TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

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

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 5 de septiembre de 2025

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: Optimización, Metaheurística, Método basado en poblaciones, Programación Paralela, Nizar

Resumen

En la Ciencia, la Ingeniería y la Industria suelen aparecer problemas de optimización en los que una función modela cierto aspecto de la realidad y hace falta encontrar, según el caso, los puntos máximos o mínimos de su dominio.

En general, según las variables implicadas y la estructura matemática de la función, existen distintos métodos para abordar este tipo de problemas, garantizando incluso soluciones óptimas. Sin embargo, especialmente con problemas complejos, es frecuente tener que renunciar a la optimalidad y usar heurísticas. Este tipo de métodos destacan por poder encontrar soluciones suficientemente buenas con un esfuerzo computacional aceptable, y definen un activo y prolífico campo de investigación.

En este trabajo se aborda el estudio y paralelización de un reciente método propuesto en esta línea, Nizar. Este algoritmo, que reportó resultados mejores a otros del estado del arte, ha cosechado un número de citas destacable en poco tiempo. Busca ser fácil de usar y necesitar pocos parámetros en contraposición a otras heurísticas en las que, a veces, encontrar una buena configuración es un problema en sí mismo.

En concreto, se pretende ampliar dicho trabajo haciendo una implementación de código abierto del método y que aplique, además, computación de altas prestaciones, lo que no se trataba en el artículo original.

Según los resultados obtenidos, la mejor implementación desarrollada es hasta 70 veces más rápida que una versión programada en MATLAB, y lo es sin perder calidad en los resultados. Además, esta es capaz de explotar la disponibilidad de múltiples núcleos de procesamiento, logrando acelerar la resolución de problemas hasta 5 veces en una máquina de 8 núcleos en memoria compartida, y hasta 13 veces combinando 16 núcleos entre dos máquinas en memoria distribuida.

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

Guillermo Dalda

Keywords: Optimization, Metaheuristic, Population-based Method, Parallel Computing, Nizar

Abstract

In Science, Engineering and Industry, optimization problems often arise where a function describes a certain aspect of reality and it is needed to find, depending on the case, the maximum or minimum points of its domain.

In general, depending on the variables involved and the mathematical structure of the function, there exists different methods to tackle these kinds of problems, even guaranteeing optimal solutions. However, especially with complex problems, it is frequent to have to renounce optimality in favor of heuristics. These types of methods stand out for being able to find good enough solutions with an acceptable computational effort, and they define an active and prolific research field.

This work addresses the study and parallelization of a recent method proposed in this line, Nizar. This algorithm, which reported better results than others from the state of the art, has collected a notable number of quotes in little time. It seeks to be easy to use and to need few parameters in contrast to other heuristics in which, sometimes, finding a good configuration is a problem in itself.

Specifically, it is intended to expand said work by doing an open source implementation of the method and that applies, also, high-performance computing, which was not discussed in the original article.

By the results obtained, the best implementation developed is up to 70 times faster than a version coded in MATLAB, and it is without losing quality in the results. On top of that, it's able to exploit the availability of multiple processing units, being able to accelerate problem resolution up to 5 times in a shared memory machine with 8 cores, and up to 13 times combining 16 cores between two distributed memory machines.

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 5 de septiembre de 2025.

D. **Nicolás Calvo Cruz**, Profesor del Departamento de Ingeniería de Computadores, Automática y Robótica de la Universidad de Granada.

D.ª **Eva Martínez Ortigosa**, Profesora del Departamento de Ingeniería de Computadores, Automática y Robótica de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Desarrollo y evaluación de una implementación de altas prestaciones de la metaheurística de optimización Nizar***, ha sido realizado bajo su supervisión por **Guillermo Dalda del Olmo**, 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 5 de septiembre de 2025.

Los directores:

Nicolás Calvo Cruz **Eva Martínez Ortigosa**

Agradecimientos

En primer lugar, a mis tutores Nicolás Calvo Cruz y Eva Martínez Ortigosa, que 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.

A la profesora Maribel García Arenas por dejarme utilizar la máquina GenMagic y al grupo Supercomputación - Algoritmos (SAL) de la Universidad de Almería por permitirme usar temporalmente una de sus máquinas.

Contenidos

Figuras

Pseudocódigo

Tablas

1. Introducción

La optimización es una rama de las matemáticas aplicadas que estudia la identificación de los extremos de funciones en su dominio. Estas, a menudo, modelan algún aspecto de la realidad, y sirven para definir problemas cuya solución se encuentra en algún extremo de dichas funciones.

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 reducir costes. Son perspectivas diferentes que al final buscan la solución óptima (de ahí surge el nombre de optimización). Y por esta razón a la función que define el problema a optimizar se le llama función objetivo, función fitness o función de coste (1).

En este trabajo se trata un algoritmo metaheurístico basado en poblaciones llamado NOA (*Nizar Optimization Algorithm*) (2) que busca el valor mínimo de la función objetivo que se le indique. Como resultado, el método devuelve tanto el mejor valor encontrado (idealmente el mínimo global del dominio de la función), como el punto en el que se da, es decir, con qué valor de las variables se encuentra. Es además importante destacar que este método espera que las variables sean continuas y que la función pueda evaluarse en todo su dominio. Sin embargo, no tiene requisitos adicionales en aspectos como la derivabilidad, y se limita a obtener y comparar puntos de la función. Por tanto, NOA se clasificaría como un algoritmo de caja negra (3).

A lo largo de las siguientes secciones se hablará de la relevancia de la optimización (con especial mención al algoritmo NOA), de la motivación del proyecto, de los objetivos planteados, de cómo se han ido completando estos objetivos y qué recursos se han utilizado para ello.

1.1. Relevancia y motivación

Muchos problemas del mundo real son resolubles de forma óptima mediante técnicas de optimización clásicas como la programación lineal. Sin embargo, estas dejan de ser aptas cuando la complejidad del problema requiere una cantidad de tiempo inviable para hallar el óptimo, solo se encuentran óptimos locales, o la función objetivo del problema no es del tipo adecuado (no es lineal, no tiene una expresión analítica cerrada…). Para estos casos se recurre a los métodos heurísticos de optimización (1) (6).

Los métodos heurísticos de optimización, o simplemente heurísticas, son algoritmos que se apoyan en ideas y estrategias de exploración aproximada del espacio de soluciones. Su eficacia no siempre se puede demostrar formalmente, pero se sabe que permiten obtener soluciones adecuadas con un esfuerzo razonable. De hecho, su propio nombre intenta plasmar esta orientación práctica a la obtención de resultados. Concretamente, la palabra heurística proviene del término "heuriskein", que en griego antiguo significa descubrir y explorar (4). Suelen utilizar métodos estocásticos, lo cual implica el uso de variables aleatorias, y tener múltiples parámetros que se deben ajustar según el problema en cuestión (5).

Esta orientación a la obtención de soluciones con un esfuerzo razonable, incluso sin garantías de optimalidad, hace que el uso de heurísticas esté muy extendido en muchos problemas que se consideraban difíciles de resolver en campos de la Ciencia y la Ingeniería (6) (7).

Posiblemente, una de las estrategias heurísticas más simples y conocidas sea la voraz (*greedy*), que compone la solución de un problema escogiendo la mejor opción para cada subproblema del mismo (6). Sin embargo, existe una gran variedad de métodos mucho más complejos y que se apoyan en diversos principios, desde la aleatoriedad, hasta la simulación de procesos físicos y biológicos (1)(6).

Entre las múltiples heurísticas que existen (4), NOA destaca por:

- Su reciente aparición, publicado originalmente en septiembre de 2023, y alto impacto, acumulando ya 29 citas.
- Su sencillez de uso, sin parámetros específicos.
- Sus resultados experimentales, en los que se mostró superior a otros 15 algoritmos del mismo tipo al optimizar tanto funciones sintéticas sin restricciones como problemas de ingeniería con restricciones.
- Sus nuevas técnicas que mejoran la exploración y explotación, alcanzando nuevas zonas del espacio de búsqueda y encontrando el mejor punto de las ya conocidas (8).

Estas últimas son las contribuciones que el artículo original presenta como las más relevantes. Sin embargo, y a pesar de lo prometedor de los resultados mostrados por los autores de NOA, no se ofrece una implementación de código abierto del método. Además, los autores originales optaron por trabajar en Matlab. Esta decisión facilita la codificación y el prototipado, pero también suele lastrar el rendimiento y limitar su uso para problemas muy costosos (8).

1.2. Objetivos

De lo expuesto anteriormente se deriva el objetivo principal del presente trabajo: la realización de una implementación de código abierto del método NOA que sea fiel a la descripción hecha del algoritmo y que, además, aplique técnicas de computación de altas prestaciones. La consecución de dicho objetivo implica, a su vez, la persecución de los siguientes objetivos parciales:

1. **Estudio del algoritmo y la optimización.** Se quiere estudiar los fundamentos de la optimización, así como comprender la propuesta del algoritmo NOA. Es fundamental analizar la descripción del algoritmo original para poder entender su funcionamiento.
2. **Implementación.** Se busca realizar una implementación propia del algoritmo en C. Este lenguaje permite la programación a más bajo nivel de abstracción que MATLAB para mejorar el rendimiento del programa y también aumentar la eficiencia gracias a las herramientas para la paralelización que existen.
3. **Contraste del funcionamiento del algoritmo.** Se quiere comprobar del 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. Es de vital importancia asegurar que la implementación del algoritmo se comporta como se describe en el artículo original (2).

4. **Implementación con paralelismo.** Se desean realizar varias versiones paralelas del método. Una de ellas utilizará *pthreads* (POSIX threads) (9) para mejorar el rendimiento en multiprocesadores. La otra combinaría *pthreads* y MPI (*Message Passing Interface* o interfaz de paso de mensajes) (10), siendo así compatible con multicomputadores y mejorando la escalabilidad del algoritmo en sistemas de computación de altas prestaciones.
5. **Comparación de las distintas versiones.** Se quieren comparar las distintas implementaciones en cuanto a la aceleración y los resultados que proporcionan. En este sentido es clave comprobar también que el aumento del rendimiento computacional no degrada la eficacia del método.
6. **Elaboración de una memoria (Documentación).** Se pretende escribir un documento que recoja el trabajo realizado y los resultados obtenidos.

Estos objetivos parciales pueden plasmarse directamente en tareas que realizar, y cuya planificación se detalla en la siguiente sección.

1.3. Planificación

El desarrollo del trabajo ha llevado unas 320 horas en total y se ha realizado de forma progresiva durante aproximadamente 47 semanas. En este tiempo se han completado los diferentes objetivos tras la realización de las tareas correspondientes. Esto se puede reflejar mediante el diagrama de Gantt que se muestra en la Figura 1.

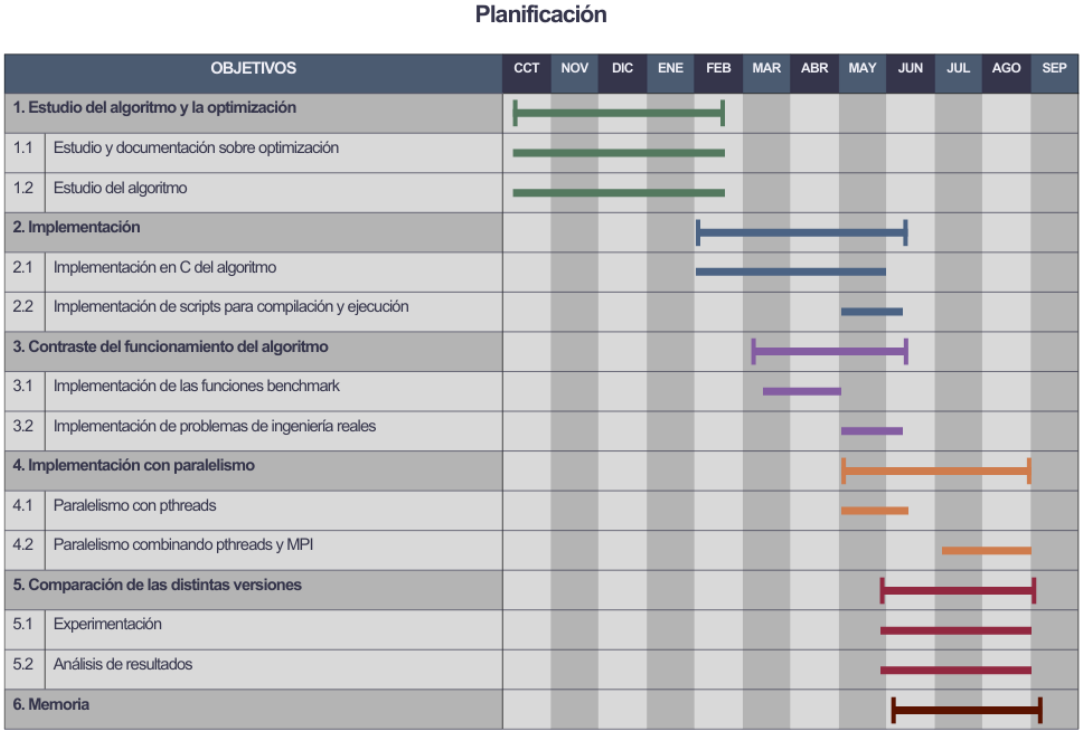

Figura 1: Diagrama de Gantt.[1]

[1] Elaboración propia

Seguidamente se procede a describir en mayor detalle el tiempo dedicado a cada objetivo, y las tareas en las que se traducen.

1er Objetivo. Estudio de la optimización y el algoritmo

Este objetivo se completó en un período de 19 semanas, desde el 14 de octubre de 2024 hasta el 24 de febrero de 2025. Se llevaron a cabo las siguientes tareas para lograrlo:

- Documentación y preparación sobre optimización, incluyendo el estudio de un optimizador simple y su implementación. Concretamente, se implementó una búsqueda aleatoria pura (11).
- Lectura y estudio del algoritmo Nizar (2).

Se realizó una reunión con los directores al comienzo del período y otra al final. A lo largo del mismo se invirtieron aproximadamente 30 horas; equivalente a 1 hora y media a la semana.

2º Objetivo. Implementación

Este objetivo se completó en un período de 15 semanas, desde el 24 de febrero de 2025 hasta el 9 de junio de 2025. Se llevaron a cabo las siguientes tareas para lograrlo:

- Implementación del programa principal.
- Implementación del algoritmo.
- Múltiples reorganizaciones de código de los ficheros del programa.
- Implementación de *scripts* para compilación, ejecución y experimentación.
- Adaptación al programa del generador de números aleatorios Mersenne Twister (12).
- Corrección de errores en la implementación.

A lo largo de este período se mantuvieron reuniones con los directores cada dos semanas. Se invirtieron aproximadamente 90 horas; equivalente a 6 horas a la semana.

3er Objetivo. Contrastar el funcionamiento del algoritmo

Este objetivo se completó en un período de 6 semanas, desde el 30 de abril de 2025 hasta el 9 de agosto de 2025. Se llevaron a cabo las siguientes tareas para lograrlo:

- Implementación de las 10 funciones *benchmark*.
- Implementación de los 2 problemas de ingeniería reales.
- Contraste de implementación entre la versión de C y la referencia de MATLAB realizada por los directores.
- Contraste del funcionamiento de las versiones paralelas respecto de las secuenciales.

A lo largo de este período se compartieron las mismas reuniones con los directores del anterior objetivo. Se invirtieron aproximadamente 25 horas, lo que sería equivalente

a 4 horas a la semana.

4º Objetivo. Paralelización del algoritmo

Este objetivo se completó en un período de 18 semanas, desde el 12 de mayo de 2025 hasta el 31 de agosto de 2025. Se llevaron a cabo las siguientes tareas para lograrlo:

- Implementación de *pthreads* sobre el algoritmo con cerrojos.
- Sustitución de cerrojos para *pthreads* sobre el algoritmo por barreras.
- Implementación de MPI sobre el programa principal en combinación con el uso de pthreads previamente planteado.
- Corrección de errores y perfilado de las versiones paralelas.

A lo largo de este período se compartieron las mismas reuniones con los directores del segundo objetivo. Se invirtieron aproximadamente 55 horas, lo que equivaldría a 3 horas a la semana.

5º Objetivo. Comprobación del rendimiento

Este objetivo se completó en un período de 12 semanas, desde el 9 de junio de 2025 hasta el 2 de septiembre de 2025. Se llevaron a cabo las siguientes tareas para lograrlo:

- Experimentación y comparación de las versiones secuencial y paralelas.
- Medición de tiempos y comparación de las versiones secuencial y paralelas.
- Búsqueda de un problema más apto para las comprobaciones de rendimiento, implementación y medición de tiempos de este.
- Experimentación y comparación entre las versiones paralelas.

A lo largo de este período no se realizaron reuniones con los directores, pero si hubo comunicación y coordinación. Se invirtieron aproximadamente 35 horas, en una distribución equivalente a 3 horas a la semana.

6º Objetivo. Elaboración de una memoria (Documentación).

La elaboración de la memoria comenzó el 9 de junio y se llevó a cabo durante 13 semanas hasta el 5 de septiembre de 2025. Se invirtieron aproximadamente 85 horas; equivalente a 6 horas a la semana.

Esta temporización se ha ajustado generalmente bien y sin grandes contratiempos con la planificación realizada.

1.4. Recursos y costes

En la realización del proyecto se ha hecho uso de distintas herramientas. Se detallan a continuación distinguiendo tres categorías, hardware, software y otros. Además, la sección concluye con un desglose de los costes que ha supuesto llevar a cabo el proyecto.

1.4.1. Hardware

La implementación y comprobación del funcionamiento del algoritmo se ha realizado en un sistema con Intel Core(TM) i3-1005G1 @ 1.20GHz (con 2 núcleos físicos y 4 lógicos, y memoria compartida) 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 con hilos respecto al secuencial se ha realizado en el servidor de GenMagic de la ETSIIT, que tiene un Intel(R) Core(TM) i7-9700F CPU @ 3.00GHz (con 8 núcleos físicos y 8 lógicos, y memoria compartida) y 31GiB de memoria principal.

Se ha comprobado en un clúster de la Universidad de Almería la versión híbrida de MPI y *pthreads*, dónde cada nodo contiene dos AMD EPYC Rome (con 24 núcleos físicos y 48 lógicos) y 512GB de memoria principal. Los nodos no tienen memoria compartida y se comunican mediante una red InfiniBand FDR de 54Gbps. Se tuvo acceso a estas máquinas en la etapa final del proyecto.

