Paralelismo de solución a TSP con Branch and Bound en sistemas distribuidos con MPI

Maria Tidball Binz

Resumen

Este informe investiga la paralelización de la solución Branch and Bound para Travelling Salesman Problem. Primero se analiza el problema conceptualmente, y se examinan soluciones importantes en el último siglo. Se profundiza con un análisis de la solución del algoritmo Branch and Bound basado en Depth Search First de Peter Pachecho en "An Introduction to Parallel Programming"[1], la implementación serial y su paralelización. Se define una versión con Static Partitioning y se plantea el problema de balance de carga.

En base a la implementación paralela desarrollada con MPI (usamos la implementación OpenMPI 4.0.2)[2], se realizaron pruebas en un cluster de 8 nodos con un máximo de 64 procesos. Se usaron grafos de 5, 8, 10, 12 y 15 ciudades para tres casos distintos; sin camino, con un solo camino y con caminos parciales. En todas las pruebas, se analizaron las diferencias en Speedup y Eficiencia entre 2, 4, 8, 16, 32 y 64 procesos, en comparación con la versión serial más rápida del mismo algoritmo.

El análisis demuestra que hay mayores beneficios en el uso de 8 procesos para la mayoría de los casos. La conclusión es que este número de procesos se condice con el promedio del pequeño rango de ciudades de prueba utilizado. También se considera la condición del cluster, donde los procesos internos a un nodo tienen menor latencia de comunicación que procesos inter-nodo. A futuro, se considera importante correr las mismas pruebas con una versión con mapeo dinámico, analizar la diferencia en Speedup y Eficiencia en base al número de procesos utilizado, comparando la diferencia con el algoritmo con mapeo estático.

Introducción

Existe un problema en informática que se llama Travelling Salesman Problem. Se puede pensar desde el punto de vista de planificar un viaje. Cuando visitamos más de una ciudad en un viaje, sabemos cuales son nuestros próximos puntos de salida. Pero,¿sabemos cuál es el camino óptimo entre los lugares que queremos visitar? Descubrir la distancia mínima entre una serie de puntos (volviendo al lugar de origen), se conoce como el Travelling Salesman Problem (o TSP) - el problema del vendedor ambulante. En TSP, un vendedor recibe una lista de ciudades que debe visitar y el costo para viajar entre cada par de ciudades. Su problema es que necesita visitar cada ciudad una sola vez, volviendo a su lugar de partida, y debe hacerlo con el menor costo posible. Una ruta que sale desde su ciudad, visita una sola vez a cada ciudad, y vuelve a su ciudad de partida se llama un camino. Por esto se puede decir que TSP es encontrar el camino de costo mínimo. Este problema tiene gran importancia en el mundo de la matemática, y consecuentemente, en el mundo de las ciencias de la computación[3]

[4]Este problema es cada vez más vigente en el mundo contemporáneo, porque afecta una serie de áreas que requieren de sistemas de alto cómputo para planificar sus operaciones. Por ejemplo, una empresa de ventas que necesita planificar el camino más corto para despachar sus productos de manera óptima alrededor del mundo es un problema similar a TSP.

En los siguientes capítulos haremos un recorrido por algunos conceptos necesarios para entender TSP. Luego, haremos un recorrido de algunas soluciones e implementaciones secuenciales conocidas. Vamos a analizar la solución e implementación secuencial Branch and Bound con mayor profundidad, analizando la forma algorítmica que toma. Luego veremos una implementación de la solución de búsqueda en un árbol, Branch and Bound, paralelizada usando una librería de Message Passing Interface (MPI) en C para sistemas distribuidos. En este último análisis entraremos en los detalles de MPI para entender los detalles del algoritmo. Comparamos métricas de resoluciones secuenciales y paralelas para entender las posibilidades de speedup y eficiencia en la paralelización en sistemas distribuidos.

---

1 Pacheco, P.S. An Introduction to Parallel Programming. 2011

2 Documentación OpenMPI https://ccportal.ims.ac.jp/en/node/2635

3 Balas, E. . *The prize collecting traveling salesman problem. Networks, 19(6), 621–636.* 1989 doi:10.1002/net.3230190602

4 David L. Applegate; Robert E. Bixby; Vašek Chvátal; William J. Cook, The Traveling Salesman Problem: A Computational Study , Princeton University Press, 2007. "TSP [..] is in fact one of the most intensely investigate problems in computational mathematics." p1, ch1

Capítulo 1: Marco Teórico

Hoy en día, no se puede hablar de una sola "solución" a TSP, ya que es un problema que sigue generando preguntas en términos generales y teóricos tanto en la matemática como en las ciencias de la computación. TSP se sigue estudiando por que no existen soluciones que sean óptimas para aplicar a cualquier versión del problema mejores que simplemente buscar por todas las posibles soluciones y compararlas. Si pudiéramos encontrar una solución a TSP que sea mejor en todos los casos que la fuerza bruta, entonces hay muchos otros problemas que podemos solucionar rápidamente. Esto es lo que lo hace un problema común de optimización.

¿Que significa que este sea considerado un problema de **optimización?**

En el cómputo se entiende así: "En cómputo, la **optimización** es el proceso de modificar un sistema para mejorar su eficiencia y el uso de los recursos disponibles (rendimiento)."[5]

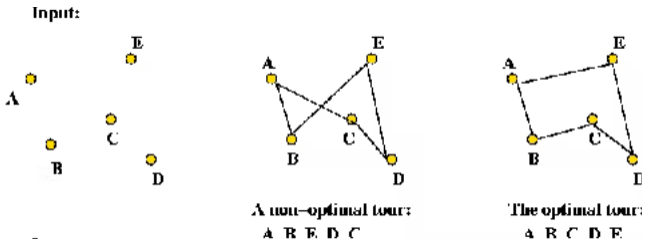
En las matemáticas la optimización hace referencia a hallar máximos o mínimos de una función:

> "De forma general, la optimización incluye el descubrimiento de los "mejores valores" de alguna función objetivo dado un dominio definido" [6]

Busquemos entonces la definición en matemáticas para Optimización:

> "En matemáticas la **optimización** **programación matemática** intenta dar respuesta a un tipo general de problemas matemáticos donde se desea elegir el mejor entre un conjunto de elementos. En su forma más simple, el problema equivale a resolver una ecuación de este tipo: $\max(\min) f(x)$ $x \in \Omega \subset \mathbb{R}^n$
> Donde $x = (x_1, \ldots, x_n)$ es un vector y representa variables de decisión, $f(x)$ es llamada función objetivo y representa o mide la calidad de las decisiones (usualmente números enteros o reales) y Ω es el conjunto de puntos o decisiones factibles o restricciones del problema. Algunas veces es posible expresar el conjunto de restricciones como Ω solución de un sistema de igualdades o desigualdades.
> $g(x_1, \ldots, x_n) \quad < \quad ($
> $h(x_1, \ldots, x_n) \quad = \quad ($
>
> Un problema de optimización trata entonces de tomar una decisión óptima para maximizar (ganancias, velocidad, eficiencia, etc.) o minimizar un criterio determinado (costos, tiempo, riesgo, error, etc). Las restricciones significan que no cualquier decisión es posible." [7]

Volviendo al tema en cuestión, vemos que la optimización necesaria en TSP es encontrar un mínimo absoluto para la suma de caminos entre un número dado (N) de nodos, sin repetición y volviendo siempre al primer nodo.



Ver referencia de imagen a pie de página. [8]

En TSP, tratamos de un grafo dirigido y pesado, es decir, las conecciones tienen dirección y valor. El valor del camino entre un par de nodos tiene un valor medible. Por ejemplo, su tiempo de recorrido o de distancia. Hay variaciones de TSP en las cuales el grafo es simétrico o asimétrico. Estas variaciones facilitan diferentes tipos de soluciones. Esto significa que dado un nodo A y un nodo B, que exista la conexión AB no significa que necesariamente haya una conexión BA, y que la distancia de un nodo A a un nodo, B, no es necesariamente equivalente a la distancia de B a A. Estas variaciones en el problema se deben tener en cuenta al momento de idear una solución. [9]

Incógnita ¿P = NP?

TSP es un problema importante para la matemática y para la computación porque entra en los problemas que se definen dentro de la incógnita "P = NP", aún irresuelta. Para la mayoría de científicos en el campo de la computación y la matemática, P no es igual a NP. ¿Qué significa esto? En resumen, consideremos que P es un conjunto de problemas que pueden ser **solucionados** en tiempo polinómico. Además, NP es el conjunto de problemas que pueden ser **verificados** en tiempo polinómico. Actualmente, no se puede comprobar que todo problema en el conjunto P, también se comprende en el conjunto NP. Si fuese así, muchos problemas que ahora son muy difíciles de resolver, se resolverán de manera fácil. En general, son problemas que en algún punto requieren de búsqueda exhaustiva de combinaciones para su resolución, por lo cual los tiempos de resolución de dichos problemas se hacen cercanos al infinito mientras crece N. Para resolver TSP se deben probar y comparar todas las combinaciones posibles. Esto lleva muy alto de resolución para Ns grandes, que es el tiempo factorial. En algunos casos usando algoritmos especiales, el tiempo de resolución puede llegar a bajar hasta ser exponencial. Adicionalmente, en TSP, no existen algoritmos

---

[5] Wikipedia - Optimización https://es.wikipedia.org/wiki/Optimizaci%C3%B3n_(c%C3%B3mputo)

[6] Wikipedia - Optimización https://es.wikipedia.org/wiki/Optimizaci%C3%B3n_(c%C3%B3mputo)

[7] Diccionario - Optimización http://diccionario.sensagent.com/Optimizaci%C3%B3n%20(matem%C3%A1tica)/es-es/

[8] Imagen de https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html

[9] Existe una librería especializada sobre problemas TSP en torno a la optimización de la universidad de Heidelberg. http://comopt.ifi.uni-heidelberg.de/projects/

de verificación; para verificar qué llegamos a la solución correcta, no queda otra manera que resolver el problema.

Se puede clasificar a los algoritmos en relación a TSP en dos campos. Dependiendo de la formulación de TSP, entra en diferentes clasificaciones de este tipo de problemas, NP- dificil o NP-completo.
Para TSP, existen algoritmos que devuelven una solución exacta (el camino más corto entre todos los caminos posibles), y algoritmos que se aproximan a un camino mínimo, pero no garantizan que sea el más corto entre todos los caminos. Estos últimos se llaman heurísticas y pueden ser muy eficientes para casos específicos. Hoy en día, se encuentran heurísticas que devuelven soluciones optimizadas para Ns de millones, con tiempos en segundos - pero aún no hay una solución exacta genérica que sea factible para cualquier N, o para cualquier variación del problema.

Dependiendo de la variación del problema, si buscamos una solución exacta, TSP se considera un problema NP-Completo. Si simplemente buscamos una solución cercana, entonces se considera NP-difícil.

> "TSP in nominal form is considered *NP-Complete*, when attempted using exact deterministic heuristics. When solving the problem using optimization algorithms and approximation, then the problem tends to be *NP-Hard*."[10]

Nos enfocaremos en una solución exacta, por lo tanto abordamos TSP como NP-completo. TSP entra en un conjunto de problemas que no tienen fácil solución y encontrar algoritmos de optimización es muy importante para una gran variedad de campos. Si fuera posible resolver TSP , se podrían resolver un gran número de otros problemas fácilmente para todos los casos. Este tema es muy amplio y basta con saber que no solo **no** hay soluciones conocidas para TSP que sean mejores en todos los casos que la búsqueda exhaustiva, sino que es muy poco probable que se encuentre una. Al estar contemplado en el marco de esta incógnita, TSP es un problema de optimización. Por esta razón, se siguen publicando estudios y buscando mejoras para números más grandes de nodos. Para leer con detalle sobre problemas NP-completos, se recomienda Computers and Intractability de Garey y Johnson. [11]

Complejidad de tiempo

La resolución exacta más inmediata para resolver TSP es la solución de fuerza bruta. Esta solución implica probar todas las combinaciones de conexiones entre nodos y comparar las soluciones entre ellas para encontrar la más corta. Este algoritmo conlleva una complejidad de tiempo factorial, el cual en la computación es un tiempo extremadamente costoso. Esta solución utiliza permutaciones, con un tiempo de ejecución de (N!) para buscar cada camino, y para encontrar el más corto.
Con esta solución el tiempo de resolución crece de manera factorial con cada incremento de N, y para Ns mayor a 20 el tiempo necesario se hace inviable. El algoritmo que analizaremos más adelante (Branch and Bound simple usando Depth First Search) no mejora este tiempo en el peor caso, aunque en la práctica podemos tener tiempos mejores al acortar el campo de búsqueda. Soluciones conocidas más sofisticadas que se basan en Branch and Bound cómo Held-Karp llegó a mejorar la complejidad de tiempo a $O(n^2 2^n)$.

---

10 Davendra, D., et al, "CUDA Accelerated 2-OPT Local Search for the Traveling Salesman Problem", in Novel Trends in the Traveling Salesman Problem. London, United Kingdom: IntechOpen, 2020 [Online]. Available: https://www.intechopen.com/chapters/72774 doi: 10.5772/intechopen.93125

11 Garey, M.R.; Johnson, D.S. Computers and Intractability: A Guide to the Theory of NP-Completeness. New York: W.H. Freeman. 1979

Capítulo 2: Cronología de orígenes y de optimizaciones sobre TSP

Basándonos en el sitio Concorde[12], haremos un recorrido sobre las optimizaciones secuenciales más conocidas para las soluciones y aproximaciones a TSP. Este sitio web del equipo de investigación de David Applegate, AT&T Labs - Research, Robert Bixby, ILOG and Rice University, Vašek Chvátal, Concordia University y William Cook University of Waterloo, acompaña al libro "In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation", de William Cook[13].

Primeras aproximaciones: Hamilton Cycle

William Cook hace un extenso recorrido de los orígenes del problema en el libro "In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation" [14]. Sin embargo, para no perder el foco de este informe, podemos tocar algunos puntos relevantes para nuestro propósito. Dos nombres importantes que dieron reconocimiento a la importancia del problema son Leonhard Euler y W.R. Hamilton en 1800s. Leonhard Euler dió pié a la teoría de grafos con su planteo del problema "los puentes de Königsberg", y Sir William Rowan Hamilton definió un problema inicialmente conocido como Hamilton Cycle, una generalización y versión simplificada de TSP. Hamilton y Euler se aproximan al problema desde perspectivas diferentes. Hamilton se pregunta cómo atravesar el grafo pasando por cada nodo una sola vez, volviendo al nodo de partida. Euler se plantea como atravesar un grafo pasando por cada vértice una sola vez. Pero aún no se considera la necesidad de encontrar el camino más corto. Estas preguntas dan inicios a las consecuentes formulaciones del problema.[15]

Consiguientemente en los 1930s, se empiezan a encontrar mayores menciones a este problema como punto de investigación en papers y conferencias. Karl Menger es uno de los tales que define el problema y busca nuevas reglas para mejorar sobre la solución con fuerza bruta.
En un coloquio de matemáticos el 5 de febrero de 1930, Menger menciona el problema del Hamilton Cycle como el "Messenger Problem" -

> "We designate as the Messenger Problem (since this problem is encountered by every postal messenger, as well as by many travelers) the task of finding, for a finite number of points whose pairwise distances are known, the shortest path connecting the points [...] This problem is of course solvable by finitely many trials. Rules that give a number of trials below the number of permutations of the given points are not known."[16]

Karl Menger considera una heurística llamada Nearest Neighbour (de ahora en más NN) para encontrar un Hamilton Cycle, considerando adicionalmente la necesidad de encontrar el camino más corto. Reconoce que NN mejora los tiempos pero no encuentra el camino más corto. "The rule, that one should first go from the starting point to the point nearest this, etc., does not in general result in the shortest path." .[17]

En los 40s aparecen publicaciones con referencias al problema y en 1949 en J.B. Robinson, "On the Hamiltonian game (a traveling-salesman problem)", RAND Research Memorandum, aparece la primera mención escrita del nombre "Travelling Salesman Problem", que es como lo conocemos hoy.[18]

En los años 50, tuvo lugar una solución innovadora muy famosa. En 1954, tres investigadores desarrollan una solución exacta usando Linear Programming con el método "cutting plane" para optimizar TSP hasta N de 49. Este paper seminal de G. Dantzig, R. Fulkerson, and S. Johnson[19] soluciona el mayor N encontrado hasta ese entonces, e indicó un gran salto en la construcción del conocimiento sobre la resolución. A pesar de que el grupo de investigación no provee un algoritmo computacional para resolver TSP, logra definir el problema matemáticamente, proveyendo los pasos de resolución matemática y las ideas necesarias que dan pié a los avances posteriores. Cabe destacar que Georg Danzig fue autor de uno de los grandes algoritmos del siglo 20 (simplex algorithm in linear programming, del año 1947), el cual fue un necesario precursor de esta solución.[20]

Otro conjunto de soluciones posteriores, tanto para soluciones exactas como para heurísticas, se basan en utilizar árboles de búsqueda, con diferentes restricciones para encontrar caminos óptimos. Entre ellos, se destacan Held-Karp, Christofedes y Lin-Kernighan. Luego nos enfocaremos en una versión simplificada y general de búsqueda en árboles con Branch and Bound para encarar la paralelización.

