Universidad Nacional Autónoma de México
Facultad de Ciencias
Complejidad Computacional & Algoritmos Paralelos
Semestre 2025-2
319526240 - Romero Palacios Santiago
320268135 - Nava Córdova Mariana
12 de Junio de 2025

Reporte de Proyecto

Implementación de un Algoritmo Genético Paralelo para el Problema del Agente Viajero

Índice

Resumen/Abstract

Elaboramos y evaluamos un programa que resuelve el problema del agente viajero mediante un algoritmo genético en paralelo, tanto en CPU como en GPU. Para ello, se realizó una implementación en secuencial y dos en paralelo: en CPU con OpenMP y GPU con CUDA. El lenguaje de elección fue C++. Indagamos qué tan eficientemente podemos obtener buenas aproximaciones de problemas NP-Duros con hardware moderno de grado de consumidor, usando el problema del agente viajero euclidiano en su versión de optimización como caso de estudio. En GPU seguimos cercanamente a Gaxiola Sánchez et al., 2014, pp. 84-90, y para la versión secuencial a Merino Trejo, 2023, pp. 55-69. Utilizamos las recomendaciones de evaluación de metaheurísticas en paralelo de Alba y Luque, 2005.

Motivación

La logística mueve a la economía mundial, siendo un factor crítico para el éxito o fracaso de muchos negocios que hacen uso de la importación y exportación de bienes. (Michigan State University, s.f.)

Las pequeñas y medianas empresas (PYMES) a menudo se enfrentan a limitaciones de recursos que hacen que las soluciones logísticas tradicionales y de alto costo sean poco viables. Esto dificulta el escalamiento y crecimiento de sus operaciones, pues rápidamente se vuelven altamente ineficientes por la intractabilidad de obtener soluciones buenas manualmente. (Michigan State University, s.f.)(Warnier, 2024)

Existen alternativas de renta de tiempo de cómputo, pero esto puede no ser conveniente si se trabaja con información sensible, además de generar una dependencia en el proveedor del servicio en las operaciones de la organización. (Warnier, 2024) Aprovechar hardware de grado de consumidor relativamente económico para resolver problemas logísticos computacionalmente difíciles es una alternativa práctica y rentable.

Nos interesa investigar cómo aprovechar al máximo los recursos computacionales de CPU y GPU económicos en la implementación de algoritmos de optimización para generar mejoras significativas en el rendimiento sin necesidad de hardware más especializado ni depender de la renta de recursos de cómputo. Usamos el Problema del Agente Viajero Euclidiano como un caso de estudio de juguete.

Preliminares

Problema del Agente Viajero

El problema del agente viajero (PAV o bien TSP) consiste en encontrar el camino más corto que permita a un viajero recorrer un conjunto de ciudades exactamente una vez y regresar al punto de origen. Dado que pertenece a la clase de problemas NP-duro (también conocido como NP-difícil), su resolución exacta se vuelve intractable conforme el número de ciudades crece, lo que hace imprescindible el uso de heurísticas y metaheurísticas para obtener soluciones aproximadas de calidad. (Merino Trejo, 2023) (Cormen et al., 2009)

La tesis de Merino Trejo, 2023 aborda diferentes enfoques heurísticos, como el método del vecino más cercano, inserción aleatorizada y la mejora mediante búsqueda local con 2-intercambio (2-opt). Estas estrategias se conocen como metaheurísticas de trayectoria, pues se sigue un camino en búsqueda del óptimo. Otra técnica que aborda, es la del algoritmo genético, que exploraremos a continuación.

Algoritmos Genéticos

Los algoritmos genéticos son una clase de algoritmos evolutivos inspirados en la teoría de la evolución de Darwin (Mitchell, 1998). Operan sobre una población de soluciones codificadas como cromosomas (en el problema del agente viajero, permutaciones de ciudades) y aplican operadores de selección, cruzamiento y mutación para generar nuevas soluciones.

Su fuerte es su capacidad para explorar amplias regiones del espacio de búsqueda y evitar caer fácilmente en óptimos locales. Algunos enfoques recientes han combinado estos algoritmos con técnicas como "extinciones masivas" (Padilla Martínez, 2019) o restricciones propias de problemas relacionados como el de rutas de vehículos (VRP) (Tavera, 2018).

Algoritmos Paralelos

Dado que muchos de los pasos en algoritmos genéticos, como la evaluación de aptitud, la selección o la cruza pueden ejecutarse de manera independiente, su implementación en paralelo suele mejorar considerablemente la eficiencia, especialmente cuando se trata de instancias de gran tamaño. (Alba & Tomassini, 2002) (Alba & Luque, 2005)

Para implementar este paralelismo, se han desarrollado herramientas como OpenMP y CUDA, ampliamente utilizadas en el cómputo de alto rendimiento.

