Estudio de la tolerancia a fallos en Algoritmos Genéticos Paralelos

Sistemas Informáticos
Curso 2005-2006

Dirigido por D. José Ignacio Hidalgo Pérez

Amelia Barroso Cuarental
David Boíllos Fernández
Ana Belén Jerónimo Pérez

Indice

1. Introducción

Algoritmos genéticos paralelos tolerantes a fallos.

El proyecto desarrollado se basa en un estudio sobre los algoritmos genéticos en el procesamiento paralelo.

Cuando varios procesadores trabajan simultáneamente ejecutando un algoritmo genético, intercambian las soluciones que van obteniendo en cada generación.

Si se da el caso de que uno de ellos falla, las soluciones a las que ha llegado hasta ese momento, son desechadas.

El objetivo del proyecto es averiguar qué es más eficaz: ¿deberíamos desechar los resultados del procesador fallido o, por el contrario, debemos continuar la ejecución teniendo en cuenta los resultados obtenidos por este procesador hasta el momento de la caída?

En el experimento, vamos a realizar todas las pruebas sobre un sistema Unix, haciendo uso de las librerías de PGAPack.

Para el paso de mensajes entre procesadores, utilizaremos las sentencias que nos proporciona MPI.

Las pruebas se están realizando sobre una máquina multiprocesador ubicada en los laboratorios de informática, que cuenta con 8 procesadores, entre los que se repartirán las diferentes ejecuciones de los algoritmos.

Realizaremos múltiples pruebas con varios algoritmos, haciendo fallar algún procesador para comprobar si se pueden aprovechar las generaciones obtenidas hasta el momento por el mismo.

Para llevar a cabo el experimento disponemos de diversos algoritmos genéticos: One-Max, F-Trap, Fmodal, Rastrigin, Schwefel y Rosenbrock, implementados en C.

Es importante la decisión de la codificación de los individuos para la ejecución de algoritmos genéticos:

Se trata de individuos con alelos binarios, cuya longitud dependerá del experimento lanzado en cada ejecución.

Para poder seguir el rastro de las generaciones, viendo por qué procesador van pasando, y quién las ha generado, añadiremos información extra en cada individuo, para un mejor seguimiento de los mismos.

En el lanzamiento de las pruebas de ejecución, podremos elegir todas las posibilidades que nos ofrece el PGA-Pack, tales como:

- Tipo de cruce
- Tipo de fitness
- Tipo de mutación
- Tipo de reemplazamiento
- Tipo de selección
- Longitud del individuo
- Tamaño de la población
- Número de iteraciones
- Cada cuántas generaciones intercambiamos
- Cuántos individuos intercambiamos cada vez

Todas estas características se explicarán más adelante.

1.1. Organización del trabajo:

En resto de documento trataremos los siguientes puntos:

1. **Algoritmos genéticos**. Definición y explicación de los mismos. Diferentes opciones a considerar para su codificación.
2. **Algoritmos paralelos.** Qué es la programación paralela. Técnicas de procesamiento paralelo utilizadas en el proyecto actual.
3. **Algoritmos genéticos paralelos**. Consideraciones a tener en cuenta para paralelizar el código para su ejecución en diferentes procesadores.
4. **PGA-Pack.** Funcionamiento. Sus principales opciones configurables y modificaciones realizadas para su uso.
5. **Funciones de prueba.** Explicación de las funciones implementadas para realizar los experimentos.
6. **Resultados experimentales.** Explicación de los resultados obtenidos.
7. **Conclusiones obtenidas.** Gráficas explicativas
8. **Futuras líneas de trabajo.**
9. **Referencias.**
10. **Bibliografía**
11. **Apéndices.**
    - Código en C.
    - Explicación de los métodos utilizados para sacar conclusiones (realización de gráficas de los datos obenidos mediante macros en Microsoft Excel).
    - Gráficas de los experimentos realizados.

2. Algoritmos genéticos

2.1. Definición

Un algoritmo genético es una técnica de programación que imita a la evolución biológica como estrategia para búsqueda y optimización en la resolución problemas.

Dado un problema específico a resolver, la entrada del algoritmo es un conjunto de soluciones potenciales a ese problema, codificadas de alguna manera, y una métrica llamada función de aptitud (o función de coste, o fitness) que permite evaluar cuantitativamente a cada candidata. Estas candidatas pueden ser soluciones que ya se sabe que funcionan, con el objetivo de que el Algoritmo Genético las mejore, pero se suelen generar aleatoriamente.

Luego el Algoritmo Genético evalúa cada candidata de acuerdo con la función de aptitud. En un acervo de candidatas generadas aleatoriamente, por supuesto, la mayoría no funcionarán en absoluto, y serán eliminadas. Sin embargo, por puro azar, unas pocas pueden ser prometedoras -pueden mostrar actividad, aunque sólo sea actividad débil e imperfecta, hacia la solución del problema.

Estas candidatas prometedoras se conservan y se les permite reproducirse. Se realizan múltiples copias de ellas, pero las copias no son perfectas; se introducen cambios aleatorios durante el proceso de copia. Luego, esta descendencia digital prosigue con la siguiente generación, formando un nuevo acervo de soluciones candidatas, y son sometidas a una ronda de evaluación de aptitud. Las candidatas que han empeorado o no han mejorado con los cambios en su código son eliminadas de nuevo; pero, de nuevo, por puro azar, las variaciones aleatorias introducidas en la población pueden haber mejorado a algunos individuos, convirtiéndolos en mejores soluciones del problema, más completas o más eficientes. De nuevo, se seleccionan y copian estos individuos vencedores hacia la siguiente generación con cambios aleatorios, y el proceso se repite. Las expectativas son que la aptitud media de la población se incrementará en cada ronda y, por tanto, repitiendo este proceso cientos o miles de rondas, pueden descubrirse soluciones muy buenas del problema.

Los algoritmos genéticos son métodos de optimización, que tratan de resolver el mismo conjunto de problemas que se ha contemplado anteriormente, es decir, hallar (xi,...,xn) tales que F(xi,...,xn) sea máximo. En un algoritmo genético, tras parametrizar el problema en una serie de variables, (xi,...,xn) se codifican en un cromosoma. Todos los operadores utilizados por un algoritmo genético se aplicarán sobre estos cromosomas, o sobre poblaciones de ellos. En el algoritmo genético va implícito el método para resolver el problema; son solo parámetros de tal método los que están codificados, a diferencia de otros algoritmos evolutivos como la programación genética. Hay que tener en cuenta que un algoritmo genético es independiente del problema, lo cual lo hace un algoritmo robusto, por ser útil para cualquier problema, pero a la vez débil, pues no está especializado en ninguno.

Las soluciones codificadas en un cromosoma compiten para ver cuál constituye la mejor solución (aunque no necesariamente la mejor de todas las soluciones posibles). El

ambiente, constituido por las otras camaradas soluciones, ejercerá una presión selectiva sobre la población, de forma que sólo los mejor adaptados (aquellos que resuelvan mejor el problema) sobrevivan o leguen su material genético a las siguientes generaciones, igual que en la evolución de las especies. La diversidad genética se introduce mediante mutaciones y reproducción sexual.

En la Naturaleza lo único que hay que optimizar es la supervivencia, y eso significa a su vez maximizar diversos factores y minimizar otros. Un algoritmo genético, sin embargo, se usará habitualmente para optimizar sólo una función, no diversas funciones relacionadas entre sí simultáneamente. La optimización que busca diferentes objetivos simultáneamente, denominada multimodal o multiobjetivo, también se suele abordar con un algoritmo genético especializado.

Por lo tanto, un algoritmo genético consiste en lo siguiente: hallar de qué parámetros depende el problema, codificarlos en un cromosoma, y se aplican los métodos de la evolución: selección y reproducción sexual con intercambio de información y alteraciones que generan diversidad. Más adelante veremos este intercambio de información (también llamado operador de cruce), y las alteraciones (llamadas mutaciones).

Para comenzar la competición, se generan aleatoriamente una serie de cromosomas. El algoritmo genético procede de la forma siguiente:

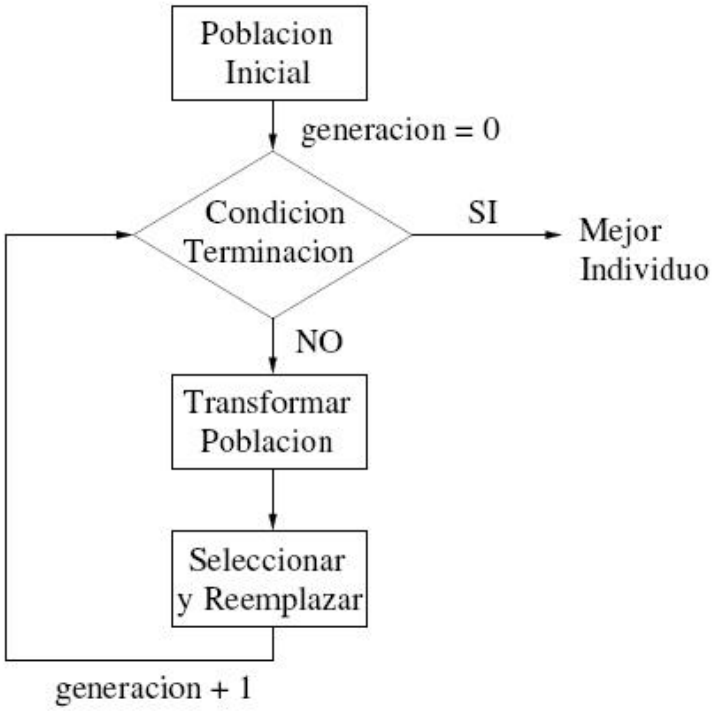
- Evaluar la puntuación (fitness) de cada uno de los genes.
- Permitir a cada uno de los individuos reproducirse, de acuerdo con su puntuación.
- Emparejar los individuos de la nueva población, haciendo que intercambien material genético, y que alguno de los bits de un gen se vea alterado debido a una mutación espontánea.

Cada uno de los pasos consiste en una actuación sobre las cadenas de bits, es decir, la aplicación de un operador a una cadena binaria. Se les denominan, por razones obvias, operadores genéticos, y hay tres principales: selección, crossover (o recombinación, u operador de cruce) y mutación; aparte de otros operadores genéticos no tan comunes, todos ellos se verán a continuación.

Un algoritmo genético tiene también una serie de parámetros que se tienen que fijar para cada ejecución, como los siguientes:

- Tamaño de la población: debe de ser suficiente para garantizar la diversidad de las soluciones, y, además, tiene que crecer más o menos con el número de bits del cromosoma, aunque nadie ha aclarado cómo tiene que hacerlo. Por supuesto, depende también del ordenador en el que se esté ejecutando.
- Condición de terminación: lo más habitual es que la condición de terminación sea la convergencia del algoritmo genético o un número prefijado de generaciones.

Veamos un diagrama que explica la estructura de los algoritmos genéticos:



2.2. Representación

Antes de que un algoritmo genético pueda ponerse a trabajar en un problema, se necesita un método para codificar las soluciones potenciales del problema en cromosomas, de forma que una computadora pueda procesarlas.

Cada cromosoma (individuo) tiene varios genes (alelos), que corresponden a sendos parámetros del problema. Para poder trabajar con estos genes en el ordenador, es necesario codificarlos en una cadena.

Un enfoque común es codificar los cromosomas como cadenas binarias: secuencias de 1s y 0s, donde el dígito de cada posición representa el valor de algún aspecto de la solución. Otro método similar consiste en codificar las soluciones como cadenas de enteros o números decimales, donde cada posición, de nuevo, representa algún aspecto particular de la solución. Este método permite una mayor precisión y complejidad que el método comparativamente restringido de utilizar sólo números binarios, y a menudo "está intuitivamente más cerca del espacio de problemas" .

Un tercer método consiste en representar a los individuos de un Algoritmo Genético como cadenas de letras, donde cada letra, de nuevo, representa un aspecto específico de la solución. Un ejemplo de esta técnica es el método basado en "codificación gramática" de Hiroaki Kitano, en el que a un AG se le encargó la tarea de evolucionar un sencillo conjunto de reglas llamadas gramática libre de contexto, que a su vez se utilizaban para generar redes neuronales para una variedad de problemas.

La virtud de estos tres métodos es que facilitan la definición de operadores que causen los cambios aleatorios en las candidatas seleccionadas: cambiar un 0 por un 1 o viceversa, sumar o restar al valor de un número una cantidad elegida al azar, o cambiar una letra por otra. Otra estrategia, desarrollada principalmente por John Koza, de la Universidad de Stanford, y denominada programación genética, representa a los programas como estructuras de datos ramificadas llamadas árboles. En este método, los cambios aleatorios pueden generarse cambiando el operador o alterando el valor de un cierto nodo del árbol, o sustituyendo un subárbol por otro.

La principal diferencia entre los llamados algoritmos evolutivos y los algoritmos genéticos es que en los primeros los individuos pueden ser cualquier estructura de datos, mientras que en los segundos trabajan con poblaciones de cadenas binarias.

Es importante señalar que los algoritmos evolutivos no necesitan representar las soluciones candidatas como cadenas de datos de una longitud fija. Algunos las representan de esta manera, pero otros no; por ejemplo, la "codificación gramatical" de Kitano, explicada arriba, puede escalarse eficientemente para crear redes neuronales grandes y complejas, y los árboles de programación genética de Koza pueden crecer arbitrariamente tanto como sea necesario para resolver cualquier problema que se les pida. Más adelante, en este mismo punto, se tratará con detalle los individuos de longitud variable.

EJEMPLO 1:

Si un atributo (tiempo) puede tomar tres valores posibles (despejado, nublado, lluvioso) una manera de representarlo es mediante tres bits de forma que:

(Tiempo = Nublado ó Lluvioso) y (Viento = Fuerte) se representaría con la siguiente cadena:

0 1 1 1 0

De esta forma podemos representar fácilmente conjunciones de varios atributos para expresar restricciones (precondiciones) mediante la concatenación de dichas cadenas de bits.

Además si tenemos otro atributo "Viento" que puede ser Fuerte o Moderado, se representaría con la siguiente cadena:

0 1 1 1 0

Las postcondiciones de las reglas se pueden representar de la misma forma. Por ello una regla se puede describir como la concatenación de la precondición y la postcondición.

La mayoría de las veces, una codificación correcta es la clave de una buena resolución del problema. Generalmente, la regla heurística que se utiliza es la llamada regla de los bloques de construcción, es decir, parámetros relacionados entre sí deben de estar cercanos en el cromosoma. De esta forma, se codifican mediante grupos de bits o bytes sucesivos en el cromosoma.

En todo caso, se puede ser bastante creativo con la codificación del problema, teniendo siempre en cuenta la regla anterior. Esto puede llevar a usar cromosomas bidimensionales, o tridimensionales, o con relaciones entre genes que no sean puramente lineales de vecindad. En algunos casos, cuando no se conoce de antemano el número de variables del problema, caben dos opciones: codificar también el número de variables, fijando un número máximo, o bien, lo cual es mucho más natural, crear un cromosoma que pueda variar de longitud. Para ello, claro está, se necesitan operadores genéticos que alteren la longitud.

Normalmente, la codificación es estática, pero en casos de optimización numérica, el número de bits dedicados a codificar un parámetro puede variar, o incluso lo que representen los bits dedicados a codificar cada parámetro. Algunos paquetes de algoritmos genéticos adaptan automáticamente la codificación según van convergiendo los bits menos significativos de una solución.

Las características de una representación perfecta respecto a los objetos respresentados son:

1. Completitud: Todos los objetos deben poder ser representados.
2. Coherencia: Únicamente debe representar objetos del problema.
3. Uniformidad: Todos los objetos deben estar representados por la misma cantidad de codificaciones.
4. Sencillez: Debe ser fácil de aplicar el mecanismo de codificación objeto $\leftrightarrow$individuo.
5. Localidad: Pequeños cambios en los individuos se han de corresponder con pequeños cambios en los objetos.

2.2.1. Cromosomas de longitud variable

Hasta ahora se han descrito cromosomas de longitud fija, donde se conoce de antemano el número de parámetros de un problema. Pero hay problemas en los que esto no sucede. Por ejemplo, en un problema de clasificación, donde dado un vector de entrada, queremos agruparlo en una serie de clases, podemos no saber siguiera cuantas clases hay. O en diseño de redes neuronales, puede que no se sepa (de hecho, nunca se sabe) cuántas neuronas se van a necesitar. Por ejemplo, en un perceptrón hay reglas que dicen cuantas neuronas se deben de utilizar en la capa oculta; pero en un problema determinado puede que no haya ninguna regla heurística aplicable; tendremos que utilizar los algoritmos genéticos para hallar el número óptimo de neuronas.

En estos casos, necesitamos dos operadores más: añadir y eliminar. Estos operadores se utilizan para añadir un gen, o eliminar un gen del cromosoma. La forma más habitual de añadir un locus es duplicar uno ya existente, el cual sufre mutación y se añade al lado del anterior. En este caso, los operadores del algoritmo genético simple (selección, mutación, cruce) funcionarán de la forma habitual, salvo, claro está, que sólo se hará cruce en la zona del cromosoma de menor longitud.

Estos operadores permiten, además, crear un algoritmo genético de dos niveles: a nivel de cromosoma y a nivel de gen. Supongamos que, en un problema de

clasificación, hay un gen por clase. Se puede asignar una puntuación a cada gen en función del número de muestras que haya clasificado correctamente. Al aplicar estos operadores, se duplicarán los alelos con mayor puntuación, y se eliminarán aquellos que hayan obtenido menor puntuación, o cuya puntuación sea nula.

Por ejemplo, en un problema de clasificación en el que hay que clasificar los puntos del cuadrado [0,10]x[0,10] en dos clases, 1 y 2, que no son linealmente separables. Inicialmente se desconoce cuantos vectores son necesarios para clasificar estas clases. El algoritmo genético es capaz de hallar un número óptimo de vectores, a cada uno de los cuales se asigna una etiqueta de clase, tales que el error se hace mínimo, en este caso 4 vectores para la primera clase y 5 para la 2ª. Cada cromosoma estará compuesto por un diccionario o conjunto de vectores, cada uno de los cuales tiene asignada una etiqueta de clase.

2.2.2. Operadores de nicho

Otros operadores importantes son los operadores de nicho. Estos operadores están encaminados a mantener la diversidad genética de la población, de forma que cromosomas similares sustituyan sólo a cromosomas similares, y son especialmente útiles en problemas con muchas soluciones; un algoritmo genético con estos operadores es capaz de hallar todos los máximos, dedicándose cada especie a un máximo. Más que operadores genéticos, son formas de enfocar la selección y la evaluación de la población.

Una de las formas de llevar esto a es la introducción del crowding (apiñamiento). Otra forma es introducir una función de compartición o sharing, que indica cómo es de similar un cromosoma al resto de la población. La puntuación de cada individuo se dividirá por esta función de compartición, de forma que se facilita la diversidad genética y la aparición de individuos diferentes.

También se pueden restringir los emparejamientos, por ejemplo, a aquellos cromosomas que sean similares. Para evitar las malas consecuencias del inbreeding (cruce entre dos descendientes del mismo padre) que suele aparecer en poblaciones pequeñas, estos periodos se intercalan con otros periodos en los cuales el emparejamiento es libre. Las consecuencias del inbreeding se tratarán un poco más abajo, en el punto del cruce.

2.2.3. Operadores genéticos: selección, cruce, inversión y mutación.

Ya que estamos hablando continuamente de estos operadores, pasaremos a explicar detenidamente en qué consta cada uno de ellos:

Inversión

La inversión es un operador de reordenación que se basa en la genética natural en la que el valor de un gen es independiente de su posición en el cromosoma de manera

que al invertir el cromosoma se guarda mucha de la información del cromosoma original.

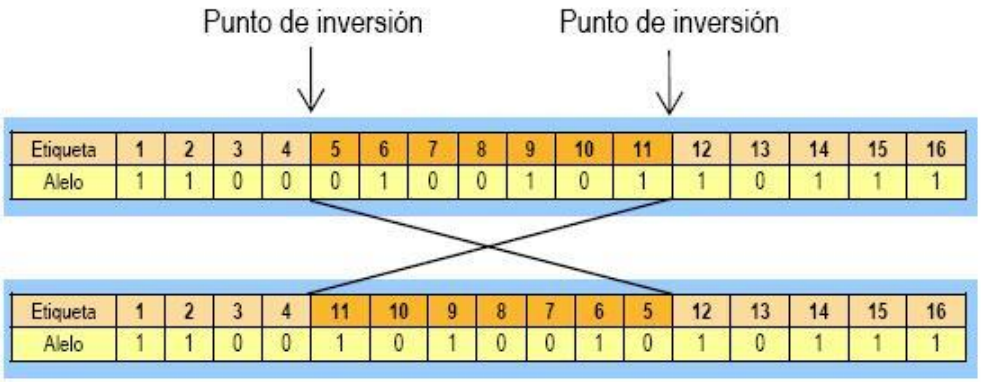
No es un operador que caracterice de forma marcada los algoritmos genéticos, como son el cruce y la mutación. Su uso se suele dar en algoritmos que emplean codificaciones binarias para escapar de situaciones en las que se ha bloqueado el mecanismo de generación de bloques constructivos.

La ordenación seleccionada para codificar soluciones a un problema (el orden de los componentes que codifican los genes en el cromosoma), puede no resultar conveniente e impedir que se logre crear bloques constructivos adecuados, haciendo que el algoritmo sea incapaz de trasladar la búsqueda a las zonas más aptas del espacio de soluciones. Estas situaciones suelen desembocar en la imposibilidad de converger y en la desorientación.

Este inconveniente puede ser resuelto introduciendo en el proceso de optimización, no sólo el fenotipo de los individuos (lo que representan las estructuras), sino también el genotipo (las propias estructuras), es decir: buscar el orden de los componentes de la cadena que permita construir bloques constructivos de alto significado y con mayor potencial evolutivo.

Uno de los mecanismos que permite hacer evolucionar las estructuras de codificación es la inversión. Este operador actúa seleccionando aleatoriamente dos puntos de la cadena e invirtiendo el orden de los elementos existentes entre ambos. Se hace necesario que el algoritmo recuerde el significado original de cada uno de los elementos de la cadena durante los diferentes cambios de orden; esto puede hacerse mediante el etiquetado de los elementos. Cabe destacar que el proceso de inversión no modifica al individuo que representa el cromosoma, sino la forma en que dicho individuo es codificado. Por esta razón, no se modifica tampoco el valor de fitness (valor de la función de coste) del cromosoma.

El operador inversión puede producir problemas a la hora de realizar el cruce.



Cruce

También llamado crossover, entrecruzamiento o recombinación.

Consiste en el intercambio de material genético entre dos cromosomas (a veces más, como el operador orgía propuesto por Eiben et al.). El cruce es el principal operador genético, hasta el punto que se puede decir que no es un algoritmo genético si no tiene cruce, y, sin embargo, puede serlo perfectamente sin mutación, según descubrió Holland. El teorema de los esquemas confía en él para hallar la mejor solución a un problema, combinando soluciones parciales.

Para aplicar el cruce, se escogen aleatoriamente dos miembros de la población. No pasa nada si se emparejan dos descendientes de los mismos padres; ello garantiza la perpetuación de un individuo con buena puntuación (y, además, algo parecido ocurre en la realidad; es una práctica utilizada, por ejemplo, en la cría de ganado, llamada inbreeding, y destinada a potenciar ciertas características frente a otras). Sin embargo, si esto sucede demasiado a menudo, puede crear problemas: toda la población puede aparecer dominada por los descendientes de algún gen, que, además, puede tener caracteres no deseados. Esto se suele denominar en otros métodos de optimización atranque en un mínimo local, y es uno de los principales problemas con los que se enfrentan los que aplican algoritmos genéticos.

En cuanto al teorema de los esquemas, se basa en la noción de bloques de construcción. Una buena solución a un problema está constituida por unos buenos bloques, igual que una buena máquina está hecha por buenas piezas. El cruce es el encargado de mezclar bloques buenos que se encuentren en los diversos progenitores, y que serán los que den a los mismos una buena puntuación. La presión selectiva se encarga de que sólo los buenos bloques se perpetúen, y poco a poco vayan formando una buena solución. El teorema de los esquemas viene a decir que la cantidad de buenos bloques se va incrementando con el tiempo de ejecución de un algoritmo genético, y es el resultado teórico más importante en algoritmos genéticos.

El intercambio genético se puede llevar a cabo de muchas formas, pero hay dos grupos principales

Cruce n-puntos:

Los dos cromosomas se cortan por n puntos, y el material genético situado entre ellos se intercambia. Lo más habitual es un cruce de un punto o de dos puntos. Veamos un ejemplo, donde aparecen señalados a partir de qué alelo se ha producido el cruce:

Padre

0 0 0 1 0 1 0 1 0 1 0 1 0 1

Madre

1 0 1 1 1 0 0 1 1 1 0 1 1 1

Hijo

0 0 0 1 0 0 0 1 1 1 0 1 1 1

Cruce uniforme:

Se genera un patrón aleatorio de 1s y 0s, y se intercambian los bits de los dos cromosomas que coincidan donde hay un 1 en el patrón. O bien se genera un número aleatorio para cada bit, y si supera una determinada probabilidad se intercambia ese bit entre los dos cromosomas. En el siguiente ejemplo aparecen señalados los bits que se han cruzado, tomando como referencia el individuo correspondiente al padre:

Padre

0 0 0 1 0 1 0 1 0 1 0 1 0 1

Madre

1 0 1 1 1 0 0 1 1 1 0 1 1 1

Hijo

0 0 1 1 0 0 0 1 1 1 0 1 1 1

Cruces especializados:

En algunos problemas, aplicar aleatoriamente el cruce da lugar a cromosomas que codifican soluciones inválidas; en este caso hay que aplicar el cruce de forma que genere siempre soluciones válidas. Un ejemplo de estos son los operadores de cruce usados en el problema del viajante.

Este problema consiste en un viajante de comercio que ha de visitar n ciudades o puntos de venta: x_1, x_2,…, x_n, siendo conocidas las n(n-1) distancias entre los puntos (d_{ij} ≡ distancia desde la ciudad x_i a la x_j; i, j =1, 2, …, n). El objetivo es pasar una sola vez por todas y cada una de ellas (volviendo a la ciudad de partida) y su problema es el de saber en qué orden debe hacerlo a fin de que la distancia total recorrida sea mínima.

Cruce segmentado:

Se trata de una variante del cruce multipunto (cruce de n-puntos), en el que el número de puntos es variable**.** En lugar de definir un valor concreto se define una probabilidad de que al llegar a cierto elemento de la cadena durante la reproducción, se introduzca o no un punto de cruce. A dicha probabilidad se le denomina "probabilidad de segmentación". Por ejemplo, un valor de la probabilidad de segmentación de 0,15 indica que en promedio, el número de puntos de cruce en cada proceso reproductivo será 0,15*long , siendo "long" la longitud de las cadenas, aunque no necesariamente tomará dicho valor.

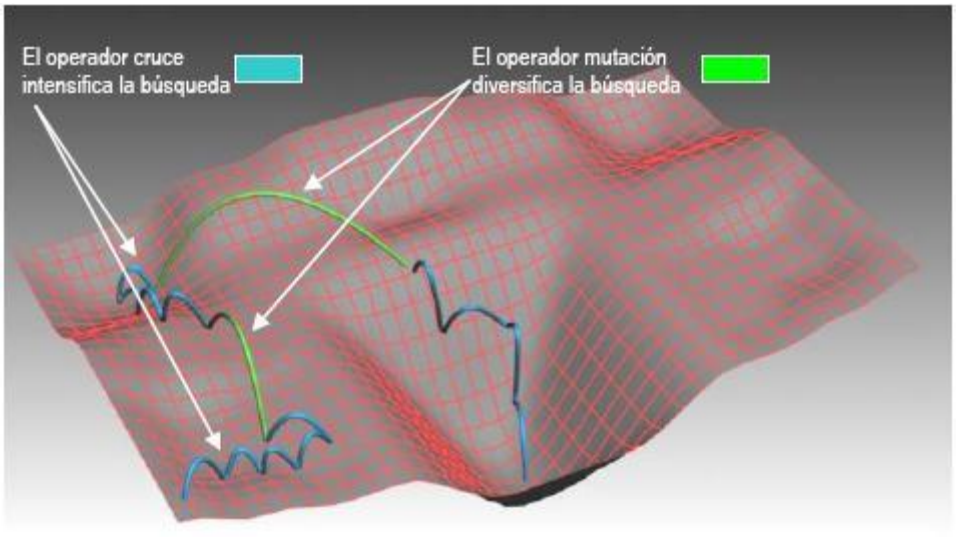
Mutación

En la Evolución, una mutación es un suceso bastante poco común (sucede aproximadamente una de cada mil replicaciones), como ya se ha visto anteriormente. En la mayoría de los casos las mutaciones son letales, pero en promedio, contribuyen a la diversidad genética de la especie. En un algoritmo genético tendrán el mismo papel, y la misma frecuencia (es decir, muy baja).

Una vez establecida la frecuencia de mutación, por ejemplo, uno por mil, se examina cada bit de cada cadena cuando se vaya a crear el nuevo individuo a partir de sus padres (normalmente se hace de forma simultánea al cruce). Si un número generado aleatoriamente está por debajo de esa probabilidad, se cambiará el bit (es decir, de 0 a 1 o de 1 a 0). Si no, se dejará como está. Dependiendo del número de individuos que haya y del número de bits por individuo, puede resultar que las mutaciones sean extremadamente raras en una sola generación.

No hace falta decir que no conviene abusar de la mutación. Es cierto que es un mecanismo generador de diversidad, y, por tanto, la solución cuando un algoritmo genético está estancado, pero también es cierto que reduce el algoritmo genético a una búsqueda aleatoria. Siempre es más conveniente usar otros mecanismos de generación de diversidad, como aumentar el tamaño de la población, o garantizar la aleatoriedad de la población inicial.

Los algoritmos genéticos son capaces de explorar y explotar el espacio de soluciones. Por explotar se entiende realizar una búsqueda exhaustiva en una zona restringida del espacio. Esta tarea se realiza mediante el operador cruce, que actúa como un intensificador de la búsqueda en la zona del espacio de soluciones actualmente ocupada por la población. El efecto exploratorio, por el que se prospectan zonas del espacio alejadas de las soluciones presentes en la población, es asumido por el operador mutación. Como se representa a continuación, mientras la población converge hacia un óptimo local, esa zona del espacio de soluciones es estudiada, posteriormente, el operador mutación hace posible el salto hacia otros lugares inexplorados.



Selección

Antes de hablar de la selección, deberemos explicar la función de evaluación de individuos o función de coste:

2.2.4. Función de coste

El otro aspecto fundamental en el desarrollo de un AG es la elección de una buena función de coste. La función de coste debe evaluar a los individuos para indicar cuál es la calidad de la solución que representan y poder realizar el proceso de selección. Un ejemplo típico es el siguiente. Supongamos que tenemos una cadena de 6 bits en la que se quiere maximizar el número de unos que contiene la cadena (algoritmo denominado "One Max", el cual se va a utilizar para nuestro proyecto). En este caso la función de coste sería f1(x) = x, donde x es el número de 1s que contiene la cadena (individuo) evaluada. Así el individuo [0 0 0 0 1 1] tendría asociado un valor de la función de coste igual a 2 y el individuo [0 1 1 0 1 1] un valor de 4.

Normalmente los AGs tratan de maximizar una función, pero si lo que se quiere es minimizar, no hay más que utilizar la inversa de la función objetivo, o bien multiplicarla por -1. Para el mismo ejemplo si se quiere minimizar el número de 1s en la cadena de caracteres, se pueden utilizar las funciones f2(x) =-x ó f3(x) = 1/x. Otra opción es cambiar el significado de x para que represente el número de ceros que contiene cada cromosoma y utilizar la función f1(x).

Veamos un ejemplo un poco más complejo. Suponiendo que tenemos 8 empleados que queremos destinar a diferentes ciudades, imaginemos que el desplazamiento de cada una de las personas tiene un coste para la empresa dependiendo a la ciudad a la que vayan (ver tabla 1).

Empleado	Coste Bilbao	Coste Málaga	Coste Granada	Coste Santander
1	22	13	14	17
2	8	9	10	26
3	7	35	46	12
4	53	28	96	15
5	61	17	38	17
6	42	32	33	15
7	21	12	21	12
8	41	29	27	35

(tabla 1)

Si se quiere minimizar el coste de la nueva distribución no habría más que utilizar una función de coste de la forma:

$$f(x) = \Sigma^{8}_{i=1} f_i(x)$$

donde fi(x) es el coste de cada empleado para la distribución evaluada. Una de las distribuciones óptimas sería la representada por el individuo:

2 1 1 4 4 4 2 3

donde cada uno de los empleados podría tomar un valor de 1 a 4, representando la ciudad a la que sería destinado, y el individuo resultante representaría para cada empleado la ciudad con menos coste a la que podría ser destinado. Como podemos ver, no se han impuesto restricciones sobre el número de empleados máximo o mínimo que debe ir a cada ciudad.

En ocasiones los AGs tienen aparte de una función de coste una función objetivo. Es decir, que aunque la evaluación se realiza de acuerdo a una función se trata de alcanzar un objetivo que se evalúa mediante otra función distinta.

Esto suele ocurrir cuando se tratan problemas con restricciones o problemas en los que se pretende optimizar distintos parámetros conocidos cómo problemas multi-objetivo. Esta función objetivo no tiene porque ser una función numérica si no simplemente que se cumpla un criterio o no.

2.2.5. Evaluación y Selección

Hay muchas formas de realizar la selección, pero siempre se debe asegurar que los mejores individuos tengan una mayor probabilidad de ser seleccionados. Una de las características más importantes de los algoritmos genéticos es que dejan un camino abierto a aquellas soluciones que no pertenecen a las mejores, para que puedan aportar parte de su información a la nueva generación.

Durante la evaluación, se decodifica el gen, convirtiéndose en una serie de parámetros de un problema, se halla la solución del problema a partir de esos parámetros, y se le da una puntuación a esa solución en función de lo cerca que esté de la mejor solución. A esta puntuación se le llama fitness. Se trata de una función heurística como las utilizadas en los algoritmos de búsqueda de Inteligencia Artificial, ya que no olvidemos que los algoritmos genéticos son una parte de esta área.

Por ejemplo, supongamos que queremos hallar el máximo de la función, una parábola invertida con el máximo en x=1. En este caso, el único parámetro del problema es la variable x. La optimización consiste en hallar un x tal que F(x) sea máximo. Crearemos, pues, una población de cromosomas, cada uno de los cuales contiene una codificación binaria del parámetro x. Lo haremos de la forma siguiente: cada byte, cuyo valor está comprendido entre 0 y 255, se transformará para ajustarse al intervalo [-1,1], donde queremos hallar el máximo de la función (ver tabla)

Valor binario	Decodificación	Evaluación f(x)
10010100	21	0,9559
10010001	19	0,9639
101001	-86	0,2604
1000101	-58	0,6636

El fitness determina siempre los cromosomas que se van a reproducir, y aquellos que se van a eliminar, pero hay varias formas de considerarlo para seleccionar la población de la siguiente generación:

- Usar el orden, o rango, y hacer depender la probabilidad de permanencia o evaluación de la posición en el orden.

- Aplicar una operación al fitness para escalarlo; como por ejemplo, el escalado sigma. En este esquema el fitness se escala

- En algunos casos, el fitness no es una sola cantidad, sino diversos números, que tienen diferente consideración. Basta con que tal fitness forme un orden parcial, es decir, que se puedan comparar dos individuos y decir cuál de ellos es mejor. Esto suele suceder cuando se necesitan optimizar varios objetivos.

- Una vez evaluado el fitness, se tiene que crear la nueva población teniendo en cuenta que los buenos rasgos de los mejores se transmitan a esta. Para ello, hay que seleccionar a una serie de individuos

Un algoritmo genético puede utilizar muchas técnicas diferentes para seleccionar a los individuos que deben copiarse hacia la siguiente generación, pero abajo se listan algunos de los más comunes. Algunos de estos métodos son mutuamente exclusivos, pero otros pueden utilizarse en combinación, algo que se hace a menudo.

- **Selección elitista**: se garantiza la selección de los miembros más aptos de cada generación. (La mayoría de los algoritmos genéticos no utilizan elitismo puro, sino que usan una forma modificada por la que el individuo mejor, o algunos de los mejores, son copiados hacia la siguiente generación en caso de que no surja nada mejor). Hay estudios que indican que un algoritmo con selección elitista asegura la convergencia del algoritmo genético hacia un óptimo global.

- **Selección proporcional a la aptitud**: los individuos más aptos tienen más probabilidad de ser seleccionados, pero no la certeza.

