DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico: Threading

Sistemas Operativos
Primer Cuatrimestre de 2022

Integrante	LU	Correo electrónico
Martín Schuster	208/18	m.a.schuster98@gmail.com
Guillermo Reyna	393/20	guille.j.reyna@gmail.com
Javier de Sa Souza	319/95	desasouza@gmail.com
Lucia Biglieri	599/16	luciabiglieri@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

Introducción

Este informe fue elaborado como parte de la cursada de la materia Sistemas Operativos de la carrera de Lic. en Ciencias de la Computación. El código descrito puede encontrarse en el siguiente link: `https://github.com/lulabiglieri/TP-SO` junto con las consignas del informe pautadas por la cátedra de la materia.

Objetivo

El presente trabajo tiene por objetivo estudiar el manejo de memoria dinámica en condiciones de concurrencia y describir los problemas que presenta. En particular, la estructura de datos que a estudiar es una tabla de hash abierta que utiliza listas enlazadas para la resolución de colisiones. Analizaremos cómo se comporta esta estructura en condiciones de concurrencia: su performance utilizando multi-threading y cómo compara con la versión single-threaded, los beneficios y las desventajas que presenta, y cómo se garantiza la correctitud de la inserción de datos en la estructura.Para el análisis mencionado se trabajará sobre cuatro aspectos de la implementación de la estructura, que serán abordados desde un diseño single-threaded y un diseño multi-threaded para luego comparar sus implementaciones bajo distintas circunstancias teóricas.

Finalmente, realizamos experimentos para medir la performance de nuestra implementación. Los datos de entrada utilizados se generaron utilizando dos distribuciones que serán analizadas por separado: uniforme, normal y bernoulli. También se realizará un análisis sobre la respuesta que tiene la estructura en función al tamaño de los datos de entrada. Esta experimentación a su vez permitirá evaluar si a partir de un determinado tamaño de entrada de datos el overhead que implica el uso de multithreads realmente justifica su uso. Por último, se estudiará si es posible encontrar una cota al número de threads que es razonable (en el sentido de costo de manejo versus beneficio en performance) utilizar, y si esa cota guarda relación alguna con la estructura de estudio.

1. Lista Atómica

Para elaborar la estructura de tabla de hash precisamos determinar el método de resolución de colisiones. Como fue mencionado, en este trabajo se utiliza una lista. En particular, se desarrolla una *ListaAtómica* para así poder trabajar de manera paralela posteriormente. En esta sección describimos la implementación del método $insertar(T\ valor)$ de la clase $ListaAtomica$. El mismo tiene la función de agregar el parámetro $valor$ al principio de la lista atómica.

Decimos que una operación es atómica cuando la misma no puede ser interrumpida. Es decir que una vez que comienza a ejecutarse, continuará hasta finalizar. De esta forma si dos o más threads desean ejecutar una operación atómica, dicha ejecución se realizará en forma secuencial, ya que el scheduler del sistema opertivo no podrá realizar una conmutación de tarea que implique desalojarlas, evitando así posibles *race conditions*[1].

Una lista atómica es esencialmente una lista enlazada con la condición adicional de que garantiza la atomicidad de sus operaciones. De esta manera, podemos tener accesos concurrentes a la misma, teniendo la certeza de que no caeremos en una *race condition* donde la lista se vea alterada durante la ejecución de una operación, dando como consecuencia un resultado distinto al esperado.

En términos generales, el funcionamiento de una lista (no atómica) es el siguiente:

1. Crear **nuevoNodo**.
2. Apuntar **nuevoNodo** $\rightarrow$ **siguiente** al nodo que actualmente cumple el rol de cabeza de la lista.
3. Setear **nuevoNodo** como la nueva cabeza.

El problema con esta implementación es que no soporta inserciones concurrentes a la lista. Esta situación se puede ilustrar con el siguiente ejemplo:

Supongamos que tenemos dos threads que quieren insertar un nuevo nodo. El thread A hace los pasos 1 y 2 pero antes de hacer el paso 3 es interrumpido en ese momento por el scheduler del sistema operativo. A continuación el thread B ejecuta el método insertar y finaliza los 3 pasos. Cuando el thread A retoma su ejecución, tiene su **nuevoNodo** apuntando a la antigua cabeza de la lista. Luego, al ejecutar el paso 3 se pierde el registro del último nodo que agregó el thread B, ya que el thread A hizo que la cabeza apuntara a su **nuevoNodo** y el nodo agregado por el thread B quedó sin ningún nodo que lo apunte como siguiente. Este es uno de los tantos problemas que trae la implementación clásica del método insertar.