OpenMP es una interfaz de programación que facilita la ejecución paralela en arquitecturas de memoria compartida, como los procesadores multinúcleo. Esto nos permite dividir automáticamente tareas repetitivas como los bucles en distintos hilos de ejecución. (Chapman et al., 2008)

Por otro lado, CUDA es una plataforma desarrollada por NVIDIA que permite ejecutar código en GPU, las cuales están diseñadas para realizar miles de operaciones en paralelo con sus numerosos CUDA cores (así como otro tipo de núcleos de procesamiento), agrupados en Streaming Multiprocessors, que a su vez se agrupan en TPCs y estos en GPCs. Esto convierte a CUDA en una herramienta práctica para acelerar cálculos intensivos, como la evaluación masiva de rutas o la aplicación simultánea de operadores genéticos. (Sanders & Kandrot, 2010) (Gaxiola Sánchez et al., 2014)

Metodología

Se elaboraron cuatro programas, los cuales constan de tres solucionadores y un configurador ejecutable.

El configurador permite intercambiar el solucionador, seleccionar el número máximo de iteraciones, seleccionar el número de ejecuciones del programa, cambiar el nivel de paralelismo, seleccionar el problema a resolver, etc.

El configurador además mostrará estadísticas de las ejecuciones del solucionador junto con la mejor solución encontrada, así como las guardará en archivos en formato JSON. Además, validará que las soluciones candidatas sean válidas y que reporten su costo adecuadamente. Así nos aseguramos de que los solucionadores sean correctos.

A continuación describiremos los solucionadores, que satisfacen la siguiente interfaz de C.

```
// La versión del solucionador.
extern "C" EXPORT u64 solver_Version();

// La versión del configurador con el cual es compatible el solucionador.
extern "C" EXPORT u64 solver_Compatibility();

// El nombre del solucionador.
extern "C" EXPORT const char * solver_Name();

// La descripción del solucionador.
extern "C" EXPORT const char * solver_Description();

// Inicialización al cargar el solucionador.
extern "C" EXPORT b32 solver_Setup();

// Realiza el proceso de desinicialización necesario al cerrar el solucionador.
extern "C" EXPORT void solver_Unload();

// Da una solución aproximada al problema.
extern "C" EXPORT b32
solver_Solve(tsp_instance *__restrict__ Tsp,
             i32 *__restrict__ out_Permutation,
             u64 *__restrict__ Iterations,
             r32 Cutoff,
             i32 Parallelism);
```

En todo caso, realizamos las pruebas de rendimiento de los solucionadores con instancias obtenidas de TSPLIB95, las cuales se encarga de leer nuestro configurador.

En particular, se realizarán dos tipos de pruebas, de esfuerzo máximo y de corte, donde en las pruebas por esfuerzo máximo corremos el solucionador un número fijo de iteraciones, mientras que mediante punto de corte, si alcanzamos un valor suficientemente cercano al óptimo conocido nos detenemos.

Esto es así pues como menciona (Alba & Luque, 2005), nos interesa encontrar el tiempo neto en que encontramos una solución aceptable, además de que nos interesa conocer el comportamiento de convergencia con respecto al tamaño del problema.

Además, tendremos que ejecutar todas las pruebas un número estadísticamente significativo de veces, pues estamos tratando con algoritmos fundamentalmente aleatorios. Esto también es algo que se menciona en (Alba & Luque, 2005).

Descripción del Programa Secuencial

Se elaboró un solucionador secuencial `cpuseq` para comparar el rendimiento y speedup con el resto de los programas.

A continuación describiremos las partes principales del algoritmo genético implementado.

Población

Se genera una población inicial aleatoria de tamaño máximo entre $2n$ y 1000, donde n es el número de vértices del problema de entrada.

Esto es conformado por un arreglo de permutaciones de números de cero hasta el tamaño de la población P.

Generación

La primera generación será construida aleatoriamente y conservaremos a los 5 mejores individuos.

Selección

La selección de los dos individuos de la población actual P^t que se cruzarán para dejar su descendiente en P_i^{t+1} será dada por P_i^t y P_j^t donde j es el ganador de un torneo de cinco individuos tomados aleatoriamente.

En otras palabras, escogeremos para la "casilla" i a quien ya está ahí y al ganador de tomar 5 otros individuos al azar.

Tomaremos a un individuo de élite como el segundo padre con baja probabilidad, en particular para las pruebas que realizamos usamos el 10 %.

Cruza

Escogeremos un intervalo aleatorio $[a, b] \subseteq \{0, 1, 2, \ldots, n-1\}$ y primero ubicamos a todos los elementos en $P_i^t[k]$ para todo $k \in [a, b]$. Después colocaremos en P_i^{t+1} para toda $k \in [0, b]$ a los valores $P_j^t[k]$ que no se encuentren entre los elementos que identificamos anteriormente. Luego, agregamos a todo $P_i^t[k]$ con $k \in [a, b]$ y finalmente agregamos a todo $P_j^t[k]$ para $k \in [b, n-1]$ que no estén en el conjunto que identificamos de P_i.

Así, en $O(n)$ construimos la cruza, donde intuitivamente el proceso que realizamos fue tomar un intervalo de un padre e insertarlo a la mitad del otro, cuidando el no repetir u omitir vértices.

