Universidad Austral de Chile

Facultad de Ciencias de la Ingeniería
Escuela de Ingeniería Civil en Informática

PARALLEL PATTERN MATCHING IN COMPRESSED SPACE

Proyecto para optar al título de
Ingeniero Civil en Informática

PROFESOR PATROCINANTE:
HÉCTOR FERRADA
DOCTOR EN CIENCIAS DE LA COMPUTACIÓN

VICENTE BENJAMÍN LETELIER LAZO

VALDIVIA – CHILE
2022

Parallel Pattern Matching in Compressed Space*

Benjamín Letelier

Facultad de Ciencias de la Ingeniería, Universidad Austral de Chile
vicente.letelier.lazo@alumnos.uach.cl

Abstract. Pattern Matching is one of the main problems in the area of text analysis.There is a wide variety of solutions, the most widely used being to solve it with data structures. This work proposes a CPU-parallel algorithm that allows solving Pattern Matching using the Hybrid Index, a set of data structures that stores the LZ77 factorization of the input text and that, based on different heuristics, manages to work in a totally compressed space, without the need to consult the original text. This parallelization seeks to improve the execution times of Pattern Matching in conventional and repetitive texts with respect to the original algorithm. Experimental results indicate that, depending on the characteristics of the text, the length of the pattern to be searched and the number of threads to be used, execution times between $\sim 1.25\times$ and $\sim 4.8\times$ better than the original version can be obtained.

Keywords: Pattern Matching · Parallel Algorithm · Compression.

1 Introducción

Sea $S[1...n]$, un *string*[1] de longitud n denominado texto, y $P[1...m]$, un *string* de longitud m denominado patrón, ambos sobre un mismo alfabeto[2] de tamaño σ, con $m \leq n$; se puede definir como *Pattern Matching* (PM) al problema que consiste en reportar la ubicación de todas las ocurrencias de P en S [HHLS04]. Se denominará como una ocurrencia de P en S, a un índice i de S, tal que $S[i...i+m\text{-}1]$ es igual a P y existirán k ocurrencias en total.

PM se soluciona a diario cuando se utilizan editores de texto o lectores de PDF, al momento de buscar un carácter o un conjunto de caracteres en el texto. Además, en muchas áreas de investigación, es un problema común que necesita ser resuelto para trabajar, como en investigaciones de: procesamiento de lenguajes naturales [PT21], ciberseguridad [GSM14], *information retrieval* [VJ16] y bioinformática [AR09], utilizando textos con tamaños que van desde los cientos de Megabytes hasta los Terabytes de memoria. Generalmente se pueden utilizar tres técnicas diferentes para atacar PM, las cuales son:

*Esta investigación ha sido parcialmente financiada por la Vicerrectoría de Investigación de la Universidad Austral de Chile en el marco del proyecto InnovING:2030, a través del Proyecto de Instalación VIDCA 2020 (VFCI) 16ENI266903.

[1]Secuencia finita de símbolos extraídos de un alfabeto.

[2]Conjunto finito de símbolos.

1. Algoritmos *online*. Recorren S de izquierda a derecha, procesando segmentos dependientes de la longitud de P para buscar las ocurrencias; los algoritmos más comunes son: Fuerza bruta, *Boyer-Moore* [BM77] y *Karp-Rabin* [KR87].
2. Índices de texto completo (IT). Preprocesan completamente S para crear una estructura de datos que permite resolver PM utilizando, necesariamente, el texto S y la estructura de datos creada; los índices de texto completo más usados son: *Suffix Trees* [Wei73] y *Suffix Array* (SA) [MM93].
3. Auto-índices comprimidos de texto completo (IC). Versión comprimida de los IT. Preprocesan completamente S para crear una estructura de datos que almacena S de manera comprimida. Trabaja en espacio comprimido y resuelve PM utilizando la estructura de datos creada; los IC más conocidos son: *Compressed Suffix Array* [GV00], *FM-Index* (FMI) [FM05] e *Hybrid Index* (HI) [FGHP14].

Los algoritmos *online* funcionan de manera distinta a los IT y a los IC, ya que operan directamente sobre S, recorriéndolos de izquierda a derecha, procesando segmentos de largo m; los mejores algoritmos logran reportar todas las ocurrencias del patrón, independiente de su largo en $O(n)$ [CCG+94]. Por otro lado, tanto los IT como los IC, necesitan preprocesar el texto S antes de la primera consulta de PM, pagando una única vez un tiempo extra para construir una estructura de datos que permite resolver PM más rápido, a costo de un espacio adicional. Por ejemplo, el SA de S se puede construir secuencialmente en $O(n)$ [KS03], gastando $O(n)$ palabras de memoria en espacio adicional. Luego, utilizando S y la estructura de datos creada, se resuelve PM mediante búsquedas binarias sobre la estructura, logrando un tiempo de $O((log_2(n) * m) + k)$.

Los algoritmos online y los IT son influenciados por diferentes variables que determinan su tiempo de ejecución final. Ambos dependen directamente de la longitud de S y, generalmente, de la longitud de P. Estas dos variables se suman a la repetitividad[3] del texto S, e influyen en el tiempo de la siguiente manera:

1. El tamaño de S. Mientras mayor sea el tamaño de S, más probable es encontrar ocurrencias. Esto se debe a que existe una mayor cantidad de caracteres donde buscar.
2. La longitud de P. Mientras menor sea la longitud de P, más probable es que exista una ocurrencia en S, ya que un patrón de menor tamaño posee una menor cantidad de caracteres que deben coincidir en S.
3. La repetitividad de S. Mientras más repetitivo sea S, y P se encuentra en S, existirá una cantidad mucho mayor de ocurrencias que en textos convencionales[4].

El depender de estas variables hace que utilizar estas metodologías y estructuras sobre textos abultados sea una mala idea, tanto en tiempo de ejecución como en espacio utilizado para almacenar las estructuras creadas. Para solventar estos

[3] Segmentos de texto de largos considerables, que aparecen muchas veces en el texto completo.

[4] Textos no repetitivos normalmente de uso cotidiano, como novelas.

problemas se idearon los IC. Estos se basan en los IT, pero utilizan el texto S solamente para crear su estructura de datos, ya que almacenan a S de manera comprimida. Esto permite utilizar textos de gran tamaño como entrada sin ocupar mucho espacio.

Es sencillo pensar que los IC presentan tiempos de ejecución altos porque reducen la memoria ocupada, pero no es el caso, ya que operan en espacio comprimido sin necesidad de descomprimir el texto, o solo realizando descompresiones parciales de S. Es decir, sus estructuras de datos y heurísticas les permiten resolver el problema completo trabajando sobre un problema de menor tamaño, otorgándoles tiempos de ejecución muy similares a los de IT. Todos los IC resuelven PM en este espacio comprimido, pero sus heurísticas se enfocan en las variables que influyen en el tiempo de ejecución, como la repetitividad de S. Por ejemplo, el mejor caso del FMI [GKK+16] es resolver PM con textos convencionales, mientras que en el HI [FGHP14] es usando textos repetitivos.

El HI es un IC creado en 2014 por Ferrada and al. [FGHP14] como una alternativa mixta, que ocupa estrategias presentes en el estado del arte para dar solución a PM en textos repetitivos. Luego, en 2018 se implementó una nueva versión del HI[5] [FKP18] que otorgaba muy buenos resultados tanto para textos repetitivos como convencionales, posicionándolo como uno de los mejores IC para cualquier tipo de texto.

