Distribución equilibrada del esfuerzo de cómputo en Algoritmos Genéticos Paralelos.

Sistemas Informáticos 2005-2006

Dirigido por D. José Ignacio Hidalgo Pérez

Manuel Arquero Portero

Juan Luis Nogueras Durán

Adrián Salvador Suz

Agradecimientos

A José Ignacio Hidalgo
por su disponibilidad, apoyo,
paciencia y buen humor
y a
Francisco Fernández de Vega
por su tiempo e ideas que nos
hizo darle un nuevo enfoque
a todo esto.
También agradecemos
a Enrique de la Torre
su apoyo técnico
con la máquina.

1.- Introducción.

Los algoritmos genéticos (AGs) son métodos de búsqueda basados en los principios de la selección natural y la genética. Han sido aplicados con éxito a múltiples problemas científicos y de ingeniería. En muchas aplicaciones prácticas los AGs encuentran una buena solución en un lapso de tiempo considerable.

Hay muchos estudios para mejorar el rendimiento de los AGs y una de las alternativas más interesantes es el uso de implementaciones paralelas. El éxito de los AGs paralelos reside en la reducción del tiempo requerido para encontrar una solución aceptable en muchos problemas complejos. Los AGs trabajan con una población de soluciones independientes, lo que hace fácil su paralelización entre varios procesadores.

A pesar de su simplicidad operacional los AGs paralelos están controlados por múltiples parámetros que afectan a la eficiencia y a la calidad de las soluciones encontradas. Fijar estos parámetros correctamente, buscando un equilibrio entre ellos, es fundamental para obtener buenas soluciones rápidamente intentado no desaprovechar recursos de computación. Concretamente, algunos de los parámetros a determinar son el número y el tamaño de las poblaciones y la frecuencia de intercambio entre ellas. Habitualmente estos parámetros se obtienen tras una experimentación sistemática.

El objetivo de este estudio es servir de referencia a la hora de fijar estos parámetros y ser una guía para elegir los valores adecuados con los que obtener soluciones eficientes y de alta calidad. Este documento combina la teoría con los resultados experimentales con los que se puede apreciar el efecto de los diversos parámetros.

Para elaborar la investigación se ha desarrollado un tipo de algoritmo genético paralelo conocido como modelo de islas (Island Model) y se ha implementado usando PGApack, librería de dominio público que permite programar gran diversidad de algoritmos genéticos. Este modelo se ejecutará bajo una arquitectura de tipo SIMD con paso de mensajes que permitirá la ejecución de hasta 8 procesos (islas) en paralelo. La

comunicación entre procesos se realiza a través de MPI y el algoritmo simulará una topología en anillo.

El modelo desarrollado se aplicará al análisis de tres tipos de problemas usados en estudios previos: la función OneMax, la función multimodal y la función trampa.

Nos centraremos en equilibrar cuatro parámetros característicos de los AGs paralelos para buscar las configuraciones que nos proporcionen mejores resultados. Estos parámetros son la frecuencia de intercambio, el número de procesadores, el tamaño de la población y el operador de mutación. También se realizará un breve estudio del operador de regeneración.

En la próxima sección presenta una introducción a los algoritmos genéticos simples y paralelos en la que se describirá como trabajan los AGs y se definirán algunos términos usados a lo largo de todo el estudio. En la sección tercera se explicaran los conceptos básicos de procesamiento paralelo así como las especificaciones del hardware paralelo utilizado. La sección cuarta detallará las funciones experimentales con las que se ha efectuado la batería de pruebas. En la quinta parte se verán las librerías y los detalles de implementación utilizados. Los resultados obtenidos aparecerán en el sexto capítulo del documento y finalmente en el apartado séptimo se presentarán las conclusiones generales del trabajo realizado.

Genetic algorithms (GAs) are search methods based on the principles of natural selection and genetics. They have been applied successfully to a lot of problems in science and engineering. In many practical applications GAs are able to find good solutions in a reasonable amount of time.

There have been many studies to increase the efficiency of GAs and one of the most powerful alternatives is the use of parallel implementations. The success of parallel GAs is to reduce the time required to find a good solution in many complex problems. GAs

work with a population of independent solutions, which makes it easy to distribute the computational load among several processors.

However, despite of their operational simplicity, parallel GAs are controlled by many parameters that affect to efficiency and quality of their search. Setting these parameters correctly is crucial to obtain good solutions quickly. In particular, some of these parameters are the number and size of populations and the exchange rate between these. Typically the parameters are found by systematic experimentation.

The goal of this study is to obtain some conclusions and references to tune the parameters and provide a guide to choose their values correctly, which find efficient solutions with high quality. This document combines theory and experimental results in order to allow the user can appreciate the effect of different parameters.

For prepare the investigation, we have developed a kind of parallel GAs known as Island Model that we have implemented using PGAPACK, a public library for programming GAs. This model will run in SIMD architecture with a message passing paradigm which can be launched up to 8 processors (islands) in parallel. The communications among processors is realized with MPI and we have implemented the algorithm with a ring topology.

This model is going to be applied to the analysis of three kinds of typical problems used in previous investigations: OneMax functions, multimodal functions and trap function.

We will try to find a balance between four characteristic parameters in parallel GAs in order to get the configurations to obtain the best results. These parameters are the change frequency, the number of processors, the size of populations and the mutation operator.

The next section introduces to simples and parallel GAs. It describes the work of GAs and defines some important terminology used along the study. In the third section the basic concepts of parallel programming and the parallel hardware specifications will be explained. The fourth section will detail the experimental functions which have been used in the executions. In the fifth section will be seen the libraries and implementation details. The results will be shown in the sixth chapter of this document and in the seventh section the main conclusions will be presented.

2.- Algoritmos Genéticos.

2.1.- Algoritmos genéticos simples

Los algoritmos evolutivos (AEs) surgen a finales de los años 60 cuando John Holland planteó la posibilidad de incorporar los mecanismos naturales de selección y supervivencia a la resolución de problemas de Inteligencia Artificial. Esta investigación fue fundamentalmente académica, siendo su realización práctica en aquella época muy difícil. La aparición de computadores de grandes prestaciones y bajo coste a mediados de los 80 permite aplicar los Algoritmos Evolutivos a la resolución de ciertos problemas de ingeniería que antes eran inabordables. A partir de entonces el desarrollo de estas técnicas ha sido continuo.[6,9]

El algoritmo de Holland es un método sistemático para pasar de una población de individuos representados por cromosomas, que son cadenas de bits que representan soluciones a un problema, a una nueva población. El mecanismo usado para progresar a nuevas poblaciones emula la selección natural, y también se introducen operadores inspirados en el cruce entre los individuos seleccionados y la mutación aleatoria de los genes de los individuos resultantes del cruce.[1]

Los Algoritmos Evolutivos se basan en un modelo de evolución biológica natural que fue propuesto por primera vez por Charles Darwin. La teoría de la evolución de Darwin explica el cambio adaptativo de las especies por el principio de la selección natural, que favorece la supervivencia y evolución de aquellas especies que están mejor adaptadas a las condiciones de su entorno [3,8]. Otro factor importante de la selección es la aparición de variaciones pequeñas, aparentemente aleatorias y sin dirección en los fenotipos (características físicas). Estas mutaciones sobreviven a la selección si demuestran su valor en el entorno en curso. La fuerza básica conductora de la selección viene dada por el fenómeno natural de producción de descendencia. Bajo condiciones ambientales ventajosas, el tamaño de la población crece exponencialmente: proceso limitado por la existencia de recursos. Cuando los recursos dejan de ser suficientes, aquellos organismos que explotan los recursos más eficientemente tienen ventaja

selectiva. En la actualidad esta descripción suele aceptarse como una explicación macroscópica correcta de la evolución.

Dentro de los AEs se engloban normalmente los algoritmos genéticos (AGs), las estrategias evolutivas (EEs) y la programación genética (PG). Todos ellos tratan de hacer una abstracción del problema para hacerlo evolucionar y buscar así una mejor solución en los que necesitan adaptación, búsqueda y optimización.[1]

Figura 2.1. Clasificación de Programas Evolutivos

Los Algoritmos Genéticos ocupan un lugar central dentro de los AEs. Son los más estudiados por numerosas razones:[10]

- Completos: Reúnen las ideas fundamentales de la Computación Evolutiva.
- Flexibles: pueden adoptar con facilidad nuevas técnicas y combinarse con otros métodos.
- Mayor base teórica.
- Generales: no requieren conocimiento específico sobre la aplicación y pueden incorporar conocimiento específico con facilidad.
- Implementación sencilla y con recursos limitados.

Los AGs son métodos de búsqueda ciega, no disponen de ningún conocimiento específico del problema, de manera que la búsqueda se basa exclusivamente en los valores de la función objetivo; trabajan de forma codificada, es decir, no trabajan directamente sobre el dominio del problema, sino sobre representaciones de sus elementos; también procesan simultáneamente un conjunto de candidatos. Esta búsqueda se refiere tanto a las fases de selección como a las de transformación.

Estas características permiten hacer eficiente la búsqueda sin perder la generalidad y viceversa. Al trabajar con una población en lugar de con un único candidato se reduce la probabilidad de estancamiento en óptimos locales. Los AGs son generales porque explotan la información disponible en cualquier problema de búsqueda: la función objetivo.

Otra característica esencial de los AGs es su capacidad de intercambio estructurado de información en paralelo (paralelismo implícito): los AGs procesan externamente cadenas de códigos, sin embargo, lo que se está procesando internamente son similitudes entre cadenas. Al procesar cada una de las cadenas de la población se están procesando a la vez todos los patrones de similitud (esquema según Holland) que contienen (muchos más). Esta propiedad hace a los AGs mucho más eficaces que otros métodos de búsqueda ciega.

Al ejecutar un AG, una población de individuos, que representan a un conjunto de candidatos a soluciones de un problema, es sometida a una serie de transformaciones con las que se actualiza la búsqueda y después a un proceso de selección que favorece a los mejores individuos. Cada ciclo de selección+búsqueda constituye una generación. Se espera que después de una serie de generaciones, el mejor individuo represente a un candidato lo suficientemente próximo a la solución buscada.

Cómo funcionan los AGs

Los algoritmos evolutivos son procedimientos heurísticos de búsqueda y optimización que tienen inspiración en el mundo biológico. Se caracterizan por imitar los comportamientos adaptativos de la naturaleza y se basan en la supervivencia del mejor individuo, siendo un individuo una solución potencial del problema que se implementa como una estructura de datos.

Trabajan sobre poblaciones de soluciones que evolucionan de generación en generación mediante operadores genéticos adaptados al problema. Los parámetros que los controlan son variables en función de como se represente a los genes, del tipo de estructura de datos que implementa una solución, del tipo de operadores y de si éstos parámetros son

variables o constantes. Según estos factores nos encontramos con un tipo de estrategia u otra.

Los algoritmos genéticos simples se basan en un principio básico de la evolución: los mejores individuos tienen una mayor probabilidad de reproducirse y sobrevivir que otros individuos menos adaptados al entorno. Para implementar este principio, los algoritmos genéticos mantienen una población que evoluciona a través del tiempo y que al final convergen a una única solución.

Los algoritmos evolutivos basan parte de sus buenos resultados en el balance entre una eficiente exploración y una eficiente explotación cuando se resuelve un problema difícil. La exploración se refiere a la capacidad de mostrar diferentes partes del espacio de búsqueda en la población del algoritmo, mientras explotación se refiere a la capacidad de tunning y combinación de las soluciones subóptimas.
La exploración es útil para evitar estancarse en óptimos locales mientras que la explotación se usa para obtener el óptimo global una vez que se ha aproximado a él lo suficiente.
En las etapas iniciales de la búsqueda, un algoritmo genético debe mostrar una gran diversidad, mientras que al final la diversidad debe disminuir para conseguir una solución.
Para tratar de mejorar la velocidad de convergencia del algoritmo genético se puede utilizar la llamada Presión Selectiva, que es aquella que se ejerce cuando en el proceso de selección se utiliza un método basado en el fitness como puede serlo el de la ruleta o el torneo. Si la presión ejercida es excesiva puede derivar en Convergencia Prematura, el algoritmo converge a un óptimo local no pudiendo salir de él porque todos los individuos de la población están muy próximos a éste.

Para implementar un algoritmo genético es necesario definir:

- Una función de coste que evalúe a los individuos.
- Una codificación que permita representar las soluciones.
- El operador de selección.
- El operador de cruce.
- El operador de mutación.
- El tamaño de la población.
- Los valores de las probabilidades con que se aplica cada uno de los operadores.

Para comenzar se genera una población inicial a partir de la que se trabaja. Esta población se suele inicializar de una manera aleatoria, aunque existen implementaciones en las que se obtiene mediante otros métodos como una búsqueda local o cualquier otro tipo de algoritmo si se pretende que la búsqueda se inicie en una determinada dirección del espacio de soluciones. Cuando se han obtenido estos individuos iniciales, se repite el proceso que se explica a continuación tantas veces como indique la condición de parada, que puede ser un numero máximo de generaciones, la convergencia de la población, etc. Dicho proceso comienza evaluando la población y seleccionando los individuos que intervendrán en la formación de la siguiente generación. Seguidamente se aplican los operadores de cruce y mutación y se obtiene la nueva población a partir de la que se repiten los mismos pasos para ir avanzando en el proceso de búsqueda.

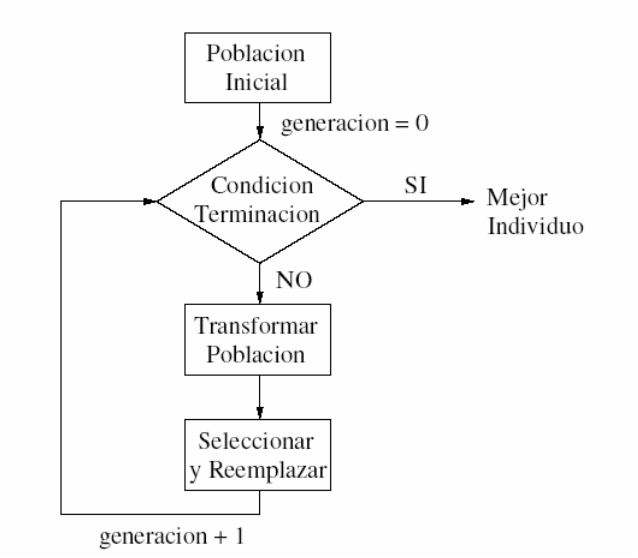

Figura 2.2. Flujo de ejecución de AG

A continuación se muestra en pseudo código la estructura general de un AG:

```
Generar una población inicial y computar la función de evaluación de cada
individuo

WHILE NOT Terminado DO

       FOR Tamaño población/2 DO
              Seleccionar dos individuos de la anterior generación, para el cruce
              Cruzar con cierta probabilidad los dos individuos
              Mutar los dos descendientes obtenidos con cierta probabilidad
              Computar la función de evaluación de los dos descendientes mutados
              Insertar los dos descendientes mutados en la nueva generación
       END
       IF la población ha convergido
              Terminado = TRUE
END WHILE
```

Representación genética

Cada individuo está representado por un cromosoma, que es una cadena de genes donde cada gen tiene un valor concreto de entre los posibles valores de dicho gen, llamados alelos. Este cromosoma codifica las variables del problema que se quiere resolver. Normalmente se utiliza un cromosoma simple compuesto por una cadena de bits de longitud fija, pero existen algoritmos genéticos que codifican el problema utilizando varios cromosomas para representar una solución y otros que utilizan cromosomas de longitud variable.

La elección de la codificación y el diseño de la función de coste son los dos puntos fundamentales en la implementación de un algoritmo genético. Las características que debe cumplir una buena codificación son las siguientes:

- Debe representar todo el espacio de soluciones.
- Debe asegurar que la aplicación de los operadores de cruce y mutación no se generan individuos incorrectos o irreales, es decir, que no representan una verdadera solución al problema.
- Deben cubrir todo el espacio de búsqueda de una manera continua, ya que si existen discontinuidades el proceso de búsqueda puede ser aleatorio.

La representación más sencilla suele utilizar un código binario, es decir, 0's y 1's. Al conjunto de caracteres que se utilizan para representar la población se le denomina alfabeto, y se representa por Ω. En el caso de la representación binaria sería: Ω={0,1}. Cada uno de estos valores se denomina alelo. Por otro lado, en determinados problemas no es suficiente con utilizar una codificación binaria y es necesario hacer uso de otra codificación, ya sea con cifras o con caracteres alfanuméricos.

Función de coste

La elección de una buena función de coste debe evaluar los individuos para indicar cuál es la calidad de la solución que representan, y poder realizar así el proceso de selección. Cada individuo tiene asignado un valor de esta función de coste, que mide la calidad de la solución y que es la herramienta que permite simular el concepto de individuos mejor adaptados. Aquellos individuos que tengan un mejor valor para la función de coste tendrán más posibilidades de ser seleccionados para construir la siguiente población y por lo tanto, pasar sus propiedades a los individuos de la siguiente generación.

Normalmente los AGs tratan de maximizar una función, aunque también puede haber problemas de minimización. En este caso basta con utilizar la inversa de la función dada.

Los algoritmos genéticos pueden tener una función objetivo además de la función de coste. Aunque la evaluación se realiza de acuerdo a una estimación de coste, se trata de alcanzar un objetivo que se evalúa mediante otra función distinta. Esta función objetivo no tiene porqué ser una función numérica, sino simplemente un indicador de si se cumple o no cierto criterio.

Operadores de Selección

El operador de selección identifica a los mejores individuos de la población actual y los utiliza para generar la siguiente población. La selección debe asegurar que los mejores individuos (mejor valor de fitness) tienen mayor probabilidad de ser seleccionados como padres en el posterior cruce. Esta probabilidad posibilita que aquellas soluciones que no pertenecen a las mejores puedan aportar información a la nueva generación.

La probabilidad de selección debe ser tal que consiga un equilibrio entre elitismo y exploración. Si la probabilidad de selección del mejor es muy alta se corre el peligro de que individuos de la población inicial con fitness superior a la media, que representan óptimos locales pero no globales, se reproduzcan en exceso provocando una pérdida de

diversidad y una convergencia prematura. En caso contrario, una proporción de selección demasiado baja puede provocar una búsqueda aleatoria.

Selección por el método de la ruleta:

> La idea es dar a cada individuo una probabilidad de ser seleccionado acorde a su función de coste, y proporcional a su calidad dentro de la población evaluada. De este modo, cuanto mejor es su valor de fitness, mayor es la probabilidad de que sea seleccionado.
>
> A continuación se calcula la probabilidad de selección acumulada para cada individuo, sumando para ello las probabilidades de selección de los individuos.
>
> Para seleccionar los individuos, se generan tantos números aleatorios como individuos se necesiten. Cada número aleatorio se compara con las probabilidades acumuladas y se escoge el individuo con una probabilidad asociada inmediatamente menor al número aleatorio generado. Al hacer los diagramas circulares correspondientes a las probabilidades acumuladas, se observa cómo cada individuo tiene una fracción proporcional a su probabilidad de selección.

Selección basada en elitismo:

> Es un variante del modelo anterior, y consiste en guardar siempre el mejor individuo de la población para la siguiente población, normalmente sustituyéndolo por el peor. Hay estudios que indican que la selección por elitismo asegura la convergencia a un óptimo global.

Selección basada en ranking:

> Es un método muy empleado cuando la población evoluciona muy rápidamente y queda atrapada en un óptimo local. En este modelo los individuos se ordenan según su valor de fitness, y su probabilidad de selección depende del ranking.

Esta selección consigue disminuir la presión de selección y, por tanto, ralentizar la convergencia de la población.

Selección por torneo:

Para cada iteración simple se toma cierto número de individuos, llamado también tamaño de torneo, y se selecciona para la siguiente generación uno de los individuos del conjunto. Esta operación se repite tantas veces como individuos haya en una población, puesto que en cada selección producimos un solo individuo de la nueva población.

Es evidente que cuanto mayor sea el número de individuos que se hacen competir cada vez, mayor es la presión de la selección para este método. Esto se debe a que el individuo más apto compite contra individuos con peor aptitud de modo que, conforme crece el tamaño de torneo, la probabilidad de tomar una decisión incorrecta también crece (proporcionalmente).

Selección sigma (Forrest):

Se trata de una técnica que intenta adaptar la selección a medida que evoluciona el AG. En ella, el valor esperado de un individuo depende de su valor de fitness, del fitness medio de la población y de la desviación estándar de la población.

Selección Boltzman:

La selección de Boltzman funciona de modo similar al enfriamiento simulado, variando la temperatura que controla la presión de selección. La temperatura inicial debe ser elevada para que la presión de selección sea baja y, por tanto, se prime la exploración. La temperatura irá bajando gradualmente de modo que se incremente la presión de selección, primando entonces la explotación.

Selección Steady-State:

En este tipo de selección, solamente unos pocos individuos son reemplazados en cada generación, normalmente un pequeño número de individuos con los peores fitness.

Operadores de Cruce

El operador de cruce implica elegir aleatoriamente a dos individuos de la población para que intercambien segmentos de su código, produciendo una "descendencia" artificial cuyos individuos son combinaciones de sus padres. Este proceso pretende simular el proceso análogo de la recombinación que se da en los cromosomas durante la reproducción sexual. Las formas comunes de cruce incluyen al cruce por un punto, en el que se establece un punto de intercambio en un lugar aleatorio del genoma de los dos individuos, y uno de los individuos contribuye todo su código anterior a ese punto y el otro individuo contribuye todo su código a partir de ese punto para producir una descendencia, y al cruce uniforme, en el que el valor de una posición dada en el genoma de la descendencia corresponde al valor en esa posición del genoma de uno de los padres o al valor en esa posición del genoma del otro padre, elegido con un 50% de probabilidad. Por otro lado se debe asegurar que la aplicación del operador de cruce no genere individuos que no sean reales.

Figura 2.3. Cruce por un punto

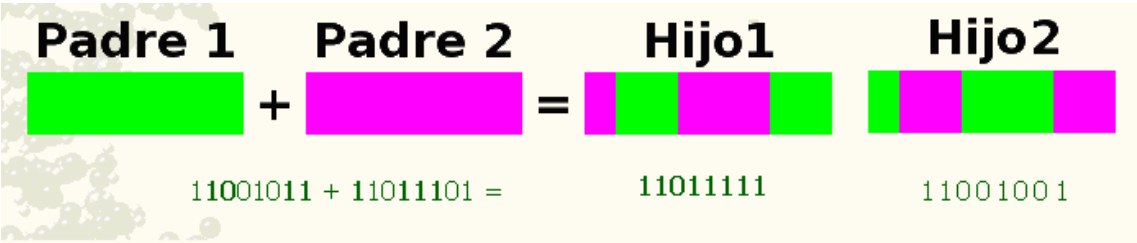

Figura 2.4. Cruce uniforme

Operadores de Mutación

Al igual que en los seres vivos, este operador cambia un gen por otro. El operador de mutación es el encargado de realizar pequeños cambios en el código con una probabilidad muy pequeña. Se utiliza para reestablecer la diversidad que se haya podido perder con la aplicación sucesiva de los operadores de selección y cruce. Normalmente se considera un operador secundario, pero esto puede ser un error ya que en ocasiones es muy útil hacer pequeñas modificaciones en la implementación para obtener mejoras importantes en el rendimiento del AG. Nosotros estudiaremos su efecto para comprobar en qué casos puede ser recomendable su uso.

El método más habitual de realizar la mutación es seleccionar un gen del individuo y cambiar su alelo por otro de los del alfabeto. La probabilidad con la que estos cambios se producen suele ser baja (normalmente 1/tamaño del individuo), para evitar que el AG realice una búsqueda aleatoria. Sin embargo en ocasiones puede ser necesario aumentar esta probabilidad para recuperar la diversidad de la población.

Figura 2.5. Operador de mutación

Tamaño de la población

Un factor muy importante para la convergencia de los AGs es el tamaño de la población. El tiempo necesario para que un AG converja a una solución única depende del tamaño de la población. Aunque los AGs son eficientes, sin embargo no garantizan la obtención de una solución óptima. Su efectividad viene claramente determinada por el tamaño de la población. Es evidente que cuanto mayor sea el número de individuos se explorarán más zonas del espacio de soluciones y será más probable que el algoritmo consiga encontrar el óptimo global. Pero también es bastante obvio que esto acarreará un costo computacional mayor, necesitando mayor tiempo de cálculo y, a veces, de convergencia. Por eso se debe buscar un compromiso entre el número de individuos utilizados y la calidad que se desea alcanzar.

2.2.- Algoritmos genéticos Paralelos

Algoritmos genéticos Paralelos

La computación paralela se ha convertido en una parte fundamental en todas las áreas de cálculo científico, ya que permite la mejora del rendimiento simplemente con la utilización de un mayor número de procesadores, memorias y la inclusión de elementos de comunicación que permitan a los procesadores trabajar conjuntamente para resolver un determinado problema. Al compartir la carga de trabajo entre N procesadores se puede esperar que el sistema trabaje N veces más rápido que con un solo procesador, lo que permite tratar problemas más grandes y complicados. Sin embargo las cosas no son tan sencillas, ya que existen varios factores de sobrecarga que hacen disminuir el rendimiento previsible. En ocasiones existen problemas cuya estructura no es lo suficientemente regular como para obtener rendimientos similares a los esperados. Otras veces los algoritmos y las técnicas utilizadas no son fáciles de paralelizar. Sin embargo, hay otros, como es el caso de los algoritmos genéticos, que tienen una estructura que se adapta perfectamente a la paralelización.[1,5]

Un programa es paralelo si en cualquier momento de su ejecución puede ejecutar más de un proceso. Para crear programas paralelos eficientes hay que poder crear, destruir y especificar procesos así como la iteración entre ellos. Existen tres formas básicas de paralelizar un programa:

- *Paralelización de grano fino*: la paralelización de un programa se realiza a nivel de instrucción.
- *Paralelización de grano medio*: los programas se paralelizan a nivel de bucle, normalmente de manera automática en el compilador.
- *Paralelización de grano grueso*: se basan en la descomposición del dominio de datos entre los procesadores, realizando cada uno cálculos sobre sus datos locales.