- **Selección por rueda de ruleta**: una forma de selección proporcional a la aptitud en la que la probabilidad de que un individuo sea seleccionado es proporcional a la diferencia entre su aptitud y la de sus competidores. (Conceptualmente, esto puede representarse como un juego de ruleta - cada individuo obtiene una sección de la ruleta, pero los más aptos obtienen secciones mayores que las de los menos aptos. Luego la ruleta se hace girar, y en cada vez se elige al individuo que “posea” la sección en la que se pare la ruleta).

- **Selección escalada**: al incrementarse la aptitud media de la población, la fuerza de la presión selectiva también aumenta y la función de aptitud se hace más discriminadora. Este método puede ser útil para seleccionar más tarde, cuando todos los individuos tengan una aptitud relativamente alta y sólo les distingan pequeñas diferencias en la aptitud.

- **Selección por torneo:** se eligen subgrupos de individuos de la población, y los miembros de cada subgrupo compiten entre ellos. Sólo se elige a un individuo de cada subgrupo para la reproducción.

- **Selección por rango**: a cada individuo de la población se le asigna un rango numérico basado en su aptitud, y la selección se basa en este ranking, en lugar de las diferencias absolutas en aptitud. La ventaja de este método es que puede evitar que individuos muy aptos ganen dominancia al principio a expensas de los menos aptos, lo que reduciría la diversidad genética de la población, obstaculizando la búsqueda de una solución aceptable, y llevándonos a una convergencia prematura, quedándonos atascados en un máximo local.

- **Selección generacional**: la descendencia de los individuos seleccionados en cada generación se convierte en toda la siguiente generación. No se conservan individuos entre las generaciones.

- **Selección por estado estacionario**: la descendencia de los individuos seleccionados en cada generación vuelven al acervo genético preexistente, reemplazando a algunos de los miembros menos aptos de la siguiente generación. Se conservan algunos individuos entre generaciones.

- **Selección jerárquica:** los individuos atraviesan múltiples rondas de selección en cada generación. Las evaluaciones de los primeros niveles son más rápidas y menos discriminatorias, mientras que los que sobreviven hasta niveles más altos son evaluados más rigurosamente. La ventaja de este método es que reduce el tiempo total de cálculo al utilizar una evaluación más rápida y menos selectiva para eliminar a la mayoría de los individuos que se muestran poco o nada prometedores, y sometiendo a una evaluación de aptitud más rigurosa y computacionalmente más costosa sólo a los que sobreviven a esta prueba inicial.

- **Selección Sigma**: Es una técnica usada para intentar adaptar la selección según evoluciona el algoritmo. Con esta técnica el valor esperado de un individuo depende de su fitness, del fitness medio de la población y de la desviación estándar de la población.

- **Selección de Boltzman**: La técnica anterior consigue una presión de selección constante a lo largo de la ejecución del algoritmo, pero en muchas ocasiones es necesaria una variación de la presión. La selección de Boltzman funciona de manera similar a como funciona el enfriamiento simulado, variando la temperatura que controla la presión de selección. La temperatura inicial debe ser muy elevada lo que significa que la presión de selección es baja y por lo tanto se prima la exploración. La temperatura se va bajando gradualmente con lo que incrementa la presión de selección.

La selección utilizada es muy importante para la correcta convergencia del algoritmo. La presión de selección debe ser tal que consiga un equilibrio entre exploración y explotación. Si la presión de selección es muy alta se corre el peligro 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 perdida de diversidad y una convergencia prematura. Una presión de selección baja puede provocar una búsqueda aleatoria o en el mejor de los casos una enorme ralentización del algoritmo. Probablemente, lo óptimo fuera un operador de selección que fuera evolucionando con el algoritmo de manera que en las primeras fases se primara la exploración y cuya presión fuera creciendo hasta que se alcanzar un punto en el que se primara la explotación.

2.2.6. Operadores especializados

En una serie de problemas hay que restringir las nuevas soluciones generadas por los operadores genéticos, pues no todas las soluciones generadas van a ser válidas, sobre todo en los problemas con restricciones. Por ello, se aplican operadores que mantengan la estructura del problema. Otros operadores son simplemente generadores de diversidad, pero la generan de una forma determinada:

Zap: en vez de cambiar un solo bit de un cromosoma, cambia un gen completo de un cromosoma.

Creep: este operador aumenta o disminuye en 1 el valor de un gen; sirve para cambiar suavemente y de forma controlada los valores de los genes.

Transposición: similar al cruce y a la recombinación genética, pero dentro de un solo cromosoma; dos genes intercambian sus valores, sin afectar al resto del cromosoma. Similar a este es el operador de eliminación-reinserción, en el que un gen cambia de posición con respecto a los demás.

Aplicando operadores genéticos

En toda ejecución de un algoritmo genético hay que decidir con qué frecuencia se va a aplicar cada uno de los algoritmos genéticos; en algunos casos, como en la mutación o el cruce uniforme, se debe de añadir algún parámetro adicional, que indique con qué frecuencia se va a aplicar dentro de cada gen del cromosoma. La frecuencia de aplicación de cada operador estará en función del problema; teniendo en cuenta los efectos de cada operador, tendrá que aplicarse con cierta frecuencia o no. Generalmente, la mutación y otros operadores que generen diversidad se suele aplicar con poca frecuencia; la recombinación se suele aplicar con frecuencia alta.

En general, la frecuencia de los operadores no varía durante la ejecución del algoritmo, pero hay que tener en cuenta que cada operador es más efectivo en un momento de la ejecución. Por ejemplo, al principio, en la fase denominada de exploración, los más eficaces son la mutación y la recombinación; posteriormente, cuando la población ha convergido en parte, la recombinación no es útil, pues se está trabajando con individuos bastante similares, y es poca la información que se intercambia. Sin embargo, si se produce un estancamiento, la mutación tampoco es útil, pues está reduciendo el algoritmo genético a una búsqueda aleatoria; y hay que aplicar otros operadores. En todo caso, se pueden usar operadores especializados.

Por ejemplo, en el algoritmo genético para jugar al MasterMind (http://kal-el.ugr.es/mastermind), se usan varios operadores genéticos: transposición, mutación y entrecruzamiento. Sin embargo, la mutación y el cruce dejan de ser efectivos en el momento que la combinación que se ha jugado tiene los colores correctos, y en cualquier caso la tasa de mutación tendrá que ser mayor cuantos menos colores haya averiguados; por eso las tasas varían durante la ejecución, convirtiéndose eventualmente en 0.

2.3. Tamaño de la población

Un factor muy importante para la convergencia de los algoritmos genéticos es el tamaño de la población. El tiempo necesario para que un AG converja a una solución única depende del tamaño de la población. Goldberg y Deb publicaron un estudio en 1991 en el que demuestran que el tiempo para que un individuo se propague a toda la población utilizando los métodos más rápidos de selección es O(n * logn) siendo n el tamaño de la población. Aunque los algoritmos genéticos 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. Pero también es bastante obvio que esto acarreará un costo computacional mayor. Por eso se debe buscar un compromiso entre el número de individuos utilizados y la calidad que se desea alcanzar.

2.4. Adaptación del código

La elección de un código fijo tanto en longitud como en orden tiene varios inconvenientes. En lo que a orden se refiere, puede ocurrir que un orden fijo produzca la ruptura de buenos esquemas debido a los efectos del cruce y de la mutación. Este conocimiento no se conoce en tiempo de "compilación", si así fuera sería tanto como conocer la respuesta al problema. Por lo tanto sería deseable que la codificación fuera cambiando el orden de los bits para evitar la ruptura y desaparición de los mejores esquemas. A este problema se le denomina el problema del "Linkage", se quiere que la funcionalidad relacionada con la posición tenga mas probabilidad de permanecer junta bajo cruces y mutación, pero no está muy claro como saber que posiciones deben conservarse en tiempo de compilación. La solución parece ser adaptar el código al tiempo que se avanza en la resolución del problema.

Otra razón para adaptar el código es que la longitud fija del cromosoma limita la complejidad de las soluciones candidatas. Existen estrategias como la programación genética que automáticamente permiten la adaptación del tamaño del código puesto que tras los cruces y las mutaciones los árboles crecen o se reducen.

2.5. Resumen

Como hemos podido observar, la principal ventaja de los algoritmos genéticos radica en su sencillez. Se requiere poca información sobre el espacio de búsqueda ya que se trabaja sobre un conjunto de soluciones o parámetros codificados (hipótesis o individuos). Se busca una solución por aproximación de la población, en lugar de una aproximación punto a punto. Con un control adecuado podemos mejorar la aptitud promedio de la población, obteniendo nuevos y mejores individuos y, por lo tanto, mejores soluciones.

Se consigue un equilibrio entre la eficacia y la eficiencia. Este equilibrio es configurable mediante los parámetros y operaciones usados en el algoritmo. Así, por ejemplo, bajando el valor del umbral conseguiremos una rápida solución a cambio de perder en "calidad". Si aumentamos dicho valor, tendremos una mejor solución a cambio de un mayor tiempo consumido en la búsqueda. Es decir, obtenemos una buena relación entre la calidad de la solución y el costo.

Quizás el punto más delicado de todo se encuentra en la definición de la función de evaluación, y de su eficacia depende el obtener buenos resultados. El resto del proceso es siempre el mismo para todos los casos.

La programación mediante algoritmos genéticos supone un nuevo enfoque que permite abarcar todas aquellas áreas de aplicación donde no sepamos como resolver un problema:

Aplicaciones de búsqueda y optimización:

Desde aplicaciones evidentes, como la biología o la medicina, hasta otros campos como la industria (clasificación de piezas en cadenas de montaje). Los algoritmos genéticos poseen un importante papel en este ámbito.

Aprendizaje automático:

La capacidad que poseen para favorecer a los individuos que se acercan al objetivo, a costa de los que no lo hacen, consigue una nueva generación con mejores reglas y, por lo tanto el sistema será capaz de ir aprendiendo a conseguir mejores resultados. Es aquí donde encuentran un estupendo marco de trabajo los algoritmos genéticos.

Pasos Algoritmo genético:

1. Evaluar la puntuación (fitness) de cada uno de los genes.

2. Permitir a cada uno de los individuos reproducirse, de acuerdo con su puntuación.

3. Emparejar los individuos de la nueva población, haciendo que intercambien material genético, y que alguno de los bits de un gen se vea alterado debido a una mutación espontánea.

3. Algoritmos paralelos

3.1. Paralelismo de código

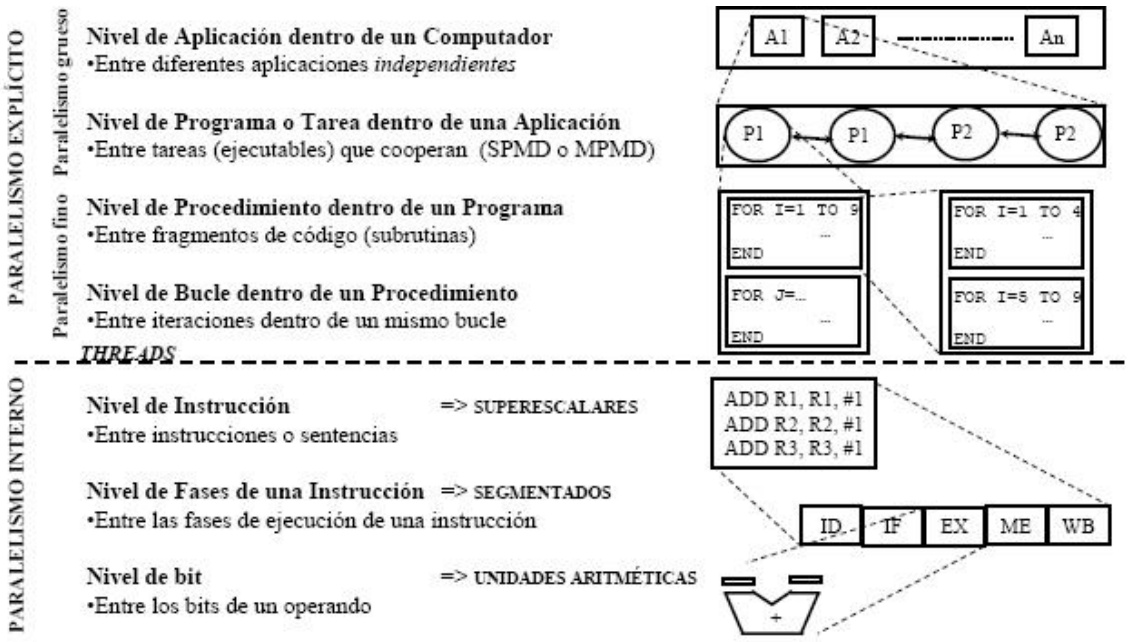
En una primera aproximación, podríamos pensar que dado un programa secuencial, podríamos ejecutarlo en N procesadores (dentro de la misma máquina o en diferentes máquinas), lo que previsiblemente ejecutaría el código N veces más rápido. Esto no es del todo cierto, ya que la ganancia es mucho menor debido a las partes secuenciales del programa, la sobrecarga debida a las comunicaciones/sincronizaciones y la desigualdad de carga en los diferentes procesadores.

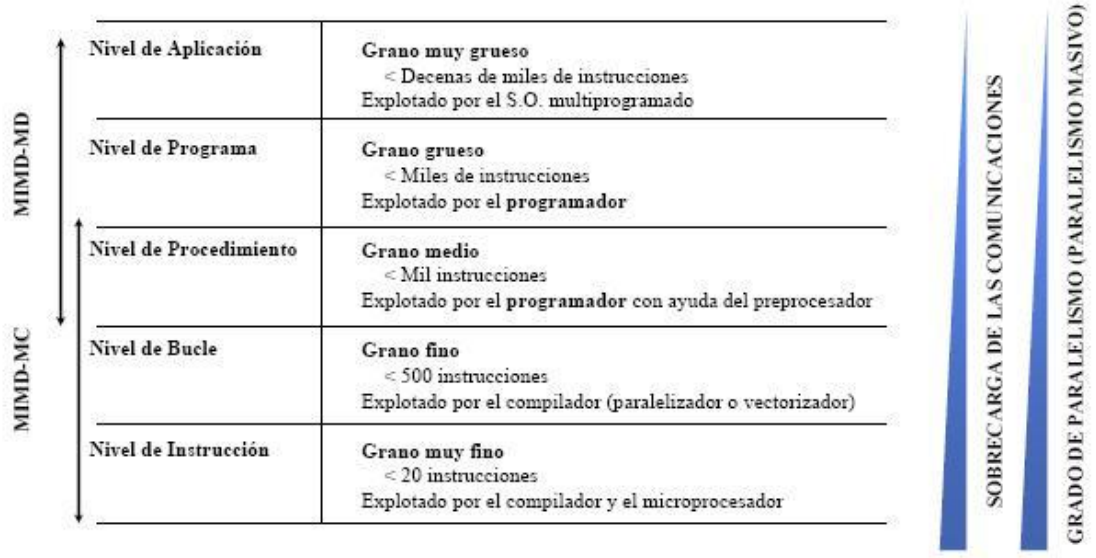
Las diferentes posibilidades de la computación paralela son:

- Resolución de problemas en menor **tiempo de ejecución**, usando más procesadores.
- Resolución de problemas con mayor **precisión**, usando mas memoria
- Resolución de problemas más reales, usando modelos matemáticos más **complejos.**

La necesidad de cálculo en este último punto puede ser crítica en campos como la predicción meteorológica. En este campo, sin la computación paralela sería inviable dar un pronóstico, ya que el tiempo necesario para todos los cálculos necesarios sería superior a la fecha en la que se quiere dar la predicción.

El tamaño de grano o granularidad es una medida de la cantidad de computación de un proceso software. Se considera como el segmento de código escogido para su procesamiento paralelo:





Pasaremos a explicar un poco más a fondo la parte del paralelismo explícito, que es del que nos encargaremos:

- Paralelización de grano fino y medio: la paralelización del programa se realiza a nivel de instrucción. La paralelización se realiza automáticamente por los compiladores. Por ejemplo se trataría de la descomposición de bucles.

- Paralelización de grano grueso: se basan en la descomposición del dominio de datos entre los procesadores, siendo cada uno de ellos el responsable de realizar los cálculos sobre sus datos locales.

Pasaremos a tratar un poco más a fondo las estrategias de paralelización de grano grueso cuyo mayor atractivo es la portabilidad, ya que se adapta perfectamente tanto a multiprocesadores de memoria distribuida como de memoria compartida. Este tipo de paralelización se puede a su vez realizar siguiendo tres estilos distintos de programación: paralelismo en datos, programación por paso de mensajes y programación por paso de datos.

- 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. Como principal ventaja presenta su facilidad de programación. Sin embargo suelen tener una eficiencia inferior a la que se consigue con el paso de mensajes. Los lenguajes de paralelismo de datos más utilizados son el estándard HPF (High Performance Fortran) y el OpenMP.

- Programación por paso de mensajes: El método más utilizado para programar sistemas de memoria distribuida es el paso de mensajes o alguna variante del mismo. 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.

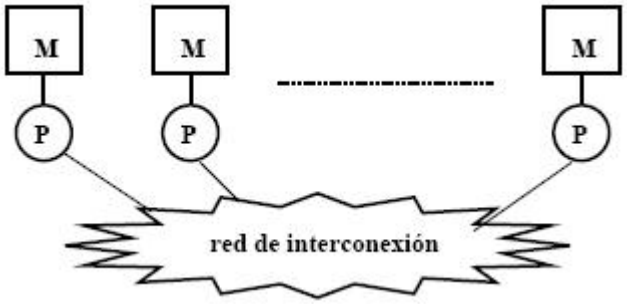
  Las librerías más utilizadas son por este orden la estándar MPI (Message Passing Interface) y PVM (Parallel Virtual Machine).

- Programación por paso de datos: A diferencia del modelo de paso de mensajes, 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, aunque en la actualidad son muy pocos los fabricantes que los soportan.

3.2. MPI (Message Passing Interface)

El fundamento del MPI es 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. La visión del programador es el lenguaje secuencial con variables privadas + Rutinas de paso de mensajes.



Como ya se ha indicado antes, la principal ventaja es que es portable de modo eficiente a cualquier tipo de arquitectura: computador paralelo, red de estaciones y a una única estación de trabajo.

Un mensaje es una transferencia de datos de un proceso a otro proceso.

La información que caracteriza el mensaje es:

- En qué variable están los datos que se envían
- Cuántos datos se envían
- Qué proceso recibe el mensaje
- Cuál es el tipo de dato que se envía
- Qué proceso envía el mensaje
- Dónde almacenar los datos que se reciben
- Cuántos datos espera recibir el proceso receptor

Existen dos tipos de envío de mensaje:

1. **Síncrono**:
   - El proceso que realiza el envío recibe información sobre la recepción del mensaje (como por ejemplo un fax).
   - La comunicación se completa cuando el mensaje ha sido recibido.

2. **Asíncrono**:
   - El proceso únicamente conoce cuándo se envía el mensaje (como por ejemplo una postal).
   - La comunicación se completa tan pronto como elmensaje ha sido enviado. Generalmente se copia en un buffer.

A su vez, existen dos tipos de **operaciones: bloqueantes y no bloqueantes**, según esperen o no a la condición de finalización.

- 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**: sólo se vuelve al programa cuando la operación ha finalizado (envío y recepción de fax estándar)

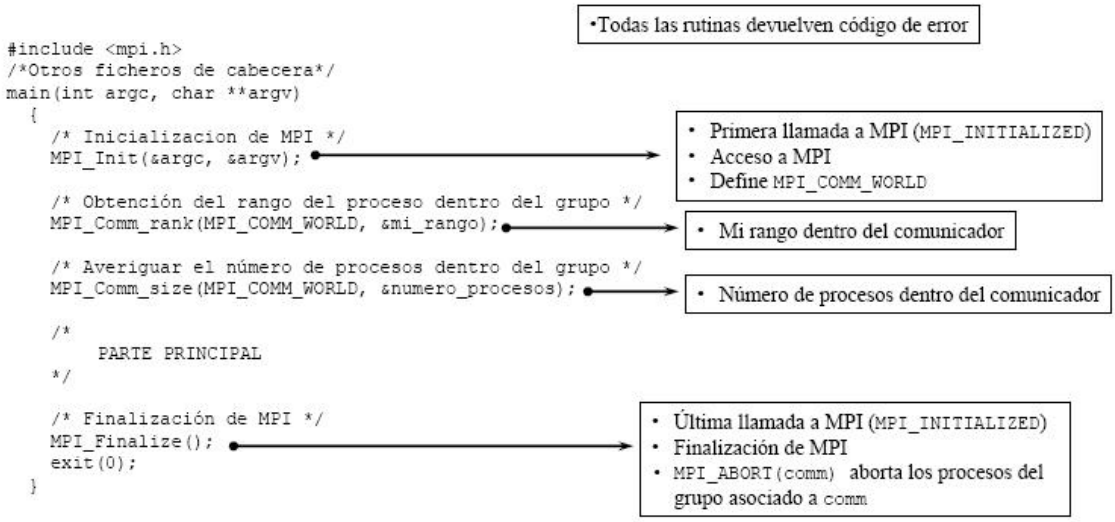
Ahora introduciremos el uso de MPI, por haber sido el estándar elegido para la realización de las pruebas.

Comunicadores:

Son conjuntos de procesos que pueden intercambiar mensajes. Cada comunicador contiene un grupo. Los procesos dentro de un grupo están numerados.

Todas las rutinas de comunicación MPI requieren un comunicador, que por defecto es MPI_COMM_WORLD, que incluye a todos los procesos que se están ejecutando cuando el programa comienza.

Veamos la estructura básica de un programa en MPI:



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

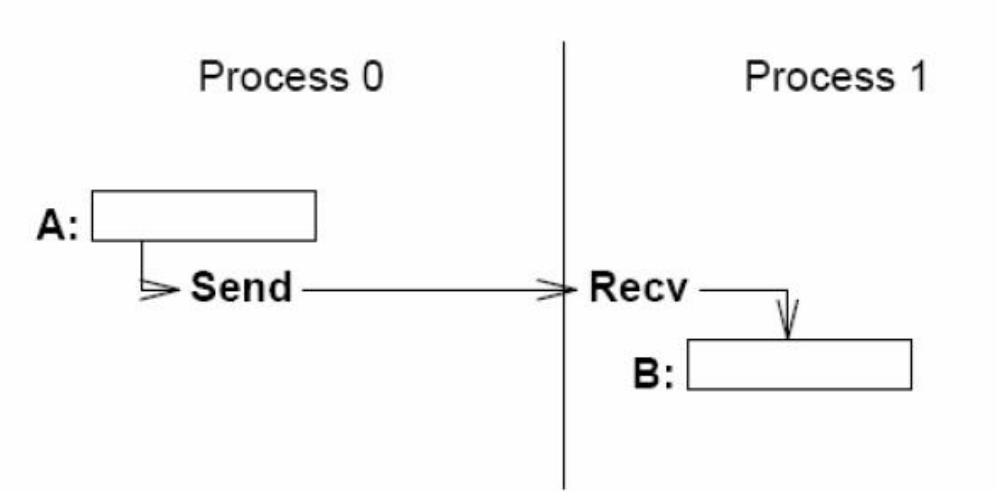
Aplicaciones MPI

MPI es muy sencillo y la mayoría de los programas se pueden realizar utilizando solo 6 funciones básicas:

- 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.

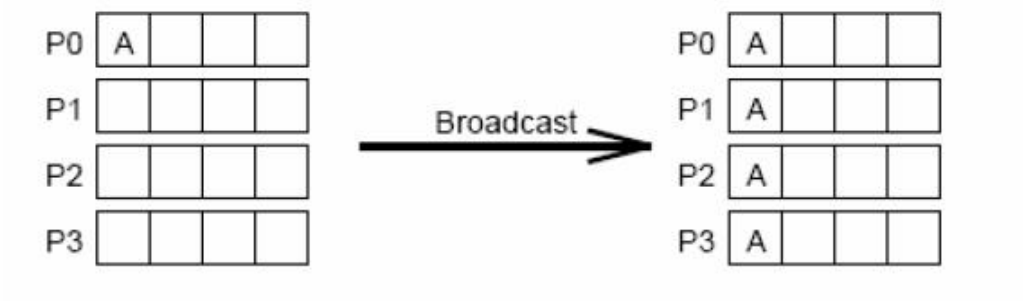
Las dos operaciones básicas para intercambiar información entre dos procesos son:

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

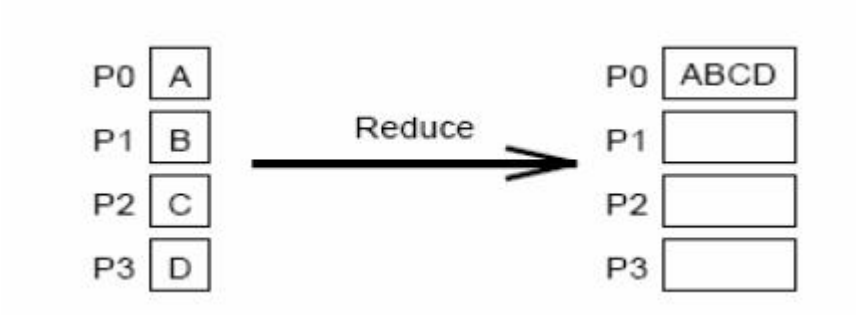


Si el programa realiza operaciones cooperantes entre todos los procesos las dos operaciones básicas son:

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



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



4. Algoritmos genéticos paralelos

Los principales métodos de paralelización de Algoritmos Genéticos consisten en la división de la población en varias sub-poblaciones (demes). 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. A continuación veremos una clasificación de los algoritmos genéticos paralelos y el funcionamiento básico de cada uno de ellos.

4.1. Clasificación de los 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 .

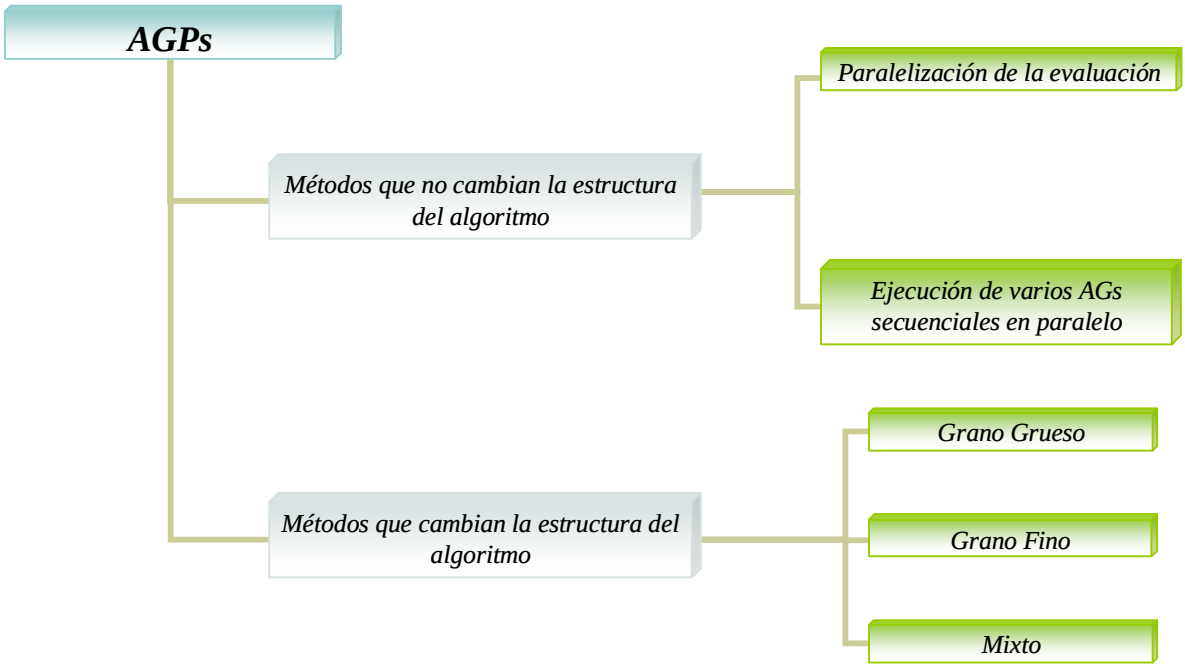
Al compartir la carga de trabajo entre N procesadores se puede esperar que el sistema trabaje N veces más rápido que con un solo procesador, lo que permite tratar problemas más grandes y complicados. Las cosas no son tan sencillas, ya que existen varios factores de sobrecarga que hacen disminuir el rendimiento previsible. En ocasiones existen problemas cuya estructura no es lo suficientemente regular cómo 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 por ejemplo los algoritmos genéticos, que tienen una estructura que se adapta perfectamente a la paralelización. De hecho la evolución natural es en sí un proceso paralelo ya que evoluciona utilizando varios individuos.

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 población. Otra forma de paralelización global consiste en realizar una ejecución de distintos Algoritmos Genéticos 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 que se denominan mixtos. En la figura podemos ver un esquema de la clasificación de los Algoritmo Genético Paralelos. Además de conseguir tiempos de ejecución menores, al paralelizar un Algoritmo Genético estamos modificando el comportamiento algorítmico, y esto hace que podamos obtener otras soluciones y experimentar con las distintas posibilidades de implementación y los distintos factores

que influyen en ella. Estas poblaciones van evolucionando por separado para detenerse en un momento determinado e intercambiar los mejores individuos entre ellas. Técnicamente hay 3 características importantes que influyen en la eficiencia de un algoritmo genético paralelo y que se analizarán más a fondo en el apartado 3.3 :

- La topología que define la comunicación entre subpoblaciones.
- La proporción de intercambio: número de individuos a intercambiar.
- Los intervalos de migración: periodicidad con que se intercambian los individuos.



4.2. Paralelización global

La paralelización global es la forma más sencilla de implementar un algoritmo genético paralelo. Cómo ya se ha dicho, consiste o en paralelizar la evaluación de los individuos o en realizar una ejecución simultánea de distintos Algoritmos Genéticos secuenciales. Sin embargo es muy útil, ya que permite obtener mejoras en el rendimiento con respecto al Algoritmo Genético secuencial muy fácilmente sin cambiar la estructura principal de éste. En la mayoría de las aplicaciones de los Algoritmos Genéticos 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.

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 habrá que tener en cuenta el costo de comunicaciones y cuánto más sencilla sea la información a enviar menor 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. A continuación se puede ver una descripción de un algoritmo genético en el que se ha paralelizado 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
```

Este método mantiene una población única y la evaluación de los individuos se realiza en paralelo. La aplicación de los operadores puede mantenerse global o realizarse en paralelo aunque generalmente el costo de comunicaciones necesario para paralelizar estas operaciones no compensa el tiempo de cómputo ahorrado.

Cuando el programa se para y espera el resultado de la evaluación para todos los individuos antes de proceder a crear la siguiente generación, se dice que es una implementación síncrona. Este algoritmo tendrá las mismas características que un algoritmo secuencial.

Si el procesador maestro no espera la llegada de todas las evaluaciones tenemos una implementación asíncrona. En este caso la estructura y comportamiento del algoritmo difiere de la versión secuencial ya que la generación de los nuevos individuos no se realiza de la misma forma. No se produce en el mismo instante de tiempo, por lo que determinados individuos pueden ser seleccionados en una generación anterior o posterior. Puede ser incluso que no sean seleccionados, bien porque su evaluación llega tarde y la calidad de la nueva población ha aumentado o bien por la mera probabilidad intrínseca a los algoritmos evolutivos. La mayoría de las implementaciones de un AGP global suelen ser síncronas por 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. En el pseudo código mostrado a continuación se puede ver un esquema de este método.

```
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
```

El modelo de paralelización global no hace ninguna distinción sobre la arquitectura del computador sobre el que se está ejecutando. Se puede implementar tanto en un computador de memoria compartida como de memoria distribuida.

En un multi-procesador de memoria compartida, la población se puede guardar en la memoria compartida y cada uno de los procesadores puede leer los individuos que tiene asignados, evaluarlos y devolver los resultados de tal forma que no se produzcan conflictos entre procesadores para acceder a la memoria y no hace falta sincronización en este paso. Los problemas pueden aparecer únicamente como consecuencia de la propia red que pueden hacer disminuir la velocidad de ejecución del algoritmo.

El número de individuos asignado a cada procesador suele ser constante, pero en algunos casos puede ser necesario equilibrar la carga entre procesadores, para lo que se puede utilizar cualquier algoritmo dinámico diseñado para este propósito. El equilibrio de la carga no es nada más que distribuir homogéneamente la cantidad de trabajo que realiza cada procesador de acuerdo a sus características. Todos los problemas tratados en este trabajo producen un equilibrio natural de la carga por lo que no ha sido necesario utilizar ninguna técnica adicional en este sentido.

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 un multiprocesador de memoria distribuida y uno de memoria compartida, la diferencia es que en un procesador de memoria distribuida se debe especificar explícitamente. Cuánto mayor es el número de esclavos utilizados mayor es el costo de comunicaciones, pero evidentemente también disminuye el número de operaciones que tiene que realizar cada uno de los procesadores.

Una consideración importante a la hora de implementar un algoritmo genético paralelo global es que si se realizan muchas comunicaciones podemos perder cualquier reducción del rendimiento alcanzada mediante la paralelización.

Esta condición indica que para problemas simples con tiempos de ejecución cortos, los Algoritmos Genéticos Paralelos globales no son una buena opción para mejorar el rendimiento, pero para problemas con tiempos de ejecución elevados consiguen una mejora sustancial.

4.3. Algoritmos Genéticos Paralelos de Grano Grueso

Las características fundamentales de un Algoritmo Genético Paralelo de grano grueso son la utilización de varias subpoblaciones relativamente grandes y la migración. Se conoce como migración al intercambio de individuos entre distintas subpoblaciones.

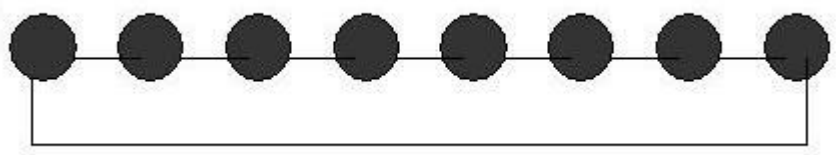
Este tipo de paralelización es el más utilizado. Las principales razones de su popularidad son:

- La forma de implementar un Algoritmo Genético Paralelo de grano grueso a partir de una versión secuencial es muy sencilla. Únicamente hay que tomar un conjunto de Algoritmos Genéticos 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 presencia de los computadores paralelos de grano grueso es habitual en la mayoría de los centros de investigación y allí donde no están disponibles son fácilmente simulables mediante software como MPI (Message Passing Interface) ó PVM (Parallel Virtual Machine).

Los Algoritmos Genéticos Paralelos de grano grueso también son conocidos como Algoritmos Genéticos distribuidos, debido a que se suelen implementar sobre máquinas de memoria distribuida. En ocasiones se les llama Algoritmos Genéticos Paralelos de "isla" (*island model*) ya que existe un modelo de poblaciones en el que las subpoblaciones están relativamente aisladas. Puede parecer, que dado que utilizamos subpoblaciones de menor tamaño que en un Algoritmo Genético secuencial (o en serie), un Algoritmo Genético Paralelo de grano grueso debería converger en un número menor de generaciones. Aunque es cierto que con una población menor un Algoritmo Genético converge más rápido, también es cierto que la calidad de la solución no tiene porque ser la misma si el problema que se está tratando es complejo.

En la figura que tenemos a continuación se puede ver un diagrama de un Algoritmo Genético Paralelo de grano grueso en forma de grafo. En él se representan las poblaciones por nodos y las aristas representan las líneas de intercambio de individuos al realizar la migración.



Los Algoritmos Genéticos Paralelos 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 podemos ver un esquema en pseudocódigo de un Algoritmo Genético Paralelo de grano grueso, en el que la frecuencia es el número de generaciones que pasa entre dos intercambios de individuos.

