



Trabajo Práctico 1

Hashmap Concurrente

Sistemas Operativos

Integrante	LU	Correo Electrónico
Tomás Donzis	443/20	donzis.tomy@gmail.com
Teo Kohan	385/20	teo.kohan@gmail.com
Fabrizio Gabriel Prida	31/20	pridafabrizio@gmail.com
Martín Santesteban	397/20	martin.p.santesteban@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

<https://exactas.uba.ar>



Índice

1	Introducción	3
2	Desarrollo	3
2.1	Ejercicio 1	3
2.2	Ejercicio 2	5
2.3	Ejercicio 3	6
2.4	Ejercicio 4	8
3	Experimentación	8
3.1	Ejercicio 5	8
3.1.1	Hipótesis	8
3.1.2	Tiempo de cómputo en función a cantidad de threads.	8
3.1.3	Tiempo de cómputo en función a cantidad de buckets y threads.	9
3.1.4	Tiempo de cómputo en función del incremento de un bucket	11
3.1.5	Tiempo de cómputo en función de la cantidad de colisiones entre sets de datos	12
4	Conclusiones	14

1 Introducción

El objetivo del presente trabajo es estudiar la gestión y efectos de la concurrencia. Para ello, se analizará la estructura `HASHMAPCONCURRENTE`, que consiste en una tabla de *hash* basada en listas enlazadas atómicas para almacenar los elementos de cada *bucket*.

En particular, analizaremos cómo implementar ciertas funcionalidades sin incurrir en condiciones de carrera. Para esto utilizaremos semáforos y objetos atómicos, procurando no generar *deadlocks* e intentando minimizar la contención.

Además, compararemos la *performance* de la estructura implementada en base al nivel de paralelismo y a la cantidad de datos procesados y su distribución.

2 Desarrollo

2.1 Ejercicio 1

Una lista enlazada de una vía es una estructura formada por nodos donde cada nodo contiene un puntero al siguiente. En una lista enlazada clásica la inserción de un nuevo nodo se realiza en tres pasos

- 1: **function** INSERTAR(valor : T) $\rightarrow \emptyset$
- 2: nuevo \leftarrow NODO(T)
- 3: nuevo.siguiente \leftarrow cabeza
- 4: cabeza \leftarrow nuevo

En un contexto de un único *thread* esta implementación es correcta.

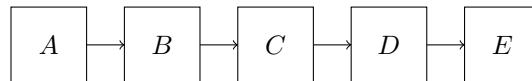


FIGURA 1: Representación de una Lista Enlazada de cinco nodos.

Supongamos ahora que estamos ejecutando un programa que manipula una lista de este tipo con dos *threads* donde cada *thread* quiere insertar un nodo en la lista enlazada.




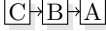
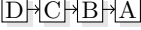
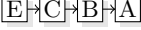
Thread 1	Thread 2	Lista Enlazada
nuevo \leftarrow NODO(D)	—	
nuevo.siguiete \leftarrow cabeza	—	
—	nuevo \leftarrow NODO(E)	
—	nuevo.siguiete \leftarrow cabeza	
cabeza \leftarrow nuevo	—	
—	cabeza \leftarrow nuevo	

FIGURA 2: Interleaving *posible* para dos threads y una lista enlazada compartida.

Como se ve en la Figura 5 este *interleaving* “pierde” un nodo. Ambos memorizan al nodo C como el nodo cabeza antes de insertar su propio nodo.

Para evitar estos casos necesitamos que la operación de asignación sea atómica, es decir, que la acción de obtener la cabeza y asignar el nuevo nodo apuntando a ella sea indivisible.

Bajo este concepto la lista es atómica ya que sus operaciones críticas no presentan condiciones de carrera. Un programa que utilice esta lista atómica queda protegido en incurrir en condiciones de carrera en lo que respecta al uso de esta estructura. Aún así puede incurrir en estas por otros medios si no se utilizan las estructuras de control necesarias.

Para lograr la atomicidad de la lista, nuestra implementación hace uso de la función provista por la biblioteca `atomic` de C++, `atomic_compare_exchange_weak`.

Un posible pseudocódigo de la misma es el de la Figura 3

```

1: function ATOMICCOMPAREEXCHANGE(obj :  $T$ , esperado :  $T$ , deseado :  $T$ )  $\rightarrow$  {0, 1}
2:   if obj = esperado then
3:     obj  $\leftarrow$  deseado
4:     return 1
5:   else
6:     esperado  $\leftarrow$  obj
7:     return 0

```

FIGURA 3: Pseudocódigo de la instrucción ATOMICCOMPAREEXCHANGE.

