UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Computación de Altas Prestaciones

<Solución de un problema mediante técnicas de paralelización>

Autores: Miguel de la Cal Bravo y Félix Ángel Martínez Muela

Titulación: Máster en Ingeniería Informática

Fecha: 07/06/2020

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN

En esta memoria se explica y desarrolla el ejercicio final de la asignatura, para el cual tenemos el objetivo de utilizar una serie de tecnologías de paralelismo, siguiendo procesos de **discretización** de *Machine Learning*, para la reducción de datos.

Gracias a los procesos de discretización, podremos dividir el rango de atributos continuos en intervalos almacenando solamente las etiquetas asociadas a los mismos. Además, resultan ser de gran importancia para aplicar reglas tanto de asociación, como de clasificación.

El enlace al repositorio es el siguiente: https://github.com/miguelcal97/hpc_ejerciciofinal

En dicho repositorio, encontramos un programa en lenguaje de programación ***C*** llamado **discretizo.c**, el cual se encarga tanto de inicializar el vector de *N* posiciones de forma aleatoria, como de aplicar aquellas técnicas y tecnologías de paralelismo estudiadas a lo largo de esta asignatura.

Adicionalmente, se ha desarrollado otro programa llamado **discretizo_hybrid.c**, donde se explotan más alternativas de paralelismo aplicando una estrategia diferente y más completa.

Finalmente, y para facilitar su compilación, se ha creado un fichero *Makefile* que automatice esta tarea antes de ejecutar nuestro programa resultante. También, dentro del repositorio se ha creado un fichero *README* que explica la compilación y ejecución del programa.

En los siguientes apartados, se describen los algoritmos desarrollados y paralelismos aplicados para un programa cuyo objetivo consiste en que, dado un vector *v* que contiene las edades de *N* ciudadanos[1], discreticemos los datos para un total de cuatro grupos de edades, aportando como resultado un vector del mismo tamaño que el original, pero en lugar de aparecer los datos de entrada, aparece el grupo al que pertenecen dichos datos.

2. VERSIÓN PARALELA DEL ALGORITMO

En esta sección se explica el planteamiento realizado para la versión paralela del algoritmo, como solución al problema dado.

Inicialmente, tendremos un vector de entrada *v*, el cual, repartiremos entre los diferentes elementos de proceso de los que dispongamos, teniendo en cuenta que cada proceso trabaja con un subconjunto de los datos de entrada siendo éstos repartidos de manera equitativa.

Para ello, será necesario disponer de varios hilos, pudiendo así ejecutar nuestra aplicación de forma paralela, aprovechando sus ventajas.

En los siguientes apartados, discutiremos la versión paralela del algoritmo que hemos diseñado para la resolución del problema, enfocándonos en la descomposición, mapeo y agrupaciones.

[1] Las edades de los ciudadanos se encuentran entre 0 y 95 años.

2.1 Descomposición en tareas y estrategia

La descomposición del problema de los datos de entrada en tareas se ve determinada por el número de intervalos de edades que se desean formar, en nuestro caso, al tener cuatro intervalos, se generarán tantas tareas como hilos tengamos, los cuales se repartirán una parte del vector y comprobarán cada uno los grupos a los que pertenecen la parte del vector asignada a dicha tarea.

Por tanto, podemos afirmar que se trata de una descomposición de datos, y más específicamente una **descomposición de los datos de entrada**.

Nuestra estrategia consistirá en hacer uso de la tecnología ***OpenMP*** para lograr la paralelización en todo lo posible del programa desarrollado. Gracias a las librerías que importamos con *omp.h*, podemos paralelizar nuestro programa haciendo uso de las directivas que comienzan con ***#pragma omp X***, donde *X* será la/s cláusula/s y opciones que deseemos utilizar.

Además, también aprovecharemos las funciones que ofrece *OpenMP* de **omp_get_num_threads()** y **omp_set_num_threads()** para obtener el número de hilos del sistema, así como cambiar en tiempo de ejecución el número total de hilos que deseamos emplear en el programa en un momento dado.