Si tras la mutación y evaluación del individuo P_i^{t+1} este tiene una calificación peor que el individuo P_i^t entonces con una probabilidad del 10 % lo puede aún reemplazar, pero si es mejor entonces siempre lo reemplazará.

Mutación

Escogeremos otro intervalo aleatorio $[a, b] \subseteq \{0, 1, 2, \ldots, n-1\}$ e invertiremos el orden de las ciudades en el intervalo en P_i^{t+1} por cada posición i de P^{t+1}.

Esto es conveniente pues notemos que las adyacencias de la ruta que describe nuestra permutación únicamente cambia en dos lugares: en los extremos del intervalo $[a, b]$. La técnica de mutación ingenua intercambiaría la posición de dos ciudades, pero notemos que eso nos cambia ambas adyacencias de las dos ciudades, resultando en un cambio de 4 adyacencias.

Generalmente es conveniente realizar mutaciones pequeñas, pues de lo contrario estamos realmente realizando una búsqueda por fuerza bruta en lugar de aprovechar del algoritmo genético.

De acuerdo a esto último que mencionamos, la mutación tiene un porcentaje asociado de ocurrir. Para este solucionador en particular, estuvimos utilizando una probabilidad de mutación del 10 %.

Elitismo

Iremos acumulando a 5 individuos de élite, los cuales son los mejores individuos que hemos visto en toda la ejecución del programa.

Descripción del Programa Paralelo en CPU con OpenMP

Generamos poblaciones aisladas por cada procesador, a las cuales llamamos islas.

Esto tiene un fundamento heurístico, pues lo que ayuda a un algoritmo genético suele ser la disponibilidad de una variedad genética diversa. Al tener poblaciones que evolucionan por separado, aunque una población converja a un cierto grupo reducido de "genotipos", es muy poco probable que varias poblaciones aisladas tengan en conjunto los mismos genes.

Para "comunicar" a las islas, lo que realizamos fue un operador que nombramos remezclado de élite, el cual realiza un shuffle sobre los individuos de élite de las islas, para que en el proceso de selección pueda efectivamente tomar a buenos individuos con genes que evolucionaron prácticamente de manera independiente.

Otro cambio que realizamos para la implementación en paralelo fue que diseñamos nuestra clase para generación de números aleatorios, pues `rand` y `srand` nos causarían condiciones de carrera porque modifican estado interno.

La paralelización se realizó con OpenMP, que llamamos dos veces, una para ejecutar el algoritmo genético sobre cada isla, y la segunda vez para la generación de la población inicial.

Descripción del Programa Paralelo en GPU con CUDA

Para esta implementación seguimos a (Gaxiola Sánchez et al., 2014), donde similarmente a la implementación con OpenMP utilizamos la noción de islas para grupos de individuos, pero además incluimos una isla de élite la cual, como su nombre lo indica, contiene a todos los individuos de élite.

En este caso, el tamaño de cada población es decidida por el hardware, de modo que es máxima mientras que el requerimiento de memoria compartida es satisfactible por bloque.

Asímismo, el número de bloques es suficiente para llenar los Streaming Multiprocessors (SM) de la GPU para alcanzar una ocupación máxima, y además para que puedan haber al menos $2n$ individuos netamente. Esto puede ser sobreescrito, pues para este solucionador el nivel de paralelismo corresponde con el número de islas, i.e. bloques. Esto se realiza con la directiva `cudaOccupancyMaxPotentialBlockSizeVariableSMem`.

Además, el tamaño de la población será de al menos 4 veces el tamaño del warp, lo cual suele ser el mínimo recomendado para eficientar los accesos a memoria, la ocupación de los SM y el *scheduling* de tareas en el GPU. (Jones, 2022)

Con respecto a nuestra isla de élite, lo que cambiará con respecto a la implementación de OpenMP es que tendrá un único individuo que represente a cada bloque, escogido por reducción paralela sobre los hilos del mismo bloque al que cuente con mejor calificación.

