Universidad de las Ciencias Informáticas
Facultad 6

Título: Algoritmos paralelos de optimización por Nube de Partículas.

Trabajo de Diploma para optar por el Título de Ingeniero en Ciencias Informáticas

Autor: Adrián Quintero Henríquez

Tutor(es): Dr. Rafael Arturo Trujillo Rasúa
Dr. Liesner Acevedo Martínez

Ciudad de la Habana, Junio del 2010

Declaración de autoría

Declaro ser autor de la presente tesis y reconozco a la UCI los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de ______________ del año ________.

____

Adrián Quintero Henríquez

____

Asesor del DDC de Programación
Dr. Rafael Arturo Trujillo Rasúa

____

Asesor del DDC de Programación
Dr. Liesner Acevedo Martínez

Datos de contacto

Tutores:

Dr. Rafael Arturo Trujillo Rasúa
Universidad de las Ciencias Informáticas, Habana, Cuba.
Email: trujillo@uci.cu

Dr. Liesner Acevedo Martínez
Universidad de las Ciencias Informáticas, Habana, Cuba.
Email: frodo@uci.cu

Agradecimientos

Primeramente quisiera agradecer a mis padres por ayudar en mi formación, apoyarme siempre que lo necesitaba y trabajar tanto para que yo cumpliera mis sueños. A mi hermano Ailán: gracias a ti me propuse esforzarme un poco más durante mi carrera y confío que algún día puedas llegar a este momento.

Agradecimiento especial a mis tutores Trujillo y Liesner por brindarme la oportunidad de pertenecer al grupo, confiar en mí, conducirme por el mundo de la programación paralela y guiarme durante estos años. Mis agradecimientos a mi colega Rigo por toda la invaluable ayuda prestada y compartir sus conocimientos conmigo desde el primer momento en que pertenecí al grupo. Agradezco a Lesley por contribuir bastante en mi formación profesional, por sus buenos consejos y brindarme su mano cuando la necesitaba.

Muchas gracias a Conchita, que junto a mi padre, dedicaron parte de su preciado tiempo en mejorar mi trabajo. Quiero agradecer a mis amistades, compañeros de aula y profesores que me han ayudado de una forma u otra a lo largo de mi carrera, en especial a Yoan, Odalis, Rosnel, Carlos, Edel, Manzano, Cuan, Yosvany, Renier, Cesar y Ana Li.

No podía dejar de agradecer a la Revolución Cubana por todas las posibilidades que me ha brindado para superarme y prepararme en el campo de la Informática.

A todos les digo sinceramente: MUCHAS GRACIAS.

Dedicatoria

A mis padres por brindarme tanto amor y ser mi inspiración.
A Ailán, Naylen y Leandro por ser magníficas personas.
A toda mi familia por su ayuda incondicional.

Resumen

Con el desarrollo de la ciencia, en la actualidad se presentan muchos problemas de optimización los cuales son computacionalmente costosos. La Optimización por Nube de Partículas es una técnica novedosa que ha obtenido buenos resultados en varios problemas donde ha sido aplicada. El presente trabajo de diploma se ha centrado en el estudio de este algoritmo y en el desarrollo de una biblioteca paralela eficiente, portable y de propósito general de la Optimización por Nube de Partículas. Se aborda la aplicación de la biblioteca en el Problema Inverso Aditivo de Valores Singulares donde se obtuvieron muy buenos resultados. Además se propone una hibridación de la Optimización por Nube de Partículas con el método de Newton-Bisección, con la cual se lograron soluciones de alta precisión, calidad y se redujo el tiempo de ejecución.

Palabras Claves: *biblioteca paralela, Optimización por Nube de Partículas, Problema Inverso Aditivo de Valores Singulares, rendimiento*

Abstract

With the development of science are currently many optimization problems which are computationally expensive. The Particle Swarm Optimization is a novel technique has good results in several problems where it has been applied. This dissertation has focused on the study of this algorithm and the parallel development of an efficient library, portable, general-purpose for Particle Swarm Optimization. It addresses the implementation of the library in Problem Inverse Additive Singular Value where we obtained very good results. Besides an Optimization by hybridization Particle Swarm Optimization with the method of Newton-Bisection, with which solutions achieved high precision, quality and reduced time implementation.

Keywords: *Inverse Additive Singular Value Problem, parallel library, Particle Swarm Optimization, performance*

Índice general

Agradecimientos I

Dedicatoria II

Resumen III

Índice de figuras VI

Índice de tablas VII

Índice de algoritmos VIII

Introducción 1

1. Revisión bibliográfica. 4

1.1. Conceptos básicos de optimización. 4
1.2. Técnicas metaheurísticas. 5
1.3. Optimización por Nube de Partículas. 8
1.3.1. Descripción de la optimización por nube de partículas. 9
1.3.2. Topologías de PSO. 13
1.3.3. Variantes de PSO para diferentes espacios de búsqueda. 15
1.4. Introducción a la computación paralela. 18
1.4.1. Arquitecturas paralelas. 18
1.4.2. Evaluación de los algoritmos paralelos. 20
1.4.3. Modelos de algoritmos paralelos 22
1.5. Optimización por nube de partículas en entornos paralelos 23
1.5.1. Modelos con población global 23
1.5.2. Modelos de islas 25
1.5.3. Otras implementaciones 26
1.6. Conclusiones del capítulo. 27

2. Herramientas y metodologías. **28**
2.1. Herramientas software. 28
2.1.1. Lenguaje de programación C/C++. 28
2.1.2. Colección de compiladores GCC. 28
2.1.3. Entorno integrado de desarrollo Code::Blocks. 29
2.1.4. Entorno de paso de mensajes 29
2.1.5. Bibliotecas de Álgebra Lineal 29
2.1.6. Kile. 30
2.1.7. KBibTex. 30
2.1.8. OpenOffice. 30
2.2. Herramientas hardware. 30
2.3. Metodologías. 30

3. Resultados y discusión. **31**
3.1. Introducción 31
3.2. Estructura de la biblioteca de optimización por Nube de Partículas. 32
3.2.1. Principales métodos de las clases de la biblioteca 33
3.3. Experimentos con algunos problemas de optimización 35
3.4. Caso de estudio: Problema Inverso Aditivo de Valores Singulares 38
3.4.1. Resolución del problema 39
3.4.2. Pruebas a las diferentes variantes implementadas 40
3.4.3. Hibridación con el método Newton-Bisección 43
3.5. Paralelización de la biblioteca en el PIAVS 44
3.5.1. Modelo Maestro-Esclavo 44
3.5.2. Modelo de Islas 50

Conclusiones **52**

Recomendaciones **53**

Referencias bibliográficas **54**

A. Pruebas de la biblioteca en el PIAVS **59**

B. Pruebas de la biblioteca híbrida en el PIAVS **63**

Glosario de términos **67**

Índice de figuras

1.1. Clasificación de las metaheurísticas. 8
1.2. Secuencia de un algoritmo de PSO. 11
1.3. Movimiento de una partícula en el espacio de soluciones. 12
1.4. Entorno geográfico y social en el espacio de soluciones. 13
1.5. Topologías de la nube de partículas. 15
1.6. Principales formas de intercambio de datos. 20
1.7. Algoritmo paralelo de PSO con población global. 24

3.1. Diagrama de clases de la biblioteca. 33
3.2. Gráfica para comparar tiempo de ejecución (Standard 2006 vs Biblioteca). 37
3.3. Comparación entre las variantes implementadas 42
3.4. Comparación cuando se aumenta el tamaño del problema 43
3.5. Rendimiento en el Modelo Maestro-Esclavo 49
3.6. Rendimiento en el Modelo de Islas 51

Índice de tablas

3.1. Comparación de la biblioteca con hibridación 44
3.2. Resumen de la paralelización Maestro-Esclavo 48

A.1. Pruebas secuenciales a la biblioteca (5) 60
A.2. Pruebas secuenciales a la biblioteca (10) 62

B.1. Pruebas secuenciales a la biblioteca hídrida (5) 64
B.2. Pruebas secuenciales a la biblioteca hídrida (10) 66

Índice de algoritmos

1. PSO Local - PSO para espacio de búsqueda continuo 14
2. PSO Global - PSO para espacio de búsqueda continuo 16
3. Modelo Maestro-Esclavo versión 1: Algortimo del *Master* 45
4. Modelo Maestro-Esclavo versión 2: Algortimo del *Master* 46
5. Modelo Maestro-Esclavo versión 1: Algortimo del *Slave* 47
6. Modelo Maestro-Esclavo versión 2: Algortimo del *Slave* 47

Introducción

Un problema de optimización consiste en encontrar una o varias soluciones, de un conjunto de soluciones admisibles, que minimiza o maximiza el rendimiento de un determinado proceso. Ejemplos de problemas de optimización se pueden encontrar en diferentes áreas de las ciencias y la ingeniería: encontrar una distribución de recursos que permita ejecutar cierta tarea en el menor tiempo posible, encontrar el camino más corto que pase por un conjunto dado de posiciones diferentes, encontrar las menores dimensiones de una viga que da soporte a cierta edificación, etcétera.

Existen problemas de optimización cuya complejidad no es significativa, pues la función a optimizar puede ser diferenciable o pueden estimarse, sin pérdida notable de precisión, sus derivadas parciales [1], y entonces pueden emplearse métodos directos (como los métodos de Newton o quasi-Newton). Pero por otro lado existen problemas de optimización mucho más complejos, bien sea porque la función no es diferenciable, o porque el problema es combinatorio (conduciendo a problemas NP-duros y NP-completos). Es en estos últimos casos cuando se emplean técnicas de optimización basadas en metaheurísticas [2].

La Optimización por Nube de Partículas o *Particle Swarm Optimization* (PSO) [3, 4] está basada en **técnicas metaheurísticas**. Esta técnica es muy reciente y novedosa, sus creadores fueron James Kennedy y Russ C. Eberhart. En 1995, desarrollaron el algoritmo experimentando con algoritmos que modelaban el comportamiento del vuelo de algunos pájaros o en el movimiento de los bancos de peces. El algoritmo se basa en la siguiente metáfora social: los individuos que son parte de una sociedad tienen una opinión influenciada por los criterios compartidos por el resto de los individuos. Cada individuo puede modificar su opinión (o estado) según tres factores: el conocimiento del entorno, los estados por los que ha pasado y los estados por los que han pasado los individuos cercanos.

La PSO ha sido aplicada a gran cantidad de problemas. La primera aplicación de esta técnica fue en el entrenamiento de redes neuronales [3] y los autores fueron sus propios creadores: Kennedy y Eberhart. Además ha sido utilizada para el diseño de circuitos lógicos combinatorios [5, 6], en el proceso de encontrar un orden óptimo de los genes de un *microarray*, denominado *Gene Ordering in Microarray Data(GOMAD)* [7] y en el campo de las telecomunicaciones, específicamente el seguimiento de la posición de los usuarios o

gestión del área de localización(*Location Area Management*) [8, 9]. Otros ejemplos donde se ha aplicado son: optimización de funciones numéricas [10], registrado de imágenes [11], aprendizaje de sistemas difusos [12] e ingeniería química [13]. Como se evidencia, esta técnica es muy utilizada y se está tratando de aplicar a muchos problemas de optimización. La mayoría de los problemas que han utilizado la PSO, obtienen buenos resultados y en tiempo razonable pero siguen siendo computacionalmente costosos.

En la PSO las partículas, peces o pájaros denotan puntos pertenecientes al dominio de la función a optimizar función objetivo. En cada iteración se trabaja con todos estos puntos a la vez, los cuales van cambiando de posición en el espacio a medida que el "enjambre" o la "nube" va adaptándose mejor al medio (o, en términos matemáticos, que con al menos uno de los puntos se va logrando un descenso en el valor de la función objetivo). Es por ello que en cada iteración ha de evaluarse la función objetivo en cada uno de esos puntos. De aquí que la complejidad de la técnica PSO es el producto de la complejidad computacional de la función objetivo, el tamaño de la población de puntos y la velocidad de convergencia (número de iteraciones). Si cualquiera de estos tres elementos aumenta, entonces la complejidad computacional de la PSO es significativamente mayor. Un ejemplo de una prueba realizada al problema *GOMAD* demuestra cuán costoso puede ser esta optimización: demoró casi 6 horas para que se pudiera obtener el resultado final [14].

La computación paralela ha devenido a ser una de las novedosas técnicas para afrontar la solución de problemas de alta complejidad computacional. Su popularidad se ha visto incrementada sobre todo por la proliferación de nuevos procesadores con arquitectura paralela (los llamados procesadores multinúcleos) en las computadoras personales. Es por ello que una de las últimas tendencias en las soluciones computacionales, es que éstas sean construidas siguiendo algoritmos paralelos que aprovechen en el mayor porciento posible las nuevas arquitecturas. A su vez, la capacidad de cálculo no sólo se puede obtener mediante un procesador multinúcleo o un supercomputador de varios procesadores, sino también conectando varias computadoras en una red y hacerlas funcionar como si fueran un supercomputador paralelo. Esta última alternativa es la que requiere menores costos de inversión y mantenimiento, por lo cual es ampliamente usada en universidades y centros de investigación.

Por su parte, la PSO es una técnica que puede ser paralelizada sin muchas dificultades, pues no existe ninguna dependencia entre las evaluaciones de la función objetivo en cada punto, y a su vez, la actualización de cada punto no depende de absolutamente todos los demás.

Dada la situación el **problema científico** a resolver es: ¿Cómo mejorar el rendimiento del algoritmo de optimización por Nube de Partículas cuando se usa en la solución de problemas computacionalmente costosos? El **objeto de estudio** es: el algoritmo de optimización por Nube de Partículas y el **campo de acción** es: el algoritmo de optimización por Nube de Partículas en entornos paralelos.

En tal sentido se propone como **objetivo general**, desarrollar una biblioteca paralela eficiente, portable y de propósito general de optimización por Nube de Partículas. Para dar cumplimiento a este objetivo se plantearon los siguientes **objetivos específicos**:

- Diseñar la biblioteca para el algoritmo de optimización por Nube de Partículas.
- Implementar la biblioteca usando modelos de paralelización conocidos.
- Realizar pruebas de rendimiento a la biblioteca.
- Aplicar la biblioteca para resolver el Problema Inverso Aditivo de Valores Singulares o *Inverse Additive Singular Value Problem* (PIAVS).

Para poder dar cumplimiento al objetivo propuesto se plantea la realización de las siguientes **tareas**:

1. Estudio del algoritmo de optimización por Nube de Partículas.
2. Diseño de la biblioteca para el algoritmo de optimización por Nube de Partículas.
3. Implementación de la versión secuencial de la biblioteca.
4. Estudio de los esquemas paralelos para el algoritmo de optimización por Nube de Partículas.
5. Implementación de la versión paralela de la biblioteca.
6. Realización de pruebas de eficiencia de la biblioteca paralela.
7. Realización de pruebas de la aplicación de la biblioteca en el PIAVS. Comparación con otras soluciones.

El documento está dividido en tres capítulos:

- El **Capítulo 1**: Revisión bibliográfica. En este capítulo se presenta una reseña bibliográfica donde se hace un análisis de la literatura y una presentación de la información recopilada que está estrechamente relacionada con el tema tratado.
- El **Capítulo 2**: Herramientas y metodologías. En este capítulo se plantean las herramientas y las metodologías utilizadas en la elaboración del presente trabajo de diploma.
- El **Capítulo 3**: Resultados y discusión. Se plantea como sería implementada la biblioteca de optimización por Nube de Partículas de forma secuencial y paralela. Se aborda la aplicación de la biblioteca, tanto secuencial como paralela, en el PIAVS.

Capítulo 1

Revisión bibliográfica.

En este capítulo se introducen los conceptos de optimización, se fundamentan las técnicas metaheurísticas y sus características. Se describe la PSO, problemas en los cuales han sido aplicados, las diferentes topologías, versiones y pseudocódigos. Además se abordan conceptos de computación paralela y los algoritmos de optimización por Nube de Partículas en entornos paralelos.

1.1. Conceptos básicos de optimización.

Los problemas que son resueltos generalmente por técnicas de optimización pueden ser clasificados en dos grupos: optimización sin restricciones y optimización con restricciones [15, 1]. Los problemas de optimización sin restricciones se presentan en aplicaciones prácticas en donde si las variables tienen algún tipo de restricción se considera que ésta no tenga efecto sobre la solución encontrada. El objetivo es mejorar (minimizar o maximizar) el valor real de una función f, por tanto se puede plantear que para encontrar el **mínimo global** será necesario encontrar un punto x^* tal que:

$$f(x^*) \leq f(x), \forall\, x \in S \tag{1.1}$$

donde S es el espacio de búsqueda y es muy común seleccionar $S = \mathbb{R}^n$ donde n es la dimensión de x. Cada elemento de S es considerado una solución candidata en ese espacio.

Los problemas de optimización con restricciones tienen restricciones explícitas sobre las variables. El problema puede tener varias condiciones, ejemplo de éstas pueden ser el establecimiento de los límites de los valores que pueden tomar una variable o involucrar desigualdades no lineales. Estos problemas de optimización minimizan una función f sobre un conjunto $U \subset \mathbb{R}^n$, $x^* \in U$ tal que:

$$f(x^*) \leq f(x),\ \forall\, x \in U \tag{1.2}$$

Para la optimización de problemas y cálculos de alta complejidad se han desarrollado múltiples técnicas y métodos. Estas técnicas de optimización se pueden clasificar en **exactas** y **aproximadas**. Las **técnicas**

exactas (enumerativas, exhaustivas, etcétera) garantizan encontrar la solución óptima de cualquier problema. Serían los métodos idóneos si no tuvieran el inconveniente de la cantidad de tiempo necesario para la resolución, el tiempo crece exponencialmente con el tamaño del problema. En determinados casos, el tiempo de resolución podría llegar a ser de varios días, meses o incluso años, lo que provoca que el problema sea inabordable con estos métodos. Las **técnicas aproximadas** sacrifican la garantía de encontrar el resultado óptimo a cambio de obtener una solución buena en un tiempo razonable. Se han venido desarrollando durante los últimos 30 años y se distinguen tres tipos:

- Los métodos constructivos: también conocidos como métodos voraces, suelen ser los más rápidos, partiendo de una solución vacía, a la cual se le va añadiendo componentes que generan una solución completa, que es el resultado del algoritmo. Generalmente las soluciones ofrecidas suelen ser de muy baja calidad ya que su planteamiento depende en gran parte del problema. Es muy difícil encontrar métodos de esta clase que produzcan buenas soluciones, y en algunas ocasiones es casi imposible, ya que por ejemplo, en los problemas con muchas restricciones puede que la mayoría de las soluciones parciales conduzcan a soluciones no factibles.

- Los métodos de búsqueda local: en otras bibliografías son conocidos también como métodos de seguimiento del gradiente. Estos usan el concepto de vecindario y se inician con una solución completa recorriendo parte del espacio de búsqueda hasta encontrar un óptimo local. El vecindario de una solución es el conjunto de soluciones que se pueden construir a partir de aquella aplicando un operador de modificación denominado movimiento. En función del operador de movimiento utilizado, el vecindario cambia y el modo de explorar el espacio de búsqueda también, pudiendo la búsqueda complicarse o simplificarse.

- Las técnicas metaheurísticas: son algoritmos no exactos, se fundamentan en la combinación de diferentes métodos heurísticos a un nivel más alto para conseguir una exploración del espacio de búsqueda más eficaz y eficiente.

1.2. Técnicas metaheurísticas.

El término **heurística** proviene de la palabra griega *heuriskein* que significa "hallar, inventar". El término metaheurística se obtiene de anteponer a **heurística** el prefijo **meta** que significa "más allá" o "a un nivel superior", por tanto las metaheurísticas son estrategias inteligentes para diseñar o mejorar procedimientos heurísticos para buscar una solución óptima o casi óptima. Este término fue introducido por primera vez por Fred Globes en 1986 [16]. Un algoritmo metaheurístico es un método heurístico para resolver un tipo de problema computacional, usando los parámetros dados por el usuario sobre unos procedimientos genéricos y abstractos. Estos algoritmos proporcionan buenas soluciones y no necesariamente la óptima, en un tiempo relativamente corto y con recursos razonables [2, 17].

