SUBSECRETARÍA DE EDUCACIÓN SUPERIOR
DIRECCIÓN GENERAL DE EDUCACIÓN
SUPERIOR TECNOLÓGICA
INSTITUTO TECNOLÓGICO DE LA PAZ

SECRETARÍA DE
EDUCACIÓN PÚBLICA

INSTITUTO TECNOLÓGICO DE LA PAZ

UN ALGORITMO HÍBRIDO PARA LA OPTIMIZACIÓN EN PARALELO DE PROBLEMAS CONTINUOS

TESIS PROFESIONAL

que para obtener el título de

INGENIERO EN SISTEMAS COMPUTACIONES

Presenta

JOEL ARTEMIO MORALES VISCAYA

Asesor

DR. MARCO ANTONIO CASTRO LIERA

La Paz, Baja California Sur, Agosto 2012.

SUBSECRETARÍA DE EDUCACIÓN SUPERIOR
DIRECCIÓN GENERAL DE EDUCACIÓN SUPERIOR TECNOLÓGICA
INSTITUTO TECNOLÓGICO DE LA PAZ

SECRETARÍA DE EDUCACIÓN PÚBLICA

Departamento: DIV.EST.PROF.
No. de Oficio: DEP-T-641-2012.

Asunto: Se autoriza impresión.

La Paz, B.C.S., **31 de Agosto del 2012.**

C. JOEL ARTEMIO MORALES VISCAYA,
PASANTE DE LA CARRERA DE INGENIERÍA EN SISTEMAS COMPUTACIONALES,
P R E S E N T E.

Con base en el dictamen de aprobación emitido por la Comisión Revisora del trabajo denominado **"UN ALGORITMO HÍBRIDO PARA LA OPTIMIZACIÓN EN PARALELO DE PROBLEMAS CONTÍNUOS"** entregado por usted para su análisis, le informo que se **AUTORIZA** la impresión.

A T E N T A M E N T E
"Ciencia es Verdad, Técnica es Libertad"

MC. PABLO PÉREZ ÁLVAREZ,
JEFE DE LA DIVISIÓN DE ESTUDIOS PROFESIONALES.

C.C.P.- Lic. Paola Cristina Torres Ortega.- Coord. Apoyo a la Titulación

Blvd. Forjadores de B.C.S. # 4720, C.P. 23080, Apdo. Postal 43-B,
Tels. (612) 121-04-24, Fax: (612) 121-12-95
email:direccion@itlp.edu.mx
www.itlp.edu.mx

SUBSECRETARÍA DE EDUCACIÓN SUPERIOR
DIRECCIÓN GENERAL DE EDUCACIÓN SUPERIOR TECNOLÓGICA
INSTITUTO TECNOLÓGICO DE LA PAZ

SEP
Institutos Tecnológicos

SECRETARÍA DE
EDUCACIÓN PÚBLICA

SEP

La Paz, B.C.S., 29/agosto/2012

M E M O R A N D U M

PARA: MSC. JAVIER ALBERTO CARMONA TROYO
JEFE DEL DEPTO. DE SISTEMAS Y COMPUTACIÓN

DE: Comisión revisora de trabajo de titulación.

FECHA: 29 de agosto del 2012.

ASUNTO: Autorización de impresión.

Por medio del presente le informamos que una vez revisado el trabajo de tesis denominado: **"UN ALGORITMO HÍBRIDO PARA LA OPTIMIZACIÓN EN PARALELO DE PROBLEMAS CONTÍNUOS"**, como opción de titulación **I TESIS PROFESIONAL** que presenta el pasante de la carrera de **INGENIERÍA EN SISTEMAS COMPUTACIONALES: JOEL ARTEMIO MORALES VISCAYA**, le informamos que esta comisión revisora ha dictaminado que el trabajo esta:

ACEPTADO (X) SUJETO A MODIFICACIONES ( ) RECHAZADO ( )

OBSERVACIONES:

A T E N T A M E N T E.

DR. MARCO ANTONIO CASTRO LIERA

DR. SAÚL MARTÍNEZ DÍAZ

MSC. ILIANA CASTRO LIERA

c.c.p. División de Estudios Profesionales.
c.c.p. Archivo.

Blvd. Forjadores de B.C.S. No. 4720 C.P. 23080 Apdo. Postal 43-B
Tels. (612) 12-107-05, 12-104-24, 12-104-26, Fax:(612) 12-112-95
email:direccion@itlp.edu.mx
www.itlp.edu.mx

Índice Temático

1. Introducción

1.1 Motivación

La optimización se puede definir de manera muy general como la ciencia encargada de encontrar las mejores soluciones a los problemas, que generalmente modelan una realidad física.

En el día a día tratamos con decenas de problemas de optimización, los cuales resolvemos muchas veces sin percatarnos de que lo estamos haciendo, son problemas simples, que pueden ser resueltos de manera analítica, o con simple observación, sin embargo, entre más grande es el abanico de posibilidades y mayor la cantidad de variables a considerar, los problemas se van volviendo más complejos y es entonces que necesitamos algoritmos y herramientas, que nos ayuden a solucionarlos.

La optimización numérica ha sido ampliamente usada para resolver problemas en áreas como la economía, ingeniería de control, arquitectura, investigación de operaciones y mecatrónica, por nombrar solo algunas. Los principales obstáculos de los algoritmos existentes es que los problemas son demasiado demandantes computacionalmente, es decir requieren recursos significativamente grandes en tiempo y hardware; y que la mayoría de los problemas están plagados de óptimos locales, soluciones que parecen ser las mejores del problema entero pero sólo lo son de una parte del mismo.

El uso eficiente de los recursos computacionales es un aspecto cada vez más importante en el desarrollo de algoritmos de optimización, partícularmente en lo que respecta al aprovechamiento de arquitecturas de cómputo paralelo, ya que, en la actualidad, prácticamente todos los modelos de computadoras personales

cuentan con procesadores multi-núcleo o con más de un procesador, por lo que el desarrollo de aplicaciones en paralelo ya no está restringido a grandes y costosos equipos especializados.

Existen además una gran cantidad de programas o librerías que facilitan la conexión de varios equipos en una red, que se comportan como una sola computadora en paralelo, lo que se conoce como Clúster.

Han sido desarrollados muchos algoritmos convencionales para la optimización de funciones, sin embargo para problemas altamente complejos: con espacios de búsqueda grandes, de muchas variables, no diferenciales, o altamente no lineales; algoritmos heurísticos basados en poblaciones han sido aplicados con éxito.

1.2 Definición del problema

La optimización se puede mostrar en dos tipos de problemas igualmente importantes, de variables continuas o de variables discretas, en este caso nos concentraremos en los de variables continuas, donde el número de valores posibles a tomar para cada variable es infinito, es decir, dentro de un rango definido, siempre, entre dos valores asignables observados podemos encontrar un tercer valor observable que también puede ser asignado.

Sugartan, Hansen *et al.* Propusieron un conjunto de funciones (o una batería de funciones) para la competencia de computación evolutiva (CEC05), el cual se usó para analizar y comparar la eficacia y eficiencia de la estrategia desarrollada contra otras ya existentes [1]. Todas las funciones que se encuentran en la batería de pruebas cumplen con la condición de ser escalables, es decir que se puede aumentar el número de variables independientes en las mismas con facilidad y

con ello volverlas más y más complejas de optimizar. Dentro del conjunto de funciones que se encuentran en la batería se eligieron algunas de las más características y complejas; las funciones Rastrigin, Schwefel y Ackley que cuenta con muchos puntos críticos, así como Weierstrass que además no es derivable, funciones para las que ya se han hecho esfuerzos por optimizarlas [2, 3, 4].

> “Se conoce como heurísticos a todos aquellos métodos que solamente necesitan la definición de un espacio de soluciones, los operadores, y la función objetivo, sin exigir el cumplimiento de ninguna característica especial, como podría ser la continuidad, la existencia de derivadas, etc. La importancia vital de estos métodos está en que son la única posibilidad para resolver problemas con las condiciones siguientes:
>
> - No existe un algoritmo eficiente para la solución partícular del mismo.
> - No es posible la exploración exhaustiva del espacio completo de búsqueda, debido a las dimensiones del mismo.
>
> Ninguno de ellos garantiza la obtención del óptimo global en el marco de condiciones prácticas. Sin embargo, ellos permiten obtener buenas soluciones que en muchos casos satisfacen a los usuarios” [5].

Los algoritmos genéticos son un heurístico basado en poblaciones propuesto por Holland dónde los individuos más aptos son los que sobreviven y producen nuevos individuos [6].

Los algoritmos basados en enjambres de partículas, conocidos como PSO son otra estrategia heurística creada por Eberhart y Kennedy basados en el movimiento de algunas especies animales dónde la mayoría de la población sigue

a los miembros que se encuentran en mejores condiciones (como bandadas de pájaros o bancos de peces) [7].

En el presente trabajo se desarrolla un algoritmo que combina dos de los principales métodos heurísticos de optimización de variables continuas, la optimización por enjambre de partículas (PSO) y los algoritmos genéticos. El algoritmo será implementado en una arquitectura en paralelo de bajo costo.

1.3 Objetivos

1.3.1 Objetivo general

Desarrollar un algoritmo de optimización para problemas de variables continuas, basado en dos estrategias distintas (Algoritmos genéticos y Optimización por enjambre de partículas), que sea capaz de mejorar la eficacia y eficiencia de los algoritmos existentes, y que pueda ser ejecutado en arquitecturas paralelas de bajo costo.

1.3.2 Objetivos específicos

- Desarrollar un algoritmo de optimización basado en algoritmos genéticos y optimización por enjambre de partículas.
- Implementar el algoritmo en una arquitectura en paralela de bajo costo (un clúster de computadoras).
- Probar la solución desarrollada con las funciones de la batería de pruebas CEC05.
- Realizar el análisis y comparación de la eficacia y eficiencia de la solución planteada contra el de trabajos previos.

2. Optimización de problemas continuos en paralelo

2.1 Inteligencia artificial y heurística

Los hombres se han denominado a sí mismos como Homo Sapiens (hombre sabio) porque nuestras capacidades mentales son muy importantes para nosotros, durante miles de años, hemos tratado de entender cómo pensamos.

Un puñado de ciencias, entre las que destaca la filosofía ha intentado desde épocas muy remotas, dar explicaciones a las interrogantes acerca del pensamiento humano y el razonamiento [8].

La inteligencia artificial (IA), término acuñado en 1956 y explotado después de la segunda guerra mundial va más allá, no sólo intenta comprender, sino que también se esfuerza en construir entes capaces de ‘pensar’ inteligentemente o razonar.

La IA toma modelos del razonamiento humano y los formaliza a través de las matemáticas, para después aplicarlos a problemas específicos, además, usa conceptos de la lógica, probabilidad, estadística y otras áreas de la ciencia.

Los problemas que trata la inteligencia artificial abarcan un espectro muy amplio y están clasificados en tareas de la vida diaria, tareas formales y tareas de los expertos [9].

Algunos ejemplos de estas tareas son el habla, la visión, el sentido común, la teoría de juegos, demostración de teoremas, sistemas de control y por supuesto la optimización.

El paradigma principal en la IA para problemas de optimización es la búsqueda heurística, un método muy general que se aplica a problemas difíciles, para encontrar soluciones en tiempos prudentes, pero, sin garantizar que sea la solución óptima, su objetivo es que dichas soluciones sean aceptablemente buenas.

Existen muchos algoritmos de búsqueda heurística, con los que se ha dado solución a problemas discretos y continuos, entre sus principales exponentes se encuentran los escaladores de colinas, recocido simulado, búsqueda tabú, métodos de inteligencia colectiva, optimización por colonia de hormigas, algoritmos genéticos y optimización por enjambre de partículas, entre otros [10].

2.2 Algoritmos genéticos

Los algoritmos genéticos (AG) propuestos por Holland son métodos heurísticos de búsqueda inspirados en lo que sabemos acerca del proceso de la evolución natural, en las teorías de Darwin y de Mendel [6].

2.2.1 Individuo, población y aptitud

En las primeras versiones de algoritmos genéticos, de variable discreta, los individuos eran vectores binarios en las que cada bit representaba un gen o una característica de la solución al problema, sin embargo en las versiones de variable real o continua, cada individuo es un vector o arreglo de números reales que representa un punto en el espacio de búsqueda R^n.

Una población P de individuos consiste en un conjunto de individuos C_i donde i va desde 1 hasta el tamaño de la población μ.

$$P = \{C_1 , C_2 , \dots , C_{\mu-1} , C_\mu\} \qquad (1)$$

Los algoritmos genéticos inician generalmente con una población generada aleatoriamente, después se evalúan sus individuos, y se aplican los operadores genéticos para producir la población de la generación siguiente, este proceso de repite hasta que se cumple con alguna condición de parada o un número determinado de generaciones (o vueltas en el ciclo).

La aptitud de los individuos nos indica que tan bien un individuo satisface o soluciona el problema de optimización, y sirve como criterio para discriminar a diferentes individuos dentro de una población.

El paso en el que se evalúa la aptitud de los individuos suele ser el más costoso para una aplicación real, puede ser una subrutina, un simulador, o cualquier proceso externo, muchas veces se usan funciones aproximadas para reducir el costo de evaluación, cuando existen restricciones éstas se pueden introducir en el costo como penalizaciones.

2.2.2 Operadores genéticos

2.2.2.1 Selección

Una vez calificados todos los individuos de una generación, el algoritmo debe seleccionar a los individuos más calificados, mejor adaptados para resolver el problema, de esta forma se incrementa la probabilidad de tener individuos o soluciones 'buenas' en el futuro.

Existen diferentes métodos para seleccionar a los individuos de la siguiente generación, uno de ellos se conoce como 'elitismo' y consiste simplemente en especificar que permanezcan en la población los mejores *n* individuos sin embargo una estrategia más utilizada es la 'selección por torneo' (figura 1), que consiste en tomar muestras de Z individuos e insertar en la siguiente generación al mejor de esos Z individuos, o a los mejores k individuos de esos Z. Valores pequeños de Z, junto a la operación genética llamada mutación suelen disminuir el riesgo de que los algoritmos caigan en óptimos locales.