Nuestra implementación del método hace uso de dos operaciones atómicas provistas por la librería de C++, $atomic\_load$ y $atomic\_compare\_exchange\_weak$, que nos garantizan la atomicidad necesaria para no perder la coherencia de datos, haciendo que el código quede de la siguiente forma:

1. Crear **nuevoNodo**.
2. Apuntar **nuevoNodo** $\rightarrow$ **siguiente** al nodo que actualmente cumple el rol de cabeza de la lista.
3. Si la cabeza actual es la misma que apunta **nuevoNodo** $\rightarrow$ **siguiente**, entonces se setea a **nuevoNodo** como la nueva cabeza. Si no, se vuelven a ejecutar los pasos 2 y 3.

Todo el paso 3 se realiza de forma atómica. Por lo tanto, tenemos la garantía de que si se setea el **nuevoNodo** como la nueva cabeza, es porque la lista no se volvió a modificar desde que seteamos **nuevoNodo** $\rightarrow$ **siguiente** en el paso 2.

[1] Notar que esto puede también evitarse mediante *mutex*, pero la utilización de variables atómicas resulta más eficiente pues no implica generar zonas de exclusión con desalojos y bloqueos.

2. Hash Map Concurrente

En esta sección abordaremos algunos métodos en la clase $HashMapConcurrente$, que implementa la estructura de tabla de hash abierta utilizando listas enlazadas para resolución de colisiones.

2.1. incrementar(string clave)

El método $incrementar(string\ clave)$ de la clase $HashMapConcurrente$ tiene por objeto el incremento de la clave pasada por parámetro en caso de ya existir, o insertar la nueva clave con valor 1 si la clave no existe en el bucket, garantizando a su vez que estas operaciones puedan ser realizadas en forma concurrente en caso de que no haya colisión entre dos claves.

A fin de poder garantizar la concurrencia generamos un algoritmo con la siguiente implementación:

1. Tener un *mutex* para cada bucket de la tabla de hashMap correspondiente a una letra.
2. Se lockea el bucket correspondiente a la clave que se desea agregar evitando así una *race condition*.
3. Se busca en la lista la clave a agregar bajo la siguiente condición:

   Si la clave ya existe entonces se incrementa en 1 su valor asociado.

   Si la clave no existe entonces se inserta la nueva clave con valor asociado fijado en 1.
4. Se finaliza la operación desbloqueando el bucket correspondiente a esa clave.

Cabe mencionar que si bien la inserción de una nueva clave se realiza llamando a la función atómica $insertar$ de la lista implementada en la Sección 1, esto por sí sólo no alcanza para evitar una *race condition*. Para ello analicemos el siguiente ejemplo:

Supongamos que tenemos dos threads que quieren insertar por primera vez una misma clave en la tabla. El thread A busca la clave en la lista correspondiente al bucket que le corresponde. Como la clave aún no fue insertada, la función procede a llamar a la función atómica insertar de la lista. En ese momento, es decir antes de ejecutar al función atómica insertar, es interrumpido por el scheduler y a continuación ejecuta el thread B. Repitiendo el mismo procedimiento, el thread B procederá a insertar la misma clave con valor asociado 1. Una vez que el thread B insertó la clave, el scheduler reanuda la ejecución del thread A, el cual procederá a insertar la misma clave con valor asociado 1.

De esta forma se romperá el invariante de la estructura, ya que en una misma lista habrá dos claves iguales insertadas con valor asociado 1, en lugar de haber una única clave con valor asociado 2. Esto determina claramente un escenario de *race condition*. De lo anterior se puede colegir que la condición de utilizar un semáforo *mutex* para cada bucket es necesaria a fin de asegurar la secuencialización de las operaciones de insertado, evitando así la existencia de *race conditions*.

2.2. claves()

El método $claves()$ de $HashMapConcurrente$ devuelve un vector de strings que representa todas las claves definidas en la estructura. Optamos por una implementación que recorre cada bucket e inserta las claves en el vector de retorno de la función.

Se garantiza que la función esté libre de inanición aún en caso de que se agreguen constantemente claves mientras se está ejecutando $claves()$, basado fuertemente en que $insertar$ agrega claves al principio de la lista. Como las listas son recorridas de principio a fin (y no, por ejemplo, desde el final hasta el principio), cualquier inserción a la lista o bien no modifica la cantidad de elementos (si se incrementa un valor ya existente) o agrega elementos en un punto anterior de la lista (en particular, al principio, en caso de que se defina una nueva clave). Entonces, se asegura que $claves()$ no recorrerá más elementos de los que habían en una lista al comenzar la iteración sobre esa lista, y se garantiza que termine.

Si se agregara la función de borrar claves a la estructura, nuestra implementación no estaría libre de *race conditions*. Para ello deberíamos utilizar los *mutex* creados para cada uno de los buckets de forma tal que se bloquee el bucket que se está recorriendo.

2.3. valor(string clave)

El método $valor(clave)$ de la clase $HashMapConcurrente$ devuelve el valor entero asociado a una clave en la estructura. En la implementación de la clase ese valor es el segundo del par $< clave, valor >$ en la lista atómica asociada al primer caracter de la clave. Para devolverlo recorremos iterativamente la lista atómica correspondiente a $clave$ hasta encontrar el par asociado y retornamos el segundo elemento de ese par.

