DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

15 de mayo de 2019 — Sistemas Operativos

Integrante	LU	Correo electrónico
Cortés Conde Tító, Javier María	252/15	javiercortescondetito@gmail.com
Alvarez Vico, Jazmín	75/15	jazminalvarezvico@gmail.com
Strika, Luciano	76/16	lucianostrika44@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) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

1.1. Objetivo

El presente trabajo tiene como objetivo comprender y utilizar la programación en paralelo, y la sincronización mediante Threads y distintos tipos de “locks” y semáforos.
Para lograr esto, el trabajo consiste en implementar y analizar un algoritmo que encuentre el árbol generador mínimo (AGM) de un grafo utilizando multithreading.

2. Implementacion

Por los resultados de la experimentación, nos parece pertinente marcar los problemas que tuvimos al momento de la implementación. Pese a que en las conclusiones vamos a listar nuestras hipótesis de cómo podríamos haberlos resuelto, creemos que algunas mejoras y cambios se entienden mejor al ir leyendo sobre la implementación.

2.1. Explicación del Algoritmo

El algoritmo para generar el AGM de manera concurrente cuenta con tres partes:

- Comer si se tiene para comer
- pintar y deliberar quien es comido y quien es comedor,
- y finalmente el comportamiento normal de un AGM secuencial.

Mas adelante detallaremos las estructuras que utilizamos, deteniéndonos en las que nos parecen relevantes. Daremos además una explicación del algoritmo en palabras. En caso de querer comprender mejor una parte, se pueden referir al código, que está apropiadamente documentado.

La forma de manejar la interacción de threads en esta implementación es bastante simple, pero a la vez restrictiva. Cada thread tiene una variable que indica quién lo va a comer, formándose un bosque dirigido de comidos. Por ejemplo, cuando thread 5 se encuentra con thread 10, primero se decide cuál es nodo raíz del ”árbol de comidos”del thread 10. En el caso de que sea mas chico que el thread 5, este es comido por la raíz y se agrega a una cola de threads que tiene que comer. En caso contrario el thread 5 se come a la raíz y pasa a ser la raíz de ese árbol. Lo que se trata de simular es que si 5 se choca con 10, pero 10 había encontrado a 6, es que 5 se chocó con 6. El caso de que 10 se encuentra con 5 es análogo.

Ahora que explicamos de forma superficial el comportamiento general de las interacciones, vamos a detallar las variables globales y las estructuras que utilizamos:

2.2. Variables Globales

- vector<atomic_int> colores: indica quien comió a que color. Es atómico, ya que no hay mucho que hacer además de pintar o no dependiendo de si ya fue pintado.
- vector<EstadoThread> ThreadInfos: Tiene la información de cada thread.
- vector<mutex> eating: bloquea un thread si le están agregando a la cola, o bloquea la cola si está comiendo. Podría reemplazarse por una Queue atómica en cada thread.
- mutex eatingRoot: cuando un thread encuentra un nodo de otro thread, se bloquea la función que busca la raíz del árbol. Esto es importante porque el árbol se puede romper en una sección y hace falta que sea consistente.
- mutex waitToCreate: este mutex es meramente para que el nodo 0 termine último. Esto es porque en caso de grafos chicos en comparación a la cantidad de threads, el 0 puede terminar antes de la creación de todos los threads.

2.3. Estructuras

Struct EstadoThread:

- int índice : identifica al thread y representa su color.
- Grafo * pintados : puntero al AGM que estamos construyendo.
- vector<int> nodosRecorridos : almacena los nodos que ya se pintaron.
- vector<int> distancia: guarda la distancia mínima del thread a todos los otros nodos del grafo.
- vector<int> distanciaNodo: guarda desde que nodo del grafo del thread, se pueden alcanzar los otros nodos con distancia mínima.
- queue<int> threadsToEat : cola con los índices de los threads que hay que comer.
- int myEater: -1 indica que no fue comido y si no tiene el id de quien lo comió.
- atomic_int estado:
  - 0 estado inicial del thread.
  - 1 listo para ser comido.
  - 2 ya fue agregado a una cola.
  - -1 se murió sin ser consumido.