Hago una mención especial a desarrollos más recientes para soluciones aproximadas de TSP, con algoritmos evolutivo genéticos y su paralelización. La mayoría de las investigaciones actuales hacen enfoque en estos métodos, inclusive para la paralelización en arquitectura distribuida con MPI[21], o con CUDA para GPU.

---

12 Concorde https://www.math.uwaterloo.ca/tsp/concorde.html

13 Cook, W. In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation, Princeton University Press, 2014

14 Cook, W. In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation, Princeton University Press, 2014

15 https://www.math.uwaterloo.ca/tsp/history/index.html

16 https://www.math.uwaterloo.ca/tsp/history/biblio/1930.html

17 https://www.math.uwaterloo.ca/tsp/history/biblio/1930.html

18 https://www.math.uwaterloo.ca/tsp/history/biblio/1940.html

19 G. Dantzig, R. Fulkerson, and S. Johnson, 1954, "Solution of a large-scale traveling-salesman problem", en *Operations Research*

20 Dantzig George
https://mbriournal.com/wp-content/uploads/2020/11/29_George-Dantzig-The-Pioneer-of-Linear-Optimization.pdfhttps://metnumun.wordpress.com/2-history-of-scientific-computing-numerical-algorithms-numerical-methods/

21 B.Wilkinson, M. Allen, Parallel Programming: techniques and applications using networked workstations and parallel computers" Pearson, 2005

Karl Menger - Nearest Neighbour - 1930

Nearest Neighbour (NN) es una heurística, es decir es un algoritmo que no devuelve soluciones exactas a TSP, sino que se aproxima a una solución óptima para algunos tipos de problemas. Para NN, se elige para cada ciudad su vecino más cercano no visitado y se elige ese como siguiente hasta volver al nodo de inicio. Para NN se debe transformar el grafo a un grafo completo.

Un breve resumen de la heurística Nearest Neighbour:

1. Empezar en un nodo, señalar como el nodo actual.
2. Desde el nodo actual, ir por por el vínculo de menor costo que lo lleve a un nodo no visitado - su vecino más cercano.
3. Marcar el nodo actual como visitado.
4. Si quedan nodos sin visitar, vuelve al paso 2.
5. Sino, termina.

Esta solución, aunque mejora el tiempo, generalmente no brinda el camino más corto. El conjunto resultante de vértices es una solución aproximada a TSP. Nearest Neighbour es un ejemplo de un algoritmo Greedy - en cada paso, elige el vértice más corto. No se toman en consideración las consecuencias posteriores de esa decisión. Es posible que elegir el vértice más corto ahora genere la necesidad de elegir uno mucho más largo en el futuro, por lo que hay instancias en el cual Nearest Neighbour tenga resultados muy ineficientes.[22]

G. Dantzig, R. Fulkerson, and S. Johnson, Cutting Plane Method - 1954

En esta solución para 42-49 ciudades[23], la realización de la solución fue instrumentando métodos de Linear Programming. En Linear Programming (de ahora en más LP), se intenta buscar la mejor resolución de un modelo matemático dadas ciertas restricciones.

En esta solución se comienza relajando la definición de recorrido para permitir encontrar fracciones de caminos en una solución intermedia, luego remediando los errores usando métodos de chequeo y eliminación de ciclos hasta subsecuentemente encontrar la solución óptima. En cada iteración subsecuente, se corrigen errores hasta arribar a una solución óptima. Se parte del algoritmo Simplex en LP desarrollado por Dantzig en 1947.

Ellos formularon el problema como un conjunto de conexiones. En vez de pensar un camino como un itinerario de nodos en un orden particular, se piensa el camino como las conexiones entre los nodos en cuestión. Se formula la pregunta, si entre cada nodo hay una conexión potencial, entonces, se hace la pregunta 'la conexión existe o no?'. La respuesta se puede representar como un valor 0 (no) o 1 (si), para cada conexión. Entonces el camino total se puede representar como la suma de todas las conexiones.

Para pensar el problema de modo matemático,se puede pensar un camino fraccional con características tales que se puede representar un camino entre un nodo i, y un nodo j como x(i,j)

- Para cada par de ciudades hay un vínculo de tipo x(i,j). Se asume que x(i,j) es igual a x(j,i), por lo tanto, solo necesita representarse un camino por cada par de ciudades.
- El total de caminos en un grafo n, es de n(n-1)/2. Por ejemplo para 6 ciudades hay 15 vínculos de este tipo.
- Cada camino se puede representar por (x = 0) si no existe el camino entre dos ciudades, ó (x = 1) si existe el camino
- Para cada x(ij) (con 1 o 0 que representa si está en el camino o no) hay una distancia c(i,j) de valor variable mayor que 0 y menor o igual a 1.
- Para tener un camino, se multiplica cada correspondiente x y c de tal manera x(ij) * c(ij) y luego se suman todos los términos. Sum(x(ij) * c(ij)) con i=1 y j=1 hasta N + 1 (número de ciudades contando dos veces la de partida)
- Se busca encontrar: Min(Sum(x(ij)*c(i,j))

Para mejorar el modelo toman en cuenta las restricciones que fueron sugeridas por Julia Robinson en 1949[24] de usar valores fraccionarios en la resolución intermedia. Ella sugiere que los valores de x en lugar de contener solo 1 o 0, deben contener un valor fraccionario de 0 ó más y requiere que el valor total de la suma de los valores de los enlaces vinculados a una ciudad deben ser un máximo de dos. Esto representa que se sale y entra a una ciudad al menos una vez (1 + 1), y que se termina en la misma ciudad que se empieza (1 solo ciclo comprende la totalidad de los nodos + 1). Puede haber infinitos enlaces posibles a cada nodo, de los cuales se deben elegir conexiones tal que deben quedar solo dos para cada nodo en el recorrido final.

Al usar valores fraccionarios, y tener el requerimiento de una suma de 2 para cada nodo, se genera que en una primera instancia pueden haber varias conexiones desde un solo nodo. Esto requiere correcciones posteriores para eliminar las conexiones innecesarias. Luego, usando lo que ellos denominan restricciones para eliminación de ciclos, se logran eliminar todas las conexiones donde se crean ciclos intermedios, y se agregan nuevas conexiones que unifican a estos.

A partir de corregir usando las restricciones, y reiterando el algoritmo hasta encontrar un camino conexo, se puede verificar una solución exacta a TSP usando dualidad LP.

1. Encontrar error: identificar una desigualdad TSP que se viola por la fracción del camino
2. Corregir error: agregar la desigualdad violada como límite en la fracción de camino LP y resolver el nuevo LP con el algoritmo Simplex
3. Verificar solución: chequear si existe un camino TSP como solución óptima a la fracción de camino LP, verificando optimalidad usando dualidad LP.

La mayoría de las soluciones de TSP desde entonces, han utilizado optimizaciones sobre esta solución, específicamente en la parte donde se computan y eliminan los ciclos.

---

22 G.Gutin and A.Yeo. The Greedy Algorithm for the Symmetric TSP, *Algorithmic Oper. Res.*, Vol.2, 2007, pp.33--36.

23 G. Dantzig, R. Fulkerson, and S. Johnson, 1954, "Solution of a large-scale traveling-salesman problem", en *Operations Research*

24 Cook W Computing in Combinatorial Optimization. In: Steffen B., Woeginger G. (eds) Computing and Software Science. Lecture Notes in Computer Science, vol 10000. Springer, Cham. 2019 https://doi.org/10.1007/978-3-319-91908-9_3

Algoritmos de búsqueda en árbol

El uso de búsqueda en árboles es intuitivo para este tipo de problemas y muchos algoritmos tanto exactos como aproximados tipo heurística usan una variación de búsqueda en árbol. La generalización es usar acotamientos en base a búsquedas en árboles mínimos de expansión, lo cual permite reducir las permutaciones en el campo de búsqueda, y por lo tanto el tiempo de ejecución. Abajo se enumeran algunos de los algoritmos más famosos.

Bellman-Held–Karp algorithm - 1962

Este es un algoritmo que encuentra la solución exacta, y es mejor conocido como simplemente Held-Karp. Es una versión del algoritmo de Branch and Bound (búsqueda en un árbol), está basado en la programación dinámica, y tiene un tiempo exponencial $T(n) = O(2^n n^2)$. Siendo un algoritmo exacto, no es apto para Ns muy grandes. Requiere un mayor espacio para poder resolverse - espacio $S(n) = O(2^n \sqrt{n})$, lo cual se debe tener en cuenta si se tienen recursos limitados.[25] [26] [27]
Held-Karp es una relajación Lagraniana de TSP[28]. Held-Karp agregan restricciones especiales a Branch and Bound, se utiliza un 1-tree y el uso de programación dinámica para calcular valores para la acotación de caminos. Una descripción paso por paso del algoritmo se puede encontrar en la solución que sigue.

Lin-Kernighan / k-opt - 1973

El Lin-Kernighan algorithm es una de las soluciones heurísticas con mejores resultados para TSP simétrico hasta hoy, logrando soluciones óptimas para valores altos de N, y con complejidad de tiempo de $O(n^{2.2})$. Su implementación utiliza árboles mínimos de expansión, intercambiando vértices o subárboles utilizando un algoritmo greedy. La implementación no es trivial, por lo cual no vamos a profundizar en el algoritmo en este estudio. Sin embargo, los detalles completos se pueden encontrar en la publicación.[29]Existen varias publicaciones más recientes que amplifican el estudio de este algoritmo.[30]

Christofides - 1976

Otro algoritmo notorio con árbol de búsqueda es Christofides, de 1976[31]. Es un algoritmo que encuentra caminos óptimos aproximados (es decir, un algoritmo de tipo heurístico), en instancias donde las distancias forman un espacio métrico (son simétricos y obedecen la desigualdad triangular). Es un algoritmo de aproximación que garantiza que la solución estará dentro de un factor de 3/2 de la solución óptima. Desde 2017, se ha comprobado como la mejor relación de aproximación para TSP en espacios métricos generales, aunque mejores aproximaciones han sido encontradas para algunos casos especiales.

Pasos básicos:

1. Encontrar un árbol mínimo de expansión (T)
2. Encontrar los vértices de T con un ángulo impar (O)
3. Encontrar vértices de pesos mínimos (M) vinculados a T
4. Generar un circuito Euleriano utilizando los vértices de M y T
5. Generar un ciclo Hamiltoniano salteando los vértices repetidos.

Metaheurísticas: enfoque evolutivo

A diferencia de los enfoques deterministas vistos en las secciones anteriores, en las últimas décadas ha surgido un campo basado en enfoques evolutivos que se denominan 'meta-heurísticas'.[32] Uno de los primeros es el trabajo seminal Ant Colony Optimization de Dorigo and Gambardella[33] en 1997. Hay varios estudios de este tipo, y abundan paralelizaciones para este tipo de estudios particularmente con el uso de tecnologías multi-core y tarjetas gráficas.[34]

Siguiente etapa: paralelización de búsqueda en un árbol

En el próximo capítulo, veremos un algoritmo muy simple para TSP basado en búsqueda en un árbol. Describiremos un algoritmo llamado Branch and Bound usando Depth First Search. Luego, en capítulo 5, veremos una posible paralelización del algoritmo Branch and Bound implementada para MPI.

---

25 Held, M. and Karp. R.M. The traveling-salesman problem and minimum spanning trees. Op.Res.18, 1970, pp.1138-1162.

26 Held, M. and Karp. R.M. , A Dynamic Programming Approach to Sequencing Problems, Journal of the Society for Industrial and Applied Mathematics, Vol. 10, No. 1 (Mar.,1962), pp. 196-210

27 C.L.Valenzuela and A.J.Jones. Estimating the Held-Karp lower bound for the geometric TSP. *European J. Op. Res.*, 102:1, 1997, pp.157-175

28 John E. Mitchell, The Held & Karp Relaxation of TSP, 2019

29 Lin, Shen; Kernighan, B. W. (1973). "An Effective Heuristic Algorithm for the Traveling-Salesman Problem". Operations Research

30
K. Helsgaun. General k-opt submoves for the Lin-Kernighan TSP heuristic. *Mathematical Programming Computation*, 2009.
K. Helsgaun. An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic, DATALOGISKE SKRIFTER (Writings on Computer Science), No. 81, 1998, Roskilde University.
Keld Helsgaun "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic", European Journal of Operational Research. 126, 106–130 2000
http://akira.ruc.dk/~keld/research/LKH/LKH-1.3/DOC/LKH_REPORT.pdf
Johnson, David S.; McGeoch, Lyle A. (1997). "The Traveling Salesman Problem: A Case Study in Local Optimization" (PDF). In E. H. L. Aarts; J. K. Lenstra (eds.). Local Search in Combinatorial Optimization. London: John Wiley and Sons. pp. 215–310
B.Chandra, H.Karloff and C.Tovey. New results on the old k-opt algorithm for the TSP. *5th ACM-SIAM Symp. on Discrete Algorithms*, 1994, pp.150-159.

31 N.Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Report No. 388, GSIA, Carnegie-Mellon University, Pittsburgh, PA, 1976.
https://apps.dtic.mil/dtic/tr/fulltext/u2/a025602.pdf

32 www.metaheuristics.org, www.aco-metaheuristic.org

33 Dorigo M, Gambardella L. Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation. April 1997; 1(1):53-66. DOI: 10.1109/4235.585892

34 Uchida A., Ito, Y., Nakano, K. An Efficient GPU Implementation of Ant Colony, Optimization for the Traveling Salesman Problem, 2012 Third International Conference on Networking and Computing
https://www.cs.hiroshima-u.ac.jp/cs/_media/4893a094.pdf
You, Y-S. Parallel Ant System for Traveling Salesman Problem on GPUs Taiwan Evolutionary Intelligence Laboratory (TEIL)http://www.gpgpgpu.com/gecco2009/4.pdf

Capítulo 3: Descripción del algoritmo Branch and Bound

Consideremos que el conjunto de ciudades y rutas que debe recorrer el vendedor ambulante de TSP se representan con un grafo dirigido y pesado. Un grafo es un conjunto de vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto. En la teoría de grafos, un grafo pesado es un grafo cuyos vértices o aristas tienen nombres o etiquetas, en este caso, las aristas tienen costos asociados. En un grafo dirigido o digrafo, las aristas tienen una orientación; si hay una arista (A, B), A es el nodo inicial y B es el nodo final de la arista. Los vértices corresponden a las ciudades, las aristas a las rutas entre ciudades, y las etiquetas de las aristas corresponden a los costos de las rutas.

