Facultad de Informática
UPV/EHU

Universidad del País Vasco Euskal Herriko Unibertsitatea

Grado en Ingeniería Informática

Arquitectura de Computadores

GII – 2º curso

Proyecto: sistemas multiprocesador

Paralelización de una aplicación utilizando **OpenMP**

Autores: Rubén Castelar, Iker González, Julen Casajús

Índice

Introducción

Durante el primer cuatrimestre hemos trabajado diferentes ámbitos de la paralelización, con el fin de mejorar y optimizar el rendimiento los diferentes programas y ejercicios que se nos han ido presentando.

En este caso, el proyecto se centra en el procesamiento del lenguaje natural (NLP) y aprendizaje automático (ML).

La velocidad de ejecución de un código puede parecer insignificante a pequeñas escalas, pero, cuando se trata de programas que tienen que procesar grandes grupos de datos, cualquier optimización puede resultar en una mejora significativa de tiempo. Este ha sido el principal enfoque de este trabajo.

En primer lugar, hemos desarrollado 4 subprogramas para asegurar la correcta ejecución del proyecto al completo. Mientras trabajamos en esto hemos podido calcular los tiempos de ejecución, que serán importantes para posteriormente comprobar si la versión paralela es eficiente o no.

Después, hemos modificado el programa para implementar la paralelización. Para lograr esto, hemos buscado las secciones críticas del código que menos eficientes son en serie. Además, hemos hecho muchas pruebas para comparar las diferentes formas de paralelizar con el fin de buscar la más óptima.

En conclusión, este proyecto se basa en conocer los beneficios de la paralelización en el contexto del procesamiento del lenguaje natural y el aprendizaje automático. Los resultados obtenidos reflejan las mejoras hechas a lo largo del desarrollo del programa.

Desarrollo del programa en serie

Como hemos mencionado anteriormente, el desarrollo se ha centrado en realizar 4 códigos diferentes para los 4 problemas propuestos.

gendist():
El primero de ellos, calcula la distancia euclídea entre 2 vectores. Para esto, hemos realizado un programa que reciba como parámetros dos punteros, en el caso de nuestro código: ***vec1** y ***vec2**.

Sabiendo que la distancia euclídea entre dos vectores es:

$$d_E(P,Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}.$$

Hemos utilizado esta fórmula para calcular la distancia de cada vector. El número de iteraciones del bucle es **NDIM**, es decir, la dimensión de cada vector (40).

grupo_cercano():
Para el siguiente ejercicio, tenemos como parámetros **nvec**, el número de vectores del ejercicio, **mvec**, matriz de palabras a procesar, **cent**, matriz de centroides, y ***popul**, que indica el cluster más cercano de cada vector. Esto último es lo que debemos calcular.

Para una comprensión más sencilla, dividiremos el ejercicio en diferentes pasos:

1) Realizar un bucle entre 0 y el número de vectores (0..**nvec**).
2) Para cada vector, haremos otro bucle entre 0 y en número de grupos (0..**ngrupos**).
3) Guardamos la distancia actual en la variable **distanciaActual**, obtenida a través de la distancia euclídea (ejercicio 1) entre la palabra **mvec[i]** y el centroide **cent[j]**.
4) Si esta es menor que la distancia que tenemos guardada en la variable **distanciaMinima**, la reemplazamos por la nueva y guardamos el centroide en la variable **Centroide**.
5) Finalmente, para cada índice **i**, guardamos el centroide más cercano en **popul[i]**.

silhouette_simple():
En tercer lugar, debemos calcular la calidad de partición de los clústeres. Este ejercicio se divide en 3 partes.

La primera, calcular la distancia intra-cluster **a(k)**. Esta se calcula sumando la distancia de todos los elementos que pertenecen al clúster, dividido entre el número de ejemplos que pertenecen al clúster **k**. La fórmula proporcionada por el enunciado está sin simplificar, lo cual supone un tiempo de ejecución más lento a grandes escalas, como se puede ver en la siguiente gráfica:

Los dos códigos que representan estas fórmulas son:

```
VERSION NO SIMPLIFICADA:
if (nvecgAux > 1) {
  for(i = 0; i < nvecgAux; i++)
    for(j = 0; j < nvecgAux; j++)
      sumaDistancias += gendist (mvec[vecgAux[i]], mvec[vecgAux[j]]);
    distIntra = sumaDistancias / (nvecgAux * (nvecgAux - 1));
} a[k] = distIntra;

VERSION NO SIMPLIFICADA:
if (nvecgAux > 1) {
  for(i = 0; i < nvecgAux-1; i++)
    for(j = i+1; j < nvecgAux; j++)
      sumaDistancias += gendist (mvec[vecgAux[i]], mvec[vecgAux[j]]);
    distIntra = (2 * sumaDistancias) / (nvecgAux * (nvecgAux - 1));
} a[k] = distIntra;
```

La suma de los valores de las distancias se guarda en la variable **sumaDistancias**, que vuelve a utilizar la distancia euclídea entre los vectores del clúster. En el caso de que el número de ejemplos que pertenecen al clúster sea menor o igual que uno, **a[k]** recibirá el valor 0, en otro caso, recibirá el valor de **SumaDistancias** entre el número de ejemplos.

