Algoritmos Paralelos para Búsqueda de Patrones: Preproceso del patrón

David A. Koufaty A. y Alejandro Teruel L.
Departamento de Computación y
Tecnología de la Información
Universidad Simón Bolívar
Apartado 89000, Caracas 1080–A, Venezuela

Resumen

En este trabajo se presentan algoritmos paralelos para el problema de búsqueda de patrones, donde el texto y el patrón son cadenas de caracteres sobre un alfabeto finito. Estos algoritmos son en su mayoría implantables sobre arquitecturas disponibles con la tecnología actual y fueron programados en un simulador de un hipercubo, el Intel iPSC/2. Los algoritmos desarrollados paralelizan el algoritmo secuencial ingenuo y los algoritmos secuenciales optimales que construyen un índice en base al patrón. Para todos ellos se efectúa el análisis de complejidad, aceleramiento y eficiencia.

Algunos de los algoritmos paralelos son optimales si la longitud del patrón está acotada por una constante, mientras que otros, a pesar de no serlo, pueden presentar un comportamiento bastante bueno en la práctica, ya que hacen intentos por mantener el balance de carga entre los procesadores.

1 Introducción

El problema de búsqueda de patrones se puede plantear de la siguiente forma: dada una cadena de caracteres de longitud n denominada texto, y otra de longitud m denominada patrón, con $m < n$, encontrar las ocurrencias del patrón en el texto. En nuestro caso nos interesa sólo el problema cuando el patrón y el texto son cadenas de caracteres sobre un alfabeto finito Σ y deseamos encontrar todas las ocurrencias del patrón, utilizando para ello arquitecturas paralelas con memoria distribuída.

Existen muchos algoritmos optimales tanto secuenciales como paralelos. En cuanto a los algoritmos secuenciales, el más ingenuo requiere un orden $O(n*m)$ en el peor caso y los optimales

requieren un orden de $O(n+m)$, basando su eficiencia en la construcción de índices auxiliares para la búsqueda, y pueden ser divididos en dos grupos: aquellos que construyen los índices en base al patrón y aquellos que lo hacen en base al texto. La mayoría de los algoritmos paralelos optimales no se basan en los algoritmos secuenciales optimales, sino que resuelven el problema de una manera completamente diferente.

Uno de ellos [MS84] resuelve en forma paralela el problema de hallar la primera ocurrencia del patrón en el texto y se basa en el algoritmo secuencial de Boyer y Moore [BM77]. El algoritmo efectúa una división del texto similar a la presentada en este trabajo, utilizando secciones de tamaño constante sobre una máquina que posee 8 procesadores y memoria compartida. El algoritmo propuesto es incompleto pues no muestra como resolver el problema de una ocurrencia en la frontera de dos secciones. Tampoco se efectúa un análisis del comportamiento teórico del algoritmo y de la magnitud de la pérdida de linealidad a medida que se aumentan los procesadores ni de la conveniencia o no de seleccionar secciones de tamaño fijo.

Otros algoritmos presentados son sólo de interés teórico, pues el modelo computacional utiliza memoria compartida con lecturas concurrentes y, en algunos casos, acceso concurrente para la escritura de una misma constante; ademas, el mínimo número de procesadores es $n/\log^2 n$. Algunos de ellos no son optimales, entre los cuales está el de Karp y Rabin [KR87], mientras que los algoritmos de Galil [Gal84, Gal85], Vishkin [Vis85] y Berkman et al. [BBG+89] si lo son, obteniendo en este útlimo un orden de $O(\log\log m)$ con $n/\log\log m$ procesadores usando el modelo computacional descrito.

En nuestro caso presentaremos algoritmos que pueden ser implementados en las arquitecturas disponibles en la actualidad. El modelo computacional usado será una máquina tipo MIMD (Multiple Instruction Multiple Data) con memoria distribuída, en la cual se puede solapar la comunicación con el cómputo. En algunos caso se presenta el algoritmos para un módelo SIMD (Single Instruction Multiple Data).

Para efectuar el análisis de los algoritmos utilizaremos varias métricas, a saber, el tiempo de ejecución $T(n)$, el aceleramiento (*speedup*) S, el número de procesadores P, el costo $C(n)$ y la eficiencia E del algoritmo.

Para el problema de búsqueda de patrones con un texto de longitud n (variable) y un patrón fijo de longitud m denotaremos con $T_{opt}(n)$ el tiempo tomado por los algoritmos secuenciales optimales y con $T_{par}(n)$ el tiempo tomado por un algoritmo paralelo usando P procesadores. Para la comunicación denotaremos con M_p al número de mensajes enviados por cada procesador más los que recibe del anfitrión y con M_c el número de caracteres transmitidos en estos mensajes.

Los algoritmos presentados aquí han sido programados en su totalidad en el lenguaje C [KR78] en un simulador de un hipercubo de Intel, el iPSC/2 [Int]. La programación y detalles

de los mismos se presenta en [Kou91].

En las secciones 2 y 3 presentaremos las versiones paralelas del algoritmo ingenuo y los algoritmos que preprocesan el patrón, respectivamente. Luego en la sección 4 se efectúa el análisis de aceleramiento y eficiencia de estos algoritmos. Por último se presentan las conclusiones.

2 Algoritmos ingenuos

El algoritmo secuencial más ingenuo para resolver el problema consiste en intentar encontrar el patrón en cada una de las posiciones del texto. Para ello se efectúa una comparación entre el patrón y m caracteres del texto, comenzando en cada una de las posiciones del texto. De esta manera el algoritmo sería:

Para $i = 1$ hasta $n - m + 1$ hacer
  Si pat[1]...pat[m] es igual a text[i]...text[$i + m - 1$]
  entonces reportar una ocurrencia en la posición i

Varios algoritmos paralelos ingenuos se obtienen a partir de éste. Si disponemos de $n-m+1$ procesadores, podemos obtener uno paralelizando el lazo exterior, donde el procesador i es responsable de buscar el patrón a partir de la posición i del texto. En el mejor de los casos si disponemos de memoria compartida con lecturas simultáneas, el algoritmo es de orden $O(m)$ y costo $(n - m + 1) * m$.