Resultados

Ejecutamos las pruebas en las siguientes máquinas, usando `fedora` como la máquina para evaluar los solucionadores en CPU y `walle` como la computadora para las pruebas en GPU.

```
Parameter,Value
Hostname,fedora
OS Info,"Fedora Linux 37 (KDE Plasma), Kernel: 6.5.12-100.fc37.x86_64"
glibc Version,2.36
CPU Model,AMD Ryzen 7 PRO 2700U w/ Radeon Vega Mobile Gfx
CPU Vendor,AuthenticAMD
CPU Flags,3dnowprefetch abm adx aes aperfmperf apic arat avic avx avx2 bmi1 bmi2 bpext clflush clflushopt cl
Logical Cores,8
Physical Cores,4
Base Frequency (MHz),1600.0
Boost Frequency (MHz),2200.0
Chipset,83DA
L1d Cache,128 KiB (4 instances)
L1i Cache,256 KiB (4 instances)
L2 Cache,2 MiB (4 instances)
L3 Cache,4 MiB (1 instance)
OpenMP Version,4.5
Total RAM (GB),14.55
```

```
CUDA Capable Device,N/A
CUDA Version,N/A
AVX Supported,Yes
SSE Supported,Yes
```

```
Parameter,Value
Hostname,WallE
OS Info,"Ubuntu 22.04.5 LTS, Kernel: 5.15.167.4-microsoft-standard-WSL2"
glibc Version,2.35
CPU Model,AMD Ryzen 5 2600 Six-Core Processor
CPU Vendor,AuthenticAMD
CPU Flags,3dnowprefetch abm adx aes apic arat avx avx2 bmi1 bmi2 clflush clflushopt clzero cmov cmp_legacy c
Logical Cores,12
Physical Cores,6
Base Frequency (MHz),3.2
Boost Frequency (MHz),3.7
Chipset,N/A
L1d Cache,192 KiB (6 instances)
L1i Cache,384 KiB (6 instances)
L2 Cache,3 MiB (6 instances)
L3 Cache,8 MiB (1 instance)
OpenMP Version,4.5
Total RAM (GB),15.60
CUDA Capable Device,Yes
NVIDIA GPU Model,NVIDIA GeForce GTX 1060 6GB
Global Memory,6.00 GB
Shared Memory per Block,49152 bytes
Warp Size,32
Streaming Multiprocessor Count,20
Compute Capability,6.1
CUDA Core Count,1280
PCIe Bus Bandwidth,<15.76GB
GPU Memory Bus Width, 192-bit
CUDA Version,12.5
AVX Supported,Yes
SSE Supported,Yes
```

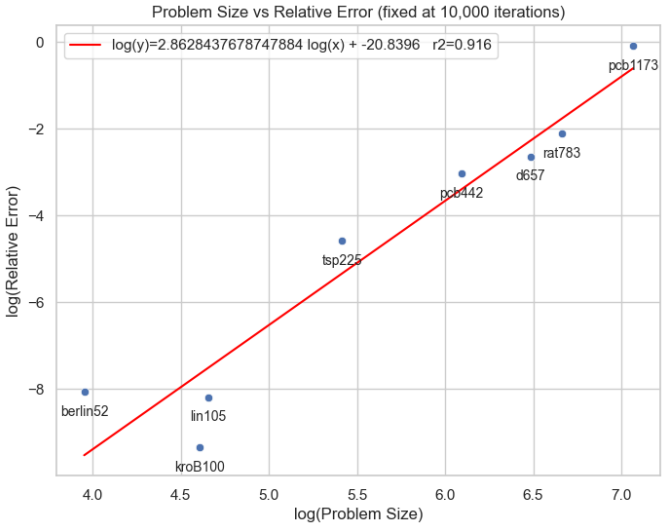
Escogimos las siguientes instancias de TSPLIB95 pues sus tamaños nos parecieron relevantes, pero no ejecutamos todos los problemas con todos los solucionadores pues el tiempo de cómputo lo prohibía por limitaciones de tiempo en cuanto a la realización del proyecto.

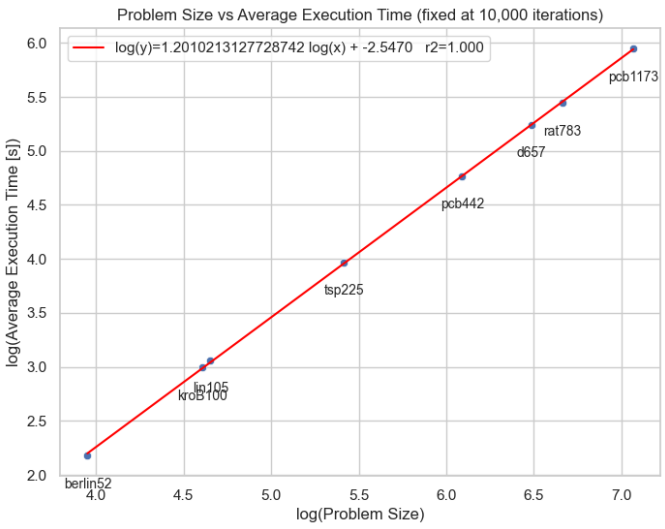
Los datos de las ejecuciones resumidas se pueden revisar lanzando el notebook de Jupyter.

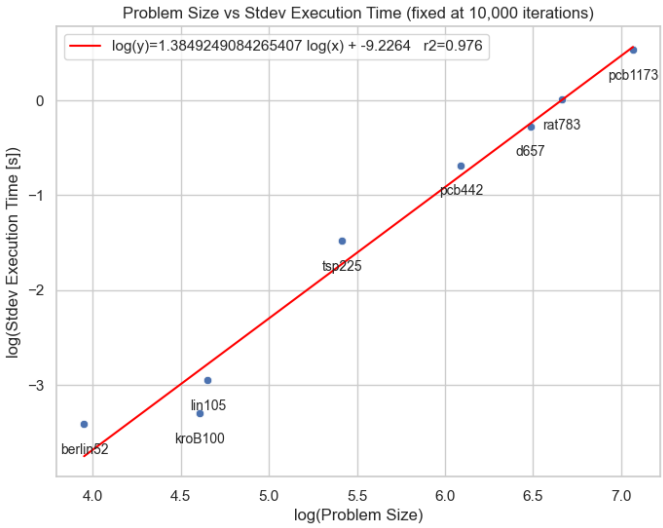
Teniendo instalado `pip` y `python3`, basta con estar sobre el directorio `/analysis` y ejecutar `./launch`. Ahí mismo se realiza una pequeña discusión a manera de análisis de los resultados.

