DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Threads

Sistemas Operativos

Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Castiglione, Rubén Adrián	818/15	adriancastiglione@gmail.com
Sassone, Federico Sebastián	602/13	fede.sassone@hotmail.com
Teren Bernal, Guido Sebastián	749/14	guidoteren@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

1. Introducción

El tp consiste de un ConcurrentHashMap, una estructura que simula un diccionario con la posibilidad de hacer reaparecer varias definiciones, y varias operaciones con el mismo. En particular se busca desarrollar distintas implementaciones de contar palabras, chequear si un elemento pertenece al diccionario y aumentar sus apariciones.

Renombres en el TP:

Definimos ParClaveApariciones como la clase que representa los pares <string, unsigned int> a guardar en nuestro diccionario (De ahora en mas, ConcurrentHashMap, o CHM). Un parClaveApariciones sabe responder a dameClave(), dameApariciones() y aumentarApariciones() (Que suma 1 a las apariciones del par).

A continuación, los métodos de la sección de Introduccion, su implementación y observaciones sobre los mismos:

1.1. ConcurrentHashMap()

- Constructor. Crea el diccionario con 26 buckets que representarán a las 26 letras del abecedario.
- Los buckets se representan con una tabla: vector < Lista < ParClaveApariciones > > tabla26
- Cada Lista será una ListaAtomica proporcionada por la cátedra
- A cada parClaveAparicion lo enviaremos al bucket correspondiente al ser ingresadas en el CHM accediendo a la primer letra de su clave.
- Cada CHM tendrá además un mutex LockMaximum, un mutex lockCargar y un vector <mutex> vectorMutex cuyos funcionamientos y utilidad explicaremos en los metodos que los llamen.

1.2. void addAndInc(string key)

- Nos pasan como un parametro una clave key. La buscamos en los parClaveAparicion de nuestro CHM. Si está agregada aumentamos en 1 sus apariciones; si no, la agregamos con apariciones = 1.
- Para implementar esta funcion creamos un char primerLetra con el primer char de key (asumimos que no nos pasan un key vacío) y utilizamos el metodo dameIndice(primeraLetra) para saber el indice del bucket al que corresponde la key (asumimos que no hay mayúsculas). a tiene asignado el bucket 0, b el 1, c el 2, ..., z el 25.
- En este momento, con el vectorMutex del CHM lockeamos el correspondiente al indice del bucket para que si dos threads tienen que trabajar con el mismo bucket del CHM no haya race conditions.
- Accedemos a la Lista en el indice correspondiente de la tabla de CHM y creamos un iterador para la misma.
- Recorremos la Lista de parClaveAparicion hasta encontrar algun par cuya clave sea igual a nuestro key o quedarnos sin elementos a recorrer. Si encontramos a algun par que satisfaga la condición, aumentamos su apariciones en 1 y desbloqueamos el mutex correspondiente antes de terminar.
- Si no encontraramos un parClaveAparicion válido, crearíamos un parNuevo con clave=key, apariciones=1 y utilizamos push_front(parNuevo) de Lista para insertarlo en la misma.
- Notese que la condicion especial de este ejercicio, çontencion en caso de colision de Hash.[es]tá contemplada. Si se llama a addAndInc() desde dos lugares con dos keys que tienen la misma letra inicial, ambas van a ir a parar al mismo bucket; por lo que la primero que acceda a esta Lista, la lockeará, evitando que se pisen.

1.2.1. push_front(const T&val)

A pesar de que llamamos a este método siempre dentro de una Lista lockeada (implicando que el mismo no va a ser interrumpido) el enunciado pedía que el mismo fuera atómico. No podemos hacerlo realmente ya que no tenemos forma de hacerlo en una única linea atómica. Lo más cercano que tenemos es entonces lockear un mutex y correr tres lineas de código, cada una atómica. Es justamente lo que hacemos. Lockeamos, creamos nuestro nuevo nodo que tendrá el parClaveAparicion a agregar, lo asignamos como siguiente al head de la lista y luego lo hacemos head de la misma; luego, desbloqueamos.

