Computación Paralela y Distribuida

Proyecto Final

Renato Bacigalupo
renato.bacigalupo@utec.edu.pe

Rodrigo Céspedes
rodrigo.cespedes@utec.edu.pe

Benjamín Diaz
benjamin.diaz@utec.edu.pe

Profesor:
José Fiestas

July 30, 2021

1. Introducción

Para esta experiencia, se ha elegido explorar el ***Travelling Salesman Problem*** o **TSP**. Este es un famoso problema del área de grafos el cual busca recorrer dicho grafo a través de un **camino hamiltoniano** mínimo, en otras palabras, encontrar una ruta para recorrer todos los nodos del grafo una única vez y regresar al origen de manera tal que la ruta que tomemos sea la de menor coste disponible.

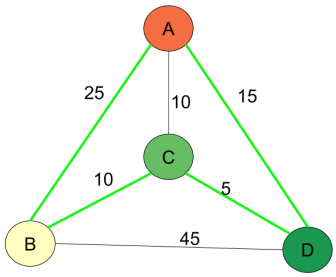

Figure 1: Ejemplo de solución de TSP

El TSP es un problema **NP-Difícil** y, si resuelto de manera ingenua con fuerza bruta, llega a tener un coste de $O(n!)$ pues si tenemos n nodos, el camino tiene que ser de largo $O(n)$ y, para cada nodo $i = 1 : n$ existen i posibilidades para escoger, lo que calculado nos da el coste de $O(n!)$. Existen diferentes métodos para abarcar este problema y volverlo más barato de resolver.

El objetivo de este proyecto es implementar la solución al TSP de manera que podamos paralelizar dicho programa.

2. Método

Para resolver el TSP de una manera más eficiente es necesario encontrar un método para reducir la cantidad de caminos que se tienen que analizar sin que esto afecte a la eficacia del algoritmo. Para ello, se va a usar la técnica de ***Branch and Bound*** la cual consta, en escencia, de *podar* (dejar de recorrer una parte de) el árbol de posibilidades cuando se encuentra una ramificación con un camino que ya no tiene posibilidad de ser mínimo.

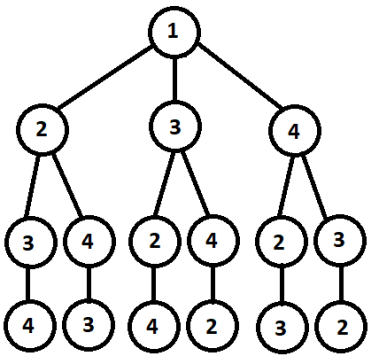

Figure 2: Árbol de posibilidades para un grafo fuertemente conexo de 4 nodos, empezando en el nodo 1

Si se observa bien, para el grafo anterior, las posibilidades de los diferentes caminos se reducen a todos los posibles caminos empezando desde la raíz y terminando en alguna hoja. Para el siguiente ejemplo, supongamos que una arista verde, es una arista que, al sumarse al total acumulado de esa ruta, no ha superado el coste total del mejor camino posible (para empezar, el mejor camino posible será el que recorre todos los nodos en orden), y una arista roja, será una tal que sumada su valor al coste de la ruta, se excede el coste total del mejor camino hasta ahora.

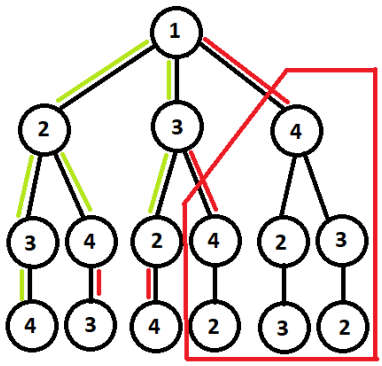

Figure 3: Representación de una poda en Branch and Bound

Se puede observar, en la derecha, como las aristas $1,4$ y $3,4$ hacen que sus rutas excedan el coste total del mejor camino $(1,2,3,4)$ por lo que seguir recorriendo el árbol por los posibles caminos siguientes, no tiene sentido. En tal caso, se hace la *poda* y no se continúa explorando esos caminos. Este va a ser el método que vamos a usar paraa nuestra implementación.