Esta implementación únicamente involucra lecturas, por lo que en ningún momento queda inconsistente el invariante de la estructura. Por lo tanto, no es necesario (ni utilizamos) un bloqueo. Asumiendo que no se realizan llamados al método $incrementar$ durante la ejecución de $valor$ (la especificación no requiere que se garantice la consistencia en este caso), el par correcto se encontrará eventualmente.

3. Elemento Máximo

La clase $HashMapConcurrente$ cuenta con un método para devolver la clave cuyo valor asociado es el máximo de todas las claves en la estructura. De este método implementamos una versión single-threaded y una versión multi-threaded.

3.1. maximo() single-threaded

La esencia del método $maximo$ en su implementación single-threaded es simple: se itera secuencialmente sobre todos los elementos de todas las listas atómicas de la estructura, guardando cada vez el elemento cuyo valor es máximo. Al finalizar la iteración, devuelve el par $< clave, valor >$ máximo.

El problema principal que surge de esta implementación es que si se llama al método $incrementar$ de forma concurrente con la ejecución de $maximo$, no se puede garantizar que $maximo$ devuelva un resultado consistente. Observemos el siguiente ejemplo.

Se llama la función $maximo$, que inicia la ejecución mirando el primer elemento (se marca con $\rightarrow$ al elemento sobre el cuál está iterando el método $maximo$); se tiene una estructura con 4 claves definidas (simplificamos la escritura para no expandir la estructura entera por razones ilustrativas):

$\rightarrow$ “arbol”: 2
“hoja”: 3
“luz”: 1
“salto”: 1

$maximo =< arbol, 2 >$

A simple vista, la clave de valor máximo es “hoja” y su valor es 3. Como sólo visitamos el primer elemento, el $maximo$ hasta este momento es “arbol”. Consideremos la situación en la que, una vez evaluado el primer elemento, pasamos al segundo, y luego se ejecuta concurrentemente 10 veces el método $incrementar(arbol)$:

“arbol”: 12
$\rightarrow$ “hoja”: 3
“luz”: 1

"salto": 1

$maximo =< hoja, 3 >$

Observamos que, como ya evaluamos la clave "arbol" cuando tenía valor 2, tomamos la clave "hoja" como la máxima con valor 3, a pesar de que claramente "arbol" es actualmente el máximo.

Consideremos el caso en que al iterar al siguiente elemento, se ejecuta concurrentemente 4 veces el método $incrementar(salto)$:

"arbol": 12
"hoja": 3
$\rightarrow$ "luz": 1
"salto": 5

$maximo =< hoja, 3 >$

La clave de valor máximo sigue siendo "arbol". Sin embargo, si finalizamos la ejecución devolvemos el par $< salto, 5 >$:

"arbol": 12
"hoja": 3
"luz": 1
$\rightarrow$ "salto": 5

$maximo =< salto, 5 >$

Observemos: que al principio de la ejecución, el elemento máximo era $< hoja, 3 >$; que al final de la ejecución, el elemento máximo era $< arbol, 12 >$; y que en ningún momento fue $< salto, 5 >$ el elemento máximo.

Para solucionar este problema, bloqueamos la iteración sobre la estructura con el mismo *mutex* que bloquea la escritura en el método $incrementar$. Inmediatamente antes de la iteración, procuramos el *mutex* para todas las listas de la estructura, e inmediatamente después de la iteración las liberamos. Con esto garantizamos que la estructura no sea modificada mientras se evalúa el máximo, y por lo tanto el elemento devuelto es aquel que es máximo en el momento en que se inicia la iteración, y que va a ser el elemento máximo al finalizar la iteración. En el caso anterior, se guardaría el elemento $< hoja, 3 >$, y luego se ejecutarían todas las instancias de $incrementar$ en algún orden.

Sin embargo, un problema que podría aparecer es que se llamara concurrentemente al método $incrementar$ entre el momento en que se llama a la función $maximo$ y el momento en que se procuran los *mutexes*, y entre que se liberan los *mutexes* y se devuelve el resultado. Consideramos que este problema es inevitable sin estructuras adicionales o cuidados del usuario, y que a lo sumo podemos garantizar la consistencia de la respuesta al momento de ser procesada la estructura, i.e. durante la iteración.

3.2. maximoParalelo() multithreaded

El método $maximoParalelo$ toma como argumento la cantidad de threads a utilizar y devuelve el elemento máximo en el $HashMapConcurrente$.

La implementación con multi-threading que diseñamos es muy similar a la versión single-threaded, con algunas adiciones para hacer uso de procesamiento concurrente. La observación importante es la siguiente: como bloqueamos modificaciones a la estructura durante la iteración, no es necesario que el procesamiento de las listas sea secuencial. Podemos repartir el trabajo en múltiples threads y luego de-

volver el máximo encontrado entre todos los threads de ejecución.

