UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

INGENIERÍA EN ...

Desarrollo y evaluación de una implementación de altas prestaciones de la metaheurística de optimización Nizar

Autor

Guillermo Dalda del Olmo

Directores

Nicolás Calvo Cruz

Eva Martínez Ortigosa

Aqui se puede incluir nombre y logo del Departamento responsable del proyecto

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 16 de Junio de 2025

3

Alternativamente, el logo de la UGR puede sustituirse / complementarse con uno específico del proyecto

UNIVERSIDAD DE GRANADA

Desarrollo y evaluación de una implementación de altas prestaciones de la metaheurística de optimización Nizar

Autor
Guillermo Dalda del Olmo

Directores
Nicolás Calvo Cruz
Eva Martínez Ortigosa

Desarrollo y evaluación de una implementación de altas prestaciones de la metaheurística de optimización Nizar

Guillermo Dalda del Olmo

Palabras clave: palabra clave1, palabra clave2, palabra clave3, ...

Resumen

Poner aquí el resumen.

Desarrollo y evaluación de una implementación de altas prestaciones de la metaheurística de optimización Nizar

Guillermo Dalda

Keywords: Keyword1, Keyword2, Keyword3, ...

Abstract

Write here the abstract in English.

Yo, **Guillermo Dalda del Olmo**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 77037106D, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Guillermo Dalda del Olmo

Granada a 16 de Junio de 2025.

D. **Nombre Apellido1 Apellido2 (tutor1)**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

D. **Nombre Apellido1 Apellido2 (tutor2)**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Título del proyecto, Subtítulo del proyecto***, ha sido realizado bajo su supervisión por **Nombre Apellido1 Apellido2 (alumno)**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 201 .

Los directores:

Nombre Apellido1 Apellido2 (tutor1) **Nombre Apellido1 Apellido2 (tutor2)**

Agradecimientos

En primer lugar, a mis tutores Nicolás Calvo Cruz y Eva Martínez Ortigosa, qué han hecho de este proyecto una gran experiencia de aprendizaje, amena y sin alterar el transcurso del resto de mis asignaturas. Han estado siempre disponibles y dispuestos a resolver dudas o proporcionar la ayuda necesaria para completar los objetivos, apoyándome en todo momento y procurando el mejor desarrollo posible del proyecto.

A mis padres, que hacen esto posible.

A aquellos profesores que durante mi vida me han motivado a aprender, pensar y razonar, además de impulsarme a dar lo mejor de mí mismo; pues esas facultades son las que me han llevado hasta aquí.

Contenidos

Figuras

Tablas

Introducción

La optimización consiste en encontrar la mejor solución para un problema específico. Esto se consigue ajustando los parámetros del problema para llegar a la solución máxima o mínima, dependiendo de cual sea el objetivo. Por ejemplo, se pueden ajustar los parámetros del modelo matemático de un proceso de fabricación para maximizar las ganancias; o a la hora de diseñar un soporte de acero que debe cargar con una cantidad de peso, minimizar la cantidad de acero utilizado para así minimizar costes. Son perspectivas diferentes que al final buscan la solución óptima (de ahí surge la optimización). Los optimizadores suelen utilizarse para hallar el mínimo porque generalmente el objetivo es reducir costes, igualmente los problemas o funciones se pueden adaptar fácilmente según se vaya a utilizar un algoritmo que quiere encontrar el mínimo o el máximo. Pero por esta razón, a estas funciones o problemas de los cuáles se quiere buscar el óptimo se les llama función objetivo, función fitness o función coste(1).

En este documento se trata un algoritmo metaheurístico basado en poblaciones llamado NOA (Nizar Optimization Algorithm)(2), que minimiza el valor de la función objetivo y devuelve **los valores de los parámetros o variables** que proporcionan el óptimo de dicha función. Esta es la solución que deben proporcionar todos los optimizadores, ya que no sirve de nada conocer el coste mínimo de un producto sin también saber la cantidad requerida de cada material que permite alcanzar este valor óptimo y realmente minimizar costes.

A lo largo de las siguientes secciones hablaré de la relevancia de la optimización, en concreto del algoritmo NOA; los aspectos que se van a trabajar de este, por qué y cómo.

1.1. Relevancia y motivación

Muchos problemas del mundo real son solubles de forma óptima mediante técnicas de optimización clásicas; pero estas dejan de ser aptas cuando la complejidad requiere una cantidad de tiempo inviable para hallar el óptimo o en su lugar se da con un óptimo local. Para estos casos se recurre a los métodos heurísticos, que incluyen a los algoritmos metaheurísticos de optimización como NOA, y que fueron ideados para proporcionar soluciones razonablemente buenas, si no las mejores en un tiempo de computación razonable. Aunque los métodos heurísticos no garantizan la optimalidad, en ciertos casos son la única opción para aproximarse a resultados concretos; de hecho, han sido ampliamente utilizados en problemas empresariales, deportivos, ambientales, estadísticos, médicos y de ingeniería qué se consideraban difíciles de resolver si no(3). Estos problemas de optimización no son situacionales o aislados, surgen continuamente, sobre todo en campos como la Arquitectura, las Ingenierías y otras Ciencias Aplicadas(4).

Aparte del interés intrínseco que los algoritmos de optimización puedan tener, NOA destaca por:

- Su reciente aparición, publicado originalmente en Septiembre de 2023 y por tanto es difícil que ya esté obsoleto o desactualizado respecto a tendencias modernas.

- Su uso simplificado, sin parámetros específicos.
- Sus resultados experimentales, superior a los otros 15 algoritmos con los que se compara tanto en funciones sintéticas sin restricciones como en problemas de ingeniería conocidos con restricciones.
- Sus nuevas técnicas, que mejoran la exploración y explotación del espacio de búsqueda de forma equilibrada.

Estas últimas son las contribuciones que el artículo original presenta como relevantes y a tener en cuenta; pero debido a que no se adjunta ningún programa con el que contrastar y confirmar las declaraciones anteriores y tampoco existe ninguna implementación pública que haya podido encontrar, la razón principal que llama a trabajar con NOA es la implementación y comprobación de su funcionalidad. Aparte, se afirma que la experimentación se realizó sobre una implementación en Matlab; por lo que existe margen de mejora respecto a la eficiencia con el uso de técnicas de computación de altas prestaciones.

De estos aspectos del algoritmo surgen los objetivos de este trabajo.

1.2. Objetivos

Se quiere complementar el trabajo realizado dando una implementación cuyo funcionamiento esté verificado a través de los resultados dados en *Nizar optimization algorithm…*(2), proporcionando un enfoque mayor hacia la eficiencia y el aumento de prestaciones; de modo que se va a implementar utilizando un lenguaje más eficiente, C, que permita el uso de técnicas de computación de altas prestaciones y comprobar la escalabilidad del algoritmo. De manera que estos serían los objetivos a alcanzar durante el trabajo:

- Proporcionar una implementación del algoritmo; pues los propios autores del artículo original no lo han hecho.
- Realizar la implementación del algoritmo en C; pues permite programación a más bajo nivel para mejorar el rendimiento del programa y también aumentar la eficiencia gracias las herramientas para la paralelización que proporciona.
- Comprobar el correcto funcionamiento de la implementación replicando la experimentación sobre 10 de las funciones benchmark y 2 de los problemas de ingeniería reales y contrastar los resultados; pues se debe asegurar que la implementación del algoritmo se comporta de forma correspondiente a tal y como se describe en el artículo original(2) o habría una inconsistencia entre los resultados obtenidos y los proporcionados.
- Realizar una implementación paralela utilizando pthreads; pues de esta forma se puede mejorar el rendimiento y la escalabilidad del algoritmo potencialmente haciéndolo válido para computación de altas prestaciones.
- Realizar una implementación paralela utilizando MPI; pues permite paralelismo real mejorando el rendimiento y la escalabilidad del algoritmo específicamente en sistemas de computación de altas prestaciones.
- Comprobar la aceleración de la versión de C sobre Matlab; pues teóricamente debería tener un mayor rendimiento, pero hay que comprobarlo empíricamente.

- Comprobar la aceleración que proporciona el paralelismo y su efecto en los resultados; pues la sobrecarga del paralelismo y el diseño de ciertos algoritmos causan a menudo que empeore el rendimiento, y en el caso de un optimizador puede también deteriorar las soluciones y necesitar más evaluaciones para encontrar el óptimo.

Para cumplir estos objetivos se ha llevado a cabo una planificación no estricta, y se han ido completando como se describe en el siguiente apartado.

1.3. Planificación

El desarrollo del trabajo ha llevado unas 286 horas en total y se ha realizado de forma progresiva durante aproximadamente 35 semanas, las cuáles se pueden dividir entre períodos de trabajo entre reuniones con los directores. Estos períodos no son uniformes ni tampoco el trabajo u horas dedicadas en ellos.

1ª reunión. Lunes, 14 de octubre 2024. Duración aproximada de 1 hora.

Avances realizados en este período:

- Lectura y estudio de *Nizar optimization algorithm…*(2).
- Documentación y preparación sobre optimización, incluyendo el estudio de un optimizador simple y su implementación.
- Comienzo de implementación de la función main.

Este período consta de unas 19 semanas y se invirtieron aproximadamente 30 horas; equivalente a 1 hora y media a la semana.

2ª reunión. Lunes, 24 de febrero 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Implementación de las fases del algoritmo.

Este período consta de 2 semanas y se invirtieron aproximadamente 13 horas; equivalente a 6 horas y media a la semana.

3ª reunión. Lunes, 10 de marzo 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Implementación de la función benchmark *sphere*.
- Contraste de implementación entre la versión de C y la de Matlab.

Este período consta de 2 semanas y se invirtieron aproximadamente 20 horas; equivalente a 10 horas a la semana.

4ª reunión. Lunes, 24 de marzo 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Corrección de errores en la implementación.
- Implementación de otras 4 funciones benchmark.

Este período consta de 2 semanas y se invirtieron aproximadamente 16 horas; equivalente a 8 horas a la semana.

5ª reunión. Lunes, 7 de abril 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Experimentación y verificación de resultados entre el programa de C y el de Matlab con las primeras 5 funciones sintéticas.
- Corrección de errores.
- Reorganización de los ficheros del programa.

Este período consta de unas 3 semanas y se invirtieron aproximadamente 20 horas; equivalente a unas 6 horas y media a la semana.

6ª reunión. Miércoles, 30 de abril 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Implementación de las otras 5 funciones sintéticas y de los 2 problemas de ingeniería reales.
- Implementación de *scripts* para compilación, ejecución y experimentación.
- Implementación de *pthreads* sobre la inicialización de la población.

Este período consta de unas 2 semanas y se invirtieron aproximadamente 28 horas; equivalente a 14 horas a la semana.

7ª reunión. Lunes, 12 de mayo 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Experimentación y verificación de resultados entre el programa de C y el de Matlab con las 7 funciones benchmark restantes.
- Implementación de *pthreads* sobre el cuerpo del algoritmo.

Este período consta de 2 semanas y se invirtieron aproximadamente 16 horas; equivalente a 8 horas a la semana.

8ª reunión. Lunes, 26 de mayo 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Implementación de un generador de aleatorios con acceso seguro para las secciones paralelas, requiriendo una versión de Windows y otra de Linux.
- Optimización de la implementación de *pthreads*, implementando dos versiones que actualizan las generaciones de forma distinta.
- Experimentación y comparación entre las versiones paralelas.
- Medición de tiempos en secuencial del programa en C y en Matlab.

Este período consta de unas 2 semanas y se invirtieron aproximadamente 28 horas; equivalente a 14 horas a la semana.

9ª reunión. Viernes, 6 de junio 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Unificación del programa de Windows y Linux utilizando el generador de números aleatorios Mersenne Twister
- Corrección de errores y perfilado de las versiones paralelas.
- Experimentación y comparación de las versiones paralelas y secuencial alterando el tamaño de la población.

Este período consta de 3 días y se invirtieron aproximadamente 16 horas; equivalente a unas 5 horas al día.

10ª reunión. Lunes, 9 de junio 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Medición de tiempos de la versión paralela para distintos tamaños de población.
- Búsqueda de un problema de mayor coste computacional.
- Elaboración de la memoria.

Este período consta de 3 días y se invirtieron aproximadamente 28 horas; equivalente a unas 9 horas al día.

11ª reunión. Jueves, 12 de junio 2025. Duración aproximada de 1 hora.

Avances realizados en este período:

- Implementación de carga artificial para un problema real y medición de tiempos en este problema para distintos tamaños de población.
- Finalización de la memoria

Este período consta de 4 días y se invirtieron aproximadamente 60 horas; equivalente a unas 15 horas al día.

1.4. Recursos

En la realización del proyecto se ha hecho uso de las siguientes herramientas; diferenciando entre el hardware y software requeridas para cada apartado.

1.4.1. Hardware

La implementación y comprobación del funcionamiento del algoritmo se ha realizado en un sistema con Intel Core i3-1005G1 1.20GHz y 8GB de memoria principal. También se ha utilizado este hardware para la comparación entre las versiones secuenciales que utilizan distinto software.

La experimentación posterior y análisis del algoritmo paralelo respecto al secuencial se ha realizado en el servidor de genmagic de la ETSIIT.(Especificaciones?)

1.4.2. Software

La versión en el lenguaje C se ha implementado con el editor Visual Studio Code, desde un SO Windows 10. La comprobación del funcionamiento se ha realizado con este software y se ha contrastado con una implementación de Matlab utilizando MATLAB R2024b. Para estas pruebas se han implementado y utilizado 10 funciones sintéticas y 2 problemas de ingeniería reales en ambos lenguajes.

La experimentación posterior y análisis del algoritmo paralelo respecto al secuencial se ha realizado sobre un sistema que funciona en Linux.

Se han utilizado lenguajes de *script* para automatizar la compilación, ejecución y toma de resultados de los programas en C:

- En Windows: Mediante *Powershell scripts*.
- En Linux: Mediante *bash scripts*.

1.4.3 Otros

Aparte de librerías del estándar de C, se ha utilizado el algoritmo de generación de números aleatorios Mersenne Twister(5) por su mejor calidad y eficiencia respecto a generadores del estándar; también por la posibilidad de crear generadores independientes para cada hebra en una versión con *pthreads* con accesos seguros. De hecho, los generadores seguros disponibles en el estándar para Linux(rand_r()) difieren de los disponibles en Windows(rand_s), por lo que inicialmente se tenían dos implementaciones del algoritmo, una para Windows(para poder ejecutarlo localmente) y otra para Linux(SO usado principalmente en entornos de paralelismo y computación de altas prestaciones).

Revisión bibliográfica sobre optimización

2.1. Tipos de optimización

Primero es importante considerar los factores que diferencian unos problemas de otros. Principalmente si están restringidos o no, si son unimodales o multimodales y si las funciones objetivo que los representan y sus restricciones(si las tienen) son lineales o no lineales.

Un problema restringido debe tener en cuenta otros resultados aparte del suyo propio, por ejemplo, si se está optimizando el coste de fabricación industrial de un producto es normal tener restricciones sobre las dimensiones y precisión con las que puede trabajar la maquinaria y por tanto invalide ciertas soluciones al problema. Mientras que en un problema no restringido solo importa el valor de la función objetivo como tal. Pero como es lógico problemas reales suelen tener restricciones debido a la cantidad de factores físicos que pueden afectarles; y los problemas sin restricciones suelen ser funciones sintéticas diseñadas para probar la efectividad de los optimizadores.

Los problemas unimodales solo tienen un óptimo, ya sea mínimo o máximo; esto simplifica la tarea de optimización, ya que no puede caer en un óptimo local.