La gran diferencia que posee el HI frente a los otros IC, es que realiza un paso extra antes de construir sus estructuras de datos. Ocupa como entrada el diccionario LZ77 S', el cual es la salida obtenida al aplicar el algoritmo de compresión LZ77 [ZL77] a S. Este paso previo se ejecuta por un motivo extra aparte de trabajar en un espacio más comprimido, y es que en 1996 Kärkkäinen and Ukkonen [KU96] demostraron que aplicar LZ77 a un texto, permite encontrar dos tipos de ocurrencias de un patrón, las ocurrencias primarias y secundarias. Además, plantearon una forma rápida para localizar las ocurrencias secundarias a partir de las primarias, utilizando recursividad y *Range Minimun Queries* [BV93] sobre una grilla que se construye a partir de S'.

El HI [FGHP14][FKP18] resuelve PM encontrando todas las ocurrencias primarias y secundarias mediante un algoritmo denominado *locate*, que realiza lo siguiente: las ocurrencias secundarias las localiza, a partir de las ocurrencias primarias, siguiendo lo planteado por [KU96], mientras que para las primarias crea un texto S'' a partir de S', con $|S''| < |S|$, el cual es un texto convencional que posee la primera ocurrencia de todo *substring* existente en S de largo menor o igual a M, para un M dado como parámetro en la construcción del HI. Esto permite utilizar cualquier IC presente en el estado del arte para resolver PM sobre este texto S'', encontrando así, toda ocurrencia primaria en S''. El HI actual [FKP18] utiliza el FMI de Gog and al. [GKK+16], que era la versión más rápida y a la vez que más comprimía de los FMI de la fecha. La implementación

[5] La implementación está disponible en: https://github.com/hferrada/HydridSelfIndex.

es parte de la librería SDSL[6] [GP14], que contiene implementaciones del estado del arte de estructuras de datos comprimidas.

El HI soluciona muchos de los problemas que aparecen al querer resolver PM, pero hay uno que no puede ser alterado aunque se trabaje en espacio comprimido, y es la cantidad total de ocurrencias a encontrar. Esta cantidad total se ve afectado por el tamaño de S y/o la longitud de P y/o la repetitividad de S. La solución común al querer acelerar el cómputo cuando se trabaja sobre un problema de gran tamaño es mediante la paralelización. Esta solución, aplicado a PM, debería funcionar mejor cuando la cantidad total de ocurrencias a encontrar es alta. Esto se debe a que la paralelización penaliza con un tiempo de configuración, y si el problema es pequeño, no vale la pena paralelizar.

Las formas más comunes de paralelización son utilizando la CPU[7] y la GPU[8]. Cada una de estas opera bajo diferentes principios permitiendo mejoras en los tiempos de ejecución. Las mejoras son medidas principalmente utilizando la ley de Amdahl [Amd67].

El presente trabajo pretende determinar si una implementación paralela del algoritmo *locate* del HI mejora los tiempos de ejecución de PM en textos repetitivos y convencionales.

Contribuciones del trabajo Los principales resultados obtenidos son: una implementación paralela del algoritmo *locate*[9] del HI con mejoras, sobre el tiempo promedio por ocurrencia, de entre los $\sim$ 1,25$\times$ y $\sim$ 4,8$\times$ con respecto al algoritmo original, tanto para el mejor caso como el peor caso.

El resto del artículo se distribuye de la siguiente manera: la **Sección 2** describe términos importantes, usados en el trabajo. La implementación realizada se especifica en la **Sección 3**. La **Sección 4** explica los experimentos a realizar, además de exhibir y analizar los resultados. Las conclusiones y reflexiones del trabajo son presentados en la **Sección 5**.

2 Preliminares

2.1 Paralelismo en CPU

Una de las técnicas más usadas para mejorar la capacidad de cómputo al querer resolver un problema es utilizar paralelismo. Esta técnica permite separar una tarea que se ejecuta de manera secuencial utilizando un núcleo, en sub-tareas que se ejecutan en simultáneo en múltiples núcleos (llamados *threads*), resolviendo su propio sub-problema. Luego, se deben procesar en conjunto los resultados de cada thread, para dar la solución al problema que la tarea paralelizada pretende resolver.

[6] La librería está disponible en: https://github.com/simongog/sdsl-lite.

[7] Unidad de procesamiento central.

[8] Unidad de procesamiento gráfico.

[9] La implementación está disponible en: https://github.com/mWrathion/Parallel-Locate.

2.2 Medidas de rendimiento de un programa paralelo

La manera más común de medir que tan bueno es el tiempo de ejecución de un algoritmo contra a otro es a través del *speedup* (S), el cual es una razón entre el tiempo del algoritmo con el que te quieres comparar (T_1) y el tiempo del nuevo algoritmo (T_2), siguiendo la siguiente fórmula: $S = \frac{T_1}{T_2}$. Esta fórmula aplica de igual manera al querer comparar los tiempos entre un algoritmo secuencial contra una de sus versiones paralela, operando sobre lo siguiente: $S_p = \frac{T_s}{T_p}$, donde S_p es el *speedup* paralelo utilizando p procesadores, T_s es el tiempo de la versión secuencial y T_p es el tiempo de algoritmo paralelo utilizando p procesadores. Cabe resaltar que: $S_p \leq p$.

Generalmente, los algoritmos paralelos siempre poseen una porción del código secuencial que no pudo ser paralelizado. Este segmento es un cuello de botella en el tiempo de ejecución, ya que siempre se ejecutará con un solo procesador, como muestra la **Figura 1**.

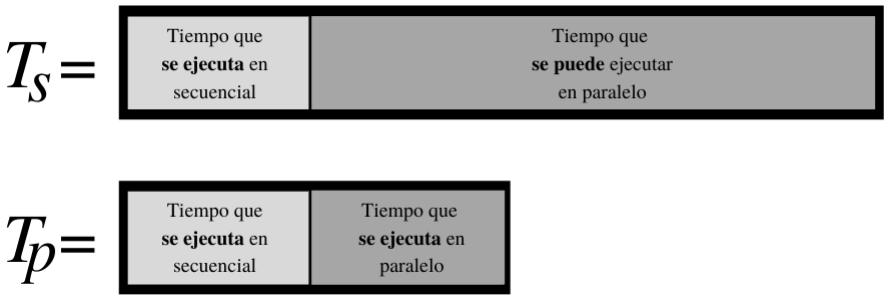

Figura 1. Ejemplo de la distribución del tiempo de un algoritmo secuencial (T_s) y la de un algoritmo paralelo (T_p) con p procesadores.

En 1967, Gene Amdahl sugirió una nueva fórmula para el *speedup* paralelo que considera los segmentos secuenciales y paralelos que existen en la comparación entre un algoritmo secuencial y su versión paralela, conocida como la Ley de Amdahl [Amd67].

Ley de Amdahl Esta ley sugiere considerar la fracción del tiempo que se ejecuta en secuencial (f_s) y la fracción del tiempo que se puede ejecutar en paralelo (f_p) dentro del algoritmo secuencial (ver **Figura 1**), obteniendo dos fórmulas que calculan speedup paralelo:

1. $S_p = \frac{1}{f_s + \frac{f_p}{p}}$, donde p es la cantidad de procesadores que se pretende utilizar. Esta fórmula asume que la cantidad de procesadores afecta de manera lineal a la fracción de tiempo paralelizable. Si se asume que existen infinitos procesadores ($p = \infty$), la fórmula se modifica a $S_p = \frac{1}{f_s}$, así sin la necesidad de

implementar un algoritmo paralelo, se puede saber cual es el mejor *speedup* que se puede obtener.

2. $S_{s,p} = \frac{1}{f_s + \frac{f_p}{s}}$, donde s es la razón entre el tiempo que se puede ejecutar en paralelo del algoritmo secuencial y el tiempo que se ejecuta en paralelo del algoritmo paralelo. Esta fórmula permite comparar que tan rápido es el algoritmo paralelo versus el secuencial, teniendo en cuenta la fracción de tiempo secuencial que ambos algoritmos comparten.