La paralelización más popular es la paralelización de grano grueso, debido a su portabilidad ya que se adapta perfectamente tanto a multiprocesadores de memoria compartida como de memoria distribuida. A su vez, este tipo de paralelización se puede llevar a cabo siguiendo tres estilos distintos de programación:

- *Paralelismo en datos*: El compilador se encarga de la distribución de los datos guiado por un conjunto de directivas que introduce el programador. Estas directivas hacen que cuando se compila el programa las funciones se distribuyan entre los procesadores disponibles. Es fácil de programar pero sin embargo suelen tener una eficiencia inferior a la que se consigue con el paso de mensajes. Para este modelo de paralelismo destacan los lenguajes HPF (High Performance Fortran) y OpenMP.

- *Programación por paso de mensajes*: Es el método más utilizado para programar sistemas de memoria distribuida. La forma más básica consiste en que los procesos coordinan sus actividades mediante el envío y la recepción de mensajes. Las principales ventajas que presentan son la flexibilidad, la eficiencia, la portabilidad y la controlabilidad del programa. Por contra el tiempo de desarrollo puede ser más elevado que para un paralelismo en datos. Las librerías más utilizadas son el estándar MPI (Message Passing Interface) y PVM (Parallel Virtual Machine).

- *Programación por paso de datos*: En este caso la transferencia de datos entre los procesadores se realiza con primitivas unilaterales tipo put-get, lo que evita la necesidad de sincronización entre los procesadores emisor y receptor. Es un modelo de programación de muy bajo nivel pero muy eficiente.

Los principales métodos de paralelización de AGs consisten en la división de la población en varias subpoblaciones. Por ello, el tamaño y distribución de la población entre los distintos procesadores será uno de los factores fundamentales a la ahora de paralelizar un algoritmo evolutivo.

Clasificación de los AGs paralelos

Existen varias formas de paralelizar un algoritmo genético. La primera y más intuitiva es la global, que consiste básicamente en paralelizar la evaluación de los individuos manteniendo una sola población. Otra forma de paralelización global consiste en realizar una ejecución de distintos AGs secuenciales simultáneamente. El resto de aproximaciones dividen la población en subpoblaciones que evolucionan por separado e intercambian individuos cada cierto número de generaciones. Si las poblaciones son pocas y grandes, tenemos la paralelización de grano grueso. Si el número de poblaciones es grande y con pocos individuos en cada población, tenemos la paralelización de grano fino. Por último, existen algoritmos que mezclan propiedades de estos dos últimos y son denominados mixtos.[1]

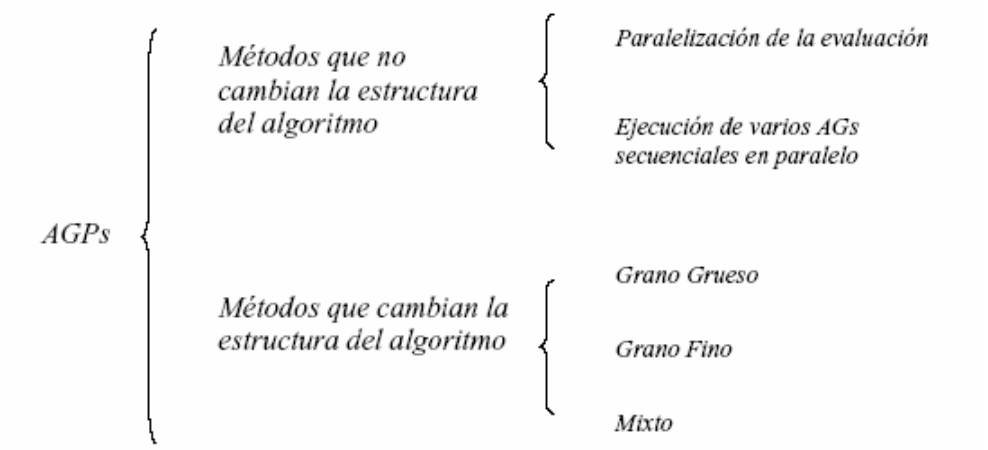

Figura 2.6. Clasificación de los AGP's

En los AGs las poblaciones van evolucionando por separado para detenerse en un momento determinado e intercambiar los mejores individuos entre ellas. Tenemos tres factores importantes que determinan la eficiencia de un algoritmo genético paralelo:[4]

- La topología de la comunicación entre los procesadores.
- La proporción de intercambio: el número de elementos que se intercambian en cada ocasión.
- La frecuencia de intercambio: periodicidad con que se intercambian los individuos.

Paralelización global

La paralelización global es la forma más sencilla de implementar un algoritmo genético paralelo. Consiste o en paralelizar la evaluación de los individuos o en realizar una ejecución simultánea de distintos AGs secuenciales. Es muy útil ya que permite obtener mejoras en el rendimiento con respecto al AG secuencial muy fácilmente, sin cambiar la estructura principal de éste. En la mayoría de las aplicaciones de los AGs la parte que consume un mayor tiempo de cálculo es la evaluación de la función de coste. En estos casos se puede ahorrar mucho tiempo de cálculo simplemente encargando la evaluación de una parte de la población a distintos procesadores y de una forma simultánea. Para problemas simples con tiempos de ejecución cortos, los AGP globales no son una buena opción para mejorar el rendimiento, pero para problemas con tiempos de ejecución elevados consiguen una mejora sustancial.

Normalmente la evaluación de un individuo cuesta exactamente igual para todos ellos y el tiempo de cálculo se puede disminuir aproximadamente en N veces, siendo N el número de procesadores. Evidentemente el costo de las comunicaciones reducirá el rendimiento, y cuanto más sencilla sea la información a enviar mayor será este. También habrá que tener en cuenta el tipo de arquitectura que se esté utilizando y su facilidad para transmitir un tipo de datos u otro. Una consideración importante a la hora de implementar un AGP global es que si se realizan muchas comunicaciones podemos perder cualquier reducción del rendimiento alcanzada mediante la paralelización.[1]

A continuación se muestra el pseudo código de un algoritmo genético en el que se ha paralelizado la evaluación de la función de coste:

```
generacion de la poblacion inicial
while ( no se cumpla la condicion de parada) do
   do in parallel
     evaluacion de los individuos
   end parallel do
   seleccion
   produccion de nuevos individuos
   mutacion
end while
```

Figura 2.7. Evaluación paralelizada de un AG

La aplicación de los operadores puede mantenerse global o realizarse en paralelo, aunque generalmente el costo de las comunicaciones necesario para paralelizar estas operaciones no compensa el tiempo de cómputo ahorrado.

Dentro de este tipo de paralelización tenemos dos tipos de implementaciones: la síncrona, que es cuando el programa se para y espera el resultado de la evaluación antes de proceder a crear la siguiente generación, teniendo así las mismas características que un algoritmo secuencial; y la asíncrona, en la que el procesador maestro no espera la llegada de todas las evaluaciones, siendo ésta la forma de implementación más usual debido a su facilidad de realización.

La otra forma de paralelización global consiste únicamente en enviar varios algoritmos a distintos procesadores y al final del proceso ver la mejor solución. El resultado es el mismo que si ejecutáramos varios AGs secuenciales y escogiéramos la mejor solución de todas las obtenidas. Aquí se muestra el pseudo código:

```
do in parallel
   generacion de la poblacion inicial
   while (no se cumpla la condicion de parada) do
     evaluacion de los individuos
     seleccion
     produccion de nuevos individuos
     mutacion
   end while
end parallel do
Escoger la mejor solucion
```

Figura 2.8. AG con varias poblaciones evolucionando en paralelo

Por último comentar que el modelo de paralelización global no hace ninguna distinción acerca de la arquitectura del computador sobre el que se está ejecutando. Se puede implementar tanto en un sistema de memoria compartida como de memoria distribuida. En un computador de memoria distribuida la población normalmente se almacena en un procesador (maestro) que se encarga de enviar los individuos al resto de procesadores (esclavos), de recoger la información y de aplicar los operadores. En cualquier caso, el costo de comunicaciones es similar para cualquiera de los dos sistemas.

AGPs de Grano grueso

Las características fundamentales de un AGP de grano grueso son la utilización de varias subpoblaciones relativamente grandes y la migración, entendiendo ésta como el intercambio de individuos entre distintas subpoblaciones [1]. Este tipo de paralelización es el más utilizado debido a:

- La facilidad de implementación: sólo hay que tomar un conjunto de AGs secuenciales, ponerlos en distintos procesadores y cada cierto número de generaciones intercambiar individuos. La mayoría del código de la versión secuencial queda exactamente igual después de paralelizar.

- La disponibilidad de los computadores paralelos en los centros de investigación es habitual, y si no son fácilmente simulables mediante software como MPI o PVM.

Los AGPs de grano grueso se suelen implementar sobre máquinas de memoria distribuida.

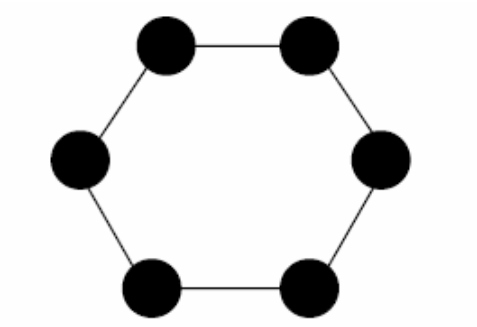

Figura 2.9: Esquema general de un AGP de grano grueso

Los AGPs de grano grueso simulan el aislamiento geográfico de las diferentes civilizaciones y el intercambio esporádico de características que se realiza con la emigración.

A continuación mostramos en pseudo código un esquema de un AGP de grano grueso:

```
 Inicializar P subpoblaciones con N individuos cada una
 Numero de generacion = 1
 while ( no se cumpla la condicion de fin) do
   for cada subpoblacion) do in parallel
     evaluar y seleccionar individuos por su funcion de coste
if (Numero de generacion mod frecuencia) = 0 then
  enviar K<N mejores individuos a poblacion vecina
  recibir K mejores individuos de poblacion vecina
  reemplazar K individuos de la poblacion
end if
     producir nuevos individuos
     aplicar operador de mutacion
   end parallel do
  Numero de generacion ++
end while
```

Figura 2.10: Pseudo código de un AGP de grano grueso

Topologías de comunicación:

La topología de comunicación es un factor fundamental en el rendimiento de un AGP, ya que determina la velocidad con que una buena solución se propaga de una subpoblación al resto. Si la topología tiene muchas conexiones entre las subpoblaciones las buenas soluciones se transmiten rápidamente de una población a otra. Por el contrario, si las poblaciones tienen poca comunicación entre ellas, las soluciones se extienden más lentamente, permitiendo la aparición de varias soluciones y una evolución más aislada de cada grupo. Otro factor en el que interviene la topología es el coste de comunicaciones. Es necesario buscar un compromiso entre la topología elegida y el costo de comunicaciones, para no perder las ventajas obtenidas sobre el rendimiento con la paralelización. La norma general es utilizar una topología estática que se mantiene constante a lo largo de toda la ejecución del algoritmo. Una segunda opción es la implementación de una topología dinámica. En ellas el intercambio entre subpoblaciones se realiza entre procesadores distintos cada vez en función de un criterio que suponga una mejora para el algoritmo.
Las topologías más utilizadas son:

- *Anillo*: las poblaciones están distribuidas en anillo y sólo hay intercambio entre vecinos.
- *Maestro – Esclavo*: todos los procesos esclavos intercambian sus mejores individuos con el maestro.
- *Todos con todos*: todos los procesadores intercambian información con cada uno del resto.

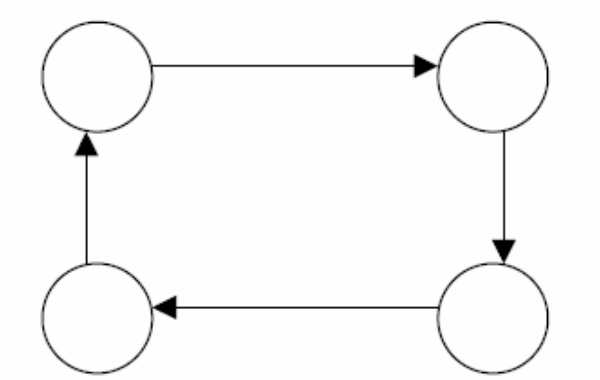

Figura 2.11. Modelo en anillo de un AGP

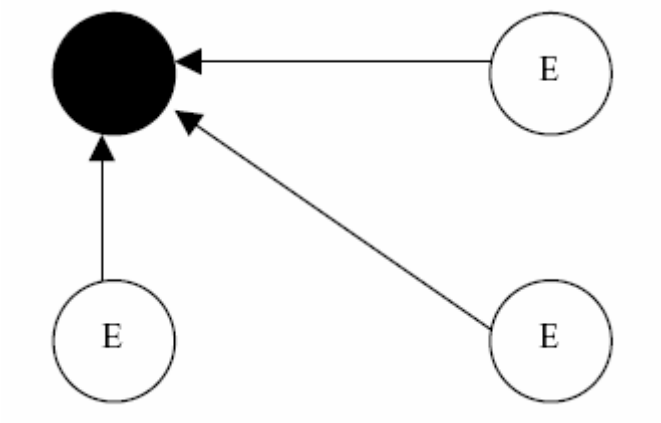

Figura 2.12. Modelo maestro esclavo de un AGP

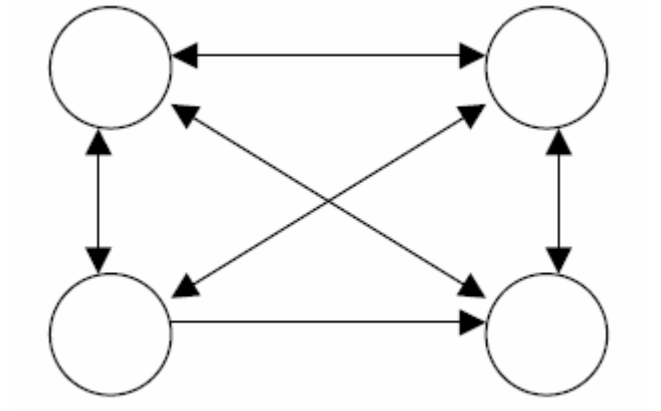

Figura 2.13. Modelo todos con todos de un AGP

Proporción y frecuencia de intercambio:

Tanto la frecuencia como la proporción de intercambio son muy importantes para la convergencia del algoritmo y para la calidad de las poblaciones. Aunque en principio se puede suponer que cuanto mayor sea el intercambio mejor se propagan las buenas soluciones, esto no es cierto totalmente, ya que puede suceder que el excesivo intercambio de individuos entre las poblaciones convierta el AGP en una búsqueda prácticamente aleatoria al no permitir que el AGP se desarrolle con normalidad.

Hay diferentes manera de realizar la implementación: en algunos casos el intercambio se produce únicamente cuando las poblaciones han convergido totalmente; en otros, el intercambio se realiza después de un determinado número de generaciones o con una

periodicidad fijada de antemano y que se mantiene constante a lo largo de toda la ejecución del programa.
Debido a la importancia ya citada de estos parámetros, es conveniente realizar un estudio de diferentes políticas de migración teniendo en cuenta la topología de las poblaciones.[4,7]

Estudiaremos el efecto de tener distintas frecuencias de intercambio para buscar las mejores configuraciones.

AGPs de Grano fino

Los AGPs de grano fino se conocen también como AGP grid o en parrilla debido a la disposición de las poblaciones sobre los procesadores. Los individuos se disponen en una parrilla de dos dimensiones, con un individuo en cada una de las posiciones de la rejilla. La evaluación se realiza simultáneamente para todos los individuos y la selección, reproducción y cruce se realizan de forma local con un reducido número de vecinos. Con el tiempo se van formando grupos de individuos que son homogéneos genéticamente como resultado de la lenta difusión de individuos. A este fenómeno se le llama aislamiento por distancia y es debido a que la probabilidad de interacción entre dos individuos disminuye con la distancia.
Este tipo de implementación simula las relaciones personales entre individuos de una misma localidad. Es decir, normalmente dos individuos que vivan cerca, tienen más probabilidad de relacionarse que dos que vivan más separados. La forma de seleccionar al individuo de la vecindad con el que se interactúa se puede hacer de diversas formas, siendo la forma más utilizada por torneo.

A continuación mostramos el pseudo código de un AGP de grano fino:

```
for cada punto de la parrilla
  do in parallel
generar un individuo aleatoriamente
end parallel do
while (no se cumpla la condicion de fin) do
  for cada punto de la parrilla k do in parallel
evaluate individual in k
seleccionar un individuo vecino q
producir descendientes de k y q
asignar uno a k
aplicar operador mutacion sobre k con probabilidad Pm
  end parallel do
end while
```

Figura 2.14. Pseudo código de un AGP de grano fino

Los AGPs de grano fino se adaptan mejor a las máquina de tipo SIMD (Single Instrucción Múltiple Data), ya que las operaciones necesarias para las comunicaciones locales son muy eficaces implementadas sobre esta arquitectura.

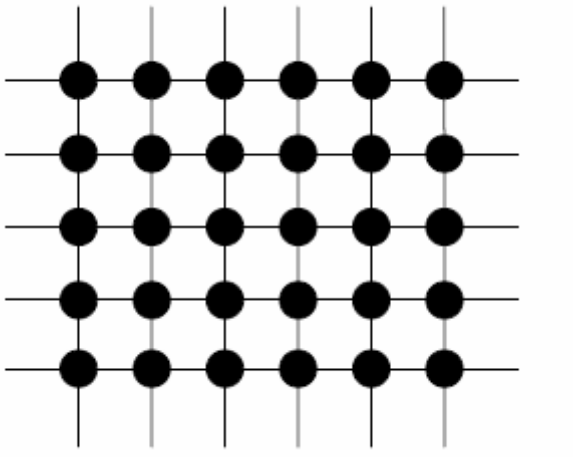

Figura 2.15. Esquema de un AGP de grano fino

AGPs Híbridos

Un AGP híbrido es una combinación de un AGP de grano grueso con un AGP de grano fino. Algunos de estos algoritmos añaden un nuevo grado de complejidad al entorno de los AGP, pero otros utilizan la misma complejidad que uno de sus componentes. Los AGPs híbridos son normalmente paralelos de grano grueso en el nivel superior. En el nivel inferior algunos tienen un AGP de grano fino y otros tienen de nuevo un AGP de grano grueso. Vamos a ver algunos esquemas de estos tipos de AGPs:

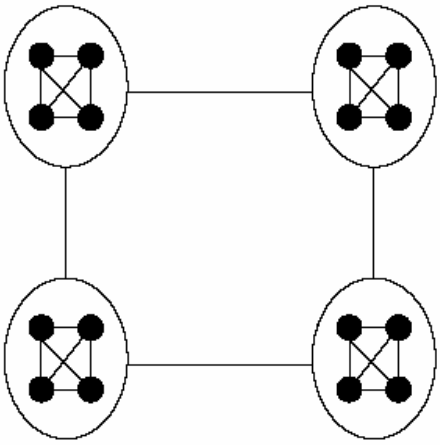

Figura 2.16. AGP híbrido combinando un AGP de grano grueso tanto en el primer como en el segundo nivel.

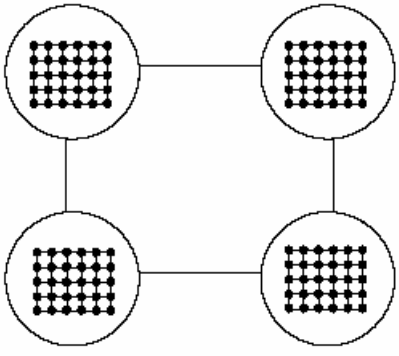

Figura 2.17. AGP híbrido que combina un AGP de grano grueso en el primer nivel con un AGP de grano fino en el segundo nivel.

3.- Procesamiento Paralelo

3.1- El cluster

Para la ejecución del algoritmo hemos utilizado una arquitectura de cuatro procesadores dobles lo que nos ha permitido hacer pruebas de hasta ocho poblaciones.

A continuación detallamos las especificaciones de la a máquina que es un modelo Proliant ML570 G2 de HP.

Es un quad de Xeon, y cada procesador es un Intel(R) Xeon(TM) MP CPU 2.80GHz stepping 05, con HT.

Figura 3.1. Intel Xeon

Cada una de los procesadores tiene la siguiente jerarquía de memoria cache:

L1: 8K

L2: 512K

L3: 2048K

Esta dotada de una memoria principal de 8575008k/8912892k disponibles (2269k kernel, 74844k reservado, 818k datos, 432k inicio, 7733212k highmem) del tipo DDR PC1600.

Posee un disco de 36Gb para el sistema y una controladora RAID con 5 discos para el espacio de los usuarios.

Figura 3.2. HP Proliant 570

Podemos ver la especificación de producto en la web de HP http://h18004.www1.hp.com/products/servers/proliantml570/specifications-g2.html

3.2- MPI

Introducción MPI.

Consiste en la replicación del paradigma de programación secuencial. El programador divide la aplicación en varios procesos que se ejecutan en diferentes procesadores sin compartir memoria y comunicándose por medio de mensajes.[12]

Objetivos:

- Total portabilidad de los códigos
- Mantener la inversión en un programa
- Desarrollar un programa en otra arquitectura antes de ejecutarlo en la arquitectura objetivo
- Permitir a los vendedores optimizar las rutinas del estándar (Hw especializado)
- Facilidad de uso
- Soportar entornos heterogéneos
- Versatilidad, con tan solo 6 rutinas se implementan la mayoría de las aplicaciones

Ventajas:

- Portable de modo eficiente a cualquier tipo de arquitectura: computador paralelo, red de estaciones y a una única estación de trabajo

Inconvenientes:

- Bastante más complicado de programar y depurar que memoria compartida
- La eficiencia depende del programador

Información que caracteriza un mensaje:

- En qué variable están los datos que se envían
- Cuantos datos se envían
- Qué proceso recibe el mensaje

- Cuál es el tipo de dato que se envía
- Qué proceso envía el mensaje
- Donde almacenar los datos que se reciben
- Cuantos datos espera recibir el proceso receptor

Envío síncrono y asíncrono (condición de finalización)

- Síncrona: El proceso que realiza el envío recibe información sobre la recepción del mensaje (fax). La comunicación se completa cuando el mensaje ha sido recibido

- Asíncrona: El proceso únicamente conoce cuando se envía el mensaje (postal). La comunicación se completa tan pronto como el mensaje ha sido enviado

Operaciones bloqueantes y no bloqueantes:

- Operación no bloqueante: Se inicia la operación y se vuelve al programa, por medio de otras funciones se puede comprobar la finalización de la operación (envío fax con memoria y recepción fax estándar)
- Operación bloqueante: Solo se vuelve al programa cuando la operación ha finalizado (envío y recepción fax estándar)

Comunicadores en MPI

Conjunto de procesos que pueden intercambiar mensajes. Cada comunicador contiene un grupo. Los procesos dentro de un grupo están numerados (rango). Todas las rutinas de comunicación MPI requieren de un comunicador. El comunicador por defecto es MPI_COMM_WORLD, que incluye a todos los proceso que se están ejecutando cuando el programa comienza.

Grupos y contextos en MPI: Colección ordenada de procesos, cada uno con un rango (número que le identifica)

Utilidad: Sirven para especificar que procesos están involucrados en una comunicación colectiva e introducir paralelismo en tareas; diferentes grupos pueden desarrollar diferentes tareas

MIMD: Cargando diferentes ejecutables en cada grupo.
SIMD: Mismo ejecutable donde cada grupo ejecuta diferentes opciones de un condicional

El número de procesos en ejecución es fijo, pero el número de grupos es dinámico, se crean, destruyen y un proceso puede estar involucrado en diferentes grupos. Un contexto es una etiqueta asociada a cada grupo por el sistema. Dos procesos que pertenecen al mismo grupo y que usan el mismo contexto pueden comunicarse. Un contexto permite la creación de diferentes flujos de mensajes. Los contextos particionan el conjunto de etiquetas de mensajes y los grupos particionan el espacio de procesos. Un proceso viene identificado por un grupo y un rango, un mensaje por un contexto y una etiqueta

Aplicaciones MPI

MPI es muy sencillo y la mayoría de los programas se pueden realizar utilizando solo 6 funciones básicas [11,13]:

- MPI_INIT: Inicializa las comunicaciones a través de MPI
- MPI_FINALIZE: Finaliza las comunicaciones a través de MPI.
- MPI_COMM_SIZE: Indica cuantos procesos intervienen en el programa.
- MPI_COMM_RANK Indica el índice del proceso actual, siendo este un número entre 0 y SIZE-1.

Las dos operaciones básicas para intercambiar información entre dos procesos son:

- MPI_SEND: Enviar información de un proceso a otro
- MPI_RECV: Un proceso recibe información de otro proceso.