1.3. Bool member(string key)

- Devolverá verdadero si la key está guardada en el CHM.
- La implementación es muy parecida a la del método anterior. Solo que sin lockear la lista.
- Volvemos a buscar el indice del bucket correspondiente, volvemos a recorrer la Lista correspondiente y volvemos a hacer el chequeo por clave igual a key en cada parClaveApariciones de la Lista.
- La diferencia está en que si encontramos una clave que satisfaga, devolveremos true. Caso contrario devolvemos false.
- Nuevamente notese que la condicion especial se cumple. El método es $wait - free$ ya que no tiene que esperar a nadie para ejecutarse.

1.4. pair<string, unsigned int>maximum(unsigned int nt)

- Devolvemos el parClaveApariciones tal que su dameApariciones devuelve el número mayor en comparación a todos los parClaveApariciones del CHM.
- El parámentro nt representa la cantidad de threads con la que debe ejecutarse el método. Que emoción, nuestro primer método con threads!
- maximum() no puede ser concurrente con addAndInc(). Para lograr esto, lockeamos cada lista en el momento previo a que un thread la acceda. Luego si el CHM se modifica una vez comenzada la ejecución por un addAndInc() el cambio no se verá reflejado en el resultado de maximum().
- Además, cada thread correrá sobre un bucket distinto y lo hará hasta que no haya buckets disponibles para procesar.
- Esto nos dió la noción de que debiamos tener una estructura de control que nos diga qué bucket ya fué procesado. Veamos la idea del algoritmo:

```
1       thread t[nt];
2       atomic<int> siguienteFilaALeer;
3       ParClaveApariciones maximo = ParClaveApariciones("a",0);
4       siguienteFilaALeer = 0;
5
6       for (int i = 0; i <nt; ++i)
7       {
8               t[i]=thread(obtenerMaximasRepeticiones,
9               ref(siguienteFilaALeer), ref(maximo), ref(*this));
10      }
11
12      for(int k=0;k<nt;k++){
13              t[k].join();
14      }
15
16      return maximo;
```

- Creamos un array de threads con la cantidad utilizada para correr el metodo
- Guardamos un int atomico el siguiente bucket libre. Al mismo accederan todos los threads, modificandolo y consultando por el siguiente bucket a recorrer. Si el mismo llega a 26, el thread queda esperando a que sus hermanos terminen.
- Creamos un parClaveApariciones contra el que todos los threads compararan y remplazarán atomicamente los maximos obtenidos en cada recorrida de un bucket si fuera necesario. Este se inicia con un apariciones en 0 para que al menos un parClaveAparicion lo reemplace.
- Seteamos la primer fila a leer en 0, (bucket del char a).
- Para cada thread corremos obtenerMaximasRepeticiones pasandole como referencia la siguiente fila sin procesar, el maximoActual y un punteo a el CHM.
- Los threads se esperan entre ellos, todos terminan juntitos y felices.
- devolvemos maximo.
- Veamos el codigo de obtenerMaximasRepeticiones:

```
1       int actual = atomic_fetch_add(&siguienteFilaALeer, 1);
2
3       while(actual<26){
4               Lista<ParClaveApariciones> *lista = &chp.tabla[actual];
5               lista->mtx.lock();
6
7               Lista<ParClaveApariciones>::Iterador iterador = lista->CrearIt();
8               ParClaveApariciones parClaveAparicionesActual;
9               ParClaveApariciones maximoLocal = ParClaveApariciones("a",0);
10              while(iterador.HaySiguiente()){
11                      parClaveAparicionesActual = iterador.Siguiente();
12                      if(parClaveAparicionesActual.dameApariciones() >=
13                      maximoLocal.dameApariciones()){
14                              maximoLocal = parClaveAparicionesActual;
15                      }
16                      iterador.Avanzar();
17              }
18              lista->mtx.unlock();
19              chp.lockMaximum.lock();
20              if(maximoLocal.dameApariciones() >= maximo.dameApariciones()){
21                      maximo = maximoLocal;
22              }
23              chp.lockMaximum.unlock();
24              actual = atomic_fetch_add(&siguienteFilaALeer, 1);
25      }
```