1.4.2. Software

La versión en el lenguaje C se ha implementado con el editor Visual Studio Code, desde un sistema operativo Windows 10. La comprobación del funcionamiento se ha realizado con este software y se ha contrastado con la implementación de MATLAB proporcionada por los tutores en 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 sobre un sistema operativo Ubuntu 22.04.4 LTS.

Se ha utilizado el compilador de gcc, y para la versión híbrida se ha acudido al compilador mpicc, que permite el uso de las librerías de MPI. Igualmente, también se ha hecho uso de la librería *pthreads*.

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 (12) por su mejor calidad de distribución numérica y eficiencia respecto a generadores del estándar.

También se ha escogido esta opción 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 (sistema operativo usado principalmente en entornos de paralelismo y computación de altas prestaciones). Ese enfoque era mucho peor desde del punto de vista de la mantenibilidad del código, por lo que su unificación con un generador externo fue también positiva.

Otro material utilizado para la documentación de la memoria es el libro (13) y la herramienta Canva (14).

1.4.4. Costes

Se ha estimado un coste del trabajo realizado de 4480€ tal y como se desglosa en la Tabla 1. En la Tabla 2 se calcula el coste estimado del equipo y las herramientas utilizadas, que llega a los 250€. En total se estima que el coste del proyecto sería de 4730€.

Personal	*Horas*	*€/hora*	*Coste total (€)*
Ingeniero informático	320	14	4480

Tabla 1: Coste de personal

Equipamiento	*Coste inicial (€)*	*Coste del uso (€)*	*Coste total (€)*
Portátil	200	30	230
Servidor	0	0	0
Clúster	0	0	0
Windows 10	20	0	20
Ubuntu	0	0	0
MATLAB R2024b	0	0	0
Visual Studio	0	0	0
Librerías	0	0	0
Total	220	30	250

Tabla 2: Coste del equipamiento

1.5. Licencia de código abierto

El trabajo desarrollado y todos los aspectos que engloba, incluyendo código, scripts y resultados obtenidos; se encuentra en el siguiente repositorio público de GitHub: https://github.com/Guillermo-Dalda/TFG.git.

El código ha sido liberado bajo los términos de la licencia MIT, permitiendo su uso, modificación y distribución, siempre que se incluya el aviso de copyright correspondiente y una copia de la licencia.

2. Revisión bibliográfica sobre optimización

En el apartado anterior ya se ha descrito brevemente qué es la optimización, conceptualmente, y su propósito en diversos ámbitos. Sin embargo para comprender mejor en qué consiste y por qué se aplican metodologías basadas en heurísticas para resolver algunos problemas (especialmente problemas no lineales), se va a realizar una revisión en profundidad del tema.

Un problema de optimización genérico se puede expresar de forma muy simple, en términos de minimización, como una función en la que se tenga que encontrar un punto $\hat{x} \in X$ de forma que (15):

$$f(\hat{x}) \leq f(x) \ \ para\ todo \ \ x \in X$$

La función f sería la conocida como función objetivo y $f(\hat{x})$ sería el óptimo global dado por $\hat{x}$. Por ejemplo, en la Figura 2 $\hat{x}$ sería "5" y proporcionaría el mínimo global de la función $f(x)$. Pero el óptimo buscado también puede ser un máximo global, dependiendo de si el objetivo es maximizar o minimizar la función. La razón por la que la expresión anterior es generalmente válida es, que para buscar un máximo global de una función $f(x)$ se puede buscar en su lugar el mínimo global de la función $-f(x)$, cuyo valor $\hat{x}$ es equivalente.

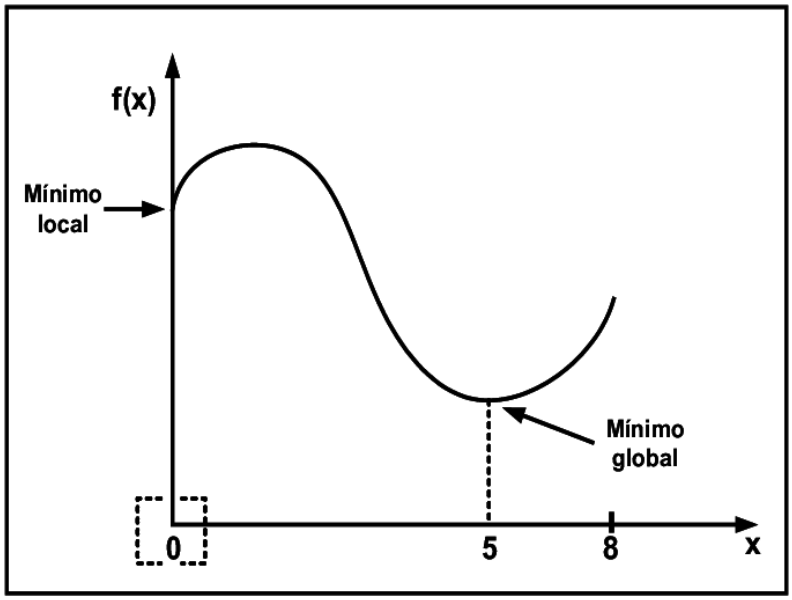

Figura 2: Ejemplo de función con mínimo local y global.[2]

En la Figura 2 también se puede observar que puede haber óptimos locales, no solo globales. Esto, y otras características, forman parte de lo que constituyen diferentes tipos de problemas de optimización; y plantean dificultades distintas. Dichas características motivan el uso de distintos métodos en función del problema planteado y sus propiedades. Por ello se va a ver cómo diferenciar los problemas según sus

[2] Fuente: https://www.researchgate.net/figure/Figura2Optimoslocalesyglobales_fig2_307181653

propiedades, qué métodos existen para optimizarlos y su viabilidad para las distintas clases de problemas.

2.1. Clases de problemas de optimización

En primer lugar, es importante considerar los factores que diferencian unos problemas de otros. Se puede ver si tiene restricciones 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 contiene otras funciones aparte de la función objetivo que actúan como condiciones que deben cumplir los resultados de este. 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.
- Por otro lado, en un problema sin restricciones solo importa el valor de la función objetivo como tal. Pero como es lógico, los 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 como la función esfera de la Figura 3 solo tienen un óptimo, ya sea mínimo o máximo. Esto simplifica la tarea de optimización, ya que no puede caer en un óptimo local.

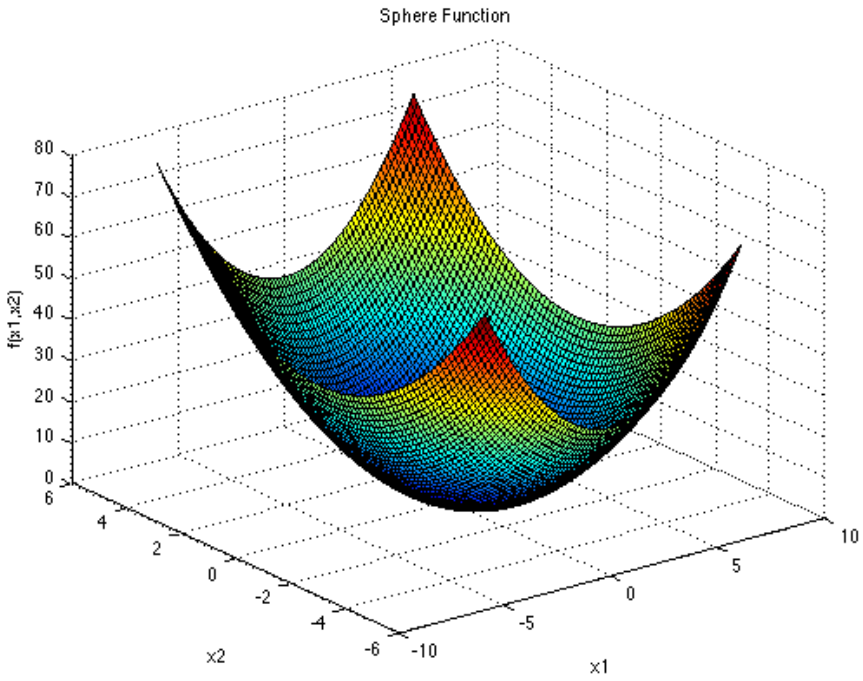

Figura 3: Ejemplo de un problema unimodal con un único mínimo global.[3]

[3] Fuente: https://www.sfu.ca/~ssurjano/spheref.html

- Los problemas multimodales tienen múltiples puntos críticos, pudiendo haber uno o varios óptimos globales y múltiples óptimos locales como ocurre con la función representada en la Figura 4. Esto dificulta el proceso de optimización, pues se tiene que evitar considerar un óptimo local como el global y explorar todas las opciones.

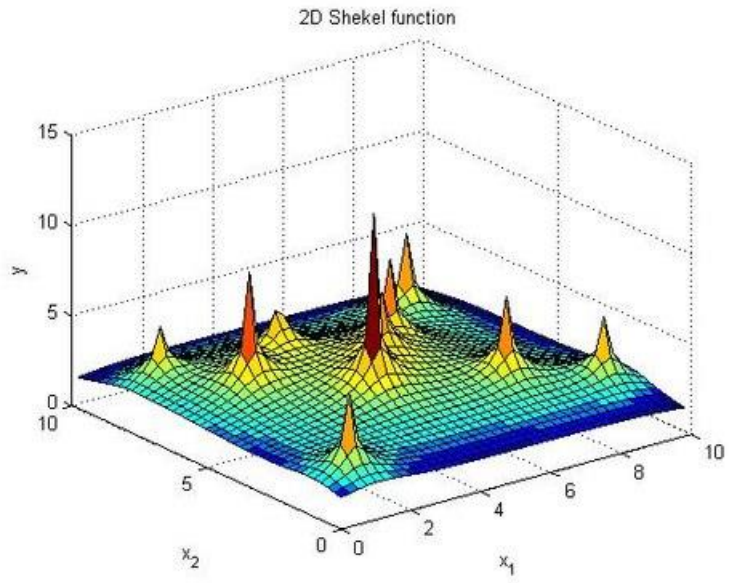

Figura 4: Ejemplo de un problema multimodal con múltiples máximos locales.[4]

- Una función lineal siempre se puede representar mediante el producto de un valor por la pendiente de una recta, por lo que nunca será multimodal.
- Una función no lineal no tiene una única pendiente y por tanto no puede representarse mediante una línea recta.

Las características anteriores describen aspectos de los problemas de optimización y dependen finalmente de si son problemas lineales o no lineales:

- Cuando se habla de problemas lineales, tanto su función objetivo como sus restricciones han de ser funciones lineales (16). Las restricciones son fundamentales, ya que sin ellas, la función objetivo podría crecer o decrecer indefinidamente, lo que haría imposible determinar un valor máximo o mínimo. Es decir, sin restricciones adecuadas, las variables tenderían hacia el infinito o menos infinito y no se podría encontrar una solución óptima (1). Por esta misma razón deben también ser siempre problemas unimodales.
- Todo problema que no se caracterice como lineal, ya sea porque su función objetivo o alguna de sus restricciones no lo son, es un problema no lineal. Estos problemas pueden ser tanto unimodales como multimodales y pueden estar restringidos o no sin que se trivialice el resultado.

[4] Fuente: https://en.wikipedia.org/wiki/Shekel_function

Los problemas no lineales se consideran más complejos que sus homólogos lineales (16). Por esto, algunos de los métodos más sencillos de optimización no suelen utilizarse para resolver este tipo de problemas y se han ido desarrollando métodos poco convencionales que sí.

2.2. Métodos de optimización

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.

En general, es importante destacar que no existe un método estrictamente mejor que otro para todos los casos. Cada uno tiene ventajas y desventajas que los hacen más o menos apropiados para un problema concreto (16). En *Numerical optimization* (17) se establece que todo buen algoritmo de optimización debe contar con tres propiedades:

- Robustez: Deben tener un buen rendimiento en una amplia variedad de problemas en su misma clase.
- Eficiencia: No deben requerir una cantidad de tiempo o espacio de computación excesiva.
- Precisión: Deben poder identificar una solución con poco margen de error.

Pero estos objetivos son conflictivos. Un algoritmo muy preciso puede requerir mayor tiempo de cómputo, o un algoritmo muy eficiente puede resultar poco robusto. El problema principal de la optimización es que siempre compensa fortaleza en algún aspecto por debilidad en otro, y generalmente la precisión y la eficiencia influyen directamente la una sobre la otra. Este par de métodos son un buen ejemplo extremo de esta situación:

- Uso de cálculo diferencial: 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 las aplicaciones prácticas (1). Es un método preciso, pero poco robusto y en muchos casos requiere un esfuerzo computacional demasiado costoso.
- Fuerza bruta: La fuerza bruta nunca es eficiente y siempre encuentra el óptimo. Lo hace a costa de explorar todas las soluciones. Por eso sólo se puede aplicar en muy pocos problemas.

Otros métodos que se utilizan de forma más amplia suelen tratar de encontrar el mejor balance posible de eficiencia y precisión. Se pueden mencionar los siguientes:

- Métodos de búsqueda lineal: Son métodos iterativos que descienden por pasos hacia un mínimo (17). Generalmente son eficientes y precisos, pero poco robustos. Por ejemplo, el método de interpolación (18) funciona bien cuando la función objetivo es suave, pero sino es mejor utilizar el método de la sección áurea (19).
- Métodos de gradiente: Son métodos que derivan de los anteriores, siguen siendo iterativos, pero además deciden la dirección por la que descender. Algunos de los más populares son el método del descenso más rápido (20) y el método del gradiente conjugado (21). Pueden resolver problemas complejos de forma eficiente, y encontrando resultados óptimos; pero no necesariamente óptimos globales (1). Son eficientes y precisos, pero dependen de que la función objetivo del problema sea unimodal.
- Métodos basados en metaheurísticas: Estos métodos suelen ser estocásticos, y 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. Se pueden clasificar de muchas formas dependiendo de las características del algoritmo, como en función del número de soluciones que se utilizan a la vez (22). Algunos trabajan con una única solución (búsqueda de un único punto) mientras la mayoría realizan la búsqueda con múltiples puntos iniciales (búsqueda basada en poblaciones). Son muy eficientes y pueden conseguir resultados bastante precisos, dependiendo del algoritmo escogido y el problema en cuestión.

Estos y otros métodos son ampliamente utilizados para optimizar, principalmente, problemas no lineales. De hecho, para problemas lineales ya existen métodos especializados muy eficaces, en concreto el método Simplex (23) y los métodos de punto interior (24).

En el trabajo realizado, el algoritmo utilizado (NOA) es un algoritmo metaheurístico basado en poblaciones. Por tanto, es pertinente desarrollar en mayor profundidad en qué consisten.

2.3. Algoritmos basados en poblaciones

Los algoritmos metaheurísticos basados en poblaciones se están volviendo poderosos métodos para resolver muchos problemas de optimización complejos del mundo real. Estos métodos utilizan diferentes heurísticas y exploran estrategias únicas, la mayoría intentando replicar patrones que se dan en la naturaleza (conocidos como algoritmos basados en la naturaleza).

Algunos se ven inspirados por la mejora gradual de las especies gracias a la evolución, herencia, genética y selección natural. De aquí surgen la Programación Evolutiva (25) y el Algoritmo Genético (26), por ejemplo.

Otros se encuentran basados en el comportamiento de plantas, humanos, enjambres, etc., siendo estos últimos los más destacados de todos por la conocida Optimización de Partículas de Enjambre (27).

También se pueden encontrar otros algoritmos basados principios de la física, química, geología, etc. Además, la actividad de este campo también ha dado lugar a otros métodos, como el *Battle Royale Optimization* (28), que no se pueden enmarcar directamente en alguna de estas categorías.

A pesar de que la mayoría de los 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. Por otro lado, los parámetros específicos son herramientas de control y configuración específicas para cada algoritmo que permiten aumentar o reducir el rendimiento de estos. 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. De hecho, la tarea de afinar todos los parámetros específicos de un 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 fomenta la búsqueda de algoritmos que no requieran parámetros específicos, como NOA.

Este algoritmo, que se explica en detalle en la siguiente sección, destaca especialmente por no requerir parámetros específicos e incorporar técnicas para equilibrar, de forma adaptativa, las fases de exploración y explotación previamente mencionadas.

3. Nizar Optimization Algorithm (NOA)

NOA se ha diseñado para resolver problemas con restricciones y sin restricciones. El método no necesita ningún parámetro específico, solo 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. Dichas técnicas se describen a continuación:

- Mapeos efectivos: Son seis mapas de vectores que se clasifican en dos tipos:
    - Mapeos de combinaciones: Construyen un vector combinando dos vectores de entrada utilizando tres mecanismos diferentes.
    - Mapeos de transformaciones: Devuelven el vector de entrada modificado o sustituido por otro en función de variables de 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.

Como ya se ha mencionado, 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 diversificación: 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 que un problema se encuentra dentro de un dominio y, por tanto, cada variable puede tomar valores acotados entre dos límites, inferior y 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 que la función objetivo calcule el valor de un punto o individuo; por tanto, el bucle externo se ejecuta (evaluaciones / tamaño de población) veces.

Tras la inicialización y evaluación de la población inicial, el algoritmo en sí se compone de varias etapas que utilizan variables estocásticas y los individuos existentes para generar nuevos, como se observa en el Pseudocódigo 1. Estas etapas se describen con más detalle en los siguientes apartados.