INSERTAR realiza esta operación en bucle con los siguientes parámetros

```

1: while  $\neg$ ATOMICCOMPAREEXCHANGE(cabeza, nuevo.siguiete, nuevo) do
2:   skip

```

Por lo tanto al realizar la nueva asignación se asegura de que el nodo *siguiete* en *nuevo* sea efectivamente la *cabeza* actual, caso contrario le asigna *cabeza* a *siguiete*.

2.2 Ejercicio 2

La implementación de HASHMAP, utiliza una función de *hash* ϕ tal que dos elementos pertenecen al mismo *hash-index* si solo si coinciden sus iniciales.

Si se utilizaran listas enlazadas no atómicas se puede perder la inserción de algunas claves como se ve en el inciso 5. Es por esto que en esta implementación de HASHMAPCONCURRENTE, se utilizan Listas Atómicas.

El método INCREMENTAR agrega una clave a la tabla o en caso de que esta tupla ya exista, incrementa en uno el valor asociado a la clave. Donde agregar una clave a la tabla implica agregar la tupla indicada a su lista atómica asociada.

Que la lista sea atómica no resuelve todos los posibles problemas de concurrencia. Si dos *threads* insertan la misma clave concurrentemente, entonces estos no serán capaces de detectar la presencia previa de la clave. Los *threads* deben coordinarse en la inserción para no agregar una misma clave más de una vez.

Una posible solución, es tener un semáforo para la inserción de nuevas claves, independientemente de la misma. De esta forma un único *thread* puede modificar el HASHMAP concurrentemente. Sin embargo, esto genera contención innecesaria; los threads deberán esperarse entre sí, por mas que deseen modificar distintas listas dentro de la tabla. Para evitar la espera, podemos asegurar la exclusión mutua por lista en vez de hacerlo globalmente. Contaremos con un arreglo de semáforos, uno por cada *bucket* en la tabla de *hash*.

```

1: function INCREMENTAR(clave : STRING)
2:   llave  $\leftarrow \phi(c)$ 
3:   WAIT(semáforos[llave])
4:   for  $\langle c, v \rangle \in \text{semáforos}[llave]$  do
5:     if  $c = \text{clave}$  then
6:        $v \leftarrow v + 1$ 
7:       SIGNAL(semáforos[llave])
8:     return
9:   tabla[llave].insertar( $\langle \text{clave}, 1 \rangle$ )
10:  SIGNAL(semáforos[llave])

```

FIGURA 4: Pseudocódigo de la implementación de INCREMENTAR.

Nuestra implementación de INCREMENTAR 4 otorga exclusión mutua para cada acceso a una lista atómica.

El método CLAVES devuelve todas las claves insertadas en el HASHMAP. Como no modifica la estructura no contiene condiciones de carrera ni contención.

El método VALOR toma como parámetro una clave, y devuelve su valor asociado. Como no modifica la estructura no contiene condiciones de carrera ni contención.

2.3 Ejercicio 3

El método MÁXIMO devuelve la tupla $\langle \text{clave}, \text{valor} \rangle$ con el valor máximo de la tabla.

Sin embargo, en el caso que se ejecute concurrentemente MÁXIMO e INCREMENTAR se pueden generar inconsistencias. Hay escenarios en los que la concurrencia entre ambos métodos hace que MÁXIMO devuelva una tupla que en ningún momento fue la tupla máxima en la tabla.

Siguiendo la Figura 5, uno de los posibles escenarios donde surge un problema es cuando MÁXIMO recorre las claves hasta cierto punto, y llegado ese instante pasa lo siguiente:

1. se recorre el HASHMAP hasta una clave B tal que el máximo parcial hasta B es B y el máximo “real” de la tabla es un valor “lejano” D .
2. se incrementa sucesivamente el valor de una clave A que MÁXIMO ya recorrió, de forma tal que pasa a ser la clave con máximo valor en el HASHMAP superando a D .
3. se incrementa el valor de una clave C que todavía no fue evaluada, tal que su valor resulte mayor al máximo parcial de MÁXIMO y D , pero menor al máximo “real” de la tabla, A .