Un algoritmo de orden constante se puede obtener paralelizando ambos ciclos, si disponemos de $n * m$ procesadores con una memoria compartida en la cual se pueden efectuar escrituras simultáneas de una misma constante 0. Para ello se requiere de un arreglo de tamaño n con cada entrada inicializada en el valor 1. El procesador (i,j), donde i varía desde 1 hasta n y j de 1 a m, es responsable de comparar text[i] con pat[j] (aunque para ser precisos, i no debe ser mayor que $n - m + 1$). En caso de ser diferentes coloca en 0 la entrada i del arreglo. Las entradas que permanecan en 1, indican las posiciones de ocurrencia del patrón. Como se ve, este algoritmo es de $O(1)$ y tiene un costo igual al caso anterior.

En los dos casos anteriores se trata de paralelizar el acceso al texto. Desde otro punto de vista podríamos tratar de paralelizar el acceso al patrón. Aunque el resultado obtenido obliga a descartar la solución, el enfoque no deja de tener interés. La idea en este algoritmo consiste en tener m procesadores, responsable cada uno de la posición del patrón correspondiente. Cada procesador debe comparar el caracter del patrón asociado a su posición con todos los caracteres del texto y determinar las posiciones en donde coinciden. Luego se debe proceder a determinar si existen m posiciones encontradas por los m procesadores que sean consecutivas (es decir el

primer procesador indica i, el segundo $i+1$, hasta el último que indica $i-m+1$). Para resolver este problema el algoritmo propuesto se basa en un *pipeline* de procesadores, donde existe además un anfitrión (*host*) que tiene comunicación con el primero y el último de los procesadores, de manera que el anfitrión y los procesadores conforman un anillo.

Los caracteres del texto entran al *pipeline* y cada nodo, dependiendo de su estado, compara el caracter del patrón que posee con el caracter del texto que le llega. Luego los caracteres son enviados al próximo procesador. Para determinar las ocurrencias los caracteres viajan junto con una etiqueta, que asegura que al salir del *pipeline* la etiqueta tiene valor m si y sólo si hay una ocurrencia del patrón que finaliza en ese caracter. El estado según el cual los procesadores deciden comparar o no los caracteres que le llegan depende del caracter anterior que le llegó al procesador. El tiempo de ejecución del algoritmo es de orden $O(n+m)$, que viene dado por el tiempo durante el cual se llena el *pipeline* mas el tiempo que tardan en salir por él n elementos. Su costo es $nm+m^2$.

A pesar de que el aceleramiento de los dos primeros algoritmos es bastante bueno, todos los algoritmos son de costo mayor o igual a $O(n*m)$. Los primeros dos algoritmos son optimales respecto al algoritmo secuencial ingenuo, pero no así con respecto a los algoritmos secuenciales optimales que, como indicamos al principio, son de orden $O(n+m)$. Con esto obtenemos que la eficiencia de estos algoritmos es:

$$E = \frac{n+m}{n*m} = \frac{1}{n} + \frac{1}{m}$$

lo que indica que mientras más procesadores se usen (o de mayor longitud sea el patrón o el texto) el algoritmo hace peor uso de los recursos.

3 Algoritmos que preprocesan el patrón

En esta sección se mostrarán algoritmos paralelos que utilizan parcialmente alguno de los algoritmos secuenciales que preprocesan el patrón. El uso de estos algoritmos secuenciales es local a cada procesador y en muchos casos es irrelevante cual de ellos es.

Existen tres algoritmos secuenciales que preprocesan el texto y que dan origen a muchas variantes y combinaciones, estos son el algoritmo de Knuth, Morris y Pratt [KMP77], el de Boyer y Moore [BM77] y el de Karp y Rabin [KR87]. Los tres algoritmos son optimales en tiempo con $T_{opt} = O(n+m)$, aunque los dos últimos lo son en el caso promedio, pues en el peor caso son de orden cuadrático $O(n*m)$ como el ingenuo. Recientemente Sunday [Sun90] presenta una familia de algoritmos que recoge las ideas de los dos primeros y genera varios algoritmos de búsqueda, resultando casos particulares los algoritmos clásicos de Knuth, Morris y Pratt y el de

Boyer y Moore.

En el algoritmo de Knuth, Morris y Pratt el texto es examinado de izquierda a derecha. El algoritmo tiene dos pasos: el primero de ellos consiste en procesar el patrón para computar una tabla que luego ayudará en el segundo paso, que es la búsqueda propiamente dicha. La fase de búsqueda del algoritmo se inicia alineando el patrón con el comienzo del texto. Las comparaciones van de izquierda a derecha. Mientras sean exitosas, se avanza una posición en el texto y en el patrón, hasta llegar al final del patrón cuando encontramos una ocurrencia. Si ocurre una falla, se desplaza el patrón a la derecha un número de posiciones, indicado por la tabla precomputada, que asegure que no se pierde ninguna ocurrencia; esto se hace sin desplazar el apuntador al texto. La cantidad por la cual se desplaza el patrón en el caso de una falla corresponde a la máxima ocurrencia del patrón en sí mismo, cantidad que garantiza que se puede continuar la búsqueda desde esa posición sin comparar los caracteres anteriores con el texto.

El algoritmo para búsqueda de patrones que más versiones ha generado es quizás el de Boyer y Moore. A diferencia del algoritmo de Knuth et al, este algoritmo al alinear el patrón en alguna posición del texto examina sus caracteres de derecha a izquierda, comenzando en el último caracter del patrón. A medida que los caracteres alineados coinciden se examinan los caracteres a la izquierda, hasta llegar al principio del patrón cuando se ha encontrado una ocurrencia. Si existe una diferencia se desplaza el patrón a la derecha de acuerdo a una cantidad precomputada y las comparaciones se reinician al final del patrón. La cantidad por la cual se desplaza el patrón viene dado por la cantidad máxima entre dos heurísticas: la de ocurrencia y la de coincidencia. La primera indica para cada caracter del alfabeto cual es su posición más a la izquierda en el patrón. La segunda es similar a la heurística del algoritmo de Knuth et al., pero considerando que los caracteres son examinados de derecha a izquierda.