```
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
```

Cómo ya se ha indicado hay 3 parámetros importantes que influyen en la eficiencia de un Algoritmo Genético Paralelo:

- La topología que define la comunicación entre subpoblaciones
- La proporción de intercambio: número de individuos a intercambiar
- Los intervalos de migración o frecuencia: periodicidad con que se intercambian los individuos.

Veamos un poco más extensamente cómo influyen estos puntos en el funcionamiento de los Algoritmos Genéticos Paralelos de grano grueso y las distintas alternativas que se presentan.

4.4. Topologías de comunicación

La topología es un factor fundamental en el rendimiento de un Algoritmo Genético Paralelo 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. Es evidente que las malas soluciones también lo harían, pero precisamente el proceso de selección gobernado por la función de coste controla este esparcimiento y lo limita a la pura probabilidad de selección que puede tener una mala solución.

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. Estas distintas soluciones se pueden utilizar posteriormente para generar individuos que superen a los obtenidos con una convergencia más homogénea.

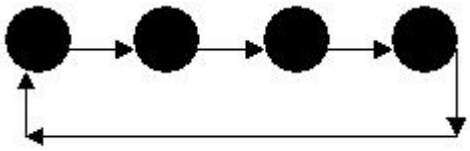
Otro factor en el que interviene la topología es el coste de comunicaciones. Aunque normalmente no se realizan tantos intercambios como para ralentizar el Algoritmo Genético Paralelo, es necesario buscar un compromiso entre la topología elegida y el costo de comunicaciones, para no perder las ventajas 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. Muchas de estas topologías estáticas aprovechan la propia topología de la red de comunicaciones entre procesadores.

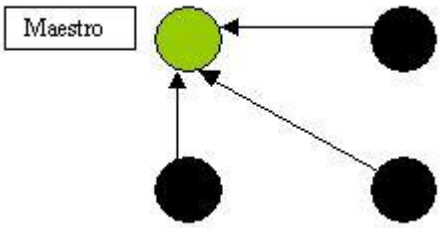
Otra opción es la implementación de una topología dinámica. En ellas el intercambio entre subpoblaciones se realiza entre procesadores distintos cada vez en función de un criterio que suponga una mejora para el algoritmo. Uno de los factores que más se tienen en cuenta para este tipo de Algoritmos Genéticos Paralelos es la diversidad de la población, es decir se trata de favorecer el intercambio con aquellas subpoblaciones con un mayor número de individuos iguales.

Topologías más utilizadas en la implementación del modelo de islas:

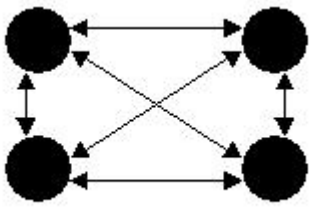
- **Topología en anillo**: las poblaciones están distribuidas en un anillo, y sólo hay intercambio de individuos entre vecinos.



- **Topología maestro-esclavo o *master-slave***: todos los procesadores esclavos intercambian sus mejores individuos con el maestro.



- **Comunicación todos con todos (all-to-all)**: todos los procesadores intercambian información con cada uno de los otros.



4.5. Proporción y frecuencia de intercambio

Tanto la frecuencia cómo 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 Algoritmo Genético Paralelo en una búsqueda prácticamente aleatoria al no permitir que el Algoritmo Genético se desarrolle con normalidad.

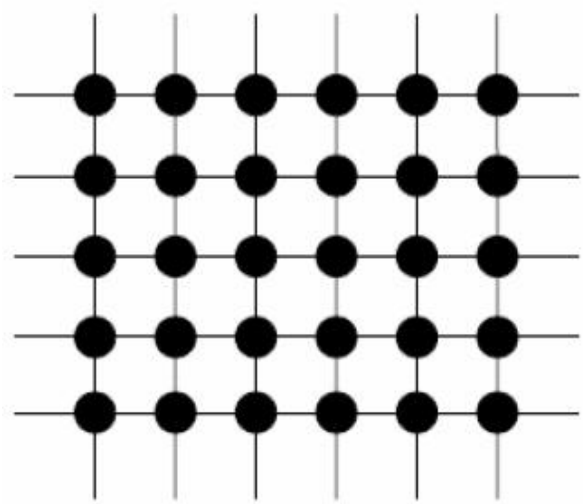
Algunas implementaciones realizan el proceso de migración o intercambio únicamente cuando las poblaciones han convergido totalmente. Con ello se trata de introducir una diversidad en las poblaciones y aliviar así posibles problemas de convergencia prematura. Otras soluciones utilizan distintas políticas de intercambio y lo realizan después de un determinado número de generaciones o bien con una periodicidad fijada de antemano y que se mantiene constante a lo largo de toda la ejecución del programa.

Si la migración se produce muy pronto, puede suceder que las propiedades de los individuos intercambiados sean débiles e influyan muy levemente en el proceso de búsqueda.

Por todo lo citado anteriormente es conveniente realizar un estudio de distintas políticas de migración, teniendo en cuenta a la vez la topología de las poblaciones. Existen trabajos en los que se han estudiado estos problemas aunque todavía son un campo emergente y es necesario un estudio particular para cada problema, cada sistema y cada implementación.

4.6. Algoritmos Genéticos Paralelos de Grano Fino

Los algoritmos genéticos de grano fino, se conocen también como Algoritmos Genéticos Paralelos *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. A continuación se puede ver un esquema en pseudocódigo de un Algoritmo Genético Paralelo 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 descendiente de k y q
            asignar uno a k
            aplicar operador mutacion sobre k con probabilidad Pm
         end parallel do
   end while
```

En la descripción anterior los vecinos que se consideran son generalmente los cuatro u ocho más próximos. La forma de seleccionar al individuo de la vecindad con el que se interactúa se puede hacer de varias formas, la más utilizada es por torneo, es decir compiten entre ellos mediante el valor de su función de coste.

De igual forma, la sustitución de los antiguos individuos por los nuevos a partir de los descendentes se puede hacer eligiendo el de mejor función de coste o mediante un proceso aleatorio.

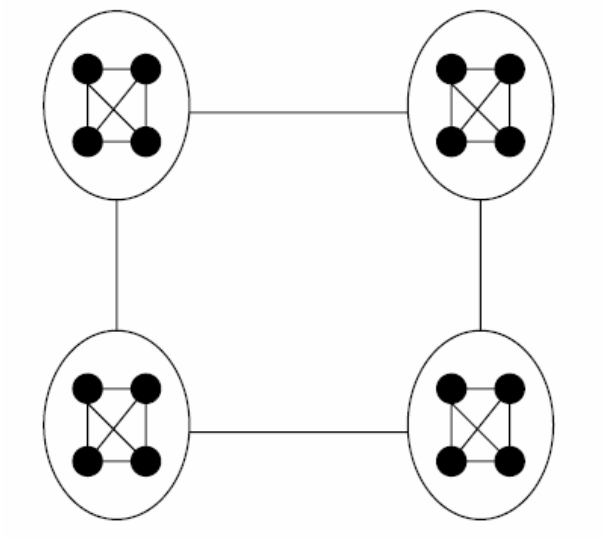
Todos estos procesos se pueden hacer de una manera dinámica o bien se pueden realizar con individuos que viajen u otras variantes que se desee implementar.

4.7. Algoritmos Genéticos Paralelos Híbridos

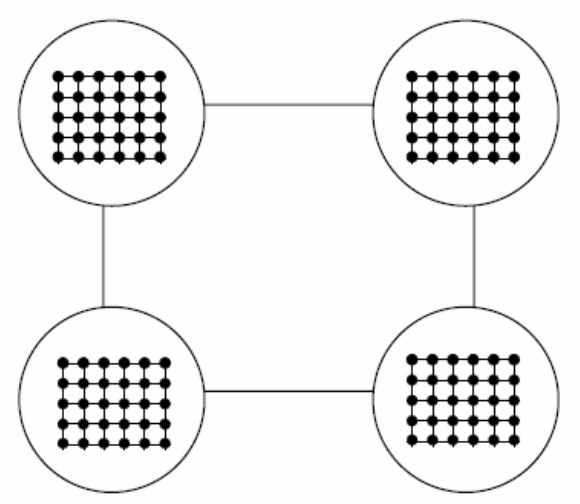
Cuando se combinan los Algoritmos Genéticos Paralelos de grano fino con los de grano grueso se produce lo que se conoce como algoritmo genético híbrido. Alguno de estos algoritmos híbridos añaden un nuevo grado de complejidad al entorno de los Algoritmos Genéticos Paralelos, pero otros utilizan la misma complejidad que uno de sus componentes.

Cuando se combinan dos métodos de Algoritmos Genéticos Paralelos forman una jerarquía. Los Algoritmos Genéticos Paralelos híbridos son normalmente paralelos de grano grueso en el nivel superior. Alguno de ellos tienen un Algoritmo Genético Paralelo de grano fino en el nivel inferior y otros tienen tanto en el primer como en el segundo nivel Algoritmos Genéticos Paralelos de grano grueso.

Veamos algún esquema de este tipo de algoritmos:



Este Algoritmo Genético Paralelo híbrido combina un algoritmo de grano grueso tanto en el primer como en el segundo nivel de jerarquía.



Este otro algoritmo híbrido combina un Algoritmo Genético Paralelo de grano grueso con uno global.

5. PGAPack

El PGAPack es una librería de algoritmos genéticos paralelos. Contiene la mayoria de las capacidades necesarias en una librería de algoritmos genéticos y es portable.

Sus principales utilidades son las siguientes:

1. Capacidad de usarlo desde Fortran o C
2. Ejecutable en uniprocesadores, multiprocesadores, multicomputadoras y redes de estaciones de trabajo.
3. tipos de datos nativos binarios, enteros, reales y char.
4. Estructura de datos orientada a objetos
5. Reemplazamiento de población parametrizada.
6. Múltiples elecciones para operadores selección, cruce y mutación.
7. Fácil integración a heurísticas de escalada.

5.1. Ejemplo de uso

Todos los programas de PGAPAck en C deben incluir el fichero de cabecera *pgapack.h*.

La llamada a *PGACreate* es siempre la primera función a la que se llama en un programa PGAPack. Esta llamada inicializa la variable de contexto *ctx*. Los parámetros de PGACreate son los argumentos del programa (dados por *argc* y *argv*), el tipo de datos seleccionado (como por ejemplo *PGA_DATATYPE_BINARY*), la longitud de la cadena, y el tipo de optimización (por ejemplo *PGA_MAXIMIZE*). La llamada a *PGASetUp* inicializa todos los parámetros y variables de las funciones no explicitadas por el usuario a los valores por defecto.

PGARun ejecua el algoritmo genético. Su segundo argumento es el nombre de la función definida por el usuario (*evaluate*) que será llamada para evaluar las cadenas. *PGADestroy* limpia la memoria reservada por el PGAPAck. Véase que todas las funciones de PGAPack tienes la variable de contexto como argumento (excepto *PGACreate*).

La funcion *evaluate* debe ser escrita por el usuario, debe devolver un *double*, y debe tener la secuencia de llamadas igual que la de ejemplo. La función *PGAGetStringLength* devuelve la longitud de la cadena. *PGAGetBinaryAllele* devuelve el valor del bit i-ésimo de la cadena p en la población pop.

Veamos el ejemplo del One Max:

```
#include "pgapack.h"
double evaluate (PGAContext *ctx, int p, int pop);

int main(int argc, char **argv)
{
    PGAContext *ctx;
    ctx = PGACreate (&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
    PGASetUp        (ctx                                                  );
    PGARun          (ctx, evaluate                                        );
    PGADestroy      (ctx                                                  );
    return;
}

double evaluate (PGAContext *ctx, int p, int pop)
{
    int i, nbits, stringlen;

    stringlen = PGAGetStringLength(ctx);
    nbits     = 0;
    for (i=0; i<stringlen; i++)
        if (PGAGetBinaryAllele(ctx, p, pop, i))
            nbits++;
    return((double) nbits);
}
```

5.2. Estructura del PGAPack

Tipos de datos nativos: soporta cuatro tipos de datos nativos: binarios, enteros, reales y caracteres, aunque se pueden extender para soportar otros tipos de datos.

Los bits (binarios) son los usados tradicionalemente en Algoritmos Genéticos. Las cadenas de bits podrían ser interpretadas literalmente o decodificadas en valores reales o enteros, usando otros códigos. Los tipos de datos enteros se usan a menudo para problemas de planificación. Los tipos de datos reales se usan en aplicaciones de optimización numérica. Los tipos de caracteres se usan para aplicaciones simbólicas.

Variable de contexto: es una estructura de datos que provee las capacidades de ocultación de código. Es un puntero a una estructura de C, que en ella misma es una colección de otras estructuras. Estas subestructuras contienen toda la información necesaria para ejecutar el algoritmo genético, incluyendo tipos de datos construidos, parámetros, qué funciones hay que llamar, parámetros del sistema operativo, flags para depuración, inicializaciones y arrays internos. Mediante la ocultación de estos tipos, las funciones en el nivel de usuario del PGAPack pueden llamarse independientemente del tipo de datos.

Casi todos los campos de la variable de contexto tienen valores por defecto. Sin embargo, el usuario puede cambiar los valores en la variable usando las funciones *PGASet*. Los valores de los campos de la variable de contexto se pueden leer mediante las funciones *PGAGet.*

Criterio de parada

El PGAPack termina cuando al menos una de las reglas de parada especificadas se alcanza. Las tres reglas de parada son:

- el límite de iteraciones se ha sobrepasado,
- la población es muy similar
- no hay cambio en la mejor solución hallada en un número dado de iteraciones.

El criterio por defecto es parar cuando el límite de iteraciones (por defecto, 1000 iteraciones) se ha alcanzado.

La elección de la regla de parada se establece con *PGASetStoppingRuleType*.

Inicialización

Las cadensa se inicializan aleatoriamente (por defecto), o se inicializan a cero. Esta elección se especifica con el segundo argumento de *PGASetRandomInitFlag*, poniéndolo a *PGA_TRUE* o a *PGA_FALSE*.

Para cadenas binarias, cada gen se inicializa a 1 o a 0 con la misma probabilidad (en inicialización aleatoria). Para cambiar la probabilidad a 0.3 de inicializar un bit 1, hay que usar la función *PGASetBinaryInitProb(ctx,0.3)*.

Selección

PGAPack soporta cuatro tipos de selección: proporcional, estocástica universal, torneo binario y torneo binario probabilístico. La elección se especifica inicializando el segundo argumento de *PGASetSelectType* a uno de *PGA_SELECT_PROPORTIONAL, PGA_SELECT_SUS, PGA_SELECT_TOURNAMENT* y *PGA_SELECT_PTOURNAMENT* para selección proporcional, estocástica, torneo y torneo probabilístico respectivamente. Por defecto, se utiliza selección por torneo.

Cruce

El tipo de cruce se puede especificar mediante *PGASetCrossoverType* a *PGA_CROSSOVER_ONEPT, PGA_CROSSOVER_TWOPT* o *PGA_CROSSOVER_UNIFORM*, para cruce de un punto, dos puntos, o cruce uniforme, respectivamente. Por defecto se utiliza el cruce de dos puntos.

Mutación

Utilizando la llamada a la función *PGASetMutationProb(ctx, 0.001)*, inicializaremos la probabilidad de mutación a 0.001.

No entraremos más en detalle del resto de las opciones de mutación, ya que sólo utilizaremos población binaria, por lo que la mutación consistirá en cambiar alguno de sus bit de 0 a 1 o al revés con cierta probabilidad.

5.3. Modificaciones realizadas en el código

En el archivo ***pgapack.h*** se han realizado las siguientes modificaciones:

```
/*****************************************
*       INDIVIDUAL STRUTURE                *
*****************************************/

typedef struct {                    /* primary population data structure   */
  double evalfunc;                  /* evaluation function value           */
  double fitness;                   /* fitness    function value           */
  int    evaluptodate;              /* flag whether evalfunc is current    */
  void   *chrom;                    /* pointer to the GA string            */
  int procesador[8]; /*En la posicion 0 procesador 0, en la 1 procesador
1,...pE guardar el arbl genealogico del individuo*/
  //int Origen; //procesador original donde se creo este individuo
} PGAIndividual;

int IdProceso;
int numProcesadores;
```

Se ha añadido la variable procesador[] en la cual se guardara la herencia de cada individuo. Es decir, el paso de cada individuo por cada procesador. Asi como las variables globales IdProceso y numProcesadores donde se guardara el identificador de proceso y el número de procesadores totales implicados en la ejecución respectivamente.

En el archivo ***binary.c*** se han realizado las siguientes modificaciones:

```
void PGABinaryPrintString( PGAContext *ctx, FILE *fp, int p, int pop ){
       …

       fprintf(fp,"Origen: ");
       fprintf(fp," Procesadores: ");
       int j=0;
       for (j=0;j<numProcesadores;j++)
              {
              int s;
              s =PGAGetIndividual(ctx, p, pop)->procesador[j];
              fprintf(fp,"%i",s);
              }

       …
}
```

A la hora de la impresion de un individuo debemos tener en cuenta la presentacion tambien de su herencia correspondiente.

```
void PGABinaryCopyString (PGAContext *ctx, int p1, int pop1, int p2, int pop2)
{
       …
       PGAIndividual *fuente, *destino;
       fuente = PGAGetIndividual ( ctx, p1, pop1 );
       destino   = PGAGetIndividual ( ctx, p2, pop2 );
       int j = 0;
       for (j=0;j<8;j++)
              destino->procesador[j] = fuente->procesador[j];
       …
}
```

Al realizar la copia de un individuo debemos copiar también su variable procesador[].

```
MPI_Datatype PGABinaryBuildDatatype(PGAContext *ctx, int p, int pop){
       …
       MPI_Address(traveller->procesador, &displs[4]);
     counts[4] = 8;
     types[4]  = MPI_INT;

     //MPI_Address(&traveller->Origen, &displs[5]);
     //counts[5] = 1;
     //types[5]  = MPI_INT;

     MPI_Type_struct(5, counts, displs, types, &individualtype);
     MPI_Type_commit(&individualtype);
       …
}
```

Cada vez que se realiza un envio un individuo a otro procesador se debe construir un tipo de MPI especifico a partir de los datos del individuo para su correcto envio y/o recepcion.

En el archivo ***create.c:***

```
void PGACreateIndividual (PGAContext *ctx, int p, int pop, int initflag)
       {
       …

       int i;
       for (i=0;i<numProcesadores;i++)
              ind->procesador[i] = 0;
       //meto un 1 en el lugar del procesador que lo creo
       ind->procesador[IdProceso] = 1;
       …
}
```

Cada vez que se crea un individuo en un procesador se debe almacenar en la variable procesador[] en que procesador se creo, escribiendo un uno en la posición correspondiente y dejando el resto a 0. Un individuo nacido en el procesador 0 tendra un 1 en la posición procesador[0] y 0 en el resto de posiciones.

Archivo ***cross.c***

```
void PGACrossover ( PGAContext *ctx, int p1, int p2, int pop1,
                    int c1, int c2, int pop2 )
      {
      …
   //la madre y el padre son p1 y p2 y los hijos son c1 y c2
   //realizo la actualizacion de la informacion de donde viene cada cruce
   int i;
   //consigo los individuos
   PGAIndividual *papa,*mama,*hijo1,*hijo2;
   papa = PGAGetIndividual(ctx, p1, pop1);
   mama = PGAGetIndividual(ctx, p2, pop1);
   hijo1 = PGAGetIndividual(ctx, c1, pop2);
   hijo2 = PGAGetIndividual(ctx, c2, pop2);

   for (i=0;i<numProcesadores;i++)
       {
       //miro los procesadores del padre
       if (papa->procesador[i] == 1)
              {
              // printf("****cruce****");
              hijo1->procesador[i]=1;
              hijo2->procesador[i]=1;
              }
       //miro los procesadores de la madre
       else if (mama->procesador[i] == 1)
              {
               //printf("****cruce****");
              hijo1->procesador[i]=1;
              hijo2->procesador[i]=1;
              }
       }
       …
}
```

Cuando se realiza el cruce entre dos individuos el hijo debe modificar su variable procesador[] para representar la herencia que ha recibido. Es decir, si considerasemos la variable procesador[] de los progenitores como un numero binario la variable procesador[] del hijo seria el resultado de realizar “OR” entre dichas variables.

6. Funciones de prueba utilizadas para el estudio:

6.1. Algoritmos defectivos

El término *deceptive* - que nosotros traduciremos como *defectivo* – fue introducido para probar las limitaciones de los AGs.

Aunque, como ya hemos visto, los AGs son algoritmos muy potentes que resuelven con éxito muchos problemas de optimización complejos, algunos otros resultan difíciles de resolver y se denominan problemas *GA-Hard*. Los problemas defectivos son una clase particular de estos problemas, y explotan la debilidad de la codificación de los cromosomas. Chow propone un método basado en algoritmos genéticos que resuelve los problemas GA-Hard, desarrollando un mapeo del genotipo al fenotipo que permite capturar también información de esquemas.

Teoría de los esquemas

Antes de definir los problemas defectivos, revisaremos la teoría básica de los esquemas a fin de poder entender mejor la naturaleza de estos problemas de optimización.

El concepto de esquemas intenta explicar la habilidad que tienen los Algoritmos Genéticos para desarrollar una búsqueda global en el espacio del problema, generalmente extenso y con gran número de dimensiones . Trabajamos con la hipótesis de la existencia de bloques de construcción (BBs, Building Blocks) en los cromosomas, a los que llamamos *esquemas.*

Un esquema es un patrón de cromosoma (de una parte del cromosoma) constituido por símbolos definidos (0 y 1) y por un símbolo comodín (*) que casa con 0 y con 1 indistintamente. Así, mientras que un cromosoma (por ejemplo el 10010011) representa un punto simple en el espacio de soluciones, un esquema del mismo número de bits representa un hiperplano en el espacio de soluciones. Por ejemplo, el esquema 10*1***1 (que casa con el cromosoma anterior) forma un hiperplano consistente en 16 puntos de datos, puesto que cada comodín * puede ser un 1 ó un 0 ($4_2 = 16$).

Se dice que un esquema *muestrea* su hiperplano asociado porque, cuando sobrevive de generación en generación, los puntos de datos de su hiperplano son probados repetidamente como posibles buenas soluciones. Así, a través de la recombinación, mutación y selección genéticas, los hiperplanos prometedores en el espacio de soluciones reciben proporciones de muestreo mayores, gracias a un incremento exponencial de las cantidades del esquema en la población de cromosomas.

Genotipo VS fenotipo

Un cromosoma binario puede consistir en fragmentos de secuencias binarias, cada uno de los cuales se llama *gen*, que corresponden a un parámetro de la función de optimización. Normalmente, se escoge un esquema particular de codificación de números binarios para trasladar cada secuencia binaria al correspondiente valor

numérico (u otro tipo de información) del parámetro. Por ejemplo, podemos usar el complemento a dos para traducir desde binario a entero.

El cromosoma binario representa el *genotipo,* mientras que la lista de parámetros decodificados representa el *fenotipo*.

6.1.1. Problemas de fondo

A continuación comentamos algunos de los problemas que subyacen al uso de este tipo de funciones de aptitud.

Métodos de codificación binaria

Plantearemos aquí el problema de la codificación cuando se traduce un parámetro del fenotipo a un número binario. El método de codificación más sencillo y usado es el complemento a dos. El problema de este tipo de codificaciones es que se pueden producir discontinuidades en el proceso de búsqueda, debido a que un paso simple en el espacio fenotípico (por ejemplo de 15 a 16) requeriría múltiples pasos en el espacio genotípico (5 pasos desde 00001111 hasta 00010000).

Se hicieron varios intentos, usando métodos de codificación alternativos como por ejemplo el código Gray, para solucionar este problema (,). La solución parecía pasar por emplear códigos tales que la distancia Hamming entre dos números (valores del fenotipo) consecutivos fuera uno.

Sin embargo, incluso usando códigos como el Gray, un espacio de problema puede seguir resultando complejo para la búsqueda que realizan los Algoritmos Genéticos.

Problemas defectivos

Como dijimos anteriormente, algunos de los problemas que resultan *duros* para los Algoritmos Genéticos son los llamados problemas defectivos (*deceptive problems*).

En primer lugar veremos qué tipos de esquemas pueden llevar a engaño. Ya sabemos que el fitness de un esquema determinado depende de los valores de fitness de sus puntos muestreados (los que forman su hiperplano). Y, por la naturaleza de los Algoritmos Genéticos, los esquemas con altos valores de fitness tienen mayor probabilidad de sobrevivir y pasar a la siguiente generación. A continuación daremos algunas definiciones útiles para el posterior planteamiento de los problemas defectivos:

- Un esquema con n bits definidos (1s ó 0s) se dice un esquema ***de orden n***. Por ejemplo, los esquemas *0**0, *0**1, *1**0 y *1**1 son todos esquemas de orden dos.

- Los esquemas con igual orden que, además, tienen definidos los bits de las mismas posiciones son ***competidores primarios*** en el proceso de selección, puesto que dichos esquemas compiten como planos ortogonales en el hiperespacio. Un hiperplano de orden inferior también

contiene una colección de hiperplanos de orden más alto, que comparten bits definidos con el hiperplano de orden inferior.

- Por ejemplo, el hiperplano de orden 2 0***0 contiene a su vez hiperplanos de orden 3 tales como 00**0 y 01**0 (entre otros), mientras que el esquema 1***0 contiene otro conjunto de hiperplanos de orden 3 tales como 10**0 y 11**0 (entre otros).

- Un esquema de orden n y los esquemas de orden k (donde n < k) que contiene se dicen ***relevantes*** uno respecto del otro. Así, cuando el esquema de menor orden compite en un proceso de selección, todos sus esquemas relevantes de nivel superior también participan en la competición. Por ejemplo, la competición entre los esquemas de orden 2 0***0, 0***1, 1***0 y 1***1 también implica las competiciones entre los esquemas de orden 3 00**0, 01**0, 00**1, 01**1, 10**0, 11**0, 10**1 y 11**1.

- Recíprocamente, las competiciones entre los hiperplanos de orden superior también implican la competición de sus hiperplanos relevantes de orden inferior.

Una vez expuestos estos conceptos, podemos definir cuándo se produce una situación defectiva: cuando la competición entre hiperplanos de un nivel superior k se encamina a una solución global que es radicalmente diferente, en términos de distancia de Hamming, de la solución global de las competiciones entre los hiperplanos relevantes de orden n, donde n < k.

Por ejemplo, sean los siguientes valores de fitness para la función defectiva f1:

F1(000)	28	F1(001)	26
F1(010)	22	F1(100)	14
F1(110)	0	F1(011)	0
F1(101)	0	F1(111)	30

Con la función f1, la solución global de una competición de hiperplanos de orden 3 es 111 (fitness 30).

Ahora consideremos el hiperplano relevante de 111 representado por *11. Cuando compite con *00, *01 y *10 en competiciones de orden 2, el fitness promedio de *11 es 15 ((0 + 30) / 2), en contraposición a los valores promedio de *00 ((28 + 14) / 2 = 21), de *01 ((26 + 0) / 2 = 13) y de *10 ((22 + 0) / 2 = 11). En función de estos valores, el esquema *00 parece el que más probablemente ganará la competición de hiperplanos de orden 2.

La diferencia, en distancia de Hamming, entre las dos soluciones globales 111 y *00 provoca una situación defectiva (*deception*).

Una vez vistos los conceptos anteriores, podemos dar al fin una definición de problema defectivo (*deceptive problem*):

Un problema defectivo desvía a los Algoritmos Genéticos a converger incorrectamente a una región en el espacio del problema llamado atractor defectivo (óptimo local).

6.2. Funciones Multimodales

Buscan un óptimo (máximo ó mínimo) global en una función con muchos óptimos locales, aunque sin llegar a ser funciones defectivas.

Sus aptitudes vienen descritas típicamente por funciones trigonométricas.

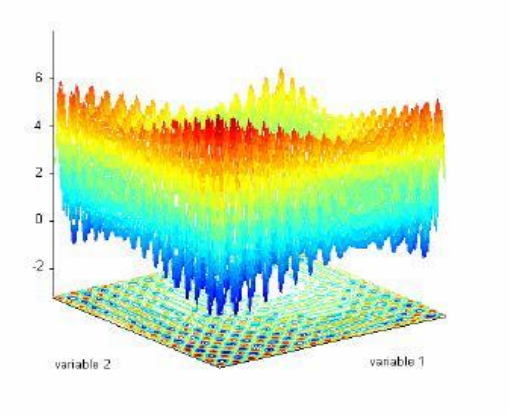
6.2.1. Fmodal

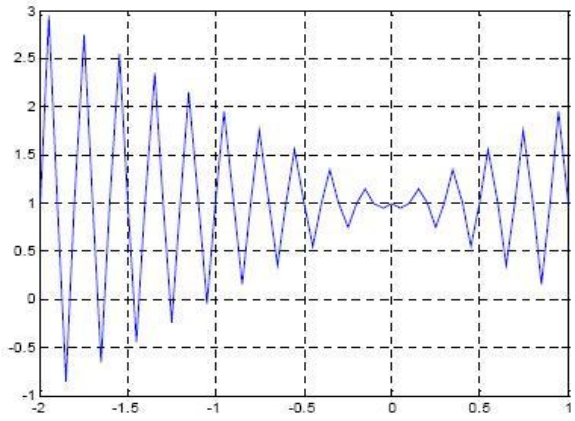
Se trata de una función multimodal. Al igual que en la función anterior evaluamos la función para cada tramo de cromosoma de longitud total partido de 10.

La fórmula utilizada para su codificación es la siguiente:

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

Las gráficas correspondientes a la función FModal son:





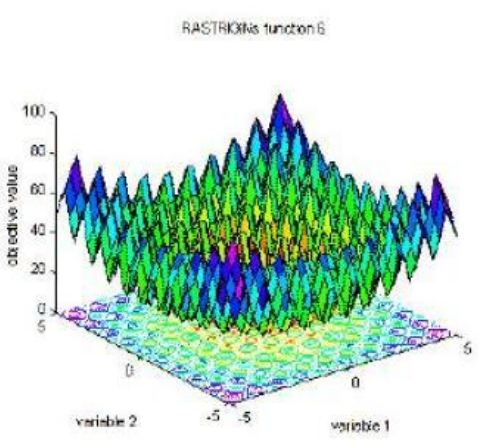
6.2.2. Rastrigin

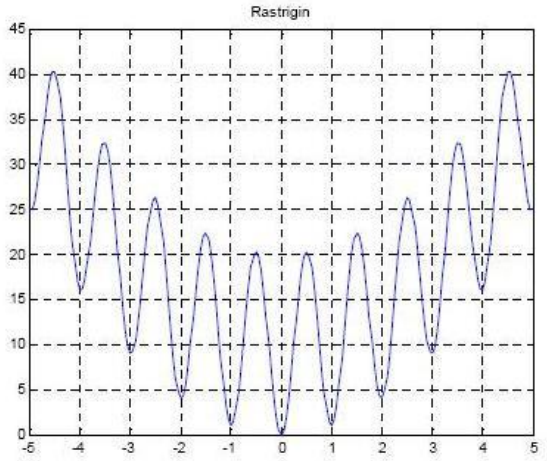
Se trata de una función multimodal. Al igual que en la función anterior evaluamos la función para cada tramo de cromosoma de longitud total partido de 10.

La fórmula utilizada para la codificación es:

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

Las gráficas que se obtienen de esta función son:





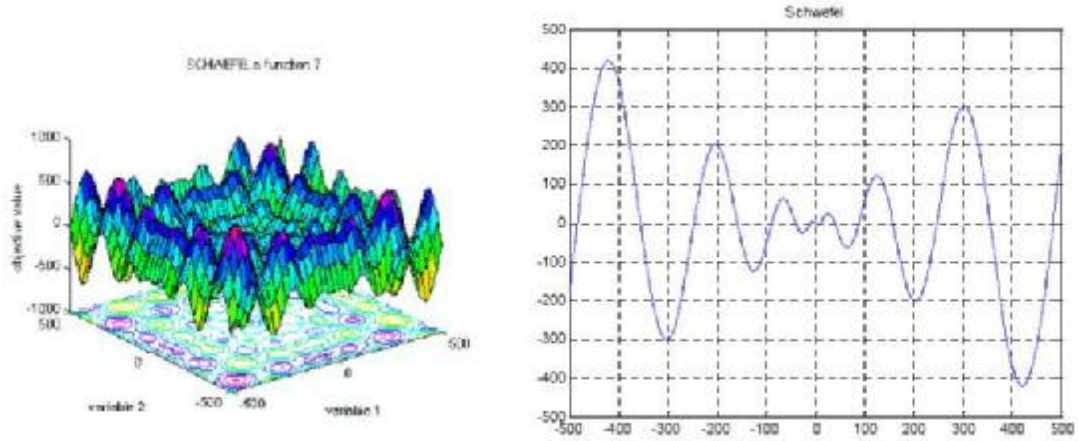
6.2.3. Schwefel

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

La fórmula que nos ha servido para implementar el código es la siguiente:

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

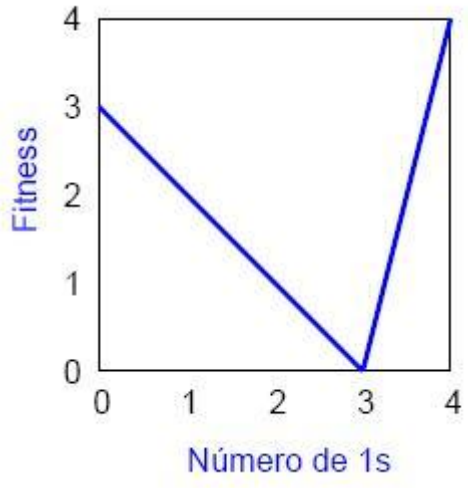
Y la gráfica obtenida es:



6.3. Ftrap (trap function): funciones trampa

Un ejemplo de las funciones defectivas son las llamadas funciones *trampa* (*trap functions*).

Por ejemplo, La función mostrada en la figura es una función trampa, puesto que hay una tendencia hacia la solución 0000 (con fitness 3), que está máximamente alejada de la mejor solución 1111 (con fitness 4).



El problema se desvía porque los pequeños cambios incrementales en la solución son recompensados, dirigiendo erróneamente la búsqueda hacia el atractor defectivo.

Para definir formalmente una función trampa, introduciremos primero la definición de otra función auxiliar. Sea $\mathbf{x} = (x_1,...,x_l)$ una cadena binaria de longitud l. La función u($\mathbf{x}$) de $\mathbf{x}$ se define como:

$$u(x)= u(x_1, \ldots, x_l) = x_1+ \ldots + x_l = \sum_{i=1}^{l} x_i$$

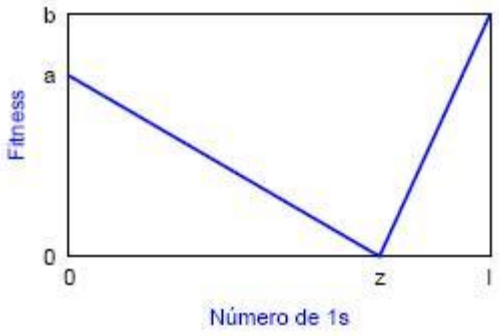
(ver la función One Max)

Una función trampa f(**x**) se define, basándose en u(**x**), del siguiente modo:

$$f(\mathbf{x}) = F(u(\mathbf{x})) = \begin{cases} \frac{a}{z}(z - u(\mathbf{x})), \text{ si } u(\mathbf{x}) \leq z \\ \frac{b}{1-z}(u(\mathbf{x}) - z), \text{ eoc} \end{cases}$$

donde *a* es el óptimo local (posiblemente defectivo), *b* es el óptimo global, y *z* es la posición del cambio de inclinación que separa la cuenca de atracción de los dos óptimos. Los valores de los parámetros *a*, *b* y *z* de la función trampa determinan el grado de dificultad que supondrá para el AG encontrar el óptimo global *b* en contraposición al óptimo local *a*.

La forma general de una función trampa de l bits es:



En nuestra implementación los valores tomados son:

a=400
b=512
z=400

por lo que el máximo local estaría en x=0, f(x) = 400, y el máximo global en x=512, f(x)= 512.

6.4. OneMax

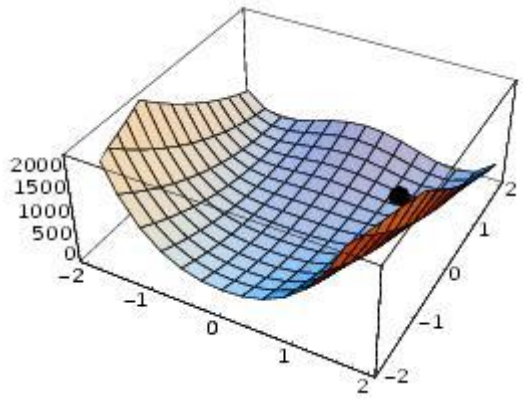
Resuelve el problema de obtener la cadena binaria con mayor número de 1s posible. Para ello define como función de aptitud:

$$F_{OneMax}(X) = \sum_{i=1}^{n} x_i$$

siendo X el vector de genes (cromosoma), n la longitud del cromosoma y x_i el gen iésimo (con valor 0 ó 1).

En nuestra implementación hemos trabajado siempre (para One Max) con individuos de longitud 512, por lo que el valor máximo que puede alcanzar la función es 512.

6.5. Rosenbrock



La función utilizada es la siguiente:

$$F=\sum_{i=1}^{n-1} 100\,(x_{i+1} - x_i^2)^2 + (1- x_i)^2$$

Se utiliza para problemas de prueba de optimización de algoritmos (donde una variación con 100 es reemplazada por 105 a veces; Germundsson 2000). Es una función unimodal con un mínimo global de 0 en el punto (1,1).