El método Branch and Bound (ramificación y acotamiento) consiste de un recorrido sobre una estructura de árbol de soluciones que utiliza una búsqueda basada en un estado óptimo, de tal manera que se enumeran los candidatos exitosos de la búsqueda y se descartan (no se exploran) los que no lo son. La decisión sobre la selección o descarte se decide usando límites que se actualizan durante el recorrido. El algoritmo Branch and Bound se suele usar como método de optimización global para problemas discretos.
Existen variaciones de este algoritmo, con mayor y menor complejidad. Veremos una versión simple que utiliza depth-first-search, o búsqueda en profundidad para resolver TSP.

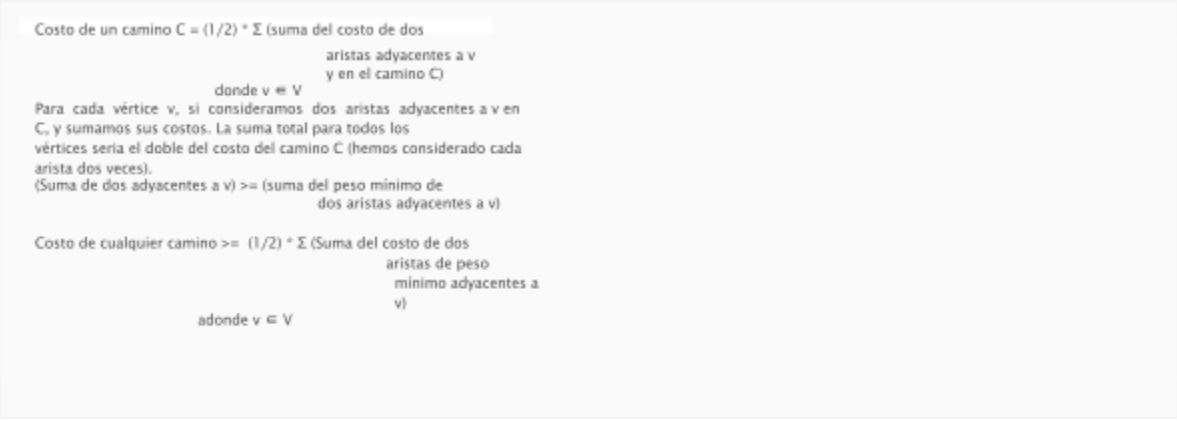
Primero se plantea el grafo expandido como un árbol (inicialmente se imagina a todos los nodos potencialmente conectados entre todos ellos). Desde éste se expande a partir del nodo "ciudad de partida" como la raíz, y los caminos son los subárboles hijos del nodo raíz hasta las hojas. Las hojas enumeran el conjunto de posibles soluciones totales o caminos, y todos los otros nodos corresponden a caminos parciales - rutas que han visitado algunas, pero no todas las ciudades. Cada nodo del árbol tiene un costo asociado, el costo del camino parcial. Se tiene registro de un costo mínimo actual durante la búsqueda. En este método, se comienza desde la raíz del árbol, o el nodo actual, en profundidad por el subárbol (izquierdo si se quiere) hasta la primera hoja, y de aquí se obtiene el primer costo del mejor camino. Si durante el resto de la búsqueda por los subsecuentes subárboles nos encontramos con que el costo del camino parcial desde la raíz a ese nodo excede la mejor solución actual establecida como acotamiento, entonces podemos ignorar el subárbol de este nodo.

> The set of all tours (feasible solutions) is broken up into increasingly small subsets by a procedure called branching. For each subset a lower bound on the length of the tours therein is calculated. Eventually, a subset is found which contains a single tour whose length is less than or equal to some lower bound for every tour.[35]

La **complejidad temporal** en el peor caso implica recorrer todos los posibles subárboles (el conjunto completo de soluciones posibles), O(n!), por lo tanto no podemos decir que implica una mejora en estos términos. Sin embargo, en términos prácticos, existen ocasiones en las cuales se percibe una mejora en el tiempo de ejecución usando Branch and Bound por sobre la solución usando permutaciones (brute force), ya que en muchos casos se pueden eliminar caminos posibles que superan el límite establecido durante la búsqueda.

> The process stops when a subset is found which contains a single solution whose value is less than or equal to the lower bounds for all the terminal nodes. (The lower bound used for a single solution is assumed to be the value of the
> solution itself.) Since the terminal nodes collected together always contain all solutions, a minimal solution has been identified.[36]

En pseudocódigo podemos pensar el problema de la siguiente manera. Consideramos el acotamiento como el cálculo de un costo mínimo, que se calcula de costos acumulados. El costo de atravesar un nodo está compuesto de dos costos
1) El costo de alcanzar el nodo desde la raíz (que conocemos al llegar al nodo)
2) El costo de un camino desde el nodo actual a una hoja (se calcula un acotamiento sobre este costo para decidir si ignorar el subárbol con el nodo o seguir por el
3) Lo más complejo es encontrar una forma de calcular el acotamiento sobre la mejor solución posible.



3.1 Algoritmo Iterativo de Branch and Bound

¿Cómo pensar esto en código? Se toma como referencia la búsqueda en profundidad en un árbol de soluciones, detallada por Peter Pacheco en "An Introduction to Parallel Programming".[37] La implementación sería sugerida por Pacheco es inicialmente una búsqueda en profundidad en el árbol utilizando recursión. Al trabajar con grafos o con árboles, el método más simple e intuitivo de la búsqueda en profundidad, es comúnmente implementado usando recursión. La recursión significa que para cada nodo en el árbol, vamos a buscar en sus subárboles hasta llegar a una hoja. Tenemos dos casos bases para retornar de la función - llegar a una hoja, o exceder el mínimo actual. Esta implementación necesita

---

35 J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel, "An algorithm for the traveling salesman problem", *Operations Research* **11,** 972-989. 1963
https://dspace.mit.edu/bitstream/handle/1721.1/46828/algorithmfortrav00litt.pdf

36 J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel, "An algorithm for the traveling salesman problem", *Operations Research* **11,** 972-989. 1963
https://dspace.mit.edu/bitstream/handle/1721.1/46828/algorithmfortrav00litt.pdf

37 Pacheco, P.S. An Introduction to Parallel Programming. 2011

de una estructura para almacenar el grafo dirigido o digrafo. Se puede pensar como una estructura que tiene el número de ciudades N, y un arreglo que almacena la matriz de adyacencias (en C se suele usar un arreglo como manera más eficiente de almacenar las matrices, usando índices de desplazamiento para direccionar las posiciones de la matriz).También vamos a necesitar representar la ciudad de partida y el mejor camino encontrado.

Solución 1: recursiva

Se llama función recursiva de "búsqueda en profundidad", a una función en la cual primero se chequea si se llegó a una hoja - si se llega a N (número de ciudades); luego se chequea si es el mejor camino (el más corto). Si fuera el mejor camino, se actualiza la variable mejor camino y se guarda el camino actual como mejor camino. Este camino se inicializa previo a la llamada de la función con un número mayor a cualquier costo de camino posible. Si no se llegó a una hoja, se puede continuar buscando en profundidad en el árbol "expandiendo el nodo actual", en otras palabras, intentando visitar otras ciudades desde la última ciudad visitada en el camino parcial. Para hacer esto, simplemente iteramos por todas las ciudades. Para cada ciudad, se chequea si la ciudad o vértice ya fue visitada; y, si no lo fué, si el costo es menor al mínimo costo de camino actual. Si la ciudad es viable, se la agrega al camino, y se llama recursivamente a "búsqueda en profundidad". Cuando se retorna de "búsqueda en profundidad", se remueve la ciudad del camino, ya que no debería ser incluida en el camino usado en las llamadas recursivas subsecuentes.

Ya tenemos un algoritmo de búsqueda en profundidad en el árbol que nos devuelve el camino mínimo. Este algoritmo funciona correctamente. Pero este tipo de funciones recursivas dificultan la paralelización. Para poder pensar la paralelización, necesitamos poder encontrar un modelo apto de forma tal que se pueda distribuir el trabajo y mapear el problema a los diferentes procesos. La recursión en este caso tiene la desventaja de que en cualquier instante del tiempo solo el nodo actual es accesible. Esto podría presentar problemas al momento de paralelizar la búsqueda en el árbol y querer dividir los nodos del árbol entre procesos.

Peter Pacheco describe dos soluciones en las cuales se replica un stack con una estructura de datos tipo arreglo, que reemplaza la pila de llamadas del sistema. De esta manera es posible pensar en manipular y dividir el problema en la paralelización. La idea básica de los algoritmos está modelada sobre la implementación recursiva. Las llamadas recursivas empujan el estado actual de la función recursiva en la pila de ejecución. Entonces, se puede simular la recursión con una estructura de control *while* y una estructura de datos que reemplaza la pila. Se empujan los datos contextuales de un nodo a un stack propio antes de buscar más profundo en el árbol. Al volver para arriba en el árbol - sea por que se llega a una hoja o a un nodo demasiado costoso - se saca de nuevo de la pila.

Hay un poco de diferencia entre las dos versiones descritas por Pacheco que utilizan una estructura de datos tipo lista para replicar la pila.

```
1  void Depth_first_search(tour_t tour) {
2     city_t city;
3
4     if (City_count(tour) == n) {
5        if (Best_tour(tour))
6           Update_best_tour(tour);
7     } else {
8        for each neighboring city
9           if (Feasible(tour, city)) {
10             Add_city(tour, city);
11             Depth_first_search(tour);
12             Remove_last_city(tour, city);
13          }
14    }
15 }  /* Depth_first_search */
```

Solución 2: primer versión iterativa

Al primer caso, lo llamaremos solución iterativa 1. En esta versión, un registro de la pila consiste de una sola ciudad. Inicialmente, con una estructura de control tipo *for*, se empujan todas las ciudades del dígrafo a la estructura stack (en orden reverso n -1 a 1 , ya que luego se sacan de la pila de manera descendente. Esto asegura que las ciudades son visitadas en el mismo orden que la función recursiva).

Luego, la estructura de control principal del algoritmo es un *while* que no se corta mientras la pila no esté vacío. Se saca una ciudad de stack. Se chequea que la ciudad sacada no sea la constante "NO_CITY" que representa haber llegado al final de la lista de hijos de un nodo, en cual caso se quita la última ciudad del camino actual para subir un nivel. Si no es "NO_CITY", se agrega nuestra ciudad al camino actual.

Seguidamente , se chequea si se ha llegado a una hoja preguntando si la cantidad de ciudades en el camino es igual a N, y si se ha llegado, se chequea si el costo es mejor que el costo mejor actual y si lo es, se actualiza el costo menor actual, de la misma forma que lo hacía la versión recursiva. También se debe sacar la última ciudad del camino para poder volver hacía arriba en el árbol.

Luego se empuja "NO_CITY" antes de empujar todos los hijos del nodo actual al stack. Entonces, se desarrolla la búsqueda en profundidad desde la ciudad actual, bajando por el árbol, preguntando si los hijos pueden producir caminos viables.

En esta versión hay un trabajo más cercano y explícito al stack, y es una versión más eficiente en su uso de memoria, aunque sea más difícil de replicar en versión paralela. La razón que es más difícil de paralelizar, es que al solo empujar ciudades al stack, no hay información contextual para que cada proceso pueda trabajar de manera independiente con diferentes partes del árbol. Se pierde toda la información del camino parcial.

```
1  for (city = n-1; city >= 1; city--)
2     Push(stack, city);
3  while (!Empty(stack)) {
4     city = Pop(stack);
5     if (city == NO_CITY) // End of child list, back up
6        Remove_last_city(curr_tour);
7     else {
8        Add_city(curr_tour, city);
9        if (City_count(curr_tour) == n) {
10          if (Best_tour(curr_tour))
11             Update_best_tour(curr_tour);
12          Remove_last_city(curr_tour);
13       } else {
14          Push(stack, NO_CITY);
15          for (nbr = n-1; nbr >= 1; nbr--)
16             if (Feasible(curr_tour, nbr))
17                Push(stack, nbr);
18       }
19    }  /* if Feasible */
20 }  /* while !Empty */
```

Solución 3: segunda versión iterativa

La tercera versión estudiada por Peter Pacheco[38] de este algoritmo, a la que llamaremos versión iterativa 2, nos permite paralelizar TSP más fácilmente de manera distribuida usando MPI[39] con el paradigma SPMD[40], utilizando búsqueda en un árbol de soluciones con Branch and Bound.

En esta versión hacemos una búsqueda similar que las anteriores, pero en vez de solo guardar ciudades en la pila, se guardan subárboles enteros. De esta manera conservamos la información contextual de cada camino parcial construido (en este sentido, es más similar a la versión recursiva).

Esta versión es más lenta, ya que requiere tiempo de alocación para los caminos parciales. Este tiempo se puede ahorrar en parte reutilizando el espacio de los caminos abandonados.

Lo que nos permite esta versión es poder dividir el digrafo en subárboles, y distribuir un subárbol por proceso, y que cada proceso pueda encargarse de manera independiente de buscar en su subárbol. En la sección siguiente, analizaremos cómo se puede paralelizar este algoritmo.

```
1  Push_copy(stack, tour);  // Tour that visits only the hometown
2  while (!Empty(stack)) {
3     curr_tour = Pop(stack);
4     if (City_count(curr_tour) == n) {
5        if (Best_tour(curr_tour))
6           Update_best_tour(curr_tour);
7     } else {
8        for (nbr = n-1; nbr >= 1; nbr--)
9           if (Feasible(curr_tour, nbr)) {
10             Add_city(curr_tour, nbr);
11             Push_copy(stack, curr_tour);
12             Remove_last_city(curr_tour);
13          }
14    }
15    Free_tour(curr_tour);
16 }
```

Pruebas sobre algoritmos seriales

Luego de ejecutar algunas pruebas, podemos ver que el algoritmo recursivo es aproximadamente 8% más lento que la primera versión iterativa. Esta primera versión iterativa es aproximadamente 5% más rápida que la segunda versión iterativa. Como es la más rápida, nos sirve para ejecutar las pruebas de comparación con las versiones paralelas. La tercera versión será la más lenta, pero nos sirve para paralelizar este problema con el paradigma SPMD, gracias a su uso de pilas de caminos separadas para cada proceso.

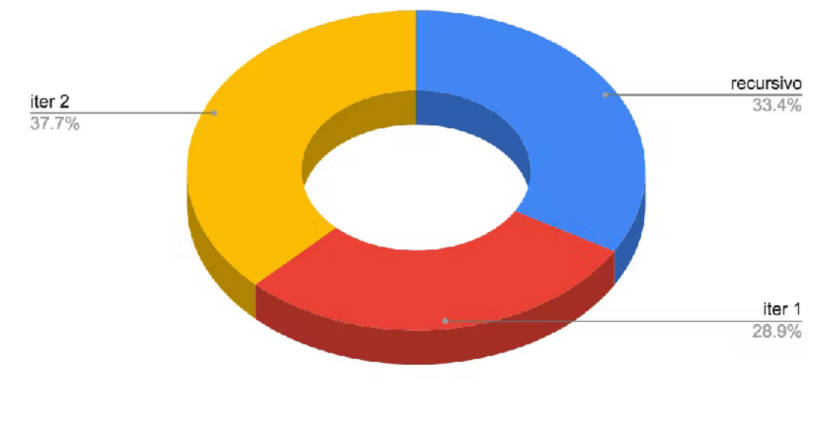
Tiempos de ejecución sobre versiones seriales

Algoritmos Iterativos	recursivo	iter 1	iter 2
n = 8	0.002761126	0.002388	0.003111124
n = 10	0.2269852	0.192179	0.2497609
n = 12	4.471921	3.948113	4.763838

38 P. Pacheco, P.S. An Introduction to Parallel Programming, 2011

39 Padua, D., Ghoting, A., Gunnels, J. A., et al. *MPI (Message Passing Interface). Encyclopedia of Parallel Computing, 1184–1190, 2011*

40 Dongarra, J.,et al (2011). SPMD Computational Model. Encyclopedia of Parallel Computing, 1933–1943. doi:10.1007/978-0-387-09766-4_26



3.2 Paralelizar Branch and Bound

Consideramos la metodología de Ian Foster[41] [42], para la paralelización de programas seriales.