Los valores de speedup mayores que 1 significa que el algoritmo creado es X veces mejor que el algoritmo original (donde X es el speedup), mientras que valores menores que 1 significa que el algoritmo original es mejor.

2.3 LZ77

En 1977 A. Lempel y J. Ziv publicaron un algoritmo de compresión sin pérdida, al que denominaron factorización LZ77 [ZL77]. En la actualidad, es la base de esquemas de compresión usados a diarios como GIF, PNG y ZIP; y se han implementado nuevas versiones de este algoritmo como el LZSS [SS82] y LZ-End [KN10].

La factorización LZ77 se basa en el concepto de *longest previous factor* (LPF). El LPF de un texto X en la posición i es un par (p_i,l_i), tal que X[$i..i+l_i-1$] es igual a X[$p_i..p_i+l_i-1$], con $p_i < i$ y el mayor l_i posible. En otras palabras, el mayor substring[10] que comienza en la posición i y que tenga al menos una ocurrencia previa en el texto.

Dado un texto X[$1..n$], la factorización LZ77 de X es un algoritmo *greedy*[11] que transforma a X en sus *longest previous factors* (LPFs), mientras lo recorre de izquierda a derecha. Como resultado se producen z LPFs, que se denominarán frases LZ. Si la posición i produce la j-ésima frase, que es representada a partir del par (p_i,l_i), entonces la (j+1)-ésima frase iniciará en la posición $i + l_i$. Existe una excepción cuando $l_i = 0$, que ocurre sí X[i] es la ocurrencia de más a la izquierda de un carácter en X, entonces la j-ésima frase es representada a partir del par (X[i], 0), y la (j+1)-ésima frase iniciará en la posición $i + 1$. Además, cuando $l_i > 0$, el *substring* X[$p_i..p_i + l_i - 1$] es denominado como *source* de la frase X[$i..i+l_i-1$] y se representa como (p_i, l_i).

Por ejemplo, la factorización LZ77 del texto X = *zzzzzapzap* produce las siguientes 5 frases LZ:

$$(z, 0), (1, 4), (a, 0), (p, 0), (5, 3)$$

Donde $(z, 0)$, $(a, 0)$ y $(p, 0)$ son las primeras ocurrencias de los caracteres z, a y p respectivamente. Mientras que $(1, 4)$ y $(5, 3)$ son los *sources* de los *substrings* X[2..5] y X[8..10] que aparecieron anteriormente en la posición X[1..4] y X[5..7] respectivamente.

[10] Secuencia contigua de caracteres dentro de un *string*.

[11] Estrategia de búsqueda por la cual se sigue una heurística consistente en hacer elecciones localmente óptimas, con la esperanza de encontrar un óptimo global.

Es bueno notar que mientras más repetitivo sea el texto, mejor será la compresión obtenida, gracias a la reutilización de los *substrings* que aparecen anteriormente. Esto implica que la factorización LZ77 de un texto repetitivo tendrá como resultado pocas frases LZ, pero el largo promedio l_i de sus frase será grande. Caso contrario ocurre cuando se aplica a un texto convencional, donde existirán muchas frases LZ, donde el valor l_i de sus frase será pequeño. Además, el diccionario resultante es siempre, sin importar si el texto de entrada es repetitivo o convencional, un texto convencional con z frases LZ.

2.4 Ocurrencias primarias y secundarias

En 1996, J. Kärkkäinen y E. Ukkonen observaron que, para un texto S y P, la factorización LZ77 de S permite hablar de dos tipos de ocurrencia de P en S: las que cruzan los límites de las frases LZ, y las que no, definiéndolas como ocurrencias primarias y ocurrencias secundarias respectivamente [KU96].

La **Figura 2** muestra gráficamente la definición anterior. La idea clave es que una ocurrencia secundaria siempre es parte de una ocurrencia previa que cruza un límite de frase o de una ocurrencia secundaria anterior (conocido como un predecesor). Esto se debe al funcionamiento de la factorización LZ77, que se basa en encontrar todos los LPF del texto de entrada.

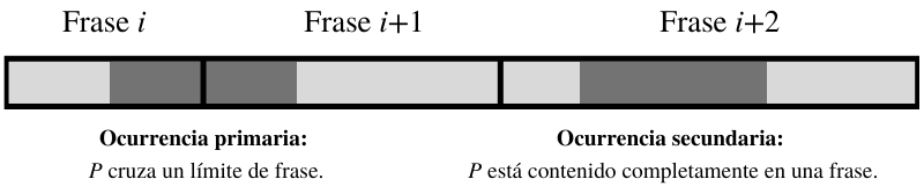

Figura 2. Representación de una ocurrencia primaria (cruza límite de una frase LZ) y una ocurrencia secundaria (completamente contenida en una frase).

Junto a la observación de que cada ocurrencia secundaria esta contenida en otra ocurrencia anterior (ya sea primaria o secundaria), J. Kärkkäinen y E. Ukkonen crearon un algoritmo recursivo y eficiente para localizar todas las ocurrencias secundarias que se forman a partir de una ocurrencia anterior. Este algoritmo depende de dos estructuras: una para encontrar el predecesor de una ocurrencia y otra para consultas RMQ [KU96]. Es bueno notar que la eficiencia de este algoritmo depende directamente de la eficiencia en las consultas a ambas estructuras, por lo tanto si la consulta sobre los predecesores y la RMQ es en tiempo constante, entonces el algoritmo recursivo recupera cada ocurrencia en tiempo constante.

2.5 Range Minimum Query Problem

El *Range Minimun Query* (RMQ) *problem* se define como: dado un arreglo A[1..n], construir una estructura de datos que ante cualquier par de posiciones (i, j), con $1 \leq i \leq j \leq n$, retorne el índice del elemento mínimo presente en A[i..j], como se aprecia en la **Figura 3**.

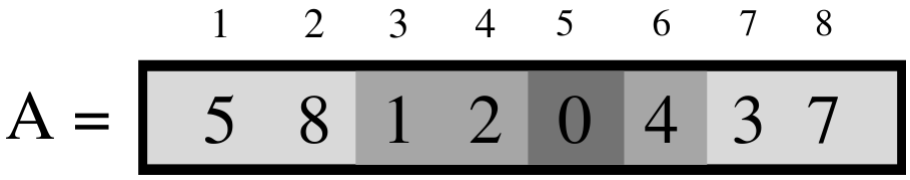

Figura 3. Ejemplo de consulta $RMQ_A(3,6)$, dando como resultado el índice 5, ya que A[5] = 0 es el valor mínimo en A[3..6].

Este es un problema muy estudiado ya que tiene relación directa con otros problemas importantes, como el *Longest Common Ancestor* (LCA) [GBT84][BV93]. Debido a esto se han creado muchas implementaciones, como la de Bender and Farach [BFC00] o la mejor solución en la actualidad implementada por Ferrada and Navarro [FN17] que responde la consulta en tiempo constante.

Es bueno notar que una RMQ también puede buscar el máximo en un rango. Esto se realiza multiplicando todas sus casilla por -1 al momento de ser creada.

2.6 Suffix Array

El SA [MM93][KS03] es el IT más simple y permite resolver PM a partir de búsquedas binarias sobre su estructura y consultas sobre S en *O((m*log(n))+k)*. Es un arreglo de tamaño n, siendo n el tamaño de S; en el cual se almacenan los índices de inicio de cada sufijo del texto en orden lexicográfico. Al necesitar tanto el arreglo de sufijos ordenados como el texto de entrada para resolver PM, no es muy útil cuando el texto es de tamaño considerable. Por esta razón nacen los IC que no necesitan del texto de entrada después de ser construidos.

