Práctica 3

Implementación distribuida de un algoritmo de equilibrado dinámico de la carga usando MPI

Introducción

En esta práctica se aborda la implementación del algoritmo de equilibrado de carga unido al de detección de fin de Dijkstra para resolver el problema del viajante de comercio (TSP) mediante algoritmo Branch&Bound, usando para ello procesos paralelos mediante OpenMPI. Se aportan todos los ficheros necesarios para la compilación y ejecución del algoritmo que se presenta.

El algoritmo se encuentra en el fichero **bbpar.cc**, y para ejecutarlo es necesario pasarle como parámetro el tamaño del problema y la matriz de adyacencias del problema, encontrados en la carpeta **tsp_problems**. Este programa devuelve como resultado el tiempo empleado para la ejecución del algoritmo (usando una barrera para que todos los procesos ejecuten el tiempo a la vez), así como el número de iteraciones (nodos del arbol explorados) que ejecuta cada proceso.

Se aporta un archivo **grafica.gp** que genera las gráficas usando la utilidad **Gnuplot** de Linux y las almacena en la carpeta **resultados**. En esta misma carpeta se encuentra el archivo **resultados.dat**, generado a mano recogiendo los distintos resultados de haber ejecutado el algoritmo.

Esta práctica está programada y ejecutada en Arch Linux x86_64 con procesador Intel Core i5-8250U @ 1.60GHz de 8 núcleos y 2 hilos por núcleo (por tanto, 16 hilos). Se usa OpenMPI en su versión 4.0.5.

Equilibrado de la carga y detección de fin para el algoritmo Branch&Bound

El algoritmo implementa el equilibrado de carga y la detección de fin, pero no implementa la difusión de la cota superior, por tanto los resultados y gráficas que se muestren a continuación han sido obtenidos sin difusión de cota superior.

Se ha usado como esqueleto del algoritmo tanto el pseudocódigo encontrado en las transparencias de la asignatura como el pseudocódigo de la descripción de la práctica. Se hacen envíos y recepciones de mensajes síncronos con *MPI_Send* y *MPI_Recv*, sin embargo, antes se sondea con *MPI_Probe* de forma síncrona para recibir cualquier tipo de mensaje y entonces discriminarlo en una de las cuatro categorías: PETICION, cuando se recibe una petición de trabajo; NODOS, cuando se obtienen nuevos nodos resultado de una petición de trabajo; TOKEN, cuando se empieza la detección de fin pasando el token y su color en anillo por todos los procesos; y FIN, cuando se realiza una segunda comprobación de fin, esta vez cada proceso enviando el valor solución que ha calculado y comparándolo con el recibido, para quedarse con el mejor y volverlo a enviar al siguiente hasta que se haya completado otra vuelta.

Para evitar terminar el algoritmo cuando aún hay procesos en espera de trabajo, algún proceso ha envíado trabajo y sigue sin resolverse, se usan estados para cada proceso (BLANCO/NEGRO, indicando si el proceso ha realizado algún envío de trabajo que no ha quedado pendiente y PASIVO/ACTIVO, indicando que el proceso está explorando el árbol o no le quedan nodos y está esperando) y token (BLANCO/NEGRO, indicando si el token está "limpio" y por tanto se puede finalizar, o si no porque aún queda trabajo en circulación y por tanto un token negro no indica finalizar el algoritmo).

Para enviar el trabajo se ha usado la función de dividir la pila, obteniendo así una nueva pila que es la mitad de la primera, y enviando el atributo "nodos" de la clase pila, con el atributo "tope" siendo el número de nodos que se envían. Es decir, no se envía el objeto de la clase si no el array de enteros que representa todos los nodos, ya que si no no se podría enviar correctamente por MPI. Cuando un proceso recibe ese trabajo, lo almacena en su propia pila, machacándola. Además, para poder saber dinámicamente el número de nodos (tamaño del array de enteros) que se han obtenido, ya que puede variar el tamaño de los nodos que se envían, se ha usado la función *MPI_Get_count* que permite obtener el número de elementos que se han recibido sin haberlos recibido aún, ya que antes se hizo una llamada a *MPI_Probe*.