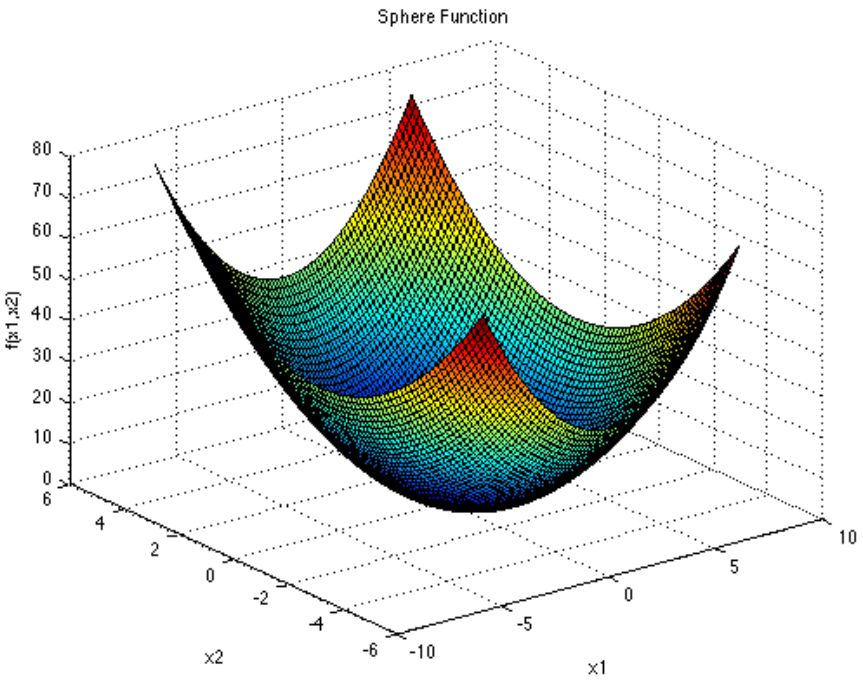

Figura 1: Ejemplo de un problema unimodal con un único mínimo global.

Los problemas multimodales tienen múltiples puntos críticos, siendo solo uno de ellos el óptimo global; esto dificulta el proceso de optimización, pues se tiene que evitar considerar un óptimo local como el global y explorar todas las opciones.

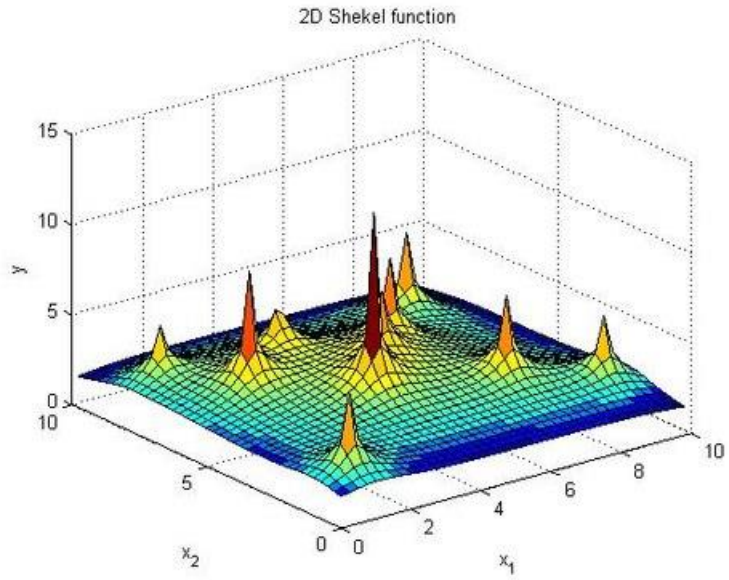

Figura 2: Ejemplo de un problema multimodal con múltiples máximos locales

Cuando se habla de problemas lineales, su función objetivo y sus restricciones son funciones lineales. Estos siempre están restringidos pues no tienen máximo o mínimo, ya que crecen o decrecen indefinidamente y sus variables siempre tenderían a infinito o a menos infinito. En cambio, problemas no lineales pueden tener múltiples soluciones locales, que son óptimas para una subregión del espacio de soluciones, pero para toda la región la solución se define como óptimo global; estos pueden estar restringidos o no sin que se trivialice el resultado.

Una vez definidos estas características sobre los problemas de optimización, se pueden plantear varios métodos para encontrar máximos o mínimos:

- El uso de cálculo: Aunque encuentra de forma efectiva los valores mínimo y máximo, la complejidad del álgebra y la cantidad de coeficientes diferenciales reduce su utilidad a los problemas más simples y se vuelve inviable para la mayoría de aplicaciones prácticas.
- Fuerza bruta: Para dos o tres variables, se pueden probar todas las combinaciones con una precisión baja y se encontrarían soluciones relativamente rápido, pero difícilmente las soluciones óptimas. Además, el aumento de la complejidad del problema o de la precisión con la que se quiere buscar rápidamente lleva los tiempos de cómputo exceder una cantidad aceptable, por lo que es inviable en la mayoría de casos.
- Métodos de gradiente: Pueden resolver problemas complejos de forma eficiente, siendo métodos de búsqueda directos viables a diferencia de los anteriores; pero se estancan en óptimos locales a menudo, siendo realmente poco eficaces para encontrar óptimos globales a no ser que haya pocos o ningún óptimo local.

- Algoritmos de optimización inspirados en la naturaleza: Generalmente realizan dos pasos; una exploración aleatoria del espacio de soluciones y luego la convergencia hacia el óptimo global. Estos pasos se repiten hasta que se obtiene la solución óptima. Este es el tipo que se va a tratar.

2.2. Algoritmos basados en la naturaleza

Los algoritmos metaheurísticos basados en poblaciones se están volviendo poderosos métodos para resolver muchos problemas de optimización complejos del mundo real. Estos algoritmos utilizan diferentes metaheurísticas y exploran estrategias únicas, con sus ventajas y desventajas respecto a cómo aproximan las soluciones y resuelven ciertos tipos de problemas. Algunos se ven inspirados por la mejora gradual de las especies gracias a la evolución, herencia, genética y selección natural. Otros se inspiran y simulan el comportamiento de la naturaleza, ya sea de enjambres, plantas, humanos u otros seres vivos. Hay algoritmos basados en las leyes científicas en que forman la física, química, geología, etc. Al resto de algoritmos se les clasifica como “Otros” (2).

A pesar de que la mayoría de algoritmos acaban siendo muy distintos, comparten a su vez una serie de parámetros generales y parámetros específicos. Los parámetros generales incluyen el tamaño de la población, cantidad de evaluaciones, ciclos, dimensiones, etc. Mientras que los parámetros específicos son herramientas de control y configuración específicas para cada algoritmo que permiten aumentar o reducir el rendimiento del mismo; el problema es que son indispensables para su funcionamiento y no hay forma de predecir qué parámetros ni bajo qué circunstancias pueden mejorar o empeorar el rendimiento. La tarea de afinar todos los parámetros específicos de in algoritmo para obtener los mejores resultados puede acabar siendo un problema en sí, dificultando su uso y ralentizando el trabajo de ingenieros e investigadores que intentan utilizar estos algoritmos. Por eso se anima al desarrollo de algoritmos que no requieran parámetros específicos, como NOA.

Un algoritmo de optimización tiene dos procesos necesarios:

- Exploración: Es el proceso de explorar posibles soluciones en un área de búsqueda amplia.
- Explotación: Es el proceso de encontrar nuevas soluciones aprovechando las soluciones actuales.

Sin un balance apropiado entre estos dos procesos la búsqueda se puede estancar en un óptimo local o tardar demasiado en converger hacia el óptimo global.

Estas debilidades que se han comentado son las que motivaron las estrategias de NOA que pretenden equilibrar la exploración y explotación, junto con la ausencia de parámetros específicos.

NOA

NOA(Nizar Optimization Algorithm) se ha diseñado para resolver problemas restringidos y no restringidos sin ningún parámetro específico, solo con parámetros generales. Siendo un algoritmo metaheurístico basado en poblaciones, se genera una población aleatoria inicial y luego se va actualizando cada generación con nuevas soluciones mejores generadas a partir de la anterior mediante dos técnicas nuevas:

- Mapeos efectivos: Son seis mapas de vectores que se clasifican en dos tipos:
    - Mapeos de combinaciones: que construyen un vector combinando dos vectores de entrada utilizando tres mecanismos diferentes.
    - Mapeos de transformaciones: que devuelven el vector de entrada modificado o sustituido por otro en función de variables condición aleatorias y utilizando tres mecanismos diferentes.
- Puntos efectivos: Se construyen haciendo uso de individuos de la generación anterior y de los mapeos efectivos. Con estos se generan nuevas soluciones para la siguiente generación.

Estos mecanismos son los que proporcionan la variedad y calidad de las soluciones. Esta variedad es la que le da al algoritmo una propiedad importante, la capacidad de salir de un óptimo local.

Un buen algoritmo de optimización debe tener un balance entre exploración y explotación; por eso el proceso de aprendizaje de NOA se divide en dos fases:

- Fase de diversidad: Las soluciones se crean usando los puntos efectivos y los individuos de la generación anterior.
- Fase de superposición: Las soluciones se crean solo con los individuos de la generación anterior.

Una vez se sabe cuáles son las características que distinguen a NOA de otros algoritmos, se puede pasar a ver cómo se utilizan estos mecanismos y cómo funcionan.

3.1. Funcionamiento de NOA

La definición del problema viene dada por la función objetivo, su dimensión y límites de cada variable. Se ha de tener en cuenta qué un problema se encuentra dentro de un dominio y por tanto cada variable puede tomar valores dentro de un rango [lim_inferior, lim_superior]. Estos límites forman parte de la definición del problema y se tienen en cuenta al inicializar la población inicial con valores aleatorios, pero dentro de los límites establecidos.

El tamaño de la población determina cuantos individuos iniciales se tienen y cuantas soluciones nuevas se crean en cada generación.

El criterio de parada determina cuántas evaluaciones va a realizar el algoritmo, siendo una evaluación una iteración de un individuo en el bucle interno; por tanto el bucle externo se ejecuta (evaluaciones / N) veces.

Algoritmo 1: Pseudocódigo del algoritmo NOA

Input: Definición del problema, tamaño N de la población, y el criterio de parada.
Output: La solución óptima $\overrightarrow{X}_{Best}$.

```
1  //Inicialización
2  Inicialización de la población;
3  Evaluación del fitness de los individuos;
4  //Bucle principal
5  while No se llega al criterio de parada do
6      Generar_coeficientes_lambda();
7      Generar_coeficientes_alfa();
8      Identificar la mejor solución actual;
9      //Bucle interno
10     for i = 1 hasta N do
11         Elegir tres individuos aleatorios X_j, X_k y X_m donde i ≠ j ≠ k ≠ m;
12         Genera dos aleatorios β1 y β2 en el rango [0,1];
13         Calcula los números δj = β1 × (-1)^j y δk = β2 × (-1)^k;
14         //Fase de diversificación (Exploración)
15         Construir el punto efectivo P_1;
16         if λ1 = 1 then
17             Actualiza la solución X'_i ;
18         else
19             Construir los vectores Vj y Vk;
20             Construir los puntos efectivos P_2 y P_3;
21             Actualiza la solución X'_i;
22         end
23         if X'_i = X_Best ∨ X_i = X_Best ∨ r < 1/4 then
24             //Fase de superposición (Explotación)
25             Construir el punto efectivo P_1;
26             Actualiza la solución X'_i;
27         end
28         Devuelve la solución X'_i al espacio de búsqueda;
29         Evaluación del fitness de la nueva solución X'_i;
30         Sustituye X_i por la nueva solución X'_i si esta es mejor;
31     end
32 end
33 return la mejor solución;
```

3.2. Extracción de los coeficientes de selección

3.3.1 Generar los coeficientes α

Para cada generación se calculan cuatro coeficientes para el proceso de selección de los mapeos efectivos:

$\alpha_1, \alpha_2, \alpha_3, \alpha_4$.

$\alpha_1 = r_1$
$\alpha_2 = r_2 - \frac{3}{8}$

$\alpha_3 = r_3 - \frac{1}{4}$

$\alpha_4 = r_4 - \frac{3}{8}$

Donde r_1, r_2, r_3, r_4 son números aleatorios con distribución uniforme en el rango [0, 1]. (2)

Función 1: Generar_coeficientes_alfa

Input: semilla para MersenneTwister
Output: Vector Alfa

```
1  Alfa[1] se genera como aleatorio en el rango [0, 1];
2  Alfa[2] se genera como aleatorio en el rango [-3/8, 5/8];
3  Alfa[3] se genera como aleatorio en el rango [-1/4, 3/4];
4  Alfa[4] se genera como aleatorio en el rango [-3/8, 5/8];
5  return Alfa;
```

3.3.1 Generar los coeficientes λ

Para cada generación se calculan ocho coeficientes para el proceso de construcción de puntos efectivos:

$\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5, \lambda_6, \lambda_7, \lambda_8$.

$\lambda_n = \text{mod}(a_n, 2)$ Esta operación devuelve el resto de la división de a_n entre 2.

Donde a_n son los valores de décima, centésima, milésima, etc. parte de un número aleatorio con distribución uniforme en el rango [0, 1]. (2)
Esto resulta algo retorcido para conseguir ocho números que valgan 0 o 1, por eso lo he simplificado como el módulo de un entero aleatorio sin signo para cada λ:

Función 2: Generar_coeficientes_lambda

Input: semilla para MersenneTwister
Output: Vector Lambda

```
1  for i = 1 hasta N do
2  |  r se genera como entero aleatorio sin signo;
3  |  Lambda[i] = r mod 2;
4  end
5  return Lambda;
```

3.3. Fase de diversificación

Por cada individuo o solución $\overrightarrow{X_i}$ de la población actual, se genera un individuo $\overrightarrow{X'_i}$ derivado utilizando los mapas y puntos efectivos en función de los valores de Alfa y Lambda; además de los individuos $\overrightarrow{X_i}$, $\overrightarrow{X_j}$, $\overrightarrow{X_k}$ y $\overrightarrow{X_m}$ (que son distintos entre sí) para calcular a su vez estos puntos y mapas efectivos:

Función 3: Fase de diversificación

Input: Los individuos $\overrightarrow{X_i}$, $\overrightarrow{X_j}$, $\overrightarrow{X_k}$, $\overrightarrow{X_m}$ y $\overrightarrow{X}_{Best}$; los valores β_1, β_2, δ_j y δ_k; los vectores Alfa y Lambda; y la dimensión D del problema.

Output: El individuo generado $\overrightarrow{X'}_i$.

```
1  P1 = Construir_P1();
2  if Lambda[1] = 1 do
3  |  S1 = Construir_S1();
4  |  X'_i = S1;
5  else
6  |  T3 = Construir_T3();
7  |  V_j = Transferencia(X_j, T3, Alfa[1], D);
8  |  V_k = Transferencia(X_k, T3, Alfa[1], D);
9  |  T2 = Construir_T2();
10 |  P2 = Construir_P2();
11 |  P3 = Construir_P3();
12 |  S2 = Construir_S2();
13 |  X'_i = S2;
14 end
15 return X'_i ;
```

Se puede ver en este primer paso cómo la solución dada para la nueva generación será $\overrightarrow{S_1}$ o $\overrightarrow{S_2}$, si λ_1 es igual a 1 o 0 respectivamente. Se verá cómo se calculan las soluciones $\overrightarrow{S_1}$ y $\overrightarrow{S_2}$ a continuación, pero antes se construye el punto efectivo $\overrightarrow{P_1}$ que se utiliza para hallar ambas.

3.3.1 Construcción de $\overrightarrow{P_1}$

Debido a que $\overrightarrow{T_1}$ solo se necesita para $\overrightarrow{P_1}$ en caso de que λ_3 valga 0, se incluye el cálculo de $\overrightarrow{T_1}$ en $\overrightarrow{P_1}$; el cuál en ese caso viene dado como una combinación lineal de $\overrightarrow{X_m}$ y $\overrightarrow{X}_{Best}$ de una forma para λ_6 igual a 1 y de otra para λ_6 igual a 0.
Si en cambio λ_3 vale 1 $\overrightarrow{P_1}$ será $\overrightarrow{X_m}$.

Función 3.1: Construir_P1

Input: Los individuos $\overrightarrow{X_m}$ y $\overrightarrow{X}_{Best}$; el valor β_1; el vector Lambda; y la dimensión D.

Output: El punto $\overrightarrow{P_1}$.

```
1  if Lambda[3] = 1 do
2  |  P1 = X_m;
3  // Si no se construye T1 y se le atribuye a P1.
4  else if Lambda[6] = 1 do
5  |  P1 = T1 = 0.5 × (X_Best + X_m);
6  else
7  |  P1 = T1 = β1 × X_m + (1 – β2) × X_Best;
8  end
9  return P1;
```

Una vez calculado $\overrightarrow{P_1}$, se comienza la construcción de $\overrightarrow{S_1}$ o $\overrightarrow{S_2}$; primero se ve la de $\overrightarrow{S_1}$,

que no requiere de puntos adicionales.