La función de Rosenbrock es un algoritmo de búsqueda de orden 0 (no necesita derivar la función objetivo). Aún así, se aproxima a una búsqueda de gradiente combinando las ventajas de estrategias de orden 0 y orden 1. Fue publicada por Rosenbrock en los 70.

En la primera iteración, es una búsqueda simple de orden 0 en las direcciones de los vectores de la base de un sistema de coordenadas n-dimensional. En caso de éxito, el ancho de búsqueda se incrementa, mientras que si hay un fallo se decrementa y se prueba en dirección contraria.

Una vez que se encuentra algún éxito y se ha explorado en todas las direcciones de la base, el sistemas de coordenadas se rota para tomar el primer punto base del vector en dirección. Deespués se inicializan todos los anchos de búsqueda y se repite el proceso con el sistema de coordenadas rotado.

7. Resultados experimentales

7.1. Introducción

En nuestra implementación hemos utilizado **elitismo**:

El modelo elitista consiste en guardar siempre el mejor individuo de la población para la siguiente generación, normalmente sustituyéndolo por el peor. Hay estudios que indican que un algoritmo con selección elitista asegura la convergencia del Algoritmo Genético hacia un óptimo global.

Para conseguir resultados diferentes en cada ejecución de un experimento necesitábamos inicializar la **generación de la secuencia de números pseudos-aleatorios** con un número verdaderamente aleatorio. Para ello utilizamos el generador de números aleatorio /dev/random del sistema operativo.

Este generador de números aleatorios recoge el ruido ambiental procedente de los manejadores de dispositivo y otras fuentes y lo guarda en un "almacén de entropía". El generador también mantiene una estimación del número de bits de ruido en el almacén de entropía. Los números aleatorios se crean a partir de este almacén.

El encargado de leer el fichero es únicamente el procesador 0 para evitar esperas innecesarias a que el sistema operativo de permiso de lectura sobre este archivo. De este modo conseguimos mayor velocidad de ejecución.

El procesador lee el archivo random lo utiliza como semilla y comienza una generación de números pseudo aleatorios que ira pasando al resto de procesadores.

Para el intercambio de individuos entre procesadores hemos utilizado una **topología en anillo**, ya que nos resultaba la más adecuada para el tipo de paralelismo (de grano grueso) que utilizamos.

Para mostrar valores más cercanos a los resultados, y no depender demasiado de la aleatoriedad de las poblaciones iniciales, cada experimento se ha realizado con un total de **100 repeticiones**, haciendo la media aritmética después de cada valor obtenido.

El **número de procesadores** utilizados ha variado desde 1 procesador hasta 8 procesadores. Para poder comprobar la tolerancia a fallos, hemos incomunicado algunos procesadores durante la ejecución, bloqueando hasta la mitad de los procesadores que se están utilizando en cada prueba. El número de generación en el que procedemos a incomunicar cada procesador también ha sido variable. Para poder realizar comparativas eficientes, hemos ejecutado cada experimento con las diferentes opciones de número de procesadores activos sin incomunicar ninguno.

La **frecuencia de intercambio** de individuos también ha variado desde el intercambio cada 10, cada 25, cada 50 y cada 100 generaciones, comprobando los resultados obtenidos para cada uno de los experimentos.

Utilizamos dos opciones diferentes para la **probabilidad de cruce y de mutación**:

- Probabilidad de cruce=1, probabilidad de mutación=0. Con esto queremos comprobar cómo influyen el resto de los factores en caso de que no exista mutación.
- Probabilidad de cruce=0.8, probabilidad de mutación=0.002.

Los resultados obtenidos los hemos analizado guardando los siguientes ficheros para cada experimento:

- fichero en el que se guardara la informacion para relizar las graficas, guardamos los mejores valores.
- fichero en el que se guardara la informacion para relizar las graficas, guardamos los peores valores.
- fichero en el que se guardara la informacion para relizar las graficas, guardamos los medios valores.
- fichero en el que se guardara la informacion para relizar las graficas, guardamos los tiempos de ejecucion.
- fichero en el que se guardara la informacion para relizar las graficas, guardamos los porcentajes de herencia.

Para todos los experimentos realizados se han escogido las siguientes opciones del PGAPack:

Inicialización de las poblaciones: los individuos son inicializados de forma aleatoria.

Criterio de parada: la parada se produce cuando la solución ha convergido. Si la solución no converge, como mucho ejecutará hasta hasta 5000 generaciones.

Tipo de selección: PGA_SELECT_TOURNAMENT (selección por torneo), valor por defecto del PGAPack.

Tipo de cruce: CROSSOVER_TWOPT, valor por defecto del PGAPack.

Tipo de fitness: FITNESS_RAW, valor por defecto del PGAPack.

Tipo de reemplazamiento: POPREPL_BEST, valor por defecto del PGAPack.

En el reemplazamiento sólo intercambiamos el mejor individuo de la población.

7.2. Función One Max:

A continuación detallaremos un pequeño índice en el que se comentaran las pruebas realizadas para este experimento:

- **Fallo cada 50 generaciones:** hemos provocado el fallo de un procesador cada 50 generaciones. Más propiamente dicho, incomunicación ya que ese procesador sigue ejecutándose aisladamente y seguimos recopilando datos de su ejecución.

  Se ha repetido el experimento para 8 procesadores, 4 procesadores y 2 procesadores con y sin mutación.

  - *Con mutación:* Probabilidad de cruce 0,8. Probabilidad de mutación *0,002.*

  - *Sin mutación:* Probabilidad de cruce 1. Probabilidad de mutación 0.

  Para todos los casos se han considerado los periodos de intercambio cada 10, 25, 50 y 100 generaciones.

  Así mismo, todas estas pruebas se han repetido con las mismas condiciones sin forzar ningún fallo en los procesadores.

  Se puede observar los resultados en forma de gráfica en las FIGURA 1 y 2, así como un resumen de los resultados en la TABLA1.

- **Fallo cada 100 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones, pero esta vez se han provocado los fallos cada 100 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 3 y 4, así como un resumen de los resultados en la TABLA 2.

- **Fallo cada 200 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones pero esta vez se han provocado los fallos cada 200 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 5 y 6, así como un resumen de los resultados en la TABLA 3.

Para una mejor comprensión de las tablas procederemos a continuación a la explicación de cada una de las columnas:

- **Experimento:** resumen del experimento al que se refieren los resultados.

- **Intercambio:** periodo de intercambio utilizado en ese experimento.

- **Herencia de todos:** en esta columna se observará si todos los procesadores que quedan "vivos" al final de la ejecución han conseguido recibir herencia de todos los procesadores con los que se inicio la ejecución.

- **Separan en generación:** en esta columna se muestra a partir de qué generación las líneas de la gráfica correspondientes al experimento con fallos y aquéllas correspondientes al experimento sin fallos comienzan a separarse indicando tener algún problema por la "muerte" de algún procesador.

- **Generacion:** número de generaciones que ha ejecutado el experimento.

- **Proc Caidos:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos.

- **%Pob Muerta:** mostramos el porcentaje de población muerte en el momento en el que se empiezan a separar las líneas.

- **Proc Caidos Total:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos al finalizar la ejecución.

- **%Pob Muerta Total:** mostramos el porcentaje de población muerta en el momento en el que se empiezan a separar la líneas al finalizar la ejecución

- **Max Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

- **Max Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

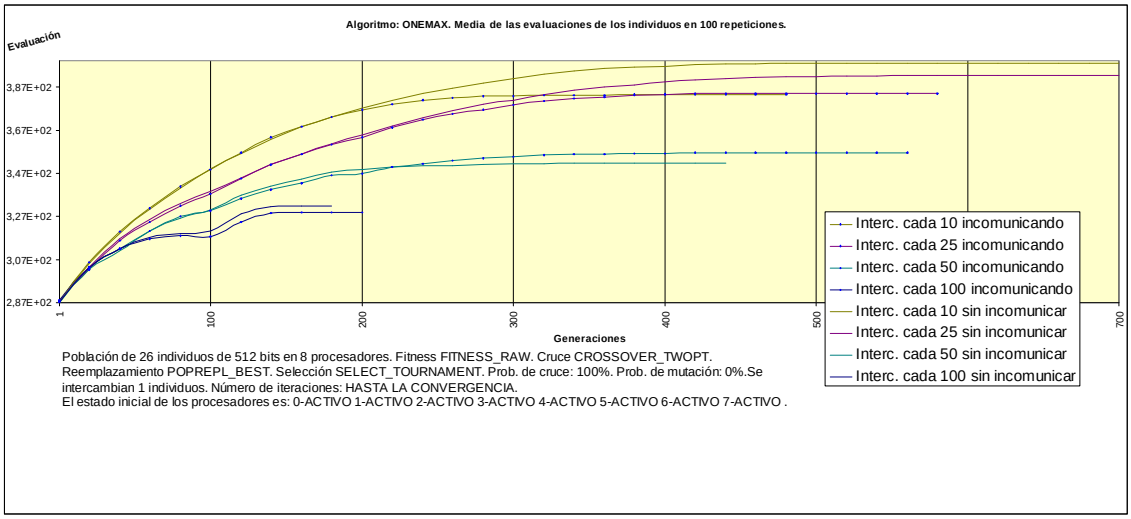
- **Tolerancia (% error):** diferencia entre el valor alcanzado con fallos en los procesadores y la ejecución sin fallos. Expresada en porcentaje.

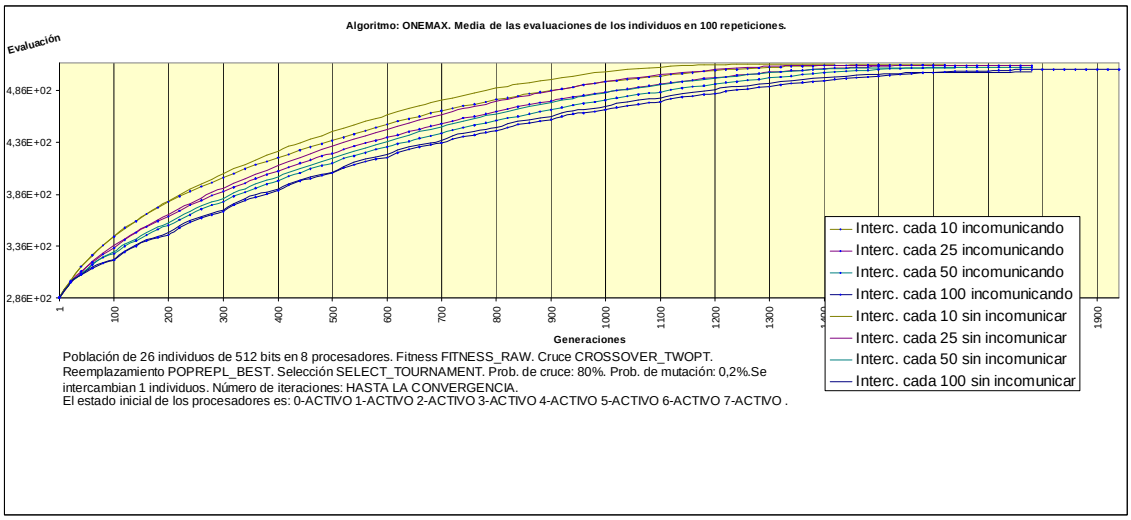
- **Min Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

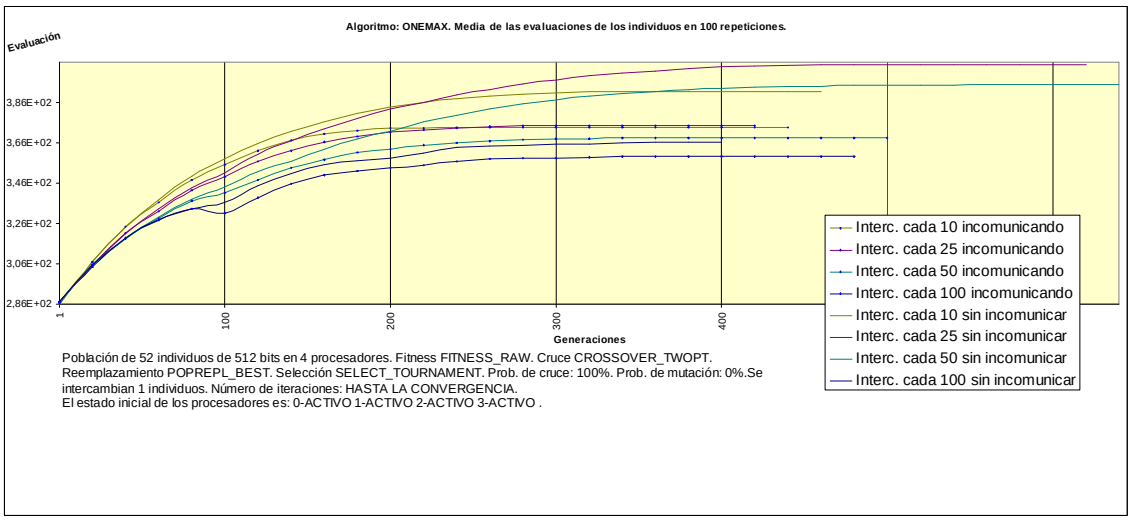
- **Min Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

A continuación de cada tabla se muestra una gráfica en la que se indica la evolución del porcentaje de tolerancia según el periodo de intercambio y si el experimento fue realizado con mutación (cm) o sin mutación (sm).

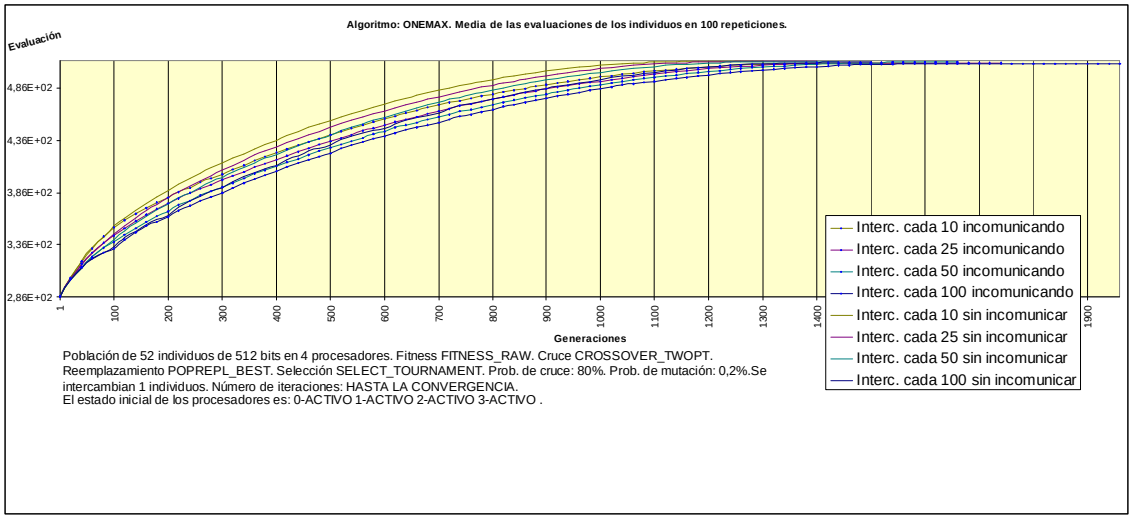
7.2.1. Fallo cada 50 generaciones

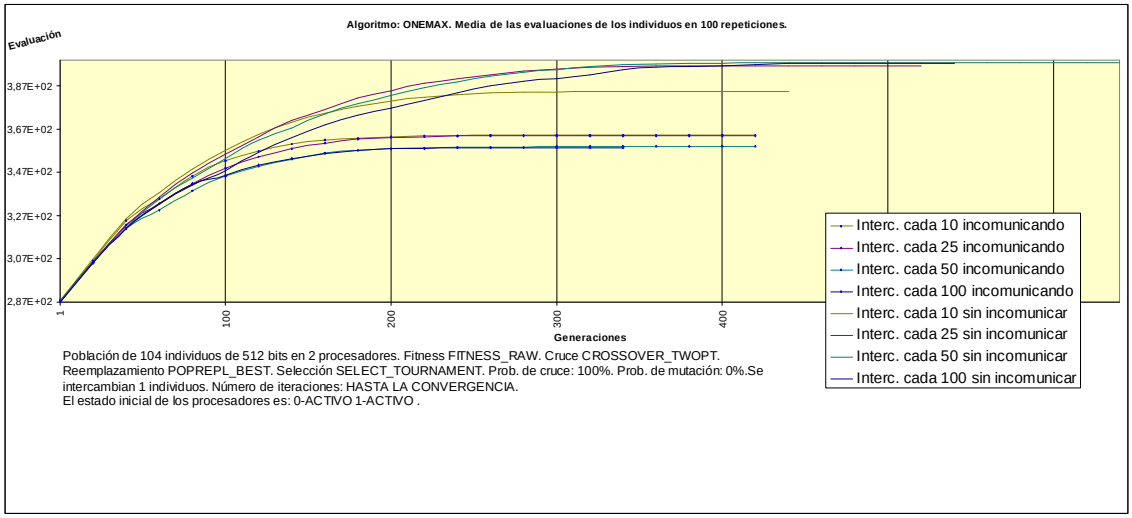


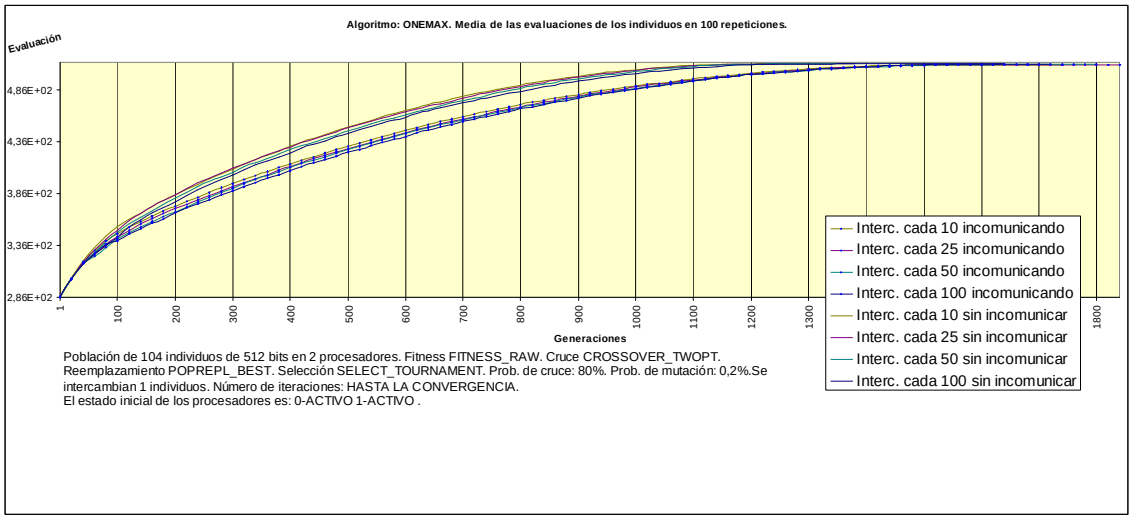




ONEMAX. .FIGURA 1 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 50 generaciones falla un procesador



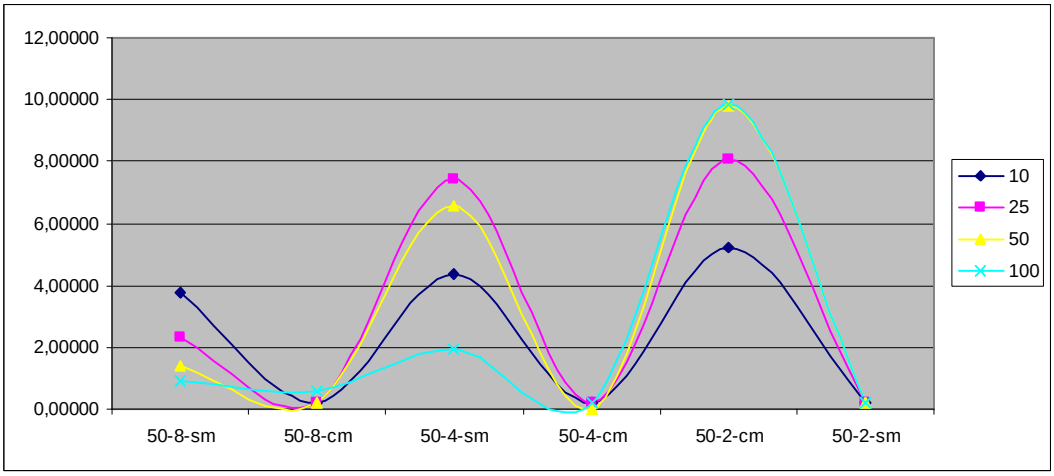




ONEMAX. FIGURA 2 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 50 generaciones falla un procesador

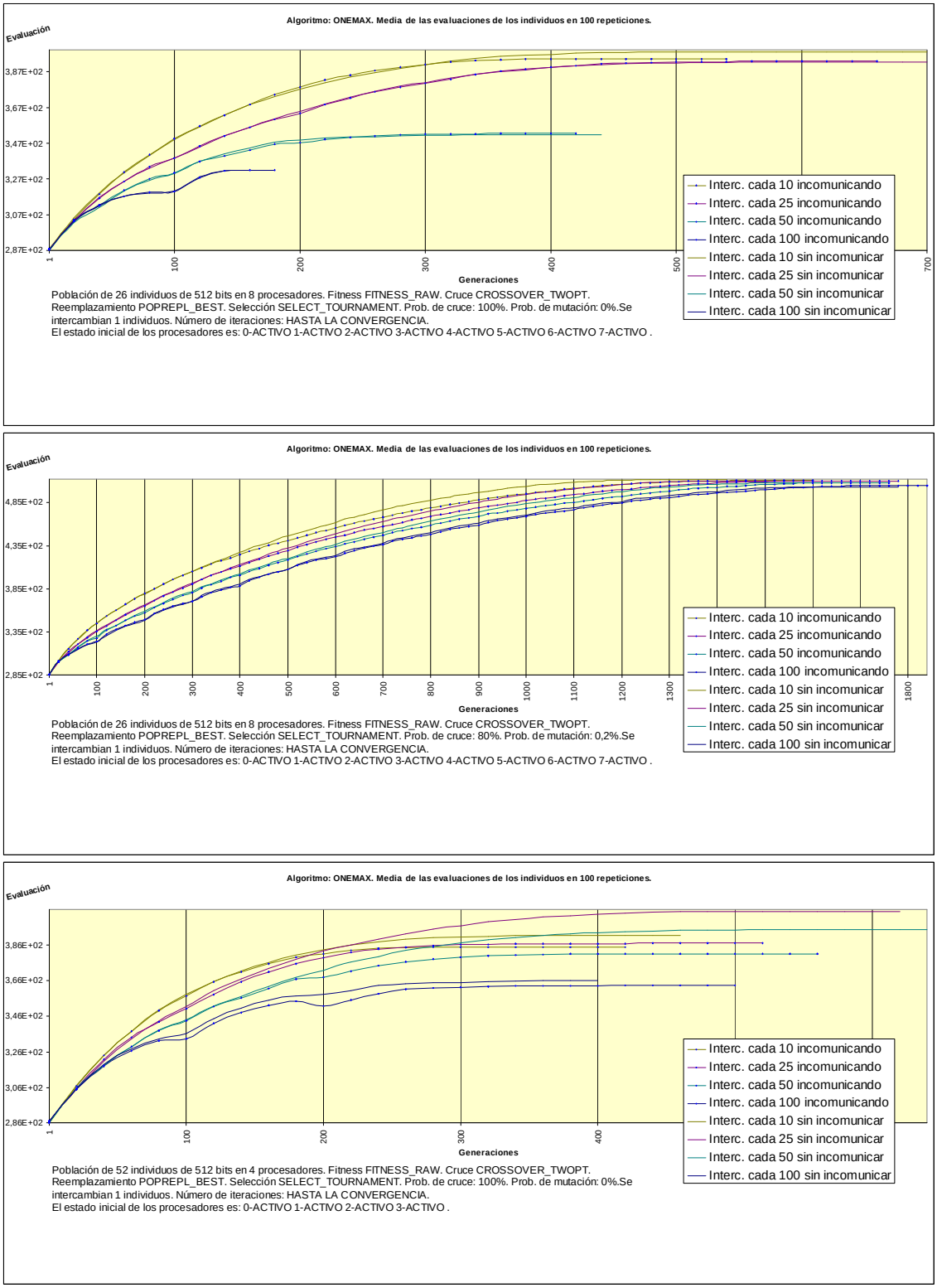
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
Onemax 50 8 sin mutación	10	si	200	480	4	50,0	4	50,0	383	398	3,768844	383	398
	25	si	250	580	4	50,0	4	50,0	384	393	2,290076	382	380
	50	no	220	560	4	50,0	4	50,0	357	352	1,420455	339	326
	100	no	100	200	2	25,0	4	50,0	329	332	0,903614	306	305
OneMax 50 8 con mutación	10	si	200	1880	4	50,0	4	50,0	511	512	0,195313	510	510
	25	si	150	1880	3	37,5	4	50,0	511	510	0,196078	508	506
	50	no	150	1880	3	37,5	4	50,0	509	508	0,19685	506	502
	100	no	150	1880	3	37,5	4	50,0	507	504	0,595238	502	496
OneMax 50 4 sin mutación	10	si	50	440	1	25,0	2	50,0	374	391	4,347826	374	391
	25	si	50	480	1	25,0	2	50,0	375	405	7,407407	375	404
	50	no	80	500	1	25,0	2	50,0	369	395	6,582278	369	384
	100	no	80	480	1	25,0	2	50,0	359	366	1,912568	352	341
OneMax 50 4 con mutación	10	si	80	1850	1	25,0	2	50,0	511	512	0,195313	510	510
	25	si	80	1850	1	25,0	2	50,0	511	512	0,195313	509	509
	50	no	80	1850	1	25,0	2	50,0	511	511	0	509	508
	100	no	80	200	1	25,0	2	50,0	510	511	0,195695	508	506
OneMax 50 2 sin mutación	10	si	50	420	1	50,0	1	50,0	364	384	5,208333	364	384
	25	si	50	420	1	50,0	1	50,0	364	396	8,080808	364	396
	50	no	50	420	1	50,0	1	50,0	359	398	9,798995	359	398
	100	no	50	340	1	50,0	1	50,0	358	397	9,823678	358	394
OneMax 50 2 con mutación	10	si	50	1720	1	50,0	1	50,0	511	512	0,195313	509	510
	25	si	50	1720	1	50,0	1	50,0	511	512	0,195313	509	510
	50	no	50	1850	1	50,0	1	50,0	511	512	0,195313	509	509
	100	no	50	1720	1	50,0	1	50,0	511	512	0,195313	509	509

de 0,5 % a 1% de 0% a 0,5%

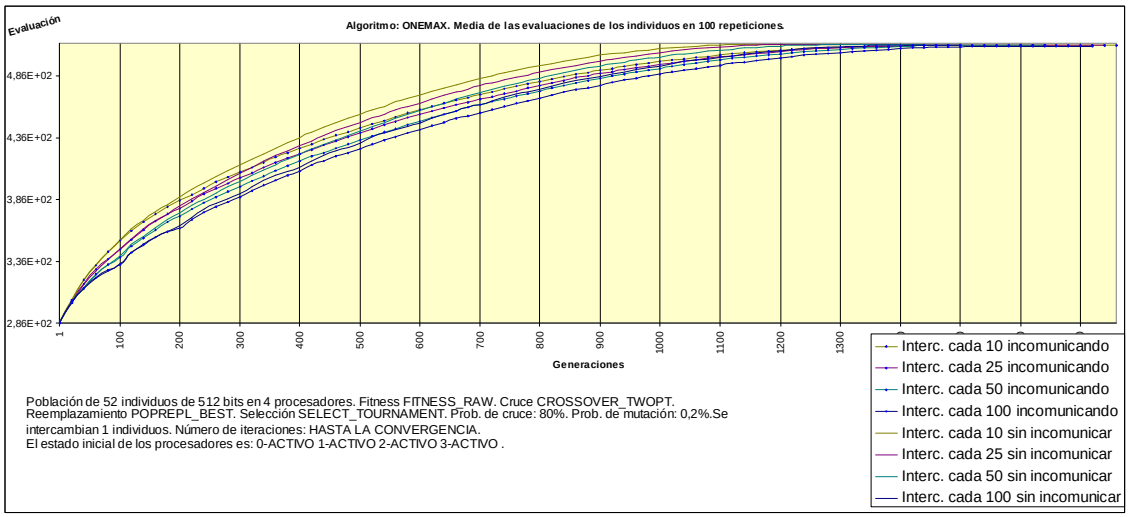


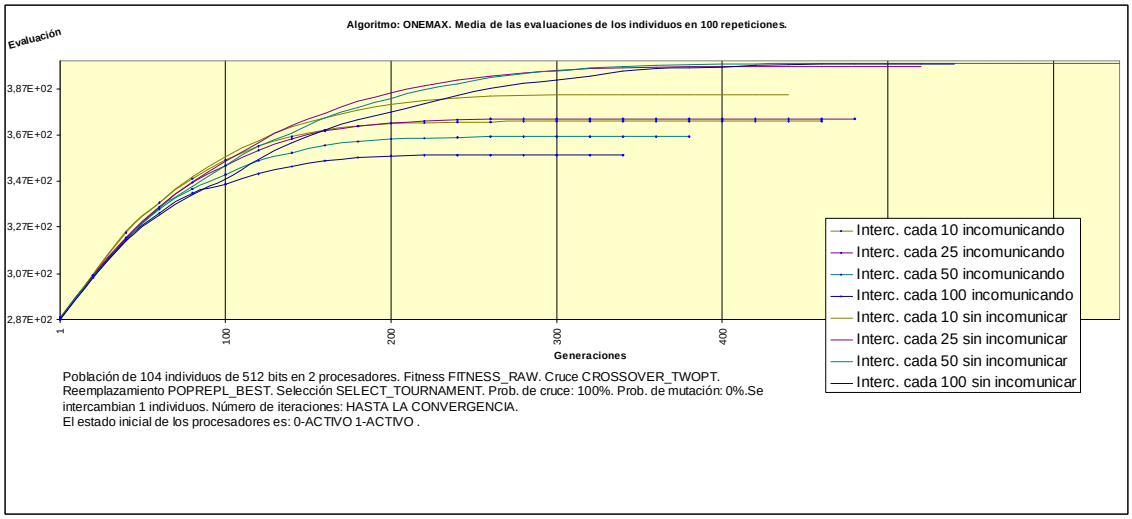
ONEMAX. TABLA 1. Resumen de resultados incomunicando cada 50 generaciones

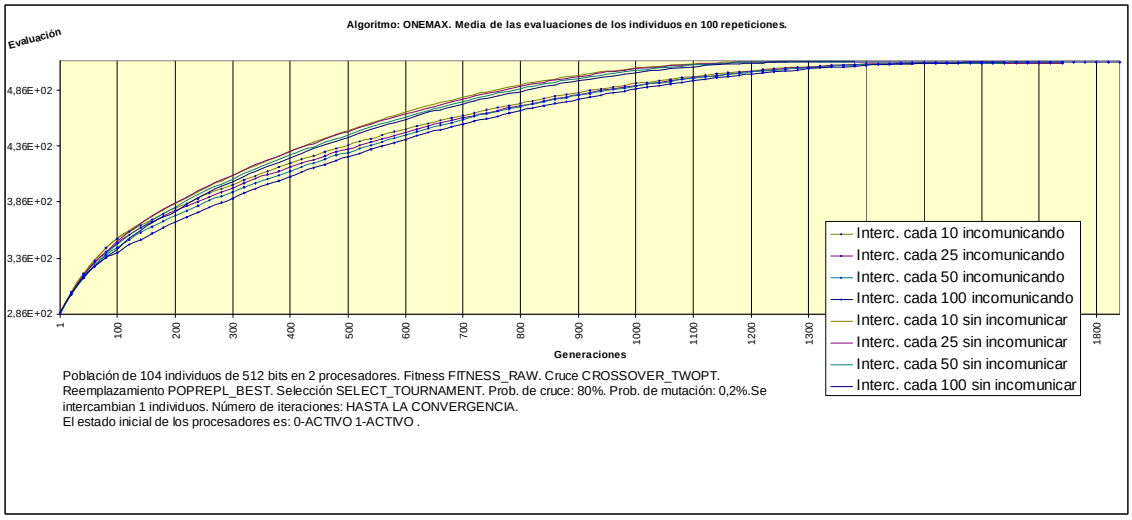
7.2.2. Fallo cada 100 generaciones



ONEMAX. .FIGURA 3 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 100 generaciones falla un procesador



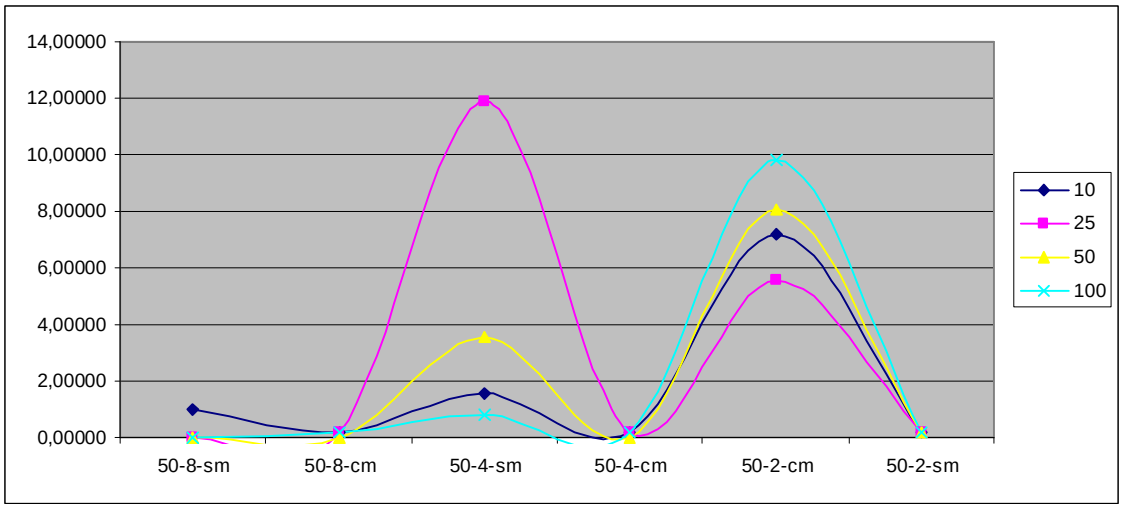




ONEMAX. FIGURA 4 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 100 generaciones falla un procesador

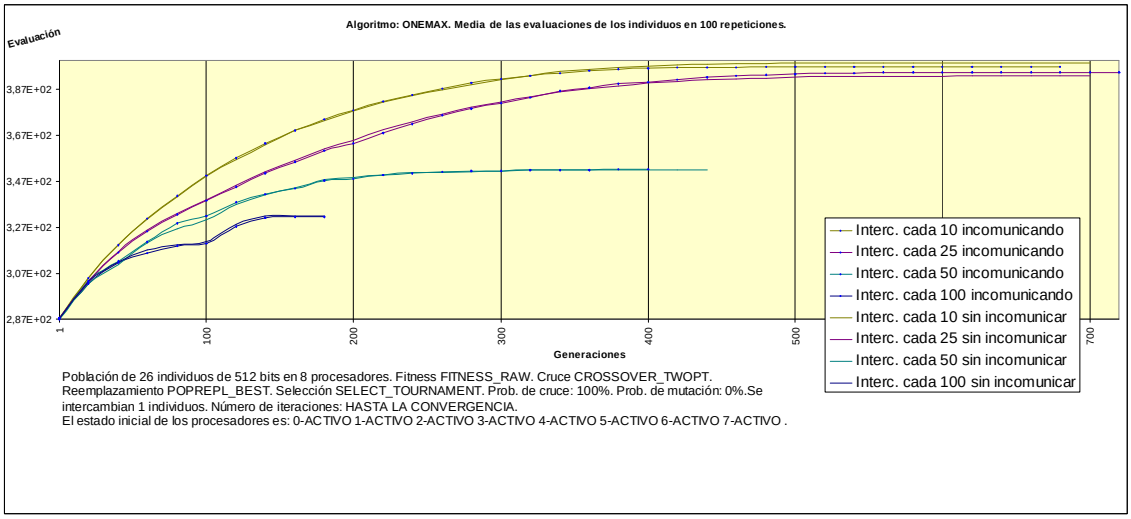
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
Onemax 100 8 sin mutación	10	si	350	550	3	37,5	4	50,0	394	398	1,005025	394	398
	25	si		650			4	50,0	393	393	0	388	380
	50	si		450			4	50,0	352	352	0	330	326
	100	no		180			1	12,5	332	332	0	305	305
OneMax 100 8 con mutación	10	si	350	1800	3	37,5	4	50,0	511	512	0,195313	510	510
	25	si	350	1800	3	37,5	4	50,0	511	510	0,196078	508	506
	50	si	350	1800	3	37,5	4	50,0	508	508	0	505	502
	100	no	650	1800	4	50,0	4	50,0	505	504	0,198413	501	496
OneMax 100 4 sin mutación	10	si	150	450	1	25,0	2	50,0	385	391	1,534527	385	391
	25	si	100	520	1	25,0	2	50,0	357	405	11,85185	387	404
	50	si	100	550	1	25,0	2	50,0	381	395	3,544304	381	384
	100	no	100	500	1	25,0	2	50,0	363	366	0,819672	350	341
OneMax 100 4 con mutación	10	si	100	1800	1	25,0	2	50,0	511	512	0,195313	510	510
	25	si	100	1800	1	25,0	2	50,0	511	512	0,195313	509	509
	50	si	100	1800	1	25,0	2	50,0	511	511	0	509	508
	100	no	100	1800	1	25,0	2	50,0	510	511	0,195695	508	506
OneMax 100 2 sin mutación	10	si	100	450	1	50,0	1	50,0	373	348	7,183908	373	384
	25	si	100	480	1	50,0	1	50,0	374	396	5,555556	374	396
	50	si	100	480	1	50,0	1	50,0	366	398	8,040201	366	398
	100	no	100	350	1	50,0	1	50,0	358	397	9,823678	358	394
OneMax 100 2 con mutación	10	si	100	1850	1	50,0	1	50,0	511	512	0,195313	509	510
	25	si	100	1850	1	50,0	1	50,0	511	512	0,195313	509	510
	50	si	100	1850	1	50,0	1	50,0	511	512	0,195313	509	509
	100	no	100	1850	1	50,0	1	50,0	511	512	0,195313	509	509

