문자열의 suffix-prefix가 일치하는 모든 쌍 찾기를 위한 상수시간 RMESH 알고리즘

우진운*

요약

문자열 연산이 계산 생물학 분야에 응용되면서 효율적인 문자열 연산을 위한 다양한 자료구조와 알고리즘이 연구되고 있다. suffix-prefix가 일치하는 모든 쌍 찾기는 두 개 이상의 문자열이 주어질 때 각 쌍의 문자열에 대해 가장 긴 suffix와 일치하는 prefix를 찾는 것으로 가장 짧은 슈퍼스트링을 검출하는 근사 알고리즘에서 사용될 뿐만 아니라 생물정보학, 데이터 압축 분야에서도 중요하게 사용된다. 본 논문에서는 RMESH(Reconfigurable MESH) 구조에서 3-차원 $n \times n \times n$ 프로세서를 사용하여 문자열들의 suffix-prefix가 일치하는 모든 쌍 찾기를 위한 알고리즘을 제안하며, 이 알고리즘은 $O(1)$ 시간 복잡도를 갖는다.

Constant Time RMESH Algorithm for Finding All-pairs Suffix-prefix Matching of Strings

JinWoon Woo*

ABSTRACT

Since string operations were applied to computational biology area, various data structures and algorithms for computing efficient string operations have been studied. The all-pairs suffix-prefix matching is to find the longest suffix and prefix among given strings. The matching algorithm is importantly used for fast approximation algorithm to find the shortest superstring, as well as for bio-informatics and data compressions. In this paper, we present an algorithm to find all-pairs suffix-prefix matchings of the given strings using three-dimensional $n \times n \times n$ processors on RMESH(Reconfigurable MESH). The algorithm has $O(1)$ time complexity.

Key Words : string, suffix-prefix matching, string matching, RMESH, time complexity

* 단국대학교 소프트웨어학과(✉jwwoo@dankook.ac.kr)
· 제1저자(First Author) : 우진운 · 교신저자(Correspondent Author) : 우진운
· 접수일(2011년 5월 20일), 수정일(1차 : 2011년 6월 15일), 게재확정일(2011년 6월 17일)

Ⅰ. 서 론

최근 문자열 연산들이 계산 생물학 분야에 응용되면서 효율적인 문자열 연산을 위한 자료구조와 알고리즘들이 연구되고 있다. 이러한 문자열 연산에는 문자열 패턴 매칭(string pattern matching), 접미사-접두사 매칭(suffix-prefix matching), 최장 공통 부분문자열(longest common substring) 찾기, 최대 반복자(maximal repeat) 찾기 등이 있다.

suffix-prefix가 일치하는 모든 쌍 찾기는 두 개 이상의 문자열이 주어질 때 각 쌍의 문자열에 대해 가장 긴 suffix와 일치하는 prefix를 찾는 것을 의미한다[7]. Suffix-prefix가 일치하는 문자열을 찾는 문제는 하나 또는 여러 개의 문자열에서 존재하는 단일 패턴 또는 멀티 패턴의 모든 위치를 찾고, 존재하는 패턴들의 출현 횟수를 검출하며[3,5,17], 가장 짧은 슈퍼스트링(shortest superstring)을 검출하는 근사 알고리즘에서 사용될 뿐만 아니라, [12]에서 연구된 C.elegans의 유전자에서 ACRs의 개수와 위치를 산출하는데 사용되는 등 생물정보학 분야에서도 아주 중요하게 사용된다[7].

문자열 연산과 관련된 병렬 알고리즘이 많이 연구되어 있다[1,2,4,6,8,9,16,17]. 대부분의 병렬 알고리즘은 CRCW-PRAM 모델에서 동작하거나[6,8,16,17], RMESH 모델에서 동작한다[2,4,9].

RMESH 모델이 제안된 이래, 영상처리 분야에서는 많은 RMESH 알고리즘들이 발표되었으나, 문자열 처리 분야에는 활발하게 발표되지 못하고 있다. Lee와 Ercal이 RMESH 모델에서 상수시간 문자열 매칭 알고리즘을 제안하였고[8], Datta와 Subbiah는 RMESH 모델에서 문자열 처리를 위한 상수시간 알고리즘들을 제안하였다[2].

본 논문은 RMESH 모델에서 문자열들의 suffix-prefix가 일치하는 모든 쌍 찾기를 위한 상수 시간 알고리즘들을 제안한다. 문자열들의 suffix-prefix가 일치하는 모든 쌍 찾기를 위한 알고리즘은 동적 프로그래밍 기법을 사용하며 RMESH 모델의 각 프록세서의 스위치를 효율적으로 작동시킴으로써 상수 시간에 동작한다.