Algoritmo: 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 P1;
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 P2 y P3;
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 P1;
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;
```

Pseudocódigo 1: Algoritmo NOA

3.2. Extracción de los coeficientes de selección

3.2.1. Generar los coeficientes α

Para cada generación se calculan cuatro coeficientes (α_1, α_2, α_3, α_4) que se usan en el proceso de selección de los mapeos efectivos. En la práctica se generan como se ve en el Pseudocódigo 2 y su descripción teórica viene a continuación:

$\alpha_1 = r_1$
$\alpha_2 = r_2 - \frac{3}{8}$
$\alpha_3 = r_3 - \frac{1}{4}$
$\alpha_4 = r_4 - \frac{3}{8}$

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 $[-\frac{3}{8}, \frac{5}{8}]$;
3 Alfa[3] se genera como aleatorio en el rango $[-\frac{1}{4}, \frac{3}{4}]$;
4 Alfa[4] se genera como aleatorio en el rango $[-\frac{3}{8}, \frac{5}{8}]$;
5 **return** Alfa;

Pseudocódigo 2: Generar coeficientes alfa

3.2.2. Generar los coeficientes λ

Para cada generación se calculan ocho coeficientes que se usan en el proceso de construcción de puntos efectivos:

λ_1, λ_2, λ_3, λ_4, λ_5, λ_6, λ_7, λ_8.

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

El valor a_n hace referencia a 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 se ha simplificado como el módulo de un entero aleatorio sin signo para cada λ, tal y como aparece en el Pseudocódigo 3.

Función 2: Generar_coeficientes_lambda

Input: semilla para MersenneTwister
Output: Vector Lambda
for i = 1 hasta N **do**
    r se genera como entero aleatorio sin signo;
    Lambda[i] = r mod 2;
end
return Lambda;

Pseudocódigo 3: Generar coeficientes 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.

Se puede ver en el Pseudocódigo 4 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. A continuación, se verá cómo se calcula el punto efectivo $\overrightarrow{P_1}$, que se utiliza para hallar las soluciones $\overrightarrow{S_1}$ y $\overrightarrow{S_2}$ como se explica más adelante.

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 $\overrightarrow{X'_i}$.
$\overrightarrow{P_1}$ = Construir_P1();
if Lambda[1] = 1 **do**
    $\overrightarrow{S_1}$ = Construir_S1();
    $\overrightarrow{X'_i} = \overrightarrow{S_1}$;
else
    $\overrightarrow{T_3}$ = Construir_T3();
    $\overrightarrow{V_j}$ = Transferencia($\overrightarrow{X_j}$, $\overrightarrow{T_3}$, Alfa[1], D);
    $\overrightarrow{V_k}$ = Transferencia($\overrightarrow{X_k}$, $\overrightarrow{T_3}$, Alfa[1], D);
    $\overrightarrow{T_2}$ = Construir_T2();
    $\overrightarrow{P_2}$ = Construir_P2();
    $\overrightarrow{P_3}$ = Construir_P3();
    $\overrightarrow{S_2}$ = Construir_S2();
    $\overrightarrow{X'_i} = \overrightarrow{S_2}$;
end
return $\overrightarrow{X'_i}$;

Pseudocódigo 4: Fase de diversificación

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, como se indica en el Pseudocódigo 5. 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 $\overrightarrow{P_1}$.

```
if Lambda[3] = 1 do
    P1 = Xm;
// Si no se construye T1 y se le atribuye a P1.
else if Lambda[6] = 1 do
    P1 = T1 = 0.5 × (X_Best + Xm);
else
    P1 = T1 = β1 × Xm + (1 – β2) × X_Best;
end
return P1;
```

Pseudocódigo 5: Construcción del punto P_1

Una vez calculado $\overrightarrow{P_1}$, se comienza la construcción de $\overrightarrow{S_1}$ o $\overrightarrow{S_2}$. Se describe en primer lugar la construcción de $\overrightarrow{S_1}$ porque no requiere de puntos adicionales.

3.3.2. Construcción de $\overrightarrow{S_1}$

En el Pseudocódigo 6 se puede observar que $\overrightarrow{S_1}$ 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
    S1 = P1 + β1 × (Xi - Xj) + β2 × (Xi - Xk);
else
    S1 = Xi + β1 × (P1 - Xj) – β2 × (P1 - Xk);
end
return S1 ;
```

Pseudocódigo 6: Construcción de la solución S_1

Si se va a construir $\overrightarrow{S_2}$ en lugar de $\overrightarrow{S_1}$; son necesarios los puntos efectivos $\overrightarrow{P_2}$ y $\overrightarrow{P_3}$ y los vectores $\overrightarrow{V_j}$ y $\overrightarrow{V_k}$ de los que depende. A su vez todos estos dependen de los puntos efectivos $\overrightarrow{T_2}$ y $\overrightarrow{T_3}$.

Por lo que se construye $\overrightarrow{T_3}$ en primer lugar, ya que también se utiliza en la construcción posterior de $\overrightarrow{T_2}$. Después se construyen los puntos $\overrightarrow{P_2}$ y $\overrightarrow{P_3}$ y finalmente la solución $\overrightarrow{S_2}$. $\overrightarrow{V_j}$ y $\overrightarrow{V_k}$ se pueden describir de la siguiente forma:

$$\overrightarrow{V_j} = Transferencia(\overrightarrow{X_j}, \overrightarrow{T_3}, Alfa[1], D)$$
$$\overrightarrow{V_k} = Transferencia(\overrightarrow{X_k}, \overrightarrow{T_3}, Alfa[1], D)$$

3.3.3. Construcción de $\overrightarrow{T_3}$

En este caso el Pseudocódigo 7 muestra la construcción de $\overrightarrow{T_3}$. 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 λ no es igual a 1, 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 $\overrightarrow{T_3}$.

```
1  if Lambda[8] = 1 do
2      T3 = X_Best;
3  else
4      T3 = Xi;
5  end
6  return T3;
```

Pseudocódigo 7: Construcción del punto 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 $\overrightarrow{T_2}$ de la siguiente forma también visible en el Pseudocódigo 8:

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 $\overrightarrow{T_2}$.

```
1  if Lambda[7] = 1 do
2      T2 = Traducción(reemplazar(T3, P1, D), β2, Alfa[4], D);
3  else
4      T2 = Traducción(mezclar(T3, P1, D), β2, Alfa[4], D);
5  end
6  return T2 ;
```

Pseudocódigo 8: Construcción del punto T_2

3.3.5. Construcción de $\overrightarrow{P_2}$

En el Pseudocódigo 9 se puede observar que $\overrightarrow{P_2}$ 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
    P2 = Xm + β1 × 1D;   //1D es un vector de unos de tamaño D.
else
    P2 = T2;
end
return P2 ;
```

Pseudocódigo 9: Construcción del punto P_2

3.3.6. Construcción de $\overrightarrow{P_3}$

Para generar el último punto efectivo se sigue el procedimiento descrito en el Pseudocódigo 10. Según se muestra, 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 $\overrightarrow{P_1}$ o aplica $\overrightarrow{P_1}$ 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
    P3 = Dilatación[Transferencia(distribuir(P1, T3, D), T2, Alfa[3], D), β2, Alfa[2], D];
else
    P3 = Dilatación[Transferencia(distribuir(T3, P1, D), T2, Alfa[3], D), β2, Alfa[2], D];
end
return P3 ;
```

Pseudocódigo 10: Construcción del punto P_3

Llegados a este punto, se puede hallar la solución $\overrightarrow{S_2}$ que se corresponde son un individuo de siguiente generación.

3.3.7. Construcción de $\overrightarrow{S_2}$

De forma semejante a $\overrightarrow{S_1}$, pero en su lugar como se describe en el Pseudocódigo 11, $\overrightarrow{S_2}$ se calcula como combinación lineal de $\overrightarrow{P_2}$, $\overrightarrow{P_3}$, $\overrightarrow{V_j}$, y $\overrightarrow{V_k}$. Como se puede apreciar, el modo de cálculo depende fundamentalmente 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 $\overrightarrow{S_2}$.

1 **if** Lambda[2] = 1 **do**
2 $\quad \overrightarrow{S_2} = \overrightarrow{P_2} + \delta_j \times (\overrightarrow{P_3} - \overrightarrow{V_j}) + \delta_k \times (\overrightarrow{P_3} - \overrightarrow{V_k})$;
3 **else**
4 $\quad \overrightarrow{S_2} = \overrightarrow{P_3} + \delta_j \times (\overrightarrow{P_2} - \overrightarrow{V_j}) + \delta_k \times (\overrightarrow{P_2} - \overrightarrow{V_k})$;
5 **end**
6 **return** $\overrightarrow{S_2}$;

Pseudocódigo 11: Construcción de la solución 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. En las secciones siguientes se verá el funcionamiento de estos mapeos y así se entenderá mejor cómo logran combinar y transformar puntos efectivos e individuos de anteriores generaciones.

3.4. Mapeos de combinación efectivos

Los mapeos de combinación efectivos buscan combinar algunos de los valores aleatorios de los individuos de la población de formas distintas y creativas. Se utilizan tres métodos, que consisten en el reemplazo, mezcla y distribución de valores.

3.4.1. Reemplazar

Como dice su nombre, este mapa reemplaza algunos valores del vector de entrada $\overrightarrow{V_{in}}$, por los respectivos del vector objetivo $\overrightarrow{V_{tg}}$, en posiciones aleatorias, tal y como se realiza en las líneas 2, 3 y 4 del Pseudocódigo 12. 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 $\overrightarrow{V_{tg}}$ y la dimensión D del problema.

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

1 **for** i = 1 hasta D **do**
2 $\quad$ r se genera como un número entero aleatorio sin signo;
3 $\quad$ **if** r es par **do**
4 $\quad\quad \overrightarrow{V}_{out}[i] = \overrightarrow{V_{tg}}[i]$;
5 $\quad$ **else**
6 $\quad\quad \overrightarrow{V}_{out}[i] = \overrightarrow{V_{in}}[i]$;
7 $\quad$ **end**
8 **end**
9 **return** $\overrightarrow{V}_{out}$;

Pseudocódigo 12: Reemplazar

Por ejemplo, si se decide reemplazar el primer valor del vector de entrada, pero no el segundo como en la Figura 5, en la primera posición del vector de salida se guardaría el “4” del vector objetivo mientras que en la segunda posición se dejaría el valor original de entrada.

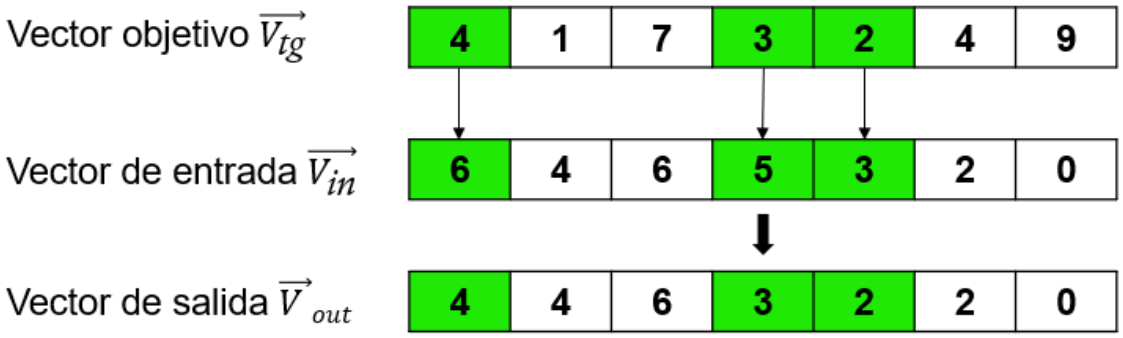

Figura 5: Ejemplo de reemplazar.[5]

3.4.2. Mezclar

Este mapa mezcla algunos valores del vector objetivo $\overrightarrow{V_{tg}}$ en el vector de entrada $\overrightarrow{V_{in}}$, sobre posiciones aleatorias como describe el Pseudocódigo 13. 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 $\overrightarrow{V_{tg}}$ y la dimensión D del problema.
Output: Vector $\overrightarrow{V}_{out}$.
$\overrightarrow{V}_{aux}$ = Fisher_Yetes_shuffle($\overrightarrow{V_{tg}}$) //Baraja el contenido del vector de forma aleatoria (29)
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] = $\overrightarrow{V_{aux}}$[i];
    else
        $\overrightarrow{V}_{out}$[i] = $\overrightarrow{V_{in}}$[i];
    end
end
return $\overrightarrow{V}_{out}$;

Pseudocódigo 13: Mezclar

Por ejemplo, los valores en primera, cuarta y quinta posición del vector de entrada se sustituyen por el tercer, primer y último valor del vector objetivo respectivamente, como aparece en la Figura 6, y el resto se mantienen. A diferencia del mapeo de reemplazar, los valores cogidos del vector objetivo no han sido aquellos que comparten la misma posición que el del vector de salida, pueden ser otros cualquiera.

[5] Elaboración propia

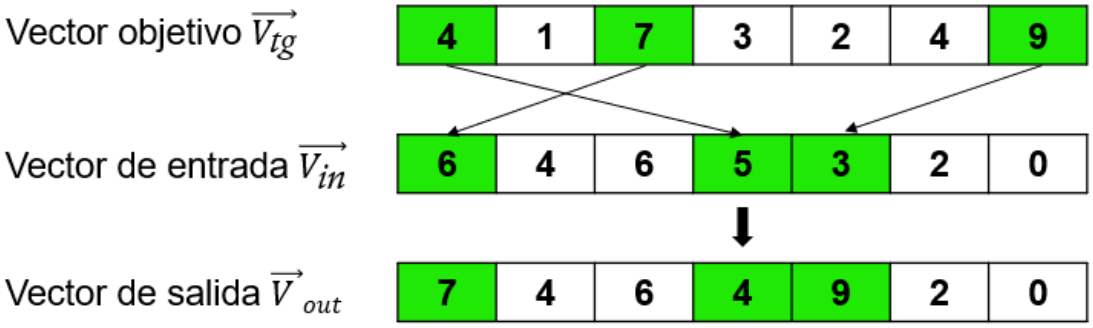

Figura 6: Ejemplo de mezclar.[6]

3.4.3. Distribuir

Como se describe en el Pseudocódigo 14, 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 $\overrightarrow{V_{tg}}$ 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 ;
```

Pseudocódigo 14: Distribuir

Por ejemplo, se coge el tercer valor del vector objetivo, que contiene un “7” como en la Figura 7, entonces se sustituyen todos los valores elegidos en el vector de salida por un “7” y el en el resto de las posiciones se mantienen los valores del vector de entrada.

[6] Elaboración propia

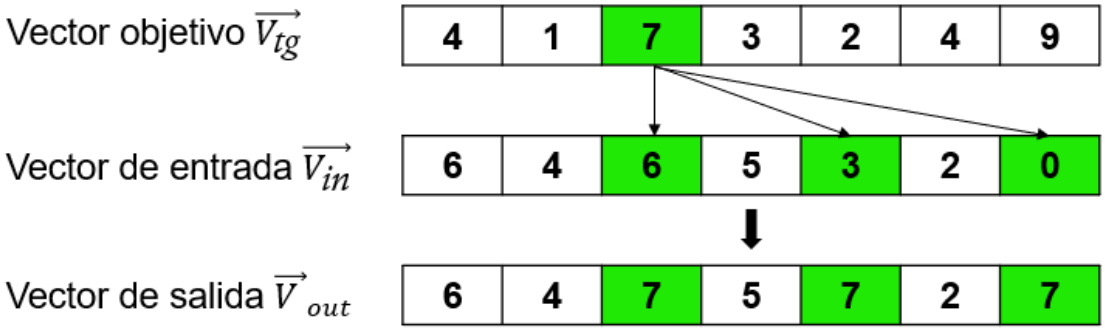

Figura 7: Ejemplo de distribuir.[7]

3.5. Mapeos de transformación efectivos

Todos los mapeos de transformación devuelven el vector de entrada sin alterar en el caso de que el valor de α sea menor o igual a un medio (0.5). En otro caso se devuelve un vector distinto o una combinación lineal de este, dependiendo a cuál de los tres métodos de transformación se haya llamado. Estos son los mapas de traducción, de dilatación y de transferencia.

3.5.1. Traducción

Como se describe en el Pseudocódigo 15, la función recibe un vector de entrada $\vec{V}_i$ y un escalar *r*. Devolverá 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 5.1: 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
    V_out = V_i;
else
    V_out = V_i + r^2 × 1_D;   //1_D es un vector de unos de tamaño D.
end
return V_out;
```

Pseudocódigo 15: Traducción

3.5.2. Dilatación

Tal y como se describe en el Pseudocódigo 16, la función 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}.

[7] Elaboración propia

Función 5.2: Dilatación

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

```
if Alfa[n] <= 0.5 do
    V_out = V_i;
else
    V_out = V_i × r^2;
end
return V_out;
```

Pseudocódigo 16: Dilatación

3.5.3. Transferencia

En el Pseudocódigo 17 se describe cómo la función recibe dos vectores de entrada $\overrightarrow{V_i}$ y $\overrightarrow{V_j}$, devolviendo un vector $\overrightarrow{V}_{out}$ que será igual a $\overrightarrow{V_i}$ o $\overrightarrow{V_j}$ 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 $\overrightarrow{V_i}$ y $\overrightarrow{V_j}$; un valor del vector Alfa; y la dimensión D del problema.
Output: El punto $\overrightarrow{V}_{out}$.

```
if Alfa[n] <= 0.5 do
    V_out = V_i;
Else
    V_out = V_j;
end
return V_out;
```

Pseudocódigo 17: Transferencia

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. Se lleva 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 $\frac{1}{4}$, siendo r número aleatorio con distribución uniforme en el rango [0, 1].

Si se cumpla alguna, como se indica en el Pseudocódigo 18, 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, sino multiplicaciones elemento a elemento. De esta forma, 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] × $\overrightarrow{X_m}$[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 $\overrightarrow{X'_i}$;

Pseudocódigo 18: Fase de superposición

Tras esta fase, se llegue a realizar o no, se llevan a cabo las siguientes acciones:

- Se verifica que la solución $\overrightarrow{X'_i}$ siga dentro de los límites del espacio de búsqueda. Todo valor que se salga de los límites permitidos de su variable correspondiente se descarta y es reemplazado por el que contenía la solución original, $\overrightarrow{X_i}$.
- Luego se evalúa el valor fitness de $\overrightarrow{X'_i}$, o lo que es lo mismo, se evalúa en la función objetivo para esa solución candidata.
- 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 de la función objetivo es mejor.

Una vez se comprende el funcionamiento del algoritmo, se puede pasar al planteamiento del paralelismo que se va a intentar explotar sobre este.

4. Estrategias de paralelización

Antes de intentar paralelizar el algoritmo, se debe entender qué aspectos de este son los que conllevan una mayor carga computacional. Por tanto, se procede a revisar 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 y N es el tamaño de la población. Tal y como detallan los autores en el artículo original (2), 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; por lo que no es necesario realizar balanceo de carga y la implementación de mecanismos de sincronización más complejos.

Por último, se puede destacar que tiene una complejidad equivalente a otros métodos basados en poblaciones y, por tanto, se pueden aplicar estrategias de paralelización equivalentes a otros optimizadores basados en poblaciones como (30). La estrategia seguida se describe y justifica en el apartado siguiente.

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 utilizadas en la experimentación, 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 que 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 (31), que consiste en la ejecución del algoritmo de forma independiente en cada hebra con sus propias variables locales. Sin embargo, este enfoque puede degradar el proceso de optimización del algoritmo al no tener tantas generaciones por unidad de ejecución como para converger en la solución.

Las opciones se reducen a dividir la generación de los individuos entre las hebras en cada iteración, que dividiría la carga computacional de NCD. Se puede concluir entonces, que el coste computacional depende fundamentalmente del tamaño de la población, del número de iteraciones y del número de dimensiones.

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 primer 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. Para que puedan reducir el tiempo de cómputo el sistema tiene la misma cantidad de núcleos físicos que hilos creados, por lo que es ideal una máquina multinúcleo.