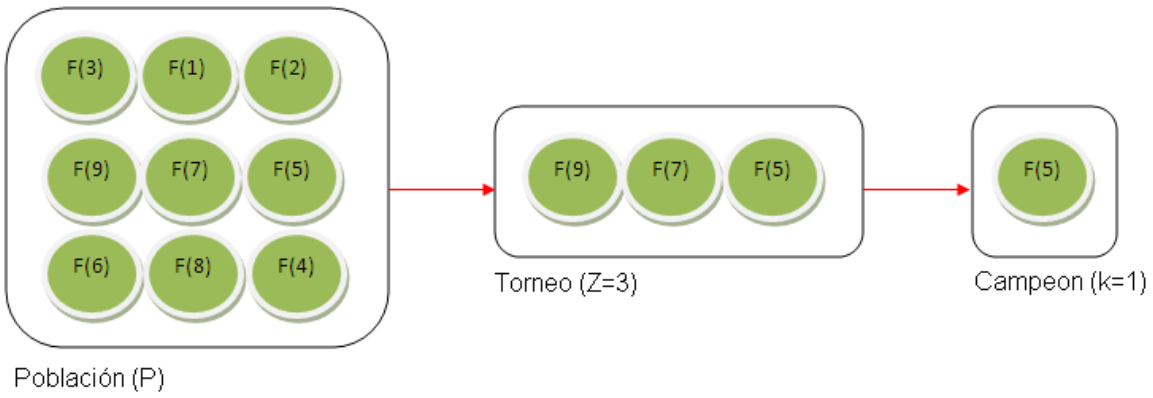

Figura 1 Selección por torneo.

2.2.2.2 Cruzamiento

Mientras que durante la selección solamente pasamos individuos de una generación a otra, es en el cruzamiento donde se producen nuevas soluciones, el cruzamiento consiste en tomar dos individuos o padres y a partir de ellos producir nuevos individuos o hijos, en AG's discretos estos nuevos individuos consisten en dividir a los individuos viejos en dos secciones y recombinarlos para formar a su descendencia, lo que implica generar un escalar i entre uno y la longitud de la cadena binaria, llamado punto de cruzamiento y a partir de esa posición recombinar (figura 2).

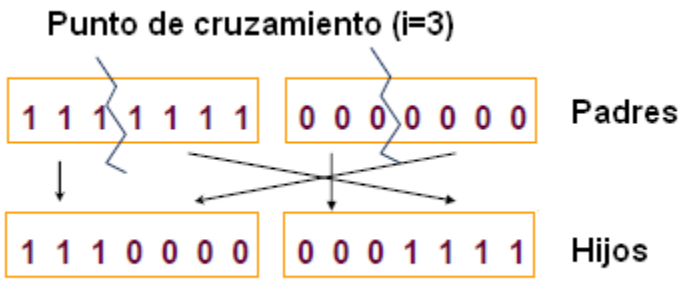

Figura 2 Cruzamiento para cadenas binarias.

Para AG's de variable real se genera un escalar λ entre cero y uno que permite generar arreglos con una mezcla proporcional de los valores de los padres:

$$C'_1 = (\lambda)(C_1) + (1- \lambda)(C_2) \qquad (2)$$

$$C'_2 = (1-\lambda)(C_1) + (\lambda)(C_2) \qquad (3)$$

Se conoce como probabilidad de cruce a la razón de la cantidad de individuos generados por este método y el tamaño de la población.

2.2.2.3 Mutación

Ocasionalmente algunos individuos se alteran a propósito, éstos se seleccionan aleatoriamente en lo que constituye el símil de una mutación. El objetivo es generar nuevos individuos que exploren regiones del dominio del problema que probablemente no se han visitado aún.

Por lo regular se genera un valor aleatorio entre cero y uno, y se compara con la probabilidad de mutación (un valor previamente definido), si es menor se procede a alterar al individuo, en caso contrario no, para los AG's discretos, se calcula nuevamente un valor aleatorio i entre uno y la longitud de la cadena y el nuevo individuo se calcula simplemente invirtiendo el bit o gen en la posición 'i' (figura 3).

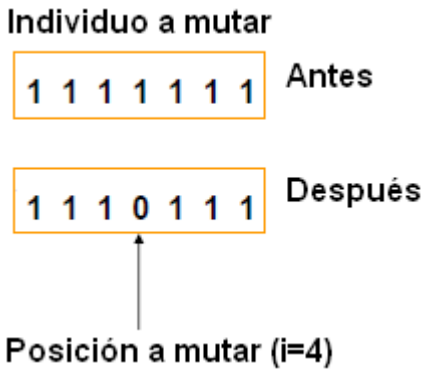

Figura 3. Mutación para cadenas binarias.

En la representación real el proceso es muy similar solamente que el nuevo valor se genera aleatoriamente en el rango de valores fijados como limites del espacio de búsqueda.

En general la probabilidad de que un individuo mute suele ser muy baja para evitar que la búsqueda se vuelva demasiado aleatoria.

2.2.3 Algoritmos genéticos en paralelo

Una función real puede tener varios puntos de inflexión, es decir puntos con una derivada total igual a cero, que son mejores que todos sus vecinos inmediatos en cualquier dirección, pero de todos los puntos de inflexión solamente uno es realmente el punto de valor óptimo en toda la función (óptimo global), los demás son llamados óptimos locales. Muchas veces los métodos heurísticos convergen a un óptimo local, en particular algoritmos genéticos que se ejecutan en un solo equipo tienen dicho problema, sin embargo los algoritmos genéticos en paralelo, al buscar al mismo tiempo en distintas partes del espacio de búsqueda minimizan las posibilidades de quedar atrapados en óptimos locales, e igualmente importante, al procesarse de manera paralela los cálculos de aptitud y las operaciones genéticas en distintas poblaciones se reduce el tiempo de corrida del algoritmo.

Nowostawski y Poli [11] proponen clasificar los algoritmos genéticos en 8 categorías, que obtuvieron considerando los siguientes rasgos:

- La forma de evaluar la aptitud y aplicar la mutación.
- Si se implementa en una sola población o varias sub-poblaciones.
- En el caso de que se implemente en varias sub-poblaciones, la forma en la que se intercambian individuos entre ellas.
- Si se hace selección global o localmente.

Las categorías propuestas son:

1. Maestro-Esclavo.
    - Una sola población.
    - Generalmente se parareliza el cálculo de aptitud y la mutación, a cargo de varios procesos esclavos.

◦ La selección y el cruce se llevan a cabo de manera global, a cargo de un proceso maestro.

◦ Puede ser:

▪ Síncrono: El maestro espera a que todos los esclavos terminen de aplicar sus operadores, antes de iniciar la siguiente generación

▪ Asíncrono: El maestro sólo espera a una fracción de los esclavos para iniciar la siguiente generación.

2. Sub-poblaciones estáticas con migración (modelo de islas).

◦ Conjunto estático de sub-poblaciones independientes.

◦ Requiere implementar un operador genético adicional: migración, que consiste en intercambiar individuos entre sub-poblaciones cada cierto número de generaciones. La migración permite compartir material genético entre las sub-poblaciones y para implementarla requiere que se incluya en el algoritmo un conjunto de parámetros adicionales.

◦ Los modelos de migración pueden ser:

▪ En anillo: Antes de iniciar el algoritmo se determina la población destino de cada elemento a migrar.

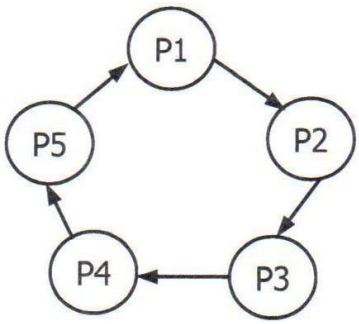

Figura 4 Migración en anillo

▪ Aleatoria: Se elige al azar la población destino, en el momento de ejecutar la migración.

Generación 1 Generación 2

Figura 5 Migración aleatoria.

◦ Las estrategias para seleccionar los individuos que migrarán pueden ser:

▪ Elitista: Se migran los mejores individuos de cada población.

▪ Por torneo: Similar a la selección por torneo.

3. Sub-poblaciones estáticas superpuestas.

◦ A diferencia de la categoría anterior, no existe un operador de migración.

◦El intercambio de información genética se hace mediante individuos que pertenecen a más de una sub-población.

4. Algoritmos genéticos masivamente paralelos.

◦ Básicamente funcionan como la categoría anterior, solo que se disminuye la cantidad de individuos en la sub-población.

◦ Se implementan en computadoras masivamente paralelas.

5. Sub-Poblaciones dinámicas.

◦ Una sola población.

◦ Se divide en sub-poblaciones en tiempo de ejecución.

6. Algoritmos genéticos de estado estable.

    ◦ Se distinguen de los algoritmos maestro-esclavo con evaluación asíncrona de aptitud por la forma en la que aplican la selección.

    ◦ El algoritmo no espera a que una fracción de la población haya concluido su evaluación, sino que continúa con la población existente.

7. Algoritmos genéticos desordenados.

    ◦ Constan de tres fases:

    ▪ Inicialización: Se crea la población inicial, usando algún método de enumeración parcial.

    ▪ Primaria: Se reduce la población mediante alguna forma de selección y se evalúan sus individuos.

    ▪ Yuxtaposición: Se unen las soluciones parciales resultantes de las fases anteriores.

8. Métodos híbridos.

    ◦ Combinan características de los métodos anteriores.

Cada una de las categorías puede presentar mejores rendimientos, dependiendo del tipo de problema a tratar y de la arquitectura de la máquina paralela en la que se implementará la solución.

En la figura 6, podemos observar el diagrama de flujo general de un algoritmo genético con migración.

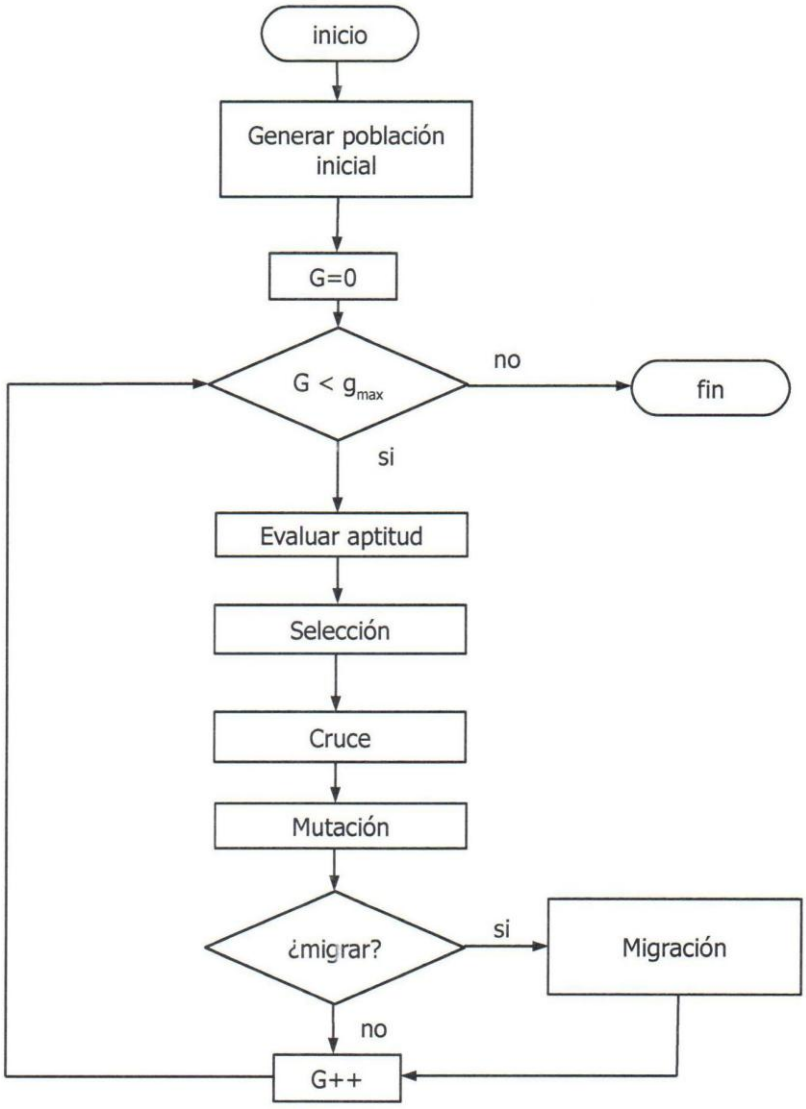

Figura 6 Diagrama de flujo de un algoritmo genético.

2.2.4 Características de los algoritmos genéticos

Todo algoritmo de búsqueda necesita establecer un equilibrio entre dos factores aparentemente contrapuestos [12]:

- Exploración del espacio de soluciones, para realizar una búsqueda en amplitud, localizando así zonas prometedoras, y
- Explotación del espacio de búsqueda, para hacer una búsqueda en profundidad en dichas zonas, obteniendo así las mejores soluciones.

Dos factores contrapuestos influyen en la efectividad de un AG:

- Convergencia: Centrar la búsqueda en regiones prometedoras mediante la presión selectiva.
- La presión selectiva permite que los mejores individuos sean seleccionados para reproducirse. Es necesaria para que el proceso no sea aleatorio.
- Diversidad: Evitar la convergencia prematura (rápida convergencia hacia óptimos locales).

Se conoce como falta de diversidad genética cuando todos los individuos en la población son parecidos, y esto conlleva siempre a una convergencia prematura hacia óptimos locales, en la práctica este proceso es irreversible.

Los algoritmos genéticos son un tipo de algoritmo de búsqueda de propósito general, cuyos operadores pueden establecer un equilibrio adecuado entre exploración y explotación.

Las principales características que hacen a los algoritmos genéticos tan atractivos desde un punto de vista computacional y que los han hecho muy populares en muchos campos, además de dicho equilibrio, son:

-Las pocas condiciones que debe cumplir el espacio de búsqueda; ya que, solo se requiere una forma de representación de las soluciones y una forma de medir el grado en que cada una de ellas satisface el problema.

-La disponibilidad de varias métodologías probadas para su paralelización.

2.3 Optimización por enjambre de partículas

La optimización por enjambre de partículas (Particle Swarm Optimization ó PSO) propuesta por Kenedy y Eberhart es un método heurístico de búsqueda inspirado en el comportamiento social del vuelo de las bandadas de aves y el movimiento de cardúmenes de peces [7].

2.3.1 Enjambre y partícula

Un enjambre es el conjunto de varias partículas (enjambre de partículas = particle swarm) que se mueven (“vuelan”) por el espacio de búsqueda durante la ejecución del algoritmo, es el conjunto de todos las partículas p_i donde i va desde 1 hasta el tamaño del enjambre μ.

$$P = \{p_1 , p_2 , \dots , p_{\mu-1} , p_\mu\} \qquad (4)$$

Dentro de PSO se conoce como partícula a cada solución potencial (equivalente a los individuos en AG’s), una partícula es una estructura de datos que en su forma más simple está compuesta por (figura 7):

-Tres vectores:

El vector X, almacena la posición actual (posición) de la partícula en el espacio de búsqueda.

El vector Y almacena la localización de la mejor solución encontrada por la partícula hasta el momento y,

El vector V almacena la dirección y magnitud hacia donde se moverá la partícula (velocidad).

-Dos escalares:

El escalar ‘ap’ que almacena la aptitud de la solución actual (del vector X).

El escalar ‘map’ que almacena la mejor aptitud histórica de la partícula (del vector Y), viene a ser el valor más alto que ha tomado la variable ‘ap’ en problemas de maximizar o el más pequeño en problemas de minimización.

