UCLM – Escuela Superior de Informática de Ciudad Real

Discretización de Datos - Trabajo Final

Computación de Altas Prestaciones

Rubén Pérez Rubio y Lydia Prado Ibáñez
7-6-2020

Índice

Introducción al problema

Todo el código referenciado en este informe se puede encontrar en el siguiente repositorio de Github: https://github.com/RubenPerez95/TrabajoFinalDiscretizacion.

El objetivo de este trabajo consiste en solucionar un problema de discretización de datos mediante el empleo de técnicas de paralelización. Este problema tiene en cuenta la utilización de una matriz de tamaño variable de edades que vayan comprendidas entre 0 y 95 y se debe realizar la discretización mediante la clasificación de estas edades en 4 grupos diferentes de edades: infantil, joven, adulto, anciano. A continuación, se muestran los rangos de edades a los que hace referencia cada grupo:

- Infantil: de 0 a 14
- Joven: de 15 a 24
- Adulto: de 25 a 64
- Anciano: mayor o igual a 65

En este documento, se muestran las estrategias seguidas y pruebas que se han realizado para conseguir una paralelización de los bucles que generen las salidas deseadas. En la sección *Descripción del algoritmo*

En esta sección se comentan las funcionalidades que se han implementado para mejor entendimiento del programa y cómo se han enfocado las salidas que debe producir.

En el programa la idea principal es la implementación de la recepción del tamaño que se le va a asignar al vector de edades a través de la línea de comandos. Esto permite realizar varias pruebas acerca de la efectividad de la paralelización con tamaños pequeños y grandes y si llega a ser efectiva su implementación.

Después, se han implementado dos tipos de bucles enfocados a dos tipos de salidas. Una salida genera un vector del mismo tamaño que el vector de entrada, en donde se sustituyen los valores de las edades por caracteres haciendo referencia a los rangos de edades, junto con la posición que ocupan en el vector original. Estos son los caracteres asignados a cada rango:

- Infantil: I
- Joven: J
- Adulto: A
- Ancianos: M

Se han implementado dos métodos aparte, uno por cada tipo de ejecución, secuencial y paralelo. Junto a esto, se ha añadido un método más, llamado *comparar*Vectores, que permite

comparar el vector de salida secuencial con el vector de salida paralelo para comprobar que las salidas sean coincidentes y no haya incongruencias en los valores generados.

La otra salida que se llega a generar es un vector de 4 posiciones en los cuales en cada una de ellas irá almacenado la cuenta total de un tipo específico de edades que existen en el vector original. Es decir, por ejemplo, en la primera posición irá almacenada la cuenta total de personas de rango infantil que se encuentren en el vector original.

Además, se han implementado los típicos métodos para las impresiones de los bucles para visualizarlos de una forma más efectiva y clara.

Pasos realizados para la paralelización se pueden ver más en detalle las salidas que se han deseado generar con la discretización de los datos y qué pruebas se han efectuado para obtener un rendimiento mejor. Finalmente, se muestran las mejoras de tiempo de ejecución y los gráficos de speed-up con respecto a cada prueba.

También se muestran las especificaciones del sistema que se ha utilizado para la resolución de este problema, puesto que esto es algo muy para tener en cuenta a la hora de valorar los resultados obtenidos frente a los obtenidos en equipos alternativos.

Especificaciones del sistema

Los resultados se han obtenido ejecutando el programa desarrollado en una máquina virtual con las siguientes características:

- Entorno de virtualización: Oracle VM VirtualBox
- RAM: 6GB
- Procesador: Intel i7-8750H @ 2.20Hz
- N.º de núcleos: 1
- Sistema operativo: Ubuntu 18.04.4 LTS

Descripción del algoritmo

En esta sección se comentan las funcionalidades que se han implementado para mejor entendimiento del programa y cómo se han enfocado las salidas que debe producir.

En el programa la idea principal es la implementación de la recepción del tamaño que se le va a asignar al vector de edades a través de la línea de comandos. Esto permite realizar varias pruebas acerca de la efectividad de la paralelización con tamaños pequeños y grandes y si llega a ser efectiva su implementación.

Después, se han implementado dos tipos de bucles enfocados a dos tipos de salidas. Una salida genera un vector del mismo tamaño que el vector de entrada, en donde se sustituyen los valores de las edades por caracteres haciendo referencia a los rangos de edades, junto con la posición que ocupan en el vector original. Estos son los caracteres asignados a cada rango:

- Infantil: I
- Joven: J
- Adulto: A
- Ancianos: M

Se han implementado dos métodos aparte, uno por cada tipo de ejecución, secuencial y paralelo. Junto a esto, se ha añadido un método más, llamado *compararVectores*, que permite comparar el vector de salida secuencial con el vector de salida paralelo para comprobar que las salidas sean coincidentes y no haya incongruencias en los valores generados.