La segunda parte, será calcular la distancia inter-clúster **b(k)**. Para ello, nos basamos en la fórmula dada y sumamos todas las distancias entre el centroide **k** y el resto de centroides, posteriormente dividiendo esto entre el número de centroides menos 1.

En tercer lugar, calcularemos el término **s(k)**, ratio entre el término **a(k)** y **b(k)**, que nos indica la calidad del clúster **k**. Para ello recorreremos cada grupo con índices desde 0 hasta **ngrupos** y dividimos **(b(k)-a(k))** entre el mayor de estos dos valores.

Finalmente, devolveremos el valor medio de los términos **s(k)**, que representa la calidad de partición de los clústeres.

Nos damos cuenta de que estas 3 partes recorren unos valores desde 0 hasta **ngrupos**, por lo que podemos unir todas las operaciones dentro de un mismo bucle for. Así, podemos evitar tener que crear los arrays **b[ ]** y **s[ ]**, y utilizamos directamente valores double.

analisis_campos():
El último ejercicio, el cuarto, debemos realizar un análisis de campos UNESCO.

Como parámetros de entrada tenemos un puntero de listas de grupos ***listag**, la matriz **mcam**, y otro puntero a estructuras analisis, ***info_cam** , los cuales contiene los resultados del análisis de los grupos.

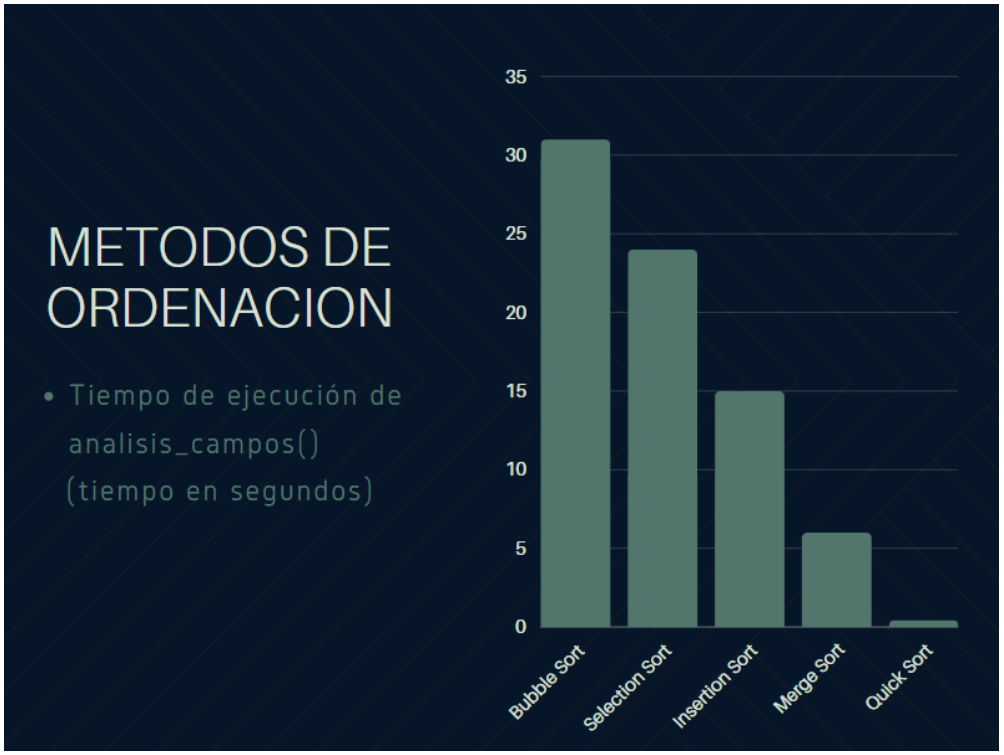
Debemos darle a este análisis los valores de la distancia máxima y mínima de cada campo a los clústeres, y en qué grupos están estos datos. Los valores tomados para determinar la

distancia campo-grupo serán la mediana del array ordenado de palabras dentro de cada grupo.

Para ello, hemos recorrido todos los valores desde 0 hasta el número de campos UNESCO (0..**NCAM**). Dentro de este bucle, recorremos otro bucle desde 0 hasta el número máximo de grupos **ngrupos**.