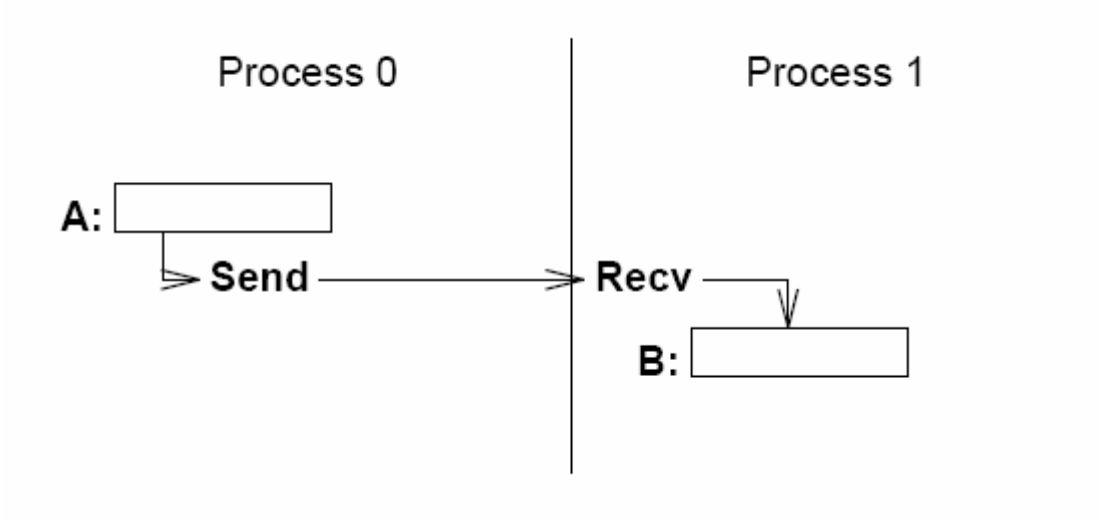

Figura 3.3. Mecanismo de paso de mensajes

Si el programa realiza operaciones cooperantes entre todos los procesos las dos operaciones básicas son:

- MPI_BCAST: Permite enviar información de un proceso al resto de procesos.

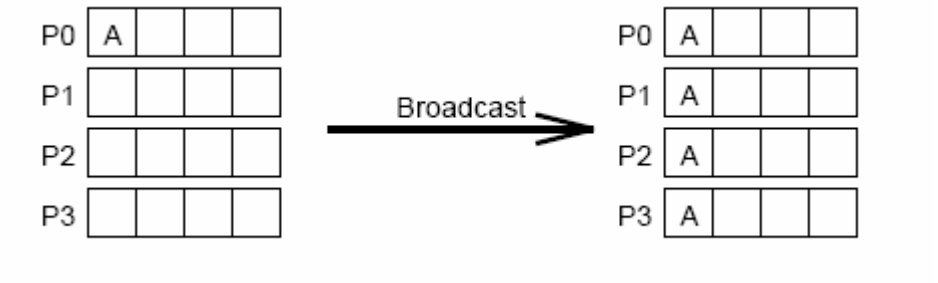

Figura 3.4. Operación broadcast

- MPI_REDUCE: Un proceso recibe información del resto y realiza una operación acumulativa con todos los datos recibidos.

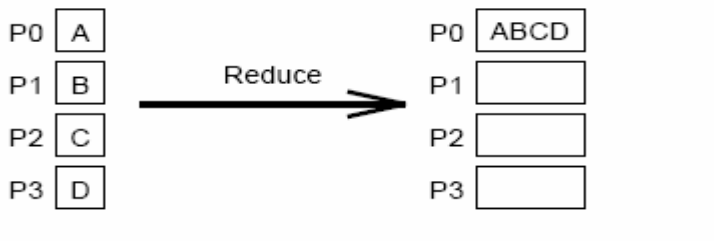

Figura 3.5. Operación Reduce

Modos de Comunicación

	Función	Condición de finalización
Send síncrono	`MPI_SSEND`	Se completa cuando el `receive` se completa
Send con buffer	`MPI_BSEND`	Se completa al copiar el mensaje en un buffer
(asíncrono)		Se debe crear y eliminar el buffer (`MPI_BUFFER_ATTACH`)
Send estándar	`MPI_SEND`	Síncrono o con buffer (según implementación)
Send preparado	`MPI_RSEND`	Siempre se completa (síncrono)
		Si no hay proceso esperando, el resultado no está definido
Receive	`MPI_RECV`	Se completa cuando se recoge el mensaje

Figura 3.6. Modos de comunicación MPI

- En el caso bloqueante la llamada vuelve cuando se satisface la condición de finalización
- En caso no bloqueante la llamada vuelve inmediatamente, devolviendo una variable, y luego se puede chequear la finalización por medio de funciones especificas usando esta variable.

4.- Funciones de prueba

Como ya hemos indicado vamos a analizar distintos tipos de funciones matemáticas con las que veremos el comportamiento del algoritmo genético. Estas funciones son las más habituales en la mayoría de los estudios sobre AGs por sus peculiaridades. A continuación describimos brevemente las características de cada una de ellas.[15]

4.1- OneMax

Descripción:

La función trata de maximizar el número de 1's del cromosoma. En nuestro caso la longitud del cromosoma es 200.

Formula:

$$f = \sum_{i=1}^{200} x_i$$

Valor óptimo:

x=200 f(x)=200

Gráfica:

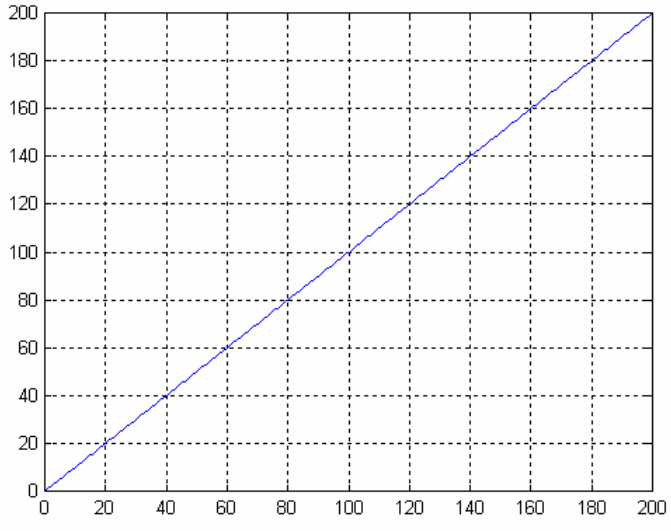

Figura 4.1. Función OneMax

4.2- Schwefel

Descripción:

Se trata de una función multimodal con varios óptimos y mínimos locales. Se ha aplicado sobre 10 variables por lo que evaluamos la función para cada tramo de cromosoma de longitud total partido de 10.

Formula:

$$f = -\sum_{i=1}^{10} x_i * \sin(\sqrt{|x_i|})$$

Valor óptimo:

$x_i = 420$ $f(x_i) = 418.7$

Gráfica:

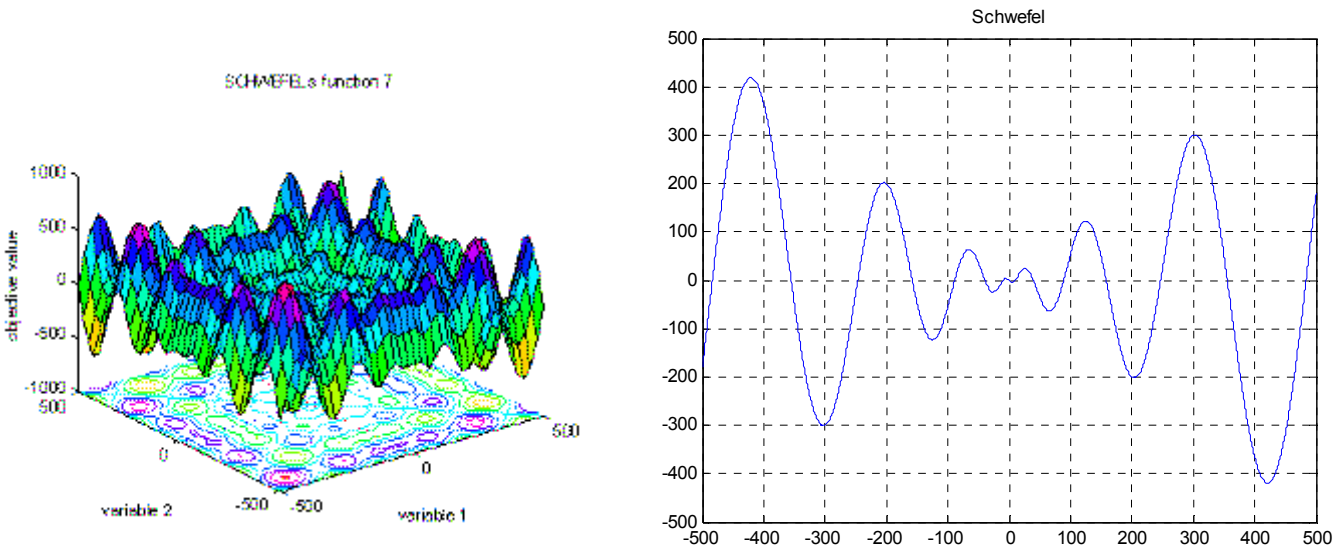

Figura 4.2. Función Schwefel

4.3- Rastrigin

Descripción:

Se trata de otra función multimodal. Al igual que en la función anterior evaluamos la función para cada tramo de cromosoma de longitud total partido de 10.

Formula:

$$f = \sum_{i=1}^{10} [x_i^{\,2} - 10 * \cos(2 * \Pi * x_i) + 10]$$

Valor óptimo:

x_i= -4.5 $f(x_i)$=40.25

Gráfica:

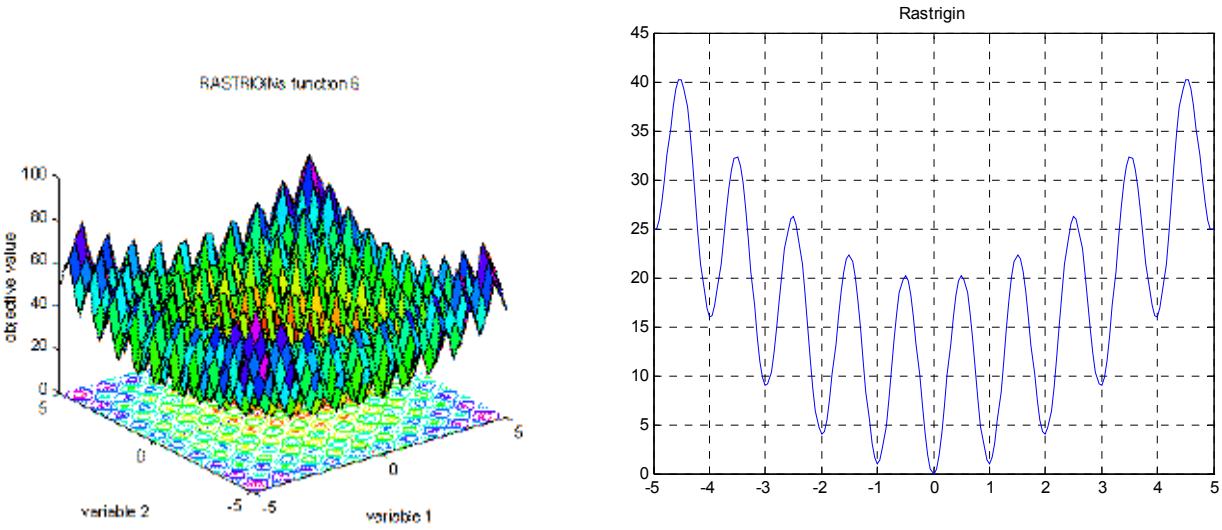

Figura 4.3. Función Rastrigin

4.4-F. Modal

Descripción:

De nuevo tratamos otra función multimodal. Al igual que en las funciones anteriores evaluamos la función para cada tramo de cromosoma de longitud total partido de 10.

Formula:

$$f = -\sum_{i=1}^{10} x_i * \sin(10 * \Pi * x_i) + 1$$

Valor óptimo:

x_i=-1.95 $f(x_i)$=2.95

Gráfica:

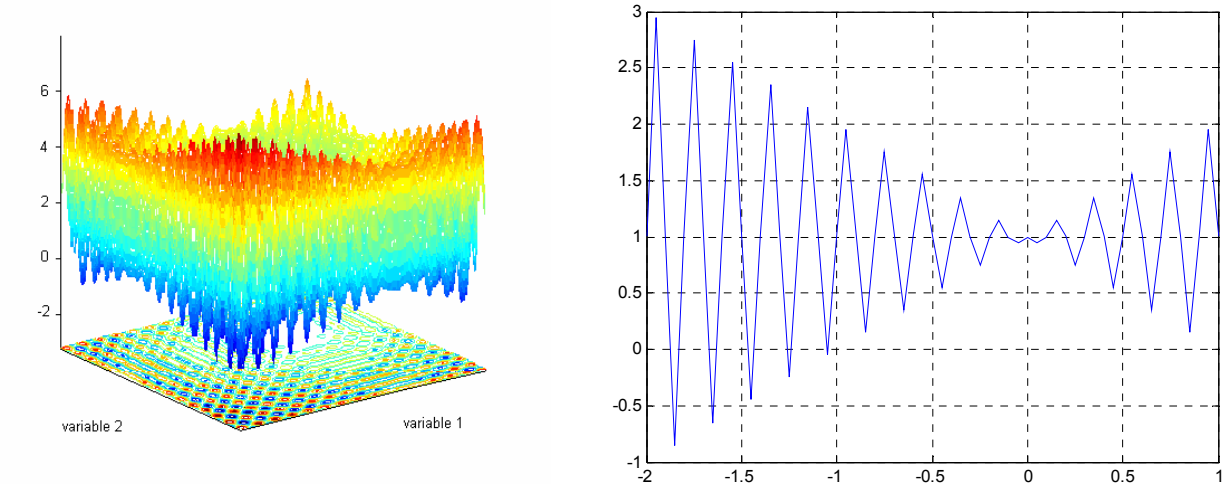

Figura 4.4. Función F. Modal

4.5-FTrap

Descripción:

Una función trampa es una función del número de 1s de una cadena de bits. Es una función defectiva porque desvía la búsqueda hacia una solución que está muy alejada de la correcta (óptimo global).

Formula:

$$u(x) = u(x_1, x_2, ...x_{200}) = \sum_{i=1}^{200} x_i$$

$$f(x) \begin{cases} \frac{a}{z}(z - u(x)) & \text{Si } u(x) \le z \\ \frac{b}{1-z}(u(x) - z) & \text{e.o.c.} \end{cases}$$

Siendo '*a*' el optimo local, '*b*' el optimo global y '*z*' el punto de inflexión entre los dos tramos de la función.

Valor óptimo:

x=200 f(x)=200

Gráfica:

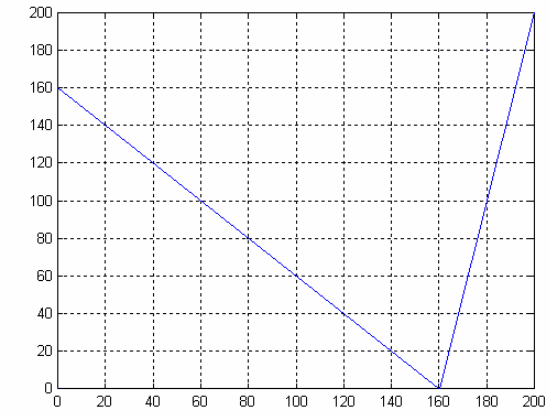

Figura 4.5. FTrap

5.- Implementación

5.1- AGP (PGAPack)

PGAPack es una librería de algoritmos genéticos paralelos escrita en ANSI C que intenta proveer la mayoría de capacidades deseadas en un paquete de algoritmos genéticos de manera integrada y portable. Estas son las principales características de PGAPack.[14]

- Capacidad para ser utilizada desde C o Fortran.
- Ejecutable en máquinas monoprocesador, multiprocesador o redes de estaciones de trabajo.
- Tipos de datos nativos: binarios, enteros, reales y caracteres.
- Diseño de estructuras de datos orientadas a objetos.
- Reemplazo parametrizado de poblaciones.
- Múltiples opciones para los operadores de selección, cruce y mutación.
- Fácil integración de heurísticas hill-climbing (ascenso de colinas).
- Soporte para nuevos tipos de datos.
- Todas las funcionalidades y los accesos a la librería se realizan a través de llamadas a funciones.

El modelo de programación paralela usado en PGAPack es el paso de mensajes, en particular el modelo SPMD (Single Program Multiple Data). Nosotros hemos construido la versión paralela de PGAPack bajo la implementación MPICH 1.2.7 de MPI.

Uso de PGAPack en el algoritmo:

En la implementación del AGP utilizaremos la técnica de selección por torneo, cruce uniforme fijando la probabilidad de cruce al 100%, la mutación estará parametrizada y en caso de usarse, la probabilidad de mutación será de 1/L, siendo L la longitud del cromosoma. Se utilizará elitismo para el reemplazo con lo que los dos mejores individuos se propagan directamente a las siguiente generación y el resto se recombina. El tamaño de población y la frecuencia de intercambio están también parametrizados.

Estructura del programa usado para las pruebas:

El programa que hemos realizado para todas las pruebas necesarias de nuestras funciones ha sido desarrollado bajo C. Todos los programas en C para PGAPack deben incluir la cabecera *pgapack.h*, aunque en nuestro caso incluimos alguna más para determinadas funciones que necesitaremos:

```
#include <pgapack.h>
#include <mpi.h>
#include <math.h>
```

Antes de nada tenemos las declaraciones de una serie de constantes:

```
#define MIN_schwefel  -500
#define MAX_schwefel  500
#define MIN_rastrigin -5
#define MAX_rastrigin 5
#define MIN_fmodal  -2
#define MAX_fmodal  1
#define PI 3.1416
#define A 160
#define B 200
#define Z 160
#define N_VAR 10
```

Ya dentro del programa principal, las primeras líneas que tenemos son las que inicializan MPI y la llamada a PGACreate, que es siempre la primera llamada dentro de un programa de PGAPack. PGACreate inicializa la variable de contexto (ctx), determina el tipo de los datos que vamos a tratar (binarios) y la longitud de cadena (len),

y también indica la dirección de optimización (en este caso nuestros problemas son de maximización):

```
MPI_Init(&argc,&argv);
ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, len, PGA_MAXIMIZE);
```

La variable de contexto es una estructura de datos que nos proporciona la capacidad de abstracción de los datos. Es un puntero a una estructura del lenguaje C, la cual es a su vez una colección de otras estructuras. Estas subestructuras contienen toda la información necesaria para ejecutar el algoritmo genético.

A continuación tenemos las llamadas a las funciones que establecerán la configuración de los parámetros de nuestra aplicación. En nuestro caso establecemos los siguientes parámetros: número máximo de generaciones, semilla aleatoria, tamaño de la población (número de individuos), tipo de fitness, número de individuos a reemplazar, política de reemplazamiento, tipo de cruce, probabilidad de cruce, indicación para ver si se debe ejecutar el cruce y la mutación o sólo una de las dos, probabilidad de mutación, frecuencia de impresión de resultados. A continuación mostramos el código de estas llamadas en el orden en que han sido enumeradas arriba:

```
PGASetMaxGAIterValue(ctx,maxiter);
PGASetRandomSeed(ctx, seed);
PGASetPopSize(ctx,pob);
PGASetFitnessType(ctx,PGA_FITNESS_RANKING);
PGASetNumReplaceValue(ctx,pob-2);
PGASetPopReplaceType(ctx,PGA_POPREPL_BEST);
PGASetCrossoverType(ctx,PGA_CROSSOVER_UNIFORM);
PGASetUniformCrossoverProb(ctx,1);
      if(mut==0){
            PGASetMutationOrCrossoverFlag(ctx,PGA_TRUE);
            PGASetMutationProb(ctx,0);
      }else{
            PGASetMutationAndCrossoverFlag(ctx,PGA_TRUE);
            PGASetMutationProb(ctx,0.05);
      }

PGASetPrintFrequencyValue(ctx,n_iteration_rep);
```

Después tenemos la llamada a PGASetUp que inicializa con valores por defecto todos los parámetros y punteros a funciones que no hayan sido declarados por el usuario y dos llamadas a funciones MPI que determinan en variables el número de procesadores y el identificador del procesador en cuestión:

```
PGASetUp(ctx);
MPI_Comm_size(MPI_COMM_WORLD,&n_pops);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

A continuación se llama a una función que a su vez llamará a la función que haya sido pasada por parámetro en el main, y que será una de las cinco de que consta nuestro estudio. También se hallará el fitness para cada uno de los individuos de la población:

```
evaluar(ctx,PGA_OLDPOP,feval);
PGAFitness(ctx,PGA_OLDPOP);
```

Ahora viene el bucle que se ejecutará mientras no se cumpla la condición de parada que hemos establecido anteriormente. Dentro del bucle se harán las llamadas para realizar la selección, el cruce y la mutación, además de llamar de nuevo a la evaluación de la población, a calcular el fitness de sus individuos, a actualizarla con los nuevos valores y, dado el caso, ejecutar el intercambio de individuos entre procesadores.

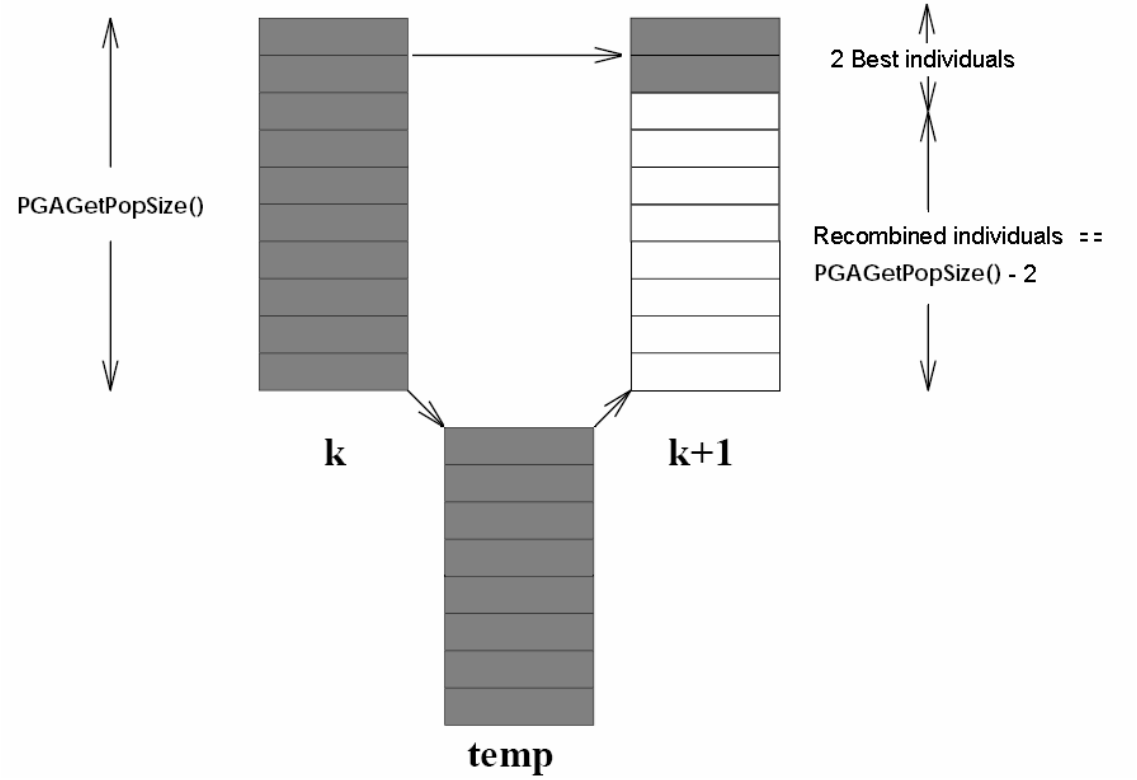

Figura 5.1. Elitismo y reemplazo

También llamamos a una función auxiliar programada por nosotros en la que mostraremos por pantalla información que nos será útil para realizar el estudio.

