Computación de Altas Prestaciones

Práctica Final

JAVIER CÓRDOBA ROMERO
JUAN JOSÉ CORROTO MARTÍN

Índice

Introducción

A continuación, se presenta la solución a la práctica final de Computación de Altas Prestaciones. El problema se ha resuelto mediante el uso de OpenMP, junto con el lenguaje C. Para alcanzar la solución se han realizado múltiples pruebas, con el fin de obtener el mejor resultado, en este caso el menor tiempo posible. Para ejecutar el código y extraer los resultados se ha usado un ordenador con las siguientes especificaciones:

- Arquitectura de 64 bits.
- CPU: Intel(R) Core(TM) i7-4720HQ
- Frecuencia de reloj: 2.60 Ghz
- Memoria RAM: 8GB DDR3
- Sistema operativo: Ubuntu 18.04.4 LTS
- Compilador: gcc 8

A continuación, se explican los pasos que se han seguido para alcanzar cada una de las soluciones. En cada sección se explicará el trabajo relativo a cada una de las soluciones que se presentarán al final.

Descripción del problema

El objetivo es lograr un algoritmo capaz de discretizar un vector de fechas en cuatro grupos:

1. De 0 a 14 años.
2. De 15 a 24 años.
3. De 25 a 64 años.
4. De 65 a 95 años.

El vector de entrada estará compuesto por N valores aleatorios en el intervalo [0, 95]. Los valores aleatorios son generados al inicio del programa, siendo la única entrada el tamaño N del vector de datos.

Hemos abordado el problema desde dos perspectivas diferentes:

1. Contabilizar la cantidad de datos en cada uno de los grupos. En este caso, la salida será un vector de 4 posiciones con la cantidad de fechas perteneciente a cada uno de los grupos.
2. Cambiar cada valor del vector por su grupo correspondiente. Esto quiere decir que el resultado será un vector del mismo tamaño N, donde cada posición contendrá el grupo al que pertenece la fecha de la misma posición en el vector original.

Cada una de ellas requiere de un algoritmo distinto, dando pie a diferentes descomposiciones. Pasamos ahora a explicar cada una de las 2 soluciones que hemos planteado.

Conteo de grupos

Descomposición en tareas.

Para esta solución el resultado debe ser un vector de 4 posiciones, cada uno de ellos con el número de fechas perteneciente a cada grupo. Puesto que sabemos el tamaño tanto del vector de entrada como el de salida, se da pie a varias posibles descomposiciones del problema. Hemos realizado las siguientes descomposiciones:

Descomposición de datos de salida.

La descomposición consiste en asignar una tarea a cada uno de los valores de salida, en este caso 4 (Ilustración 1). Cada una de las tareas debe por tanto leer todo el vector inicial, contando únicamente aquellas fechas que pertenezcan a su grupo. Esta descomposición sigue la regla de “Su

propietario lo computa", pues cada tarea es propietaria de una posición del vector resultado. Tampoco existe ninguna comunicación entre procesos.

Esta descomposición presenta una desventaja: cada una de las tareas debe leer el vector inicial entero, por lo que el número de lecturas es 4 veces mayor. A su vez, como cada una de las tareas debe leer el vector completo, se puede pensar que cada una de las tareas tardará casi lo mismo que una ejecución secuencial (aunque cada una de las tareas tenga que hacer menos comparaciones y menos escrituras).