Thread 1	Thread 2	Max	Hashmap
$\text{max} \leftarrow \text{max} \{ \text{max}, A \}$	—	D	
$\text{max} \leftarrow \text{max} \{ \text{max}, B \}$	—	D	
—	INCREMENTAR($A, 5$)	A	
—	INCREMENTAR($C, 2$)	A	
$\text{max} \leftarrow \text{max} \{ \text{max}, C \}$	—	A	
$\text{max} \leftarrow \text{max} \{ \text{max}, D \}$	—	A	

FIGURA 5: Interleaving posible que retorna un máximo que nunca fue máximo real del hashmap. En la columna Max el máximo “real” del HASHMAP en este instante, En la columna Hashmap la barra oscura representa la clave seleccionada como máximo por el algoritmo y el pequeño triángulo apunta a la clave siendo evaluada en este instante.

Cuando se reanude la ejecución de MÁXIMO, el cambio en la clave A no será notado, y se considerará que C es la clave con valor máximo (cuando en realidad nunca fue un máximo real de la tabla).

Para evitar estas inconsistencias, en nuestra implementación de MÁXIMO se hace uso del arreglo de semáforos global definido anteriormente. Antes de iterar sobre todos los *buckets* de la tabla, se ejecuta un WAIT por semáforo, bloqueando el acceso a todos los demás *threads*.

```

1: function MÁXIMO(Hm : HASHMAP)
2:   for semáforo  $\in$  sémáforos do
3:     WAIT(semáforo)
4:   max  $\leftarrow \langle \emptyset, 0 \rangle$ 
5:   for bucket  $\in$  Hm do
6:     for  $\langle$ clave, valor $\rangle \in$  bucket do
7:       if valor > max.valor then
8:         max  $\leftarrow \langle$ clave, valor $\rangle$ 
9:     SIGNAL(semáforos[bucket])
10:  return max

```

FIGURA 6: Pseudocódigo de nuestra implementación de MÁXIMO.

Como se ve en la Figura 6 luego de obtener la exclusión mutua se busca el máximo secuencialmente liberando el *bucket* cuando terminamos de procesarlo a devolver para finalmente hacer los SIGNAL pertinentes.

Además de la función MÁXIMO, se implementó una alternativa utilizando múltiples *threads* que se ejecutan concurrentemente, MÁXIMOPARALELO. Esta toma un valor representando la cantidad de *threads* a utilizar.

Para obtener el máximo utilizando una cantidad fija n de *threads*, se utilizó la siguiente estrategia: cada thread se encuentra constantemente iterando sobre algún *bucket* del HASHMAP, y estos nunca son recorridos más de una vez.

Se utilizará un entero atómico en memoria compartida, *vanguard*, que indicará el índice del primer *bucket* que no fue recorrido. Una vez inicializado cada *thread*, lo primero que hará será ejecutar `atomic_fetch_add(vanguard)` Figura 7. De esta forma obtiene el índice del *bucket* que le corresponde recorrer e incrementa el valor de *vanguard* atómicamente. Se inicializa un arreglo de 26 posiciones para almacenar las tupla máximas de cada *bucket* (la i -ésima posición almacena a la tupla máxima del i -ésimo *bucket*). Al terminar de recorrer un *bucket* el *thread* pide otro índice, y vuelve a repetir el proceso hasta que el índice devuelto no se corresponda con un *bucket*. Al utilizar un entero atómico junto a una operación atómica, todo *bucket* se recorre exactamente una vez, consecuentemente la función no presenta condiciones de carrera.

```

1: function ATOMICFETCHADD(valor :  $\mathbb{Z}$ )  $\rightarrow \mathbb{Z}$ 
2:   anterior  $\leftarrow$  valor
3:   valor  $\leftarrow$  valor + 1
4:   return anterior

```

FIGURA 7: Pseudocódigo de la instrucción ATOMICFETCHADD.

Una vez finalizan todos los *threads* se retorna el máximo valor del arreglo que contiene los máximos de cada *bucket* y se desbloquean todos los *buckets*.

2.4 Ejercicio 4

Para llenar un HASHMAP de claves y testear su comportamiento, insertamos a través de *file-streams* cada clave. Para esto completamos la función `cargarArchivo` que justamente toma como parámetro el `path` relativo hacia el archivo y crea un HASHMAP con su contenido.