Otra función muy útil para la medición de tiempos es **omp_get_wtime()**, que utilizaremos antes y después de la inicialización y discretización del vector, para poder medir la diferencia y determinar los tiempos de cómputo en ambas fases.

En el apartado *3. Descripción e implementación del algoritmo*, se amplía la explicación de las distintas directivas utilizadas con *OpenMP* para el paralelismo de nuestro programa.

2.2 Análisis de tamaños e interacciones entre tareas

En el problema planteado, se parte de un vector *v* con tamaño *N* definido antes de su inicialización, en la que no se producen interacciones entre tareas.

En la parte correspondiente a la clasificación o discretización, no se producirán interacciones entre tareas, ya que el vector inicial será repartido de forma equitativa entre los diferentes hilos que tengamos. Así, si tuviéramos un total de cuatro hilos, cada uno de ellos se encargaría de procesar *N/4* elementos del vector de edades.

De esta manera, también podríamos decir que cada hilo se correspondería con un proceso, el cual se encarga de procesar una cuarta parte de todos los elementos del vector.

2.3 Mapeo de tareas y agrupaciones

En la línea del subapartado anterior, el mapeo de tareas tendrá que ver con el número de hilos en los que podamos agrupar el trabajo.

En este caso, a cada hilo o elemento de proceso podríamos mapearle un grupo de elementos del vector inicial, que quedaría repartido de la manera más equitativa posible en base al número de edades de *N* ciudadanos del vector.

En la imagen de la *Figura 1*, podemos observar gráficamente el ejemplo anteriormente mencionado para un total de cuatro hilos.

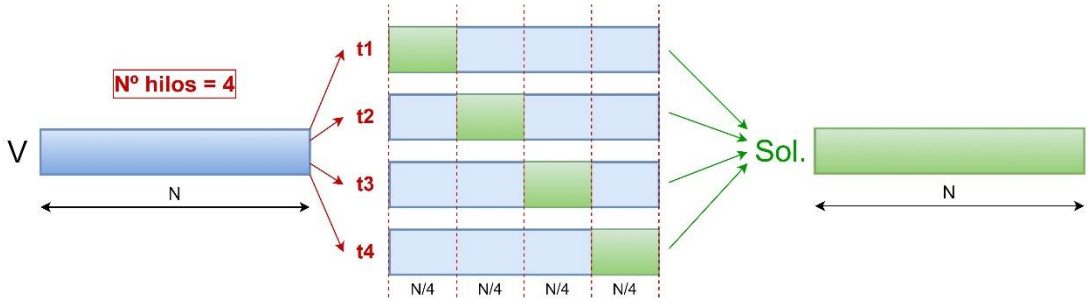

FIGURA 1. EJEMPLO DE PARALELISMO CON CUATRO HILOS

Vemos que partimos del vector inicial *v*, el cual dividimos en cuatro tareas y cada una de ellas se encarga de computar su parte correspondiente del vector solución resultante. Al repartir el trabajo entre diferentes hilos, podremos obtener la solución final en menos tiempo aprovechando esta estrategia de paralelismo

Una vez que cada tarea haya terminado, se conforma el vector con la solución final, compuesto por las soluciones parciales obtenidas en la fase anterior. Como se observa, dicho vector solución tiene el mismo tamaño que el vector inicial, pero en este caso almacena el grupo al que pertenece cada elemento del vector inicial.

Hecho esto, tendremos nuestro problema resuelto, entendiendo cómo se mapean las tareas y sus agrupaciones utilizando la tecnología de *OpenMP*.

A la hora de acceder a almacenar la solución en el vector de salida, se ha tenido en cuenta el paralelismo de manera que los hilos puedan acceder de manera concurrente al vector solución y almacenar los valores, ya que los valores del vector de entrada van a coincidir con las posiciones del vector salida y, por tanto, el valor quedará guardado por la tarea encargada de clasificarla en un grupo.

2.4 Decisión paradigma de programación

A la hora de enfrentarnos a la decisión de optar por un paradigma de **paso de mensajes (MPI)**, o **compartición de memoria (OpenMP)**, optándose por el paradigma de compartición de memoria (ver *Figura 2*).

