PaREM (Parallel Regular Expression Matching)

1st Gabriel A. Spranger
Dpto. Ciencia de la Computación
UTEC
Lima, Perú
gabriel.spranger@utec.edu.pe

2nd Macarena M. Oyague
Dpto. Ciencia de la Computación
UTEC
Lima, Perú
macarena.oyague@utec.edu.pe

3rd Alejandro Goicochea
Dpto. Ciencia de la Computación
UTEC
Lima, Perú
alejandro.goicochea@utec.edu.pe

***Abstract*—Se presenta la implementación del algoritmo *Parallel Regular Expression Matching* (PaREM) y *Parallel Automaton Matching* (PaAM). Evaluamos el *performance* de ambos algoritmos donde obtuvimos un speedup favorable que incrementaba con el tamaño del input. El algoritmo secuencial se presta a la paralelización lo cual se noto en los resultados de tiempos de ejecución donde una mayor cantidad de procesos, una vez que se llega a un input de tamaño considerable, reducía el tiempo total.**

I. Introduction

El problema de pareo de expresiones regulares tiene distintas aplicaciones. Por ejemplo, se puede diseñar una expresión regular para encontrar una subsecuencia exacta dentro de una secuencia de ADN. Los métodos conocidos hasta ahora para el pareo de expresiones regulares, solo se pueden hacer en $\mathcal{O}(n)$, es decir, se traduce la expresión regular a un autómata y luego se evalúa cada caracter, se pasa al estado correspondiente a través de la transición correspondiente y luego se repite el proceso hasta que ya no hayan más caracteres que analizar, donde si se llegó a un estado final o no, decide si se acepta el *string* o no. Dado que las secuencias de ADN son enormes, hay un impedimento de usar un algoritmo que corra en $\mathcal{O}(n)$. Es por eso que en este informe, presentamos la implementación de *PaREM* y su variante *PaAM*. Con esto, obtenemos una complejidad resultante de $\mathcal{O}(\frac{n}{p})$ (donde n es el tamaño del *string* y p es el número de *threads* usadas por el algoritmo, lo cual es una gran mejora comparado con el método anterior. Los detalles del análisis y resultados empíricos se detallan en la sección III.

La diferencia entre *PaREM* y *PaAM* es que como describen sus nombres, *PaREM* paraleliza el pareo de expresión regular, mientras *PaAM* paraleliza el pareo de un *string* con un patrón descrito por un autómata. Ambos algoritmos son muy parecidos. *PaREM* consta de dividir el *string* o *input* en p pedazos o *chunks*. La idea es hacer el pareo en cada *chunk*, pero empezando el pareo por cada estado de un conjunto de estados inciales R el cual es igual a la intersección entre otros dos conjuntos de estados S y L, donde S es el conjunto de estados a los que se puede llegar con el primer caracter del *chunk* asignado al *thread* i y L es el conjunto de estados a los que se pueden llegar con el último caracter del *chunk* del *thread* $i-1$. Por cada estado de R, se hace el pareo desde ese estado evaluando todo el *chunk*. Si en dicho pareo, llegamos a un estado final, entonces hay una alta probabilidad de que el *string* cumpla con el autómata del *PaREM* o *PaAM*. Se resalta la *alta probabilidad* porque en la reducción aseguramos que exista conectividad entre los estados inciales y finales de todos los *chunks*. Si esta conectividad no existe, entonces no hay pareo exitoso en el caso de *PaAM*, mientras que en el caso de *PaREM* solo basta con que un estado sea el final, no que todos los *chunks* se conecten entre sí. En la sección II se detalla a más detalle la implementación.