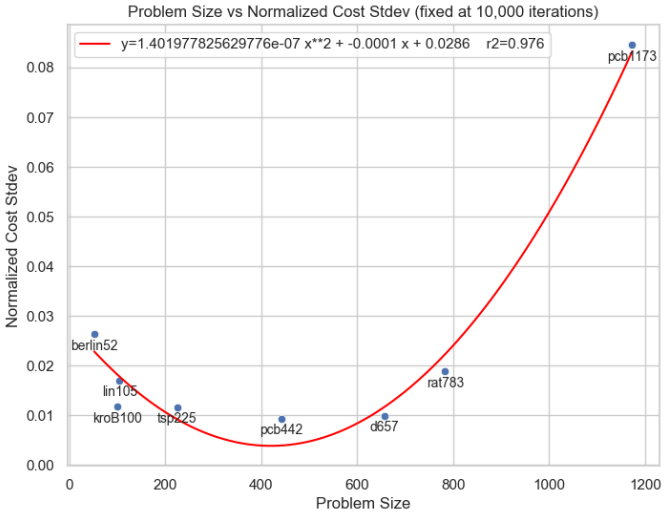
Para evitar duplicar toda la información del notebook, en este reporte dejaremos algunos datos importantes e interesantes de lado, y nos enfocaremos únicamente en los resultados de tiempos de ejecución. Recomendamos explorar el notebook con mayor detalle para ver los resultados completos.