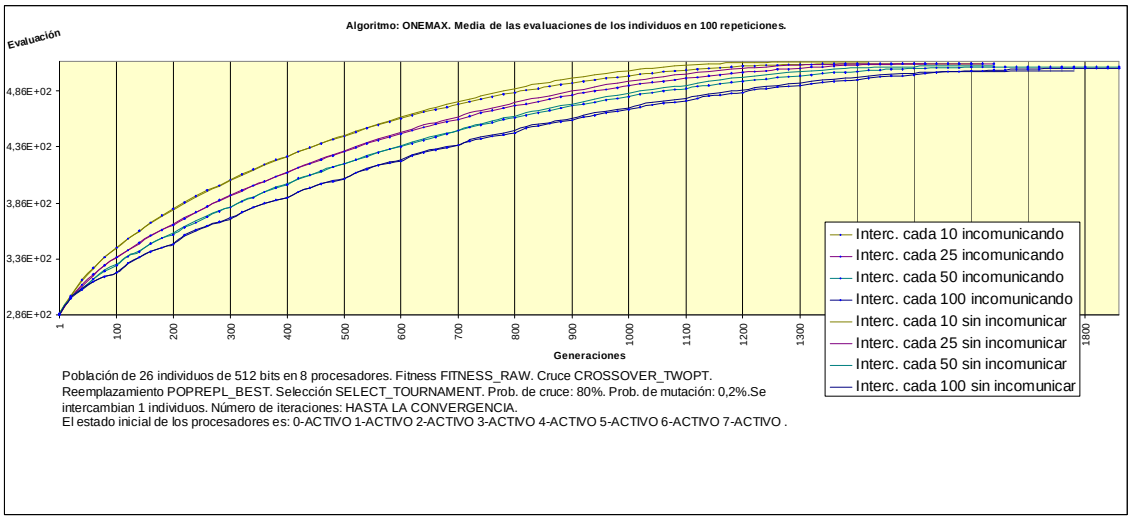
de 0,5 % a 1% de 0% a 0,5%

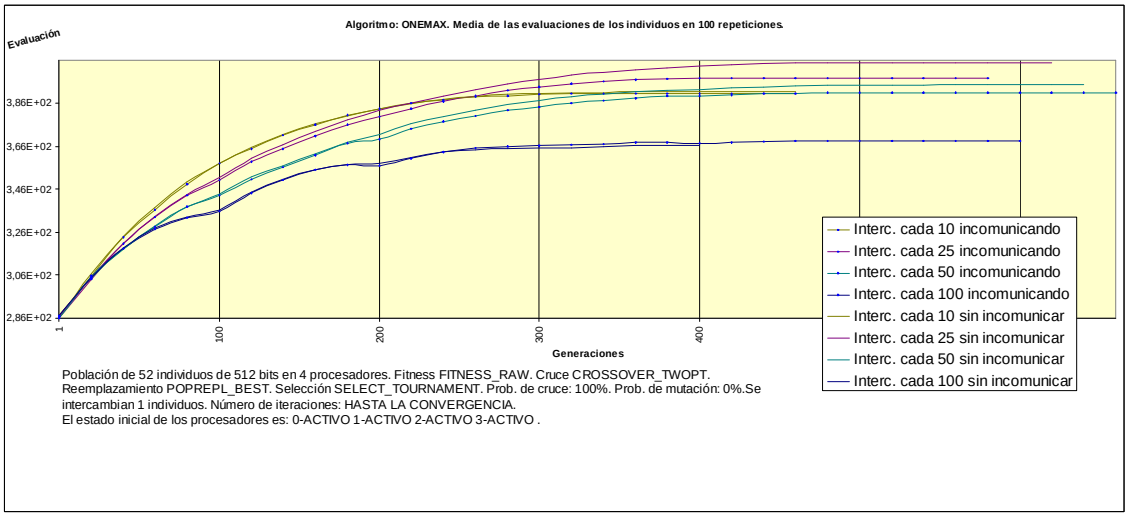


ONEMAX. TABLA 2. Resumen de resultados incomunicando cada 100 generaciones

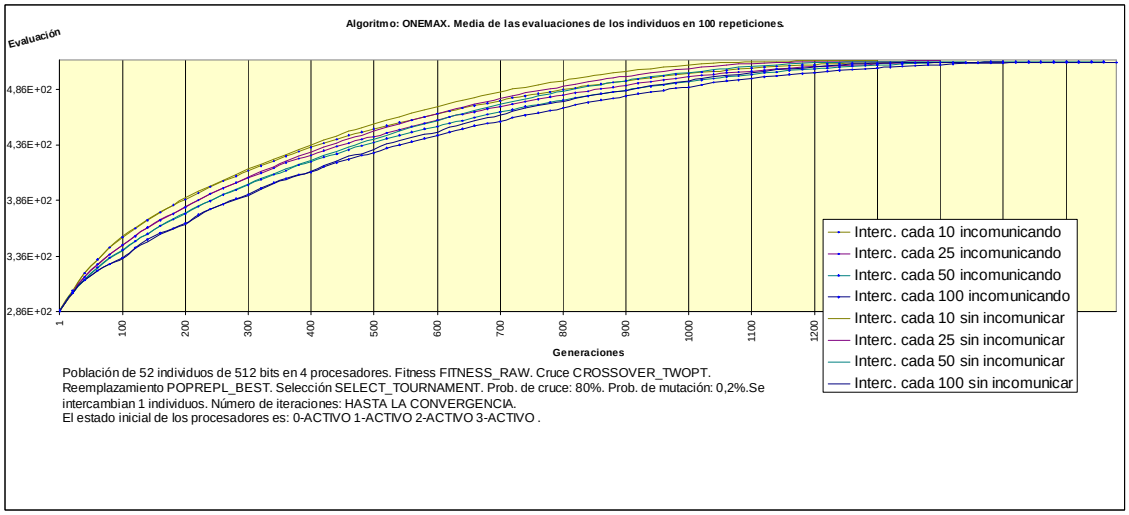
7.2.3. Fallo cada 200 generaciones

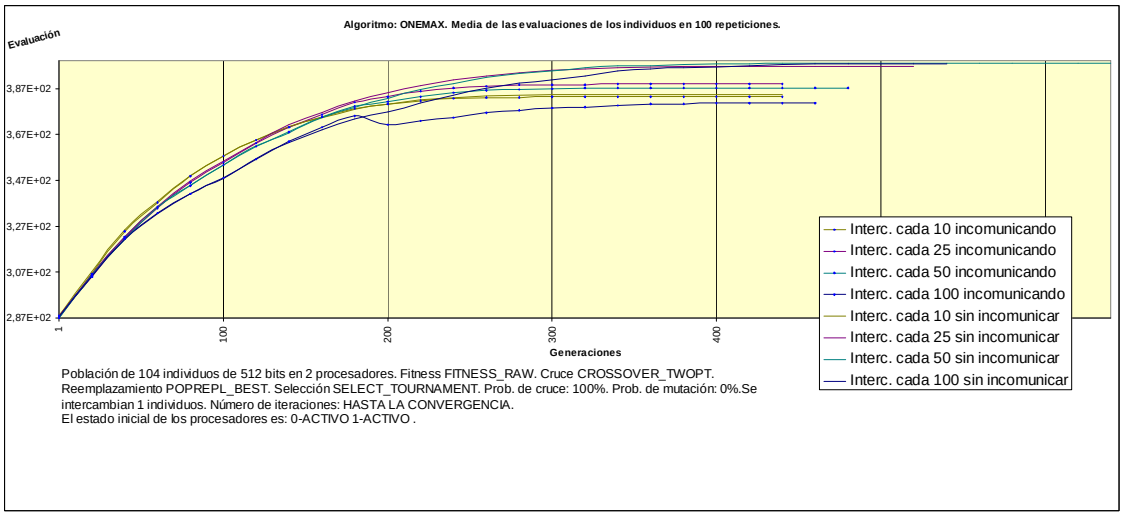


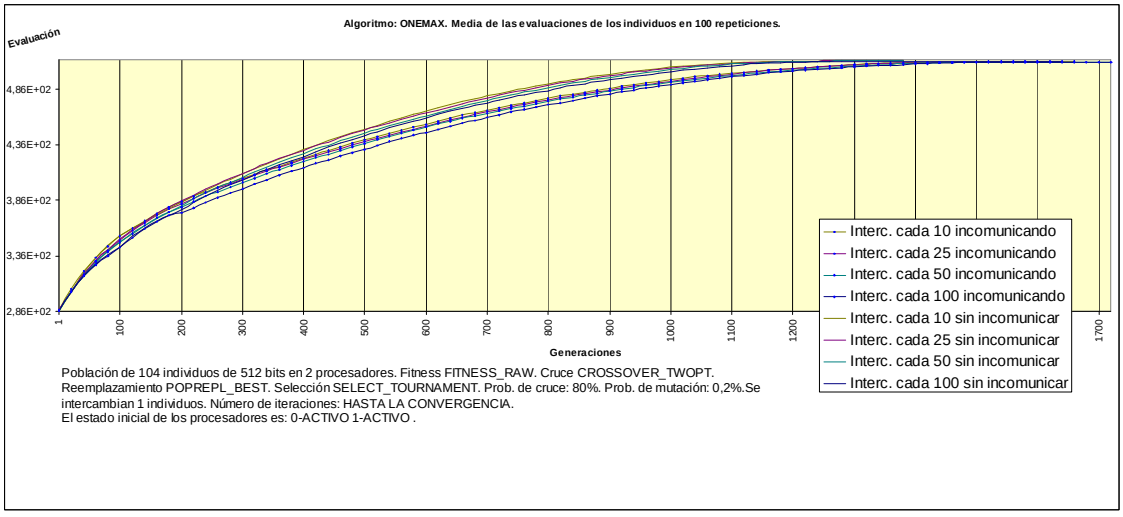




ONEMAX. .FIGURA 5 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 200 generaciones falla un procesador



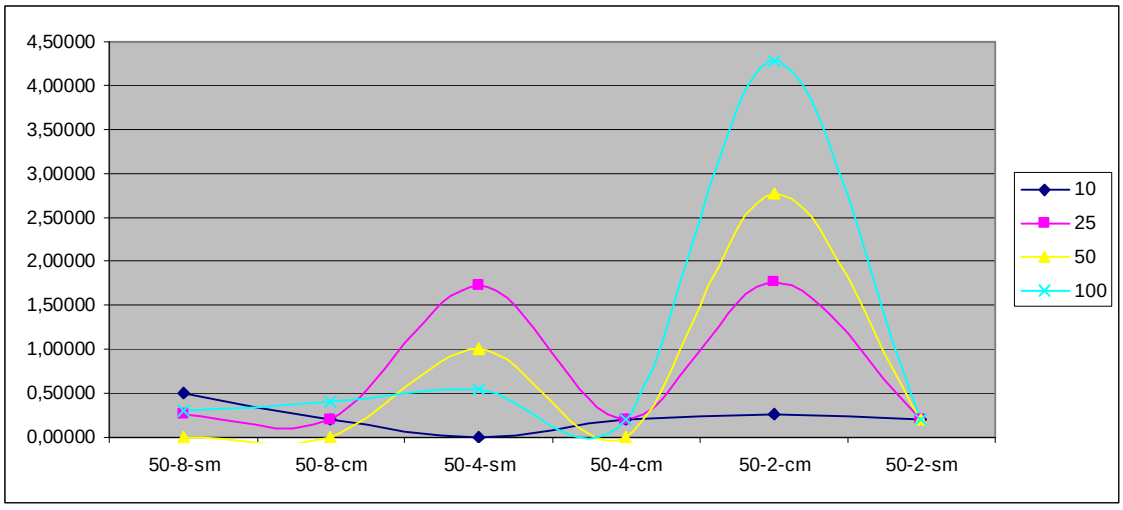




ONEMAX. .FIGURA 6 . 4 procesadores sin mutación. 2 procesadores sin y con mutación. Cada 200 generaciones falla un procesador

experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
Onemax 200 8 sin mutación	10	si	300	680	1	12,5	3	37,5	396	398	0,502513	396	398
	25	si	400	750	2	25,0	3	37,5	394	393	0,254453	383	380
	50	si		400			2	25,0	352	352	0	327	326
	100	no		180			0	0,0	331	332	0,301205	305	305
OneMax 200 8 con mutación	10	si	600	1400	3	37,5	4	50,0	511	512	0,195313	510	510
	25	si	550	1650	2	25,0	4	50,0	511	510	0,196078	508	506
	50	si	750	1900	3	37,5	4	50,0	508	508	0	504	502
	100	si	900	1680	4	50,0	4	50,0	506	504	0,396825	501	496
OneMax 200 4 sin mutación	10	si		460			2	50,0	391	391	0	391	391
	25	si	200	580	1	25,0	2	50,0	398	405	1,728395	398	404
	50	si	200	700	1	25,0	2	50,0	391	395	1,012658	387	384
	100	si	250	600	1	25,0	2	50,0	368	366	0,546448	348	341
OneMax 200 4 con mutación	10	si	380	1300	1	25,0	2	50,0	511	512	0,195313	509	510
	25	si	380	1300	1	25,0	2	50,0	511	512	0,195313	509	509
	50	si	380	1300	1	25,0	2	50,0	511	511	0	509	508
	100	si	380	1700	1	25,0	2	50,0	510	511	0,195695	508	506
OneMax 200 2 sin mutación	10	si		440			1	50,0	383	384	0,260417	383	384
	25	si	200	440	1	50,0	1	50,0	389	396	1,767677	389	396
	50	si	200	480	1	50,0	1	50,0	387	398	2,763819	387	398
	100	si	200	460	1	50,0	1	50,0	380	397	4,282116	375	394
OneMax 200 2 con mutación	10	si	200	1720	1	50,0	1	50,0	511	512	0,195313	510	510
	25	si	200	1720	1	50,0	1	50,0	511	512	0,195313	509	510
	50	si	200	1720	1	50,0	1	50,0	511	512	0,195313	509	509
	100	si	200	1720	1	50,0	1	50,0	511	512	0,195313	509	509

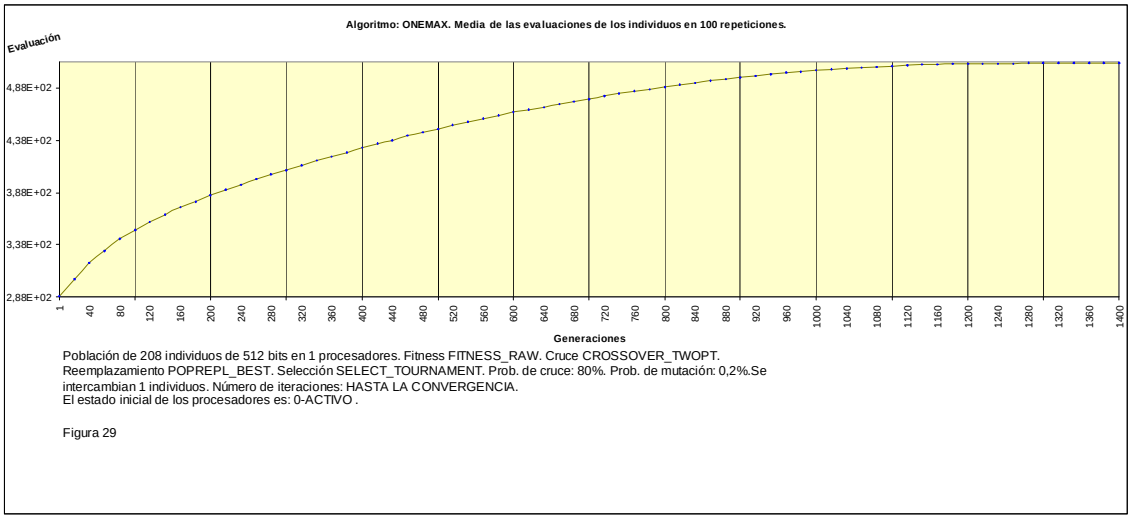
de 0,5 % a 1% de 0% a 0,5%



ONEMAX. TABLA 3. Resumen de resultados incomunicando cada 200 generaciones

Efectos de la mutación

Como primera conclusión sobre esta función, y como veremos en sucesivos experimentos, podemos concluir que la mutación puede sustituir al intercambio entre procesadores en el modelo de islas, siempre y cuando se realice intercambio y no mutación.



ONEMAX Figura 7. Ejecución en un procesador. Probabilidad de cruce 80% y probabilidad de mutación 0.002%.

Podemos observar cómo en un procesador obtenemos el valor óptimo, 512, mientras que si observamos las tablas sin mutación nunca llega a obtener este mejor valor. Hay que destacar que con intercambio y mutación sí se consigue llegar al óptimo prácticamente en todos los casos.

Otro efecto de la mutación es el aumento de la tolerancia a fallos incluso cuando parece que no se ha conseguido tolerar la muerte de algún procesador. Observemos las gráficas correspondientes a los experimentos con mutación en las FIGURAS 1, 2, 3, 4, 5 y 6. Podemos comprobar que, aun cuando las líneas de los experimentos en los que se producen fallos en algún procesador están, prácticamente durante todas las generaciones, por debajo de las correspondientes a los experimentos sin incomunicación, al finalizar la ejecución todos los experimentos convergen al mismo valor. Es decir, **gracias a la mutación conseguimos recuperarnos de la ausencia de procesadores. Conseguimos tolerar la pérdida de hasta un 50% de los procesadores en todos los experimentos realizados con OneMax.**

Tolerancia según el fallo de procesadores

Provocando una muerte cada 50 generaciones: cuando tenemos 8 procesadores se consiguen mejores tolerancias que con 4 y 2 procesadores. 4 y 2 procesadores no toleran ni el primer fallo de un procesador (lo que supone que muera un 25% de la población total) mientras que el experimento con 8 procesadores tolera bastante bien hasta la cuarta y última muerte. A partir de aquí no consigue recuperarse. Además, tolera mejor los fallos cuando la población total esta más divida, de manera que en cada muerte de un procesador se pierde menos población total.

Matando cada 100: se puede observar prácticamente la misma evolución que provocando fallos cada 50 generaciones. Con 8 procesadores tolera mejor que con 4 y con 2, que no llegan a tolerar ni una sola muerte, si bien cabe destacar que se consiguen mejores tolerancias con 8 procesadores que en el caso en que se producían fallos cada 50 generaciones.

Matando cada 200: aquí podemos observar cómo en el caso de 4 procesadores se mejoran los resultados con respecto a las pruebas con muertes cada 100 generaciones, no llegando de todos modos a tolerar la muerte del primer procesador.

Aquí hay que tener en cuenta que para 8 procesadores llegamos a provocar el fallo en solo 3 de ellos. Por lo tanto, mirando los resultados anteriores, vemos que el experimento con 8 procesadores es capaz de tolerar hasta 4 muertes (un 50 %) si los fallos se producen cada 100 generaciones.

De manera general y observando las tablas resumen, podemos apreciar: si el periodo en el que provocamos los fallos es mayor se obtienen mejores tolerancias a fallos que si este periodo es inferior. Es decir, **la consecución sucesiva de fallos en los procesadores con poca frecuencia de intercambio (cada 50 generaciones) obliga a considerar como no válidas las condiciones obtenidas a no ser que se utilice la mutación como método para sobrellevar estos fallos.**

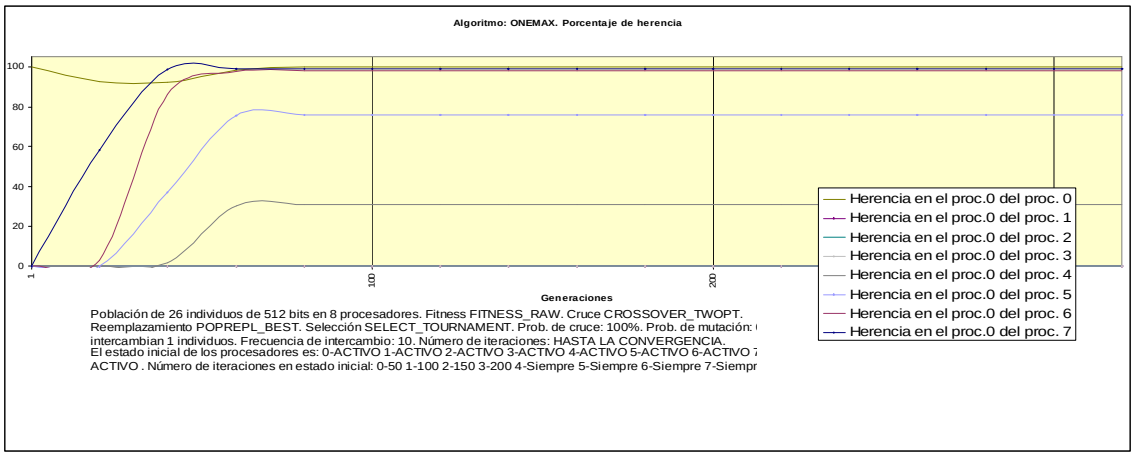
Efectos de la frecuencia de intercambio

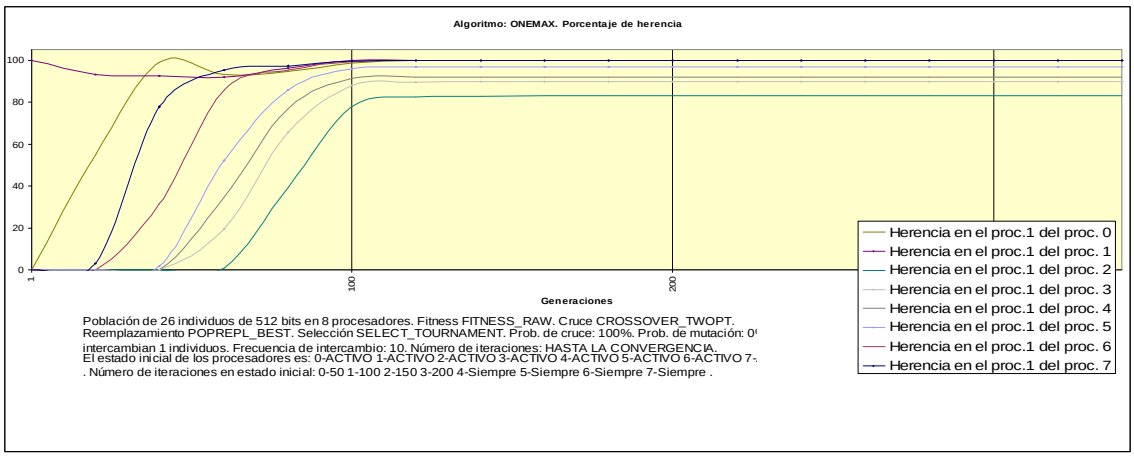
Revisando las TABLAS 1, 2 y 3 y fijándonos en los apartados correspondientes a experimentos sin mutación, podemos observar que se consiguen mejores resultados con frecuencias de intercambio mayores, sin embargo, se debe notar que no se consiguen resultados óptimos como con la mutación. Esta última observación nos lleva a justificar otra vez la importancia de la mutación para conseguir mejores resultados.

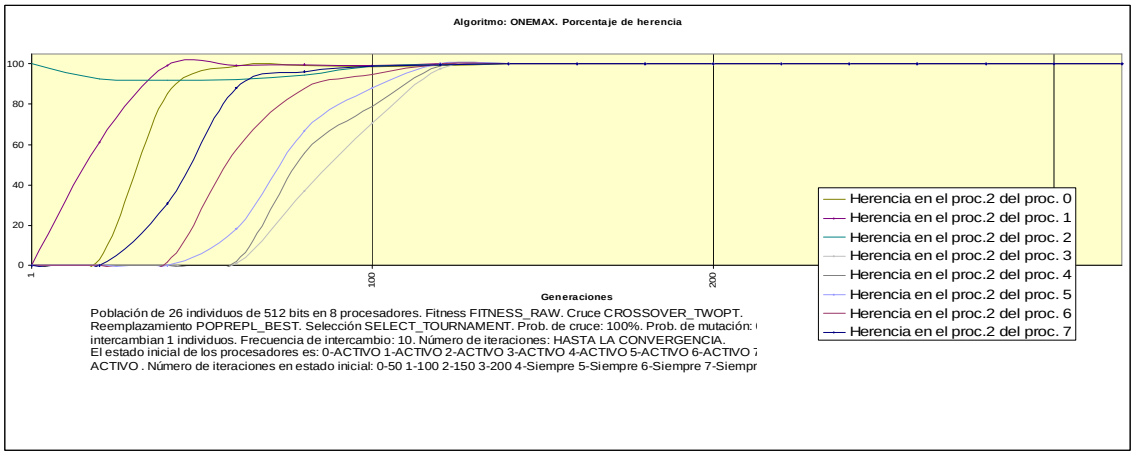
Efectos de la herencia

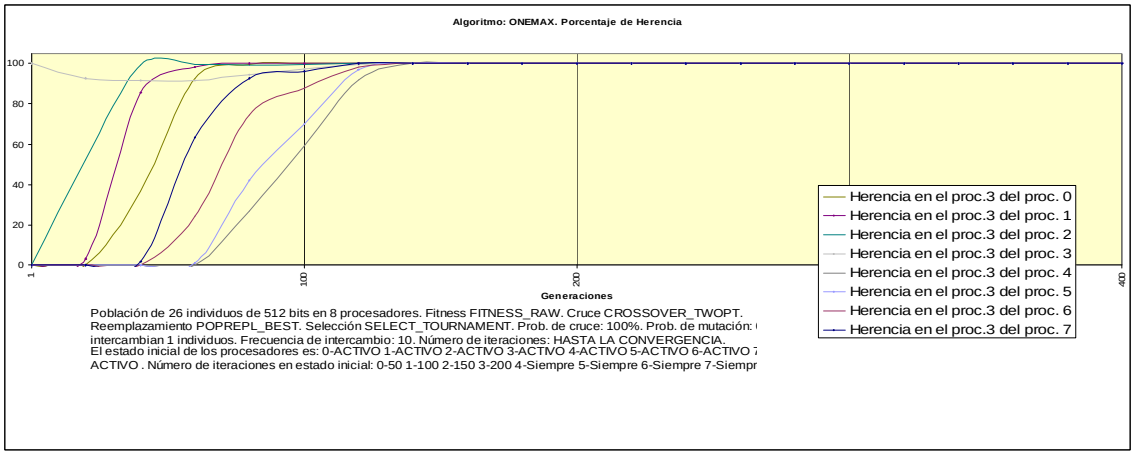
A continuación pasaremos a comentar los efectos de la herencia en la tolerancia a los fallos del experimento.

El experimento con este algoritmo, OneMax, presenta unas características apreciables en la herencia. A pesar de que falle un procesador, éste consigue transmitir la información (los esquemas correctos) a todas las poblaciones, como mostramos en las FIGURAS 8 y 9 (herencia del experimento con 8 procesadores sin mutación y con fallos cada 50 generaciones). Con 8 procesadores se consigue transmitir las propiedades y buenas soluciones para construir la solución final. El periodo de intercambio cada 10 generaciones consigue buenos resultados, lo que parece indicar que los esquemas que mayor influencia tienen en la formación de una buena solución se producen al principio. Esto es razonable puesto que se trata de un problema sencillo.

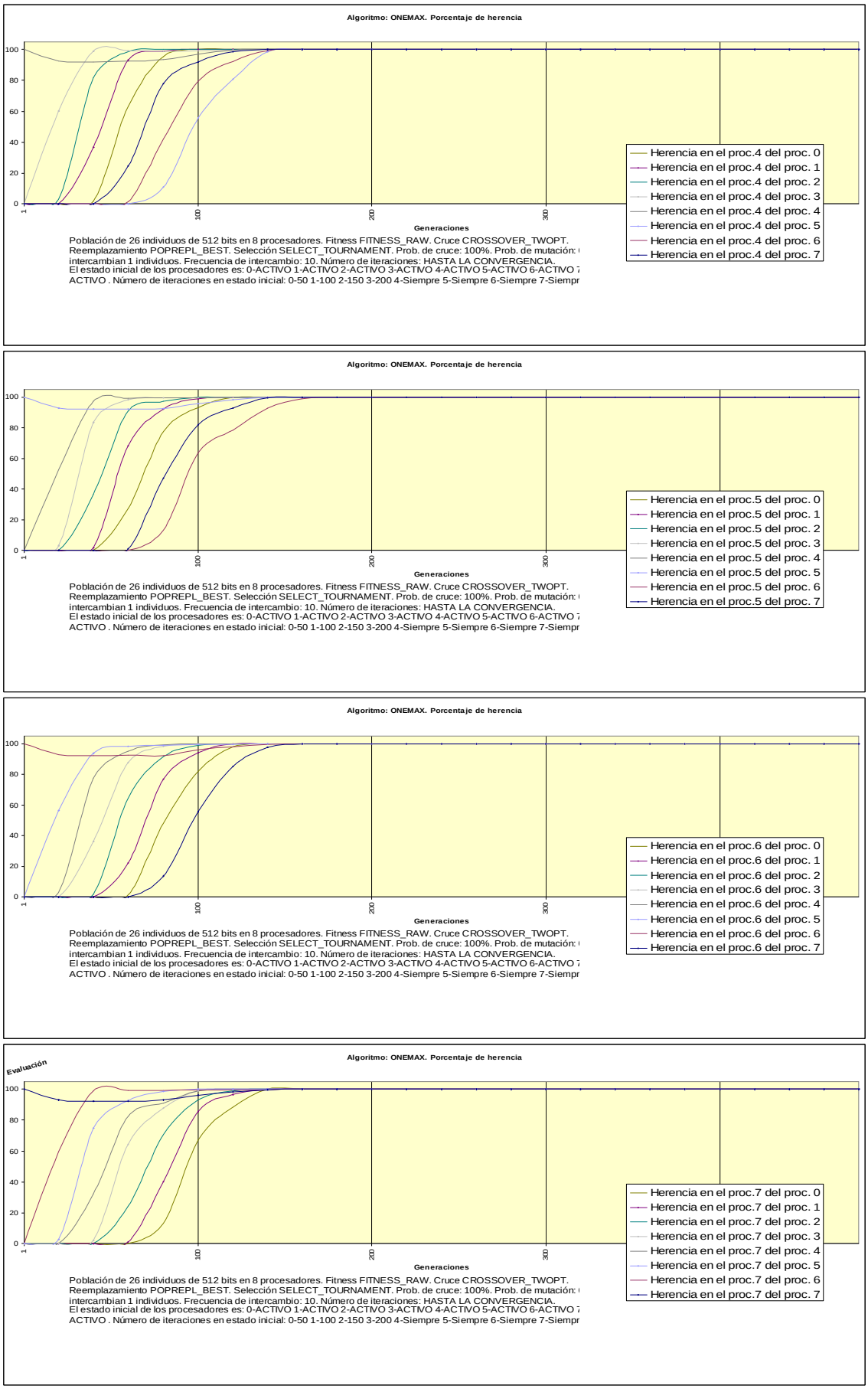






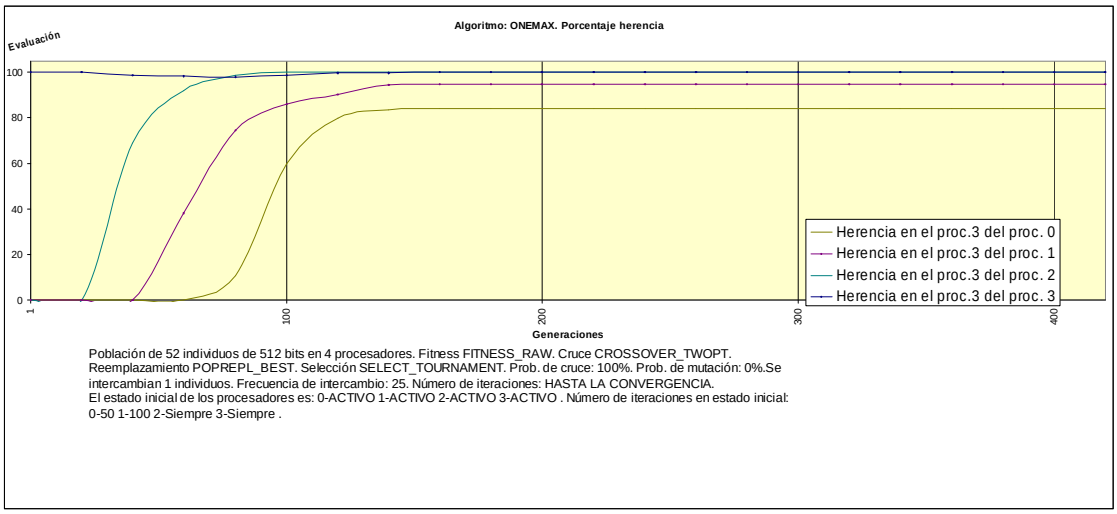


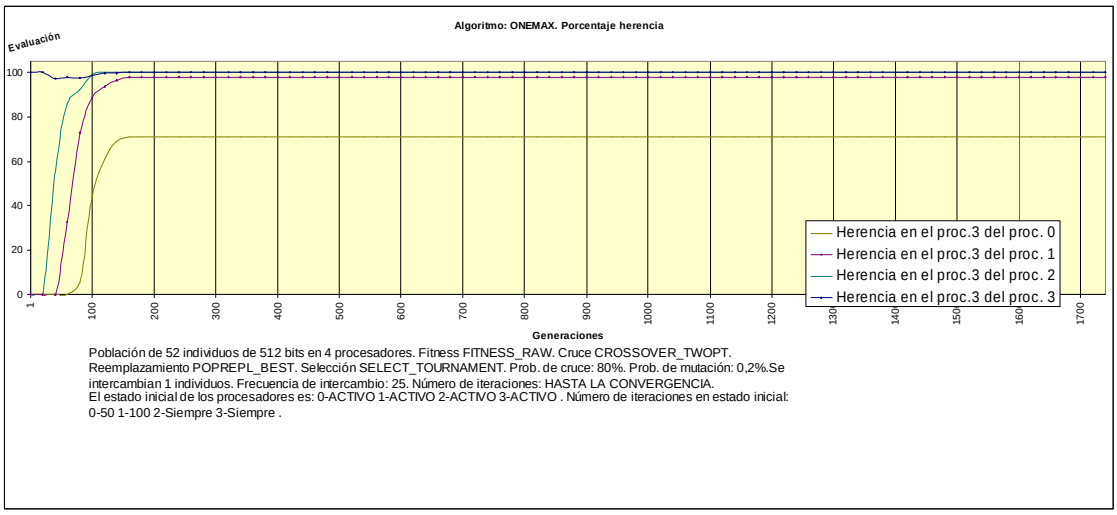
ONEMAX Figura 8. Herencia del procesador 0 al 3. 8 procesadores. Prob. Cruce 100%. Prob. Mut. 0 %. Fallos cada 50 generaciones, intercambio cada 10.



ONEMAX Figura 9. Herencia del procesador 4 al 7. Prob. Cruce 100%. Prob. Mut. 0 %. Fallos cada 50 generaciones, intercambio cada 10.