```
while(!PGADone(ctx,NULL)){
	PGASelect(ctx,PGA_OLDPOP);
	if(mut==0)
		PGARunMutationOrCrossover(ctx,PGA_OLDPOP,PGA_NEWPOP);
	else
		PGARunMutationAndCrossover(ctx,PGA_OLDPOP,PGA_NEWPOP);

	evaluar(ctx,PGA_NEWPOP,feval);
	PGAFitness(ctx,PGA_NEWPOP);		PGAUpdateGeneration(ctx,NULL);
	if(PGAGetGAIterValue(ctx)%n_iteration_rep==0)
		changeElementRing(ctx,nElemChange,feval);
	imprimir(ctx,myid,n_pops,feval);
}
```

Después vienen las llamadas que finalizan la ejecución del programa principal, destruyendo el contexto y cerrando la conexión MPI:

```
PGADestroy(ctx);
MPI_Finalize();
```

Dentro de nuestro programa están declaradas las cinco funciones que hemos usado en nuestro estudio. Cada una de ellas devuelve un valor que será el que determine el fitness de los individuos. Aquí vemos el código de una de ellas (Schwefel):

```
double evaluate2(PGAContext *ctx, int p, int pop) {
	double x;
	int i,ini;
	int len_var;
	double sol,aux;
	sol=0;
	ini=0;
	len_var=PGAGetStringLength(ctx)/N_VAR;
	for(i=0;i<N_VAR;i++){
		x=getX(ctx,p,pop,ini,len_var-1,MIN_schwefel,MAX_schwefel);
		aux=((double) -x*sin(sqrt(fabs(x))));
		sol=sol+aux;
		ini=ini+len_var;
	}
	return sol;
}
```

Dentro de esta función se llama a una función auxiliar getX que nos devolverá un valor en un intervalo dependiendo de la función que se trate.

Por último comentamos la función que realiza el intercambio de individuos entre procesadores con una frecuencia determinada de antemano a través de un parámetro como hemos visto en el programa principal. En esta función de nuevo se determina el número de procesadores y el identificador de cada uno de ellos. Luego simplemente (cada procesador cuando le toque) se ordenará la población en función al fitness de sus individuos, se obtendrá el índice de los mejores y peores individuos (en nuestro caso hemos establecido que sólo se intercambiará un elemento, de manera que sólo nos quedamos con el índice del mejor y el índice del peor) y se llamará al intercambio a través del comunicador MPI (MPI_COMM_WORLD). Cada procesador envía al inmediatamente posterior su mejor individuo y por otra parte reserva una plaza para recibir el que le envíe el inmediatamente anterior (sustituye su peor elemento por el mejor del procesador anterior). Se hace uso de una serie de cerrojos (barreras) para que los intercambios se realicen de manera correcta.

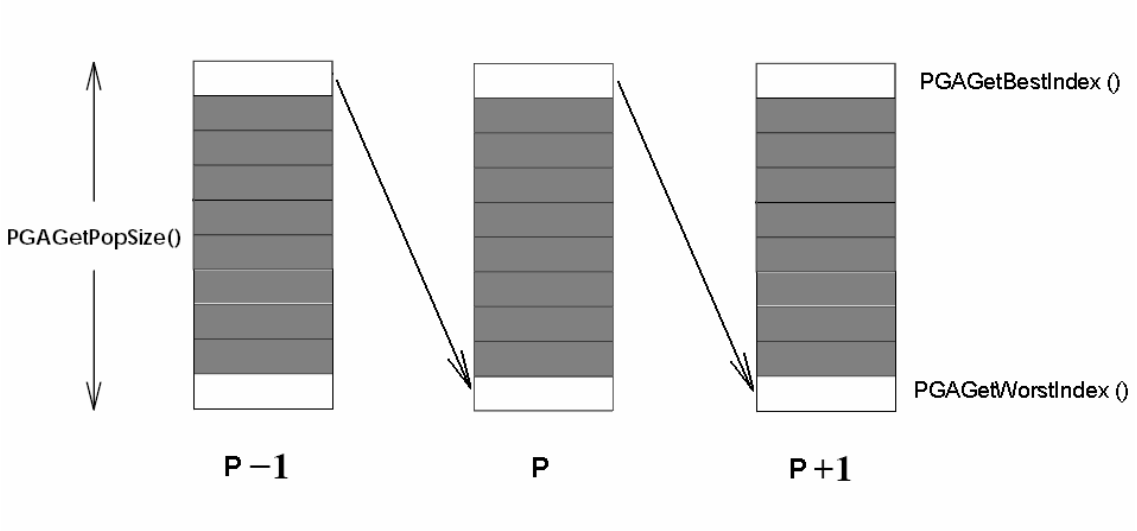

Figura 5.2. Intercambio en anillo

```
void changeElementRing(PGAContext *ctx,int nElemChange, int feval) {
      int myid, n_pops, tag, m,elemCambiado;
      MPI_Status  status;
      int i,j,k,ant;
      tag=2001;

      MPI_Comm_size(MPI_COMM_WORLD,&n_pops);
      MPI_Comm_rank(MPI_COMM_WORLD,&myid);

      //Intercambiar n peores por los n mejores
      PGASortPop(ctx,PGA_OLDPOP);
      for(i=0;i<nElemChange;i++){
            j=PGAGetSortedPopIndex(ctx,i);
            k=PGAGetSortedPopIndex(ctx,(PGAGetPopSize(ctx)-1)-i);
            if(myid==0)
                  ant=n_pops-1;
            else
                  ant=myid-1;

      PGASendIndividual(ctx,j,PGA_OLDPOP,(myid+1)%n_pops,tag,MPI_COMM_WORLD);
      PGAReceiveIndividual(ctx,k,PGA_OLDPOP,ant,tag,MPI_COMM_WORLD,&status);

      MPI_Barrier(MPI_COMM_WORLD);
      }
}
```

5.2- Shell script

El Shell o intérprete de órdenes de UNIX es un programa ejecutable (como cualquier otro programa de usuario) que se encuentra en cualquier entorno UNIX y cuyo cometido es leer órdenes para el sistema operativo, analizarlas y realizar las llamadas al sistema que sean necesarias para ejecutarlas.[16]
Este intérprete define al igual que ocurre con cualquier otro intérprete o traductor un lenguaje de programación que posee características como:

- Procedimientos.
- Palabras y caracteres reservados.
- Variables.
- Estructuras de control.
- Manejo de interrupciones.

Realmente en un sistema UNIX existen varios intérpretes de órdenes, pudiendo elegir cada usuario el que prefiera. El intérprete de órdenes más básico es el Bourne Shell o sh que esta disponible en cualquier sistema. Utilizaremos el Bourne-Again shell que es totalmente compatible con el sh.
Se denomina Shell Script a un fichero que contiene órdenes para ser ejecutadas por el shell.
Nuestro script únicamente consta de una serie de bucles en los que variamos los parámetros con los que se lanzará la aplicación island_model. Para ejecutar dicha aplicación se hace una llamada a mpirun para que sea lanzada de forma paralela en el número de procesadores especificado. Los parámetros que variamos son los siguientes:

- Mutación: valores de 1 y 0 para indicar si se usa o no este operador.
- Función: valores 1-5 para cada una de las 5 funciones con las que se realizado nuestro estudio.
- Número de procesadores: valores 1-8.
- Frecuencia de intercambio: valores 10, 25, 50.
- Semilla: valores 1-10 para que la semilla sea inicializada con diferentes valores.

Además hacemos la llamada con otros dos parámetros que están fijados de antemano, el tamaño de cada individuo que será 200, y el número de elementos a intercambiar cuando se de el caso que será 1.

El número de individuos por población lo determinamos de forma dinámica, dependiendo del número de generaciones y del número de procesadores de que dispongamos en cada ejecución. Partiendo de un esfuerzo de cómputo fijo (cota) calcularemos este valor para el número de individuos.

Comentar también que el resultado de cada ejecución se redirige a un archivo de texto que estará localizado en la ruta que se indica de forma dinámica.

A continuación se muestra el código del script con el que se han realizado las pruebas:

```
#!/bin/bash
for MUTAR in 0 1
do
for CONTGEN in  400
do
  COTA=CONTGEN*8*30
  FUN=1
  while test $FUN -lt 6
  do
      CONTPROC=1
      while test $CONTPROC -lt 9
      do

            for CONTFREC in 10 25 50
            do
                  AUX=$(($CONTGEN * $CONTPROC))
                  INDIVIDUOS=$(($COTA / $AUX | bc ))
                  CONTSEED=1