3. Implementación

Para la implementación de la resolución del TSP, se realizaron dos versiones del problema. Una recursiva y otra iterativa usando el método de Branch and Bound, siendo esta última la que intentaremos paralelizar. En todas las versiones el camino empieza en Lima Centro (nodo 0).

3.1. Recursiva

Para la implementación de la versión recursiva utilizamos un `stack<int>` y una variable `cur_dist` que mantiene la distancia recorrida actual, además de esto utilizamos dos variables globales `mejor_camino` y `mejor_camino_distancia`. La llamada inicial es con un stack con 0 (nodo inicial) y `cur_dist = 0`, lo que realiza la función es itera sobre todas las ciudades y calcula la distancia con la última ciudad visitada, en caso esta distancia sea menor que la mejor distancia global, la ciudad actual se *pushea* al `stack`, se llama recursivamente y luego se *popea*. El caso base de la función es cuando el `stack` es de tamaño N, en caso `cur_dist` sea menor a `mejor_camino_distancia` se actualiza `mejor_camino` y `mejor_camino_distancia` con el camino y distancia actual. Cabe resaltar que además de esto se utiliza `bool visitados[N]` para validar que las ciudades se visitan solo 1 vez.

```cpp
void BranchAndBound(stack<int> path, float cur_dist) {
    if (path.size() == N) {
        float dist = cur_dist + GRAFO[LIMA_CENTRO][path.top()];
        if (dist < mejor_camino_distancia) {
            mejor_camino = path;
            mejor_camino_distancia = dist;
        }
    } else {
        int ultima_ciudad = path.top();
        for (const auto& ciudad : ciudades) {
            float dist = cur_dist + GRAFO[ciudad][ultima_ciudad];
            bool esOptimo = dist < mejor_camino_distancia;

            if (esOptimo && !visitados[ciudad]) {
                visitados[ciudad] = true;
                path.push(ciudad);
                BranchAndBound(path, dist);
                path.pop();
                visitados[ciudad] = false;
            }
        }
    }
    path.pop();
}
```

Listing 1: Versión recursiva

3.2. Iterativa

Para la versión iterativa, vamos a utilizar un método de reducción utilizando matrices. Inicialmente vamos a tener la matriz de adyacencia del grafo, luego pasaremos a reducirla, es decir, obteniendo el menor valor del cada fila y columna y restándole a la misma ese valor. Además, necesitamos sumar los números de reducción para obtener el costo de reducción total de esa matriz. Una vez tengamos ese valor vamos a definir el costo de camino desde el nodo `root` hasta el nodo actual como $C(j) = dist(j, j-1) + c(j-1) + c\_red(j)$, donde $dist(j, j-1)$ es la distancia del nodo anterior al nodo actual, pero calculada usando la matriz reducida del nodo anterior, $c(j-1)$ es el costo calculado del nodo anterior, y $c\_red(j)$ es el costo de reducción de la matriz actual. En la primera iteración $c(j-1)$ es 0, y en las siguientes iteraciones calculamos la matriz reducida a base de la matriz reducida del nodo anterior, pero con la condición de que la fila y columna de los nodos anterior y actual son llenadas con 0 o ∞, después se reduce como explicamos anteriormente. Así se calcula para todos los nuevos nodos añadidos en forma de árbol hasta que se llega a el primer nodo hoja, una vez llegado a este calculamos el costo de ese nodo y lo determinamos como `costo_minimo_actual`, luego en las siguientes iteraciones solo procedemos a calcular si es que el nodo tiene un costo menor o igual a el de `costo_minimo_actual`, si no lo tiene no calculamos este subárbol camino.

Para la implementación utilizamos un `priority_queue` para siempre calcular el nodo con menor costo y un `while` que solo corre si es que el `priority_queue` tienen nodos dentro. En cada iteración, añadimos todos los nodos a los que podemos llegar al árbol y calculamos el costo del nodo actual (es decir, la matriz reducida), además hacemos uso de un `unordered_map` para no volver a calcular los nodos que ya hemos visitado en