3.3.2 Construcción de $\overrightarrow{S_1}$

$\overrightarrow{S_1}$ se calcula como combinación lineal de $\overrightarrow{P_1}$, $\overrightarrow{X_i}$, $\overrightarrow{X_j}$, y $\overrightarrow{X_k}$; variando en función de si λ_2 vale 0 o 1.

Función 3.2: Construir_S1

Input: El punto $\overrightarrow{P_1}$; los individuos $\overrightarrow{X_i}$, $\overrightarrow{X_j}$ y $\overrightarrow{X_k}$; los valores β_1 y β_2; el vector Lambda; y la dimensión D del problema.
Output: El punto $\overrightarrow{S_1}$.

if Lambda[2] = 1 **do**
  $\overrightarrow{S_1} = \overrightarrow{P_1} + \beta_1 \times (\overrightarrow{X_i} - \overrightarrow{X_j}) + \beta_2 \times (\overrightarrow{X_i} - \overrightarrow{X_k})$;
else
  $\overrightarrow{S_1} = \overrightarrow{X_i} + \beta_1 \times (\overrightarrow{P_1} - \overrightarrow{X_j}) - \beta_2 \times (\overrightarrow{P_1} - \overrightarrow{X_k})$;
end
return $\overrightarrow{S_1}$;

Si se va a construir $\overrightarrow{S_2}$ en lugar de $\overrightarrow{S_1}$; antes se construyen los puntos efectivos $\overrightarrow{P_2}$ y $\overrightarrow{P_3}$ y los vectores $\overrightarrow{V_j}$ y $\overrightarrow{V_k}$ de los que depende, y como su vez todos estos dependen de los puntos efectivos $\overrightarrow{T_2}$ y $\overrightarrow{T_3}$ se deben construir estos últimos los primeros. Concretamente se construye $\overrightarrow{T_3}$, que también se utiliza para luego construir $\overrightarrow{T_2}$.

3.3.3 Construcción de $\overrightarrow{T_3}$

En este caso, si λ es igual a 1, $\overrightarrow{T_3}$ simplemente tomará el valor de la mejor solución o mejor individuo actual($\overrightarrow{X}_{Best}$); si no tomará el valor del individuo que se está evaluando ($\overrightarrow{X_i}$).

Función 3.3: Construir_T3

Input: Los individuos $\overrightarrow{X_i}$ y $\overrightarrow{X}_{Best}$; el vector Lambda; y la dimensión D del problema.
Output: El punto $\overrightarrow{T_3}$.

if Lambda[8] = 1 **do**
  $\overrightarrow{T_3} = \overrightarrow{X}_{Best}$;
else
  $\overrightarrow{T_3} = \overrightarrow{X_i}$;
end
return $\overrightarrow{T_3}$;

3.3.4 Construcción de $\overrightarrow{T_2}$

Con $\overrightarrow{T_3}$ calculado, se puede utilizar junto a $\overrightarrow{P_1}$ para construir $\overrightarrow{T_2}$. Para esto se utiliza el mapa de traducción; sobre el vector resultante de aplicar el método de reemplazo sobre $\overrightarrow{T_3}$ con $\overrightarrow{P_1}$ cuando λ_7 sea 1 y sobre el vector resultante de aplicar el método de mezcla sobre $\overrightarrow{T_3}$ con $\overrightarrow{P_1}$ cuando λ_7 sea 0.

Función 3.4: Construir_T2

Input: Los puntos $\overrightarrow{P_1}$ y $\overrightarrow{T_3}$; el valor β_2; los vectores Alfa y Lambda; y la dimensión D del problema.

Output: El punto $\overrightarrow{T_2}$.

if Lambda[7] = 1 **do**
  $\overrightarrow{T_2}$ = Traducción(reemplazar($\overrightarrow{T_3}$, $\overrightarrow{P_1}$, D), β_2, Alfa[4], D);
else
  $\overrightarrow{T_2}$ = Traducción(mezclar($\overrightarrow{T_3}$, $\overrightarrow{P_1}$, D), β_2, Alfa[4], D);
end
return $\overrightarrow{T_2}$;

3.3.5 Construcción de $\overrightarrow{P_2}$

$\overrightarrow{P_2}$ será igual a $\overrightarrow{T_2}$ en caso de que λ_4 sea distinto de 1, situación en la que $\overrightarrow{P_2}$ es la suma de $\overrightarrow{X_m}$ y un vector de tamaño D que contiene el valor β_1 en todas sus posiciones; o, dicho de otra forma, el resultado de añadir β_1 a todas las componentes de $\overrightarrow{X_m}$.

Función 3.5: Construir_P2

Input: El punto $\overrightarrow{T_2}$; el individuo $\overrightarrow{X_m}$; el valor β_1; el vector Lambda; y la dimensión D del problema.

Output: El punto $\overrightarrow{P_2}$.

if Lambda[4] = 1 **do**
  $\overrightarrow{P_2} = \overrightarrow{X_m} + \beta_1 \times 1_D$; //$1_D$ es un vector de unos de tamaño D.
else
  $\overrightarrow{P_2} = \overrightarrow{T_2}$;
end
return $\overrightarrow{P_2}$;

3.3.6 Construcción de $\overrightarrow{P_3}$

Para generar el último punto efectivo, se aplican varios mapas de combinación y transformación de forma concatenada; empezando por el de distribución, aplicando su resultado a un mapa de transferencia junto a $\overrightarrow{T_3}$ y por último un mapa de dilatación. La diferencia entre si λ_5 toma valor 1 o 0 es, respectivamente, si el mapa de distribución aplica $\overrightarrow{T_3}$ sobre $\overrightarrow{P_2}$ o aplica $\overrightarrow{P_2}$ sobre $\overrightarrow{T_3}$.

Función 3.6: Construir_P3

Input: Los puntos $\overrightarrow{P_1}$, $\overrightarrow{T_2}$ y $\overrightarrow{T_3}$; el valor β_2; los vectores Alfa y Lambda; y la dimensión D del problema.

Output: El punto $\overrightarrow{P_3}$.

if Lambda[5] = 1 **do**
  $\overrightarrow{P_3}$ = Dilatación[Transferencia(distribuir($\overrightarrow{P_1}$, $\overrightarrow{T_3}$, D), $\overrightarrow{T_2}$, Alfa[3], D), β_2, Alfa[2], D];
else
  $\overrightarrow{P_3}$ = Dilatación[Transferencia (distribuir ($\overrightarrow{T_3}$, $\overrightarrow{P_1}$, D), $\overrightarrow{T_2}$, Alfa[3], D), β_2, Alfa[2], D];
end
return $\overrightarrow{P_3}$;

Ya se puede hallar la solución $\overrightarrow{S_1}$ que se corresponde son un individuo de siguiente generación.

3.3.7 Construcción de $\overrightarrow{S_2}$

De forma semejante a $\overrightarrow{S_1}$, $\overrightarrow{S_2}$ se calcula como combinación lineal de $\overrightarrow{P_2}$, $\overrightarrow{P_3}$, $\overrightarrow{V_j}$, y $\overrightarrow{V_k}$; variando en función de si λ_2 vale 0 o 1.

Función 3.7: Construir_S2

Input: Los puntos $\overrightarrow{P_2}$ y $\overrightarrow{P_3}$; los vectores $\overrightarrow{V_j}$ y $\overrightarrow{V_k}$; los valores δ_j y δ_k; el vector Lambda; y la dimensión D del problema.

Output: El punto $\overrightarrow{S_2}$.

if Lambda[2] = 1 **do**
  $\overrightarrow{S_2} = \overrightarrow{P_2} + \delta_j \times (\overrightarrow{P_3} - \overrightarrow{V_j}) + \delta_k \times (\overrightarrow{P_3} - \overrightarrow{V_k})$;
else
  $\overrightarrow{S_2} = \overrightarrow{P_3} + \delta_j \times (\overrightarrow{P_2} - \overrightarrow{V_j}) + \delta_k \times (\overrightarrow{P_2} - \overrightarrow{V_k})$;
end
return $\overrightarrow{S_2}$;

Durante esta fase de diversificación o proceso de exploración de soluciones se han utilizado los puntos efectivos (como ya se ha visto) y los mapas efectivos; por lo que se verá en los siguientes puntos el funcionamiento de estos mapeos y así entender mejor como logran combinar y transformar puntos efectivos e individuos de anteriores generaciones.

3.4. Mapeos de combinación efectivos

3.4.1. Reemplazar

Como dice el nombre, este mapa reemplaza algunos valores del vector de entrada $\overrightarrow{V_{in}}$, en posiciones aleatorias, por los respectivos del vector objetivo $\overrightarrow{V_{tg}}$. El vector de salida $\overrightarrow{V}_{out}$ será $\overrightarrow{V_{in}}$ con nuevos valores pertenecientes al $\overrightarrow{V_{tg}}$, pero sin llegar a modificar ninguno de los vectores originales.

Función 4.1: Reemplazar

Input: Los vectores $\overrightarrow{V_{in}}$ y $\overrightarrow{V_{tg}}$, la semilla para Mersenne Twister y la dimensión D del problema.

Output: Vector $\overrightarrow{V}_{out}$.

for i = 1 hasta D **do**
  r se genera como un número entero aleatorio sin signo;
  if r es par **do**
    $\overrightarrow{V}_{out}$[i] = $\overrightarrow{V_{tg}}$[i];
  else
    $\overrightarrow{V}_{out}$[i] = $\overrightarrow{V_{in}}$[i];
  end
end
return $\overrightarrow{V}_{out}$;

He aquí un ejemplo para visualizar el funcionamiento de reemplazar:

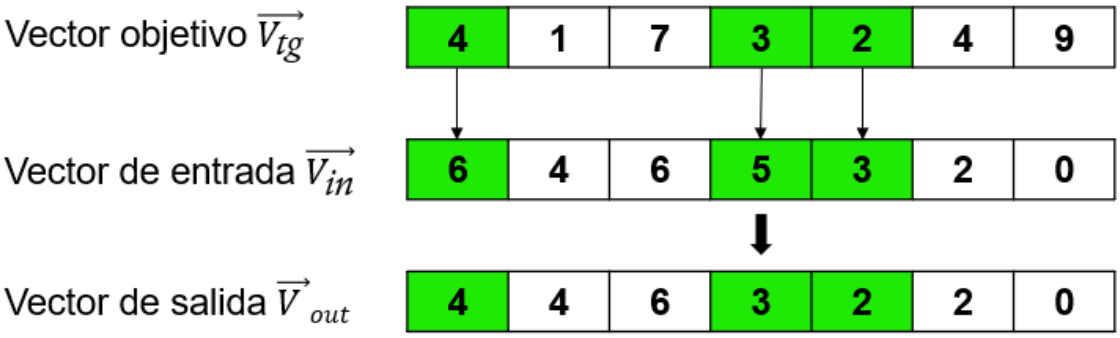

Figura 3: Función Reemplazar

3.4.2. Mezclar

Como dice el nombre, este mapa mezcla algunos valores del vector objetivo $\overrightarrow{V_{tg}}$ en el vector de entrada $\overrightarrow{V_{in}}$, sobre posiciones aleatorias. El vector de salida $\overrightarrow{V}_{out}$ será $\overrightarrow{V_{in}}$ con nuevos valores pertenecientes al $\overrightarrow{V_{tg}}$, pero sin llegar a modificar ninguno de los vectores originales.

Función 4.2: Mezclar

Input: Los vectores $\overrightarrow{V_{in}}$ y $\overrightarrow{V_{tg}}$, la semilla para Mersenne Twister y la dimensión D del problema.
Output: Vector $\overrightarrow{V}_{out}$.

```
1  V_aux = Fisher_Yetes_shuffle(V_tg) //Baraja el contenido del vector de forma aleatoria(6)
1  for i = 1 hasta D do
2      r se genera como un número entero aleatorio sin signo;
3      if r es par do
4          V_out[i] = V_aux[i];
5      else
6          V_out[i] = V_in[i];
7      end
8  end
9  return V_out ;
```

He aquí un ejemplo para visualizar el funcionamiento de mezclar:

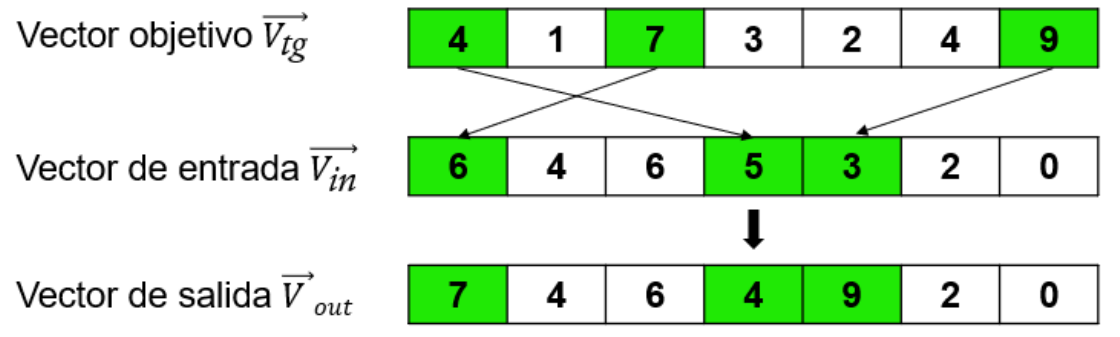

Figura 4: Función Mezclar

3.4.3. Distribuir

Como dice el nombre, este mapa distribuye un valor aleatorio del vector objetivo $\overrightarrow{V_{tg}}$ en posiciones aleatorias del vector de entrada $\overrightarrow{V_{in}}$. El vector de salida $\overrightarrow{V}_{out}$ será $\overrightarrow{V_{in}}$ con nuevos valores pertenecientes al $\overrightarrow{V_{tg}}$, pero sin llegar a modificar ninguno de los vectores originales.

Función 4.3: Distribuir

Input: Los vectores $\overrightarrow{V_{in}}$ y $\overrightarrow{V_{tg}}$, la semilla para Mersenne Twister y la dimensión D del problema.
Output: Vector $\overrightarrow{V}_{out}$.

```
1  valor_objetivo = valor aleatorio de V_tg;
1  for i = 1 hasta D do
2      r se genera un número entero aleatorio sin signo;
3      if r es par do
4          V_out[i] = valor_objetivo;
5      else
6          V_out[i] = V_in[i];
7      end
8  end
9  return V_out ;
```

He aquí un ejemplo para visualizar el funcionamiento de distribuir:

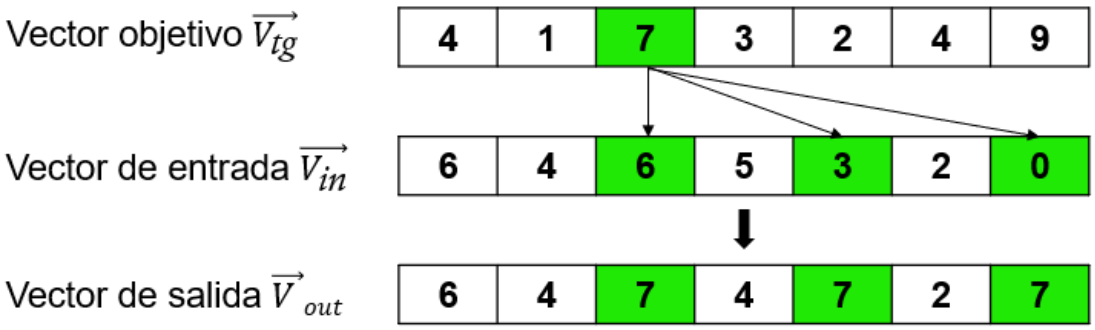

Figura 5: Función Distribuir

3.5. Mapeos de transformación efectivos

Todos los mapeos de transformación devuelven el vector de entrada sin alterar en al caso de que el valor de α sea menor o igual a un medio(0.5).

3.5.1. Traducción

Recibe un vector de entrada $\vec{V}_i$ y un escalar *r*, devolviendo un vector $\overrightarrow{V}_{out}$ que será igual a $\vec{V}_i$ o una combinación lineal de este con *r* en función del valor de α_n para n ∈ {1, 2, 3, 4}; es decir, uno de los valores del vector Alfa[n] para n ∈ {1, 2, 3, 4}.