El algoritmo de Karp y Rabin es un algoritmo probabilístico, que usa la idea de clasificación (*hashing*) para la búsqueda de patrones. El algoritmo básicamente computa una función (firma) para el patrón y lo compara con el valor de la función en cada cadena de m caracteres del texto. Inicialmente se computa la firma del patrón y de los primeros m caracteres del texto. Dada la firma de una posición $i-1$, el cálculo de la firma de la posición i se puede hacer en orden constante. Sin embargo si dos firmas son iguales (hay una colisión) no se garantiza que haya una ocurrencia, por lo cual es necesario una comparación adicional con el texto.

La idea principal de los algoritmos paralelos de esta sección consiste en paralelizar el segundo paso de los algoritmos secuenciales optimales. Estos algoritmos consisten todos de un preproceso y luego una búsqueda. En base a esto, el problema de hallar las coincidencias entre un texto de longitud n y un patrón de longitud m, se puede dividir en varios subproblemas, cada uno con

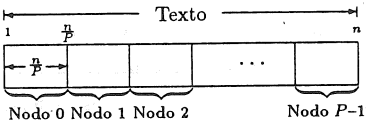

Figura 1: Partición del problema

el mismo patrón pero con una sección de texto de menor tamaño y, como veremos adelante, un trabajo posterior adicional.

El algoritmo genérico propuesto en base a esta observación es:

1. El anfitrión asigna a cada procesador secciones de texto.
2. Cada procesador aplica localmente uno de los algoritmos optimales de de búsqueda secuencial en cada sección asignada.
3. Cada ocurrencia es reportada al anfitrión.

Los procesadores son responsables de encontrar las ocurrencias dentro de las secciones de texto a él asignadas. Un ejemplo de tal asignación se muestra en la figura 1, donde a cada procesador se le asigna una sección de n/P caracteres. Es evidente que el número de procesadores P debe ser a lo sumo n, caso en el cual a cada procesador le corresponde un caracter del texto.

En este algoritmo surgen dos problemas básicos:

1. Cómo se efectúa la asignación de texto a los procesadores, y
2. Cómo se detectan las ocurrencias del patrón que estén en el límite entre dos secciones consecutivas asignadas a dos procesadores distintos.

La asignación de texto a los procesadores puede ser prefijada (estática) o dinámica. La diferencia entre ellas está íntimamente ligada al problema de balanceo de carga: cómo evitar que un procesador no tenga asignado un trabajo mientras los demás sí.

El segundo problema puede ocurrir cuando los últimos k caracteres de una sección coinciden con los primeros k caracteres del patrón y los primeros $m-k$ caracteres de la siguiente sección coinciden con los últimos $m-k$ caracteres del patrón. De esta forma la coincidencia no detectada por ninguno de los dos procesadores asignados a estas dos secciones, tal como lo muestra la figura 2. Es evidente que las únicas ocurrencias que un procesador no puede detectar son aquellas que comienzan en una posición mayor de $n/P - m$ relativa al comienzo de la sección.

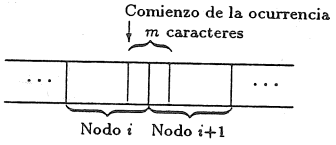

Figura 2: Ocurrencia no detectada en el algoritmo genérico

3.1 Asignación de secciones

Si asignamos estáticamente las secciones al momento de comenzar la ejecución el tamaño total de las secciones que le corresponderán a cada procesador. Si disponemos de P procesadores el número total de caracteres que deberá analizar cada uno es de n/P. En este caso lo más conveniente es que todos los caracteres asignados a un procesador sean consecutivos, generando una única sección de tamaño n/P para cada uno, evitando la proliferación de ocurrencias en la frontera. Si denotamos con q el tiempo necesario para resolver el problema de las ocurrencias en la frontera de una sección, el orden de algoritmo un algoritmo con este esquema es:

$$T_{par}(n) = T_{opt}\left(\frac{n}{P}\right) + q = O\left(\frac{n}{P} + q + m\right)$$

La asignación estática sufre de un problema básico en todo algoritmo paralelo: balanceo de carga. El trabajo de cada procesador es fijado a priori, por lo cual es posible que un procesador quede ocioso mientras otros trabajan, ya que, por ejemplo, en el algoritmo de Knuth et al., la constante del número de comparaciones es 2 en el peor caso, pero en el mejor caso es exactamente 1. Esto indica que puede ocurrir que algunos procesadores terminen su trabajo en la mitad del tiempo que otros.

Para tratar de aminorar el efecto que tiene la diferencia en la dificultad de cada sección, la siguiente alternativa es no asignar las secciones a priori, sino entregar a cada procesador secciones de tamaño constante s a medida que finalice el análisis de la sección que posee. s debe ser menor que n/P para que el algoritmo mejore con respecto al anterior. El número total de secciones de tamaño s a enviar es n/s. Supondremos en las siguientes subsecciones que el número de secciones de texto asignadas a cada procesador es en promedio $n/(sP)$. Observese que en este caso el orden de la búsqueda en cada sección es $O(s)$, a lo cual hay que agregar el proproceso del patrón una sola vez. En este caso el orden de un algoritmo es:

$$T_{par}(n) = (T_{opt}(s) + q)\frac{n}{sP} + m = (O(s) + q)\frac{n}{sP} + m = O\left(\frac{n}{P} + q\frac{n}{sP} + m\right)$$

Este esquema de asignación aventaja al estático:

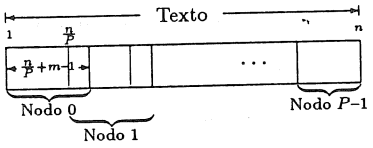

Figura 3: Solapamiento de secciones

1. El balanceo de carga es más justo.
2. El espacio necesario en cada procesador es $O(s+m)$, el texto y el patrón, mientras que en los algoritmos anteriores es $O(n/P+m)$. Si s es mucho menor que n/P, el ahorro es significativo.

