DANIEL MONJAS MIGUÉLEZ
70274432-W
18/04/2022

ACAP: PRÁCTICA 2

Paralelización del código

Ejercicio 1: Para paralelizar el ejercicio 1 lo que he hecho es darle a cada hebra un trozo (de igual tamaño) del vector ingresos y otro trozo (también de igual tamaño) del vector gastos. Cada hebra calcula localmente la suma de su trozo de vector y se lo manda a la hebra 0 que es el encargado de mostrar el resultado.

Para distribuir la carga lo que se ha hecho (en la hebra 0) es dividir el tamaño de los vectores (se ha supuesto que ambos vectores tienen igual tamaño) entre el número de procesos, ajustando esta división para que el tamaño no tenga que ser múltiplo del número de hebras. Tras esto utilizando la función *MPI_Scatterv* le paso a cada hebra la parte que le corresponde de los vectores gastos e ingresos. Finalmente, la hebra 0 hace su parte del cómputo y con un *MPI_Reduce (*opción MPI_SUM) hace la suma de las sumas locales de gastos e ingresos, almacenando el resultado de estas sumas en un vector de dos componentes. Por último la hebra 0 muestra el resultado haciendo la diferencia de los componentes de este vector.

Por su parte cada hebra reserva la cantidad necesaria de memoria para recoger sus trozos de los vectores (se puede hacer, pues está programado de forma que todas las hebras conocen el número total de hebras), los recibe con *MPI_Scatterv,* hace su parte del cómputo y lo manda a la hebra 0 con *MPI_Reduce.*

Ejercicio 2: Para paralelizar este caso (y no repetir el enfoque del ejercicio 1), de nuevo la hebra 0 es la encargada de calcular que trozo del vector le corresponde a cada una de las hebras, pero esta vez en lugar de mandárlo con *MPI_Scatterv* , manda con *MPI_Send* de forma que se hacen tantos envíos como número de hebras haya. Tras esto la hebra 0 calcula el mínimo de su trozo del vector y lo manda con *MPI_Reduce* a la hebra 1, que es la encargada de mostrar cual será el mínimo por pantalla. He utilizado *MPI_Reduce,* pues trae implementada la opción *MPI_MIN,* la cual te calcula el mínimo de entre los número recibidos.

Las hebras worker (rank distinto de 0), comprueban si hay un mensaje con *MPI_Probe*, si lo hay, guardan en una variable el tamaño de este (*MPI_Get_count*) y reserva el espacio justo para recibirlo. Recibe finalmente el mensaje con *MPI_Recv.* Tras esto calcula el mínimo local y lo manda a la hebra 1, de nuevo con *MPI_Reduce.*

Por último se hace un *if,* de forma que si la hebra tiene rank 1, a parte de hacer el *MPI_Reduce*, reserva una variable double que será la que contendrá el mínimo total, que será el que mostrará por pantalla.

Ejercicio 3: En este ejercicio lo que he hecho es que al inicio del programa cada hebra va a su función según cual sea su rank (la hebra 0 a la función 0, la hebra 1 a la función 1, …).

Las hebras con rango mayor que cero lo que hacen es ejecutar un bucle infinito. En este bucle infinito lo que hace primero es un *MPI_Probe,* luego quedan bloqueadas hasta la recepción de un mensaje. Si el mensaje que llega tiene TAG == FIN, lo que hace es salir del bucle infinito, imprimir un mensaje por pantalla de que está abortando su ejecución y finalizar.
Si el TAG!=FIN, lo que hace es la funcionalidad que se pedía para cada una de ellas.

Por su parte la hebra 0 ejecuta la función 0. En este función, al igual que en las otras se ejecuta un bucle infinito, en este bucle si la variable *num_ant < 4* (explico su utilidad más abajo) entonces se pide al usuario que introduzca un número, se comprueba que el número esté en el rango esperado, y sino se volverá a pedir otro número.

Tras esto se ejecuta un *switch*, en función de el número introducido el *swith* nos lleva a una funcionalidad u otra.