Para `cargarArchivo` no necesitamos tener en cuenta ningún problema de sincronización, ya que invocamos a la función INCREMENTAR que garantiza la contención en el caso en que múltiples *threads* deban incrementar claves con la misma inicial, el cuál sería el mayor problema de concurrencia ante el cuál podríamos enfrentarnos en este escenario.

Por este motivo al implementar la función `cargarMultiplesArchivos`, cada *thread* ejecuta la función `cargarArchivo` con el siguiente *file-path* a procesar. Esta noción de siguiente la damos utilizando un entero atómico y la función `atomic_fetch_add` al igual que en el Ejercicio 3b (2.3). Determinando así el siguiente índice del vector de *file-paths* a cargar en el HASHMAP.

3 Experimentación

3.1 Ejercicio 5

3.1.1 Hipótesis

1. A mayor grado de paralelización menor tiempo de cómputo.
2. Sean A_1, A_2, \dots, A_n multiconjuntos de palabras e I_i el multiconjunto de iniciales correspondientes a A_i para todo $1 \leq i \leq n$. Sea $I_\Omega = \bigcap_{i=1}^n I_i$ y sean α y $\beta \#I_\Omega$ y el promedio de los cardinales de los A_i , respectivamente, entonces $\frac{\alpha}{\beta}$ es un predictor de las colisiones entre los conjuntos A_1, A_2, \dots, A_n .
3. Sea b^* el *bucket* con mayor cantidad de claves en un HASHMAP y $|b^*| \gg |b|$ para todo b *bucket* del HASHMAP $b^* \neq b$. Entonces la diferencia de *performance* entre la versión *single-threaded* y *multi-threaded* es negligible. Si la varianza de $|b|$ para todo b *bucket* del HASHMAP es chica, entonces la *performance* del algoritmo *multi-threaded* es mayor a la del algoritmo *single-threaded*.
4. Sea p la cantidad de *threads* y b la cantidad de *buckets* no vacíos en el HASHMAP. Sean todos los *buckets* de aproximadamente la misma longitud, entonces la *performance* en el tiempo de cómputo de MÁXIMOPARALELO para p threads está dada por $\lceil b/p \rceil$. i.e. la cantidad de “tandas” de *buckets* a procesar.

3.1.2 Tiempo de cómputo en función a cantidad de threads.

El objetivo de este experimento es exponer el tiempo de cómputo para la ejecución de `cargarMultiplesArchivos` y `maximoParalelo` variando la cantidad de *threads*. Se tomó un subconjunto del *dataset* de las 333.333 palabras más comunes de Kaggle[1], y se midió el tiempo de cómputo de ambos programas en función a la cantidad de *threads* utilizados.

Lo esperado es que a medida que aumenta la cantidad de *threads*, los tiempos de cómputo se vuelven menores.

Resultados

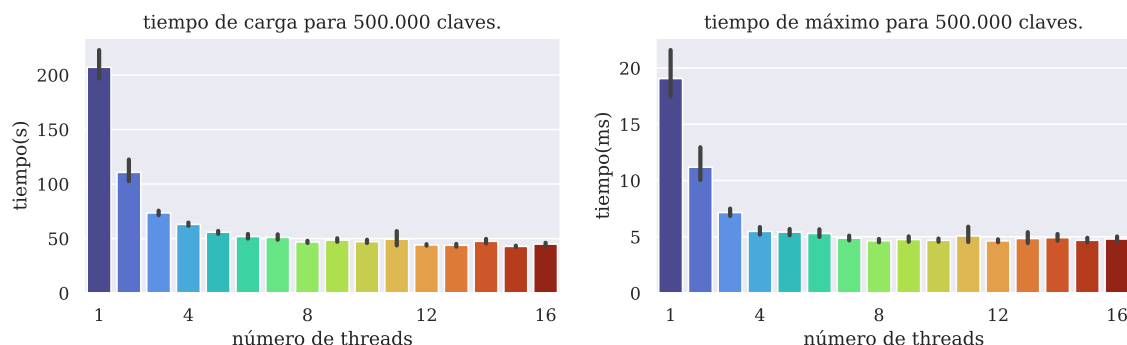


FIGURA 8: A la izquierda tiempos de `cargarMultiplesArchivos` a la derecha de `maximoParalelo`.