3.2 Ocurrencias en la frontera

La primera alternativa consiste en asignar los caracteres en la frontera por duplicado, solapando las secciones asignadas. Bastaría con que el solapamiento sea de $m-1$ caracteres, ya que de ser menor aún habría ocurrencias no detectadas y de ser mayor se reportarían dos veces. Así, a cada sección de texto asignada se le deben agregar los $m-1$ caracteres correspondientes al solapamiento. Por ejemplo si utilizamos un esquema de asignación estática combinado con el solapamiento de las secciones obtenemos la asignación presentada en la figura 3. El orden adicional de detectar las ocurrencias en la frontera es $O(m)$, ya que el texto es ahora $m-1$ caracteres mayor.

Estudiando detenidamente el algoritmo de Knuth, Morris y Pratt, se pueden hacer las siguientes afirmaciones:

1. El algoritmo puede examinar más de una vez cada caracter del texto (comparándolo con el patrón).
2. Una vez que avanza al próximo caracter del texto no retrocede. Esto indica que el apuntador al texto siempre se incrementa.
3. Cuando el algoritmo llega al final del texto, se tiene un índice sobre el patrón que indica la máxima coincidencia entre el texto y el patrón al final del texto. Esto corresponde con la semántica del índice del patrón, ya que mantiene en todo momento la máxima coincidencia hasta la posición actual en el texto.

Si anexamos otros caracteres al final del texto la búsqueda se podría reiniciar con sólo saber el valor final del índice j del patrón con el cual concluyó la búsqueda sobre el texto inicial. La búsqueda a partir de la posición $j+1$ del patrón y el primero de los nuevos caracteres.

La otra alternativa consiste en explotar esa particularidad para evitar las secciones solapadas. Cada procesador recibe su sección de texto de acuerdo al esquema de asignación utilizado. y aplica el algoritmo de Knuth et al. Al finalizar cada procesador sabe cual fue la coincidencia máxima entre el patrón y la parte final de su sección y la envía al procesador que posee la sección siguiente para que continúe haciendo la búsqueda. Este último paso consiste en aplicar el mismo algoritmo pero partiendo con un valor inicial para el índice del patrón, siguiendo el algoritmo hasta la posición $m-1$ del texto, pues de ahí en adelante ya el procesador efectuó la búsqueda (en realidad basta con buscar hasta que el índice del texto supere al del patrón, pues en ese caso la ocurrencia finalizaría al menos en la posición m y éstas ya han sido reportadas). El orden adicional para las ocurrencias en la fronteras es $O(m)$, ya que se aplica el algoritmo de Knuth et al. con un texto y un patrón de longitud m.

A pesar de no mejorar el orden de la búsqueda de las ocurrencias en la frontera, este esquema presenta varias ventajas con respecto al de solapamiento de secciones:

1. En el caso de modelos con memoria distribuída el algoritmo es mejor, pues se evita la transmisión adicional de $(m-1)(P-1)$ caracteres y tener por duplicado algunas secciones del texto.
2. De trabajar en un modelo con memoria compartida no se necesitaría el acceso concurrente a ninguna sección del texto.

Como se desprende del esquema, la topología ideal para la misma es una topología lineal, pues la comunicación sólo es necesaria entre procesadores que tienen secciones consecutivas.

Para garantizar que cada procesador pueda finalizar la búsqueda iniciada por el anterior es conveniente que la sección asignada sea de al menos de tamaño $m-1$, ya que de lo contrario sería necesario la coloboración de más de un procesador para detectar las ocurrencias compartidas, lo cual aumenta la complejidad comunicacional. Por ello es recomenedable que $m \leq n/P$, lo cual nos da una indicación sobre el número de procesadores del algoritmo:

$$P \leq n/m$$

Como se ve, la diferencia en tiempo entre los algoritmos esta en determinación del esquema de asignación de secciones, mientras que la diferencia en comunicación depende del de detección de ocurrencias en la frontera. Combinando los equemas de asignación estática y dinámica con los de solapamiento y colaboración se obtienen varios algoritmos, cuyo comportamiento teórico se resume en las tablas 1 y 2. En el caso del número de caracteres no se incluyen los $n/P+m$ caracteres indispensables que debe recibir cada procesador.

	Asignación estática	Asignación dinámica
Tiempo	$\frac{n}{P} + m$	$\frac{n}{P} + m\frac{n}{sP}$
Eficiencia	$\frac{n+m}{n+mP}$	$\frac{n+m}{n+mn/s}$

Tabla 1: Tiempo y Eficiencia

	Asignación estática		Asignación dinámica	
	Solapadas	Colaboración	Solapadas	Colaboración
Mensajes M_p	1	2	$2\frac{n}{sP} - 1$	$3\frac{n}{sP}$
Caracteres M_c	$m - 1$	1	$m\frac{n}{sP} - 1$	$4\frac{n}{sP} - 1$

Tabla 2: Comunicación

3.3 Descentralización de la administración

La idea de este algoritmo es descentralizar la administración del texto, manteniendo en lo posible el balance de carga entre los procesadores. Uno de los problemas que surge en el algoritmo general propuesto antes, consiste en el posible cuello de botella en el que se puede convertir el anfitrión, quien en ese caso se desenvuelve como administrador del texto. Una solución al problema consiste en combinar la asignación estática y la dinámica. El algoritmo descrito a continuación es sobre una topología de hipercubo y tiene sentido cuando el número de procesadores disponibles es al menos 8 (cubo de dimensión 3), ya que las ideas expuestas requieren que cada nodo posea al menos 3 vecinos.

