analyzed the algorithms. S.P. implemented and experimented with the algorithms, and wrote the draft of the paper. Y.K. and J.S.S. reviewed and revised the paper. J.S.S. analyzed the algorithm, provided algorithmic support, and was the project manager. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2022-00155915, Artificial Intelligence Convergence Innovation Human Resources Development (Inha University)), and by INHA UNIVERSITY Research Grant.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare they have no conflicts of interest.

References

1. Kim, J.; Eades, P.; Fleischer, R.; Hong, S.H.; Iliopoulos, C.S.; Park, K.; Puglisi, S.J.; Tokuyama, T. Order-preserving matching. *Theor. Comput. Sci.* **2014**, *525*, 68–79. [CrossRef]
2. Kim, J.; Amir, A.; Na, J.C.; Park, K.; Sim, J.S. On representations of ternary order relations in numeric strings. *Math. Comput. Sci.* **2017**, *11*, 127–136. [CrossRef]
3. Cho, S.; Na, J.C.; Park, K.; Sim, J.S. A fast algorithm for order-preserving pattern matching. *Inf. Process. Lett.* **2015**, *115*, 397–402. [CrossRef]

4. Kim, Y.; Kim, Y.; Sim, J.S. An improved order-preserving pattern matching algorithm using fingerprints. *Mathematics* **2022**, *10*, 1954. [CrossRef]
5. Na, J.C.; Lee, I. A simple heuristic for order-preserving matching. *IEICE Trans. Inf. Syst.* **2019**, *102*, 502–504. [CrossRef]
6. Aho, A.V.; Corasick, M.J. Efficient string matching: An aid to bibliographic search. *Commun. ACM* **1975**, *18*, 333–340. [CrossRef]
7. Han, M.; Kang, M.; Cho, S.; Gu, G.; Sim, J.S.; Park, K. Fast multiple order-preserving matching algorithms. In Proceedings of the International Workshop on Combinatorial Algorithms, Verona, Italy, 5–7 October 2015; pp. 248–259.
8. Park, J.; Kim, Y.; Sim, J.S. A space-efficient hashing-based algorithm for order-preserving multiple pattern matching problem. *KIISE Trans. Comput. Pract.* **2018**, *24*, 399–404. [CrossRef]
9. Park, K.B.; Kim, Y.; Sim, J.S. Parallel implementation of the order-preserving multiple pattern matching algorithm using the Karp-Rabin algorithm. *J. KIISE* **2021**, *48*, 249–256. [CrossRef]
10. Kubica, M.; Kulczyński, T.; Radoszewski, J.; Rytter, W.; Waleń, T. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.* **2013**, *113*, 430–433. [CrossRef]
11. Knuth, D. *The Art of Computer Programming, Seminumerical Algorithms*; Addison-Wesley: Boston, MA, USA, 1997; Volume 2.
12. Mareš, M.; Straka, M. Linear-time ranking of permutations. In Proceedings of the Algorithms–ESA 2007: 15th Annual European Symposium, Eilat, Israel, 8–10 October 2007; pp. 187–193.
13. Chhabra, T.; Tarhio, J. A filtration method for order-preserving matching. *Inf. Process. Lett.* **2016**, *116*, 71–74. [CrossRef]
14. Williamson, S. Daily Closing Values of the DJA in the United States, 1885 to Present, Measuring Worth. Available online: https://www.measuringworth.com/datasets/DJA/index.php (accessed on 17 December 2021).
15. Goldberger, A.L.; Amaral, L.A.N.; Glass, L.; Hausdorff, J.M.; Ivanov, P.C.; Mark, R.G.; Mietus, J.E.; Moody, G.B.; Peng, C.K.; Stanley, H.E. Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals. *Circulation* **2000**, *101*, e215–e220. Available online: http://circ.ahajournals.org/content/101/23/e215 (accessed on 20 September 2022). [CrossRef] [PubMed]
16. Park, S.; Kim, Y.; Sim, J.S. Comparison of order-isomophism verification times of two strings according to their representations. In Proceedings of the Korean Institute of Next Generation Computing Spring Conference; Gwangju, Republic of Korea, 1–13 May 2021; Korean Institute of Next Generation Computing: Seoul, Republic of Korea, 2021; pp. 350–353.