Los tipos de metaheurísticas se establecen en función del tipo de procedimientos a los que se refiere [18]. Se dividen principalmente en cuatro grupos:

- Las metaheurísticas de relajación: se refieren a procedimientos de resolución de problemas que utilizan relajaciones del modelo original (es decir, modificaciones del modelo que hacen al problema más fácil de resolver), cuya solución facilita la solución del problema original.
- Las metaheurísticas constructivas: se orientan a los procedimientos que tratan de la obtención de una solución a partir del análisis y selección paulatina de las componentes que la forman.
- Las metaheurísticas de búsqueda: guían los procedimientos que usan transformaciones o movimientos para recorrer el espacio de soluciones alternativas y explotar las estructuras de entornos asociadas.
- Las metaheurísticas evolutivas: están enfocadas a los procedimientos basados en conjuntos de soluciones que evolucionan sobre el espacio de soluciones.

Existen muchas formas diferentes de clasificar y describir las técnicas metaheurísticas [19]. Dependiendo de las características que se seleccionen se pueden obtener diferentes taxonomías: basadas en la naturaleza o no basadas en la naturaleza, basadas en memoria o sin memoria, con función objetivo estática o dinámica, etcétera. La clasificación de las técnicas metaheurísticas más empleada es la que se basa en si la técnica utiliza un único punto del espacio de búsqueda o trabaja sobre una población o conjunto. Esta taxonomía se muestra en la Figura 1.1. De acuerdo a esta clasificación las metaheurísticas se dividen en dos grupos:

- Metaheurísticas basadas en trayectoria: en estas técnicas se parte de un punto inicial y se actualiza la solución explorando el vecindario, así se va formando una trayectoria. La búsqueda finaliza cuando se alcanza un número de máximo de iteraciones, se encuentra una solución con una calidad aceptable o se detecta un estancamiento del proceso. Algunas de las técnicas basadas en esta clasificación son:
    - Enfriamiento Simulado o *Simulated Annealing* (SA): el nombre e inspiración viene del proceso de **recocido** del metal, una técnica que consiste en calentar y luego enfriar de forma controlada un material para aumentar el tamaño de sus cristales y reducir sus defectos. [20, 21, 22]
    - Búsqueda Tabú o *Tabu Search* (TS): es una técnica para resolver problemas combinatorios de gran dificultad. La idea básica de la búsqueda tabú es el uso explícito de un historial de la búsqueda, es decir, una memoria de corto plazo. [16, 23]
    - Búsqueda en Vecindario Variable o *Variable Neighborhood Search* (VNS): aplica explícitamente una estrategia para cambiar entre diferentes estructuras de vecindario de entre un conjunto de ellas definidas al inicio del algoritmo. [24, 25]

    - Procedimiento de Búsqueda Miope Aleatorizado y Adaptativo o *Greedy Randomized Adaptive Search Procedure* (GRASP): es una metaheurística simple que combina heurísticos constructivos con búsqueda local. Es un procedimiento iterativo compuesto de dos fases: primero una construcción de una solución y después un proceso de mejora. [26, 18].

- Metaheurísticas basadas en población: en estas técnicas se tiene un grupo de individuos los cuales representan las soluciones, en vez de utilizar una única solución como ocurre en el caso de las metaheurísticas basadas en trayectoria. La eficiencia y resultado depende de cómo se manipula la población en cada iteración. Ejemplos de técnicas que utilizan esta clasificación son:

    - Algoritmos evolutivos o *Evolutionary Algorithms* (EA): están inspirados en los principios de la genética y la selección natural. Cada individuo representa una solución potencial al problema que se está resolviendo. La modificación de la población se lleva a cabo mediante tres operadores: selección, recombinación y mutación [27, 28, 26].
    - Búsqueda Dispersa o *Scatter Search* (SS): es un procedimiento basado en estrategias para combinar reglas de decisión, así como en la combinación de restricciones. El método SS opera sobre un conjunto de soluciones, llamado conjunto de referencia, combinando éstas para crear nuevas soluciones de modo que mejoren a las que las originaron. [2, 29]
    - Sistemas Basados en Colonias de Hormigas o *Ant Colony Optimization* (ACO): están inspiradas en el comportamiento de las hormigas cuando realizan la búsqueda de comida: inicialmente, las hormigas exploran el área cercana a su nido de forma aleatoria. Tan pronto como una hormiga encuentra la comida, la lleva al nido y mientras que realiza este camino, va depositando una sustancia química denominada **feromona**. Esta comunicación indirecta entre las hormigas mediante el rastro de **feromona** las capacita para encontrar el camino más corto entre el nido y la comida. [30, 31]
    - PSO: técnica inspirada en el comportamiento social del vuelo de las bandadas de aves o el movimiento de los bancos de peces[1]. Se fundamenta en los factores que influyen en la toma de decisión de un agente que forma parte de un conjunto de agentes similares. La toma de decisión por parte de cada agente se realiza conforme a una componente social y una componente individual, mediante las que se determina el movimiento (dirección) de este agente para alcanzar una nueva posición en el espacio de soluciones. Simulando este modelo de comportamiento se obtiene un método para resolver problemas de optimización. [4, 3, 32]

[1]En la literatura lo podemos encontrar como población, cúmulo, enjambre o colmena de partículas. En este trabajo se usará el término "nube".

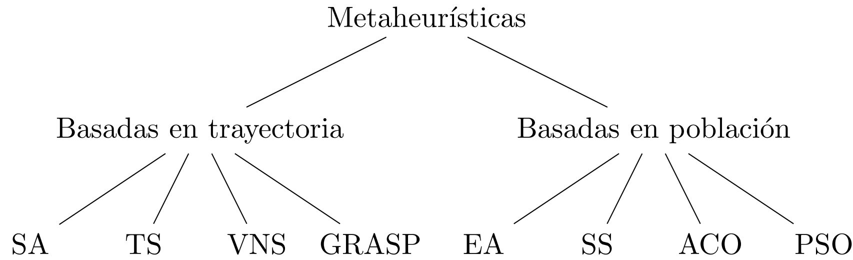


Figura 1.1: Clasificación de las metaheurísticas.

1.3. Optimización por Nube de Partículas.

La optimización por Nube de Partículas es una técnica metaheurística bastante reciente, fue propuesta por el sociólogo James Kennedy y el ingeniero Russ C. Eberhart en 1995 [3]. Este algoritmo está inspirado en el comportamiento social de los individuos dentro de los enjambres, ejemplo de ello son el vuelo de las bandadas de aves y los bancos de peces. Cada individuo evolucionará su comportamiento para asemejarse a los individuos con más éxito dentro de su entorno. En la simulación cada individuo está afectado por un factor individual y uno social.

La PSO es un sistema multiagente donde cada partícula es un agente y cada uno se guía por tres principios fundamentales: evaluar, comparar e imitar. Desde el punto computacional significa moverse por el espacio de búsqueda y encontrar una solución, compararla con la propia y con la de los demás agentes, en el caso de que la solución sea la mejor del todo el sistema comunicarla a los demás, en caso contrario imitar al mejor agente.

Las principales características de la PSO son las siguientes:

- Los agentes modifican su posición de acuerdo a las posiciones de los agentes del vecindario.
- Cada agente almacena su "experiencia" propia, y determina su nueva dirección en función de la mejor posición por la que haya pasado anteriormente.
- Por lo general converge rápido a buenas soluciones.
- La población de agentes se inicia de forma aleatoria y va evolucionando en cada iteración.
- El algoritmo tiene operadores de movimiento pero no mutación o cruzamiento como ocurre en los algoritmos genéticos.
- Durante la ejecución nunca muere o nace un nuevo agente.

1.3.1. Descripción de la optimización por nube de partículas.

Un algoritmo de PSO consiste en un proceso iterativo y **estocástico** que opera sobre una nube de partículas. Generalmente una partícula p_i está compuesta por:

1. Tres vectores
   - El vector $x_i = (x_{i,1}, x_{i,2}, x_{i,3}, ..., x_{i,N})$ almacena la posición actual de la partícula en el espacio de búsqueda.
   - El vector velocidad $v_i = (v_{i,1}, v_{i,2}, v_{i,3}, ..., v_{i,N})$ almacena el dirección según el cual se moverá la partícula.
   - El vector $pBest_i = (pBest_{i,1}, pBest_{i,2}, pBest_{i,3}, ..., pBest_{i,N})$ almacena la posición de la mejor solución encontrada por la partícula hasta el momento.
2. Dos valores de aptitud
   - El valor de aptitud $fitness_i$ almacena la adecuación, es decir, el valor de la función objetivo en el punto x_i.
   - El valor de aptitud $fitness_pBest_i$ almacena el valor de la adecuación de la mejor solución local encontrada hasta el momento (vector $pBest_i$).

La nube se inicializa generando las posiciones de las partículas de forma aleatoria, regular o combinando ambos. Además se generan las velocidades aleatoriamente en un intervalo establecido $[-V_{max}, V_{max}]$, donde V_{max} será la velocidad máxima que se pueda tomar una partícula en cada movimiento. No es conveniente fijarlas en cero pues no se obtienen buenos resultados [4]. Una vez generadas las posiciones, se calcula la aptitud de cada una y se actualizan los valores de $fitness_i$ y $fitness_pBest_i$.

Inicializando la nube, las partículas deben moverse dentro del proceso iterativo. Una partícula se mueve desde una posición del espacio de búsqueda hasta otra, adicionando al vector posición x_i^k el vector v_i^{k+1} para obtener un nuevo vector posición:

$$x_i^{k+1} = x_i^k + v_i^{k+1} \tag{1.3}$$

Una vez calculada la nueva posición de la partícula, se evalúa actualizando el $fitness_i$. Si la nueva aptitud es la mejor encontrada hasta el momento, se actualizan los valores de mejor posición $pBest_i$ y aptitud $fitness_pBest_i$. El vector velocidad de cada partícula es modificado en cada iteración utilizando la velocidad anterior, el componente **cognitivo** y el componente **social**. El modelo matemático resultante viene representado por las siguientes ecuaciones:

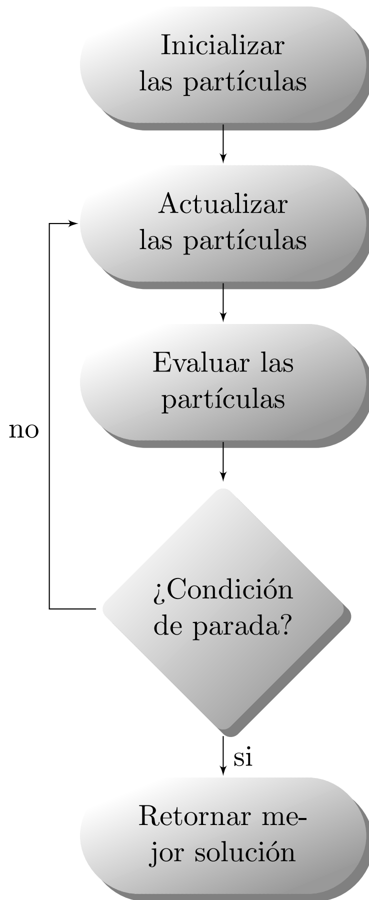
$$v_i^{k+1} = v_i^k + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i^k) + \varphi_2 \cdot rand_2 \cdot (gBest - x_i^k) \tag{1.4}$$

donde

- $x_i^k \equiv$ vector posición de la partícula i en la iteración k.
- $v_i^k \equiv$ vector velocidad de la partícula i en la iteración k.
- $\varphi_1 \equiv$ peso que controla el componente cognitivo.
- $\varphi_2 \equiv$ peso que controla el componente social.
- $rand_1 \equiv$ número aleatorio en el intervalo [0, 1].
- $rand_2 \equiv$ número aleatorio en el intervalo [0, 1].
- $pBest_i \equiv$ mejor posición encontrada por la partícula i hasta el momento que posee la mejor solución.
- $gBest \equiv$ representa la posición de la partícula con la mejor solución o aptitud.

En la ecuación (1.4) se actualiza el vector velocidad de cada partícula i en la iteración k. El componente **cognitivo** en la ecuación es el factor $\varphi_1 \cdot rand_1 \cdot (pBest_i - x_i^k)$ e indica la decisión que tomará la partícula en dependencia de su propia experiencia. El componente **social** en la ecuación es el factor $\varphi_2 \cdot rand_2 \cdot (gBest - x_i^k)$ y representa la decisión que tomará la partícula según la influencia que el resto de la nube ejerce sobre ella.

El siguiente pseudocódigo describe el algoritmo clásico de PSO:



$t = 0$
Nube ←Inicializar Nube de Partículas
mientras no se alcance condición de parada **hacer**
  $t = t + 1$
  para $i = 0$ **hasta** *tamaño(Nube)* **hacer**
    Evaluar cada partícula x_i de la *Nube*
    si $fitness_i$ es mejor que $fitness_pBest_i$ **entonces**
      $pBest_i \leftarrow x_i$
      $fitness_pBest_i \leftarrow fitness_i$
    fin si
    si $fitness_pBest_i$ es mejor que $fitness_gBest$ **entonces**
      $gBest \leftarrow pBest_i$
      $fitness_gBest \leftarrow fitness_pBest_i$
    fin si
  fin para
  para $i = 0$ **hasta** *tamaño(Nube)* **hacer**
    Calcular la velocidad v_i de x_i en base a x_i, $pBest_i$ y $gBest$
    Calcular la nueva posición de x_i, de su valor actual y v_i
  fin para
fin mientras
Devuelve la mejor solución encontrada

Figura 1.2: Secuencia de un algoritmo de PSO.

Existen muchas formas para calcular la velocidad de las partículas, pero existen dos que son las más utilizadas. En el año 1998 [33], Shi y Eberhart propusieron añadirle a la ecuación (1.4) un peso a la inercia (ω) conocido como **factor de inercia**, con este se garantiza **controlar** el vector velocidad v_i sin necesidad de usar V_{max}. Además este factor es usado para el balance entre la exploración y la explotación en el espacio de búsqueda. Si $\omega \geq 1$ y se va incrementando durante la ejecución la nube va a divergir, mientras que si $0 < \omega < 1$ y se va disminuyendo la nube va a converger. Por esto es que los autores proponen ir disminuyendo el **factor de inercia** durante la ejecución, para así lograr que pase de la fase de exploración a explotación. En pruebas realizadas, los valores $0{,}4 < \omega < 0{,}9$ han obtenido buenos resultados [34]. La ecuación modificada quedaría:

$$v_i^{k+1} = \omega \cdot v_i^k + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i^k) + \varphi_2 \cdot rand_2 \cdot (gBest - x_i^k) \tag{1.5}$$

La otra variante de acuerdo al cálculo de velocidad es el **factor de estrechamiento** que fue propuesta por Kennedy y Clerc [35]. No utiliza el valor de inercia, como en la versión anterior de PSO, donde el **factor**

de inercia es utilizado para controlar la velocidad de la partícula y reducir la influencia de la dirección de búsqueda previamente recorrida por la partícula. La variante con el **factor de estrechamiento** representado por K (*constriction factor*) no multiplica la velocidad actual de la partícula sino que afecta a la velocidad actualizada calculada. La fórmula de vuelo modificada es la siguiente:

$$v_i^{k+1} = K \cdot [v_i^k + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i^k) + \varphi_2 \cdot rand_2 \cdot (gBest - x_i^k)] \tag{1.6}$$

$$K = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4 \cdot \varphi}|} \quad donde\ \varphi = \varphi_1 + \varphi_2, \varphi > 4 \tag{1.7}$$

En la Figura 1.3 se muestra el movimiento de una partícula en el espacio de solución. Las flechas de líneas discontinuas negras representan la dirección de los componentes **cognitivo** y **social** respectivamente. La fecha discontinua roja representa el vector velocidad de la partícula y la flecha azul es la dirección que toma la partícula para moverse hacia su próxima posición, es decir, desde x^k hasta x^{k+1}.

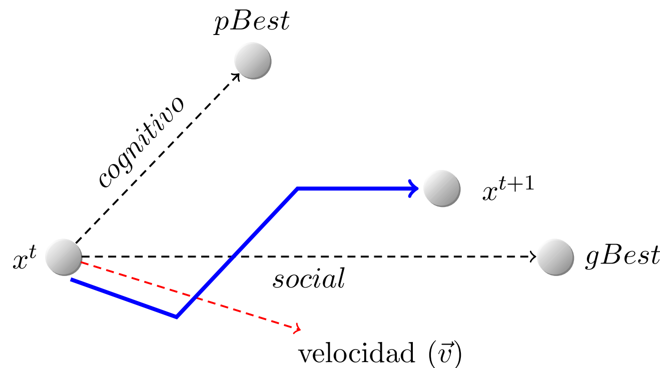


Figura 1.3: Movimiento de una partícula en el espacio de soluciones.

Modelos de optimización por Nube de Partículas.

La PSO se divide en diferentes tipos teniendo en cuenta diversos **factores de configuración**. Según la importancia de los pesos **cognitivo** y **social** se dividen en:

- **Modelo Completo**: $\varphi_1, \varphi_2 > 0$. Los dos componentes intervienen en el movimiento de la partícula.
- **Modelo sólo Cognitivo**: $\varphi_1 > 0$ y $\varphi_2 = 0$. El componente *cognitivo* es el único que interviene en el movimiento de la partícula.
- **Modelo sólo Social**: $\varphi_1 = 0$ y $\varphi_2 > 0$. El componente *social* es el único que interviene en el movimiento de la partícula.

- **Modelo sólo Social exclusivo**: $\varphi_1 = 0, \varphi_2 > 0$ y $gBest \neq x_i$. La posición de la partícula en sí no puede ser la mejor de su entorno.

1.3.2. Topologías de PSO.

El desarrollo de una partícula individual depende tanto de la versión del algoritmo como de la topología de la nube. Las topologías definen el entorno de interacción de una partícula individual con su vecindario. Los entornos pueden ser de dos tipos:

- Geográficos: se calcula la distancia de la partícula actual al resto y se toman las más cercanas para componer su entorno.

- Sociales: se define a priori una lista de vecinas para la partícula, independientemente de su posición en el espacio. Son los más empleados.

En la Figura 1.4 se muestran ejemplos de entornos geográficos y sociales. Cuando el tamaño del entorno es toda la nube de partículas, el entorno es la vez geográfico y social, obteniendo un PSO Global.

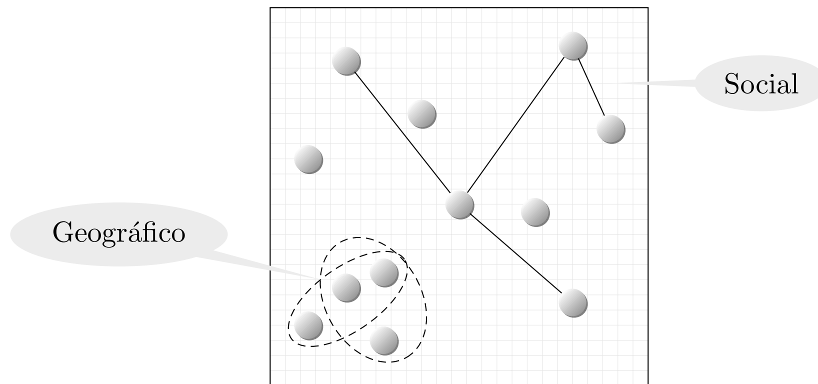


Figura 1.4: Entorno geográfico y social en el espacio de soluciones.

En una nube de partículas, cada uno de los individuos puede estar conectado a algún otro mediante una gran variedad de topologías. Sin embargo, la mayoría de las implementaciones utilizan alguna de las dos principales llamadas *Global best (GBest)* y *Local best (LBest)*. El modelo *GBest* conecta a todos los individuos de la nube, que tiene como efecto que cada partícula estará influenciada por la partícula que haya tenido la mejor **adaptación**. Este modelo converge más rápido hacia la solución porque la visibilidad de cada partícula es mejor y se acercan más a la mejor de la nube favoreciendo la intensificación, por esta razón, también cae más fácilmente en óptimos locales [36].

El modelo *LBest* crea un vecindario para cada partícula, que comprende a él mismo y a sus k vecinos, la partícula estará influenciada por la de mejor **adaptación** de una cierta porción de la nube. Esta topología ofrece la ventaja de establecer varias subnubes que realizan búsquedas en diversas regiones del espacio. Por sus características le cuesta más converger favoreciendo en este caso la diversificación, por lo tanto no cae fácilmente en óptimos locales [37]. El siguiente algoritmo muestra el pseudocódigo de la versión PSO Local.