1. ***Partitioning*** Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
2. ***Communication*** Determine what communication needs to be carried out among the tasks identified in the previous step.
3. ***Agglomeration or aggregation*** Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. ***Mapping*** Assign the composite tasks identified in the previous step to processes/ threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

Esta metodología se puede aplicar a diferentes modelos generales para paralelizar algoritmos. En nuestro caso tenemos un modelo basado principalmente en master-slave o coordinador y trabajadores (una versión de SPMD con dos tipos de tareas en vez de una) con mapeo estático de tareas, y divide and conquer, ya que cada proceso trabajara sobre su subárbol de manera independiente. Al final del trabajo principal se hace una reducción para encontrar el mínimo. En este modelo de datos paralelos, la descomposición y mapeo se hace sobre los datos, las tareas (luego de dividir entre coordinador y trabajadores) son fijas, los datos se mapean estáticamente, y cada tarea hace la misma operación sobre los datos asignados. Todos los procesos hacen la misma tarea y no hay comunicación entre procesos durante ese trabajo.

El algoritmo de Branch and Bound se basa en la búsqueda en el árbol de soluciones utilizando un costo mínimo actualizable que vimos en la sección anterior. Trabajar con árboles hace bastante accesible la paralelización ya que diferentes procesos pueden buscar diferentes partes del árbol iterando independientemente, por lo cual el programa se hace "embarrassingly parallel". Se divide el árbol de soluciones en subárboles, se recorren los posibles caminos y se comparan sus costos acumulados. Al usar una estructura de tipo árbol tiene sentido asociar tareas con nodos del árbol. La forma mas natural sería alocar una parte del árbol desde un nodo en particular a un proceso. Este proceso busca en su propio área del árbol. Al hacer esto, asumimos que las tareas se comunicaran de arriba para abajo: un padre comunica su camino parcial a un hijo, pero un hijo solo vuelve para arriba en caso de terminar y retornar.

Para la paralelización, surge la necesidad de usar búsqueda en anchura (Breadth First Search) para poder particionar el árbol, abriendo todos los subárboles de un nodo en vez de explorar a profundidad desde un primer paso. Para distribuir equitativamente el árbol entre procesos, conviene particionar el árbol desde un nivel que tenga suficientes nodos para repartir entre los procesos. De esta manera mejoramos la distribución de carga que si intentamos empezar con una búsqueda en profundidad. El proceso coordinador se puede encargar de hacer esta primera división. Hace una búsqueda en anchura, busca lo más amplio posible en el árbol, hasta encontrar un nivel que tiene al menos el número de procesos, y así dividir el nivel en número de procesos usando un **block-cyclic partition**[43] . Si hay más procesos que subárboles, algunos procesos tendrán +1 subárbol. El coordinador envía los subárboles a cada proceso usando una estructura especial para esta tarea. Cada proceso puede tomar la tarea de explorar un subárbol desde el primer nivel donde el número de procesos se aproxime al número de nodos que haya en ese nivel.

Cada proceso, de manera independiente, hace búsqueda en profundidad en su subárbol o subárboles, y para cada nodo compara el costo contra un mínimo local. Se usan estructuras de datos auxiliares para poder almacenar los caminos localmente y poder ir recorriendo los caminos de cada proceso. Se saca el camino parcial de la pila local (inicialmente puede ser de un solo nodo), y se pregunta si llegó al largo de N, si no, se continúa haciendo una búsqueda en profundidad. Para cada nodo, se pregunta si el costo actual es mayor al mínimo local, y si lo es, no se continúa buscando profundidad por ese subárbol. Al llegar a una hoja, con comunicación grupal se pide un costo mínimo global, y si el proceso tiene el camino de menor costo global, manda comunicación grupal para que se actualice el costo. Para mejorar los tiempos, solo se envía el costo, y no el camino. Un proceso que ya no tiene más trabajo que hacer, espera ocioso en una barrera, hasta que hayan terminado todos los procesos para poder finalizar. Cuando todos los procesos han terminado de buscar, se hace una reducción global para poder encontrar el camino con el menor costo. El proceso que tiene el menor costo puede enviar su camino y su ID al proceso 0, que es el proceso master o coordinador (si es el coordinador, no necesita enviarlo). El proceso coordinador tiene control sobre I/O para informar el resultado.

Veremos con más detalle la implementación con MPI en el capítulo 5.

---

[41] P. Pacheco, P.S. An Introduction to Parallel Programming, 2011.p66

[42] I. Foster, Designing and Building Parallel Programs, Addison-Wesley, Reading, MA, 1995.

[43] P. Pacheco, P.S. An Introduction to Parallel Programming, 2011.p110

Algunos desafíos:

Todos los procesos no tienen la misma carga de trabajo, y algunos pueden estar ejecutando mucho tiempo, mientras otros terminan de trabajar y no toman más trabajo. Cuando no se le puede asignar la misma carga de trabajo a cada proceso, tenemos un problema de balance de carga o "load balancing". El balance de carga es un problema que surge debido al uso de una estructura irregular. Se debe tomar en cuenta a la par de la necesidad de minimizar el overhead de comunicación entre procesos que se hace necesaria en un mapeo dinamico. A pesar de asignar un subárbol de igual tamaño a cada proceso, puede que un proceso no tenga trabajo si el costo actual supera el costo mejor actual de ese subárbol.

> "El objetivo primario del cómputo paralelo es reducir el tiempo de ejecución haciendo uso eficiente de los recursos. El balance de carga es un aspecto central y consiste en, dado un conjunto de tareas que comprenden un algoritmo y un conjunto de procesadores, encontrar el mapeo (asignación) de tareas a procesadores tal que cada una tenga una cantidad de trabajo que demande aproximadamente el mismo tiempo. Esto es más complejo si los procesadores (y comunicaciones) son heterogéneos."[44]

En la implementación que vimos se utiliza static partitioning (se la asigna una sección igual del árbol a cada proceso, sin re-asignar trabajo de los que tienen mucho a los que se quedan sin trabajo) y presenta un problema de load balancing que se puede resolver ajustando el programa para utilizar dynamic scheduling. En la solución de dynamic scheduling, cuando un proceso se queda sin trabajo se le intenta asignar trabajo para aliviar la carga de otros procesos. Esto último presenta sus propios problemas de sincronización que hacen que la implementación se haga más compleja, y tenga mayor overhead de comunicaciones. Como no tenemos forma de saber cuáles procesos tendrán más trabajo que otros, el proceso de repartirse la carga implica que se necesitan más comunicaciones grupales, y mayor tiempo de sincronización entre procesos para determinar el reparto.

Por estas razones, TSP en general no es simple de implementar de manera paralela. Obviando las diferencias en lo que son arquitecturas o posibles herramientas dentro de lo que es el campo de la concurrencia, el problema en sí es difícil de dividir de manera pareja. Esto es porque no se cuenta con una estructura regular, siendo el centro del problema conocer el mejor recorrido de un grafo, cuyo árbol de expansión tiene límites desconocidos, y en este problema en particular, cuyos pesos de adyacencia son irregulares. [45]

Capítulo 4: librería MPI para C

MPI[46] es una especificación para la implementación de paralelización usando pasaje de mensajes. El API de MPI facilita el uso de primitivas para la comunicación entre procesos. Fue diseñado para el uso de procesamiento paralelo usando pasaje de mensajes sobre memoria distribuida. Las implementaciones vigentes tienen soporte también para memoria compartida. Hay varias implementaciones de la especificación como lo son MPICH y openMPI. *En este informe se utiliza OpenMPI 4.0.2 que es la implementación de MPI instalada en el cluster de la Universidad Nacional de La Plata, que utilizaremos para correr las pruebas de TSP con MPI.*[47]

En MPI, no se automatiza la paralelización implícitamente, sino que se debe implementar de manera explícita por el programador. La forma habitual de implementar la paralelización, es comenzar con un algoritmo secuencial en el cual se identifican secciones paralelizables, luego se hace re-ingeniería para paralelizar dichas secciones utilizando MPI.

Siendo MPI un paradigma SPMD (Single Program Multiple Data)[48]se puede utilizar una estrategia de "divide and conquer" donde cada proceso ejecuta el mismo programa pero se divide entre ellos el problema. A partir del algoritmo secuencial, se particionan los datos en partes iguales y se dividen entre los procesos. Cada uno toma una pequeña parte para mejorar la eficiencia de procesamiento (aunque cada procesos levanta la misma aplicación, cuenta con su propia sección de memoria para procesar su sección de datos).

Una vez desarrollada la reingeniería del programa, esté siempre corre de manera serial en su principio, y en el momento de querer inicializar la paralelización, se debe explicitar la inicialización con MPI_Init(). Luego se utilizan los primitivos de pasaje de mensaje para trabajar los datos en paralelo. Luego del procesamiento paralelo, se debe indicar el final de la sección con MPI_Finalize(). Luego corre de manera serial hasta finalizar el programa.

Se debe tener en cuenta al momento de programar que conviene procesar I/O de un proceso a la vez para evitar cuellos de botella. Se previene cuello de botella de I/O (además de otros problema relacionados con sincronización) eligiendo un solo proceso para este tipo de operaciones. Citamos[49]:

> Recall that different parallel systems vary widely in their I/O capabilities, and with the very basic I/O that is commonly available it is very difficult to obtain high performance. This basic I/O was designed for use by single-process, single-threaded programs, and when multiple processes or multiple threads attempt to access the I/O buffers, the system makes no attempt to schedule their access.
> [..] we generally assume that one process/thread does all the I/O, and when we're timing program execution, we'll use the option to only print output for the final timestep.

En el caso del algoritmo implementado, solo el procesos 0 lee el archivo que almacena en una matriz los vértices del digrafo para poder comunicarlo a otros procesos. El desarrollo del algoritmo se verá en detalle en el próximo capítulo

---

44 Naiouf,M. et al.Fundamentos de cómputo paralelo y distribuido de altas prestaciones. Construcción y evaluación de aplicaciones. 2014, p2

45 Monteiro, M. J. T. P./ Parallelizing Irregular Algorithms: a Pattern Language.Conference on Pattern Languages of Programs. 2011. pp. 4:1-4:17 https://www.hillside.net/plop/2011/papers/A-24-Monteiro.pdf

46 Padua, D., Ghoting, A., Gunnels, J. A., et al. *MPI (Message Passing Interface). Encyclopedia of Parallel Computing, 1184–1190. 2011* doi:10.1007/978-0-387-09766-4_222

47 Documentación OpenMPI https://ccportal.ims.ac.jp/en/node/2635

48 Dongarra, J.,et al (2011). SPMD Computational Model. Encyclopedia of Parallel Computing, 1933–1943. doi:10.1007/978-0-387-09766-4_26

49 Pacheco, P.. An Introduction to Parallel Programming. 2011. p280

Capítulo 5: Implementación algoritmo Branch and Bound paralelizado con MPI

El código trabajado en este capítulo se basa en la implementación que se describe en el libro de Peter Pachco "An Introduction to Parallel Programming, en el capítulo 6. Tiene 3 secciones principales - el inicio, donde se incluye la librería, se definen los tipos de datos, se declaran las variables globales y se inicializa la paralelización con MPI, guardando los datos correspondientes a la comunicación entre procesos. La sección principal, que es el procesamiento de datos con MPI. Al final, se hace la finalización de la sección MPI, liberación de memoria, el informe al usuario, etc

Inicialización

Se inicializa la paralelización con la primitiva MPI_Init(&argc, &argv). Los argumentos recibidos de la línea de comandos, argc y argv representan respectivamente el número de argumentos y el vector de argumentos. Luego se llaman las primitivas que nos permiten tener para cada proceso su ID (mi_id_proceso), y el total de procesos (num_de_procesos).

```
MPI_Init(&argc, &argv);
universo = MPI_COMM_WORLD;
MPI_Comm_size(universo, &num_de_procesos);
MPI_Comm_rank(universo, &mi_id_proceso);
```

MPI_COMM_WORLD es el comunicador predefinido por MPI_Init(), guardado en 'universo'. El comunicador es una colección de procesos que se pueden comunicar con mensajes. Se pueden definir nuevos comunicadores que definen un universo de procesos que se pueden comunicar entre sí. A partir del comunicador podemos conocer el ID o rango de los procesos y el tamaño de la colección. El comunicador siempre se debe especificar en las primitivas de comunicación. En este caso solamente usaremos el comunicador predefinido MPI_COMM_WORLD que contempla todos los procesos que usaremos.

Para iniciar a trabajar con el dígrafo, se debe leer desde un archivo de entrada. Los archivos de los dígrafos son archivos de texto plano que contienen el dato N como primer dato, y luego una matriz de adyacencias. Cada posición Xij de la matriz (i y j son índices de 0 a n-1) corresponde a la distancia entre el nodo i y el nodo j en la "ciudad". Las distancias Xij no son iguales a las distancias Xji. En la diagonal principal, donde i = = j, el dato no se usa, por lo cual se guarda como 0. Como vamos a usar al proceso 0 como coordinador, y le vamos a otorgar acceso único a IO, proceso 0 se encargará de leer los datos del archivo y repartirlos para el resto de los procesos en el universo.

El proceso 0 abre el archivo que contiene la matriz de adyacencias. Este fue pasado como argumento al programa en modo lectura y está en argv[1].

```
if (mi_id_proceso == 0) {
   archivo_de_matriz = fopen(argv[1], "r");
}
```

El proceso 0 también es el que lee el tamaño de N, el número de ciudades. Luego envía este dato, indicando que es integer con MPI_INT en &n, con un broadcast usando la funcion MPI_Bcast() (bloqueante) a todos los procesos. Todos los procesos llaman la función MPI_Bcast(), y solo se continúa ejecutando el programa una vez que todos los procesos hicieron el llamado a la función.

```
if (mi_id_proceso == 0) fscanf(archivo_de_matriz, "%d", &n);
MPI_Bcast(&n, 1, MPI_INT, 0, universo)
```

Todos los procesos alocan memoria para el dígrafo, un espacio de tamaño n*n del tipo dato costo, costo_t (es un int que representa el costo de una ruta entre dos nodos). Luego, proceso 0 usa dos estructuras de *for* anidadas para direccionarse en la matriz de adyacencias del archivo del digrafo, y lee cada dato haciendo un desplazamiento usando los índices i y j para acceder a las posiciones.

```
if (mi_id_proceso == 0) {
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
         fscanf(archivo_de_matriz, "%d", &matriz_de_adyacencias[i*n + j]);
       }
}
```

Nuevamente, todos los procesos llaman a la función MPI_Bcast(), adonde el proceso 0 envía al resto de los procesos un tipo de dato compuesto de MPI_INT de tamaño n*n en el puntero matriz_de_adyacencias, previamente alocado. El proceso 0 cierra el archivo de la matriz.

```
MPI_Bcast(matriz_de_adyacencias, n*n, MPI_INT, 0, universo);
if (mi_id_proceso == 0) fclose(archivo_de_matriz);
```

MPI tiene una serie limitada tipos de datos predefinidos que se declaran en el momento de envío y recepción, que no incluyen datos compuestos. Sin embargo, hay varias funciones para declarar nuevos tipos de datos, particularmente para datos compuestos. Luego de alocar e inicializar los structs correspondientes a los tours locales, se necesita declarar un tipo de datos nuevo de tipo arreglo para poder enviar los subárboles a los procesos en el universo. Ya que se trata de un arreglo de datos integer, podemos definirlo con la primitiva MPI_Type_contigous() que es la primitiva más básica para esta funcionalidad. La firma de MPI_Type_contiguous en los docs de OpenMPI es la siguiente:

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
 MPI_Datatype *newtype)
```

Usamos MPI_INT como oldtype, &camino_mpi_t como new type (pasada como puntero), y el count es de n+1 ya que necesitamos contar N mas el nodo que es el punto de retorno. Para definir el nuevo tipo de datos MPI, se declara el nuevo tipo entre las variables globales, se utiliza la primitiva MPI_Type_contigous() para declarar un tipo

contiguo en memoria, y se usa MPI_Type_commit para asentar la definición nueva. Al final del programa, necesariamente liberaremos el nuevo tipo con MPI_Type_free().