En cada iteración debemos:

1) Calcular el valor de cada mediana dentro del cluster, para lo cual utilizaremos el método auxiliar de ordenación. Primero, optamos por el método más sencillo que conocíamos, el bubblesort. Tras muchas pruebas e investigación, construimos la siguiente gráfica de eficiencia de diferentes métodos de ordenación:



Concluimos que el método quicksort es considerablemente más eficiente.

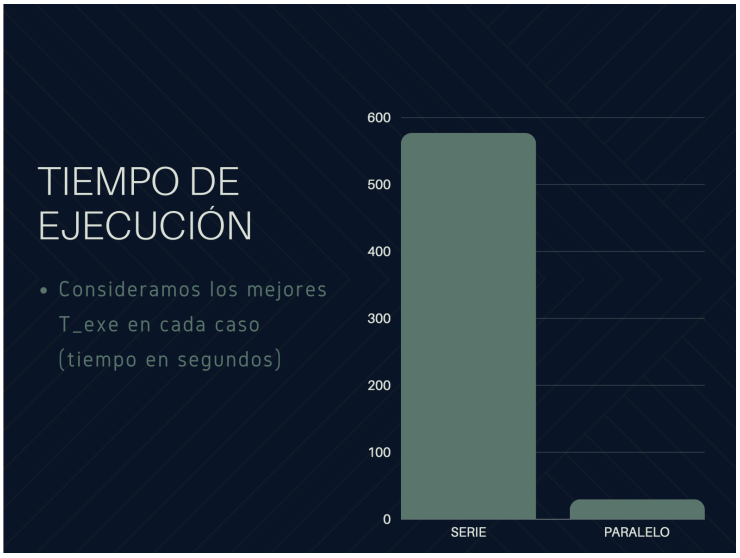
2) Tras obtener el valor de la mediana, comprobamos si ésta es menor que el valor mínimo guardado o mayor que el valor máximo guardado, en cuyo caso cambiamos ese dato y también apuntamos el grupo en el que se encuentra.

3) Finalmente, con los valores máximos y mínimos guardados y los grupos correspondientes, se los asignamos a la variable **info_cam**.

La versión paralela del programa

El último proceso en cuanto a código se refiere, es paralelizar el ejercicio con el fin de conseguir un mejor tiempo de ejecución.

Para comprender la importancia de paralelizar este programa, hemos creado una gráfica comparando el tiempo de ejecución en serie y en paralelo:



El factor de aceleración de la versión paralela es de 19,23 con respecto a la versión en serie.

Para explicar el proceso de paralelización, iremos ejercicio por ejercicio, explicando las decisiones tomadas.

Paralelización de fun_p.c:

gendist():
Comenzando por el primer ejercicio. Después de realizar algunas pruebas con código en paralelo, hemos llegado a la conclusión de que era ineficiente, por tener que trabajar con un número de operaciones (**NDIM**=40) fijo e independiente al número de vectores que se trabajan.

Habíamos probado:

#pragma omp parallel for reduction(+ : sumaCuadrados) private(i) schedule (static),

Pero hemos decidido dejarlo como en serie.

grupo_cercano():
El segundo ejercicio, sin embargo, sí que ha obtenido mejoras. Hemos paralelizado el loop dependiente del número de vectores de la prueba, número elegible que es suficientemente grande como para notar esta diferencia.

En un principio, por tener todas las instrucciones un parecido tiempo de ejecución, pensábamos que lo óptimo sería elegir un **schedule(static)**:

#pragma omp parallel for private(i, j, posCentroide, distanciaMinima, distanciaActual) shared(popul) schedule(static)

Tras hacer las pruebas hemos visto que ambos métodos, static y dynamic, operan en tiempos de ejecución muy parecidos. Por lo que ambos métodos son válidos.

Como las variables **i**, **j**, **posCentroide**, **distanciaMinima** y **distanciaActual** reciben su valor en cada iteración del bucle, y no deben guardar ningún dato, es preciso asignarles el valor "private" para que cada hilo trabaje con sus propias variables.

La variable **popul** es común para cada iteración del bucle, es decir, debemos indicar su valor "shared" para el correcto funcionamiento. Como cada hilo trabajará con un valor **i** diferente, no hay manera de que **popul[i] = posCentroide**;

silhouette_simple():
Para el tercer ejercicio, el código de la versión paralela es muy parecido al del segundo ejercicio. Hemos utilizado un código en paralelo únicamente para calcular la distancia intra-cluster, por depender ésta del número de vectores nvec.

Al igual que anteriormente, pensamos que la mejor forma de paralelizar sería de manera **static**, por realizarse en el bucle una suma y una llamada a otro subprograma de tiempo de ejecución no muy variable.