> **Máximo esfuerzo GPU**

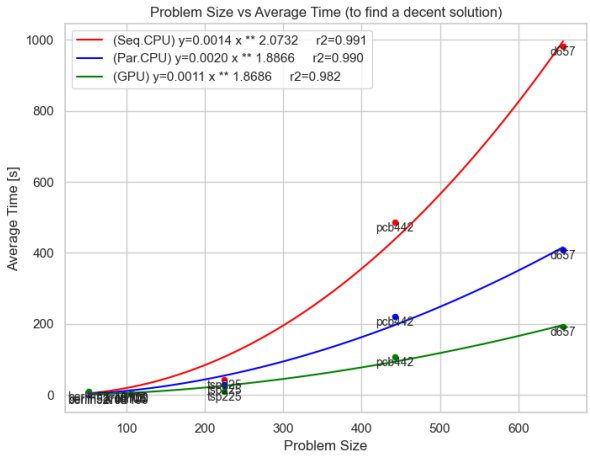


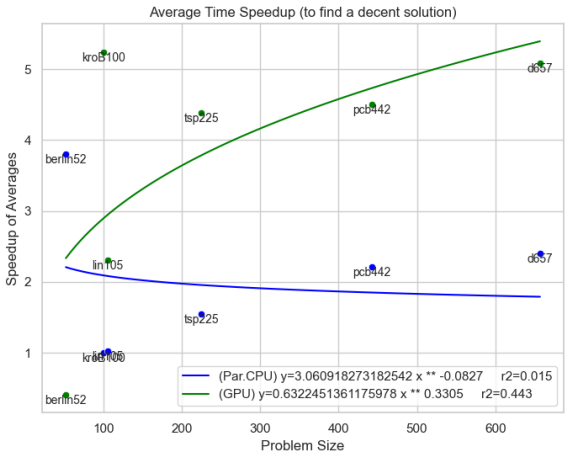


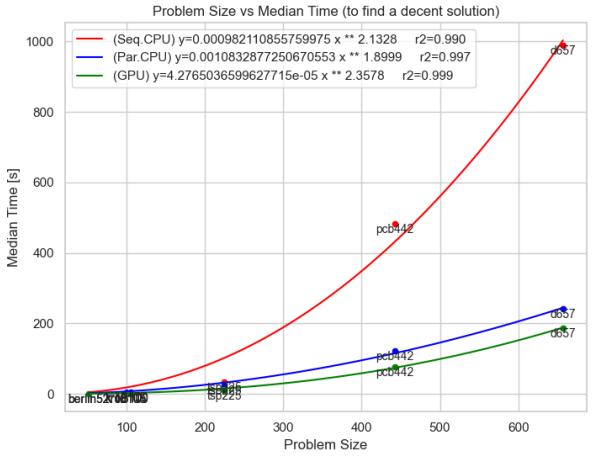


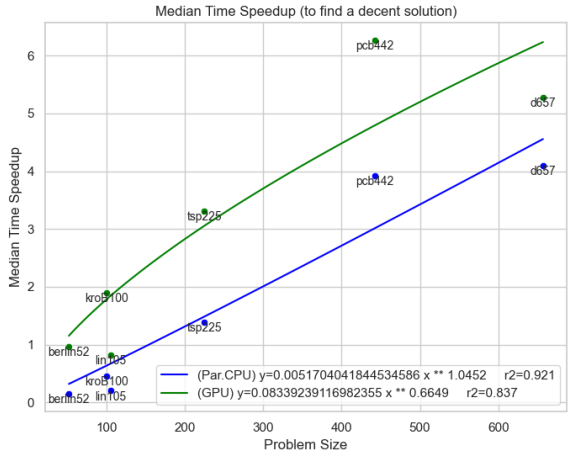


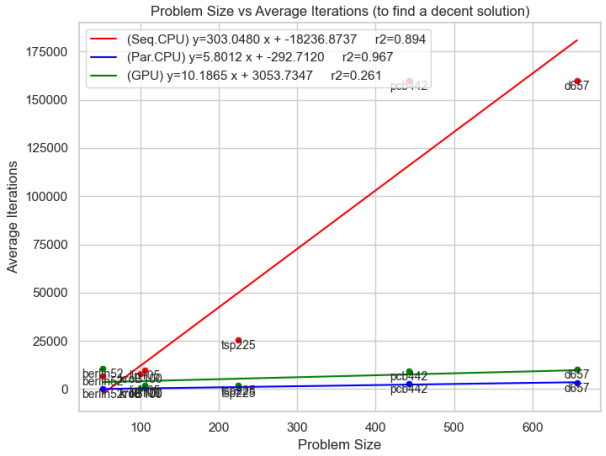
Punto de Corte

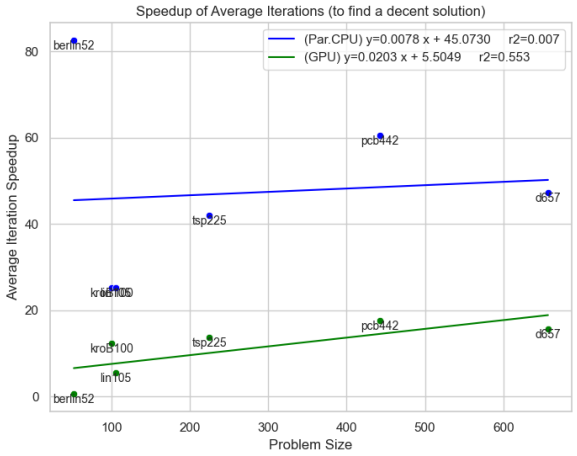


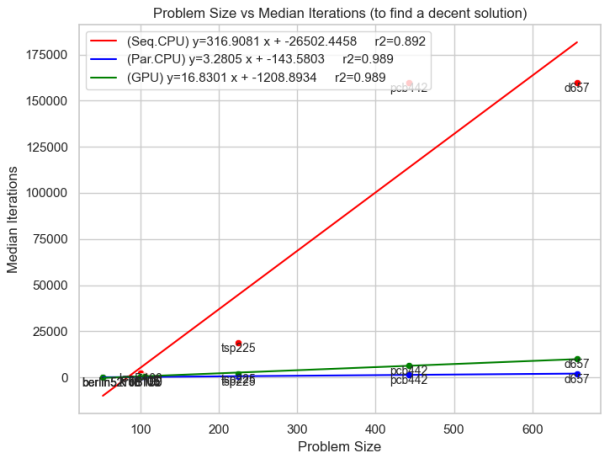


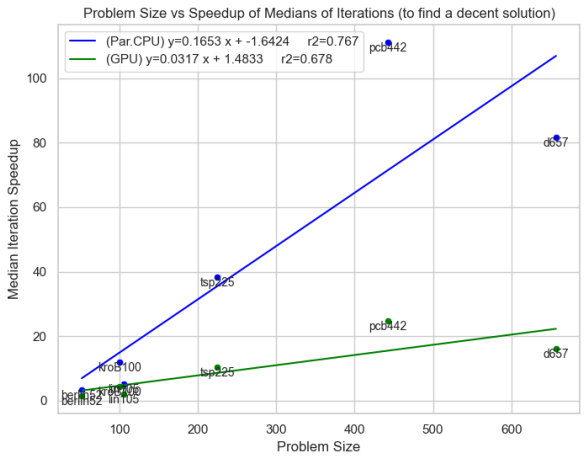












Análisis de Resultados

Igual que en la sección anterior, referimos al notebook realizado, pero compartimos algunas observaciones relevantes.

Una observación importante es para las gráficas en un espacio log-log, debemos interpretar rectas como polinomios con el

mismo grado que la pendiente. Esto es porque rectas en el espacio log-log se transforman en leyes de potencias:

$$\log y = m \log x + b \iff e^{\log y} = e^{m \log x + b} \iff y = e^{b} x^{m}$$

Esto es particularmente práctico para un análisis empírico de algoritmos pues únicamente incluye al término polinomial dominante y la constante oculta correspondiente.