```
MPI_Datatype camino_mpi_t;
MPI_Type_contiguous(n+1, MPI_INT, &camino_mpi_t);
MPI_Type_commit(&camino_mpi_t);
```

Finalmente, implementamos una barrera MPI_barrier() previo a la medición del tiempo del algoritmo paralelo, y una reducción con MPI_MAX posterior a la medición, para obtener el tiempo máximo de la búsqueda entre todos los procesos, siguiendo los consejos de Pacheco:[50]

> "[..] timing parallel code is more complex, since ideally we'd like to synchronize the processes at the start of the code, and then report the time it took for the "slowest" process to complete the code. MPI Barrier does a fairly good job of synchronizing the processes. A process that calls it will block until all the processes in the communicator have called it."

```
MPI_Barrier(universo);
comienzo = MPI_Wtime();
// CÓDIGO A MEDIR
fin = MPI_Wtime();
duracion_local = fin - comienzo;
MPI_Reduce(&duracion_local, &duracion, 1, MPI_DOUBLE, MPI_MAX, 0, universo);
```

Partición inicial del árbol

La partición inicial del árbol y la distribución de los subárboles correspondientes a cada proceso lo hace proceso 0. Proceso 0 se hace responsable de hacer una búsqueda en anchura para repartir, entre el número de procesos, los nodos correspondientes a los subárboles o caminos parciales que corresponden a un nivel en el árbol. Se usa una estructura de tipo cola para la búsqueda en anchura y almacenar los caminos parciales. A continuación se describe en detalle este proceso, paso a paso.

El proceso 0 primero necesita saber un límite superior para el tamaño de la cola. Se llama a la función Limite_superior_para_cola(), que encuentra un límite superior para el tamaño de la cola en relación al número de ciudades y de procesos, usando un incremento factorial. Este tamaño se guarda en la variable tam_de_cola. Luego se aloca espacio en la cola para todos los caminos.

```
cola_t Inicializar_cola(int tamanio) {
  cola_t cola_nueva = malloc(sizeof(struct_de_cola));
  cola_nueva->lista = malloc(tamanio*sizeof(camino_t));
  cola_nueva->lista_alloc = tamanio;
  cola_nueva->cabeza = cola_nueva->final = cola_nueva->llena = 0;
  return cola_nueva;
}
```

Para poder hacer la búsqueda en anchura, primero se declara un camino "vacío", un struct camino_t, en donde se guarda la ciudad 0 como ciudad de partida, con contador de ciudades en 1, el costo en 0 y el camino vacío.

Antes de empezar la búsqueda en anchura, se encola una copia del camino vacío. Se libera la memoria del camino ya que fue encolado. La variable tam_actual que representa el tamaño de la cola, inicializada en 0, ahora se incrementa a 1.
Mientras la cantidad de caminos en la cola sea menor que la cantidad de procesos, se itera en el *while*. Se inicia desencolando el primer camino, por lo cual se decrementa el contador de la cola en uno. Se usa una iteración en un *for* de 1 hasta n (ciudades) para hacer una búsqueda en anchura, encolando cada camino parcial en la cola. Por ejemplo, en la primera iteración por un grafo de 4 ciudades, la cola saca primero del nivel 0 la ciudad 0, y luego encola los caminos parciales 0→1, 0→2 y 0→3 correspondientes a nivel 1 del árbol. En el *for* se reutiliza el mismo struct de camino para cada camino parcial ahorrando tiempo y espacio. Al terminar cada iteración del *while*, se libera el camino. Al volver a iterar en el *while*, se desencola el primer camino parcial del nivel encolado previamente usando la estructura liberada, y se iteran las ciudades no visitadas para agregar los caminos parciales que siguen a ese camino parcial. Se hace lo mismo para cada camino parcial del nivel anterior, recorriendo cada nivel del árbol. Una vez que se llega a un nivel cuya cantidad de caminos es superior al número de procesos, se corta el *while.*

```
while (tam_actual < num_de_procesos) {
    camino = Sacar_primero(cola);
    tam_actual--;
    for (nbr = 1; nbr < n; nbr++)
       if (!Visitada(camino, nbr)) { n $
          Agregar_ciudad_al_camino(camino, nbr);
          Encolar(cola, camino);
          tam_actual++;
          Quitar_la_ultima_ciudad(camino);
       }
    Liberar_camino(camino, NULL);
 }
```

50 Pacheco, P.S. An Introduction to Parallel Programming. 2011 p.139

Repartir los datos

Luego de particionar el árbol, se copia todos los caminos que hay en la cola a un solo arreglo lista_de_colas. Se retorna el puntero de la lista_de_colas, y se guarda el número de caminos de la cola, tam_actual, en el puntero &contador_de_caminos. Proceso 0 hace un MPI_Bcast() a todos los procesos enviando el valor del contador.

```
MPI_Bcast(&contador_de_caminos, 1, MPI_INT, 0, universo);
```

Luego de construir la lista de colas inicial, proceso 0 se encarga de enviarle a cada proceso su subárbol con MPI_Scatterv() usando un contador y un desplazamiento para cada proceso. MPI_Scatterv() a diferencia del MPI_Scatter() común, nos permite distribuir los subárboles entre los procesos. Se usa en casos como este, donde el número de subárboles no se puede dividir perfectamente por el número de procesos. Cada proceso sabe donde esta(n) su(s) subárbol(es) gracias al valor de desplazamiento que se calcula en la función Colocar_caminos_iniciales(). Esta función usa el número de caminos (contador_de_caminos) y número de procesos (num_de_procesos), calcula el cociente y el módulo de la división entre ambos, y luego calcula el desplazamiento para cada proceso y devuelve los arreglos cuantos y desplazamientos y la variable mi_contador (que tiene el número de elementos en el subárbol que le corresponde a cada proceso). También se aloca la lista de caminos lista_de_caminos_de_proceso, en la cual cada proceso va a poder recibir su porción en el MPI_Scatterv() usando **block-cyclic partition**[51]

```
void Colocar_caminos_iniciales(int contador_de_caminos, int cuantos[], int desplazamientos[],
    int* contador_de_proceso, ciudad_t** lista_de_caminos_de_proceso) {
 int quotient, remainder, i;

 quotient = contador_de_caminos/num_de_procesos;
 remainder = contador_de_caminos % num_de_procesos;
 for (i = 0; i < remainder; i++)
    cuantos[i] = quotient+1;
 for (i = remainder; i  < num_de_procesos; i++)
    cuantos[i] = quotient;
 *contador_de_proceso = cuantos[mi_id_proceso];
 desplazamientos[0] = 0;
 for (i = 1; i < num_de_procesos; i++)
    desplazamientos[i] = desplazamientos[i-1] + cuantos[i-1];

 *lista_de_caminos_de_proceso = malloc((*contador_de_proceso)*(n+1)*sizeof(int));
}
```

```
 MPI_Scatterv(lista_de_colas, cuantos, desplazamientos, camino_mpi_t,
       lista_de_caminos, mi_contador, camino_mpi_t, 0, universo)
```

Los primeros tres datos para el envío, son arreglos. La variable lista_de_colas contiene todos los caminos encolados en la lista que construimos previamente. La variable cuantos contiene información de cuantos elementos le corresponden a cada proceso. La variable desplazamientos envía información sobre dónde se encuentra el camino correspondiente en el arreglo lista_de_colas para el proceso receptor. Para estos tres arreglos la posición *i* en cada arreglo corresponde a los datos para el proceso *i*. El tipo de datos camino_mpi_t, es el tipo de datos MPI que registramos en la sección inicial con MPI_Type_contiguous.
Para la recepción, se usa el puntero lista_de_caminos para recibir el camino o caminos parciales correspondiente al proceso receptor (previamente alocado), mi_contador es el número de elementos recibidos en lista_de_caminos, y nuevamente camino_mpi_t es el tipo de datos MPI registrado previamente. Se envían los datos desde el proceso 0 usando el comunicador "universo".

```
 Construir_pila_inicial(pila, lista_de_caminos, mi_contador);
```

Luego de recibir su lista de caminos con MPI_Scatterv(), cada proceso construye su pila con los subárboles que le corresponden. Esta pila será la base para la búsqueda en profundidad Branch and Bound.
Luego, cada proceso llama Construir_pila_inicial() con la pila vacía, la lista de caminos recibida, y el contador mi_contador. Luego de armar las pilas, el proceso 0 libera la lista de colas, y todos los procesos liberan su lista de caminos propia.

```
void Construir_pila_inicial(pila_propia_t pila, ciudad_t lista_de_caminos[], int mi_contador) {
 int i;
 camino_t camino = Alocar_camino(NULL);

 for (i = mi_contador-1; i >= 0; i--) {
    Crear_camino_desde_lista(lista_de_caminos + i*(n+1), camino);
    Empujar_copia(pila, camino, NULL);
 }
 Liberar_camino(camino, NULL);
}
```

51 Pacheco, P.S. An Introduction to Parallel Programming. 2011. p110

Dentro de la función Construir_pila_inicial() se aloca un camino vacío. Se itera usando un *for* reverso desde el tamaño del contador de caminos del proceso mi_contador-1, bajando hasta 0. En el *for*, el proceso llama a Crear_camino_desde_lista() con el puntero de la lista de caminos del proceso, lista_de_caminos. Se usa aritmética de punteros para sumar el valor del índice actual del *for i* por n+1 a la dirección de la lista_de_caminos pasada por parámetro. Esto direccionará en el arreglo lista_de_caminos en el punto que le corresponde a cada camino del proceso. Le llegará la dirección a la función Crear_camino_desde_lista() en la dirección del camino *i*. También se le envía el tipo camino_t alocado previamente para recibir el camino. Una vez que se copian las ciudades y el costo del camino al camino_t tour, se vuelve al *for* de la función Construir_pila_inicial() y se empuja una copia del camino a la pila. De esta manera el proceso tendrá una pila con todos los caminos parciales que le corresponden de su subárbol. Con estos caminos parciales, luego construirá todos los caminos completos que sean viables en relación a los costos mínimos, local y global, en la búsqueda con Branch and Bound.

Búsqueda con Branch and Bound

El algoritmo principal de Branch and Bound es la búsqueda en profundidad usando una pila. Cada proceso itera sobre la pila con a su(s) subárbol(es) recibido(s). Mientras su pila no esté vacía, itera sobre esta, y hace una búsqueda en profundidad para cada subárbol.

Lo primero que hace al entrar al *while*, es sacar de la pila un camino. Pregunta si se ha completado el camino (se tiene Cant_ciudades() == n en el camino actual). En afirmativo, se pregunta si es el mejor camino con la función Mejor_camino(). En esta función se llama a Buscar_mejores_caminos() donde se chequea si hay un mejor costo global entre los otros procesos, usando la comunicación grupal MPI_Iprobe(), con el tag TAG_CAMINO (este tag sirve para distinguir este tipo de mensajes); si existen mensajes de otros procesos que hayan enviado un costo mínimo nuevo, se entra en el *if* y se reciben con MPI_Recv(). Importante notar que no podemos usar MPI_Recv() directamente ya que es bloqueante, necesitamos usar MPI_Iprobe() para verificar qué hay mensajes en el buffer y recién ahí permitir al proceso recibir con MPI_Recv() , sino se quedará esperando en MPI_Recv() y no volverá a la búsqueda en su subárbol. Al recibir un costo_camino, lo chequea contra su mejor camino. Si su camino es mejor, actualiza su mejor camino (costo_del_mejor_camino) con el recibido. Esta variable se ha inicializado en un valor máximo (100000) en un principio, y luego se va actualizando con los costos de los caminos propios o ajenos. Se utiliza esta variable en la búsqueda para descartar caminos con costos más altos.

```
int Mejor_camino(camino_t camino) {
    costo_t costo_actual = Costo_camino(camino);
    ciudad_t ultima_ciudad = Ultima_ciudad(camino);

    Buscar_mejores_caminos();

    if (costo_actual + Costo(ultima_ciudad, ciudad_de_partida) < costo_del_mejor_camino)
       return TRUE;
    else
       return FALSE;
 }

 void Buscar_mejores_caminos(void) {
    int terminado = FALSE, msg_disponible, costo_camino;
    MPI_Status status;

    while(!terminado) {
       MPI_Iprobe(MPI_ANY_SOURCE, TAG_CAMINO, universo, &msg_disponible,
             &status);
       if (msg_disponible) {
          MPI_Recv(&costo_camino, 1, MPI_INT, status.MPI_SOURCE, TAG_CAMINO,
                universo, MPI_STATUS_IGNORE);
          if (costo_camino < costo_del_mejor_camino) costo_del_mejor_camino = costo_camino;
       } else {
          terminado = TRUE;
       }
    }
 }
```

Luego, volviendo a la búsqueda principal, si el camino tiene el mejor costo a nivel global, entra en el *if* y llama a la función Actualizar_el_mejor_camino(), en la cual le agrega su ciudad de partida (nodo 0) al camino para terminar, y llama Bcast_costo_camino(). En esta función envía el nuevo costo mínimo al resto de los procesos menos a sí mismo. Se usa un loop de MPI_Bsend() para enviar de manera no bloqueante el mejor camino a todos los procesos, indicando que envía costo de camino con el tag TAG_CAMINO.

```
 void Bcast_costo_camino(int costo_camino) {
    int destino;

    for (destino = 0; destino < num_de_procesos; destino++)
       if (destino != mi_id_proceso)
          MPI_Bsend(&costo_camino, 1, MPI_INT, destino, TAG_CAMINO, universo);
 }
```

Si aún no entra en el *if* que pregunta si se llegó al final, se hace una búsqueda en profundidad por el subárbol actual para seguir armando el camino. En la estructura iterativa, *for* recorre las ciudades de manera inversa, de n-1 a 1 preguntando en cada una si ya fue visitada y si el costo actual es mayor al mínimo local. La ciudad es viable si no fue visitada, y si el costo es menor que el mínimo local. Si es viable, se agrega al camino, y se empuja una copia del camino a la pila. Luego de empujar una copia del camino, se saca la última ciudad. Se vuelve a iterar, se agrega la próxima ciudad viable, se empuja una copia del nuevo camino a la pila, y así repetidamente hasta completar un camino (o descartarlo).
Cada proceso construye los caminos viables para cada subárbol usando búsqueda en profundidad y luego, al llegar al final de los caminos, hace los chequeos correspondientes con comunicación grupal para comparar con el mínimo global.
En cada iteración se usa un solo struct para construir los caminos. Este se libera en cada iteración que se entra al else, y al volver a iterar se usa nuevamente para el próximo camino actualizado que se saca de la pila. De esta manera se puede reutilizar el espacio sin volver a alocar para cada vuelta. Esto ahorra algo de espacio y tiempo.
Como ejemplo, se puede pensar que si se está trabajando con 5 ciudades (0, 1, 2, 3, 4), y el subárbol inicial es 0→1, imaginando que todas las ciudades son viables por costo, entonces se agregarán los caminos 0 →1→4, 0→1→3 y 0→1→ 2 a la pila. Luego se sacará 0→ 1-→2, y se copiarán 0→1→ 2→ 3 y 0 →1→2 → 4 a la pila. Se sacará 0→1→2 →3 y se copiará 0→1→2→ 3→4. En la próxima iteración, se sacará 0→ 1→2→3→4, entrará en el *if* por tener n ciudades, y se evaluará su costo. Al ser el primer camino evaluado, se guardará su costo como el mejor costo local (será mejor que el valor máximo de 100000 con el cual se inicializa la variable de mejor costo). Se chequea si hay mensajes de costos mejores de parte de otros procesos. Si su costo es el mejor costo de todos los recibidos, se enviará este costo al resto de los procesos.