La otra salida que se llega a generar es un vector de 4 posiciones en los cuales en cada una de ellas irá almacenado la cuenta total de un tipo específico de edades que existen en el vector original. Es decir, por ejemplo, en la primera posición irá almacenada la cuenta total de personas de rango infantil que se encuentren en el vector original.

Además, se han implementado los típicos métodos para las impresiones de los bucles para visualizarlos de una forma más efectiva y clara.

Pasos realizados para la paralelización

Para comenzar, se ha utilizado para la resolución del problema la tecnología de paralelización de OpenMP. Esto se ha tenido en cuenta desde el principio puesto que es necesario plantearlo para poder enfocar la resolución de manera más efectiva y estudiar las diferentes estrategias que se pueden aplicar. Una vez se han implementado los métodos para generar las salidas de forma secuencial, se han estudiado las diferentes directivas a aplicar para comprobar su efectividad en los tiempos de ejecución.

Se ha tenido en cuenta que la ejecución del programa se ha realizado bajo las condiciones de una máquina virtual, en cual contenía un solo núcleo, por lo que es necesario establecer el número de núcleos para permitir la paralelización. Por lo general, en las computadoras convencionales y de normal uso se utilizan 4 núcleos, por eso, se ha establecido este número de núcleos mediante la función de *omp_set_num_threads(4)*. Esto permitirá dividir la tarea principal que es cada bucle que genera cada salida en subtareas más pequeñas, asignando el bucle a cada núcleo.

Como se ha comentado en la sección anterior, una vez se han desarrollado los bucles secuenciales, se han desarrollado dos bucles de forma paralela, en los cuales se han implementado varias pruebas para cada uno mediante combinaciones de primitivas para observar la viabilidad y mejoras en los tiempos de ejecución con respecto a aquellos de los bucles secuenciales. Se ha de destacar que en este documento se harán referencia a las combinaciones de primitivas a través de otro tipo de nomenclatura puesto que son un tanto complejas de mencionar cuando se vean más adelante los resultados generados por cada una. A continuación, se muestran las transformaciones en las nomenclaturas:

- Bucle de posiciones:
  - #pragma omp for ordered ➔ Prueba 1
  - #pragma omp parallel for shared(edades, vectorParaleloSalidaIntervalo, vectorEdades) private(i) ➔ Prueba 2
  - #pragma omp for ➔ Prueba 3
- Bucle contador:
  - #pragma omp for ordered ➔ Prueba 1
  - #pragma parallel for default(none) shared(edades, vectorEdades, totalRangosParalelo) schedule(static) ➔ Prueba 2
  - #pragma omp parallel for shared(edades, vectorEdades) private(i) reduction(+:totalRangosParalelo) ➔ Prueba 3
  - #pragma omp for ➔ Prueba 4

Como se puede observar, estas son las primitivas que se han llegado a probar teniendo en cuenta la naturaleza de cada bucle que se quería paralelizar. Para el caso del primer bucle, se han utilizado las primitivas simples de ordered y for, para comprobar la efectividad de la paralelización. En la prueba 2 se puede ver como se han usado más primitivas las cuales sirven para especificar que variables son compartidas entre todas las tareas y se privatiza el índice *i* del bucle, para que las tareas no lo reutilicen y acaben produciendo valores incorrectos o distintos al bucle secuencial, ya que pueden existir dependencias entre los datos.

Para el caso del bucle contador, se han vuelto a emplear las directivas ordered y for con el mismo propósito. También, al ser el cálculo de operaciones enteras, se ha utilizado la primitiva reduction, la cual permite realizar los cálculos recurrentes de forma paralela. También se ha utilizado la primitiva schedule para planificar la distribución de las tareas, habiendo probado con las variables static, dynamic y guided. Con la que mejor resultados se obtenían era con static, por lo que se ha dejado como la opción por defecto y la que se verá más adelante en los análisis de los resultados.

Finalmente, estos bucles han sido medidos mediante el uso de la función de *omp_get_wtime()*. Se han insertado uno antes del cada bucle y otro después y se ha aplicado la diferencia de la segunda a la primera para obtener el tiempo total. Se ha podido observar que, puesto que en este problema se realizan cómputos sencillos y de forma trivial, los tiempos han sido extremadamente pequeños, por lo que se han adaptado a que se generen en microsegundos en lugar de segundos, ya que en segundos se generaban datos con demasiados ceros en los decimales y haciendo imposible las comparaciones entre los tiempos secuenciales y los paralelos.

Resultados

En esta sección, se muestran los resultados obtenidos en cuanto a los tiempos de ejecución obtenidos, sus medias, desviaciones estándar y los speed-ups, junto con las gráficas a modo de mejor entendimiento. Se ha de tener en cuenta que se han realizado 5 repeticiones por cada prueba de directivas de paralelización OpenMP y secuencial. Finalmente, lo que se obtienen son sus medias y se plasman en las gráficas.