También podemos observar como en el caso de la herencia con mutación y sin mutación, aun teniendo peores condiciones de herencia en el caso de la mutación, ésta consigue mejores resultados. Veamos las dos siguientes gráficas:





En estas dos gráficas se muestra la herencia para el procesador 3 con y sin mutación. Observamos como aun teniendo la gráfica de herencia referente a la mutación menos herencia procedente del procesador 0, los resultados obtenidos con mutación son mejores que sin ella.

7.3. Función Rastrigin:

A continuación detallaremos un pequeño índice en el que se comentaran las pruebas realizadas para este experimento:

- **Fallo cada 50 generaciones:** hemos provocado el fallo de un procesador cada 50 generaciones. Más propiamente dicho, incomunicación ya que ese procesador sigue ejecutándose aisladamente y seguimos recopilando datos de su ejecución.

  Se ha repetido el experimento para 8 procesadores, 4 procesadores y 2 procesadores con y sin mutación.

  - *Con mutación:* Probabilidad de cruce 0,8. Probabilidad de mutación *0,002.*
  - *Sin mutación:* Probabilidad de cruce 1. Probabilidad de mutación 0.

  Para todos los casos se han considerado los periodos de intercambio cada 10, 25, 50 y 100 generaciones.

  Así mismo, todas estas pruebas se han repetido con las mismas condiciones sin forzar ningún fallo en los procesadores.

  Se puede observar los resultados en forma de gráfica en las FIGURA 1 y 2, así como un resumen de los resultados en la TABLA1.

- **Fallo cada 100 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones, pero esta vez se han provocado los fallos cada 100 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 3 y 4, así como un resumen de los resultados en la TABLA 2.

- **Fallo cada 200 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones pero esta vez se han provocado los fallos cada 200 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 5 y 6, así como un resumen de los resultados en la TABLA 3.

Para una mejor comprensión de las tablas procederemos a continuación a la explicación de cada una de las columnas:

- **Experimento:** resumen del experimento al que se refieren los resultados.
- **Intercambio:** periodo de intercambio utilizado en ese experimento.

- **Herencia de todos:** en esta columna se observará si todos los procesadores que quedan "vivos" al final de la ejecución han conseguido recibir herencia de todos los procesadores con los que se inicio la ejecución.

- **Separan en generación:** en esta columna se muestra a partir de qué generación las líneas de la gráfica correspondientes al experimento con fallos y aquéllas correspondientes al experimento sin fallos comienzan a separarse indicando tener algún problema por la "muerte" de algún procesador.

- **Generacion:** número de generaciones que ha ejecutado el experimento.

- **Proc Caidos:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos.

- **%Pob Muerta:** mostramos el porcentaje de población muerte en el momento en el que se empiezan a separar las líneas.

- **Proc Caidos Total:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos al finalizar la ejecución.

- **%Pob Muerta Total:** mostramos el porcentaje de población muerta en el momento en el que se empiezan a separar la líneas al finalizar la ejecución

- **Max Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

- **Max Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

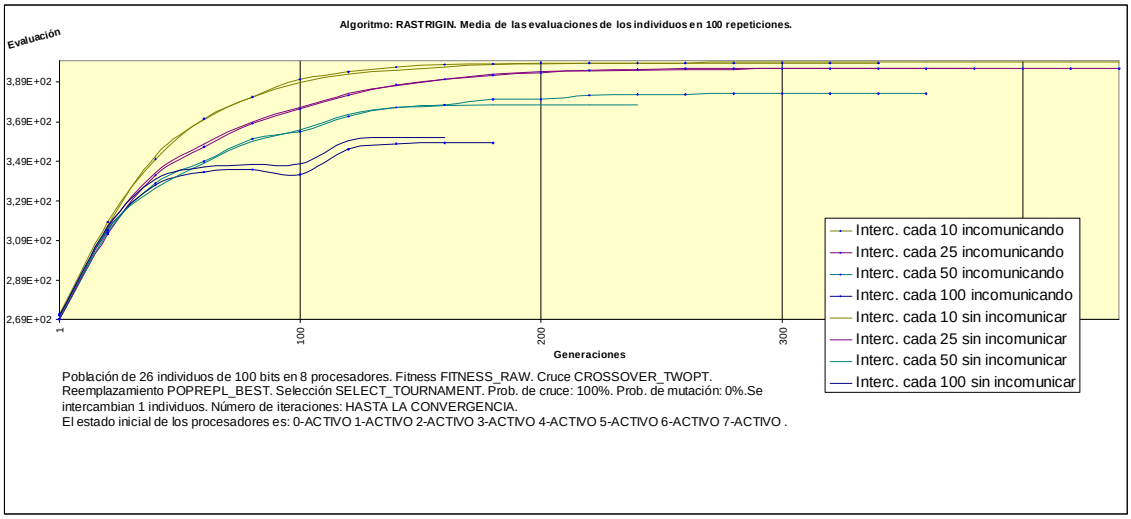
- **Tolerancia (% error):** diferencia entre el valor alcanzado con fallos en los procesadores y la ejecución sin fallos. Expresada en porcentaje.

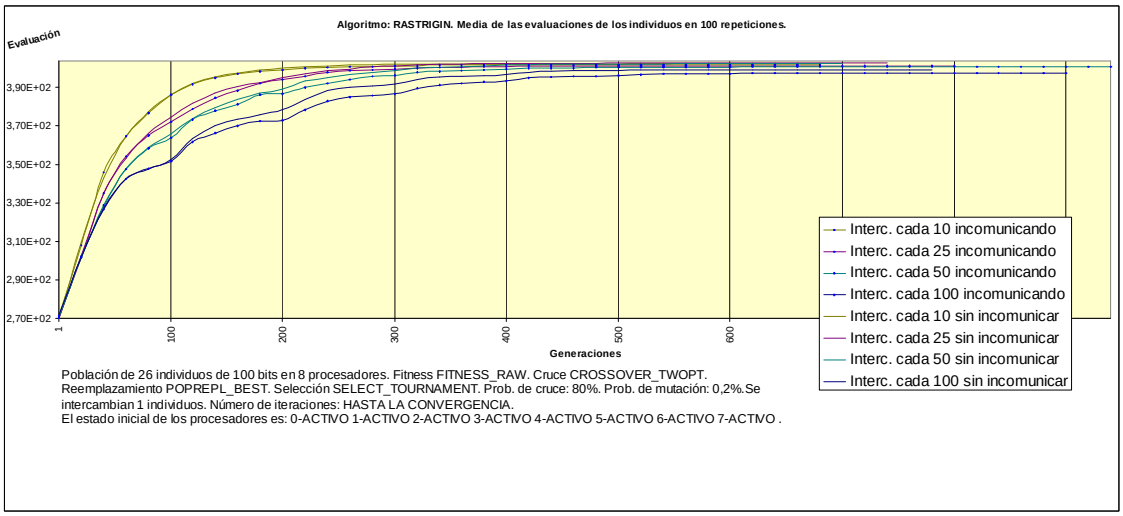
- **Min Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

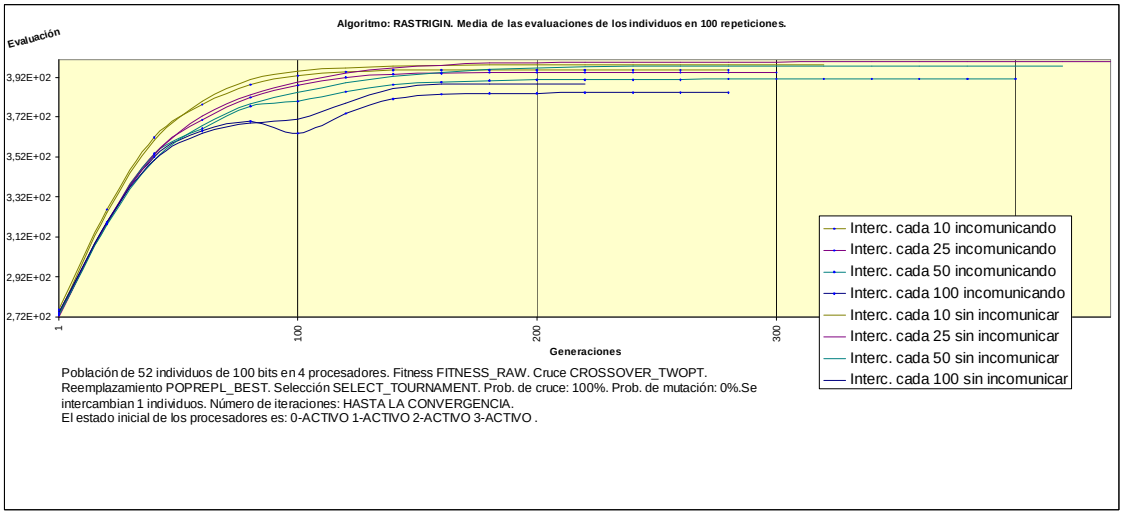
- **Min Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

A continuación de cada tabla se muestra una gráfica en la que se indica la evolución del porcentaje de tolerancia según el periodo de intercambio y si el experimento fue realizado con mutación (cm) o sin mutación (sm).

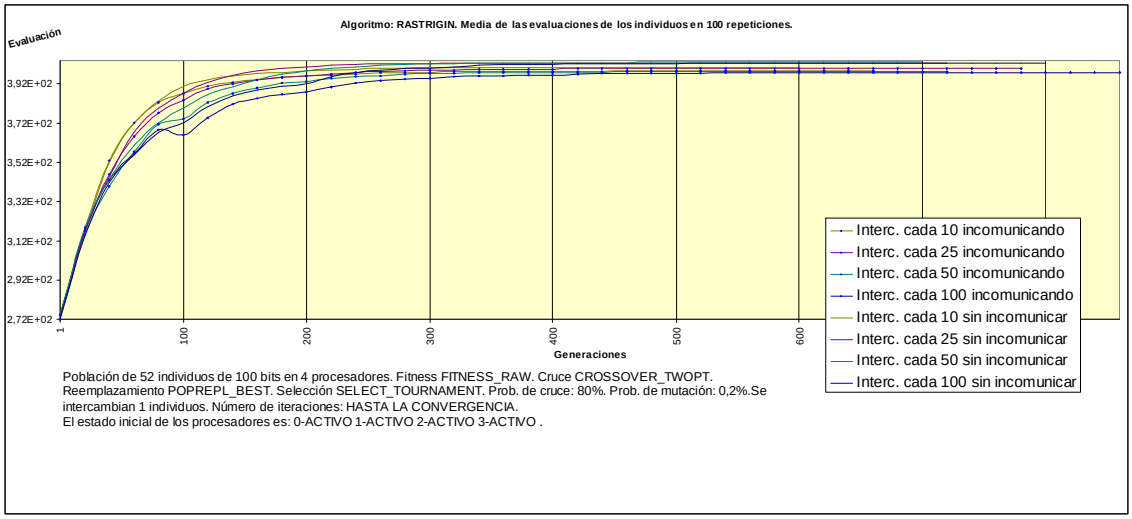
7.3.1. Fallo cada 50 generaciones

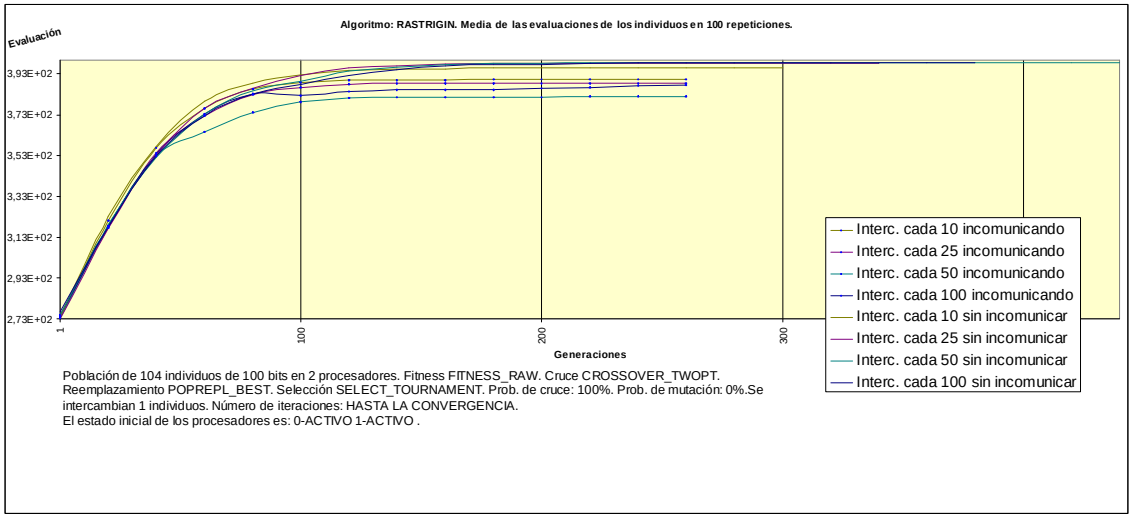


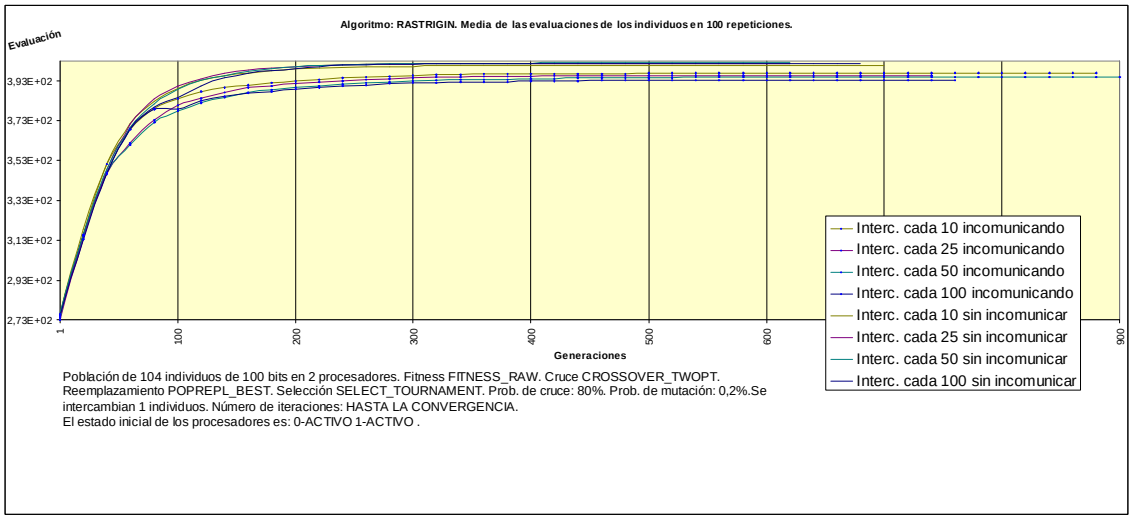




RASTRIGIN. .FIGURA 1 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 50 generaciones falla un procesador



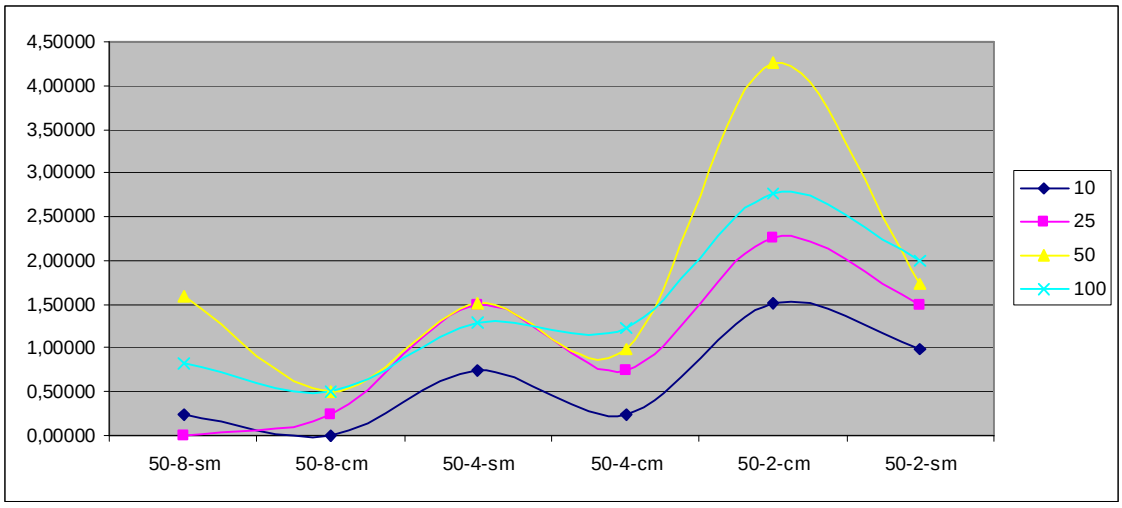




RASTRIGIN. FIGURA 2 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 50 generaciones falla un procesador

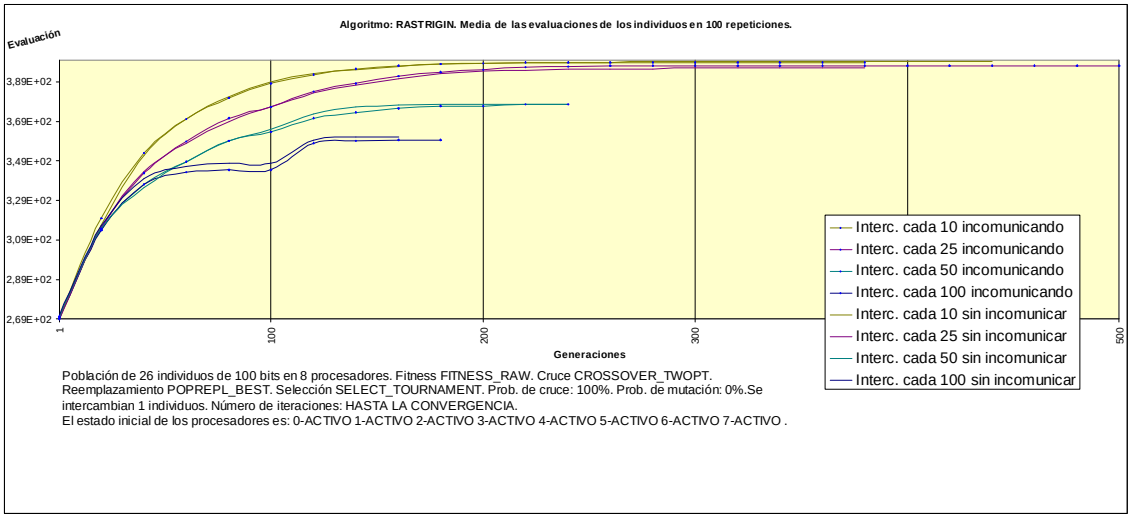
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
rastrigin 50 8 sin mutación	10	si		380			4	50,0	398	399	0,25063	398	399
	25	si		500			4	50,0	395	395	0,00000	391	379
	50	no	160	240	3	37,5	4	50,0	383	377	1,59151	348	329
	100	no	50	180	1	12,5	3	37,5	358	361	0,83102	305	393
rastrigin 50 8 con mutación	10	si		700			4	50,0	402	402	0,00000	402	402
	25	si	100	900	2	25,0	4	50,0	401	402	0,24876	401	401
	50	no	100	800	2	25,0	4	50,0	400	402	0,49751	399	392
	100	no	100	900	2	25,0	4	50,0	397	399	0,50125	390	376
rastrigin 50 4 sin mutación	10	si	80	300	1	25,0	2	50,0	395	398	0,75377	395	398
	25	si	80	340	1	25,0	2	50,0	394	400	1,50000	394	399
	50	no	80	480	1	25,0	2	50,0	391	397	1,51134	391	386
	100	no	80	280	1	25,0	2	50,0	384	389	1,28535	366	356
rastrigin 50 4 con mutación	10	si	80	760	1	25,0	2	50,0	399	400	0,25000	399	400
	25	si	80	740	1	25,0	2	50,0	400	403	0,74442	400	403
	50	no	80	780	1	25,0	2	50,0	399	403	0,99256	396	403
	100	no	80	860	1	25,0	2	50,0	398	403	1,24069	397	400
rastrigin 50 2 sin mutación	10	si	50	280	1	50,0	1	50,0	390	396	1,51515	390	396
	25	si	80	280	1	50,0	1	50,0	389	398	2,26131	389	398
	50	no	50	300	1	50,0	1	50,0	382	399	4,26065	382	399
	100	no	80	260	1	50,0	1	50,0	388	399	2,75689	385	397
rastrigin 50 2 con mutación	10	si	80	680	1	50,0	1	50,0	397	401	0,99751	397	401
	25	si	80	780	1	50,0	1	50,0	396	402	1,49254	395	402
	50	no	80	820	1	50,0	1	50,0	395	402	1,74129	395	401
	100	no	80	800	1	50,0	1	50,0	394	402	1,99005	393	402

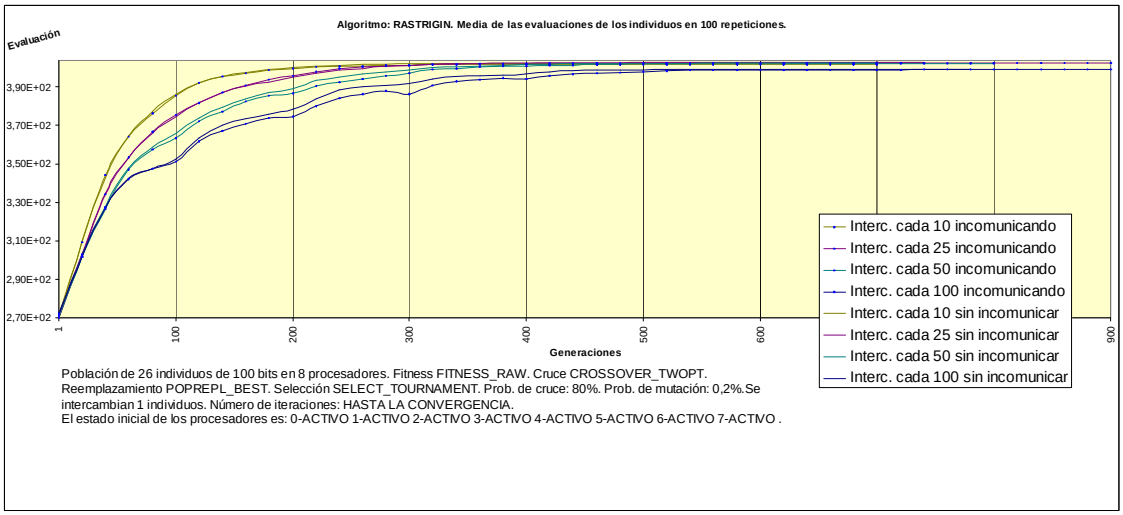
de 0,5 % a 1% de 0% a 0,5%

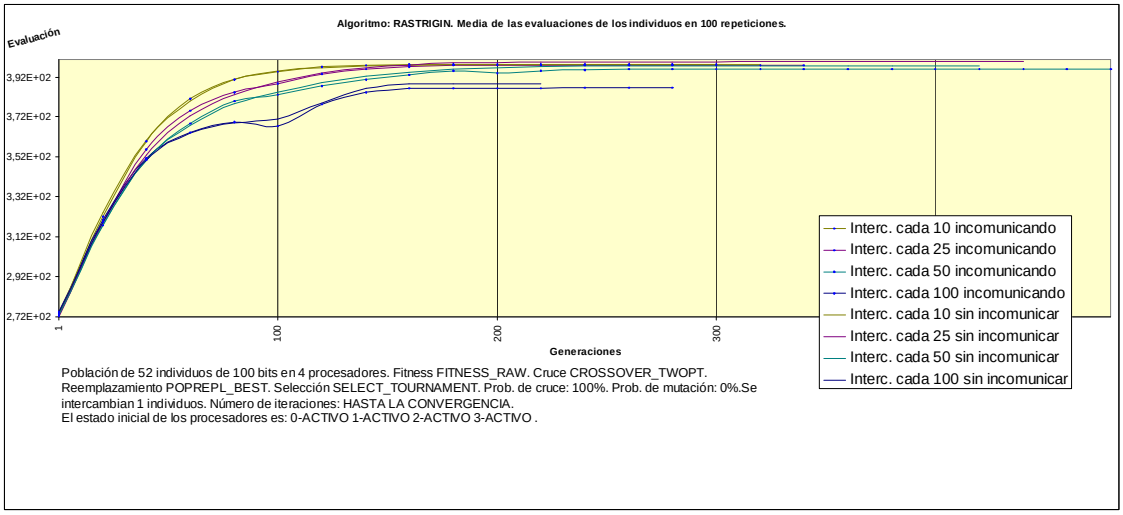


RASTRIGIN. TABLA 1. Resumen de resultados incomunicando cada 50 generaciones

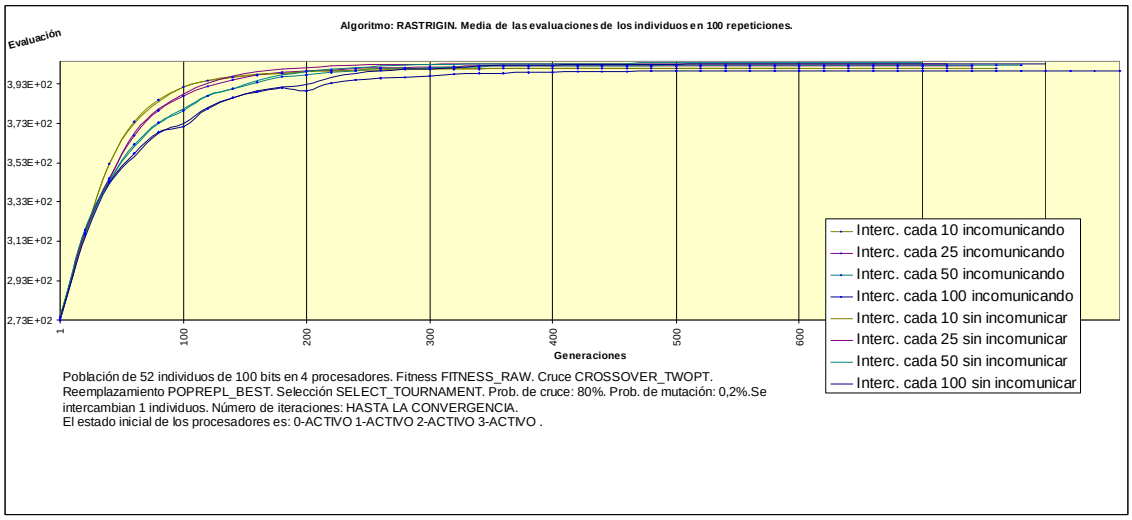
7.3.2. Fallo cada 100 generaciones

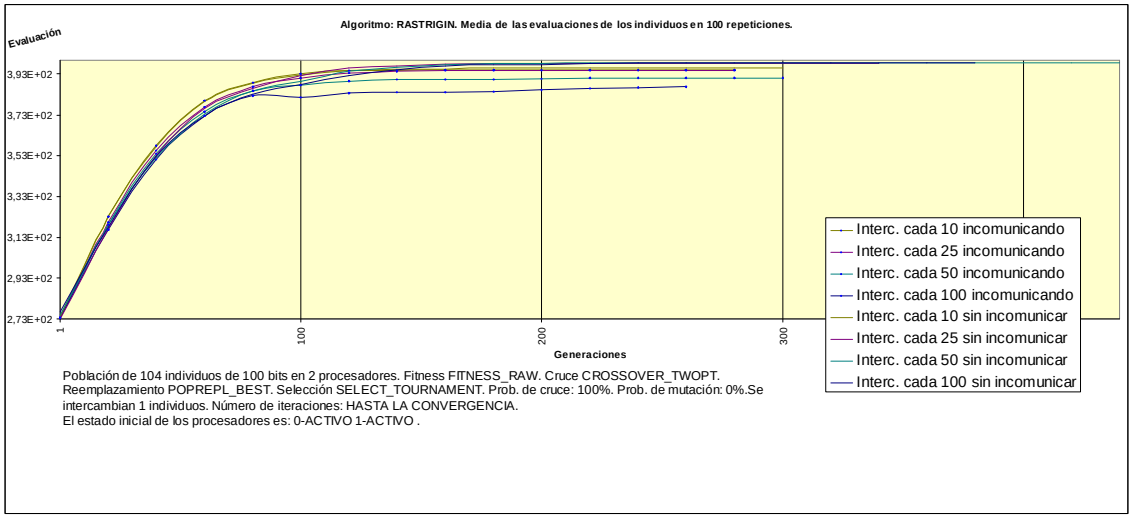


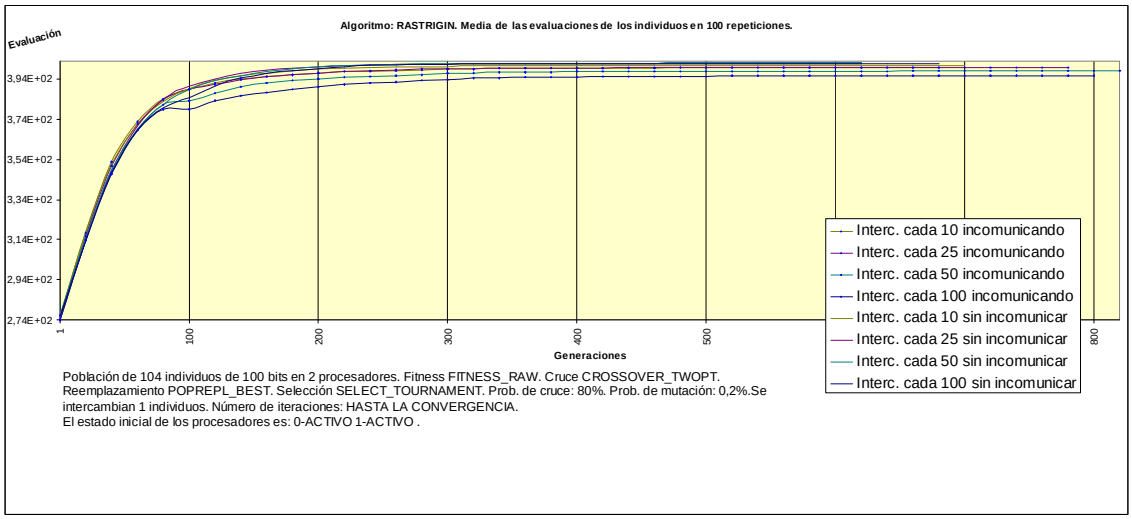




RASTRIGIN. .FIGURA 3 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 100 generaciones falla un procesador



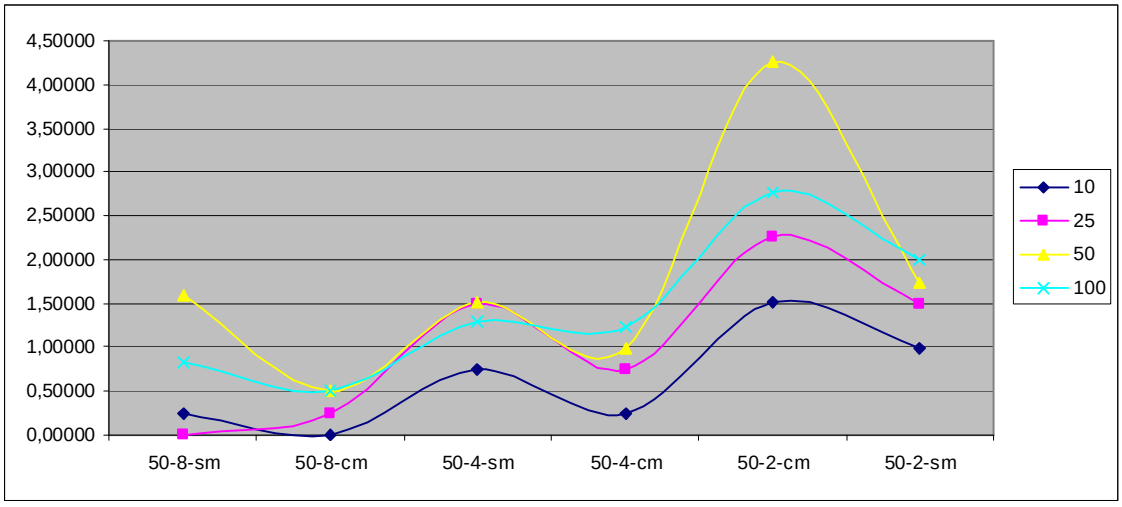




RASTRIGIN. FIGURA 4 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 100 generaciones falla un procesador