Figura 7. Anatomía de una partícula

2.3.2 Movimiento de una partícula

Al iniciar el algoritmo el vector X de todas las partículas se genera aleatoriamente entre los límites del espacio de búsqueda, es decir en el rango en el rango $[-x_{max}, x_{max}]$.

Para el vector V es común que se tengan valores máximos distintos a los del espacio de búsqueda, generalmente submúltiplos de este, de tal forma que V se inicializa con valores aleatorios pero en un intervalo más pequeño.

$$V_{max} = k * x_{max} \text{ , donde } 0.1 < k < 1 \qquad (5)$$

Una partícula se mueve simplemente añadiendo el vector velocidad V al vector posición X:

$$X(t+1)_i = X(t)_i + V(t)_i \qquad (6)$$

Una vez calculada la nueva posición de la partícula, se evalúa ésta. Si su aptitud nueva es mejor que la que tenía hasta ahora, entonces:

$$map = ap \qquad (7)$$

De este modo, el primer paso es ajustar el vector velocidad, para después sumárselo al vector posición, el cálculo del vector velocidad está determinado por la velocidad actual de la misma (inercia), por la mejor posición histórica de la partícula y por la mejor posición de un conjunto de partículas (figura 8).

Existen dos estrategias para calcular el vector velocidad de PSO, según sea el conjunto de partículas considerado, la estrategia global y la basada en vecindarios. La diferencia de ambas estrategias es que la primera considera a todo el enjambre de partículas y la segunda solamente a las partículas más cercanas (geográfica o lógicamente) de la que queremos ajustar su velocidad. La estrategia global converge más rápido pero cae con mayor frecuencia en óptimos

locales, al contrario de la segunda, en lo sucesivo llamaremos a un conjunto de partículas como vecindario (las más cercanas, que no necesariamente las incluye a todas) y a todas las partículas o el conjunto de todos los vecindarios como enjambre.

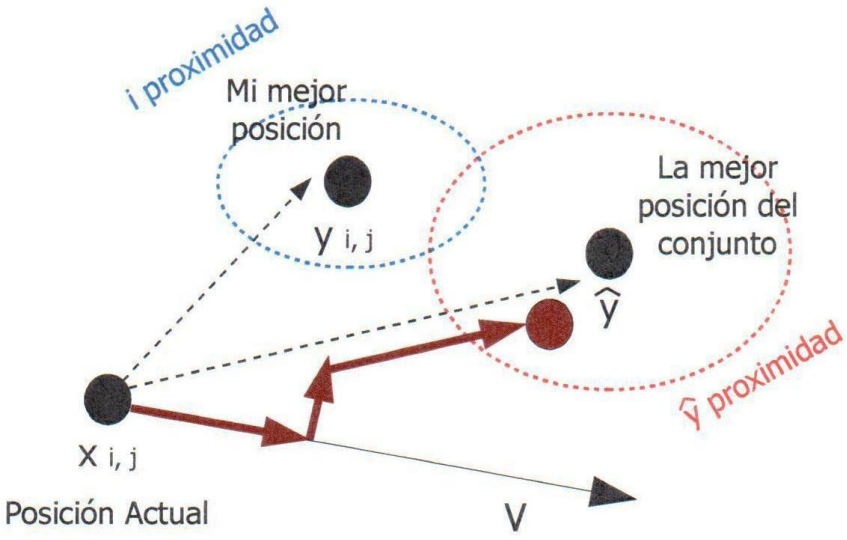

Figura 8 Representación gráfica del movimiento de una partícula.

Durante cada iteración del algoritmo, se cambia la velocidad de la partícula, para acercarla con su mejor posición y la del vecindario.

El algoritmo utiliza dos valores aleatorios independientes entre cero y uno, r_1 y r_2; los cuales son potenciados por los coeficientes de aceleración o 'pesos' C_1 y C_2, dónde C_1 controla la componente cognitiva y C_2 la componente social de la velocidad.

La velocidad se actualiza por separado para cada dimensión *n* en R^n del problema, es decir para cada uno de los componentes de la misma, o visto de manera más práctica, para cada variable problema.

De tal manera que $V_{i,n}$ denota la enésima dimensión del vector velocidad asociado con la partícula ‘i’. De esta forma la ecuación con la que se actualiza la velocidad de la partícula en la posición ‘i’ es:

$$v_{i,n}(t+1)=v_{i,n}(t)+c_1 r_{1,n}(t)[y_{i,n}(t)-x_{i,n}(t)]+c_2 r_{2,j}(t)[y_n(t)-x_{i,n}(t)] \quad (8)$$

Donde y es la mejor posición de toda la población para la estrategia global y la del vecindario para la estrategia basada en los mismos.

Visto de otra manera tenemos dos componentes que modifican la velocidad en el tiempo:

$$v_{i,j}(t+1)=v_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t)-x_{i,j}(t)] + c_2 r_{2,j}(t)[\hat{y}_j(t)-x_{i,j}(t)]$$

Componente **cognitivo** Componente **social**

El componente cognitivo, es cuánto confía la partícula en su propio rendimiento y el social es cuánto lo hace en su entorno.

Kennedy [7] identifica las siguientes topologías de PSO:

1. Completo, C_1 y $C_2 > 0$

2. Solo cognitivo $C_1 > 0$ y $C_2 = 0$

3. Solo social $C_1 = 0$ y $C_2 > 0$

4. Solo social exclusivo $C_1 > 0$ y $C_2 = 0$ dónde $\hat{y} \neq i$ (la mejor partícula del entorno no debe ser la misma partícula).

Un problema habitual de los algoritmos de PSO es que la magnitud de la velocidad suele llegar a ser muy grande durante la ejecución por lo que las partículas se mueven demasiado rápido por el espacio, lo cual disminuye el rendimiento del

algoritmo, un método para controlar dicho crecimiento excesivo es el de colocar un factor de inercia w que multiplique el valor de la velocidad en el tiempo anterior.

$$v_{i,n}(t+1)=w\,v_{i,n}(t)+c_1 r_{1,n}(t)[y_{i,n}(t)-x_{i,n}(t)]+c_2 r_{2,j}(t)\ [y_n(t)-x_{i,n}(t)] \quad (9)$$

Se han hecho muchas investigaciones para encontrar los valores de C_1, C_2 y w que aseguren la convergencia del algoritmo; para w, Shi y Eberhart [13] sugieren un valor en el intervalo, $0.8 < w < 1.2$; y conjuntamente con Clerc en [14] sugieren $C_1 = C_2 = 1.4692$.

Si se decide trabajar con la estrategia basada en vecindarios, es importante decidir la topología de los vecindarios, Kennedy [15] propone topologías en estrella o en anillo para los vecindarios, con variaciones aleatorias en la comunicación entre sus nodos (figura 9):

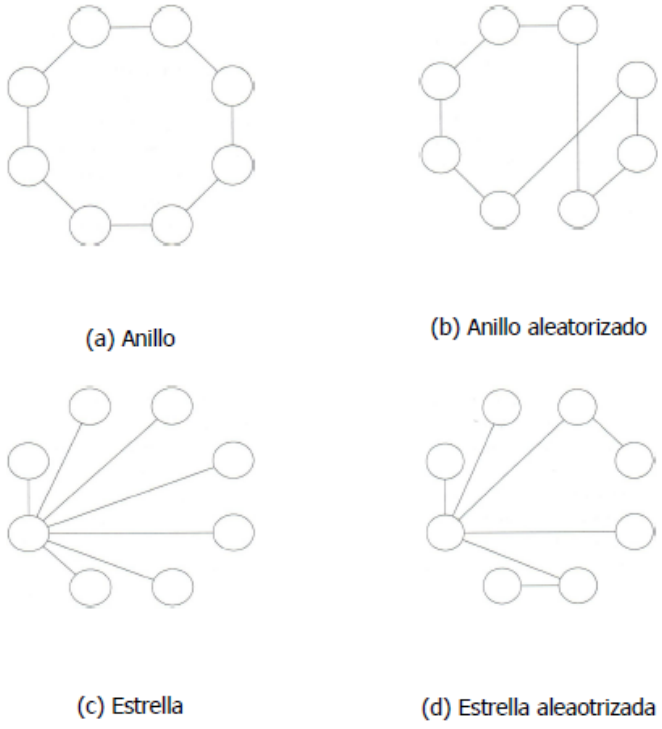

Figura 9 Topologías de vecindarios

2.3.3 Optimización por enjambre de partículas en paralelo

La optimización por enjambre de partículas tiene características que lo hacen factible de paralelizar. Una de ellas es el manejo de sub-poblaciones.

Dependiendo del tamaño de las sub-poblaciones o vecindarios y de la forma en que se intercambia la información entre dichos vecindarios, la paralelización puede ser de grano grueso o de grano fino [16].

En la estrategia que es conocida como de grano grueso o basada en islas, se procesan varios vecindarios de manera independiente, excepto por un intercambio ocasional de partículas entre ellos tras cierto número de generaciones o iteraciones (figura 10).

La estrategia llamada de grano fino, que consta de un solo vecindario organizado en una malla ortogonal donde los vecinos intercambian información.

Si se utiliza paralelización de grano grueso, la actualización de los parámetros de velocidad y posición de la partícula i en un vecindario k se hace hasta que se actualizaron los de la partícula $i-1$. La paralelización consiste en que se hace esa actualización simultáneamente en todos los vecindarios que se hayan definido.

En la implementación de grano fino, se actualizan simultáneamente los parámetros de velocidad y posición de todas las partículas del enjambre.

No obstante que la paralelización del algoritmo disminuye notablemente el tiempo de ejecución del algoritmo, a mayor grado de paralelización son necesarias mayor cantidades de iteraciones para que el algoritmo converja [17].

Al igual que en otras heurísticas basadas en poblaciones, las estrategias en paralelo que consideran la migración de individuos produce una mayor diversidad

y evita caer fácilmente en óptimos locales, así como reducen el tiempo de corrida (al realizar más procesos de manera simultánea).

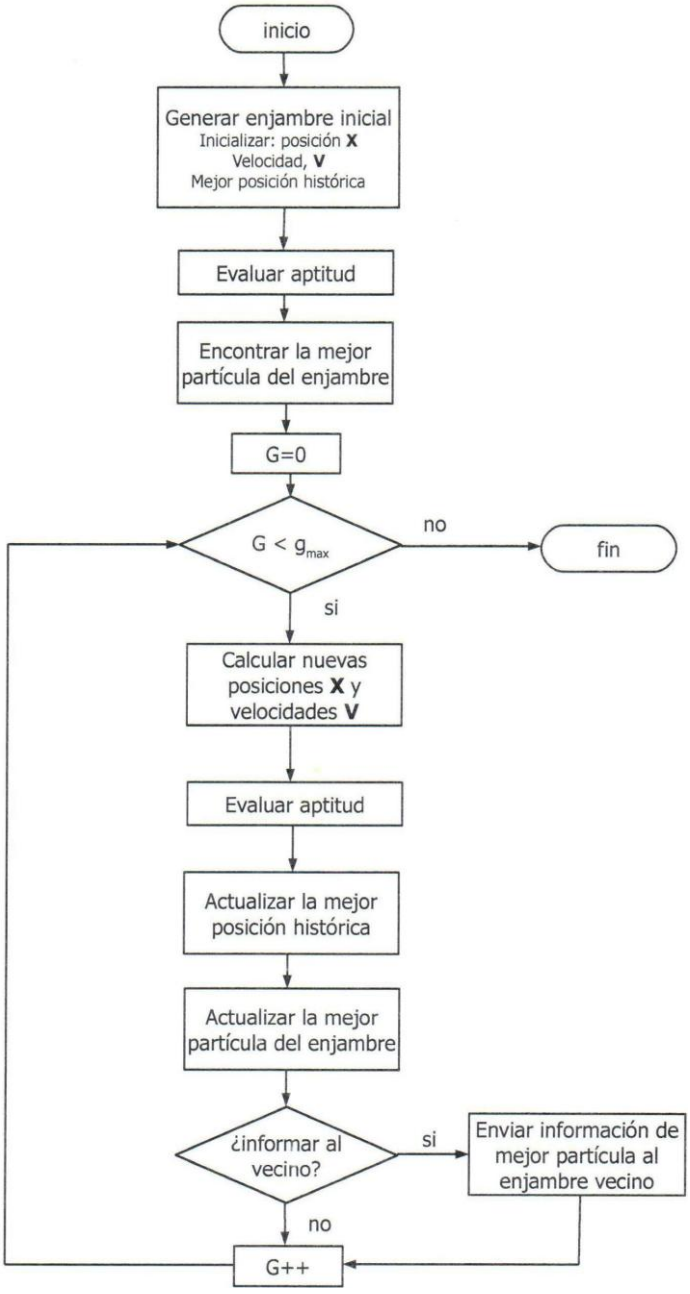

Figura 10 Diagrama de flujo de PSO

2.3.4 Características de PSO

PSO es un algoritmo de búsqueda heurística que cada vez se encuentra más en uso en diversas áreas de la ingeniería, con una gran potencialidad de aplicaciones, debido a que su implementación es muy sencilla, a que utiliza pocos parámetros y sobre todo a que converge muy rápido a buenas soluciones, incluso con problemas no tan sencillos.

Si bien al igual que un AG, se pueden adaptar algunas de sus operaciones, concretamente el cálculo de la velocidad, así como utilizar modelos basados en vecindarios y paralelizarlo, para evitar una convergencia prematura, PSO es generalmente una estrategia de optimización con una convergencia más rápida que en un AG, pero también que suele caer más en óptimos locales, y que no tiene ninguna clase de respuesta a la falta de diversidad en sus soluciones (no hay nada similar a la operación genética de mutación en un AG, por ejemplo).

Por este motivo nunca se deben sacar conclusiones con una única ejecución del algoritmo, se deben utilizar medidas estadísticas como medias o medianas y un número suficiente de ejecuciones independientes.

PSO es particularmente veloz en problemas que no poseen demasiados puntos de inflexión o con espacios de búsqueda reducidos; en espacios ya sesgados aplicar dicho algoritmo es bastante eficiente.

2.4 Métodos híbridos

Todos los que hemos trabajado con algoritmos de optimización, sabemos que no existe uno mejor que otro, que simplemente algunos son mejores a los demás en ciertas circunstancias, la mayoría tiene sus ventajas y desventajas claramente marcadas y por ello no es de extrañarse que se haya intentado con éxito desarrollar algoritmos híbridos, es decir, derivados de otros o mezclas que intentan sacarle el mayor provecho a las ventajas de cada uno de ellos.

A continuación algunos ejemplos exitosos:

En 2007, Petcu y Faltings [18] desarrollaron un algoritmo para optimización combinatoria basado en métodos de búsqueda locales (que solo toman decisiones basados en información local) y métodos completamente inferenciales.

En 2010, Elhossini et al. [19] Presentaron una eficiente estrategia basada en PSO y el ‘frente de Pareto’ que generalmente se ve en algoritmos evolutivos, para resolver problemas de optimización multiobjeto.

