PROGRAMACIÓN PARALELA Y DISTRIBUIDA

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

Autor:
Jorge Ibáñez Romero

Índice

Página

1. 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.

1.1. 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:

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

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:

$$\Delta\tau_{xy}^{k} = \frac{Q}{L_k}$$

donde L_k es el costo de la ruta de la hormiga k, y Q es una constante.

2. Análisis del algoritmo ACO

2.1. 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.

2.2. 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 (**ACO.c**) se ha creado una matriz dinámica 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.

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.

3. 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, y posteriormente se aplicará distribución mediante el uso de MPI, además de realizar cambios para modificar la estrategia y volver a hacer uso de OpenMP donde sea conveniente.

3.1. Paralelización mediante el uso de OpenMP

El bucle que ejecuta las hormigas se paralelizará mediante el uso de la directiva *#pragma omp parallel for shared(rutas, feromonas, mejor_ruta, mejor_longitud)*. De esta manera, cada thread se encargará de un grupo de hormigas. Debido a la necesidad de proteger el acceso a la función *tiempo()* por el uso de pipes nombrados, se ejecutarán las llamadas a esta función en exclusión mutua, haciendo uso de la directiva *#pragma omp critical*, al igual que se realizará de la misma manera a la hora de actualizar la mejor longitud conseguida, para evitar una posible pérdida de la mejor ruta calculada.

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) shared(feromonas)*, 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 threads 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.

3.2. 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 un reparto entre los nodos disponibles hasta 16.

Cada 2 iteraciones, todos los procesos convertirán sus matrices de feromonas en un array de feromonas llamado *vector_feromonas*, para posteriormente acumular los valores de cada proceso en un nuevo array llamado *vector_total* mediante el uso de la instrucción *MPI_Reduce()* hacia el proceso raíz. Una vez recibido, este array se

repartirá hacia todos los procesos mediante la instrucción *MPI_Broadcast()*, y una vez lo hayan recibido, actualizarán sus matrices de feromonas con la media de todos los procesos, asignando cada posición correspondiente del array entre el número de nodos del programa, 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 de todos los procesos, haciendo uso de las instrucciones *MPI_Send()* y *MPI_Recv()*, y se elegirá la mejor solución entre los 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) shared(feromonas, vector_feromonas)*, y en el cálculo de la media se usará *#pragma omp parallel for shared(feromonas, vector_total)*.

4. Implementación de la paralelización del ACO

Debido a que el programa se puede ejecutar con varios procesos, y las pruebas se van a realizar en una sola máquina, se han creado copias de **script.py** llamadas **scriptN.py**, siendo N el número del proceso al que le corresponda más 1 (el proceso raíz es el proceso 0), que hará las mismas funciones, pero realizará la comunicación usando pipes distintos, 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 seis 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 secuencial: gcc ACO.c -o ACO -lm
- Ejecución ACO secuencial: ./ACO
- Compilación ACO paralelo: mpicc -fopenmp -o ACOparalelo ACOparalelo.c -lm
- Ejecución ACO: mpirun --oversubscribe -np N ./ACOparalelo (siendo N el número de nodos)
- Ejecución script Python: python3 script.py latitud_inicial longitud_inicial latitud_final longitud_final resolucion velocidad (tanto en ACO.c como en ACOparalelo.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).

5. Resultados, experimentación, discusión y conclusión

Para realizar las pruebas de las distintas implementaciones del algoritmo se ha usado un equipo con las siguientes especificaciones:

- Procesador: Intel® Core™ i5-12600 (frecuencia de base de 3,3 GHz, hasta 4,8 GHz, 18 MB de caché, 6 núcleos con *Hyper-Threading*)
- Memoria RAM: 16 GB (2 x 8 GB) DDR4 3200 MHz
- Sistema operativo: Windows 10 con WSL para usar Ubuntu 24.04.1 LTS
- 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.

Las pruebas de ejecución de las distintas implementaciones del algoritmo se han realizado para un tamaño de 10 iteraciones y 100 hormigas sobre 100 puntos del mapa. Se han realizado 10 ejecuciones para cada tamaño y se ha realizado la media entre los 10. Se han realizado pruebas del algoritmo en un único nodo MPI, variando el número de threads, en 2 nodos MPI, variando el número de threads, y en varios nodos MPI con un único thread.