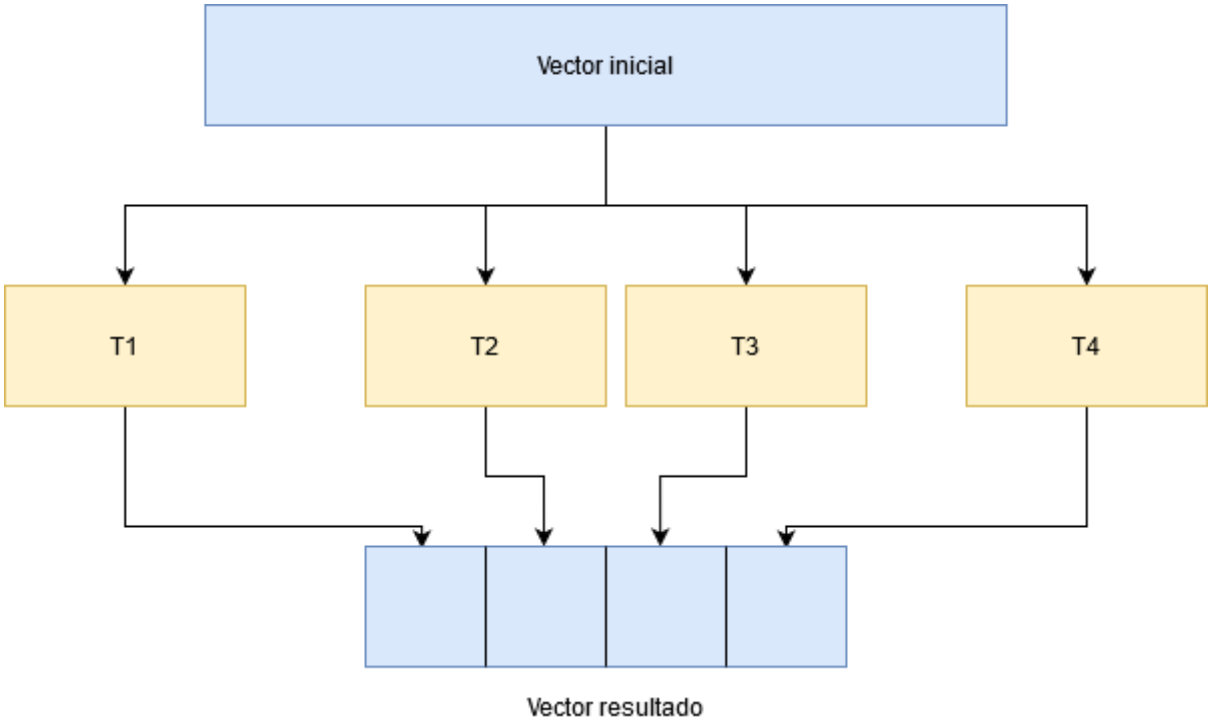

Ilustración 1: Descomposición de datos de salida.

Descomposición de datos de entrada.

Esta descomposición divide el vector de entrada en varios subconjuntos, asignando cada subconjunto a una tarea. Cada tarea es la encargada de contabilizar el resultado de su propio vector, obteniendo un vector de 4 posiciones por cada tarea. En este sentido, al finalizar el conteo debe existir otra tarea adicional para combinar los resultados de todas las tareas (Ilustración 2). Puesto que estamos usando OpenMP y estamos en un entorno de memoria compartida, no es necesaria la comunicación entre tareas, pero al menos una de ellas debe de acceder a los resultados parciales y juntarlos. Esta descomposición también sigue la regla "Su propietario lo computa", sólo que esta vez cada tarea es propietaria de un subconjunto del vector de entrada, y es ella la encargada de calcular su resultado parcial.

La principal desventaja de esta descomposición es el esfuerzo extra requerido para juntar los resultados, necesitando computación extra que no existiría en una ejecución secuencial.

Descomposición de entrada y salida.

También hemos combinado ambas descomposiciones. En este caso, hemos intentado solucionar la principal debilidad de la descomposición de datos de salida: que cada tarea tenga que leer todo el vector de entrada.