  esta variable se usa para saber si se puede comer el thread, o si se puede encolar. Siendo una variable atómica, garantizamos que no habrá problemas de sincronización a la hora de averiguar o cambiar el estado del thread. Esto es importante ya que lo necesitamos en la lógica de çomer"threads.
- mutex mutexMyEater bloquea el uso de myEater en momentos de comparación, es decir antes de entrar al ciclo, cuando se encuentra con un nodo distinto a el, si ya tiene que ser comido no queremos hacer nada con ese hallazgo, y en la función que busca la raíz del árbol. Este mutex no puede ser remplazado, pero el uso de myEater podría verse reducido en la función principal usando la variable estado.
- mutex deadThread este mutex se explicara en mejor detalle cuando se entiendan los pasos del algoritmo.

Ahora que tenemos una noción general de las estructuras y cuándo se van a usar, detallaremos los pasos a seguir de cada thread.

- Primero, el thread revisa si debe comer a otro thread, consultando su cola de threads a comer. De ser así, el comedor hace busy waiting, esperando a que el otro thread termine de hacer cosas, entre ellas pintar un nodo, o comer. Cuando el comido avisa, cambiando su estado a 1, se ejecutan los siguientes pasos:
  - El grafo del thread comido pasa a ser parte del grafo del comedor.
  - Se actualizan las estructuras internas del thread comedor con la información que no tenía del thread comido, la frontera del grafo consumido, qué nodos lo alcanzan, sus distancias. También consume la cola de threads del comido.

  Antes de empezar a comer se llama a un mutex específico para bloquear el acceso a la cola del comedor, para no tener race conditions. Esto podría ser resuelto con una cola atómica. También el busy waiting parece ser una idea óptima si la forma de comer no fuese tan estructurada, ya que todos los threads cuando mueren comen y después se dejan ser comidos.
  Por ejemplo, si la lista es A come a B, B come a C y C a D, A hace busy waiting mientras C come a D y después B come a C y después finalmente A come a B. Tener en cuenta que B también hace busy waiting.

- Luego, el thread agrega un nodo nuevo a su lista de pintados siguiendo el criterio de generación de AGM. Esto es, el nodo de menor distancia con los nodos del árbol del thread.

- A la hora de agregar el nodo, el thread revisará si este efectivamente no estaba pintado. Al ser una variable atómica, se utiliza un compare and exchange. De no estarlo, lo agrega y termina este ciclo. De estarlo se ejecutan los pasos siguientes:
  - Se busca al comedor Supremo (getBigEater): esto quiere decir ir por el árbol de myEater hasta que se llega a -1, encontrando al thread mas chico que tiene que comer. En esta función se usan tres mutex con distintos propósitos:
    - bloqueamos la función entera, ya que se necesita tener este árbol de forma consistente. esto genera un cuello de botella, ya que se tiene que esperar a que otro thread termine de deliberar su posición en la lista de comidos antes de poder deliberar la nuestra.
    - Mientras recorremos los myEater bloqueamos el que estamos modificando, ya que al final del recorrido puede ser que la cabeza este desfasada (es decir, que e.g. el thread 0 este siendo comido por el 4). Esto se arregla al final de la función, pero necesitamos tenerlo bloqueado.
    - Al deliberar, el que va a ser comido se bloquea el deadThread, este es esencial para que se agregue a la cola sin que el otro thread cambie de estado antes.
  - Luego se decide quien se come a quien, y se encola en su lista de comidos, desbloqueando el deadThread así el thread comido puede setear su estado.
- se sigue con la lógica normal del algoritmo de AGM secuencial.
- al final del ciclo se chequea que no este generado todo el grafo y que myEater no este seteado, de estarlo se sale del ciclo y se ejecutan los siguientes pasos.
  - en el caso de ser el thread cero se espera que se mueran los demás threads así se imprime al fin de todos los threads en caso de que el grafo sea chico o los threads sean muchos y el thread cero ya haya pintado todos los threads antes de que otros hayan arrancado a recorrer el grafo.
  - se come lo que haya quedado en la cola de la misma forma que en el ciclo.
  - se setea su estado, si ya había sido agregado a una cola, se pone listo para ser comido. En caso contrario se setea como muerto, así si alguien lo encuentra se puede encolar. la diferenciación de estados es para evitar encolar varias veces en una misma cola, es decir, nosotros encolamos si nadie lo encolo antes. Como dijimos antes esta parte esta bloqueada para que se lea bien el estado.