Ⅱ. RMESH 구조

RMESH는 Reconfigurable MESH의 약어로서 기존의 메쉬(mesh) 구조에 동적으로 재구성 가능한 버스 시스템을 결합한 구조로서 Miller, Prasanna-Kumar, Reisis, Stout에 의하여 제안되었으며[14], 구조적인 장점 때문에 다양한 분야에서 연구되었고 다양한 상수시간 알고리즘들이 개발되었다[1,10].

2.1 2-차원 RMESH

크기가 $n \times n$인 2-차원 RMESH의 기본 구조는 메쉬이며 프로세서들 사이의 통신을 위하여 브로드캐스트 버스(broadcast bus)가 존재한다. 예를 들어, 그림 1은 4×4 RMESH 구조를 보여준다. 프로세서들을 식별하기 위해 각 프로세서에게 $PE\ (i, j)$ 를 부여한다. 이때 $0 \leq i, j < n$, i는 행의 인덱스이고, j는 열의 인덱스이다.

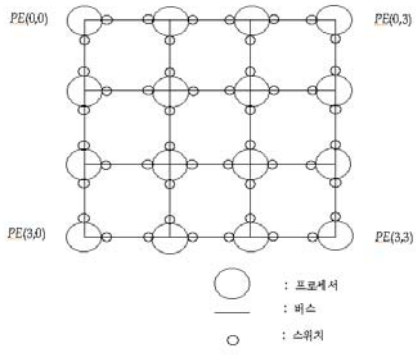
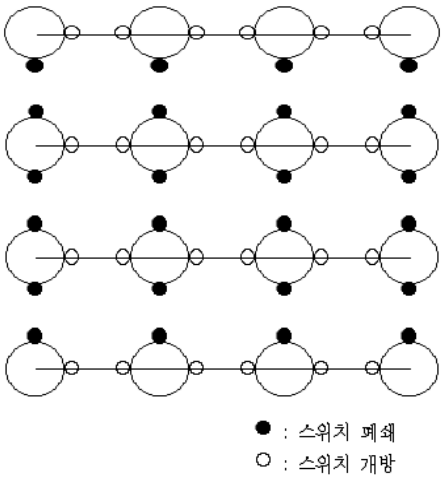


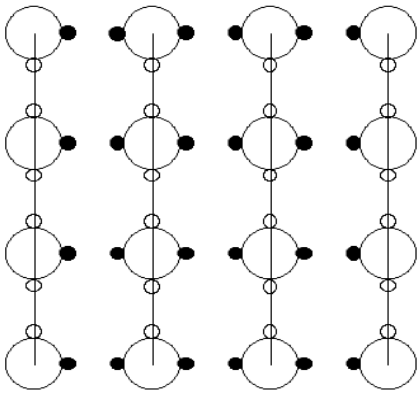
그림 1. 4×4 RMESH 구조
Fig. 1. RMESH structure

브로드캐스트 버스상의 통신 제어를 위하여 버스 스위치가 있다. 버스 스위치들은 각 프로세서의 상, 하, 좌, 우에 하나씩 존재하는데, 이를 각각 N(north), S(south), W(west), E(east)라 한다. 버스 스위치는 각 프로세서의 소프트웨어에 의하여 $O(1)$ 시간에 조작되며, 스위치의 개폐 여부에 따라 브로드캐스트 버스를 다수의 서브버스(subbus)들로 재구성이 가능하다.

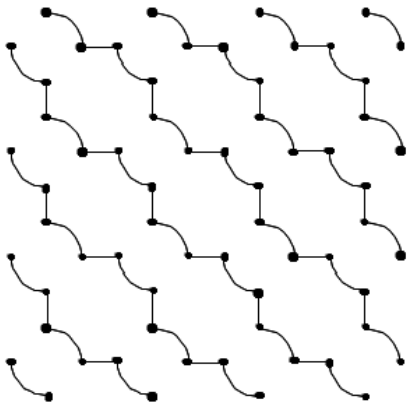
예를 들어, 각 프로세서가 자신의 S와 N 스위치를 끊고 E와 W 스위치를 연결한다면 여러 개의 서브버스가 형성되는데, 이를 그림 2(a)와 같은 행 버스(row bus)라 하고, 자신의 E와 W 스위치를 끊고 S와 N 스위치를 연결한다면 여러 개의 서브버스가 형성되는데, 이를 그림 2(b)와 같은 열 버스(column bus)라 한다. 또한 각 프로세서가 외부적으로 S와 N 스위치, E와 W 스위치를 연결하고, 내부적으로 N과 E 스위치, S와 W 스위치를 연결하게 되면 여러 개의 서브버스가 형성되며 이를 그림 2(c)와 같은 대각선 버스(diagonal bus)라 한다.