- Cada thread actualiza y carga siguienteFilaALeer múltiples veces dentro de ObtenerMaximasRepeticiones. Esto finaliza cuando se alcanza 26 (letra fuera del abecedario)
- Con el índice del bucket a recorrer, nos traemos un puntero al mismo, lo lockeamos con su mutex y le creamos un iterador.
- Mientras haya siguiente comparamos las apariciones...
- Desbloqueamos la lista y lockeamos el Maximo global.
- Comparamos los valores y si nuestro maximo local es mayor, lo reemplazamos atomicamente.
- Traemos el valor de siguienteFilaALeer (Puede haber sido modificada por otro thread) y seguimos loopeando.

2. Ejercicios

2.1. push front e implementacion de ConcurrentHashMap

Explicado en la sección 1.2.1

2.2. ConcurrentHashMap count words(string arch)

- Esta función toma un archivo de texto como parámetro y crea un CHM cargandole las palabras del mismo como claves para todo parClaveAparicion en el mismo. Las palabras repetidas aumentan apariciones. Será implementado de manera no concurrente.
- Veamos el código:

```
1       ConcurrentHashMap j;
2       ifstream TestEntrada;
3       string aux;
4       TestEntrada.open(arch.c_str());
5       while(!TestEntrada.eof()){
6
7               TestEntrada >> aux;
8               if(TestEntrada.eof()) break;
9               j.addAndInc(aux);
10      }
10      return j;
```

- Se crean un nuevo CHM, un ifstream y un string aux.
- Se abre el archivo con el ifstream y mientras no lleguemos al final:
- Parseamos un string (solo tomamos palabras separadas por espacios) lo tomamos como key y lo agregamos al CHM con addAndInc(key).

2.3. ConcurrentHashMap count words(list <string> archs)

- Misma idea que el ejercicio anterior pero ahora pasamos como parámetro una lista de archivos en vez de uno sólo. Debemos cargar todas las palabras de todos los archivos a un mismo CHM y adicionalmente, cada archivo debe ser procesado con un thread distinto.
- Código:

```
1       ConcurrentHashMap res;
2       thread t[archs.size()];
3       list<string>::iterator pos;
4       pos = archs.begin();
5       int i = 0;
6        while( pos != archs.end())
7        {
8               t[i]=thread(cargarConcurrentHashMap,ref(res),ref(*pos));
9               pos++;
10              i++;
11       }
12              for(int k=0;k<archs.size();k++){
13                      t[k].join();
14              }
15      return res;
```

- La idea es lanzar un thread para cada archivo y utilizar en cada uno de ellos la misma idea del ejercicio anterior. Como addAndInc() es concurrente, no debería haber mayores complicaciones.
- Se crea un CHM, un vector de threads de tamaño igual a la cantidad de archivos a procesar, un iterador de Lista<string> y se lleva un index para el array de threads.
- Luego, mientras que no recorramos todos los archivos, lanzamos un thread por archivo y movemos el indice y el iterador. El thread llamará a cargarConcurrentHashMap con parámetros res (que será el CHM final) y un puntero al comienzo de su archivo particular.
- Finalmente, los threads se esperan entre sí y se devuelve el CHM final.
- Veamos ahora el código de cargarConcurrentHashMap:

```
1       ifstream TestEntrada;
2       string aux;
3       TestEntrada.open(arch.c_str());
4       while(!TestEntrada.eof()){
5               TestEntrada >> aux;
6               if(TestEntrada.eof()) break;
7               chp.addAndInc(aux);
8       }
```

- Con la referencia de CHM y el puntero al arch, el thread ejecutará una versión muy parecida al ejercicio anterior.