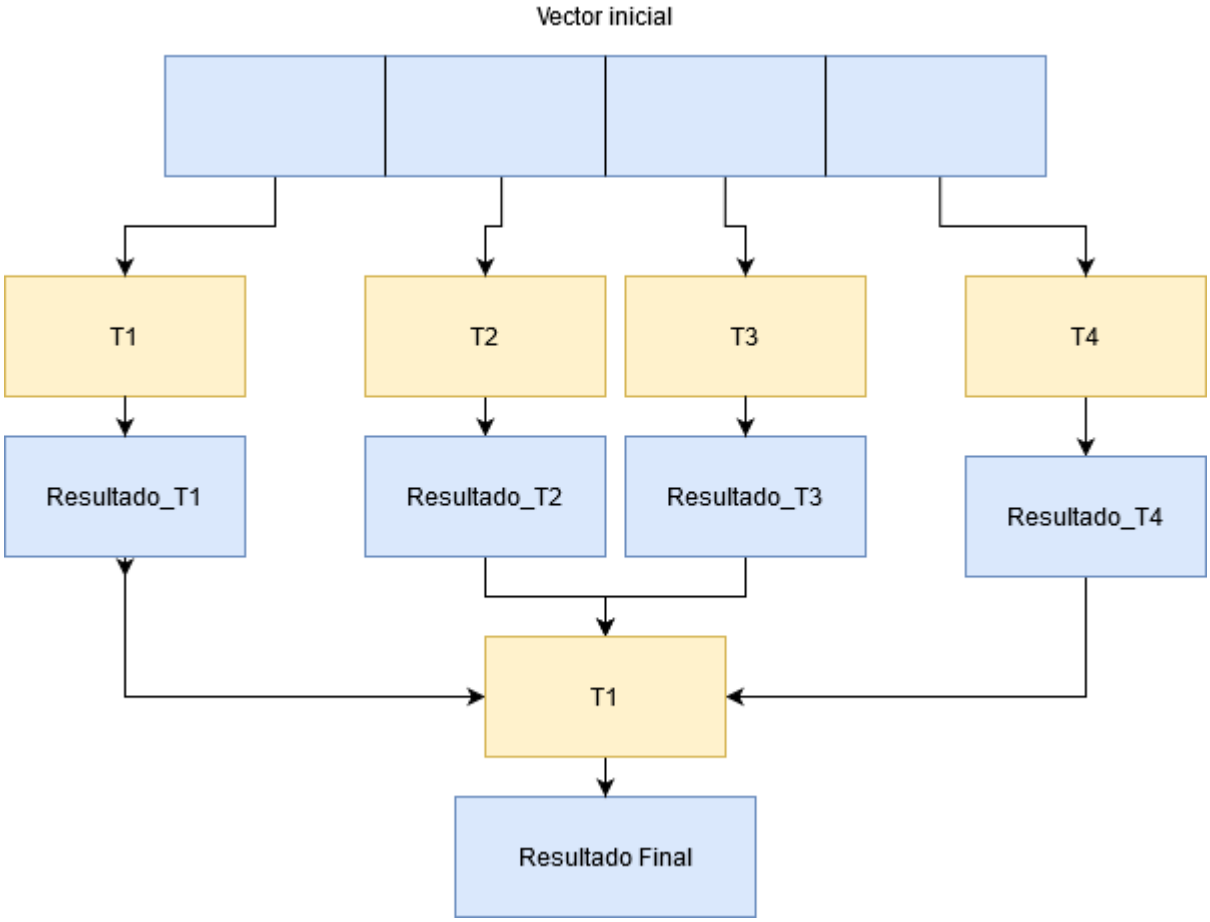

Ilustración 2: Descomposición de datos de entrada.

Con una descomposición de datos de salida y de entrada, tenemos cuatro tareas que toman como entrada todo el vector inicial, cada una de estas cuatro tareas encargándose de una posición del resultado final. Cada una de estas cuatro tareas divide el vector inicial en subconjuntos, y se originan más subtareas para encargarse de calcular los resultados parciales de cada uno de los grupos. El resultado es que cada una de las 4 tareas tendrá varios resultados parciales, esta vez siendo sólo números: el número de fechas del grupo que tienen asignado en cada uno de los subconjuntos de sus subtareas. Estas 4 tareas iniciales deben, por tanto, sumar estos resultados parciales para conseguir el resultado final.