Función 5.1: Traducción

Input: El vector $\vec{V_i}$; un escalar r; un valor del vector Alfa; y la dimensión D del problema.
Output: El punto $\vec{V}_{out}$.

if Alfa[n] <= 0.5 **do**
  $\vec{V}_{out} = \vec{V_i}$;
else
  $\vec{V}_{out} = \vec{V_i} + r^2 \times 1_D$; //$1_D$ es un vector de unos de tamaño D.
end
return $\vec{V}_{out}$;

3.5.2. Dilatación

Recibe un vector de entrada $\vec{V_i}$ y un escalar *r*, devolviendo un vector $\vec{V}_{out}$ que será igual a $\vec{V_i}$ o una combinación lineal de este con *r* en función del valor de α_n para n ∈ {1, 2, 3, 4}; es decir, uno de los valores del vector Alfa[n] para n ∈ {1, 2, 3, 4}.

Función 5.2: Dilatación

Input: El vector $\vec{V_i}$; un escalar r; un valor del vector Alfa; y la dimensión D del problema.
Output: El punto $\vec{V}_{out}$.

if Alfa[n] <= 0.5 **do**
  $\vec{V}_{out} = \vec{V_i}$;
else
  $\vec{V}_{out} = \vec{V_i} \times r^2$;
end
return $\vec{V}_{out}$;

3.5.3. Transferencia

Recibe dos vectores de entrada $\vec{V_i}$ y $\vec{V_j}$, devolviendo un vector $\vec{V}_{out}$ que será igual a $\vec{V_i}$ o $\vec{V_j}$ en función del valor de α_n para n ∈ {1, 2, 3, 4}; es decir, uno de los valores del vector Alfa[n] para n ∈ {1, 2, 3, 4}.

Función 5.3: Transferencia

Input: Los vectores $\vec{V_i}$ y $\vec{V_j}$; un valor del vector Alfa; y la dimensión D del problema.
Output: El punto $\vec{V}_{out}$.

if Alfa[n] <= 0.5 **do**
  $\vec{V}_{out} = \vec{V_i}$;
Else
  $\vec{V}_{out} = \vec{V_j}$;
end
return $\vec{V}_{out}$;

3.6. Fase de superposición

Esta es la última fase del algoritmo, que se corresponde con la fase de explotación de un algoritmo de optimización; llevándose a cabo en caso de que se cumpla una de las siguientes condiciones:

- La solución generada en la fase de diversificación, $\overrightarrow{X'_i}$, es igual a la mejor solución actual, $\overrightarrow{X}_{Best}$.
- El individuo de la evaluación actual, $\overrightarrow{X_i}$, es igual a la mejor solución actual, $\overrightarrow{X}_{Best}$.
- El valor de *r* es menor o igual a $\frac{1}{4}$(0.25), siendo r número aleatorio con distribución uniforme en el rango [0, 1].

Una vez se cumpla alguna, se han de reconstruir $\overrightarrow{S_1}$ y $\overrightarrow{P_1}$ de nuevo reemplazando β_1 y β_2 por los vectores $\overrightarrow{\beta_1}$ y $\overrightarrow{\beta_2}$ de dimensión D y cuyos valores son números aleatorios uniformemente distribuidos en el rango [-1, 1]. Es **importante** destacar que los productos realizados con estos vectores **no son productos vectoriales, son multiplicaciones elemento a elemento**, es decir, el vector resultante $\overrightarrow{V_r}$ se calcula como $\overrightarrow{V_r}[1] = \overrightarrow{\beta_1}[1] \times \overrightarrow{X_m}[1]$, $\overrightarrow{V_r}[2] = \overrightarrow{\beta_1}[2] \times \overrightarrow{X_m}[2]$, …, $\overrightarrow{V_r}[D] = \overrightarrow{\beta_1}[D] \times \overrightarrow{X_m}[D]$.

Función 6: Fase de superposición

Input: Los individuos $\overrightarrow{X'_i}$, $\overrightarrow{X_i}$, $\overrightarrow{X_m}$ y $\overrightarrow{X}_{Best}$; el vector Lambda; y la dimensión D.
Output: La solución actualizada $\overrightarrow{X'_i}$.

r se genera como un número aleatorio en el rango [0, 1];
if $\overrightarrow{X'_i} = \overrightarrow{X}_{Best} \vee \overrightarrow{X_i} = \overrightarrow{X}_{Best} \vee r <= \frac{1}{4}$ **then**
  $\overrightarrow{\beta_1}$[D] se rellena con números aleatorios en el rango [-1, 1]
  $\overrightarrow{\beta_2}$[D] se rellena con números aleatorios en el rango [-1, 1]
  if Lambda[3] = 1 **do**
    $\overrightarrow{P_1} = \overrightarrow{X_m}$;
  else if Lambda[6] = 1 **do**
    $\overrightarrow{P_1} = 0.5 \times (\overrightarrow{X}_{Best} + \overrightarrow{X_m})$;
  else
    $\overrightarrow{P_1} = \overrightarrow{\beta_1} \times \overrightarrow{X_m} + (1_D - \overrightarrow{\beta_2}) \times \overrightarrow{X}_{Best}$; //$1_D$ es un vector de unos de dimensión D
  end
  if Lambda[2] = 1 **do**
    $\overrightarrow{S_1} = \overrightarrow{P_1} + \overrightarrow{\beta_1} \times (\overrightarrow{X_i} - \overrightarrow{X_j}) + \overrightarrow{\beta_2} \times (\overrightarrow{X_i} - \overrightarrow{X_k})$;
  else
    $\overrightarrow{S_1} = \overrightarrow{X_i} + \overrightarrow{\beta_1} \times (\overrightarrow{P_1} - \overrightarrow{X_j}) - \overrightarrow{\beta_2} \times (\overrightarrow{P_1} - \overrightarrow{X_k})$;
  end
  $\overrightarrow{X'_i} = \overrightarrow{S_1}$;
end
return $\overrightarrow{X'_i}$;

Tras esta fase, se llegue a realizar o no:

- Se devuelve la solución $\overrightarrow{X'_i}$ al espacio de búsqueda, es decir, si contiene valores que exceden límites inferior o superior se le reasigna su valor respectivo del individuo $\overrightarrow{X_i}$.
- Luego se evalúa el valor fitness de $\overrightarrow{X'_i}$, o lo que es lo mismo, se evalúa en la función objetivo.
- Por último, antes de volver al inicio del bucle, se sustituye el individuo $\overrightarrow{X_i}$ por la

nueva solución $\overrightarrow{X'}_i$ si su valor fitness es mejor.

Una vez se tiene la implementación secuencial del algoritmo, se puede pasar al planteamiento del paralelismo que se va a realizar sobre este.

Estrategias de paralelización

Antes de intentar paralelizar el algoritmo, se debe entender qué aspectos del son los que generan carga computacional; por tanto, se revisa la complejidad computacional de NOA(2).

Se parte de las siguientes asignaciones; T es la cantidad de iteraciones o generaciones, C es el coste computacional de la función objetivo, D es la dimensión del problema and N es el tamaño de la población. La complejidad temporal de NOA se calcula de la siguiente manera:

- La complejidad temporal de inicializar la población es O(ND + 1).
- La complejidad temporal de todas las evaluaciones de la función objetivo es O(TCN).
- La complejidad temporal de actualizar todas las soluciones es O(TND) + 2O($\frac{1}{2}$TN).

Se sabe que $1 < TCN$, $1 < TND$, $TN < TCN$ y $TN < TND$, por lo que la complejidad temporal de NOA se puede evaluar asintóticamente como:

$$O(NOA) = O(ND + TCN + TND + TN + 1) \cong O(TCN + TND)$$

Se sabe entones que, dada esta complejidad temporal, la velocidad de ejecución solo se ve afectada por parámetros de entrada al optimizador y no por los métodos y operaciones realizadas por el algoritmo. Esto es algo muy deseable a la hora de paralelizar, pues se puede dejar de lado el paralelismo funcional y centrarse en el paralelismo de datos; lo que significa no tener que realizar balanceo de carga y la implementación de mecanismos de sincronización más complejos.

4.1. Tareas paralelizables en el algoritmo NOA

Se ha visto que el rendimiento del algoritmo se ve afectado por N, T, D y C; por lo que puede parecer que se puede paralelizar en muchos aspectos, pero realmente no es posible paralelizar las funciones objetivo ni las operaciones con una población de dimensión D directamente, se debe paralelizar el funcionamiento interno de estas lo que supondría una sobrecarga por la sincronización necesaria tras cada operación. Como estas operaciones se realizan muchas veces no tiene sentido se paralelicen internamente, y en su lugar se debería paralelizar el proceso que las contiene, que en este caso es el bucle de cálculo de generaciones del algoritmo.

Realmente las opciones son limitadas, pues es un algoritmo de optimización cuyas fases de exploración y de explotación son dependientes entre sí y deben realizarse de forma secuencial, además cada generación de soluciones se debe basar en la anterior para mejorar progresivamente y si se calculan múltiples de ellas simultáneamente simplemente tendríamos múltiples algoritmos secuenciales, pero con menos evaluaciones. Este último planteamiento se podría ajustar recurriendo al paralelismo por islas, que consiste en la ejecución del algoritmo de forma independiente en cada hebra con sus propias variables locales; pero esto denigraría el proceso de optimización del algoritmo al no tener tantas generaciones por hebra como para converger en la solución, y en su lugar se dependería de la posibilidad de que al

estar, efectivamente ejecutando múltiples algoritmos a la vez, una de las hebras llegase a encontrar el óptimo por azar.

Las opciones se reducen a dividir la generación de los individuos entre las hebras en cada iteración, que dividiría la carga computacional de NCD.

En los siguientes apartados se concretan las herramientas y la metodología utilizadas para aplicar esta última estrategia de paralelismo sobre el algoritmo NOA.

4.2. Herramientas para paralelizar

Se ha utilizado *phtreads* como mecanismo de paralelización, el cual crea hebras que realizan la tarea que se les encomiende. Pero *phtreads* no genera paralelismo real, las hebras se ejecutan concurrentemente en núcleos lógicos del procesador y se requieren de múltiples núcleos físicos para empezar a funcionar de forma paralela; llegando poder ejecutarse de forma completamente paralela si el sistema tiene la misma cantidad de núcleos físicos que hebras creadas. Aún así la concurrencia significa que si el sistema tiene recursos disponibles, el programa los va a aprovechar, resultando en una ganancia en la aceleración aunque no se estén ejecutando todas las hebras al mismo tiempo siempre.

4.3. Implementación de *phtreads*

La implementación con *phtreads* requiere de una estructura de datos en la que se almacenen los datos compartidos que vayan a necesitar cada hebra, luego cada una crea las variables locales que necesita para almacenar datos independientes.

Una vez se comienza la ejecución del algoritmo, se realiza lo siguiente:

- Se inicializa el generador de números aleatorios para que cada hebra tenga el suyo propio; esto se debe hacer por primera vez de forma atómica, aislando con mutex la línea de código.
- Se dividen los individuos de la población entre el número de hebras por bloques para reducir los fallos de páginas, estableciendo el inicio y final de los bucles para cada hebra (en las variables *start* y *end*).
- Para evitar condiciones de carrera, se sitúan barreras o *barriers* en los puntos en los que toda la población tiene que estar actualizada; esto ocurre después de la inicialización de la población, tras el cálculo de cada nueva generación de individuos y después de actualizar los vectores compartidos con las soluciones calculadas.

Es importante recalcar que la actualización de la generación actual con los nuevos individuos se realiza una vez todos se han calculado, a diferencia del funcionamiento secuencial que actualiza la población cada vez que se encuentra una solución mejor. Esto se hace para que no hayan accesos simultáneos a los individuos por parte de las hebras y es indiscutible, pero significa que el algoritmo puede converger hacia el valor óptimo más lentamente y podría deteriorar los resultados respecto a la versión secuencial.

Experimentación y resultados

Se han utilizado 10 de 60 funciones benchmark y 2 de los problemas reales considerados en *Nizar optimization algorithm…*(2).

Para mantener la consistencia con respecto a los resultados presentados en el documento original, se realizan las comprobaciones del funcionamiento del algoritmo implementado en Matlab y en C con los mismos parámetros; realizando ejecuciones con tamaño de población 25 y 35000 evaluaciones de la función objetivo, además de los parámetros específicos de cada problema. Pero como se verá en el apartado de experimentación, hay razones por la que cambiar el tamaño de la población puede resultar interesante.

Pero antes y de forma inmediata se describen los problemas elegidos y sus parámetros específicos.

5.1. Problemas de prueba(benchmark) de optimización global

Antes es importante recalcar que no se ha usado la expresión referenciada en *Nizar optimization algorithm…*(2) para la mayoría de las funciones por inconsistencias en la notación del artículo y erratas en algunas de las expresiones, siendo incorrectas; en su lugar se ha contrastado la descripción con otras fuentes.

Se han escogido a propósito 5 funciones benchmark con óptimo en 0 y otras 5 con óptimo menor que 0 para asegurar que el algoritmo converge correctamente hacia cualquier resultado y no solo hacia 0, lo que podría significar un error en la implementación. También se ha escogido una multimodal, siendo las otras 9 unimodales, para comprobar que la fase de exploración no se queda encallada en óptimos locales y así de nuevo se asegura una implementación correcta del algoritmo.

Cuanta más variedad de problemas distintos se prueben mayor es la seguridad de que se comporta como debería y estos 10 problemas no tienen restricciones; por lo que se han escogido 2 problemas de ingeniería del mundo real que si están restringidos.

Por último, debido a que el coste computacional de todos los problemas planteados es muy bajo(con tiempos de ejecución del orden de milisegundos), se ha adaptado uno de ellos para añadirle operaciones que no alteren el resultado pero simulen una carga computacional mayor aumentando significativamente el tiempo de ejecución.

5.1.1. Sphere

Es una función unimodal con componentes en D dimensiones y viene descrita por (7):

$$f(x) = \sum_{i=1}^{D} x_i^2$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 50.
- Dominio o rango de la función delimitado por [-100, 100].

5.1.2. Quartic

Es una función unimodal con componentes en D dimensiones y que está personalizada en el artículo original (2), viene descrita por:

$$f(x) = \sum_{i=1}^{D} ix_i^4$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 50.
- Dominio o rango de la función delimitado por [-1.28, 1.28].

5.1.3. Powell Sum

Es una función unimodal con componentes en D dimensiones y viene descrita por (8):

$$f(x) = \sum_{i=1}^{D} |x_i|^{i+1}$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 50.
- Dominio o rango de la función delimitado por [-1, 1].

5.1.4. Sum Squares

Es una función unimodal con componentes en D dimensiones y viene descrita por (9):

$$f(x) = \sum_{i=1}^{D} ix_i^2$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 50.
- Dominio o rango de la función delimitado por [-10, 10].

5.1.5. Schwefel's 2.20

Es una función unimodal con componentes en D dimensiones y viene descrita por (10):

$$f(x) = \sum_{i=1}^{D} |x_i|$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 50.
- Dominio o rango de la función delimitado por [-100, 100].

5.1.6. Stepint

Es una función unimodal con componentes en D dimensiones y viene descrita por (11):

$$f(x) = 25 + \sum_{i=1}^{D} \lfloor x_i \rfloor$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 50.
- Dominio o rango de la función delimitado por [-5.12, 5.12].

5.1.7. Ridge

Es una función unimodal con componentes en D dimensiones y que está personalizada en el artículo original (2), viene descrita por:

$$f(x) = x_1 + \sqrt{\sum_{i=2}^{D} x_i^2}$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 50.
- Dominio o rango de la función delimitado por [-5, 5].

5.1.8. Neumaier's N. 3

Es una función unimodal con componentes en D dimensiones y viene descrita por (12)(12):

$$f(x) = \sum_{i=1}^{D} (x_i - 1)^2 - \sum_{i=2}^{D} x_i x_{i-1}$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 15.
- Dominio o rango de la función delimitado por [-100, 100].

5.1.9. Ackley N. 2

Es una función unimodal con componentes en D dimensiones y viene descrita por (13):

$$f(x) = -200\exp\left(-0.2\sqrt{x_1^2 + x_2^2}\right)$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 2; que en este caso está prefijada.
- Dominio o rango de la función delimitado por [-32, 32].