Saltándonos directamente a las ejecuciones por punto de corte, pues nos interesa comparar la parelelización, podemos notar que en general para los tiempos: la implementación en GPU es la más rápida, después la implementación en CPU en paralelo, y finalmente en secuencial.

A pesar de ello, entre las implementaciones paralelas el speedup de la versión en CPU se acerca mucho más al speedup lineal, mientras que la GPU con más de 1000 CUDA cores no alcanza ni el $1\,\%$ de eficiencia.

Además, al notar que la versión paralela en CPU nos reduce en cerca de 80 veces el número de iteraciones que realiza la simulación, entonces podemos argumentar que si optimizamos más el tiempo que toma una única iteración podríamos fácilmente obtener un speedup superlineal, lo cual es reportado en la bibliografía. (Alba & Luque, 2005)

Esto nos quiere decir que también nuestra implementación en GPU debería aún poder verse mucho más eficiente, pues el speedup es extremadamente bajo. Sospechamos que tiene que ver con el costo de la instrumentación, que involucra además constantes movimientos por el bus PCIe entre GPU y CPU.

Así, nuestra versión con OpenMP pareciera ser escalable, pero como vemos en alguna de las figuras, el comportamiento del tiempo es tan errático que seguramente depende de la configuración particular geométrica del problema en lugar de únicamente su tamaño. Por lo tanto, habría que realizar aún más mediciones.

Otra consideración es que teníamos un techo en el número de iteraciones, lo que nos sugiere que el speedup real podría ser mucho mayor, pero se cortó el experimento en un techo alto antes de ello. Esto quiere decir que nuestro speedup es una cota inferior al speedup que observaríamos realmente.

Conclusiones

En cualquier caso definitivamente fue mejor contar con el cómputo paralelo para resolver estos problemas con nuestro algoritmo genético. A pesar del tamaño de los problemas, además en todo caso los resolvimos rápidamente en un tiempo razonable para la operación pequeña y mediana.

A pesar de ello, notamos que queda mucho espacio para optimización, lo cual representa una oportunidad real para seguir iterando sobre esta solución, que toma una cantidad de tiempo y recursos computacionales totalmente razonable para su propósito.

Sería interesante analizar cómo escalan estas soluciones al hacer uso de cómputo distribuido, pero para la meta que nos habíamos planteado, el resultado fue satisfactorio y con un amplio espacio para seguir iterando y mejorando.

Apéndices

Manual de Usuario

TSP Paralelo

Esta es una herramienta para utilizar y comparar distintos solucionadores para el TSP Euclidiano en 2D.

Documentación

Compilación

Se requiere `lualatex` y varios paquetes de LaTeX. Si usas Linux, se recomienda buscar en el repositorio de tu distribución un paquete como `texlive-all` o `texlive-full`.

Dentro del directorio `latex` de cada documentación, si estás en Linux, puedes simplemente ejecutar:

```
./run
```

Índice

- Propuesta del proyecto: `docs/propuesta/latex`
- Informe del proyecto: `docs/reporte/latex`

Compilación

Se asume que estás en Linux, ya que actualmente no brindamos soporte para otras plataformas. Como mínimo, necesitarás una CPU `x86_64` y el compilador `g++`, que a menudo se incluye en un paquete de tu distribución llamado `build-essential` o `base-devel`.

Punto de entrada

Primero debes compilar el programa que configura la ejecución. Este programa se encuentra en `code/src/main/`. Puedes modificar el script `build` a tu gusto y luego ejecutar la compilación con:

```
./build
```

Si ya tienes solucionadores compilados, puedes ejecutar el programa con:

```
./run
```

Solucionadores

Son bibliotecas dinámicas que buscan las soluciones candidatas para las instancias del TSP Euclidiano 2D. Funcionan como programas independientes del ejecutable `main`, ya que pueden requerir soporte de hardware y/o software especializado y permiten depuración casi en tiempo real al recargarlas en ejecución. Es decir, puedes ejecutar el binario `./main` y luego recompilar cualquier solucionador cuando quieras hacer cambios sin reiniciar el programa. El único requisito es descargar la biblioteca antes de compilar, cambiando a otro solucionador.

Dummy El solucionador `dummy` devuelve los nodos en el orden recibido; es solo para depuración. Para compilarlo, ingresa a `code/src/dummy/` y ejecuta:

```
./build
```

Genético secuencial El solucionador `cpuseq` implementa un algoritmo genético en ejecución secuencial y sirve como punto de comparación. Para compilarlo, ingresa a `code/src/cpu/sequential/` y ejecuta:

```
./build
```

Genético paralelo El solucionador `cpupar` aplica la técnica fork-join al mismo algoritmo que `cpuseq`. Sirve como punto de comparación para una implementación paralela ingenua en CPU. Para compilarlo es necesario instalar OpenMP, pero si ya cuentas con el archivo de objeto compartido (`.so`), no lo necesitas para ejecutar. Para compilarlo, ve a `code/src/cpu/parallel/` y ejecuta:

```
./build
```