En la Figura 8 se puede ver como el tiempo requerido para la ejecución de los algoritmos baja a medida que aumenta la cantidad de *threads*. Se observa que para 1 a 4 *threads* los tiempos se reducen significativamente. Utilizar más de 6 ~ *threads* no aporta mejoras significativas. El estancamiento para cantidades mayores de *threads* suponemos se debe a que cada *thread* tiene una menor carga de trabajo, y aumenta la contención al haber tantos trabajadores operando sobre una misma estructura de datos. Además agregar el *overhead* generado por lanzar y terminar los *threads*.

Por otra parte, comparando los dos gráficos se puede ver que los tiempos de ejecución de la función `maximoParalelo` son mucho más bajos que los de `cargarMultiplesArchivos`. Esto era esperable, ya que `maximoParalelo` recorre una única vez todo el HASHMAP de forma paralela mientras que `cargarMultiplesArchivos` debe recorrer un *bucket* por cada palabra nueva a insertar, y los tamaños de los *buckets* van en aumento a medida que se insertan palabras.

3.1.3 Tiempo de cómputo en función a cantidad de buckets y threads.

Este experimento analiza cómo afecta al tiempo de cómputo de `maximoParalelo` la elección de la cantidad de *threads* utilizados y el número de *buckets* en el HASHMAPCONCURRENTE.

La idea inicial es que al asumir que todos los *buckets* tienen un tamaño aproximadamente igual, cada *thread* los procesará en un tiempo t muy similar. En consecuencia, si se tienen p *threads*, la diferencia en el tiempo de procesamiento al considerar cualquier cantidad de *buckets* en el rango $p \cdot i + 1, p \cdot i + 2, \dots, p \cdot (i + 1)$ será despreciable. En otras palabras, la *performance* en el tiempo de cómputo de `máximoParalelo` está dada por $\lceil b/p \rceil$.

Concretamente, para este experimento vamos a considerar *buckets* con 500000 datos. Variaremos la cantidad de *buckets* considerados así como también el número de *threads* empleados para la

ejecución.

Resultados

La Figura 9 nos permite ver el tiempo de ejecución de MÁXIMOPARALELO en función de la cantidad de *buckets* considerados en el HASHMAPCONCURRENTE, utilizando dos *threads*. Se puede observar claramente una “escalera”, lo cual confirma nuestras hipótesis: dado $i \in \mathbb{N}$, los tiempos de ejecución para $2 \cdot i - 1$ y $2 \cdot i$ buckets son muy parecidos, mientras que entre los tiempos para $2 \cdot i$ y $2 \cdot i + 1$ se puede observar un salto. En otras palabras, los tiempos al considerar (1, 2) buckets es muy similar, y lo mismo sucede tomando (3, 4), (5, 6), etc. Por otra parte, vemos que hay una mayor diferencia al tomar (2, 3) buckets, y de igual forma sucede para (4, 5), (6, 7), etc.

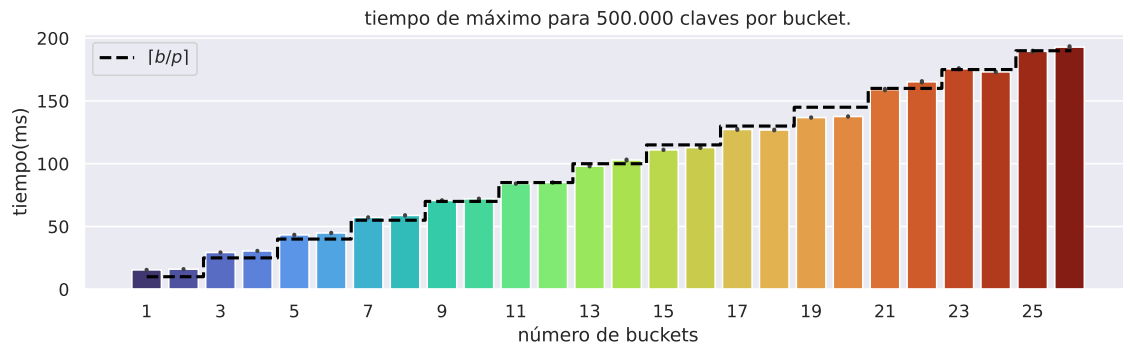


FIGURA 9: *Tiempos de MÁXIMOPARALELO en función de la cantidad de buckets. Cada bucket contiene 500000 datos y 2048 muestras.*

Por ahora tenemos que, al tomar dos *buckets*, se cumple lo que imaginábamos. A continuación procedemos a realizar la misma experimentación, pero con distinta cantidad de *threads*.

