Paralelización del algoritmo Prefix Doubling para la construcción del Suffix Array

1st V. Benjamín Lazo Letelier
Instituto de Informática, FCI
Universidad Austral de Chile
Valdivia, Chile
vicente.lazo@alumnos.uach.cl

***Abstract*—Suffix array is a data structure widely used to solve problems of pattern recognition, compression and indexing of text. Prefix doubling is a simple algorithm that constructs the suffix array of a string in $O(n*log_2(n))$. When the algorithm is confronted with a string of considerable size, it has high construction times of the suffix array. This paper proposes a parallel version of the prefix doubling algorithm that implements radix sort parallel as an ordering algorithm. This parallelization seeks to decrease the suffix array construction time when the string size is large. In first instance we present the implementation of the prefix doubling algorithm. Then the steps to achieve the parallelization of the algorithm are listed. Finally it is demonstrated that our parallel version is 4.1 times faster than the sequential version.**

***Index Terms*—High-Performance Computing, parallelism, string, suffix array.**

I. Introducción

El *arreglo de sufijos* o *suffix array* (**SA**), es una estructura de datos creada a partir del orden lexicográfico de todos los sufijos de un string. Esta estructura almacena solamente los índices de los sufijos en un arreglo. El SA es usado en algoritmos de procesamiento de strings, principalmente en los que resuelven problemas de reconocimiento de patrones, compresión, indización de textos, entre otros. En el año 1970 Weiner [1] propuso una estructura de datos llamada *suffix tree*, que desde entonces ha sido ampliamente usada en algoritmos que resuelven problemas similares. Esta estructura de datos tiene una alta complejidad de implementación y requiere una alta cantidad de memoria para ser construida y almacenada. Es por esto que en 1990 Manber y Myers [2], propusieron el SA como una opción simple y que presenta un uso más eficiente de la memoria, frente al suffix tree.

En la actualidad existen diversos algoritmos para la construcción del SA [3], como son el algoritmo fuerza bruta (**FB**), la construcción a partir de un suffix tree y los informalmente llamados *prefix doubling* (**PD**), *recursive* e *induced copying*, teniendo una complejidad en su peor caso de $O(n^2 * log(n))$, $O(n)$, $O(n * log(n))$, $O(n)$ y $O(n^2)$ respectivamente, donde *n* corresponde al número de caracteres total que componen el string.

PD es un algoritmo sencillo propuesto por Manbern y Myers en 1993 [6]. En 1999 Larsson y Sadakane [4] implementaron un algoritmo 10 veces más rápido que el propuesto por Manbern y Myers, ocupando el mismo espacio y teniendo la misma complejidad [3]. PD busca ordenar iterativamente los sufijos por sus prefijos, duplicando la longitud de éstos en cada iteración. Su complejidad promedio es igual a la del peor caso, es decir $O(n * log(n))$, lo que puede significar un problema con strings de tamaño considerable, produciendo largos tiempos de espera para la construcción de su SA. Este problema lleva a plantear que *la paralelización del algoritmo prefix doubling podría mejorar el tiempo de construcción del SA*. Así, el objetivo del estudio es *desarrollar un algoritmo a través de la paralelización de prefix doubling para mejorar el tiempo de construcción del SA*, al cuál llamaremos prefix doubling paralelo (**PDP**).

II. Metodología

Para la realización de esta investigación se implementó el algoritmo PD. Luego este se paralelizó, generando el algoritmo PDP. La entrada a la que se sometieron los algoritmos PD y PDP fue la cadena de caracteres aleatorio ***s*** de la forma **s**[1..n], donde *n* es el largo de s y s_i es el i-ésimo carácter de s. SA_s es un arreglo, de la forma SA_s[1..n], de los índices de todos los sufijos de s, luego de que estos hayan sido ordenados lexicográficamente. La Figura 1 presenta un ejemplo de la construcción del SA de la palabra *s=banana*, con n=6, usando el algoritmo FB.