La idea principal es asignar una sección de tamaño $k * n/P$ caracteres a un grupo de k procesadores, al cual denominaremos macro-procesador, y que entre todos hallen las ocurrencias en esa sección. El caso descrito a continuación consta de 4 procesadores por macro-procesador.

Este grupo debe tener la particularidad que uno de los procesadores es vecino del resto de los elementos del grupo. Al procesador central del macro-procesador lo denominaremos M-nodo. Este será el administrador de la sección de texto asignada al macro-procesador, convirtiéndose así en un anfitrión local para los otros procesadores de macro-procesador.

Cada uno de estos M-nodos recibe los caracteres correspondientes a todo el macro-procesador, es decir, $4(n/P)$ caracteres. Las acciones que toma este nodo son las mismas que tomaba el anfitrión en el caso del algoritmo de asignación dinámica, excepto que sólo sobre los nodos del macro-procesador. Además de administrar, cada M-nodo también efectúa las funciones de

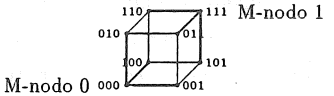

Figura 4: Cobertura de un cubo de dimensión 3 por macro-procesadores

búsqueda, ya que con tres procesadores dependientes es poco el trabajo de administración.

Para efectuar la implementación del algoritmo en el iPSC/2 se requiere primero encajar la topología del algoritmo en el hipercubo. Se escogen como M-nodos todos aquellos procesadores cuya representación binaria termine con los dígitos 000 ó 111. Los tres vecinos de cada M-nodo en el macro-procesador se obtienen invirtiendo cada uno de los tres últimos bits de la representación binaria de su número en el hipercubo. Para el caso de un M-nodo de la forma X...X000 sus vecinos son el X...X001, el X...X010 y el X...X100. De esta forma queda cubierto un cubo de dimensión 3 por cada dos macro-procesadores, como lo muestra la figura 4.

En el caso de los caracteres en la frontera entre dos M-nodos, se puede seguir cualquiera de las políticas descritas antes: secciones solapadas o colaboración entre procesadores. Si deseamos usar el segundo método, debemos hacer algunas consideraciones sobre la asignación de segmentos a los M-nodos, de manera de minimizar la distancia entre ellos. Es posible lograr que todas las secciones consecutivas estén en procesadores a distancia 1, excepto dos que está a distancia 3.

La idea para minimizar la distancia entre los M-nodos consiste en encajar dos anillos, un anillo en cada uno de los dos sub-hipercubos de dimensión $d-3$ que forman los macro-procesadores que terminan en 000 y 111, y luego establecer la conexión entre ellos. Para encajar los anillos en cada uno de los sub-hipercubos usamos el código de Gray y escribimos una lista con los procesadores adayacentes consecutivamente. Las secciones de texto son asignadas a los M-nodos de la primera lista desde el primero al último y luego a los de la segunda lista desde el último hasta el primero. De esta manera todos los M-nodos con secciones consecutivas están a distancia 1, excepto los últimos de cada lista que están a distancia 3, ya que los últimos procesadores de ambas listas son iguales a excepción de los tres últimos bits, que son 000 para uno y 111 para el otro.

Para el caso de los caracteres en la frontera de las secciones dentro de un macro-procesador, dado que los procesadores están muy cercanos entre sí, resulta muy conveniente la utilización del algoritmo de colaboración entre procesadores y no el de solapamiento.

La complejidad computacional de este algoritmo permanece invariable respecto a los algoritmos de secciones dinámicas, ya que se efectúan exactamente el mismo algoritmo de distribución de secciones de tamaño s, excepto que localmente.

La complejidad comunicacional en cuanto al número total de caracteres enviados M_c, aumenta con repecto a los algoritmos de secciones dinámicas, ya que, como se observa en el algoritmo, el texto se retransmite en la red; una vez lo envía el anfitrión a los M-nodos y luego los M-nodos lo retransmiten al resto de los procesadores. Por esto se crea un factor adicional de n en el total de caracteres enviados. El número de comunicaciones M_p efectuadas por cada procesador se mantiene como en el algoritmo de secciones dinámicas en los procesadores no M-nodos y aumenta en los M-nodos, ya que deben efectuar el trabajo de distribución al resto. Sin embargo el número total de transmisiones en la red disminuye con respecto al de secciones dinámicas, ya que, las transmisiones de distribución están contadas dos veces (una en los M-nodos y otras en los no M-nodos); si descontamos de las transmisiones de los M-nodos las necesarias para la distribución de las secciones a los procesadores no M-nodos, los M-nodos sólo efectúan una comunicación con el anfitrión. La diferencia está en que el M-nodo recibe todas sus secciones en una sola comunicación (aunque en un principio no sabe cuales son).

Es importante destacar que todas las transmisiones en el algoritmo, a diferencia de los anteriores, se pueden realizar en paralelo y por canales de comunicación distintos, ya que se aseguró por la topología usada que la distancia entre los procesadores que cooperan es mínima y lo hacen por canales distintos.

3.4 Balanceo automático de carga

En este algoritmo la idea básica es hacer una asignación inicial de texto estática, pero que a medida que los procesadores terminen, le ofrezcan ayuda a los procesadores que aún están ocupados. Los procesadores ocupados deben decidir si aceptan la ayuda o no. La ayuda es rechazada si el tiempo que le tomaría al nodo procesar la fracción que le quede es inferior al tiempo que se tardarán los caracteres enviados en llegar al otro procesador mas el tiempo en que éstos serán procesados. Si denotamos con k el número de caracteres por procesar y con j el número de caracteres a enviar, el óptimo de tal relación se obtiene con:

$$T_{opt}(k-j) = T_{transmisión}(j) + T_{opt}(j)$$

Este tiempo de transmisión lo podemos expresar como $t_0 + t * j$, donde t_0 es el tiempo inicial necesario para establecer la comunicación (*setup*) y t es el tiempo requerido para la transmisión de cada caracter (ancho de banda del canal). Además podemos expresar el tiempo optimal de procesar i caracteres como $t_p * i$, donde t_p es el tiempo promedio de los algoritmos secuenciales para examinar cada caracter. Entonces se requiere que:

$$(k-j) * t_p = t_0 + j * t + j * t_p k = \frac{t_0}{t_p} + \left(\frac{t}{t_p} + 2\right) * j \qquad (1)$$