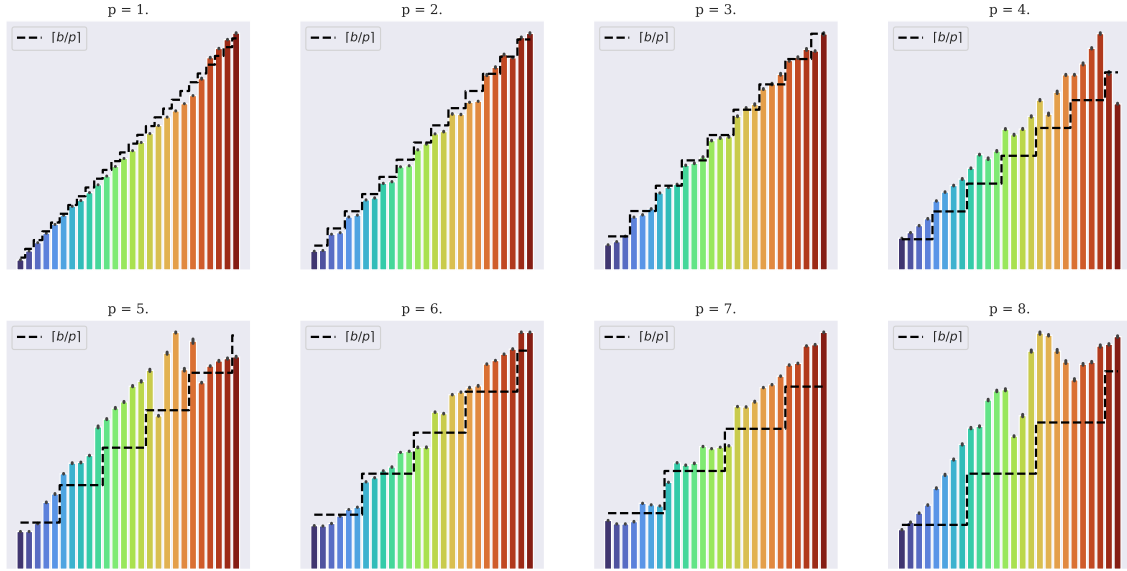


FIGURA 10: Ajustes de MÁXIMOPARALELO en función de la cantidad de buckets para p threads $1 \leq p \leq 8$ a una función $O(\lceil b/p \rceil)$ con buckets de 500.000 claves y 2048 muestras.

En la Figura 10 observamos un comportamiento similar a lo que obtuvimos con dos *threads* para 1, 2 y 3. En mayor o menor medida, se cumple que la performance de `maximoParalelo` es $\lceil b/p \rceil$, donde b es la cantidad de *buckets* y p es el número de *threads*. A medida que aumentamos la cantidad de *threads* esta relación se distorsiona más, en gran medida por la varianza de los datos y que se requerirían cada vez mayores cantidades de samples para obtener resultados más estables.

3.1.4 Tiempo de cómputo en función del incremento de un bucket

El objetivo de este experimento es analizar el rendimiento de MAXIMOPARALELO teniendo un *bucket* b^* tal que $|b^*| \gg |b|$ para todo $b \neq |b^*|$.

Tengan todos los *buckets* un tamaño inicial K y se agreguen nuevas palabras de una clave específica, esperamos ver que, a medida que se incrementa el tamaño del *bucket* a desbalancear, las diferencias entre los tiempos de cómputo de MAXIMOPARALELO utilizando cada vez mas *threads* se va achicando. Al tener mas de un único *thread*, uno de ellos se asignará al *bucket* desbalanceado, mientras que los restantes procesarán los demás *buckets* de menor tamaño en un tiempo mucho menor, y de esta manera el tiempo de cómputo estará dado por lo que se tarde en procesar el *bucket* grande. Por otra parte, al tener un único *thread*, el tiempo que tarde estará dado por tener que procesar el *bucket* desbalanceado, sumado al tiempo de procesar todos los restantes.

Con esta idea en mente, se estudio el tiempo de computo para 1, 2, 4, 8 y 16 *threads* en función a la cantidad de claves que se le incrementaron al *bucket* a desabalancear (n), y se compararon los resultados para $K = 0$ y $K = 10000$. Para $K = 0$, independientemente de la cantidad de *threads* se espera que la diferencia en los tiempos sea casi despreciable, mientras que en el segundo escenario

debería ser sustancial. Como se ve en la Figura 8, los tiempos de carga de las claves se disminuyen significativamente hasta 4 threads, luego la diferencia es mínima. Consecuentemente se espera ver lo mismo en este experimento.