(a) 행 버스



(b) 열버스



(c) 대각선 버스

그림 2. 행, 열, 대각선 서브버스
Fig. 2. row, column, and diagonal bus

두 개의 프로세서들은 충돌이 없는 한 공통된 하나의 특정 스위치를 동시에 개폐할 수 있다. 버스상에는 특정 시간에 단 하나의 프로세서만이 데이터를 실을 수 있으며, 서브버스 위에 실린 데이터는 단위 시간에 그 버스에 연결된 모든 프로세서에게 전달될 수 있다. 만약 한 프로세서가 서브버스상에 있는 모든 프로세서에게 레지스터(register) X의 값을 브로드캐스트하려면 broadcast(X) 명령을 사용하고, 브로드캐스트 버스의 내용을 읽어 레지스터 R에 저장하려면 R :=

content(broadcast bus) 명령을 사용한다. 따라서 데이터 브로드캐스트는 $O(1)$ 시간에 수행된다.

2.2 3-차원 RMESH

2-차원 RMESH를 확장하여 3-차원 RMESH를 구성할 수 있다. 3-차원 RMESH에서는 각 프로세서에게 PE (l, i, j)를 부여한다. 이때 $0 \le l, i, j < n$, l은 각 프로세서가 위치한 계층(layer)이고, i와 j는 계층 l에서의 행과 열의 인덱스이다. 예를 들어, 그림 3은 $4\times4\times4$ RMESH를 보여준다. 버스 스위치들은 기본적으로 2-차원 RMESH와 같이 N, S, W, E 스위치가 존재하며, 추가적으로 각 프로세서마다 계층을 연결하는 U(up)와 D(down) 스위치가 존재한다. 그리고 모든 프로세서의 N, S, W, E 스위치를 끊고 U와 D 스위치를 연결하면 여러 개의 서브버스가 형성되는데, 이를 UD 버스라 한다.

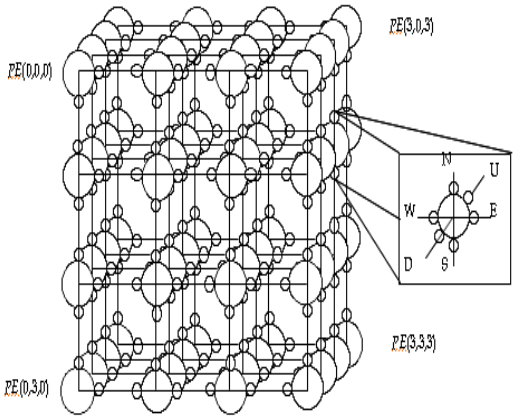


그림 3. $4\times4\times4$ RMESH 구조
Fig. 3. $4\times4\times4$ RMESH structure

III. Suffix-Prefix가 일치하는 모든 쌍 찾기

suffix-prefix가 일치하는 모든 쌍 찾기는 두 개 이상의 문자열이 주어질 때 각 쌍의 문자열에 대해 가장 긴 suffix와 일치하는 prefix를 찾는 문제이다[7].

예를 들어, 세 개의 문자열 S1=xbaxab, S2=abxb, S3=axabaxba 에 대해 suffix-prefix가 일치하는 모든 쌍 찾기의 결과는 다음과 같다. S1과 S2의 suffix-prefix는 'ab', S1 과 S3의 suffix-prefix는 'axab'가 되며, S2 와 S1의 suffix-prefix는 'xb', S2 와 S3의 suffix-prefix는 존재하지 않는다. 마지막으로, S3 와 S1의 suffix-prefix는 'xba'가 되고, S3 와 S2의 suffix-prefix는 'a'가 된다. 이러한 결과를 표시하면 표 1과 같게 된다. 표 1에서 숫자는 Si 와 Sj 의 suffix-prefix가 일치하는 Si의 가장 긴 suffix의 길이에 해당한다. 그리고 0은 일치하는 suffix-prefix가 존재하지 않음을 의미한다[5].

suffix-prefix가 일치하는 모든 쌍 찾기를 위해 k 개의 문자열, S1, S2, ..., Sk 가 주어질 때, 각 문자열의 길이를 ni $(1 \le i \le k)$ 라 하자. 접미사 배열을 이용하기 위해 k 개의 문자열을 연결하여 하나의 긴 문자열 S = { S1, S2, ..., Sk }을 생성하며, 문자열들을 구별하기 위해 연결되는 문자열 사이에 특수 문자 '$'을 넣는다. S 의 길이 n = $\Sigma n_i + m + 1$ 이며 m 은 문자열들의 수이다.