Como era de esperar el resultado es muy dependiente de la velocidad relativa de transmisión con repecto al cómputo. Por otro lado, sabemos que j debe ser mayor o igual a m, pues por las mismas razones expuestas antes es conveniente que ninguna sección sea menor de m caracteres, con lo cual el mínimo número de caracteres con lo cual sería compartida la carga es:

$$k \geq \frac{t_0}{t_p} + \left(\frac{t}{t_p} + 2\right) * m \tag{2}$$

En base a esto se compartiría la carga si k es al menos el valor indicado en la ecuación 2 y los caracteres a enviar serían, en el caso óptimo, los indicados por el valor de j en 1. Observemos que si, por ejemplo, el valor de t_0 es 1000 veces mayor que t_p, el valor de t es 50 veces peor que el de t_p y estamos buscando un patrón de longitud 30, el mínimo número de carateres con los cuales se comparte la carga es 2560, de lo cual parece razonable esperar que el algoritmo sólo tiene utilidad con textos muy grandes.

Una consideración necesaria sobre el algoritmo es cómo se ofrece o rechaza la ayuda entre los procesadores. Una posibilidad sería que los procesadores libres efectuaran una difusión de su disponibilidad y los procesadores ocupados que así lo deseen soliciten la ayuda; para el caso que más de un procesador solicite la ayuda, se debe esperar la confirmación o rechazo del procesador que la ofrece. Otra posibilidad que involucra menos comunicación en la red, consiste en mantener en el anfitrión el estado de trabajo de cada procesador. Cuando un procesador queda libre envía la notificación al anfitrión y éste le indica algún procesador ocupado. Si el procesador indicado por el anfitrión no acepta la ayuda se marca como desocupado y se repite el proceso hasta que todos los procesadores queden desocupados.

Como se ve la implementación eficiente de este algoritmo requiere conocer el tiempo de comunicación promedio entre los procesadores, dato que no se obtuvo, por lo cual este algoritmo no fue implementado.

La complejidad computacional de este algoritmo es difícil de determinar de manera exacta, ya que depende de las características del patrón y el texto. Evidentemente resulta inferior a los de secciones dinámicas. Si los procesadores ocupados deciden no distribuir la carga entre los libres, la complejidad se mantiene igual a la de los algoritmos de secciones estáticas; pero en el caso de haber distribución de carga, se espera que sea ligeramente superior. Lo mismo ocurre con la complejidad comunicacional. Por ello no podemos dar una fórmula exacta para la complejidad promedio de este algoritmo, pero será un valor intermedio entre las complejidades de los algoritmos de secciones estáticas y los de secciones dinámicas, mas cercano a los primeros que a los segundos.

4 Análisis de los algoritmos

4.1 Aceleramiento y eficiencia

Como hemos calculado previamente tenemos básicamente dos ordenes distintos en los algoritmos, los presentados en la tabla 1.

4.1.1 Secciones estáticas

Para el caso de los algoritmos de secciones estáticas el aceleramiento es:

$$S = \frac{n+m}{n/P+m}$$

Nos interesa observar el comportamiento de esta función. Para ello veremos tres casos de patrones: patrones de longitud 1, patrones de longitud n/P y patrones de longitud n.

1. Para el caso de patrones muy pequeños tomaremos como caso extremo $m = 1$. Desarrollando para el caso de que el número de procesadores está entre 2 y n, obtenemos que el aceleramiento está entre $S = 2(n+1)/(n+2) \cong 2 = P$ y $S = (n+1)/2 = (P+1)/2 \cong P/2$. Esto nos indica que la eficiencia para el caso de patrones pequeños está entre $1/2$ y 1.

2. En el otro caso extremo tenemos que $m = n$, con lo cual $S = \frac{n+n}{n/P+n} = \frac{2P}{P+1} \cong 2$ En este caso el aceleramiento es siempre cercano a 2, por lo cual el algoritmo está muy lejos de hacer un buen uso de los recursos que posee y a medida que se aumentan los procesadores la eficiencia disminuye.

3. En el caso medio tomamos $m = n/P$ (o equivalentemente $P = n/m$). Obtenemos $S = \frac{n+n/P}{2n/P} = \frac{P+1}{2}$. Esto indica que el algoritmo obtiene una eficiencia superior a $1/2$ si logramos mantener el número de procesadores inferior al número de secciones de tamaño m que podemos formar.

En conclusión podemos afirmar que los algoritmos de secciones estáticas presentados tienen una eficiencia teórica superior a $1/2$ si el número de procesadores usados en el algoritmo es $P \leq n/m$. El algoritmo se comporta ineficientemente en cualquier otro caso, llegando a obtener en el peor caso un aceleramiento de apenas 2, aún con un gran número de procesadores, por lo cual es recomendable disminuir el número de procesadores a medida que la longitud del patrón se acerca a la del texto.

4.1.2 Secciones dinámicas

El aceleramiento obtenido para los algoritmos de secciones dinámicas es:

$$S = \frac{n+m}{n+mn/s}P$$

Presentaremos el análisis en base al tamaño de la sección s.

1. En el caso de secciones de tamaño reducido, $s = 1$, el aceleramiento es:

$$S = \frac{(n+m)P}{(m+1)n} < \frac{(n+m+1)P}{(m+1)n} = \left(\frac{1}{m+1} + \frac{1}{n}\right)P$$

   lo cual es bastante deficiente, ya que el aceleramiento y la eficiencia resultan inversamente proporcional a las longitudes del texto y el patrón.

2. En el caso de tomar secciones de m caracteres obtenemos que el aceleramiento es $S = \frac{n+m}{n+n}P = \frac{n+m}{2n}P$. Sabemos que el patrón tiene longitud entre 1 y n, con lo cual la eficiencia esta entre $1/2$ y 1.