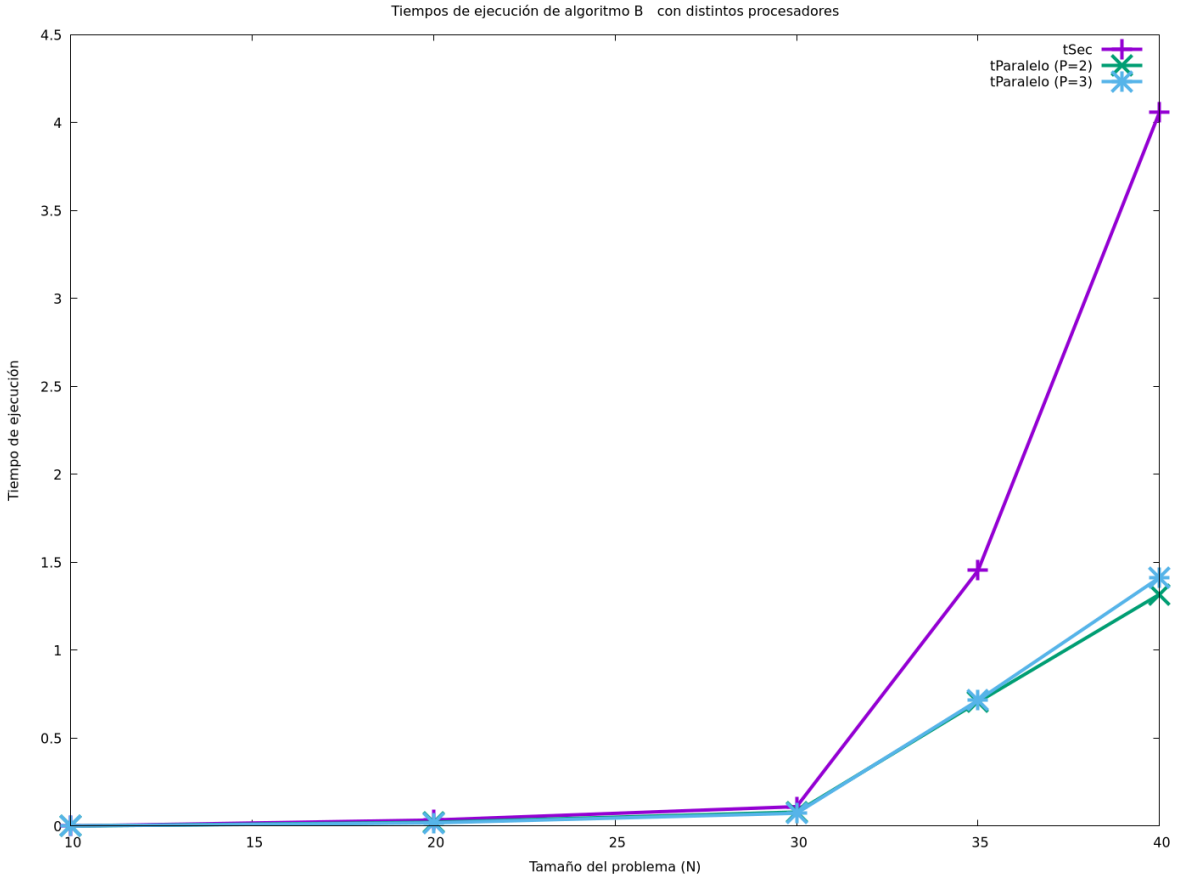
Resultados obtenidos

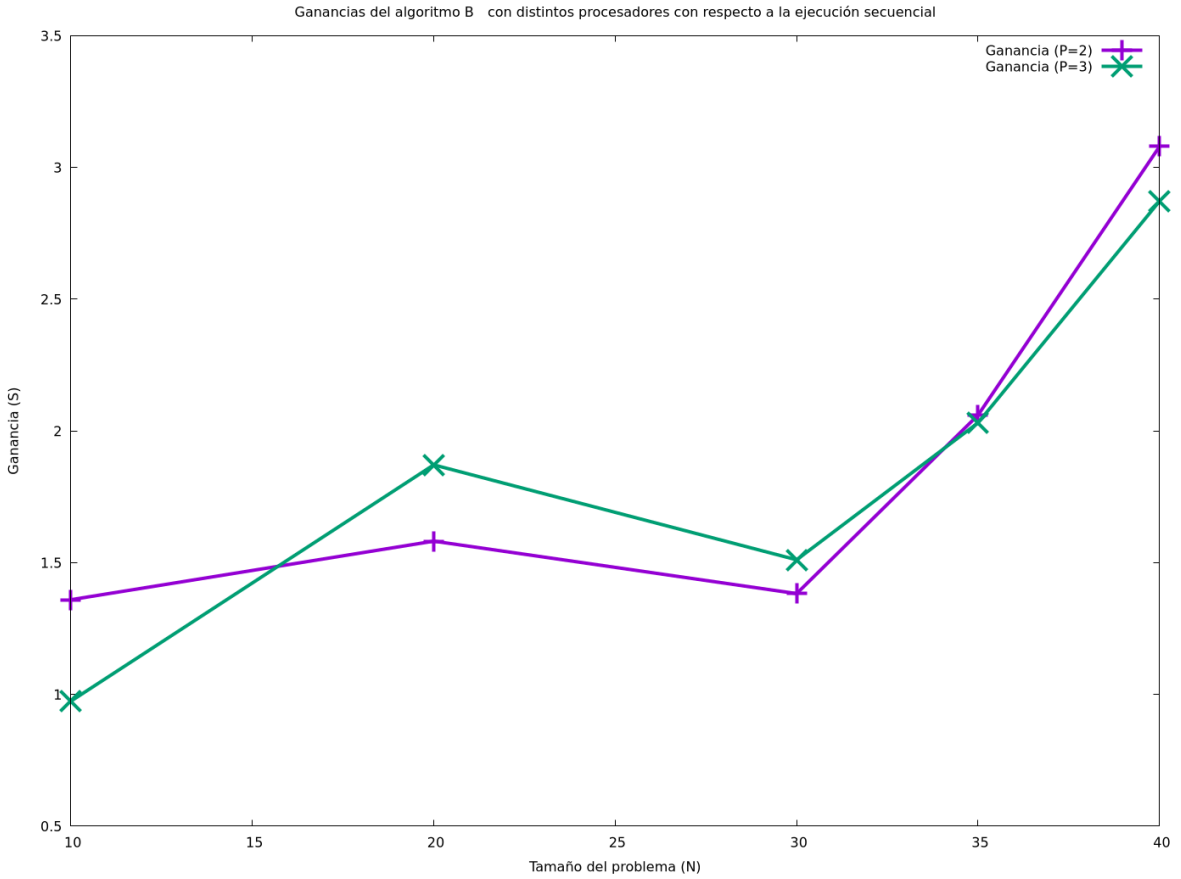
Como se puede ver, se obtienen ganancias de velocidad cercana al número de procesadores o superior para tamaños a partir de 30. Además, el número de nodos explorados por cada proceso es próximo a la media de nodos por proceso (todos los procesos exploran un número parecido de nodos). Esto nos indica que tenemos una buena solución, además de haber comprobando previamente que los resultados obtenidos por el algoritmo son los correctos.

Tamaño del problema	P = 1	P = 2	Ganancia (P=2)	P = 3	Ganancia (P=3)	Nodos explorados (P=1)	Nodos explorados (P=2)	Nodos explorados (P=3)
10	0.000808788	0.000594962	1.359394381	0.000830298	0.974093639	207	P0 = 144 P1 = 96	P0 = 199 P1 = 88 P2 = 114
20	0.0345362	0.0218356	1.581646486	0.0184575	1.871120141	3755	P0 = 2590 P1 = 2385	P0 = 2191 P1 = 2126 P2 = 2033
30	0.110667	0.0800457	1.38254772	0.0732475	1.510863852	6957	P0 = 5635 P1 = 5570	P0 = 5223 P1 = 5045 P2 = 5012