En la Ilustración 1 se exponen las salidas para el bucle que asigna a cada posición del vector original con su rango de edad correspondiente. Como se puede ver, el vector original muestra la edad generada para cada posición y los dos resultados, tanto de la ejecución secuencial como de la ejecución en paralelo, muestran el rango de edad al que pertenece cada edad de dicho vector.

```
Valores generados para el vector de edades:
Posición 0(74) Posición 1(84) Posición 2(54) Posición 3(86) Posición 4(93) Posición
5(16) Posición 6(5) Posición 7(38) Posición 8(28) Posición 9(65) Posición 10(17) Pos
ición 11(27) Posición 12(16) Posición 13(4) Posición 14(49) Posición 15(22) Posición
 16(46) Posición 17(80) Posición 18(56) Posición 19(67)

Resultado de la ejecución secuencial del vector:
Posición 0(M) Posición 1(M) Posición 2(A) Posición 3(M) Posición 4(M) Posición 5(J)
Posición 6(I) Posición 7(A) Posición 8(A) Posición 9(M) Posición 10(J) Posición 11(A
) Posición 12(J) Posición 13(I) Posición 14(A) Posición 15(J) Posición 16(A) Posició
n 17(M) Posición 18(A) Posición 19(M)

Resultado de la ejecución en paralelo del vector:
Posición 0(M) Posición 1(M) Posición 2(A) Posición 3(M) Posición 4(M) Posición 5(J)
Posición 6(I) Posición 7(A) Posición 8(A) Posición 9(M) Posición 10(J) Posición 11(A
) Posición 12(J) Posición 13(I) Posición 14(A) Posición 15(J) Posición 16(A) Posició
n 17(M) Posición 18(A) Posición 19(M)
```

Ilustración 1. Salida del bucle posiciones.

En la Ilustración 2 se muestra el resultado del bucle que cuenta cuántas edades hay para cada uno de los rangos.

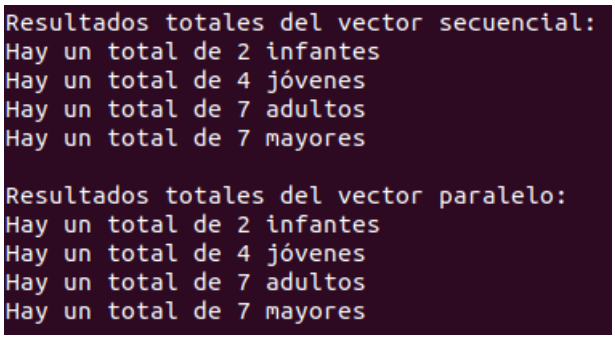

Ilustración 2. Salida del bucle contador.

Para las dos ilustraciones anteriores, se ha reflejado el resultado tanto de la ejecución secuencial como de la ejecución en paralelo para que se pueda comprobar que la ejecución en paralelo se ha realizado de forma correcta y que no ha habido problemas intermedios que desemboquen en resultados diferentes. La ejecución de este código ha sido para un número de edades de 20.

Los datos obtenidos para las siguientes tablas y gráficos se han realizado ejecutando el código con sobre 900000 edades, de forma que el problema a abordar sea algo más complejo debido al tamaño.

En la Tabla 1 se muestran los resultados numéricos del speed-up medio, el tiempo medio de ejecución y la desviación estándar de cada una de las pruebas realizadas con la paralelización del código que pone, para cada posición del vector, el rango de edad al que corresponde.

Nº Prueba	Speed-up	Media (ms)	Desviación std. (ms)
1	1.013	700.696	192.649
		691.517	179.688
2	0.736	684.883	160.053
		930.887	177.727
3	1.067	680.943	72.492
		638.006	65.642

Tabla 1. Bucle posiciones.

Con el speed-up se puede ver cuántas veces mejor es el tiempo de ejecución del código paralelizado con respecto al del código secuencial, en caso de que sea mejor. El speed-up se obtiene con la siguiente fórmula:

$$speedup = \frac{tiempo\ medio\ del\ código\ secuencial}{tiempo\ medio\ del\ código\ paralelizado}$$

Los valores de speed-up superiores a 1 indican que ha habido una mejora en el tiempo de ejecución del código paralelizado con respecto al código secuencial. Los valores que se encuentran por debajo de 1 indican que no ha habido una mejora en el tiempo de ejecución paralelo. Para la anterior tabla se puede ver que el speed-up es mejor para las pruebas 1 y 3 y peor para la prueba 2. En general las mejoras obtenidas con el código paralelizado no son resaltables, ya que las medias de sus tiempos son muy similares y pequeñas.

