Informe Proyecto

Palabra

Facultad de informática

Arquitectura de computadores

Autores:

Ander Serrano

Aitor Malo

Fecha:

Fecha 23 de Diciembre de 2023

Índice:

1. **Portada**
2. **Índice**
3. **Introducción**
4. **Desarrollo del programa en serie**

4.1. Decisiones (Serie)

4.2. Evaluación del programa (Serie)

**5. Versión paralela del programa**

5.1 Pasos para crear el código en paralelo

i) Estrategias evaluadas

ii) Necesidades de sincronización

**6.** **Análisis del rendimiento**

6.1 Metodología

6.2 Región paralela

i) Rendimiento

ii) Reflexión

6.3 Sistema entero

i) Regiones ejecutadas en paralelo

ii) Rendimiento, fa,ef

**7. Conclusiones**

**8. Conclusiones técnicas**

**Introducción**:

El trabajo consta de 2 fases.La primera trata en agrupar en diferentes grupos 211.640 representaciones vectoriales de palabras, utilizando la cercanía de los significados de las palabras. Y la segunda fase,en cambio, pide que se analice la relación de cada grupo con los diferentes campos semánticos de las palabras, ya sean científicos, tecnológicos etc…

Estas dos primeras fases se harán en una versión en serie, donde luego se compilara el código escrito. Y finalmente se creará una versión paralela del código, obteniendo un tiempo de ejecución diferente al de la versión paralela. Luego se valorarán los resultados obtenidos.

**Desarrollo del programa en serie**

En lo que respecta a la versión en serie hemos optado por dejar como estaba el fichero gengrupos\_s.c y hemos completado fun\_s.c. Para completar el último nos hemos guiado en las fórmulas del enunciado proporcionado por el docente en papel (se encuentra en eGela también).

Se ha importado la librería float.h para poder poner el máximo y mínimo valores representables.

Se ha creado una función auxiliar ordenar,que utiliza el algoritmo de la burbuja.La cual nos ayudará en un futuro a buscar la mediana de cada cluster. Ya que ordenará el array de menor a mayor, y podremos obtener la mediana de cada grupo (cluster).

void ordenar (float \*array, int tamano){

for(int i = 0; i < tamano; i++;){

for (int j = 0; j < tamano-1; j++){

if(array[j+1] < array[j]){

float aux = 0.0;

aux = array[j];

array[j] = array[j+1]:

array[j+1] = aux;

}

}

}

}

Este se utiliza en la subrutina analisis\_campos para ordenar el vector de probabilidades. Cabe resaltar que hemos creado el script.txt para compilar y ejecutar la versión en serie, para hacer las pruebas mientras completamos esta primera versión.

**Descripcion del codigo:**

**Función gendist:**

Calcula la distancia euclídea entre dos vectores. Por tanto creamos un bucle for que recorre las dimensiones de cada vector (NDIM=40) para comparar la distancia y la guardamos en la variable dist. Con la cual se le aplicará la raíz del valor obtenido.

double gendist (float \*vec1, float \*vec2){

double dist = 0.0;

for (int i = 0; i < NDIM; i++){

dist += pow(vec1[i] - vec2[i],2);

}

dist = sqrt (dist);

return dist;

}

**Función grupo\_cercano:**

Se utilizan 2 bucles for. El primero para recorrer cada palabra hasta **nvec** y el segundo para recorrer todos los grupos (centroides), compara todas las distancias con los grupos y guarda en **popu**l la menor distancia . Así va guardando en cada **popul [ j ]** la menor distancia de todas las palabras hasta **nvec**.

**Función Silhouette\_simple:**

La primera parte de la función es rellenar el vector **a**, para ello hemos empleado 3 bucles for.

for(int i=0; i < ngrupos ; i++){

float sum= 0.0;

for(int k = 0 ; k < listag[i].nvecg; k++){

for (int j = 0; j < listag[i].nvecg; j++ ){

float dist = (float) gendist(mvec(listag[i].vecg[k]],mvec[listag[i].vecg[j]] );

sum = sum +dist;

}

}

El primero sirve para iterar los grupos (clusters).Y los 2 siguientes sirven para calcular las distancias entre todos los elementos del grupo.