5.1.10. Shekel 10

Es una función multimodal con componentes en D dimensiones y viene descrita por (14)(13):

$$f(x) = -\sum_{i=1}^{m} \left(\sum_{j=1}^{D} (x_j - a_{ij})^2 + c_i \right)^{-1}$$

Donde *a* es una matriz de tamaño *m* × D y *c* es un vector de longitud *m*. Para *m* = 10 y D = 4 se dan de la siguiente forma:

$$a = \begin{pmatrix} 4.0 & 1.0 & 8.0 & 6.0 & 3.0 & 2.0 & 9.0 & 8.0 & 6.0 & 7.0 \\ 4.0 & 1.0 & 8.0 & 6.0 & 7.0 & 9.0 & 3.0 & 1.0 & 2.0 & 3.6 \\ 4.0 & 1.0 & 8.0 & 6.0 & 3.0 & 2.0 & 9.0 & 8.0 & 6.0 & 7.0 \\ 4.0 & 1.0 & 8.0 & 6.0 & 7.0 & 9.0 & 3.0 & 1.0 & 2.0 & 3.6 \end{pmatrix}^T$$

$$c = \frac{1}{10}(1, 2, 2, 4, 4, 6, 3, 7, 5, 5)$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 4.
- Dominio o rango de la función delimitado por [0, 10].

5.1.11. PVD

El problema de diseño de un depósito bajo presión o PVD (pressure vessel design) tiene el objetivo de minimizar el coste de fabricación, que se puede describir de la siguiente forma(15):

- Grosor del depósito: x_1.
- Grosor de la cabeza: x_2.
- Radio del depósito: x_3.
- Longitud del depósito: x_4.

$$f(x) = 0.6224x_1x_3x_4 + 1.7781x_2x_3^2 + 3.1661x_1^2x_4 + 19.84x_1^2x_3$$

Sujeto a cuatro restricciones:

- Tensión del aro: g_1.
- Tensión longitudinal: g_2.
- Volumen del depósito completo: g_3.
- Longitud del depósito completo: g_4.

$$g_1(x) = -x_1 + 0.0193x_3 \leq 0$$
$$g_2(x) = -x_2 + 0.00954x_3 \leq 0$$
$$g_3(x) = -\pi x_3^2 x_4 - \frac{4}{3}\pi x_3^3 + 1296000 \leq 0$$
$$g_4(x) = x_4 - 240 \leq 0$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 4; que en este caso está prefijada.
- Dominio o rango de las variables delimitado por: $x_1 \in [0, 100]$, $x_2 \in [0, 100]$, $x_3 \in [10, 200]$ y $x_4 \in [10, 200]$.

5.1.12. TCSD

El problema de diseño de un muelle de compresión/tensión o TCSD (tension/compression spring design) tiene el objetivo de minimizar el peso del muelle, que se puede describir de la siguiente forma(16):

- Diámetro del alambre: x_1.
- Diámetro del devanado: x_2.
- Cantidad de bobinas activas: x_3.

$$f(x) = (x_3 + 2)x_2x_1^2$$

Sujeto a cuatro restricciones:

- Deflexión mínima: g_1.
- Tensión cortante: g_2.
- Frecuencia de oscilación: g_3.
- Diámetro: g_4.

$$g_1(x) = 1 - \frac{x_2^3 x_3}{71875 x_1^4} \leq 0$$

$$g_2(x) = \frac{4x_2^2 - x_1 x_2}{12566(x_2 x_1^3 - x_1^4)} + \frac{1}{5108 x_1^2} - 1 \leq 0$$

$$g_3(x) = 1 - \frac{140.45 x_1}{x_2^2 x_3} \leq 0$$

$$g_4(x) = \frac{x_1 + x_2}{1.5} - 1 \leq 0$$

Para su optimización con el algoritmo NOA se han utilizado los siguientes parámetros:

- Dimensión D = 3; que en este caso está prefijada.
- Dominio o rango de las variables delimitado por: $x_1 \in [0.05, 2]$, $x_2 \in [0.25, 1.3]$ y $x_3 \in [2, 15]$.

5.1.13. Artificial

Se utiliza como base el problema de TCSD y se le añade carga adicional con las siguientes operaciones matriciales:

- Se realizan operaciones de filas sobre la matriz identidad I_m para generar A_m.
- Se calcula la inversa de A_m , que sabemos que tiene porque partimos de I_m.
- Se realiza el producto $A_m \times A_m^{-1}$ que va a devolver la matriz I_m.
- Se realiza el producto de los valores en la diagonal y el resultado se multiplica por el resultado del TCSD, poque si no forman parte del resultado el compilador ignoraría estos cálculos. Como se puede esperar la diagonal de la matriz I_m son todo unos, por lo que realmente se devuelve 1 × TCSD sin alterar el funcionamiento del problema y añadiendo carga computacional que de forma elegante dependerá del tamaño de las matrices cuadradas.

  En este documento se han utilizado matrices de tamaño m = 30, y siempre son cuadradas.

5.2. Prestaciones de la versión secuencial

Se comprueba que el funcionamiento de las versiones de Matlab y C es correcto y se comparan para comprobar que además se comportan igual. Los resultados calculados son promedios de 30 ejecuciones en ambos programas en todas las funciones benchmark:

Función	Matlab	C	Óptimo global
Sphere	0	0	0
Quartic	0	0	0
Powell Sum	0	0	0
Sum Squares	0	0	0
Schwefel 2.20	0	0	0
Stepint	-275	-275	-275
Ridge	-4,99999008	-4,9999661	-5
Neumaier N. 3	-664,999913	-665	-665
Ackley N. 2	-200	-200	-200
Shekel 10	-10,1758812	-10,3561457	-10,53644
PVD	5885,33277	5885,33278	5885,3327736164
TCSD	0,01266524	0,012665	0,0126652327883

Tabla 1: Resultados promedios obtenidos con NOA en Matlab y en C

Luego se comparan los tiempos de ejecución para comprobar si realmente la versión de C va a ser mucho más eficiente que la de Matlab. Los resultados calculados son promedios de 30 ejecuciones en ambos programas en todas las funciones benchmark:

Función	Matlab (s)	C (s)	Aceleración de C sobre Matlab
Sphere	0,54426203	0,04328997	12,5724751
Quartic	0,7906349	0,0459512	17,2059684
Powell Sum	0,76055275	0,07287837	10,4359193
Sum Squares	0,53407083	0,04354313	12,2653284
Schwefel 2.20	0,54598672	0,04683873	11,6567354
Stepint	0,53373293	0,04650393	11,4771568
Ridge	0,51492746	0,04205177	12,245085
Neumaier N. 3	0,46631755	0,01444287	32,2870494
Ackley N. 2	0,44824343	0,0076167	58,8500838
Shekel 10	0,43105459	0,00701073	61,4849502
PVD	0,43855928	0,00688	63,7440809
TCSD	0,43930163	0,0062259	70,5603409

Tabla 2: Aceleración de NOA en C sobre Matlab

5.3. Aceleración de *pthreads* sobre secuencial

Debido a que el paralelismo implementado divide la cantidad de individuos a calcular por cada generación entre las hebras, los parámetros utilizados en la experimentación original(2) perjudican ligeramente este diseño(para problemas computacionalmente muy costosos no es tan denigrante). ¿Por qué?

Con N = 25 y el total de evaluaciones siendo 35000, se calculan 1400 generaciones de 25 individuos; dividiendo el trabajo entre 4 hebras significaría que cada una solo haría 7, 6, 6 y 6 evaluaciones antes de cada sincronización, aumentando mucho la cantidad de sincronizaciones que se realizan y por tanto la sobrecarga que causan respecto al trabajo realizado entre ellas.

Por esto a continuación se realizan pruebas de la aceleración del programa paralelo sobre el secuencial utilizando N = 25, N = 100, N = 250, N = 500 y N = 1000. Esto no

altera la cantidad total de evaluaciones(35000) que se realizan para alcanzar el óptimo, únicamente afecta a la cantidad de trabajo que realiza cada hebra por generación reduciendo cuántas generaciones se calculan; esto tiene el efecto secundario de que puede afectar a la velocidad de convergencia hacia el óptimo global, deteriorando los resultados. Pero esto también se va a comprobar con el propósito de ver las prestaciones que alcanza la versión paralela con problemas de poca carga computacional.

5.3.1. Sphere

La aceleración es positiva incluso para N = 25.

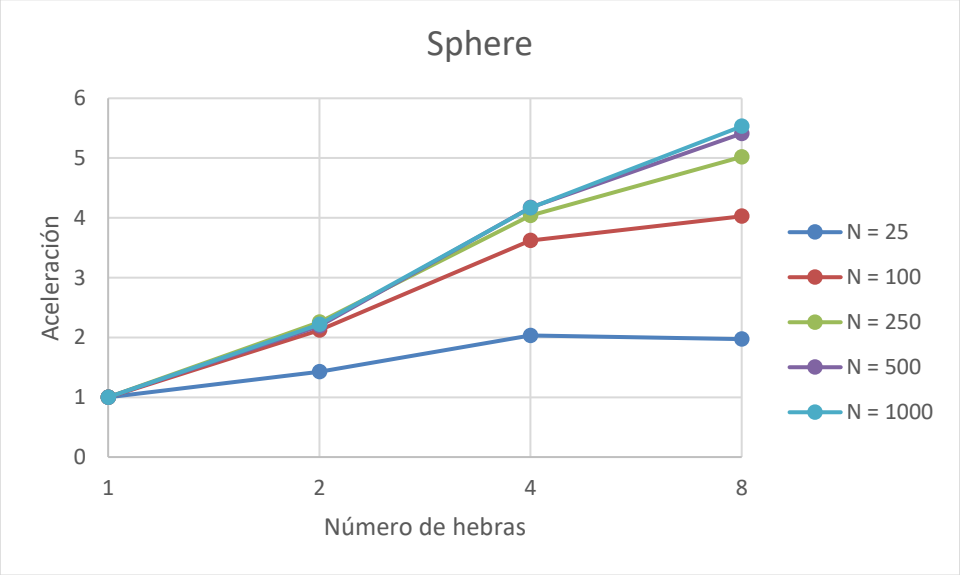

Figura 6: Aceleración sobre la función Sphere

No hay deterioro en las soluciones hasta N = 500, una vez alcanzada prácticamente la máxima aceleración.

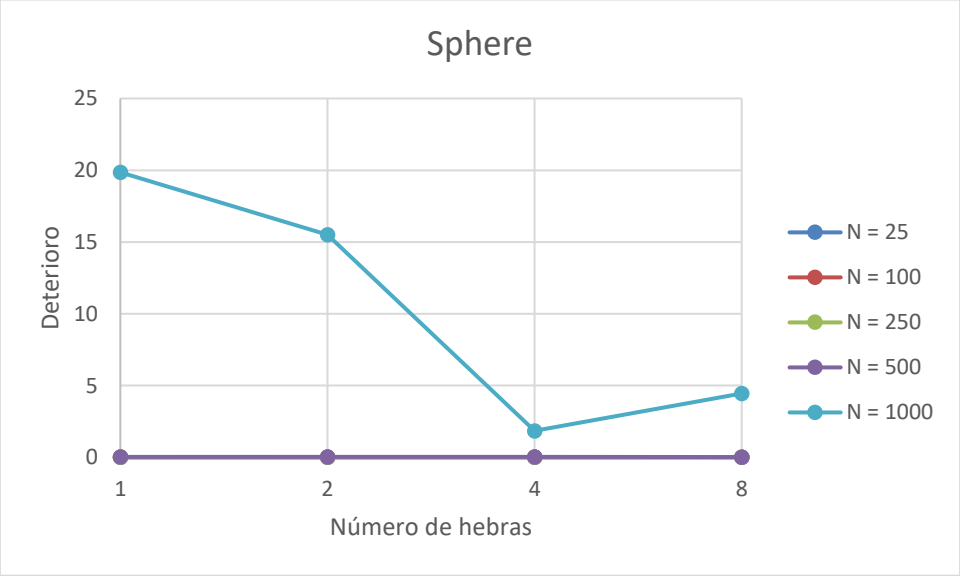

Figura 7: Deterioro de los resultados de la función Sphere

5.3.2. Quartic

La aceleración es positiva incluso para N = 25.

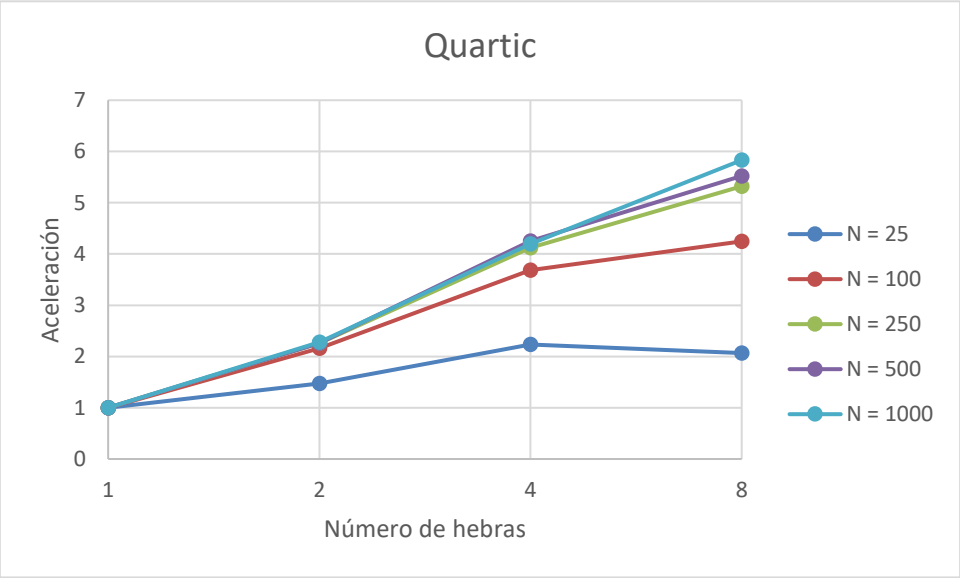

Figura 8: Aceleración sobre la función Quartic

No hay deterioro en las soluciones hasta N = 500, una vez alcanzada prácticamente la máxima aceleración.

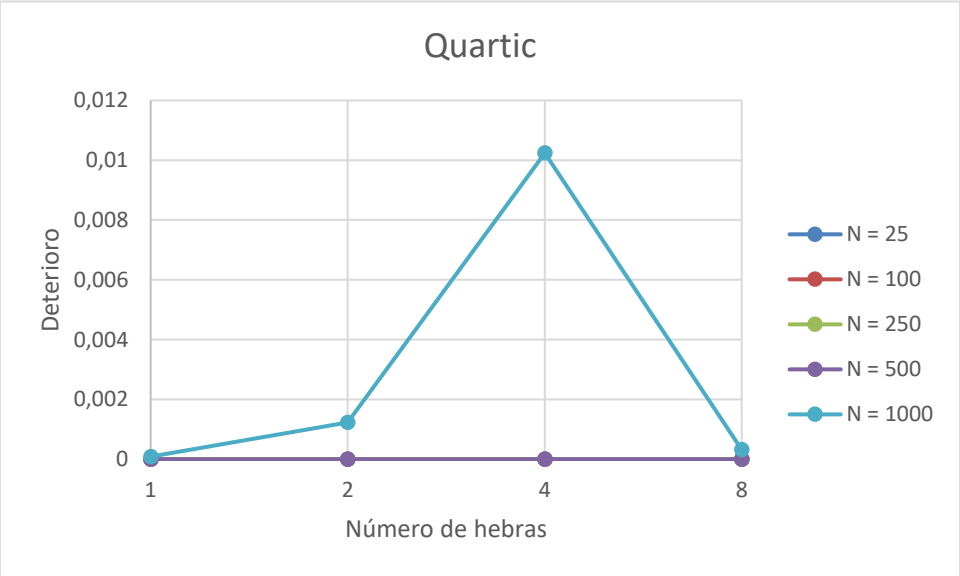

Figura 9: Deterioro de los resultados de la función Quartic

5.3.3. Powell Sum

La aceleración es positiva incluso para N = 25.