Para aprovechar el paralelismo real que es posible en multicomputadores, además de en máquinas multinúcleo, se ha implementado una versión híbrida que combina MPI con *pthreads*. MPI es una interfaz de paso de mensajes que permite crear varios procesos que realicen tareas en núcleos de procesamiento distintos y que se comuniquen entre ellos, ya se encuentren en la misma máquina u otra. Los procesos son inherentemente paralelos, por lo que pueden ejecutarse tantos como se quiera simultáneamente, siempre que haya recursos disponibles.

El uso de estas dos herramientas en un entorno con suficientes prestaciones combina el paralelismo que otorga MPI con la concurrencia de varios hilos, pudiendo proporcionar una aceleración sobre la versión secuencial que se acerque a la máxima.

4.3. Implementación únicamente con *phtreads*

La implementación con *phtreads* requiere de una estructura de datos en la que se almacenen los datos compartidos que vaya a necesitar cada hebra. Cada una creará después las variables locales que necesite para almacenar datos independientes.

Una vez se comienza la ejecución del algoritmo en cada hebra, se realizan los siguientes pasos para paralelizarlo correctamente:

- Se inicializa el generador de números aleatorios (línea 1 del Pseudocódigo 20) 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. Si esto no se hace, tener un generador único causaría un cuello de botella, pues las hebras tendrían que esperarse unas a otras cada vez que quisieran acceder.

- Se dividen los individuos de la población entre el número de hebras por bloques, estableciendo el inicio y final de los bucles para cada hebra (en las variables *start* y *end*). Por ejemplo, en la Figura 8 se separa una población de 16 individuos entre 4 hebras; a cada una se le asigna una parte para evitar accesos simultáneos. El bloque del hilo 1 tendría el comienzo (*start*) en la posición 0 del vector y el final (*end*) en la posición 3, el del hilo 2 comenzaría en la posición 4 y terminaría en la 7…

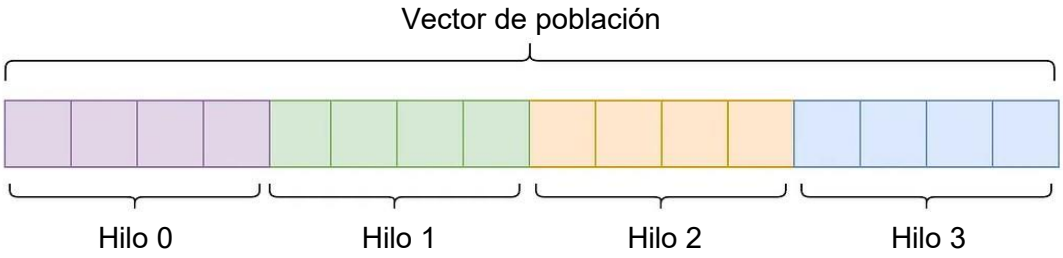

Figura 8: Vector dividido entre hebras.[8]

En caso de que la división no fuese entera, se asignan los bloques de tamaño lo más similar posible en función del resto. Se empezaría añadiendo una posición más al final del primer hilo como aparece en la línea 4 del Pseudocódigo 19 (y al inicio de todos los demás en consecuencia como aparece en la línea 3 del Pseudocódigo 19), continuando con el segundo hilo y así sucesivamente. El proceso que se describe en el Pseudocódigo 19 se realizaría por todos los hilos y cada uno obtendría el inicio y final de su bloque de individuos.

Función: Dividir_población_hebras

Input: Tamaño N de la población, la cantidad de hebras *n* y el identificador *id* de cada hebra.
Output: El inicio y fin del bloque de la hebra.

```
1  Bloque = N / n;
2  Resto = N % n; //Se guarda el resto de la división
3  Start = id * Bloque; if id < Resto then Start = Start + id; else Start = Start + Resto; end
4  End = Start + Bloque; if id < Resto then Start = Start + 1; end
5  return [Start, End]
```

Pseudocódigo 19: Proceso de división de la población entre hebras

- Para evitar condiciones de carrera, se sitúan barreras o *barriers* en los puntos en los que toda la población tiene que estar actualizada. Estos puntos son los siguientes: después de la inicialización de la población (línea 8 del Pseudocódigo 20), tras el cálculo de cada nueva generación de individuos (línea 31 del Pseudocódigo 20) y después de actualizar los vectores compartidos con las soluciones calculadas (de nuevo línea 8 del Pseudocódigo 20). Esta última barrera se ha incluido en el mismo punto que la que sincroniza la inicialización de la población, reduciendo el total de sincronizaciones necesarias a solo dos por iteración del bucle principal y resultando algo más eficiente.

[8] Elaborada a partir de la que se encuentra en la fuente: https://medium.com/javarevisited/fork-join-thread-pool-framework-a-powerful-tool-for-task-splitting-computation-8c8da92708a4

Algoritmo: Pseudocódigo del algoritmo NOA paralelizado mediante *pthreads*

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

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

Pseudocódigo 20: Algoritmo NOA paralelizado mediante pthreads

Es importante recalcar que la actualización de la generación actual con los nuevos individuos se realiza una vez que todos se han calculado (división en dos del bucle en las líneas 30-32 del Pseudocódigo 20), a diferencia del funcionamiento secuencial (Pseudocódigo 1) que actualiza la población cada vez que se encuentra una solución mejor. Esto se hace para que no haya accesos simultáneos a los individuos por parte

de las hebras y es indiscutible, pero significa que el algoritmo podría converger hacia el valor óptimo más lentamente y deteriorar los resultados respecto a la versión secuencial.

4.4. Implementación híbrida de MPI y *phtreads*

Para realizar la implementación híbrida se ha partido de la versión descrita en el apartado anterior que paralelizaba utilizando *pthreads*. A esta se le han incluido las llamadas de MPI necesarias para una combinación efectiva, eficiente y que no altera el funcionamiento original del algoritmo. Los pasos añadidos son los siguientes:

- En el fichero “main” se han de inicializar los procesos de MPI, pero solo el proceso “0” o Maestro recoge los parámetros del programa y los envía al resto de procesos. Esto implica el uso de los correspondientes mensajes bloqueantes (con sincronización implícita).
- El siguiente grupo de mensajes se establece en la inicialización de la población vista en el Pseudocódigo 19. Se realiza de modo que, aunque todos los procesos tengan hebras, solo el proceso Maestro y sus hebras realicen la inicialización de la misma forma descrita en el apartado anterior. De nuevo, tras esto se deben comunicar los procesos para recibir la población inicializada por el Maestro.
- Por último, se debe realizar la sincronización y unificación de las soluciones de los procesos de la misma forma que se ha hecho para las hebras, después de tener todas las soluciones de la siguiente generación. Concretamente, primero se deben actualizar las propias hebras de cada proceso como se describe en las líneas 31-35 del Pseudocódigo 20, y después todos los procesos fusionan sus vectores mediante la directiva MPI_Allgatherv (32).

5. Experimentación y resultados

Se han utilizado 10 de las funciones *benchmark* y 2 de los problemas reales considerados en el artículo original (2).

Para mantener la consistencia con respecto a los resultados presentados en el documento original, se han realizado las comprobaciones del funcionamiento del algoritmo implementado en MATLAB y en C con los mismos parámetros. Concretamente se ha trabajado con un tamaño de población de 25 individuos y 35000 evaluaciones de la función objetivo, además de los parámetros específicos de cada problema. No obstante, como se verá en el apartado de experimentación, hay razones por las que cambiar el tamaño de la población puede resultar interesante.

A continuación, se pasa a describir los problemas elegidos y sus parámetros específicos.

5.1. Problemas de prueba (*benchmark*) de optimización global

Antes es importante recalcar que además de la expresión referenciada en (2) se ha contrastado la descripción con otras fuentes por rigurosidad.

Se han escogido a propósito 5 funciones *benchmark* con óptimo en 0 y otras 5 con óptimo distinto de 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. Por la misma razón se ha escogido una función 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.

Además, en el ámbito de los métodos de optimización es también recomendable probar funciones con óptimos que sean distintos. Algunos métodos tienen, por problemas de diseño, una inercia general a converger a ciertos puntos, como el origen, que da una falsa sensación de buen funcionamiento (33).

5.1.1. Sphere

Es una función unimodal con componentes en D y se ha tomado como viene descrita en el trabajo de referencia (2) y en (34):

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

Se ha fijado la dimensión a D = 50 y el dominio de las variables ha sido delimitado por [-100, 100]. El valor óptimo se puede encontrar en $x_i = 0 \ para \ i = 1{,}2 \dots D$, dónde $f(x) = 0$.

5.1.2. Quartic

Es una función unimodal con componentes en D dimensiones y se ha tomado

como viene descrita en el artículo original (2); parece estar personalizada ya que en otros artículos (35) se le suman números aleatorios:

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

Se ha fijado la dimensión a D = 50 y el dominio de las variables ha sido delimitado por [-1.28, 1.28]. El valor óptimo se puede encontrar en $x_i = 0 \ para \ i = 1,2 \dots D$, dónde $f(x) = 0$.

5.1.3. Powell Sum

Es una función unimodal con componentes en D dimensiones y se ha tomado como viene descrita en (2):

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

Se ha fijado la dimensión a D = 50 y el dominio de las variables ha sido delimitado por [-1, 1]. El valor óptimo se puede esperar entorno a $x_i = 0 \ para \ i = 1,2 \dots D$, dónde $f(x) = 0$.

5.1.4. Sum Squares

Es una función unimodal con componentes en D dimensiones y se ha tomado como viene descrita en (2) y en (36):

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

Se ha fijado la dimensión a D = 50 y el dominio de las variables ha sido delimitado por [-10, 10]. El valor óptimo se puede encontrar en $x_i = 0 \ para \ i = 1,2 \dots D$, dónde $f(x) = 0$.

5.1.5. Schwefel's 2.20

Es una función unimodal con componentes en D dimensiones y se ha tomado como viene descrita en (2) y en (37):

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

Se ha fijado la dimensión a D = 50 y el dominio de las variables ha sido delimitado por [-100, 100]. El valor óptimo se puede encontrar en $x_i = 0 \ para \ i = 1,2 \dots D$, dónde $f(x) = 0$.

5.1.6. Stepint

Es una función unimodal con componentes en D dimensiones y se ha tomado como viene descrita en (2):

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

Se ha fijado la dimensión a D = 50 y el dominio de las variables ha sido delimitado por [-5.12, 5.12]. El valor óptimo se puede esperar a partir de $x_i < -5 \; para \; i = 1,2 \dots D$, dónde $f(x) = -275$.

5.1.7. Ridge

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

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

Se ha fijado la dimensión a D = 50 y el dominio de las variables ha sido delimitado por [-5, 5]. El valor óptimo se puede encontrar en $x_1 = -5 \, ; x_i = 0 \; para \; i = 2,3 \dots D$, dónde $f(x) = -5$.

5.1.8. Neumaier's N. 3

Es una función unimodal con componentes en D dimensiones y se ha tomado en (38), ya que podría haber un error tipográfico en el artículo original.

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

Se ha fijado la dimensión a D = 15 y el dominio de las variables ha sido delimitado por [-100, 100]. El valor óptimo se puede esperar entorno a $x_1 = 15, x_2 = 28, x_3 = 39, x_4 = 48, x_5 = 55, x_6 = 60, x_7 = 63, x_8 = 64, x_9 = 63, x_{10} = 60, x_{11} = 55, x_{12} = 48, x_{13} = 39, x_{14} = 28, x_{15} = 15,$; dónde $f(x) = -665$.

5.1.9. Ackley N. 2

Es una función unimodal con componentes en 2 dimensiones y se ha tomado como viene descrita en el artículo de referencia (2), aunque dependiendo de las fuentes se puede encontrar el exponente multiplicado por "-0.02" (35) o por "-0.2" (39):

$$f(x) = -200 \exp\left(-0.02 \sqrt{x_1^2 + x_2^2}\right)$$

En este caso la dimensión no es variable, y el dominio de las variables ha sido delimitado por [-32, 32]. El valor óptimo se puede encontrar en $x_1 = 0 \; y \; x_2 = 0$, dónde $f(x) = -200$.

5.1.10. Shekel 10

Es una función multimodal con componentes en 4 dimensiones y se ha tomado como viene descrita en (38), ya que podría haber un error tipográfico en el artículo original.

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

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:

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

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

En este caso la dimensión no es variable, y el dominio de las variables ha sido delimitado por [0, 10]. El valor óptimo se puede encontrar en $x_1 = 4, x_2 = 4, x_3 = 4\ y\ x_4 = 4$, dónde $f(x) = -10.5364098$.

5.2. Problemas de ingeniería reales

Cuanta más variedad de problemas distintos se prueben, mayor es la seguridad de que el método se comporta como debería. Dado que estos 10 problemas no tienen restricciones también se han escogido 2 problemas reales de ingeniería que tienen restricciones.

Las restricciones en general se definen, como se resalta en (1), mediante funciones de la forma:

$$g_i(x) \leq 0, \qquad i = 1, 2, \ldots p$$
$$h_j(x) = 0, \qquad j = 1, 2, \ldots m$$

En este trabajo los problemas utilizados están sujetos a restricciones definidas por desigualdades (g(x)), por lo que, para tenerlas en cuenta en la función objetivo del problema, no deben influir en la solución si se cumplen y perjudicar significativamente al valor *fitness* en caso contrario. También es importante para esto que la solución vaya mejorando cuando los valores de las restricciones se acercan a "0". Se pueden cumplir con estos objetivos al mismo tiempo si planteamos el resultado dado por la función objetivo mediante penalización estática (40),de la siguiente forma:

$$Valor = f(x) + g_i(x)\ (si\ g_i(x) > 0\ para\ i = 1, 2, \ldots p)$$

Esto se puede ajustar dependiendo del problema para evitar posibles resultados en los que el que g(x) sea mayor que "0" no sea lo suficientemente perjudicial.

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.

5.2.1. Problema de Diseño de un Depósito Bajo Presión

El problema de diseño de un depósito bajo presión o PVD (P*ressure Vessel Design*) tiene el objetivo de minimizar el coste de fabricación, y se ha tomado como viene descrito en (41), ya que podría haber un error tipográfico en el artículo original.

Las variables que se observan en la Figura 9 se corresponden con las siguientes características del depósito y forman la función que se ha de optimizar:

- Grosor del depósito: $T_s = x_1$.
- Grosor de la cabeza: $T_h = x_2$.
- Radio del depósito: $R = x_3$.
- Longitud del depósito: $L = x_4$.

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

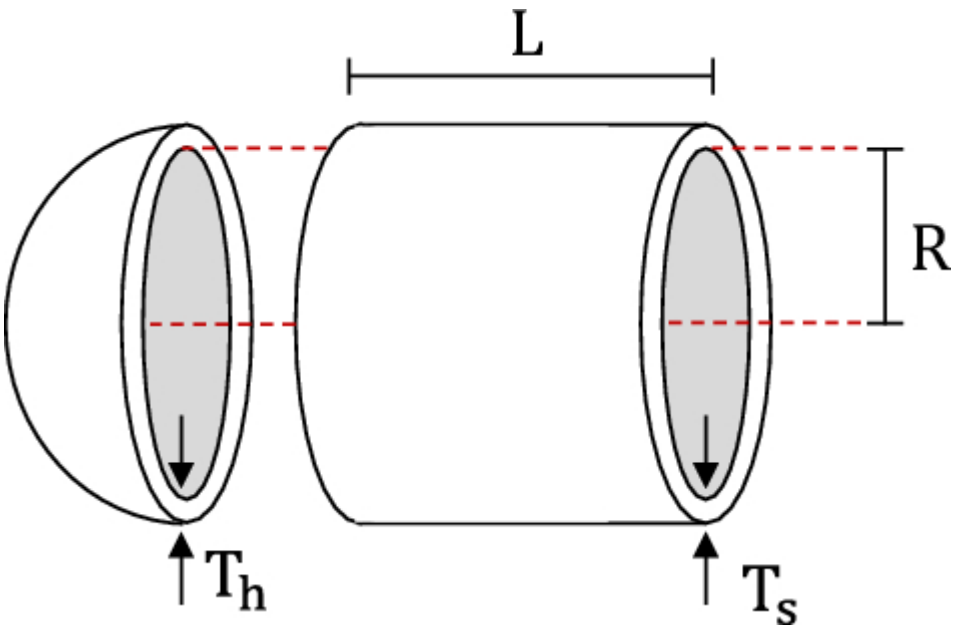

Figura 9: Disposición del problema de diseño de un depósito bajo presión.[9]

El problema además está sujeto a cuatro restricciones:

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

[9] Fuente: https://link.springer.com/article/10.1007/s11831-023-09975-0/figures/10

- Longitud del depósito completo: g_4.

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

El número de dimensiones del problema D = 4, que en este caso no es variable. El dominio de las variables delimitado por: $x_1 \in [0, 100]$, $x_2 \in [0, 100]$, $x_3 \in [10, 200]$ y $x_4 \in [10, 200]$. Y el valor óptimo se puede esperar entorno a $x_1 = 0.778169, x_2 = 0.384649, x_3 = 40.319619\ y\ x_4 = 200$, dónde $f(x) = -5885.332774$.

Debido a que el valor óptimo de este problema es mayor que 0, se ha optado por que las restricciones influyan muy significativamente en el valor de la función objetivo (asegurándose así de que nunca se devuelva una solución que incumpla las restricciones):

$$Valor = f(x) + (10000 + g_i(x))\ (si\ g_i(x) > 0\ para\ i = 1, 2, \ldots p)$$

5.2.2. Problema de Diseño de un Muelle de Compresión/Tensión

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, y se ha tomado como viene descrito en (2) y en (42).