Resultados

En la Figura 11a se puede observar como a medida que incrementa la cantidad de claves en el *bucket* a desbalancear aumentan los tiempos de cómputo significativamente de forma constante. La diferencia entre los escenarios *singlethreaded* y los *multithreaded* se mantiene constante y mínima. Sin embargo, en la Figura 11b se puede observar que al incrementar K , esta diferencia aumenta enormemente, debido a que en los casos *multithreaded* no se presenta *idleness* por los *threads* a los que no les fue asociado el *bucket* a desbalancear.

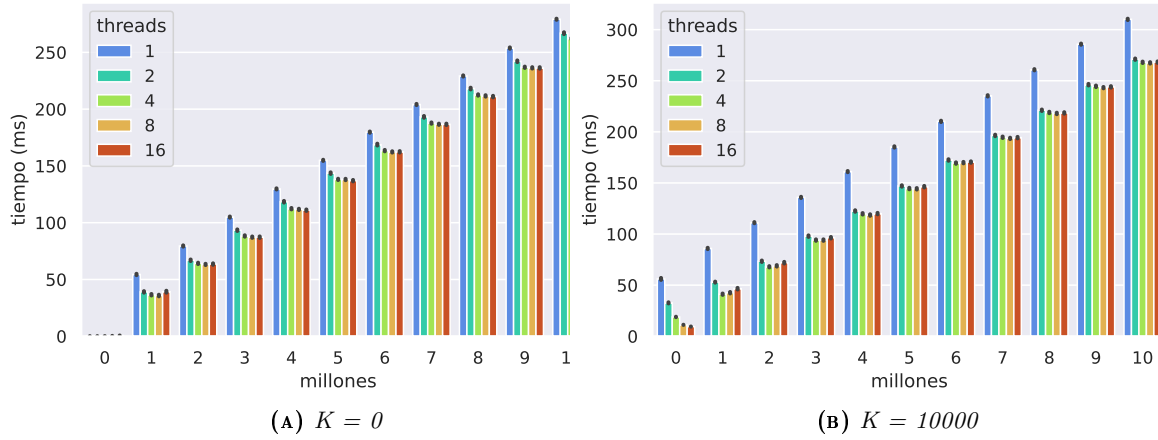


FIGURA 11: Tiempos de *MáximoParalelo* en función del tamaño del *bucket* desbalanceado.

3.1.5 Tiempo de cómputo en función de la cantidad de colisiones entre sets de datos

Este experimento busca analizar el tiempo de cómputo para la carga concurrente de archivos, mediante la función `CARGARMULTIPLESARCHIVOS` con respecto a la cantidad de “colisiones” entre los sets de datos provistos como *input*.

Consideramos “colisiones” a aquellas palabras en los *datasets* que coinciden en inicial, estas serán contenidas cuando se desea incrementar concurrentemente dos palabras con el mismo *hash*.

Para el experimento construimos tres escenarios posibles. Sean A, B, \dots, Z conjuntos que contienen todas palabras que comienzan con esa iniciales. Sea $\Omega = \{A, B, \dots, Z\}$.

El primero ocurre cuando todos los *datasets* tienen las mismas palabras y todas las palabras tienen la misma inicial. Sean $S_1^{(1)}, S_2^{(1)}, \dots, S_n^{(1)}$ los conjuntos a evaluar entonces $S_1^{(1)} = S_2^{(1)} = \dots = S_n^{(1)}$ y $\exists \ell \in \Omega : S_i^{(1)} \subseteq \ell$ para todo $S_i^{(1)}$.

El segundo apunta a lo contrario, es decir, *datasets* sin ninguna colisión de iniciales entre sí, donde cada *dataset* consta de palabras con una única inicial. Sean $S_1^{(2)}, S_2^{(2)}, \dots, S_n^{(2)}$ los conjuntos a evaluar entonces $S_i^{(2)} \neq S_j^{(2)}$ para todo $i \neq j$ y $\forall S_i^{(2)} \exists \ell \in \Omega : S_i^{(2)} \subseteq \ell$.