cada posible camino. Utilizamos un `struct nodo` donde guardamos todo lo que vamos a necesitar como la matriz reducida de ese nodo, su `unordered_map`, su costo y su nodo padre.

```
while(pq.empty() == false) {
    auto temp = pq.top();
    pq.pop();
    if (temp->coste <= upper) {
        vector<nodo*> nodos_a_los_que_llegas;
        bool parada = false;
        for (int i = 0; i < N; i++) {
            if (GRAFO[temp->ciudad][i] != DBL_MAX && temp->nodos_visitados[i] == false) {
                nodo* nodo_nuevo = new nodo(temp, nullptr, DBL_MAX, i, temp->nodos_visitados);
                nodos_a_los_que_llegas.push_back(nodo_nuevo);
                parada = true;
            }
        }

        if (parada == false) {
            upper = temp->coste;
            if (mejor_camino == nullptr) { mejor_camino = temp; }
            else {
                mejor_camino = temp->coste < mejor_camino->coste ? mejor_camino = temp : mejor_camino = mejor_camino;
            }
        }
        else {
            for (auto n : nodos_a_los_que_llegas) {
                auto data = reducir(temp->mat, temp->ciudad, n->ciudad);
                n->set_matrix(data.first);
                n->coste = temp->mat[temp->ciudad][n->ciudad] + temp->coste + data.second;
                n->nodos_visitados[n->ciudad] = true;
                pq.push(n);
            }
        }
    }
}
```

Listing 2: Versión iterativa

3.3. Paralelización

Para paralelizar este código (utilizando OpenMP) es necesario analizar que parte es la que mejor se puede paralelizar. Si vemos el `while` explicado anteriormente vemos que este tiene una dependencia considerable. Es necesario la información del nodo anterior para poder calcular la información del nodo actual. Sin embargo, es posible paralelizarlo de en niveles del árbol, es decir si ya tenemos la información del nodo `root` (o un nodo interno), sus nodos hijos se pueden calcular al mismo tiempo, ya que la información del padre ya está calculada. Sin embargo, usando directivas OpenMP no podemos lanzar una cierta cantidad de nodos que calculen el mismo task para cada "iteración" del `while`. Y si fuéramos a separar este `while` la idea inicial del algoritmo de *podar* no estaría siendo respetada, porque tendríamos que calcular muchos nodos

más de los que probablemente calcularíamos si fuéramos *podando* más, haciendo así al código menos escalable. Si no podemos paralelizar el `while`, la siguiente parte que se puede paralelizar más es la función donde se calcula la matriz reducida de cada nodo. El cálculo matricial es perfectamente paralelizable, y nosotros tratamos de llegar a esta perfección paralelizando el primer `for` que engloba el cálculo del menor y el cálculo de los nuevos valores de la matriz reducida. Además, es necesario calcular el costo de reducción para esto utilizamos una reducción de suma para poder sumar todos los subvalores de `acumulado_reduccion`. Y por último, utilizamos `#pragma omp atomic update` y `#pragma omp atomic write`.

```
pair<double**, double> reducir(double **mati, int from, int to){

    double **mat = new double *[N];
    for (int i = 0; i < N; i++) {
        mat[i] = new double[N];
    }

    for(int i=0; i<N; ++i){
        for(int j=0; j<N; ++j){
            mat[i][j] = mati[i][j];
        }
    }

    for(int i=0; i<N; ++i){
        mat[i][i] = DBL_MAX;

        if(from != to) mat[i][to] = DBL_MAX;
        if(from != to) mat[from][i] = DBL_MAX;
    }


    mat[to][from] = DBL_MAX;

    double acumulado_reduccion = 0;

    double min_fila;
    double min_columna;
    int i,j;
    #pragma omp parallel for private(i, min_fila, min_columna, j)
   reduction(+:acumulado_reduccion)
    for(i=0; i<N; ++i){
        min_fila = DBL_MAX;
        min_columna = DBL_MAX;
        for(j=0; j<N; ++j) {
            #pragma omp atomic write
            min_fila = min(min_fila, mat[i][j]);
            #pragma omp atomic write
            min_columna = min(min_columna, mat[j][i]);
        }
        if(min_fila > 0 && min_fila != DBL_MAX) {
            for(j=0; j<N; ++j) {
                if(mat[i][j] != DBL_MAX)
                    #pragma omp atomic update
                    mat[i][j] -= min_fila;
            }
            if(min_fila != DBL_MAX)
                #pragma omp atomic update
```

```
47                  acumulado_reduccion += min_fila;
48
49          }
50          if(min_columna > 0 && min_columna != DBL_MAX) {
51              for(j=0; j<N; ++j) {
52                  if(mat[j][i] != DBL_MAX)
53                      #pragma omp atomic update
54                      mat[j][i] -= min_columna;
55              }
56              if(min_columna != DBL_MAX)
57                  #pragma omp atomic update
58                  acumulado_reduccion += min_columna;
59
60          }
61      }
62      return {mat, acumulado_reduccion};
63  }
```