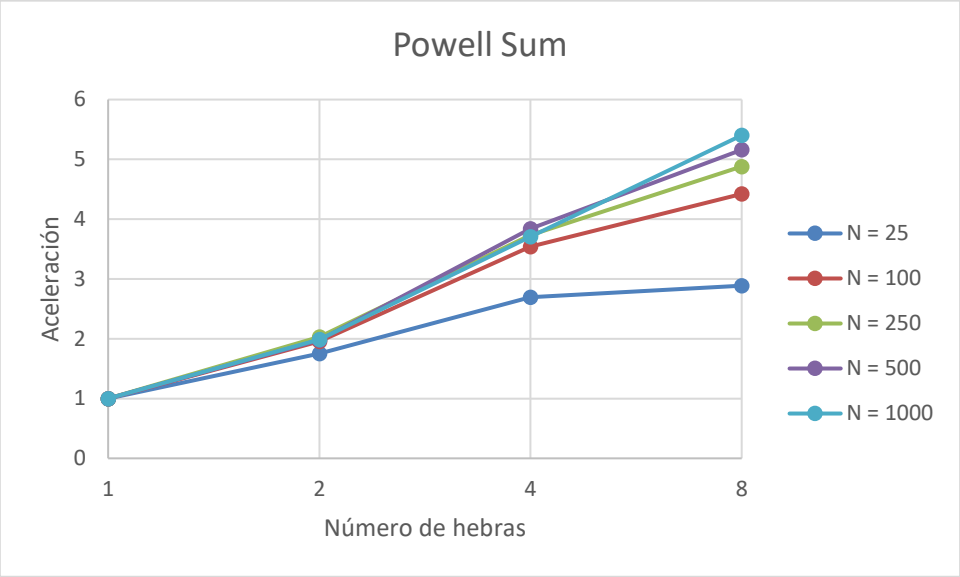

Figura 10: Aceleración sobre la función Powell Sum

No hay deterioro en las soluciones hasta N = 250, una vez alcanzada prácticamente la máxima aceleración.

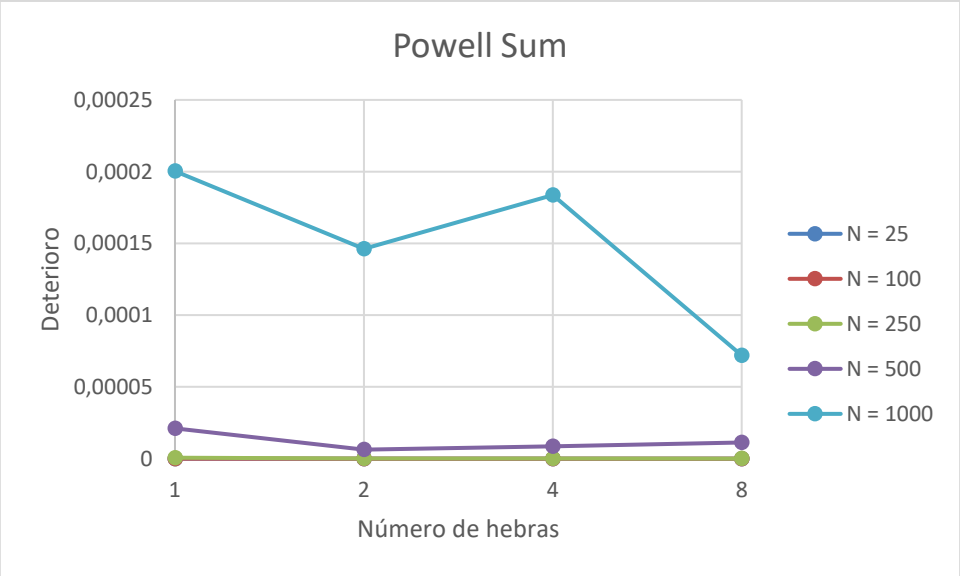

Figura 11: Deterioro de los resultados de la función Powell Sum

5.3.4. Sum Squares

La aceleración es positiva incluso para N = 25.

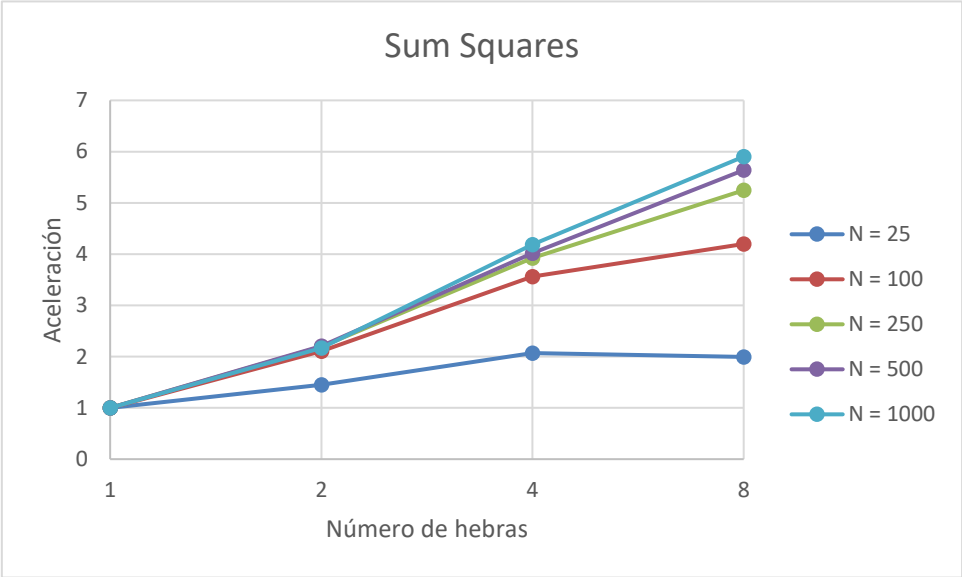

Figura 12: Aceleración sobre la función Sum Squares

No hay deterioro en las soluciones hasta N = 500, una vez alcanzada prácticamente la máxima aceleración.

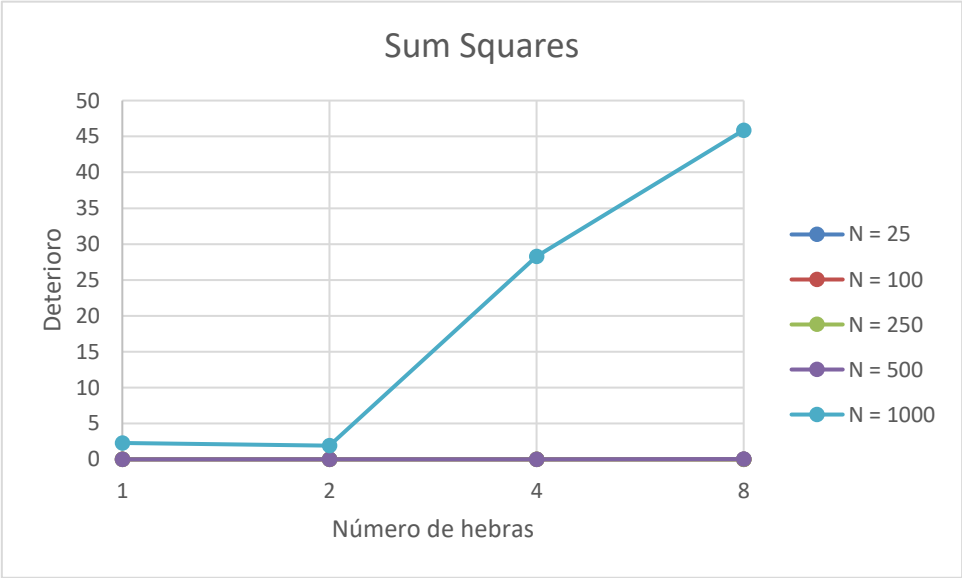

Figura 13: Deterioro de los resultados de la función Sum Squares

5.3.5. Schwefel's 2.20

La aceleración es positiva incluso para N = 25.

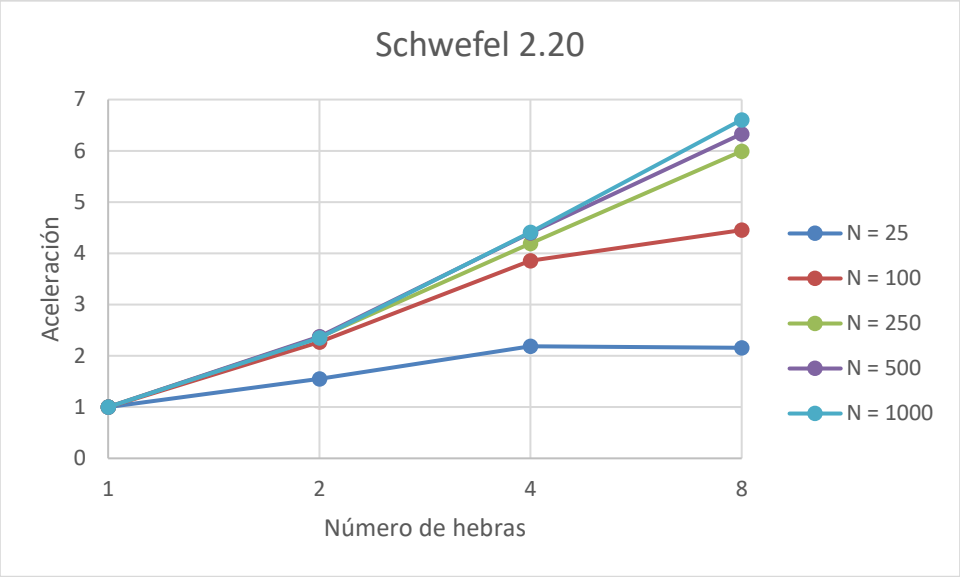

Figura 14: Aceleración sobre la función Schwefel 2.20

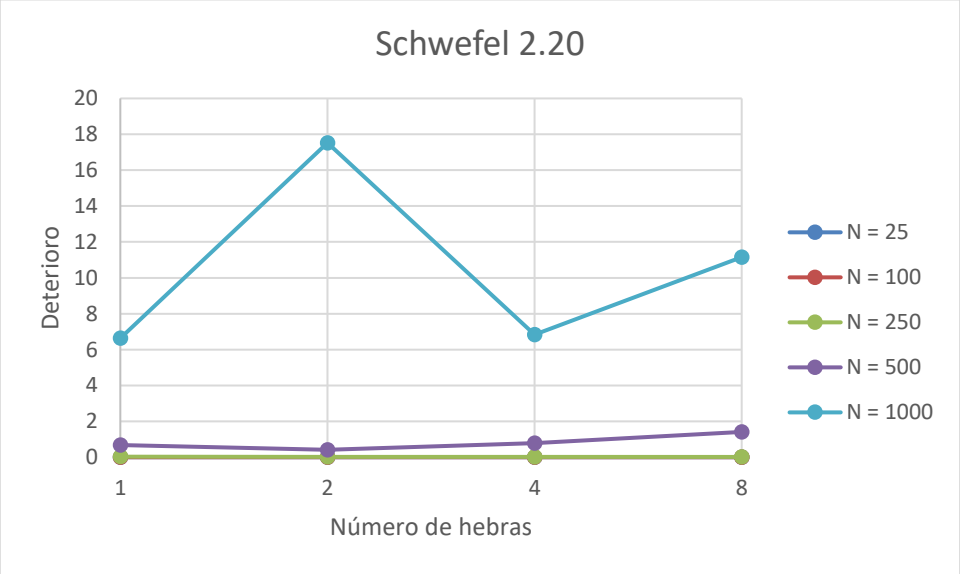

Figura 15: Deterioro de los resultados de la función Schwefel 2. 20

No hay deterioro en las soluciones hasta N = 250, una vez alcanzada prácticamente la máxima aceleración.

5.3.6. Stepint

La aceleración es positiva incluso para N = 25.

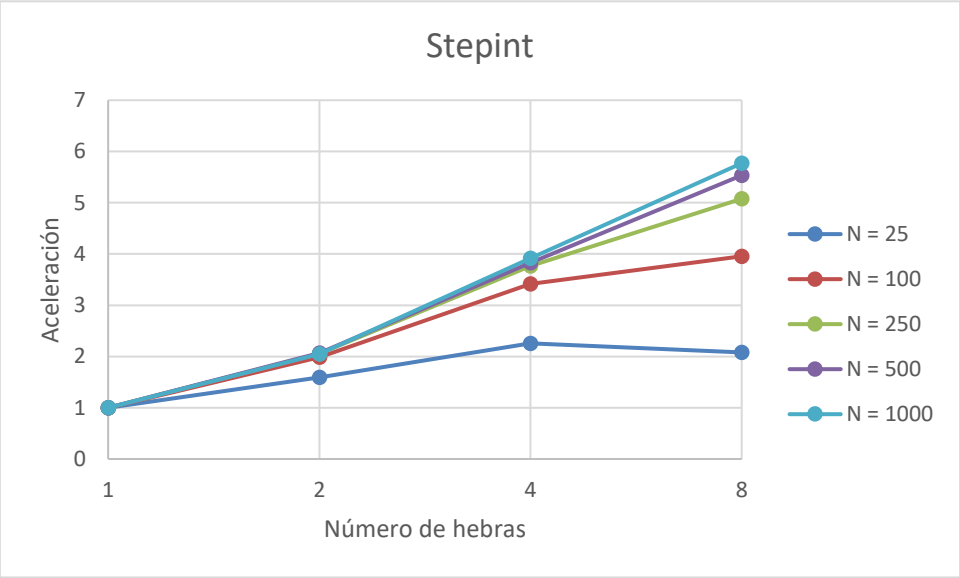

Figura 16: Aceleración sobre la función Stepint

No hay deterioro en las soluciones hasta N = 100, lo cual no proporciona la mejor aceleración para 8 hebras, pero prácticamente la máxima con 4 hebras.

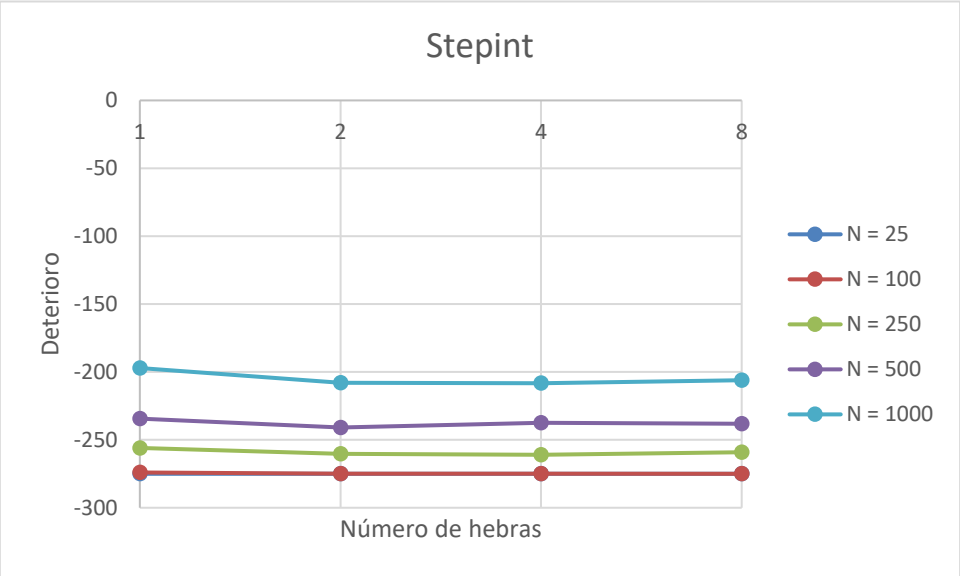

Figura 17: Deterioro de los resultados de la función Stepint

5.3.7. Ridge

La aceleración es positiva incluso para N = 25.

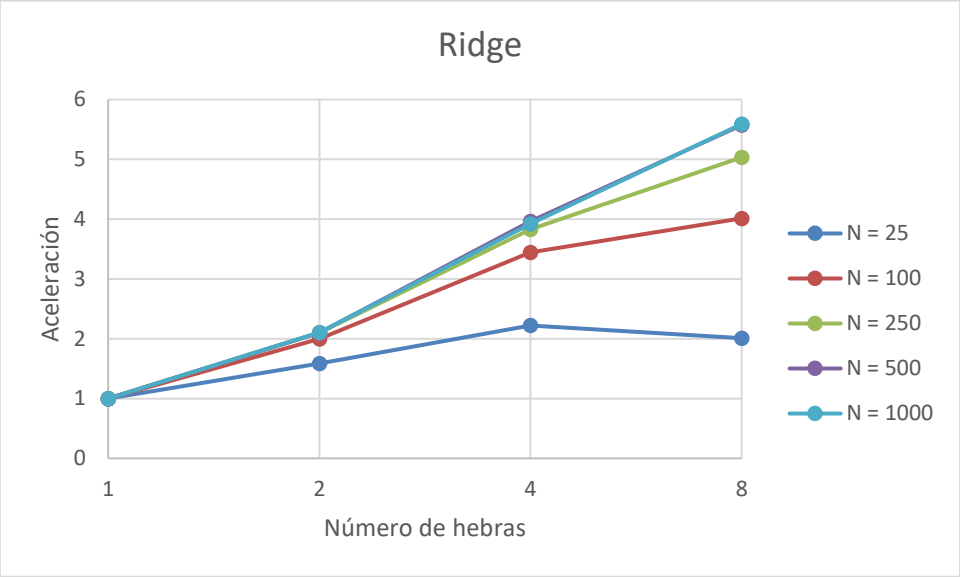

Figura 18: Aceleración sobre la función Ridge

No hay deterioro en las soluciones hasta N = 100, lo cual no proporciona la mejor aceleración para 8 hebras, pero prácticamente la máxima con 4 hebras.