2.7 Auto-índices comprimidos de texto completo

Los IC son un conjunto de estructuras de datos que almacenan S de manera comprimida y pueden trabajar sobre el texto completo sin la necesidad de descomprimirlo. Estos suplen la necesidad presentada en los IT y logran usar un espacio menor, resolviendo los mismos problemas.

En la actualidad los IC más usados y conocidos son: Compressed Suffix Array [GV00], el Full-text index in Minute space [FM05] y el Hybrid Index [FGHP14], donde cada uno aplica diferentes heurísticas para poder almacenar S de manera comprimida y poder resolver PM.

1. **CSA**. Elaborado por Roberto Grossi and Jeffrey Vitter en el año 2000, es la versión comprimida del SA, pero aún necesitando el texto para operar. La estructura consiste en un diccionario en el formato (c_i, p_i), donde c_i es el primer carácter del sufijo i y p_i es la primera posición de ese sufijo en el SA. En 2003 Sadakane implementó una mejora sobre CSA añadiendo un segundo arreglo compresible denominado Ψ permitiendo representar S con las estructuras [Sad03].
2. **FMI**. Otra representación del SA creado por Ferragina and Manzini en 2005. En este caso el SA se representa como la inversa del arreglo Ψ, lo que permite retroceder sobre el SA; resolviendo PM utilizando *Backwards search*[FM05] en vez de búsqueda binaria como lo hace el SA y el CSA. En la actualidad se han implementado muchas mejoras sobre este IC, mejorando sus tiempos de ejecución [GKK+16].

2.8 Hybrid Index

El HI es un IC elaborado por Ferrada and al. en el año 2014 [FGHP14], que permitía resolver PM en textos repetitivos de manera rápida y ocupando poco espacio gracias a heurísticas que aprovechan la repetitividad del texto para funcionar. Luego, en 2018 Ferrada et al. [FKP18] mejoraron el HI otorgándole buenos resultados de PM sobre textos convencionales, mejorando los tiempos en textos repetitivos y siendo el IC que menos espacio ocupa en disco.

Este IC no utiliza el texto original como entrada, sino a un diccionario creado a partir de la factorización LZ77 de S, denominado S'. Luego, el HI es construido a partir de S' y un largo óptimo de patrón M (entregado por el usuario), el cual permite al HI responder consultas de PM para cualquier patrón P de cualquier largo m; siendo muy eficiente para largos $m \leq M$, y penalizando el tiempo para largos mayores.

Construir el HI significa fabricar y almacenar en memoria, una única vez por texto y antes de aplicar PM, un conjunto de estructuras de datos que permiten, a partir del diccionario de entrada, mapear las ocurrencias encontradas con su posición original en S, sin la necesidad de descomprimir S'. Dado el texto de entrada $S[1..n]$ y el diccionario LZ77 S' de z frases, las estructuras más importantes que construye el HI para resolver PM son:

1. $\mathbf{S''}$, un texto formado por la concatenación de *substrings* de S, con $|S''| < |S|$; donde S'' se construye a partir del texto original S y el diccionario LZ77, el cual es un diccionario creado a partir de S'; donde se reduce el número de frases LZ de z a z' gracias a la creación de frases literales. Para esto, se aplican las siguientes dos heurísticas: *(1)* Si el largo de las i-ésima e *(i+1)*-ésima frases LZ son menores o iguales que M se unirán formando una frase literal, *(2)* si solamente el largo de la i-ésima es menor o igual que M se marcará como frase literal. Todos los demás puntos serán denominados como *sources* y almacenarán un par $(inicio, fin)$, que representa el inicio y el fin de la frase en el texto original. Es bueno notar que S'' ayuda a los textos convencionales, ya que sus largos de frase promedio son mucho menores que

en los textos repetitivos, por lo que las condiciones tienden a cumplirse con mayor facilidad.

2. **Un IC creado sobre S″**, el cual nos permite encontrar todas las ocurrencias existentes en S''. El HI acepta cualquier IC del estado del arte y actualmente se ocupa el FMI de Gog and al. [GKK+16], ya que posee muy buenos resultados en textos convencionales como lo es S''. De esta manera se resuelve el problema de encontrar las ocurrencias primarias, ya que el FMI devuelve todas las ocurrencias del patrón en S'' cuyo texto contiene todo posible substring de S de largo menor o igual a M; además, es posible reportar *strings* de largos mayores mediante uniones de consultas de *substrings* de largo óptimo M.
3. **Arreglo de predecesores**, el cual es un arreglo ordenado de menor a mayor creado a partir de S'' y que posee el inicio de todos los *source* que no cumplieron con las condiciones de frases literales. Este arreglo permite retornar, mediante una búsqueda binaria, el índice del mayor inicio de un *source* que sea menor o igual a la posición de una ocurrencia encontrada.
4. **Búsqueda de rangos en dos dimensiones**. Corresponde a la propuesta de Kärkkäinen and Ukkonen [KU96], implementada mediante un conjunto de estructuras que permiten encontrar, recursivamente, todas las ocurrencias secundarias desde una ocurrencia ya encontrada. Es creada a partir de los pares $(inicio, fin)$ de cada *source* existente en S''. Se utiliza el arreglo de predecesores (creado con todos los valores *inicio* ordenados de menor a mayor) y una RMQ (que busca el máximo en un rango) con todos los valores fin, utilizando el mismo orden que el arreglo de predecesores. Cada valor almacenado en la RMQ posee, como dato satélite, el punto de partida de esa frase sobre el texto original, de esta manera se puede recuperar a partir de cualquier frase, el inicio de esa frase en S. Esta estructura busca el índice del predecesor de una ocurrencia primaria encontrada en $xPos$ utilizando el arreglo de predecesores, retornando el índice r. Luego, utilizando la RMQ$[1..r]$ se obtiene el índice c, con $1 \leq c \leq r$, el cual es el índice del *source* que posee el mayor fin. Finalmente, se obtiene un *source* que inicia en $Predecesor[c]$, termina en el fin_c, el fin del c-ésimo *source* ordenado; y aparece en la posición j del texto original, por lo que se reporta como ocurrencia secundaria la posición $j + xPos - Predecesor[c]$ si y sólo sí $fin_c \geq xPos + m$, para después buscar posibles nuevas ocurrencias secundarias a partir de la recientemente encontrada y seguir buscando por ocurrencias secundarias de $xPos$ en los rangos $[1..c-1]$ y $[c+1, r]$.

El HI responde PM utilizando su algoritmo *locate*, desarrollado por Ferrada and al. [FKP18], el cuál necesita un HI almacenado en memoria y uno o más patrones de largo m a buscar. Este algoritmo consiste en tres pasos, los cuales son:

1. Encontrar las posibles ocurrencias primarias utilizando un IC del estado del arte (en este caso se utiliza el FMI de Gog and al. [GKK+16]).
2. Recorrer de manera iterativa cada una de estas posibles ocurrencias primarias y verificar cuales son realmente ocurrencias del patrón.

3. Por cada ocurrencia primaria real calcular sus ocurrencias secundarias de manera recursiva utilizando la estructura de búsqueda de rangos en dos dimensiones.

El **Algoritmo 1** presenta en forma de pseudocódigo los pasos descritos para realizar *locate* sobre un texto utilizando el HI, mientras que el **Algoritmo 2** presenta como conseguir las ocurrencias secundarias a partir de las primarias de manera recursiva.