Se ha optado por esto debido a que, de esta manera, el acceso a la solución es posible de manera concurrente por parte de los hilos y sin que se tenga que añadir código extra para el *merge* del vector final.

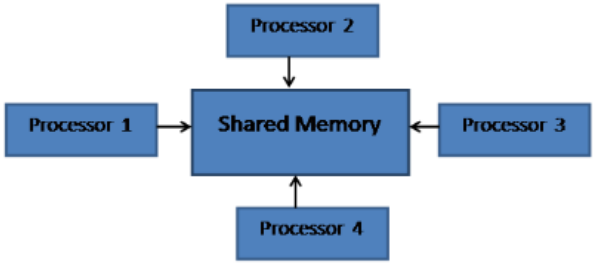

FIGURA 2. PARADIGMA P. OPENMP

Evitaremos también los *overheads* al tener acceso a la memoria compartida y el permitir la modificación del propio vector directamente por los hilos.

OpenMP nos permite discernir entre memoria privada y compartida, por lo que en los casos que se ha necesitado que cada hilo tenga su propia memoria, se ha podido conseguir de manera sencilla, aportando incluso los *reduction* para pasar de una memoria privada a una compartida automáticamente.

También el hecho de que los hilos de ejecución sea algo inherente al código nos permite una ejecución mucho más limpia y rápida, ya que no tenemos que decidir cuantos hilos van a ser los encargados de ejecutar dicho proceso, sino que será el propio código el que se encargue de solucionar estos problemas.

3. DESCRIPCIÓN E IMPLEMENTACIÓN DEL ALGORITMO

El programa *discretizo.c* encontramos desarrollado los algoritmos y técnicas de *OpenMP* empleadas para obtener la paralelización de nuestro problema.

Inicialmente se importan las librerías y se definen las constantes que definirán el tamaño del problema a resolver. Algunas de ellas, las modificaremos en el apartado *4. Estudio de tiempos de ejecución*, con el fin de estudiar el comportamiento del algoritmo implementado.

Tras ello, se inicializa el vector *v* con las edades de *N* ciudadanos de forma aleatoria, y, seguidamente, se procede a discretizar el vector en los diferentes grupos de edades.

Aquí es donde entra el uso de *OpenMP* y sus directivas, las cuales explicaremos a continuación en función de su utilización en el programa desarrollado.

Entrando en la paralelización de bucles, en el primer bucle *for*, para la inicialización del vector de edades, utilizaremos ***#pragma omp parallel for***, para así aprovechar los hilos inicializando el vector empleando todos ellos y acelerar su inicialización.

En cuanto a la discretización del vector, hemos utilizado ***#pragma omp parallel for***, con las opciones de *schedule* (indicando el tamaño de cada trozo o *chunk*) y *reduction* para las operaciones de suma de las variables que almacenan el número de elementos por cada grupo, paralelizando en diferentes trozos el conteo de elementos de cada grupo de edades.

A la hora de realizar una suma de los grupos que forman el vector final, se ha decidido optar por una solución basada en ***reduction***, ya que, de esta manera, cada hilo tendrá una copia local de dicho valor, la cual cuando finalice el bucle *for* se pondrá en común con los demás hilos y se realizará la correspondiente suma, de esta manera se evitan las colisiones y valores erróneos en caso de dos hilos acceder al mismo valor de memoria.

Para terminar, se imprimirán los parámetros de la ejecución establecidos junto con los tiempos de cómputo para la fase de discretización del vector inicial. También, se comprueba que la suma total del número de elementos de cada grupo se corresponde con el tamaño *N* del vector.

Solución alternativa - discretizo_hybrid.c

Para hacer un mayor uso de las directivas que nos ofrece *OpenMP*, se ha desarrollado una solución alternativa que contempla más posibilidades de paralelismo con esta tecnología.

Se ha decidido implementar un programa que ejecute una solución de paralelismo basada en datos de entrada y de salida:

El paralelismo de **datos de entrada** parte de la premisa de dividir el vector inicial en tantas partes (*chunks*) en función del número de hilos encargados de ejecutar dicho programa, de manera que esa parte es igual a la del programa *discretizo.c*.