En el CIE 2011 Tanda Martinez y Aguado Behar [20] utilizaron una estrategia híbrida basada en algoritmos genéticos y en el método simplex para la identificación experimental de sistemas de lazo abierto, obteniendo resultados ‘muy superiores a los obtenidos por métodos convencionales’, una estrategia muy similar llamada ‘Algoritmo Simplex Génetico’ se puede ver en [21].

Algoritmos híbridos se han utilizado además para la reconstrucción de imágenes [22], para el clásico problema del comerciante viajero [23], entre muchos otros [24], [25].

2.5 Arquitecturas paralelas de bajo costo

2.5.1 GPGPU

Desde que, en el 2001, la compañía NVIDIA liberó la serie de tarjetas gráficas GeForce 3, que tuvieron la capacidad de ejecutar tanto instrucciones de propósito general como para el manejo de gráficos en ellas, la programación en paralelo ha avanzado rápidamente. Se ha evolucionado desde tener que “engañar” a la GPU (graphics processing unit) utilizando para otros propósitos sus instrucciones para el manejo de pixeles, hasta contar con las interfaces de programación específicas que existen actualmente, por ejemplo CUDA [3].

La arquitectura CUDA (Compute Unified Device Architecture) surgió en 2006, cuando NVIDIA lanzó sus tarjetas GeForce 8800 GTX las que por primera vez incluyeron elementos dirigidos específicamente a posibilitar la solución de problemas de propósito general. Poco después, NVIDIA lanzó el compilador CUDA C, el primer lenguaje de programación para desarrollar aplicaciones de propósito general sobre una GPU; con la finalidad de captar la mayor cantidad de programadores posibles que adoptasen esta arquitectura, CUDA C es un lenguaje muy similar a C, al cual se le agregaron un conjunto de instrucciones que harían posible la programación en paralelo en un sólo equipo de cómputo. [27]

2.5.2 Clúster

Una forma de resolver el problema de los altos costos al desarrollar aplicaciones paralelas, que se viene aplicando en varias instituciones científicas y en partícular de corte educativo, es la creación de máquinas paralelas virtuales (también conocidas como clúster). Un clúster es un conjunto de computadoras interconectadas a través de una red, que trabajan de forma cooperativa mediante protocolos de paso de mensajes para emular una computadora paralela [26].

Actualmente, el ancho de banda de comunicación inter-procesadores en un clúster es menor que aquel que puede alcanzarse usando el bus interno de una computadora. Sin embargo, es posible considerar esta limitación al momento de diseñar aplicaciones para su ejecución en un clúster tratando de minimizar la cantidad de datos que deban transferirse entre procesadores.

En la actualidad las librerías más comúnmente utilizadas para la implementación de clúster son PVM (Parallel Virtual Machine) y MPI (Message Passing Interface), las dos implementaciones más populares para MPI en clúster son LAM/MPI y MPICH.

2.5.2.1 Estructura de PVM

Se decidió desarrollar el algoritmo utilizando PVM, un paquete de software con el que es posible simular un clúster a partir de un conjunto heterogéneo de computadoras.

Se eligió PVM por las principales características del mismo [26].

- Conjunto de máquinas definido por el usuario: El usuario puede elegir a través de una consola, de entre las máquinas conectadas al clúster, en cuales de ellas se va a ejecutar una tarea. Incluso las máquinas que forman parte de ese conjunto pueden ser agregadas o eliminadas dinámicamente, lo cual es una característica importante para la implementación de tolerancia a fallos.
- Acceso traslúcido al hardware: Los usuarios pueden utilizar un acceso transparente al hardware, en el que se permite que PVM decida la asignación de tareas o puede asignarse de manera explícita cuál máquina ejecutará cada tarea, para explotar mejor las capacidades del hardware.
- Computación basada en procesos: La unidad de paralelismo en PVM es una tarea, constituyendo un hilo independiente de control que se alterna entre los estados de comunicación y cálculos. No existe un mapeo directo entre tareas y procesadores. De hecho es posible ejecutar varias tareas en una sola máquina.
- Modelo de paso de mensajes explícito: Un conjunto de tareas cooperan en la solución de un problema enviándose mensajes entre sí. El tamaño de los

mensajes sólo está limitado por la cantidad de memoria de cada computadora.

- Soporte de heterogeneidad: PVM soporta la heterogeneidad en términos de computadoras, redes y aplicaciones. Con respecto al paso de mensajes, PVM permite que mensajes que contienen más de un tipo de datos puedan ser intercambiados entre computadoras con diferentes formas de representación de los mismos.
- Soporte de multiprocesadores: PVM utiliza las facilidades nativas de paso de mensaje en computadoras paralelas, para hacer uso de las ventajas de este tipo de hardware subyacente. Algunos distribuidores proveen versiones de PVM optimizadas para sus arquitecturas particulares, que pueden comunicarse con las versiones públicas de PVM.

El sistema PVM está compuesto por dos partes: la primera parte es un proceso llamado pvmd3, a veces abreviado como pvmd que se ejecuta en segundo plano en cada computadora conectada a la máquina virtual. Este proceso es el responsable de mantener la comunicación que permite el paso de mensajes y la creación y eliminación de tareas en el clúster. La segunda parte es un conjunto de bibliotecas que permiten a las aplicaciones hacer uso de las características que ofrece PVM.

Para instalar PVM, es necesario que los nodos que van a ser incluidos en el clúster puedan comunicarse entre ellos mediante sus nombres en la red. Esto puede lograrse configurando un DNS (Domain Name Service) o agregando dichos nombres en los archivos de hosts de cada uno de los nodos.

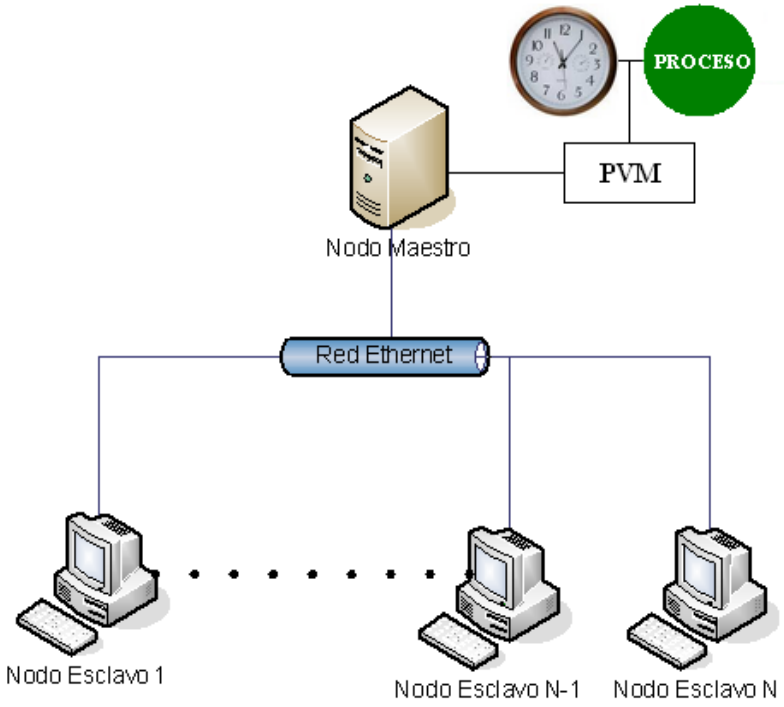

Figura 11. Estructura de un clúster PVM

Un clúster PVM está compuesto de un nodo maestro y varios nodos esclavos, como se muestra en la figura 11.

3. Algoritmo propuesto

3.1 Estructura

Se propone un algoritmo que se puede dividir en tres fases, la inicialización de los procesos esclavos, que incluye el envío de parámetros a los mismos. El núcleo o la optimización en sí, que se puede dividir a su vez en otras dos fases, la primera realiza una búsqueda más en anchura que en profundidad, con el fin de reducir el espacio de búsqueda apoyándose en un algoritmo genético y la segunda más profunda con el objetivo de encontrar el óptimo global de la función, utilizando para ello una estrategia de PSO; en ambas subfases hay una constante comunicación entre los nodos. La última fase consiste en que cada proceso esclavo envíe su mejor resultado al proceso maestro para que filtre al mejor de todos, incluye la escritura en el archivo de los resultados y el cerrado del mismo.

De tal manera que la tarea o proceso maestro del algoritmo se encarga de:

- Generar un proceso esclavo para cada subpoblación.
- Declarar los parámetros de prueba, que son enviados como mensajes en tiempo de ejecución para cada proceso esclavo (figura 12).
- Declarar y crear los archivos en los que se guarden los resultados de las pruebas.
- Recibir los resultados de los n procesos esclavos que se generaron, y escribir en los archivos dichos resultados (número total de evaluaciones, mejor solución para el problema, aptitud de dicha mejor solución así como tiempo de ejecución).

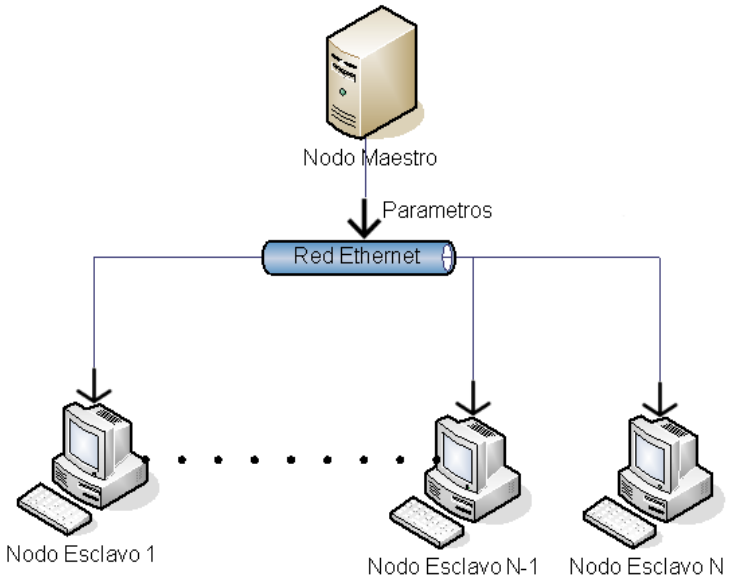

Figura 12. Primera fase, paso de parámetros.

Mientras que los procesos esclavos serían básicamente cada una de las subpoblaciones en las cuales se ejecuta el algoritmo híbrido en sí; una vez que se reciben los parámetros, y se cumple un cierto número de generaciones, cada nodo esclavo comparte información (individuos o partículas) cómo se mencionó anteriormente, con su vecino inmediato, en un símil a la migración (figura 13).

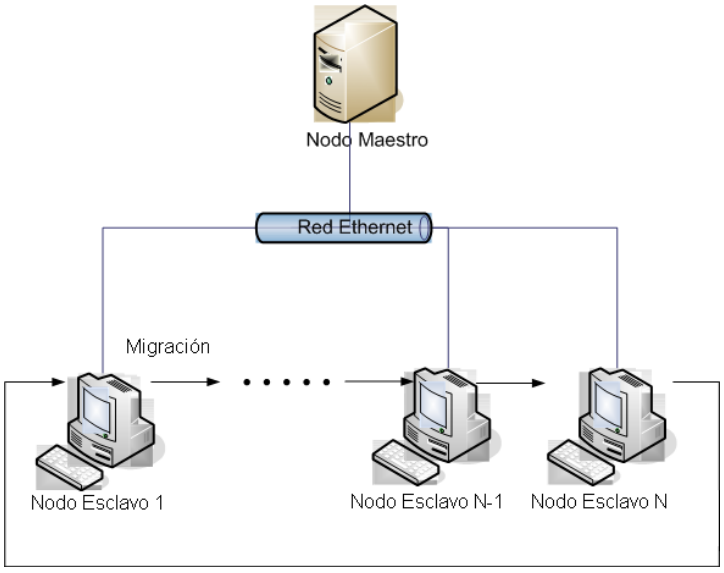

Figura 13. Segunda fase, migración

Una vez que se cumple la condición de parada, el número de generaciones o vueltas del algoritmo, cada nodo esclavo envía sus resultados hacia el nodo maestro, esto se ilustra en la figura 14.

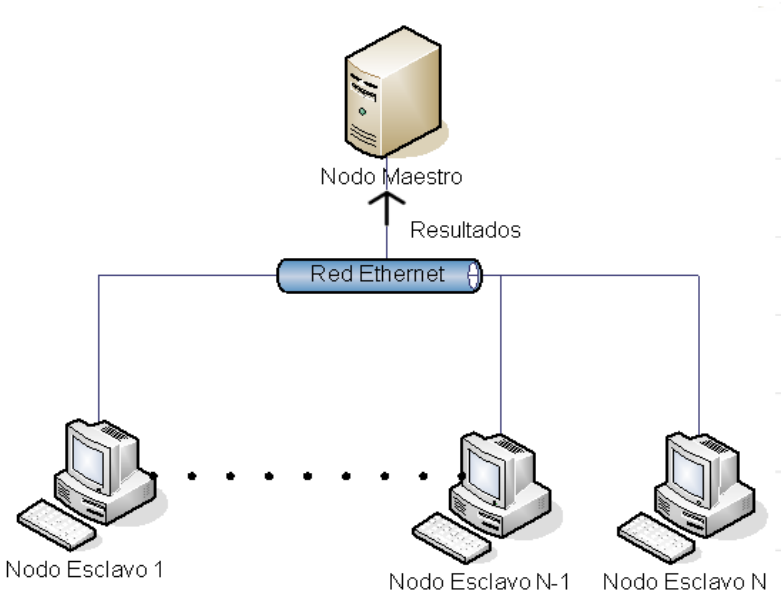

Figura 14. Tercera fase, compartición de resultados.

3.2 Algoritmo híbrido

3.2.1 Paso de parámetros

El proceso que se ejecuta en cada uno de los nodos esclavos del clúster se inicia recibiendo los parámetros de prueba del proceso maestro, estos parámetros incluyen:

- Número de generaciones (que se utiliza como condición de parada).
- Tamaño de la población y del enjambre (número de individuos en cada parte del proceso).
- Periodo migratorio (con qué frecuencia se migran los individuos, una variable para la parte de AG y otra para el PSO).
- Constantes de PSO (C_1, C_2, w y V_i; peso cognitivo, peso social, factor de amortiguamiento y factor de velocidad inicial).
- Constantes de AG (Z, Pc y Pm; tamaño del torneo, probabilidad de cruzamiento y probabilidad de mutación).
- Destino (con el ID del proceso esclavo hacia el cuál debe migrar a los individuos).
- Paridad (con un valor binario que indica si va migrar antes que recibir o viceversa; los procesos con ID impar envían primero a sus individuos y los pares reciben primero, así se optimiza el uso del ancho de banda y se minimizan errores).
- $\propto$ (O Constante de proporcionalidad, esta variable indica que proporción de ciclos se realizaran con cada una de las estrategias, es un valor entre cero

y uno que indica el porcentaje de ciclos del total que se ejecutaran como AG, el restante se ejecutan como PSO).