                  rmmut$MUTAR/gen$CONTGEN/fun$FUN/proc$CONTPROC/
                  frec$CONTFREC/*

                  while test $CONTSEED -lt 11
                  do
                        mpirun   -np   $CONTPROC     island_model   200
                        $CONTGEN  $INDIVIDUOS  $CONTFREC  $CONTSEED  1
                        $FUN                                        $MUTAR
                        >>mut$MUTAR/gen$CONTGEN/fun$FUN/proc$CONTPROC
                        /frec$CONTFREC/seed$CONTSEED.txt
                        CONTSEED=`expr $CONTSEED + 1`
                  done
            done
            CONTPROC=`expr $CONTPROC + 1`
      done
      FUN=`expr $FUN + 1`
  done
done
done
```

5.3- Análisis de pruebas

Para extraer los resultados de la ejecución del algoritmo hemos usado una aplicación java que lee los ficheros de texto generados por el script y los introduce en una serie de bases de datos que luego usaremos para mostrar resultados y gráficas. La aplicación, desarrollada con JDK 1.4, consta de dos paquetes, uno que contiene las clases que generan las tablas para los resultados con mutación y otro para el caso en el que no se haya usado este operador. A continuación se detallan las clases de los paquetes (análogas para los dos casos comentados):

- *Bloque.java:* clase auxiliar para almacenar la información de cada iteración y que leemos de los archivos de texto.

- *GeneradorEstadisticas.java:* clase que genera las estadísticas generales para todos los casos que se dan en la ejecución. Generará 3 tablas por cada función: una general con todos los resultados y 2 reducidas obteniendo medias (en el primer caso la media de las 10 semillas y en el segundo además también la de las tres frecuencias de intercambio distintas).

- *GeneradorEvolucionFitnessMejores.java:* esta clase generará una seria de tablas con la traza de la evolución del fitness de los mejores individuos para las distintas configuraciones de número de procesadores y frecuencia de intercambio para una determinada función. En el main de la clase por lo tanto hay que pasarle el número de la función que se desea evaluar. Los resultados obtenidos aparecen en la base de datos auxiliar TrazasEvoluciones.mdb.

- *GeneradorEvolucionFitnessMedias.java:* análoga a la clase anterior pero para el caso de los resultados promedio de fitness de los individuos de las poblaciones. Los resultados nos aparecen en la misma base de datos que en el caso anterior.

- *GeneradorEvolucionFitnessPeores.java:* de nuevo análoga a las dos clases anteriores. En este caso se obtendrá la traza de fitness para los peores individuos en cada generación. La base de datos donde se almacenan los resultados es la misma que anteriormente.

- *GeneradosPorcentajeConvergidos.java:* esta clase extraerá los resultados del tanto por ciento de individuos de una población que han alcanzado el óptimo, entendiendo como óptimo el valor máximo que se dará a lo largo de toda la ejecución en esa población. De nuevo habrá que pasar en el main la función que nos interesa estudiar. Los resultados se muestran también en este caso en la base de datos TrazasEvoluciones.mdb.

- *GeneradorEvolucionPoblacionMejores.java:* clase que obtendrá un análisis poblacional de los mejores individuos. Veremos las estadísticas, en tanto por ciento, de la contribución de los diferentes procesadores a la formación de los mejores individuos. Es decir, conoceremos qué tanto por ciento de "mejores individuos" han pasado por un solo procesador, qué tanto por ciento lo ha hecho por dos, etc. En este caso los resultados obtenidos se guardan en la base de datos auxiliar TrazasPoblaciones.mdb y de nuevo tendremos que introducir el número de función deseada.

- *GeneradorEvolucionPoblacionTodos.java:* clase similar a la anterior, en la que, en vez de limitarnos a estudiar sólo a los mejores individuos, estudiaremos las poblaciones completas, es decir, todos los individuos. Veremos de nuevo reflejados los porcentajes comentados más arriba en la misma base de datos e introduciendo también la función deseada.

- *Estadistica.java:* clase auxiliar para guardar la información necesaria de cada generación para luego hacer el estudio de las dos clases anteriores.

- *GeneradorXML:* clase que generará los archivos .xml con los mismos resultados que aparecen en las bases de datos EstadisticasSinMutacion y

EstadisticasConMutacion. Para la compilación de esta clase es necesario el archivo jdom.jar.

Comentar también que habrá que configurar los Orígenes de Datos ODBC. El driver usado es JdbcOdbcDriver, y dependiendo de las clases usaremos diferentes URL's de las bases de datos. Estas son las correspondencias entre los nombres de orígenes de datos usados en las clases .java y las bases de datos Access:

Nombre de Origen de Datos	**Base de Datos Access**
EstadisticasSinMutacion	EstadisticasSinMutacion.mdb
EstadisticasConMutacion	EstadisticasConMutacion.mdb
Trazas	TrazasEvoluciones.mdb
TrazasEvolucionPoblacionMejores	TrazasPoblaciones.mdb
TrazasEvolucionPoblacionesTodos	TrazasPoblaciones.mdb

Aclarar que las bases de datos TrazasEvoluciones.mdb y TrazasPoblaciones.mdb son auxiliares, ya que sólo pueden almacenar las trazas de todas las configuraciones para una función en concreto. Una vez generadas las trazas de una función habría que guardar la base de datos para que no se sobrescriba en la próxima ejecución.

6.- Resultados experimentales

6.1. Introducción a los estudios.

En los casos de estudio abarcaremos distintos análisis basados en diferentes aspectos del rendimiento de los algoritmos para los casos de mejores individuos, medias de todas las poblaciones y peores individuos. Así mismo nos adentraremos también en el estudio de la convergencia de las poblaciones y de la contribución de los procesadores a la formación de los distintos individuos a lo largo de las generaciones de las pruebas. Por último analizaremos los efectos del operador de regeneración en las funciones estudiadas.

La mayor parte del análisis se realiza a partir de las gráficas obtenidas mediante la representación de las pruebas lanzadas para las diferentes configuraciones y funciones de estudio.

La dinámica de estudio para cada caso consta de diferentes apartados donde veremos el efecto de fijar los diferentes parámetros variando uno de ellos. En general el orden seguido será, efecto de número de poblaciones, efecto de la frecuencia de intercambio y efecto de la mutación.

Por último, aclaramos que en las leyendas gráficas se representa realmente el periodo de intercambio. Nosotros hablaremos de frecuencia en su lugar, por ello habrá que tener en cuenta que la frecuencia es la inversa del periodo. También comentar que nos referiremos indistintamente a poblaciones y procesadores, ya que a cada procesador le corresponde una población.

6.2. Estudio fitness mejores.

En este caso vamos a estudiar la evolución del valor del fitness de los mejores individuos a lo largo de las generaciones. Para ello seleccionaremos en cada generación al mejor individuo de todos los procesadores, y luego obtendremos un resultado promedio para las 10 ejecuciones con distintas semillas.

6.2.1.- Efecto del número de procesadores.

En primer lugar vamos a estudiar el efecto que produce el disponer de un número mayor o menor de procesadores, tratado de encontrar el número óptimo de procesadores y evitar así una utilización innecesaria de recursos. Como ya sabemos, el número de individuos por población depende directamente del número de procesadores (poblaciones) de la prueba. Veremos como, en general, el modelo con varias poblaciones alcanza antes la solución que el modelo de una sola población.

- Función: OneMax
- Parámetro Fijado: Sin mutación, frecuencia
- Parámetro variable: Número de procesadores 1,2,4,8
- Gráfica:

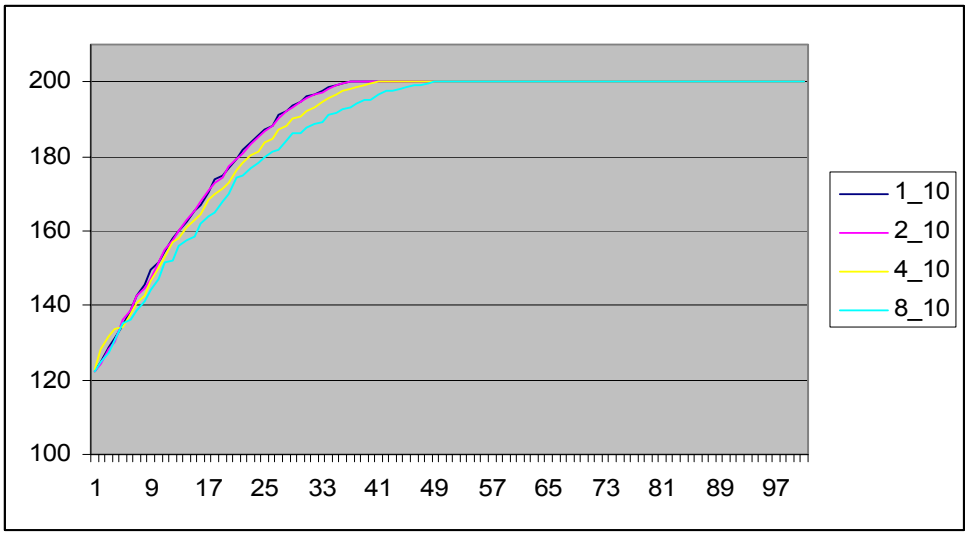

Figura 6.1. OneMax con frecuencia fija

- Observaciones: Vemos en la gráfica que todas las configuraciones convergen, siendo las de mayor número de poblaciones las que tardan alguna generaciones más. Esto confirma el beneficio que en muchos casos se obtiene debido a la diversidad provocada por tener una única población.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, Frecuencia de intercambio 50
- Parámetro variable: Número de procesadores 1,2,4,8
- Gráfica:

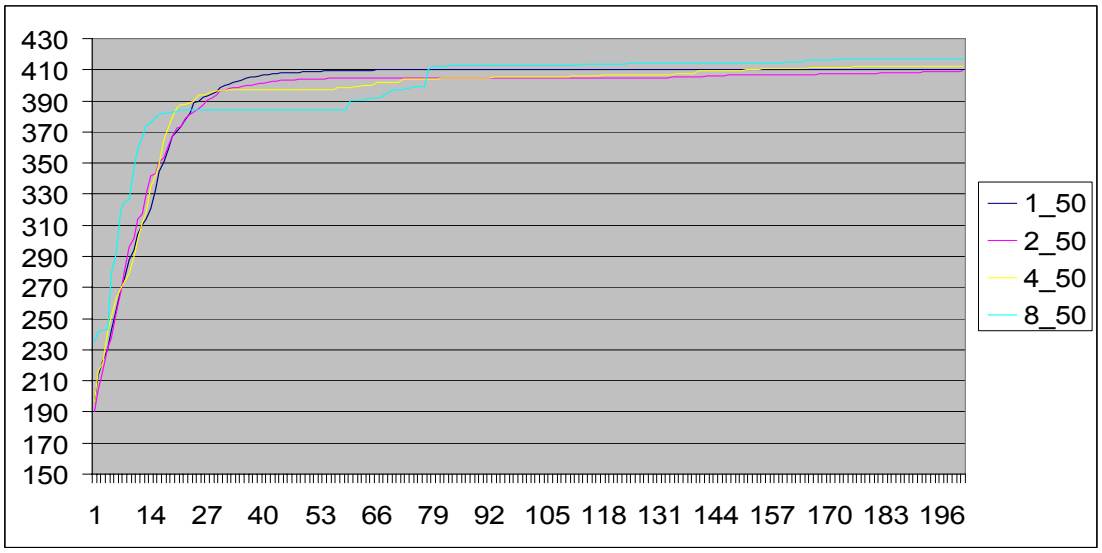

Figura 6.2. Schwefel con frecuencia fija

- Observaciones: Inicialmente, cuanto mayor es el número de procesadores, el valor de fitness converge antes a un valor menor. Conforme se van realizando los intercambios la situación se va invirtiendo según avanzan las generaciones, quedando las configuraciones con más poblaciones por encima. Este incremento es mas acusado conforme mas alto es el número de procesadores. En la generación 400, no representada en la gráfica, hemos comprobado que el valor de fitness de las configuraciones está establemente ordenada de mayor a menor número de poblaciones.

- Función: Rastrigin
- Parámetro Fijado: Sin mutación, Frecuencia de intercambio 25
- Parámetro variable: Número de procesadores 1,2,4,8
- Gráfica:

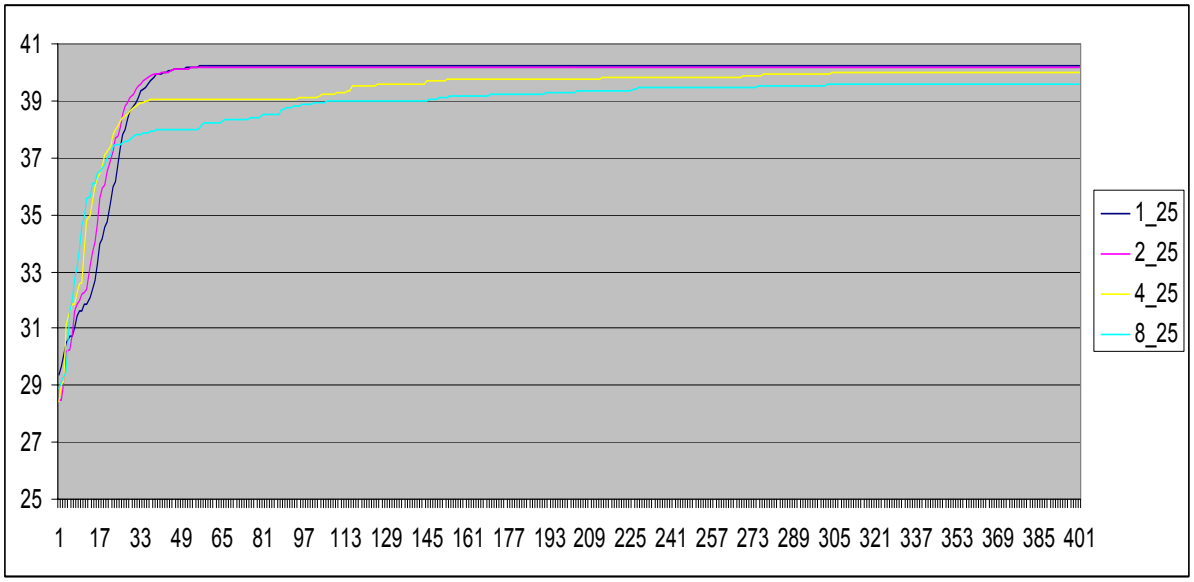

Figura 6.3. Rastrigin con frecuencia fija

- Observaciones: En la función Rastrigin observamos un comportamiento análogo a la multimodal anterior. De nuevo, las configuraciones con más islas van acercándose progresivamente a las de menos poblaciones. Hemos determinado que por la naturaleza de la propia función, las configuraciones no llegan a cruzarse, quedando estancadas al final de la ejecución y ordenadas de menor a mayor número de procesadores.

6.2.2- Efecto de la frecuencia de intercambio

Sabemos que con diferentes frecuencias de intercambio se obtienen diferentes resultados. Veremos que, como cabe esperar, en general las configuraciones que tienen una frecuencia de intercambio más alta proporcionan mejores valores de fitness en un menor número de generaciones, ya que los mejores individuos se propagaran más rápidamente de unos procesadores a otros. Como vimos anteriormente, un número de poblaciones (procesadores) elevado nos será útil cuando dispongamos de frecuencias de intercambio altas o bien un número elevado de generaciones.

- Función: OneMax
- Parámetro Fijado: Sin mutación, número de procesadores 4
- Parámetro variable: Frecuencia de intercambio 10, 25, 50
- Gráfica:

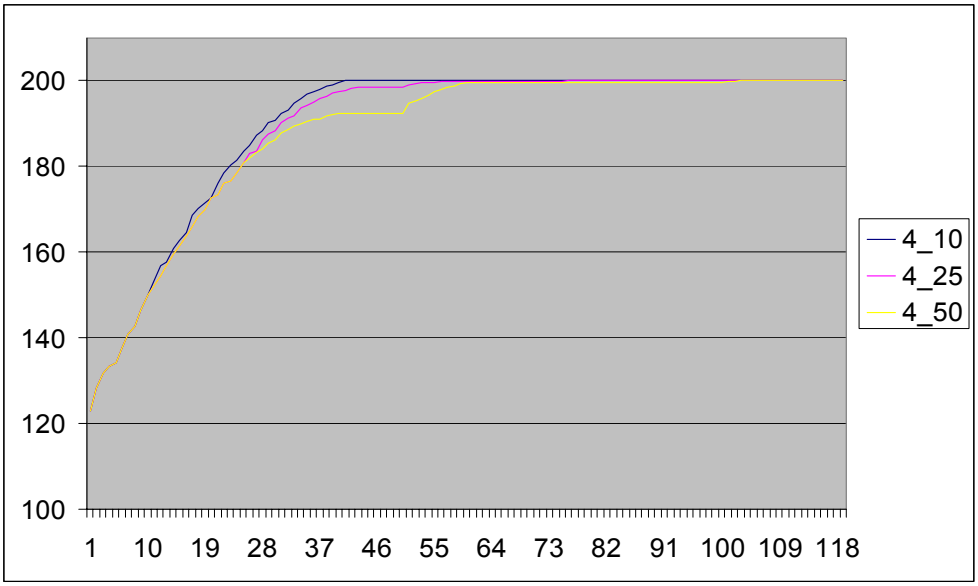

Figura 6.4. OneMax con nº poblaciones fijas

- Observaciones: Inicialmente las configuraciones con mayor frecuencia de intercambio llegan a la convergencia en menor número de generaciones. Una vez que se van produciendo los intercambios de las distintas configuraciones todas convergen al óptimo. El escalón producido por el intercambio es más acusado en aquellas de menor frecuencia ya que no se han desarrollado de manera tan progresiva.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, número de procesadores 8 y 4
- Parámetro variable: Frecuencia de intercambio 10, 25, 50
- Gráficas:

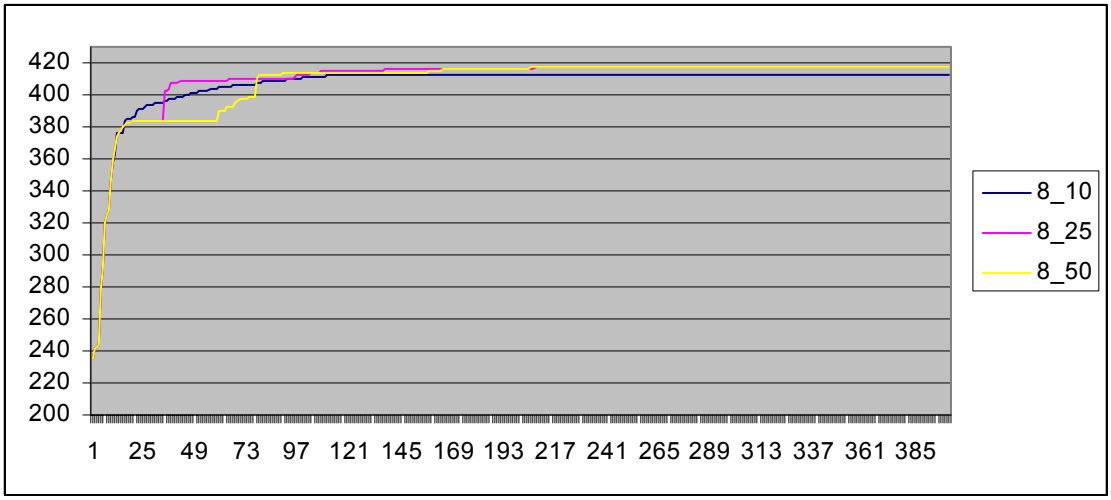

Figura 6.5. Schwefel con nº poblaciones fijas

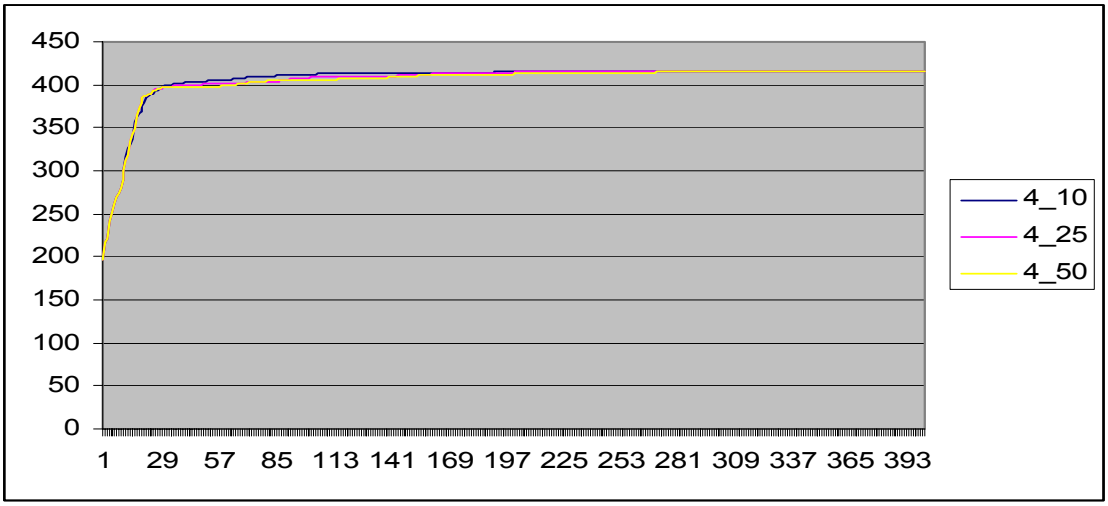

Figura 6.6. Schwefel con nº poblaciones fijas

- Observaciones: Inicialmente observamos que las configuraciones con menor frecuencia de intercambio se estancan antes y posteriormente el escalón producido al recuperarse de ese estancamiento es más acusado. Ocurre ya que se benefician más de recibir individuos que hayan convergido, con lo que aportan un valor de fitness alto a la población a la que llegan. Al final de la ejecución, las configuraciones quedan ordenadas de menor (50) a mayor (10) frecuencia. Esto es debido a que la población es tan pequeña (en el caso de 8 procesadores), que al

distanciar más los intercambios la diferencia de fitness entre unas islas y otras es mayor haciendo que el efecto del intercambio tenga consecuencias más acentuadas. En las configuraciones con 4 islas se encuentra la convergencia de las tres configuraciones a un mismo punto. Esta configuración tiene el tamaño de población intermedio de tal forma que la frecuencia no tiene un efecto significativo en el comportamiento. Sin embargo, poblaciones mayores que esta se verán favorecidas por una frecuencia de intercambio mayor.