- Si el número introducido es 0, se le manda a cada hebra un mensaje con etiqueta FIN, de forma que todas las hebras aborten, la hebra 0 sale del bucle infinito también y termina su ejecución.
- Si el número introducido es 1, se recoge con *fgets*, el texto introducido al usuario, se le manda a la hebra con rango 1, la cual usará la función *toupper*, carácter a carácter y nos devolverá la salida a la hebra 0, la cual la mostrará por pantalla.
- Si el número introducido es 2, se crea un vector de números reales cuyo contenido son {1.1, 2.2, 3.3, …, 10.10}. Este vector se manda con *MPI_Send* a la hebra 2, la cual calcula la suma de estos números y la raíz de la suma (la raíz se hace utilizando la función *sqrt* de la librería math.h). Tras esto se manda el resultado a la hebra 0 usando un vector de dos componentes y esta lo muestra por pantalla.
- Si el número introducido es 3, se le manda a la hebra 3 el propio número, la hebra 3 imprime por pantalla el mensaje *Entrando en funcionlidad 3*, calcula la suma de los códigos de los caracteres del mensaje y se la devuelve a la hebra 0 con *MPI_Send*, tras esto la hebra 0 mostrará el resultado por pantalla.
- Si el número introducido es 4, se pone *num_ant = 4* de esta forma en la siguiente iteración del bucle no se pedirá al usuario un número, sino que se pondrá dicho número a 1, tras la ejecución de la funcionalidad 1, se pondrá el número a 2, y tras la ejecución de la funcionalidad 2 se pondrá el número a 3. Tras ejecutar la funcionalidad 3 siempre se comprueba si *num_ant* es 4, en cuyo caso se establece a 0 para que en la siguiente iteración se pida número al usuario.

Ejercicio 4: Para el ejercicio 4 no me interesaba utilizar un puntero a punteros, pues se colocarían los valores de la matriz contiguamente en memoria. Para solucionar esto he utilizado la abstracción de matriz hecha en el *archivo42_matrices.c* del curso de C. De esta forma declaro un vector de tamaño filas por columnas y otro que sea un puntero a punteros. El puntero a punteros lo inicializo como un vector con tantos elementos como filas tenga la matriz y en cada uno de estos elementos almaceno la dirección de inicio de cada fila en el vector largo. De esta forma se pueden trabajar con dos índices y me aseguro de que todas las filas estén contiguas en memoria, mientras sigo usando memoria dinámica.

Una vez hecho esto simplemente divido el número de celdas de la matriz entre el número de procesos (ajustando para que no sea necesario que el número de celdas sea múltiplo del número de procesos). Aprovechando la abstracción hecha de la matriz mando con *MPI_Scatterv,* a cada hebra un número de celdas de la matriz. Para esto solo tengo en cuenta la dirección donde empieza el vector largo. Una vez mandado a cada hebra el trozo de matriz que le corresponde hacen el cálculo local de su trozo, y por último utilizando la función *MPI_Reduce,* con la opción *MPI_SUM,* se hace la suma de todas las sumas locales en la hebra 0, que muestra el resultado por pantalla.

Ejercicio 5: Para el ejercicio 5 utilizo la misma abstracción de matriz que he utilizado para el ejercicio 4. También he aprovechado que todas las hebras conocen las filas de A, columnas de A y columnas de B, pues se pasan como parámetros al ejecutar el programa y estos valores están contenidos en el vector *argv[]*.

Conocido las filas y columnas de una y otra matriz conozco también las filas y columnas de la matriz resultado. Sabiendo esto, lo que hecho es dividir el número de filas de la matriz resultado entre el número de hebras disponibles, de forma que cada hebra se encarga de calcular un número determinado de filas de la matriz resultado (he considerado que si el número de filas de la matriz resultado es menor que el número de hebras disponibles no tiene sentido paralelizar).