2.4. ConcurrentHashMap count words(unsigned int n, list<string>archs)

- Parecida al anterior solo que ahora específicamos la cantidad de threads a utilizar y hay que asignarlos inteligentemente de forma tal que nunca queden ”libres”si aun quedan archivos por procesar. Se utilizará para este objetivo una idea parecida a la usada en addAndInc(). Un int atómico que todos los threads modifiquen, el cual indicará el siguiente archivo aun sin procesar:

```
1       ConcurrentHashMap res;
2       thread t[n];
3       atomic<int> siguiente(0);
4       for (int i = 0; i < n; ++i){
5               t[i]=thread(cargarConcurrentHashMapThread,
6               ref(res),ref(siguiente),ref(archs));
7       }
8       for(int k=0;k<n;k++){
9               t[k].join();
10      }
11      return res;
```

- Se crean un CHM final, el array de n threads y el entero atómico que se utilizará por los threads para saber qué archivo es el siguiente a procesar.
- Para cada thread se llama a carcagConcurrentHashMapThread que tiene la lógica del algoritmo:

```
1               int actual = atomic_fetch_add(&siguiente, 1);
2               while(actual < archs.size()){
3                       auto it = archs.begin();
4                       advance(it, actual);
5                       string archivo(*it);
6                       cargarConcurrentHashMap(chp, archivo);
7                       actual = atomic_fetch_add(&siguiente, 1);
8               }
```

- Se busca y carga atómicamente el indice del siguiente archivo a leer y si el mismo es un número menor a la cantidad de archivos, se avanza con un iterador sobre la Lista de archivos hasta el mismo. Luego se carga el archivo al CHM con el primer count_words y se loopea el comportamiento.

- Finalmente los threads se esperan entre si y se devuelve el CHM.

2.5. pair<string, unsigned int>maximum(unsigned int p archivos, unsigned int p maximos, list<string>archs)

- Devuelve el parClaveAparicion con mayor apariciones para un CHM creado a partir de una lista de archivos. Toma como parámetro una cantidad de threads a utilizar para leer archivos y otra para calcular maximos. También una lista de archivos a procesar. Tenemos que no utilizar las versiones concurrentes de count_words.

```
0       int n = archs.size();
1       vector<ConcurrentHashMap> chms(n, ConcurrentHashMap());
2       thread t_archivos[p_archivos];
3       auto it = archs.begin();
4       for (int i = 0; i < p_archivos; ++i){
5               int desde = i * ( archs.size() / p_archivos );
6               int hasta = (i+1) * ( archs.size() / p_archivos ) - 1;
7               if ( i == p_archivos - 1 ) hasta = archs.size()-1;
8               t_archivos[i]=thread(cargarConcurrentHashMapThreadMaximumLectura,
9               ref(chms),it,desde,hasta);
10      }
11      for(int k=0;k<p_archivos;k++){
12                      t_archivos[k].join();
13      }
14      int apariciones;
15      for(int i=1;i<n;i++){
16              for (int j = 0; j < 26; j++) {
17                      for (auto it = chms[i].tabla[j].CrearIt();
18                      it.HaySiguiente(); it.Avanzar()) {
19                              auto t = it.Siguiente();
20                              apariciones = t.dameApariciones();
21                              for(int r=0;r<apariciones;r++){
22                                      chms[0].addAndInc(t.dameClave());
23                              }
24                      }
25              }
26      }
27      ParClaveApariciones maximo = chms[0].maximum(p_maximos);
28      return maximo;
```

- Guardamos la cantidad de archivos, creamos un vector de n CHM's, nuevamente un array de threads y un iterador de archivos.

- Para asignarle a cada conjunto de archivos un thread específico, tomamos el numero total de archivos y lo dividimos en los threads disponibles. Luego le decimos a cada thread en qué sección de la lista de archivos comenzar a cargar archivos a su CHM por medio de cargarConcurrentHashMapThreadMaximumLectura. Este método no es nada diferente a los ya vistos que cumplen funciones parecidas, solo que es precavido de solo leer los archivos que le corresponden. Para asignarlos, tomamos los primeros n/p y los enviamos al primer thread, y así sucesivamente, teniendo en cuenta el caso borde.

