PROGRAMACIÓN PARALELA Y DISTRIBUIDA

Grado en Ingeniería Informática

COMPUTACIÓN PARALELA PARA LA OPTIMIZACIÓN DE RUTAS MARÍTIMAS CON COLONIA DE HORMIGAS

Jorge Ibáñez Romero

Índice

Contexto

El problema a resolver en este proyecto es el cálculo de ruta de barcos. Este es un problema de logística marítima y optimización que consiste en determinar la ruta más rápida para el transporte marítimo, ya sea de bienes o personas. Este problema es especialmente importante debido a la necesidad de optimizar el tiempo del viaje, los costos de transporte y el impacto medioambiental que este supone.

La complejidad de este problema radica en que, a diferencia del transporte por tierra, el transporte marítimo varía constantemente debido a las condiciones climáticas (lluvia, viento, mareas, ciclones), por lo que las corrientes marítimas no son siempre las mismas en cada zona, lo que provoca que una ruta que tiene una menor longitud que otra sea más lenta debido a las corrientes, además de que las condiciones estén en constante cambio.

Ant Colony Optimization

Para desarrollar la solución a este problema vamos a diseñar un algoritmo de colonia de hormigas (Ant Colony Optimization). A partir de este punto, nos referiremos a este algoritmo como **ACO**. Este algoritmo es una técnica probabilística de optimización basada en el comportamiento de las colonias de hormigas.

Las hormigas esparcen una sustancia química llamada feromona cuando vuelven a su colonia después de encontrar comida, las cuales dejan un rastro tras de sí. Cuantas más hormigas pasen por esa ruta, el rastro será mayor, por lo que la probabilidad de que más hormigas crucen por esa ruta irá en aumento. Hay que tener en cuenta que, con el paso del tiempo, estas feromonas se van evaporando, por lo que las rutas más cortas acaban con una mayor densidad de feromonas que las rutas más largas.

La probabilidad de que una hormiga se mueva de un punto i a un punto j se define de la siguiente manera:

$$p_{xy}^{k} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\Sigma(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}$$

donde τ_{xy} es la cantidad de feromonas depositadas desde x hasta y, $0 \leq \alpha$ es la influencia de τ_{xy}, η_{xy} es la heurística asociada y $\beta \geq 1$ es la influencia de la heurística.

La evaporación y depósito de las feromonas se realiza de la siguiente manera:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \sum_{k} \Delta\tau_{xy}^{k}$$

donde ρ es la tasa de evaporación $(0 < \rho < 1)$ y $\Delta\tau_{xy}^{k}$ es la cantidad de feromonas depositada por la hormiga k dado por:

$$\Delta\tau_{xy}^{k} = Q/L_{k}$$

donde L_{k} es el costo de la ruta de la hormiga k, y Q es una constante.

Análisis del algoritmo ACO

Recursos proporcionados

Para el desarrollo del algoritmo ACO se nos han proporcionado los siguientes archivos y códigos:

- uo.csv y **vo.csv**: Estos ficheros en formato CSV(comma-separated values) contienen los datos de las corrientes en el océano, los cuales se usarán para los cálculos de tiempo entre puntos.

- **data.py**: Este código de Python contiene la definición de la clase Ocean, que se usará para representar el océano, con sus dimensiones y corrientes. De esta clase podríamos considerar como más importantes dos métodos:

- *crop_data()*: Esta función se encarga de recortar las dimensiones del océano dados unos valores iniciales y finales de latitud y longitud. De esta manera, podremos trabajar solo con la zona del océano que necesitemos.
- *get_currents()*: Esta función devuelve las corrientes de un punto del océano dadas una latitud y una longitud.

- **simulation.py**: Este código de Python contiene datos necesarios para el cálculo de tiempo entre los puntos del océano, tales como el radio de la tierra. También contiene la siguiente función:

- *compute_time()*: Esta función es la que se encarga de calcular cuánto tiempo tarda un barco en desplazarse de un punto a otro, dados los puntos de origen y destino, representados mediante sus coordenadas de latitud y longitud, la velocidad del barco y las corrientes marinas obtenidas a partir del objeto de la clase Ocean.

Algoritmo secuencial

Para diseñar e implementar el algoritmo secuencial, era necesario usar un método de comunicación entre el algoritmo y los códigos proporcionados, ya que están escritos en lenguajes de programación que no son compatibles entre sí. De esta manera, se ha decidido hacer uso de pipes nombrados (también conocido como FIFO). Para ello, se ha implementado el código **script.py**, el cual actúa como un “servidor”, solo que en lugar de comunicarse mediante sockets, lo hace mediante pipes. Este código se encarga de precargar los datos del océano, crear los pipes para la comunicación y realizar los cálculos necesarios para la ejecución del ACO.