3. En el caso de secciones de n/P caracteres, obtenemos el algoritmo de secciones estáticas.

Resulta evidente que el comportamiento de los algoritmos mejora sensiblemente con la selección de secciones de texto relativamente grandes, siendo recomendable tomar un mínimo de m caracteres por sección. Este es un resultado esperado ya disminuyendo el tamaño de las secciones, el número de secciones de texto resulta mayor y con ello la cantidad de caracteres en las fronteras, los cuales son revisados dos veces por el algoritmo.

Algo que se deduce de ambos esquemas de asignación de secciones es que los algoritmos están muy cerca de los optimales si los patrones son pequeños relativos al texto. En los casos prácticos de búsqueda son frecuentes los patrones de pocos caracteres (a lo sumo algunas decenas de ellos) y texto de millones de caracteres.

4.2 Comportamiento límite

La ley de Amdahl coloca un límite al aceleramiento que puede ser obtenido en los algoritmos paralelos. Si definimos α, denominado fracción de Amdahl, como la fracción del algoritmo que debe correr secuencialmente (no es paralelizable), tenemos que:

$$T_{par}(n) = \alpha T_{opt}(n) + \frac{(1-\alpha)T_{opt}(n)}{P}$$

y

$$S = \frac{P}{1+(P-1)\alpha}$$

Por lo tanto hay un límite en el aceleramiento alcanzado dado por:

$$\lim_{P\to\infty} S = \frac{1}{\alpha}$$

Realmente α no es una constante, sino que depende del tamaño del problema. A continuación efectuaremos el estudio del comportamiento de los algoritmos cuando tenemos patrones muy grandes.

4.2.1 Secciones estáticas

En el caso de los algoritmos de secciones estáticas, la fracción de Amdahl corresponde a dos factores. Si suponemos que el texto está inicialmente distribuído uniformemente entre los procesadores, a excepción de la diferencia de 1 caracter que se puede dar si n no es divisible por P, la contribución del algoritmo a la fracción de Amdahl está determinado por el tiempo de distribución del patrón y el trabajo adicional de las ocurrencias en la frontera (que en cualquiera de las dos soluciones presentadas es $O(m)$). En el hipercubo la distancia que tienen que viajar los mensajes está acotada por $\log_2 P$, por lo que el tiempo de transmisión del patrón es $O(m \log_2 P)$. En este caso la fracción quedaría determinada por

$$\alpha = \frac{O(m\log_2 P) + O(m)}{T_{par}(n)} = \frac{O(m\log_2 P) + O(m)}{O(n/P + m)} = O\left(\frac{m(\log_2 P + 1)}{n/P + m}\right)$$

Si consideramos que m está acotada por una constante, lo cual en la práctica es cierto, y que disponemos de un número finito de procesadores tenemos que

$$\lim_{n\to\infty} \alpha = 0$$

por lo cual se puede concluir que el algoritmo obtiene un aceleramiento cercano al lineal, $S = P$, para texto de longitud muy grande si la memoria de los procesadores (para un número fijo de procesadores) es suficientemente grande para almacenar la fracción de texto que le corresponde.

Una conclusión similar se obtiene analizando el comportamiento de la función del aceleramiento derivada antes, ya que con las mismas consideraciones anteriores:

$$\lim_{n\to\infty} S = \lim_{n\to\infty} \frac{n+m}{n/P+m} = P$$

4.2.2 Secciones dinámicas

En los algoritmos de secciones dinámicas se tiene que:

$$S = \frac{(n+m)sP}{(s+m)n}$$

En este caso el comportamiento límite viene dado por:

$$\lim_{n\to\infty} S = \lim_{n\to\infty} \frac{(n+m)sP}{(s+m)n} = \frac{s}{s+m}P$$

El valor $s/(s+m)$ es siempre menor que 1 y se acerca a 1 a medida que la sección de texto s es mayor, con lo cual el aceleramiento es menor que P y la eficiencia menor que 1. Para mejorar la eficiencia las secciones de texto deben ser relativamente grandes. Cuando la sección es n/P se obtiene el mejor aceleramiento, por lo cual el algoritmo de secciones estáticas es mejor en el caso límite.

4.3 Complejidad comunicacional

Como la comunicación entre procesadores y con el anfitrión es de orden lineal en el tamaño del problema, la complejidad computacional del algoritmo es al menos lineal (caso óptimo). Sin embargo existen diferencias en cuanto al número de comunicaciones necesarias en cada algoritmo que alteran el comportamiento real de los mismos. Todos los algoritmos requieren como mínimo la transmisión del texto y el patrón; como se requiere que todos los procesadores posean el patrón, el mínimo número de caracteres transmitidos a cada procesador es $n/P + m$ con una transmisión.

Observando la tabla 2, podemos afirmar que los algoritmos de solapamiento de secciones resultan más económicos en el número de transmisiones efectuadas, ya que al haber colaboración, el número de transmisiones en cada procesador aumenta por la cantidad de secciones que recibe el procesador, lo cual es 1 en el caso de secciones estáticas y $n/(sP)$ para el de secciones dinámicas. Sin embargo resultan más costosos en cuanto a número total de caracteres enviados, ya que el número de caracteres enviados, adicionales al texto y el patrón, en los algoritmos de secciones solapadas es cerca de m veces el número de secciones asignadas al procesador, mientras que en los de colaboración es una constante por el número de secciones asignadas. Esto es sencillo de explicar, lo primero se debe a que en los algoritmos con solapamiento de secciones los procesadores no se comunican entre sí para hallar la solución y lo segundo a que en los algoritmos de colaboración no se duplica ninguna sección del texto.

Sin embargo los algoritmos de secciones estáticas son en todos los aspectos de comunicación mejores que los de secciones dinámicas. Es de esperar que el comportamiento del algoritmo de descentralización sea similar al de secciones dinámicas.