Los resultados de tiempo de ejecución son los siguientes:

Configuración	**Tiempo de ejecución**
Secuencial	1569.3 segundos
1 nodo MPI 1 thread	1595.8 segundos
1 nodo MPI 2 threads	1512.4 segundos
1 nodo MPI 3 threads	1626.6 segundos
1 nodo MPI 4 threads	1629.2 segundos
2 nodos MPI 1 thread	823.1 segundos
2 nodos MPI 2 threads	782.2 segundos
2 nodos MPI 3 threads	810.0 segundos
2 nodos MPI 4 threads	812.2 segundos
3 nodos MPI 1 thread	570.6 segundos
4 nodos MPI 1 thread	419.8 segundos
5 nodos MPI 1 thread	350.6 segundos
6 nodos MPI 1 thread	282.2 segundos
7 nodos MPI 1 thread	241.6 segundos
8 nodos MPI 1 thread	216.2 segundos
9 nodos MPI 1 thread	201.6 segundos
10 nodos MPI 1 thread	185.6 segundos
11 nodos MPI 1 thread	168.1 segundos
12 nodos MPI 1 thread	158.2 segundos
13 nodos MPI 1 thread	151.5 segundos
14 nodos MPI 1 thread	149.7 segundos
15 nodos MPI 1 thread	144.8 segundos
16 nodos MPI 1 thread	142.6 segundos

Cuadro 1: Tiempos de ejecución calculados para las diferentes configuraciones

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 las versiones de un único nodo MPI son los siguientes:

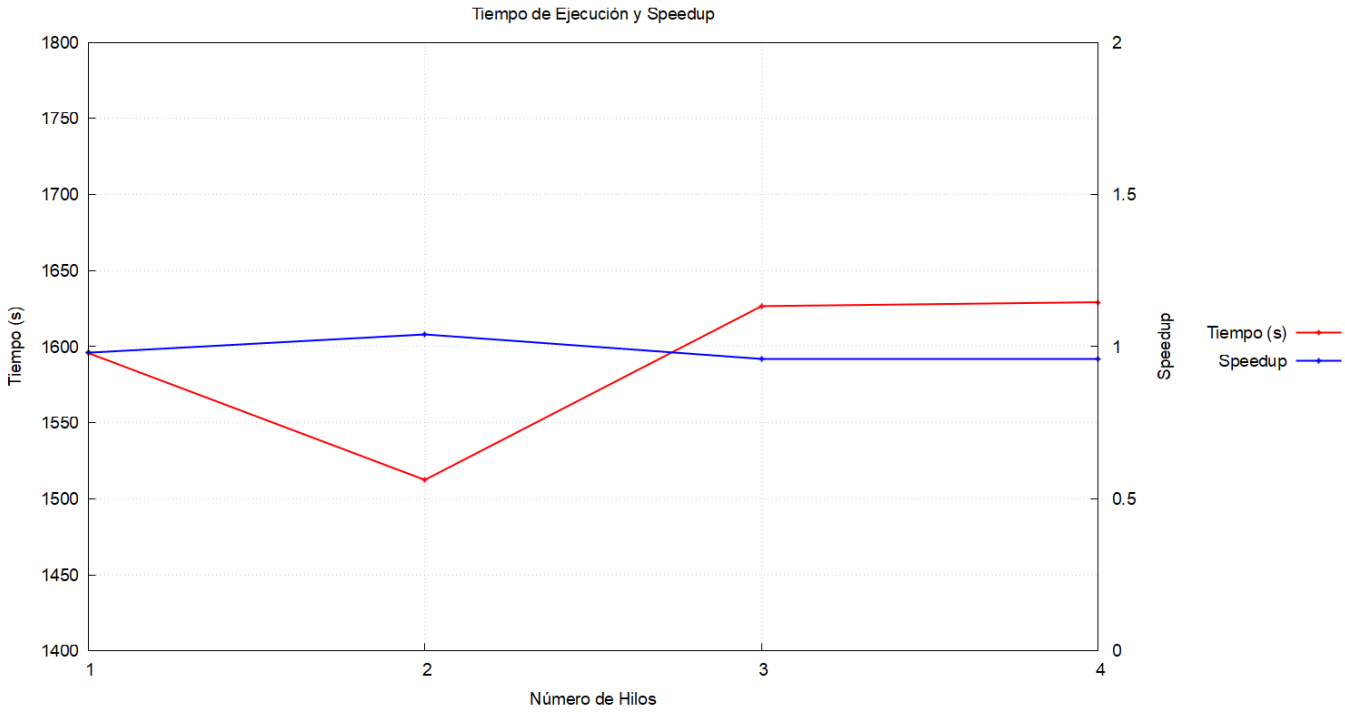

Figura 1: Resultados de Speedup de las versiones de 1 nodo

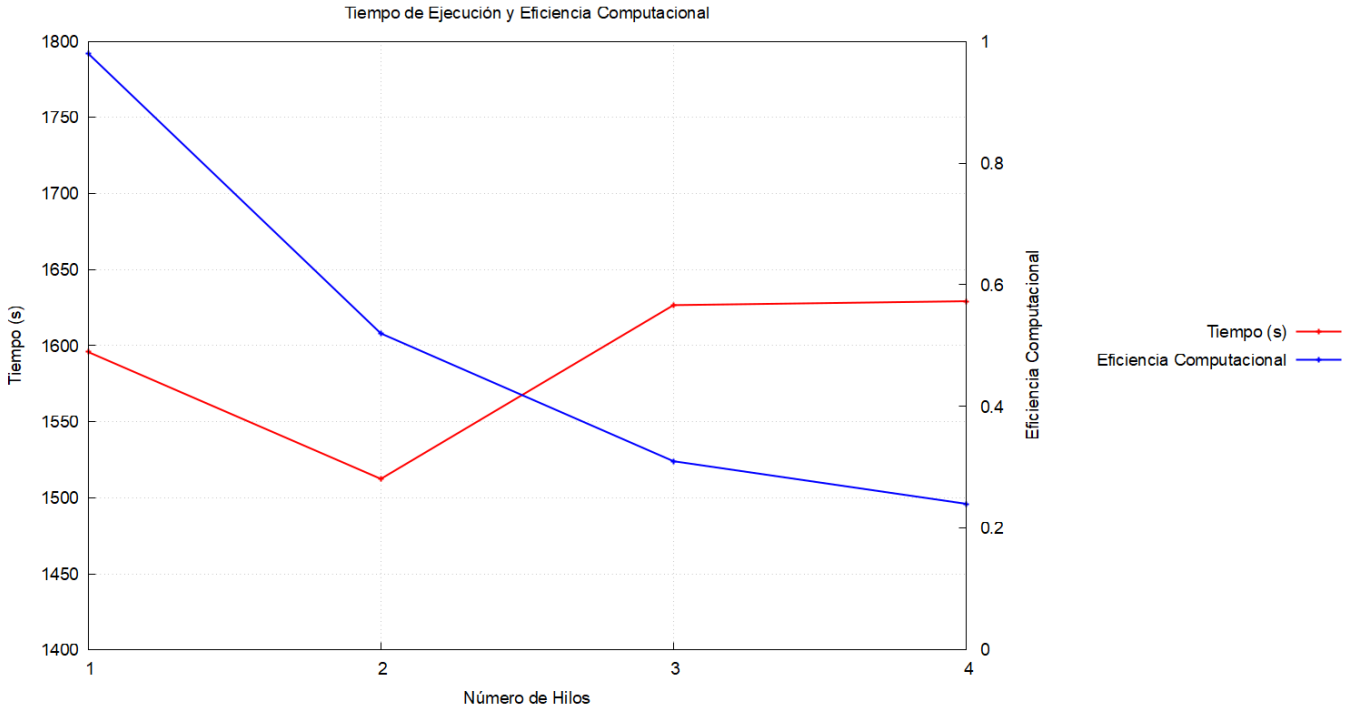

Figura 2: Resultados de Eficiencia Computacional de las versiones de 1 nodo

Como podemos observar en las gráficas mostradas, los resultados muestran un Speedup sublineal. Hay que tener en cuenta que existen zonas del código que no se pueden paralelizar, y la comunicación mediante pipes nombrados produce un fuerte cuello de botella. Debido a estos factores, la mejora de rendimiento no es muy destacable al aumentar el número de threads, aunque conseguimos una mejora al usar 2 threads, pese a que al volver a aumentar la cantidad, esta mejora desaparezca.