El esquema de nuestra implementación es el siguiente:

Primero, definimos un vector auxiliar de pares $< clave, valor >$ cuyo tamaño (en elementos) es la cantidad de listas atómicas en la estructura, y cuyos valores representan el elemento máximo en cada lista. En particular, el vector tiene 26 elementos correspondientes a cada letra.

Cuando se llama a $maximoParalelo$, se procuran los *mutexes* correspondientes a cada lista. Luego, se lanzan la cantidad de threads indicados (dato pasado por parámetro) que procesan las listas y guardan el elemento máximo en el vector auxiliar anteriormente definido. Cuando no hay más listas que procesar, los threads terminan su ejecución y se liberan los *mutexes*. Finalmente, se recorre el vector auxiliar y se devuelve el elemento máximo encontrado en el mismo.

El punto de interés en este esquema es la implementación de threading. Debemos asegurarnos, por un lado, que cada thread procese una lista diferente, y por otro lado, que se procesen todas las listas. Para ello, definimos un entero atómico $lastProcessedList$ que se inicializa en 0 y una función auxiliar $maximoDeLista$, que toma como parámetros la última lista que fue procesada y el vector auxiliar anteriormente definido.

La función auxiliar utiliza el método atómico provisto por la biblioteca estándar de C++ $fetch\_add$ (*get and increase*) sobre $lastProcessedList$ para conseguir el índice de la lista que debe procesar y luego incrementarlo de manera atómica, indicando así que esa lista fue atendida y asegurando que nunca dos threads atiendan la misma lista. Una vez seleccionada la lista a recorrer busca su máximo elemento. Esto se repite en un while loop hasta que $lastProcessedList$ sea mayor a la cantidad de letras, caso en el que la función termina, asegurando que se atiendan todas las listas (ni más ni menos).

Esta implementación funciona para cualquier cantidad de threads, ya que cada thread ejecuta la función y se termina cuando la función retorna, y la función asegura que se van a atender todas las listas. Sin embargo, no tiene sentido utilizar más de 26 threads porque la mínima unidad de trabajo distribuible en este esquema es una lista.

4. Cargar Archivos

La clase $HashMapConcurrente$ debe poder contar las apariciones de palabras en una lista de archivos de texto. Para ello, implementamos dos funciones: $cargarArchivo$, que cuenta las apariciones de palabras en un único archivo con un único thread, y que toma como parámetros un $HashMapConcurrente$ en donde contar las palabras, y un string $filePath$ que indica el archivo a leer; y $cargarMultiplesArchivos$, que utiliza multithreading para contar las apariciones de palabras en múltiples archivos, y toma como parámetros el $hashMap$, la cantidad de threads a utilizar, y un vector de $filePaths$.

4.1. cargarArchivos() single-threaded

Se utiliza la biblioteca estándar de C++ para abrir el archivo correspondiente a $filePath$ en un *stream*, leer cada palabra en el *stream*, e incrementarla en el $hashMap$. No son necesarias consideraciones especiales: como $incrementar$ es atómico y el proceso es secuencial y single-threaded, el *stream* se avanza por un único thread. Luego, se cierra el archivo y se retorna.

4.2. cargarArchivosParalelo() multithreaded

Nuestra implementación con multithreading de la función $cargarArchivosParalelo$ tiene un esquema similar al método $maximoParalelo$ presentado en la Sección 3.2: utilizamos una variable atómica

$lastProcessedFile$ y una función auxiliar $cargarArchivosParalelo$ que la toma como parámetro.

Se genera la cantidad de threads indicada por argumento y a cada thread se le asigna la lectura de un archivo distinto (determinado por $lastProcessedFile$) a través de la función $cargarArchivo$. Cada thread continúa con el siguiente archivo hasta que no haya ningún archivo más para ser procesado y luego termina.

La decisión principal que tomamos fue la de la unidad de trabajo distribuible. Elegimos que cada thread atienda un archivo distinto (i.e. que un mismo archivo no sea atendido por dos threads distintos) para evitar que múltiples threads manipulen el mismo stream (ya que no queremos que se cuente la misma palabra más de una vez). En particular, esto significa que se puede utilizar una cantidad arbitraria de threads, pero que no es razonable utilizar una cantidad de threads mayor a la cantidad de archivos.

Finalmente, no son necesarias más consideraciones adicionales para la sincronización. La función asegura que los threads se encarguen de tareas distintas, y de que se encarguen de todas las tareas, y la función $cargarArchivo$ asegura que la tarea se realice correctamente.

5. Parallelism idleness

Una métrica interesante cuando hablamos de ejecución concurrente es el nivel de ***parallelism idleness***, el cual definimos como el tiempo en que los threads se encuentran bloqueados o sin hacer trabajo útil.