- Función: F. Modal
- Parámetro Fijado: Sin mutación, número de procesadores 2
- Parámetro variable: Frecuencia de intercambio 10, 25, 50
- Gráfica:

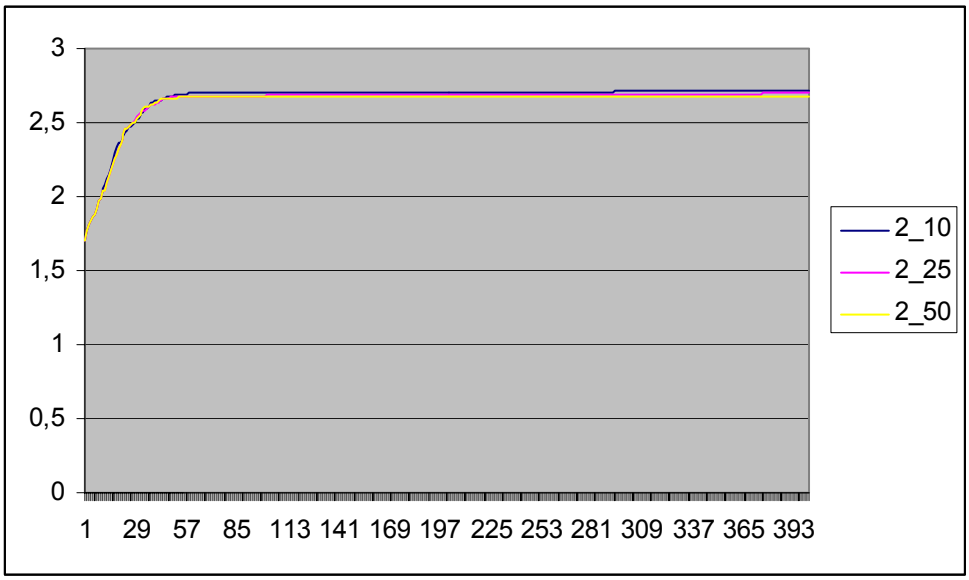

Figura 6.7. F. Modal con nº poblaciones fijas

- Observaciones: En estas configuraciones, el gran tamaño de las poblaciones hace que se vea favorecida por una frecuencia de intercambio alta. En esta función se observan muchos menos picos y menos acusados. En las últimas generaciones, las configuraciones quedan ordenadas con estabilidad de alta a baja frecuencia.

6.2.3- Efecto de la mutación.

Como comentamos en el estudio general, al introducir mutación en los algoritmos hay una mayor probabilidad de encontrar el óptimo debido a que se produce una mayor diversidad en la población, evitando que la función se quede en un óptimo local. El grado de aleatoriedad que se introduce con este operador nos permite obtener resultados pico que de otra forma serían más difíciles de desarrollar. El uso de la mutación produce que las configuraciones obtengan valores cuando menos muy próximos al óptimo disponiendo del número de generaciones que hemos fijado.

- Función: Schwefel
- Parámetro Fijado: Frecuencia 50 y Número de procesadores 4
- Parámetro variable: Mutación
- Gráfica:

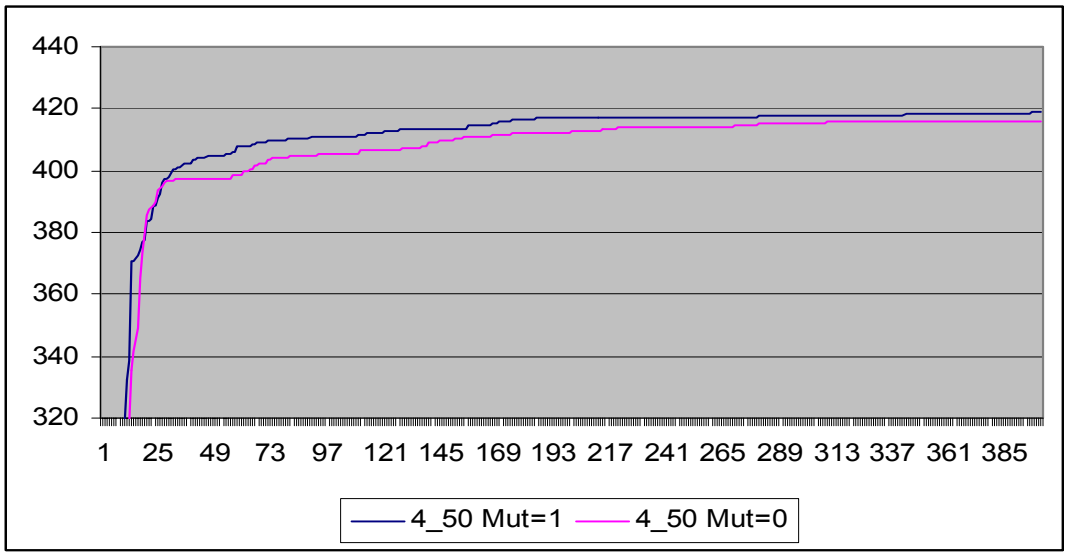

Figura 6.8. Schwefel con frecuencia y nº poblaciones fijas

- Observaciones: Se puede observar que el fitness es superior utilizando mutación en las mismas configuraciones de frecuencia y poblaciones. Además la diferencia es más acusada conforme disminuye el tamaño de la población debido al factor de diversidad que introduce el operador.

- Función: OneMax
- Parámetro Fijado: Frecuencia 10 y Número de procesadores 4
- Parámetro variable: Mutación
- Gráfica:

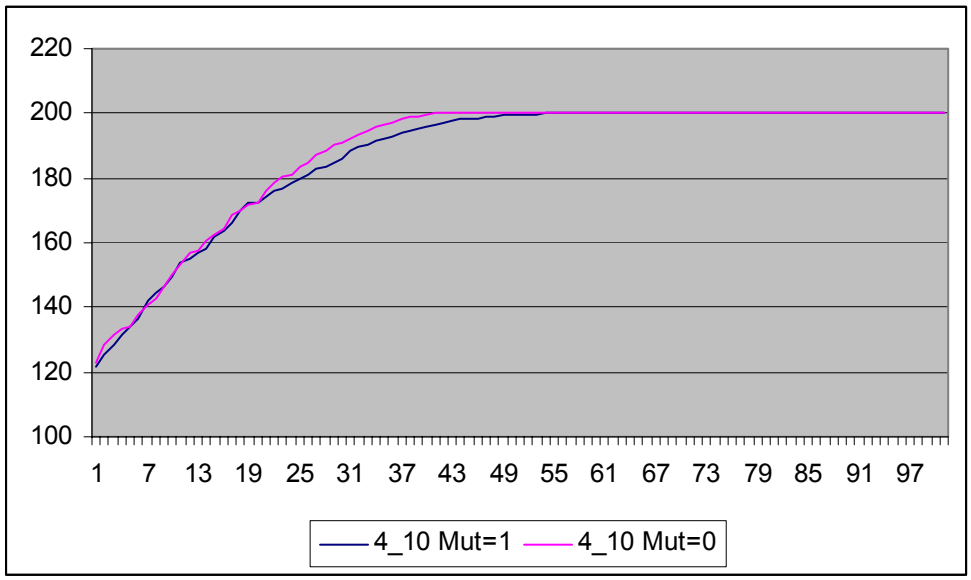

Figura 6.9. OneMax con frecuencia y nº poblaciones fijas

- Observaciones: En este caso podemos ver como la aleatoriedad introducida por el operador de mutación en esta configuración produce que un suavizado en la subida de la curva debido a que contrarresta la maximización del numero de 1's introducida por el cruce y la selección. De esta manera vemos como alcanza el fitness óptimo antes la configuración que no sufre mutación. La explicación se encuentra en la sencillez de la naturaleza de la función de modo que tan solo con los operadores de cruce y selección son suficientes para dirigir la búsqueda hacia el óptimo.

6.2.4.- Efecto del tamaño de la población.

Vamos a ver el efecto que supone el disponer de una población de menor tamaño. En el estudio anterior disponíamos de 240 individuos repartidos equitativamente entre los procesadores; en este caso utilizaremos 208 individuos.

- Función: Rastrigin
- Parámetro Fijado: Sin mutación, 2 procesadores, frecuencia de intercambio 10.
- Parámetro variable: número de individuos
- Gráfica:

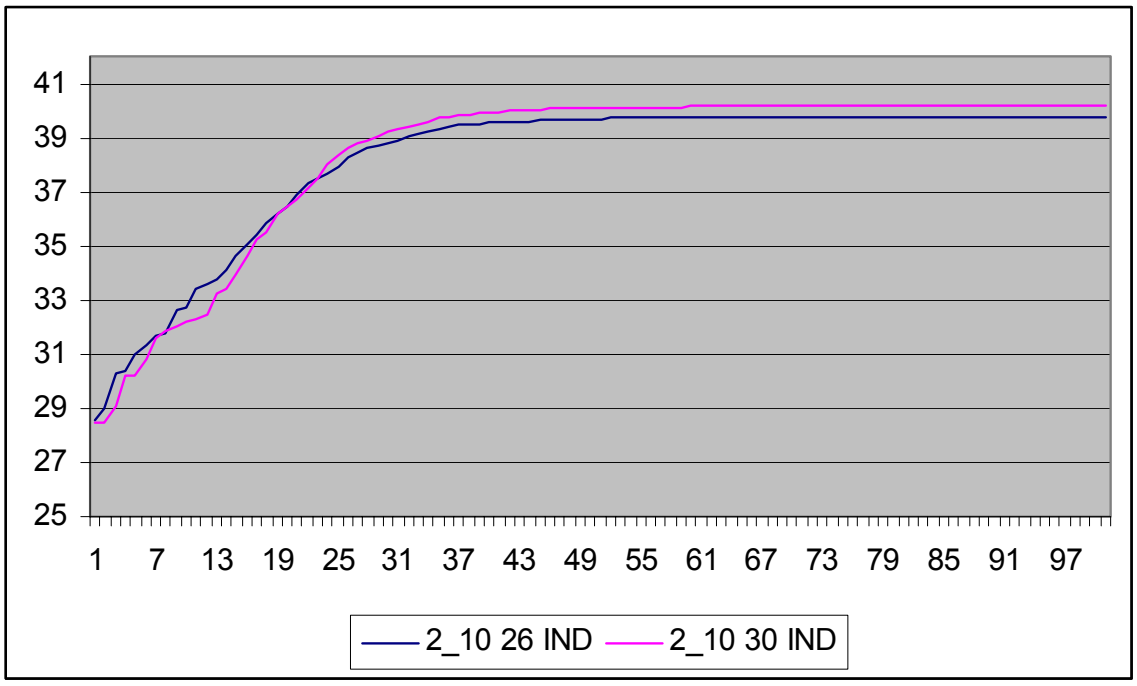

Figura 6.10. Rastrigin con variación del tamaño de la población

- Observaciones: Como podemos observar, el reducir el tamaño de la población a 208 individuos apenas afecta al fitness de los mejores elementos, aunque si se puede apreciar una pequeña mejoría al disponer de una población mayor.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, 8 procesadores, frecuencia de intercambio 50.
- Parámetro variable: número de individuos
- Gráfica:

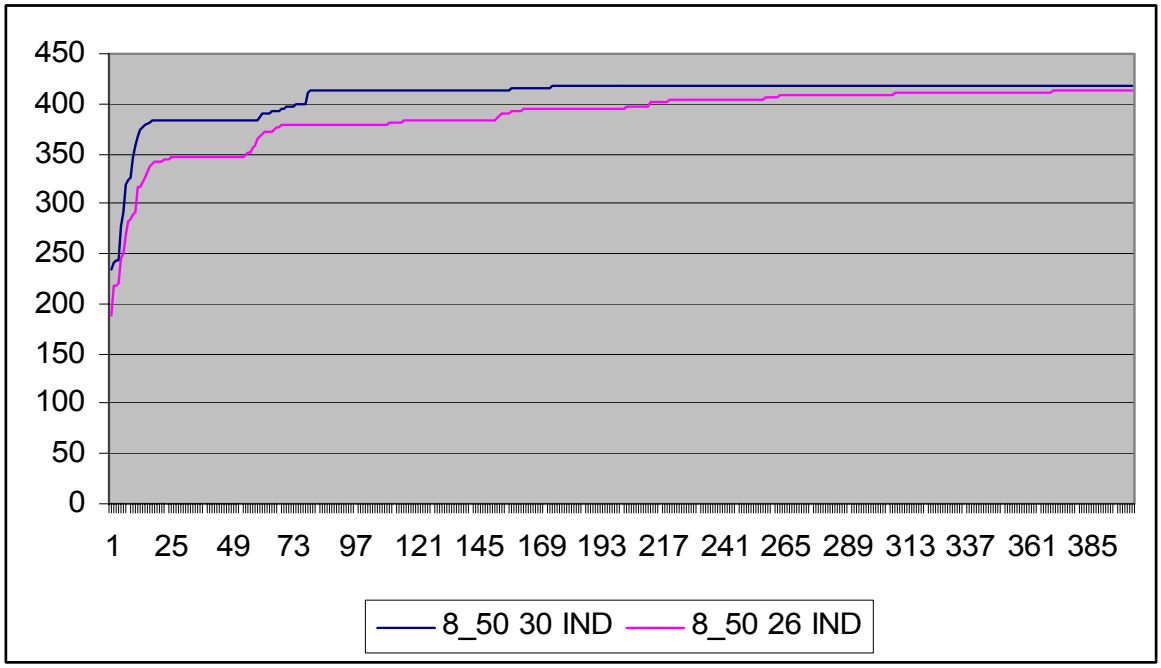

Figura 6.11. Schwefel con variación del tamaño de la población

- Observaciones: De nuevo se puede ver cómo con una población con más individuos se obtienen mejores resultados picos de fitness. En este caso es más acentuado que en el caso anterior de 2 procesadores debido a que ahora las poblaciones con 8 procesadores son más pequeñas.

6.2.5. Conclusiones generales.

Si disponemos de un número elevado de procesadores parece conveniente establecer una frecuencia de intercambio baja. De este modo permitimos que esas poblaciones más pequeñas separadas se desarrollen más y, así el intercambio favorezca más al incremento del fitness.

Si por el contrario disponemos de menos procesadores será conveniente establecer una frecuencia de intercambio más alta, de manera que los peores individuos de la población se desechen más frecuentemente.

El punto de inflexión se encuentra en las poblaciones de tamaño intermedio. Aquí parece que la frecuencia de intercambio apenas produce efectos apreciables.

Se puede concluir por lo tanto que la frecuencia de intercambio y el número de procesadores son inversamente proporcionales.

Parece muy apropiado hacer uso del operador de mutación ya que sin duda favorece la generación de individuos de fitness mejores al ayudar a diversificar la población en momentos de estancamiento.

También se ha comprobado que al tener las poblaciones mayor número de individuos se obtienen valores de fitness mejores.

6.2. Evolución del fitness medio.

A continuación vamos a estudiar la evolución del fitness promedio de todos los individuos a lo largo de las generaciones. Obtendremos el resultado medio de entre las 10 ejecuciones con distintas semillas.

6.2.1. Efecto del número de procesadores.

Vemos de nuevo el efecto del número de procesadores en el fitness medio de las poblaciones. Trataremos de encontrar el número óptimo de procesadores y evitar así un exceso de recursos innecesarios. De nuevo el número de individuos por población depende directamente del número de procesadores de la prueba.

- Función: OneMax
- Parámetro Fijado: Sin mutación, frecuencia
- Parámetro variable: Número de procesadores 1,2,4,8
- Gráfica:

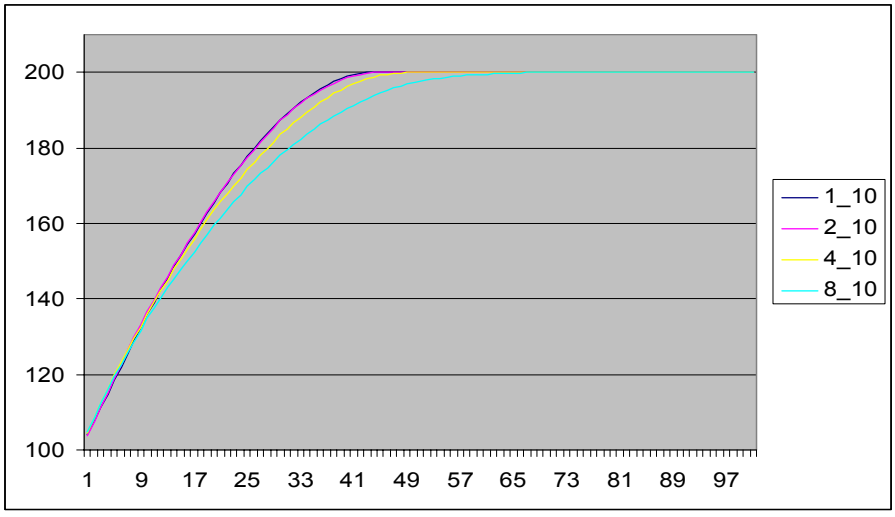

Figura 6.12. OneMax medias con frecuencia fija

- Observaciones: Como en los mejores vemos en la gráfica que todas las configuraciones convergen, siendo las de mayor número de poblaciones las que tardan algunas generaciones más. La convergencia del fitness medio de toda la población se efectúa aproximadamente unas 10 generaciones después de haber convergido el fitness del mejor.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, Frecuencia de intercambio 10
- Parámetro variable: Número de procesadores 1,2,4,8
- Gráfica:

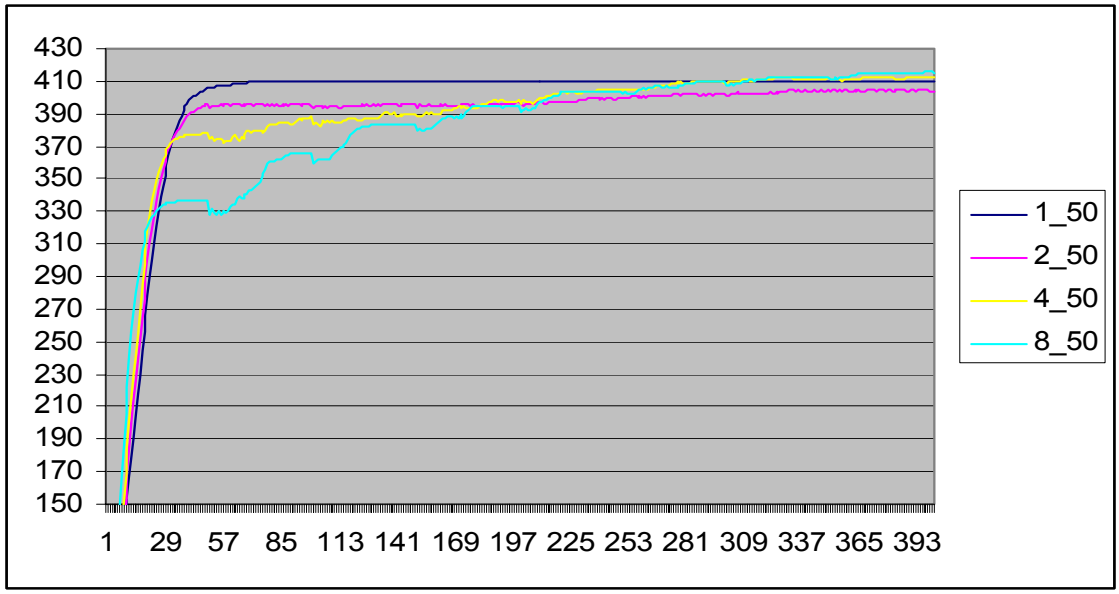

Figura 6.13. Schwefel medias con frecuencia fija

- Observaciones: Nuevamente cuanto mayor número de procesadores, el valor de fitness medio de la población converge a un valor menor en la primeras iteraciones. Conforme se van realizando los intercambios la situación se va invirtiendo según avanzan las generaciones quedando las configuraciones con más poblaciones por encima. Se observa que en las iteraciones donde se producen intercambios la media del fitness baja debido a que se pueden estar intercambiando individuos con peor fitness que los existentes en la iteración anterior, pero que al ser diferentes la aleatoriedad que introducen hace que en pocas recombinaciones la población mejore sus valores de fitness. En la última generación se observa que el valor de fitness de las configuraciones está establemente ordenada de mayor a menor número de poblaciones.

- Función: Rastrigin
- Parámetro Fijado: Sin mutación, Frecuencia de intercambio 25
- Parámetro variable: Número de procesadores 1,2,4,8
- Gráfica:

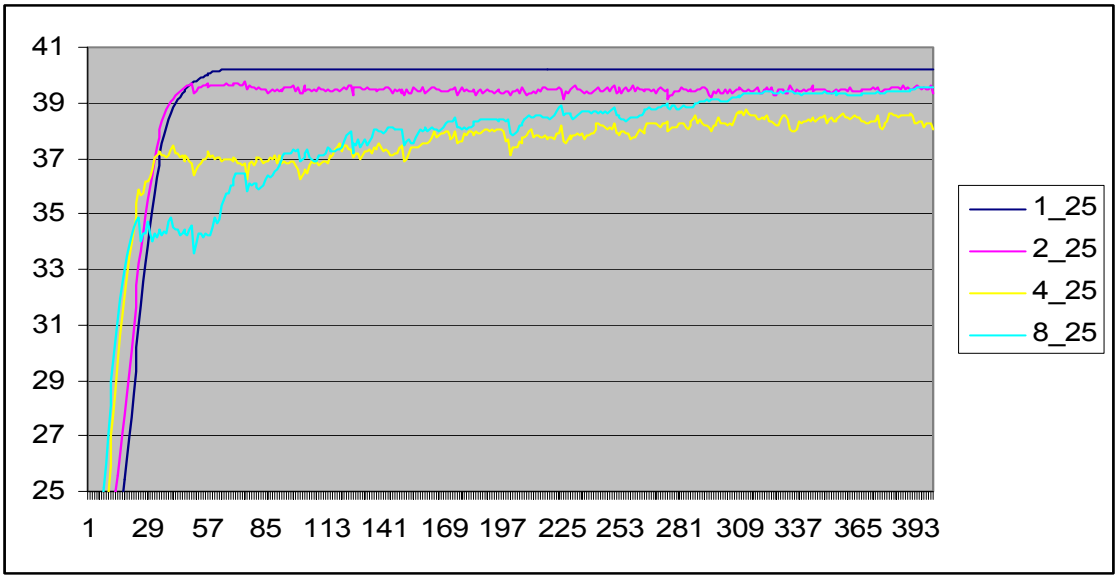

Figura 6.14. Rastrigin medias con frecuencia fija

- Observaciones: En la función Rastrigin observamos un comportamiento análogo a la grafica de los mejores (figura 2.3). De nuevo, las configuraciones con más islas van acercándose progresivamente a las de menos poblaciones. En la graficas de medias de toda la población se observa que en las configuraciones con poblaciones más pequeñas (8 poblaciones con 30 individuos) el efecto de conservar el mismo número de mejores que en otras configuraciones con poblaciones mayores produce que el valor del fitness medio de las primeras supere al de estas poblaciones, como se ve en relación de trazas entre 4 procesadores y 8. La desviación entre los valores de una misma traza se acentúa conforme aumenta el número de poblaciones. Se debe a que las poblaciones grandes son más homogéneas al aplicarse el cruce para cada generación entre muchos más individuos.

6.2.2. Efecto de la frecuencia de intercambio.

En general las configuraciones que tiene una frecuencia de intercambio más alta proporcionan mejores valores de fitness en un menor número de generaciones, ya que los mejores individuos se propagaran más rápidamente de unos procesadores a otros.

- Función: OneMax
- Parámetro Fijado: Sin mutación, número de procesadores 4
- Parámetro variable: Frecuencia de intercambio 10, 25, 50
- Gráfica:

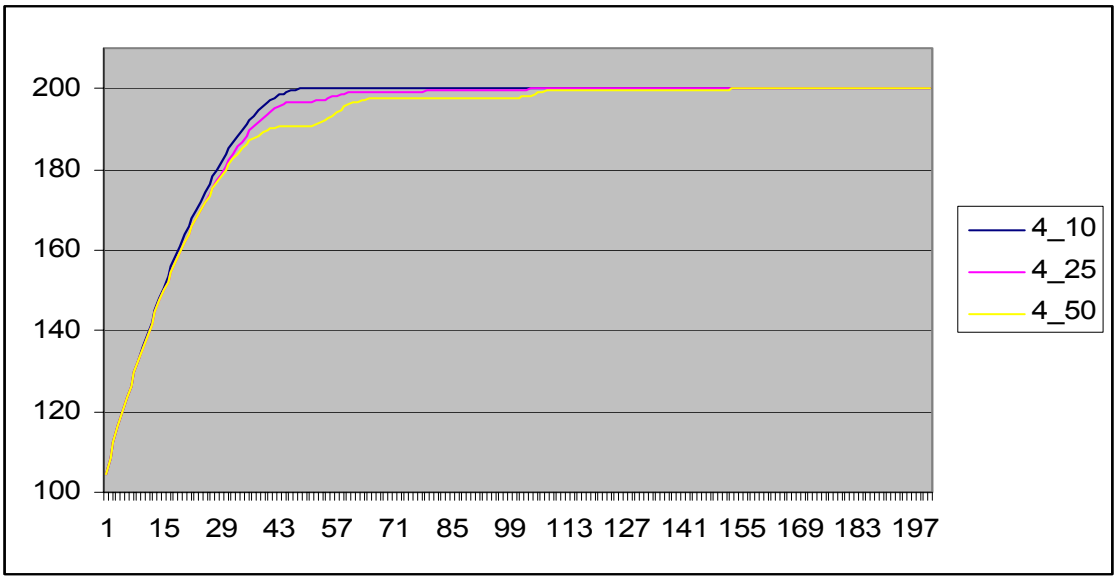

Figura 6.15. OneMax medias con nº poblaciones fijas

- Observaciones: Inicialmente las configuraciones con mayor frecuencia de intercambio llegan a la convergencia en menor número de generaciones. Una vez que se van produciendo los intercambios de las distintas configuraciones todas convergen al óptimo. El escalón producido por el intercambio es más acusado en aquellas de menor frecuencia.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, número de procesadores 8 y 4
- Parámetro variable: Frecuencia de intercambio 10, 25, 50

- Gráfica:

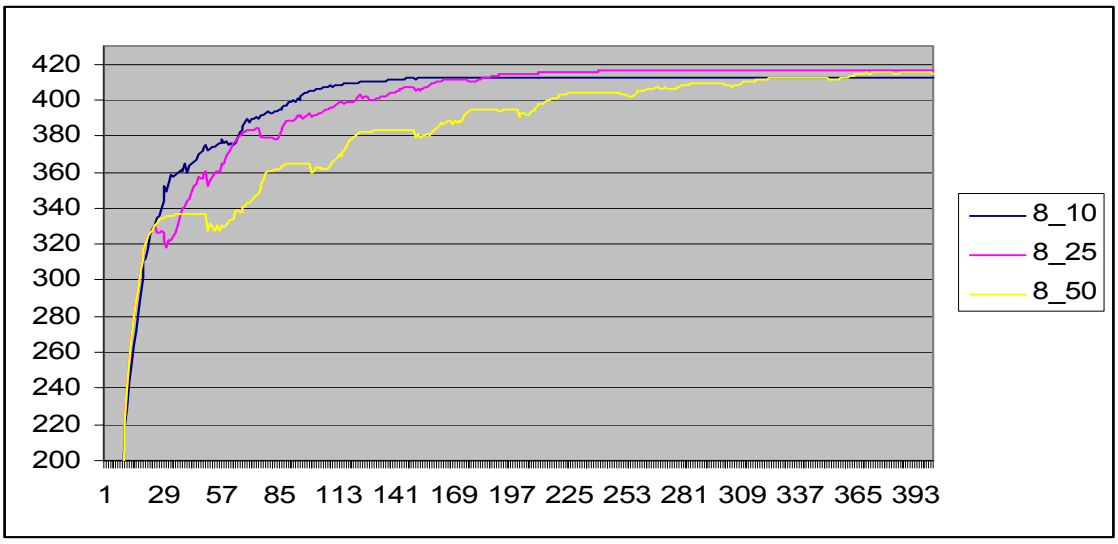

Figura 6.16. Schwefel medias con nº poblaciones fijas

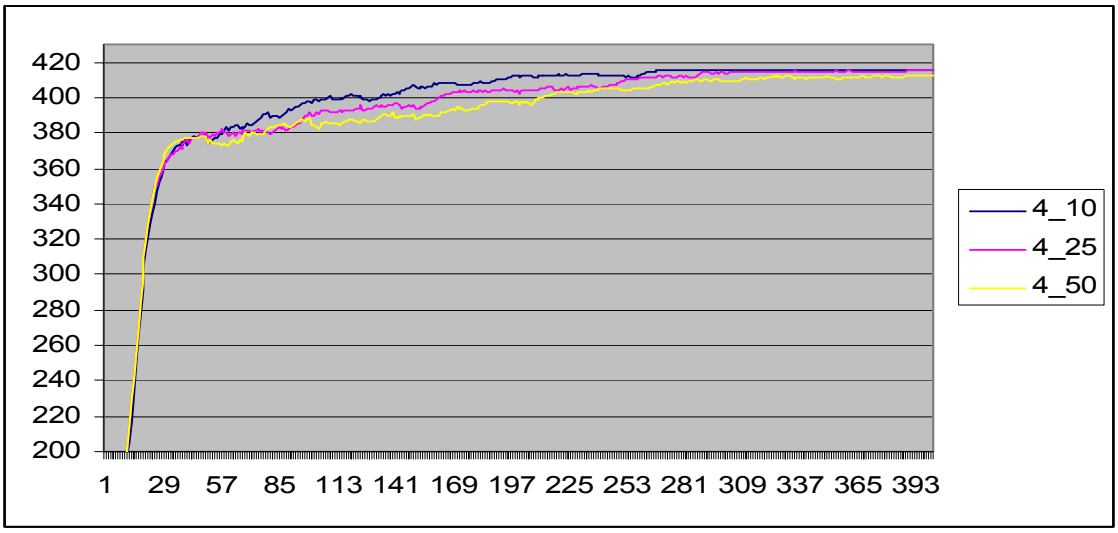

Figura 6.17. Schwefel medias con nº poblaciones fijas

- Observaciones: En las primeras iteraciones observamos que las configuraciones con menor frecuencia de intercambio se estancan antes y el escalón producido posteriormente es más acusado. Los escalones son mas cortos y hay más cantidad con respecto a la grafica del mejor (figura 2.5). Esto indica que las poblaciones de menor tamaño requieren más intercambios para la convergencia del fitness medios. Con poblaciones de tamaño intermedio las diferencias son más pequeñas entre configuraciones de distintas frecuencias para el mismo número de poblaciones.

- Función: F. Modal
- Parámetro Fijado: Sin mutación, número de procesadores 2
- Parámetro variable: Frecuencia de intercambio 10, 25, 50
- Gráfica:

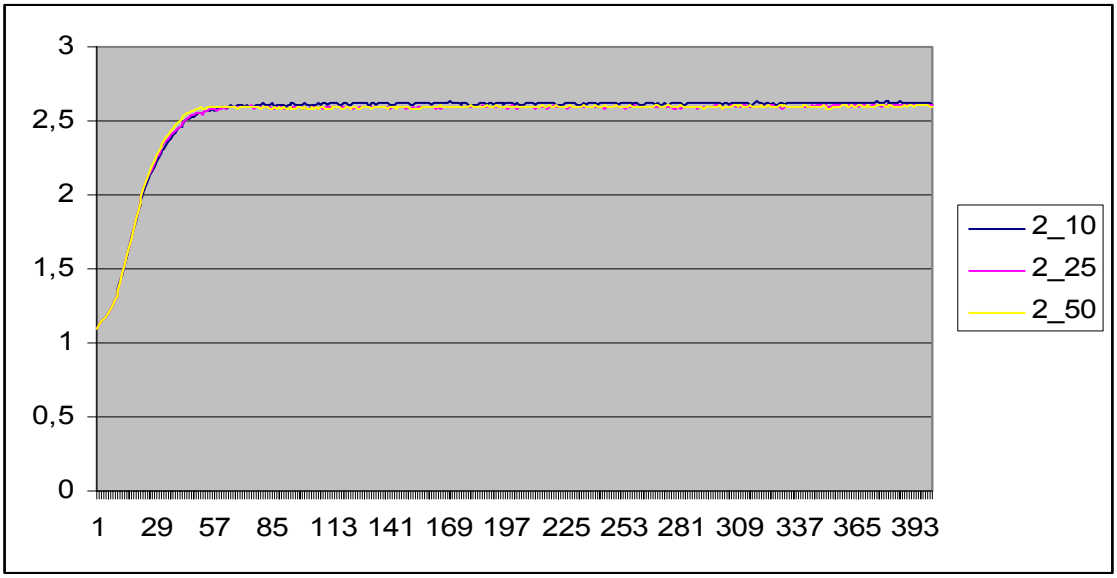

Figura 6.18. F. modal medias con nº poblaciones fijas

- Observaciones: Observamos en esta grafica que la convergencia del fitness medio se realiza varias generaciones después y la diferencia con distintas frecuencias de intercambio es mínima.

6.2.3. Efecto de la mutación.

Veremos que la aleatoriedad introducida por este operador reduce el grado de desarrollo de las poblaciones obteniéndose peores valores promedios de fitness. De modo que si tenemos un problema en el que nos interese que en promedio todos los individuos sean mejores, aunque no lleguemos a alcanzar el óptimo, no deberíamos introducir la mutación, y si lo que deseamos es resolver un problema de optimización, la mutación nos puede ayudar a alcanzar ese valor.

- Función: Schwefel
- Parámetro Fijado: Frecuencia 50 y Número de procesadores 4
- Parámetro variable: Mutación
- Gráfica:

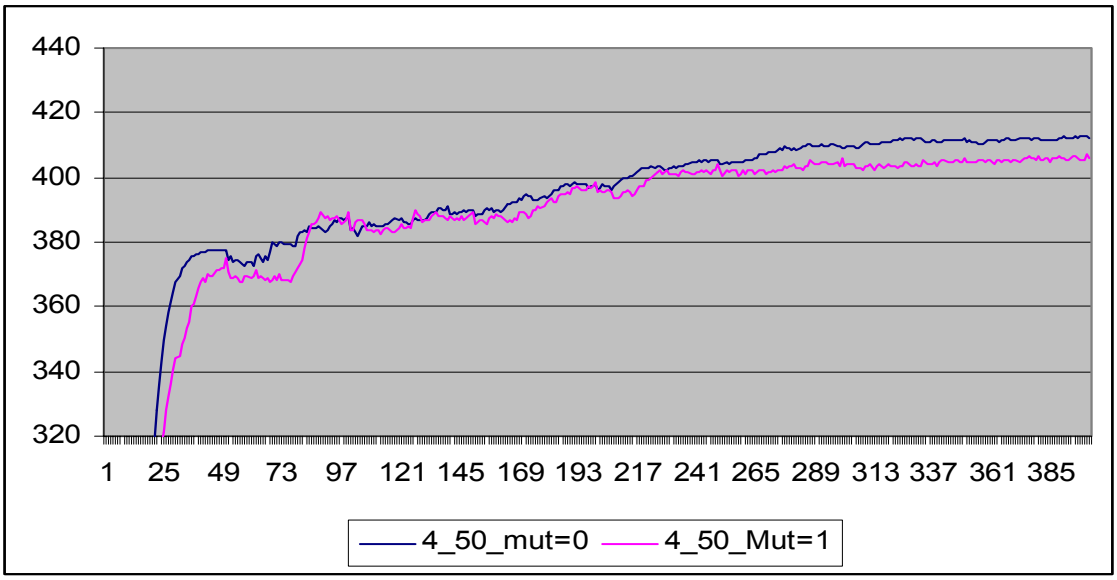

Figura 6.19. Schwefel medias con nº poblaciones y frecuencia fijas

- Observaciones: Hemos observado que al introducir el operador de mutación los valores de fitness medio son peores que los obtenidos sin utilizarla.

6.2.*4. Efecto del tamaño de la población.*

- Función: Schwefel
- Parámetro Fijado: Sin mutación, 4 procesadores, frecuencia de intercambio 25
- Parámetro variable: número de individuos
- Gráfica:

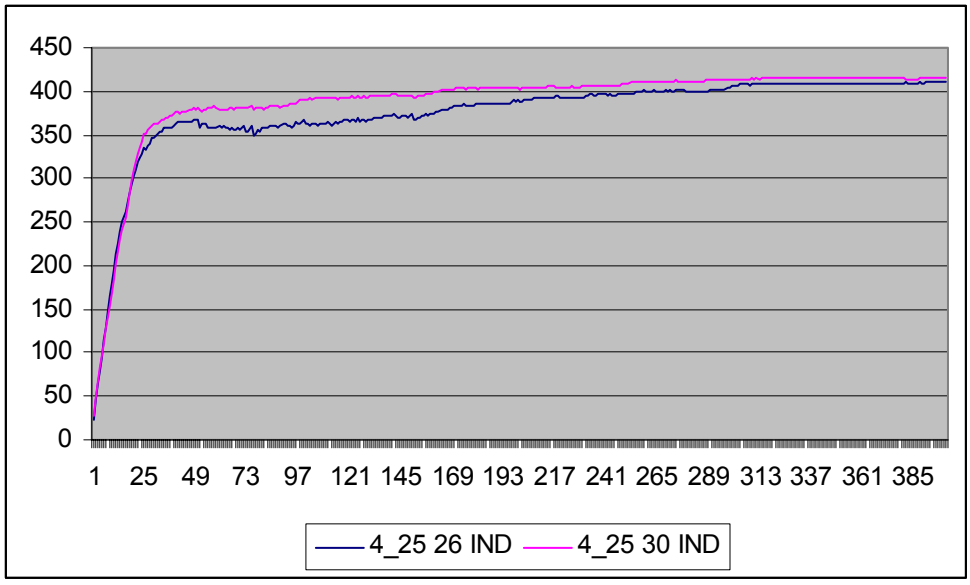

Figura 6.20. Rastrigin con variación del tamaño de la población

- Observaciones: Podemos observar que con poblaciones de mayor tamaño el fitness promedio de los individuos es superior al de las poblaciones más pequeñas. Vemos que en las últimas generaciones los valores se aproximan bastante, llegando a igualarse si dispusiéramos de más esfuerzo de cómputo.

- Función: Rastrigin
- Parámetro Fijado: Con mutación, 6 procesadores, frecuencia de intercambio 25
- Parámetro variable: número de individuos
- Gráfica:

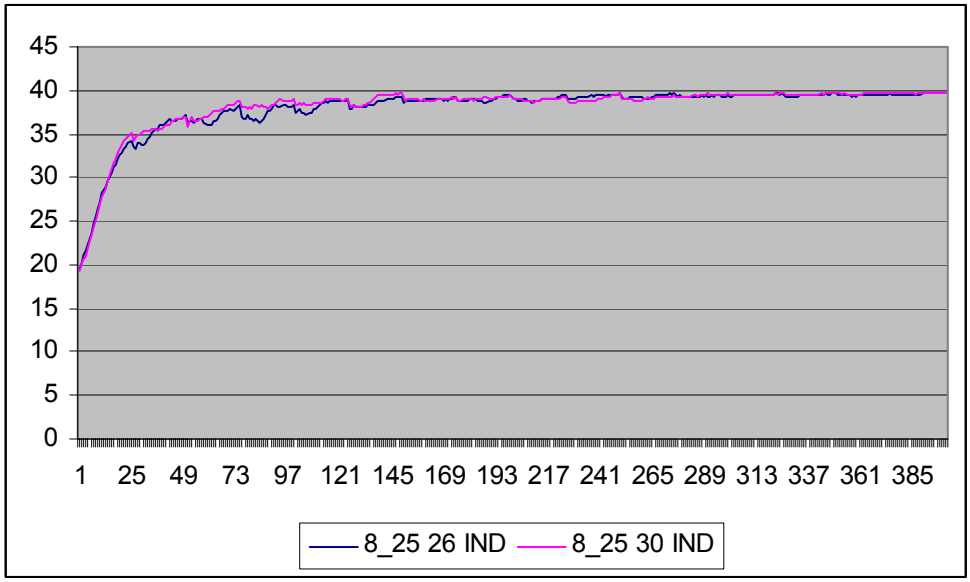

Figura 6.21. Rastrigin con variación del tamaño de la población