Un hecho interesante se obtiene si tabla anterior denotaremos con N_p el número de secciones que recibe cada procesador en los algoritmos (1 para el de secciones estáticas y $n/(sP)$ para el de secciones dinámicas). La misma tabla, pero expresada de esta forma nos queda:

	Secciones estáticas		Secciones dinámicas	
	Solapadas	Colaboración	Solapadas	Colaboración
Mensajes M_p	N_p	$2N_p$	$2N_p - 1$	$3N_p$
Caracteres M_c	$(m-1)N_p$	N_p	$mN_p - 1$	$4N_p - 1$

En la tabla se observa que en todos los algoritmos el número de comunicaciones es directamente proporcional al número de secciones que recibe cada procesador. Además en los algoritmos de secciones solapadas, se depende adicionalmente del tamaño del patrón.

5 Conclusiones

Algunos de los algoritmos presentados en este trabajo no son implantables con la tecnología disponible. Tal es el caso de los dos primeros algoritmos ingenuos de las sección 2, los cuales usan $O(n)$ y $O(n * m)$ procesadores respectivamente y necesitan memoria compartida para lectura. El tercer algoritmo ingenuo obtiene resultados muy pobres que obligan a descartarlo. El resto de los algoritmos presentan ventajas y desventajas relativas.

Si denotamos con N_s el número de secciones en las cuales se divide el texto en cada algoritmo (P o n/s) y con N_p el número de secciones que recibe cada procesador (1 o $n/(sP)$) podemos escribir el tiempo y la eficiencia de la tabla 1 como:

$$T_{par}(n) = \frac{n}{P} + m * N_p$$

$$E = \frac{n+m}{n+m*N_s}$$

En base a esto, es evidente que la complejidad de los algoritmos aumenta proporcionalmente al número de secciones asignadas a cada procesador. Por ello el algoritmo de secciones estáticas resulta mejor que el de secciones dinámicas, teóricamente. Sin embargo, el algoritmo de secciones dinámicas puede obtener mejores resultados en la práctica para algunos casos, ya que en problemas que presenten mal comportamiento con respecto al balanceo de carga, utilizando secciones suficientemente grandes (por ejemplo, $n/(2P)$) es posible superar a los algoritmos de secciones estáticas. Lo mismo ocurre con los algoritmos de descentralización y de balanceo automático de carga.

La selección entre los algoritmos de secciones solapadas y los de colaboración entre vecinos dependerá fundamentalmente del tamaño del patrón, ya que con patrones muy cortos el solapamiento genera duplicación de pocos caracteres, mientras que con patrones largos los caracteres a transmitir pueden ser demasiados. Si ademas el texto puede ser modificado y queremos evitar la posible inconsistencia entre los caracteres distribuídos resulta mejor utilizar los algoritmos de

colaboración y evitar la duplicación de partes del texto. Es necesario considerar además el costo real de las comunicaciones adicionales a causa de la colaboración y determinar si estas resultan mas costosas que el tiempo de transmisión de los caracteres solapados.

En algoritmo de descentralización puede ser útil en el caso de que el texto esté distribuído en varios discos fijos, de manera que pueda ser accesado concurrentemente por un grupo de procesadores (tal es el caso del hipercubo). En este caso es necesario transmitir al resto de los procesadores las secciones de texto que analizarán y resulta conveniente el esquema de asignación de secciones.

El rebalanceo automático puede ser superior a todos los algoritmos anteriores, pero requiere un esfuerzo de entonación sobre las constantes que representan el tiempo promedio de analizar un caracter y el tiempo promedio real de transmisión del mismo.

El Intel iPSC/2, arquitectura usada para la prueba de los algoritmos, presentó la ventaja de la gran independencia existente entre la arquitectura y los algoritmos: los mensajes son enviados y recibidos en diversas modalidades que permiten un manejo transparente de la comunicación e independiente de la arquitectura. Por otra parte al perder el control del enrutamiento de mensajes no existe forma de indicar por cuál canal debe salir un mensaje. Para asegurar un manejo eficiente de la topología de hipercubo es necesario diseñar algoritmos muy particulares (tal es el caso del algoritmo de descentralización) que aseguren un buen uso de la misma; con lo cual los algoritmos dejan de ser "portables" de una máquina a otra, ya que la topología usada en los mismos en muy rígida. Lamentablemente no se contó con otros simuladores u otras máquinas con las cuales efectuar una comparación de sus ventajas y desventajas durante la programación.

Además de los algoritmos presentados en [Kou91] se presentan algoritmos paralelos basados en la versión secuencial que construye los índices en base al texto.

Referencias

[BBG+89] Omer Berkman, Dany Breslayer, Zvi Galil, Baruch Schieber, y Uzi Vishkin. Highly parallelizable problems. In *Proc. of the* 21^{th} *ACM Symposium on Theory of Computing*, páginas 309–319, 1989.

[BM77] Robert S. Boyer y J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[Gal84] Zvi Galil. Optimal parallel algorithms for string matching. In *Proc. of the* 16^{th} *ACM Symposium on Theory of Computing*, páginas 240–248, 1984.

[Gal85] Zvi Galil. Optimal parallel algorithms for string matching. *Information and Control*, 67:144–157, 1985.

[Int] Intel Corp. *iPSC/2 Simulator Manual.*

[KMP77] Donald E. Knuth, James H. Morris, y Vaugham R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[Kou91] David A. Koufaty A. Algoritmos paralelos para búsqueda de patrones. Tesis de Maestría, Universidad Simón Bolívar, Caracas, 1991.

[KR78] Brian W. Kernighan y Dennis M. Ritchie. *The C Programming Language.* Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[KR87] Richard M. Karp y Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–261, 1987.

[MS84] Peter Moller-Nielsen y Jorgen Staunstrup. Experiments with a fast string searching algorithm. *Information Processing Letters*, 18:129–135, 1984.

[Sun90] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[Vis85] Uzi Vishkin. Optimal parallel pattern matching in strings. *Information and Control*, 67:91–113, 1985.