Esta descomposición puede verse como una combinación de las 2 anteriores, donde usamos una descomposición de datos de entrada en cada una de las 4 tareas que se originaban en la descomposición de datos de salida.

Este enfoque logra resolver los 2 problemas que tenían las 2 descomposiciones anteriores: El trabajo necesario para juntar los resultados parciales pasa a ser una suma, que puede ser resulta de forma más rápida que la suma de vectores. Por otro lado, el trabajo de cada una de las tareas no es la lectura del vector completo, por lo que, aunque sigan habiendo lecturas repetidas, el trabajo de cada una de las tareas no es igual de largo que el de una ejecución secuencial.

Mapeo de tareas y algoritmos.

Puesto que hemos usado OpenMP, el mapeo de tareas a procesos específicos es algo que no se puede hacer de forma explícita. Sin embargo, cada tarea en OpenMP es mapeada a un hilo del procesador de la máquina donde se ejecute. Esto quiere decir que cada tarea se realizará en un hilo paralelo, dependiendo de la arquitectura donde se ejecute. En el caso del procesador que hemos utilizado para el cálculo de los tiempos que presentaremos más adelante, podemos tener hasta 8 hebras en paralelo. Esto quiere decir que tenemos hilos de sobra para las 2 primeras descomposiciones, pero no para la descomposición de datos de entrada y de salida, donde algunas tareas tendrán que esperar a que otras terminen.

Por otra parte, un caso específico son las tareas de "agregación de datos parciales", en el caso de las dos últimas descomposiciones. Este trabajo ha sido mapeado al hilo de ejecución de usuario, es decir, es ejecutado fuera de la zona paralela de ejecución.

En cuento al algoritmo usado, se han utilizado bucles for paralelos en los 3 casos. Además, hemos probado a utilizar la primitiva *reduction* que nos ofrece OpenMP para la "agregación de datos parciales", permitiendo reducir aún más el tiempo de ejecución.

En el caso secuencial cada iteración se puede resolver realizando 3 comparaciones y una escritura (la suma). Para las descomposiciones de datos de salida, hemos utilizado un vector auxiliar llamado *bounds*, de tal modo que cada hilo pueda saber cuáles son los límites de su intervalo. En este caso, los hilos usarán su identificador para acceder al vector *bounds*. Para la descomposición de datos de entrada, hemos usado un vector de 16 posiciones para acumular los resultados parciales, y más adelante sumarlos. Esto no es necesario si usamos la primitiva *reduction* (ver código en *parallel_inputdec_reduction.c*).

Resultados

Para obtener resultados hemos ejecutado cada una de las variantes que hemos programado 5 veces, y además probando dos tamaños de entrada: 1 y 100 millones. Para esta solución, tenemos 5 soluciones aportadas: descomposición de datos de salida, descomposición de datos de salida y *reduction*, descomposición de datos de entrada, descomposición de datos de entrada y *reduction*, descomposición de datos de entrada y salida.

La siguiente tabla resume los tiempos de ejecución de cada solución, además del tiempo secuencial, en milisegundos.

Tiempo (ms)	Media 10^6	Desviación sd. 10^6	Media 10^8	Desviación sd. 10^8
Secuencial	5,3024	0,181912891	517,6692	4,244905028
D. salida	5,3174	0,109331148	513,634	5,082759142
D. salida reduction	4,945	0,040230585	482,4006	5,406249097
D. entrada	2,9224	0,145138899	274,6606	5,854847675
D. entrada reduction	2,5122	0,086508381	214,9642	0,87478123
D. entrada salida	6,5762	0,193410703	482,8002	5,01940865