Este código recibe las coordenadas de latitud y longitud de los puntos de origen y destino, la resolución de los puntos del océano y la velocidad del barco.

Para la implementación secuencial (**ACOsecuencial.c**) se ha creado una matriz dinámica global de feromonas, la cual se asigna con el valor 1.0 en todas sus posiciones y que se actualizará en cada iteración.

Para solicitar los puntos del mapa, las coordenadas de los puntos de origen y destino, y los tiempos entre cada punto se ha diseñado la función *comunicacion()*, la cual se encarga de convertir el mensaje para después enviarlo hacia el pipe de escritura y leer el contenido del pipe de lectura para devolverlo. Antes de la escritura en el pipe se ha añadido una espera de 200 microsegundos para evitar posibles interbloqueos producidos por la comunicación entre los códigos mediante los pipes.

La ejecución del algoritmo se realiza de la siguiente manera:

- El ACO solicita al script el número de puntos del mapa y las coordenadas de los puntos de origen y destino, e inicializa la matriz de feromonas y el vector que contiene la mejor ruta.

- Se ejecuta un bucle que se repite MAX_ITERACIONES veces (el número de iteraciones), en el que se inicializan las filas de una matriz dinámica de rutas que contendrá para cada fila la ruta de cada hormiga, y un vector de longitudes donde cada posición será la longitud total de la ruta de cada hormiga.

- Dentro del primer bucle hay un segundo bucle que se repite NUM_HORMIGAS veces. Dentro de este bucle se inicializan las columnas de la matriz de rutas con el valor -1, y un vector que contiene los puntos visitados en dicha ruta. Se ejecuta un bucle que se repite hasta que se haya visitado el punto de destino o se visiten todos los puntos, donde se seleccionará un punto para calcular el tiempo del anterior hasta este, y se marcará el punto como visitado. A la hora de seleccionar el punto, se calculan los tiempos entre puntos para calcular las probabilidades, las cuales se normalizan, y por último, se selecciona el punto.

- Una vez acabe el bucle en el que se crea la ruta, se calcula la longitud de la ruta de la hormiga, y si es menor a la mejor longitud guardada, se actualiza a esta y se copia la ruta a la mejor ruta guardada.

- Cuando se ejecuten todas las hormigas, se actualiza la matriz de feromonas, aplicando la evaporación y el depósito de feromonas. Al ejecutarse todas las iteraciones del primer bucle, se muestra la mejor ruta conseguida y el tiempo de esta, se cierra la comunicación entre el ACO y el script y se cierra el programa.

Estudio de paralelización del ACO

Para desarrollar la implementación paralela del algoritmo se ha decidido partir de la versión secuencial debido a la complejidad del problema, ya que así podremos asegurarnos de que funcione correctamente. Para ello, se ha decidido comenzar usando OpenMP para aprovechar los recursos en las zonas del código donde no se vayan a producir condiciones de carrera. También se han realizado cambios en la declaración de la matriz de feromonas y las funciones que hacen uso de ella para evitar errores.

Paralelización mediante el uso de OpenMP

En la función *actualizar_feromonas()* podemos acceder a la matriz de feromonas para aplicar la evaporación sin peligro de que se produzcan condiciones de carrera, por lo que haremos uso de la directiva *#pragma omp parallel for collapse(2)*, usando *collapse(2)* para convertir los dos bucles anidados en un único espacio de iteraciones.

Así mismo, también podemos usar la misma clausura para el bucle que se encarga de aplicar el depósito de feromonas. En este caso, debido a las condiciones del segundo bucle, no haremos uso de *collapse(2)*, y, debido a que varios hilos podrían acceder a las mismas posiciones de la matriz de feromonas para actualizarla, se hace uso de la directiva *#pragma omp atomic* para asegurar que estas operaciones se realicen de forma atómica.

Distribución mediante el uso de MPI

Una vez terminada la paralelización mediante OpenMP, se distribuirá el algoritmo mediante el uso de MPI. La idea pensada para distribuir en este algoritmo es repartir la cantidad de hormigas que se van a ejecutar. Para ello, se ha decidido por realizar una división entre 2 nodos, debido a los recursos disponibles. Cada nodo recibirá una cantidad de hormigas sobre el total, y se ejecutará de igual manera que en secuencial.