Las variables que se observan en la Figura 10 se corresponden con las siguientes características del muelle y forman la función que se ha de optimizar:

- Diámetro del alambre: d = x_1.
- Diámetro del devanado: D = x_2.
- Cantidad de bobinas activas: N = x_3.

$$f(x) = (x_3 + 2)x_2 x_1^2$$

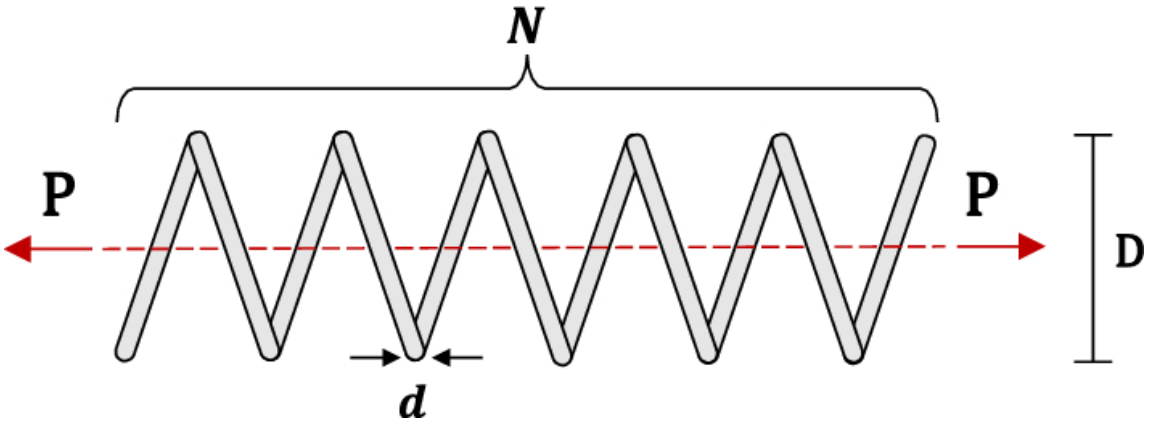

Figura 10: Disposición del problema de diseño de un muelle de compresión/tensión.[10]

[10] Fuente: https://link.springer.com/article/10.1007/s11831-023-09975-0/figures/11

El problema además está sujeto a cuatro restricciones:

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

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

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

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

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

El número de dimensiones del problema D = 3, que en este caso no es variable. El dominio de las variables delimitado por: $x_1 \in [0.05, 2]$, $x_2 \in [0.25, 1.3]$ y $x_3 \in [2, 15]$. Y el valor óptimo se puede obtener entorno a $x_1 = 0.051732, x_2 = 0.357761$ y $x_3 = 11.228079$, dónde $f(x) = 0.01266524$.

5.2.3. Problema ralentizado artificialmente

Como se mencionó anteriormente, el reducido coste computacional de los problemas de prueba hace difícil apreciar bien el beneficio de la paralelización de NOA. Por tanto, se ha desarrollado una variante ralentizada de uno de los problemas previos. Este se basa en el problema de TCSD y se le añade carga adicional con las operaciones matriciales descritas en el Pseudocódigo 21:

- Se realizan operaciones de filas sobre la matriz identidad I_m para generar A_m.
- Se calcula la inversa de A_m, y se sabe 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.

Función: Pseudocódigo de la carga artificial sobre la función objetivo TCSD

```
    Output: El determinante de la matriz resultante (que será la matriz identidad).
 1  //Se inicializa la matriz I_m (todo 0s excepto la diagonal con todo 1s)
 2  for i = 0 hasta m do
 3      for j = 0 hasta m do
 4          if i = j then
 5              I[i][j] = 1;
 6          else
 7              I[i][j] = 0;
 8          end
 9      end
10  end
11  //Se realizan operaciones de por filas sobre A_m (no influye que siempre sean las mismas)
12  for i = 0 hasta m do
13      for j = 0 hasta m do
14          for k = 0 hasta m do
15              A[i][k] = A[i][k] + I[j][k] * j;
16          end
17      end
18  end
19  //Se haya la inversa de la matriz resultante A_m
20  A_m^-1 = inversa(A_m);
21  //Se multiplican A_m y A_m^-1 (se sabe que el resultado será la matriz identidad)
22  for i = 0 hasta m do
23      for j = 0 hasta m do
24          for k = 0 hasta m do
25              I[i][j] = A[i][k] * A^-1[k][j];
26          end
27      end
28  end
29  //Se calcula el determinante de I_m (como se sabe que es la identidad se simplifica)
30  determinante = 1;
31  for i = 0 hasta m do
32      determinante = determinante * I[i][i];
33  end
34  return determinante;
```

Pseudocódigo 21: Función de carga artificial mediante operaciones matriciales

En este trabajo se han utilizado matrices de tamaño m = 50, y siempre son cuadradas para que las operaciones sean posibles.

5.3. 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 y presentes en la Tabla 3 son promedios de 30 ejecuciones en ambos programas en todas las funciones *benchmark*. Y como puede observarse los resultados son correctos en ambas versiones, con una desviación típica del orden de cienmilésimas en los peores casos.

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	-4,99999	-4,99995	-5
Neumaier N. 3	-664,99991	-665	-665
Ackley N. 2	-200	-200	-200
Shekel 10	-10, 5364098	-10,5364098	-10,5364098
PVD	5885,332774	5885,332775	5885,3327736164
TCSD	0,01266524	0,01266524	0,0126652327883

Tabla 3: Resultados promedios obtenidos con NOA en MATLAB y en C

Seguidamente, se procede a comparar 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 y presentes en la Tabla 4 son promedios de 30 ejecuciones en ambos programas en todas las funciones *benchmark*. Se ha tomado la aceleración como la fracción de la velocidad original entre la velocidad de la versión de la que se quiere conocer la propia aceleración:

$$A = \frac{Velocidad\ de\ la\ versión\ original\ en\ segundos\ (Matlab)}{Velocidad\ de\ la\ versión\ de\ la\ que\ se\ quiere\ conocer\ la\ aceleración\ en\ s.(C)}$$

Al igual que para los valores óptimos, la variación entre medidas de tiempo de un mismo problema es del orden de milésimas, por lo que se puede omitir la desviación típica.

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
TCSD con carga artificial	40,83834887	7,2461936	5,635834636

Tabla 4: Aceleración de NOA en C sobre MATLAB

Resulta evidente que la versión implementada en C es mucho más rápida que la implementada en MATLAB, llegando a ser 10 veces más veloz para la mayoría de los problemas. Y aunque el caso con carga artificial es el más representativo, en este sigue siendo por lo menos 5 veces más rápido.

5.4. 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) no son los más adecuados para apreciar los beneficios de la paralelización. La situación se agrava especialmente dado el bajo coste computacional de cada evaluación para los problemas considerados.

Específicamente, se puede observar que 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 ese motivo, 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 totales (35000) que se realizan para alcanzar el óptimo. Únicamente afecta a la cantidad de trabajo que realiza cada hebra por generación, reduciendo así el número de generaciones que se calculan. Esto tiene el efecto secundario de disminuir 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.

5.4.1. Sphere

La gráfica presentada en la Figura 11 muestra la aceleración, representada en el eje Y de la gráfica, y cómo cambia conforme se utilizan más hilos durante la optimización de la función (en este caso de la esfera). Además, según el tamaño de la población N se pueden observar cambios distintos en la aceleración. Todas las figuras que representan una gráfica de aceleración a lo largo de los siguientes apartados se han de interpretar de esta forma.

Volviendo a la Figura 11, se puede ver que existe aceleración para el tamaño de población N = 25 con dos y cuatro hebras. Pero es más destacable el aumento en la aceleración para poblaciones mayores, tal y como se había predicho. Concretamente, para dos hebras la aceleración es máxima a partir de N = 100, para cuatro hebras la aceleración es máxima a partir de N = 250 y para ocho hebras la aceleración es máxima a partir de N = 500. En general la aceleración máxima alcanzada sobre esta función no requiere llegar a un tamaño de población de N = 1000.

Es interesante determinar para qué tamaño de población se encuentra la mayor aceleración posible de la optimización de un problema, pues ya se ha explicado que el aumento de esta puede deteriorar los resultados como se observa en la Figura 12. Esta gráfica representa el deterioro causado conforme se aumenta el tamaño de N.

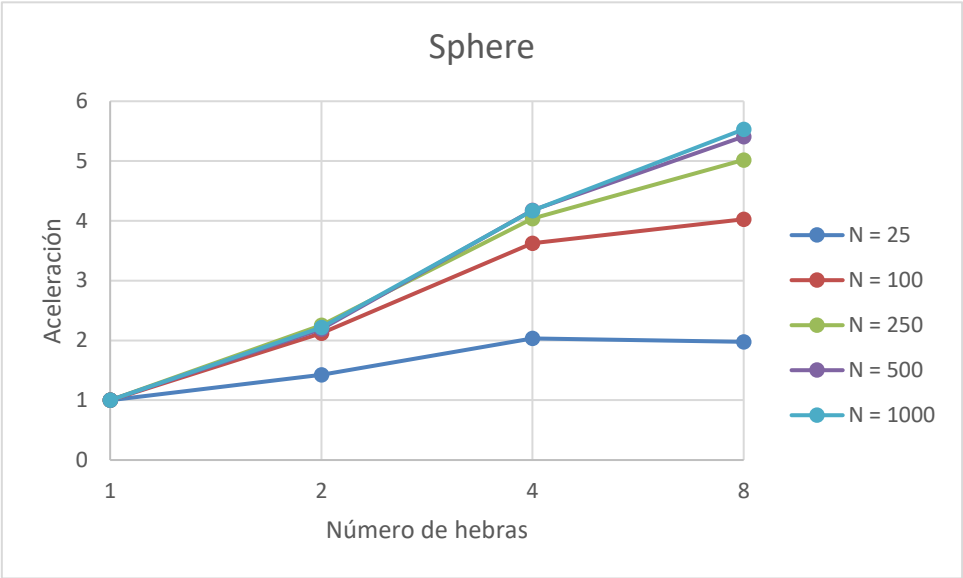

Figura 11: Aceleración sobre la función Sphere

Todas las gráficas de deterioro de los apartados siguientes se interpretan como la de la Figura 12. Los resultados del problema se reflejan en el eje Y, siendo el óptimo el que se encuentra en la intersección con el eje X. Todo valor que sea mayor que el óptimo, por tanto, representa un deterioro en la solución obtenida por el algoritmo.

En este caso podemos tener en cuenta que no hay deterioro en las soluciones hasta N = 500, como se muestra en la Figura 12, y por tanto se puede afirmar que, una vez alcanzada la máxima aceleración posible para este problema, todavía se obtiene la solución óptima en esta función sin deterioro. Se puede concluir entonces que este problema se puede optimizar hasta 5 veces más rápido sin deteriorar la solución.

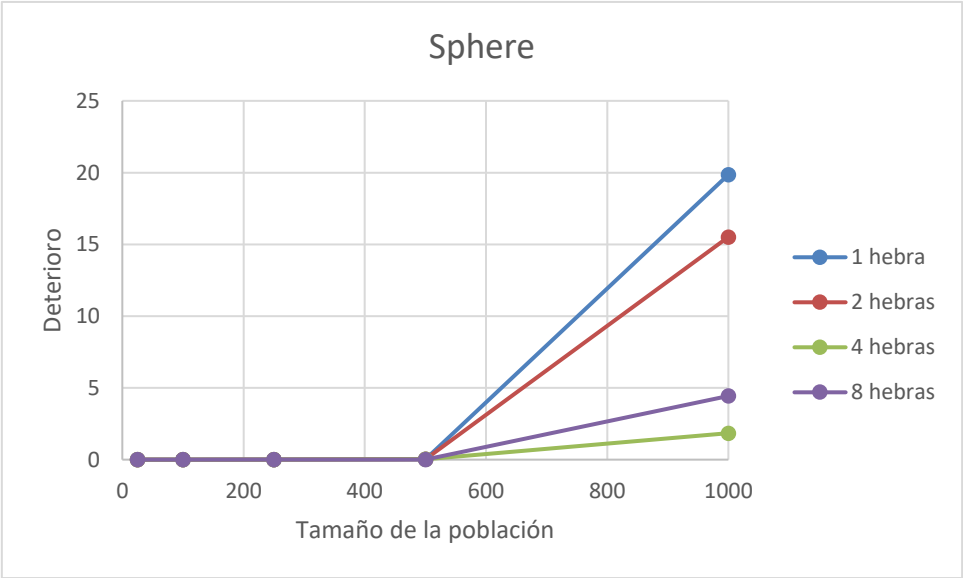

Figura 12: Deterioro de los resultados de la función Sphere

5.4.2. Quartic

Se puede observar en la Figura 13 que la aceleración para el tamaño de población N = 25 con dos y cuatro hebras. Pero es más destacable que para dos hebras la aceleración es máxima a partir de N = 100, para cuatro hebras la aceleración es máxima a partir de N = 250 y para ocho hebras la aceleración es máxima prácticamente también. En general la aceleración sigue un comportamiento semejante al de la función anterior.

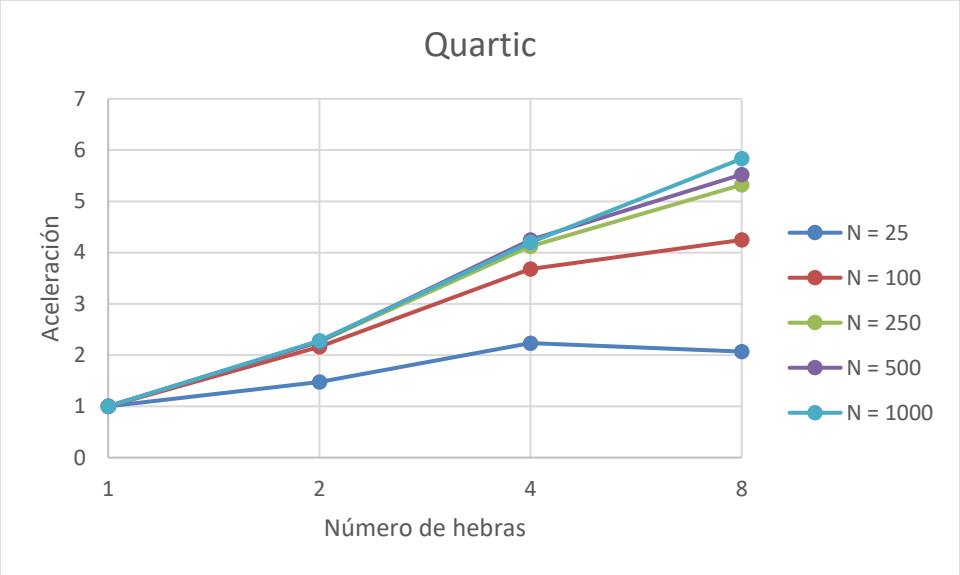

Figura 13: Aceleración sobre la función Quartic

Cuando tenemos en cuenta que no hay deterioro en las soluciones hasta N = 500, como se muestra en la Figura 14, se puede afirmar que se alcanza la máxima aceleración posible para este problema mucho antes de que se deterioren los resultados, de forma semejante a la función anterior.

Figura 14: Deterioro de los resultados de la función Quartic

5.4.3. Powell Sum

Se puede observar en la Figura 15 que sí existe aceleración para el tamaño de población N = 25. Además, es destacable que para dos hebras la aceleración es máxima a partir de N = 100, y para cuatro y ocho hebras la aceleración es máxima a partir de N = 250. En general la aceleración máxima alcanzada sobre esta función no requiere llegar a un tamaño de población de N = 500.

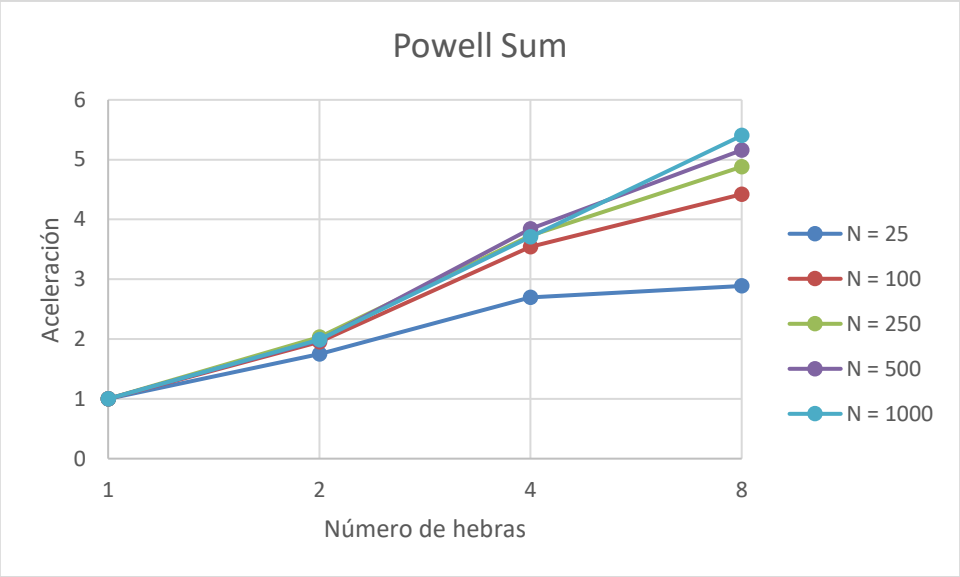

Figura 15: Aceleración sobre la función Powell Sum

Esto es muy relevante en esta función, ya que como se muestra en la Figura 16, en esta ocasión para el tamaño de población N = 500 los resultados ya empiezan a deteriorarse. Igualmente se puede afirmar que se alcanza el máximo rendimiento de esta función antes de que se deterioren los resultados.

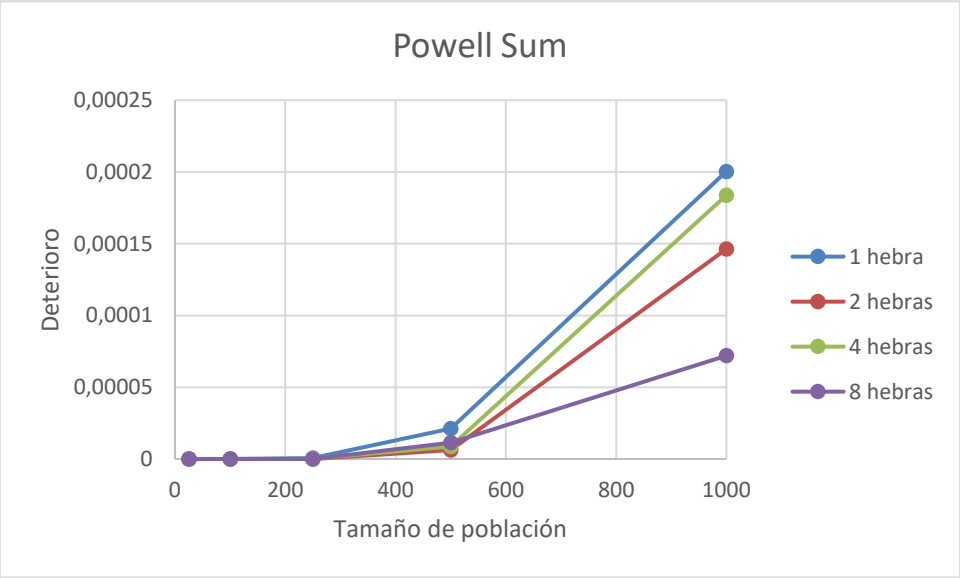

Figura 16: Deterioro de los resultados de la función Powell Sum

5.4.4. Sum Squares

Después de analizar varias funciones unimodales que convergen en el origen (en el 0), se observa un rendimiento casi idéntico para las distintas cantidades de hebras y tamaños de población. Y como de nuevo se puede observar en la Figura 17 la máxima aceleración alcanzada sobre el algoritmo secuencial se encuentra entre 5 y 6, paralelizando con 8 hebras y a partir de un tamaño de población mayor o igual que 250.