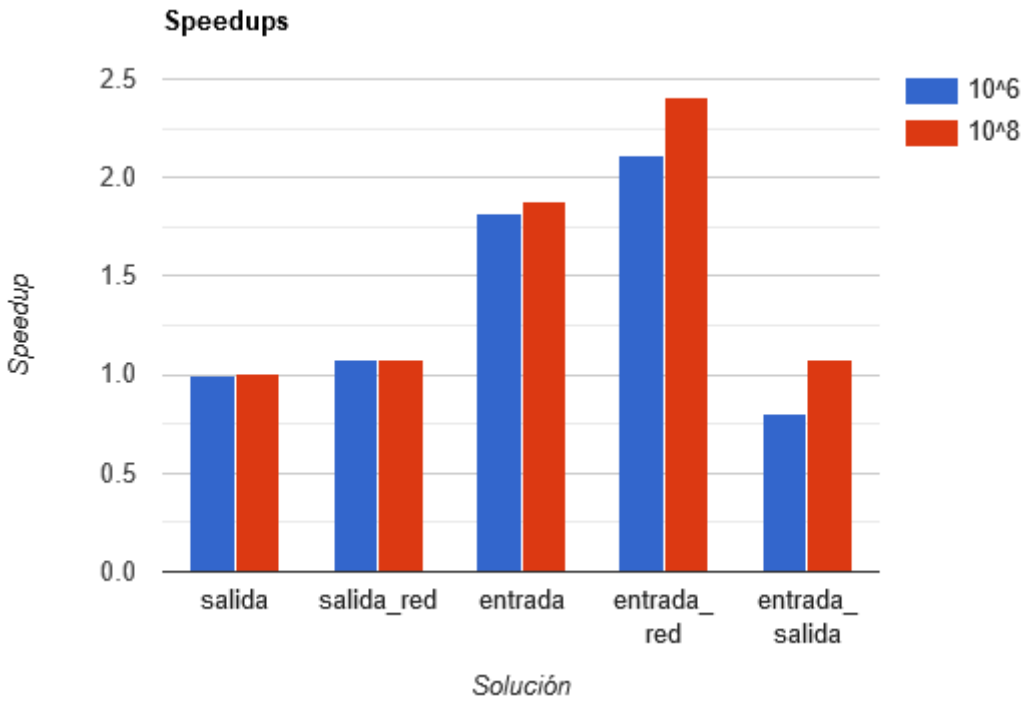

Ilustración 3: Speedups conteo de grupos

En la Ilustración 3 también podemos ver los speedups de cada una de las soluciones. Como se puede ver, la mejor solución es la que usa descomposición de datos de entrada usando *reduction*. También se puede ver cómo, al aumentar el tamaño del problema, todas las soluciones o mejoran o siguen igual. Esto puede ser debido a que, siendo mayor el tamaño del problema, es menor el *overhead* que se añade al utilizar paralelismo con respecto al resto del trabajo. Es interesante sobre todo el caso de la tercera descomposición, que es más lenta que un tiempo secuencial con 10^6 de tamaño de entrada, pero al aumentar el tamaño pasa a ser más rápido. También creemos que la descomposición de datos de entrada y salida funcionaría mejor con más hilos de ejecución, ya que permite la ejecución de más tareas en paralelo, en este caso, más de las que soporta nuestra máquina.

Por otra parte, se puede ver como las soluciones basadas únicamente en descomposición de datos de salida se acercan a speedups de 1, tal y como ya anticipábamos. Puesto que cada una de las tareas debe leer el vector entero, no se produce una gran mejora en el rendimiento (aunque empleando la cláusula *reduction* conseguimos al menos “algo de mejora”).

Transformación del vector

Descomposición en tareas y mapeo

Este problema tiene una especificación ligeramente diferente, la salida tiene el mismo tamaño que el vector de entrada, pero esta diferencia es significativa. También tiene una característica importante: al ser una transformación del vector, cada posición del vector resultado depende únicamente de la misma posición que el vector de entrada. Esto significa que no habrá secciones críticas ni necesidad de comunicación entre tareas.

