Proyecto Final

Morales Alcalde, Piero
piero.morales@utec.edu.pe

Motta Paz, Angel
angel.motta@utec.edu.pe

Barreto Zavaleta, Jeanlee
jeanlee.barreto@utec.edu.pe

30 de Julio de 2021

Introducción

Elección del proyecto

Se ha realizado el desarrollo del proyecto de **TSP** (*Travelling Salesman Problem* o *Problema del Viajero*), que es un proyecto **Tipo A** (*Desarrollo de código en paralelo*).

Descripción del proyecto

Se trata de un vendedor que debe minimizar el recorrido entre n ciudades. Puede empezar en cualquiera de ellas y terminar en la misma luego del recorrido. Cada ciudad debe ser visitada solo una vez.

Una de las soluciones es con el método de **búsqueda primero en profundidad** (DFS en inglés), se consideran las posibles ciudades que se pueden visitar desde una ciudad de partida. Así sucesivamente, y calcular el camino mínimo de todas las posibilidades. Por ello, para n ciudades, obtenemos una cantidad total de caminos de (n-1)!

Para este proyecto se va a usar el método de **branch and bound** se verifica si el camino encontrado hasta el momento, es mayor que el mejor camino encontrado. En caso lo sea, se detiene la búsqueda por ese camino. El algoritmo recursivo correspondiente es el siguiente:

```
BB(camino){
    if (camino tiene longitud n) {
        if (camino es el nuevo mejor camino) {
            mejor camino = camino
        }
        else: {
```

```
            for (para todos los caminos aun no recorridos i)
                if (camino U i es menor que el actual mejor camino )
                    BB(camino U i)
            }
        }
    }
}
```

En su versión no-recursiva el algoritmo realiza lo siguiente:

```
camino={0}
push(camino)
while pop(camino){
    if (camino tiene longitud n) {
        if (camino es el nuevo mejor camino) {
            mejor camino = camino
        }
        else: {
            for (para todos los caminos aun no recorridos i) {
                if (camino U i es menor que el actual mejor camino )
                    push(camino U i)
            }
        }
    }
}
```

Para su implementación se utiliza una pila (stack), que se inicializa con la ciudad **0**. En cada iteración se retira la ciudad del tope de la pila. Si el camino tiene longitud n se evalúa, de lo contrario se añaden ciudades si la longitud del camino no es mayor que el del mejor actual. El programa termina cuando la pila está vacía.

Objetivo del proyecto

Una agencia de transporte de materiales, quiere optimizar sus operaciones en Lima. La agencia trabaja en los siguientes distritos: Lima Centro, Lince, Miraflores, Barranco, Rimac, Los Olivos, La Molina, La Victoria, Magdalena, San Borja.

Se necesita elaborar un software en paralelo que resuelva el problema TSP entre estos distritos de Lima. Para ello debe elaborar una matriz con distancias entre estos.

Método

Para solucionar el Problema del Viajero (TSP por sus siglas en inglés) de manera paralela existen muchos enfoques que se han ido propuesto a lo largo del

tiempo. Al ser este un problema NP-Hard, su costo computacional es siempre un factor a tomar en cuenta. Un enfoque muy conocido, y del que hemos estado hablando en la sección anterior, es el método Branch and Bound.

Con este método evitamos realizamos los (n-1)! caminos posibles para obtener la respuesta correcta. Solo se avanza en un camino si y solo sí este se proyecta como uno que minimizará el costo en comparación a un camino previo encontrado.

Para realizar la implementación del método Branch and Bound en paralelo consideramos dividir las tareas en dos clases de nodos. Es así que el nodo maestro será el encargado de coordinar y verificar los caminos que se van encontrando en la marcha. Por otro lado están los nodo trabajadores o esclavos, los cuales se encargan de realizar los cálculos más pesados como son el estimar un límite de costo (bounding), a los posibles caminos óptimos (branching) e ir avanzando en estos si y solo sí el master aprueba dicho camino.

A continuación procederemos a explicar en detalle las tareas que realizan los dos tipos de nodos en nuestra implementación.

Algoritmo general del proceso maestro

El proceso maestro realiza las siguientes actividades:

1. Lectura de la matriz de costos, para la cual se utiliza un arreglo unidimensional para el almacenamiento de las distancias. Esto para facilitar el envío de datos mediante la operación MPI Send.

2. Operación de Broadcast hacia los procesos esclavos de la cantidad de vertices y la matriz de adyacencia.

3. El maestro toma el ID del primer esclavo disponible y lo pone n estado ocupado. El maestro mantiene el control de los procesos esclavos disponibles.

4. El maestro define un costo óptimo global inicialmente con un valor de *infinito.*

5. El maestro crea un arreglo de longitud igual al número total de vertices incialmente incializado en -1 indicando que no hay vertices visitados.