```
while (!Pila_esta_vacia(pila)) {
   camino_actual = Sacar_ultimo(pila);
   if (Cant_ciudades(camino_actual) == n) {
      if (Mejor_camino(camino_actual)) {
         Actualizar_el_mejor_camino(camino_actual);
      }
   } else {
      for (nbr = n-1; nbr >= 1; nbr--)
         if (Viable(camino_actual, nbr)) {
            Agregar_ciudad_al_camino(camino_actual, nbr);
            Empujar_copia(pila, camino_actual, disponible);
            Quitar_la_ultima_ciudad(camino_actual);
         }
   }
   Liberar_camino(camino_actual, disponible);
}
```

Allreduce()

Luego de terminar de recorrer la pila construyendo los caminos y enviando los costos, se hace liberación de memoria de la pila, y se usa una barrera de sincronización para esperar a que todos los procesos terminen su trabajo. Esto se hace con MPI_Barrier().

"The MPI collective communication function MPI_Barrier ensures that no process will return from calling it until every process in the communicator has started calling it." [52]

```
Liberar_pila(pila);
Liberar_pila(disponible);
MPI_Barrier(universo);
Obtener_mejor_camino_global();
```

Luego de la barrera, se llama Obtener_mejor_camino_global() en el cual se hace un MPI_Allreduce() para deducir el costo total del camino mínimo de todos los procesos.
MPI_Allreduce() recibe valores de todos los procesos en el comunicador indicado, y comunica los resultados de la operación usando un patrón mariposa, guardando el valor de retorno en el comunicador en lugar de en un proceso particular (en este caso se usa el operador de la función mínimo MPI_MINLOC, que indica una reducción a un mínimo "local", o sea, se relacióna al ID del proceso "ganador". Como además del valor mínimo, también comunica el id del proceso que envió ese costo mínimo, y se utiliza MPI_2INT para enviar ambos valores). MPI_Allreduce() luego de computar el mínimo, comunica el resultado de la operación a todos los procesos llamantes.

"MPI_Allreduce stores the result on all the processes in the communicator."[53]

MPI_Allreduce() es bloqueante, ningún proceso puede retornar de la función hasta que todos los procesos la han llamado. Esto garantiza que siempre recibamos el mejor camino, aún cuando algunos procesos ya han encontrado su mejor camino mientras otros están buscando.
Si el proceso 0, coordinador, tiene el mejor camino, entonces retorna de la función. Si es otro el que tiene el mejor camino, este proceso envía su mejor camino al proceso 0 para que lo pueda procesar.

```
MPI_Allreduce(&data_local, &data_global, 1, MPI_2INT, MPI_MINLOC, universo);
if (data_global.id == 0) return;
if (mi_id_proceso == 0) {
   MPI_Recv(mejor_camino_local->ciudades, n+1, MPI_INT, data_global.id,
         0, universo, MPI_STATUS_IGNORE);
   mejor_camino_local->costo = data_global.costo;
   mejor_camino_local->contador = n+1;
} else if (mi_id_proceso == data_global.id) {
   MPI_Send(mejor_camino_local->ciudades, n+1, MPI_INT, 0, 0, universo);
}
```

52 Pacheco, P.S. An Introduction to Parallel Programming. 2011.p.122

53 Pacheco, P.S. An Introduction to Parallel Programming. 2011.p.137

Finalizando

Como pueden haber quedado mensajes del mejor camino sin recibir en el comunicador, usamos una función para “limpiar” el buffer. La funcion Limpiar_cola_mensages() usa MPI_Iprobe() e itera en un *while*. Mientras haya mensajes en el buffer, llama MPI_Recv(). De esta manera “limpiamos”los mensajes que hayan quedado esperando recepción. Es necesario hacer esto para qué MPI_Buffer_detach() y MPI_Finalize() que necesitamos llamar al final del programa puedan ejecutarse normalmente y sin errores.