Algoritmo 1: Locate HI secuencial

Entrada : Patrón a buscar P de largo m, todas las estructuras ya construidas del HI.
Salida : Arreglo *occs* con la posición de todos las ocurrencias encontradas.

```
1  procedimiento locateSec(P, m, HI)
2      posiblesOccsPrims ← FMI.locate(P);
3      nLoc ← tamaño de posiblesOccsPrims;
4      si nLoc ≤ 0 entonces
5          retornar ; /* No existe ninguna posible ocurrencia primaria */
6      fin
7      Sea occs un arreglo de enteros de largo nLoc;
8      nOccs ← 0;
9      para i = 0 hasta i < nLoc hacer
10         si posiblesOccsPrims[i] es frase literal o cruza un límite de frase
           entonces
11             occs[nOccs] ← posición original de la ocurrencia primaria ;
12             nOccs++;
13         fin
14     fin
15     para i = 0 hasta i < nOccs hacer
16         si occs[i] tiene predecesor entonces
17             r ← predecesor de occs[i];
18             locateSecondSec(1, r, occs[i], m, nLoc, nOccs, occs);
19         fin
20     fin
21     retornar occs;
22 fin
```

De esta manera nacen las siguientes preguntas de investigación: *¿Es factible paralelizar las consultas locate del HI?*, de ser afirmativo, *¿cuál es el speedup que se obtiene en textos con diferente grados de compresibilidad?*.

Algoritmo 2: Buscar ocurrencias secundarias HI secuencial

Entrada : Valores del rango l y r para las búsquedas de la RMQ, posición de la ocurrencia original encontrada $xPos$, largo de patrón m, largo del arreglo original $nLoc$, cantidad de ocurrencias encontradas $nOccs$, arreglo $occs$.

Salida : Arreglo $occs$ con la posición de todas las ocurrencias encontradas.

```
1  procedimiento locateSecondSec(l, r, xPos, m, nLoc, nOccs, occs)
2      c ← RMQ(l, r);
3      inicio_c ← Predecesor[c];
4      fin_c ← Obtener el final del c-ésimo source ordenado gracias a la RMQ y el
         diccionario LZ77;
5      j ← inicio de la frase en el texto original;
6      si fin_c < xPos + m entonces
7          retornar; ;                              /* Condición de término */
8      fin
9      si nOccs ≥ nLoc entonces
10         Hacer doubling array para occs[];
11         nLoc ← nLoc * 2;
12     fin
13     nuevaOcc ← j + xPos − inicio_c;
14     occs[nOccs] ← nuevaOcc;
15     nOccs++;
16     si nuevaOcc tiene predecesor entonces
17         nuevoR ← predecesor de nuevaOcc;
18         locateSecondSec(1, nuevoR, nuevaOcc, m, nLoc, nOccs, occs);
19     fin
20     si c > l entonces
21         locateSecondSec(l, c-1, xPos, m, nLoc, nOccs, occs);
22     fin
23     si c < r entonces
24         locateSecondSec(c+1, r, xPos, m, nLoc, nOccs, occs);
25     fin
26 fin
```

3 Paralelización del algoritmo locate

Al querer desarrollar un algoritmo paralelo utilizando CPU, hay que tener en cuenta algunas heurísticas que van de la mano al querer mejorar el tiempo de ejecución, las cuales son:

1. **Trabajar en celdas contiguas de memoria**. Esto quiere decir que recorrer un arreglo/vector A de tamaño N utilizando nt threads, implica que cada thread trabaje sobre un rango contiguo de tamaño $\frac{N}{nt}$ sobre A denominado *chunk*. La **Figura 4** presenta gráficamente como se distribuye un trabajo paralelo en CPU sobre un arreglo A de tamaño 11 usando 3 threads, en este caso los dos primeros threads tienen un *chunk* de 4 mientras que el último tiene un *chunk* de 3.

2. **Asegurar que la escritura de cualquier valor en una celda de memoria sea en paralelo sin hacer uso de *locks***[12]. Para cumplir esta escritura en paralelo, se pueden reemplazar las variables en las cuales se quiere escribir por vectores de tamaño nt y ocupar el tid[13] como índice del nuevo vector. De esta manera, cada thread tiene un espacio de memoria propio para resolver su sub-problema, el cual es accesible con su tid.
3. **Unificar los resultados de todos los threads para dar solución al problema original de manera paralela en un único arreglo solución**. Al tener todas las soluciones locales y sabiendo cuántas soluciones tiene cada thread se puede calcular, mediante un *prefix-sum* secuencial (se hace secuencial ya que el *prefix-sum* es de largo $nt+1$), el tamaño final del arreglo solución (con el valor de *prefix-sum*$[nt]$), el nuevo inicio de cada thread (con el valor de *prefix-sum*$[tid]$) y el *chunk* de cada thread (con la cantidad de soluciones encontradas por ese thread). Estas tres variables permiten crear un mapa entre las soluciones locales de un thread con su posición final en el arreglo solución. De esta manera se permite una escritura en el arreglo solución de manera paralela, pagando un pequeño costo de $O(nt+1)$ para calcular la *prefix-sum* secuencial.

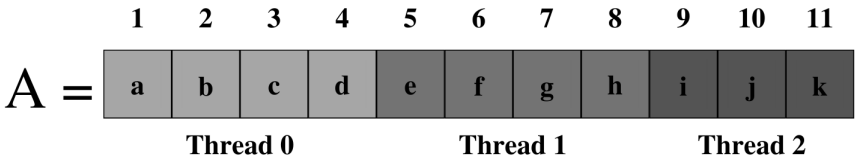

Figura 4. Ejemplo de trabajo paralelo en CPU sobre un arreglo con $N = 11$ y $nt = 3$.

En este trabajo se desarrolló un algoritmo paralelo que sigue los pasos del algoritmo secuencial original de Ferrada and al. [FKP18] y añade las heurísticas ya descritas para realizar la paralelización. De esta manera, se obtienen los **Algoritmos 3** y **4**.

El **Algoritmo 3** funciona similar al **Algoritmo 1** con la diferencia que cada thread calcula su zona de trabajo (como muestra la **Figura 4**), almacenan las ocurrencias encontradas con el uso de dos vectores (*occsTid* las ocurrencias que cada thread encuentra y *contTid* la cantidad de ocurrencias encontrada por thread) y un tercer vector que calcula la *prefix-sum* secuencial (*psTid*, entre las líneas 32 y 37) que permite unificar todas las ocurrencias encontradas por cada thread en un solo vector solución de manera paralela (entre las líneas 39 y 41).

[12] Mecanismo que bloquea una celda de memoria cuando es usada por un thread, haciendo esperar a los demás threads que quieren usarla.

[13] Acrónimo de *Thread ID*, cada thread tiene un ID único que puede tomar un valor en el rango $[\mathbf{0}..\mathbf{nt}-\mathbf{1}]$

El **Algoritmo 4** es realizado por cada thread de manera paralela. Esto quiere decir que se ejecutan nt recursividades a la vez buscando cada una sus ocurrencias secundarias y aumentado los contadores si es que son encontrados.

4 Resultados experimentales

4.1 Entorno de prueba

Este trabajo realizó los experimentos en el servidor Patagón de la Universidad Austral de Chile, con 2x CPU AMD EPYC 7742 (2.6GHz, 64-cores, 256MB L3 cache) y 1 TB RAM DDR4-3200Hz. El servidor no ejecutaba nigún otro trabajo significante en CPU y se ocuparon 1, 8, 16, 24 y 32 threads por experimento.

4.2 Textos utilizados

Se utilizaron 3 textos presentes en el dataset de textos reales y altamente repetitivos de la Universidad de Chile[14] y 1 texto presente en el dataset de textos convencionales de la Universidad de Chile[15], formando un conjunto de pruebas de 4 textos con diferentes pesos, tamaños de alfabeto y repetitividad.

El **Cuadro 1** presenta las características de cada texto, donde se aprecia que los textos *world_leaders*, *cere* y *einstein* son altamente repetitivos con porcentajes de compresión del 97.872 %, 97.789 % y 99.881 % respectivamente, mientras que *sources* es un texto convencional con un porcentaje de compresión del 66.794 %.