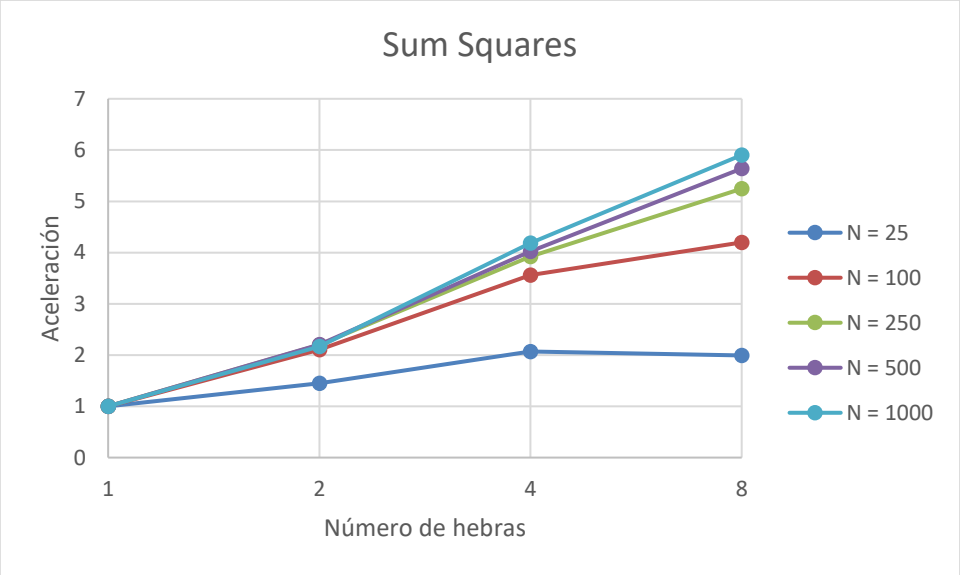

Figura 17: Aceleración sobre la función Sum Squares

Además, para ese tamaño de población de N = 250 las soluciones no llegan a deteriorarse en ninguna de las funciones anteriores, lo cual es ideal. Y por otro lado, en todos los casos los resultados se alejan significativamente del valor óptimo al alcanzar una población de tamaño N = 1000, de la misma manera que en la Figura 18.

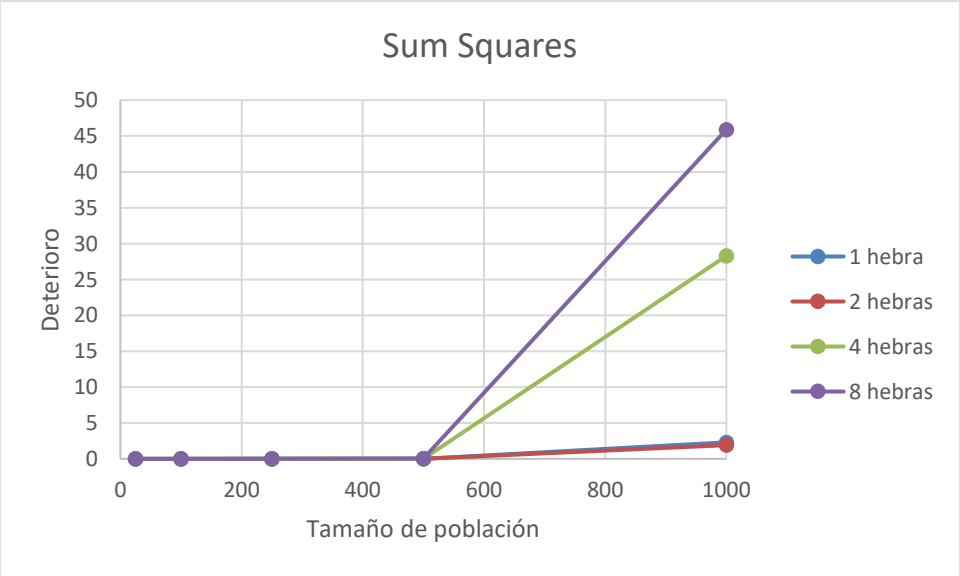

Figura 18: Deterioro de los resultados de la función Sum Squares

5.4.5. Schwefel's 2.20

Otro caso en el que se puede observar una aceleración casi lineal sobre una función unimodal que converge en el origen. Aunque estas 5 funciones compartan estas características, el rendimiento observado en la Figura 19 puede ser representación directa del rendimiento del algoritmo paralelizado mediante *pthreads*.

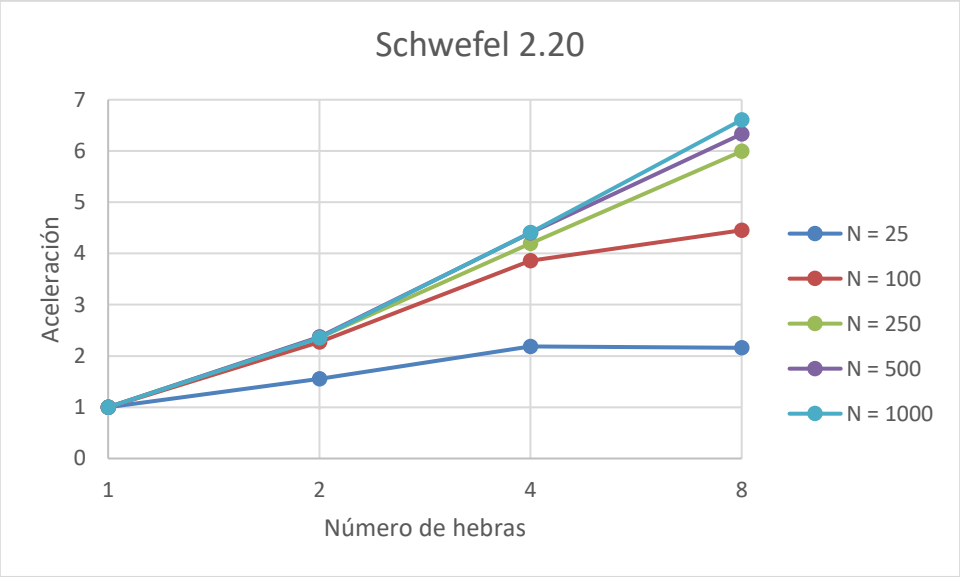

Figura 19: Aceleración sobre la función Schwefel 2.20

Por otro lado, las gráficas del deterioro sí parecen depender más de la definición matemática de la propia función, ya que, como se muestra en la Figura 20, en esta ocasión para el tamaño de población N = 500 los resultados empiezan a deteriorarse. Pero se seguiría pudiendo acercarse a la máxima aceleración sin necesidad de incrementar la población, y por tanto sin deteriorar los resultados.

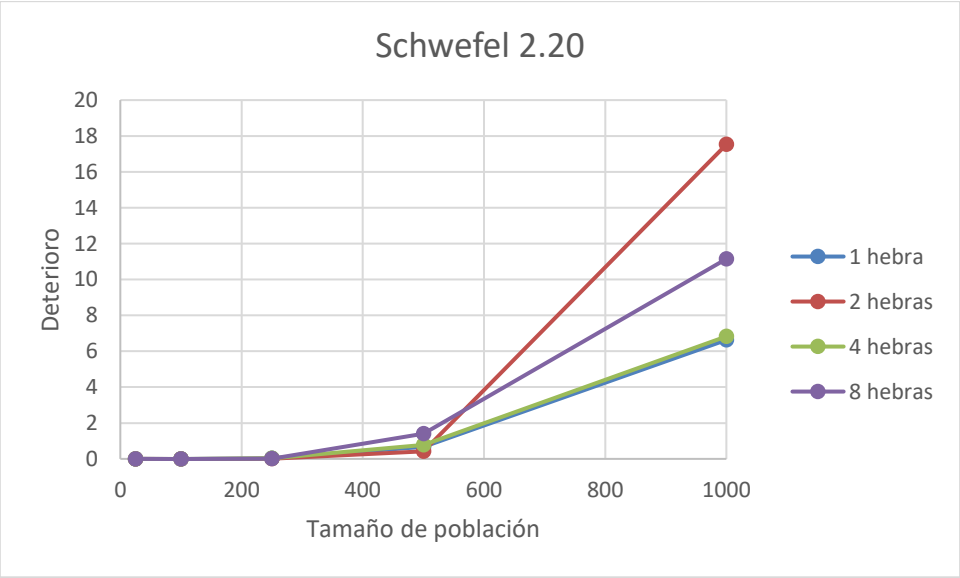

Figura 20: Deterioro de los resultados de la función Schwefel 2. 20

5.4.6. Stepint

Esta es la primera función que no converge en el origen, pero sigue siendo unimodal. Se puede observar en la Figura 21 que para el tamaño de población N = 25 hay una mejora de rendimiento hasta llegar a las 4 hebras, como todas las anteriores funciones. Por tanto, se podría descartar que el rendimiento del algoritmo esté ligado al valor concreto en el que convergen las funciones.

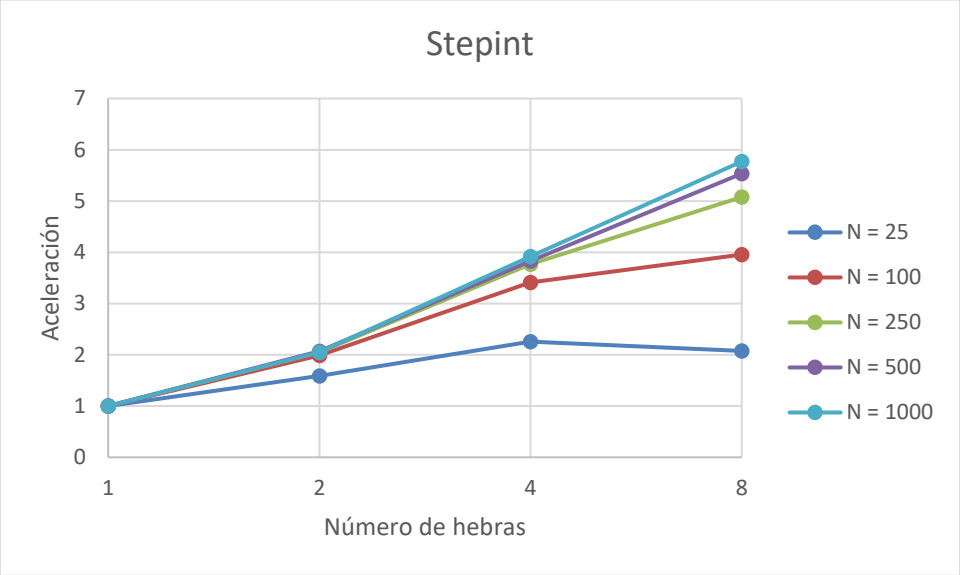

Figura 21: Aceleración sobre la función Stepint

Aparte, el deterioro aparece en las soluciones a partir de N = 100 (ver Figura 22), lo cual no proporciona la mejor aceleración para 8 hebras, pero prácticamente la máxima con 4 hebras. En este caso, y a diferencia de los anteriores, se deberían mantener las mismas evaluaciones por individuo que las utilizadas en la versión secuencial.

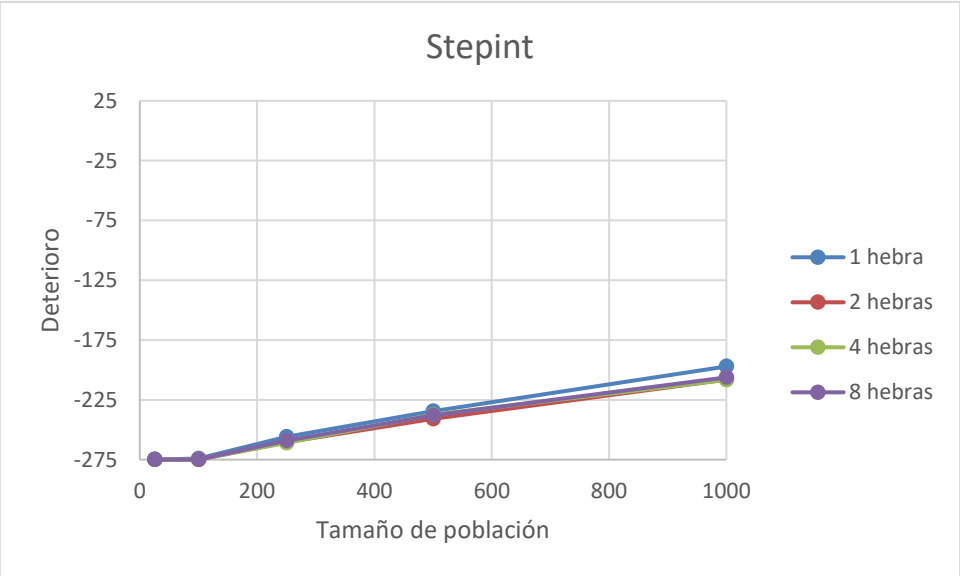

Figura 22: Deterioro de los resultados de la función Stepint

5.4.7. Ridge

Se puede observar en la Figura 23 que solo existe aceleración hasta 4 hebras para el tamaño de población N = 25. Además, es destacable que para dos hebras la aceleración es máxima a partir de N = 100, y para cuatro hebras la aceleración es máxima a partir de N = 250. En general se puede obtener el mejor rendimiento sobre esta función con un tamaño de población menor que N = 500.

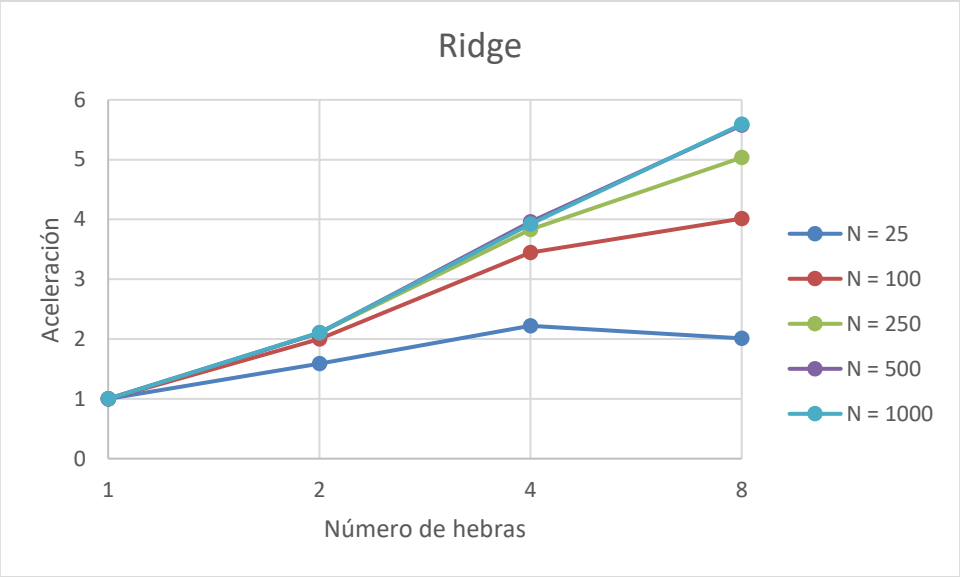

Figura 23: Aceleración sobre la función Ridge

Por otro lado, y al igual que en la función anterior, los resultados se deterioran a partir de N = 100, como se observa en la Figura 24 lo cual no proporciona la mejor aceleración para 8 hebras, pero prácticamente la máxima con 4 hebras. De nuevo no es recomendable aumentar el tamaño de la población demasiado.

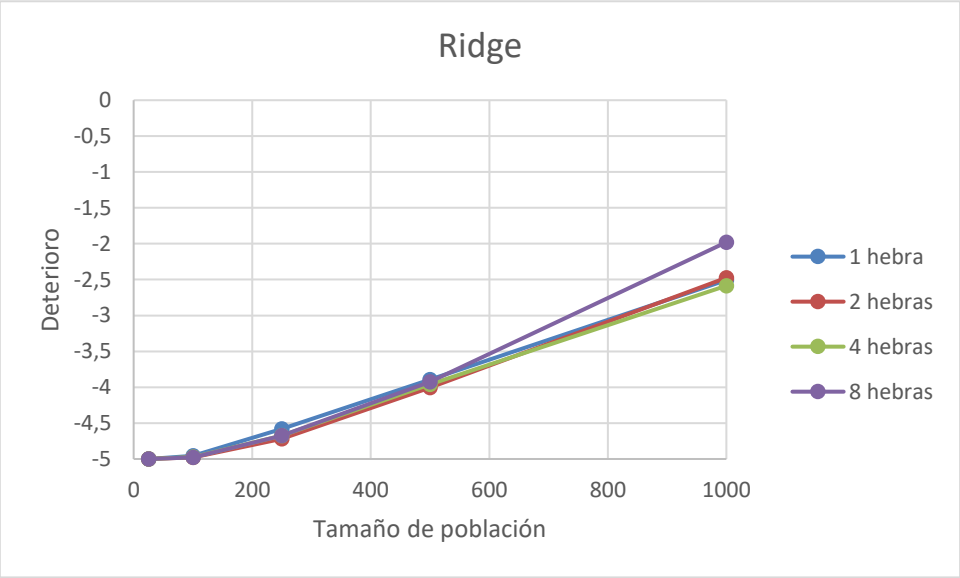

Figura 24: Deterioro de los resultados de la función Ridge

5.4.8. Neumaier's N. 3

Esta es la primera función con dimensión menor que 50, e inmediatamente se observa en la Figura 25, que solo hay aceleración real si se eleva el tamaño de la población por encima de N = 25. Se puede descartar entonces que el rendimiento esté asociado a funciones unimodales, pues esta también es unimodal.