Una vez recibidos todos los parámetros por parte del maestro, se realizan algunas operaciones simples de gestión de memoria, consolidación de archivos y del generador de números aleatorios y se procede a inicializar la población.

3.2.2 Algoritmo genético

El algoritmo empieza generando una población inicial de individuos de manera aleatoria, la cual pasará por el proceso que ya describimos en el capitulo dos:

Primero buscamos al individuo más apto dentro de los que se generaron aleatoriamente, lo guardamos en una estructura de datos aparte, realizamos las operaciones genéticas sobre la población, el número de generación se incrementa y se repite el ciclo hasta que llegue la hora de migrar, cada proceso migrará ‘n’ veces donde ‘n’ se determina dividiendo el número de vueltas del proceso de AG ($\propto$ * Numero de generaciones) entre el periodo migratorio de AG.

El destino de la migración ya se determinó con anterioridad, cada proceso migra con el proceso con el ID siguiente, y el último con el primero a manera de anillo.

Después de acuerdo al valor de paridad, el proceso esperará a recibir los individuos de la población vecina, o enviará primeramente sus individuos a la siguiente subpoblación.

Al recibir individuos de otra población, se busca a los peores individuos para compararlas y/o reemplazarlos con los que llegaron, en caso de que estos sean mejores.

Una vez llevada a cabo las ‘n’ migraciones, se considera que la búsqueda más en anchura que en profundidad, llevada a cabo por el AG ha finalizado y se procede a continuar con un algoritmo PSO para encontrar el óptimo global del problema.

3.2.3 PSO

Primeramente se libera la memoria empleada en las estructuras de datos del AG y después se inicializa el enjambre de partículas, para esto se redefinen un poco las operaciones de PSO, de tal manera que el vector X ó posición de cada partícula toma las coordenadas de cada uno de los individuos del AG, al no ser necesariamente las dos poblaciones de idéntico tamaño, algunos individuos de la primera parte del proceso quedan descartados y no son asignados a ninguna partícula, de lo que sí nos aseguramos es de que el individuo más apto no se pierda, asignándolo a una estructura de datos independiente.

El proceso viene a ser muy similar, se calcula cuantas veces se va a migrar, se actualizan los vectores posición y velocidad en cada generación, cuando llega el momento se migra de acuerdo a las mismas variables que en la primera parte ($\propto$, Paridad y Destino), en esta parte del proceso no se migra realmente a la partícula completa con todos sus atributos, simplemente se envía la aptitud y se compara con la mejor aptitud de la otra población, de tal manera que su posición, si es la mejor que se vio en la población, sirva como punto de referencia solamente.

Al final de todo el proceso se envía al proceso maestro la aptitud y las coordenadas de la partícula con la mejor solución al problema.

3.3 Desarrollo e implementación

Para el desarrollo del proyecto se utilizó Fedora 14, una distribución basada en el sistema operativo Linux, muy estable, libre y que facilita la instalación de PVM.

El código está en lenguaje C, compilado con GCC 4.7, un conjunto libre de compiladores distribuidos bajo la licencia GPL.

Las pruebas fueron hechas en un clúster de 16 equipos Dell con procesadores Pentium Core 2 Duo de 2.6 Ghz, con 2 GB de memoria RAM

Como parte del proyecto, se desarrollaron los siguientes archivos:

- Mixto-m.c (Código fuente del proceso maestro).
- Mixto-esc.c (Código fuente del proceso esclavo).
- Aptitud.h (Archivo de encabezado donde se define la función de aptitud y sus características, dimensiones y sesgo).

Así como once funciones de aptitud distintas, pertenecientes a la batería de pruebas CEC2005 que se utilizaron para las pruebas: Ackley, Eliptical, Griewank, Quadric, Rastrigin, Rosenbrock, Scaffer, Schwefel, Schwefel-noise, Spherical, Weierstrass.

3.4 Pruebas y Resultados

3.4.1 Funciones probadas

De la batería de pruebas se seleccionaron las siguientes:

Schwefel:

La ecuación de schwefel se muestra en la figura 13, y está definida por la siguiente función:

$$f(x)=\sum_{i=1}^{D}\left(\sum_{j=1}^{i} x_j\right)^2 \qquad (9)$$

Dónde D = Número de dimensiones

Con el espacio de búsqueda entre -100 y 100 para cada dimensión, y las propiedades de ser unimodal, desplazable, no separable y escalable.

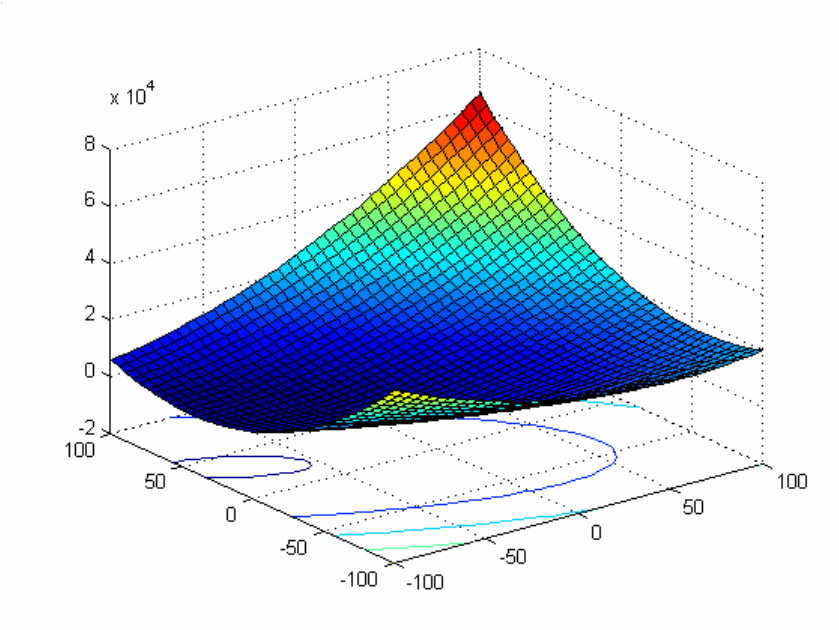

Figura 13 Función Schwefel para D=2

Rastrigin:

La ecuación de rastrigin se muestra en la figura 14, y está definida por la siguiente función:

$$f(x)=\sum_{i=1}^{D}\left(x_i^{\,2}-10\cos(2\Pi\, x_i)+10\right) \qquad (10)$$

Dónde D = Número de dimensiones

Con el espacio de búsqueda entre -5 y 5 para cada dimensión, y las propiedades de ser multimodal, desplazable, separable y escalable; y con una gran cantidad de óptimos locales.

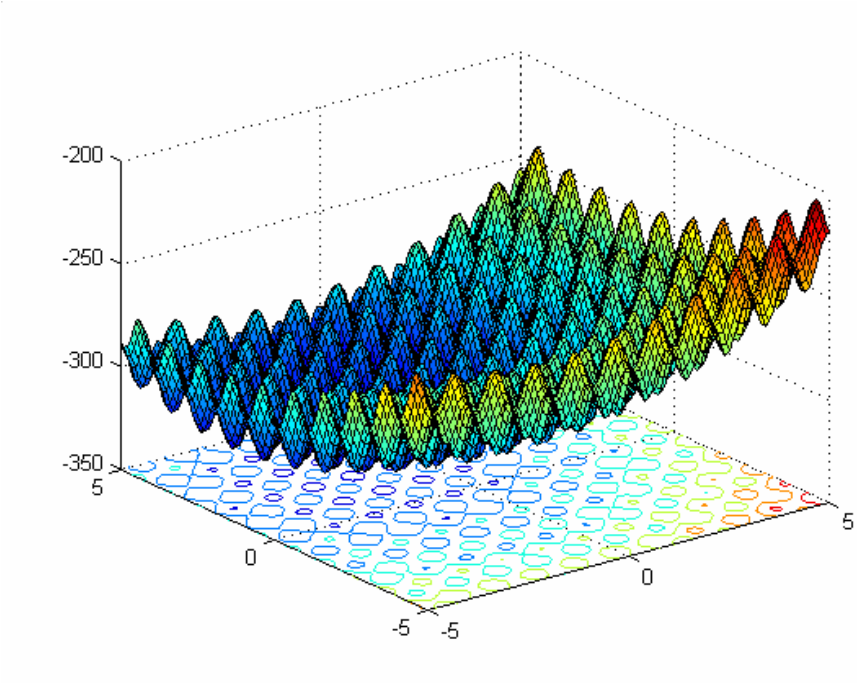

Figura 14 Función Rastrigin para D=2

Ackley:

La ecuación de ackley se muestra en la figura 15, y está definida por la siguiente función:

$$f(x) = -20\ exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^{\,2}}\right) - exp\left(\frac{1}{D}\sum_{i=1}^{D} cos(2\,\Pi\ x_i)\right) + 20 + e \qquad (11)$$

Dónde D = Número de dimensiones

Con el espacio de búsqueda entre -32 y 32 para cada dimensión, y las propiedades de ser multimodal, rotable, desplazable, no separable y escalable; con una gran cantidad de óptimos locales y el óptimo global cerca de la frontera.

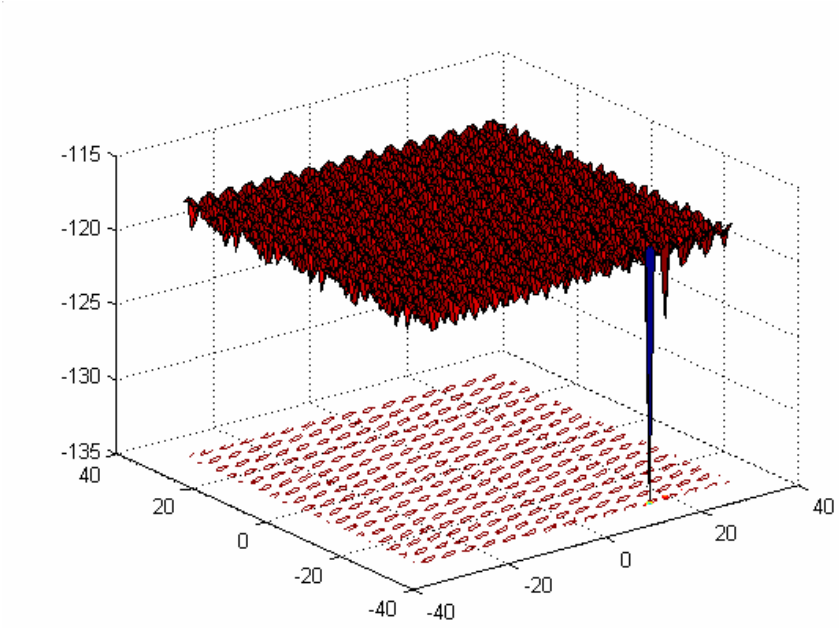

Figura 15 Función Ackley para D=2

Weierstrass:

La ecuación de weierstrass se muestra en la figura 16, y se define como:

$$f(x)=\sum_{i=1}^{D}\left(\sum_{k=0}^{k_{max}}\left[a^k \cos\left(2\Pi b^k\left(x_i+0.5\right)\right)\right]\right) - D\sum_{k=0}^{k_{max}}\left[a^k \cos\left(2\Pi b^k \cdot 0.5\right)\right] \qquad (12)$$

Dónde:

D = Número de dimensiones

a = 0.5

b = 3

k_{max} = 20

Con el espacio de búsqueda entre -0.5 y 0.5 para cada dimensión, y las propiedades de ser multimodal, rotable, desplazable, no separable y escalable; con una gran cantidad de óptimos locales, continua pero diferenciable solamente en algunos intervalos.

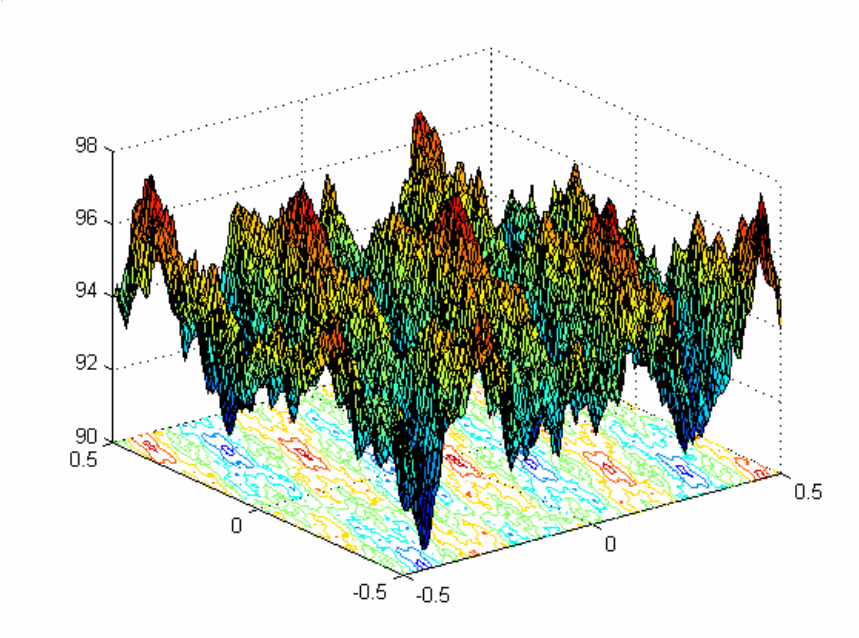

Figura 16 Función Weierstrass para D=2

3.4.2 Metodología de las pruebas

La primera parte de las pruebas consistió en determinar el valor óptimo de la variable de proporcionalidad $\propto$, al ser un poco más rápido PSO comparado con AG, empezamos desde un valor de 0 (es decir, resolviendo los problemas utilizando solamente PSO) y subimos progresivamente el valor de $\propto$, se realizaron pruebas preliminares de 100 muestras, de las cuales se seleccionaban aquellas con porcentaje de éxito mayor a 95% sobre un error permitido de $1x10^{-6}$; las pruebas seleccionadas como candidatas más prometedoras (con los tiempos menores) son después reproducidas pero con muestras de 10,000 corridas, para aumentar la certidumbre en los resultados.

Una vez que encontramos el parámetro $\propto$ con el que se puede resolver el problema con menos errores que utilizando la estrategia de PSO pura, la segunda parte de las pruebas consistió en reducir el número de vueltas o generaciones de corrida del algoritmo, hasta encontrar el punto crítico en el que el problema se resolviera en menos tiempo que utilizando estrategias basadas solamente en PSO o en un AG puro.

3.4.3 Determinación de los parámetros de las pruebas

Para llevar a cabo las pruebas se utilizaron las funciones mencionadas anteriormente con un valor D=30, es decir con espacios de búsqueda de 30 dimensiones, los cuáles han sido ampliamente utilizados cómo un estándar de un problema altamente complejo de resolver. [2,3]

Para la parte de PSO las constantes C_1, C_2 = 1.62, y w = 0.8 han sido sugeridas por obtener buenos resultados [4, 28-29], así como la constante v_i = 10% y un periodo migratorio de 10% para un tamaño del enjambre de 20 partículas [2].