- Observaciones: En este caso, aunque en un principio parece que la configuración con mayor número de individuos es ligeramente mejor, el efecto del operador de mutación introduce una aleatoriedad similar a la que podría introducir el disponer de algunos individuos más, por lo que acaban siendo prácticamente iguales a mitad de ejecución.

3.5. Conclusiones generales.

De este estudio obtenemos que la configuración más adecuada para tener la media de toda la población más alta posible, serán poblaciones grandes, con alta frecuencia de intercambio y sin mutación. Esto nos proporcionará muchas soluciones “buenas” pero no necesariamente una óptima. Esto será útil para problemas en los que nos interese obtener una gran cantidad de soluciones suficientemente buenas aunque se pueda prescindir de que sea óptima. Estos problemas existen en ingeniería como por ejemplo puede ser la distribución o partición de circuitos.

6.3. Estudio de la convergencia y comparación de los mejores y peores individuos de las poblaciones.

A continuación vamos a ver la desviación de valores de fitness que encontramos en las poblaciones. Veremos como evolucionan los valores promedio de los mejores y peores individuos de las poblaciones.

6.3.1.- Efecto del número de procesadores.

Tratamos de encontrar el número óptimo de procesadores y evitar así un exceso de recursos innecesarios y como afecta esto al intervalo entre mejores y perores de las poblaciones.

- Función: FTrap
- Parámetro Fijado: Sin mutación, frecuencia 25
- Parámetro variable: Número de procesadores 4, 8
- Gráfica:

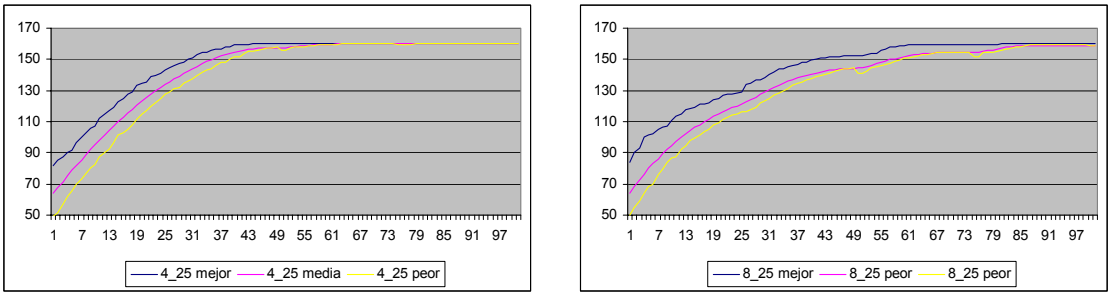

Figura 6.22. Ftrap con frecuencia fija 4 y 8 procesadores

- Observaciones: Vemos en la gráfica que todas las configuraciones convergen, siendo las de mayor número de poblaciones las que tardan alguna generaciones más. Esto se debe a la diversidad contenida en las poblaciones grandes. En la de más poblaciones se pueden apreciar los efectos de los intercambios en los valores de los fitness.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, Frecuencia de intercambio 50
- Parámetro variable: Número de procesadores 2,8
- Gráfica:

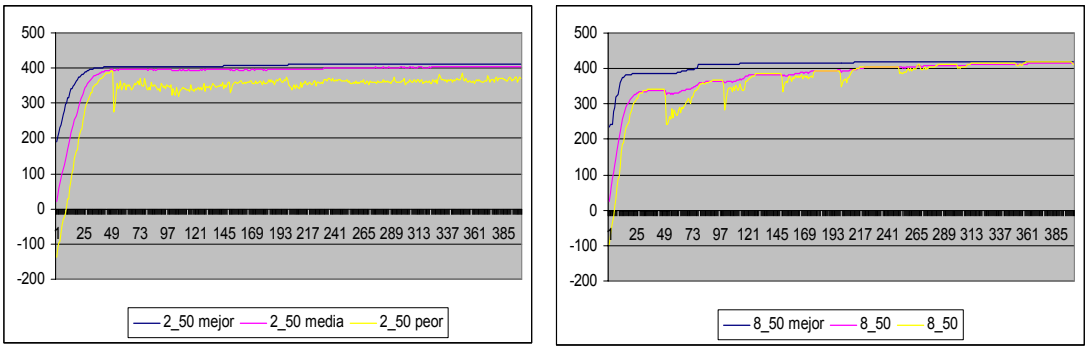

Figura 6.23. Schwefel con frecuencia fija

- Observaciones: Observamos que en las configuraciones con poblaciones grandes no convergen los valores peor y mejor debido a que hay mucha mas diversidad de individuos. En el caso de ocho procesadores, con población, por tanto, más pequeña, al final de las generaciones lanzadas prácticamente han convergido. En los peores se puede apreciar el efecto del intercambio y como los elementos recibidos tienen un fitness menor.

6.3.2- Efecto de la frecuencia de intercambio

Sabemos que con diferentes frecuencias de intercambio se obtienen diferentes resultados.

- Función: OneMax
- Parámetro Fijado: Sin mutación, número de procesadores 8
- Parámetro variable: Frecuencia de intercambio 10, 50
- Gráfica:

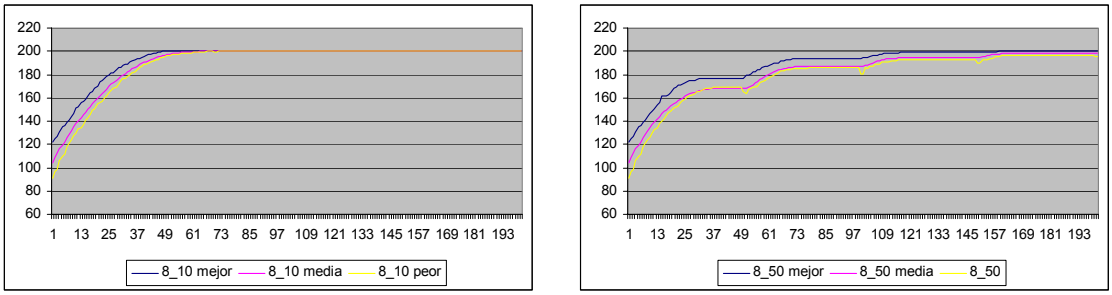

Figura 6.24. OneMax con nº poblaciones fijas

- Observaciones: Inicialmente las configuraciones con mayor frecuencia de intercambio llegan a la convergencia en menor número de generaciones. El escalón producido por el intercambio es más acusado en aquellas de menor frecuencia ya que no se han desarrollado de manera tan progresiva.

- Función: F. Modal
- Parámetro Fijado: Sin mutación, número de procesadores 4
- Parámetro variable: Frecuencia de intercambio 10, 25
- Gráfica:

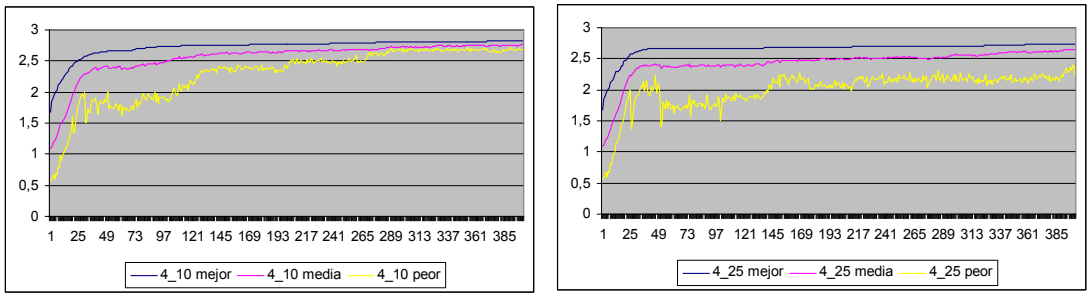

Figura 6.25. F. Modal con nº poblaciones fijas

- Observaciones: Como ya hemos comprobado con anterioridad se confirma que las configuraciones con frecuencia de intercambio alta favorecen la convergencia de la población.

6.3.3- Efecto de la mutación.

Como comentamos en el estudio general, al introducir mutación en los algoritmos hay una mayor probabilidad de encontrar el óptimo debido a que se produce una mayor diversidad en la población, evitando que la función se quede en un óptimo local. El grado de aleatoriedad que se introduce con este operador nos permite obtener resultados pico que de otra forma serían más difíciles de desarrollar. El uso de la mutación produce que las configuraciones obtengan valores cuando menos muy próximos al óptimo disponiendo del número de generaciones que hemos fijado.

- Función: Schwefel
- Parámetro Fijado: Frecuencia 50 y Número de procesadores 4
- Parámetro variable: Mutación
- Gráfica:

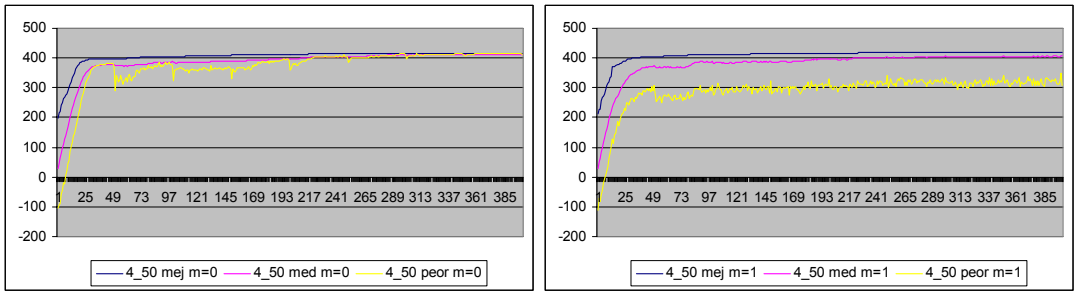

Figura 6.26. Schwefel con frecuencia y nº poblaciones fijas

- Observaciones: Se puede observar que el fitness es superior utilizando mutación en las mismas configuraciones de frecuencia y poblaciones. Se comprueba que la diferencia entre mejores y peores es mucho mas acusada con mutación ya que este operador introduce la aleatoriedad que impide desarrollar y converger a toda una misma población.

6.3.4. Conclusiones generales.

Destaca del estudio realizado que será más beneficioso desde el punto de vista de la convergencia del fitness de toda la población, (el fitness de mejores y peores se aproximan), el uso de configuraciones con poblaciones grandes, frecuencia de intercambio alta y sin mutación. Habrá que tener en cuenta que esta configuración no deberá necesariamente obtener el valor óptimo de la función.

6.4. Estudio del porcentaje de óptimos.

Veremos en este estudio qué porcentaje de individuos alcanzan el valor máximo que se alcanzará en su población en una ejecución. Este valor máximo no tiene por qué ser el óptimo global, ya que lo que nos interesa ver es cómo los mejores individuos hacen evolucionar a su población. A este valor máximo nos referiremos como MVP (máximo valor por población).

6.4.1.- Efecto del número de procesadores.

Vamos a ver el efecto de disponer un número mayor o menor de procesadores para ver si las poblaciones locales convergen a sus MVP.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, frecuencia de intercambio 10
- Parámetro variable: número de procesadores
- Gráfica:

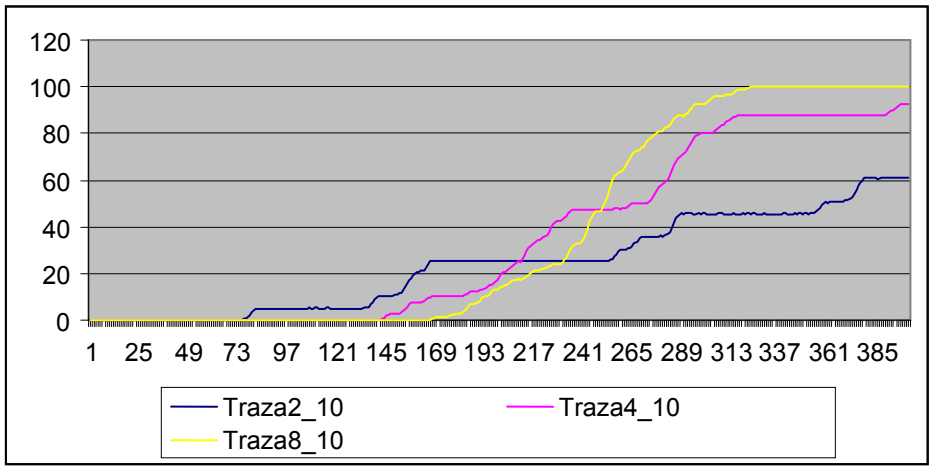

Figura 6.27. Schwefel con frecuencia fija

- Observaciones: Observamos que en el caso de mayor número de poblaciones, el valor máximo tarda más en aparecer, sin embargo, al tener menos individuos cada procesador, este valor se propaga más rápidamente entre la población local. De hecho, sólo en la configuración con 8 procesadores toda la población local converge al MVP.

6.4.2.- Efecto de la frecuencia de intercambio.

Estudiamos ahora el efecto de variar la frecuencia de intercambio para ver cómo se propagan los mejores y ver cómo afecta esto a la convergencia de las poblaciones.

- Función: F. modal
- Parámetro Fijado: Sin mutación, 4 procesadores
- Parámetro variable: frecuencia de intercambio 10, 25, 50.
- Gráfica:

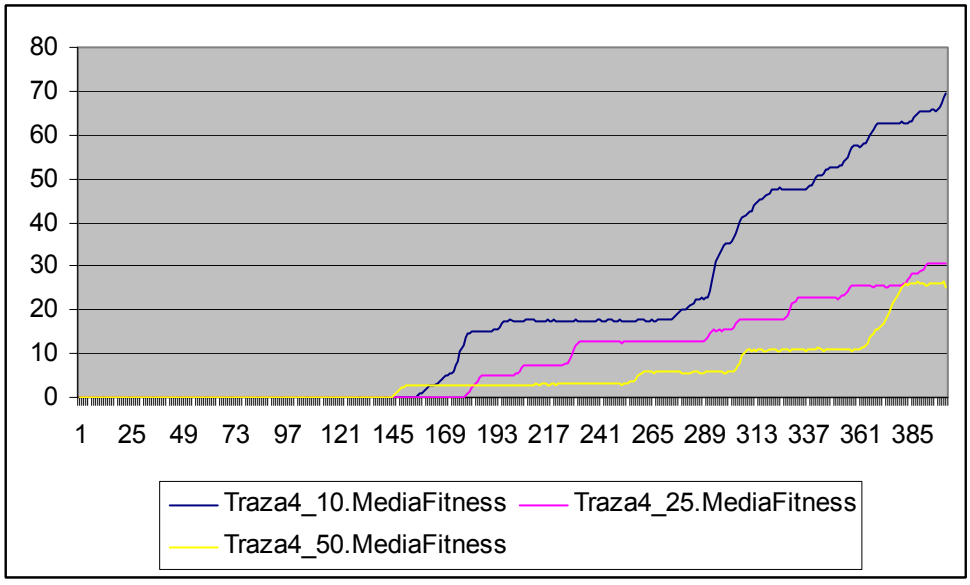

Figura 6.28. F. modal con número de procesadores fijo

- Observaciones: Se puede observar que al tener una frecuencia de intercambio elevada, los mejores individuos se propagan antes y por lo tanto el resto de la población tiene más tiempo para evolucionar al valor MVP.

6.4.3.- Efecto de la mutación:

Vamos a ver ahora el efecto que produce en la convergencia de las poblaciones la aleatoriedad que proporciona el operador de mutación.

- Función: FTrap
- Parámetro Fijado: frecuencia de intercambio 10, 8 procesadores.
- Parámetro variable: mutación
- Gráfica:

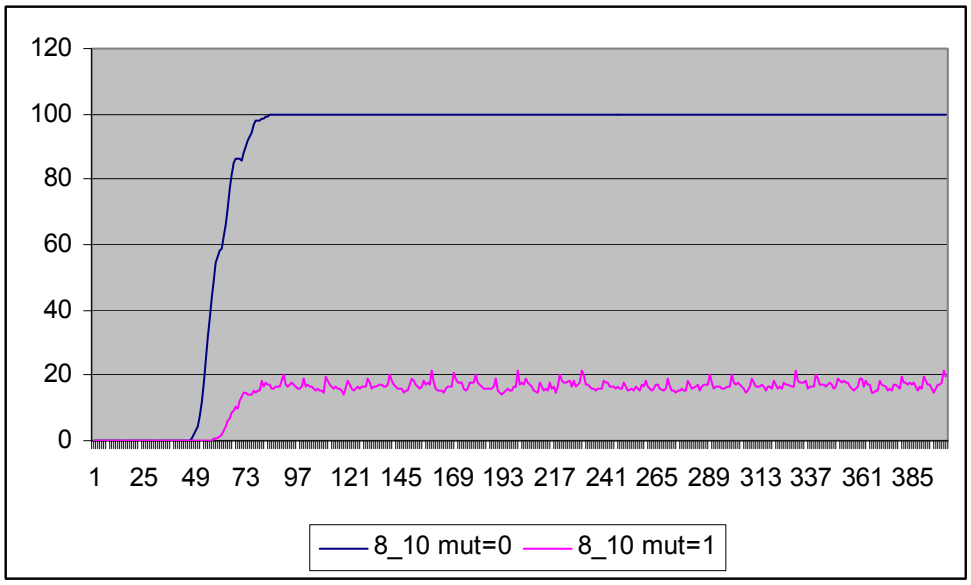

Figura 6.29. FTrap con frecuencia de intercambio y número de procesadores fijo

- Observaciones: Podemos ver, como ya hemos comentado en varias ocasiones, que la aleatoriedad que conlleva la mutación de los individuos no permite desarrollarse a la población, haciendo que los valores MVP oscilen continuamente. Esto explica que, como se observa en la gráfica, en el caso de hacer uso de la mutación sólo un porcentaje pequeño de las poblaciones converge al valor MVP, mientras que sin hacer uso de la misma las poblaciones convergen a este valor en un número relativamente pequeño de generaciones.

6.4.4.- Conclusiones generales:

Según lo visto las configuraciones con número elevado de poblaciones con pocos individuos cada una y una frecuencia de intercambio alta, que favorezca los intercambios rápidos, hacen que converjan a un optimo local. Esto no es deseable si el valor MVP es mucho menor que el óptimo global ya que hará que la población se estanque en ese valor. En este caso el uso del operador de mutación evitará que el MVP se estanque permitiendo a toda la población salir del óptimo local.

6.5. Estudio poblacional de los mejores individuos.

En este apartado estudiaremos la configuración de los mejores individuos de cada población, entendiendo ésta como los distintos procesadores que han intervenido en la formación de estos individuos. Mostraremos los resultados como porcentajes que indican el tanto por ciento de individuos que han pasado por un número determinado de procesadores.

6.5.1.- Efecto del número de procesadores.

El efecto del número de procesadores es interesante ya que influye en el tamaño de las poblaciones.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, frecuencia 25
- Parámetro variable: Poblaciones 4, 8
- Gráficas:

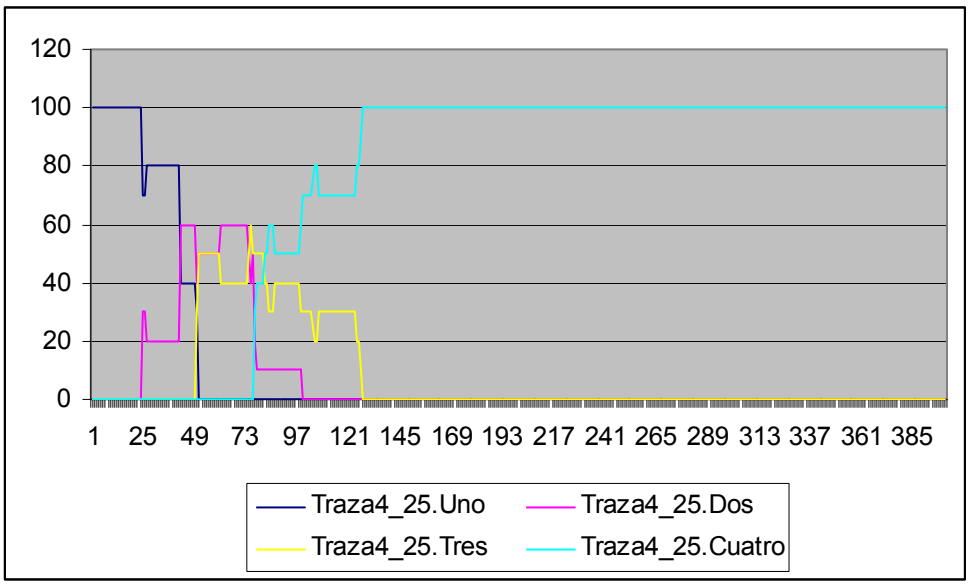

Figura 6.30. Schwefel poblacional frecuencia 25, 4 poblaciones

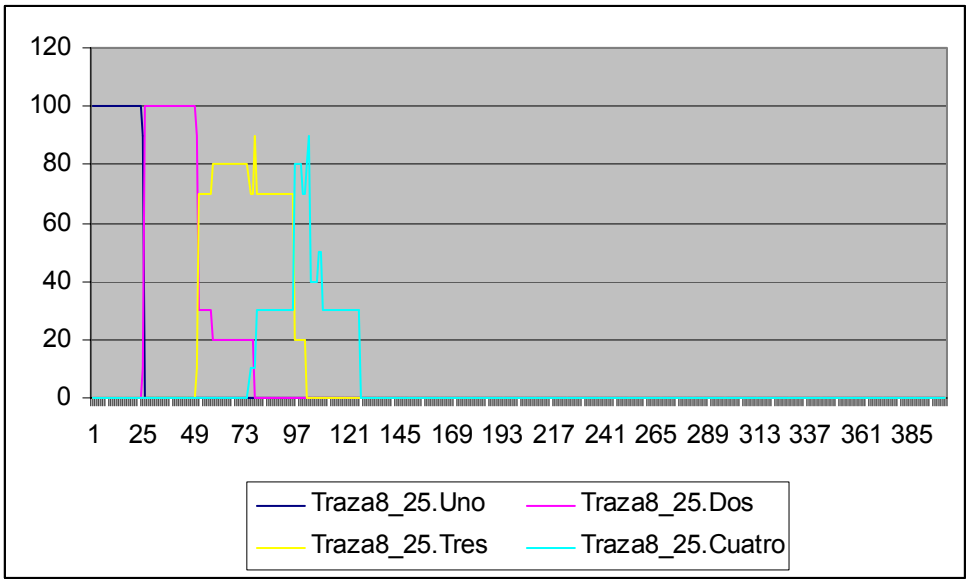

Figura 6.31. Schwefel poblacional frecuencia 25, 8 poblaciones

- Observaciones: Con poblaciones más grandes la transmisión de individuos foráneos tiene menos probabilidad con respecto al resto de individuos de la población por los que el número de generaciones con los que una configuración de procesadores ha contribuido a la formación de individuos es mayor. Por ejemplo podemos comprobar que para el porcentaje de 3 procesadores con 4 poblaciones (traza amarilla) se ubica entre las iteraciones 52 a la 129 y con 8 poblaciones la misma configuración se ubica ente las iteraciones 51 a la 102

6.5.2- Efecto de la frecuencia de intercambio

Vamos a estudiar el efecto de la frecuencia de intercambio, ya que ésta será la que determine la frecuencia con la que los individuos se mueven por las poblaciones.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, número de procesadores 4
- Parámetro variable: Frecuencia de intercambio 10, 50

- Gráficas:

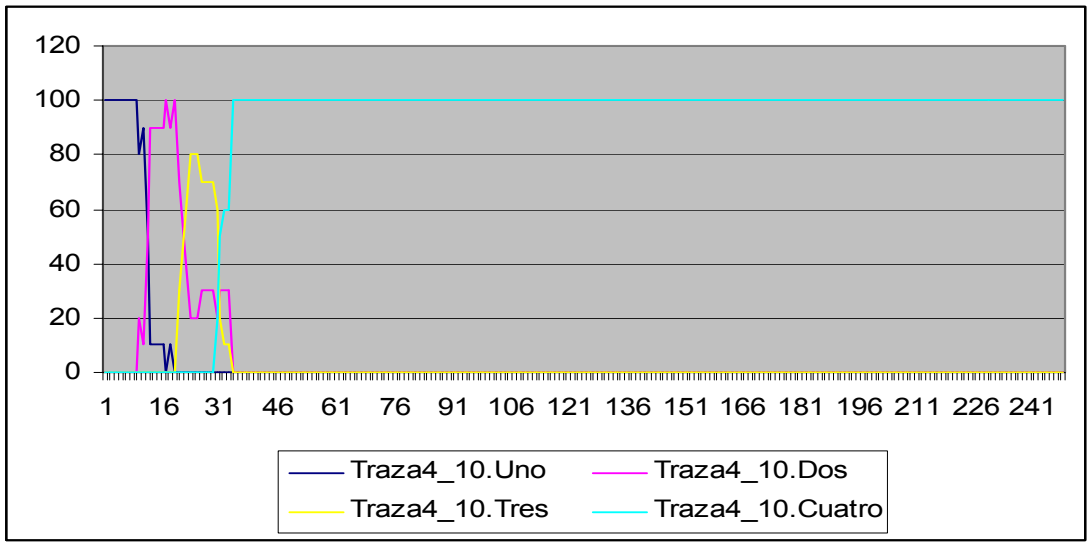

Figura 6.32. Schwefel poblacional frecuencia 10

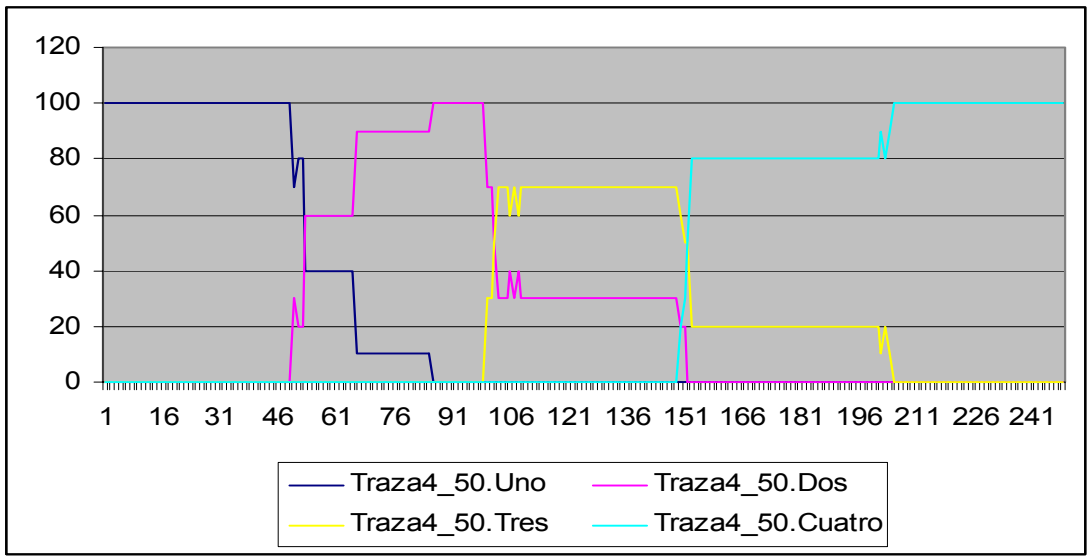

Figura 6.33. Schwefel poblacional frecuencia 50

- Observaciones: Se observa que al ser la frecuencia más baja en la segunda que en la primera grafica, tarda más en propagarse el mejor individuo a las demás poblaciones. Aún así en ambos casos el mejor necesita pasar por todas las poblaciones para llegar a ser óptimo. En la configuración de la figura 6.30 el óptimo se alcanza aproximadamente en la iteración 242 mientras que en la segunda configuración lo alcanza en la 364.

- Función: F. Modal
- Parámetro Fijado: Sin mutación, número de procesadores 8
- Parámetro variable: Frecuencia de intercambio 10, 50
- Gráfica:

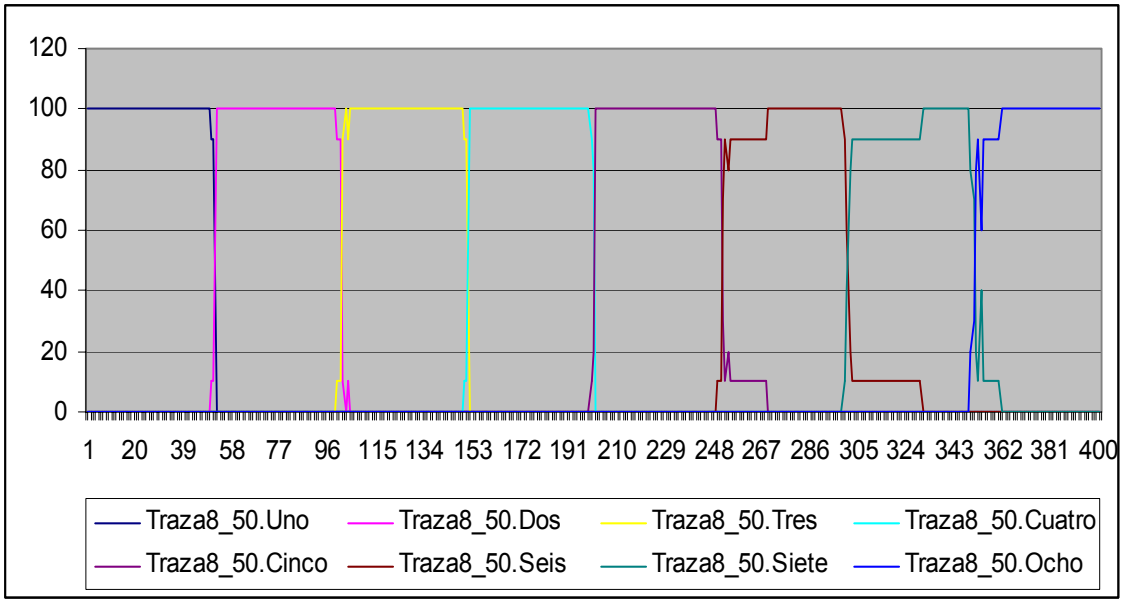

Figura 6.34. F. Modal poblacional frecuencia 50

- Observaciones: Incluso en el caso extremo de tener 8 poblaciones y frecuencia 50, el mejor individuo, que aparece aquí en la generación 364, ya ha pasado por todos los procesadores. Le ha sido suficiente con 7 intercambios para ello.

6.5.3- Efecto de la mutación.

Como comentamos en el estudio general, al introducir mutación en los algoritmos hay una mayor probabilidad de encontrar el óptimo debido a que se produce una mayor diversidad en la población, evitando que la función se quede en un óptimo local. El grado de aleatoriedad que se introduce con este operador nos permite obtener resultados pico que de otra forma serían más difíciles de desarrollar. El uso de la mutación produce que las configuraciones obtengan valores cuando menos muy próximos al óptimo disponiendo del número de generaciones que hemos fijado.

- Función: Rastrigin
- Parámetro Fijado: Frecuencia 50 y Número de procesadores 4
- Parámetro variable: Mutación

- Gráfica:

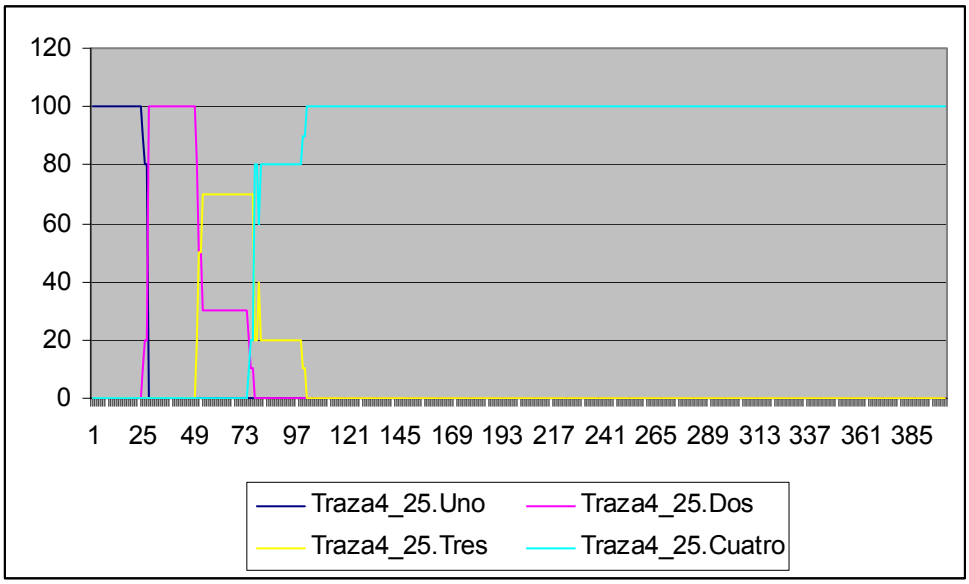

Figura 6.35. Rastrigin poblacional sin mutación

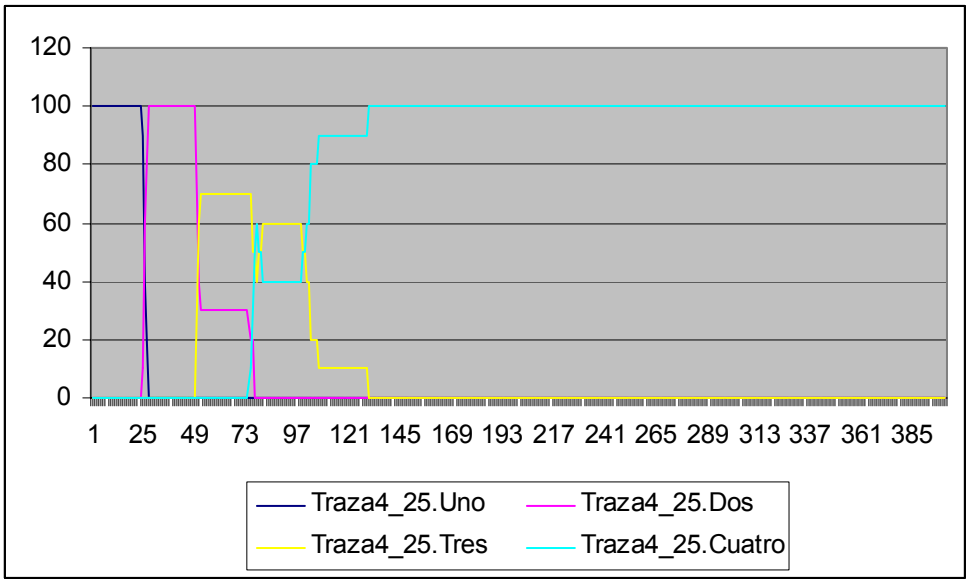

Figura 6.36. Rastrigin poblacional con mutación