```
MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, universo, &msg_recibido, &status);
while (msg_recibido) {
   MPI_Recv(buf_trabajo, 100000, MPI_BYTE, status.MPI_SOURCE,
         status.MPI_TAG, universo, MPI_STATUS_IGNORE);
   if (status.MPI_TAG == TAG_CAMINO)
      cuantos[1]++;
   else
      cuantos[0]++;
   MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, universo, &msg_recibido, &status);
}
```

Luego llamamos a MPI_Buffer_detach(), y el proceso 0 se encarga de I/O, imprimiendo el mejor camino a stdout, para informar el resultado de la búsqueda.
En MPI necesitamos explicitar que se ha terminado de trabajar paralelamente llamando a MPI_Finalize(). Previo a esto, liberamos memoria de variables alocadas durante la ejecución, y liberamos los tipos de datos registrados para MPI durante el programa. En nuestro caso, usamos MPI_Type_free() para el arreglo que usamos para almacenar caminos &camino_mpi_t.

Capítulo 6: Diseño del estudio

Las pruebas se hacen con diferentes tamaños de ciudades, con 5, 8, 10, 12 y 15 nodos, para medir adecuadamente las diferencias entre el algoritmo iterativo elegido y la implementación paralela con MPI de una implementación con mapeo estático. Reducimos los casos de prueba a este número ya que con un algoritmo exacto, números de ciudades mayores a 15 se hacen inviables.. Para obtener un speedup absoluto, se utiliza el algoritmo serial más rápido entre los tres que se describen en el capítulo 3. También se toman varias veces los tiempos para cada prueba y se usa el mínimo como métrica Esta recomendación en Pacheco[54] evita las variaciones en tiempos que pueden aparecer en sistemas paralelos:

> A further problem with taking timings lies in the fact that there is ordinarily considerable variation if the same code is timed repeatedly. For example, the oper- ating system may idle one or more of our processes so that other processes can run. Therefore, we typically take several timings and report their minimum.

Se toman tiempos de ejecución seriales y paralelos, y se mide el Speedup absoluto y la eficiencia para cada caso. Se toman en cuenta los consejos de Pacheco[55] como referencia para los resultados.

> **Parallel overhead** is the part of the parallel run-time that's due to any additional work that isn't done by the serial program. In MPI programs, parallel overhead will come from communication. When *p* is large and *n* is small, it's not unusual for parallel overhead to dominate the total run-time and speedups and efficiencies can be quite low.

Las fórmulas para speedup y eficiencia que se usan en este estudio son las siguientes (se utiliza Speedup absoluto, usando el peor tiempo del algoritmo secuencial más rápido. Speedup relativo mediría el tiempo secuencial corriendo el algoritmo paralelo con un solo proceso)[56]:

Speedup:

$$\Psi\ n, p = \frac{\textit{sequential execution time}}{\textit{parallel execution time}}$$

Eficiencia:

$$\varepsilon\ n, p = \frac{\textit{sequential execution time}}{\textit{processors used} \times \textit{parallel execution time}}$$

Para la paralelización, se hacen pruebas con diferentes números de procesos; 2, 4, 8, 16, 32 y 64. La idea es poder medir las diferencias en speedup y eficiencia con diferentes números de procesos y ver los resultados. Como estamos trabajando con una arquitectura de granularidad gruesa, con un diseño de algoritmo que se adapta a esta modalidad donde hay poca comunicación entre procesos y cada proceso hace mucho trabajo independiente. Se espera un speedup sublineal ya que hay una parte importante del programa que se hace secuencial donde se particiona el árbol y se reparten los datos. A partir de esta situación, se plantea una hipótesis de que es muy probable que cuando el número de procesos se aproxime al número de n (ciudades) se den las mejores métricas, donde hay suficientes procesos para repartirse el trabajo, pero también un mínimo adonde se divide el árbol en los primeros niveles (se minimiza la sección no paralelizable) y no hayan demasiadas comunicaciones grupales. A pesar de que la búsqueda en un árbol puede dar resultados de Speedup superlineal por qué realmente se hace menos trabajo en el algoritmo paralelo, en este caso estamos ante un algoritmo con desbalance de carga y un porcentaje del código que no permite la paralelización total, además del overhead de las comunicaciones grupales.

Tenemos que el tiempo secuencial iteramos al menos N × (N - 1) veces, en las cuales se hacen en el pero caso N comparaciones (se chequea si ya fue visitada la ciudad iterando por las ciudades restantes, y se chequea el costo contra el mejor costo), y una asignación (se empuja a la pila una ciudad). Es un tiempo factorial en el peor caso (usualmente se descartan secciones del árbol, pero debemos pensar siempre en el peor de los casos).

$$\textit{sequential execution time} = n \times (n-1) \times (n+2)$$

El código de tiempo paralelo es en la búsqueda en el árbol, donde se itera N/p × (N - 1) veces, y en cada vez se hacen en el peor caso N comparaciones (se chequea si ya fue visitada la ciudad iterando por las ciudades restantes, y luego se chequea el costo contra el mejor costo), y tres 'asignaciones'; se agrega la ciudad al camino, se empuja a la pila el camino y se libera la ciudad.

$$\textit{parallel execution time} = (n/p) \times (n-1) \times (n+3)$$

Entonces se estima que el Speedup debería corresponder con la siguiente fórmula:

$$\text{Speedup} = \frac{n \times (n-1) \times (n+2)}{(n/p) \times (n-1) \times (n+3)}$$

Deberíamos ver que la eficiencia se acerca a la fórmula:

$$\text{Eficiencia} = \frac{n \times (n-1) \times (n+2)}{(\textit{processors used} \times (n/p) \times (n-1) \times (n+3))}$$

Se debe considerar que los tiempos tomados incluyen la partición en el árbol, aunque no sea paralela esta parte del algoritmo. Van a haber overheads de paralelismo para las comunicaciones grupales y el tiempo necesario para alocaciones de memoria. En el código, se usa una barrera previo a medir el tiempo y una función de reducción para obtener el tiempo del proceso más lento.

Probamos con tres tipos de casos - matrices de adyacencia donde no hay caminos óptimos (se usan todos valores iguales), matrices de adyacencias donde hay un camino mejor desde principio a fin (y solo uno), y matrices donde hay caminos mejores parciales para poder evidenciar patrones de comportamiento en relación a diferentes tipos de casos.

Nota: *Inicialmente el estudio se realizó con matrices de valores de adyacencias random. Pero esos valores se descartaron ya que al haber mucha diferencia entre diferentes N, no era fácil de deducir patrones en las pruebas. Nuestro objetivo es comparar el speedup entre diferentes números de N, diferentes casos especificos, entre algoritmos iterativos y con paralelización y diferentes números de procesos, y ver patrones de comportamiento. Esto requiere mayor control sobre los casos medidos. Se debe notar que con este tipo de algoritmo, el tipo de matriz de adyacencias devuelve resultados muy diferentes, y sin control de casos no se pueden evidenciar patrones con seguridad, ya que con demasiada randomización puede que algunas matrices sean más fáciles o rápidas de atravesar que otras, no por el N, sino por la dificultad de encontrar un camino óptimo en ese caso (o vice versa).*

---

54 Pacheco, P.S. An Introduction to Parallel Programming. 2011.p.139

55 Pacheco, P.S. An Introduction to Parallel Programming. 2011.p.139

56 Aunque en este estudio, se recomienda usar Speedup relativo. Sun, X.-H. and J.L. Gustafson, Toward a better parallel performance metric, Parallel Computing 17, 1093-1109. 1991.

Se utiliza una arquitectura de tipo cluster con 8 nodos, cada uno con cuatro núcleos con dos hilos de procesamiento cada uno. En este cluster es posible tener 64 procesos paralelos corriendo simultáneamente.

Carasteristicas de cluster HPC (salida del comando lscpu)

Orden de los bytes:	Little Endian	
Tamaños de las direcciones:	40 bits physical, 48 bits virtual	
CPU(s):	8	
Lista de la(s) CPU(s) en línea:	0-7	
Hilo(s) de procesamiento por núcleo:	2	
Núcleo(s) por «socket»:	4	
«Socket(s)»	1	
Modo(s) NUMA:	1	
ID de fabricante:	GenuineIntel	
Familia de CPU:	6	
Modelo:	26	
Nombre del modelo:	Intel(R) Xeon(R) CPU	E5520 @ 2.27GHz
Revisión:	5	
CPU MHz:	1595.926	
CPU MHz máx.:	2262,0000	
CPU MHz mín.:	1596,0000	
BogoMIPS:	4521.75	
Virtualización:	VT-x	
Caché L1d:	32K	
Caché L1i:	32K	
Caché L2:	256K	
Caché L3:	8192K	
CPU(s) del nodo NUMA 0:	0-7	

Capítulo 7: Resultados y Análisis

Como fue discutido en el capítulo anterior, para el estudio se utilizaron tres tipos de casos base para diferentes valores de N. Los casos son los siguientes:

Caso 1, no hay camino (todos los valores son iguales)
Caso 2, camino parcial intermedio (principio y fin indeterminados)
Caso 3, camino predeterminado de comienzo a final

El primer caso, en el cual no hay camino, nos permite analizar el "peor caso", en el cual se deben buscar en todos los subárboles para encontrar un mejor camino. Como no se encuentra uno, se debe seguir buscando hasta zanjar todas las posibilidades. En este caso, la búsqueda en n = 15 se hizo inviable, tardando horas, y hasta días. Por esto no se incluyeron los resultados en este estudio. Se iniciaron algunas pruebas, pero al demorarse tanto, se decidió desistir al pasar varias horas de espera.

Este caso y el caso en el cual se diseña un solo camino desde principio a fin, sirven para ilustrar casos extremos donde sabemos bien los resultados. En el caso de tener un camino desde el primer punto hasta el final, hace fácil el trabajo para la mayoría de los procesos, que podrán descartar sus subárboles con costo excesivo desde temprano. En este caso, la búsqueda se hace rápida, y un n = 15 que en el peor caso tardó varias horas, aquí tardó solo minutos.

Un caso más común es el segundo caso, donde no se sabe desde un principio como se va a proceder, y solo se elabora un comienzo de una solución en el intermedio. Esto permite ver una aproximación a casos comunes, aunque podríamos probar muchas más variaciones (por ejemplo, tres caminos de menor costo), dado el tiempo adecuado.

Para calcular Speedup y Eficiencia de manera absoluta, los tiempos iterativos se toman ejecutando el algoritmo iterativo discutido en capítulo 3 que da el mejor tiempo (el segundo caso). Esta versión implementa un stack que emula la pila del sistema con una lista, pero no guarda el camino entero en la pila, solo la ciudad actual. De esta manera el algoritmo se hace más eficiente, aunque menos paralelizable.

También debemos tener en cuenta la ley de Amdahl[57], ya que los procesos deben esperar mientras un solo proceso, proceso 0, calcula el tamaño de las pilas, y los desplazamientos, y construye la lista de pilas para todos los procesos usando la búsqueda en anchura. Aunque la búsqueda en un árbol es un algoritmo que puede, en algunos casos, lograr un Speedup superlineal, en este caso vemos que hay severos límites al speedup en todos los casos. Por un lado, podemos hipotetizar que esto es porque una parte importante del programa se ejecuta por un solo proceso, y por otra parte la formulación irregular del problema produce un desbalance en la carga de trabajo, cuya resolución también agregaría déficit en performance por necesidad de sincronización y comunicación, aunque esto debería ser testeado con diferentes números de procesos para verificar la hipótesis. Por otro lado se ven variaciones en relación al tamaño de datos en el caso de camino parcial con N= 15, donde se ven los mejores números de Speedup, teniendo en cuenta la ley de Gustavson, donde si agregamos trabajo al mismo tiempo que agregamos más procesadores puede verse aminorada la situación señalada por Amdahl

> Gustafson's law addresses the shortcomings of Amdahl's law, which is based on the assumption of a fixed problem size, that is of an execution workload that does not change with respect to the improvement of the resources. Gustafson's law instead proposes that programmers tend to increase the size of problems to fully exploit the computing power that becomes available as the resources improve.[58]

En el caso del camino parcial intermedio es donde vemos una mejora importante en speedup > 5 con N= 15 con más de 8 procesos, y las mejores métricas en eficiencia (cercanas a 0,6) en todo el estudio. Hay algunas anomalías en los patrones del speedup y la eficiencia en las métricas del estudio en general.. En particular notamos que el uso de entre 4 y 16 procesos, y usando 8 procesos específicamente, tiene cierta ventaja en muchos puntos del estudio. Esto podría ser porque es un promedio del número de ciudades que usamos en estos casos, y 8 es un número de procesos que se aproxima al número de ciudades promedio. Otra hipótesis es que el cluster utilizado tiene 8 procesos por nodo, y la comunicación inter-nodo es mucho más lenta que la comunicación interna al nodo. Estas pueden ser algunas de las razones de por qué se nota una mejora en el Speedup en la mayoría de los casos con 8 procesos en las tablas del caso sin camino, y con un solo camino. Para analizar con más detenimiento este punto, se hicieron tablas que promediaron los tiempos de ejecución para los tamaños de n = 5 a n = 12 (ya que el primer caso no tiene n = 15 y se hace más fácil la comparación entre casos usando las mismas métricas), separando por número de procesos. En todos los casos, los tiempos promediados de ejecución más rápidos fueron con 8 procesos.