- Una vez más unimos los threads. Luego, tendremos que unir los n CHM's creados y encontrar el maximum de la union.
- Para hacerlo, buscamos el enfoque más sencillo, para los n-1 CHM's después del primero, para cada uno de sus buckets, mientras no lo hayamos recorrido entero, tomamos cada parClaveAparicion definido, leemos su clave y apariciones y lo agregamos a el primero de todos los CHM con addAndInc(clave) apariciones-veces. Esto es maravillosamente poco performante; pero estabamos muy cansados.
- Finalmente calculamos el maximo del primer CHM ahora modificado y convertido en un super-CHM con toda la informacion de los demás y lo devolvemos.

2.6. Version concurrente de 2.5

```
1       ConcurrentHashMap chm = count_words(p_archivos,archs);
2       ParClaveApariciones res = chm.maximum(p_maximos);
3       return res;
```

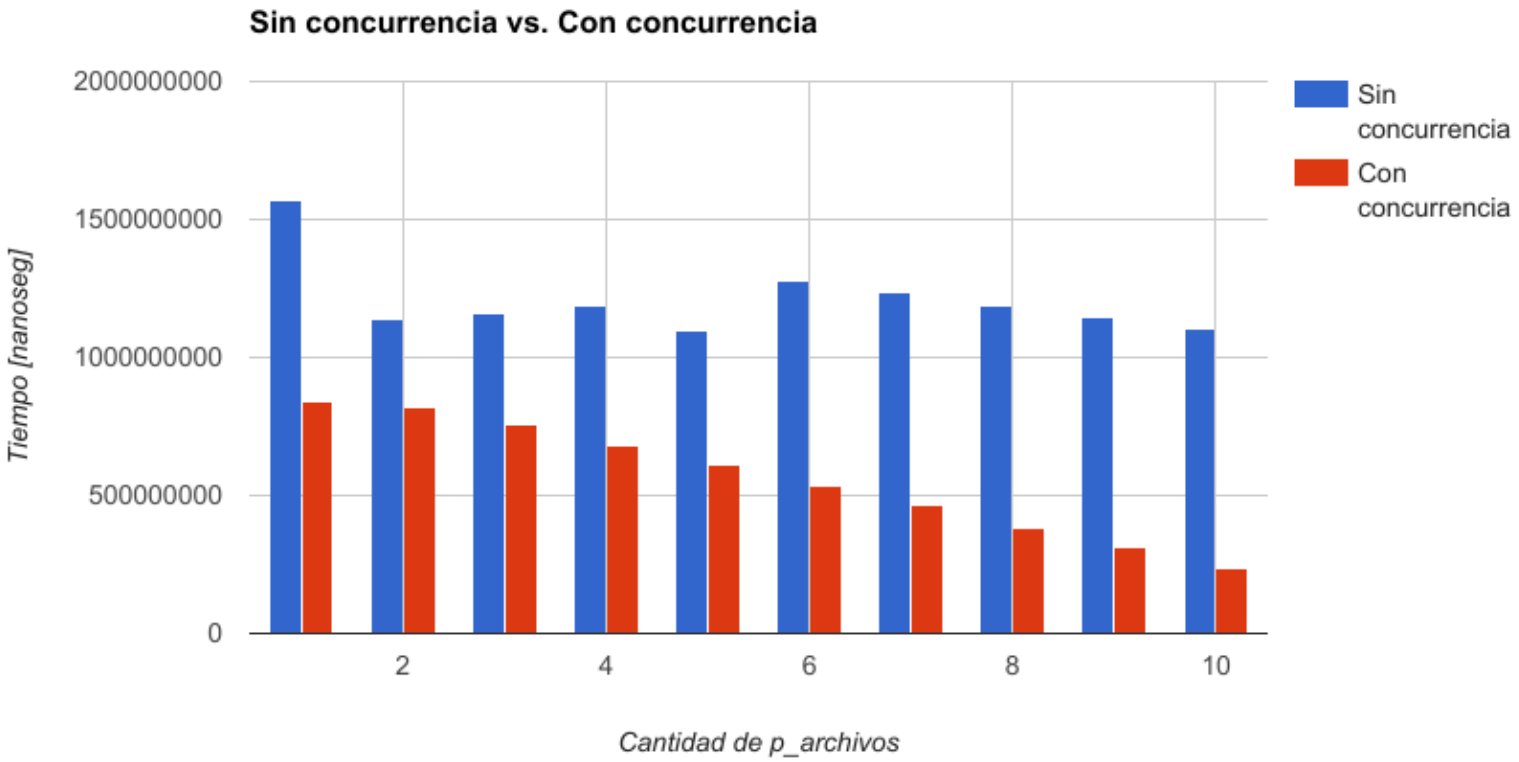
- No hay mucho para decir. Utilizamos los metodos definidos anteriormente. Creamos un CHM con p_archivos threads de lectura para archs.size() archivos.
- Luego devolvemos el parClaveApariciones resultante de buscar maximum con p_maximos threads de corrida de maximos.