Sin embargo, el paralelismo de **datos de salida** se ha realizado, para cada hilo del vector de entrada se generen otros dos hilos, los cuales serán los encargados de revisar al grupo que puede pertenecer dicho valor, de manera que se produce una ejecución especulativa para comprobar los grupos, y solo en caso de que sea dicho grupo, se añadirá al grupo correcto, quedando descartadas las ejecuciones que no nos lleven a un resultado correcto.

Al darnos cuenta de que la solución que hemos planteado no mejora en nada la solución de la partición de los datos de entrada, se ha decidido optar por explicar en más detalle la otra y mantener esta como una solución alternativa, pero bastante peor a la otra en cuanto a ejecución de esta.

4. ESTUDIO DE TIEMPOS DE EJECUCIÓN

Una vez explicado y analizado la solución propuesta para el problema plantado, se llevará a cabo un estudio de los tiempos de ejecución a lo largo del programa desarrollado, variando aquellos parámetros que se consideren convenientes para confeccionar el informe de tiempos.

El parámetro principal que podremos variar se trata del **tamaño *N*** del vector de edades, ya que al aumentar o disminuir este número los tiempos de cómputo variarán de una ejecución a otra.

Adicionalmente, introduciremos una variante en el algoritmo para ejecutar el programa cambiando el **número de hilos**, con lo que observaremos la evolución de tiempos y paralelismo en base al tamaño del problema con un número de hilos determinado.

Para que las prueban sean más justas, se ha decidido mantener el mismo vector de números aleatorios para todas las pruebas que impliquen el mismo tamaño del vector, en conclusión, manteniéndose para distintos números de hilos y variando para distinto tamaño de vector.

Al final del programa desarrollado, se imprimirán los resultados de tiempos correspondientes a la de fase de discretización, para un tamaño *N* y un número de hilos determinado.

Dichos resultados los hemos recopilado en un fichero *Excel*, mediante el que realizaremos medias de tiempos entre diferentes ejecuciones y pruebas del programa, con el fin de obtener unos resultados más fiables de unas pruebas a otras.

A continuación, en la *Tabla 1* se muestran los tiempos de ejecución recogidos en las pruebas:

Vector size / #Cores	Test #1	Test #2	Test #3	Test #4	Test#5	Avg Time
10000 / 1	0,0993	0,1000	0,0996	0,1030	0,0995	**0,10**
10000 / 2	0,0568	0,0657	0,0708	0,0583	0,0513	**0,06**
10000 / 3	0,0379	0,0368	0,0344	0,0366	0,0593	**0,04**
10000 / 4	4,0062	0,0302	0,0305	0,0275	0,0286	**0,82**
100000 / 1	1,1438	1,1181	1,1200	1,1361	1,1218	**1,13**
100000 / 2	0,5071	0,5095	0,7414	0,5706	0,5519	**0,58**
100000 / 3	0,3334	0,8456	0,3445	0,3313	0,3231	**0,44**
100000 / 4	3,5896	0,2456	0,2647	0,2598	0,2445	**0,92**
1000000 / 1	11,5724	11,3600	11,3511	11,3200	11,4600	**11,41**
1000000 / 2	6,1123	6,2400	5,8403	5,8099	5,7888	**5,96**
1000000 / 3	9,1745	3,4546	3,4486	3,4888	3,7198	**4,66**
1000000 / 4	8,7104	2,4168	2,5731	2,5925	2,6914	**3,80**
10000000 / 1	112,2085	128,4866	112,9446	112,4656	113,7732	**115,98**
10000000 / 2	56,3574	56,8692	56,6071	56,6273	56,4762	**56,59**
10000000 / 3	33,7291	34,1714	33,6507	35,1465	33,6362	**34,07**
10000000 / 4	25,8223	25,9997	25,6106	26,1580	26,0605	**25,93**

TABLA 1. RESULTADOS DE TIEMPOS DE EJECUCIÓN (EN MS) DE LOS TESTS

En la *Tabla 2*, se muestran las ganancias (*speedup*) de utilizar paralelismo multihilo, frente a una ejecución secuencia con un único hilo:

Speedup	Test #1	Test #2	Test #3	Test #4	Test#5	Avg Speedup
10000 / 2 VS 10000 / 1	175%	152%	141%	177%	194%	**166%**
10000 / 3 VS 10000 / 1	262%	272%	290%	281%	168%	**245%**
10000 / 4 VS 10000 / 1	2%	331%	327%	375%	348%	**12%**
100000 / 2 VS 100000 / 1	226%	219%	151%	199%	203%	**196%**
100000 / 3 VS 100000 / 1	343%	132%	325%	343%	347%	**259%**
100000 / 4 VS 100000 / 1	32%	455%	423%	437%	459%	**122%**
1000000 / 2 VS 1000000 / 1	189%	182%	194%	195%	198%	**192%**
1000000 / 3 VS 1000000 / 1	126%	329%	329%	324%	308%	**245%**
1000000 / 4 VS 1000000 / 1	133%	470%	441%	437%	426%	**301%**
10000000 / 2 VS 10000000 / 1	199%	226%	200%	199%	201%	**205%**
10000000 / 3 VS 10000000 / 1	333%	376%	336%	320%	338%	**340%**
10000000 / 4 VS 1000000 / 1	435%	494%	441%	430%	437%	**447%**

TABLA 2. SPEEDUP O GANANCIA OBTENIDA CON PARALELISMO MULTIHILO

A primera vista, podemos observar que cuanto más pequeño es el tamaño del vector, menor es la ganancia respecto a la ejecución secuencial, ya que no compensa el ***overhead*** que se necesita para sincronizar los diferentes hilos de ejecución respecto a que la realice un solo hilo secuencialmente. En ocasiones, esta causa será un gran problema en la ejecución.

A medida que va creciendo el tamaño del vector y el número de hilos, observamos una ganancia cada vez mayor. Aquí el *overhead* inicial deja de ser un problema tan notable como al principio, ya que el tamaño del problema a medida que se crecen, mejor aprovecharemos los hilos disponibles en nuestros paralelismos.

NOTA: tanto el Excel, como los ficheros .txt con los resultados de tiempos de ejecución, se adjuntan en el repositorio de Github del proyecto, en la carpeta /results.

De esta manera, completamos el apartado correspondiente a los tiempos de ejecución del problema planteado, variando los diferentes parámetros relevantes en el programa.

Podemos concluir afirmando que el paralelismo es muy útil a medida que el tamaño de nuestro problema crece, aprovechando las ventajas que nos ofrece el desarrollo de aplicaciones paralelas, en nuestro caso gracias a la tecnología de *OpenMP*.

5. CONCLUSIONES

Antes de dar por terminado este trabajo, daremos nuestras conclusiones y opiniones personales sobre las tecnologías y técnicas utilizadas para resolver el problema.

En primer lugar, hemos optado por utilizar la tecnología que nos ofrece *OpenMP* debido a su variedad y versatilidad para lograr buenos resultados con técnicas de paralelismo basada en **memoria compartida**. Gracias a su *API* portable, obtenemos una interfaz muy flexible para desarrollar aplicaciones paralelas de forma sencilla.

Gracias al paralelismo, podemos dividir un problema en subproblemas o tareas más pequeñas, las cuales serán asignadas a diferentes procesos obteniendo así mejores resultados en cuanto a tiempos de cómputo.

Además, se ha observado que, a medida que el tamaño del problema crece, las mejoras obtenidas gracias a las tecnologías de paralelismo son más notables, dado que se reduce el *overhead* necesario para sincronizar los procesos.

Tras probar diferentes combinaciones y posibilidades de paralelismo, hemos logrado no solo pensar en cómo resolver un problema de forma paralela, sino de llevarlo a la práctica en este trabajo siguiendo una estrategia, con una descomposición de datos, mapeos y agrupaciones determinadas explorando el amplio abanico de posibilidades dentro del mundo de las aplicaciones paralelas.

Así, concluimos afirmando y reiterando la gran importancia del paralelismo hoy en día, permitiéndonos aprovechar al máximo nuestros recursos acelerando la computación en nuestras aplicaciones para computación de altas prestaciones.