Hay pequeñas alteraciones en los patrones en el caso del camino parcial, donde es esperable que no se siga una moda específica, ya que los caminos parciales intermedios generan las irregularidades comunes en TSP, basado en que no hay un comportamiento esperado en cuanto la búsqueda y encuentro de caminos. Sin embargo, justamente en este caso, se nota una mayor ventaja en cuanto a la eficiencia en torno a agregar procesos. Mientras en los casos predecibles es notoria la falta de eficiencia pasados los 8 procesos, en el caso menos predecible del camino parcial, sigue siendo viable la eficiencia para un número de procesos mayor a 8 para Ns grandes.

---

57 https://demonstrations.wolfram.com/AmdahlsLaw/

58 Gustafson, J. L. (May 1988). "Reevaluating Amdahl's Law". *Communications of the ACM*. **31** (5): 532–3

Resultados y gráficas

Caso 1: No existe un mejor camino

Tiempo de ejecución iterativo (sin camino)

n = 5	n = 8	n = 10	n = 12	n = 15
0.0000231266	0.003890038	0.3243999	45.85871	

Tiempo de ejecución paralelo (sin camino)

procesos	n = 5	n = 8	n = 10	n = 12	n = 15
2	0.000073054	0.0042372	0.3502244	45.02683	
4	0.000111886	0.002232197	0.2165592	22.82423	
8	0.02292791	0.03730816	0.1492446	15.54695	
16	0.02661057	0.04234065	0.4948537	86.23886	
32		0.0436964	1.249451	112.6817	
64		0.2336378	1.027771	160.5826	

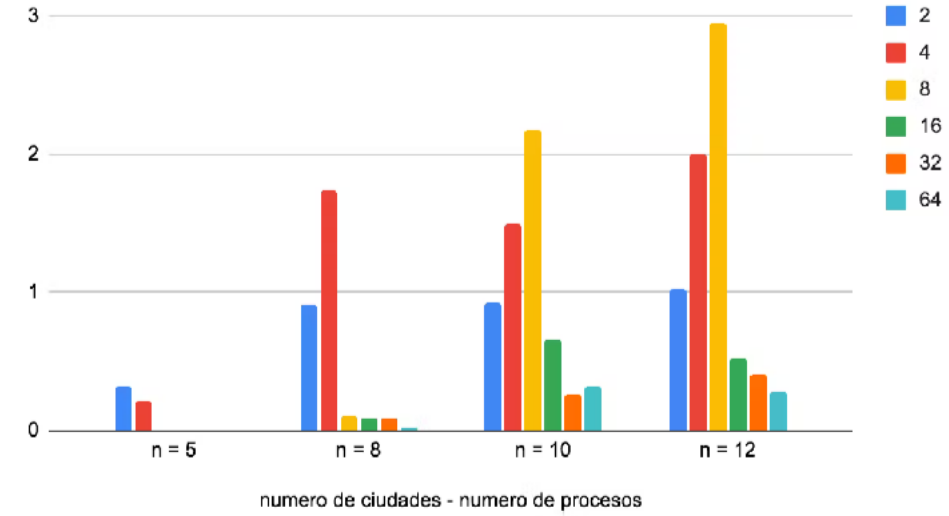
Speedup (sin camino)

Procesos	n = 5	n = 8	n = 10	n = 12	n = 15
2	0.3165685657	0.9180680638	0.9262629902	1.018475207	
4	0.2066978889	1.742694753	1.497973302	2.009211702	
8	0.001008665857	0.1042677527	2.173612312	2.94969174	
16	0.0008690757094	0.09187478227	0.6555470839	0.5317638707	
32		0.08902422168	0.2596339512	0.4069756669	
64		0.01664986573	0.3156344166	0.28557708	

Eficiencia (sin camino)

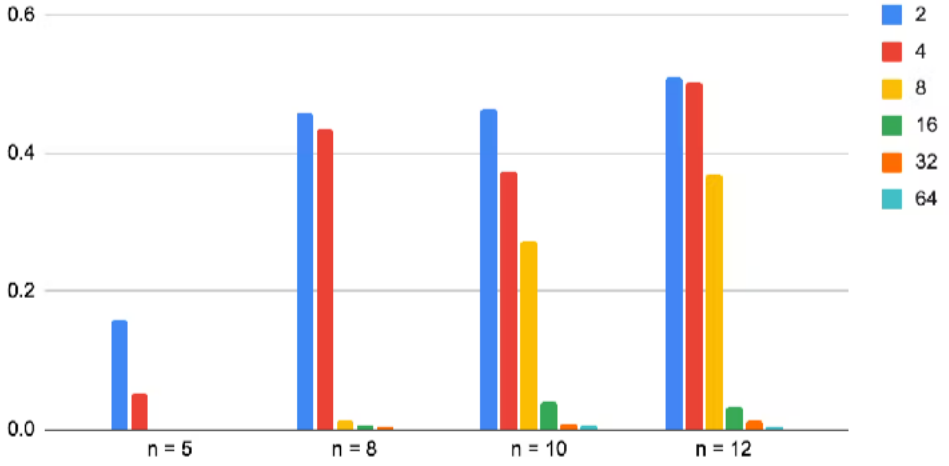
Name	n = 8 2	n = 8	n = 10	n = 12	n =15
2	0.158284283	0.459034	0.4631315	0.50924	
4	0.051674472	0.435673688	0.3744933	0.5023	
8	0.00012608	0.01303347	0.2717015	0.36871	
16	0.00005432	0.00574217	0.0409717	0.03324	
32		0.002782	0.0081136	0.01272	
64		0.0002602	0.0049318	0.00446	

Speedup: sin camino

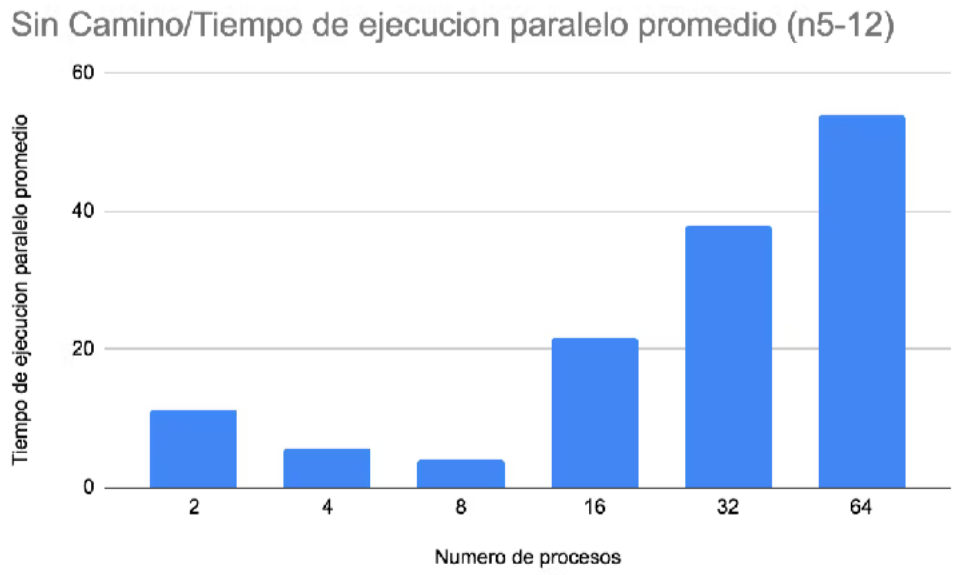


numero de ciudades - numero de procesos

Eficiencia: sin camino



numero de ciudades - numero de procesos



Caso 2: Existe un mejor camino parcial intermedio

Tiempo de ejecución iterativo (camino parcial)

n = 5	n = 8	n = 10	n = 12	n = 15
0.00001978874	0.002364159	0.191402	3.932629	9214.981

Tiempo de ejecución paralelo (camino parcial)

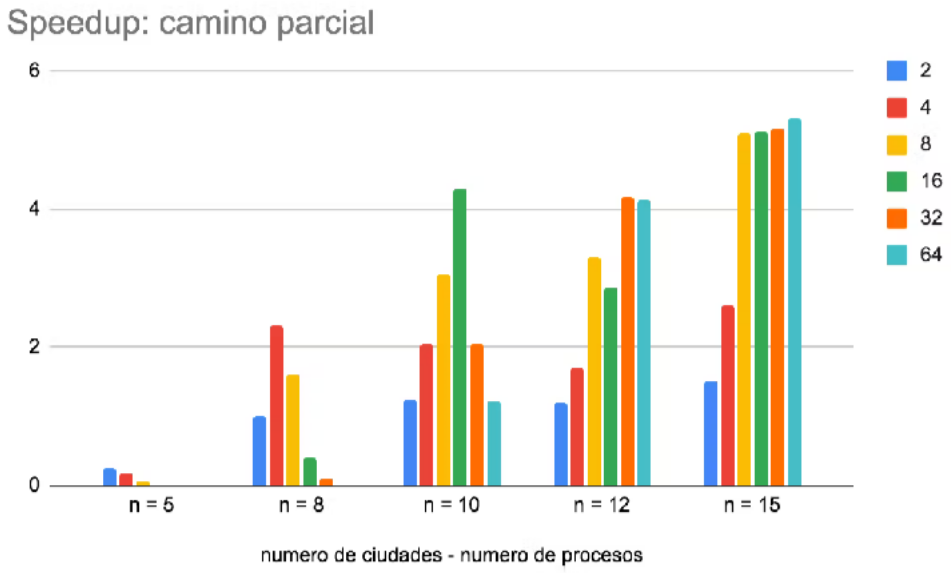
procesos	n = 5	n = 8	n = 10	n = 12 n = 15
2	0.000081994	0.002332279	0.1522438	3.275998 6109.936
4	0.000104482	0.00102513	0.09377705	2.292064 3532.138
8	0.00037407	0.001478819	0.06266016	1.192487 1802.372
16	0.005018551	0.005790932	0.0445849	1.373752 1798.652
32		0.02406084	0.09313007	0.9395402 1777.24
64		0.327415	0.1563893	0.9496293 1731.016

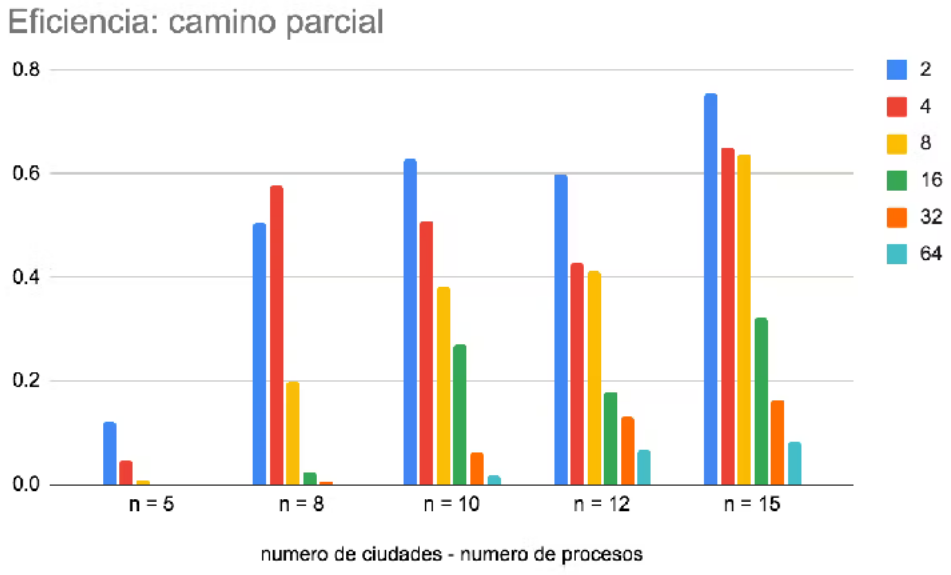
Speedup (camino parcial)

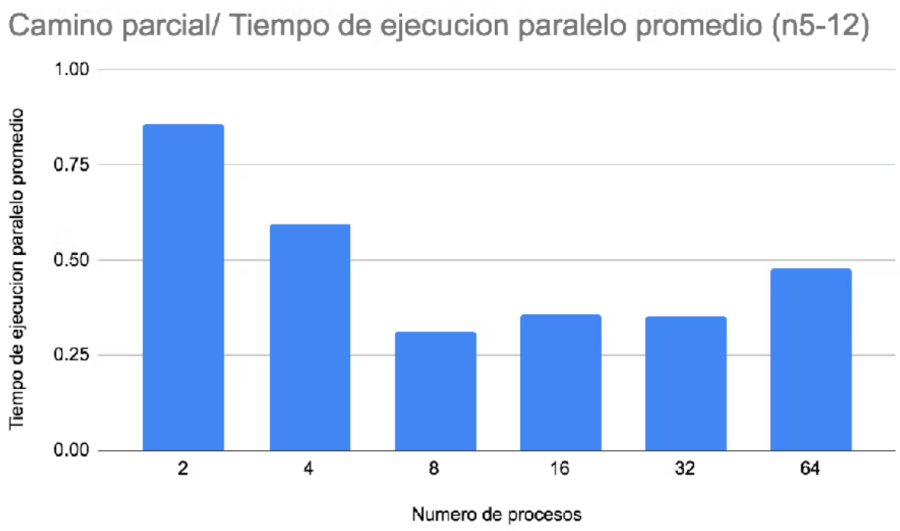
procesos	n = 5	n = 8	n = 10	n = 12	n = 15
2	0.2413437569	1.013669034	1.25720719	1.200436936	1.508195994
4	0.1893985567	2.306204091	2.041032427	1.71575881	2.608896085
8	0.05290116823	1.598680434	3.054604393	3.297838048	5.112696491
16	0.003943118243	0.4082519014	4.292978116	2.862692102	5.123270649
32		0.09825754213	2.055211598	4.185695301	5.184995274
64		0.007220680177	1.223881685	4.141225423	5.323452238

View of Eficiencia, camino mejor parcial

procesos	n = 5	n = 8	n = 10	n = 12	n = 15
2	0.1206718784	0.5068345168	0.628603595	0.6002184678	0.7540979971
4	0.04734963917	0.5765510228	0.5102581069	0.4289397024	0.6522240213
8	0.006612646029	0.1998350542	0.3818255491	0.412229756	0.6390870614
16	0.0002464448902	0.02551574384	0.2683111322	0.1789182564	0.3202044156
32		0.003070548192	0.06422536244	0.1308029781	0.1620311023
64		0.0001128231278	0.01912315133	0.06470664724	0.08317894123







Caso 3: Existe un camino de principio a fin

Tiempo de ejecución iterativo (un camino)

n = 5	n = 8	n = 10	n = 12	n = 15
0.000008821487	0.0003709793	0.01105404	0.648442	110.9308

Tiempo de ejecución paralelo (un camino)

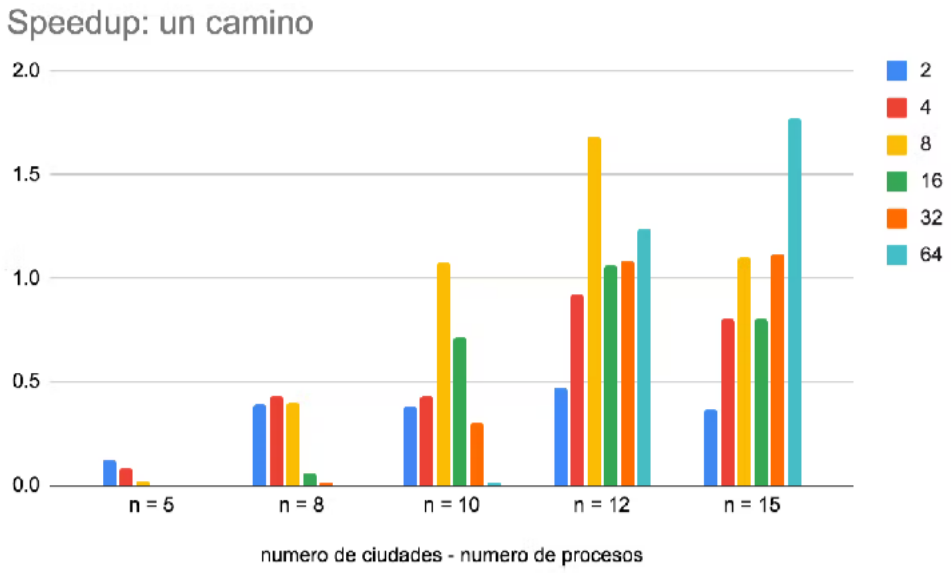
procesos	n = 5	n = 8	n = 10	n = 12	n = 15
2	0.000071588	0.000950058	0.02871672	1.366815	303.9704
4	0.000103366	0.0008644310	0.02580409	0.6984877	137.4295
8	0.000434765	0.0009277780	0.01028483	0.3857512	100.6779
16	0.01494418	0.006055630	0.01554126	0.6104376	138.3322
32		0.02937712	0.03631297	0.5989708	99.09387
64		1.073074	0.9954581	0.5216227	62.43953

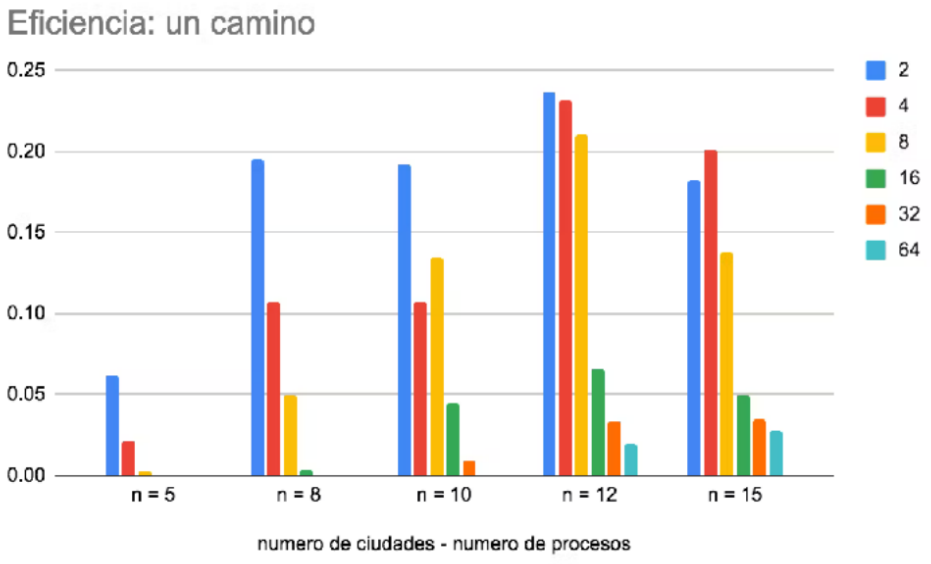
Speedup (un camino)

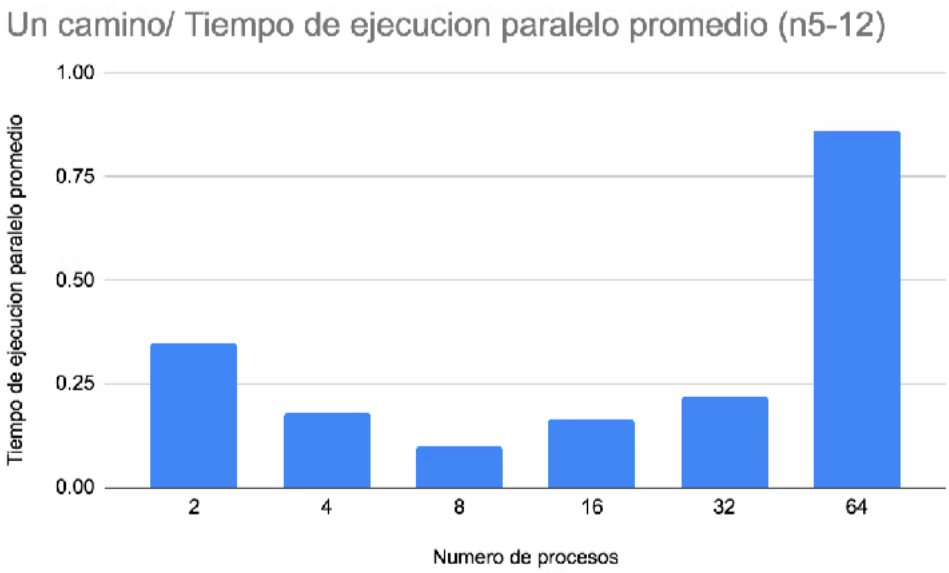
procesos	n = 5	n = 8	n = 10	n = 12	n = 16
2	0.1232257781	0.3904806864	0.384933934	0.4744182644	0.3649394809
4	0.08534224987	0.4291601065	0.4283832524	0.9283513511	0.8071833194
8	0.02029024185	0.3998578324	1.074790735	1.680985049	1.101838636
16	0.0005902958208	0.06126188357	0.7112705147	1.062257633	0.8019159675
32		0.01262817118	0.3044102424	1.082593676	1.119451688
64		0.0003457164184	0.01110447542	1.24312458	1.776611707

Eficiencia (un camino)

procesos	n = 5	n = 8	n = 10	n = 12	n = 15
2	0.06161288903	0.1952403432	0.192466967	0.2372091322	0.1824697405
4	0.02133556247	0.1072900266	0.1070958131	0.2320878378	0.2017958299
8	0.002536280232	0.04998222905	0.1343488419	0.2101231312	0.1377298295
16	0.0000368934888	0.003828867723	0.04445440717	0.06639110206	0.05011974797
32		0.0003946303492	0.009512820075	0.03383105237	0.03498286524
64		0.000005401819038	0.0001735074284	0.01942382157	0.02775955793







Capítulo 8: Conclusiones

El problema TSP es amplio y no tiene soluciones conocidas para todos los casos que sean mejores que la solución de fuerza bruta. Este es un tema conocido y se conoce como parte de la incógnita P = NP. Recorrimos una serie de soluciones exactas y soluciones aproximadas en la historia de este problema. Como el objetivo de este estudio no es conocer una mejor solución, sino poder paralelizar una solución serial y hacer pruebas para comparar los tiempos de ejecución, se elige una solución simple de paralelizar, sabiendo que no es la mejor opción.

La consigna fue elaborar una solución paralela para arquitectura distribuida, por lo cual se trabajó con el protocolo MPI, específicamente la implementación openMPI, sobre un cluster de 8 nodos con 4 núcleos por nodo, y 2 hilos de procesamiento por núcleo. Se desarrolló la solución en base del estudio sobre TSP publicado en el libro "An introduction to parallel programming" del Peter Pacheco. En las pruebas pudimos deducir que hay mejor performance con un número de procesos entre 8 y 16. A futuro podríamos versionar un estudio similar haciendo pruebas con 8, 9 10, 11, 12, 13, 14, 15 y 16 procesos para ver adonde se encuentra el punto óptimo para la performance en problemas de este tipo y tamaño.

El algoritmo con mapeo estático nos demuestra solo una cara de la solución. Trabajando con paralelismo es importante aprovechar al máximo el potencial de trabajo de los procesos disponibles. En el estudio se considera el desafío de balance de carga, se reconoce que el estudio podría ampliarse, desarrollando la implementación del algoritmo paralelo con mapeo dinámico de tareas, para ver si un mejor balance de carga se justifica contra el mayor overhead de paralelización en relación con la comunicación y sincronización extra necesaria entre procesos.

Bibliografía

Andrews G. “Foundations of Multithreaded, Parallel and Distributed Programming”, Addison Wesley, 2000

Balas, E. . The prize collecting traveling salesman problem. Networks, 19(6), 621–636. 1989 doi:10.1002/net.3230190602

Ben-Ari, M. “Principles of Concurrent and Distributed Programming, 2/E”. Addison-Wesley. 2006. ISBN 0-321-31283-X

Brinch Hansen, P., “Studies in Computational Science. Parallel Programming Paradigms”, Prentice Hall, 1995.

Chandra,B. et al. New results on the old k-opt algorithm for the TSP. 5th ACM-SIAM Symp. on Discrete Algorithms, 1994, pp.150-159.

Chandy, Misra, “Parallel Program Design. A Foundation”, Addison Wesley, 1988.

Christofides, N. Worst-case analysis of a new heuristic for the travelling salesman problem. Report No. 388, GSIA, Carnegie-Mellon University, Pittsburgh, PA, 1976. https://apps.dtic.mil/dtic/tr/fulltext/u2/a025602.pdf"

Concorde https://www.math.uwaterloo.ca/tsp/concorde.html

Cook W Computing in Combinatorial Optimization. In: Steffen B., Woeginger G. (eds) Computing and Software Science. Lecture Notes in Computer Science, vol 10000. Springer, Cham. 2019 https://doi.org/10.1007/978-3-319-91908-9_3

Cook, W. In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation, Princeton University Press, 2014

Dantzig, G., Fulkerson, R., and Johnson, S. 1954, "Solution of a large-scale traveling-salesman problem", en Operations Research

Davendra, D., et al, "CUDA Accelerated 2-OPT Local Search for the Traveling Salesman Problem", in Novel Trends in the Traveling Salesman Problem. London, United Kingdom: IntechOpen, 2020 [Online]. Available: https://www.intechopen.com/chapters/72774 doi: 10.5772/intechopen.93125

David L. Applegate; Robert E. Bixby; Vašek Chvátal; William J. Cook, The Traveling Salesman Problem: A Computational Study , Princeton University Press, 2007.

Documentación OpenMPI https://ccportal.ims.ac.jp/en/node/2635

Dongarra, J.,et al SPMD Computational Model. Encyclopedia of Parallel Computing, 1933–1943. 2011. doi:10.1007/978-0-387-09766-4_26

Dorigo M, Gambardella L. Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation. April 1997; 1(1):53-66. DOI: 10.1109/4235.585892

Filminas de las clases teóricas de la cátedra de Programación Concurrente ATIC, Info, UNLP, 2021

Foster, I. Designing and Building Parallel Programs, Addison-Wesley, Reading, MA, 1995.

Gao, Y. Heuristic Algorithms for the Traveling Salesman Problem, 2020 https://medium.com/opex-analytics/heuristic-algorithms-for-the-traveling-salesman-problem-6a53d8143584

Garey, M.R.; Johnson, D.S. Computers and Intractability: A Guide to the Theory of NP-Completeness. New York: W.H. Freeman. 1979

Grama A., Gupta A., Karypis G., Kumar V., "An Introduction to Parallel Computing. Design and Analysis of Algorithms", Pearson Addison Wesley, 2nd Edition, 2003

Gustafson, J. L. (May 1988). "Reevaluating Amdahl's Law". *Communications of the ACM*. **31** (5): 532–3

Gutin, G. and Yeo, A.. The Greedy Algorithm for the Symmetric TSP. Algorithmic Oper. Res., Vol.2, 2007, pp.33--36.

Held, M. and Karp. R.M. The traveling-salesman problem and minimum spanning trees. Op.Res.18, 1970, pp.1138-1162.

Helsgaun, K.. General k-opt submoves for the Lin-Kernighan TSP heuristic. Mathematical Programming Computation, 2009. http://akira.ruc.dk/~keld/research/LKH/LKH-1.3/DOC/LKH_REPORT.pdf

John E. Mitchell, The Held & Karp Relaxation of TSP, 2019

Johnson, David S.; McGeoch, Lyle A. (1997). "The Traveling Salesman Problem: A Case Study in Local Optimization" (PDF). In E. H. L. Aarts; J. K. Lenstra (eds.). Local Search in Combinatorial Optimization. London:

John Wiley and Sons. pp. 215–310

Jordan H.F., Alaghband G., Jordan H.E., "Fundamentals of Parallel Computing", Prentice Hall, 2002

Keld Helsgaun “An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic“, European Journal of Operational Research. 126, 106–130 2000

Libreria TSP Universidad de Heidelberg. http://comopt.ifi.uni-heidelberg.de/projects/

Lin, Shen; Kernighan, B. W. (1973). "An Effective Heuristic Algorithm for the Traveling-Salesman Problem". Operations Research

Little,J.D.C., et al, "An algorithm for the traveling salesman problem", Operations Research 11, 972-989. 1963
https://dspace.mit.edu/bitstream/handle/1721.1/46828/algorithmfortrav00litt.pdf

McCool, Michael; et al (2013). Structured Parallel Programming: Patterns for Efficient Computation. Elsevier. p. 61. ISBN 978-0-12-415993-8.

Metaheuristicas, sitios web www.metaheuristics.org, www.aco-metaheuristic.org

Monteiro, M. J. T. P../ Parallelizing Irregular Algorithms: a Pattern Language.Conference on Pattern Languages of Programs. 2011. pp. 4:1-4:17
https://www.hillside.net/plop/2011/papers/A-24-Monteiro.pdf

Naiouf,M. et al.Fundamentos de cómputo paralelo y distribuido de altas prestaciones. Construcción y evaluación de aplicaciones.2014

Pacheco, P. An Introduction to Parallel Programming. 2011

Padua, D., Ghoting, A., Gunnels, J. A., et al. MPI (Message Passing Interface). Encyclopedia of Parallel Computing, 1184–1190, 2011

Raynal M. “Concurrent Programming: Algorithms, Principles, and Foundations”. Springer, 2012.

Sun, X.-H. and Gustafson,J.L. . Toward a better parallel performance metric, Parallel Computing 17, 1093-1109. 1991.

Taubenfeld, G. “Synchronization Algorithms and Concurrent Programming”. Prentice Hall. 2006

Uchida A., Ito, Y., Nakano, K. An Efficient GPU Implementation of Ant Colony, Optimization for the Traveling Salesman Problem, 2012 Third International Conference on Networking and Computing https://www.cs.hiroshima-u.ac.jp/cs/_media/4893a094.pdf

Valenzuela, C.L. and Jones, A.J. Estimating the Held-Karp lower bound for the geometric TSP. European J. Op. Res., 102:1, 1997, pp.157-175

Wilkinson, B, Allen, M., Parallel Programming: techniques and applications using networked workstations and parallel computers” Pearson, 2005

Writing Efficient Programs https://www.youtube.com/watch?v=-40jMw4HN-o&list=PLGvfHSgImk4aweyWlhBXNF6XISY3um82_&index=91

You, Y-S. Parallel Ant System for Traveling Salesman Problem on GPUs Taiwan Evolutionary Intelligence Laboratory (TEIL)http://www.gpgpgpu.com/gecco2009/4.pdf