Algoritmo 1 PSO Local - PSO para espacio de búsqueda continuo

1: $S \leftarrow InicializarNube()$
2: **mientras** no se alcance condición de parada **hacer**
3:   **para** $i = 1$ **hasta** $size(S)$ **hacer**
4:     Evaluar cada partícula x_i de S
5:     **si** $fitness_i$ es mejor que $fitness_pBest_i$ **entonces**
6:       $pBest_i \leftarrow x_i$
7:       $fitness_pBest_i \leftarrow fitness_i$
8:     **fin si**
9:   **fin para**
10:   **para** $i = 1$ **hasta** $size(S)$ **hacer**
11:     Escoger $lBest_i$, la partícula con mejor fitness del entorno de x_i
12:     $v_i \leftarrow v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (lBest_i - x_i)$
13:     $x_i \leftarrow x_i + v_i$
14:   **fin para**
15: **fin mientras**
16: **Salida**: la mejor solución encontrada

Otra variante, propuesta por Kennedy y Mendes [38] es la denominada **Von Neumann o Cuadrada** donde plantean que se pueden obtener mejores resultados que en la variante *GBest*. Se ha comprobado que no se puede afirmar que existe una topología óptima para cualquier problema, las pruebas realizadas han demostrado que en varios problemas algunas variantes han sido superiores al resto pero para otras situaciones han sido deficientes. En nuestro trabajo haremos las pruebas con las más usadas: *GBest* y *LBest*.

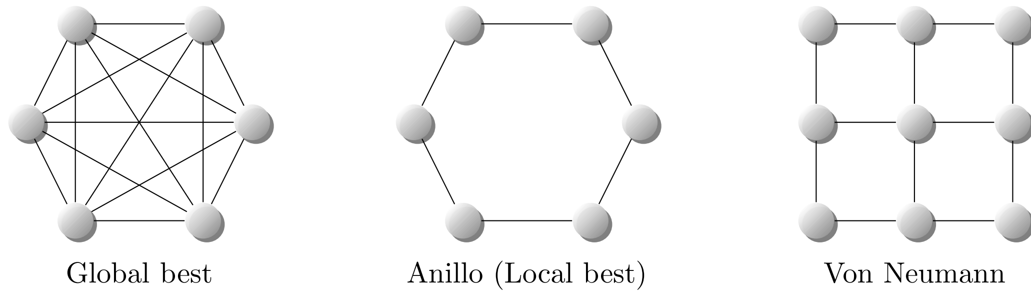


Figura 1.5: Topologías de la nube de partículas.

1.3.3. Variantes de PSO para diferentes espacios de búsqueda.

PSO para espacio de búsqueda continuo

Es la versión original y más utilizada, concebida en espacios de búsqueda con variables continuas. En esta versión del PSO, los individuos son vistos como puntos en el espacio donde el cambio a lo largo del tiempo es visto como un movimiento de puntos o partículas. En el siguiente algoritmo se muestra el pseudocódigo de la versión PSO Global.

Algoritmo 2 PSO Global - PSO para espacio de búsqueda continuo

1: $S \leftarrow InicializarNube()$
2: **mientras** no se alcance condición de parada **hacer**
3: **para** $i = 1$ **hasta** $size(S)$ **hacer**
4: Evaluar cada partícula x_i de S
5: **si** $fitness_i$ es mejor que $fitness_pBest_i$ **entonces**
6: $pBest_i \leftarrow x_i$
7: $fitness_pBest_i \leftarrow fitness_i$
8: **fin si**
9: **si** $fitness_pBest_i$ es mejor que $fitness_gBest$ **entonces**
10: $gBest \leftarrow pBest_i$
11: $fitness_gBest \leftarrow fitness_pBest_i$
12: **fin si**
13: **fin para**
14: **para** $i = 1$ **hasta** $size(S)$ **hacer**
15: $v_i \leftarrow v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (gBest - x_i)$
16: $x_i \leftarrow x_i + v_i$
17: **fin para**
18: **fin mientras**
19: **Salida**: la mejor solución encontrada

PSO para espacio de búsqueda binario

En este modelo cada individuo de la población sólo tiene en mente la decisión binaria que tiene que tomar: si/no, falso/verdadero, etcétera. Cada individuo está rodeado por otros que también tienen que tomar su propia decisión. La probabilidad de que un individuo seleccione el "si" depende de qué tan exitoso haya resultado el "si" con respecto al "no" en el pasado. El individuo debe tomar una serie de decisiones de manera que todas ellas sean adecuadas en conjunto, por lo que debe de ser capaz de evaluar, comparar e imitar una cierta cantidad de decisiones binarias simultáneamente.

Matemáticamente, Kennedy y Eberhart [32] proponen un modelo donde la probabilidad de que un individuo decida si/no, falso/verdadero o alguna otra decisión binaria es una función que depende de factores personales y sociales:

$$P[x_{i,d}^{t} = 1] = f[x_{i,d}^{t-1}, v_{i,d}^{t-1}, p_{i,d}, p_{g,d}]$$

donde:

- $P[x_{i,d}^t = 1]$ es la probabilidad que el individuo i seleccione 1 para el bit en la posición d de la cadena binaria.
- $x_{i,d}^t$ es el estado actual del bit d de la cadena binaria.
- $v_{i,d}^{t-1}$ es la medida de la predisposición individual o la probabilidad actual de seleccionar 1.
- $p_{i,d}$ es el mejor estado encontrado para el bit d de la cadena binaria de acuerdo a la experiencia personal del individuo.
- $p_{i,d}$ es el mejor estado encontrado para el bit d de la cadena binaria encontrado por el mejor individuo del vecindario.

Si $v_{i,d}^t$ es grande, el individuo tiene una mayor predisposición a seleccionar 1, mientras que valores pequeños favorecen al 0. Para ubicar el valor de este parámetro dentro del rango [0, 1], se utiliza la siguiente función:

$$sig(V_{i,d}) = \frac{1}{1 + exp(-V_{i,d})} \tag{1.8}$$

donde V_{id} es la velocidad de la partícula i en la dimensión d.

El valor ρ es un número aleatorio comprendido en el intervalo [0, 1] y viene dado por la siguiente expresión:

$$Si\ \rho_i < sig(V_{i,d})\ entonces\ x_{i,d} = 1;\ si\ no\ x_{i,d} = 0$$

Otras variantes de PSO

Existen muchas variantes de PSO, por tanto solo se hará referencias a algunas:

- *Hybrid of Genetic Algorithm and* PSO (GA-PSO): combina las ventajas de la **inteligencia** de los enjambres y un mecanismo de selección natural, tal es el caso de Algoritmos Genéticos, a fin de aumentar el número de agentes mejores adaptados, mientras que disminuye el número agentes peores adaptados en cada paso de iteración [39, 40].
- *Adaptive* PSO: otros autores han sugerido otros ajustes a la parámetros de configuración del algoritmo, por ejemplo, añadir un componente aleatorio a la inercia [33] y la aplicación de lógica difusa [41].
- *Multiobjective* PSO (MOPSO): los problemas de optimización multiobjetivos consisten en varios objetivos que deben ser alcanzados de manera simultánea. Una forma sencilla para abordar estos problemas consiste en agregar múltiples objetivos en una función teniendo en cuenta el peso, que puede ser fijo o puede cambiar dinámicamente durante el proceso de optimización [42]. El principal inconveniente de este

enfoque es que no siempre es posible encontrar la función con ponderación adecuada y por otra parte, a veces se desea considerar las compensaciones entre los múltiples objetivos.

- *Gaussian* PSO (GPSO): el algoritmo clásico PSO para realizar la exploración, así como la convergencia de la nube en la zona óptima, depende de los valores de los parámetros tales como la aceleración y la inercia. Con el fin de corregir estas deficiencias percibidas, algunos autores han introducido las funciones de Gauss para guiar los movimientos de las partículas [43, 44]. En este enfoque, la constante de inercia ya no es necesaria y la aceleración es remplazada por números aleatorios con distribución gaussiana.
- *Stretching* PSO (SPSO): el problema principal en muchas técnicas de optimización es el problema de la convergencia en la presencia de mínimos locales. En estas condiciones, la solución puede caer en los mínimos locales, cuando comienza la búsqueda, y puede estancarse en sí. Parsopoulos y Vrahatis [45] presentaron este algoritmo modificado de PSO que está orientado para resolver el problema de encontrar todos los mínimos globales.

1.4. Introducción a la computación paralela.

A lo largo de la historia de la Informática ha sido una necesidad incrementar la potencia de cálculo disponible. En la actualidad existen muchos problemas complejos que su resolución tiene un costo computacional muy alto, manejando grandes volúmenes de datos. A pesar de la vertiginosa evolución de las arquitecturas hardware, el desarrollo de software con mayores necesidades de recursos siempre es mucho más rápido [46]. Por esta necesidad se plantea proponer nuevos modelos capaces de adaptarse a las necesidades de los científicos e ingenieros de una manera más flexible y económica. Frente a la imposibilidad de construir procesadores con capacidad suficiente para llevar a cabo estas tareas con un costo y en un tiempo razonable, la solución más idónea es la introducción de paralelismo.

La computación paralela es el nombre que se le da al procesamiento de datos de manera simultánea y es considerada una técnica de programación [47]. El procesamiento paralelo funciona bajo el principio de división del trabajo en subtareas y asignación de éstas a distintos elementos computacionales trabajadores, con el fin de conseguir un menor tiempo de ejecución. Éstos deben comunicarse entre si cuando es necesario, para llevar a cabo ciertas subtareas.

1.4.1. Arquitecturas paralelas.

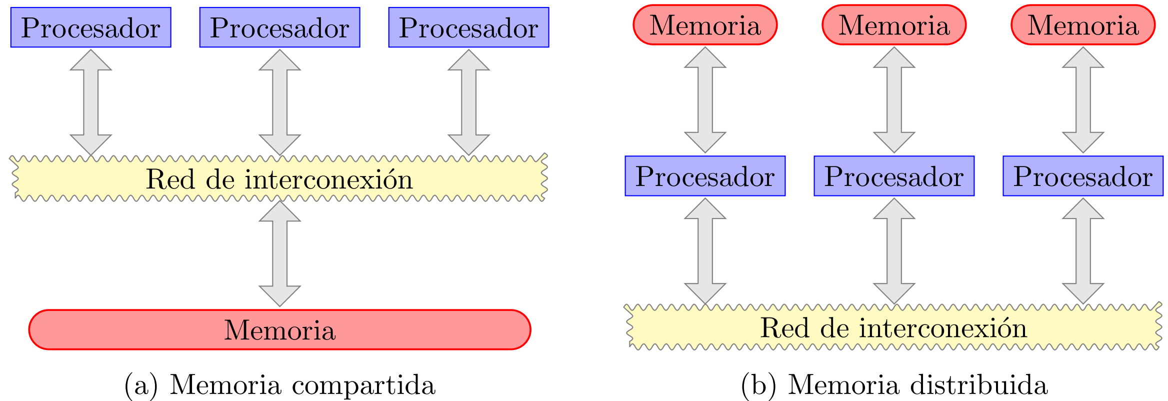
Existen varias maneras de clasificar el procesamiento paralelo. Puede considerarse a partir de la organización interna de los procesadores, desde la estructura de interconexión entre los procesadores o desde del flujo de información a través del sistema [48]. La secuencia de instrucciones leídas de la memoria constituye un flujo de instrucciones y las operaciones ejecutadas sobre los datos en el procesador constituyen un flujo de datos.

El procesamiento paralelo puede ocurrir en el flujo de instrucciones, en el flujo de datos o en ambos. Una clasificación presentada por Michael. J. Flynn considera la organización de un sistema de computadora mediante la cantidad de instrucciones y unidades de datos que se manipulan en forma simultánea. La clasificación de Flynn divide a las computadoras en cuatro grupo principales [49]:

- *Simple Instruction Simple Data (SISD)*: Modelo secuencial tradicional. Computador secuencial que no explota el paralelismo en las instrucciones ni en flujos de datos.
- *Multiple Instructions Simple Data (MISD)*: Poco común debido al hecho de que la efectividad de los múltiples flujos de instrucciones suele precisar de múltiples flujos de datos. Se han propuesto algunas arquitecturas teóricas que hacen uso de *MISD*, pero ninguna llegó a producirse en masa. Su desarrollo se limita al campo de la investigación.
- *Simple Instruction Multiple Data (SIMD)*: Un computador que explota varios flujos de datos dentro de un único flujo de instrucciones para realizar operaciones que pueden ser paralelizadas de manera natural.
- *Multiple Instructions Multiple Data (MIMD)*: Varios procesadores autónomos que ejecutan simultáneamente instrucciones diferentes sobre datos diferentes. Los sistemas distribuidos suelen clasificarse como arquitecturas *MIMD*; bien sea explotando un único espacio compartido de memoria, o uno distribuido. Este es el modelo que implementan la mayoría de los computadores paralelos.

Hay dos formas principales de intercambio de datos entre las tareas paralelas:

- Arquitectura de memoria compartida: Todos los procesadores del sistema pueden acceder directamente a todas las posiciones de memoria, es decir, el espacio de memoria física es único y global. Las restricciones de acceso a la memoria suelen ser únicamente de tipo físico. Una red de interconexión une los distintos elementos de proceso con los módulos de memoria, compartida por todos ellos. La comunicación entre los distintos elementos de proceso se hace utilizando la memoria. Para ello basta con que un determinado procesador escriba una variable en memoria y ésta sea leída por otro procesador.
- Arquitectura de memoria distribuida (paso de mensajes): Los procesadores del sistema pueden acceder directamente sólo a las posiciones de memoria locales, es decir, cada procesador tiene su propio espacio de memoria física privado. Una red de interconexión une los distintos elementos de proceso entre sí. La comunicación entre los procesadores se realiza mediante paso de mensajes.



(a) Memoria compartida

(b) Memoria distribuida

Figura 1.6: Principales formas de intercambio de datos.

Cluster

Un *cluster* es un multicomputador cuyos nodos son computadores de componentes comunes y relativamente baratos y que están conectados por una red de alta velocidad mediante un *switch*. El objetivo de un *cluster* es obtener una máquina de mayor potencia a partir de computadores no necesariamente muy potentes y a un precio razonable. Al usar componentes más o menos comunes y habituales en el mercado, el costo de un *cluster* en relación a la potencia de cálculo obtenida es bajo.

El modelo de programación de un *cluster* es necesariamente paso de mensajes entre nodos a través de la red. Para ello existe una capa de software intermedio, o *middleware* de comunicaciones, que facilita esta labor. Un *cluster* es tremendamente escalable y muy flexible, ya que se le pueden añadir o sustituir nodos con relativa facilidad. No es necesario que todos los nodos de un *cluster* tengan la misma arquitectura, ni siquiera es necesario que tengan el mismo sistema operativo. Esto es lo que se denomina un *cluster* heterogéneo, frente al concepto de *cluster* homogéneo, donde sí son iguales todos los nodos.

1.4.2. Evaluación de los algoritmos paralelos.

La evaluación de los algoritmos puede hacerse teórica y experimentalmente. Para ello es necesario contar con varias métricas de evaluación de prestaciones, que consideren el tamaño de la entrada al algoritmo n y el número de procesadores P. Las métricas más utilizadas son [48]:

Tiempo de ejecución.

Es un parámetro absoluto pues permite medir la rapidez del algoritmo sin compararlo con otro. En el caso de un programa secuencial, consiste en el tiempo transcurrido desde que se lanza su ejecución hasta que finaliza.

En el caso de un programa paralelo, el tiempo de ejecución es el tiempo que transcurre desde el comienzo de la ejecución del programa en el sistema paralelo hasta que el último procesador culmine su ejecución [48]. El tiempo aritmético se expresa en cantidad de FLOPS, donde el FLOPS (*floating-point operations per second*) es una medida que indica la velocidad que tarda el procesador en realizar una operación aritmética en punto flotante.

Para sistemas paralelos con memoria distribuida el tiempo paralelo con p procesadores, T_P, se determina de modo aproximado mediante la fórmula:

$$T_P \approx T_A + T_C - T_{SOL} \tag{1.9}$$

donde T_A es el tiempo aritmético, es decir, es el tiempo que tarda el sistema multiprocesador en hacer las operaciones aritméticas; T_C es el tiempo de comunicación, o sea, el tiempo que tarda el sistema multiprocesador en ejecutar transferencias de datos; y T_{SOL} es el tiempo de solapamiento, que es el tiempo que transcurre cuando las operaciones aritméticas y de comunicaciones se realizan simultáneamente. El tiempo de solapamiento suele ser muchas veces imposible de calcular, por lo cual se condiciona que se realice la aproximación:

$$T_P \approx T_A + T_C \tag{1.10}$$

Ganancia de velocidad *(Speed-Up)*.

El *Speed-Up* (S_P), para p procesadores, se determina mediante la fórmula:

$$S_P = \frac{T_S}{T_P} \tag{1.11}$$

donde T_S es el tiempo de ejecución de un programa secuencial y T_P es el tiempo de ejecución de la versión paralela de dicho programa en p procesadores. Esta métrica indica la ganancia de velocidad que se ha obtenido con la ejecución en paralelo, respecto al algoritmo secuencial. En el mejor de los casos el tiempo de ejecución de un programa en paralelo con p procesadores será p veces inferior al de su ejecución en un sólo procesador, teniendo todos los procesadores igual potencia de cálculo. Generalmente el tiempo nunca se verá reducido en un orden igual a p, ya que hay que contar con las sincronizaciones y dependencias entre los procesadores.

Eficiencia

La **eficiencia** significa el grado de aprovechamiento de los procesadores para la resolución del problema. El valor máximo que puede alcanzar es 1, que significa un 100 % de aprovechamiento.

$$E = \frac{S_P}{p} \tag{1.12}$$

Escalabilidad

La escalabilidad es la capacidad de un determinado algoritmo de mantener sus prestaciones cuando aumenta el número de procesadores y el tamaño del problema en la misma proporción. Ésta nos indica la capacidad del algoritmo de utilizar de forma efectiva un incremento en los recursos computacionales. La escalabilidad puede evaluarse mediante diferentes métricas [48]. Es conveniente tener en cuenta las características de los problemas con los que se está tratando para elegir la métrica adecuada de escalabilidad.

1.4.3. Modelos de algoritmos paralelos

Un modelo de algoritmo es normalmente una forma de estructurar un algoritmo paralelo mediante la selección de una descomposición, asignación y la aplicación de la estrategia adecuada para minimizar las interacciones [48, 50]. Un concepto a tener en cuenta es la granularidad, que es el tamaño de las piezas en que se divide una aplicación. Dichas piezas pueden ser una sentencia de código, una función o un proceso en sí que se ejecutarán en paralelo. La granularidad es categorizada en paralelismo de grano fino y paralelismo de grano grueso. De grano fino es cuando el código se divide en una gran cantidad de piezas pequeñas. Es a un nivel de sentencia donde un ciclo se divide en varios subciclos que se ejecutarán en paralelo. De grano grueso es a nivel de subrutinas o segmentos de código, donde las piezas son pocas y de cómputo más intensivo que las de grano fino.

Modelo datos en paralelo

El modelo datos en paralelo es uno de los modelos más simple. Las tareas son asignadas a cada uno de los procesos de forma estáticamente o semi-estática, y cada tarea ejecuta operaciones similares con datos diferentes. El trabajo puede hacerse en fases y los datos operados en éstas pueden ser diferentes. Normalmente, los datos de las fases se entremezclan con las interacciones para sincronizar las tareas o para obtener los nuevos datos. Dado que todas las tareas deben realizar cálculos similares, la descomposición del problema generalmente se basa en fragmentar los datos, debido a que si se logra repartir los datos uniformemente entonces se garantiza que la carga de trabajo esté balanceada.