II. Método

Se ha utilizado el lenguaje de C++ para la elaboración de los códigos. Se pueden encontrar en el siguiente repositorio:

A. DFA

Debido a que es crucial la utilización de un Autómata de Estados finitos, se ha implementado una clase para poder abstraer los atributos y métodos que serán utilizados para el algoritmo principal. Los atributos esenciales que comprenden el autómata son los siguientes:

- *Q*: arreglo de estados del autómata
- *Alphabet*: arreglo de caracteres válidos para el lenguaje que corresponde al autómata
- *q0*: estado inicial del autómata
- *F*: arreglo de estados finales del autómata
- *Tt*: tabla de transiciones correspondientes al autómata, que corresponden a el salto entre un estado inicial hacia un estado subsiguiente por medio de una transición correspondiente al alfabeto

Los métodos esenciales que comprenden el autómata a manera de subrutina del algoritmo PaREM son los siguientes:

- *belongs_to_Q (qi, s)*: verifica si es que partiendo desde el estado *qi* y utilizando la transición *s* se puede llegar hacia un estado válido para retornar verdadero. Esto se debe a que no siempre existirá una transición válida correspondiente a un caracter del alfabeto a partir de un estado. Por ello, se consulta en la *Tt* a cuál estado llegaría partiendo de *qi* utilizando *s*, y si es que no se llega a ninguno, se retorna falso. Se podría utilizar un estado que no se encuentra dentro del conjunto *Q* como un estándar.
- *belongs_to_F (qi)*: verifica si es que el estado *qi* pertenece al conjunto de estados finales. Retorna verdadero si es que pertenece y falso si es que no.

B. Algoritmo PaREM

- *Recibe:* DFA: dfa, T: cadena de caracteres, p: número de procesos, mode: RegexMatch o AutomatonMatch
- *Devuelve:* verdadero si es que hay match y falso si es que no

Debido a que las condiciones del problema no requieren de una cantidad de memoria elevada, se ha paralelizado por medio de memoria compartida utilizando OpenMP. Por ello, se ha dividido la cadena de entrada del algoritmo en función a la cantidad de threads que ejecutarán.

En el inicio del algoritmo se define *chunk_size*, el tamaño de cada chunk en función a la longitud de la cadena. Además, y se definen arreglos que serán compartidos por todas las threads. La posición de cada elemento en la posición correspondiente al arreglo, que es correspondiente a su vez con el id del thread, puede ser entendida como una pareja debido a que *I* y *hasF* hacen referencia a la misma unidad de data, pero por cuestiones de tener un código más legible se optó por ser comprendidos en los siguientes arreglos de vectores: *I*, que es donde se almacenarán arreglos que comprendan los posibles estados visitados por cada proceso en función a cada elemento de $S \cap L$; y *hasF*, que contiene un booleano por cada vector anteriormente para poder responder a si es que alguno de esos estados se encuentra en la lista de estados finales *F*. Acto seguido se entra a la región paralelizada teniendo únicamente como variable privada *start_position* porque tendrá que cambiar su valor según el thread en el que nos encontremos. Se settea *i* llamando a la función `omp_get_thread_num()` para que cada thread pueda tener almacenado su id propio, el cual servirá para settear la posición de la subcadena en donde el thread operará.

```
bool PaREM(DFA& dfa, const string& T, int p, Mode
    mode) {

    int start_position;
    const int chunk_size = T.length()/p;

    vector <vi> I [p];
    vector <bool> hasF [p];

    #pragma omp parallel private(start_position)
    {

        int i = omp_get_thread_num();
        start_position = i*(chunk_size);
        ...
    }
    ...
}
```

Luego de ello, se tiene que realizar un análisis para saber las intersecciones posibles que pueden darse a partir del último caracter de la subcadena del proceso anterior, a quien llamaremos p_{i-1}, y el primer caracter de la subcadena perteneciente al proceso actual, p_i. Debido a que el primer thread no tendrá un estado anterior, se realiza un filtro para realizar el análisis con los procesos cuyo identificador *i* sea distinto a 0 que es el inicial.