Listing 3: Versión paralelizada

Otro acercamiento a la paralelización es dividir el recorrido del árbol de posibilidades en varios hilos desde la raíz, de manera tal que se vayan analizando arboles de caminos completos en varios hilos. Cada hilo realizará Branch and Bound en el subarbol correspondiente. Entonces tendríamos algo como esto.

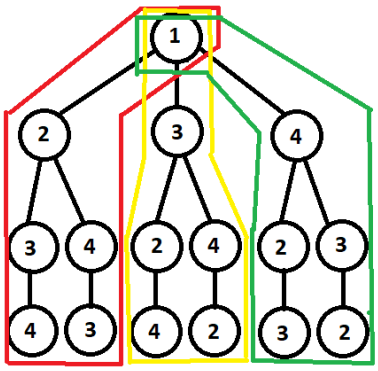

Figure 4: Representación de Parallel Search, en donde cada color corresponde a una búsqueda distinta en cada hilo

Para implementar este sistema, lo que hicimos fue convertir el while anterior, en una función, que solo *pusheará* un vecino solamente en la primera iteración, para así general los subarboles. Esta función se llamará una vez por cada vecino del nodo raíz. El problema con este método, es que para paralelizarlo, utilizamos sections, pero no encontramos una forma de crearlos dinámicamente, por lo que las pruebas fueron simplemente de concepto y escribiendo cada section manualmente para diferentes N. El código es demasiado grande y tedioso para exponerlo aquí en el reporte, pero la idea es tal cual la mostrada en el gráfico, cada color va a un hilo y se trata de hallar el mínimo camino en paralelo.

4. Resultados

N	Secuencial
n = 5	0.00005
n = 8	0.003956
n = 10	0.158424

Figure 5: Resultados sin gasolina. (segundos)

N	Secuencial
n = 5	0.00005
n = 8	0.002449
n = 10	0.060802

Figure 6: Resultados con gasolina. (segundos)

N	Iterativo	Parall Redux	Parall Search
n = 10	0.001026	0.0134042	0.0192634
n = 15	0.006537	0.0096221	0.039609
n = 20	0.046557	0.061687	0.14524
n = 25	1.12887	1.03375	0.707459
n = 30	9.11051	7.17261	17.4251

Figure 7: Resultados sin gasolina. (segundos)

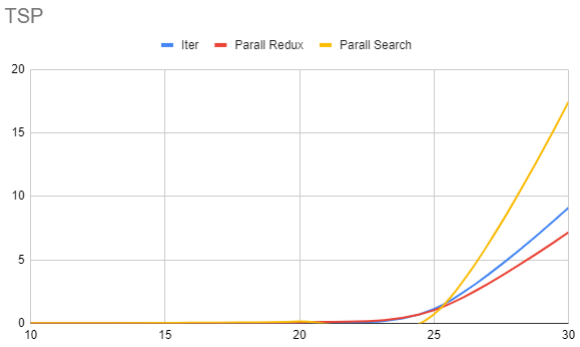

Figure 8: Gráfico

N	Iterativo	Parall Redux	Parall Search
n = 10	0.00094	0.118538	0.0094419
n = 15	0.004701	0.0335857	0.0439378
n = 20	0.120528	0.17957	0.314048
n = 25	1.18213	0.950216	3.13729
n = 30	12.7545	19.6222	25.7872

Figure 9: Resultados con gasolina. (segundos)