6. El maestro adicionalmente crea un arreglo de longitud 4, para guardar información de control (metadata) que recibirá del esclavo. La información de control esta almacenado de la siguiente forma:

   - *metadata[0]:* costo óptimo global
   - *metadata[1]:* costo actual obtenido por el esclavo
   - *metadata[2]:* último vértice visitado del camino.
   - *metadata[3]:* 1 para indicar si el esclavo ha llegado a nivel hoja, 0 en caso contrario.

Inicialmente estos valores están inicializados en 0.

7. El maestro realiza 2 operaciones Send, en el primero envia el arreglo camino creado previamente y luego envia el arreglo metadata.
8. El maestro entra en un bucle mientras exista algún proceso en estado *ocupado*
9. El maestro al identificar que todos los procesos esclavos se encuentra sin hacer nada realiza una operación Send para solicitarles que terminen su ejecución *(MPI_Finalize())*
10. Finalmente el maestro muestra los resultados de: costo mínimo y el camino óptimo.

Algoritmo general procesos esclavos

Las tareas para los procesos esclavos son las siguientes:

1. El nodo recibe el número de vértices total y la matriz de adyacencia enviados por el maestro (mediante el Broadcast).
2. Crea una lista para los posibles caminos a procesar
3. Recibe una orden desde un nodo α, pudiendo ser este el maestro u otro nodo esclavo. La orden recibida puede ser:
   a) 0 parar: Orden dada solo por el master para finalizar actividades, va al paso 10.
   b) 1 avanzar: Se le asigna un posible camino y luego de ser agregado a la lista de caminos el nodo continua con el proceso.
4. De existir al menos un camino dentro de la lista, va al paso siguiente. De lo contrario, el nodo notifica al master que está libre y retorna al punto 3.
5. Extrae un camino de la lista. De no ser uno completo prosigue al paso 6.
   - Camino completo: Si su costo es menor al costo actual, el nodo esclavo envía el camino al maestro y se regresa al punto 4.
6. Para el camino extraído calcula el lower bound (posible costo mínimo de referencia) y compara dicha proyección con el costo mínimo actual.
7. Si el lower bound no es menor al costo actual, se regresa al punto 4.
8. Notifica al maestro que encontró un posible camino óptimo. Si este aprueba el camino, prosigue al punto 9, caso contrario solo actualiza su costo mínimo local con el global dado por el maestro y retorna al paso 4.
9. Agrega a la lista los posibles caminos obtenidos a partir del camino aprobado (branching) y los distribuye para que sean procesados por los nodos que le solicita al master.
10. Finaliza su trabajo

Resultados

Validación de correctitud

Se hicieron pruebas de correctitud del modelo implementado. En la Figura 1 se muestra un grafo pequeño donde se aprecia de forma clara el camino óptimo.

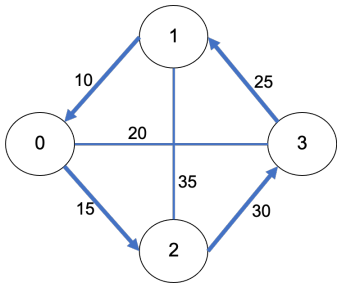

Figura 1: Grafo de 4 vértices - Correctitud

Como se puede ver en la Figura 2 el modelo implementado responde con exactitud el camino óptimo.

```
angelinux@greendev TSP_Project % make run
mpirun -np 2 ./tsp.out

        ***BIENVENIDO***

Elija su opcion:
1: Ingresar matriz de distancias
2: Agregar una nueva ciudad
0: Salir
1
Numero de ciudades: 4

matriz de adyacencia:
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0

Elija su opcion:
1: Ingresar matriz de distancias
2: Agregar una nueva ciudad
3: Calcular TSP
0: Salir
3

==  Resultados TSP
        Costo mínimo:   80
        Camino optimo:  0 2 3 1 0
        Tiempo:         0.00426698 segundos
```

Figura 2: Resultado TSP - Correctitud

Pruebas de escalabilidad

Se han realizado las pruebas de escalabilidad con tamaños de 8, 11, 14 y 17 ciudades (vértices) y con 2, 4, 8, 16 y 32 procesos y se han obtenido los resultados mostrados en la Tabla 1.

Tiempo (seg)	**8 ciudades**	**11 ciudades**	**14 ciudades**	**17 ciudades**
2 procesos	0.0079	0.6248	23.2317	1048.85
4 procesos	0.0052	0.1041	7.1098	529.44
8 procesos	0.0046	0.0462	3.4725	233.49
16 procesos	0.0028	0.0318	1.9887	118.85
32 procesos	0.0027	0.0283	1.8136	96.97

Tabla 1: Tiempos globales para las ciudades y los procesos

Así mismo, para la evaluación de los resultados también se ha construido una gráfica para cada cantidad de ciudades tal como se muestran a continuación.