Se crea un vector booleano *S* y uno *L* para almacenar partiendo de qué estado existe una transición válida para el último caracter del p_{i-1} y para el p_i, respectivamente. Por este motivo es que se asigna el tamaño de ambos vectores a la cantidad de estados válidos que se encuentran en el arreglo *Q* del *DFA* correspondiente a validar. Acto seguido, se itera por cada estado correspondiente al arreglo *Q* y se llama a la subrutina mencionada anteriormente *belongs_to_Q* del *DFA*. De esta manera se podrá saber si es que existe una transición válida utilizando el primer caracter de la subcadena de p_i a partir de cada estado *q* por el cual se está iterando. Si es que si existe, se marcará a *true* el estado. De manera similar se llama nuevamentea la subrutina *dfa_belongs_to_Q* pero ahora con el último caracter de la subcadena de p_{i-1}. Si es que sí existe una transición válida utilizando el caracter, en esta oportunidad no se marcará el estado del que proviene y, de lo contrario, se marcará a *true* el estado al cual se llegará al partir *q* con la transición dicha.

Esto se hace con la finalidad de poder tener el estado a partir del cual tendrá que partir el caracter de la subcadena del thread que. está ejecutandose. Por ello, para realizar el respectivo *match* se requiere hacer una reducción de cuáles son los estados hacia los cuáles el último caracter de p_{i-1} podrá dirigirse y al mismo tiempo a partir del cual el primer caracter de p_i pueda partir. El último *for* verifica en qué estados está prendido tanto para *S* como para *L* para conocer los estados requeridos. Si es que coinciden, o en otras palabras, hay una intersección entre ellos, se añadirán a un arreglo de enteros *R*, que es definido como el estado inicial para cada subcadena. En el caso del primer thread, sin embargo, el único estado inicial será el correspondiente al estado inicial del *DFA* conocido como *q0*.

```
...
#pragma omp parallel private(start_position)
{
    ...
    vi R;
    int j = 0;

    if (i != 0) {
        vb S(dfa.Q.size(), false);
        vb L(dfa.Q.size(), false);
        for (auto& q : dfa.Q) {
            if (dfa.belongs_to_Q(q, T[
start_position]))
                S[j] = true;
            j++;
        }
        for (auto& q : dfa.Q) {
            if (dfa.belongs_to_Q(q, T[
start_position-1]))
                L[dfa.Tt[q][string(1, T[
start_position-1])]] = true;
        }
        for (j = 0; j < dfa.Q.size(); j++){
            if (S[j] && L[j]) R.PB(j);
        }
    }

    else R.PB(dfa.q0);
    ...
}
...
```

Luego de haber obtenido $S \cap L$, se debe iterar por cada posible estado inicial de la subcadena que fue añadido en *R*. Se crea un vector auxiliar. *Rr* para conocer todas los estados por los que se pasa para cada caracter de la subcadena. Antes de realizar ello, se settea la posición final en *end_position* debido a que, si no existió una división perfecta para *chunk_size* porque no hubo una división exacta del tamaño de la cadena entre el numero de procesos *p*, el último proceso p_{p-1} tendrá sobrecarga. Para cada iteración, se debe verificar que partiendo de *r* es posible obtener una transición válida para cada caracter de la subcadena por medio de la llamada a *dfa_belongs_to_Q*, ya que si no es posible, no se debe tomar en cuenta ese *r* como un posible estado inicial de la subcadena. Si esto ocurre, se settea un flag y se intenta con el siguiente *r* del vector *R*. Asimismo, para el caso del Regex Match se necesita saber si alguno de los estados por los que se pasa es un estado final, debido a que al momento de realizar la reducción se podrá hacer una optmización si es que se tiene preprocesada dicha información. Cabe recalcar que para avanzar de un estado al siguiente, se prosigue el camino de estados teniendo a cada caracter como transición, y se debe hallar el siguiente estado haciendo una consulta en la *Tt* del *DFA*. Si es posible llegar al final de la cadena, se almacenan el vector que contiene todos los estados por los cuales pasa al iniciar en *r* y se realiza un *push_back* en el arreglo *I* en la posición proveniente al identificador *i* del proceso, marcando a su vez si es que se tuvo como estado final a alguno de ellos.

```
...
#pragma omp parallel private(start_position)
{
    ...
    int k;

    for (auto r : R) {
        bool flag = 0, foundQ = false;
        vi Rr;
        k = r;
        Rr.PB(r);
        int end_position;
        if (i < p-1)
            end_position = start_position+
chunk_size;
        else end_position = T.length();
        for (j = start_position; j <
end_position; ++j) {
            if (!dfa.belongs_to_Q(k, T[j]))
                { flag = 1; break; }

            if (dfa.belongs_to_F(k))
                foundQ = true;

            k = dfa.Tt[k][string(1, T[j])];
            Rr.PB(k);
        }
        if (flag) continue;
        I[i].PB(Rr);
        hasF[i].PB(foundQ);
    }
    ...
}
...
```