El último representa un caso intermedio, en el que la mitad de las palabras no colisionan con ningún *dataset* y la otra mitad es un subconjunto del resto de los sets. Sean $S_1^{(3)}, S_2^{(3)}, \dots, S_n^{(3)}$ los conjuntos a evaluar entonces $S_i^{(3)} \neq S_j^{(3)}$ para todo $i \neq j$, $S^{(3)*} \subseteq S_i^{(3)}$ para todo $S_i^{(3)}$ donde $S^{(3)*} \cap S_i = \emptyset$ y $\forall S_i^{(3)} \exists \ell \in \Omega : (S_i^{(3)} - S^{(3)*}) \subseteq \ell$.

Además se tiene que $|S_i^{(k)}|$ es igual para todo $1 \leq k \leq 3, 1 \leq i \leq n$.

Resultados

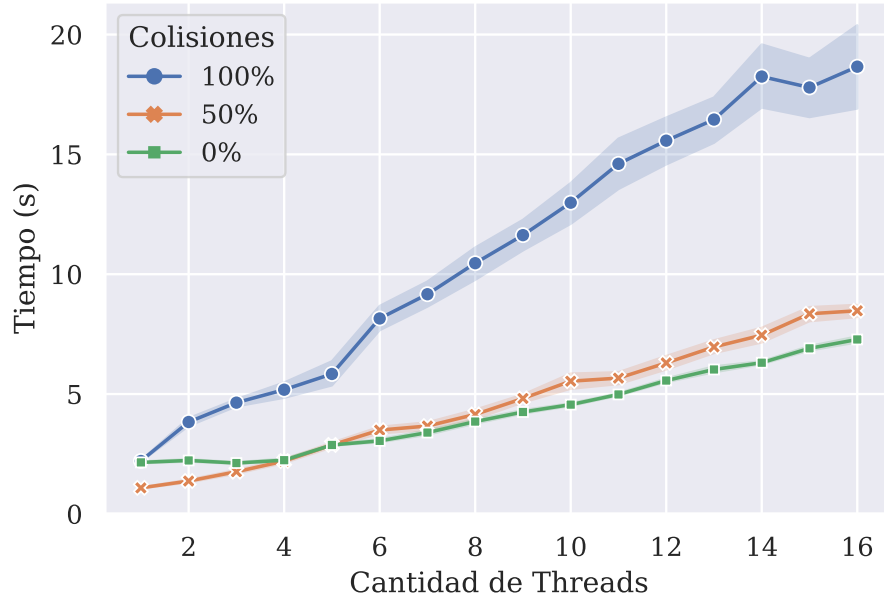


FIGURA 12: *Tiempos de `cargarMultiplesArchivos` en función de la cantidad de threads, considerando 100%, 50% y 0% de “colisiones” respectivamente.*

La Figura 12 nos permite ver, tal como esperábamos, que el tiempo de ejecución aumenta a medida que la cantidad de colisiones entre los conjuntos aumenta. De hecho, podemos notar que para cualquier cantidad de *threads* disponibles empleados, el tiempo de cómputo del caso en que todas las palabras e iniciales son idénticas domina a las otras dos.

De todas formas es interesante observar que recién a partir de cierta cantidad de *threads*, 4 en este caso, la performance del caso en que un 50% de iniciales colisiona es superada por escenario en que no hay coincidencias ni en palabras ni en iniciales. Creemos que esto se puede deber a que el caso sin colisiones trabaja con *buckets* de n elementos, mientras que el otro caso lo hace con *buckets* de

$\frac{n}{2}$ elementos, por lo que la diferencia de cómputo recién puede apreciarse a partir de cierta cantidad de *threads*.

Aún así es relevante notar que el porcentaje de colisiones no es despreciable a la hora de comparar entre estos dos escenarios, es pareja la diferencia temporal entre un caso en que no hay necesidad de contención y otro en que en la mitad de los casos los *threads* están esperando acceder a una sección crítica del código.

4 Conclusiones

Vimos que aumentar la cantidad de threads implica una mejoría en los tiempos de cómputo muy significativa. Sin embargo, se alcanza un techo rápidamente. A partir de los cuatro threads la mejoría no es sustancial, y hasta puede ser contraproducente debido a la contención y el diseño del `HASHMAPCONCURRENT`. Nos sorprendió el enorme grado de varianza que presentaron los experimentos, forzando a que se utilicen enormes cantidades de datos, sin duda también instados en gran medida por la contención y otros procesos de fondo ejecutándose en la PC.

Referencias

- [1] *Kaggle*. URL: <https://www.kaggle.com>.