35	1.45378	0.705569	2.060436329	0.715741	2.031153727	71107	P0 = 38059 P1 = 38556	P0 = 39144 P1 = 38503 P2 = 38677
40	4.06029	1.31838	3.079756974	1.41362	2.872264116	158556	P0 = 56600 P1 = 57466	P0 = 61699 P1 = 58812 P2 = 60638

Gráficas

Podemos observar en las gráficas como los algoritmos paralelos nos dan una ganancia bastante alta de entre 2 y 3 con respecto a la solución secuencial, y que entre ellos no hay mucha diferencia en tiempos salvo en tamaños muy grandes en los cuales con dos procesadores es ligeramente mejor. Sin embargo, para tamaños de entre 20 y 30 el algoritmo con 3 procesadores nos da unos mejores resultados.


Tiempos de ejecución de algoritmo B con distintos procesadores
tSec
tParalelo (P=2)
tParalelo (P=3)
Tiempo de ejecución
Tamaño del problema (N)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
10
15
20
25
30
35
40


Ganancias del algoritmo B con distintos procesadores con respecto a la ejecución secuencial
Ganancia (P=2)
Ganancia (P=3)
Ganancia (S)
Tamaño del problema (N)
0.5
1
1.5
2
2.5
3
3.5
10
15
20
25
30
35
40