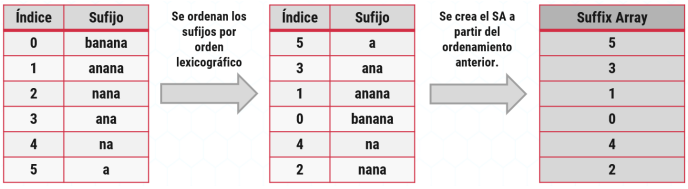

Fig. 1. Ejemplo de construcción del SA de la palabra *banana* usando FB.

La Sección II-A detalla la implementación seguida para construir el algoritmo PD. La Sección II-B muestra todos los pasos a seguir para paralelizar PD.

A. Algoritmo PD

Este estudio implementó el algoritmo propuesto por Papadopoulos y col. en el año 2016 [5]. La idea principal fue crear una *n-permutación*, iterando sobre los prefijos de los sufijos, usando un largo de prefijo que se va duplicando en cada iteración, es decir prefijos de largo $l_k = 0, 1, 2, 4, ..., log_2(n)$ y calculando el *ranking* de cada par de prefijo[i], prefijo[i+l_k], donde l_k es el largo del prefijo en la iteración k. Como el algoritmo duplica el valor de l_k en cada iteración, entonces el máximo número de interaciones para formar la n-permutación será de $log_2(n)$.

Algorithm 1: PrefixDoubling(s,SA)

```
Result: Suffix Array almacenado en SA
n = s.Length;
Sea R[1,n] arreglo de tripletes de la forma: (ranking,
  ranking+k, ind);
for i = 0 to n - 1 do
    R[i].first = ASCII(s[i]);
    R[i].second = ASCII(s[i+1]) : -1;
    R[i].third = i;
    //Se computa el valor del arreglo R_0, R_0 + k a partir
      de su código ASCII
end
Sort(R);
for k = 2 to n, k = k * 2 do
    rank = 0;
    R[0].first = rank;
    ind_real[R[0].third] = 0;
    for i = 2 to n do
        if R[i].first = R[i-1].first and R[i].second =
          R[i-1].second then
            R[i].first = rank;
        else
            rank = rank + 1;
            R[i].first = rank;
        end
        ind_real[R[i].third] = i;
        //Se computa el valor del arreglo R_{k/2}
    end
    for i = 1 to n do
        R[i].second = R[ind_real[R[i].third+k]].first : -1;
        //Se computa el valor del arreglo R_{k/2} + k
    end
    Sort(R);
end
for i = 1 to n do
    SA[i] = R[i].third;
    //Se computa el Suffix Array
end
```

El Algoritmo 1 detalla la implementación del algoritmo PD. Este tiene una complejidad de $O(n * log_2^2(n))$, donde su complejidad viene de que se realizan $log_2(n)$ iteraciones, y cada iteración realiza $O((2n)+(n*log_2(n)))$, donde $2n$ viene de los for, y $n*log_2(n)$ del algoritmo de ordenamiento (*Quick Sort*), lo que se reduce a $O(n * log_2^2(n))$. Si se utiliza *Radix Sort* como algoritmo de ordenamiento se puede disminuir la complejidad de cada iteración a $O(3n)$, ya que éste puede funcionar en $O(n)$ [7], logrando una complejidad teórica de $O(n * log_2(n))$.

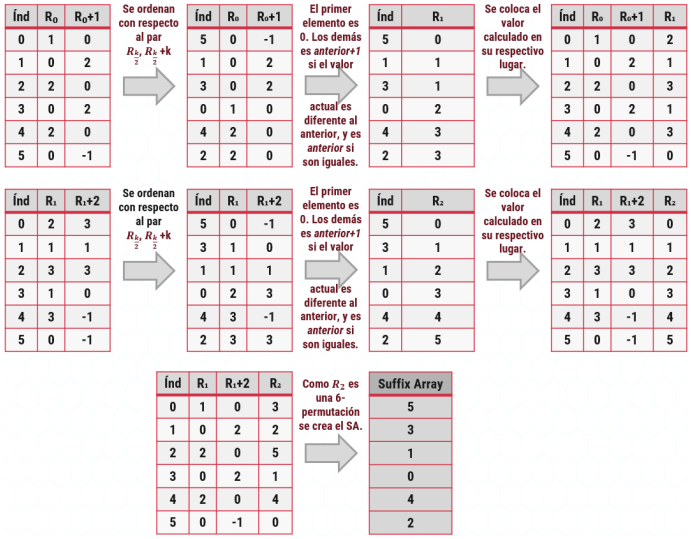

Fig. 2. Ejemplo de construcción del SA de la palabra *banana* usando PD.