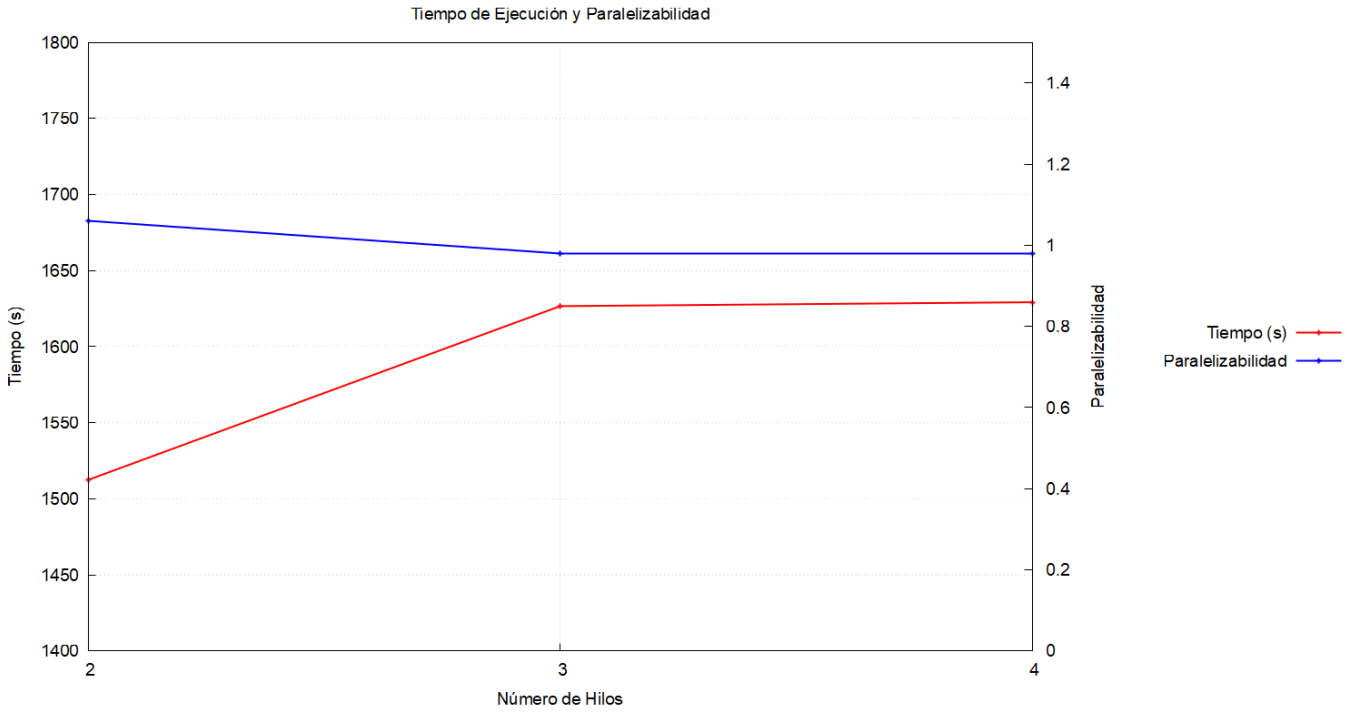

Figura 3: Resultados de Paralelizabilidad

Por otro lado, podemos ver que la eficiencia decrece debido al aumento de threads usados, y que la paralelizabilidad se mantienen en niveles muy similares debido a la poca variación de rendimiento al variar el número de threads.