Una vez estén todas las distancias sumadas se dividen entre el número de elementos del grupo elevado a 2, como indica la fórmula proporcionada en egela.

Si el número de elementos del grupo es menor que 2, se guardará el valor 0,0 en la respectiva posición de **a**, si no se guardará la media de las distancias calculadas anteriormente.

sum = sum / (float)pow(listag[i].nvecg,2);

if(lisag[i].nvecg < 2){

sum = 0.0;

a[i]= 0.0;

}

else{

a[i] 0 sum;

}

Para rellenar el vector **b**, hay que calcular la media de todas las distancias entre el centroide **k** y los otros centros.Esto nos da la idea de cuán separado está el clúster **k** de otras.

Para realizarlo hemos utilizado 2 bucles for, que van desde el primer grupo hasta el último (ngrupos).

Sumando todas las distancias y dividiendo al final por el número de grupos que hay.

Hemos tenido en cuenta que la distancia entre el centroide **j** y el mismo es 0.

Para calcular el vector **s** hemos seguido también la fórmula proporcionada. Para conseguir el mayor elemento entre **b**[ i ] y **a**[ i ] hemos utilizado la función fmax.

Por último hemos hecho la media de los valores del vector **s** y lo hemos devuelto.

Con este valor conseguimos saber la calidad de la partición de clústeres, definido así, el rango de este índice es [-1, 1], donde -1 indica una mala partición y el 1 una buena partición.

double media = 0.0;

for(int i = 0 i < ngrupos; i++){

media = (double) media +s[i];

}

media = media / ngrupos;

return media;

**Función analisis\_campos:**

Al principio la función consta de 3 bucles for, el primero sirve para acceder a cada campo desde 0-23, donde en cada iteración se crea un array llamado mediana.

El siguiente bucle accede a todos los grupos de palabras, y en cada iteración crea otro vector llamado **s**.

Finalmente el tercer bucle accede a los índices de las palabras de cada grupo. Metiendo en **s** las probabilidades de las palabras respecto al campo respectivo.

Después ordena el vector **s** mediante una función auxiliar que hemos creado previamente descrita.

Esta función ordenar, ordena el vector mediante el método de la burbuja.

ordenar (s,listag[j].nvecg);

Una vez ordenado el vector **s**, se mete en el vector mediana la mediana del vector **s**. Por lo que para campo, en el vector mediana estarán las medianas de todos los grupos.

Cuando el vector medianas esté completado, hay que mirar cual es la mayor y menor mediana y en qué grupo está.

Finalmente cuando se calcule esto, se guardará en **info\_cam [ i ]**, **i** siendo el campo desde 0-23;

Cabe recalcar que para coger la mediana del vector **s**, hemos utilizado 3 if.

El primero si el número de elementos del grupo es mayor a 1, coge el índice número de elementos del grupo / 2.

El segundo, si el número de elementos es 1, la mediana sería ese mismo elemento.

Y por último si el número de elementos es 0, que guarda el valor 10,0 en **mediana[ j ]**.

if (listag[j].nvecg > 1 ){

mediana[j] = s [listag[j].nvecg/2];

}

if (listag[j].nvecg == 1){

mediana[j] = s[0];

}

if (listag[j].nvecg == 0){

mediana[j] = 10.0;

}

Este último valor sirve para que el programa funcione correctamente, si no, produce errores en los valores.

Esto se ha solucionado poniendo ese valor en el vector mediana cuando un grupo no tiene elementos, y posteriormente cuando se recorre este vector para encontrar el máximo y mínimo.

Sí **mediana [ s ]** = 10.0 , no hace nada y pasa al siguiente elemento.