RMESH 구조에서 문자열의 suffix-prefix가 일치하는 모든 쌍 찾기를 위해서는 동적 프로그래밍 기법을 적절히 이용한다. 동적 프로그래밍은 두 문자열의 편집 거리(edit distance) 계산을 통한 방식으로 suffix-prefix가 일치하는 모든 쌍을 찾는다[13,15]. 이차원 테이블에서 각 테이블 원소의 행과 열에 해당하는 문자가 일치하면 1, 그렇지 않으면 0으로 표시한 후에 대각선 방향으로 연속된 1의 개수가 편집 거리에 해당한다. 이때 편집 거리가 1 이상이 되는 부분문자열들 중 앞 뒤에 '$' 문자와 연결된 것이 suffix-prefix 또는 prefix-suffix가 일치하는 부분으로 suffix-prefix 부분만을 선택한다.

표 1. suffix-prefix가 일치하는 모든 쌍 찾기의 예
Table 1. an example for finding all pairs suffix-prefix matching

	S_1=xbaxab	S_2=abxb	S_3=axabaxba
S_1=xbaxab	-	2	4
S_2=abxb	2	-	0
S_3=axabaxba	3	1	-

그림 4와 같게 된다. 그림 4에서 '$' 문자를 행으로 받

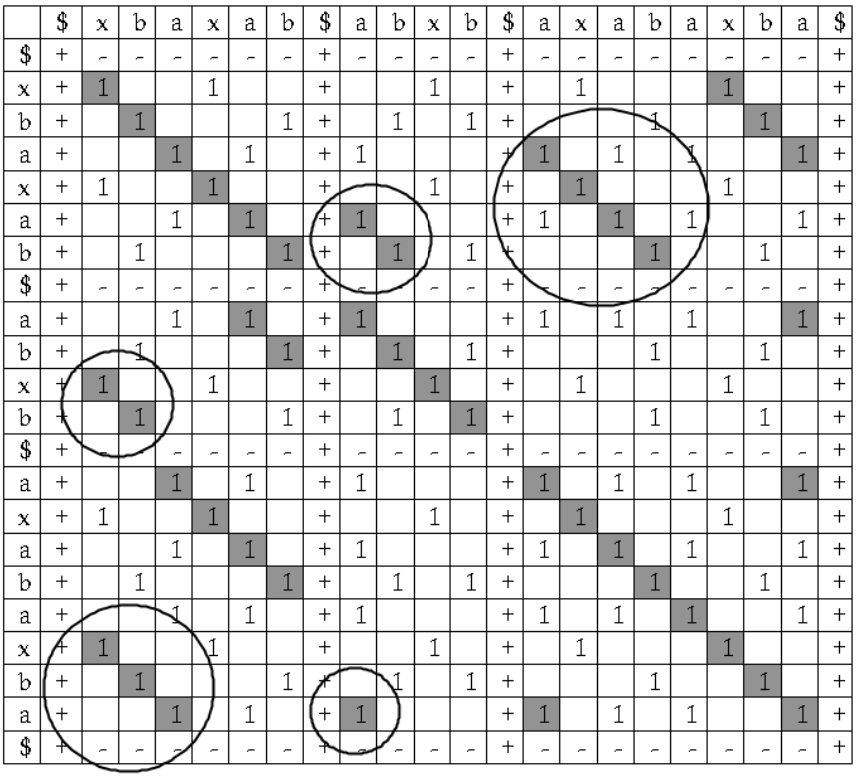


그림 4. suffix-prefix가 일치하는 모든 쌍 찾기 예
Fig 4. an example for finding all pairs suffix-prefix matching

이 절의 앞에서 소개된 예를 들어보자. S1=xbaxab, S2=abxb, S3=axabaxba 에 대해 이들을 연결한 긴 문자열을 만들면 S =$xbaxab$abxb$axabaxba$ 가 되며, 이 차원 테이블에서 각 테이블 원소의 행과 열에 해당하는 문자가 일치하면 1, 그렇지 않으면 0 으로 표시하면 은 원소는 '-', '$' 문자를 열로 받은 원소는 '+'로 표시하며, 행과 열로 동시에 받은 원소는 '+'로 표한다.

그림 4에서 짙게 표시된 부분은 앞 뒤에 '+' 문자 또는 '-' 문자와 연결된 편집 거리가 1 이상인 부분들로 '+' 와 '-' 로 연결된 부분은 suffix-prefix 가 일치하는

부분이고, '-' 와 '+' 로 연결된 부분은 prefix-suffix가 일치하는 부분이다. 따라서 suffix-prefix가 일치하는 부분만을 추출하면 되며 그림 4에서 원 안에 표시된다.