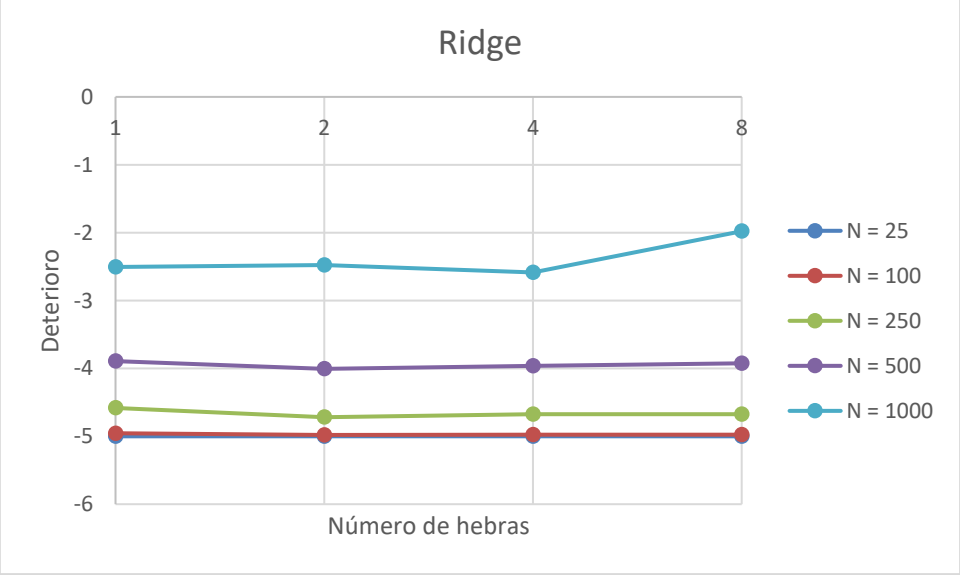

Figura 19: Deterioro de los resultados de la función Ridge

5.3.8. Neumaier's N. 3

No hay aceleración real si no se eleva el tamaño de la población por encima de 25.

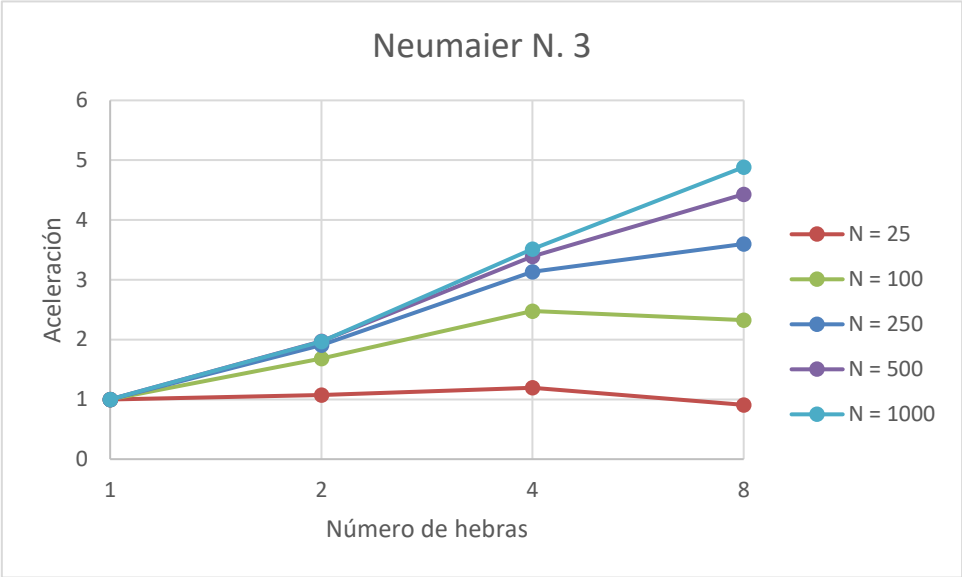

Figura 20: Aceleración sobre la función Neumaier N. 3

No hay deterioro en las soluciones hasta N = 100, lo cual no proporciona la mejor aceleración para 8 hebras, pero prácticamente la máxima con 4 hebras.

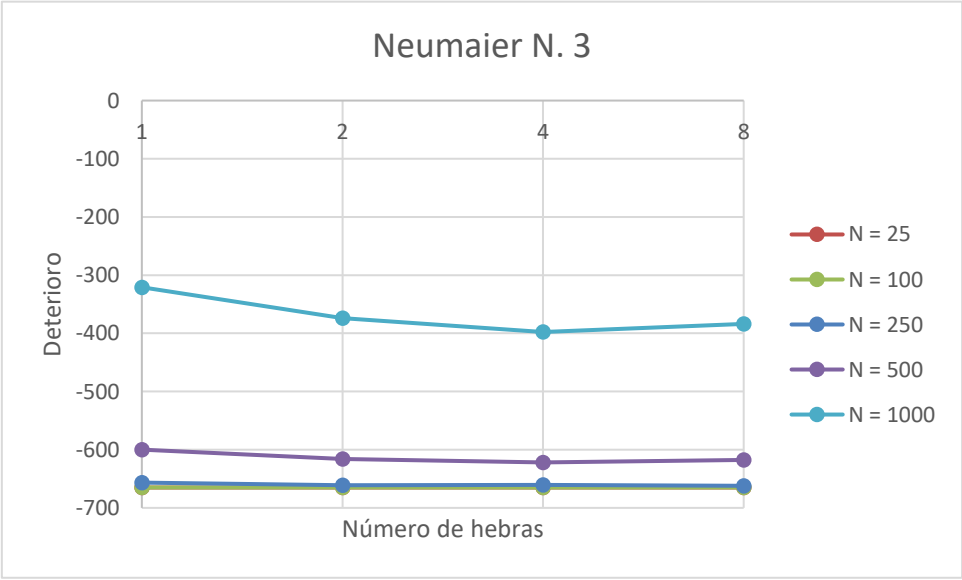

Figura 21: Deterioro de los resultados de la función Neumaier N. 3

5.3.9. Ackley N. 2

No hay aceleración real si no se eleva el tamaño de la población por encima de 25, siendo el paralelismo perjudicial en el rendimiento para N = 25.

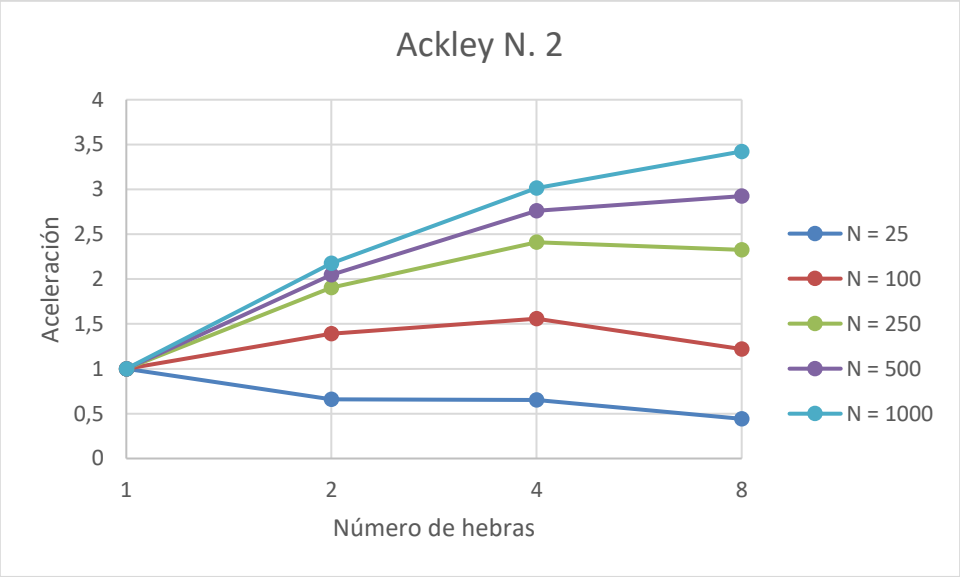

Figura 22: Aceleración sobre la función Ackley N. 2

No hay deterioro en las soluciones hasta N = 250, lo cual no proporciona la mejor aceleración para 8 hebras, pero cerca de la máxima con 4 hebras.

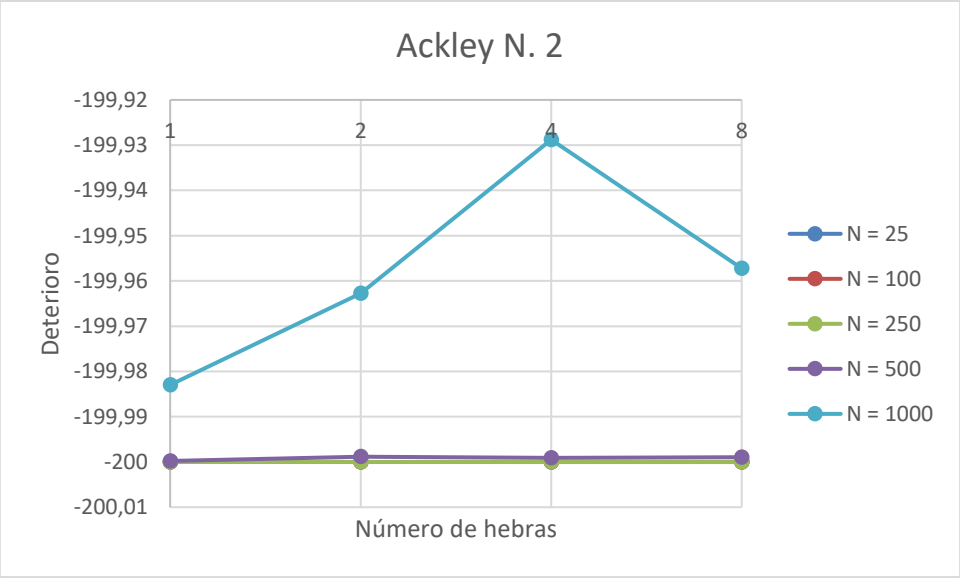

Figura 23: Deterioro de los resultados de la función Ackley N. 2

5.3.10. Shekel 10

No hay aceleración real si no se eleva el tamaño de la población por encima de 25, siendo el paralelismo perjudicial en el rendimiento para N = 25.

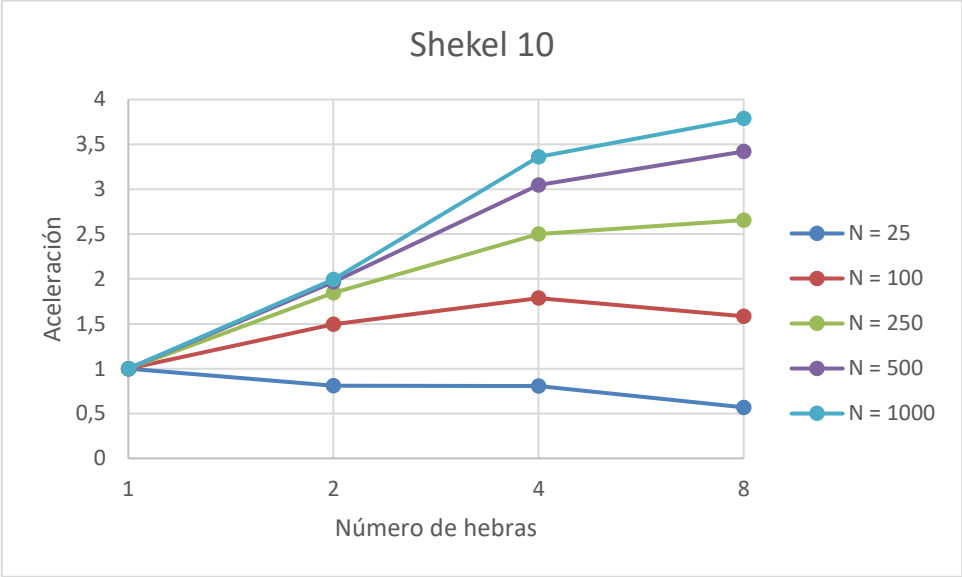

Figura 24: Aceleración sobre la función Shekel 10

No hay deterioro en las soluciones hasta N = 250, lo cual no proporciona la mejor aceleración para 8 hebras, pero cerca de la máxima con 4 hebras.

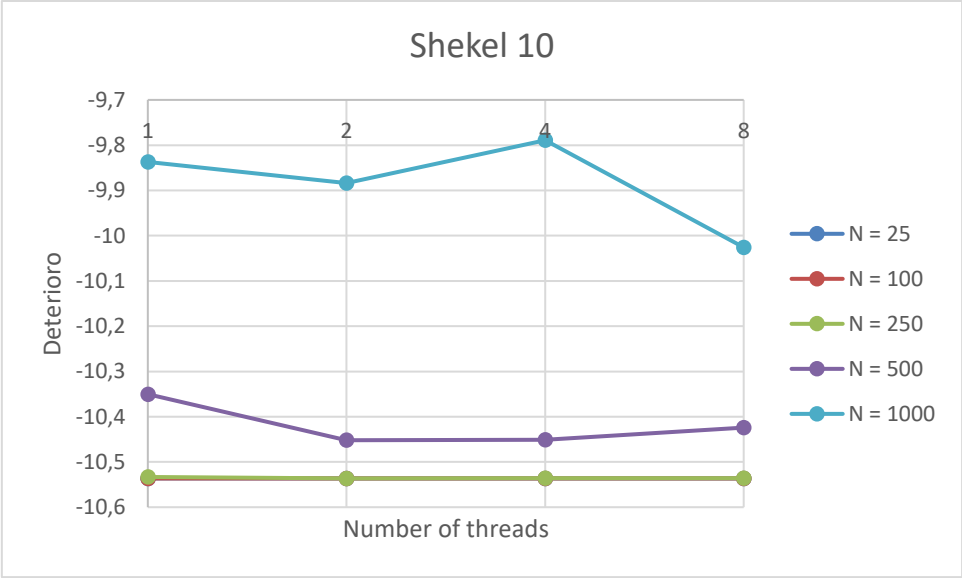

Figura 25: Deterioro de los resultados de la función Shekel 10

5.3.11. PVD

No hay aceleración real si no se eleva el tamaño de la población por encima de 25, siendo el paralelismo perjudicial en el rendimiento para N = 25.

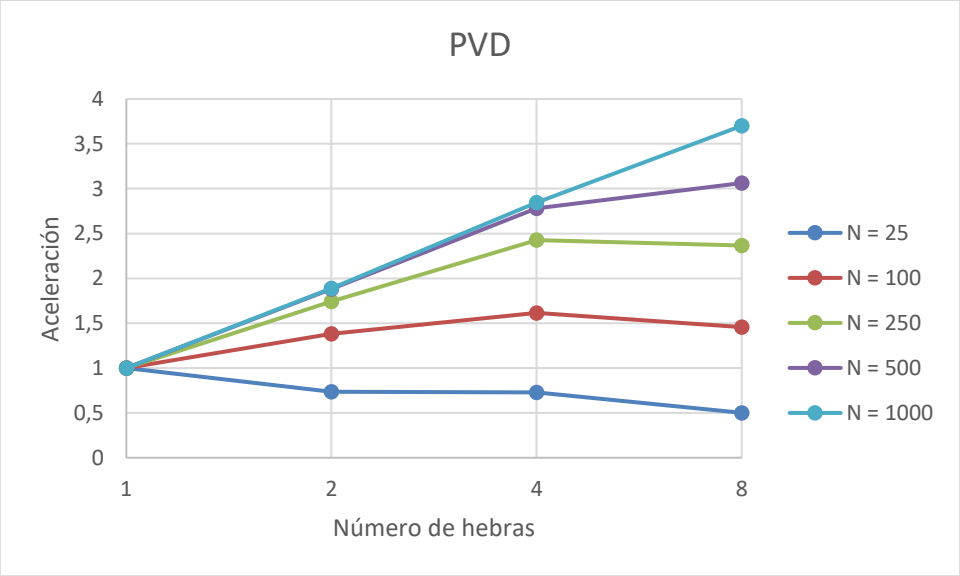

Figura 26: Aceleración sobre el problema de PVD

No hay deterioro en las soluciones hasta N = 100, y en este caso la aceleración ni siquiera mejora con más hebras.