**Evaluación en serie** La evaluación del programa en serie: tiempos y resultados obtenidos con la versión en serie.

Los resultados obtenidos del programa en serie son los siguientes (segundos) :

**T\_lec =** 3.524096

**T\_clust =** 522.894039

**t\_camp =** 73.843544

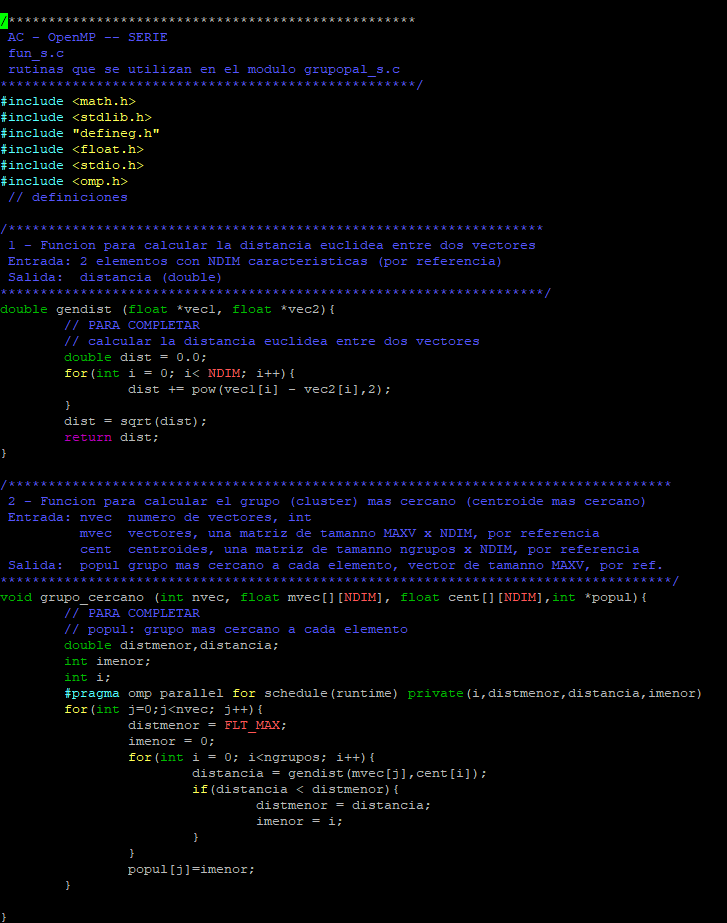
**Texe\_s =** 600.262568

**Versión paralela del programa**

Para cada sección del código describe los pasos seguidos para crear el código en paralelo. Justifica las secciones que se pueden paralelizar y las que no. Asimismo indica:

Para empezar al principio del archivo fun\_p.c que es donde vamos a paralelizar todo el código, pondremos el siguiente trozo de código: #include <omp.h> . Para incluir la librería.

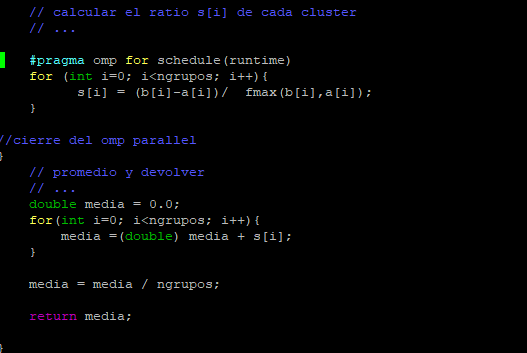
Para la función grupo\_cercano hemos paralelizado con el siguiente trozo de código.