Para la parte del AG los parámetros fueron obtenidos empíricamente mediante pruebas preliminares para conseguir una mayor diversidad y evitar caer en óptimos locales; un tamaño de torneo de 2, una probabilidad de cruce de 60%, una probabilidad de mutación de 30% y un periodo migratorio de 10% para una población de 160 individuos, resultaron constantes satisfactorias.

Se utilizaron 32 subpoblaciones o procesos (uno por cada núcleo, 16 computadoras con 2 núcleos cada una), un error máximo permitido de $1x10^{-6}$ y un tamaño de muestra de 10,000.

3.4.4 Resultados

Para las funciones mencionadas con anterioridad con D=30, se encontraron primeramente los siguientes valores (tabla 1):

Función	∝
Schwefel	5%
Rastrigin	20%
Ackley	5%
Weierstrass	(No se mejora el resultado de PSO puro)

Tabla 1. Menor valor de ∝ con el que se puede resolver.

Y después se obtuvieron los resultados que se pueden ver para cada función, Schwefel (tabla 2), Rastrigin (tabla 3), Ackley (tabla 4)

∝	Generaciones	Porcentaje de éxito	Error Máximo Permitido	Tiempo de ejecución
5 %	40’000	100%	$1x10^{-5}$	5,77095 s
5 %	45’000	100%	$1x10^{-6}$	**6,4051 s**

Tabla 2. Resultados para 10’000 muestras, de Schwefel en 30 dimensiones

∝	Generaciones	Porcentaje de éxito	Error Máximo Permitido	Tiempo de ejecución
20 %	2’500	99.30%	$1x10^{-6}$	0,5385 s
20 %	3’000	100%	$1x10^{-6}$	**0,5938 s**
25 %	2’500	99.90%	$1x10^{-6}$	0,5740 s

Tabla 3. Resultados para 10’000 muestras, de Rastrigin en 30 dimensiones

∝	Generaciones	Porcentaje de éxito	Error Máximo Permitido	Tiempo de ejecución
5 %	3’500	23%	$1x10^{-6}$	0,5763 s
5 %	4’000	100%	$1x10^{-6}$	**0,6697 s**

Tabla 4. Resultados para 10’000 muestras, de Ackley en 30 dimensiones

3.4.5 Conclusiones

Se logró desarrollar el algoritmo de optimización basado en los algoritmos genéticos y en optimización por enjambre de partículas, sacando provecho de lo mejor de ambas estrategias.

Se implementó sin ninguna clase de contratiempos el algoritmo en lenguaje de programación C, en una arquitectura paralela de bajo costo (un clúster de computadoras perteneciente a la división de estudios de posgrado e investigación del Instituto Tecnológico de La Paz).

Se implementaron además cabeceras que permiten utilizar dicha implementación para probar cualquier clase de función sin necesidad de ajustar directamente el código fuente del programa, y se añadieron cabeceras para hasta once funciones de la batería de pruebas CEC05.

Debido a que el programa genera documentos con estadísticas acerca de sus resultados, resultó sencillo comparar su desempeño con el de trabajos previos; el algoritmo híbrido fue incapaz de resolver Weierstrass en menos de 15 segundos, de acuerdo a trabajos como [3] el AG es muy malo resolviendo dicha función, de

ahí que la estrategia híbrida no sea capaz de mejorar los resultados obtenidos utilizando solamente PSO.

En la tabla 5 se ve que para el resto de las funciones, los resultados son mejores que los obtenidos en otros trabajos utilizando estrategias basadas solamente en PSO o AG, comparando los resultados obtenidos en [2] utilizando un clúster similar de 16 computadoras y en [3] utilizando una estrategia de paralelización distinta (una arquitectura CUDA):

Función	CUDA PSO	CUDA AG	Clúster PSO	Clúster Híbrido
Ackley	1,02 s	25,08 s	2,00 s	0,6697 s
Rastrigin	5,89 s	4,33 s	8,20 s	0,5938 s
Schwefel	5,38 s *	>47 s	Sin datos	6,4051 s
Weierstrass	5,86 s *	>320 s	Sin datos	>14 s

Tabla 5. Algoritmo Híbrido vs otras estrategias

Hay que considerar que en [3] las funciones Ackley y Weierstrass se consideraron resueltas, tolerando errores mayores a $1x10^{-6}$ ($2.5x10^{-6}$ y $5.7x10^{-6}$ respectivamente).

3.5 Trabajo Futuro

Los resultados conseguidos son más eficientes (mejores en tiempo y en resultados) que los que se han obtenido utilizando clústers de computadoras, e incluso mejores para algunas funciones que utilizando otras estrategias de paralelización como CUDA, con estrategias puras de optimización (sólo AG o PSO), sin embargo sería muy interesante implementar una estrategia hibrida de este tipo en arquitecturas de paralelización distintas como son los GPU's, que recientemente han obtenido resultados mejores en relación a los clústers.

Si bien los resultados ya son prometedores, queda para futuras investigaciones probar esta estrategia en el resto de las funciones de la batería de pruebas del CEC2005, para obtener conclusiones definitivas; otra cuestión pendiente para trabajar en un futuro sería probar esta estrategia hibrida en problemas que modelen cuestiones de la vida real.

4. Referencias

[1] Sugatan, P.N. et al “Problem Definitions and Evaluation Criteria for the CEC 2005 Special Session on Real-Parameter Optimization”, KanGAL, Tech. Rep 2005005, Nanyang Technological University, Singapore., Mayo 2005.

[2] Castro, M.A. Morales, J.A. Castro, I. Castro, J.A. and Cárdenas L.A., “Distributed Particle Swarm Optimization Using Clusters and GPGPU”, enviado y aceptado para publicación al I Congreso Internacional de Ingeniería Electrónica y Computación 2011. Minatitlán, Veracruz, México.

[3] Castro, I. “Paralelización de Algoritmos de Optimización Basados en poblaciones por medio de GPGPU”, Tesis para obtener el grado de maestro en sistemas computacionales. La Paz BCS, Julio 2011.

[4] Clerc, M. The Swarm and the Queen: Towards a Deterministic and Adaptive Particle Swarm Optimization. En Proceedings or the Congress on Evolutionary Computation, páginas 1951-1957, Washington DC, USA, Julio 1999, IEEE Service Center, Piscataway, NJ.

[5] Rosete, A. “Una Solución Flexible y Eficiente para el Trazado de Grafos Basada en el Escalador de Colinas Estocástico” Tesis presentada en opción al grado científico de Doctor en Ciencias Técnicas. La Habana. 2000.

[6] Holland, J.H., Adaptation in Natural and Artificial Systems. (1975) 6 ed. 2001, Michigan: MIT Press. 205.

[7] Kennedy, J. and Eberhart, R. (1995), “Particle Swarm Optimization”, Proc. 1995 IEEE Intl. Conf. on Neural Networks, pp. 1942-1948, IEEE Press.

[8] Russel S. and Norvig P. “Inteligencia Artificial, un enfoque moderno”, páginas 2-33, Person Education, Madrid España 2004.

[9] Rich E., Knight K. 1994. Inteligencia Artificial. Mc GrawHill. 2da. Edición. United States.

[10] Franco J.A. “Un Algoritmo Basado en la Optimización por Enjambre de Partículas para el problema de Asignación Axial 3-Dimensional” Tesis para obtener el grado de maestro en sistemas computacionales. La Paz BCS, Septiembre 2011.

[11] Nowostawski, M. and Poli R. Parallel Genetic Algorithm Taxonomy. Third International Conference of Knowledge-Based Intelligent Information Engineering Systems, 1999.

[12] Granada, U.d. Soft Computing and Intelligent Information Systems. 2008; Available from: http://sci2s.ugr.es/.

[13] Eberhart, R.C. and Shi Y. 2001. Particle Swarm Optimization: Developments, Applications and Resources. Evolutionary Computation 2001. Vol. 1. 81-86 p.

[14] Clerc, M. The Swarm and the Queen: Towards a Deterministic and Adaptive Particle Swarm Optimization. En Proceedings or the Congress on Evolutionary Computation, páginas 1951-1957, Washington DC, USA, Julio 1999, IEEE Service Center, Piscataway, NJ.

[15] Kennedy, J.. Stereotyping: Improving Particle Swarm Performance With Cluster Analysis. En Proceedings of the Congress on Evolutionary Computing, páginas 1507-1512, San Diego, USA, 2000. IEEE Service Center, Piscataway, NJ.

[16] Waintraub, M., Schirru, R., Pereira, C. Multiprocessor modeling of Particle Swarm Optimization applied to nuclear engineering problems, Progress in Nuclear Engineering 51, 680-688, 2009

[17] Clerc, M. Particle Swarm Optimization. Capítulo 15 páginas 195-199. 1st English de. Newport Beach ISTE Ltd, USA, 2006.

[18] Petcu A. and Faltings B. "A Hybrid of Inference and Local Search for Distributed Combinatorial Optimization" EPFL, Switzerland 2007

[19] Elhossini A. et al. "Strength Pareto Particle Swarm Optimization and Hybrid EA-PSO forMulti-Objective Optimization" School of Engineering, University of Guelph, Guelph, ON, N1G 2W1, Canada 2010 by the Massachusetts Institute of Technology

[20] Martinez T. and Behar A. "Estrategia Híbrida AG-SIMPLEX para la identificación experimental de sistemas" Departamento de Control Automático, Instituto de Cibernética, Matemática y Física (ICIMAF) CIE2011

[21] Hongfeng X. et al. "Large Scale Function Optimization or High-Dimension Function Optimization in Large Using Simplex-based Genetic Algorithm" School of Information Science and Engineering, Central South University, Shangai China 2009

[22] Liu C. et al. "Image based Reconstruction using Hybrid Optimization of Simulated Annealing and Genetic Algorithm" Shanghai University June 2009

[23] Zhou Q. et. al "A Hybrid Optimization Algorithm for the Job-shop Scheduling Problem" Department of Computer Science and Technology Chuzhou University, Chuzhou, China, June 2009

[24] K. Weinert et al. "On the Use of Problem-Specific Candidate Generators for the Hybrid Optimization of Multi-Objective Production Engineering Problems" TU Dortmund University, Germany 2009 by the Massachusetts Institute of Technology

[25] Mohan V. and Mala J. "Quality Improvement and Optimization of Test Cases – A Hybrid Genetic Algo-rithm Based Approach" ACM SIGSOFT Software Engineering May 2010

[26] Castro Liera, M. A. “Un algoritmo Genético Distribuido con aplicación en la identificación difusa de un proceso fermentativo”, Tesis presentada en opción al grado científico de Doctor en Ciencias Técnicas, Universidad Central “Marta Abreu” de las villas, Santa Clara Cuba 2009

[27] Sanders, J. Y Kandrot, E. Cuda by Example. An Introduction to General-Purpose GPU Programming. 1 Ed. Ann Harbor, Michigan, Pearson, 290 páginas.

[28] Van de Berg, F. “An Analysis of Particle Swarm Optimizers”, PhD Dissertation Faculty of Agricultural and Natural Science, Pretoria University, Pretoria South Africa, November 2001.

[29] Mussi, L Daolio, F and Cagnoni, S., “GPU-based Road Sign Detection Using Particle Swarm Optimization”, 9th International Conference on Intelligent Systems Design and Applications, 2009