Viendo como hemos hecho la división, cada hebra no requiere la totalidad de las matrices A y B, sino que requiere la matriz B entera, la cual se envía a todas las hebras con la función *MPI_Bcast*, y un número determinado de filas de la matriz A que será igual para todas las hebras (excepto ajustes). Para el envío de la matriz A, como es conocido cuanto hay que mandar a cada hebra, y por como se guarda en memoria he usado *MPI_Scatterv.* Tras esto cada hebra hace localmente el cálculo de las filas que le corresponda de la matriz y la almacenan en una matriz cuyo número de filas es el que calcula cada hebra localmente y cuyo número de columnas es el mismo que el de la matriz B.

Finalmente uno el resultado en una matriz de tamaño filas de A por columnas de B utilizando la función *MPI_Gatherv.* He usado *Gatherv* y no *Gather*, ya que con ajustes puede ser que algunas hebras calculen una fila más que otras, luego no tienen porque hacer exactamente el mismo cálculo, aunque no difiere demasiado. La hebra 0 mostrará por pantallas las matrices A, B y resultado si las filas y las columnas del resultado son menor que 7 .

Ejercicio 6: Para el ejercicio 6, una vez más he calculado el tamaño de vector que le corresponde a cada hebra y se lo he pasado utilizando *MPI_Scatterv*. Tras esto cada hebra hace y almacena localmente en un vector de tamaño 3, el producto escalar de los trozos que dispone, la suma de los cuadrados de los elementos del trozo del vector A y la suma de los cuadrados del trozo del vector B.

Tras esto se hace la suma de los resultados locales y de nuevo se almacena en un vector de longitud 3 en la hebra 0. De esta forma tendremos almacenados en un vector el producto escalar de A y B, la suma de los cuadrados de A y la suma de los cuadrados de B. Con estos datos la hebra 0 calcula la similitud coseno y la almacena en un variable double, haciendo la división del primer miembro del vector entre el producto de las raíces del segundo (modulo de A) y tercer miembro del vector (modulo de B). Es decir, la hebra 0 tiene en una variable el producto escalar de dos vectores partido del producto de sus módulos, luego tiene la similitud del coseno, y la muestra por pantalla.

Ejercicio 7: Para el ejercicio 7 he utilizado el ejercicio 5 y ATCGRID para así poder usar 16 cores. En primer lugar he ido probando tamaños hasta obtener uno que en secuencial tardase entre 20 y 30 segundos.

PRUEBAS TAMAÑO	
Tamaño	Tiempo
100	0,013428
1000	9,886823
1250	18,723566
1400	26,197617

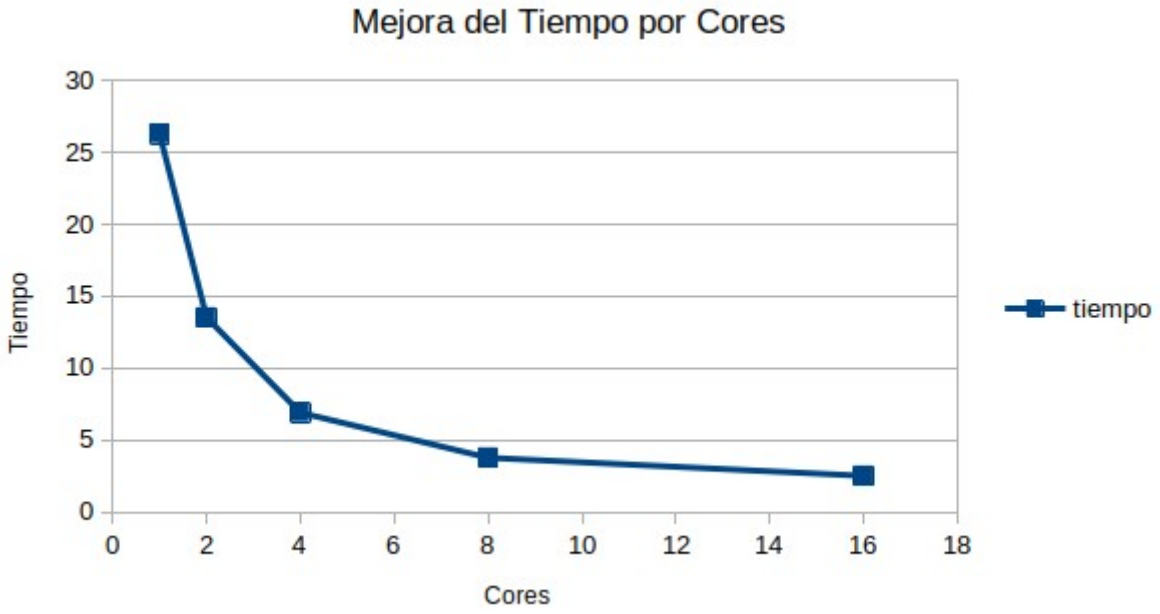
Una vez conocido el tamaño he tomado 5 tiempos usando 1, 2, 4, 8 y 16 cores y he obtenido la siguiente tabla