Listing 1: Evaluar cada *chunk* con el autómata.

La reducción sobre I la implementamos con un algoritmo similar al DFS (*Depth First Search*). Recordar que I es un vector que tiene un elemento por cada *thread*, es decir, tiene p elementos. Cada elemento, es un vector de vectores, que guarda la lista de estados visitados al evaluar cada caracter del *chunk* sobre el autómata, empezando desde cada estado del conjunto R. En otras palabras, dependiendo del tamaño de R para el *thread* i, entonces el elemento i-ésimo de I será un vector de vectores con exactamente $|R|$ listas.

Teniendo esto en cuenta, el DFS empieza desde $I[0]$ e itera por cada lista de estados visitados, revisa el último estado visitado de cada una de estas listas y revisa si este es igual a el primer estado visitado en alguna de las listas de $I[1]$. Si son iguales, entonces se llama DFS nuevamente, pero ahora empezando desde $I[1]$ y concentrándose en la j-ésima fila (se asume que el primer estado vistado de $I[1]$ estaba en la lista j-ésima). Si el algoritmo es *PaAM*, entonces el DFS retorna `true` si existen *matches* hasta $I[p-1]$ y el último estado de esa lista de estados visitados, pertenece al conjunto F de estados finales. Mientras que si el algoritmo es *PaREM*, basta con que algún estado de la última lista conectada de estados visitados pertenezca al conjunto F de estados finales, para que el DFS retorne `true`. Cabe resaltar, que uno pensaría que el DFS de *PaREM* tomaría $\mathcal{O}(n)$, pero no es así, ya que nos apoyamos de un *clon* de I que tiene solo *booleanos* la cual llenamos mientras llenamos las listas de estados iniciales, para mno afectar la complejidad total mas que por un factor constante. De esta manera, con una simple consulta en $\mathcal{O}(1)$, podemos revisar si algún estado dentro de la actual lista de estados visitados, pertenece a F. La complejidad de esta reducción (DFS) es $\mathcal{O}(p)$.

```
bool PaREM(DFA& dfa, const string& T, int p, Mode
    mode) {
    ...
    return dfs(I, hasF, p, dfa, mode);
}

bool dfs(vector<vi>* I, vector<bool>* hasQ, int p,
    DFA& dfa, Mode mode) {
    bool si = false;
    for (auto v : I[0]) {
        int j = 0;
        for (auto e : I[1]) {
            if (v.back() == e.front()) {
                si = dfs_helper(dfa, I, hasQ, 1, j,
    p, mode);
                if (si) return true;
            }
            j++;
        }
    }
    return false;
}

bool dfs_helper(DFA& dfa, vector<vi>* I, vector<bool
    >* hasQ, int idx, int jdx, int p, Mode mode) {
    int j = 0;
    if (idx != p-1) {
        if (mode == RegexMatch && hasQ[idx][jdx])
    return true;
        bool si = false;
        int v = I[idx][j].back();
```

```
            for (auto e : I[idx+1]) {
                if (v == e.front()) {
                    si = dfs_helper(dfa, I, hasQ, idx+1,
     j, p, mode);
                    if (si) return true;
                }
                j++;
            }
            return false;
        } else {
            int v = I[idx][j].back();
            if (dfa.belongs_to_F(v)) {
                return true;
            } else {
                return false;
            }
        }
}
```