이제 추출된 부분의 문자열들에 대해 편집거리를 결정하여 suffix-prefix가 일치하는 모든 쌍 찾기를 완성한다. 앞의 예에서 세 개의 문자열 S1=xbaxab, S2=abxb, S3=axabaxba 에 대해 S1과 S2의 suffix-prefix는 'ab'로서 길이가 2, S1 과 S3의 suffix-prefix는 'axab'로서 길이가 4, S2 와 S1의 suffix-prefix는 'xb'로서 길이가 2, S2 와 S3의 suffix-prefix는 존재하지 않으며, S3 와 S1의 suffix-prefix는 'xba'로서 길이가 3, S3 와 S2의 suffix-prefix는 'a'로서 길이가 1이다. 이것은 표 1에서 표시되며, 그림 4에서 원 안에 표시된 부분 문자열들의 편집거리에 해당한다. 주어진 문자열들을 연결하여 하나의 긴 문자열로 만들었을 때 길이가 n 이라면 (문자열을 구별하기 위해 삽입하는 문자 '$'을 포함), $n \times n \times n$ 3차원 RMESH 구조에서 suffix-prefix가 일치하는 모든 쌍 찾기를 위한 알고리즘을 요약하면 알고리즘 1과 같이 7 단계로 구성된다.

초기에, 연결된 긴 문자열이 개별 문자열 번호와 함께 $n \times n \times n$ RMESH의 계층 0에 속하는 프로세서들의 첫 번째 행에 저장되어 있다고 가정한다. 즉, 계층 0의 프로세서 $PE(0,0,j)$ $(0 \le j \le n-1)$는 연결된 긴 문자열의 한 문자와 그 문자가 속한 개별 문자열 번호를 가진다.

예를 들어 그림 4에서 첫 문자열의 시작 문자 'x'는 계층 0의 프로세서 $PE(0,0,1)$에 문자열 번호 1과 함께 저장되며, 계층 0의 프로세서 $PE(0,0,13)$는 세 번째 문자열의 시작 문자 'a'와 문자열 번호 3을 저장한다.

[단계 1]: 첫 번째 행의 프로세서에 저장된 문자열을 첫 번째 열의 프로세서에 복사한다.

[단계 2]: 계층 0의 첫 번째 행에 속하는 프로세서 $PE(0,0,j)$ $(0 \le j \le n-1)$는 자신의 문자와 문자열 번호를 열 버스를 이용하여 브로드캐스트하고, 계층 0의 첫 번째 열에 속하는 프로세서 $PE(0,i,0)$ $(0 \le i \le n-1)$는 자신의 문자와 문자열 번호를 행 버스를 이용하여 브로드캐스트한다.

[단계 3]: 계층 0의 모든 프로세서 $PE(0,i,j)$는 행과 열 버스로 전달받은 두 문자를 서로 비교하여 같으면 대각선 버스를 형성하고, 같지 않으면 대각선 버스를 차단한다. '$' 문자를 포함하는 프로세서는 대각선 버스를 형성한다.

[단계 4]: 대각선 버스를 이용하여 프로세서들을 세그먼트로 분리하며, 세그먼트의 첫 프로세서는 *Start* 레지스터를 1로, 마지막 프로세서는 *End* 레지스터를 1로 설정한다.

[단계 5]: *Start* = 1 인 프로세서는 자신의 프로세서 *id* 값을 대각선 버스를 통하여 브로드캐스트한다.

[단계 6]: *End* = 1 인 프로세서는 전달받은 프로세서의 *id* 값을 이용하여 편집 거리인 *Distance* 를 계산하여, *(k, l, Distance)* 값을 저장한다. 이 때 *k* 는 행 버스로 전달받은 문자열 번호이고, *l* 는 열 버스로 전달받은 문자열 번호이다.

[단계 7]: *(k, l, Distance)* 크기의 오름차순으로 정렬한다.

알고리즘 1. suffix-prefix가 일치하는 모든 쌍 찾는 RMESH 알고리즘

알고리즘 1을 pseudocode로 나타내면 알고리즘 2와 같다.

Assume S is the concatenated string of given strings with length *n*.

[step 1]: $PE(0,0,j)(0 \le j \le n-1) \leftarrow S_j$

[step 2]: $PE(0,0,j)(0 \le j \le n-1)$ broadcasts (S_j, j) using the column bus.
$PE(0,i,0)\ (0 \le i \le n-1)$ broadcasts (S_i, i) using the row bus.