La Figura 2 es un ejemplo de como funciona el Algoritmo 1 usando la palabra s = *banana*. Cabe destacar que $R_0[i]$ (ranking del prefijo de largo 0, del sufijo i) es el carácter mismo s_i. Una opción para el valor de $R_0[i]$ es su código de la tabla ASCII. Para el ejemplo planteado en la Figura 2 se asume que la tabla ASCII posee 3 elementos con los valores $a = 0$, $b = 1$ y $n = 2$. Como $s_1 = b$ entonces $ASCII(s_1) = 1$, por lo tanto $R_0[1] = 1$.

B. Algoritmo PDP

La implementación de PDP consiste en paralelizar directamente el Algoritmo 1 con la debida precaución en dos zonas importantes. Estas zonas son: el algoritmo de ordenamiento y el cálculo de $R_{\frac{k}{2}}$, y se deben abordar utilizando herramientas que facilitarán la paralelización.—

1) Paralelización del algoritmo de ordenamiento: Al tener la libertad de paralelizar cualquier algoritmo de ordenamiento, se optó por paralelizar el algoritmo Radix Sort, que busca ordenar un arreglo *A* en orden de unidades, decenas, centenas , ..., $log_{10}(d)$, donde d es el número de dígitos que posee el máximo número del arreglo A (ver Algoritmo 2). Este algoritmo usa como sub-rutina de ordenamiento el algoritmo *Counting Sort* paralelizado, que ordena contando el dígito presente en la iteración $d_i = [unidad, decena, ..., d]$ (ver Algoritmo 3).

El Algoritmo 2 realiza *d* subrutinas del Algoritmo 3. Cada ejecución del Algoritmo 3 realiza $(\frac{n}{p} + 10)$ instrucciones. Por lo que la complejidad final de Radix Sort Paralelo es de $O(\frac{n}{p})$.

Algorithm 2: RadixSortParalelo(R,AUX)

Result: Arreglo *R* y *AUX* ordenado
Sea **p** el número de threads;
d = ParallelMaxScanDigits(R);
for *i = 1 to d* **do**
  CountingSortParalelo(R,AUX, i);
end

Algorithm 3: CountingSortParalelo(R,AUX, dig)

Result: Arreglo *R* y *AUX* ordenado con respecto al dígito *dig*
Sea C[1..p][0..9] un arreglo doble de enteros de tamaño *p* y 9;
Sea tid el índice del thread que se ejecuta, en el rango [1..p];
sum = 0;
parallel for *i = 0 to R.Length - 1* **do**
  C[tid][R[i]] = C[tid][A[i]] + 1;
end
for *i = 0 to 9* **do**
  for *j = 1 to p* **do**
    aux = C[j][i];
    C[j][i] = sum;
    sum = sum + aux;
  end
end
parallel for *i = 0 to R.Length - 1* **do**
  AUX[C[tid][R[i]]] = R[i];
  C[R[i]] = C[R[i]] + 1;
end
parallel for *i = 0 to R.Length - 1* **do**
  R[i] = AUX[i];
end

Algorithm 4: PrefixDoublingParalelo(s,SA)