Modelo maestro-esclavo

El modelo maestro-esclavo consiste en tener un proceso maestro (pueden ser varios también) el cual debe generar el trabajo y asignarlo a los procesos esclavos. Las tareas pueden asignarse a priori si se puede estimar el tamaño de las tareas, y en caso de que la asignación se haga de forma aleatoria se debe garantizar que este balanceado la carga de trabajos. Otra forma es atribuirle a los esclavos trabajos más pequeños en diferentes momentos. Este último es recomendado cuando le lleva mucho tiempo al maestro crear los trabajos y por lo tanto no es conveniente que los esclavos estén esperando hasta que se hayan generado todos estos. Cuando se

utilice este modelo se debe tener cuidado porque se debe asegurar que el maestro no se convierta en un **cuello de botella**, lo cual puede suceder si las tareas son demasiadas pequeñas (o los esclavos son relativamente rápidos). La granularidad de las tareas debe ser elegida de manera que el costo de hacer el trabajo sea mayor que el costo de la transferencia y la sincronización.

Modelo piscina de trabajo

El modelo piscina de trabajo o piscina de tareas se caracteriza por una asignación dinámica de tareas en procesos para equilibrar la carga en el cual cualquiera de las tarea pueda ser realizada por cualquier proceso. La asignación puede ser centralizada o descentralizada. Los punteros de las tareas (dirección en memoria) pueden almacenarse en una lista compartida, cola de prioridad, tabla hash, un árbol o en estructura datos distribuida físicamente. El trabajo puede estar disponible estáticamente en el comienzo, o puede ser generado dinámicamente, es decir, los procesos pueden generar trabajos y añadirlos a la piscina global. Si el trabajo se genera de forma dinámica y se utiliza la asignación descentralizada, entonces será necesario que haya un algoritmo de detección de terminación [48] para que todos los procesos puedan saber cuando el programa finaliza en su totalidad (por ejemplo, el agotamiento de todas las tareas posibles) y dejar de buscar más trabajo.

Modelo productor-consumidor

En el modelo productor-consumidor (también conocido como tubería), un flujo de datos se transmite a través de una sucesión de procesos, donde cada uno realiza algunas tareas. Esta ejecución simultánea de diferentes programas en un flujo de datos se llama paralelismo de flujo. Con la excepción de los procesos iniciales, la llegada de nuevos datos desencadena la ejecución de una nueva tarea por un proceso en la tubería. Los procesos pueden crear tales tuberías en forma de matrices lineales o multidimensionales, árboles o grafos generales con o sin ciclos. Este modelo por lo general implica una asignación estática de las tareas en los procesos.

1.5. Optimización por nube de partículas en entornos paralelos

A continuación se mencionarán algunas de las aportaciones clásicas, aceptadas como las mejores alternativas para llevar a cabo la paralelización de la PSO:

1.5.1. Modelos con población global

En estos modelos de la PSO paralela, el esquema básico del algoritmo se mantiene igual que el secuencial, por tanto, tendremos una sola población global. El objetivo será repartir la carga de trabajo entre varios procesadores. Se han definido tres aproximaciones diferentes [51] para implementar este enfoque.

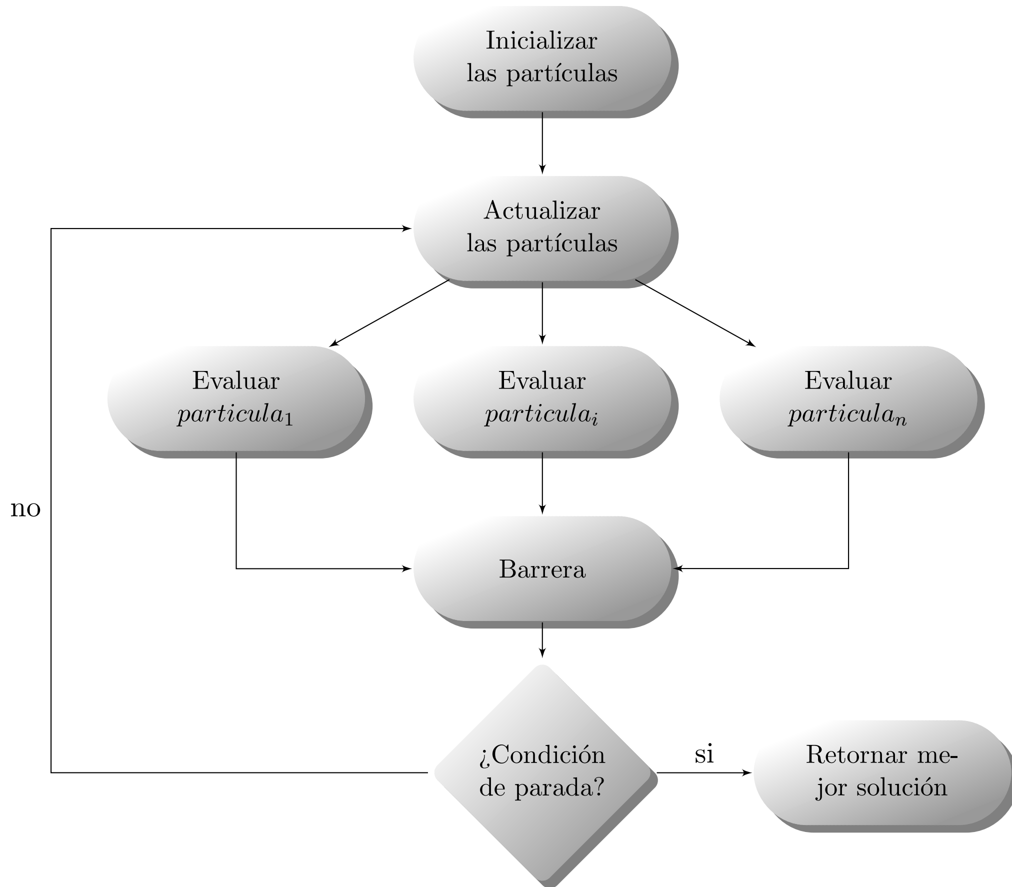


Figura 1.7: Algoritmo paralelo de PSO con población global.

Variante síncrona con maestro-esclavos

Uno de los procesadores adopta el papel de maestro, y el resto el de esclavos. El procesador maestro realiza los cálculos necesarios para generar los nuevos individuos, mientras que los nodos esclavos se encargan de calcular el *fitness* de los individuos generados de forma distribuida, es decir, cada esclavo se hace cargo del cálculo del *fitness* de una parte de la población generada en cada iteración.

Este enfoque será más o menos inteligente en función de la complejidad del cálculo del *fitness*, debido a su naturaleza síncrona. Si el costo computacional del cálculo del *fitness* es complejo, entonces sí obtendremos un beneficio notable de repartir este cómputo entre los nodos esclavos. Sin embargo, si el cálculo del *fitness* no requiere gran potencia de cálculo, el procesador maestro estará notablemente más sobrecargado que el resto de los esclavos, lo que provocará un desbalanceo preocupante de la carga, desperdiciando recursos.

Variante semi-síncrona con maestro-esclavos

Esta variante es muy similar a la anterior con la diferencia de que el trabajo de los esclavos se estructura de manera distinta. En lugar de fragmentar la población entre los procesadores esclavos disponibles, se divide en partes más pequeñas. Inicialmente, cada esclavo recibe como carga de trabajo una de estas pequeñas particiones de la población, de la cual calcula su *fitness*. Cuando los procesadores terminan los cálculos que les han sido encomendados, solicitan una nueva partición para calcular el *fitness*. Con esta variante conseguimos un mejor equilibrio de carga entre los esclavos ya que, si algunos de ellos estuvieran saturados, no supondría un especial retraso del tiempo de cómputo total.

Variante asíncrona concurrente

En las variantes anteriores el mayor problema residía en el reparto no equitativo entre el maestro y los esclavos. En esta propuesta se pretende solventar este problema eliminando el proceso maestro haciendo que todos los procesadores trabajen cooperando al mismo nivel. Para esto es necesario mantener la población global en memoria compartida, de tal modo que cada nodo sea capaz de generar, de manera independiente, una parte de los nuevos individuos hijos y, también, de calcular su *fitness*.

1.5.2. Modelos de islas

En la paralelización con islas no existe el concepto de población global, sino que cada unión de nodo y proceso se ve como si fuese una isla con su propia población local. Cada proceso supone una instancia de nuestro algoritmo, que ejecuta de forma secuencial el mismo problema. Cada población local es inicializada independientemente, de manera que cada isla partirá de poblaciones distintas y obtendrá resultados diferentes. Al final de la ejecución, las poblaciones de cada una de las islas son combinadas para conformar una población global de la que seleccionaremos el mejor individuo.

Este enfoque tiene la ventaja de que es mucho más resistente a óptimos locales durante el proceso de búsqueda, debido a que cada población es teóricamente independiente de las demás. Una característica muy interesante de este modelo es la introducción del concepto de migración. La migración entre islas es un artefacto que proporciona el intercambio periódico de individuos entre islas de acuerdo a una red de interconexión definida.

Se pueden distinguir diferentes modelos de islas en función de la comunicación entre las subpoblaciones. Algunas comunicaciones típicas son las siguientes:

- **Comunicación en estrella**, en la cual existe una subpoblación que es seleccionada como maestra (aquella que tiene mejor media en el valor de la función objetivo), siendo las demás consideradas como

esclavas. Todas las subpoblaciones esclavas mandan sus p_i mejores individuos ($p_i \geq 1$) a la subpoblación maestra, la cual a su vez manda sus p_j mejores individuos ($p_j \geq 1$) a cada una de las subpoblaciones esclavas.

- **Comunicación en red**, en la cual no existe una jerarquía entre las subpoblaciones, mandando todas y cada una de ellas sus p_i mejores individuos al resto de las subpoblaciones.
- **Comunicación en anillo**, en la cual cada subpoblación envía sus p_i mejores individuos a una población vecina, efectuándose la migración en un único sentido de flujo.

A la hora de definir este nuevo modelo entran en juego cuatro conceptos a tener en cuenta:

- Topología de interconexión de las islas: define cuales islas deben migrar individuos y se representan como un grafo dirigido.
- Periodicidad de la migración.
- Cantidad y calidad de los individuos a migrar.
- Sincronización entre islas: la manera en que las islas se coordinan para migrar los individuos nos plantea dos alternativas.
  - Modelo síncrono: en este modelo las islas se configuran para que intercambien los individuos cada cierto número de generaciones, de manera que cada isla ejecutará de forma secuencial el algoritmo y, llegados a cierta generación preseleccionada, pasará a la fase de migración. En la fase de migración las islas han de sincronizarse y esperar a que se envíen y reciban los individuos. Esto provoca retardos, ya que las islas más rápidas deberán esperar por las más lentas.
  - Modelo asíncrono: en el modelo asíncrono las fases de evolución y de migración están acopladas. Los envíos se realizan de forma asíncrona y, periódicamente, se comprueba si hay inmigrantes. De este modo, no se producen retrasos, ya que cada nodo ejecutará a la mayor velocidad posible sin tener que esperar por los nodos más lentos.

1.5.3. Otras implementaciones

En la actualidad existe una biblioteca para el diseño de PSO, y se ha guiado por la arquitectura de la biblioteca **MALLBA** [52] desarrollada en el proyecto **TIC1999-0754-C03**, en el que colaboran las Universidades de Málaga (**MA**), La Laguna (**LL**) y la Universidad Politécnica de Cataluña (**BA**). La implementación se realizó en el lenguaje de programación C++ [53] y se desarrollaron distintas versiones de PSO (secuencial, distribuido en LAN y en WAN). Esta biblioteca presenta algunas dificultades, por ejemplo, en la base del proyecto se debe utilizar código desarrollado por otras personas, lo que implica cierta dificultad ya que hay

que familiarizarse con un código no desarrollado por ti mismo. Para el paso de mensajes utilizan la biblioteca *NetStream* [54] la cual se apoya en *Message Passing Interface* (MPI), proporcionando una interfaz sencilla para aquellos usuarios que no son expertos en programación paralela, pero a la vez pierde eficiencia respecto a MPI [55]. Este proyecto no es actualizado desde el año 2002.

Otra biblioteca es CILib (*Computational Intelligence Library*) liberada bajo la segunda versión de la licencia *GNU General Public License (GPL)*. Esta biblioteca ha sido desarrollada por un grupo de investigación de la Universidad de Pretoria y por algunos usuarios de la comunidad de *Source Forge*. La principal desventaja es que fue implementada en Java, por lo cual no puede ser utilizada en el *cluster* de nuestra universidad.

1.6. Conclusiones del capítulo.

En el presente capítulo se realizó una introducción a las técnicas metaheurísticas donde se hizo un breve resumen de las técnicas principales. Además se describieron las características de la PSO, se mencionaron algunos ejemplos de problemas a los que ha sido aplicada la PSO, las variantes y topologías más utilizadas. Se realizó una introducción a la computación paralela, a las arquitecturas, modelos y formas de evaluación de los algoritmos paralelos, así como los diferentes modelos de paralelización que han sido aplicados a la PSO.

Como hemos visto, muchos problemas que para su resolución han utilizado la PSO han obteniendo buenos resultados, con la paralelización de la PSO debe mejorar la calidad de las respuestas y reducirse el tiempo de ejecución del programa.

Capítulo 2

Herramientas y metodologías.

En este capítulo se describen las herramientas software y las herramientas hardware para la programación y ejecución de los programas secuenciales y paralelos desarrollados en el presente trabajo. Además se especifican las características de cada una de las herramientas hardware, la descripción del conjunto de bibliotecas y entornos usados en la realización de este trabajo, y los procedimientos metodológicos.

2.1. Herramientas software.

Las herramientas software se componen por el ambiente paralelo de programación, el lenguaje de programación, y las bibliotecas secuenciales y paralelas de álgebra lineal numérica.

2.1.1. Lenguaje de programación C/C++.

Lenguaje de programación creado para realizar modificaciones al sistema operativo Unix, rápido y eficiente, junto al FORTRAN son los lenguajes que con más frecuencia son usados para la creación de aplicaciones de optimización, su versión orientada a objeto C++, permite agregar un nivel más de abstracción a nuestra solución. Es un lenguaje muy eficiente puesto que es posible utilizar sus características de bajo nivel para realizar implementaciones óptimas. A pesar de su bajo nivel es el lenguaje más portado en existencia, habiendo compiladores para casi todos los sistemas conocidos y proporciona facilidades para realizar programas modulares y/o utilizar código o bibliotecas existentes.

2.1.2. Colección de compiladores GCC.

Colección de compiladores creados por GNU, dentro de dicha colección se encuentran compiladores para ADA, JAVA, C, C++, FORTRAN, entre otros; el compilador para C y C++ es uno de los que más se acerca al estándar de C y C++, además ser distribuido bajo la licencia GPL y LGPL.

2.1.3. Entorno integrado de desarrollo Code::Blocks.

Entorno integrado de desarrollo, *Integrated Development Environment* (IDE), para el lenguaje C y C++, muy ligero y multiplataforma. También reconoce varios compiladores para dichos lenguajes de programación, algunos ejemplos de estos son GCC, Borland C++ Compiler, Intel C++ Compiler y Microsoft Visual Studio Toolkit; el código de la aplicación está escrito completamente por medio de este IDE y está bajo la licencia GPL.

2.1.4. Entorno de paso de mensajes

Nuestros algoritmos se apoyan en la biblioteca MPI [56, 57] debido a su actualidad y disponibilidad de distribuciones prácticamente en cualquier arquitectura. Esta biblioteca proporciona tipos de datos, procedimientos y funciones con las cuales el programador puede desarrollar aplicaciones paralelas para sistemas multiprocesadores, tanto para redes locales (LAN) como para redes de área amplia (WAN). La ventaja de MPI sobre otras bibliotecas de paso de mensajes, es que los programas que utilizan la biblioteca son portables (dado que MPI ha sido implementado para casi toda arquitectura de memoria distribuida) y rápidos (porque cada implementación de la biblioteca ha sido optimizada para el hardware en la cual se ejecuta). Además existen implementaciones libres como MPICH y LAM-MPI.

2.1.5. Bibliotecas de Álgebra Lineal

Con el objetivo de lograr portabilidad y eficiencia en las rutinas, tanto secuenciales como paralelas, se usaron en la implementación funciones de las bibliotecas secuenciales estándar de álgebra lineal *Basic Linear Algebra Subprograms* (BLAS) [58] y *Linear Algebra PACKage* (LAPACK) [59].

La biblioteca BLAS se compone de funciones de alta calidad que se basan en la construcción de bloques para efectuar operaciones básicas con vectores y matrices. BLAS está construido en tres niveles: el nivel 1 que efectúa operaciones vector-vector, el nivel 2 que efectúa operaciones matriz-vector y el nivel 3 que efectúa operaciones matriz-matriz. Esta herramienta es eficiente, portable y de amplia disponibilidad, por lo que se utiliza comúnmente en el desarrollo de software del álgebra lineal de altas prestaciones.

La biblioteca LAPACK proporciona rutinas para resolver sistemas de ecuaciones, problemas de mínimos cuadrados, problemas de valores propios, vectores propios y valores singulares. También proporcionan rutinas para la factorización de matrices, tales como LU, Cholesky, QR, SVD, Schur; tanto para matrices con estructuras específicas o matrices generales.

2.1.6. Kile.

Entorno integrado de LaTeXpara el escritorio KDE. Proporciona la habilidad de utilizar todas las funcionalidades de LaTeXcon una interfaz gráfica, autocompletamiento de comandos de LaTeX, resaltado de la sintaxis, proporcionándole un acceso sencillo, inmediato y personalizado a todos los programas para la compilación, post-procesamiento, depuración, conversión y visualización de LaTeX. Es distribuido bajo la licencia GPL.

2.1.7. KBibTex.

Es una herramienta para manejar las listas de referencias con formato BibTex, utilizadas para la preparación de documentos LaTeX. KBibTeX facilita la realización de citas bibliográficas de un modo consistente mediante la separación de la información bibliográfica de la presentación de esta información. Como está diseñado para el escritorio KDE se puede integrar con Kile, además es liberado bajo la licencia GPL.

2.1.8. OpenOffice.

Suite ofimática de software libre y código abierto de distribución gratuita que incluye herramientas como procesador de textos, hoja de cálculo, presentaciones, herramientas para el dibujo vectorial y base de datos. Está disponible para muchas plataformas como Microsoft Windows y sistemas de tipo Unix como GNU/Linux, BSD, Solaris y Mac OS X. El código fuente de la aplicación está disponible bajo la licencia LGPL.

2.2. Herramientas hardware.

En este trabajo de diploma se utilizaron computadoras personales interconectadas a una red Fast-Ethernet. Se trabajó en la máquina paralela Atropos, que es una red de computadoras funcionando con el sistema operativo GNU/Linux Debian Lenny 5.0 y consta de ocho nodos con dos procesadores Intel(R) Core(TM)2 Duo CPU E4500 2.20GHz. Los nodos están interconectados mediante Ethernet Switch 10/100 Mbps, con una topología Beowulf o de anillo y cada nodo consta de 1 Gigabyte de memoria RAM. Todos los nodos están disponibles para cálculo científico.

2.3. Metodologías.

El método teórico utilizado para el cumplimiento de los objetivos del trabajo es el analítico-sintético, el que nos permitió analizar las teorías, documentos, etcétera; permitiendo la extracción de los elementos más importantes que se relacionan con el objeto de estudio. Además se empleó el método empírico de la observación, mediante la cual se realizan modificaciones precisas en el modelad de los casos de estudio, permitiendo obtener buenos resultados.

Capítulo 3

Resultados y discusión.

En este capítulo se exponen y discuten los resultados alcanzados con el desarrollo del trabajo de diploma, los cuales dan respuesta a los objetivos planteados para la investigación. Se describen las clases y los principales métodos de la biblioteca. Además se explica el modelado, implementación y pruebas al PIAVS.

3.1. Introducción

La biblioteca desarrollada es de propósito general, ha sido diseñada para resolver varios problemas de optimización que pueden ser diferentes entre sí. Es importante señalar que la biblioteca no puede solucionar un problema sin que haya sido adaptado a ésta, ya que usuario debe dar ciertas informaciones sobre el problema en concreto. Para lograr esta reusabilidad se ha utilizado la programación orientada a objetos. Además se emplearon estructuras de datos y algoritmos de la biblioteca *Standard Template Library* (STL), la cual utiliza contenedores genéricos para el lenguaje C++. La biblioteca brinda la posibilidad de ser utilizada tanto para una ejecución secuencial como para una paralela.