3. Pruebas de rendimiento

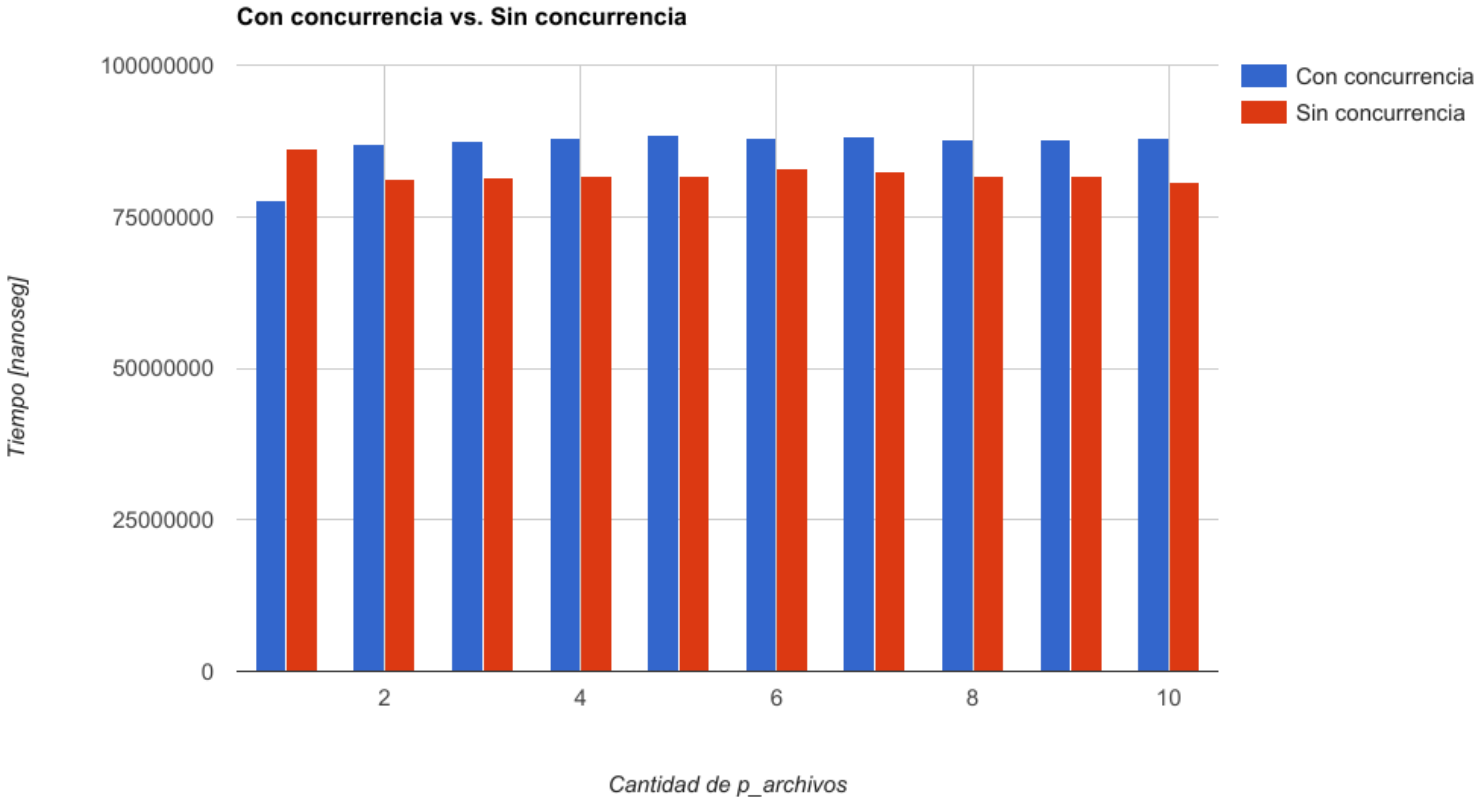
En esta sección nos proponemos medir el impacto en el rendimiento que puede lograrse mediante concurrencia. Utilizaremos las funciones maximum que calculan el maximo entre diferentes archivos, utilizando las versiones con y sin concurrencia de count words.