Result: Suffix Array almacenado en *SA*
n = s.Length;
Sea R[1,n] arreglo de tripletes de la forma: *(ranking, ranking+k, ind)*;
parallel for *i = 1 to n* **do**
  R[i].first = ASCII(s[i]);
  R[i].second = ASCII(s[i+1]) : -1;
  R[i].third = i;
  //Se computa el valor del arreglo $R_0, R_0 + k$ a partir de su código ASCII
end
RadixSortParalelo(R, AUX);
for *k = 2 to n, k = k * 2* **do**
  helper = ParallelNotEqualScan(R, AUX);
  su = ParallelSumScan(helper);
  parallel for *i = 1 to n* **do**
    R[i].first = su[i];
    ind_real[R[i].third] = i;
    //Se computa el valor del arreglo $R_{\frac{k}{2}}$
  end
  parallel for *i = 1 to n* **do**
    R[i].second = R[ind_real[R[i].third+k]].first : -1;
    //Se computa el valor del arreglo $R_{\frac{k}{2}} + k$
  end
  RadixSortParalelo(R, AUX);
end
parallel for *i = 0 to n - 1* **do**
  SA[i] = R[i].third;
  //Se computa el Suffix Array
end

2) Paralelización del cálculo de $R_{\frac{k}{2}}$: La paralelización de esta zona se separó en dos partes:

- Aplicar el algoritmo **Not Equal Scan Paralelo** entre los arreglos entregados por el Algoritmo 2, creando un arreglo auxiliar que almacenará el resultado de la comparación $A_i \neq AUX_{i-1}$.
- Al arreglo creado en el paso anterior, se le aplica el algoritmo de **Sum Scan Paralelo**, entregándo el resultado de $R_{\frac{k}{2}}$.

Resuelto el problema en ambas zonas, se paralelizaron los *for* que poseía el Algoritmo 1, creando así el Algoritmo 4. Este tiene una complejidad de $O(\frac{n}{p} * log_2(n))$, ya que tanto RadixSortParalelo y los for paralelos toman $O(\frac{n}{p})$ operaciones, los scans hace $O(log_2(n))$ operaciones y el algoritmo se realiza $O(log_2(n))$ veces. Uniendo todo queda $O((\frac{n}{p} + log_2(n)) * log_2(n))$ y reduciendo se llega a $O(\frac{n}{p} * log_2(n))$.

La Figura 3 muestra gráficamente como se ejecuta PDP para la palabra *s=banana*.

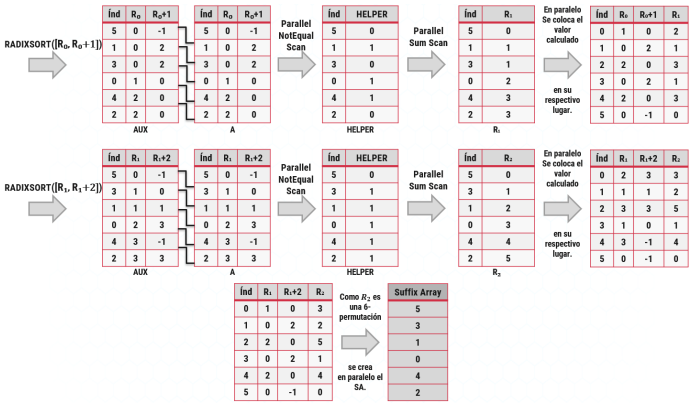

Fig. 3. Ejemplo del cálculo de R_1 y R_2 para la palabra *banana* usando PDP.

III. Experimentación

Para las pruebas se utilizó el servidor Guanaco del Instituto de Informática de la Universidad Austral de Chile. Éste posee 2 x Intel Xeon Gold 5117 @ 2.00GHz - 14 núcleos (28 en total), 256GB de RAM y ArchLinux x64.

Se implementó la versión PD y PDP. Se usaron strings aleatorios de tamaño dentro del rango entero $[2^{10}, 2^{25}]$. Cada prueba se ejecutó 10 veces para conseguir los promedios de cada tamaño de string. Las pruebas siguieron el Algoritmo 5.

Algorithm 5: Pruebas

```
Result: Tiempo de ejecución de PD y PDP para cada
        valor de n.
for i = 10 to 25 do
    seed = i;
    n = 2^i;
    for j = 1 to 10 do
        Ejecutar PD usando seed y n;
        Ejecutar PDP con 16 threads usando seed y n;
        Ejecutar PDP con 32 threads usando seed y n;
    end
end
```

IV. Resultados

Los resultados fueron obtenidos a partir de la ejecución del Algoritmo 5.