Para resolver un problema sólo es necesario adaptar algunas clases y métodos, por ejemplo definir cómo estaría conformada la partícula, cuál sería la función objetivo a evaluar, el criterio de convergencia, etcétera. Cuenta con la implementación los principales métodos, como por ejemplo, la evolución de la nube de forma paralela o secuencial, la actualización del vector mejor posición global de las partículas con la versión del anillo y *GBest*, empaquetar y desempaquetar las partículas. También permite la herencia de clases y que se puedan redefinir la mayoría de los métodos para que los usuarios más avanzados realicen sus propias implementaciones.

Además se definieron algunas clases que cuentan con tres métodos (`PACK`, `UNPACK` y `PACKSIZE`), los cuales son utilizados en las versiones paralelas que utilicen el envío de mensajes. Las funcionalidades `PACK` y `UNPACK` van a permitir empaquetar y desempaquetar respectivamente. El empaquetado no es más que guardar en un flujo de datos informaciones de la clase, mientras que cuando se desempaqueta se realiza la operación inversa,

del flujo de datos se extrae la información para ser utilizada. El método `PACKSIZE` determina el tamaño en *bytes* del paquete.

3.2. Estructura de la biblioteca de optimización por Nube de Partículas.

Para resolver cualquier problema se necesita definir cómo está conformada la partícula, la cual no es más que una solución del problema en cuestión. La clase que representa la partícula en la biblioteca es `PSO_Particle`, es abstracta y para crear nuestra propia partícula se hereda de la clase abstracta. Por cuestiones de flexibilidad se agrega a la biblioteca la clase `PSO_SetUpParams`, en donde se van a agrupar algunos atributos o parámetros que puedan ser utilizados por la partícula para su inicialización o ejecución.

La subnube (`PSO_SSwarm`) va a estar compuesta por un cúmulo de partículas y es la encargada de realizar la evolución de las mismas dentro su entorno y determina para cada partícula cuál es la mejor de su entorno. Estos métodos pueden ser redefinidos de acuerdo con el algoritmo que se vaya a utilizar. La nube (`PSO_Swarm`) va a contener una o varias subnubes y decidirá cuándo se ha alcanzado la condición de terminación del algoritmo, la nube cuenta con la funcionalidad que especifica cómo es que va a evolucionar.

Otras clases utilizadas son `PSO_Migration` y `PSO_OFunction`. La primera va a determinar cómo se realiza la migración de partículas entre diferentes nubes, de acuerdo a cómo se haya definido. Es aconsejable utilizarla en el caso de que existan varias nubes, para que puedan intercambiar partículas y así se puede evitar o reducir las posibilidades que el algoritmo quede atrapado en óptimos locales. La segunda clase va a contener la definición de la función objetivo, correspondiente al problema problema de optimización, la que calculará el valor de adaptación de una partícula especificada. En el siguiente diagrama [3.1] se puede ver la relación entre las clases de la biblioteca:

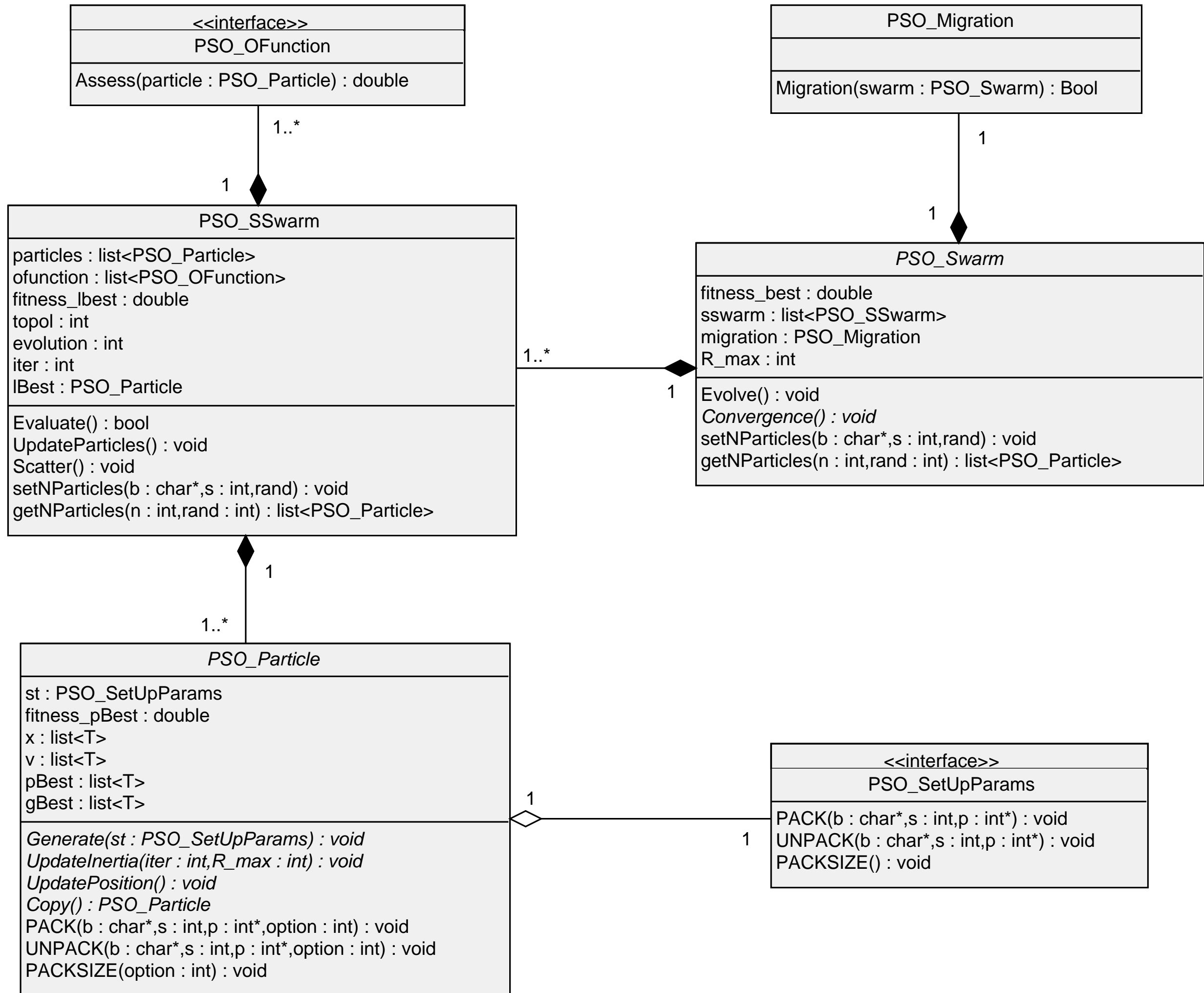


Figura 3.1: Diagrama de clases de la biblioteca.

3.2.1. Principales métodos de las clases de la biblioteca

A continuación se describen brevemente los métodos más significativos de las clases más relevantes en el funcionamiento de la biblioteca.

- `PSO_Particle`
  - `void Generate( PSO_SetUpParams* st )`: inicializa la partícula.
  - `void UpdateInertia( int iter, int R_max )`: actualiza el factor de inercia de la partícula, el cual depende de la iteración actual (`iter`) y la cantidad máxima de iteraciones (`R_max`).

- `void UpdatePosition()`: actualiza los vectores velocidad y posición, que no es más que mover la partícula a otra posición en el espacio de búsqueda.
- `void PACK( char* buffer, int size, int* pos, int option )`: es el encargado de empaquetar la partícula y almacenarla en el vector `buffer`. El parámetro `option` determina la cantidad de vectores que deben empaquetarse y en el orden siguiente: vector posición, vector mejor posición personal, vector velocidad y vector mejor posición global.
- `void UNPACK( char* buffer, int size, int* pos, int option )`: es el responsable de desempaquetar el vector `buffer` y guardar la información que tenga en la partícula. El parámetro `option` va a cumplir la misma función que cuando se empaqueta, pero con la diferencia que determina la cantidad de vectores que deben empaquetarse.
- `void PACKSIZE( int option )`: calcula el tamaño en *bytes* de la partícula que se vaya a empaquetar o desempaquetar. La variable `option` va a indicar la cantidad de vectores a empaquetar o desempaquetar.

- `PSO_SSwarm`
  - `bool Evaluate()`: define cómo evoluciona la subnube, donde se evalúan todas las partículas de la subnube para determinar cuán adaptadas están en su entorno, actualizándose el vector mejor posición personal en caso de cumplir la condición, va a definir para cada partícula cuál es la mejor de su entorno, y se actualiza la próxima posición de cada una. Devuelve verdadero cuando haga falta terminar el algoritmo, y es utilizado principalmente por el modelo Maestro-Esclavo. Por defecto está implementado de tres maneras:
    - Secuencial: versión secuencial del algoritmo.
    - Maestro-Esclavo versión 1: versión paralela donde se empaquetan todos los vectores de las partículas y se envían a los procesadores esclavos, los cuales van a calcular el valor de adaptación de las partículas y además van a actualizar la próxima posición de éstas. Cuando terminan de realizar estas operaciones se empaquetan las "nuevas partículas" y se envían al proceso Maestro. Es recomendado cuando los procesos de calcular el valor de la función objetivo y actualizar la posición son muy costosos computacionalmente.
    - Maestro-Esclavo versión 2: versión paralela donde se empaqueta solamente el vector posición de las partículas y se distribuye en los esclavos. Éstos van a calcular el valor de adaptación de las partículas y enviar esos valores al Maestro. Puede ser utilizado cuando el cálculo de la función objetivo es muy complejo.
  - `void UpdateParticles()`: se actualiza en cada partícula el vector mejor posición global. Por defecto se puede seleccionar la versión mejor local (anillo) o la mejor global (*gBest*), se puede seleccionar si la actualización de los vectores va ser de forma síncrona o asíncrona. La primera consiste en

actualizar el mejor global de las partículas cuando todas hayan terminado de evaluarse. Mientras que en la segunda variante, cuando se termina de evaluar la partícula, ella actualiza su vector mejor global y se mueve a la siguiente posición.

  - `void Scatter()`: es utilizado para "dispersar" las partículas cuando la subnube esté estancada en un óptimo local. Ésto consiste en inicializar todos los vectores de la partícula para que vuelvan a empezar la búsqueda.

- `PSO_Swarm`
  - `void Evolve()`: es el responsable de "guiar" la evolución de la(s) subnube(s). En la versión implementada no existe comunicación entre las subnubes, cada una evolucionará de forma independiente.
  - `bool Convergence()`: devuelve verdadero o falso si la nube ha alcanzado la condición de terminación.

- `PSO_Migration`
  - `bool Migration( PSO_Swarm* swarm )`: en el método se realiza la migración de partículas de una nube a otra. En caso de que las partículas migren correctamente, retorna verdadero. La migración de partículas es utilizada en la versión paralela con el modelo de islas.

- `PSO_OFunction`
  - `double Assess( PSO_Particle* particle )`: calcula el valor de adaptación (*fitness*), es decir, evalúa la función objetivo en el punto representado por la partícula pasada por parámetro y es retornado ese valor.

3.3. Experimentos con algunos problemas de optimización

En esta sección se van a realizar varias pruebas a la biblioteca con cuatro funciones matemáticas de optimización, clásicas en la computación evolutiva. Estas funciones matemáticas, que detallaremos más adelante, trabajan sobre dominios continuos y son habitualmente utilizadas para evaluar el rendimiento de los algoritmos de optimización numérica. [60, 61, 62]

El principal objetivo de estas pruebas es comparar el tiempo de ejecución de la biblioteca con el de otra implementación, específicamente la versión estándar de la PSO [63], la cual fue validada por algunos investigadores del campo, en particular, James Kennedy y Maurice Clerc.

Las funciones matemáticas seleccionadas fueron:

- **Spherical**: cualquier algoritmo numérico de optimización debe resolverla sin ningún problema. Es una función unimodal muy simple, no tiene interacción entre sus variables y la información del gradiente siempre apunta hacia el mínimo global. El mínimo global está ubicado en $x^* = 0$ con $f_1(x^*) = 0$

$$f_1(x) = \sum_{i=1}^{n} x_i^2 \quad \forall\ i \in [1..n], |x_i| \leq 100 \tag{3.1}$$

- **Rosenbrock**: no todas las funciones unimodales son fáciles de resolver. En esta función las variables son muy dependientes y la información del gradiente a menudo induce errores. Es una función unimodal donde el mínimo global está en el punto $x^* = 1$ con $f_2(x^*) = 0$

$$f_2(x) = \sum_{i=1}^{n-1} \left[(1 - x_i)^2 + 100 \left(x_{i+1} - x_i^2 \right)^2 \right] \quad \forall\ i \in [1..n], |x_i| \leq 30 \tag{3.2}$$

- **Griewank**: define un problema de minimización multimodal con interacción significativa entre las variables, causado por el término producto. Además tiene una interesante propiedad, que la cantidad de mínimos locales aumenta con la dimensionalidad. Sin embargo, la influencia del término producto disminuye en estas circunstancias. El mínimo global de la función está en $x^* = 0$, con $f_3(x^*) = 0$.

$$f_3(x) = \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}} \right) + 1 \quad \forall\ i \in [1..n], |x_i| \leq 600 \tag{3.3}$$

- **Rastrigin**: una versión multimodal no lineal de la función **Spherical** que se caracteriza por mínimos locales profundos dispuestos en forma de protuberancias sinusoidales. El algoritmo de optimización puede fácilmente quedar atrapado en un mínimo local. El mínimo global está en el punto $x^* = 0$ donde $f_4(x^*) = 0$.

$$f_4(x) = \alpha n \sum_{i=1}^{n} \left(x_i^2 - \alpha \cos(2\pi x_i) \right) \quad \forall\ i \in [1..n], |x_i| \leq 5{,}12 \tag{3.4}$$

Resultados de las pruebas

Comenzaremos por definir la partícula. Para estos problemas, la partícula va a representar una posición del espacio de búsqueda, es decir, una tupla de n elementos donde n es la dimensión de este espacio. Es importante señalar que el principal objetivo de estas pruebas es poder comparar la biblioteca con otras implementaciones, para medir el tiempo de ejecución. Todas las pruebas realizadas utilizaron los siguientes parámetros:

- Cantidad de partículas: 100

- Factor de inercia: 1,6609640
- Peso cognitivo y social: 0,8010299
- Cantidad máxima de iteraciones: 1000
- Topología: *Global best*

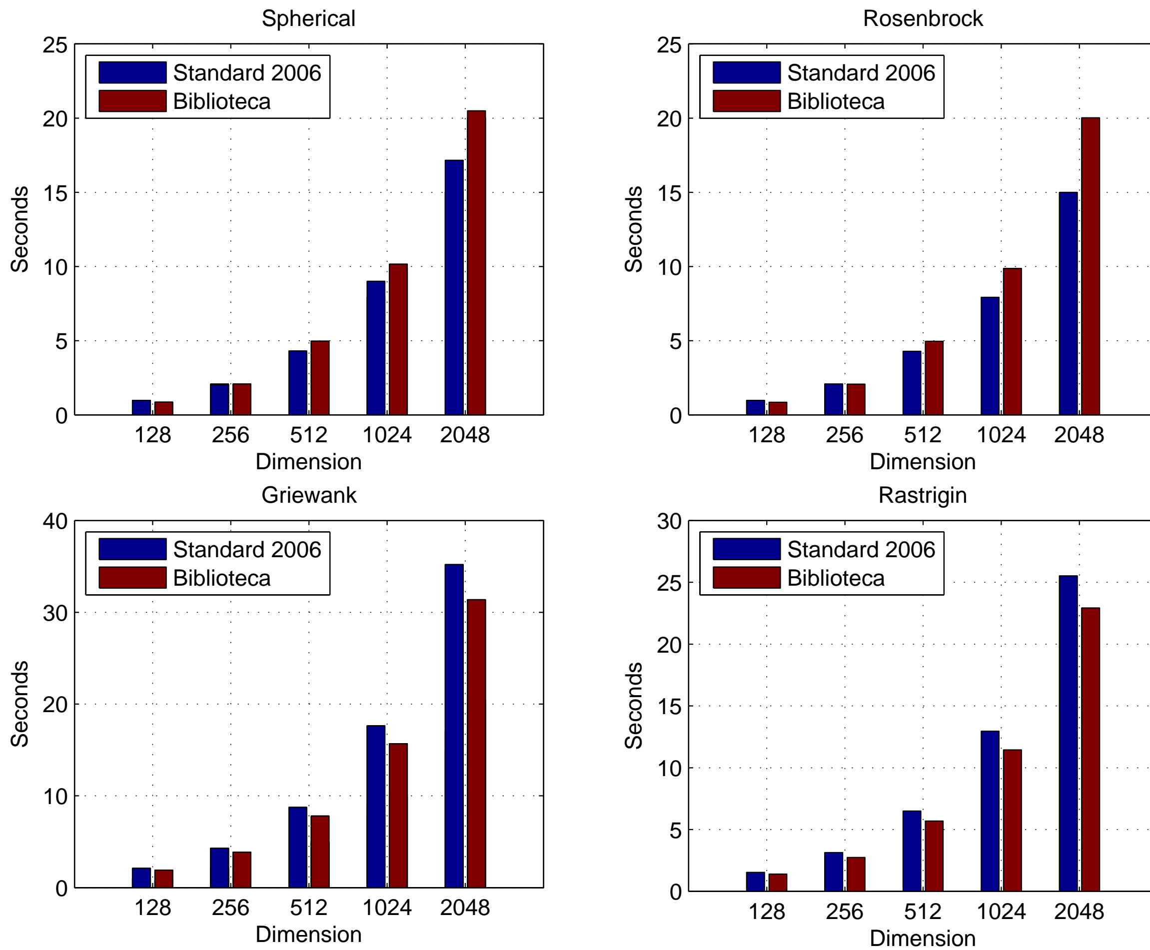


Figura 3.2: Gráfica para comparar tiempo de ejecución (Standard 2006 vs Biblioteca).

Como se puede observar, no existen muchas diferencias entre los tiempos de ejecución de las dos implementaciones. La versión estándar, a pesar no presentar una jerarquía de clases (utiliza la programación estructurada) y trabajar con arreglos, no todos los casos obtiene el menor tiempo de ejecución.

3.4. Caso de estudio: Problema Inverso Aditivo de Valores Singulares

Los problemas inversos son aquellos donde los valores de algunos parámetros del modelo deben ser obtenidos de los datos observados, en otras palabras se podría describir como problemas en los que se conoce la respuesta pero no la pregunta. Estos problemas son de gran importancia para diferentes aplicaciones de la ciencia y la ingeniería.

Existen varios ejemplos donde se aplican estos problemas, como es el caso de la tomografía, que no es más que el procesado de imágenes por secciones y es usado en medicina, arqueología, biología, geofísica, oceanografía y pruebas no destructivas. Un problema inverso específico es el Problema Inverso de Valores Singulares [64], y dentro de éste, el PIAVS [65], el cual se define como:

Dado un conjunto de matrices reales $A_0, A_1, ..., A_n \in \mathbb{R}^{m\times n}(m \geq n)$ y un conjunto de valores reales no negativos $\sigma^* = \{\sigma_1^*, \sigma_2^*, ..., \sigma_n^*\}$ tales que $\sigma_1^* \geq \sigma_2^* \geq ... \geq \sigma_n^*$, encontrar un vector $c = (c_1, c_2, ..., c_n)^t \in \mathbb{R}^n$ de modo que el conjunto de valores singulares de

$$A(c) = A_0 + c_1A_1 + ... + c_nA_n \tag{3.5}$$

sea precisamente σ^*.

Este problema generalmente se formula como un sistema de ecuaciones no lineales y se resuelve con diferentes variantes de métodos de Newton. En [65] se analizan algunas de éstas. Por otro lado, este problema puede verse también como un problema de optimización, pues se desea que la distancia entre los valores singulares dados σ^* y los valores singulares de $A(c)$, denotado como $\sigma(c)$, sea mínima. De modo que se define la función $f : \mathbb{R}^n \to \mathbb{R}$ como $f(c) =\| \sigma^* - \sigma(c) \|_2$ y el problema se pasa a ser en encontrar un vector $c = (c_1, c_2, ..., c_n)^t \in \mathbb{R}^n$ que minimice $f(c)$.

Este es un problema muy complejo, en primer lugar por el coste elevado de la evaluación de la función f, pues solamente el cálculo de los valores singulares de una matriz es aproximadamente $\frac{38}{3}n^3$; y en segundo lugar porque a medida que aumenta el tamaño del problema se hace más irregular la función y es más complicado encontrar el mínimo.