Genético GPU El solucionador `gpu` implementa un algoritmo genético adaptado a la arquitectura GPU y, por lo tanto, requiere hardware y software especializados. Para compilarlo es necesario instalar el CUDA Toolkit, aunque no requieres una GPU de inmediato: basta con tener disponible y funcional el compilador `nvcc`. Para ejecutarlo, necesitas una GPU NVIDIA de arquitectura Pascal o superior; sin embargo, si ya dispones del archivo compartido (`.so`), no requieres el CUDA Toolkit. Para compilarlo, ingresa a `code/src/gpu/` y ejecuta:

```
./build
```

Ejecución

Por defecto, tras compilar el programa principal y algunos solucionadores, sus archivos ELF se ubican en `code/data/`. Puedes ejecutar `./main` desde ahí o entrar a `code/src/main/` y ejecutar `./run`.

Dentro del programa, escribe:

```
help
```

para ver las instrucciones. Hay 10 comandos, cada uno admite 0 o 1 argumento:

- `help` Muestra la ayuda.
- `problem` Carga un problema en `code/data/tsp/*.tsp`.
- `solver` Carga un solucionador, descargando el activo previamente.
- `iterations` Establece el número máximo de iteraciones por ejecución.
- `executions` Establece la cantidad de ejecuciones desde cero.
- `cutoff` Establece el umbral de costo de solución.
- `parallelism` Define el número de "unidades" paralelas.
- `config` Muestra la configuración actual para lanzar el solucionador.
- `exit` Sale del programa, también con Ctrl-D.
- `run` Ejecuta el solucionador con la configuración actual.

Por ejemplo, `problem a280` carga el problema del archivo `code/data/tsp/a280.tsp`, y `solver dummy` carga la biblioteca `code/data/dummy.so`. La diferencia entre `iterations` y `executions` es que una iteración suele depender de la anterior, mientras que cada ejecución reinicia el solucionador desde cero, sin considerar las previas. Realizar múltiples ejecuciones permite obtener la media, la desviación estándar muestral u otras métricas estadísticas en configuraciones idénticas.

Al ejecutarse, el programa muestra información de la ejecución generada por los solucionadores y la guarda en `code/data/logs/` como un archivo JSON con marca de tiempo.

Agregar solucionadores

Crea una biblioteca dinámica que incluya, o al menos cumpla con, la interfaz definida en `code/src/include/solver.h`. Asegúrate de implementar todas las funciones. Agrega tu archivo `.so`, preferiblemente con un nombre corto y único, en `code/data/`.

Eso es todo. Así de simple.

Ahora puedes cargar, por ejemplo, `my_solver.so` con:

```
solver my_solver
```

Liga al repositorio

https://github.com/srp-mx/parallel-tsp

Referencias

Gaxiola Sánchez, L. N., Tapia Armenta, J. J., & Díaz Ramírez, V. H. (2014). Parallel Genetic Algorithms on a GPU to Solve the Travelling Salesman Problem. *Difu100ci@, Revista de difusión científica, ingeniería y tecnologías*, *8*(2), 84-90. http://difu100cia.uaz.edu.mx/index.php/difuciencia/article/view/145

Merino Trejo, A. O. (2023). *Algunas heurísticas aplicadas al problema del agente viajero* [Licenciatura]. Universidad Nacional Autónoma de México [Facultad de Ciencias, UNAM]. http://132.248.9.195/ptd2023/septiembre/0846778/Index.html

Alba, E., & Luque, G. (2005). Measuring the Performance of Parallel Metaheuristics. En *Parallel Metaheuristics* (pp. 43-62). John Wiley & Sons, Ltd. https://doi.org/https://doi.org/10.1002/0471739383.ch2

Michigan State University. (s.f.). *Why Logistics Is Fundamental to Supply Chain Success*. Michigan State University. Consultado el 12 de junio de 2025, desde https://www.michiganstateuniversityonline.com/resources/supply-chain/logistics-fundamental-to-supply-chain-success/

Warnier, A. (2024). *How Curri's Advanced Logistics Help Small Businesses Compete*. Consultado el 12 de junio de 2025, desde https://www.curri.com/article/advanced-logistics-for-small-business

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd). MIT Press.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press.

Padilla Martínez, I. R. (2019). *Algoritmos genéticos y extinciones masivas en el problema del agente viajero* [Tesis de licenciatura]. Universidad Nacional Autónoma de México.

Tavera, F. M. (2018). *Un algoritmo genético para tratar el problema de rutas de vehículos* [Tesis de licenciatura]. Universidad Nacional Autónoma de México.

Alba, E., & Tomassini, M. (2002). Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, *6*(5), 443-462. https://doi.org/10.1109/TEVC.2002.802452

Chapman, B., Jost, G., & van der Pas, R. (2008). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press.

Sanders, J., & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.

Jones, S. (2022, 11 de mayo). *GTC 2022 - How CUDA Programming Works - Stephen Jones, CUDA Architect, NVIDIA*. Consultado el 12 de junio de 2025, desde https://www.youtube.com/watch?v=QQceTDjA4f4