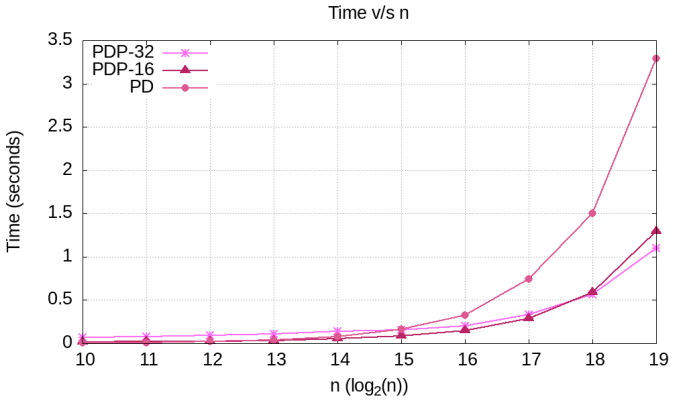

Fig. 4. Tiempo vs tamaño del string, en el rango $[2^{10}, 2^{19}]$.

La Figura 4 y 5 corresponden al tiempo promedio de 10 de ejecuciones de los algoritmos implementados para un tamaño de string en el rango $n = [2^{10}, 2^{25}]$.

La Figura 6 demuestra el speed-up por cada n, para el algoritmo PDP usando 32 (**PDP-32**) y 16 threads (**PDP-16**), el cálculo de esto se realiza de la forma $Sp = \frac{T_{PD}}{T_{PDP}}$.

La Figura 7 demuestra la eficiencia por thread para cada valor de n, de PDP-32 y PDP-16, el cálculo de esto se realiza de la forma $Ef = \frac{Sp}{p}$.

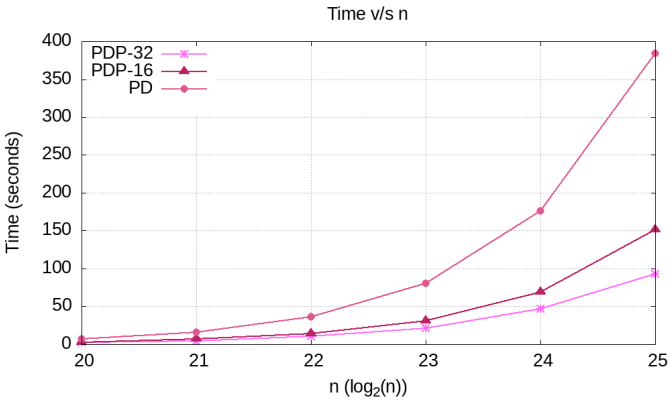

Fig. 5. Tiempo vs tamaño del string, en el rango $[2^{20}, 2^{25}]$.

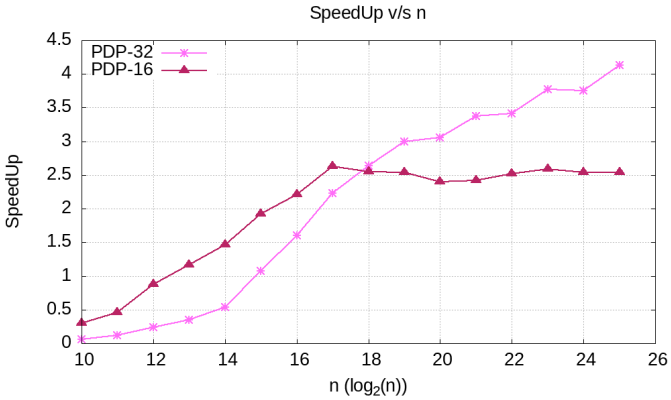

Fig. 6. SpeedUp vs tamaño del string.