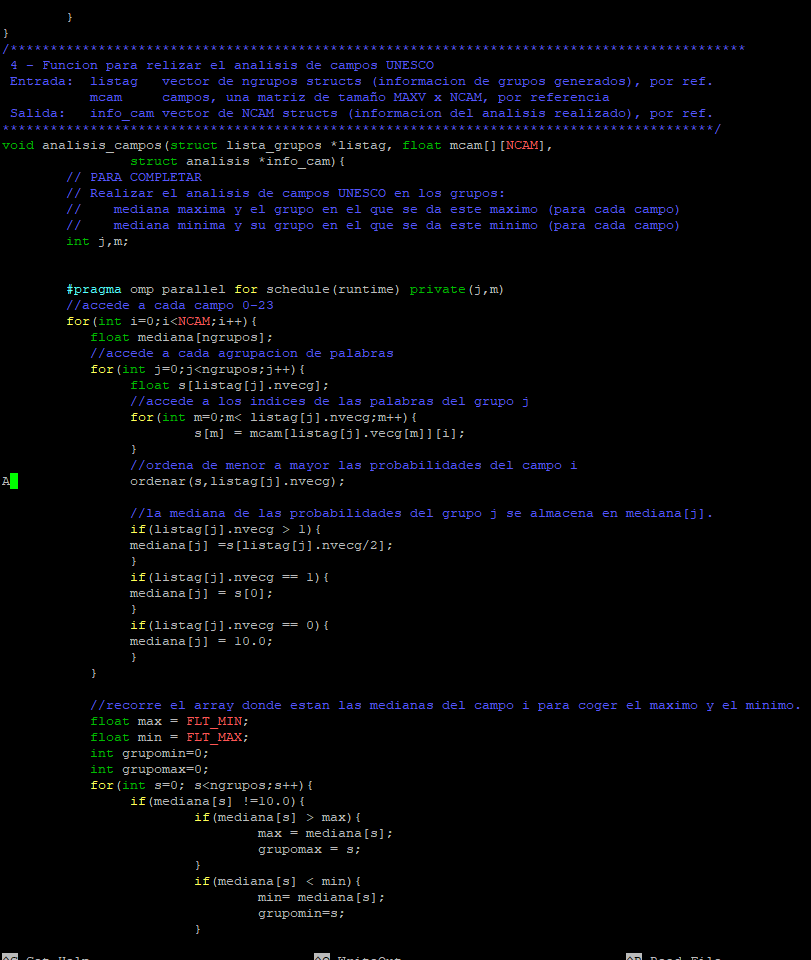
Las variables (i,distmenor,distancia,imenor) son privadas. Y cada hilo crea su propia variable, sin inicializar. Al finalizar la ejecución de un hilo, se borra la copia de la variable en ese hilo.

Para la función silhouette simple hemos paralelizado de la siguiente forma.



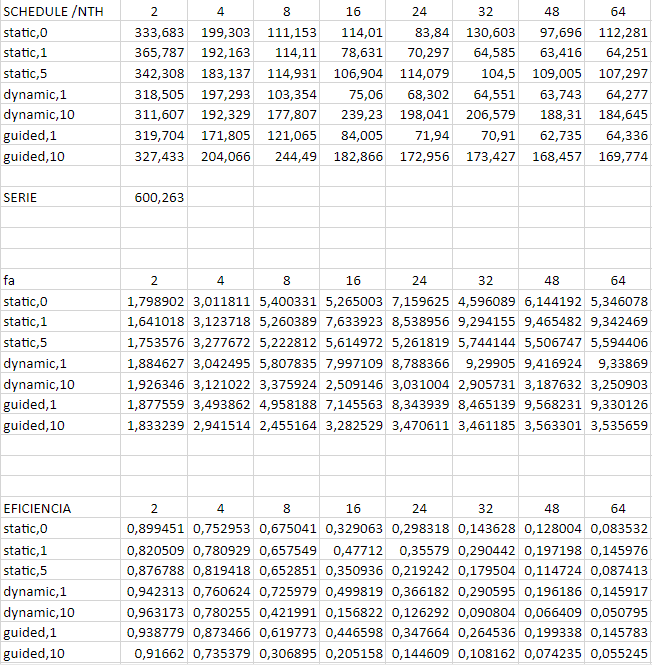


Para la función analisis\_campos , hemos paralelizado de la siguiente manera:



**Análisis del rendimiento:**

**Los resultados de la tabla entera están en segundos.**

****

Como se puede apreciar, tanto **Static 1, Guided 1 como Dynamic 1** tienen unos tiempos de ejecución parecidos a partir de los 24 hilos. Pero que se llega a apreciar mejor a partir de los 32 hilos donde la diferencia ya es de 1 solo segundo. Lo que da a pensar, por lo general, cuantos más hilos se usen, el tiempo de ejecución menor será.