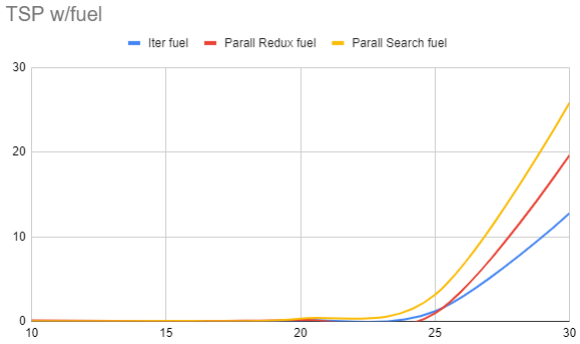

Figure 10: Gráfico

5. Conclusiones

Podemos concluir que TSP es un problema que se puede paralelizar hasta cierto punto (dependiendo en la implementación). Es posible usar la metodología explicada anteriormente en la sección de Paralización, en donde cada hilo calcula un nodo y para optimizar más dentro de esos nodos podemos ejecutar código en GPU. Al ser cálculos matriciales estos se pueden calcular mucho más rápido en una tarjeta gráfica. Esto aumentaría mucho más la performance y la escalabilidad del algoritmo. Sin embargo, para alcanzar eso es necesario hacer algunos cambios a nuestra implementación, primero no podríamos usar OpenMP, ya que las directivas no se ajustan a la idea de paralización que proponemos, sería apropiado usar otras librerías como `pthread` por ejemplo. Donde podemos tener mucho más control de cada thread individual. Por otro lado, necesitaríamos definir una función que sea de tipo observer, es decir necesitamos que todos los hilos estén al tanto del cambio a la variable `costo_minimo_actual`. Debido a que, una vez este se actualice es necesario que todos los hilos determinen si es necesario que sigan corriendo o no. Si no necesario que corran deben terminar para dar más recursos computacionales a el cálculo del camino óptimo actual.

Es interesante ver cómo para Ns pequeños, el más rápido es el algoritmo iterativo, pero cuando el N va creciendo, el algoritmo de *Parallel Search* y el de *Parallel Redux* aumentan su velocidad respecto al iterativo. Creemos que esto se debe a que para valores de N pequeños los dos algoritmos de paralelización crean demasiado overhead al crear y destruir hilos lo cual aumenta mucho el tiempo de ejecución. Luego, para valores de N intermedios como vemos en la gráfica el algoritmo de *Parallel Search* es el más eficiente (como se ve en la gráfica), esto se debe a que al no tener tantos nodos que calcular es mejor que cada thread pueda calcular un subárbol grande en

vez de generar más threads para calcular valores de las matrices como lo hace *Parallel Redux*. Por último, para valores de N altos el algoritmo de *Parallel Redux* es el más eficiente, esto se debe a que al aumentar valores de N, el algoritmo de *Parallel Search* empieza a calcular muchos nodos que no se deberían tener que calcular, ya que en otro algoritmo estarían *podados*. Esto aumenta mucho la complejidad y hace que el algoritmo de *Parallel Redux* que acelera las operaciones matriciales que son las que aumentan complejidad.

Por otro lado, el algoritmo tomando en cuenta otro factor como la Gasolina. Tiene valor bastante diferente a los demás como se ve en la gráfica. En este algoritmo toma en cuenta otro factor para calcular costo que es la gasolina que se gasta de viajar de una ciudad a otra. Una vez se termina esta es necesario rellenar el tanque lo que aumenta costo al par de nodos que se estén calculando en un camino dado. Para los resultados de este caso podemos intuir que al añadir este factor extra la naturaleza del código inicial es algo corrompida. Ya que, es posible que se esté calculando el camino óptimo en un momento, pero este el momento que se queda sin gasolina aumenta su costo considerablemente. Esto tiene dos consecuencias, la primera es que se empezará a calcular muchos más nodos de lo normal (evadiendo un poco la idea de *podar* de el algoritmo inicial y la segunda es que debido a estos cálculos extra los algoritmos en paralelo para valores de N todavía algo pequeños como los que hemos utilizado se ven perjudicados por el overhead de crear tantas threads, aumentando así su costo computacional. Para poder ver una mejora es necesario usar valores de N mucho más grandes y además hacer uso de un cluster potente o una computadora potente, para tomar ventaja de la gran cantidad de hilos que se podrían usar al mismo tiempo.

6. Repositorio

- `https://github.com/MenuMarino/Proyecto-paralela`