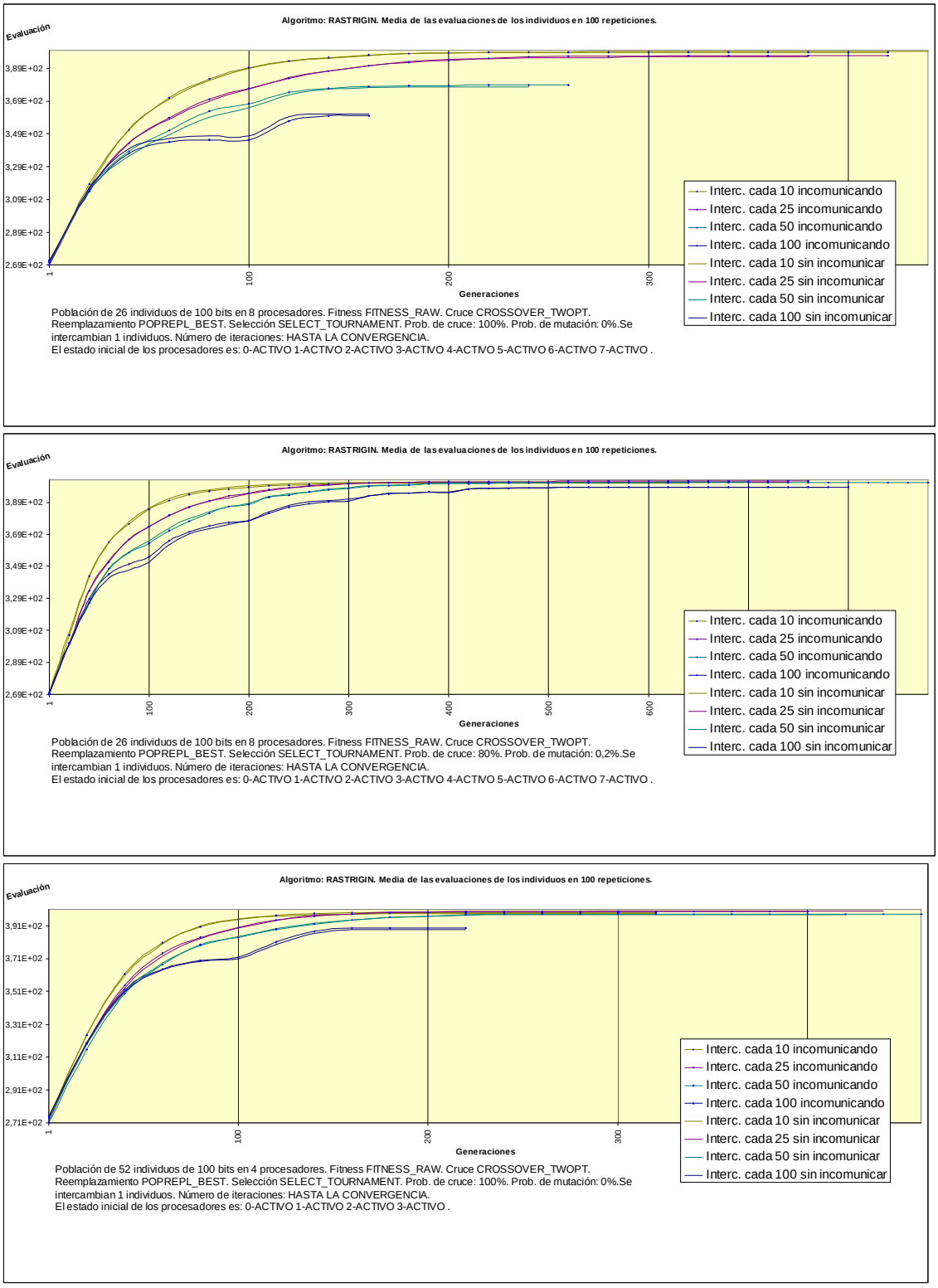
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
rastrigin 100 8 sin mutación	10	si		380			3	37,5	398	399	0,25063	398	399
	25	si	220	500	2	25,0	4	50,0	397	395	0,50633	383	379
	50	no		240			2	25,0	377	377	0,00000	335	329
	100	no	50	180	0	0,0	1	12,5	359	361	0,55402	299	293
rastrigin 100 8 con mutación	10	si		700			4	50,0	402	402	0,00000	401	402
	25	si		900			4	50,0	402	402	0,00000	402	401
	50	si	100	800	1	12,5	4	50,0	402	402	0,00000	398	392
	100	no	100	900	1	12,5	4	50,0	399	399	0,00000	389	376
rastrigin 100 4 sin mutación	10	si		300			2	50,0	398	386	3,10881	398	398
	25	si		340			2	50,0	398	398	0,00000	398	399
	50	si	100	480	1	25,0	2	50,0	396	400	1,00000	390	386
	100	no	100	280	1	25,0	2	50,0	386	397	2,77078	362	356
rastrigin 100 4 con mutación	10	si		760			2	50,0	400	400	0,00000	400	400
	25	si	100	740	1	25,0	2	50,0	402	403	0,24814	402	403
	50	si	200	780	2	50,0	2	50,0	402	403	0,24814	402	403
	100	no	200	860	2	50,0	2	50,0	399	403	0,99256	399	400
rastrigin 100 2 sin mutación	10	si		280			1	50,0	395	396	0,25253	395	396
	25	si		280			1	50,0	395	398	0,75377	395	398
	50	si	100	300	1	50,0	1	50,0	392	399	1,75439	391	399
	100	no	100	260	1	50,0	1	50,0	387	399	3,00752	384	397
rastrigin 100 2 con mutación	10	si		680			1	50,0	400	401	0,24938	400	401
	25	si	120	780	1	50,0	1	50,0	400	402	0,49751	400	402
	50	si	100	820	1	50,0	1	50,0	398	402	0,99502	398	401
	100	no	100	800	1	50,0	1	50,0	396	402	1,49254	396	402

de 0,5 % a 1% de 0% a 0,5%

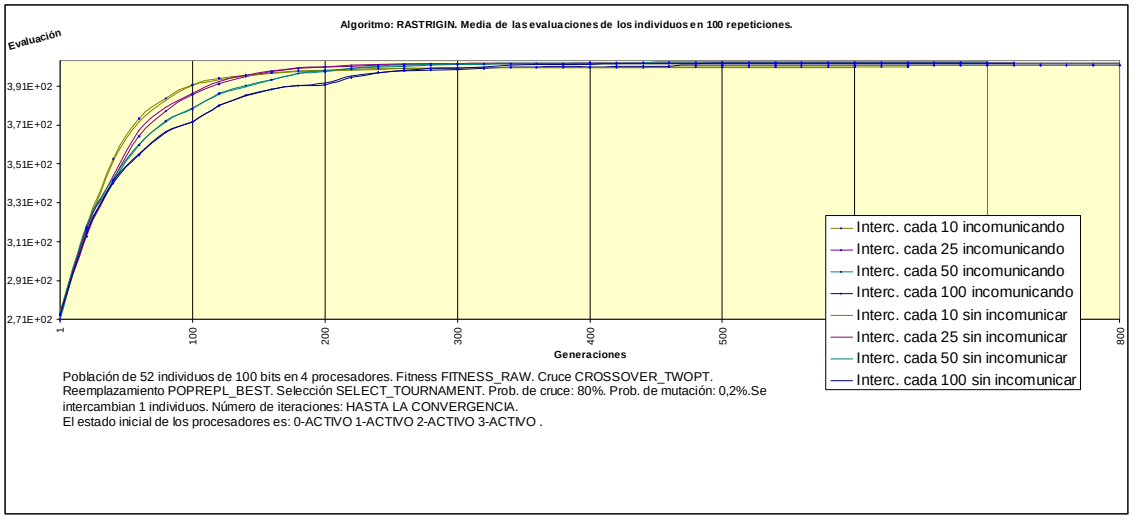


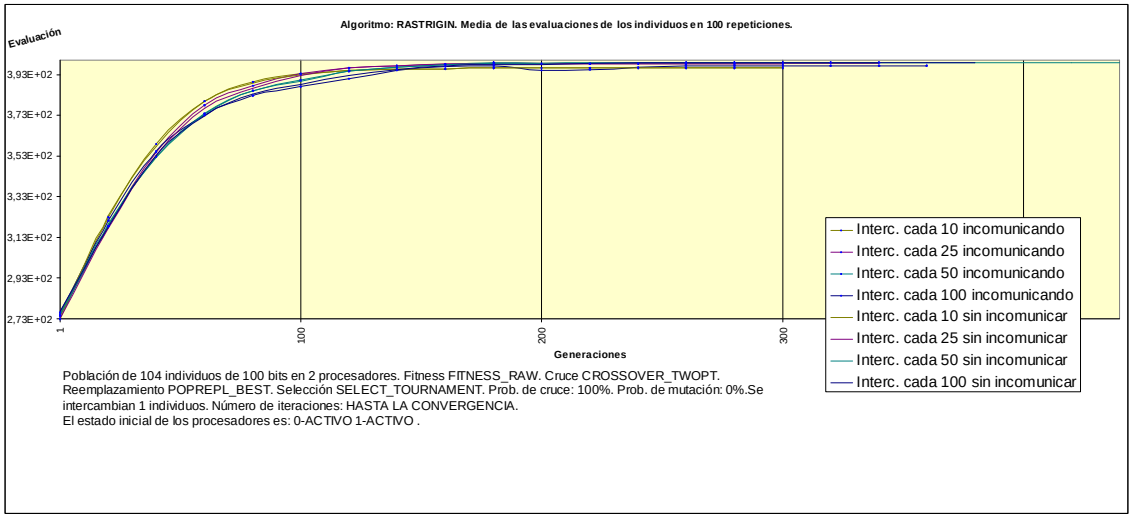
RASTRIGIN. TABLA 2. Resumen de resultados incomunicando cada 100 generaciones

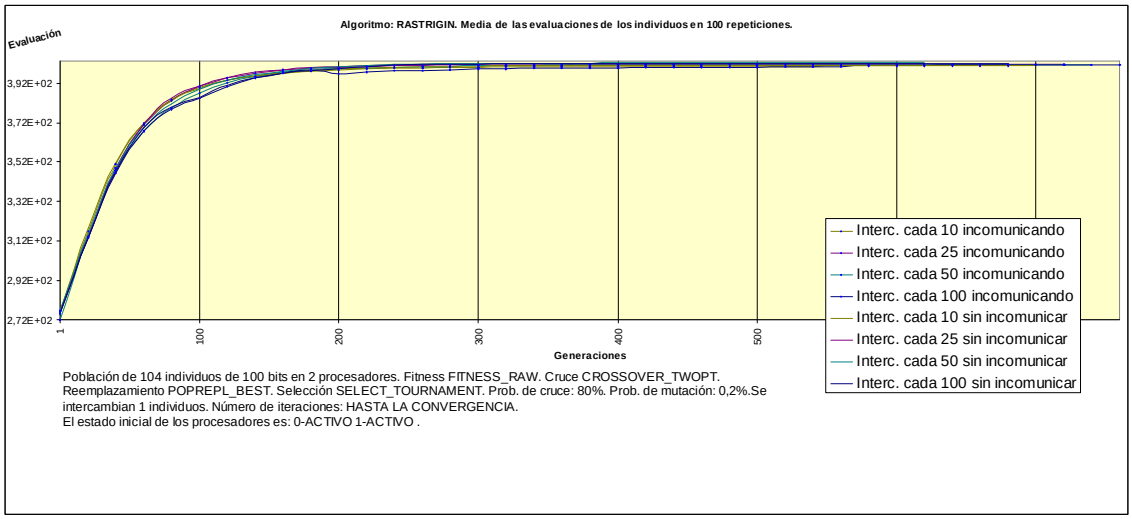
7.3.3. Fallo cada 200 generaciones



RASTRIGIN. .FIGURA 5 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 200 generaciones falla un procesador



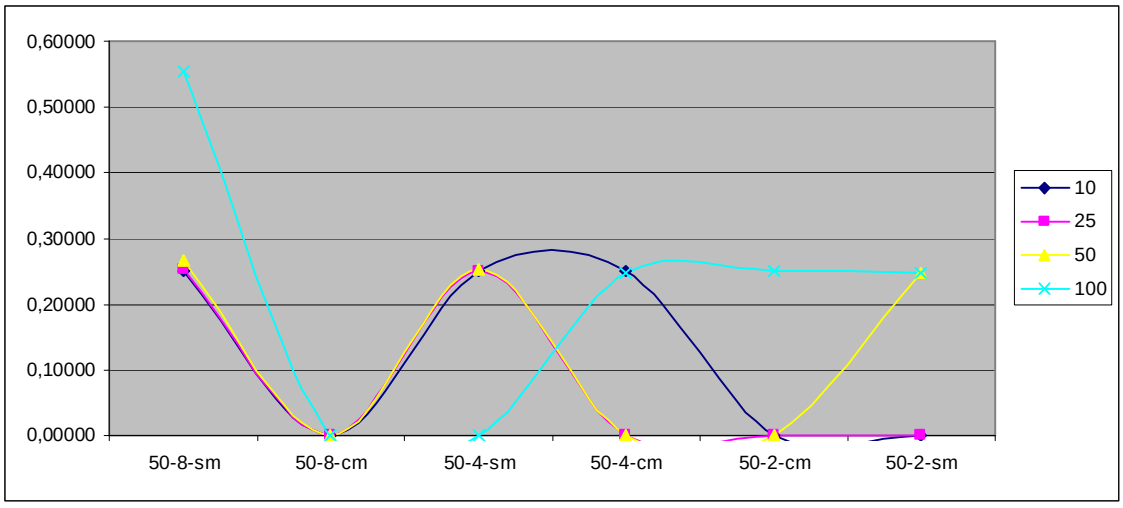




RASTRIGIN. .FIGURA 6 .4 procesadores sin mutación. 2 procesadores sin y con mutación. Cada 200 generaciones falla un procesador

experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
rastrigin 200 8 sin mutación	10	si		420			2	25,0	398	399	0,25063	398	399
	25	si		420			2	25,0	396	395	0,25316	381	379
	50	no	100	260	0	0,0	1	12,5	378	377	0,26525	330	329
	100	no	100	160	0	0,0	0	0,0	359	361	0,55402	299	293
rastrigin 200 8 con mutación	10	si		640			3	37,5	402	402	0,00000	402	402
	25	si		760			3	37,5	402	402	0,00000	401	401
	50	si		880			4	50,0	402	402	0,00000	396	392
	100	si		800			4	50,0	399	399	0,00000	382	376
rastrigin 200 4 sin mutación	10	si		320			1	25,0	399	398	0,25126	399	398
	25	si		400			2	50,0	399	400	0,25000	399	399
	50	si		460			2	50,0	398	397	0,25189	390	386
	100	no		220			1	25,0	389	389	0,00000	357	356
rastrigin 200 4 con mutación	10	si		640			2	50,0	401	400	0,25000	401	400
	25	si		680			2	50,0	403	403	0,00000	402	403
	50	si		720			2	50,0	403	403	0,00000	403	403
	100	si		800			2	50,0	402	403	0,24814	401	400
rastrigin 200 2 sin mutación	10	si		300			1	50,0	396	396	0,00000	396	396
	25	si		300			1	50,0	398	398	0,00000	398	398
	50	si		340			1	50,0	399	399	0,00000	399	399
	100	si		360			1	50,0	398	399	0,25063	394	397
rastrigin 200 2 con mutación	10	si		680			1	50,0	401	401	0,00000	401	401
	25	si		720			1	50,0	402	402	0,00000	401	402
	50	si		740			1	50,0	401	402	0,24876	400	401
	100	si		760			1	50,0	401	402	0,24876	400	402

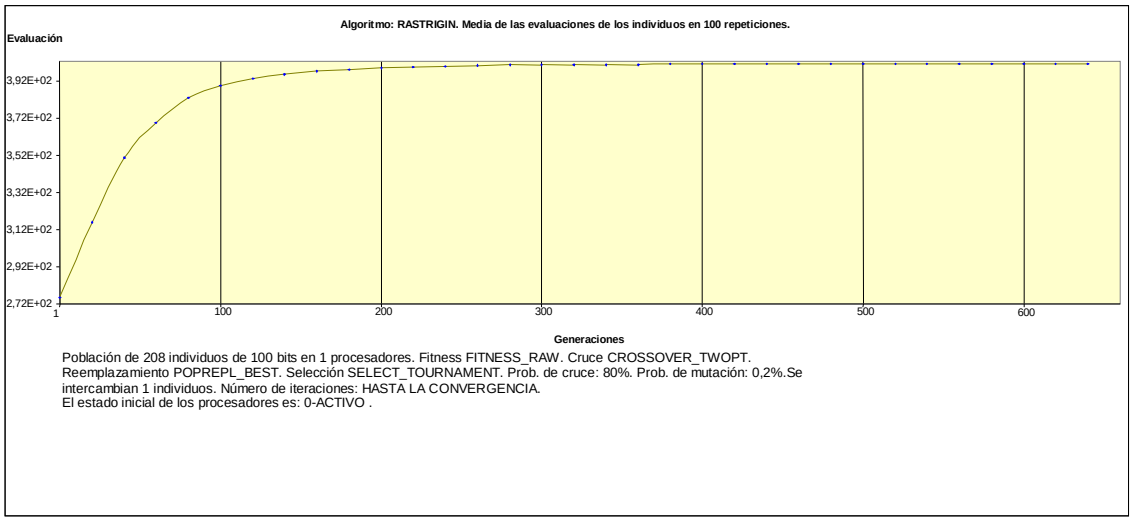
de 0,5 % a 1% de 0% a 0,5%



RASTRIGIN. TABLA 3. Resumen de resultados incomunicando cada 200 generaciones

Efectos de la mutación

Como primera conclusión sobre esta función, al igual que como sucedió con OneMax, podemos concluir que **la mutación puede sustituir al intercambio entre procesadores en el modelo de islas**, siempre y cuando se realice sólo intercambio (y no mutación al mismo tiempo).



RASTRIGIN Figura 7. Ejecución en un procesador. Probabilidad de cruce 80% y probabilidad de mutación 0.002%.

Observamos que un procesador obtenemos el valor 402 o similar, mientras si miramos las tablas correspondientes a experimentos sin mutación nunca llegamos a obtener este valor. Sin embargo, hay que destacar que con intercambio y mutación si se consigue llegar a este valor.

Otro efecto de la mutación es el aumento de la tolerancia a fallos incluso cuando parece que no se ha conseguido tolerar la muerte de algún procesador. Observemos las gráficas correspondientes a los experimentos con mutación en las FIGURAS 1, 2, 3, 4, 5 y 6. Podemos comprobar que, aun cuando las líneas de los experimentos en los que se producen fallos en algún procesador están, prácticamente durante todas las generaciones, por debajo de las correspondientes a los experimentos sin incomunicación, al finalizar la ejecución todos los experimentos convergen al mismo valor (ver, por ejemplo, primera gráfica de la FIGURA 2). Es decir, **gracias a la mutación conseguimos recuperarnos de la ausencia de procesadores. Conseguimos tolerar la pérdida de hasta un 50% de los procesadores en todos los experimentos realizados con Rastrigin.**

Tolerancia según el fallo de procesadores

Provocando una muerte cada 50 generaciones: se ve cómo los únicos experimentos en los que se toleran fallos hasta el 50% de los procesadores es en los que utilizamos 8 procesadores y unas frecuencias de intercambio elevadas, cada 10 y cada 50 generaciones. En la parte en la que comentaremos la herencia comentaremos como,

con altas frecuencias de intercambio, tarde o temprano se conseguirá recibir herencia de todos los procesadores, lo que puede justificar este buen comportamiento.

Del mismo modo, vemos como en los experimentos con 4 y 2 procesadores sin mutación no se llega a tolerar ningún fallo, es decir, la muerte de un 25% y un 50% de la población es suficiente para provocar fallos irrecuperables en la ejecución.

Como concluimos también en el experimento con la función OneMax, podemos decir aquí otra vez que es más fácil tolerar los fallos cuando la población esta más dividida entre los diferentes procesadores. De este modo, la muerte de un procesador no supone una gran pérdida. La muerte en un experimento con 8 procesadores de una isla solo supone la muerte de un 12,5% de la población total, mientras que en un experimento con solo 4 procesadores representaría la muerte de un 25% y, en dos procesadores, de un 50%, lo cual significa la muerte de una sección demasiado amplia de la población, lo que puede anular fuentes muy importantes de soluciones.

También se observa en estos resultados cómo el efecto beneficioso de la mutación para la recuperación de fallos no consigue el efecto deseado nada más que para 8 procesadores. Para 4 y 2 procesadores no consigue recuperar los fallos como en funciones más simples (OneMax).

Provocando una muerte cada 100 generaciones: con muertes más distanciadas en el tiempo conseguimos mejores resultados en todos los casos en los que con fallos cada 50 generaciones no conseguíamos tolerar adecuadamente las caídas de los procesadores. Mejoramos tanto con mutación como sin ella. Se consigue tolerar hasta el 50% de fallos en los procesadores para 8 y 4, mientras que para 2 mejora mucho con respecto al apartado anterior.

Provocando una muerte cada 200 generaciones: para una frecuencia de fallo baja como es este caso, conseguimos una tolerancia en todos los experimentos de hasta el 50% de fallos en los procesadores. También se vuelve a observar como la tolerancia mejora con el uso de la mutación consiguiendo una diferencia prácticamente del 0% con respecto al mismo experimento sin provocar muertes.

De manera general, funciones similares a Rastrigin no consiguen tolerar muertes con periodos de fallos de 50 o inferiores, siendo a partir de 100 cuando se consiguen resultados aceptables y a partir de 200 cuando se puede considerar como perfectamente validas las conclusiones obtenidas. Y otra vez se observa el beneficio de dividir la población en un mayor número de procesadores.

Efectos de la frecuencia de intercambio

Observando las figuras 1, 2, 3, 4, 5 y 6 vemos que, a mayor frecuencia de intercambio, quedan más definidas y separadas las líneas que corresponden a una frecuencia de intercambio con las de frecuencias sucesivas. Es decir, a mayor frecuencia de intercambio se obtienen mejores resultados.

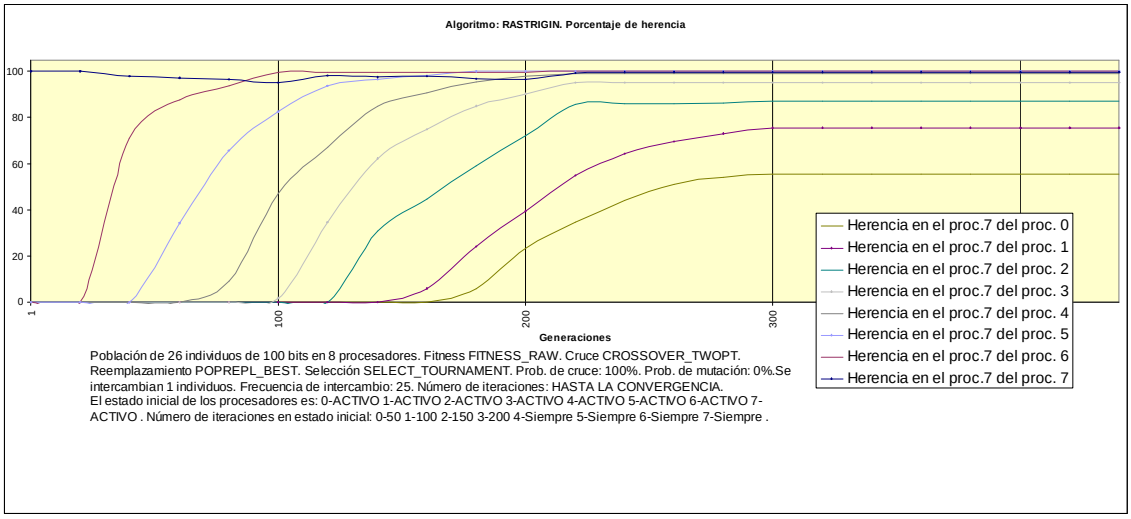
Estas diferencias se ven más claramente para experimentos con 8 procesadores (poblaciones mas pequeñas en más islas) que en experimentos con 4 y 2 procesadores (poblaciones más grandes en menos islas). En experimentos con poblaciones pequeñas

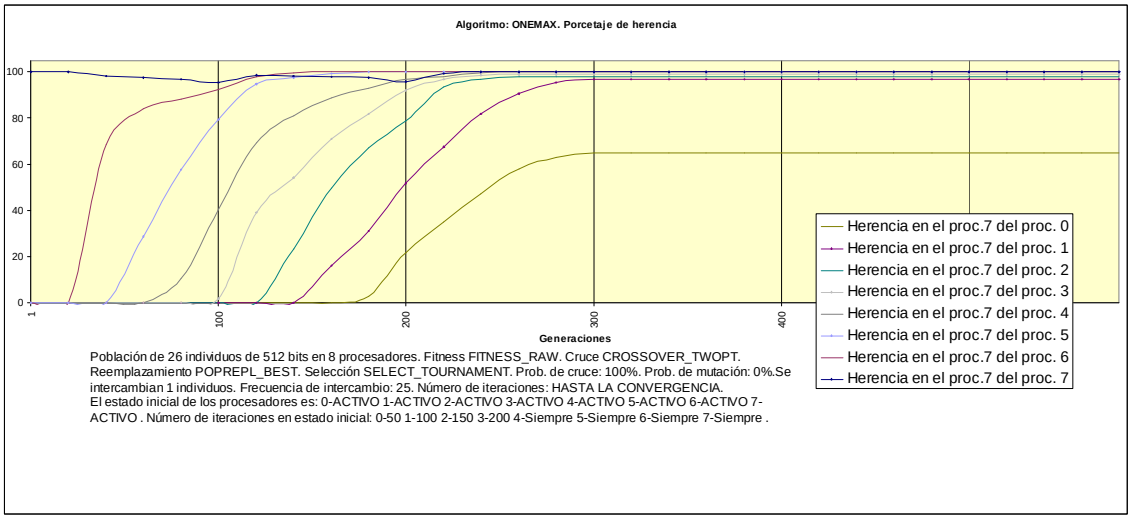
la diversidad obtenida gracias al intercambio de individuos es muy importante para desarrollar una solución buena, mientras que en poblaciones más grandes pueden suplir esta falta de intercambio con la propia diversidad encontrada en su población.

Efectos de la herencia

Para este tipo de experimentos se aprecia como la herencia depende de la distancia entre los procesadores, si bien en OneMax se consigue distribuir la herencia a todos los procesadores incluso después del fallo de un procesador. Esto lo consigue con mayor facilidad igualándolo con Rastrigin a igualdad de periodo de intercambio. En este tipo de experimentos parece no conseguirse esto, puesto que después de que un procesador haya fallado no aumenta considerablemente la información de este en el resto de procesadores, notándose mas este hecho con periodos de intercambio mayores.

A continuación mostramos la herencia del procesador 7 de Rastrigin y esta misma gráfica para OneMax. En ambos experimentos no se ha utilizado mutación y la frecuencia de intercambio es cada 25 generaciones.

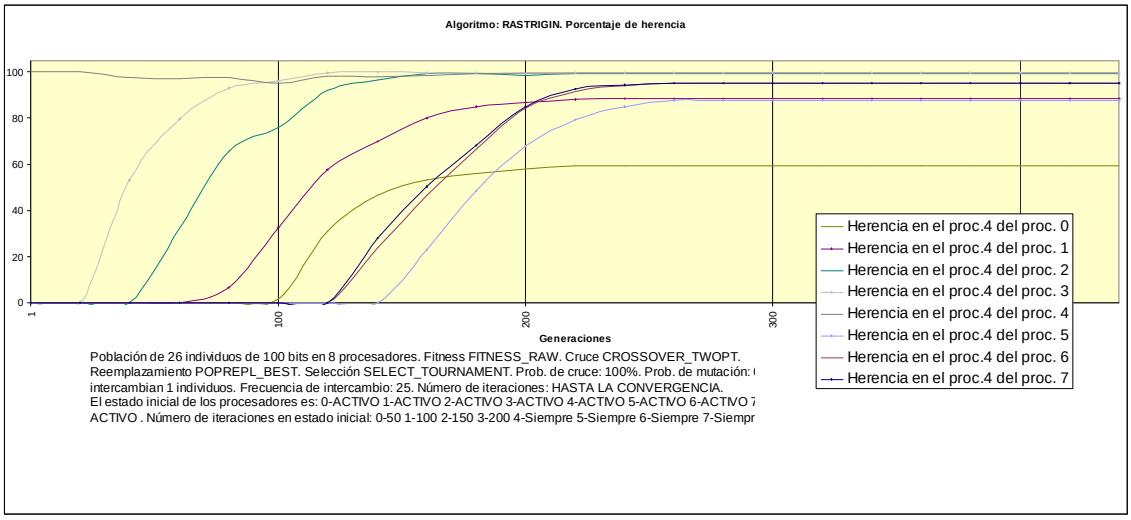




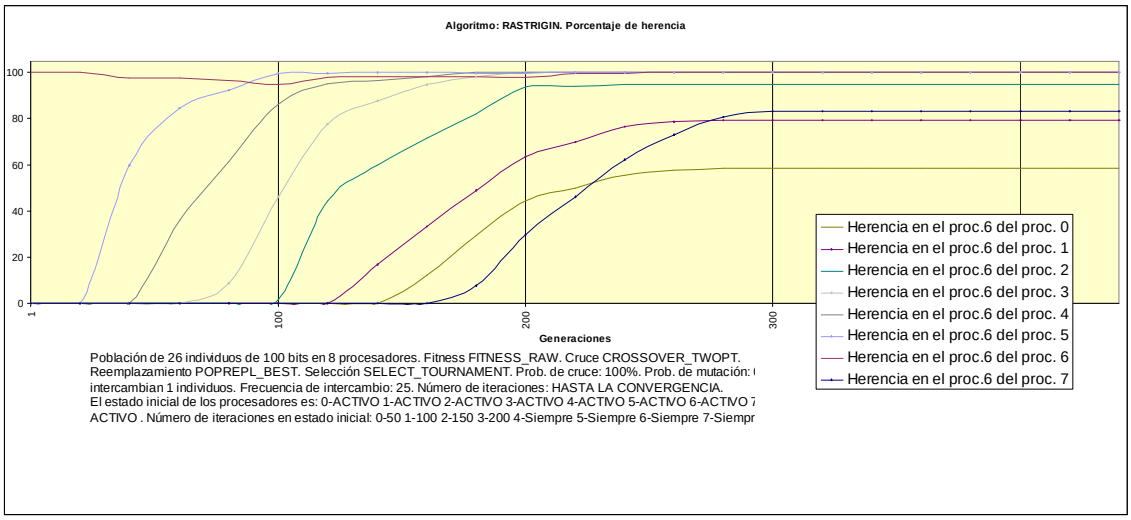
Observamos cómo en iguales condiciones OneMax consigue distribuir más herencia que Rastrigin.

Continuando con el estudio de las gráficas podemos ver que hay procesadores que contribuyen más a las soluciones que otras.

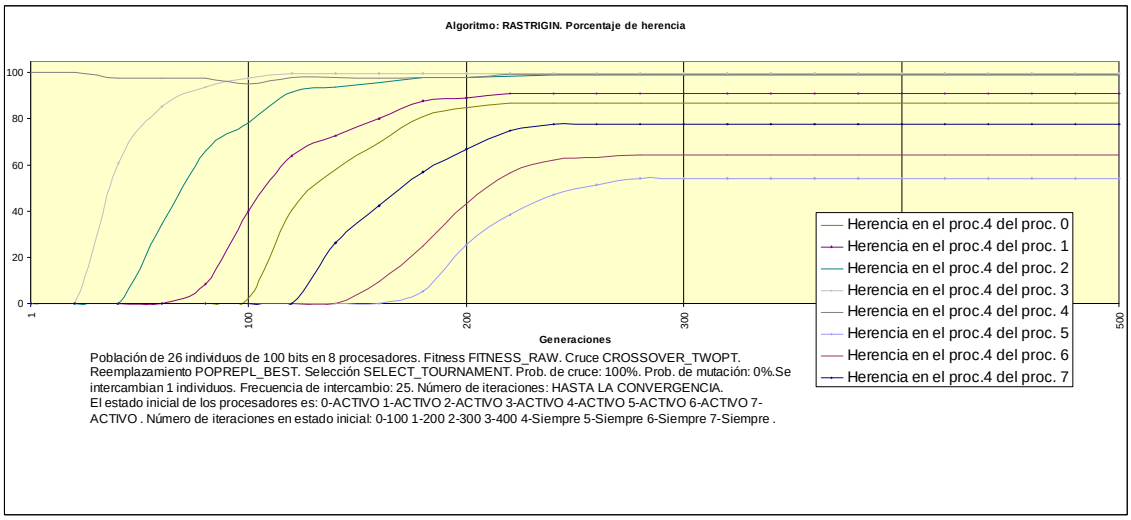
Pasemos a estudiar la gráfica que tenemos inmediatamente debajo de este texto. Primero tengamos en cuenta que matamos los procesadores 0, 1, 2 y 3 en ese orden. Tengamos presente también la distribución en anillo que realizamos. Teniendo en cuenta que estamos observando la gráfica correspondiente a la herencia del procesador 4. Podemos observar como se distribuye rápidamente en este procesador la herencia proveniente de los 5, 6 y 7, los cuales son los más alejados en el anillo de intercambio. De este modo vemos que la herencia de estos procesadores llega a influir más que la del 0 o el 1 que están más cerca. Esto nos hace pensar que, por ejemplo, el procesador 0 tuviese quizá buenos esquemas que enviar pero al sufrir un fallo no pudo enviarlos y por este motivo la herencia de los procesadores 5, 6 y 7 se propaga con tanta rapidez. De no haber fallado este procesador puede que hubiese colaborado con buenos esquemas y nos hubiese hecho conseguir mejores resultados y mayor tolerancia a los fallos que la que hemos obtenido (ver TABLA 1)

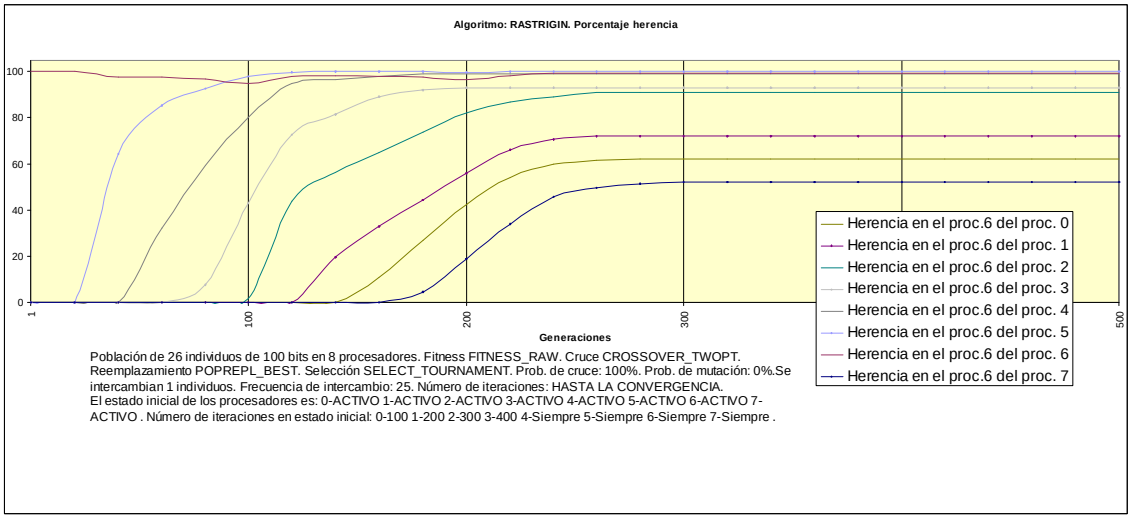


Para esta gráfica correspondiente al procesador 6 se puede observar algo parecido en la transmisión de la herencia a lo explicado en la anterior.

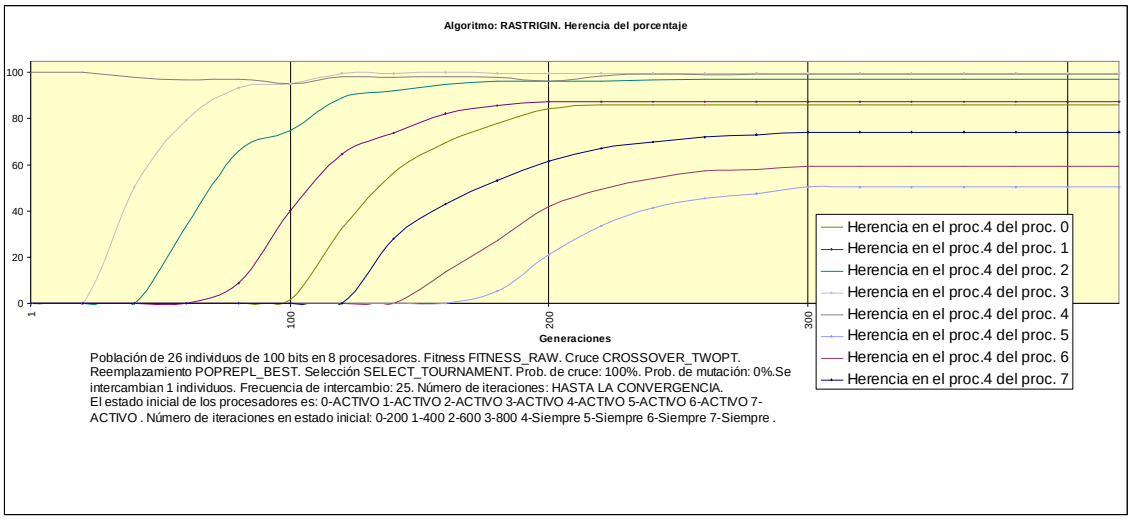


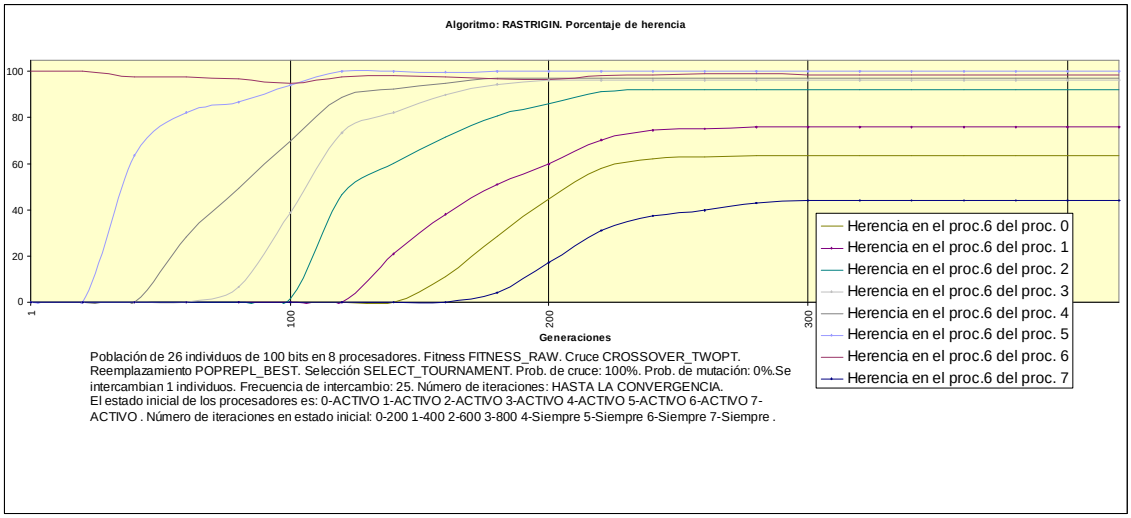
Hagamos ahora una comparación con las mismas gráficas de herencia pero con fallos cada 100 generaciones.





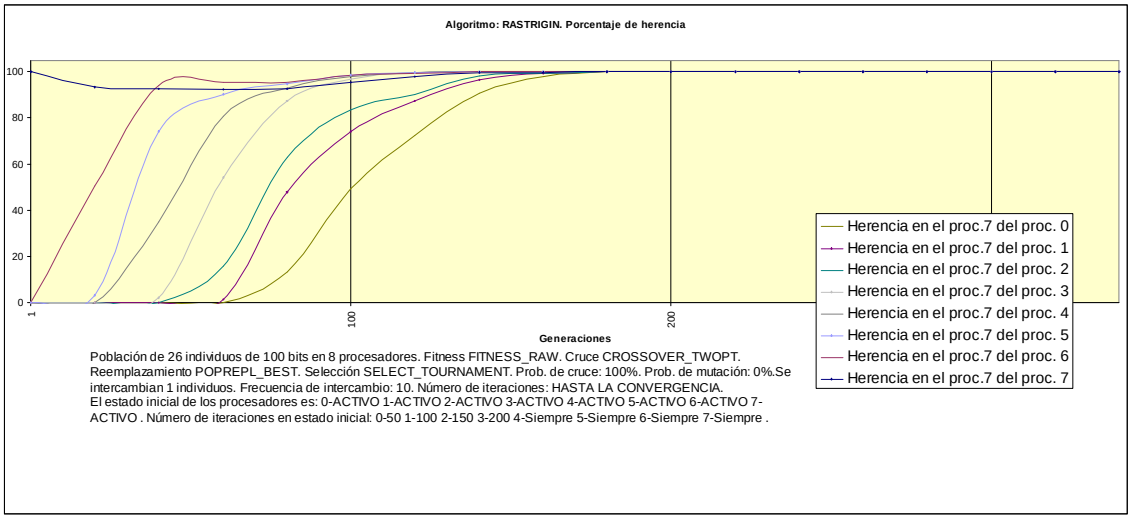
Y por último estudiemos las gráficas con fallos cada 200 generaciones.