3.1. Primer Test

En este primer test utilizaremos 10 archivos de 2500 palabras cada uno, e iremos aumentando la cantidad de threads que procesan los archivos, a esta cantidad la llamamos p_archivos. Se espera que a medida que estos threads aumentan, la versión concurrente vaya mejorando su rendimiento. A continuación se muestran los resultados obtenidos (midiendo tiempo):



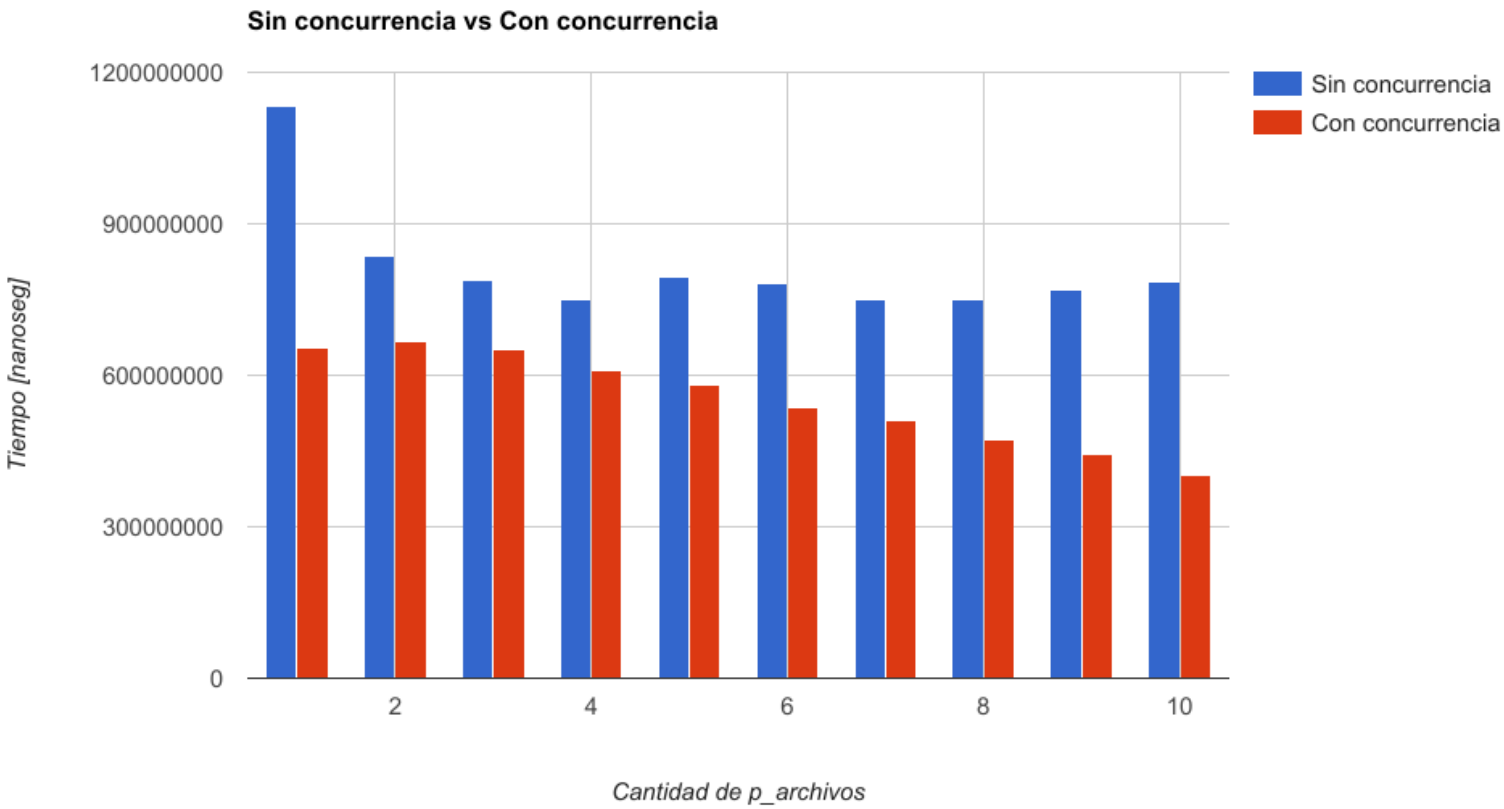
Tal y como se esperaba, la versión concurrente obtiene una diferencia de rendimiento apreciable. Sin embargo no siempre que aumentemos la cantidad de threads vamos a notar este tipo de mejoras. A continuación mostramos la misma prueba pero utilizando archivos de mucho más pequeños, de 250 palabras:



Como puede verse, la diferencia entre ambas versiones en mínima. Por otro lado, no se obtienen mejoras en rendimiento al aumentar los threads. Esto nos da el indicio de que utilizar muchos threads en un programa es útil cuando se realizarán procesos que requieran de mucho tiempo de cpu.

3.2. Segundo Test

Para este segundo test utilizaremos un archivo muy grande (8000 palabras) y un montón de archivos chicos en comparación (2000 palabras). Recordemos que el maximum sin concurrencia asigna una cantidad de archivos fijos a cada thread, por lo que el thread al que se le asigne el archivo grande deberá tardar más que los demás. Asimismo, si se le asignan más archivos, éstos podrían quedarse a la espera de que se procese el archivo grande. En la versión concurrente, mientras que un thread procesa el archivo grande, todos los demás archivos se procesan por el resto de threads.



En este caso se vuelven a apreciar diferencias entre ambas implementaciones, sin embargo es preciso notar que la versión concurrente no gana en rendimiento tan rápido como lo hacía en el test con archivos iguales. Podemos afirmar que el tamaño de los archivos (y por consiguiente la cantidad de palabras) influyen en el rendimiento de 2 maneras: por un lado aumenta el tiempo de ejecución (lo cual es totalmente esperable), y por otro reduce la escalabilidad, es decir que aumentar los threads no impacta en el rendimiento tan notoriamente como antes.

3.3. Tercer Test

Realizamos un test para verificar que la ListaAtomica funcione correctamente.Para lo cual agregaremos los numeros del 1 al 100000 a una lista.Para ello decidimos crear 50 threads que agregarán los números a la lista. Cada uno de ellos agregará 2000 de estos números. Luego de que todos los threads terminen, verificaremos que todos los numeros hayan sido agregados correctamente a la lista. Este test se puede encontrar en el archivo test-alumnos.cpp