[step 3]: $PE(0,i,j)\ (0 \le i,j \le n-1)$ compares S_i and S_j.
if (S_i == S_j) set up the diagonal bus.
else block the diagonal bus.
Processor with '$' sets up the diagonal bus.

[step 4]: Form segments (SG_1, SG_1, ... SG_k) using the diagonal bus.
Start of the Fisrt PE of SG_i = 1;
End of the Fisrt PE of SG_i = 1; $(1 \le i \le k)$

[step 5]: For each processor PE, if *Start* == 1, PE broadcasts processor id using the diaognal bus.

[step 6]: For each processor PE, if *End* == 1, PE computes the edit distance *Distanct* and stores *(k, l, Distance)*. *k* is the string number broadcasted by the row bus and *l* is the string number broadcasted by the column bus.

[단계 7]: sort *(k, l, Distance)* by the non-decreasing order.

알고리즘 2. suffix-prefix가 일치하는 모든 쌍 찾는 pseudocode

Suffix-prefix가 일치하는 모든 쌍 찾기를 위한 RMESH 알고리즘이 수행되는 과정을 단계별로 살펴보자.

[단계 1]에서는 계층 0의 첫 번째 행의 프로세서 $PE(0,0,j)\ (0 \le j \le n-1)$가 자신의 문자와 문자열 번호를 계층 0의 첫 번째 열의 프로세서 $PE(0,i,0)\ (0 \le i \le n-1)$에게 전달한다. 이 과정을 위해 $PE(0,0,j)\ (0 \le j \le n-1)$는 열 버스를 이용하여 브로드캐스트한 후, 대각선 상의 프로세서 $PE(0,i,j)\ (i=j)$만이 전달된 문자와 문자열 번호를 행 버스를 이용하여 브로드캐스트한다. 이때 $PE(0,i,0)\ (0 \le i \le n-1)$ 만이 문자와 문자열 번호를 레지스터에 저장한다. 이 단계는 두 번의 브로드캐스트 만을 수행하므로 $O(1)$ 시간에 수행 가능하다.

[단계 2]에서 계층 0의 프로세서 $PE(0,0,j)\ (0 \le j \le n-1)$는 자신의 문자와 문자열 번호를 열 버스를 이용하여 브로드캐스트하고, 계층 0의 프로세서 $PE(0,i,0)\ (0 \le i \le n-1)$는 자신의 문자와 문자열 번호를 행 버스를 이용하여 브로드캐스트한다. 이때 각 프로세서 $PE(0,i,j)$는 두개의 문자와 문자열 번호를 저장하게 된다. 따라서 이 단계는 두 번의 브로드캐스트 만을 수행하므로 $O(1)$ 시간에 수행 가능하다.

[단계 3]에서 계층 0의 모든 프로세서는 행과 열 버스를 통해 전달받은 두 문자를 비교한다. 두 문자가 같으면 '1'로 표시하고, '$' 문자를 열 버스로 전달받은 프로세서는 '+'로 표시하고, '$' 문자를 행 버스로 전달받은 프로세서는 '-'로 표시한다. '$' 문자를 열 버스와 행 버스로 모두 받은 프로세서는 '+'로 표시한다.

표시된 문자에 따라 대각선 버스를 형성한다. 먼저 '1' 문자를 가진 프로세서는 양쪽 방향 대각선 버스를 형성하고, '+' 문자를 가진 프로세스는 우측 대각선 버스만을 형성하고, '-' 문자를 가진 프로세서는 좌측 대각선 버스만을 형성한다. 이러한 관계는 그림 4에 표현된다.

이 결과로 계층 0의 프로세서들이 대각선 방향으로 세그먼트를 형성하게 된다. 이 과정은 동적프로그래

밍 기법에서 설명된 바와 같이 두 문자가 같으면 1, 같지 않으면 0으로 나타내는 것과 같다. 이 단계는 단순히 한 번의 문자 비교와 버스의 개폐만 하게 되므로 $O(1)$ 시간에 수행 가능하다.

[단계 4]에서는 대각선 방향으로 세그먼트를 형성하게 되며, 대각선 버스에서 왼쪽에 '+' 문자를 가진 프로세서는 *Start* 레지스터에 1을 저장하고, 그렇지 않으면 0을 저장한다. 여기서 *Start* 레지스터가 1인 프로세서는 하나의 세그먼트 시작을 의미한다. 오른쪽에 '-' 문자를 가진 프로세서는 *End* 레지스터에 1을 저장하고, 그렇지 않으면 0을 저장한다. 여기서 *End* 레지스터가 1인 프로세서는 세그먼트의 마지막에 해당한다. 이 단계는 대각선 버스의 인접한 프로세서만을 확인하므로 $O(1)$ 시간에 수행 가능하다.