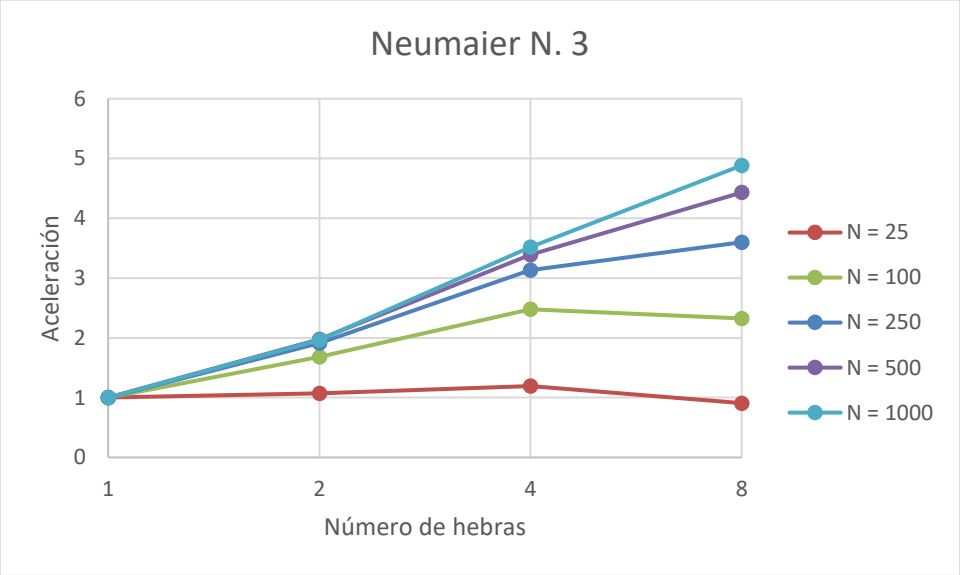

Figura 25: Aceleración sobre la función Neumaier N. 3

En cambio, no hay deterioro visible en la Figura 26 hasta apenas llegar a N = 250. Después de diferentes funciones que se deterioran en menor o mayor medida, se puede afirmar que este hecho solo depende de lo profundo que se encuentre el óptimo y por tanto de cuántas evaluaciones se le realicen a cada individuo.

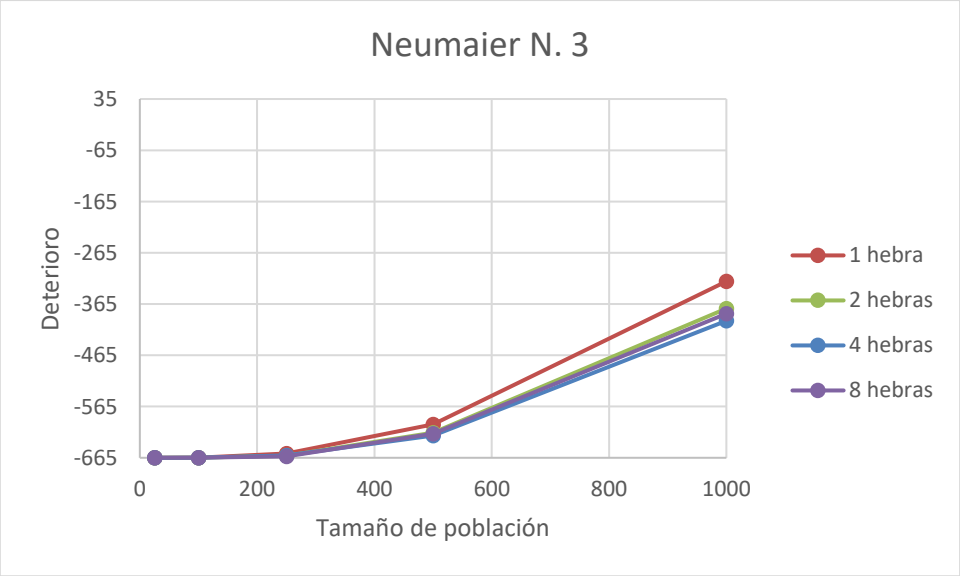

Figura 26: Deterioro de los resultados de la función Neumaier N. 3

5.4.9. Ackley N. 2

Se puede observar en la Figura 27 que no existe aceleración si no se eleva el tamaño de la población por encima de 25, siendo el paralelismo perjudicial en el rendimiento para N = 25.

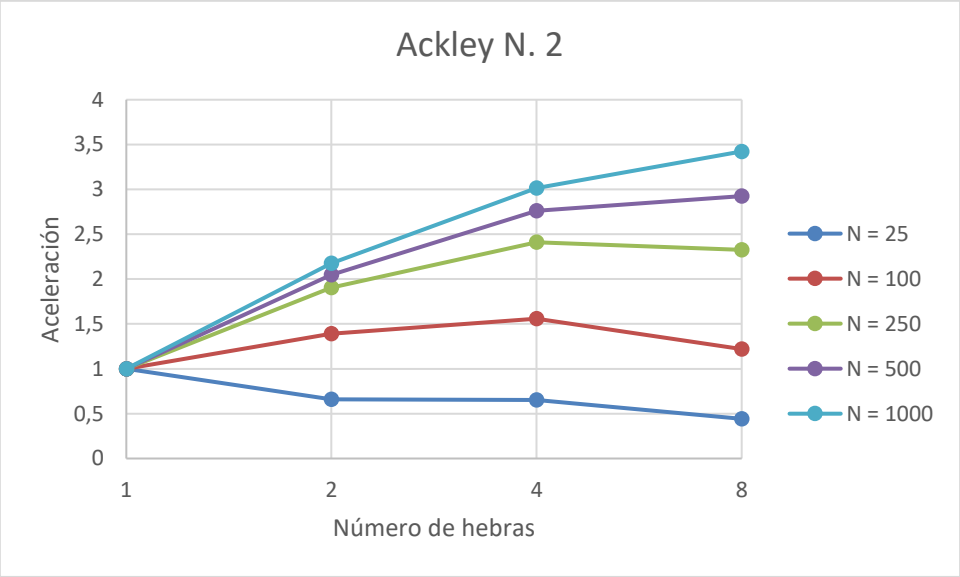

Figura 27: Aceleración sobre la función Ackley N. 2

En cambio, no hay apenas deterioro en las soluciones hasta N = 500, por lo que se podría conseguir un rendimiento aceptable (algo por encima del doble de velocidad que en secuencial) si se utilizan 4 hebras y un tamaño de población que no deteriore significativamente la solución (N = 500 como se interpreta en la Figura 28).

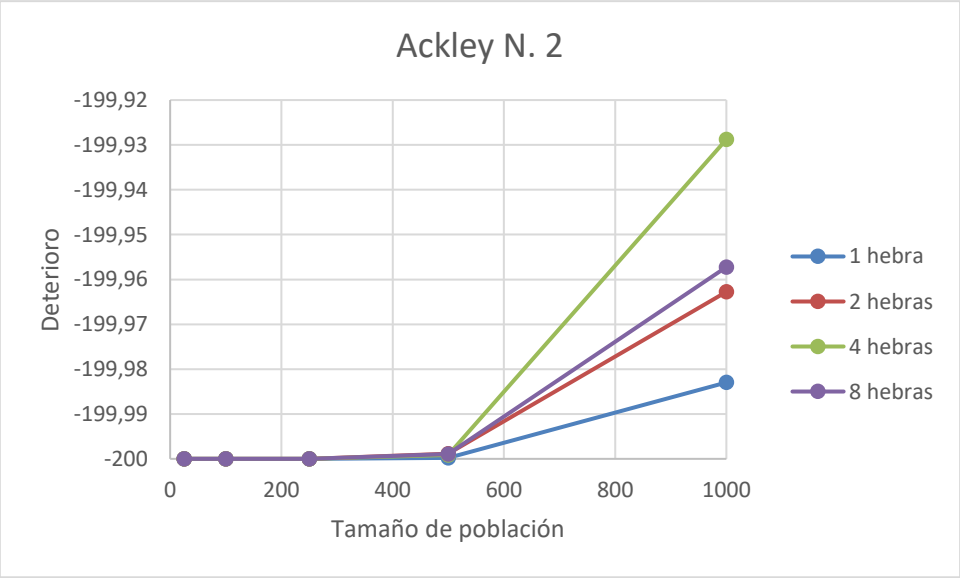

Figura 28: Deterioro de los resultados de la función Ackley N. 2

5.4.10. Shekel 10

Se puede observar en la Figura 29 un rendimiento similar al de la Figura 27, siendo el paralelismo perjudicial en el rendimiento para N = 25. En este caso la función es multimodal, descartando de nuevo que esta característica tenga que ver con la velocidad de ejecución del algoritmo.

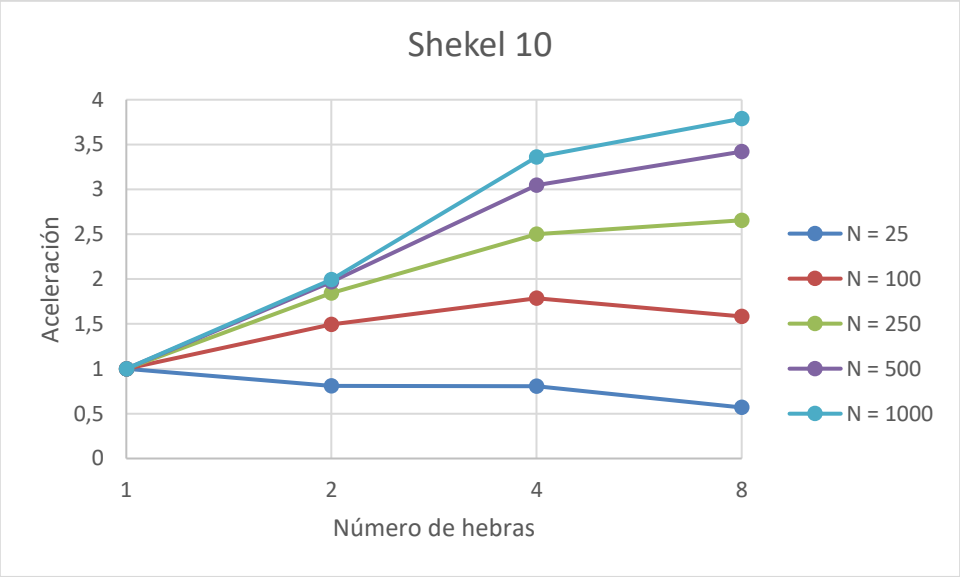

Figura 29: Aceleración sobre la función Shekel 10

En la gráfica de deterioro de esta función, visible en la Figura 30, se observa de nuevo un deterioro de la solución a partir de un tamaño de población superior a 250. Si se quisiese conseguir el mayor rendimiento para esta función sin deteriorar el óptimo, se debería aumentar el tamaño de población a N = 250 y generar la cantidad de hebras adecuadas para que el algoritmo aprovechase todos los recursos sin sobrecargar el sistema (en este caso 8).

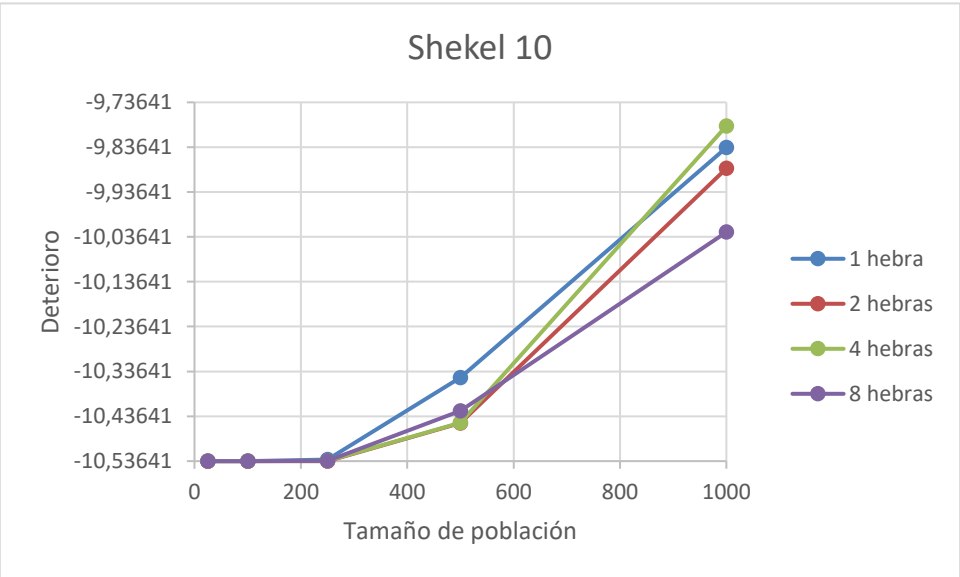

Figura 30: Deterioro de los resultados de la función Shekel 10

5.4.11. PVD

Este es el primer problema de ingeniería real, y como se puede observar en la Figura 31, parece que la sobrecarga de la sincronización de hebras es mayor que la carga que supone resolver la función objetivo para tamaños de población menores que 250. De hecho, esto es lo que se ha observado en todas las gráficas de aceleración de funciones con una dimensión menor que 50.

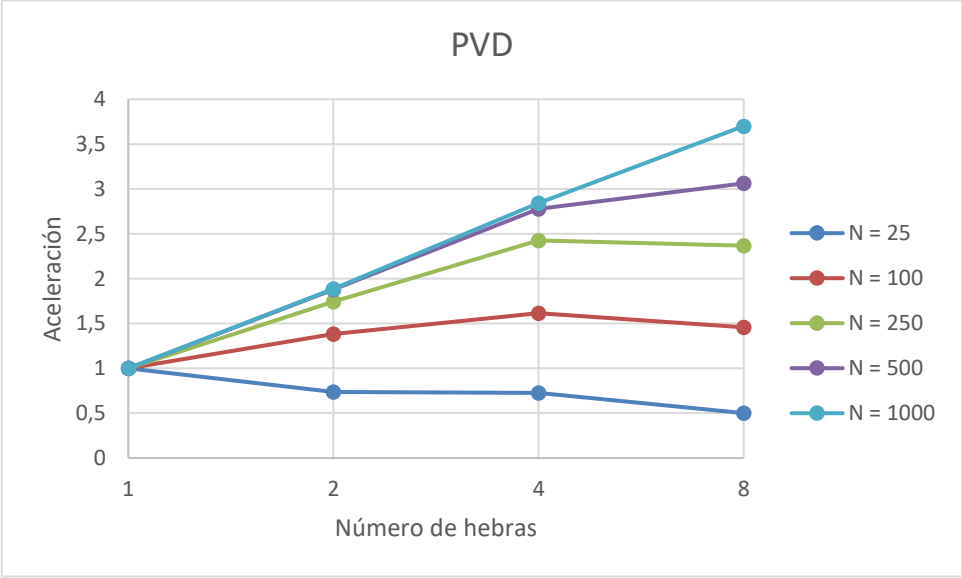

Figura 31: Aceleración sobre el problema de PVD

Además, como se puede ver en la Figura 32, a partir de un tamaño de población mayor que 100 ya existe deterioro. Esto puede deberse a que es un problema con restricciones y la velocidad de convergencia es aún menor que en las últimas funciones analizadas. Por lo que apenas se puede acelerar sobre la versión secuencial.

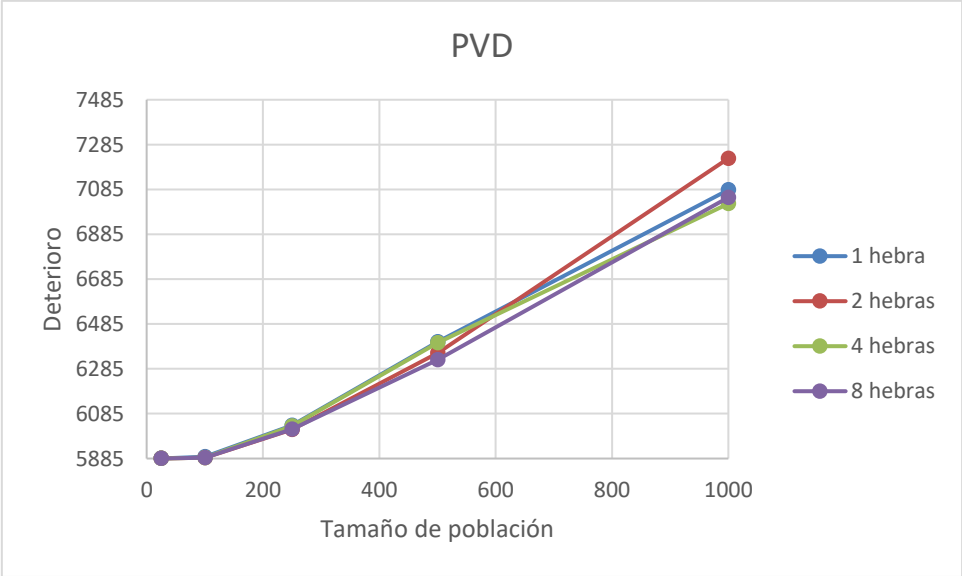

Figura 32: Deterioro de los resultados del problema de PVD

5.4.12. TCSD

En esta caso, igual que ocurre en el problema anterior, se obtiene (Figura 33) un rendimiento muy pobre, o incluso negativo, para tamaños de población menores que 250.

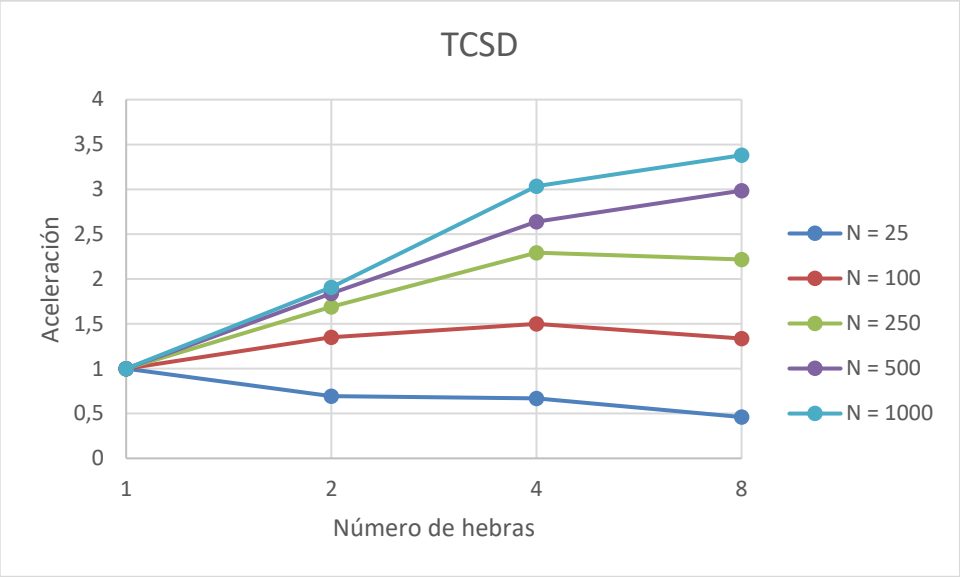

Figura 33: Aceleración sobre el problema de TCSD

Y de nuevo en la gráfica sobre el deterioro, presente en la Figura 34, se observan resultados mayores al óptimo tras aumentar el tamaño de la población hacia N = 250. Esto no permite un aprovechamiento de la versión paralela con *pthreads* del algoritmo, pero como se ha deducido, se debe a una baja carga computacional en la función objetivo y las restricciones del problema.