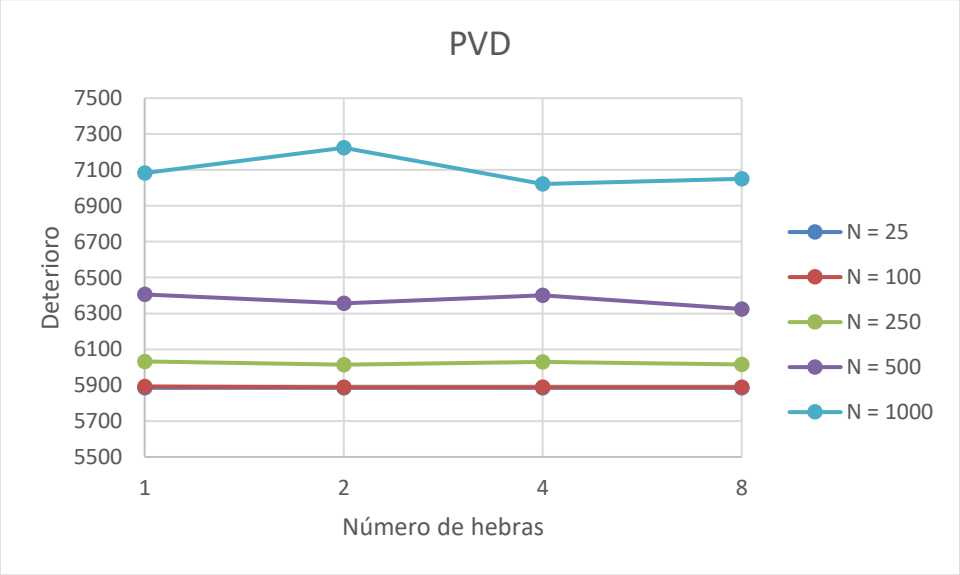

Figura 27: Deterioro de los resultados del problema de PVD

5.3.12. TCSD

No hay aceleración real si no se eleva el tamaño de la población por encima de 25, siendo el paralelismo perjudicial en el rendimiento para N = 25.

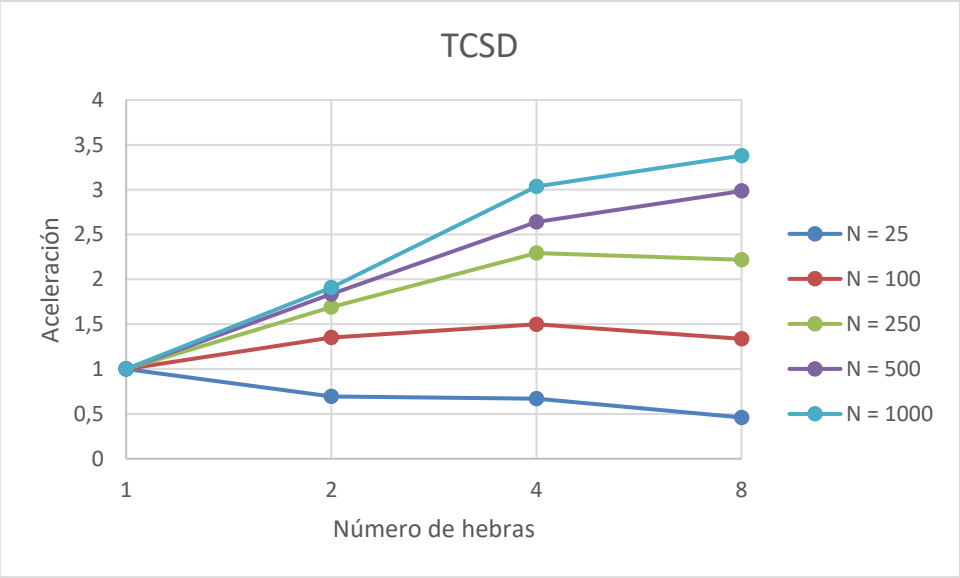

Figura 28: Aceleración sobre el problema de TCSD

Hay un deterioro casi inmediato al aumentar el tamaño de la población, por lo que el rendimiento del algoritmo paralelo es muy pobre en este caso.

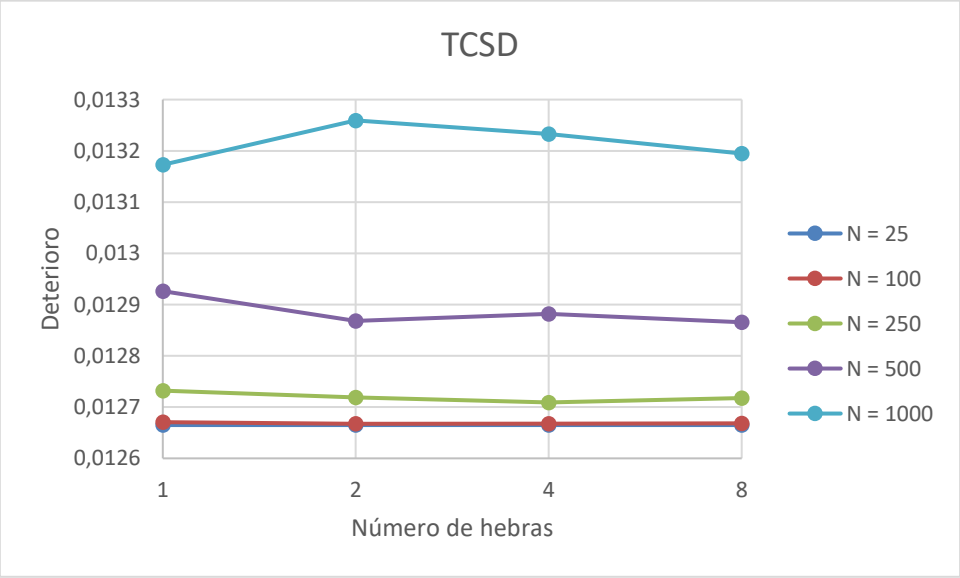

Figura 29: Deterioro de los resultados del problema de TCSD

5.3.13. Artificial

En un problema con mayor carga computacional, el tamaño de la población representa un menor efecto en el rendimiento de la versión paralela.

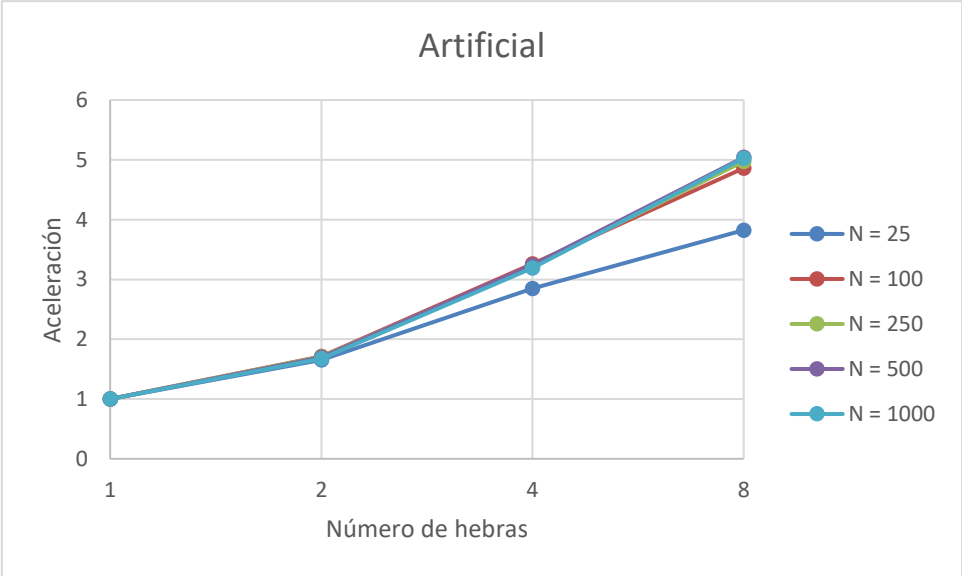

Figura 30: Aceleración sobre el problema con carga artificial

Hay un deterioro casi inmediato al aumentar el tamaño de la población; pero este es innecesario, ya que el rendimiento ya es muy alto con N = 25.

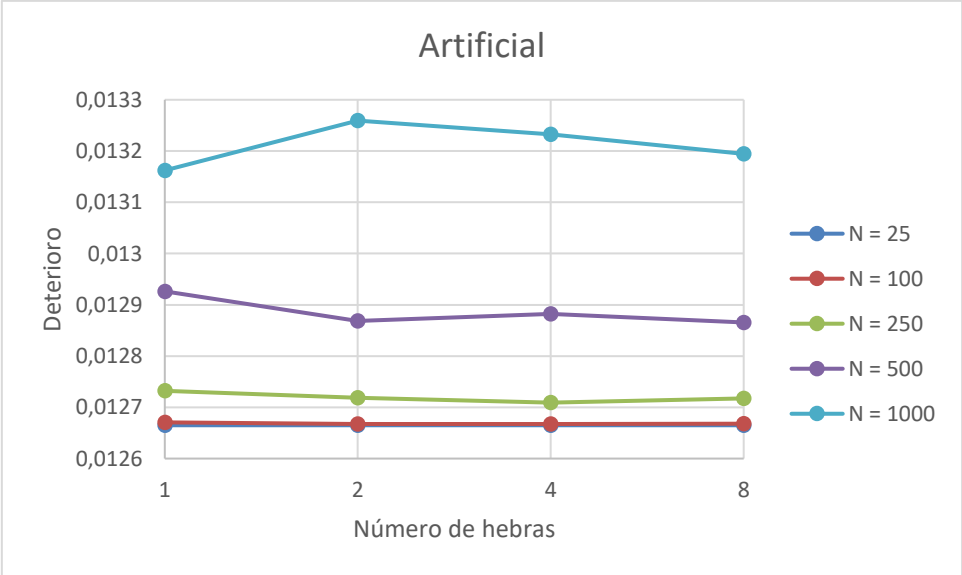

Figura 31: Deterioro de los resultados del problema con carga artificial

5.4. Versiones alternativas con pthreads(terminar)

El tiempo de ejecución es ligeramente inferior con menos memcpy…

Conclusiones y trabajo futuro

6.1. Conclusiones

6.1.1. Matlab o C

Los resultados muestran que el algoritmo en C es por lo menos 10 veces más rápido que Matlab, lo que sugiere que independientemente del problema que se tuviese que optimizar el uso de NOA en C es mucho más eficiente que en Matlab. Los resultados son equivalentes en todas las funciones por lo que se puede afirmar que el funcionamiento es el mismo y no existen aspectos de la implementación que sesguen la eficiencia de ninguna de las versiones.

6.1.2. Merece la pena paralelizar

Se puede apreciar que la aceleración en funciones más costosas, concretamente en todas aquellas que trabajan con dimensión mayor D = 50, mejora las prestaciones del algoritmo. Por otro lado, en problemas con dimensiones pequeñas, D < 15, la sobrecarga es mayor que el reparto de tareas; demostrando que el paralelismo se refleja en la carga puesta por N(tamaño de población), C(coste de la función objetivo) y D(dimensión del problema) tal y como se había supuesto. ¿Cómo?

Bueno, el coste de las funciones benchmark(excluyendo la que tiene carga artificial) es muy bajo, siendo casi despreciable, por lo que no es razón suficiente para paralelizar. Si la dimensión es pequeña como se ha dicho, de NCD acaba importando solo N, el tamaño del problema. Y por esto es que se decidió realizar la experimentación variando el tamaño de N, que como se puede apreciar supone una mejoría sustancial en la aceleración del algoritmo paralelo respecto al secuencial. Por otro lado el aumento de N conlleva consecuencias, los valores óptimos se deterioran, en algunos casos inmediatamente; lo cual también se preveía antes de la comprobación empírica.

En cualquier caso esto no indica que el paralelismo sea peor que la secuencialidad, los casos anteriores ya hemos visto qué no se benefician de un C grande, y si el valor de D es mayor que 15 el rendimiento sigue mostrándose mejor aún con funciones con poco coste computacional. Esto se observa para la función de Neumaier N. 3, cuya dimensión es D = 15 y la aceleración es 1 respecto al secuencial; lo que indica que conforme aumente D se irá acercando al rendimiento observado para las funciones con D = 50.

Se puede afirmar entonces que para problemas computacionalmente simples con pocas dimensiones como las funciones de prueba o *benchmark*, es mejor utilizar una versión secuencial del algoritmo. Mientras que si tienen dimensiones más grandes, aún con poca carga computacional, la versión paralela siempre es superior.

6.1.3. Problemas computacionalmente costos(terminar)

Para problemas con mucha carga computacional, como el que se ha implementado con una carga artificial, se ve una clara superioridad de la versión paralela sobre la secuencial sin necesidad de alterar el tamaño de la población.

De hecho, prácticamente se llega a la aceleración máxima del programa sobre la función para N = 25, lo que significa que sin distorsionar las soluciones la versión paralela es muy superior a la secuencial para problemas computacionalmente costosos.

6.2. Conclusión final

El objetivo de un algoritmo paralelo es reducir el tiempo de computación de ejecuciones muy prolongadas, habiendo problemas de optimización que tardan horas o incluso días en resolverse con un optimizador; el rendimiento inferior de un programa paralelo respecto al secuencial en ejecuciones de milisegundos no solo es esperable, sino que es lo normal.

Por esto se puede ver el resultado como un éxito; ya que se han cumplido los objetivos de llegar a una implementación paralela en C para computación de altas prestaciones del algoritmo de optimización Nizar(NOA), que otorga mayores prestaciones que la versión secuencial de C y esta a su vez mejora las prestaciones del diseño original en Matlab.

Se tiene finalmente un algoritmo con muy buenas prestaciones todos los ámbitos; buena optimización de problemas como se ve en el artículo original(2), buena eficiencia por el uso de un lenguaje de más bajo nivel(concretamente C) y buena escalabilidad debido al paralelismo implementado.

6.3. Futuro

El siguiente paso sería la implementación MPI para añadir paralelismo con diferentes procesos que deberían permitir paralelismo real siempre; algo que se encontraba como objetivo inicial pero no se ha podido terminar por falta de tiempo.

También se podría experimentar con problemas de mucha más carga computacional y realizar comparaciones con otros algoritmos actuales sobre problemas de optimización muy complejos.

Referencias

(1) Lindfield, G., & Penny, J. (2017). *Introduction to nature-inspired optimization*. Academic Press.

(2) Khouni, S. E., & Menacer, T. (2024). Nizar optimization algorithm: a novel metaheuristic algorithm for global optimization and engineering applications. *The Journal of Supercomputing*, *80*(3), 3229-3281.

(3) Salhi, S. (2017). Heuristic search: The emerging science of problem solving.

(4) Cruz, N. C., Redondo, J. L., Ortigosa, E. M., & Ortigosa, P. M. (2022, July). On the design of a new stochastic meta-heuristic for derivative-free optimization. In *International Conference on Computational Science and Its Applications* (pp. 188-200). Cham: Springer International Publishing.

(5) Matsumoto, M., & Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *8*(1), 3-30.

(6) Fisher, R. A., & Yates, F. (1938). Statistical tables for biological, agricultural aad medical research.

(7) https://www.sfu.ca/~ssurjano/spheref.html

(8) https://search.r-project.org/CRAN/refmans/smoof/html/makePowellSumFunction.html

(9) https://www.sfu.ca/~ssurjano/sumsqu.html

(10) https://infinity77.net/global_optimization/test_functions_nd_S.html#go_benchmark.Schwefel20

(11) https://al-roomi.org/benchmarks/unconstrained/n-dimensions/195-stepint-function

(12) https://al-roomi.org/benchmarks/unconstrained/n-dimensions/247-neumaier-s-function-no-03-or-trid-s-function

(13) https://www.indusmic.com/post/ackleyn-2function

(14) https://www.sfu.ca/~ssurjano/shekel.html

(15) https://neorl.readthedocs.io/en/latest/examples/ex8.html

(16) Tzanetos, A., & Blondin, M. (2023). A qualitative systematic review of metaheuristics applied to tension/compression spring design problem: Current situation, recommendations, and research direction. *Engineering Applications of Artificial Intelligence*, *118*, 105521.