Otro punto interesante a notar en el **Cuadro 1** es la cantidad de Frazes LZ y el largo promedio que poseen (directamente relacionada con la repetitividad y compresibilidad del texto), ya que está implica que tan extensa será la búsqueda de posibles ocurrencias primarias sobre el FMI (trabajo secuencial) que se realiza antes de iniciar el trabajo paralelo.

4.3 Experimentos

Los pasos presentados a continuación detallan los experimentos realizados en este trabajo.

1. Iniciar con un tamaño de patrón igual a 2.
2. De cada texto extraer 1000 patrones del largo seleccionado a partir de posiciones aleatorias del texto original.
3. Aplicar la Factorización LZ77 a cada texto, a la cual denominaremos S'.
4. Construir un HI por cada S', con un largo de patrón óptimo igual al largo de patrón seleccionado.
5. Por cada HI, calcular locate utilizando los patrones extraídos del texto que le corresponde. Este paso se realiza utilizando la versión secuencial original y utilizando la versión paralela con 8, 16, 24 y 32 threads.

[14] Se pueden encontrar en http://pizzachili.dcc.uchile.cl/real-stats.html
[15] Se puede encontrar en http://pizzachili.dcc.uchile.cl/texts.html

Algoritmo 3: Locate HI paralelo

Entrada : Patrón a buscar P de largo m, cantidad de threads a usar nt, todas las estructuras construidas del HI.
Salida : Arreglo *occs* con la posición de todos las ocurrencias encontradas.

```
1  procedimiento locatePar(P, m, nt, HI)
2      posiblesOccsPrims ← FMI.locate(P);
3      nLoc ← tamaño de posiblesOccsPrims;
4      si nLoc ≤ 0 entonces
5          retornar ;  /* No existe ninguna posible ocurencia primaria */
6      fin
7      Sea occsTid un vector de vectores de enteros de tamaño nt;
8      Sea contTid un vector de enteros de tamaño nt;
9      Sea psTid un vector de enteros de tamaño (nt+1);
10     chunk ← nLoc/nt;
11     nOccs ← 0;
12     en paralelo con nt Threads
13         tid ← obtener ID del thread;
14         inicio ← tid * chunk;
15         final ← inicio + chunk;
16         si tid es igual a nt-1 entonces
17             final ← nLoc;
18         fin
19         para i = inicio hasta i < final hacer
20             si posiblesOccsPrims[i] es frase literal o cruza un límite de frase
                entonces
21                 occsTid[tid].añadir(posición original de la ocurrencia
                    primaria);
22                 contTid[tid]++;
23             fin
24         fin
25         esperar a todos los Threads
26         para i = 0 hasta i < contTid[tid] hacer
27             si occsTid[tid][i] tiene predecesor entonces
28                 r ← predecesor de occsTid[tid][i];
29                 locateSecondSec(1, r, occsTid[tid][i], m, tid, contTid, occsTid);
30             fin
31         fin
32         esperar a todos los Threads
33         en secuencial solamente 1 Thread
34             para i = 0 hasta i < nt hacer
35                 psTid[i + 1] ← (psTid[i] + contTid[i]);
36             fin
37             nOccs ← psTid[nt];
38             Sea occs un arreglo de enteros de largo nOccs;
39         esperar a todos los Threads
40         para i = 0 hasta i < contTid[tid] hacer
41             occs[(i + psTid[tid])] ← occsTid[tid][i];
42         fin
43     fin
44     retornar occs;
45 fin
```

Algoritmo 4: Buscar ocurrencias secundarias HI paralelo

Entrada : Valores del rango l y r para las búsquedas de la RMQ, posición de la ocurrencia original encontrada $xPos$, largo de patrón m, ID del thread tid, vector $contTid$, vector $occsTid$.

Salida : Cada thread entrega en su vector $occsTid$ las posiciones de todas las ocurrencias encontradas.

```
1  procedimiento locateSecondPar(l, r, xPos, m, tid, contTid, occsTid)
2      c ← RMQ(l, r);
3      inicio_c ← Predecesor[c];
4      fin_c ← Obtener el final del c-ésimo source ordenado gracias a la RMQ y el
         diccionario LZ77;
5      j ← inicio de la frase en el texto original;
6      si fin_c < xPos + m entonces
7          retornar; ;                              /* Condición de término */
8      fin
9      nuevaOcc ← j + xPos − inicio_c;
10     occsTid[tid].añadir(nuevaOcc);
11     contTid[tid]++;
12     si nuevaOcc tiene predecesor entonces
13         nuevoR ← predecesor de nuevaOcc;
14         locateSecondSec(1, nuevoR, nuevaOcc, m, tid, contTid, occsTid);
15     fin
16     si c > l entonces
17         locateSecondSec(1, c-1, xPos, m, tid, contTid, occsTid);
18     fin
19     si c < r entonces
20         locateSecondSec(c+1, r, xPos, m, tid, contTid, occsTid);
21     fin
22 fin
```

Texto	Alfabeto	PesoTexto	PesoLZ77	FrasesLZ	LenPromedioFrasesLZ
worlds_leader	89	47MiB	1MiB	175,908	267
sources	230	52.4MiB	17.4MiB	3,307,448	15.9
cere	5	461.3MiB	10.2MiB	1,700,859	271.2
einstein	139	467.6MiB	557.6KiB	91,036	5,136.7

Cuadro 1. Características principales de los textos utilizados en las pruebas de locate.

6. Calcular y almacenar los tiempos de cada locate ejecutado.
7. Aumentar en 1 el largo de patrón.
8. Si el largo del patrón es mayor que 10 termina, si es menor o igual a 10 volver al punto 2.

De esta manera sobre cada texto se realiza un locate secuencial y cuatro locate paralelos utilizando los mismos 1,000 patrones extraídos aleatoriamente, y todo esto realizado con 9 tamaños de patrón diferente (desde 2 hasta 10).

4.4 Resultados

Este trabajo estudió el comportamiento entre el algoritmo locate original (**Algoritmos 1** y **2**) su versión paralela implementada (**Algoritmos 3** y **4**), siguiendo las pruebas descritas en la **Sección 4.3**.

La **Figura 5** presenta el resultado del tiempo promedio por ocurrencias (μs) de 1000 patrones de largo 2 (desglosado en tiempo secuencial y tiempo paralelo/paralelizable), sobre cada uno de los textos de prueba presentes en el **Cuadro 1**.

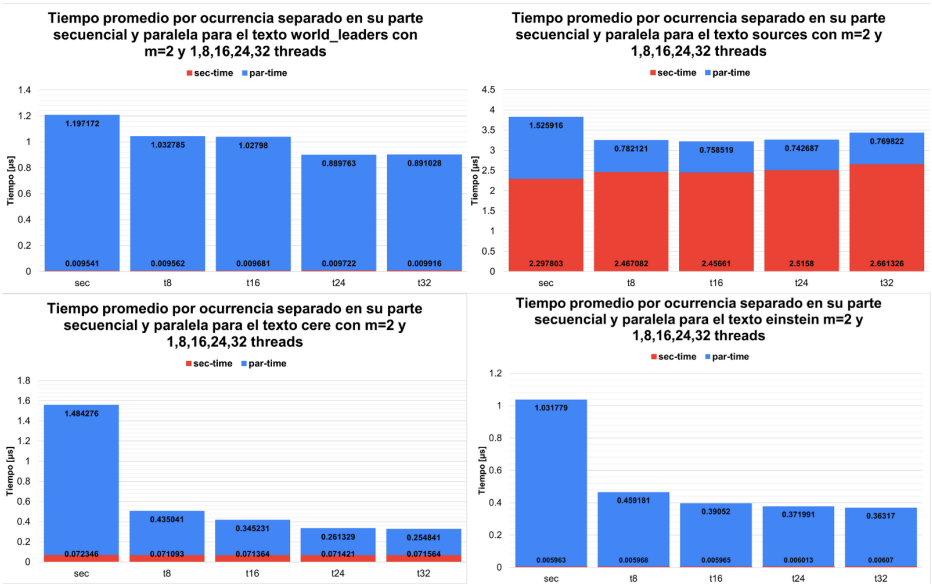

Figura 5. Tiempos promedio por ocurrencia [μs], desglozado en su tiempo secuencial y parelelo, con m=2 para los textos world_leaders, sources, cere y einstein, usando el algoritmo original y el algoritmo paralelo con 8, 16, 24 y 32 threads.