Los resultados de las versiones de 2 nodos MPI son los siguientes:

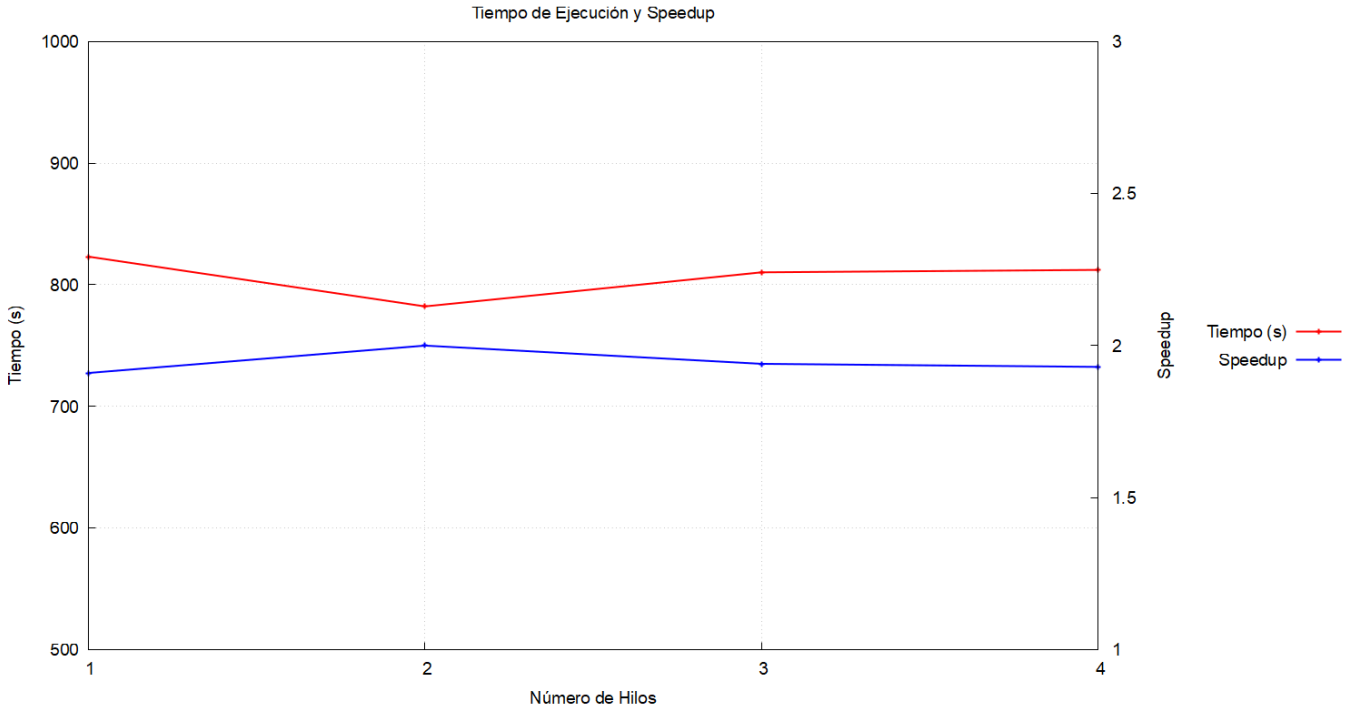

Figura 4: Resultados de Speedup de las versiones de 2 nodos

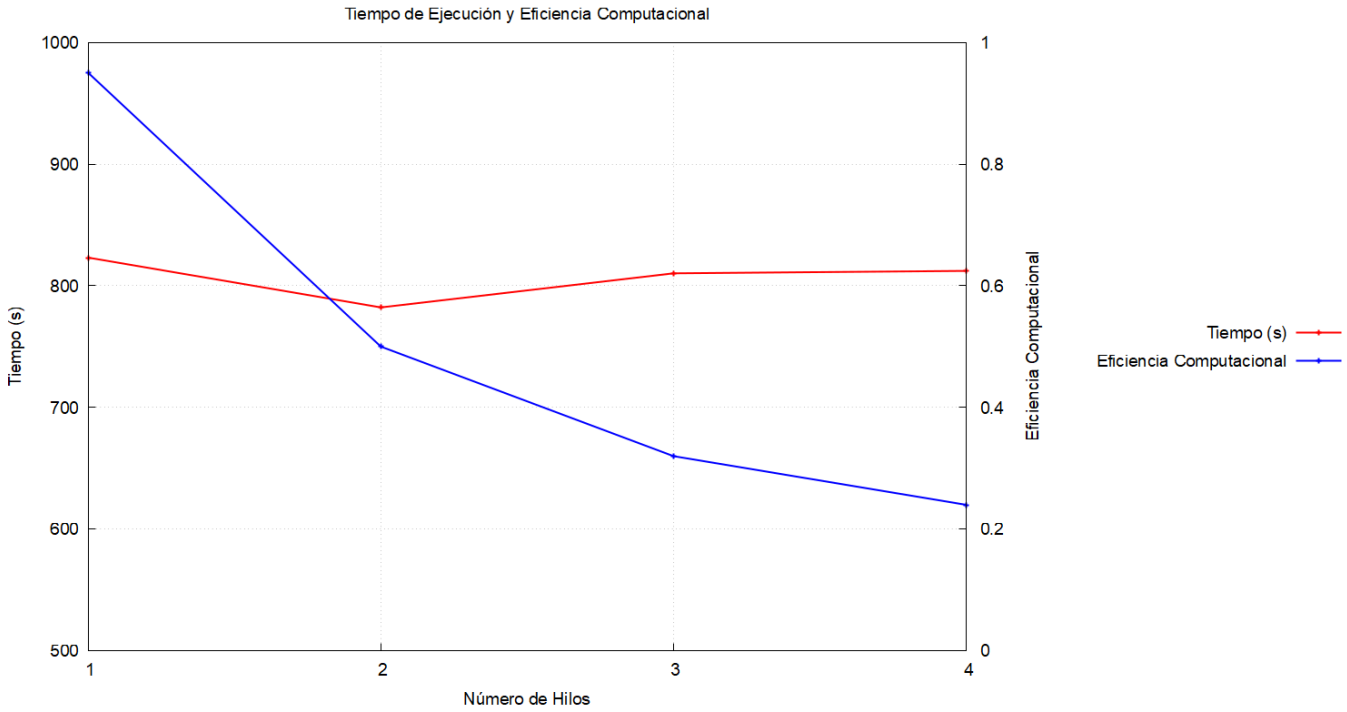

Figura 5: Resultados de Eficiencia Computacional de las versiones de 2 nodos

Como podemos ver, el Speedup de las versiones de 2 nodos sigue una variación similar a las versiones de un único nodo. La diferencia en este caso es que el Speedup ha aumentado considerablemente. Esto se debe a que, al usar pipes exclusivos para cada nodo, se dividen las comunicaciones, por lo que el cuello de botella será menor. Por otro lado, la carga se reparte equitativamente entre los nodos, por lo que cada nodo tendrá una carga menor, y a su vez, un menor tiempo de ejecución, además de que no existen muchas comunicaciones entre los nodos.

Aun así, el aumento de threads no influye significativamente en el aumento del rendimiento, y al superar los 2 threads, este disminuye, al mismo tiempo que la eficiencia decrece notablemente al aumentar el número de threads.

Los resultados de las versiones de varios nodos MPI son los siguientes:

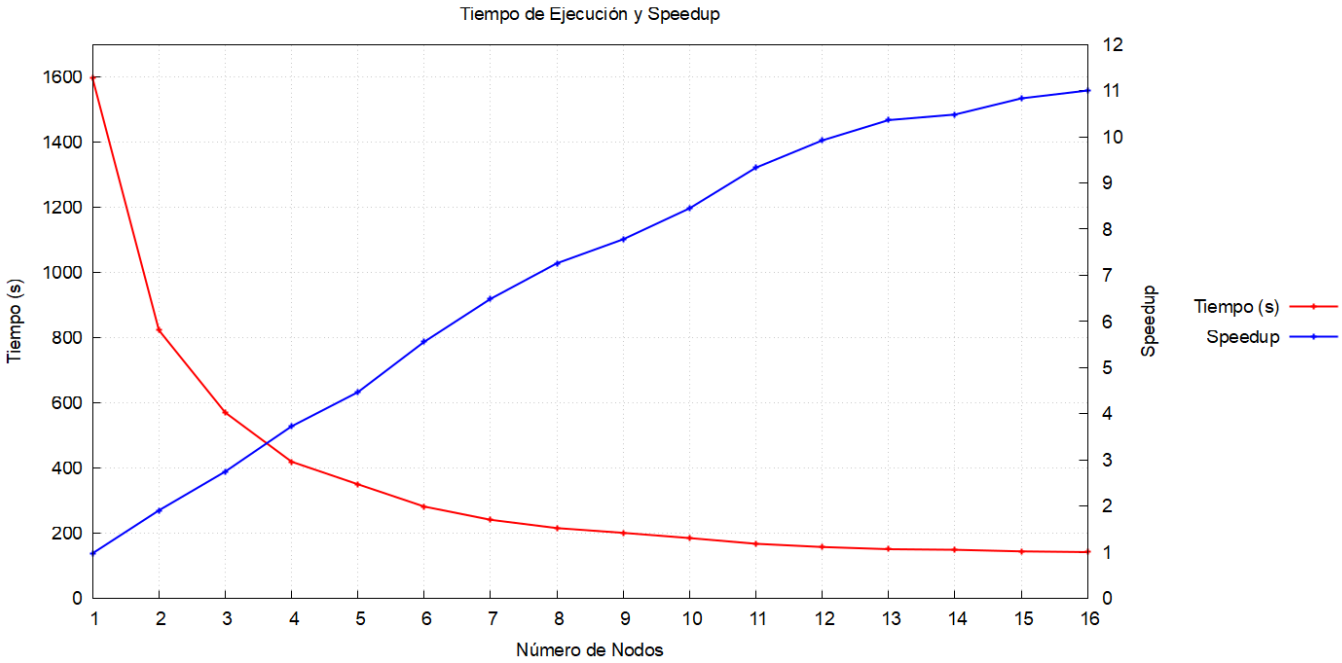

Figura 6: Resultados de Speedup de las versiones de varios nodos

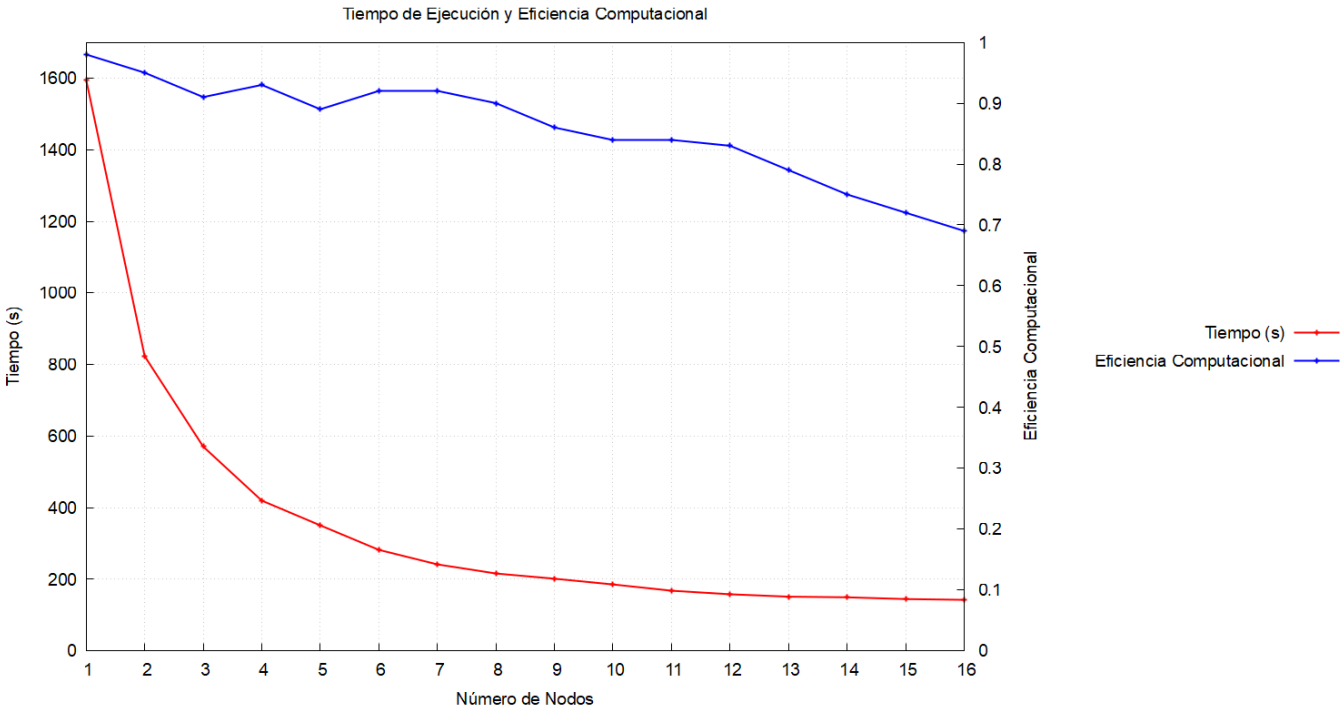

Figura 7: Resultados de Eficiencia Computacional de las versiones de varios nodos

Los resultados de estas versiones son bastante diferentes a los anteriores. Como se comentó anteriormente, el uso de pipes exclusivos para cada nodo hace que se reduzca el cuello de botella, por lo que el rendimiento aumenta considerablemente, al igual que el reparto de carga entre los distintos nodos.

Como podemos apreciar en la gráfica de Speedup, este se mantiene lineal, con valores cercanos a los ideales. Pese a que al superar los 6 nodos se haga uso de *Hyper-Threading*, esta tendencia se mantiene, solo que reduciendo levemente esta mejora.

A partir de los 13 nodos, la mejora vuelve a disminuir debido a los cambios de contexto producidos al tener un número de procesos mayor al de procesadores de la máquina, pero no llega a empeorar el rendimiento global.

Por lo tanto, mientras se reparta la carga correctamente entre los nodos, podemos ir aumentando la cantidad de nodos y mantener una mejora del rendimiento. Esta mejora se irá estabilizando a medida que aumente la cantidad de nodos, ya que la reducción de carga de trabajo será menos significativa. También hay que tener en cuenta que, al aumentar la cantidad de nodos, no es siempre posible asignar la misma carga a todos, por lo que en algunos casos, no podremos obtener una mejora significativa, como se puede ver por ejemplo al pasar de 13 a 14 nodos.

En la gráfica de eficiencia se puede ver que los valores se mantienen a niveles muy altos en todos los casos, disminuyendo levemente al superar los 6 nodos, y otra bajada un poco más pronunciada al superar los 12 nodos, pero manteniendo valores altos.