  Usar myEater como condición de salida implica bloquearlo varias veces, en cambio se podría usar la variable atómica estado y tener un comportamiento mas ameno, a la vez no tendríamos que usar el mutex deadThread.

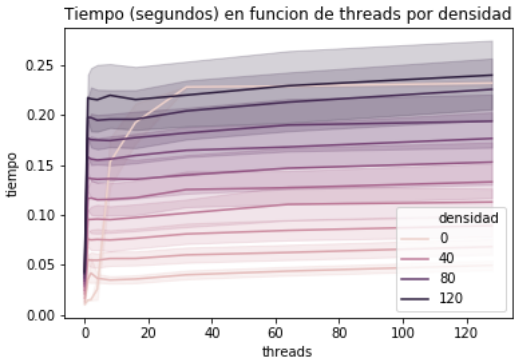
3. Experimentación y Análisis de resultados

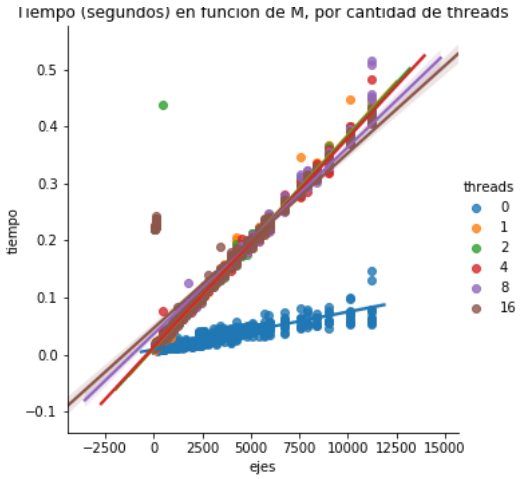
Para evaluar la performance del algoritmo en paralelo, en comparación con su versión secuencial, generamos grafos aleatorios y ejecutamos el algoritmo, variando la cantidad de threads entre 1 y 128 (probando con las potencias de 2).

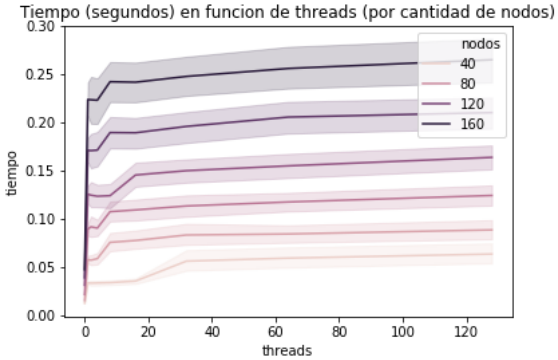
3.1. Experimentos con grafos pequeños y densos

Para el primer experimento, se midió el tiempo de ejecución del algoritmo secuencial, comparándolo contra su versión paralela usando K threads, donde K tomó todos los valores de potencias del 2 entre 1 y 128 inclusive.
Los grafos usados como input tenían entre 50 y 150 nodos, saltando de a 20, y densidades de entre 0 y

100 %, creciendo de a 10 %, la densidad es una funcion lineal que cuando se evalua en 0 te devuelve la cantidad de ejes necesarios para tener un arbol y cuando se evalua en 100 te da los ejes necesarios para tener un knn.
Se repitió el experimento para cada input 10 veces, para descartar la posibilidad de outliers, y mitigar el efecto de la varianza en el tiempo medido.
Independientemente de la densidad de los grafos, de su cantidad de vértices o de su cantidad de aristas, el resultado fue siempre el mismo: la versión single-threaded era la más rápida. Creemos que esto se debe a que los grafos que usamos eran demasiado pequeños, volviendo al overhead de la paralelización demasiado significativo respecto a los tiempos del algoritmo mismo.
Por este motivo, en el siguiente experimento probamos ejecutar el algoritmo con inputs significativamente más grandes.