En nuestro caso, hemos decidido realizar una descomposición de datos de entrada y salida. La descomposición consiste en dividir el vector de entrada en subvectores, cada uno de ellos asignado a una tarea, y cada tarea será la encargada de transformar su subvector para calcular su parte de su

solución. Cada una de las tareas es la propietaria de una parte del vector de entrada y una parte del vector de salida. Usando esta descomposición no es necesario ningún tipo de computación extra ni comunicación.

Suponiendo que descomponemos en 4 tareas, la idea se puede ver en Ilustración 4. Sin embargo, esta descomposición admite un número mayor de tareas. Hemos decidido dejar que OpenMP decida cuantos hilos debe utilizar, mapeando una tarea diferente a cada uno.

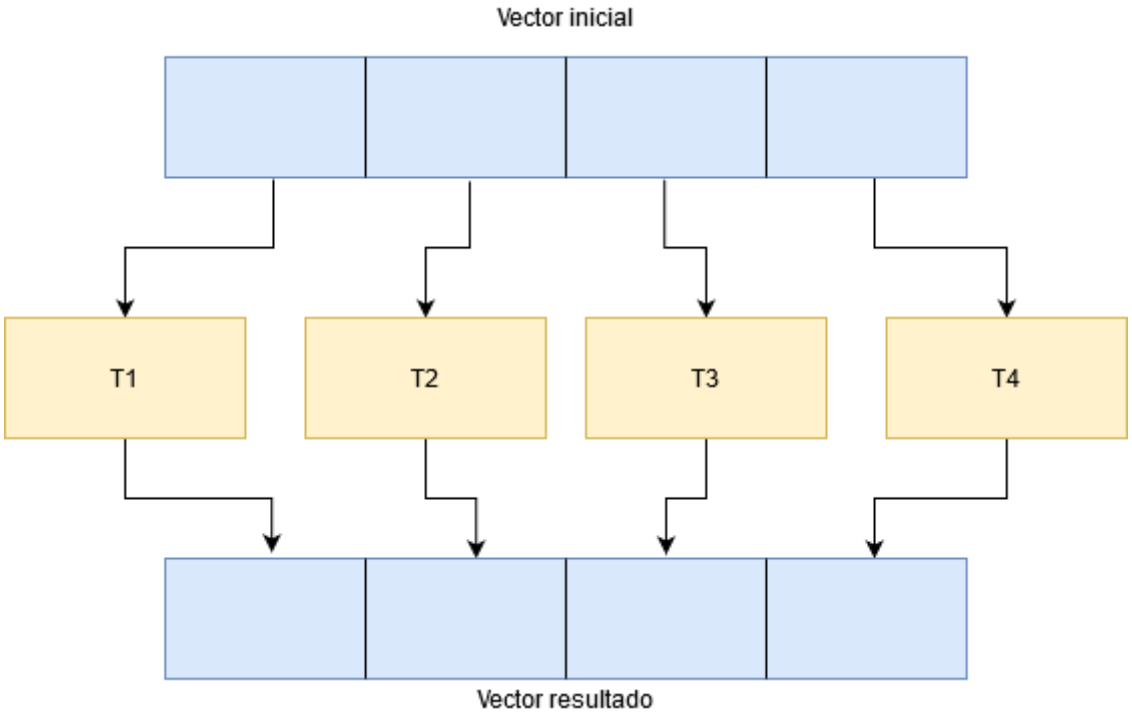

Ilustración 4: Descomposición de entrada y salida

Algoritmo

Esta descomposición se puede resolver mediante un bucle *for* paralelo, usando OpenMP. Hemos decidido calcular el tiempo con las 3 posibilidades que nos ofrece OpenMP para el *scheduling* de bucles *for*, con el objetivo de conseguir el mejor tiempo posible. Hemos intentado hacer el algoritmo secuencial lo más rápido posible para tener una buena base a la hora de paralelizar. Para ello, inicializamos el vector de resultados a 0 directamente usando la primitiva *calloc* en lugar de *malloc*, de tal modo que sólo necesitemos modificar aquellas posiciones de grupos diferentes al primero. Esto nos ahorra todas las escrituras y comparaciones relativas al grupo 0. Este proceso también se ha repetido en la versión paralela.