La **Figura 6** presenta el speedup, utilizando el tiempo promedio por ocurrencias, conseguido sobre los 4 textos de pruebas con largos de patrón en el rango [2..10].

En la **Figura 5** se aprecia como el tiempo secuencial puede afectar notoriamente al tiempo final del algoritmo paralelo. Esto se da en el texto sources, que es el único texto convencional además de poseer el menor largo promedio en sus Frases LZ según el **Cuadro 1**; donde su tiempo secuencial ocupa en promedio un 73 % del tiempo total. Para el texto cere el tiempo secuencial afecta en promedio un 15 % mientras que en world_leaders y einstein no afecta de manera notoria al tiempo promedio por ocurrencias. Los resultados obtenidos por world_leaders no presentan grandes mejorías con respecto a la versión secuencial, como sí lo hacen

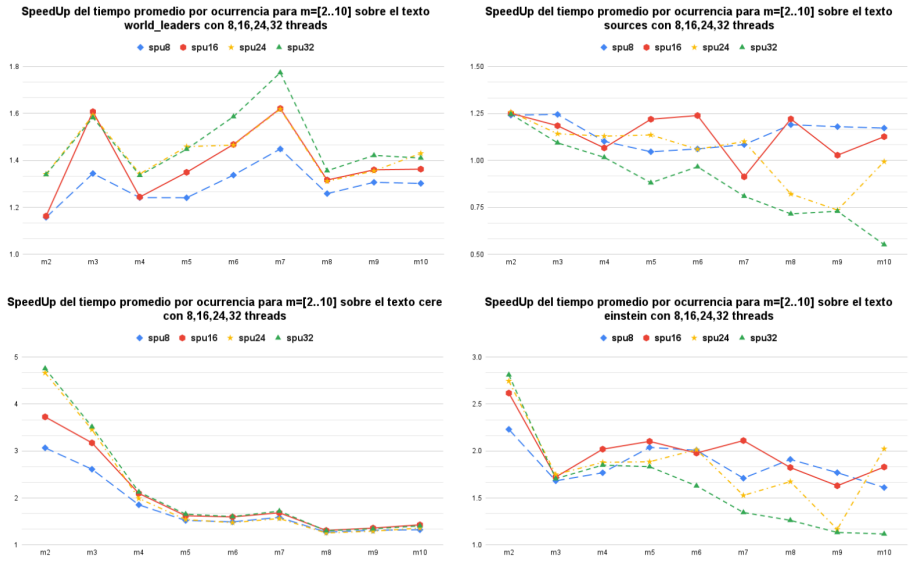

Figura 6. Speedup del algoritmo paralelo sobre el secuencial sobre los textos world_leaders, sources, cere y einstein, con m en el rango [2..10], utilizando 8, 16, 24 y 32 threads.

cere y einstein. Esto se debe a la distribución del trabajo paralelo al momento de encontrar ocurrencias primarias y/o secundarias, ya que cada thread va encontrando sus propias ocurrencias y pueden quedar desbalanceados[16], provocando un cuello de botella al tiempo final. Por último, vemos que en general el tiempo promedio por ocurrencias cuando $m = 2$ es mejor cuando se ocupa cualquier versión paralela. Esto se debe a que un largo de patrón pequeño poseerá muchas más ocurrencias que un largo de patrón de mayor tamaño, por lo que la paralelización tiene un mayor área que abarcar al querer encontrar todos los patrones presentes.

Los valores de speedup presentados en la **Figura 6**, indican que el algoritmo paralelo desarrollado es mejor que el algoritmo original para todos los casos en los textos world_leaders, cere y einstein. Los mejores resultados se ven en cere (altamente repetitivo y de gran tamaño), donde existen valores de speedup de 3.1 (8 threads) hasta 4.8 (32 threads) cuando $m = 2$ y va decayendo respecto al largo de patrón. Los tres textos repetitivos tienden a estabilizarse sobre un speedup del 1.4 pero sus mejores valores se encuentran en los primeros largos de patrón, debido a que la cantidad de patrones a encontrar es mucho mayor y existe una mejor distribución del trabajo en todos los hilos. Para el texto sources el speedup

[16]Cuando uno o más threads hacen una cantidad de trabajo mucho mayor que los demás, haciendo que uno o más threads terminen antes y deban esperar los resultados del/los threads que aún siguen trabajando.

positivo depende de la cantidad de threads que son ocupados, ya que crear X threads toma un tiempo de inicialización que produce un cuello de botella al querer encontrar una cantidad muy pequeña de ocurrencias. En el texto einstein y sources se aprecia como a medida que aumenta el largo del patrón los mejores speedup son encontrados en la menor cantidad de threads, lo que es explicado con la distribución del trabajo paralelo, donde utilizar más threads implica una cantidad muy pequeña de ocurrencias por thread.

Al comparar la **Figura 5** con la **Figura 6** nace la pregunta: ¿Cuánto afecta al speedup esta porción secuencial de tiempo en los textos cere y sources?. La **Figura 7** presenta el speedup de ambos textos teniendo en cuenta solamente el tiempo de ejecución paralelo.

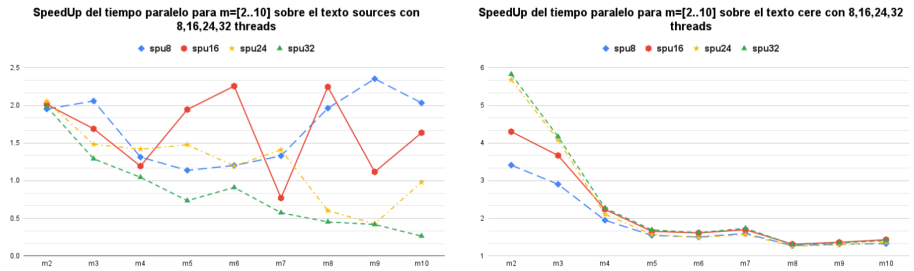

Figura 7. Speedup del algoritmo paralelo sobre el secuencial sobre los textos sources y cere, con m en el rango [2..10], utilizando 8, 16, 24 y 32 threads.

En la **Figura 7**, se distingue una diferencia notoria en el speedup con respecto a la **Figura 6**, aumentando, en los mejores casos (largo de patrón pequeño), hasta 5.8 en cere cuando se ocupan 32 threads y hasta 2 en sources con cualquier cantidad de threads. En el caso de cere el speedup tiende a estabilizarse al mismo punto que antes, indicándonos que el trabajo secuencial disminuye notoriamente a medida que el largo de patrón disminuye. Para sources este speedup de 2 se mantiene con 8 threads debido a que el trabajo secuencial es estable a medida que se aumenta el largo de patrón y el trabajo paralelo disminuye con este largo, explicando los resultados negativos de speedup al utilizar una gran cantidad de threads. La **Figura 8** afirma que el trabajo secuencial de sources se mantiene a medida que el largo de patrón aumenta y que el trabajo secuencial en cere disminuye.