Y mientras que usando más hilos el factor de aceleración aumenta, por otro lado, la eficiencia a medida que vamos utilizando más hilos disminuye, lo cual indica que cuantos más hilos peor eficiencia se consigue.

Con esto , respondemos a la pregunta ¿merece la pena paralelizar?.

Y la respuesta es sí, aunque hace falta indagar un poco y hallar el número de hilos en el que se equilibre la balanza de la eficiencia, y el factor de aceleración. El hecho de poder reducir el tiempo de ejecución de programas que requieren mucho tiempo tanto de compilación como de ejecución. Es muy favorable y hace que merezca la pena totalmente.

**Rendimiento: factor de aceleración y eficiencia.**

Para comparar **guided,1 static,1 y dynamic,1** vamos a hacer un gráfico para que se vea mejor el comportamiento de estos 3, en cuanto a su eficiencia como a su factor de aceleración.

**Factor de aceleración**

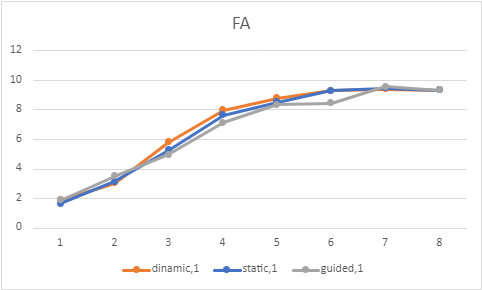


Imagen gráfica fa (Factor de aceleración)

en segundos

(Los valores 1,2,3,4,5,6… son el número de hilos)

**Eficiencia**

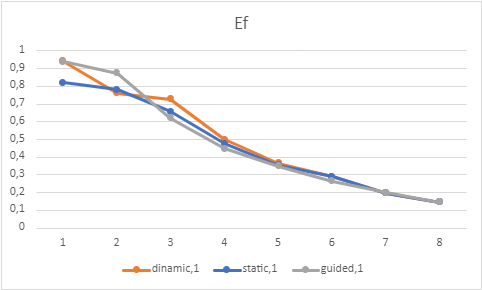


Imagen gráfica ef (Eficiencia)

en segundos

(Los valores 1,2,3,4,5,6… son el número de hilos)

Como previamente hemos mencionado, el comportamiento es prácticamente idéntico en cada uno, habiendo solamente diferencias menores cuando hay pocos hilos.

**Conclusiones:**

Como podemos ver, a medida de que el número de hilos va aumentando, por norma general el factor de aceleración aumenta.

Esto quiere decir que el tiempo de ejecución del programa es menor cuantos más hilos lo ejecuten.

A pesar de esto, no siempre lo más rápido es lo más eficiente.

Como podemos ver en la tabla, la eficiencia va disminuyendo cuando vamos utilizando más hilos.

Siendo el número de hilos más eficiente los 2 hilos.

Como conclusión, ejecutar el programa en paralelo es mejor que en serie.

Ya que se consigue repartir el trabajo, disminuyendo el tiempo de ejecución.

**Conclusiones técnicas:**

Es posible buscar un algoritmo de ordenación más eficiente que el de la burbuja. Ya sea el **MergeSort,** con un coste Θ(n log n) tanto en el mejor como en el peor de los casos , o el algoritmo de ordenación **QuickSort.** Con un coste de **Θ(n log n)** en el mejor de los casos, y con un coste de **Θ(n^2 )** en el peor de los casos.

Mientras que el método de la burbuja tiene un coste de **Θ(n)** en el mejor de los casos, y **Θ(n^2 )** en el peor de los casos**.**

Esto haría que el tiempo de ejecución del programa sea menor.

**Bibliografía:**

https://egela.ehu.eus/pluginfile.php/8882266/mod\_resource/content/0/508\_para\_elaborar\_el\_informe\_tecnico.pdf

https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/