Anexo: Código fuente del programa

1. Proceso Maestro

```
//            mixto-m.c (Proceso Maestro)
//
//            Copyrigth (c) 2012 Joel Artemio Morales Viscaya : jaamoon@hotmail.es
//                                    Marco Antonio Castro Liera : mcastroliera@gmail.com
//
//     This program is free software; you can redistribute it and/or modify
//     it under the terms of the GNU General Public License as published by
//     the Free Software Foundation; either version 2 of the License, or
//     (at your option) any later version.
//
//     This program is distributed in the hope that it will be useful,
//     but WITHOUT ANY WARRANTY; without even the implied warranty of
//     MERCHANTABILITY or FITNESS FOR A PARTÍCULAR PURPOSE.  See the
//     GNU General Public License for more details.
//
//     You should have received a copy of the GNU General Public License
//     along with this program; if not, write to the Free Software
//     Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
//     MA 02110-1301, USA.

//Cabeceras y librerías necesarias

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>
#include "aptitud.h"
#include "pvm3.h"

//Parámetros globales

#define GMAX 2500             //Numero de ciclos del algoritmo
#define POBLACIONES 32        //Numero de subpoblaciones o procesos
#define TMUESTRA 30           //Numero de veces que se corre el programa
#define EMAX 0.0000001        //Error máximo permitido (Errores mayores se consideran prematuros)
#define PROP 0.05             //Valor de alfa (Porcentaje del total de ciclos que se ejecuta el AG)

//Parámetros del PSO

#define TENJ 20               //Tamaño del enjambre
#define C1 1.62               //Confianza cognitiva
#define C2 1.62               //Confianza social
#define W 0.8                 //Constante de amortiguamiento
#define FVi 10                //Constante de la velocidad inicial
#define PSOpm 0.10            //Periodo de migración (Porcentaje de GMAX*PROP)

//Parámetros del AG

#define MU 160                //Tamaño de la población
#define Z 2                   //Tamaño del torneo (para la selección)
#define PC 0.60               //Probabilidad de cruce
#define PM 0.30               //Probabilidad de mutación
#define AGrm 0.03             //Razón de migración [Porcentaje de individuos a migrar (de MU)]
#define AGpm 0.01             //Periodo de migración (Porcentaje de GMAX*PROP)

main()
{
        //Declarar archivos

        FILE *informe;
        FILE *tiempos;

        //Estructuras para leer la hora y zona horaria.
```

```
struct timeval tv;
struct timezone tz;

//Declaración de variables

long si,ui,sf,uf;                       //Variables para el control del tiempo
double tiempo;                          //Variable para el almacenamiento del tiempo de ejecución
char salida[256];                       //Cadena de caracteres para la creación del archivo
int mytid;                              //Mi identificador de tarea (task id)
int tids[POBLACIONES];                  //Identificadores de tarea esclavos
int n,nproc,numt,i,j,k,l,r,d,msgtype,nhost,narch,prematuros,sendint;        //Variables auxiliares
double resultado,mresultado,mx[DIMENSIONES],eprom,error,sendouble;          //Para la estructura de PVM
double xin[DIMENSIONES];
struct pvmhostinfo * hostp;
long evals,getlong;

//Inicializar el numero de prematuros y errores

prematuros = 0;
error = 0;

//Calcular variables para el control de la migración

int MPPSO = GMAX*PROP*PSOpm;
int MPAG = GMAX*PROP*AGpm;
int MR = MU*AGrm;

//Crear archivos

sprintf(salida,"MIXTO:%f-%s-%02d-%03d-%05d.txt",PROP,FUNCION,POBLACIONES,DIMENSIONES,GMAX);
informe = fopen(salida,"w");
sprintf(salida,"t-MIXTO:%f-%s-%02d-%03d-%05d.txt",PROP,FUNCION,POBLACIONES,DIMENSIONES,GMAX);
tiempos = fopen(salida,"w");

for(j=0;j<TMUESTRA;j++)                 //Ciclo principal
{
        gettimeofday(&tv,&tz);          //Obtener la hora, fecha y zona horaria
        si = tv.tv_sec;                 //segundos
        ui = tv.tv_usec;                //microsegundos
        mytid = pvm_mytid();            //Agregarse a la pvm
        pvm_config(&nhost,&narch,&hostp);   //Determinar cuántos host va tener la pvm
        nproc = POBLACIONES;            //Iniciar un proceso para cada población

        //Iniciar los esclavos

        numt = pvm_spawn("mixto-esc", (char**)0,0,"",nproc,tids);       //Engendrar los esclavos

        //Si engendraron menos que el número de poblaciones...

        if (numt<nproc)
        {
                printf("\nError Creando Esclavos, codigo de error: \n");  //Desplegar mensaje de error
                for(i=numt; i<nproc; i++)
                {
                        printf("TID %d\n",i,tids[i]);   //Desplegar que procesos no se engendraron
                }
                for(i=0; i<nproc; i++)
                {
                        pvm_kill(tids[i]);      //Quitar de PVM los que se alcanzaron a engendrar
                }
                pvm_exit();             //Salir de PVM
                exit(1);                //Salir del programa
        }

        //Enviar Parámetros a cada esclavo

        msgtype=5;
        for(i=0;i<nproc;i++)
        {
```

```
                pvm_initsend(PvmDataDefault);
                sendint=TENJ;                   //Tamaño del enjambre
                pvm_pkint(&sendint,1,1);
                sendint=MPAG;                   //Periodo migratorio AG
                pvm_pkint(&sendint,1,1);
                sendint=MPPSO;                  //Periodo migratorio PSO
                pvm_pkint(&sendint,1,1);
                sendint=GMAX;                   //Generación maxima
                pvm_pkint(&sendint,1,1);
                sendouble=C1;                   //peso C1
                pvm_pkdouble(&sendouble,1,1);
                sendouble=C2;                   //peso C2
                pvm_pkdouble(&sendouble,1,1);
                sendouble=W;                    //peso W
                pvm_pkdouble(&sendouble,1,1);
                sendouble=FVi;                  //peso FVi
                pvm_pkdouble(&sendouble,1,1);
                sendint = MU;                   //Tamaño de la población
                pvm_pkint(&sendint,1,1);
                sendint = Z;                    //Torneo
                pvm_pkint(&sendint,1,1);
                sendint = MR;                   //Razón de Migración
                pvm_pkint(&sendint,1,1);
                sendouble = PC;                 //Probabilidad de cruce
                pvm_pkdouble(&sendouble,1,1);
                sendouble = PM;                 //Probabilidad de Mutación
                pvm_pkdouble(&sendouble,1,1);
                sendouble =          PROP;      //Alfa o constante de proporcionalidad
                pvm_pkdouble(&sendouble,1,1);
                sendint = i % 2;                //Si es par o impar
                pvm_pkint(&sendint,1,1);
                sendint = tids[(i+1)%nproc];            //Calcular el destino
                 pvm_pkint(&sendint,1,1);
                pvm_send(tids[i],msgtype);              //Enviar Parámetros
        }
evals=0;        //Reseteamos las evaluaciones
        for(i=0;i<nproc;++i)            //Ciclo para recibir los resultados de los n' procesos
        {
                pvm_recv(-1,msgtype);
                pvm_upkdouble(&resultado,1,1);
                resultado = -resultado;                 //Cambio de signo porque es minimización
                printf("%f\n",resultado);
                pvm_upkdouble(xin,DIMENSIONES,1);
                pvm_upklong(&getlong,1,1);
                evals+=getlong;                 //Acumulamos las evaluaciones de cada proceso
                        if(i==0 || resultado < mresultado) //Si el resultado recibido es mejor que el anterior
                        {
                                mresultado = resultado;                 //Pasa a ser considerado el mejor
                                for(d=0;d<DIMENSIONES;d++)
                                {
                                        mx[d] = xin[d];                 //Y se importan sus coordenadas
                                }
                        }
        }
        if(mresultado>EMAX)
        {
                prematuros++;       //Si el error es mayor al umbral EMAX, se considera prematuro
        }
        else
        {
                error += mresultado;            //Si no, se acumula el error
        }
        //Usamos con los segundos y milisegundos transcurridos para calcular el tiempo
        gettimeofday(&tv,&tz);
        sf = tv.tv_sec;
        uf = tv.tv_usec;
        sf -= si;
        if(ui>uf)
        {
                sf--;
```

```
                        uf+=1000000;
                }
                uf-=ui;
                tiempo = (double) sf;
                tiempo += (double) uf / 1000000;
                printf("%.06f\t%lu\n",tiempo,evals);        //Escribimos en el archivo el tiempo total de corrida
                fprintf(tiempos,"%.06f\t%lu\n",tiempo,evals);
                printf("%d\tf(X)=%f",j+1,mresultado);
                for (d=0;d<DIMENSIONES;d++)
                {
                        printf("\t%f",mx[d]);
                }
                printf("\n");
                fprintf(informe,"%10f\n",mresultado);      //Desplegamos el resultado
                pvm_exit();
        }
        eprom=error/(TMUESTRA-prematuros);                //Calculamos el error promedio (sin prematuros)
        fprintf(informe,"PREMATUROS ...: %d", prematuros);
        fprintf(informe,"ERROR PROMEDIO ...: %d", eprom);          //Terminamos el archivo escribiendo

        //Cerramos los archivos

        fclose(informe);
        fclose(tiempos);
}

//Fin del programa maestro
```

2. Proceso Esclavo

```
//              mixto-esc.c (Proceso Esclavo)
//

//Cabeceras y librerías necesarias

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>
#include <gsl/gsl_rng.h>
#include "aptitud.h"
#include "pvm3.h"
#define UNI gsl_rng_uniform_pos(r)

//Variables Globales

int GMAX,MPPSO,MPAG;
int destino, impar, msgtype;
double resultado, enviadoble,PROP;
double posiciónes[DIMENSIONES];
gsl_rng * r;

//Variables del PSO

int TENJ;
double W, C1, C2, FVi;

//Variables Globales del AG

int  MU, GMAX, Z, MR;
double PM, PC, CZ;

//Estructura individuo (AG)

struct individuo
 {
   double x[DIMENSIONES];          //Posición
   double ap;                      //Aptitud
 };
            struct individuo * p;          //Población de individuos
            struct individuo * psig;       //Población siguiente de individuos
            struct individuo * aux;        //Estructura auxiliar para intercambiar individuos
            struct individuo mejant;       //Mejor individuo de la población anterior
            int    * migrables;            //Individuos Migrables
            int    mejorp=0;               //Posición del mejor individuos

//Método que recorre toda la población en busca del mejor individuo

void buscamejor(void)
 {
   int i;
   for(i=0;i<MU;i++)
     if(p[i].ap>p[mejorp].ap)
       mejorp=i;
 }

//Método que busca los 'MR' individuos a migrar

void busca_migrables(void)
 {
   int i,j;
   for(j=0;j<MR;j++)
     migrables[j]=0;
   for(i=1;i<MU;i++)
     for(j=0;j<MR;j++)
       if(p[i].ap>p[migrables[j]].ap)
         {
```

```
          migrables[j]=i;
          j=MR;
        }
  }

//Método que busca los peores individuos (Para compararlos y/o reemplazarlos con los migrables)

void busca_peores(void)
  {
    int i,j;
    for(j=0;j<MR;j++)
      migrables[j]=0;
    for(i=1;i<MU;i++)
      for(j=0;j<MR;j++)
        if(p[i].ap<p[migrables[j]].ap)
          {
            migrables[j]=i;
            j=MR;
          }
  }

//Método que simula la selección natural (Que individuos de 'p', pasan a 'psig')

void seleccion(void)
  {
    int i,j,k,l,d;
    for(i=0;i<MU;i++)
     {
       k = UNI*MU;
       for(j=1;j<Z;j++)
         {
           l = UNI*MU;
           if(p[k].ap<p[l].ap)
             k = l;
         }
       for(d=0;d<DIMENSIONES;d++)
         psig[i].x[d] = p[k].x[d];
       psig[i].ap = p[k].ap;
     }
  }

//Método que simula el cruce [Que genera nuevos individuos a partir de los ya existentes (hijos)]

void cruce(void)
  {
    double xnew1,xnew2;
    int i,j,k,d;
    float lambda;
    lambda = UNI;
    for(i=0;i<CZ;i++)
     {
        j=UNI*MU;
        k=UNI*MU;
        for(d=0;d<DIMENSIONES;d++)
        {
          xnew1 = psig[j].x[d]*lambda+psig[k].x[d]*(1-lambda);
          xnew2 = psig[j].x[d]*(1-lambda)+psig[k].x[d]*lambda;
          psig[j].x[d] = xnew1;
          psig[k].x[d] = xnew2;
        }
        psig[j].ap = aptitud(psig[j].x);
        psig[k].ap = aptitud(psig[k].x);
     }
  }

//Método que simula la mutación de un individuo (Genera una variación aleatoria de una de las variables)

void mutacion(int g)
  {
    float m,h;
```

```
   int i,gen;
   for (i=0;i<MU;i++)
    {
      m = UNI;                              //Para decidir si se aplica el operador en base a PM
      if(m<=PM)
       {
          gen = UNI*DIMENSIONES;        //Para decidir cual gen mutar x[0], x[1]....x[gen]
          h = UNI;                              //Incremento o decremento
          if(h<0.5)
            psig[i].x[gen] = psig[i].x[gen]+(UNI*(XMAX-psig[i].x[gen]))*(1-g/GMAX);
          else
            psig[i].x[gen] = psig[i].x[gen]-(UNI*(psig[i].x[gen]+XMAX))*(1-g/GMAX);
          psig[i].ap = aptitud(psig[i].x);
       }
    }
 }

//Método que lleva a cabo la migración de individuos

void enviaAG(void)
 {
    int i;
    double resultado;
    busca_migrables();          //Migrar los MR mejores
    for(i=0;i<MR;i++)
     {
        pvm_initsend(PvmDataDefault);
        resultado = p[migrables[i]].ap;
        pvm_pkdouble(&resultado,1,1);
        pvm_pkdouble(p[migrables[i]].x,DIMENSIONES,1);
        pvm_send(destino,msgtype);
     }
 }

//Método que recibe a los individuos migrados desde otra población

void recibeAG(void)
 {
    int i;
    double result;
    busca_peores();
    for(i=0;i<MR;i++)
     {
        pvm_recv(-1,msgtype);
        pvm_upkdouble(&result,1,1);
        p[migrables[i]].ap = result;
        pvm_upkdouble(p[migrables[i]].x,DIMENSIONES,1);
     }
 }

//Estructura partícula (PSO)

struct partícula
{
            double x[DIMENSIONES];              //Posición
            double v[DIMENSIONES];              //Velocidad
            double ap;                          //Aptitud
            double p[DIMENSIONES];              //Mejor posición
            double pap;                         //Mejor aptitud
};

struct partícula * P2;          //Enjambre

struct posición
{
            double x[DIMENSIONES];              //Posición
            double ap;                          //Aptitud
} mejor;            //Estructura donde se va guardar la mejor posición del enjambre

//Método que calcula la velocidad con la que se va mover una partícula
```

```
void velocidad(int i)
{
   int d;
      for(d=0;d<DIMENSIONES;d++)
       {
         P2[i].v[d] = W*P2[i].v[d] + C1*UNI*(P2[i].p[d]-P2[i].x[d]) + C2*UNI*(mejor.x[d]-P2[i].x[d]);
         if(P2[i].v[d] > XMAX)
            P2[i].v[d] = XMAX;
         else
           if(P2[i].v[d] < -XMAX)
             P2[i].v[d] = -XMAX;
       }
}

//Método que calcula la nueva posición de una partícula (Posición nueva = Posición anterior + velocidad)

void posición(int i)
{
   int d;
   for(d=0;d<DIMENSIONES;d++)
                {
                  P2[i].x[d] += P2[i].v[d];
                  if(P2[i].x[d] > XMAX)
                                P2[i].x[d] = XMAX;
                  else
                                if(P2[i].x[d] < -XMAX)
                                  P2[i].x[d] = -XMAX;
                }
   P2[i].ap = aptitud(P2[i].x);
                if(P2[i].ap > P2[i].pap)
                {
                  P2[i].pap = P2[i].ap;
                  for(d=0;d<DIMENSIONES;d++)
                    {
                                P2[i].p[d]=P2[i].x[d];
                    }
                 //Preguntamos si es la mejor global
                  if(P2[i].pap > mejor.ap)
                                {
                                  mejor.ap = P2[i].pap;  //Si la aptitud de P2 es la mejor, pasa a ser la mejor partícula del enjambre
                                  for(d=0;d<DIMENSIONES;d++)
                                    {
                                                mejor.x[d]=P2[i].p[d];
                                    }
                                }
                }
}

//Método que migra la mejor partícula hacia el enjambre 'destino'

void envia(void)
 {
                //Enviar la posición del mejor

                pvm_initsend(PvmDataDefault);                   //Se envía el valor default
                enviadoble=mejor.ap;
                pvm_pkdouble(&enviadoble,1,1);                  //Se empaqueta la aptitud del mejor
                pvm_pkdouble(mejor.x,DIMENSIONES,1);            //Se empaqueta la posición del mejor
                pvm_send(destino,msgtype);                      //Se envía hacia otro enjambre

 }

//Método que recibe la mejor posición de otro enjambre

void recibe(void)
 {
                int d;
                pvm_recv(-1,msgtype);
                pvm_upkdouble(&resultado,1,1);                  //Se recibe la aptitud de otro enjambre
```

```
            pvm_upkdouble(posiciónes,DIMENSIONES,1);    //Se recibe la posición de la mejor aptitud
            if(resultado > mejor.ap)                    //Si su mejor posición es mejor...
             {
                        mejor.ap = resultado;           //Reemplaza la mejor posición actual
                        for(d=0;d<DIMENSIONES;d++)
                         {
                           mejor.x[d] = posiciónes[d];  //Y sus coordenadas
                         }
             }
 }

//Método que inicializa las partículas del enjambre con las posiciones finales de la población y velocidades aleatorias

void inicializa(void)
{
            int i,d;
            float aux;
            for(i=0;i<TENJ;i++)
            {
                        for(d=0;d<DIMENSIONES;d++)
                        {
                          aux = UNI;
                          #ifdef ASIMETRICO
                                    p[i].x[d] = UNI*XMAX;       //UNI distribución uniforme (0,1)
                          #else
                                    if(UNI < 0.5)
                                      p[i].x[d] = UNI*XMAX;
                                    else
                                      p[i].x[d] = -(UNI*XMAX);
                          #endif
                       if(aux>0.5)
                                    P2[i].v[d] = UNI*XMAX/FVi;          //Velocidad inicial
                       else
                                    P2[i].v[d] = -UNI*XMAX/FVi;         //Velocidad inicial
                        }
                        P2[i].pap = P2[i].ap = aptitud(P2[i].x);    //La mejor aptitud inicial es la primer aptitud
            }
            mejor.ap = p[mejorp].ap;                    //Colocamos como mejor partícula la más apta
            for(d=0;d<DIMENSIONES;d++)
            {
                        mejor.x[d]=p[mejorp].x[d];
            }
}

main()
{
            int mytid;              //Mi identificador de tarea (task id)
            int master;             //Parámetro de PVM

            //Declaración de variables

            int i,j,k,d,g,periodo,veces;

            //Estructuras para leer la hora y zona horaria.

            struct timeval tv;
            struct timezone tz;

            //Ingresar a la PVM

            mytid = pvm_mytid();            //Se nos asigna un identificador de tarea
            msgtype = 5;

            //Recibir Parametros del PSO (enviados desde mixto-m)

            pvm_recv(-1, msgtype);
            pvm_upkint(&TENJ,1,1);                  //Tamaño del enjambre
            pvm_upkint(&MPAG,1,1);                  //Periodo migratorio AG
            pvm_upkint(&MPPSO,1,1);                 //Periodo migratorio PSO
            pvm_upkint(&GMAX,1,1);                  //Número de generaciones
```

```
pvm_upkdouble(&C1,1,1);                 //Peso c1
pvm_upkdouble(&C2,1,1);                 //Peso c2
pvm_upkdouble(&W,1,1);                  //Peso W
pvm_upkdouble(&FVi,1,1);                //Peso de Vi
pvm_upkint(&MU,1,1);                    //Tamaño de población
pvm_upkint(&Z,1,1);                     //Tamaño del torneo
pvm_upkint(&MR,1,1);                    //Periodo de migración
pvm_upkdouble(&PC,1,1);                 //Probabilidad de cruce
pvm_upkdouble(&PM,1,1);                 //Probabilidad de mutación
pvm_upkdouble(&PROP,1,1);               //alfa o constante de proporcionalidad
pvm_upkint(&impar,1,1);                 //Paridad
pvm_upkint(&destino,1,1);               //Destino

//Pasos previos

CZ = MU*PC/2;                           //Cantidad de cruces: cruzados=MU*PC,
                                        //Se divide entre dos pues hay dos hijos
                                        //por cruce
veces = (GMAX*PROP)/MPAG;               //El número de veces que se ejecutara la migración en AG
p = malloc(sizeof(struct individuo)*MU);        //Gestiona memoria a la p actual
psig = malloc(sizeof(struct individuo)*MU);     //Gestiona memoria a la p siguiente
migrables = malloc(sizeof(int)*MR);             //Gestiona la memoria de los índices de los migrables
gettimeofday(&tv,&tz);                          //Se obtiene la hora del sistema
srandom(tv.tv_usec);                            //Inicialización del generador de
r =gsl_rng_alloc(gsl_rng_mt19937);              //pseudo-aleatorios
gsl_rng_set(r,random());
evaluaciones = 0;                               //Variable para llevar un control de las veces que se ejecuta la
                                                //función de aptitud
for(i=0;i<MU;i++)               //Generación aleatoria de la primera población
{
        for(d=0;d<DIMENSIONES;d++)      //valores aleatorios para cada dimensión del vector x
    {
                #ifdef ASIMETRICO
                        p[i].x[d] = UNI*XMAX;           //UNI distribución uniforme (0,1)
                #else
                        if(UNI < 0.5)
                           p[i].x[d] = UNI*XMAX;
                        else
                           p[i].x[d] = -(UNI*XMAX);
                #endif
        }
        p[i].ap = aptitud(p[i].x);      //Se calcula la aptitud inicial
}

//Ciclo del AG (Primera parte)

for(k=0;k<veces;k++)
{
        for(g=0;g<MPAG;g++)
        {
                buscamejor();           //Buscamos al mejor individuo
                for(d=0;d<DIMENSIONES;d++)          //Guardamos al mejor de la generación anterior
                        mejant.x[d] = p[mejorp].x[d];
                mejant.ap = p[mejorp].ap;

                //Inicio de los operadores genéticos

                seleccion();
                cruce();
                mutacion(g+(MPAG*k));

                //'p' se vuelve 'psig'

                aux = p;
                p = psig;
                psig = aux;
                buscamejor();

                if(p[mejorp].ap < mejant.ap)            //Si el mejor de p-1 es mejor que el
                {                                       //actual se incluye
```

```
                                        for(d=0;d<DIMENSIONES;d++)
                                                p[MU-1].x[d] = mejant.x[d];
                                        p[MU-1].ap = mejant.ap;
                                }
                        }

                        //Inicio de la migración

                        if(impar)
                          {
                            enviaAG();
                            recibeAG();
                          }
                        else
                          {
                            recibeAG();
                            enviaAG();
                          }
        }

buscamejor();

        //Una vez terminado el ciclo del AG, empezamos con la 2da parte (PSO)
        free(psig);
        free(migrables);
        P2 = malloc(sizeof(struct partícula)*TENJ);          //Gestiona memoria para el enjambre
        inicializa();                   //inicializa el enjambre con las coordenadas de los individuos de la población
        free(p);

        veces = GMAX*(1.0-PROP) / MPPSO;                //Cuantas veces se va migrar
for(k=0;k<veces;k++)
        {
                        //Para cada generación
                        for (g=0;g<MPPSO;g++)
                        {
                                //Para cada partícula calculamos su movimiento
                                for (i=0;i<TENJ;i++)
                                {
                                        velocidad(i);
                                        posición(i);
                                }

                        }

                        //Inicio de la migración

                        if(impar)
        {
          envia();
          recibe();
        }
        else
        {
          recibe();
          envia();
        }
        }

        //Enviar al maestro la mejor partícula del enjambre

        master = pvm_parent();
        pvm_initsend(PvmDataDefault);                   //Se envía el valor default
        resultado=mejor.ap;
        pvm_pkdouble(&resultado,1,1);                   //Se empaqueta la aptitud del mejor
        pvm_pkdouble(mejor.x,DIMENSIONES,1);            //Se empaqueta la posición del mejor
        pvm_pklong(&evaluaciones,1,1);
        pvm_send(master,msgtype);                       //Se envía todo al proceso maestro

        //Liberamos la memoria asignada
```

```
        gsl_rng_free(r);
        free(P2);
        pvm_exit();          //Se saca al proceso de la PVM, para terminar
}

//Fin del proceso esclavo
```