Cada 2 iteraciones, cada proceso enviará su matriz de feromonas al otro proceso. Para facilitar la comunicación entre ellos, la matriz se convertirá en un vector, y este se enviará usando *MPI_Send()* y se recibirá con *MPI_Recv()*. Una vez los procesos reciban estos vectores, se sumará el valor de cada posición correspondiente a la de

la matriz con el valor de la posición de la matriz que posee el proceso, y se actualizará con la media de ambos, permitiendo aumentar la exploración de la solución en ciertos casos y equilibrar en situaciones donde la carga sea distinta en cada proceso.

Una vez acaben todas las iteraciones, el primer proceso recibirá la mejor longitud y la mejor ruta del segundo proceso, y se elegirá la mejor solución entre los dos procesos.

Al añadir los vectores de feromonas y el cálculo de la media, podemos volver a usar OpenMP, ya que no existen condiciones de carrera en esta sección del código, por lo que al convertir la matriz de feromonas en un vector se usará la directiva *#pragma omp parallel for collapse(2)*, y en el cálculo de la media se usará *#pragma omp parallel for***.**

Implementación de la paralelización del ACO

Debido a que el programa se ejecuta usando dos procesos, y las pruebas se van a realizar en una sola máquina, se ha creado una copia de **script.py** llamada **script2.py**, que hará las mismas funciones, pero realizará la comunicación usando pipes distintos al otro código, debido a que, si usamos los mismos pipes, se podrán producir entrelazados entre los procesos que ejecutan el ACO.

En el caso de ejecutarlo en varias máquinas, podríamos suprimir este archivo y sustituir su uso por el de **script.py**, ya que la comunicación entre pipes nombrados solo se puede realizar entre procesos de la misma máquina, por lo que solo tendríamos que ejecutar este archivo en cada máquina para poder realizar la comunicación correctamente.

Para realizar el balanceo de carga, se ha creado el fichero **nodes.txt**. Este fichero debe contener dos números enteros separados por un espacio, que representarán el porcentaje de rendimiento del nodo sobre el rendimiento total, por lo que es responsabilidad del usuario conocer el rendimiento de sus máquinas e introducirlo correctamente. El ACO leerá este fichero para asignar el porcentaje correspondiente de hormigas, y si los valores están mal escritos o la suma de ambos no sea la correcta, se mostrará un mensaje de error y se cerrará la ejecución.

Las instrucciones de compilación y ejecución de los códigos son las siguientes:

- Compilación ACO: mpicc -fopenmp -o ACO ACO.c -lm
- Ejecución ACO: mpirun -np 2 ./ACO
- Ejecución script Python: python3 script.py latitud_inicial longitud_inicial latitud_final longitud_final resolucion velocidad (en ACO.c se deben modificar los valores de las variables lat_i, lon_i, lat_f y lon_f para que coincidan con las del script).

Resultados, experimentación, discusión y conclusión

Para realizar las pruebas de las distintas implementaciones del algoritmo se ha usado el siguiente equipo:

- Nombre del equipo: HP OMEN 15-dc0035ns

- Procesador: Intel® Core™ i7-8750H (frecuencia de base de 2,2 GHz, hasta 4,1 GHz, 9 MB de caché, 6 núcleos con Hyper-Threading)
- Memoria RAM: SDRAM DDR4-2666 de 16 GB (1 x 16 GB)
- Sistema operativo: Linux Mint 21.2 Cinnamon
- Versión Python: Python 3.10.12
- Versión GCC: GCC 11.4.0
- Versión OpenMP: OpenMP 4.5
- Versión OpenMPI: OpenMPI 4.1.2

También es necesario instalar las librerías de Python del fichero **requirements.txt** para ejecutar el programa.

Para la implementación paralela sobre 2 nodos, se asignan automáticamente al equipo 2 threads para cada proceso MPI.

Las pruebas de ejecución de las distintas implementaciones del algoritmo se han realizado para un tamaño de 10 iteraciones y 100 hormigas, variando el número de puntos del mapa. Se han realizado 3 ejecuciones para cada tamaño y se ha realizado la media entre los 3, y los resultados obtenidos del tiempo de ejecución son los siguientes:

Nº de puntos	Tiempo secuencial	Tiempo 1 nodo MPI 1 thread	Tiempo 1 nodo MPI 2 threads	Tiempo 1 nodo MPI 4 threads	Tiempo 2 nodos MPI (2 threads cada nodo)
12	42,81 segundos	43,90 segundos	40,83 segundos	38,70 segundos	25,09 segundos
56	11 minutos y 35,32 segundos	13 minutos y 42,61 segundos	12 minutos y 9,37 segundos	11 minutos y 32,48 segundos	5 minutos y 56,12 segundos
88	30 minutos y 41,98 segundos	31 minutos y 47,12 segundos	29 minutos y 20,84 segundos	26 minutos y 24,71 segundos	14 minutos y 1,93 segundos

Para evaluar el rendimiento de la implementación paralela y distribuida, calcularemos el Speedup algorítmico sobre la implementación secuencial desarrollada, así como la eficiencia computacional. También se evaluará la paralelizabilidad del algoritmo paralelo, comparando las versiones de un nodo MPI.