Primero analicemos los posibles escenarios en que un thread se encuentra en estado $idle$:

- El thread en ejecución debe esperar a un dispositivo de E/S por lo que cambia al estado *BLOCKED*. El tiempo que permanece en este estado y en *READY* hasta volver a retomar su ejecución es considerado $idle$.
- Finaliza el quantum del thread en ejecución y procede a ser desalojado por el scheduler. El tiempo que transcurre desde dicho desalojo hasta su retorno al estado *RUNNING* es considerado $idle$.
- El thread en ejecución se encuentra con un lock ocupado y debe quedarse esperando hasta que sea liberado y le permita el ingreso a la sección crítica. El tiempo transcurrido hasta que puede ingresar a dicha sección también debe considerarse como $idle$.

En el tercer caso, podemos medir el tiempo transcurrido utilizando REALTIME-CLOCKS antes y después de cada lock en el código concurrente. Luego basta con tomar la diferencia entre ellos para saber cuánto tiempo el thread se encontró en estado $idle$.
El segundo caso es un poco más complejo: necesitaríamos una manera de saber cuándo un thread es desalojado y cuando retoma su ejecución. A nivel usuario esto no es posible y requiere de una modificación en el scheduler para que guarde un timestamp cada vez que un thread cambia de estado (indicando de qué thread se trata).

- timestamp cada vez que el thread cambia al estado running
- timestamp cada vez que lo dealoja
- timestamp cada vez que cambia a estado blocked.

Notemos que estos escenarios pueden combinarse. Por ejemplo, un thread esperando a que se libere un lock puede ser desalojado por el scheduler. En este caso basta con quedarse con la diferencia de tiempo mayor para obtener el nivel de idleness.

Conociendo los tiempos en que un thread entró o salió del estado running y los que se encontraba esperando un lock y, por supuesto, los tiempos totales de ejecución de nuestros algoritmos, podremos evaluar el parallelism idleness de nuestro código.

6. Experimentación

6.1. Metodología experimental

El objetivo de la experimentación es lograr comparar el desempeño de los distintos algoritmos para diferentes parámetros, contrastando sus tiempos de ejecución. Para ello, generamos un conjunto de *datasets* de dimensión consdierable ($\#dataset \approx 10000$) con diferentes tipos de distribución (Bernoulli, Normal y Uniforme). Luego los algoritmos fueron ejecutados para los distintos *datasets* 20 veces, de manera tal que luego se pudieran obtener resultados estadísticos libres de sesgos.

6.1.1. Especificaciones técnicas

Para estandarizar los resultados de la experimentación utilizamos una computadora con las siguientes especificaciones técnicas:

- 8th Gen Intel(R) Core (TM) i5-8400 [2]
  - 6 cores
  - 6 threads (1 thread per core)
- Disco Sólido SSD
- 16Gb RAM

6.1.2. Datasets

Como fue mencionado anteriormente, generamos un conjunto de *datasets* para evaluar el comportamiento y eficacia de los algoritmos desarrollados. Para ello, utilizamos tres *datasets* de 10000 palabras y con distinto tipo de distribución.

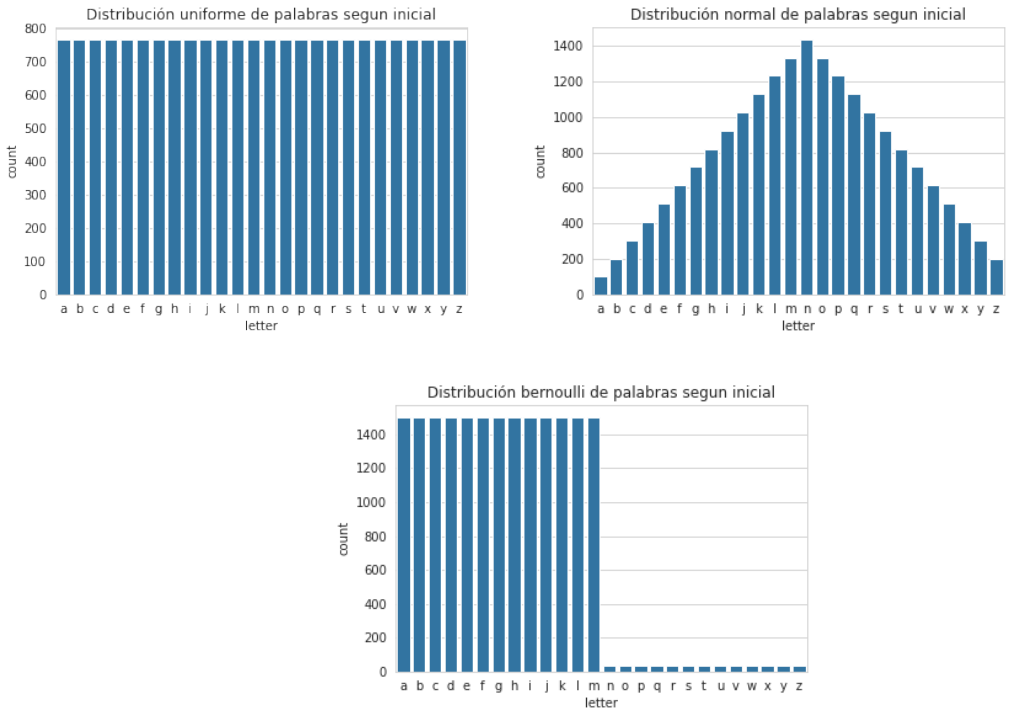

Figura 1: Datasets graficados según distribución, mostrando la cantidad de palabras **distintas** que contiene según cada inicial.