Los experimentos se realizaron con matrices que presentan la siguiente definición:

$$A_0 = 0,\ A_k = \left(A_{ij}^k\right) \in \mathbb{R}^{n\times n}\ A_{ij} = \begin{cases} 1 & si\ i-j = k-1 \\ 0 & en\ otro\ caso \end{cases}$$

y el vector c^*, vector para el cual se cumple $\sigma(A(c^*)) = \sigma^*$, es aleatorio. Se escogió este caso en específico por su bajo costo en almacenamiento, pues la matriz $A(c)$ tendrá siempre la siguiente estructura:

$$A(c) = \begin{bmatrix} c_1 & & & \\ c_2 & c_1 & & \\ \vdots & \vdots & \ddots & \\ c_n & c_{n-1} & \cdots & c_1 \end{bmatrix}$$

3.4.1. Resolución del problema

Para resolver el PIAVS se utilizó la versión de la PSO en el espacio de búsqueda continuo y se definieron e implementaron las clases y métodos necesarios para este problema en específico. También se utilizan las bibliotecas BLAS y LAPACK para las operaciones con vectores y matrices. A continuación se describirán las clases implementadas:

Clase `PSO_ParticleIASVP`

La partícula va a representar el vector $c = (c_1, c_2, ..., c_n)^t$ con el cual se construye la matriz 3.4. La clase `PSO_ParticleIASVP` hereda de `PSO_Particle`. Se implementaron o redefinieron los siguientes métodos:

- `void UpdateInertia( int iter, int R_max )`: el factor de inercia, ω, se va a ir reduciendo progresivamente en cada iteración aplicando la siguiente ecuación:

$$\omega = \omega_{max} - \frac{\omega_{max} - \omega_{min}}{iter_{max}} \cdot iter \tag{3.6}$$

  donde ω_{max} y ω_{min} van a ser el valor máximo y mínimo respectivamente que puede alcanzar ω. Mientras, $iter_{max}$ va ser la cantidad máxima de iteraciones que puede hacer el algoritmo e $iter$ va ser la iteración actual.

- `void UpdatePosition()`: en este método se actualizan los vectores posición y velocidad y se realiza utilizando la ecuación propuesta por Shi y Eberhart 1.5.

- `void UpdatepBest()`: para actualizar el vector mejor posición personal (*pBest*) se hace una copia de los datos del vector posición para él.

Clase `PSO_SwarmIASVP`

La clase `PSO_SwarmIASVP` hereda de `PSO_Swarm` y sólo se implementó el método `bool Convergence()` el cual determina cuándo la nube ha convergido. Para que la nube converja tiene que cumplir al menos una de estas condiciones:

- si la iteración actual es mayor o igual que la cantidad máxima de iteraciones.
- si la distancia entre los valores singulares dados σ^* y los valores singulares de la partícula mejor adaptada, es menor o igual que un valor especificado.

Clase `PSO_OFunctionIASVP`

La clase `PSO_OFunctionIASVP` hereda de `PSO_OFunction`. La funcionalidad `double Assess(PSO_Particle* particle)` calcula la distancia entre los valores singulares σ^* y los valores singulares de la partícula pasada por parámetro.

Clase `PSO_SetUpParamsIASVP`

La clase `PSO_SetUpParamsIASVP` implementa la interfaz `PSO_SetUpParams`. La clase va estar compuesta por los siguientes atributos:

- dimension: indica la dimensión del espacio de búsqueda
- W_max: valor máximo que puede alcanzar el factor de inercia
- W_min: valor mínimo que puede alcanzar el factor de inercia
- V_max: valor máximo que puede alcanzar el vector velocidad
- D_max: valor máximo que puede alcanzar el vector posición
- D_min: valor mínimo que puede alcanzar el vector posición
- w: factor de inercia
- c1: peso cognitivo
- c2: peso social

3.4.2. Pruebas a las diferentes variantes implementadas

Como se planteaba anteriormente, están implementadas diferentes variantes de la PSO, por ello, se decide hacer unas pruebas para comparar el rendimiento de cada una en el PIAVS. Las versiones que vamos a comparar son:

PSO-1: variante donde la topología es *lBest*, específicamente la versión del anillo y con actualización sincronizada.

PSO-2: la topología utilizada es la del anillo y se actualizarán las partículas de forma asíncrona.

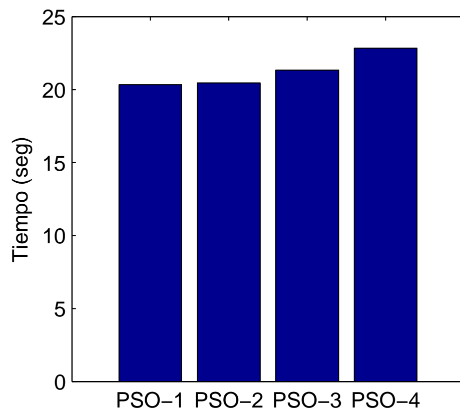
PSO-3: variante donde la topología es *gBest* y con actualización síncrona.

PSO-4: también utiliza la topología *gBest* pero la actualización será asíncrona.

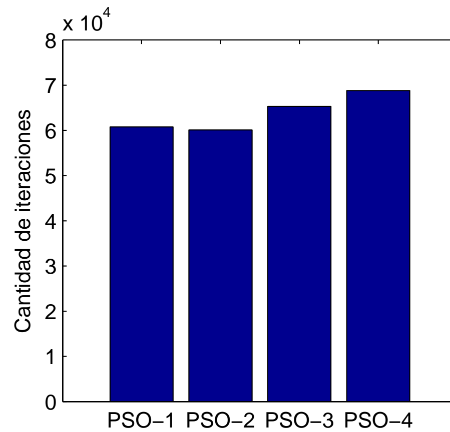
Para los experimentos se utilizaron los siguientes valores:

- Dimensión: 5
- Fitness: 1,00E-11
- Cantidad de subnubes: 1
- Cantidad de partículas: 20
- Cantidad máxima de iteraciones: 500000
- Valor del factor de inercia máximo: 0.8
- Valor del factor de inercia mínimo: 0.3
- Rango de los datos para vector posición: [-32; 32]
- Rango de los datos para vector velocidad: [-32; 32]

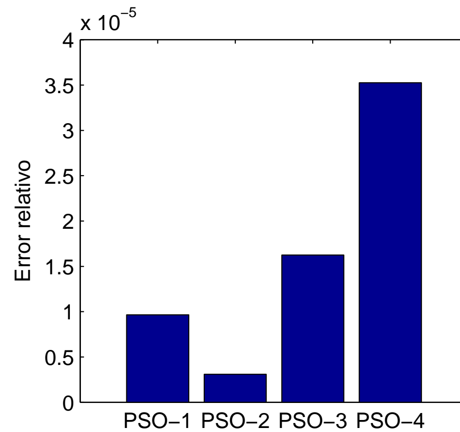
En las siguientes gráficas se muestra el promedio de todos los resultados obtenidos en cuanto a tiempo de ejecución y cantidad de iteraciones:



(a) Gráfica de tiempo



(b) Gráfica de iteraciones



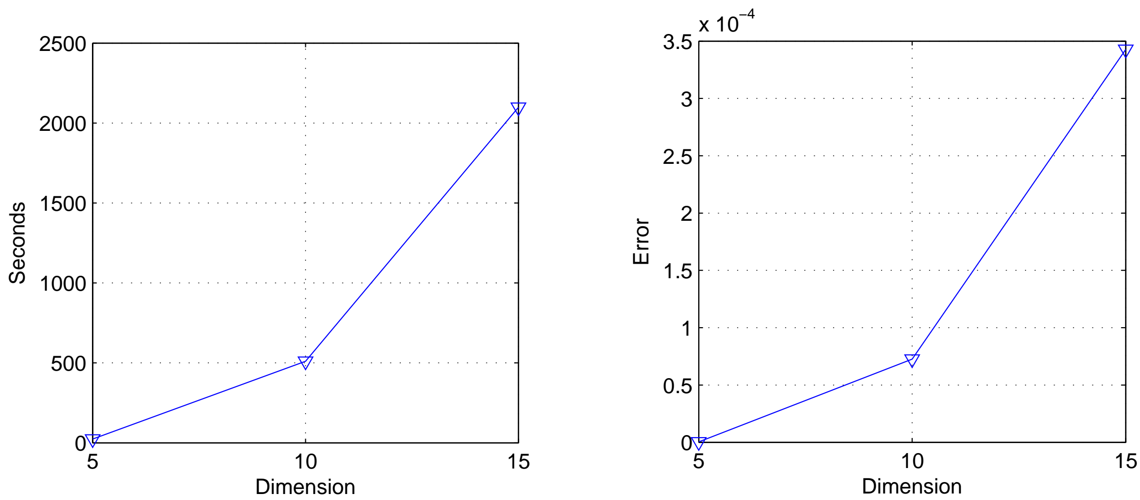
(c) Gráfica de error

Figura 3.3: Comparación entre las variantes implementadas

Como se puede apreciar las versiones que utilizan la topología del anillo dan los resultados en menor tiempo y realizan menos cantidad de iteraciones, y en estos aspectos las dos versiones tienen resultados muy similares. Sin embargo, en cuanto a la calidad de las soluciones el **PSO-2** es superior a las demás y quedó menos veces atrapado en óptimos locales. Por estas razones, en los próximos experimentos se utilizará la versión **PSO-2** y los mismos valores definidos con anterioridad, excepto la dimensión y la cantidad de partículas, donde esta última va ser igual a cuatro veces la dimensión.

Aumentando el tamaño del problema

En el próximo experimento se va a ir incrementando la dimensión del problema para comprobar la efectividad del algoritmo en instancias más complejas. Los resultados de las pruebas se encuentran en el Apéndice A. En los siguientes diagramas se resumen los resultados obtenidos. Es importante señalar que de los experimentos realizados, se descartaron los 4 mejores y los 4 peores resultados.



(a) Gráfica de tiempo

(b) Gráfica de error

Figura 3.4: Comparación cuando se aumenta el tamaño del problema

Como se esperaba, al aumentar el tamaño del problema el tiempo también debía crecer, pero en estas pruebas el tiempo se extiende extremadamente y la calidad de las soluciones empeora. A partir de la dimensión 10, varias pruebas no convergen por la calidad de la solución, sino porque han alcanzado el máximo número de iteraciones. En el caso de la dimensión 15, todas las soluciones convergieron por llegar a la cantidad máxima de iteraciones permitidas.

3.4.3. Hibridación con el método Newton-Bisección

Se demostró anteriormente, que al aumentar el tamaño del problema el algoritmo **PSO-2** no es factible para resolver el PIAVS. A causa de ésto se decide hibridar la PSO con el método de Newton-Bisección [66, 67], para que con la combinación de ambos se obtengan mejores resultados. La PSO sería la encargada de realizar la búsqueda aplicando énfasis en la exploración mientras que Newton-Bisección se enfocaría en la explotación (convergencia más rápida hacia una solución). Para ello se propone actualizar primeramente por la ecuación 1.5, y a ese nuevo vector posición se le aplica el procedimiento de Newton-Bisección, actualizándose nuevamente la posición de la partícula. En la siguiente tabla se muestra el promedio de las pruebas realizadas (Apéndice B):

Con la hibridación se obtuvieron excelentes resultados en todos los aspectos. La calidad de las soluciones se mantuvo con muy buenos valores en todos los casos y el tiempo de ejecución, que es un factor muy importante, se redujo significativamente siendo como mínimo doscientas veces menor con respecto al original.

	PSO			**PSOH**		
Dimensión	5	10	15	5	10	15
Fitness	9,5253E-12	3,4111E-03	2,4269E-02	8,9997E-13	7,0751E-13	6,9309E-13
Error relativo	4,4848E-13	7,8689E-05	4,2252E-04	3,9722E-14	1,5280E-14	1,0984E-14
Tiempo de ejecución	10,2833	521,2471	2092,8843	0,0321	0,7191	4,4457
Cantidad de iteraciones	43177	316471	500001	2	2	2

Tabla 3.1: Comparación de la biblioteca con hibridación

3.5. Paralelización de la biblioteca en el PIAVS

A pesar de que la variante con hibridación obtuvo los mejores resultados, sus tiempos de ejecución fueron elevados en los problemas donde la dimensión era superior de 30. En esta sección se expondrán las variantes de paralelización de los métodos de la PSO en entornos paralelos distribuidos.

3.5.1. Modelo Maestro-Esclavo

En el algoritmo paralelo inicialmente el procesador Maestro crea la nube y en el método `bool Evaluate()` de la subnube es donde se distribuyen las tareas. A cada esclavo, incluyendo el procesador Maestro, se le asignan una cantidad de partículas que van a procesar. Para que la carga sea lo más equilibrada posible la cantidad de partículas debe ser divisible por la cantidad de procesadores. En los algoritmos 3 y 4 se formalizan los pasos que ejecuta el proceso Maestro en la versión 1 y 2 respectivamente.

Algoritmo 3 Modelo Maestro-Esclavo versión 1: Algortimo del *Master*

1: Enviar a todos mensaje CONTINUAR
2: Empaquetar y enviar a todos los parámetros
3: $amount_particles = total_particles/nprocesors$
4: **para toda** $particle$ **hacer**
5: Empaquetar $particle_i$
6: **fin para**
7: Enviar a todos las partículas que le corresponden
8: **para** $i = 1$ hasta $amount_particles$ **hacer**
9: Evaluar $particle_i$
10: **si** $particle_i_x$ es mejor que $particle_i_pBest$ **entonces**
11: $particle_i_pBest \leftarrow particle_i_x$
12: **fin si**
13: **si** $particle_i$ es mejor que $lBest$ **entonces**
14: $lBest \leftarrow particle_i$
15: **fin si**
16: Actualizar $particle_i$
17: **fin para**
18: Recibir de todos las partículas actualizadas
19: **para** $i = amount_particles + 1$ hasta $total_particles$ **hacer**
20: Desempaquetar $particle_i$
21: **si** $particle_i$ es mejor que $lBest$ **entonces**
22: $lBest \leftarrow particle_i$
23: **fin si**
24: **fin para**

Algoritmo 4 Modelo Maestro-Esclavo versión 2: Algortimo del *Master*

1: Enviar a todos mensaje CONTINUAR
2: $amount_particles = total_particles/nprocesors$
3: **para toda** $particle$ **hacer**
4: Empaquetar $particle_i_x$
5: **fin para**
6: Enviar a todos los vectores posición que le corresponden
7: **para** $i = 1$ hasta $amount_particles$ **hacer**
8: Evaluar $particle_i$
9: **si** $particle_i_x$ es mejor que $particle_i_pBest$ **entonces**
10: $particle_i_pBest \leftarrow particle_i_x$
11: **fin si**
12: **si** $particle_i$ es mejor que $lBest$ **entonces**
13: $lBest \leftarrow particle_i$
14: **fin si**
15: **fin para**
16: Recibir de todos los valores de la evaluación
17: **para** $i = amount_particles + 1$ hasta $total_particles$ **hacer**
18: **si** $valor_i$ es mejor que $particle_i_fitnesspBest$ **entonces**
19: $particle_i_fitnesspBest \leftarrow valor_i$
20: $particle_i_pBest \leftarrow particle_i_x$
21: **fin si**
22: **si** $particle_i$ es mejor que $lBest$ **entonces**
23: $lBest \leftarrow particle_i$
24: **fin si**
25: **fin para**

Los dos algoritmos para la versión Maestro-Esclavo son muy parecidos. La diferencia está en que la primera versión empaqueta la partícula completa para que los esclavos actualicen la próxima posición, mientras que la otra versión sólo le asigna a los esclavos la tarea de calcular el valor de la función objetivo, por tanto sólo hace falta que empaquete el vector posición. Las dos versiones tienen ventajas y desventajas, dependiendo del problema que se vaya a resolver se selecciona la adecuada. La versión 1 se recomienda utilizarla cuando calcular la próxima posición de la partícula es muy costoso. En los algoritmos 5 y 6 se muestran los pasos que ejecutarían los procesos esclavos.

Algoritmo 5 Modelo Maestro-Esclavo versión 1: Algortimo del *Slave*

1: **mientras** true **hacer**
2: $amount_particles = total_particles/nprocesors$
3: Recibir mensaje del proceso *Master*
4: **si** mensaje es TERMINAR **entonces**
5: **terminar**
6: **fin si**
7: Recibir del procesor *Master* los parámetros y las partículas correspondientes
8: **para** $i = 1$ hasta $amount_particles$ **hacer**
9: Desempaquetar $particle_i$
10: Evaluar $particle_i$
11: **si** $particle_i_x$ es mejor que $particle_i_pBest$ **entonces**
12: $particle_i_pBest \leftarrow particle_i_x$
13: **fin si**
14: Actualizar $particle_i$
15: Empaquetar $particle_i$
16: **fin para**
17: Enviar partículas al proceso *Master*
18: **fin mientras**

Algoritmo 6 Modelo Maestro-Esclavo versión 2: Algortimo del *Slave*

1: **mientras** true **hacer**
2: $amount_particles = total_particles/nprocesors$
3: Recibir mensaje del proceso *Master*
4: **si** mensaje es TERMINAR **entonces**
5: **terminar**
6: **fin si**
7: Recibir del procesor *Master* los vectores posición correspondientes
8: **para** $i = 1$ hasta $amount_particles$ **hacer**
9: Desempaquetar $particle_i_x$
10: Evaluar $particle_i$
11: **fin para**
12: Enviar los valores de la evaluación al proceso *Master*
13: **fin mientras**

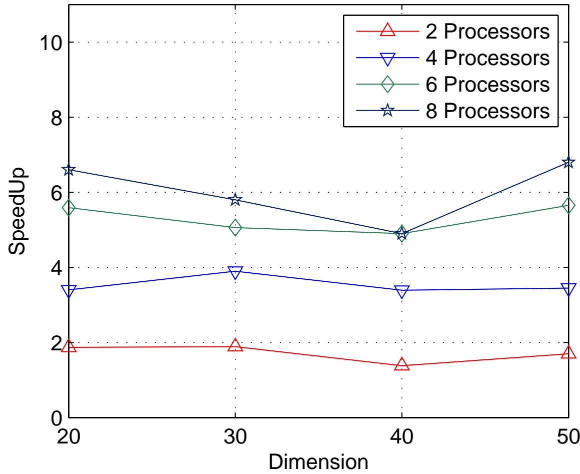
Para resolver el PIAVS con la hibridación, actualizar la próxima posición de la partícula es muy costoso porque en este método (`void UpdatePosition()`) es donde se llama al método de Newton-Bisección. Por tanto sólo haremos pruebas con el modelo Maestro-Esclavo versión 1 ya que va ser más rápida que la versión 2. Hay que tener en cuenta dos aspectos importantes en la versión 1, lo primero es que se van a actualizar las partículas de forma asincrónica y no va a dar la misma solución que el secuencial, como ocurre en la versión 2. Ésto se debe a que en la ecuación para actualizar el vector velocidad [1.5] se utilizan números aleatorios por tanto no van a coincidir estos números en las diferentes PC's. Si se quieren obtener los mismos resultados se pueden fijar los números aleatorios.