Nº Prueba	Speed-up	Media (ms)	Desviación std. (ms)
1	1.019	688.358	138.110
		675.575	151.356
2	1.006	667.421	153.762
		663.586	159.057
3	1.066	755.300	94.480
		708.867	91.775
4	1.019	741.892	161.026
		728.898	163.405

Tabla 2. Bucle contador.

En la Tabla 2 se ven los resultados obtenidos de la ejecución del bucle que cuenta cuántas edades hay para cada uno de los rangos de edad. En el caso de estas paralelizaciones, se puede ver que todas las pruebas muestran un speed-up superior a 1, por lo que los tiempos de ejecución mejoran. Pero, al igual que con el bucle anterior, las mejoras obtenidas no son algo reseñable.

Tiempos de ejecución

En la Ilustración 3 se muestran el gráfico generado por los tiempos medios de la ejecución de cada una de las primitivas, comparándolos con los tiempos de esa misma ejecución del código secuencial.

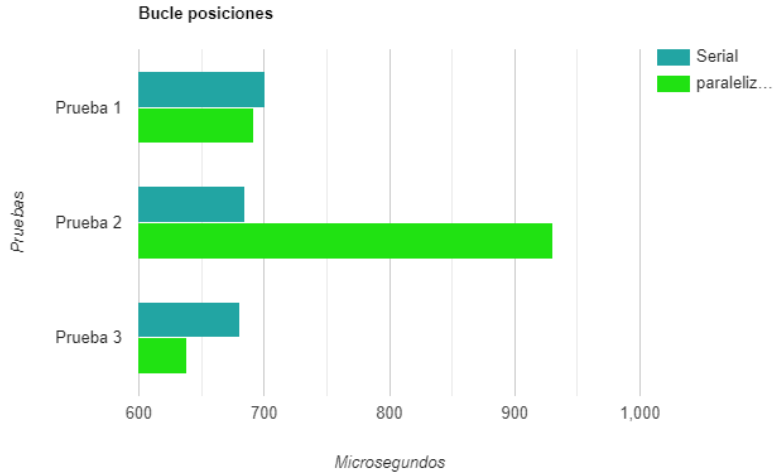

Ilustración 3. Tiempos de ejecución medios del bucle posiciones.

En este gráfico se puede ver de forma algo más clara las diferencias que hay entre los tiempos de cada prueba comparados entre ellos y con la ejecución secuencial. Se observa también que el peor resultado de tiempos se obtiene con la prueba 2 para el primer bucle.

Lo mismo a lo anteriormente mencionado se observa para el gráfico de la Ilustración 4, pero en este caso para el bucle contador. Los tiempos de las pruebas muestran mejor resultado para el código paralelizado, en especial para lo obtenido con la prueba 3.

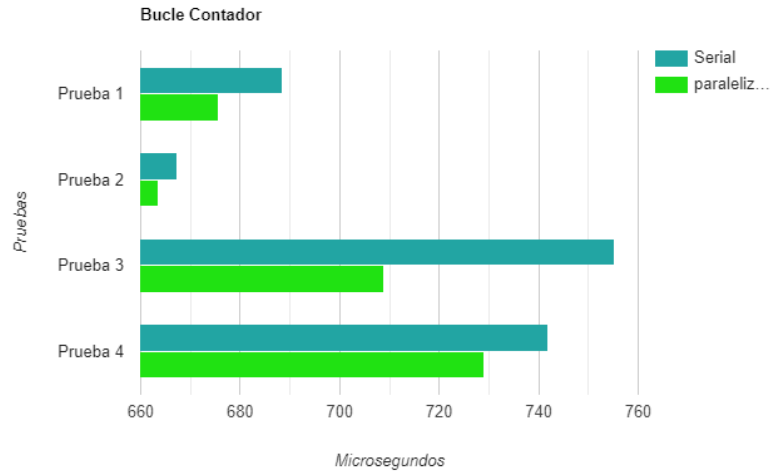

Ilustración 4. Tiempos de ejecución medios del bucle contador.

Como ya se ha mencionado, teniendo en cuenta las unidades de tiempo en la que se mueve la ejecución de código, las mejoras obtenidas con el código paralelizado no pueden destacarse demasiado.

Conclusiones

Tras el análisis y estudio de las diferentes primitivas para paralelizar este programa, se han obtenido las siguientes conclusiones:

- Las primitivas de paralelización consiguen en la mayor parte de los casos reducir el tiempo de ejecución del programa.
- A pesar de reducirse los tiempos de ejecución, son tan pequeños y la diferencia de tiempos entre el código secuencial y el paralelizado es también tan pequeña, que no merece la pena hacer el esfuerzo de análisis que requiere el código para realizar la paralelización.