[2] https://www.intel.com/content/www/us/en/products/sku/140644/intel-core-i58400-processor-9m-cache-up-to-4-00-ghz-includes specifications.html

A su vez, para evaluar el algoritmo de carga de archivos utilizamos 60 archivos con distribución uniforme, 30 de ellos con sus palabras ordenadas alfabéticamente según su inicial y los restantes con su contenido ordenado de forma aleatoria.

6.1.3. Pasos para reproducir los experimentos

Todo lo necesario para la experimentación se encontrará dentro del directorio `Exp/`, salvo el ejecutable cuyo path es `codigo/build/ContarPalabras`. Dentro del directorio de experimentación se encuentran dos jupyter-notebook:

- generador-palabras.ipynb
- experimentos.ipynb

El primero genera los datasets necesarios para correr los experimentos. Los mismos ya se encuentran creados dentro del dir `Exp/data/`. El segundo notebook es el encargado de correr dichos experimentos.

1. Generar el ejecutable:

   Abrir una terminal sobre el directorio `codigo/`.

   Ejecutar el comando `make`.

2. Generar un kernel de jupyter-notebook y abrir experimentos.ipynb
3. Ejecutar **en orden** todas las celdas.

6.2. Experimento 1: Multithreading en máximo paralelo

Lo primero a analizar fue la relación entre la eficiencia temporal del algortimo $maximoParalelo$ y la cantidad de threads con los que se ejecuta. Para ello, utilizamos los tres datasets de 10000 palabras con los 3 tipos de distribución mencionados anteriormente.

6.2.1. Hipótesis

Creemos que el tiempo de ejecución del algoritmo irá disminuyendo a medida que aumente la cantidad de threads. Al tener sólo 26 listas que recorrer (una por cada letra o bucket), asignarle más threads que la cantidad de listas no traerá ningún beneficio. Incluso podría llegar a ser perjudicial considerando el costo de crear cada uno. Siendo así, tampoco esperamos una mejoría en la performance al utilizar más threads que los soportados por el hardware, ya que la paralelización está acotada por la cantidad de procesos que puede ejecutar en simultáneo el sistema.
A su vez, si tratamos con listas muy pequeñas que se recorren rápidamente, es probable que no notemos diferencia relevante al agregar hilos de ejecución. Bajo este supuesto, creemos que la distribución de los datos puede afectar la eficiencia del algoritmo. En este sentido, el algoritmo sobre una distribución Bernoulli debería tardar más que sobre una distribución Uniforme o una Normal.