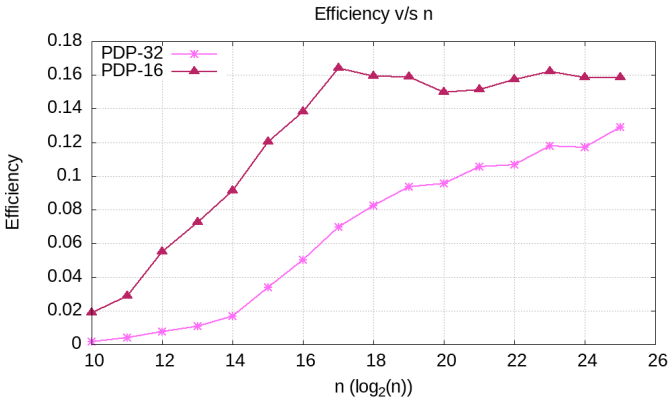

Fig. 7. Eficiencia por tread vs tamaño del string.

V. Análisis

Si se analizan las Figuras 1, 2 y 3, se puede concluir que los tres algoritmos realizan los mismos cálculos para construir el mismo SA, pero con diferentes complejidades. PDP es el que menos complejidad presenta con $O(\frac{n}{p} * log_2(n))$.

Las Figuras 4 y 5 muestra como el algoritmo PD se comporta de peor manera que la versión paralela.

El rango en que PDP-16 es mejor que PDP-32 es desde $k = [2^{10}, 2^{17}]$. En este rango se puede observar en la Figura 6 la clara superioridad de PDP-16 vs PDP-32, llegando a ser 2.6 veces más rápido que PD. Desde $m = [2^{18}, n = 2^{25}]$ PDP-16 no presenta mejoras y se mantiene igual a medida que se aumenta el tamaño del string, esto es apreciable en las Figuras 6 y 7.

La Figura 6 muestra como PDP-32 es 4.1 veces más rápido que PD, y su comportamiento denota que mientras más aumentemos el tamaño de string más aumentaría el speed-up de PDP-32.

La razón del porque PDP-16 es mejor que PDP-32 en el rango *k* es presentada en la Figura 7, que demuestra que siempre PDP-16 es más eficiente en el uso de sus threads que PDP-32, pero en el rango restante se mantiene igual, mientras que PDP-32 sigue aumentando su eficiencia a medida que aumenta el tamaño del string.

VI. Conclusión

Se logró implementar una solución paralela que es 4.1 veces más rápida que la versión secuencial. Para un $n = 2^{25}$, PDP-32 demoró 95 segundos en construir el SA, mientras PD demoró 390 segundos.

A mayor cantidad de núcleos mayor speed-up, ya que la complejidad de PDP depende de la cantidad de threads *p*.

Paralelizar un algoritmo no es sencillo, muchas veces existen pasos que no pueden ser paralelizados, por lo que debemos cambiar la perspectiva por donde se ataca el problema para lograr encontrar una solución paralela.

Como trabajo a futuro se pretende implementar PDP en GPU, donde se puede asumir que $n = p$, lo que disminuiría el orden de PDP a $O(log_2(n))$.

References

[1] P. Weiner.: Linear pattern matching algorithm. 14th Annual IEEE Symposium on Switching and Automata Theory, pages 1–11, 1973.

[2] S Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Philadelphia, PA, USA, pp. 319–327 (1990)

[3] Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2) (July 2007)

[4] Larsson, J. N. and Sadakane, K. 1999.: Faster suffix sorting. Tech. Rep. LU-CS-TR:99-214 [LUNFD6/(NFCS-3140)/1-20/(1999)], Department of Computer Science, Lund University, Sweden

[5] Papadopoulos, Agathoklis & Christodoulakis, Manolis & Theocharides, Theo. (2016).: Towards Hardware-accelerated Suffix Array Construction Architecture for the de novo DNA Sequence Assembly. 10.1109/MELCON.2016.7495406.

[6] Manber, U. and Myers, G. W. 1993.: Suffix arrays: a new method for on-line string searches. SIAM Journal of Computing 22, 5, 935–948.

[7] Davis, Ian. (1992).: A Fast Radix Sort.. Comput. J.. 35. 636-642. 10.1093/comjnl/35.6.636.