5 Conclusión

Este trabajo logró construir un algoritmo paralelo, a partir de la versión original, que encuentra patrones de es más rápido encontrando patrones en textos convencionales y repetitivos.

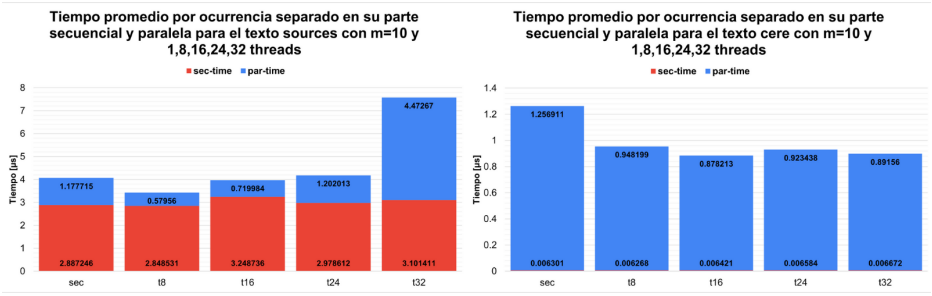

Figura 8. Tiempos promedio por ocurrencia, desglosado en su tiempo secuencial y paralelo, con m=10 para los textos sources y cere, usando el algoritmo original y el algoritmo paralelo con 8, 16, 24 y 32 threads.

Se demostró que, en textos repetitivos de tamaño considerable y con un largo de patrón pequeño, la paralelización es una gran ayuda al querer reducir el tiempo de ejecución del algoritmo, obteniendo resultados de 1.34, 2.8 y 4.8 veces mejor que la versión secuencial. Esto sirve como referencia de comportamiento al querer usar patrones de mayor tamaño en textos repetitivos de gran volumen, que poseen pesos desde los GiB hasta los TiB de información (por ejemplo en secuencias de ADN); donde se espera un escenario cuyo comportamiento sea similar al mostrado en este trabajo al trabajar con textos y patrones pequeños.

La cantidad de threads a utilizar depende completamente del tipo de texto al que se quiere aplicar locate. En textos convencionales se recomienda utilizar entre 8 y 16 threads para tener un speedup positivo independiente del tamaño del patrón. En textos repetitivos como world_leaders y cere se recomienda usar la mayor cantidad de threads posibles, ya que el speedup es mejor en cualquier largo de patrón al momento de utilizar una cantidad de threads mayor. Para textos extremadamente repetitivos como einstein, se recomienda utilizar una estrategia variada. A medida que aumenta el largo de patrón, disminuir la cantidad de threads a usar. De esta manera poder reducir el desbalance de trabajo entre los threads.

La sección secuencial de trabajo previo al momento de querer aplicar locate en el HI puede ser considerable dependiendo del tipo de texto que se utiliza, empeorando considerablemente el rendimiento del algoritmo paralelo.

El desbalance del trabajo paralelo en los threads es un problema que afecta directamente al tiempo de ejecución del algoritmo. Este problema es difícil de solucionar, ya que cada thread va determinando, si su set de posibles ocurrencias primarias lo son en realidad. De esta manera no existe un rango de trabajo real definido previo a encontrar las ocurrencias primarias reales. De la misma manera las ocurrencias secundarias se van encontrando de manera recursiva sin seguir una norma específica, por lo que un thread puede encontrar muchas más ocurrencias secundarias que otros.

Como trabajo a futuro se pretende diseñar e implementar un nuevo IC, con la posibilidad de locate en paralelo, que nos permita reemplazar el actual

FMI y así resolver la sobrecarga de trabajo secuencial en textos convencionales. Además, estudiar el paralelismo dinámico para poder atacar el problema de desbalance del trabajo paralelo.

Agradecimientos

Principalmente agradecer tanto a mi madre, Claudia Letelier, como a mi compañera de vida, Paula Mancilla, por el gran apoyo de todo tipo y la cantidad de amor que me han brindado a lo largo de estos años. A mi padre, hermanos, abuela y animales por brindarme cariño y un lugar donde poder despejarme de la Universidad. A mis compañeros y muy buenos amigos Stevenson, Mathi, Eduardo, Angelo, Heinz y Anti por brindarme risas, charlas, ganas de seguir adelante y un increíble ambiente de estudio. A la música por darme relajo y concentración y así poder dar mi máximo cada día. A la mayoría de mis profesores por brindar su máximo para entregar el gusto de la informática a las personas. Y por último pero no menos importante, a mi profesor, Héctor Ferrada, por estar siempre ahí para resolver dudas, escucharme y entregarme el gusto que tengo por los algoritmos y la investigación.

Referencias

Amd67. Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485. Association for Computing Machinery, 1967.

AR09. Pankaj Agarwal and SAM Rizvi. Pattern matching based technique to solve motif-finding problem. *Bharati Vidyapeeth's Institute of Computer Applications and Management*, page 17, 2009.

BFC00. Michael A. Bender and Martín Farach-Colton. The lca problem revisited. In Gaston H. Gonnet and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics*, pages 88–94, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

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

BV93. Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

CCG+94. Maxime Crochemore, Artur Czumaj, Leszek Gasieniec, Stefan Jarominek, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4):247–267, 1994.

FGHP14. Héctor Ferrada, Travis Gagie, Tommi Hirvola, and Simon J Puglisi. Hybrid indexes for repetitive datasets. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2016):20130137, 2014.

FKP18. Héctor Ferrada, Dominik Kempa, and Simon J Puglisi. Hybrid indexing revisited. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–8. SIAM, 2018.

FM05. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

FN17. Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *Journal of Discrete Algorithms*, 43:72–80, 2017.

GBT84. Harold N Gabow, Jon Louis Bentley, and Robert E Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, 1984.

GKK+16. Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Faster, minuter. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagrista, and James A. Storer, editors, *DCC 2016*, IEEE Data Compression Conference, pages 53–62, United States, 2016. IEEE. Data Compression Conference ; Conference date: 29-03-2016 Through 01-04-2016.

GP14. Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

GSM14. Hossein Gharaee, Shokoufeh Seifi, and Nima Monsefan. A survey of pattern matching algorithm in intrusion detection system. In *7'th International Symposium on Telecommunications (IST'2014)*, pages 946–953. IEEE, 2014.

GV00. Roberto Grossi and Jeffrey Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). *SIAM Journal on Computing*, 35, 03 2000.

HHLS04. Trinh ND Huynh, Wing-Kai Hon, Tak-Wah Lam, and Wing-Kin Sung. Approximate string matching using compressed suffix arrays. In *Annual Symposium on Combinatorial Pattern Matching*, pages 434–444. Springer, 2004.

KN10. Sebastian Kreft and Gonzalo Navarro. Lz77-like compression with fast random access. pages 239–248, 01 2010.

KR87. Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260, 1987.

KS03. Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer Berlin Heidelberg, 2003.

KU96. Juha Kärkkäinen and Esko Ukkonen. Lempel-ziv parsing and sublinear-size index structures for string matching (extended abstract). In *Proc. 3rd South American Workshop on String Processing (WSP'96*, pages 141–155. Carleton University Press, 1996.

MM93. Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

PT21. RK Pandey and S Taruna. Prevalent exact string-matching algorithms in natural language processing: A review. In *Journal of Physics: Conference Series*, volume 1854, page 012042. IOP Publishing, 2021.

Sad03. Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

SS82. James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, October 1982.

VJ16. S. Vijayarani and R. Janani. String matching algorithms for reteriving information from desktop — comparative analysis. In *2016 International Conference on Inventive Computation Technologies (ICICT)*, volume 3, pages 1–6, 2016.

Wei73. Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.

ZL77. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.