6.2.2. Resultados y discusión

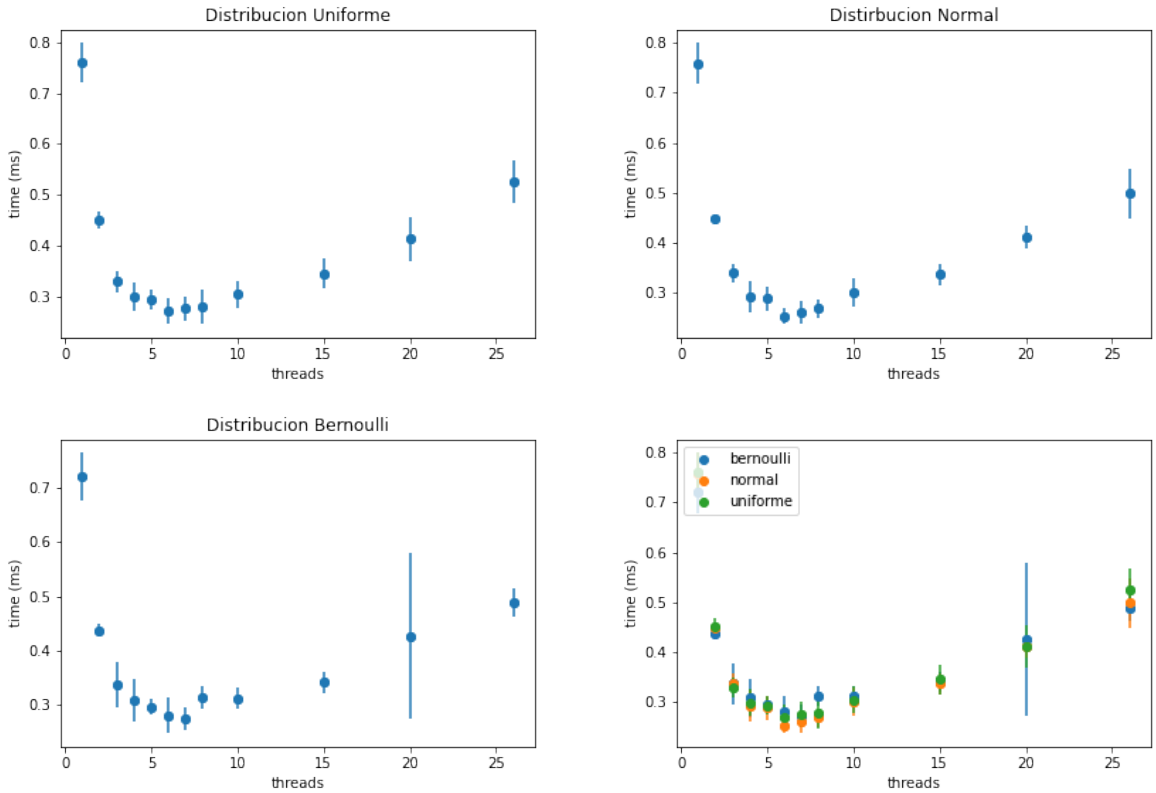

Figura 2: Variación de los tiempos de ejecución del algoritmo *maximoParalelo* según cantidad de threads utilizados para datasets de 10000 palabras con distribuciones Uniforme, Normal y Bernoulli y la comparación entre ellas. Se realizaron 20 ejecuciones para cada combinación *thread-dataset* y se grafican la media y desvío estándar de cada una.

De la Figura 2 podemos observar que el tiempo disminuye a medida que agregamos threads pero sólo hasta que se llega al limite de multithreading de la computadora utilizada. Superando ese límite vemos que sólo se incrementa, es decir que incluso resulta perjudicial para el algoritmo superar esta cota. A su vez, notamos que la distribución de los elementos no tuvo impacto relevante en la performance del algoritmo. Esto puede deberse a que al evitar el lockeo de threads para calcular el máximo paralelo por la utilización del *lastProcessedList* (ver 3.2), cada thread solo es interrumpido por el *clock* y el trabajo en paralelo se da en general de manera equitativa y contínua.

6.3. Experimento 2: Multithreading en cargar múltiples archivos

Al igual que en el experimento anterior, queremos comparar el tiempo de ejecución del algoritmo *cargarMultiplesArchivos* según la cantidad de threads asignadas. Para ello, utilizaremos el *dataset* de 60 archivos con distribución uniforme antes mencionados.

6.3.1. Hipótesis

En cuanto a la relación tiempo - cantidad de threads suponemos que tendremos resultados similares al experimento anterior.

Para los archivos ordenados alfabéticamente creemos que el tiempo de ejecución será mayor respecto a los randomizados. Esto es porque las primeras palabras de cada archivo corresponderán al mismo bucket, seguidas de las correspondientes al siguiente bucket y así sucesivamente. De esta manera, el

método *incrementar* tendrá muchas llamadas concurrentes para un mismo bucket, el cuál será bloqueado por su *mutex* asociado (ver 2.1) y todos los threads se verán afectados por dicho bloqueo.