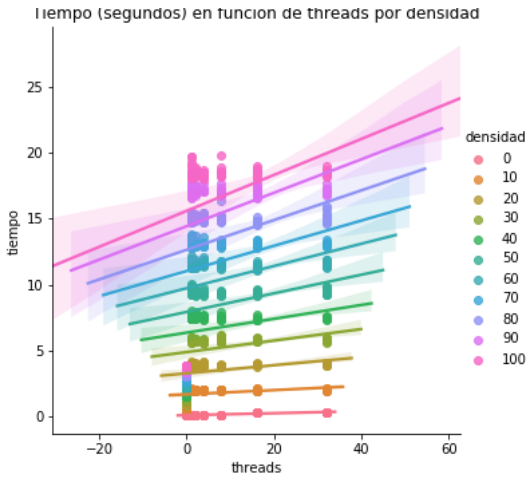




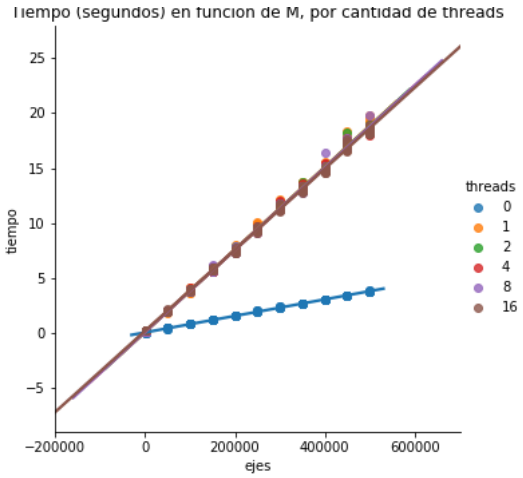


3.2. Experimentación con grafos de 1000 nodos

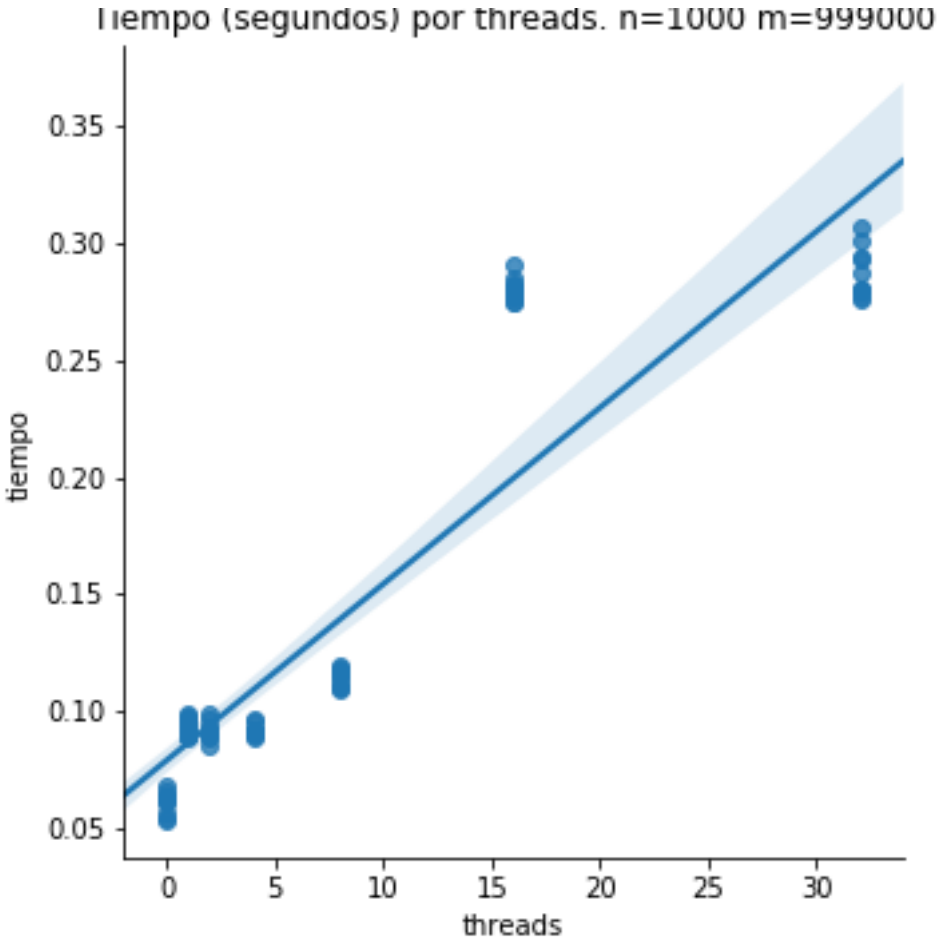
Para el siguiente experimento, se midieron los tiempos de ejecución en función de la densidad de grafos conexos de 1000 nodos, con densidades del 10 % al 100 %, moviéndose de a 10 puntos.
Llamamos "densidad 0.ªl caso particular en el que el grafo es un árbol ($n - 1$ aristas)
Se comparó la versión secuencial del algoritmo con la concurrente usando 1, 2, 4, 8, 16 y 32 threads, llegando a la conclusión de que nuestra implementación usando threads es altamente subóptima (siendo la secuencial significativamente mejor en todos los casos).