III. Resultados

Para obtener los resultados, se ejecutó el programa con 4 tamaños (n) diferentes. Cada uno de estos tamaños se ejecutó con una cantidad diferente de threads y varias veces para asi obtener un resultado promedio para cada caso. Las secciones en las que se midió el tiempo fueron la generación del DFA, el cual sesolo de genera una vez y se reusa para todos los casos y la función de PaREM en si la cual toma como input el string y la cantidad de threads que utilizaremos. Vale recalcar que la cantidad de threads se define antes de que empieze la función mediante `omp_set_num_threads(i)`. Los resultados fueron los siguientes:

Generación de DFA: 560.209 µs

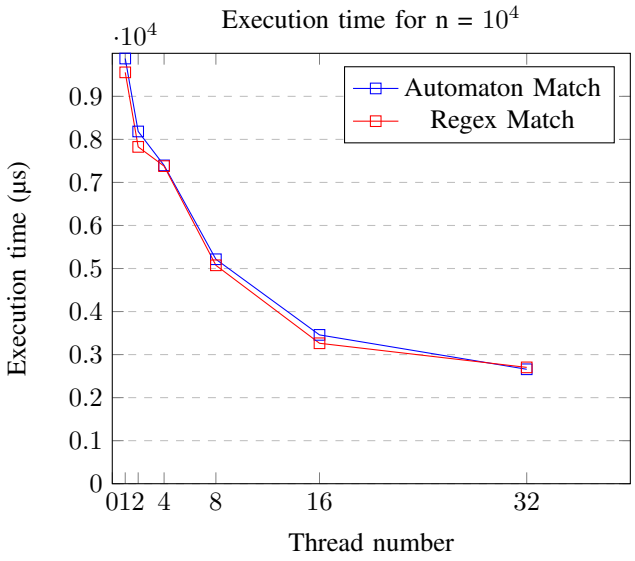
Función PaREM:

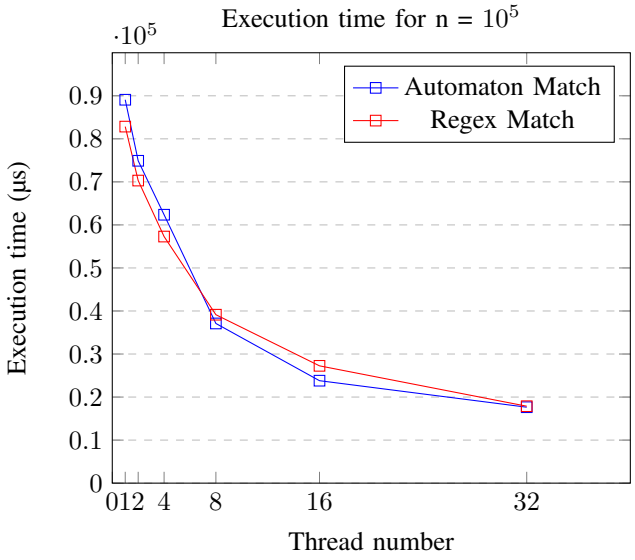
TABLE I: Automaton Match

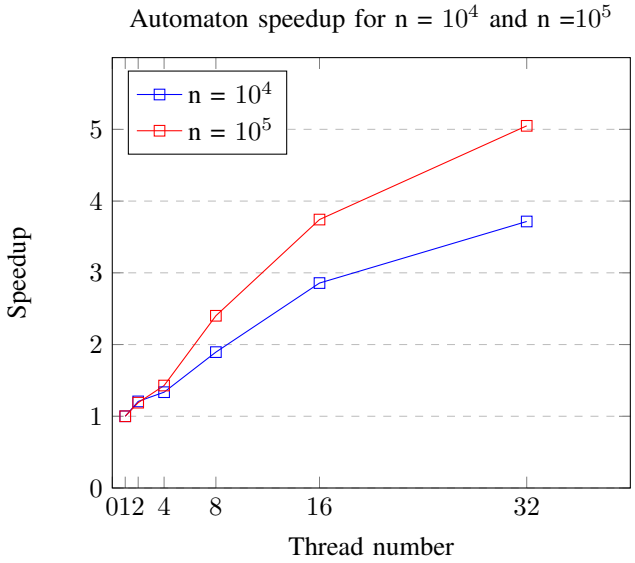
		n			
	µs	10^2	10^3	10^4	10^5
p	1	375.89	704.05	9879.06	89081.18
	2	439.413	623.392	8184.08	74896.4
	4	308.198	676.08	7396.08	62362.5
	8	265.855	693.14	5215.01	37101.2
	16	297.25	772.879	3457.59	23803.1
	32	382.969	615.168	2658.76	17642.8

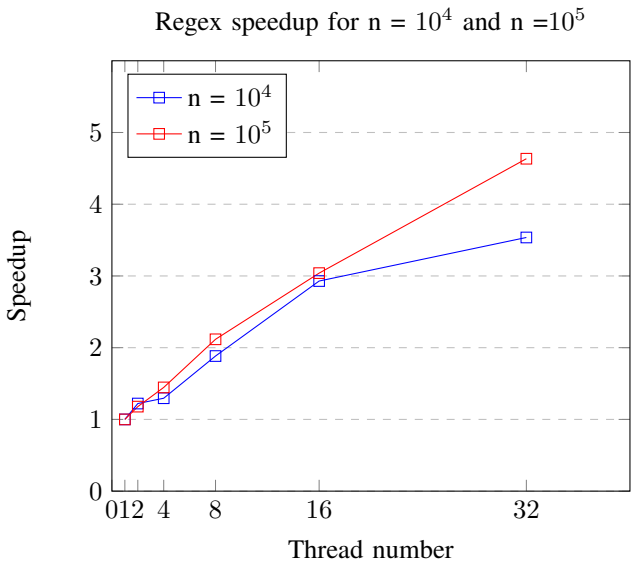
TABLE II: Regex Match

		n			
	µs	10^2	10^3	10^4	10^5
p	1	379.7326	781.5111	9557.708	82840.64
	2	436.557	680.51	7825.88	70303.5
	4	284.543	819.852	7378.97	57293.3
	8	266.451	707.707	5074.49	39134.4
	16	298.603	776.811	3262.99	27254.5
	32	338.036	591.708	2702.7	17875.6









IV. Conclusiones

De los resultados se puede apreciar el impacto que tiene el algoritmo paralelo en el tiempo de ejecución. Una vez que el tamaño del input llega a un nivel donde la paralelización se vuelve deseable, vemos como el tiempo de ejecución baja rápidamente con el incremento de threads para ambos algoritmos. Para los dos primeros tamaños, se espera que no tenga tanto impacto debido a que el overhead de usar tantos threads para un input tan chico no resulta eficiente. El problema secuencial tiene una complejidad de $\mathcal{O}(n)$ y se espera que el algoritmo paralelo tenga complejidad $\mathcal{O}(n/p)$. viendo los resultados podemos concluir que nuestro algoritmo estuvo muy cerca a la complejidad esperada, mostrando un speedup casi lineal con el incremento de procesos. Además, se puede ver como el speedup incrementa con el tamaño del problema ya que el overhead de crear varios threads ocupa un menor tiempo del total. En conclusión, podemos decir que nuestro algoritmo ha logrado reducir el tiempo de ejecución considerablemente, teniendo mayor impacto con inputs más grandes, lo que se espera al paralelizar un algoritmo. Los resultados del tiempo de ejecución y el speedup fueron favorables y se podría comparar con un modelo de memoria distribuida o un modelo híbrido en el futuro.

V. Bibliografía

- Memeti, S., Pllana S. (Diciembre, 2014). PaREM: A Novel Approach for Parallel Regular Expression Matching. *IEEE*. https://arxiv.org/pdf/1412.1741.pdf