Pero estábamos equivocados. En este caso la configuración **schedule(dynamic)** sí reduce el tiempo de ejecución por una diferencia a tener en cuenta. Así, el código utilizado es:

#pragma omp parallel for reduction(+ : sumaDistancias) private(i, j) schedule(dynamic)

Con el "reduction", hacemos que los diferentes hilos realicen una operación de reducción, en este caso una suma, de tipo RMW, de forma atómica para calcular **sumaDistancias**. Así evitamos solapamientos al sumar y posibles errores de cálculo.

analisis_campos():
Finalmente, en el ejercicio análisis campos, tras varias pruebas con diferentes métodos, el mejor rendimiento lo ha dado la versión en serie, por lo que hemos decidido no paralelizar el ejercicio. Llegando nvecg a veces hasta 10000 vectores, pensábamos que paralelizando el código referente a nvecg reduciría el tiempo de ejecución, pero no es así.

Paralelización de grupopal_p.c:

Como en todos los subprogramas anteriores, paralelizar variables constantes o de valor relativamente pequeño no sale rentable. La única variable a paralelizar es entonces nvec que, en este proyecto, asciende a más de 200 mil vectores o palabras.
Hemos decidido no paralelizar nada de grupopal, ya que no era eficiente y nos aparecían errores.

Análisis del rendimiento

Para calcular el rendimiento de la paralelización, hemos tenido como referencia el tiempo de la ejecución en serie. El mejor tiempo que hemos obtenido ha sido de unos 563,9 segundos aproximadamente, casi 10 minutos.

Partiendo de esto, los primeros tiempos obtenidos con la paralelización habían sido muy parecidos, por lo que no íbamos bien encaminados. Trás algunos cambios, logramos mejorar el tiempo a 125 segundos (2 minutos), con un factor de aceleración de 4,5.

Para este tiempo, estábamos utilizando el método de ordenación bubblesort y no habíamos hecho pruebas con diferentes “schedule” lo que hacía que la organización de los hilos no fuese la mejor. Además, habíamos paralelizado de forma incorrecta el subprograma analisis_campos(), haciendo que incluso retrasase la ejecución.

Este fué el primer cambio a realizar. Tras algunas pruebas, nos dimos cuenta que no nos merecía la pena paralelizar analisis_campos(), por lo que lo dejamos en serie.

Después, probamos a utilizar “schedule(dynamic)” en los ejercicios paralelizados. Con esto obtuvimos un tiempo de 90 segundos, lo cual supone un factor de aceleración de 6,3 respecto al ejercicio en serie y de 1,4 con respecto a la anterior paralelización.

Pero el tiempo seguía sin ser suficiente, por lo que investigamos acerca de métodos de ordenación. Entonces, decidimos cambiar el método de bubblesort a quicksort, reduciendo el tiempo de ordenación de 30 a 0,45 segundos, con un factor de aceleración de 66,67.

Hemos intentado paralelizar grupopal, pero paralelizar códigos printf o scanf es erróneo pues no se ciñe al orden, y el código que crea los vectores de cada cluster no hemos podido paralelizar sin evitar errores de compilación.

Hemos construido una tabla a partir de las pruebas realizadas con todos los tipos de datos, para comprobar que combinación era la más eficiente. Por cada combinación anotada en la tabla se han realizado al menos 5 pruebas para asegurar buena precisión.

En caso de ser de interés: DATOS PARALELISMO

Conclusiones

Los principales objetivos del proyecto han sido, por un lado, aprender sobre el procesamiento del lenguaje natural, y el aprendizaje automático.

Hemos tratado conceptos que hasta ahora no habíamos escuchado, como por ejemplo clusters, centroides, diferentes representaciones de las palabras y cómo estas pueden estar cerca o lejos entre sí (Distancias intra/inter clúster, calidad de partición de los clusters, etc).

Y por otro lado, la gran mejora que supone paralelizar de forma óptima un código. Como hemos mencionado a lo largo del informe, la mejora total que hemos obtenido respecto al programa en serie, ha sido de un factor aceleración de 12. Es decir, el programa, una vez paralelizado tarda 12 veces menos en ejecutarse. Con los datos que hemos trabajado, la diferencia de tiempo es simplemente de minutos, pero con programas a grandes escalas, esta mejora marca una gran diferencia.

Conclusiones técnicas y de otros tipos

Ha habido un factor que ha hecho que calcular los tiempos de ejecución no haya sido tan fácil como pensábamos. El servidor (arqXX@dif-cluster.ehu.es) se saturaba dependiendo de cuánta gente había conectado a él, por lo que el tiempo de ejecución era mayor. Tanto que la ejecución en serie ha llegado a tardar 40 minutos. Había que conectarse bien a primeras horas de la mañana bien a últimas horas de la noche para poder calcular los tiempos sin este “problema” añadido.

Más allá de eso, hemos aprendido nuevos términos y conceptos que seguro serán útiles de aquí en adelante, desde que es un clúster, hasta la mejor manera de optimizar cada programa.