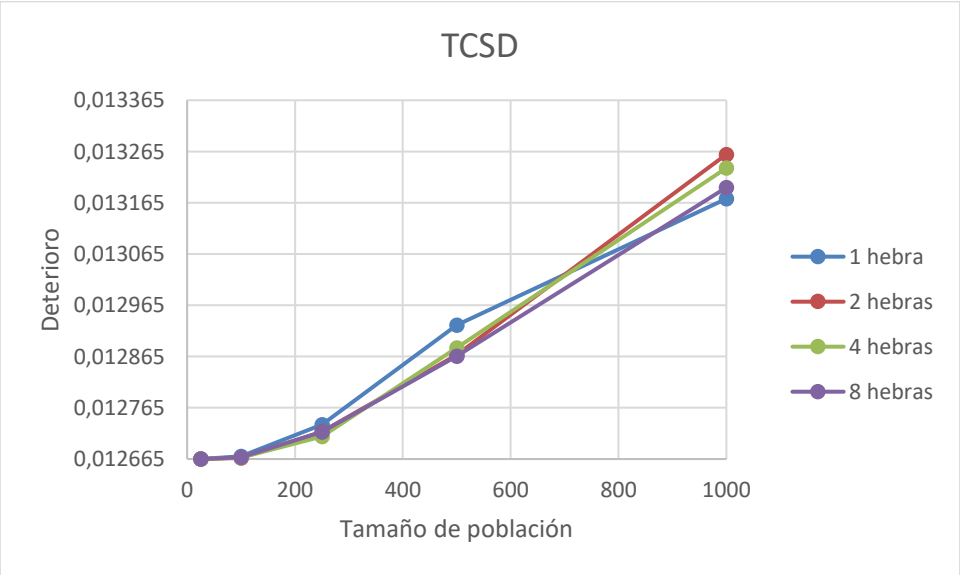

Figura 34: Deterioro de los resultados del problema de TCSD

5.4.13. Problema ralentizado artificialmente

Antes del análisis de las mediciones tomadas se razonó la posibilidad de que las funciones fuesen demasiado poco costosas computacionalmente y cómo se podrían deteriorar los resultados al intentar tener un tamaño de población que mejorase el rendimiento de la versión con paralelismo por hebras. En la Figura 35 se demuestra inmediatamente cómo un problema con mayor carga computacional, en este caso generada artificialmente, aprovecha al máximo el rendimiento de la versión paralela, independientemente de prácticamente cualquier tamaño de población..

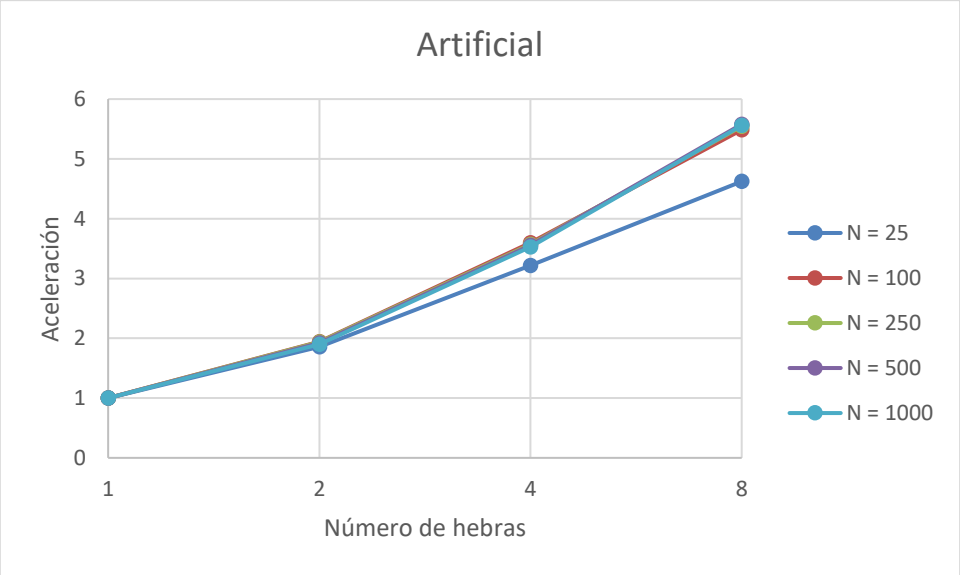

Figura 35: Aceleración sobre el problema con carga artificial

El deterioro es el mismo observado en el problema anterior, de hecho, la Figura 34 y la Figura 36 son iguales. Ya se había observado que el deterioro es mayor cuanto más se tarda en converger en el óptimo, y por tanto depende de la complejidad de la función objetivo del problema (y de las restricciones si las tiene).

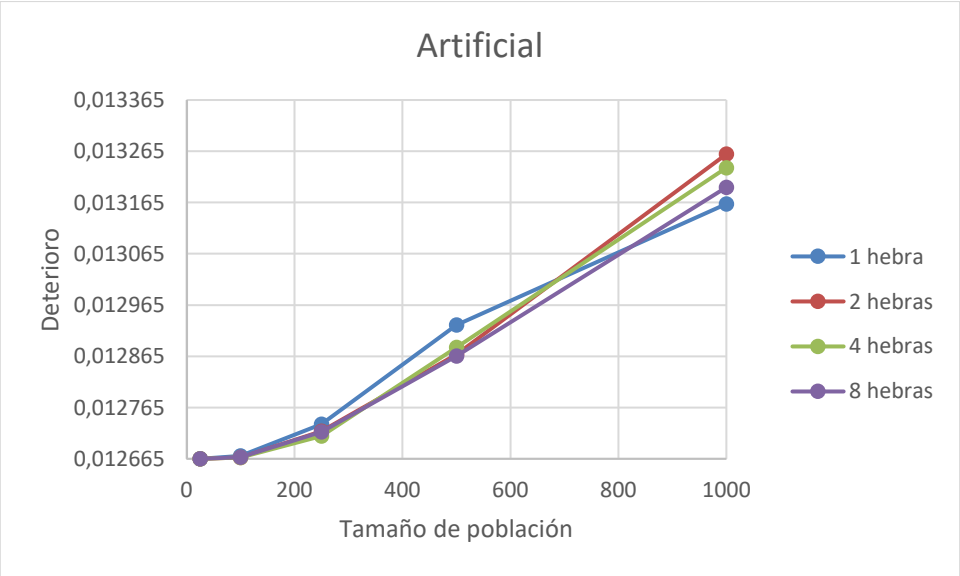

Figura 36: Deterioro de los resultados del problema con carga artificial

5.5. Rendimiento de la versión híbrida combinando *pthreads* y MPI

Para comprobar la efectividad de esta versión, no tiene sentido realizar mediciones que se comparen con las realizadas con la versión de *pthreads*, con la misma cantidad de unidades de procesamiento. Por ejemplo, si se realizase una ejecución con 2 procesos y 4 hebras cada uno, sería equivalente a un proceso con 8 hebras en una máquina con memoria compartida. Por esta razón se ha realizado una comprobación de rendimiento en un multicomputador con 2 nodos que se deben comunicar utilizando los mensajes de MPI.

Como en el apartado anterior se ha determinado claramente que es necesario medir un problema suficientemente costoso para que las ventajas del paralelismo entren en juego, la versión híbrida se ha probado únicamente con el problema ralentizado artificialmente.

En primer lugar, se ha ejecutado la versión secuencial como referencia, en la misma máquina (no tiene sentido comparar el rendimiento en máquinas distintas porque se estaría comprobando entonces cuál de ellas es más rápida y no el rendimiento del paralelismo). A continuación, la versión híbrida se ha probado con 2 procesos y 8 hilos cada uno, 16 hilos en total. Ambas ejecuciones se han realizado 10 veces, y sus promedios se muestran en la Tabla 5. También se ha obtenido la aceleración de la versión híbrida sobre la secuencial.

Función	***Secuencial (s)***	***Híbrida (s)***	***Aceleración***
TCSD con carga artificial	9,5587688	0,732138	13,05596595

Tabla 5: Aceleración de la versión híbrida en un clúster

En esta simulación se ve que el programa escala bien y que el hecho de contar con una versión híbrida permite poder explotar ese tipo de configuraciones. Pues si no se tuviese una versión híbrida, no se podrían combinar máquinas. Y este tipo de configuraciones son comunes, especialmente en clústeres de bajo coste (43).

Si se tiene en cuenta que la máxima aceleración que se puede obtener es 16, el valor obtenido es muy buen resultado. De hecho, este rendimiento es el que cabría esperar en algoritmos paralelizables para computación de alto rendimiento (HPC).

6. Conclusiones y trabajo futuro

6.1. Conclusiones

En este trabajo se ha estudiado la paralelización de un reciente algoritmo de optimización llamado Nizar con el fin de desarrollar una implementación paralela de código abierto de dicho método. Realizando finalmente tres implementaciones, una secuencial, otra paralelizando únicamente con *pthreads,* y una última combinando el paralelismo de *pthreads* y MPI.

6.1.1. Ventajas del uso de lenguajes de programación compilados

El algoritmo ya se ejecuta por lo menos 5 veces más rápido en su versión C que en su equivalente en MATLAB. Esto sugiere que, independientemente del problema que se tuviese que optimizar, el uso de NOA en C es mucho más eficiente que en MATLAB. Además, 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.

Esto era de esperar, y se ha comentado varias veces a lo largo del trabajo, pues C es un lenguaje de programación de más bajo nivel, que por ejemplo te da un control claro sobre la memoria, lo que no ocurre con MATLAB. Pero aunque fuese lo esperado, la comprobación empírica es necesaria para que sea irrebatible.

6.1.2. Beneficios del uso de computación paralela

Se puede apreciar que la aceleración en funciones más costosas, concretamente en todas aquellas que trabajan con dimensión mayor que D = 50 (como se ha destacado durante el análisis de los datos), mejora las prestaciones del algoritmo. Por otro lado, en problemas con dimensiones pequeñas, D < 15, la sobrecarga es mayor que lo que se obtiene al repartir el trabajo. 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.

Esto se debe a que el coste de las funciones *benchmark* (excluyendo la ralentizada artificialmente) 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 modificando el tamaño de N, que como se puede apreciar supone una mejora 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 que también se preveía antes de hacer la comprobación empírica. Esto se podría haber compensado manteniendo el ratio de evaluaciones por individuo, pero entonces también se habría tenido que modificar el parámetro de ejecución de “número de evaluaciones”, el cual se establecía a 35000 de forma constante para todo el trabajo original (2) (y ya se comentó que para no llegar a soluciones distintas a las originales también se mantendría este valor).

En cualquier caso, esto no indica que el paralelismo no conlleve beneficios sobre la versión secuencial. En los casos anteriores ya se ha visto que 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. Pero si tienen dimensiones más grandes, aún con poca carga computacional, la versión paralela siempre es más rápida. O lo que es lo mismo, la versión paralela puede evaluar muchas más soluciones por unidad de tiempo que la secuencial, pudiendo trabajar con más individuos y evaluaciones en el mismo tiempo efectivo que la versión secuencial.

6.1.3. Potencial para problemas computacionalmente costosos

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 alcanzada por el programa sobre la función para el tamaño de población original y menor de todos los medidos, N = 25.

Este gran rendimiento se ve tanto en la versión de *phtreads* como en la híbrida. Lo que significa que el uso del paralelismo supone siempre un aumento de rendimiento en estos casos y por ende el algoritmo NOA es escalable y viable para la computación de altas prestaciones.

6.3. Trabajo futuro

El siguiente paso sería realizar experimentos donde se mantiene el ratio de evaluaciones de los individuos.

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: SE Khouni, T. Menacer. *The Journal of Supercomputing*, *80*(3), 3229-3281.

(3) Costa, A., & Nannicini, G. (2018). RBFOpt: an open-source library for black-box optimization with costly function evaluations. *Mathematical Programming Computation*, *10*(4), 597-629.

(4) Schwarzbach, L., & Schmitt, R. (2024). Heuristic Methods. In *Operations Research and Management: Quantitative Methods for Planning and Decision-Making in Business and Economics* (pp. 131-154). Cham: Springer Nature Switzerland.

(5) Boussaïd, I., Lepagnot, J., & Siarry, P. (2013). A survey on optimization metaheuristics. *Information sciences*, *237*, 82-117.

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

(7) 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.

(8) Calvo Cruz, N., Puertas Martín, S., L Redondo, J., & M Ortigosa, P. (2023). An Effective Solution for Drug Discovery Based on the Tangram Meta-Heuristic and Compound Filtering.

(9) Nichols, B., Buttlar, D., & Farrell, J. (1996). *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.".

(10) Trobec, R., Slivnik, B., Bulić, P., & Robič, B. (2018). *Introduction to parallel computing: from algorithms to programming on state-of-the-art platforms*. Springer.

(11) Cruz, N. C., Redondo, J. L., Álvarez, J. D., Berenguel, M., & Ortigosa, P. M. (2018). Optimizing the heliostat field layout by applying stochastic population-based algorithms. *Informatica*, *29*(1), 21-39.

(12) 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.

(13) Guillén Perales, A., Martínez Cámara, E., Fernández Luna, J. M., García Sánchez, P., Noguera García, M., Rodríguez Fórtiz, M. J., & Romero Zaliz, R. C. (2025). Cómo escribir la memoria de tu TFG del Grado en Ingeniería Informática y presentarlo sin morir en el intento.

(14) https://www.canva.com. Accedido el 5 de septiembre de 2025.

(15) Ruszczynski, A. (2011). *Nonlinear optimization*. Princeton university press.

(16) Pedregal, P. (2004). *Introduction to optimization* (Vol. 46). New York: Springer.

(17) Wright, S., & Nocedal, J. (1999). Numerical optimization. *Springer Science*, *35*(67-68), 7.

(18) Powell, M. J. (1994). A direct search optimization method that models the

objective and constraint functions by linear interpolation. In *Advances in optimization and numerical analysis* (pp. 51-67). Dordrecht: Springer Netherlands.

(19) Pronzato, L., Wynn, H. P., & Zhigljavsky, A. A. (1998). A generalized golden-section algorithm for line search. *IMA Journal of Mathematical Control and Information*, *15*(2), 185-214.

(20) Meza, J. C. (2010). Steepest descent. *Wiley Interdisciplinary Reviews: Computational Statistics*, *2*(6), 719-722.

(21) Lasdon, L., Mitter, S., & Waren, A. (2003). The conjugate gradient method for optimal control problems. *IEEE Transactions on Automatic Control*, *12*(2), 132-138.

(22) Beheshti, Z., & Shamsuddin, S. M. H. (2013). A review of population-based meta-heuristic algorithms. *Int. j. adv. soft comput. appl*, *5*(1), 1-35.

(23) Nelder, J. A., & Mead, R. (1965). A simplex method for function minimization. *The computer journal*, *7*(4), 308-313.

(24) Wright, S. J. (1997). *Primal-dual interior-point methods*. Society for Industrial and Applied Mathematics.

(25) Fogel, D. B., & Fogel, L. J. (1995, September). An introduction to evolutionary programming. In *European conference on artificial evolution* (pp. 21-33). Berlin, Heidelberg: Springer Berlin Heidelberg.

(26) Holland, J. H. (1992). Genetic algorithms. *Scientific american*, *267*(1), 66-73.

(27) Kennedy, J., & Eberhart, R. (1995, November). Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks* (Vol. 4, pp. 1942-1948). ieee.

(28) Rahkar Farshi, T. (2021). Battle royale optimization algorithm. *Neural Computing and Applications*, *33*(4), 1139-1157.

(29) Fisher, R. A., & Yates, F. (1953). *Statistical tables for biological, agricultural and medical research*. Hafner Publishing Company.

(30) Cruz, N. C., Redondo, J. L., Álvarez, J. D., Berenguel, M., & Ortigosa, P. M. (2017). A parallel Teaching–Learning-Based Optimization procedure for automatic heliostat aiming. *The Journal of Supercomputing*, *73*(1), 591-606.

(31) Latter, B. D. H. (1973). The island model of population differentiation: a general solution. *Genetics*, *73*(1), 147.

(32) https://www.mpich.org/static/docs/v3.2/www3/MPI_Allgatherv.html. Accedido el 5 de septiembre de 2025.

(33) Pickard, J. K., Carretero, J. A., & Bhavsar, V. C. (2016). On the convergence and origin bias of the teaching-learning-based-optimization algorithm. *Applied Soft Computing*, *46*, 115-127

(34) Librería virtual de experimentos de simulación: Funciones test para problemas de optimización, por Sonja Surjanovic & Derek Bingham, Simon Fraser University, 2013. https://www.sfu.ca/~ssurjano/spheref.html. Accedido el 19 de agosto de 2025.

(35) Jamil, M., & Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, *4*(2), 150-194.

(36) Librería virtual de experimentos de simulación: Funciones test para problemas de optimización, por Sonja Surjanovic & Derek Bingham, Simon Fraser

University, 2013. https://www.sfu.ca/~ssurjano/sumsqu.html. Accedido el 19 de agosto de 2025.

(37) Zhao, J., & Gao, Z. M. (2020, November). An improved grey wolf optimization algorithm with multiple tunnels for updating. In *Journal of Physics: Conference Series* (Vol. 1678, No. 1, p. 012096). IOP Publishing.

(38) Ali, M. M., Khompatraporn, C., & Zabinsky, Z. B. (2005). A numerical evaluation of several stochastic algorithms on selected continuous global optimization test problems. *Journal of global optimization*, *31*(4), 635-672.

(39) https://www.indusmic.com/post/ackleyn-2function. Accedido el 19 de agosto de 2025.

(40) Yeniay, Ö. (2005). Penalty function methods for constrained optimization with genetic algorithms. *Mathematical and computational Applications*, *10*(1), 45-56.

(41) Yang, X. S., Huyck, C., Karamanoglu, M., & Khan, N. (2013). True global optimality of the pressure vessel design problem: a benchmark for bio-inspired optimisation algorithms. *International Journal of Bio-Inspired Computation*, *5*(6), 329-335.

(42) 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.

(43) Diwedi, D. V., & Sharma, S. J. (2018, November). Development of a low cost cluster computer using Raspberry Pi. In *2018 IEEE Global Conference on Wireless Computing and Networking (GCWCN)* (pp. 11-15). IEEE.