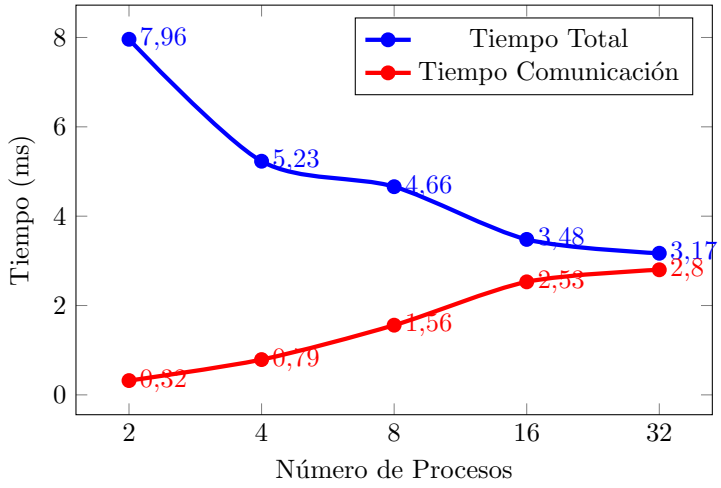

Figura 3: Gráfica de escalabilidad para 8 ciudades

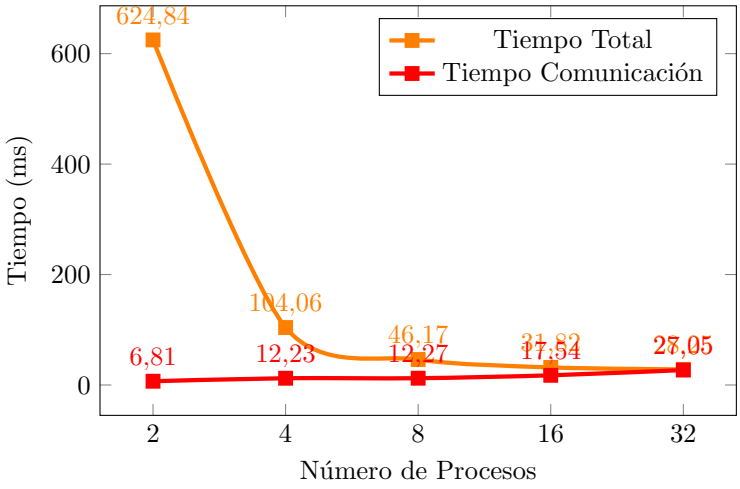

Figura 4: Gráfica de escalabilidad para 11 ciudades

Como se puede observar en la Figura 3 y 4, para un procesamiento de 8 y 11 ciudades el tiempo de ejecución es bastante breve (en el orden de milisegundos) sin embargo se puede notar como el tiempo se reduce a medida que se incrementan el número de procesos.
De forma análoga, se puede apreciar en las Figuras 5 y 6 como para problemas de mayor tamaño (duración en segundos) el tiempo de ejecución se reduce.