Podemos concluir en que la paralelización a nivel de threads no ofrece un aumento significativo del rendimiento, aunque podemos obtener una pequeña mejora al usar 2 threads, lo cual resultaría más relevante al aumentar el tamaño del problema. Por otro lado, la distribución y el reparto de carga ofrece una mejora muy significativa sobre la versión secuencial.

5.1. Conclusión

La implementación desarrollada consigue desempeño aceptable, 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 y produce un cuello de botella que perjudica a la paralelización del algoritmo. Una posible solución sería usar distintos pipes a nivel de threads. De esta manera conseguiríamos mejorar el Speedup en la versión de 1 nodo, y además no necesitaríamos controlar el acceso a las comunicaciones, ya que cada thread tendría sus propios pipes.

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 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.

6. Referencias

- Wikipedia. *Algoritmo de la colonia de hormigas*: https://es.wikipedia.org/wiki/Algoritmo_de_la_colonia_de_hormigas
- Christian Blum. *Ant colony optimization: A bibliometric review*: https://www.sciencedirect.com/science/article/pii/S1571064524001258?via%3Dihub
- Dorigo, M., & Stuetzle, T. *Ant Colony Optimization: Overview and Recent Advances*: https://www.sciencedirect.com/science/article/pii/S1571064524001258?via%3Dihub
- OpenMP. *OpenMP: The Open Multi-Processing API*: https://www.openmp.org/
- Open MPI. *Open MPI: A High-Performance Message Passing Library.*: https://www.open-mpi.org/
- Gnuplot. *Gnuplot: An interactive plotting program*: http://www.gnuplot.info/