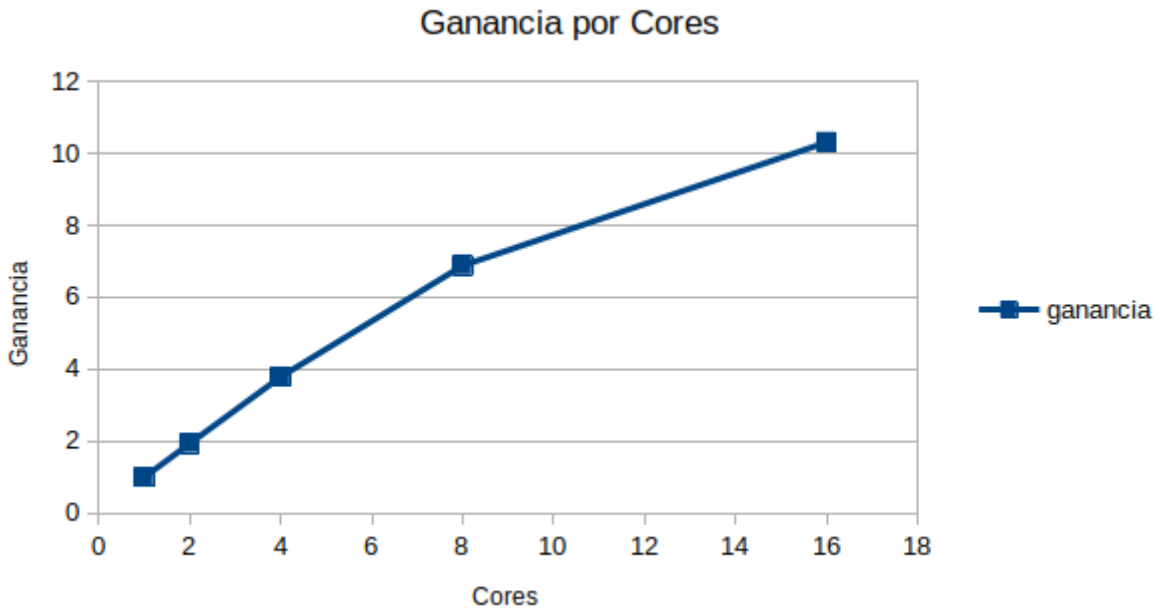
Cores	Tiempo 1	Tiempo 2	Tiempo 3	Tiempo 4	Tiempo 5
1	26,197617	26,418393	26,380955	26,153219	26,171098
2	13,538621	13,520502	13,539924	13,51375	13,51289
4	6,9133	6,910495	6,922372	6,891799	6,91231
8	3,807532	3,77014	3,835011	3,788875	3,738253
16	2,542346	2,546445	2,53009	2,529236	2,508542