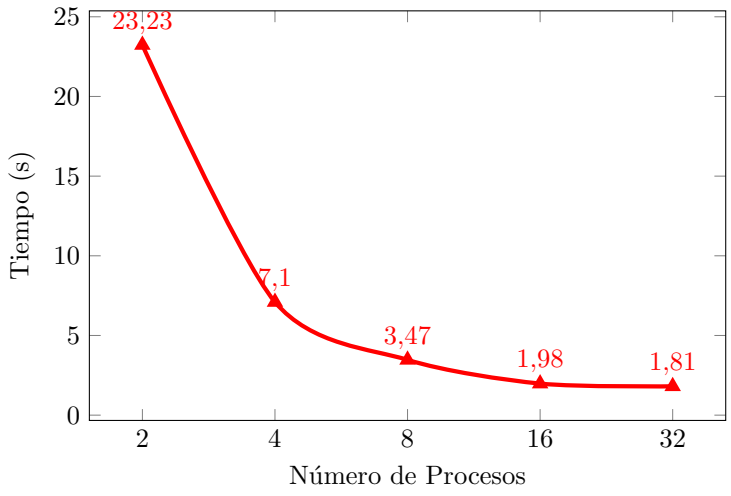

Figura 5: Gráfica de escalabilidad para 14 ciudades

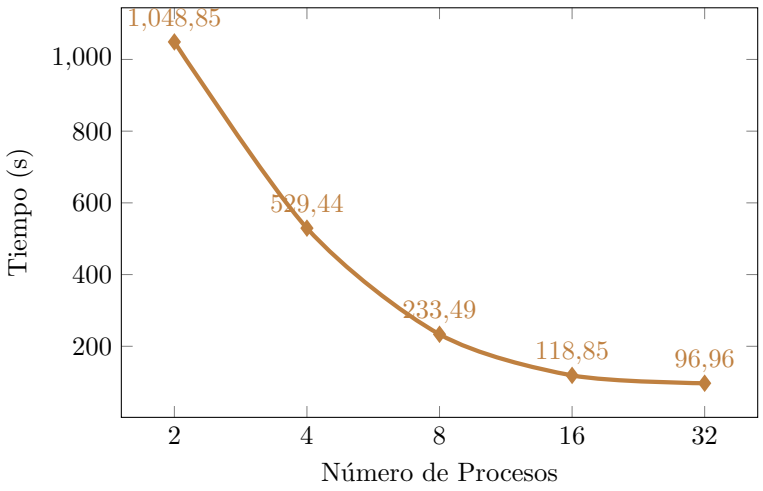

Figura 6: Gráfica de escalabilidad para 17 ciudades

Distritos de Lima

Se han tomado las distancias de 10 distritos de Lima, los cuales son Lima Centro, Lince, Miraflores, Barranco, Rimac, Los Olivos, La Molina, La Victoria, Magdalena, San Borja. La matriz de distancias se muestran en la Tabla 2.

Distancias	Lima Centro	Lince	Miraflores	Barranco	Rimac	Los Olivos	La Molina	La Victoria	Magdalena	San Borja
Lima Centro	0	3.8	7.3	10.6	4.4	13.7	15.9	5	6.3	8.4
Lince	3.8	0	3.8	7.9	7.8	17.6	14	3.5	5	6.5
Miraflores	7.3	3.8	0	4.7	10.4	21.8	15.2	6.7	6	5.5
Barranco	10.6	7.9	4.7	0	14.4	24.8	17.7	9.2	9.1	10.4
Rimac	4.4	7.8	10.4	14.4	0	12.3	19.2	8	10.6	17.2
Los Olivos	13.7	17.6	21.8	24.8	12.3	0	28.9	17.7	19.5	27
La Molina	15.9	14	15.2	17.7	19.2	28.9	0	13.7	18.7	10.7
La Victoria	5	3.5	6.7	9.2	8	17.7	13.7	0	7.4	4.6
Magdalena	6.3	5	6	9.1	10.6	19.5	18.7	7.4	0	9.6
San Borja	8.4	6.5	5.5	10.4	17.2	27	10.7	4.6	9.6	0

Tabla 2: Matriz de distancias de los distritos

Luego de introducir estos valores al software, se puede calcular el camino óptimo para recorrer todos los distritos, como se puede ver en la Figura 7.

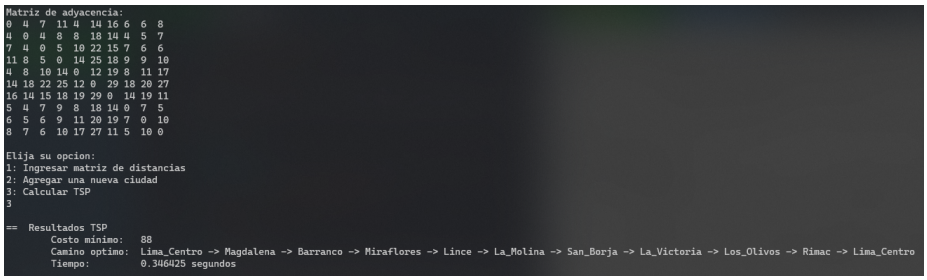

Figura 7: Resultado TSP - Distritos de Lima

Consideraciones

- El modelo implementado, funciona con un mínimo de 2 procesos, ya que el proceso master necesita al menos un proceso esclavo para realizar el computo del TSP.
- El tiempo de comunicación mostrado en las gráficas es un estimado el cual podemos tomar como referencia, ya que este tiempo incluye el costo de comunicación del maestro así como la de los procesos esclavos, sin embargo los procesos esclavos se ejecutan en simultaneo por lo que el costo total de comunicación mostrado es mayor al real.

Conclusiones

Luego de realizar la implementación y hacer las pruebas correspondientes, se ha llegado a las siguientes conclusiones.

- En la implementación realizada con MPI, se pudo comprobar la escalabilidad del modelo al evaluar su rendimiento con 2, 4, 8, 16 y 32 procesos. Esto se ve de forma clara en las gráficas mostradas anteriormente, en donde se ve una tendencia decreciente.
- Se logró realizar la actividad de optimización para que el software sea interactivo mediante un menú para tener la funcionalidad de añadir más ciudades al grafo y se actualice el costo mínimo y la ruta óptima.

Repositorio

Link del código GitHub. Las versiones *beta* solicitadas, se pueden ver en los commits realizados en el repositorio. Además, se ha colocado un archivo `Makefile` para realizar la compilación y ejecución de manera directa.