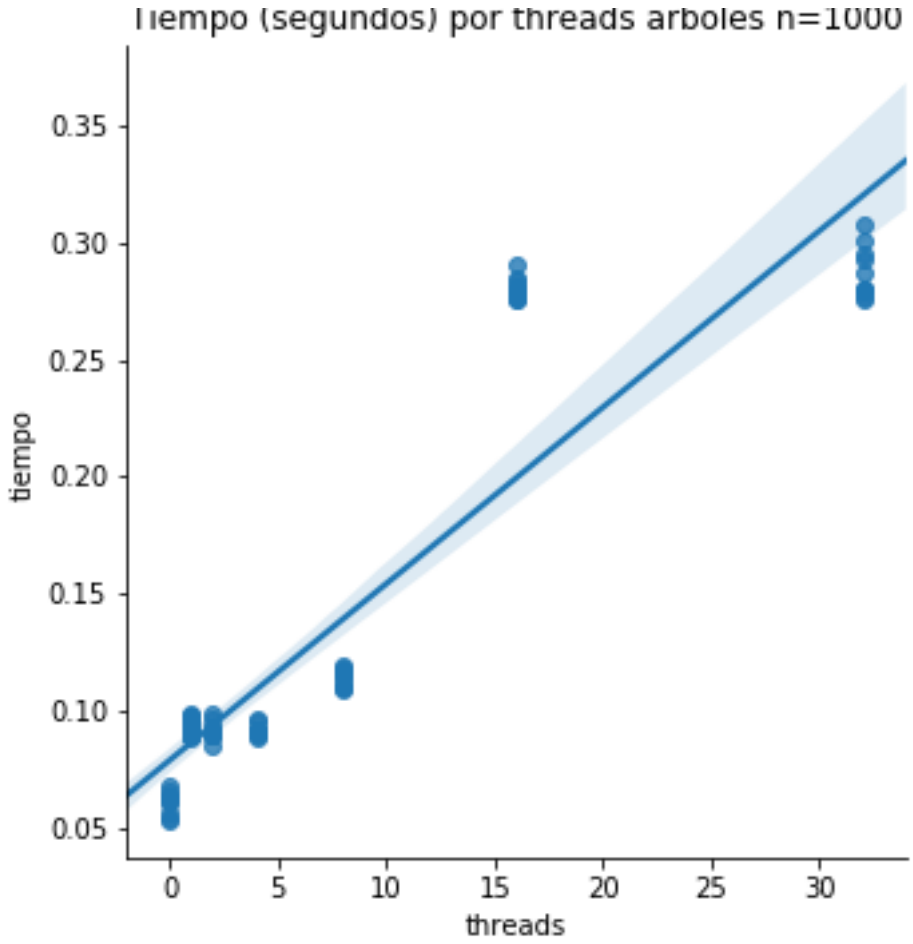


Puede verse que, a diferencia de la densidad del grafo, que claramente predice un mayor tiempo de computo a medida que se incrementa, la cantidad de threads usados no juega un rol en lo más mínimo significativo a la hora de evaluar cuánto tiempo tardará en ejecutarse el programa.



Lo mismo puede observarse al mirar la cantidad de ejes de los grafos.
Por último, elegimos hacer foco en dos tipos de grafo particulares: árboles y grafos completos, por ser los de menor y mayor cantidad de ejes respectivamente, propiedad que nos pareció significativa para evaluar tiempos.
Los resultados son terribles: La performance es casi inversamente proporcional a la cantidad de threads usados (siendo el tiempo virtualmente lineal en la cantidad de threads).


Tiempo (segundos) por threads. n=1000 m=999000
tiempo
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
5
10
15
20
25
30
threads



Creemos que las discrepancias se deben a el exceso de sincronización en algunas operaciones que podrían haberse ejecutado en paralelo. Particularmente bloquear la ejecución de todos los threads cuando dos chocan. Dados los tiempos acotados con los que contábamos, no llegamos a evaluar si esta era la causa del problema, aunque proponemos varias hipótesis y posibles soluciones.

4. Hipótesis y oportunidades de mejora en la implementación

Para tener un algoritmo con threads que funcione mejor proponemos los siguientes cambios fundamentales:

- Cambiar la forma de asignar threads comidos. En vez de pensar en todo el árbol, solo considerar como mucho a 3 threads. Supongamos que A encuentra a B y B es comido por C, si A tiene que ser comido por B el comportamiento es igual. En el caso contrario, en vez de buscar la cabeza de la raíz, buscamos en la cola de C a B y lo encolamos en A, después A es encolado en C.
  Esto distribuye la carga de quien tiene que comer a quien de forma mas homogénea, en vez de que recaiga todo sobre la raíz del árbol, a la misma vez saca el mutex myEater por que lo podemos modificar de forma atómica, y el mutex eating root, sacando un proceso que bloquea a cualquiera que quiera deliberar en temas de comida.

- Usar estado como variable para una condición de terminación del ciclo en vez de myEater, ya que este que genera 1 bloqueo por iteración de ciclo todo el tiempo y 1 mas en casos de que ese thread tenga que comer o ser comido y la generación de otro mutex para manejar la llegada asíncrona a setear el estado, eso le da un costo altísimo a cada ciclo del programa, esto nos dejaría que myEater sea atómica realmente y poder sacar su mutex y el deadEater.

- hacer que la cola sea atómica, esto sacaria el mutex de eating.
- Sacar el bussy waiting en comer, por la estructura del programa hacer bussy waiting requiere que otros threads terminen de comer y se mueran. Por la estructura e ideas del algoritmo, se tendría que modificar demasiado como comer y como termina cada thread, para que que se pueda usar bussy waiting de forma eficiente. Usar un mutex con esta estructura parece ser a priori mucho mejor.

Por razones externas e internas a nosotros, no pudimos lijar la implementación para llevar a cabo un código mas eficiente, para cuando nos dimos cuenta que el código no era eficiente, rehacerlo, experimentar no eran una opción viable.