- Observaciones: La mutación introduce aleatoriedad a la configuración. Esta hace que aumente la probabilidad de que los mejores individuos sean los nativos. De este modo aumenta levemente el número de generaciones necesario para que los procesadores contribuyan a la formación de todos los individuos. En la configuración sin mutación se alcanza en la generación 293 mientras que con el operador de mutación se alcanza en la generación 365. En los dos casos, por lo tanto, se ha pasado por todos los procesadores.

6.5.4. Conclusiones generales.

Según este estudio, para la formación de los mejores individuos es necesaria la intervención de todos los procesadores. Cada vez que se producen intercambios cambia drásticamente la configuración de las poblaciones gracias a la recombinación de los individuos.

6.6.- Estudio poblacional completo.

En el punto anterior estudiábamos la configuración poblacional de los mejores individuos. Ahora lo que haremos será centrarnos en el estudio poblacional de todos los individuos, es decir, de las poblaciones enteras. Veremos de nuevo en porcentajes la colaboración de los procesadores en la formación de los individuos de las poblaciones. Trataremos de ver también la relación entre la convergencia de los individuos al mejor resultado obtenido en su población (punto que se ha estudiado en el apartado 6.4) y la configuración poblacional de que trata este punto.

6.6.1.- Efecto del número de procesadores.

En este caso veremos que el número de procesadores de que dispongamos no influye en la configuración poblacional.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, frecuencia de intercambio 25
- Parámetro variable: 4 y 8 procesadores
- Gráfica:

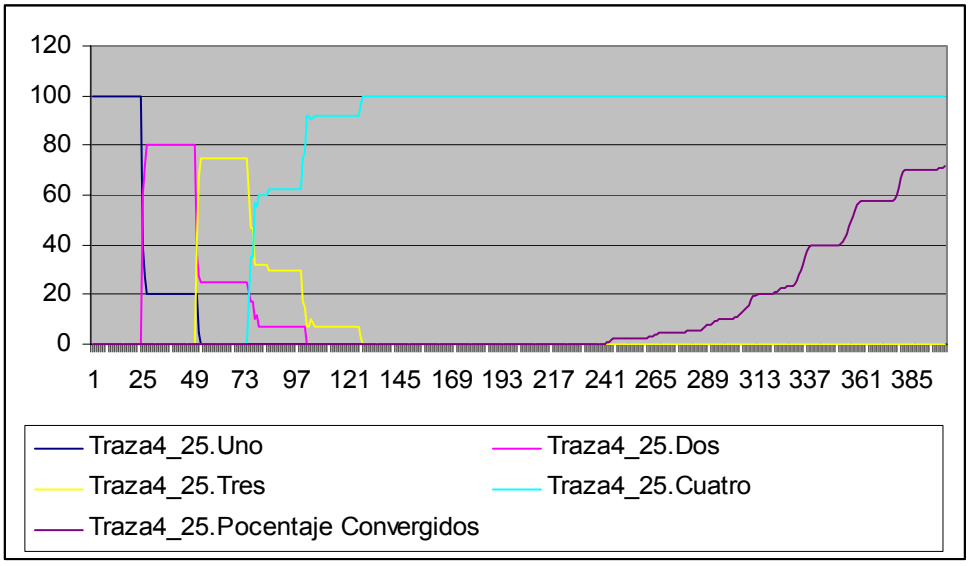

Figura 6.37. Schwefel poblacional completa 4 procesadores

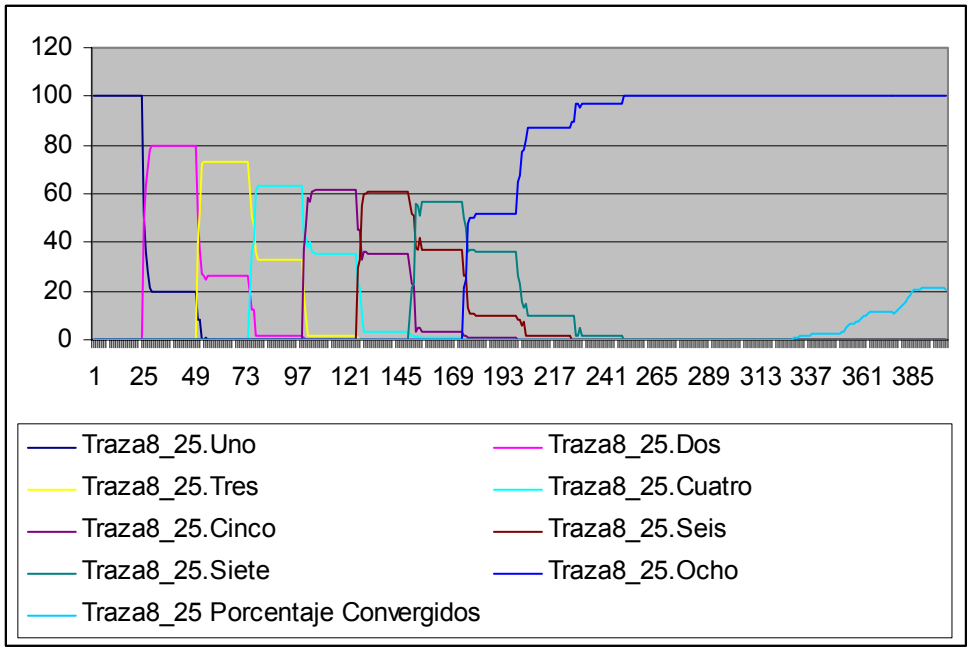

Figura 6.38. Schwefel poblacional completa 8 procesadores

- Observaciones: Al contrario que en el estudio poblacional de los mejores individuos, en el caso del estudio de las poblaciones enteras el número de procesadores podemos ver que es indiferente, obteniéndose dos gráficas prácticamente idénticas para los casos de 4 y 8 procesadores. Únicamente se observa cómo en el segundo caso la curva para 4 procesadores cae debido a que después viene la traza para 5 procesadores. En ambos casos hemos representado una curva con el porcentaje de individuos que han alcanzado el óptimo de sus respectivas poblaciones (esto ya se vio en el punto 6.4). Podemos ver que en el momento de aparecer el primer óptimo ya todos los individuos han pasado por todos los procesadores.

6.6.2- Efecto de la frecuencia de intercambio

Aquí de nuevo la frecuencia de intercambio determinará las diferentes configuraciones de las poblaciones.

- Función: Schwefel
- Parámetro Fijado: Sin mutación, número de procesadores 4
- Parámetro variable: Frecuencia de intercambio 10, 50
- Gráficas:

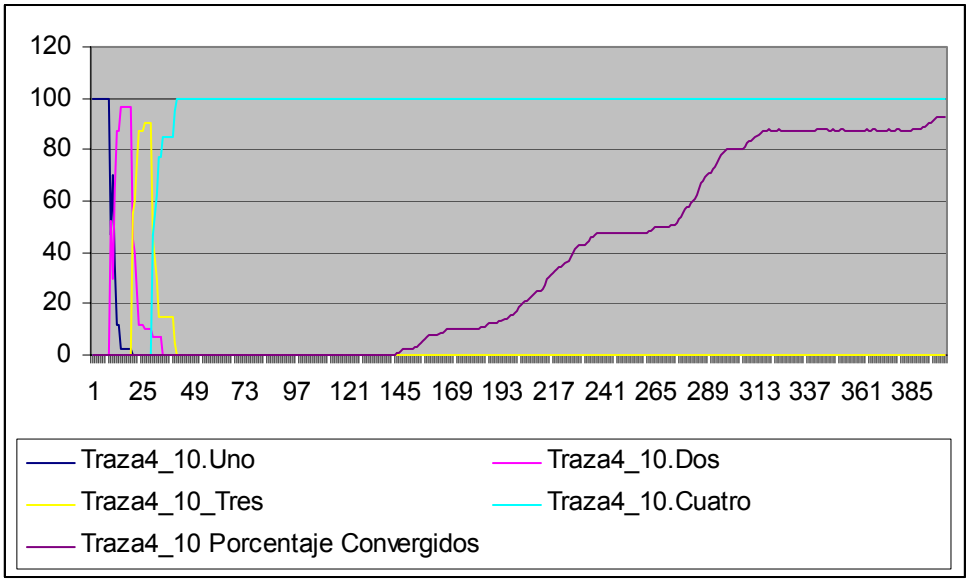

Figura 6.39. Schwefel poblacional completa frecuencia 10

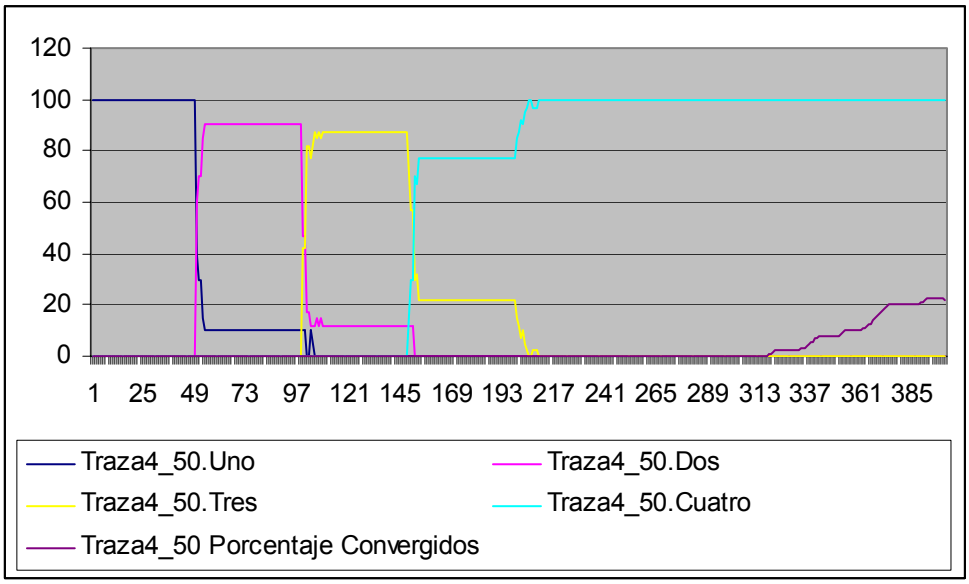

Figura 6.40. Schwefel poblacional completa frecuencia 50

- Observaciones: Observamos que al ser la frecuencia más baja en la segunda que en la primera gráfica, tarda más en propagarse el mejor individuo a las demás poblaciones. De nuevo en ambos casos el mejor necesita pasar por todas las poblaciones para llegar a ser óptimo, viéndose que en la primera configuración casi toda la población llega a converger al óptimo.

6.6.3- Efecto de la mutación.

Estudiaremos de nuevo el efecto de este operador para ver ahora el efecto sobre la configuración global de las poblaciones.

- Función: Rastrigin
- Parámetro Fijado: Frecuencia 50 y Número de procesadores 4
- Parámetro variable: Mutación
- Gráfica:

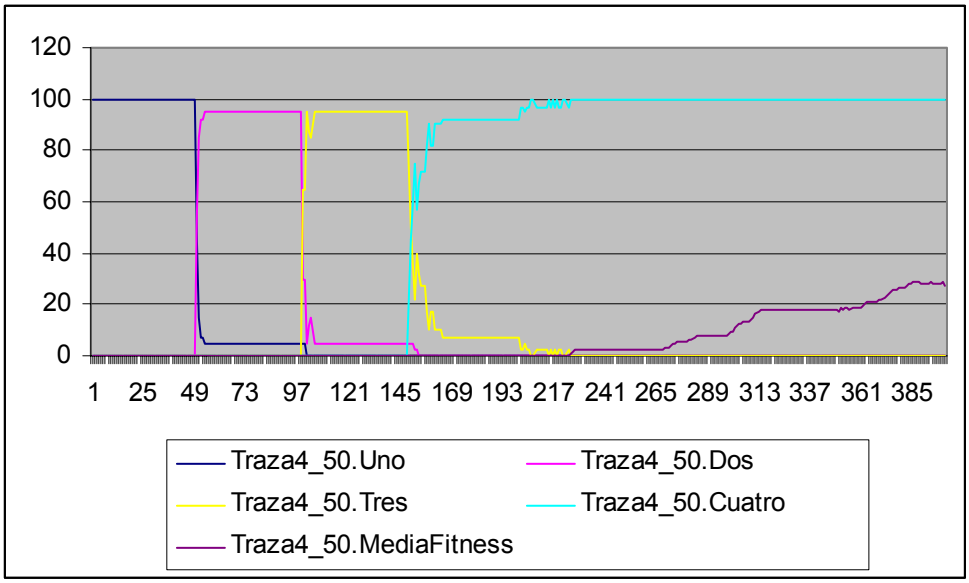

Figura 6.41. Rastrigin poblacional completa sin mutación

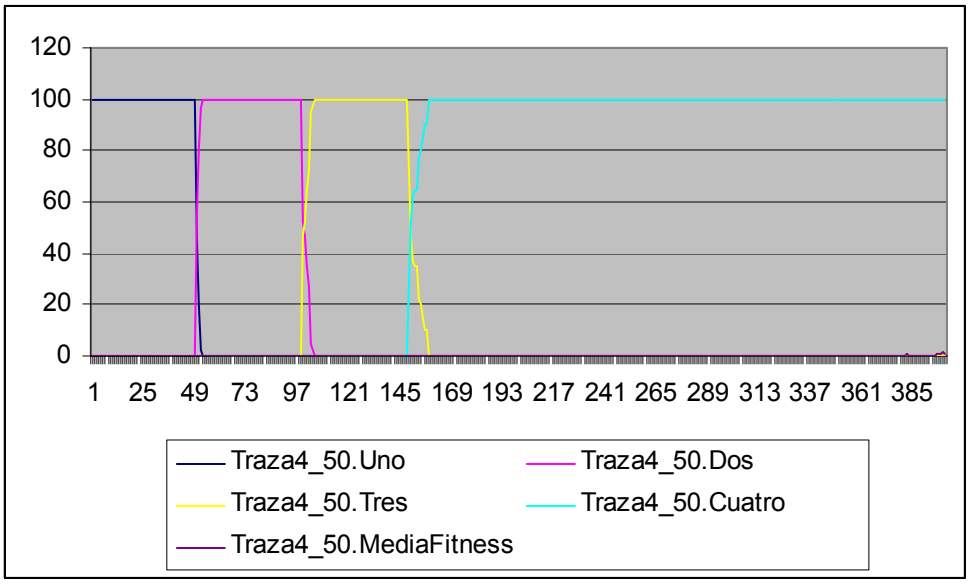

Figura 6.42. Rastrigin poblacional completa con mutación

- Observaciones: En este caso la influencia de la mutación es menor que en el estudio de las configuraciones de los mejores del punto 6.5. De hecho solamente se aprecia que en el caso de usar este operador los cambios entre configuraciones (trazas) son más limpios y se producen más rápidamente.

6.6.4. Conclusiones generales:

De nuevo concluimos que en todos los casos estudiados, para que un individuo alcance el óptimo, es necesario que atraviese todas las poblaciones. En este caso, más que en el anterior, el parámetro más influyente es el de la frecuencia de intercambio, siendo los otros dos (número de procesadores y mutación) apenas significativos.

6.7. Estudio del operador de regeneración.

6.7.1. El operador de regeneración.

Con este nuevo operador, el algoritmo detecta si el valor de fitness del mejor individuo es el mismo durante 10 generaciones consecutivas. En este caso se elimina la cuarta parte de la población, que es sustituida por nuevos individuos cuyos alelos han sido generados al azar. De esta manera conseguiremos introducir la misma aleatoriedad en una población pequeña que la que tendríamos en una población grande.[2]

6.7.2. Análisis del operador

- Función: Schwefel
- Parámetro Fijado: Poblaciones 2, Frecuencia 50, Sin Mutación
- Parámetro variable: Operador de regeneración
- Gráfica:

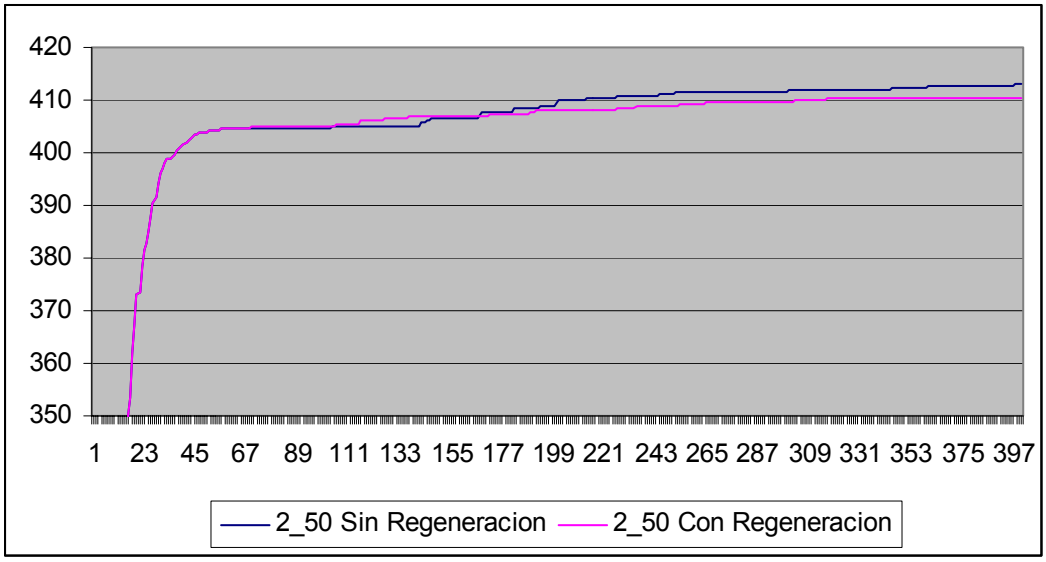

Figura 6.43. Schwefel regeneración 2 poblaciones

- Observaciones: Se observa que el operador en general no beneficia el valor de fitness de los mejores de la configuración. Esto se debe a que los elementos regenerados, con muy alta probabilidad tienen un valor de fitness bajo. Se observa en la gráfica que, puntualmente, alrededor de la generación 120 la configuración con regeneración mejora momentáneamente a la otra. Sin embargo en cuanto esta aplique un intercambio volverá a ser mejor que aquella.

- Función: Schwefel
- Parámetro Fijado: Poblaciones 8, Frecuencia 50, Sin Mutación
- Parámetro variable: Operador de regeneración
- Gráfica:

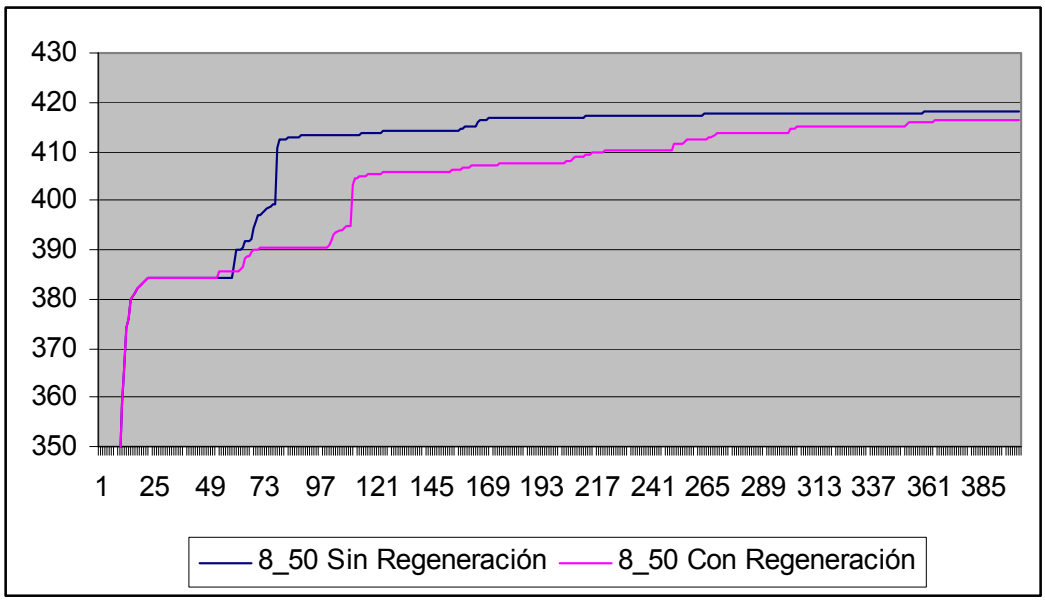

Figura 6.44. Schwefel regeneración 8 poblaciones

- Observaciones: Nuevamente, en esta población más pequeña que la vista en el caso de estudio anterior, se repite el resultado de que la configuración con regeneración da resultados peores que la que no tiene regeneración. También ocurre que durante un breve intervalo de generaciones, alrededor de la 50, la configuración con regeneración supera a la otra, aunque luego se queda muy abajo. Como diferencia se observa que con estas poblaciones más pequeñas, la diferencia de valores de fitness entre las dos configuraciones representadas es mucho mayor que en el caso de menos poblaciones más grandes.

6.7.3. Conclusiones generales.

Este estudio parece indicar que no va a ser muy apropiado el uso del operador de regeneración en funciones como las que hemos utilizado. En los caso que se han visto, este no ayuda a mejorar el valor de fitness de los mejores de las poblaciones y ha hecho bajar el valor de fitness de las medias de toda la población, ya que introduce individuos que con mucha probabilidad tendrán fitness bajo.

Cabe pensar que si podría ser interesante comprobar si, en funciones donde tengamos óptimos parciales de muy bajo valor fitness (comparado con el óptimo global) y en los cuales se quede estancada la población, esta introducción de elementos aleatorios de bajo fitness podría introducir aleatoriedad sin perjudicar tanto los valores de fitness de la población y así ayudar a superar ese estancamiento.

7.- Conclusiones

Después de haber realizado todos los casos experimentales: evolución del fitness, estudios poblacionales, estudio del operador de regeneración, con las distintas configuraciones, hemos obtenido las siguientes conclusiones que se verán a continuación.

La mayoría de los problemas para los que se aplican los algoritmos genéticos son problemas de optimización de un valor. Si nos basáramos únicamente en este criterio tendríamos claramente que tener configuraciones con poblaciones grandes distribuidas en pocos procesadores, que se interrelaciones por medio de intercambios con altas frecuencias. De este modo favoreceremos que los mejores individuos se puedan propagar por el resto de las poblaciones actuando como una única población aún más grande. Esto provoca que las poblaciones que reciben estos "mejores" se beneficien de las buenas características de estos individuos. Se ha comprobado también que el uso de mutación beneficiará la aparición de individuos con alto nivel de fitness gracias a la aleatoriedad que introduce.

Si en un problema que tratamos de resolver nos interesase obtener muchos resultados validos aunque entre estos no se encuentre un óptimo podría interesarnos buscar una configuración en la que obtengamos muchos individuos de las poblaciones con un fitness relativamente elevado. De nuevo la configuración recomendada constaría como en el caso de búsqueda del óptimo de poblaciones grandes y frecuencia de intercambio elevada. Por el contrario no se recomienda el uso de la mutación debido a que el efecto de este operador no permite que toda la población se desarrolle en una misma dirección de fitness. Introduce tanto individuos "buenos" como otros "peores" que los ya existentes con lo que dificultan la convergencia de toda la población a un único valor de fitness.

Del estudio de porcentaje de MVP obtenemos que para evitar estancarse en un óptimo local, deberemos tener mas cuidado al utilizar poblaciones pequeñas con frecuencias de intercambio altas. La mutación nos ayudará a evitar este estancamiento.

En el análisis poblacional, hemos determinado que los mejores individuos, sea cual sea el número de procesadores disponibles, han sido formados al pasar por todas las poblaciones existentes en las configuraciones estudiadas. La frecuencia de intercambio es determinante en cuanto que beneficia la propagación de individuos por los distintos procesadores. La mutación, por el contrario, hace que los individuos foráneos de una población tengan menos probabilidad de saltar a la siguiente, debido a que la aleatoriedad beneficia a que aparezcan individuos nativos mejores que los foráneos que se encuentran en minoría.

Por último se ha estudiado el efecto de operador de regeneración. Como se ha visto en el apartado 6.7, hemos llegado a la conclusión de que con las funciones y configuración con las que se ha desarrollado este trabajo no parece conveniente que sea aplicado.

Apéndice I. Documentación y bibliografía

1. Hidalgo, I., Lanchares, J, Ibarra, A.: Introducción a los Algoritmos Evolutivos. Departamento de Arquitectura de Computadores y Automática de la UCM. (2001)
2. Hidalgo, I, Fernadez de Vega, F.: Distribución equilibrada del esfuerzo de cómputo en AG. Actas del IV Congreso Español sobre Meta heurísticas, Algoritmos Evolutivos y Bioinspirados (2005)
3. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. 3erd edition. Springer-Verlag, Berlin Heidelberg New York (1996).
4. Cantu Paz, E.: Efficient and Accurate Paralell Genetic Algorithms. Kluwer Academic Publishers (2000).
5. Cantu-Paz, E. Goldberg, D.E.: Modeling idelaized bounding cases of parallel genetic algorithms. In Koza, J., et al., eds.: Proceedings of the Second Annual Genetic Programming Conference, Morgan Kaufmann (1997).
6. Goldberg, D.E.: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, MA (1989).
7. Cantu-Paz, E.: A Survey of Paralell Genetic Algorithm. IlliGAL Report No. 97003. University of Illinois at Urbana-Champaign (1997).
8. Davis: Handbook of Genetic Algorithms. Van Nostrand Reinhold (1991).
9. Mitchell,M.: An introduction to genetic algorithms. Cambridge [etc.]: MIT Press, 1997.
10. Araujo Serna, Lourdes: Apuntes de la asignatura Programación Evolutiva. Facultad de Informática UCM.
11. Pacheco, Peter S.: Paralell Programming with MPI. Morgan Kaufmann Publishers, cop. 1997.
12. Martín Llorente, Ignacio: Apuntes de la asignatura Procesamiento Paralelo. Facultad de Informática UCM.
13. MPI: http://www.mpi-forum.org/
14. PGAPack: http://www.mcs.anl.gov/pgapack.html
15. Multimodal function: http://www.geatbx.com/docu/fcnindex-01.html
16. Kernighan, Brian W., Ritchie, Dennis M.: The C Programming Language. Prentice Hall, cop. 1988.

Apéndice II. Índice de figuras