Los resultados de Speedup algorítmico y eficiencia computacional son los siguientes:

	Versión 1 nodo MPI 1 thread	Versión 1 nodo MPI 2 threads	Versión 1 nodo MPI 4 threads	Versión 2 nodos MPI (2 threads cada nodo)
12 puntos	S = 42,81 / 43,90 = 0,98 E = 0,98 / 1 = 0,98	S = 42,81 / 40,83 = 1,05 E = 1,05 / 2 = 0,53	S = 42,81 / 38,70 = 1,11 E = 1,11 / 4 = 0,28	S = 42,81 / 25,09 = 1,71 E = 1,71 / 4 = 0,43
56 puntos	S = 695,52 / 822,61 = 0,85 E = 0,85 / 1 = 0,85	S = 695,52 / 729,37 = 0,95 E = 0,95 / 2 = 0,48	S = 695,52 / 692,48 = 1 E= 1 / 4 = 0,25	S = 695, 52 / 356,12 = 1,95 E = 1,95 / 4 = 0,49
88 puntos	S = 1841,98 / 1907,12 = 0,97 E = 0,97 / 1 = 0,97	S = 1841,98 / 1760,84 = 1,05 E = 1,05 /2 = 0,53	S = 1841, 98 / 1584,71 = 1,16 E = 1,16 / 4 = 0,29	S = 1841,98 / 841,93 = 2,19 E = 2,19 / 4 = 0,55

Como se puede apreciar en la tabla anterior, los valores obtenidos corresponden a un Speedup sublineal, ya que existen ciertas funciones que no resultan paralelizables, debido a la reserva de memoria y a las comunicaciones mediante pipes nombrados, además de las comunicaciones existentes en la versión con 2 nodos MPI.

Por otro lado, los resultados obtenidos de la paralelizabilidad del algoritmo son los siguientes:

	Versión 1 nodo MPI 2 threads	Versión 1 nodo MPI 4 threads
12 puntos	P = 43,90 / 40,83 = 1,08	P = 43,90 / 38,70 = 1,13
56 puntos	P = 822,61 / 729,37 = 1,13	P = 822,61 / 692,48 = 1,19
88 puntos	P = 1907,12 / 1760,84 = 1,08	P = 1907,12 / 1584,71 = 1,2

Como podemos ver, existe una mejora al aumentar la cantidad de hilos, pero el valor de paralelizabilidad aumenta levemente sin mejorar de forma notable, debido a las zonas no paralelizables, como se comentó anteriormente.

Los resultados obtenidos cumplen con lo esperado, teniendo en cuenta los recursos disponibles. La versión paralela y distribuida del ACO mejora notablemente debido al reparto de carga entre los nodos, además de que no existen muchas comunicaciones entre los procesos MPI, lo cual podría empeorar el rendimiento total.

Conclusión

La implementación desarrollada consigue un buen desempeño, aunque hay factores que afectan a la decisión de escoger este modelo sobre otros.

La comunicación mediante pipes nombrados, aunque tenga un buen rendimiento, no es la mejor opción, ya que implica un mayor gasto de recursos de cómputo, además de la posibilidad de interbloqueos entre los procesos.

El modelo implementado está pensado para 2 nodos, por lo que habría que adaptarlo a más nodos si fuera necesario, ya que no maneja una cantidad mayor. Esto implicaría cambios en la implementación, ya que aumentaría considerablemente la complejidad y las comunicaciones si se quisiera mantener la misma estructura.

Otra posible mejora sería cambiar el método para el balanceo de carga entre los nodos, añadiendo una prueba rápida de benchmark para estudiar el rendimiento disponible del sistema y no responsabilizar al usuario de repartir la carga.

Por lo tanto, aunque esta implementación sea funcional y ofrezca resultados aceptables, existen otras opciones que pueden ofrecer un mejor rendimiento, como por ejemplo, traducir los códigos de Python proporcionados a C, ya que C es un lenguaje bastante más rápido que Python, y el ACO podría acceder directamente a los datos del océano y realizar los cálculos de tiempos. De esta manera, no sería necesario usar pipes nombrados, y se reducirían considerablemente las comunicaciones entre procesos.

Aún así, el enfoque desarrollado ha sido útil para el aprendizaje sobre comunicación entre procesos, así como la paralelización y distribución de código y el estudio del rendimiento.