Es importante señalar que para todas las pruebas siguientes, la cantidad de partículas será ocho veces el tamaño de la dimensión, el fitness tendrá valor 0,0001 y la cantidad máxima de iteraciones será igual a 1000. Primeramente se realizará unas pruebas donde en la fórmula de la velocidad [1.5] se sustituyeron los números aleatorios por el valor 0.2 para que de esta forma se obtuvieran los mismos resultados y así comprobar la eficiencia y ganancia de velocidad de la biblioteca. Los resultados se encuentran en la siguiente tabla:

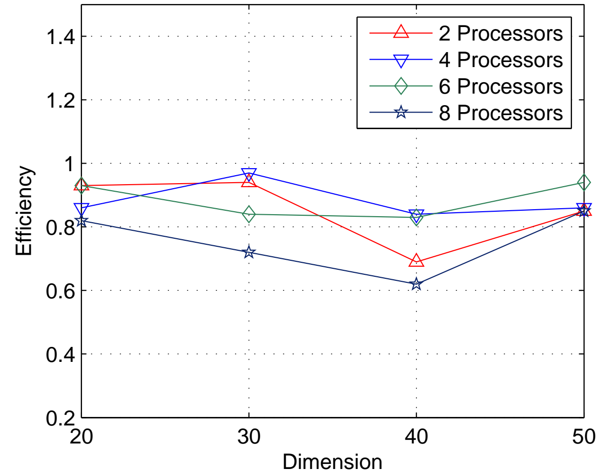
Procesadores	Vector	Dimensión	Iteraciones	Tiempo	SpeedUp	Eficiencia
1	c1x20	20	2	45,98	1,00	1,00
1	c1x30	30	2	185,72	1,00	1,00
1	c1x40	40	18	7265,20	1,00	1,00
1	c1x50	50	34	37488,90	1,00	1,00
2	c1x20	20	2	23,56	1,95	0,98
2	c1x30	30	2	95,60	1,94	0,97
2	c1x40	40	18	3633,40	1,99	0,99
2	c1x50	50	34	18862,53	1,99	0.99
4	c1x20	20	2	12,88	3,57	0,89
4	c1x30	30	2	49,94	3,72	0,93
4	c1x40	40	18	1851,22	3,92	0,98
4	c1x50	50	34	9572,82	3,92	0,98
8	c1x20	20	2	7,42	6,20	0,77
8	c1x30	30	2	26,45	7,02	0,88
8	c1x40	40	18	974,38	7,46	0,93
8	c1x50	50	34	5095,30	7,36	0,92

Tabla 3.2: Resumen de la paralelización Maestro-Esclavo

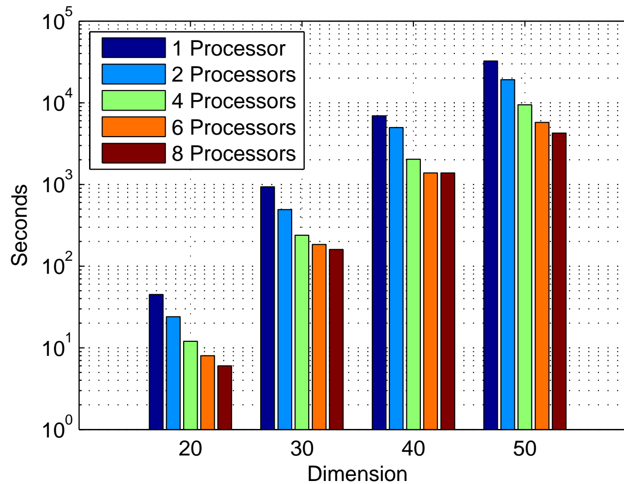
Como se pudo comprobar, con el modelo Maestro-Esclavo se obtuvieron excelentes resultados en cuanto a ganancia de velocidad y eficiencia, factores muy importantes en la programación paralela. Sabiendo que con la implementación se obtienen buenos resultados, se analizará cómo se comporta cuando se utiliza la aleatoriedad. En las siguientes gráficas se resumen las pruebas realizadas:



(a) Gráfica de ganancia de velocidad



(b) Gráfica de eficiencia



(c) Gráfica de tiempo de ejecución

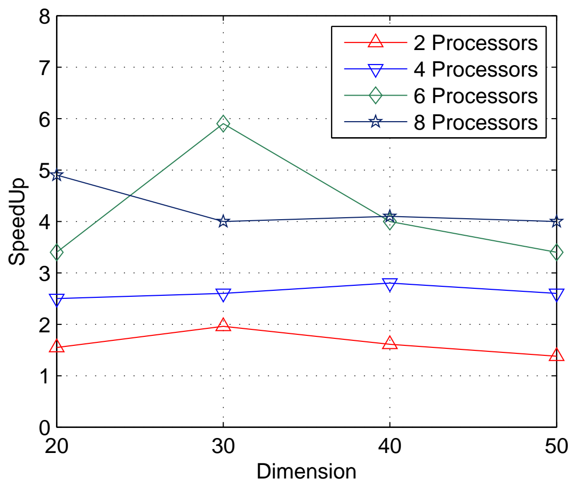
Figura 3.5: Rendimiento en el Modelo Maestro-Esclavo

3.5.2. Modelo de Islas

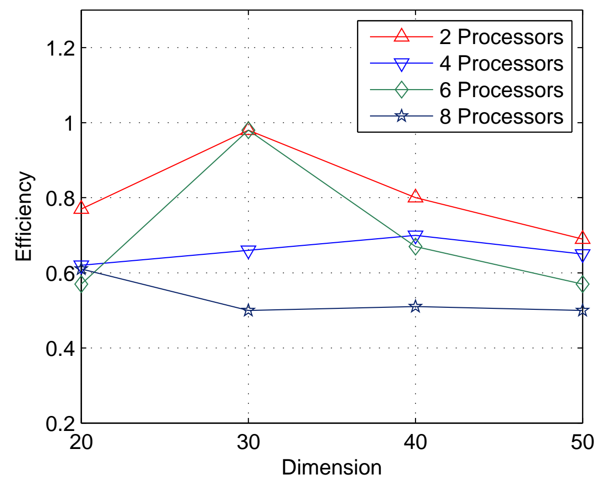
En el algoritmo paralelo cada procesador va a crear su propia nube por tanto no va a existir el concepto de población global como en el caso de Maestro-Esclavo y cada proceso se ejecuta de forma secuencial. Se debe tener en cuenta que cada cierta periodicidad deben migrar partículas de una nube a otra, ya que cada nube va ser independiente de las demás y si migran partículas entre ellas existe menos probabilidad de que queden estancadas en óptimos locales.

Para la migración de las partículas se implementó la variante de comunicación en anillo con el modelo síncrono. Debe especificarse la cantidad de partículas que van a migrar y con la frecuencia que se realizan las migraciones. También se debe definir si las partículas que van a migrar se escogen de forma aleatoria o van a ser las mejores, para esta última opción se realiza el ordenamiento de la lista de partículas de la clase `PSO_SSwarm` y las partículas que migran van a remplazar a las partículas peores adaptadas de la nube. Para el ordenamiento de las partículas se utilizó la propuesta por la biblioteca STL y su complejidad es aproximadamente $\Theta(N \log N)$, donde N sería la cantidad de elementos de la estructura.

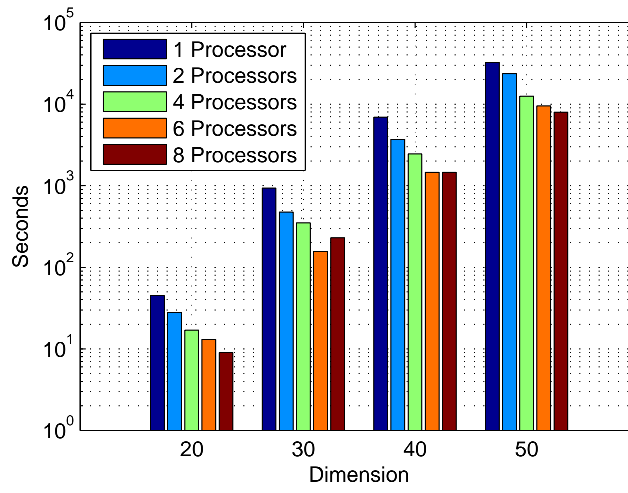
Para estas pruebas se utilizaron los mismos parámetros que el modelo Maestro-Esclavo. Cada cinco iteraciones van a migrar las partículas, migrará un solo individuo y siempre será el mejor. En las siguientes tablas se resumen las pruebas realizadas a la biblioteca:



(a) Gráfica de ganancia de velocidad



(b) Gráfica de eficiencia



(c) Gráfica de tiempo de ejecución

Figura 3.6: Rendimiento en el Modelo de Islas

Conclusiones

En el presente trabajo de diploma se realizó el diseño e implementación de una biblioteca paralela para la Optimización por Nube de Partículas, la que podrá ser aplicada a gran variedad problemas de optimización. Para ello se ha realizado un estudio profundo y detallado de la Optimización por Nube de Partículas, de sus principales variantes y los modelos de paralelización. La biblioteca fue aplicada a un problema altamente costoso como es el Problema Inverso Aditivo de Valores Singulares, y ha demostrado que es eficiente en la resolución de problemas computacionalmente costosos.

Con el modelo Maestro-Esclavo la biblioteca alcanzó mejores resultados en cuanto a ganancia de velocidad y eficiencia, comparado con el modelo de Islas. En el Problema Inverso Aditivo de Valores Singulares se realizó una hibridación de la Optimización por Nube de Partículas con el método de Newton-Bisección, donde la Optimización por Nube de Partículas se encarga de realizar la exploración del espacio de búsqueda mientras que el otro algoritmo, al ser un optimizador local realiza la explotación lográndose así una convergencia más rápida y mejores resultados.

Recomendaciones

1. Implementar algoritmos paralelos en arquitecturas con memoria compartida y para Unidad de Procesamiento Gráfico o *Graphics Processing Unit* (GPU).

2. Investigar variantes de paralelismo asíncrono.

3. Aplicar la biblioteca a otros problemas de optimización.

Referencias bibliográficas

[1] Kelley CT. Iterative Methods for Optimization. Society for Industrial Mathematics; 1987.

[2] Glover F, Kochenberger G. Handbook of Metaheuristics. Norwell: Kluwer Academic Publishers; 2002.

[3] Kennedy J, Eberhart R. Particle Swarm Optimization. In: Proceedings oi the 1995 IEEE International Conference on Neural Networks. Piscataway, New Jersey; 1995. p. 1942–1948.

[4] Kennedy J, Eberhart R, Shi Y. Swarm Intelligence. San Francisco, California: Morgan Kaufmann Publisher; 2001.

[5] Xiaohui H, Eberhart R, Shi Y. Swarm Intelligence for permutation optimization. In: Proceedings of the IEEE Swarm Intelligence Symposium. Indianapolis. USA; 2003. p. 243–246.

[6] Luna EH. Diseño de circuitos lógicos combinatorios usando optimización mediante cúmulos de partículas. Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional. México; 2004.

[7] Cotta C, Mendez A, Franca P, Moscato P. Applying Memetic Algoritms to the Analysis of Microarray Data. In: Proceedings of the Applications of Evolutionary Computing. vol. 2611. Berlin; 2003. p. 22–32.

[8] Gondim P. Genetic Algorithms and the Location Area Partitioning Problem in Cellular Networks. In: Proceedings of IEEE the 46th Vehicular Technology Conference; 1996. p. 1835–1841.

[9] Subrata R, Zomaya AY. Evolving Cellular Automata for Location Management in Mobile Computing Networks. In: IEEE Trans. Parallel Distrib. Syst.; 2003. p. 13–23.

[10] Xie X, Zhang W, Yang Z. Solving Numerical Optimization Problems by Simulating Particulates in Potential Field with Cooperative Agents; 2002.

[11] Omran M, Salman A, Engelbrecht A. Image Classification Using Particle Swarm Optimization. In: Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution and Learning; 2002. p. 370–374.

[12] Parsopoulos KE, Papageorgiou E, Groumpos P, Vrahatis MN. A First Study of Fuzzy Cognitive Maps Learning Using Particle Swarm Optimization. In: Proceedings of the IEEE Congress on Evolutionary Computation; 2003. .

[13] Ourique C, Biscaia E, Pinto J. The Use of Particle Swarm Optimization for Dynamical Analysis in Chemical Processes. In: Proceedings of the Computers and Chemical Engineeringges. vol. 26; 2002. p. 1783–1793.

[14] Nieto JMG. Algoritmos Basados en Cúmulos de Partículas Para la Resolución de Problemas Complejos; 2006.

[15] Nocedal J, Wright SJ. Numerical optimization. New York: Springer Publisher; 1999.

[16] Glover F. Future Paths for Integer Programming and Links to Artificial Intelligence. Computers & Operations Research. 1986;p. 533–549.

[17] Blum C, Roli A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Computing Surveys. 2003;p. 268–308.

[18] Resende M, González-Verlarde J. GRASP: Procedimientos de búsqueda u miopes aleatorizados y adaptativos. In: Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial; 2003. .

[19] Crainicand T, Toulouse M. Parallel Strategies for Metaheuristics. In: Handbook of Metaheuristics. Kluwer Academic Publisher; 2003. p. 475–513.

[20] Kirkpatrick S, Gelatt C, Vecchi M. Optimization by Simulated Annealing. Science. 1983;p. 671–680.

[21] Van Laarhoven PJM, Aarts EHL. Simulated Annealing: Theory and Applications. Kluwer Academic Press; 1987.

[22] Dowsland K, Díaz BA. Diseño de heurísticas y fundamentos del recocido simulado. Revista Ibeoramiericana de Inteligencia Artificial. 2003;.

[23] Glover F, Laguna M. Tabu Search. Kluwer Academic Publishers; 1997.

[24] Mladenovic N, Hansen P. Variable Neighborhood Search. Computers Oper Res,. 1997;p. 1097–1100.

[25] Hansen P, Mladenovic N. Variable Neighborhood Search: Principle and Applications. European Journal of Operational Research. 2001;p. 449–467.

[26] Haupt RL, Haupt SE. Practical Genetic Algorithms. A John Wiley & Sons; 2004.

[27] Bäck T, Fogel D, Michalewicz Z. Handbook of Evolutionary Computation. New York and Bristol (UK): IOP Publishing and Oxford University Press; 1997.

[28] Davis L. Handbook of Genetic Algorithm; 1991.

[29] Laguna M, Martí R. Scatter Search Methodology and Implementations in C. Kluwer Academic Publishers; 2002.

[30] Dorigo M. Optimization, Learning and Natural Algorithms. Politécnico di Milano; 1992.

[31] Dorigo M. The Ant Colony Optimization Metaheuristic: Algorithms, Applications and Advances. Universidad Libre de Bruxelles; 2000.

[32] Kennedy J, Eberhart R. A Discrete Binary Version of the Particle Swarm Algorithm. In: IEEE International Conference on Systems. vol. 5; 1997. p. 4104–4109.

[33] Shi Y, Eberhart R. A modified particle swarm optimizer. In: IEEE Congress on Evolutionary Computation. Piscataway, USA; 1998. p. 69–73.

[34] Eberhart R, Shi Y. Comparing inertia weights and constriction factors in particle swarm optimization. In: IEEE Congress on Evolutionary Computation. vol. 1. San Diego; 2000. p. 84–88.

[35] Kennedy J, Clerc M. The particle swarm–explosion, stability and convergence in a multidimensional complex space. IEEE Transactions on Evolutionary Computation. 2002;.

[36] Eberhart R, Simpson P, Dobbins R. 6. In: Computational Intelligence PC Tools. Academic Press Professional; 1996. p. 212–226.

[37] Kennedy J. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In: IEEE Swarm Intell. Symp.. vol. 3; 1999. p. 1931–1938.

[38] Kennedy J, Mendes R. Neighborhood Topologies in Fully Informed and Best-of-Neighborhood Particle Swarms. In: Man and Cybernetics. IEEE Transactions on Systems; 2006. p. 515–519.

[39] El-Dib A, Youssef H, El-Metwally M, Osman Z. Load flow solution using hybrid particle swarm optimization. In: Proc. Int. Conf. Elect., Electron., Comput. Eng.; 2004. p. 742–746.

[40] Naka S, Genji T, Yura T, Fukuyama Y. A hybrid particle swarm optimization for distribution state estimation. In: IEEE Trans. Power System; 2003. p. 60–68.

[41] Eberhart R, Shi Y. Fuzzy adaptive particle swarm optimization. In: Proceedings of the IEEE Congress on Evolutionary Computation. vol. 1; 2001. p. 101–106.

[42] Parsopoulos KE, Vrahatis MN. Particle swarm optimization method in multiobjective problems. In: Proc. ACM Symp. Appl. Comput.; 2002. p. 603–607.

[43] Secrest B, Lamont G. Visualizing particle swarm optimization - Gaussian particle swarm optimization. In: Proceedings of the IEEE Swarm Intelligence Symposium; 2003. p. 198–203.

[44] Krohling R. Gaussian swarm: A novel particle swarm optimization algorithm. In: Proc. IEEE Conf. Cybern. Intell. Syst.. vol. 1; 2004. p. 372–376.

[45] Parsopoulos KE, Vrahatis MN. Recent approaches to global optimization problems through particle swarm optimization. In: Natural Computing. vol. 1; 2002. p. 235–306.

[46] Vidal A. Presente y futuro de la Computación Paralela; 2000.

[47] Almasi GS, Gottlieb A. Highly Parallel Computing. Redwood City, CA: Benjamin-Cummings Publishers; 1989.

[48] Kumar V, Gramar A, Grupta A, Kerypis G. Introduction to Parallel Computing, Second Edition. Addison Wesley; 2003.

[49] Flynn MJ. Some Computer Organizations and Their Effectiveness. IEEE Trans Comput. 1972;C-21.

[50] Stender J. Parallel Genetic Algorithm: Theory and Applications. IOS Press; 1993.

[51] Pit LV. Parallel genetic algorithms. Leiden University; 1995.

[52] Alba E, group group group group M. Mallba: A Library of Skeletons for Combinatorial Optimisation. In: Proceedings of the Euro-Par; 2002. p. 927–932.

[53] Oualline S. Practical C++ Programming. O'Reilly; 2002.

[54] Alba E. Netstream: A Flexible and Simple Message Passing Service for LAN/WAN Utilization. ETSII. 2001;.

[55] Guerrero F. Algoritmos para el Problema de la Asignación de Frecuencias a Enlaces de Teléfonos Móviles. Universidad de Málaga. 2001;.

[56] Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. MPI: The Complete Reference. MIT Press; 1996.

[57] Pacheco P. Parallel Programming With MPI. Morgan Kaufmann; 1996.

[58] Hammarling S, Dongarra J, Croz JD, Hason RJ. An extended set of fortran basic linear algebra subroutines; 1998.

[59] Anderson E, Bai Z, Bischof C, Demmel J, Dongarra J, Croz JD. LAPACK User Guide; 1995.

[60] Eberhart R, Shi Y. Empirical study of particle swarm optimization. Evolutionary Programming VII. 1998;p. 591–600.

[61] Benchmark functions;. Available from: http://www.cil.pku.edu.cn/resources/benchmark_pso/.

[62] Biswas A, Dasgupta S, Das S, Abraham A. Synergy of PSO and Bacterial Foraging Optimization: A Comparative Study on. In: Numerical Benchmarks, Second International Symposium on Hybrid Artificial Intelligent Systems (HAIS 2007), Advances in Soft computing Series. Springer Verlag; 2007. p. 255–263.

[63] Standard PSO 2006;. Available from: http://www.particleswarm.info/Programs.html.

[64] Chu MT. Numerical Methods for Inverse Singular Value Problems. SIAM, Journal Numerical Analysis. 1991;29(3):885–903.

[65] Flores G, García VM, Vidal AM. Numerical Experiments on the solution of the Inverse Additive Singular Value Problem.. Atlanta, GA, USA: Computational Science - 5th International Conference; 2005.

[66] Holland J. Adaptation in Natural and Artificial Systems. University of Michigan Press; 1975.

[67] Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical Recipes in C. Second edition ed. Cambridge University Press; 1992.