[단계 5]에서 *Start* = 1 인 프로세서는 세그먼트 내의 프로세서들에게 자신의 프로세서 *id* 값을 대각선 버스를 통하여 브로드캐스트 하는데, 프로세서 *id* 는 프로세서의 행과 열 번호로 나타낸다. 즉, 프로세서 $PE(0,i,j)$의 경우, *(i, j)*를 브로드캐스트한다. 이 과정은 세그먼트를 나타내는 대각선 버스 상에서 한 번의 브로드캐스트만 일어나므로 $O(1)$ 시간에 수행된다.

[단계 6]에서 *End* = 1 인 프로세서는 전달받은 프로세서의 *id* 를 이용하여 *(k, l, Distance)* 값을 저장한다. 여기서 *Distance*는 편집 거리에 해당하며 자신의 열 번호에서 전달받은 프로세서의 *id* 의 j 값을 뺀 값이다. 즉, 프로세서 $PE(0,i',j')$가 *(i, j)* 값을 전달받았다면, *Distance* = *j'-j+1* 로 계산된다. 그리고 *k* 는 단계 2에서 행 버스로 전달받은 문자열 번호이고, *l* 는 열 버스로 전달받은 문자열 번호이다. 한편, *End* = 0 인 프로세서는 $(\infty,\infty,0)$의 값을 저장한다. 이 단계에서는 세그먼트의 마지막 프로세서가 *(k, l, Distance)* 필드를 계산하는 시간만 소요하므로 걸리는 시간은 $O(1)$이다.

[단계 7]은 *(k, l, Distance)* 의 *k* 와 *l* 값을 기준으로 하여 오름차순으로 정렬한다. Nigam과 Sahni[11]는 rotate sort 알고리즘을 3-차원 $n\times n\times n$ RMESH에 적용하여 n^2 개의 데이터를 $O(1)$ 시간에 정렬할 수 있는 알고리즘을 제안하였는데, 이 알고리즘에서 초기의 n^2 개의 데이터는 계층 0에 속하는 $PE(0,i,j)$ $(0\le i,j<n)$에 존재하며, 정렬된 결과는 계층 0에 속하는 프로세서에 row-major 순서로 하나씩 저장된다. 즉, $PE(0,0,0)$, $PE(0,0,1)$, $\cdots$, $PE(0,1,0)$, $PE(0,1,1)$, $\cdots$, $PE(0,n-1,n-1)$의 순서로 저장된다. 이 단계에서 이 정렬 알고리즘을 이용할 때 *(k, l, Distance)* 값들을 $O(1)$ 시간에 정렬할 수 있다. 이때, 계층 0의 프로세서에 저장된 *(k, l, Distance)* 은 *k* 번째 문자열과 *l* 번째 문자열이 *Distance* 길이만큼 suffix-prefix가 일치하다는 것을 나타낸다.

예를 들어, 그림 4의 문자열에 대해 알고리즘1의 단계7까지 적용하면 최종적으로 (1,2,2), (1,3,4), (2,1,2), (3,1,3), (3,2,1)가 생성된다. 문자열 S_1=xbaxab, S_2=abxb, S_3=axabaxba 에 대해, (1,2,2)는 S_1과 S_2의 suffix-prefix의 길이가 2로서 'ab', (1,3,4)는 S_1 과 S_3의 suffix-prefix의 길이가 4로서 'axab', (2,1,2)는 S_2 와 S_1의 suffix-prefix의 길이가 2로서 'xb', (3,1,3)은 S_3 와 S_1의 suffix-prefix의 길이가 3으로서 'xba', (3,2,1)은 S_3와 S_2의 suffix-prefix의 길이가 1로서 'a'임을 나타낸다. 그리고 S_2 와 S_3의 suffix-prefix는 존재하지 않으며, 이것은 표 1에서 표시되고, 그림 4에서 원 안에 표시된 부분에 해당한다.

지금까지 알고리즘 1의 각 단계가 모두 $O(1)$ 시간에 수행될 수 있음을 설명하였으며, 다음 정리와 같이 요약할 수 있다.

정리: 주어진 문자열의 길이가 n 일 때, $n\times n\times n$ 3차원 RMESH 구조에서 suffix-prefix가 일치하는 모든

쌍 찾는 알고리즘은 $O(1)$ 시간에 수행된다.