3. Cabeceras (Aptitud.h)

```
//aptitud.h

#define DIMENSIONES 30
#define FBIAS 7

double BIAS[DIMENSIONES] = {0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2,3,4};

//Función de aptitud (descomentar una)

//#include "ackley.h"
//#include "eliptical.h"
//#include "spherical.h"
//#include "griewank.h"
//#include "quadric.h"
#include "rastrigin.h"
//#include "rosenbrock.h"
//#include "scaffer.h"
//#include "schwefel.h"
//#include "schwefel-noise.h"
//#include "weierstrass.h"
```

4 Funciones de la batería de pruebas CEC2005

```
//Función 1, Pag 4 CEC05

#define FUNCION "spherical"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266625
#define XMAX 100

long evaluaciones = 0;
//función de aptitud
double aptitud(double x[DIMENSIONES])
 {
   int i;
   double z,res;
   evaluaciones++;
   res = FBIAS;
   for (i=0; i<DIMENSIONES; i++)
   {
      z = x[i]+BIAS[i];
      res  += z*z;
   }
   return (-res);
 }

//Función 2, Pag 5 CEC05

#define FUNCION "schwefel"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266625
#define XMAX 100

long evaluaciones = 0;
//funcion de aptitud
double aptitud(double x[DIMENSIONES])
 {
   int i, j;
   long double z, sum1, sum2;
   sum1 = FBIAS;
   evaluaciones++;
   for (i=0; i<DIMENSIONES; i++)
    {
      sum2 = 0.0;
      for (j=0; j<=i; j++)
        {
          z = x[j]+BIAS[j];
          sum2 += z;
        }
      sum1 += sum2*sum2;
    }
   return (-sum1);
 }

//Función 3, Pag 6 CEC05

#define FUNCION "eliptical"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266625
#define XMAX 100

long evaluaciones = 0;
//función de aptitud
double aptitud(double x[DIMENSIONES])
 {
   int i;
   double z,division,coef,res;
   evaluaciones++;
   res = FBIAS;
   for (i=0; i<DIMENSIONES; i++)
   {
```

```
      z = x[i]+BIAS[i];
      division = (double) i;
      division /= (double) (DIMENSIONES-1);
      coef = pow(1000000,division);
      res  += coef*z*z;
   }
   return (-res);
}

//Función 4, Pag 7 CEC05

# include <gsl/gsl_rng.h>
# include <sys/time.h>

gsl_rng * q;

#define FUNCION "schwefel-noise"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266249
#define XMAX 100

long evaluaciones = 0;
//funcion de aptitud
double aptitud(double x[DIMENSIONES])
 {
   struct timeval tv;
   struct timezone tz;
   int i, j;
   long double z, sum1, sum2, noise;
   gettimeofday(&tv,&tz); //se obtiene la hora del sistema
   srandom(tv.tv_usec);
   q =gsl_rng_alloc(gsl_rng_mt19937);
   gsl_rng_set(q,random());
   sum1 = FBIAS;
   evaluaciones++;

   for (i=0; i<DIMENSIONES; i++)
    {
      sum2 = 0.0;
      for (j=0; j<=i; j++)
       {
         z =  x[j] + BIAS[j];
         sum2 += z;
       }
      sum1 += sum2*sum2;
    }
   sum1 *= (1 + 0.4 * gsl_rng_uniform_pos(q));
   gsl_rng_free(q);
   return (-sum1);
}

//Función 5, Pag 8 CEC05

#define FUNCION "quadric"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266249
#define XMAX 100

long evaluaciones = 0;
//función de aptitud
double aptitud(double x[DIMENSIONES])
 {
   int i,j;
   double z,sum,res;
   evaluaciones++;
   res = FBIAS;
   for (i=0; i<DIMENSIONES; i++)
   {
      sum = 0;
      for(j=0;j<=i;j++)
```

```
        {
          z = x[j]+BIAS[j];
          sum += z*z;
        }
      res  += sum;
   }
   return (-res);
}

//Función 6, Pag 9 CEC05

#define FUNCION "rosenbrock"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266625
#define XMAX 100

long evaluaciones = 0;

//función de aptitud
double aptitud(double x[DIMENSIONES])
 {
   int i;
   double z,znxt,res;
   evaluaciones++;
   res = FBIAS;
   for (i=0; i<DIMENSIONES-1; i++)
    {
      z = x[i]+BIAS[i];
      znxt = x[i+1]+BIAS[i+1];
      res += 100.0*pow(((z+1)*(z+1)-(znxt+1)),2.0) + pow(z,2.0);
    }
   return (-res);
}

//Función 7, Pag 10 CEC05

#define FUNCION "griewank"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266625
#define XMAX 600
#define ASIMETRICO

long evaluaciones = 0;

//función de aptitud
double aptitud(double x[DIMENSIONES])
{
   int i;
   long double z,s, p;
   long double res;
   evaluaciones++;
   s = 0.0;
   p = 1.0;
   for (i=0; i<DIMENSIONES; i++)
   {
      z = x[i]+BIAS[i];
      s += z*z;
      p *= cos(z/sqrt(1.0+i));
   }
   res = 1.0 + s/4000.0 - p + FBIAS;
   return (-res);
}

//Función 8, Pag 11 CEC05

#define FUNCION "ackley"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266625
#define XMAX 32
```

```
long evaluaciones = 0;
//función de aptitud
double aptitud(double x[DIMENSIONES])
 {
   int i;
   double z, sum1, sum2, res;
   evaluaciones++;
   sum1 = 0.0;
   sum2 = 0.0;
   for (i=0; i<DIMENSIONES; i++)
   {
      z = x[i]+BIAS[i];
      sum1 += z*z;
      sum2 += cos(2.0*PI*z);
   }
   sum1 = -0.2*sqrt(sum1/DIMENSIONES);
   sum2 /= DIMENSIONES;
   res = 20.0 + E - 20.0*exp(sum1) - exp(sum2) + FBIAS;
   return (-res);
 }

//Función 9, Pag 12 CEC05

#define FUNCION "rastrigin"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266625
#define XMAX 5

long evaluaciones = 0;

//función de aptitud
double aptitud(double x[DIMENSIONES])
 {
   int i;
   double z,res;
   evaluaciones++;
   res = FBIAS;
   for (i=0; i<DIMENSIONES; i++)
    {
      z = x[i]+BIAS[i];
      res += (z*z - 10.0 * cos(2.0*PI*z) + 10.0);
    }
   return (-res);
 }

//Función 10, Pag 13 CEC05

#define FUNCION "scaffer"
#define XMAX 100

long evaluaciones = 0;

//función de aptitud
double aptitud(double x[DIMENSIONES])
 {
   evaluaciones ++;
   int i;
   double z,znxt;
   double a=0;
   double b=0;
   double sum=0;
   for (i=0; i<DIMENSIONES-1; i++)
   {
      z = x[i] + BIAS[i];
      znxt = x[i+1] + BIAS[i+1];
            a=pow(sin(sqrt((z*z) + (znxt*znxt))),2) - 0.5;
            b=pow(1+0.001*((z*z)+(znxt*znxt)),2);
      sum += 0.5 + (a/b);
   }
   sum += FBIAS;
```

```
    return (-sum);
 }

//Función 11, Pag 14 CEC05

#define FUNCION "weierstrass"
#define PI 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266625
#define XMAX 0.5

long evaluaciones = 0;
//función de aptitud
double aptitud(double x[DIMENSIONES])
  {
    int i, j;
    double res;
    double sum;
    double a, b, z;
    int k_max;
    evaluaciones++;
    a = 0.5;
    b = 3.0;
    k_max = 20;
    res = FBIAS;
    for (i=0; i<DIMENSIONES; i++)
    {
        z = x[i]+BIAS[i];
        sum = 0.0;
        for (j=0; j<=k_max; j++)
        {
            sum += pow(a,j)*cos(2.0*PI*pow(b,j)*(z+0.5));
        }
        res += sum;
    }
    sum = 0.0;
    for (j=0; j<=k_max; j++)
      {
        sum += pow(a,j)*cos(2.0*PI*pow(b,j)*(0.5));
      }
    res-=DIMENSIONES*sum;
    return (-res);
}
```