Resultado

De nuevo, hemos ejecutado cada solución 5 veces para calcular el tiempo media. La siguiente tabla resume los tiempos de cada ejecución para los mismos tamaños de entrada que antes:

Tiempo (ms)	Media 10^6	Desviación sd. 10^6	Media 10^8	Desviación sd. 10^8
Secuencial	6.5476	0.043747	693.1928	20.1616
Paralelo static	2.6422	0.3019	129.167	9.69
Paralelo dynamic	27.9884	0.8175	2722.134	25.1046
Paralelo guided	2.2222	0.5564	121.0152	2.5901

También mostramos en Ilustración 5 los speedups de cada solución, para cada tamaño de vector de entrada. Lo primero que llama la atención es el gran bajón en rendimiento que da usar *Dynamic scheduling*, probablemente debido al gran overhead que introduce. Hay que tener en cuenta que estamos usando un tamaño de chunk de 1 (el predeterminado), por lo que es probable que sea esto lo que añade tanto overhead que el rendimiento disminuye mucho. Por otra parte, *static* y *guided scheduling* dan resultados similares, siendo guided ligeramente superior. *Guided* es similar a Dynamic, pero usando tamaños de chunk variables empezando con tamaños grandes. Quizás sea esto lo que provoca que *guided* tenga un rendimiento mayor.

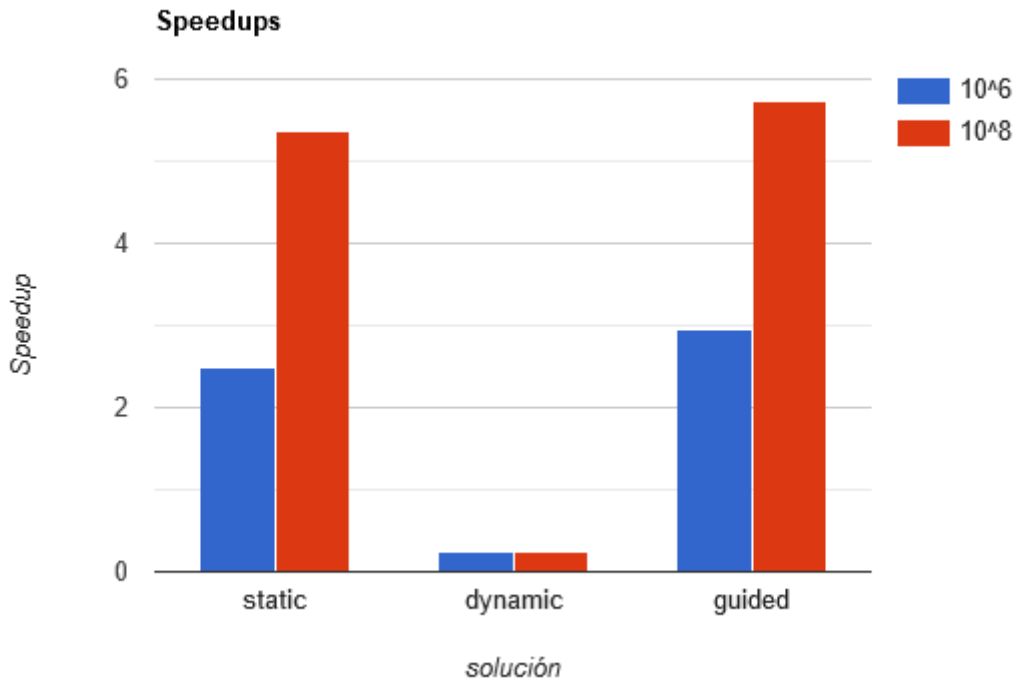

Ilustración 5: Speedups transformación

Repositorio

Todo nuestro código, junto con un Excel con los tiempos específicos de cada ejecución en todas las soluciones, se puede encontrar en https://github.com/Juanjoglvz/HPC-discretizar.