6.3.2. Resultados y discusión

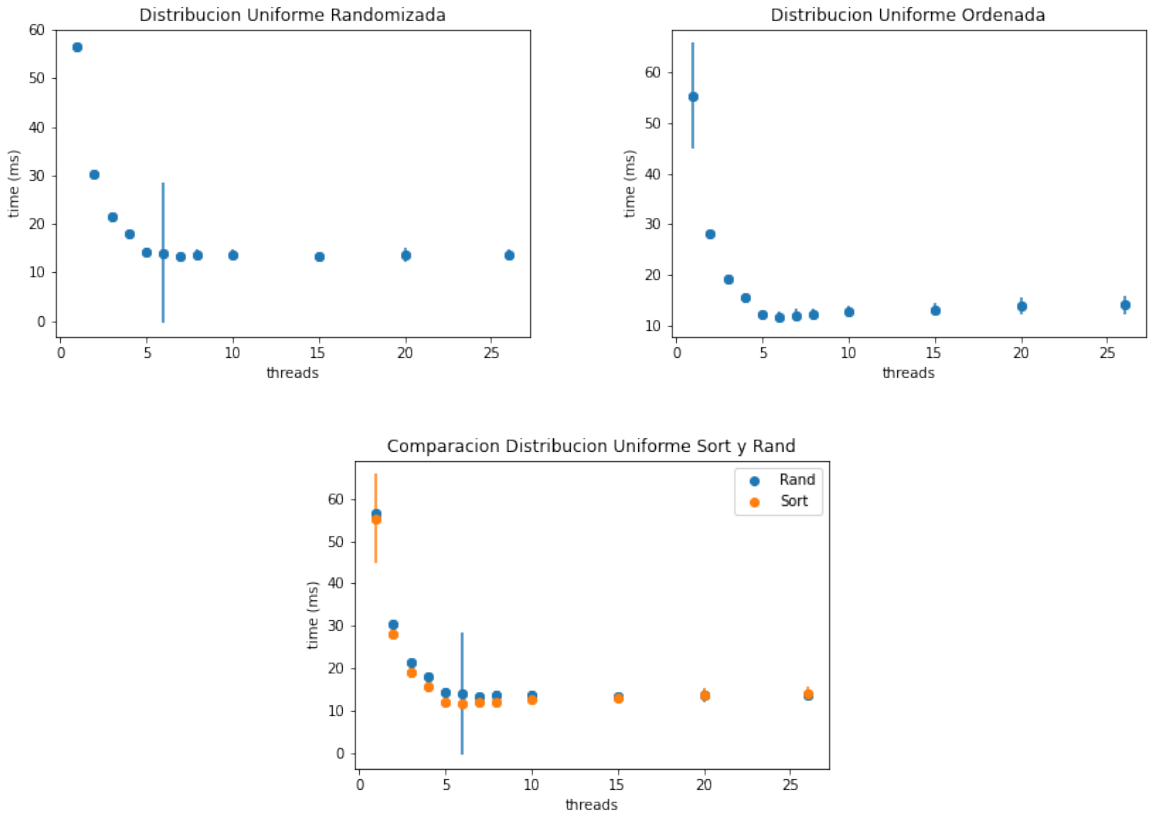

Figura 3: Variación de los tiempos de ejecución del algoritmo *cargarArchivosParalelo* según cantidad de threads utilizados para 60 archivos de 150 palabras con distribución uniforme, 30 de ellos ordenados alfabéticamente y los otros 30 con entradas randomizadas. Se realizaron 20 ejecuciones para cada combinación *thread-dataset* y se grafican la media y desvío estándar de cada una.

Anulando nuestra hipótesis, la Figura 3 muestra que se obtuvieron mejores tiempos para los archivos ordenados. Al observar estos resultados creemos que esto podría deberse a la memoria *cache*, pues al estar siempre accediendo al mismo bucket, es probable que haya muchos más *cache hit* que con archivos randomizados, donde estamos queriendo acceder a buckets distintos constantemente.

Respecto a la relación de tiempo y cantidad de threads, podemos observar que a diferencia del experimento anterior, al superar el límite de threads de nuestra computadora no crece el tiempo, sino que se mantiene constante. Esto puede deberse a la gran cantidad de operaciones de *E/S*. Al estar constantemente accediendo a memoria, los procesos se bloquearán esperando los datos solicitados, dejando lugar a otros threads que estén listos para ser ejecutados, es decir que ya cuenten con los resultados de su pedido a memoria. A su vez, esta gran cantidad de desalojos podría hacer que disminuya considerablemente la cantidad de *cache hits* para los archivos ordenados, explicando la desaparición de su ventaja al superar la cota de threads soportados por el hardware.

7. Conclusiones y trabajo futuro

Nuestra experimentación da evidencia a la utilidad y el poder de la paralelización de procesamiento y el *multi-threading*. Observamos que a pesar de agregar overhead, la paralelización en nuestro caso ofreció incrementos de *performance* hasta cuanto es posible explotar las capacidades de un procesador.

Sin embargo, observamos que la paralelización de un proceso no es un ejercicio trivial. Por un lado, la cantidad de threads que puede utilizar un proceso antes de empezar a perder performance depende de múltiples factores, a veces difíciles de controlar: las estructuras de datos utilizadas (en nuestro caso, un limitante para $maximoParalelo$), la implementación de los algoritmos (en otras palabras, el esquema de bloqueo utilizado), el tamaño de las entradas (como en el caso de $cargarMultiplesArchivos$), y la máquina en la que se ejecuta (que limita la cantidad de cores y threads utilizables). Hay que tener en consideración todos estos factores al decidir si un esquema de paralelización es beneficioso para un sistema o proceso dado, y para los tipos de estados que manejará comúnmente ese sistema.

Por otro lado, observamos que puede ser difícil razonar abstractamente sobre esquemas de paralelización y su impacto en la performance en la práctica. En nuestro caso, los resultados obtenidos fueron inesperados basado en nuestro análisis, porque fallamos en considerar el efecto de la cache en la performance del sistema.

A pesar de estas consideraciones, observamos que el multithreading es una herramienta poderosa, y que analizar la paralelizabilidad de un sistema es un ejercicio beneficioso, si bien dificultoso.

Consideramos que un interesante ángulo de estudio futuro es la exploración de los tipos de distribuciones de datos que se benefician particularmente del paralelismo, y en qué medida es por propiedad de las distribuciones en sí o por las propiedades del sistema.