Apéndice A

Pruebas de la biblioteca en el PIAVS

Vector	Dimensión	Población	Iteraciones	Fitness	Error relativo	Tiempo (seg)
c2x40	5	20	1549	7,1609E-12	2,6203E-13	0,5490
c2x40	5	20	1562	8,0749E-12	2,9548E-13	0,5521
m10x640	5	20	1481	8,9978E-12	3,6857E-13	0,5556
c3x50	5	20	1552	7,2119E-12	3,2937E-13	0,5586
c2x30	5	20	1611	9,9229E-12	3,2088E-13	0,5767
c2x40	5	20	1629	9,9974E-12	3,6582E-13	0,5806
m50x640	5	20	1542	9,5115E-12	6,4463E-13	0,5820
c1x50	5	20	1618	9,9909E-12	4,1391E-13	0,5868
c2x30	5	20	1688	8,3811E-12	2,7102E-13	0,6149
c3x50	5	20	1692	9,9569E-12	4,5474E-13	0,6156
m10x640	5	20	1669	9,7782E-12	4,0053E-13	0,6236
c1x40	5	20	1662	8,0465E-12	4,2573E-13	0,6258
c1x50	5	20	1805	9,2317E-12	3,8246E-13	0,6494
m20x640	5	20	1941	8,3683E-12	5,1372E-13	0,6598
m50x640	5	20	1760	7,9937E-12	5,4176E-13	0,6612
c3x50	5	20	1874	9,8668E-12	4,5062E-13	0,6760
c3x30	5	20	2048	8,4445E-12	4,1374E-13	0,7027
c1x40	5	20	1891	9,9711E-12	5,2756E-13	0,7037
m50x640	5	20	1912	8,8106E-12	5,9713E-13	0,7160
c2x50	5	20	2125	9,2650E-12	3,3220E-13	0,7554
m40x640	5	20	2289	9,8175E-12	6,8891E-13	0,7570
m20x640	5	20	2234	9,9354E-12	6,0993E-13	0,7592
c1x40	5	20	2079	8,8589E-12	4,6871E-13	0,7766
c1x20	5	20	2254	9,6830E-12	3,6499E-13	0,7994
c3x30	5	20	2449	9,2815E-12	4,5475E-13	0,8352
m10x640	5	20	2288	9,2711E-12	3,7976E-13	0,8506
c1x20	5	20	2392	9,9865E-12	3,7643E-13	0,8580
c1x50	5	20	2552	7,0639E-12	2,9265E-13	0,9199
m30x640	5	20	2734	9,9001E-12	5,0752E-13	0,9239
c2x50	5	20	2668	9,9056E-12	3,5517E-13	0,9426
c1x20	5	20	2828	9,4534E-12	3,5634E-13	1,0018
c3x10	5	20	2935	9,7731E-12	4,2366E-13	1,0277
c3x10	5	20	2964	8,6449E-12	3,7476E-13	1,0289
c3x10	5	20	3328	9,9198E-12	4,3002E-13	1,1730

m40x640	5	20	3757	9,2693E-12	6,5044E-13	1,2298
c2x10	5	20	4077	9,9966E-12	4,8919E-13	1,2999
m20x640	5	20	4387	9,8618E-12	6,0541E-13	1,4726
m30x640	5	20	4455	9,8799E-12	5,0648E-13	1,4991
c3x30	5	20	4922	9,5854E-12	4,6964E-13	1,6938
c2x10	5	20	6537	9,9941E-12	4,8907E-13	2,1026
c2x50	5	20	6391	9,9687E-12	3,5744E-13	2,2967
c2x10	5	20	8436	9,9823E-12	4,8849E-13	2,7324
c3x40	5	20	10887	9,6685E-12	4,0401E-13	3,7545
m40x640	5	20	12585	9,4951E-12	6,6629E-13	4,1181
c3x20	5	20	15352	9,7239E-12	3,7099E-13	5,2547
c3x20	5	20	15977	9,9898E-12	3,8113E-13	5,3437
c3x40	5	20	18314	9,7977E-12	4,0941E-13	6,2608
c3x40	5	20	19960	9,8398E-12	4,1116E-13	6,9082
c2x30	5	20	25578	9,5920E-12	3,1018E-13	9,5166
m30x640	5	20	27952	9,7209E-12	4,9833E-13	9,5202
c3x20	5	20	27851	9,9927E-12	3,8124E-13	9,5717
c1x30	5	20	101699	9,9985E-12	4,1693E-13	32,5551
c2x20	5	20	173772	9,9321E-12	4,8384E-13	58,7182
c2x20	5	20	240957	9,9986E-12	4,8708E-13	81,7028
c1x30	5	20	356257	9,9974E-12	4,1688E-13	116,7479
c2x20	5	20	445522	9,9996E-12	4,8713E-13	147,4498
c1x30	5	20	500001	2,7651E-10	1,1530E-11	162,3187
c1x10	5	20	500001	5,6864E-04	2,4533E-05	178,4077
c1x10	5	20	500001	2,4988E-05	1,0781E-06	249,2119
c1x10	5	20	500001	1,5933E-05	6,8740E-07	285,9258
Promedio			60104	1,0159E-05	4,3831E-07	23,5469
Desviación			142090	7,2806E-05	3,1410E-06	59,9747

Tabla A.1: Pruebas secuenciales a la biblioteca (5)

Vector	Dimensión	Población	Iteraciones	Fitness	Error relativo	Tiempo (seg)
c1x20	10	40	11444	9,9978E-12	2,2091E-13	18,7853
c1x20	10	40	11636	9,5432E-12	2,1086E-13	19,0424
m10x640	10	40	15825	9,9025E-12	2,2267E-13	25,4883
c1x10	10	40	17364	9,9118E-12	2,2880E-13	27,3183
c2x40	10	40	28306	9,9966E-12	1,9845E-13	45,9456
c2x30	10	40	29204	9,9622E-12	1,7795E-13	46,5280
c2x30	10	40	31364	9,9930E-12	1,7850E-13	50,3698
m50x640	10	40	32070	9,9362E-12	3,2789E-13	50,8607
c3x40	10	40	43133	9,9778E-12	2,2903E-13	68,0039
c3x50	10	40	44608	8,9725E-12	2,2523E-13	71,6909
c1x20	10	40	45423	9,9935E-12	2,2081E-13	74,6510
m10x640	10	40	45671	9,7937E-12	2,2022E-13	76,0333
c2x30	10	40	50742	9,9947E-12	1,7853E-13	80,5655
m50x640	10	40	52304	9,8629E-12	3,2546E-13	83,9209
m10x640	10	40	55063	9,9561E-12	2,2387E-13	90,5573
c3x40	10	40	57957	9,9561E-12	2,2853E-13	92,0349
c2x40	10	40	62667	9,9262E-12	1,9705E-13	103,4214
m50x640	10	40	81679	9,9519E-12	3,2840E-13	130,9445
c1x40	10	40	112969	9,9887E-12	3,4829E-13	199,8605
c2x40	10	40	150386	9,9953E-12	1,9842E-13	244,1404
c1x40	10	40	162887	9,9976E-12	3,4859E-13	284,9917
m30x640	10	40	193278	9,9990E-12	2,7601E-13	307,1729
c3x50	10	40	188704	9,9296E-12	2,4926E-13	309,4714
m30x640	10	40	209195	9,9599E-12	2,7492E-13	332,7541
c1x50	10	40	209966	9,9878E-12	2,1139E-13	350,6653
c1x40	10	40	234807	9,9748E-12	3,4780E-13	416,4260
c1x50	10	40	309586	9,9991E-12	2,1163E-13	518,8384
m40x640	10	40	500001	1,0586E-02	2,7419E-04	725,1358
m40x640	10	40	500001	7,3292E-06	1,8984E-07	725,5170
m40x640	10	40	500001	6,2425E-03	1,6169E-04	737,2993
c2x10	10	40	500001	5,7968E-03	1,3351E-04	765,3151
c2x20	10	40	500001	2,0710E-03	4,8629E-05	765,5134
c2x10	10	40	500001	1,2497E-03	2,8782E-05	779,9978
c3x10	10	40	500001	1,9383E-10	4,6110E-12	780,2643
c2x20	10	40	500001	2,0693E-03	4,8590E-05	780,3186
c2x20	10	40	500001	2,5803E-03	6,0588E-05	782,6275
c2x10	10	40	500001	3,5788E-03	8,2426E-05	785,5347
c3x10	10	40	500001	2,3421E-10	5,5715E-12	786,2142

c1x30	10	40	500001	3,4196E-05	7,3626E-07	790,0302
m20x640	10	40	500001	5,3491E-08	1,5629E-09	793,4133
m20x640	10	40	500001	9,5713E-05	2,7966E-06	794,1470
c3x10	10	40	500001	3,4883E-11	8,2983E-13	795,1242
c1x30	10	40	500001	3,9626E-05	8,5317E-07	797,5444
c1x30	10	40	500001	3,9590E-05	8,5239E-07	800,6201
m20x640	10	40	500001	1,4825E-11	4,3317E-13	801,5965
c1x10	10	40	500001	2,5826E-06	5,9616E-08	807,1666
m30x640	10	40	500001	2,8901E-11	7,9776E-13	810,6320
c1x10	10	40	500001	1,2074E-01	2,7871E-03	812,8286
c3x40	10	40	500001	1,1087E-10	2,5450E-12	817,7188
c3x30	10	40	500001	2,9332E-08	7,4560E-10	823,3709
c3x50	10	40	500001	5,9526E-07	1,4943E-08	828,6250
c2x50	10	40	500001	2,2109E-02	4,5765E-04	830,4228
c3x30	10	40	500001	7,7620E-05	1,9730E-06	831,9725
c3x30	10	40	500001	1,4894E-06	3,7859E-08	832,8567
c1x50	10	40	500001	3,2396E-11	6,8568E-13	845,9164
c2x50	10	40	500001	5,7848E-05	1,1974E-06	847,2795
c3x20	10	40	500001	5,4536E-04	1,2029E-05	854,5448
c2x50	10	40	500001	1,8573E-05	3,8445E-07	855,9356
c3x20	10	40	500001	1,5564E-03	3,4328E-05	857,7224
c3x20	10	40	500001	9,7850E-03	2,1582E-04	864,7317
Promedio			316471	3,1547E-03	7,2574E-05	510,4737
Desviación			210130	1,5696E-02	3,6158E-04	337,5928

Tabla A.2: Pruebas secuenciales a la biblioteca (10)

Apéndice B

Pruebas de la biblioteca híbrida en el PIAVS

Vector	Dimensión	Población	Iteraciones	Fitness	Error relativo	Tiempo (seg)
c2x50	5	20	2	4,3977E-15	1,5768E-16	0,0127
c3x20	5	20	2	2,7287E-13	1,0411E-14	0,0130
c1x30	5	20	2	5,9957E-13	2,5001E-14	0,0143
c1x30	5	20	2	2,3110E-13	9,6366E-15	0,0158
c3x30	5	20	2	2,2204E-15	1,0879E-16	0,0167
m20x640	5	20	2	4,1968E-14	2,5764E-15	0,0175
c3x20	5	20	2	3,8607E-14	1,4729E-15	0,0186
m20x640	5	20	2	3,6418E-15	2,2357E-16	0,0197
c1x10	5	20	2	7,6404E-15	3,2963E-16	0,0221
m40x640	5	20	2	2,1791E-15	1,5291E-16	0,0233
c3x20	5	20	2	4,8806E-15	1,8621E-16	0,0234
c2x50	5	20	2	5,0683E-15	1,8173E-16	0,0239
c2x10	5	20	2	9,8580E-13	4,8240E-14	0,0245
c1x30	5	20	2	3,4544E-14	1,4404E-15	0,0248
c2x10	5	20	2	1,4026E-12	6,8639E-14	0,0253
c1x50	5	20	2	1,7938E-12	7,4313E-14	0,0256
c3x50	5	20	2	2,4099E-12	1,1006E-13	0,0259
c2x10	5	20	2	9,4568E-15	4,6277E-16	0,0264
c3x50	5	20	2	6,8367E-15	3,1223E-16	0,0270
c3x40	5	20	2	3,4293E-12	1,4330E-13	0,0273
m30x640	5	20	2	6,4851E-15	3,3245E-16	0,0274
c3x50	5	20	2	1,8182E-13	8,3040E-15	0,0280
m10x640	5	20	2	4,5288E-15	1,8551E-16	0,0282
c1x20	5	20	2	9,2977E-12	3,5047E-13	0,0284
m20x640	5	20	2	4,1414E-12	2,5424E-13	0,0287
m30x640	5	20	2	3,6298E-12	1,8608E-13	0,0288
c1x40	5	20	2	4,2826E-15	2,2659E-16	0,0290
c2x50	5	20	2	8,5545E-12	3,0673E-13	0,0304
c1x20	5	20	2	3,9721E-15	1,4972E-16	0,0311
c3x30	5	20	2	5,3395E-15	2,6161E-16	0,0312
m30x640	5	20	2	1,8971E-15	9,7255E-17	0,0312
m10x640	5	20	2	8,3319E-15	3,4129E-16	0,0312
c2x30	5	20	2	2,8436E-15	9,1953E-17	0,0316
c1x20	5	20	2	2,1629E-14	8,1527E-16	0,0321

m40x640	5	20	2	1,3518E-15	9,4857E-17	0,0323
m40x640	5	20	2	3,6014E-15	2,5271E-16	0,0329
m50x640	5	20	2	1,6012E-15	1,0852E-16	0,0335
m50x640	5	20	2	2,5864E-12	1,7529E-13	0,0341
c1x40	5	20	2	1,9322E-14	1,0223E-15	0,0349
c1x50	5	20	2	1,2184E-13	5,0475E-15	0,0356
c3x10	5	20	2	8,3144E-14	3,6043E-15	0,0370
c3x10	5	20	2	6,7442E-13	2,9236E-14	0,0370
c3x10	5	20	2	4,5193E-15	1,9591E-16	0,0372
c1x40	5	20	2	1,3323E-14	7,0489E-16	0,0376
c3x40	5	20	2	1,3866E-13	5,7940E-15	0,0377
c2x40	5	20	2	1,7311E-12	6,3345E-14	0,0379
c2x20	5	20	2	3,2102E-15	1,5638E-16	0,0382
c2x40	5	20	2	4,0701E-15	1,4893E-16	0,0384
c3x30	5	20	2	3,3894E-15	1,6606E-16	0,0398
c2x20	5	20	2	5,3291E-15	2,5961E-16	0,0406
c2x30	5	20	2	3,5316E-14	1,1420E-15	0,0431
m50x640	5	20	2	2,7013E-15	1,8308E-16	0,0456
c3x40	5	20	2	3,2024E-15	1,3381E-16	0,0458
c1x50	5	20	2	5,0928E-12	2,1099E-13	0,0473
c2x30	5	20	2	2,2181E-13	7,1728E-15	0,0561
c2x20	5	20	2	4,1453E-15	2,0194E-16	0,0568
c2x40	5	20	2	4,4409E-15	1,6250E-16	0,0601
m10x640	5	20	3	1,6208E-14	6,6390E-16	0,0716
c1x10	5	20	5	9,1317E-12	3,9397E-13	0,0926
c1x10	5	20	6	6,2371E-12	2,6909E-13	0,1299
Promedio			2,1333	1,0549E-12	4,6244E-14	0,0346
Desviación			6,4464E-01	2,2579E-12	9,5029E-14	0,0184

Tabla B.1: Pruebas secuenciales a la biblioteca hídrida (5)

Vector	Dimensión	Población	Iteraciones	Fitness	Error relativo	Tiempo (seg)
c1x30	10	40	2	8,7572E-15	1,8855E-16	0,4280
c3x30	10	40	2	9,1378E-15	2,3227E-16	0,4835
c2x10	10	40	2	9,9884E-15	2,3005E-16	0,4857
c1x30	10	40	2	1,5463E-14	3,3292E-16	0,5019
m30x640	10	40	2	8,9827E-15	2,4795E-16	0,5056
c2x20	10	40	2	4,0281E-14	9,4584E-16	0,5162
c2x30	10	40	2	3,0203E-12	5,3949E-14	0,5166
m20x640	10	40	2	8,2885E-15	2,4218E-16	0,5207
m30x640	10	40	2	6,0871E-15	1,6803E-16	0,5308
c2x10	10	40	2	8,9447E-12	2,0601E-13	0,5446
c1x50	10	40	2	9,3160E-15	1,9718E-16	0,5542
c3x40	10	40	2	3,0578E-13	7,0189E-15	0,5662
m10x640	10	40	2	1,6503E-14	3,7109E-16	0,5704
m30x640	10	40	2	4,4511E-13	1,2287E-14	0,5811
m40x640	10	40	2	1,0115E-14	2,6199E-16	0,5838
c3x20	10	40	2	1,0362E-14	2,2854E-16	0,5860
c2x20	10	40	2	5,0860E-13	1,1942E-14	0,5961
c1x40	10	40	2	1,6109E-13	5,6169E-15	0,6039
c2x40	10	40	2	2,1575E-13	4,2830E-15	0,6176
c3x10	10	40	2	3,6969E-13	8,7945E-15	0,6215
c3x30	10	40	2	8,4651E-15	2,1517E-16	0,6216
c2x30	10	40	2	1,1307E-14	2,0197E-16	0,6233
c3x40	10	40	2	4,9364E-15	1,1331E-16	0,6317
c2x30	10	40	2	2,4220E-12	4,3263E-14	0,6391
c1x30	10	40	2	7,2581E-15	1,5627E-16	0,6395
c1x50	10	40	2	1,0267E-14	2,1730E-16	0,6403
c2x20	10	40	2	9,9177E-15	2,3288E-16	0,6475
c3x10	10	40	2	8,5446E-15	2,0327E-16	0,6509
c3x40	10	40	2	1,4470E-14	3,3214E-16	0,6539
m50x640	10	40	2	4,5402E-14	1,4982E-15	0,6567
c3x10	10	40	2	9,2146E-15	2,1921E-16	0,6705
m40x640	10	40	2	1,7678E-12	4,5790E-14	0,6706
c1x20	10	40	2	3,0500E-14	6,7393E-16	0,6737
c2x50	10	40	2	5,1379E-12	1,0635E-13	0,6958
m10x640	10	40	2	9,6583E-15	2,1718E-16	0,6998
m40x640	10	40	2	7,4212E-15	1,9222E-16	0,7014
c2x50	10	40	2	1,2468E-14	2,5809E-16	0,7117
c3x50	10	40	2	9,1578E-15	2,2989E-16	0,7308

c3x50	10	40	2	1,1733E-14	2,9452E-16	0,7373
c1x40	10	40	2	7,8135E-15	2,7244E-16	0,7694
c2x40	10	40	2	8,6569E-15	1,7185E-16	0,7792
c3x30	10	40	2	7,3375E-14	1,8651E-15	0,7919
c2x40	10	40	2	7,0776E-15	1,4050E-16	0,7969
m20x640	10	40	2	3,6519E-15	1,0670E-16	0,8069
c3x20	10	40	2	1,0779E-14	2,3775E-16	0,8152
c1x20	10	40	2	7,9689E-15	1,7608E-16	0,8259
m50x640	10	40	2	7,0637E-15	2,3309E-16	0,8431
c2x10	10	40	2	1,6956E-12	3,9054E-14	0,8555
c3x50	10	40	2	5,7432E-15	1,4417E-16	0,9318
m50x640	10	40	2	9,6097E-15	3,1711E-16	1,0140
c1x20	10	40	2	9,8303E-15	2,1721E-16	1,0288
c2x50	10	40	2	9,1493E-12	1,8939E-13	1,0522
c3x20	10	40	2	1,7656E-14	3,8942E-16	1,0571
m20x640	10	40	2	9,5096E-15	2,7786E-16	1,0918
m10x640	10	40	2	2,1417E-12	4,8160E-14	1,1074
c1x40	10	40	2	5,6409E-15	1,9669E-16	1,1140
c1x50	10	40	3	1,1754E-14	2,4878E-16	1,2613
c1x10	10	40	13	5,3905E-12	1,2443E-13	5,4995
c1x10	10	40	17	2,6656E-12	6,1530E-14	7,2359
c1x10	10	40	34	1,6505E-12	3,8100E-14	13,9697
Promedio			2,9833	7,7587E-13	1,6998E-14	1,1210
Desviación			2,9997E+00	1,9115E-12	4,1675E-14	1,9715

Tabla B.2: Pruebas secuenciales a la biblioteca hídrida (10)

Glosario de términos

PSO Optimización por Nube de Partículas o *Particle Swarm Optimization*

SA Enfriamiento Simulado o *Simulated Annealing*

TS Búsqueda Tabú o *Tabu Search*

VNS Búsqueda en Vecindario Variable o *Variable Neighborhood Search*

ILS Búsqueda Local Iterada o *Iterated Local Search*

EA Algoritmos evolutivos o *Evolutionary Algorithms*

SS Búsqueda Dispersa o *Scatter Search*

ACO Sistemas Basados en Colonias de Hormigas o *Ant Colony Optimization*

GRASP Procedimiento de Búsqueda Miope Aleatorizado y Adaptativo o *Greedy Randomized Adaptive Search Procedure*

PIAVS Problema Inverso Aditivo de Valores Singulares o *Inverse Additive Singular Value Problem*

BLAS *Basic Linear Algebra Subprograms*

LAPACK *Linear Algebra PACKage*

MPI *Message Passing Interface*

STL *Standard Template Library*

GPU Unidad de Procesamiento Gráfico o *Graphics Processing Unit*