Ⅳ. 결론

suffix-prefix가 일치하는 모든 쌍 찾기는 두 개 이상의 문자열이 주어질 때 각 쌍의 문자열에 대해 가장 긴 suffix와 일치하는 prefix를 찾는 문제이다. 이러한 문자열 연산들은 패턴 매칭, 유사도 측정 등의 문자열 처리 분야에서 중요하게 사용되고 있으며, 생물학 분야에서는 유전자의 패턴 출현 횟수와 위치를 산출하는 등에 적용 가능하다.

본 논문에서는 3-차원 $n \times n \times n$ RMESH 구조에서 문자열들의 suffix-prefix가 일치하는 모든 쌍을 찾는 상수 시간 알고리즘을 제안하였다. 이 알고리즘은 3-차원 RMESH 구조에서 상수 시간 데이터 브로드캐스팅과 같은 특성을 사용함으로써 $O(1)$ 상수 시간 복잡도를 가진다.

참고문헌

[1] 김경훈, 우진운, "*RMESH 구조에서 unaligned 선형사진 트리의 alignment를 위한 상수시간 알고리즘*", 정보과학회논문지, 제31권 1,2호, pp. 10-18, 2004.

[2] 우진운, "*문자열의 최대 palindrome을 구하기 위한 상수시간 RMESH 알고리즘*", 한국지식정보기술학회 논문지, 제 5권 제2호, pp. 59-67, 2010.

[3] 이상구 외, "*OpenMP를 이용한 새로운 COG 방식의 고속 병렬 퍼지 제어기*", 한국지식정보기술학회 논문지, 제 5권 제1호, pp 77-84, 2010.

[4] 한선미, 우진운, "*문자열의 최장 공통 부분문자열과 최대 반복자를 구하기 위한 상수시간 RMESH 알고리즘*", 정보처리학회 논문지 A, 제16-A권, 제5호, pp. 319-326, 2009.

[5] 한선미, 우진운, "*접미사 배열을 이용한 Suffix-Prefix가 일치하는 모든 쌍 찾기*", 정보처리학회 논문지:A, 제17권 제5호, pp. 221-228, 2010.

[6] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin, *"Parallel Construction of a Suffix Tree with Application"*, Algorithmica, vol. 3, pp. 347-365, 1988.

[7] D. Gusfield, *"Algorithms on Strings, Trees, and Sequences"*, Computer Science and Computational Biology, CamBridge University Press, 1997.

[8] G. Landau, and U. Vishkin, *"Fast parallel and serial approximate string matching"*, J. of Algorithms, vol. 10, pp. 157-169, 1989.

[9] H. Lee, and F. Ercal, *"RMESH Algorithms for Parallel String Matching"*, International Symposium on Parallel Architectures, Algorithms and Networks, pp. 223-226, 1997.

[10] J. Jang, H. Park, and V. Prasanna, *"A Fast Algorithm for Computing Histogram on a Reconfigurable Mesh"*, IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 17, No.2, pp.97-106, 1995.

[11] M. Nigam and S. Sahni, *"Sorting n Numbers On $n \times n$ Reconfigurable Meshes With Buses"*, Proceedings 7th Intern`ational Parallel Processing Symposium, pp.174-181, 1993.

[12] P. Green, D. Lipman, D. Hillier, R. Waterston, D. States, and J. M. Claverie. *"Ancient conserved regions in new gene sequences and the protein databases"*, Science, Vol.259, pp.1711-1716, 1993.

[13] R. A. Wagner, Michael and J. Fischer, *"The String-to-String Correction Problem"*, Journal of the ACM (JACM), Volume 21 , Issue 1, pp. 168 - 173, 1974.

[14] R. Miller, V. Prasanna-Kumar, D. Reisis, and Q. Stout, *"Parallel Computation on Reconfigurable Meshes"*, IEEE Transactions on Computers, Vol.42, No.6, pp.678-692, 1993.

[15] W. Masek and M. Paterson, *"A fast algorithm computing string edit distances"*, J. of Computer and System Sciences, 20(1), pp13-31, 1980.

[16] Z. Galil, *"A Constant Time Optimal Parallel String Matching Algorithm"*, J. of ACM, vol 42, pp. 908-918, 1995.

[17] Z. M. Kedem. G. M. Landau, and K. V. Palem, *"Parallel suffix-prefix-Matching Algorithm and Applications"*, SIAM Journal on Computing, Vol.25, NO.5, pp.998-1023, 1996.

감사의 글

이 연구는 2010년도 단국대학교 대학연구비의 지원으로 연구되었음

저자소개

우 진 운 (Jinwoon Woo)

1980년 서울대학교 수학교육과 (학사)

1989년 미국 University of Minnesota 전산학과 (박사)

1989년 ~ 현재 단국대학교 소프트웨어학과 교수

※ 관심분야: 분산 및 병렬처리, 병렬알고리즘, 자료구조