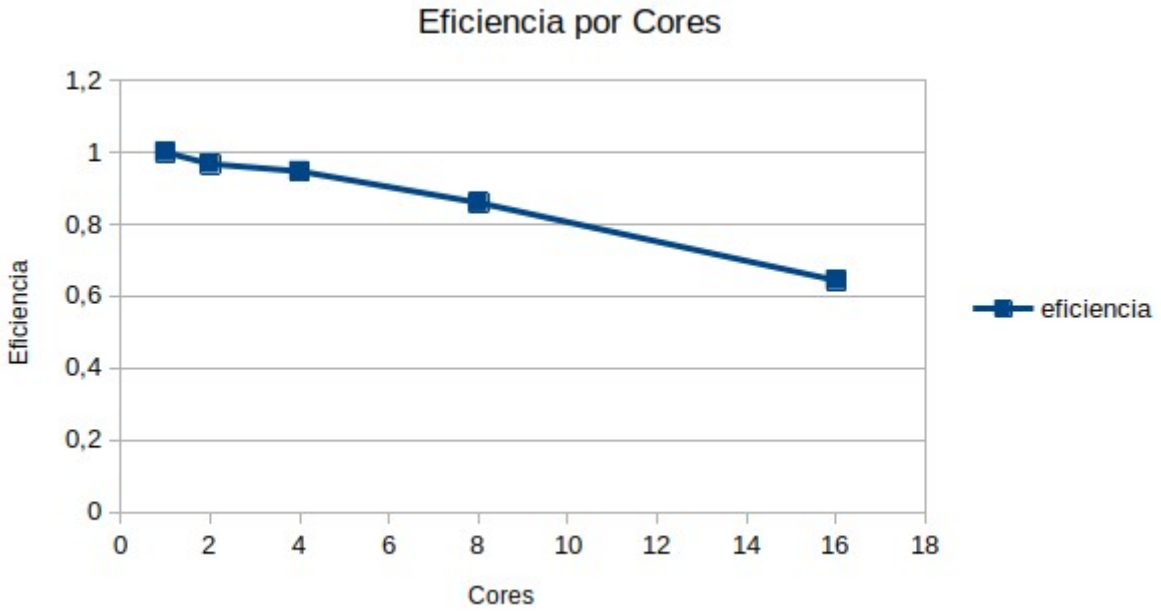
De la cual he sacado los siguiente valores

Media	Desviacion	Ganancia	Eficiencia
26,2642564	0,125324834	1	1
13,5251374	0,0132439574	1,9350284641	0,9675142321
6,9100552	0,0111892658	3,7894517813	0,9473629453
3,7879622	0,0366948063	6,8804719172	0,8600589897
2,5313318	0,0147904601	10,304504973	0,6440315608

Y las siguientes gráficas:







De estas tablas y estas gráficas se puede apreciar que conforme aumentamos el número de cores si bien la ganancia sigue aumentando y los tiempos disminuyendo, la eficiencia va disminuyendo, hasta dejarnos con 16 cores un valor de 0.6 aproximadamente, el cual no es demasiado bueno. Creo que si merece la pena la paralelización, ya que el producto de matrices es una operación bastante costosa y que puede conllevar largos tiempos para matrices cuyo tamaño sea grande, aunque atendiendo a los datos no merece la pena utilizar demasiados cores, pues aunque el tiempo se reduzca la mejora es demasiado pequeña. Basta fijarse que la diferencia entre 8 cores y 16 cores es poco más de un segundo, lo cual no justificaría desperdiciar tiempo de cómputo de unos cores que podrían necesitarse para otro cálculo.

Destacar que para estas pruebas se han utilizado matrices cuadradas, aunque el programa se ha probado y funciona correctamente con matrices de distintos tamaños, no siempre cuadradas. También destacar que por limitaciones de ATCGRID, para utilizar 16 cores se han requerido utilizar la opción -N 2 al lanzar el proceso a la cola, pues un solo nodo no disponía de los cores necesarios (en atcgrid tienen un tope de 6 cores físicos, 12 lógicos por nodo).