Vemos aquí como a todos los procesadores le da tiempo suficiente a distribuir su herencia de mejor manera que con fallos cada 50 y cada 100, corroborando así que cuanto mayor es la aportación de herencia de todos los procesadores, mejores resultados se obtiene. (Véase la TABLA 3 y los resultados para 8 procesadores sin mutación).

Podemos paliar este comportamiento aumentando la frecuencia de intercambio, de modo que conseguimos que tarde o temprano se propague completamente la herencia antes de producirse algún fallo. Observemos la siguiente gráfica en la que mostramos la herencia del procesador 7 para un periodo de intercambio de 10 generaciones.



Ahora como la herencia de los procesadores se distribuye de manera parecida. Ninguna herencia se queda estancada como le sucedía a la del 0. El procesador 0 esta vez consigue distribuir su herencia al resto de procesadores colaborando así con los esquemas que antes no pudo. Con la aportación del procesador conseguimos mejores resultados que anteriormente.

Como conclusión, si tenemos problemas de este tipo y tenemos algún procesador que sufre algún fallo debemos intentar que la ejecución dure más. O dicho de otra forma, si tenemos dudas sobre la fiabilidad de los procesadores podemos aumentar un poco la mutación y programar una condición de parada que asegure la ejecución de un mayor número de programas. Esto se podría aplicar a las típicas ejecuciones nocturnas que se producen en todo centro de investigación, en el que no suele haber tanto problema de tiempo.

7.4. Función Schwefel:

A continuación detallaremos un pequeño índice en el que se comentaran las pruebas realizadas para este experimento:

- **Fallo cada 50 generaciones:** hemos provocado el fallo de un procesador cada 50 generaciones. Más propiamente dicho, incomunicación ya que ese procesador sigue ejecutándose aisladamente y seguimos recopilando datos de su ejecución.

  Se ha repetido el experimento para 8 procesadores, 4 procesadores y 2 procesadores con y sin mutación.

  - *Con mutación:* Probabilidad de cruce 0,8. Probabilidad de mutación *0,002.*

  - *Sin mutación:* Probabilidad de cruce 1. Probabilidad de mutación 0.

  Para todos los casos se han considerado los periodos de intercambio cada 10, 25, 50 y 100 generaciones.

  Así mismo, todas estas pruebas se han repetido con las mismas condiciones sin forzar ningún fallo en los procesadores.

  Se puede observar los resultados en forma de gráfica en las FIGURA 1 y 2, así como un resumen de los resultados en la TABLA1.

- **Fallo cada 100 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones, pero esta vez se han provocado los fallos cada 100 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 3 y 4, así como un resumen de los resultados en la TABLA 2.

- **Fallo cada 200 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones pero esta vez se han provocado los fallos cada 200 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 5 y 6, así como un resumen de los resultados en la TABLA 3.

Para una mejor comprensión de las tablas procederemos a continuación a la explicación de cada una de las columnas:

- **Experimento:** resumen del experimento al que se refieren los resultados.

- **Intercambio:** periodo de intercambio utilizado en ese experimento.

- **Herencia de todos:** en esta columna se observará si todos los procesadores que quedan "vivos" al final de la ejecución han conseguido recibir herencia de todos los procesadores con los que se inicio la ejecución.

- **Separan en generación:** en esta columna se muestra a partir de qué generación las líneas de la gráfica correspondientes al experimento con fallos y aquéllas correspondientes al experimento sin fallos comienzan a separarse indicando tener algún problema por la "muerte" de algún procesador.

- **Generacion:** número de generaciones que ha ejecutado el experimento.

- **Proc Caidos:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos.

- **%Pob Muerta:** mostramos el porcentaje de población muerte en el momento en el que se empiezan a separar las líneas.

- **Proc Caidos Total:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos al finalizar la ejecución.

- **%Pob Muerta Total:** mostramos el porcentaje de población muerta en el momento en el que se empiezan a separar la líneas al finalizar la ejecución

- **Max Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

- **Max Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

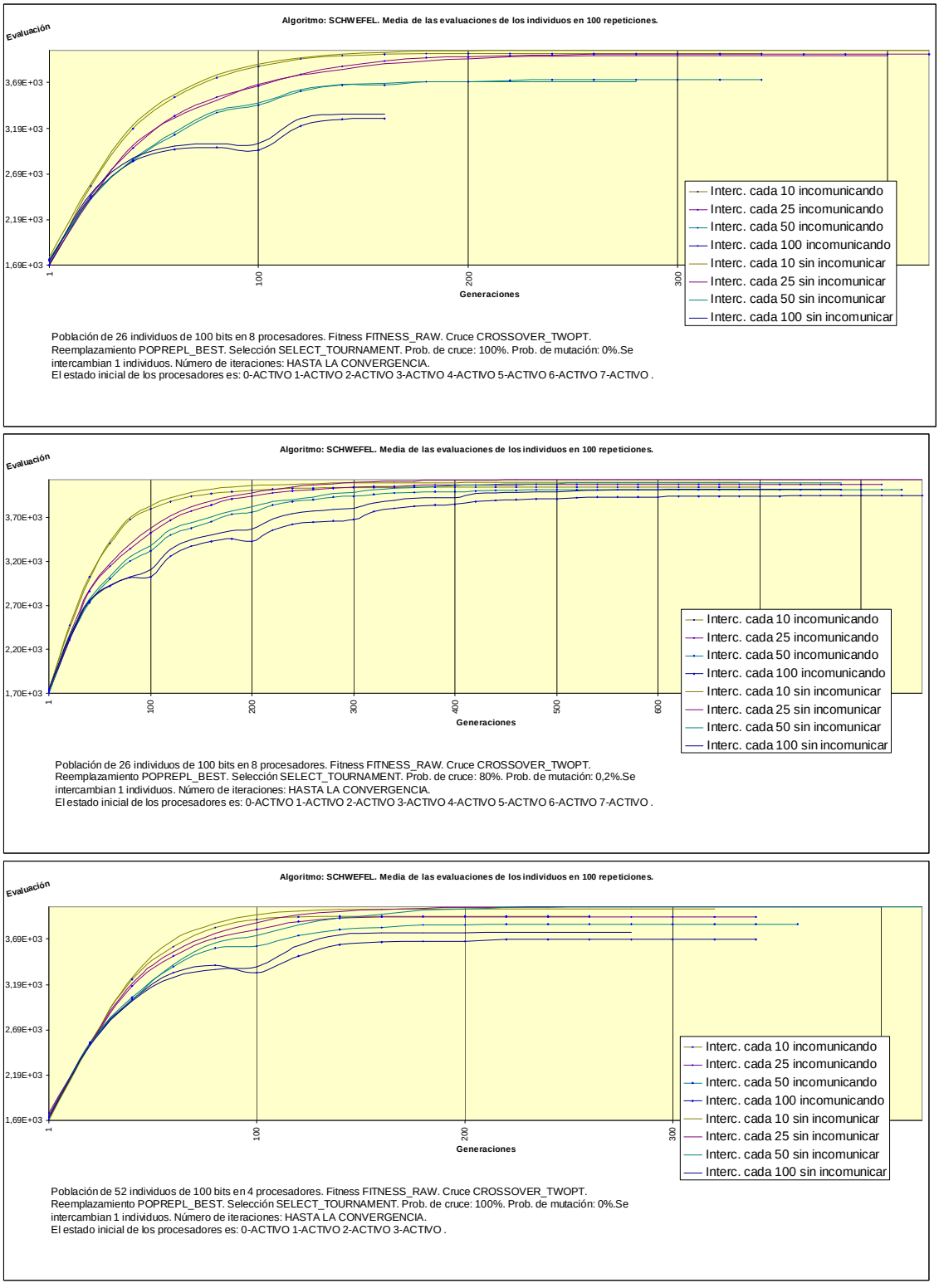
- **Tolerancia (% error):** diferencia entre el valor alcanzado con fallos en los procesadores y la ejecución sin fallos. Expresada en porcentaje.

- **Min Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

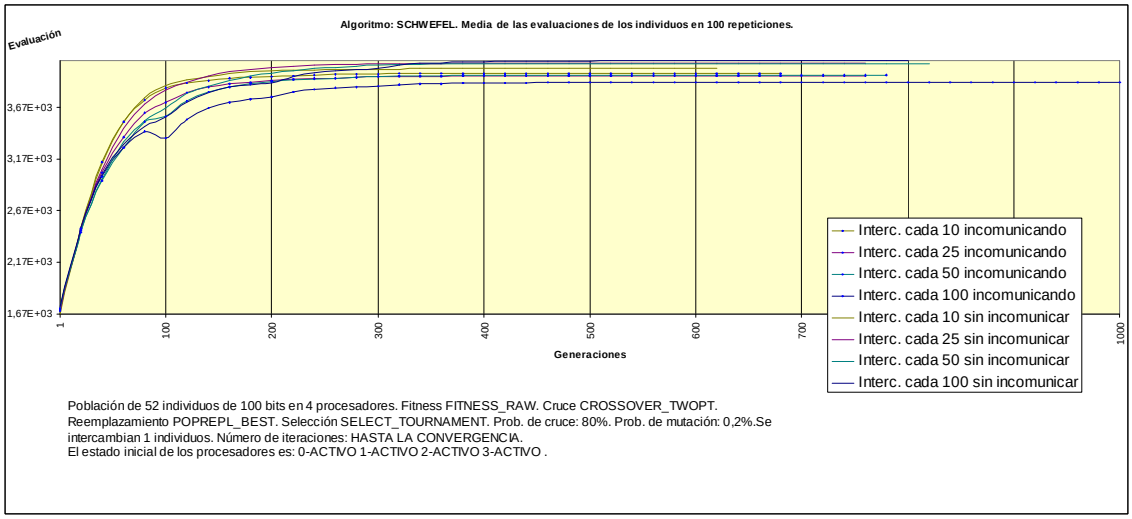
- **Min Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

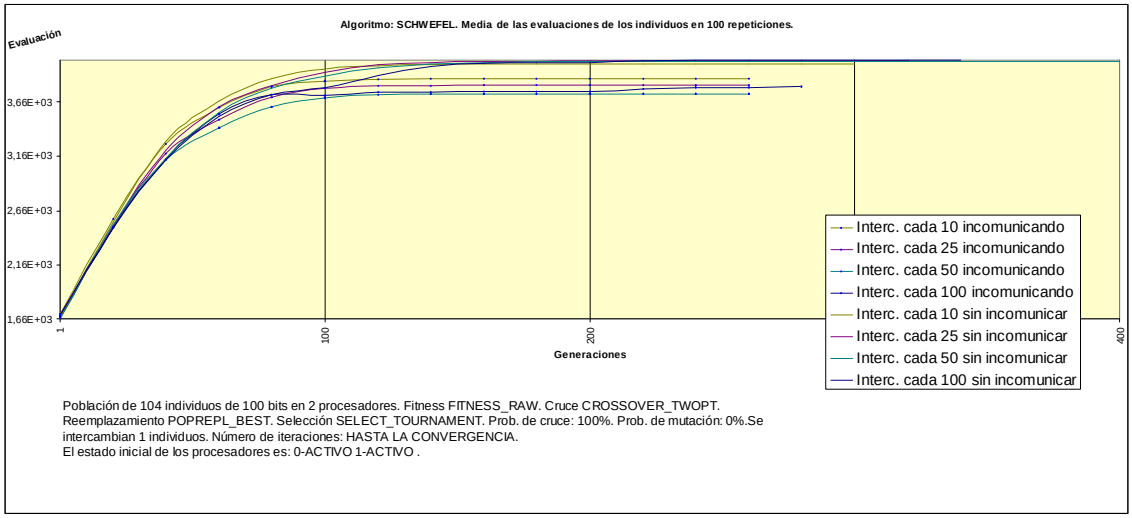
A continuación de cada tabla se muestra una gráfica en la que se indica la evolución del porcentaje de tolerancia según el periodo de intercambio y si el experimento fue realizado con mutación (cm) o sin mutación (sm).

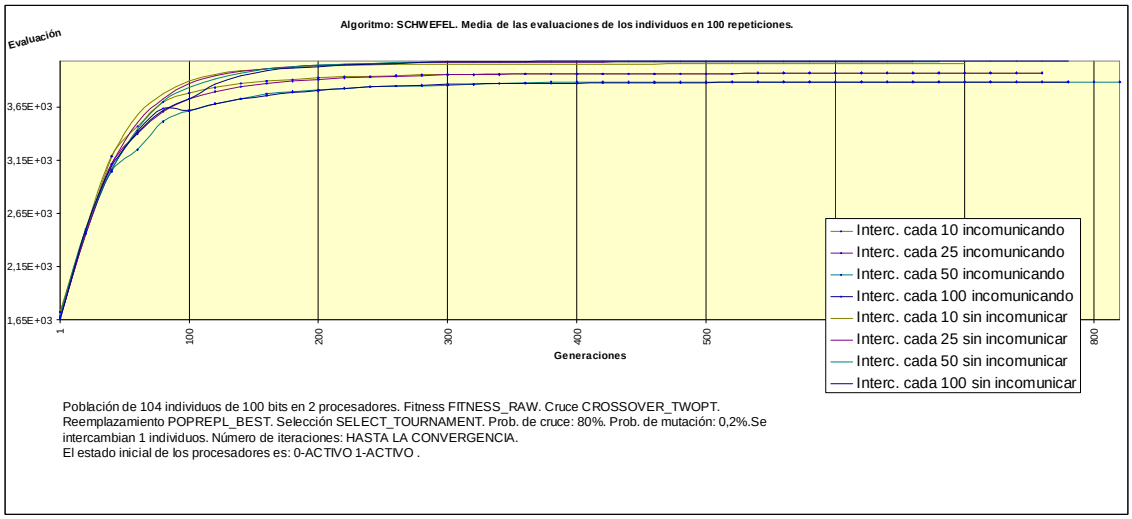
7.4.1. Fallo cada 50 generaciones



SCHWEFEL. .FIGURA 1 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 50 generaciones falla un procesador



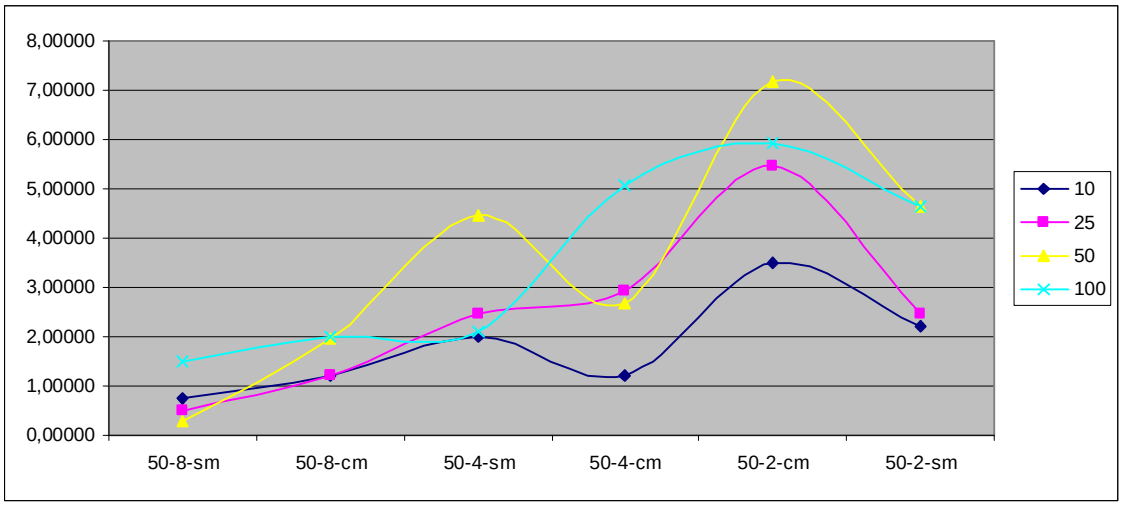




SCHWEFEL. FIGURA 2 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 50 generaciones falla un procesador

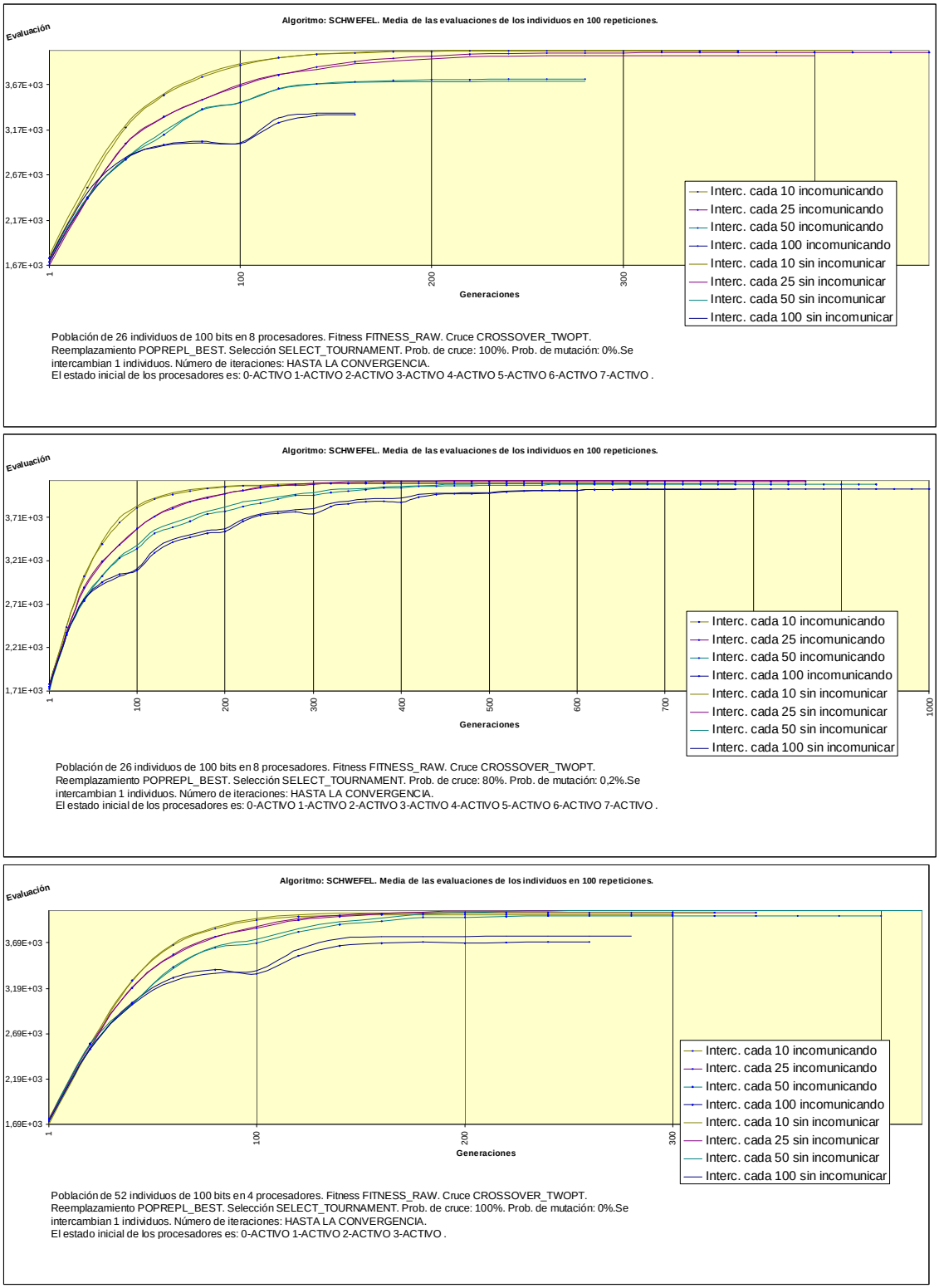
Experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
Schwefel 50 8 sin mutación	10	si	50	340	1	12,5	4	50,0	4020	4050	0,74074	4020	4050
	25	si	50	420	1	12,5	4	50,0	4010	3990	0,50125	3840	3670
	50	no	50	340	1	12,5	4	50,0	3720	3710	0,26954	2950	2770
	100	no	80	160	1	12,5	3	37,5	3300	3350	1,49254	2270	2240
Schwefel 50 8 con mutación	10	si	50	700	1	12,5	4	50,0	4050	4100	1,21951	4050	4100
	25	si	50	820	1	12,5	4	50,0	4080	4130	1,21065	4070	4100
	50	no	50	840	1	12,5	4	50,0	4020	4100	1,95122	3990	3950
	100	no	50	860	1	12,5	4	50,0	3950	4030	1,98511	3790	3620
Schwefel 50 4 sin mutación	10	si	50	260	1	25,0	2	50,0	3940	4020	1,99005	3940	4020
	25	si	50	340	1	25,0	2	50,0	3940	4040	2,47525	3940	4040
	50	no	50	360	1	25,0	2	50,0	3860	4040	4,45545	3850	3840
	100	no	50	340	1	25,0	2	50,0	3690	3770	2,12202	3380	3190
Schwefel 50 4 con mutación	10	si	100	680	2	50,0	2	50,0	4010	4060	1,23153	4010	4050
	25	si	50	760	1	25,0	2	50,0	3980	4100	2,92683	3980	4090
	50	no	50	780	1	25,0	2	50,0	3990	4100	2,68293	3990	4090
	100	no	50	1000	1	25,0	2	50,0	3920	4130	5,08475	3920	4070
Schwefel 50 2 sin mutación	10	si	50	260	1	50,0	1	50,0	3870	4010	3,49127	3870	4010
	25	si	50	260	1	50,0	1	50,0	3810	4030	5,45906	3810	4030
	50	no	50	260	1	50,0	1	50,0	3740	4030	7,19603	3730	4030
	100	no	50	280	1	50,0	1	50,0	3800	4040	5,94059	3750	3990
Schwefel 50 2 con mutación	10	si	50	760	1	50,0	1	50,0	3970	4060	2,21675	3970	4050
	25	si	50	760	1	50,0	1	50,0	3970	4070	2,45700	3960	4050
	50	no	50	820	1	50,0	1	50,0	3890	4080	4,65686	3890	4060
	100	no	50	780	1	50,0	1	50,0	3890	4080	4,65686	3880	4080

de 0,5 % a 1% de 0% a 0,5%

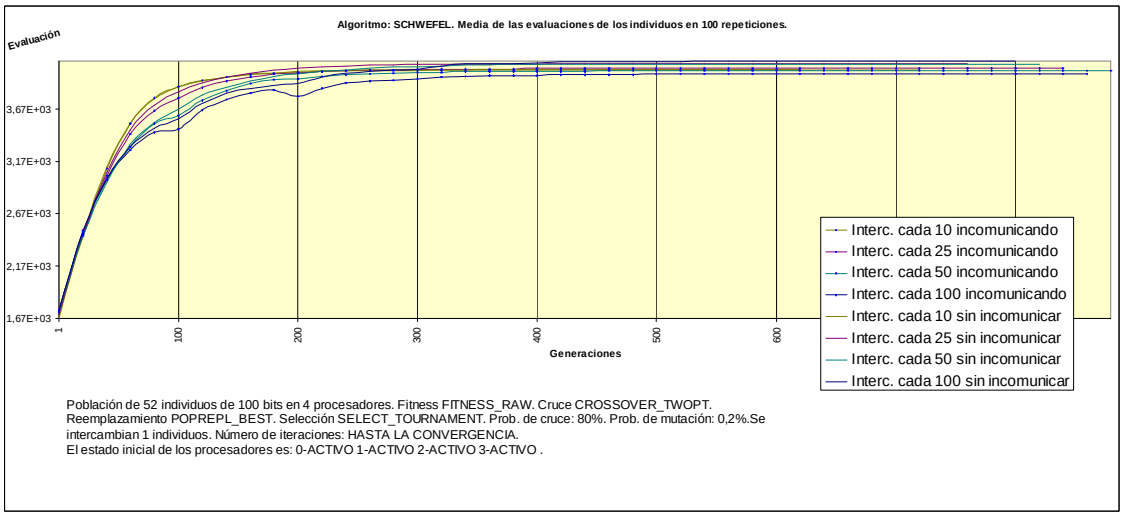


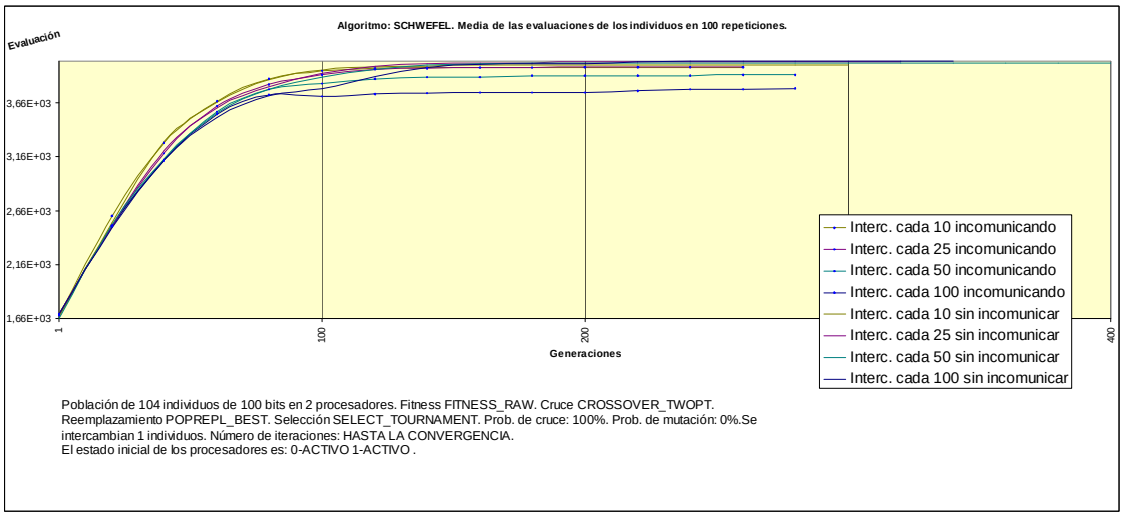
SCHWEFEL. TABLA 1. Resumen de resultados incomunicando cada 50 generaciones

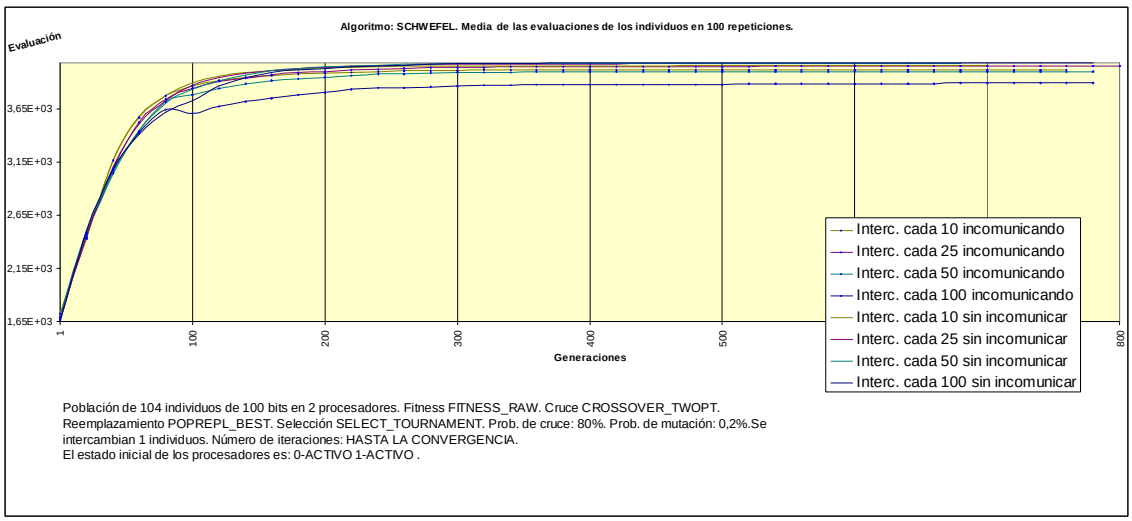
7.4.2. Fallo cada 100 generaciones



SCHWEFEL. .FIGURA 3 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 100 generaciones falla un procesador



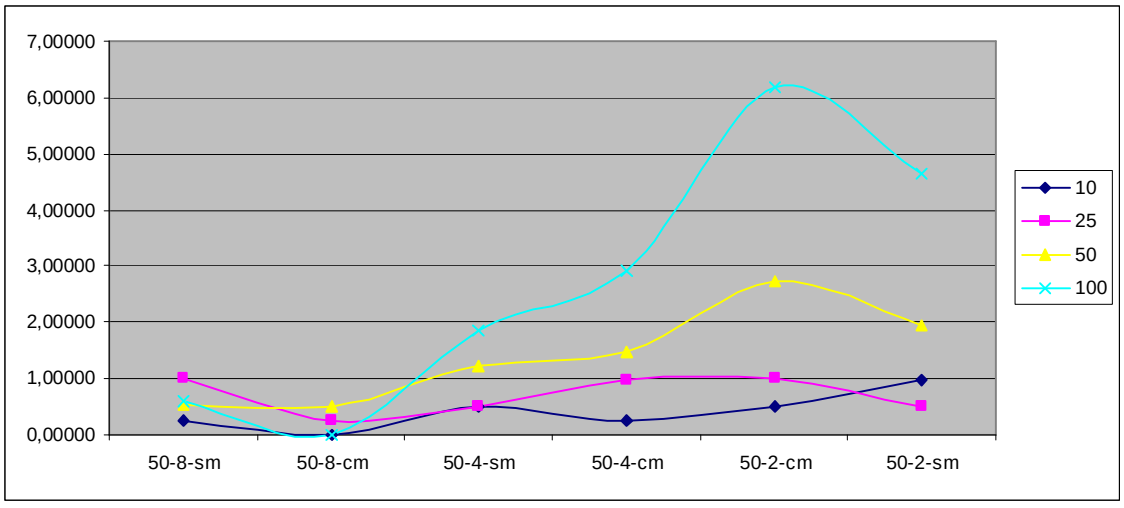




SCHWEFEL. FIGURA 4 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 100 generaciones falla un procesador

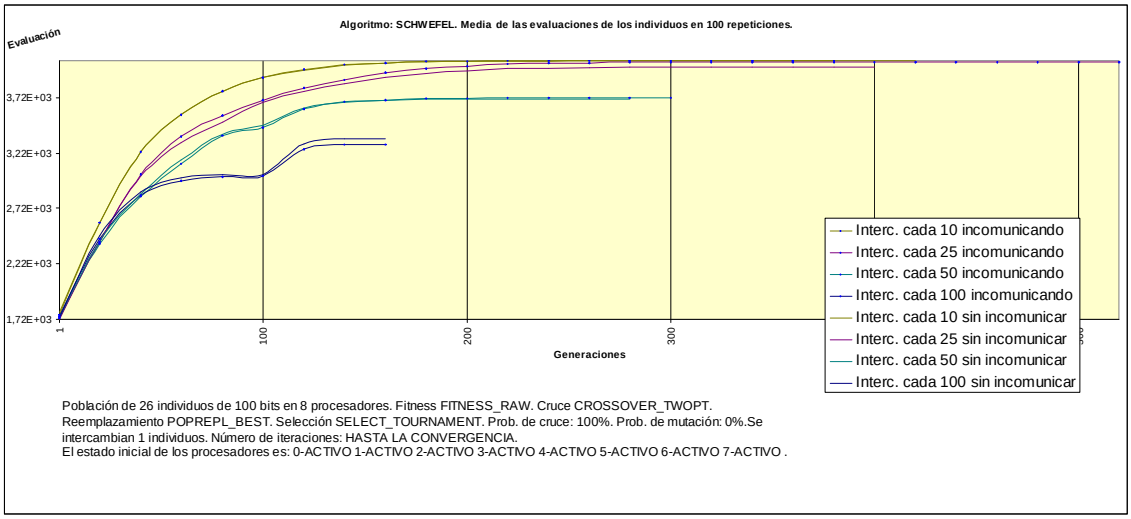
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
Schwefel1008 sin mutación	10	si		360			3	37,5	4040	4050	0,24691	4040	4050
	25	si		460			4	50,0	4030	3990	1,00251	3820	3670
	50	si		280			2	25,0	3730	3710	0,53908	2890	2770
	100	no		160			1	12,5	3330	3350	0,59701	2260	2240
Schwefe1008 con mutación	10	si		740			4	50,0	4100	4100	0,00000	4100	4100
	25	si		860			4	50,0	4120	4130	0,24213	4110	4100
	50	si	100	940	1	12,5	4	50,0	4080	4100	0,48780	4030	3950
	100	no	100	1000	1	12,5	4	50,0	4030	4030	0,00000	3860	3620
Schwefel 100 4 sin mutación	10	si		300			2	50,0	4000	4020	0,49751	4000	4020
	25	si		340			2	50,0	4020	4040	0,49505	4020	4020
	50	si	80	400	0	0,0	2	50,0	3990	4040	1,23762	3900	3840
	100	no	80	260	0	0,0	2	50,0	3700	3770	1,85676	3230	3190
Schwefel 100 4 con mutación	10	si		720			2	50,0	4050	4060	0,24631	4050	4050
	25	si	80	840	0	0,0	2	50,0	4060	4100	0,97561	4050	4090
	50	si	80	880	0	0,0	2	50,0	4040	4100	1,46341	4020	4090
	100	no	80	860	0	0,0	2	50,0	4010	4130	2,90557	3980	4070
Schwefel 100 2 sin mutación	10	si		260			1	50,0	3990	4010	0,49875	3990	4010
	25	si	120	260	1	50,0	1	50,0	3990	4030	0,99256	3990	4030
	50	si	80	280	0	0,0	1	50,0	3920	4030	2,72953	3900	4030
	100	no	80	280	0	0,0	1	50,0	3790	4040	6,18812	3750	3990
Schwefel 100 2 con mutación	10	si	80	760	0	0,0	1	50,0	4020	4060	0,98522	4010	4050
	25	si	80	800	0	0,0	1	50,0	4050	4070	0,49140	4050	4050
	50	si	50	780	0	0,0	1	50,0	4000	4080	1,96078	4000	4060
	100	no	50	780	0	0,0	1	50,0	3890	4080	4,65686	3880	4080

de 0,5 % a 1% de 0% a 0,5%

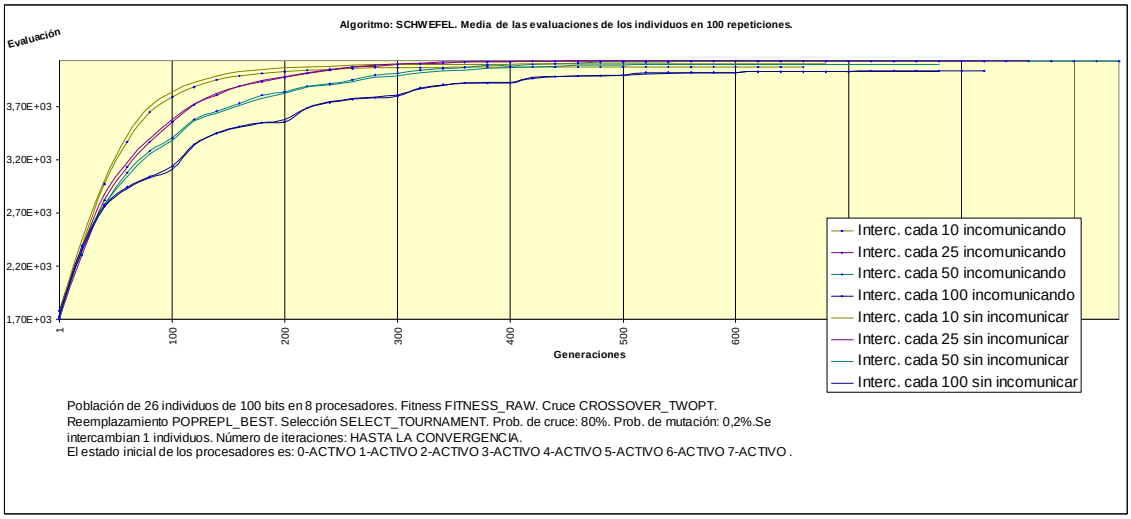


SCHWEFEL. TABLA 2. Resumen de resultados incomunicando cada 100 generaciones

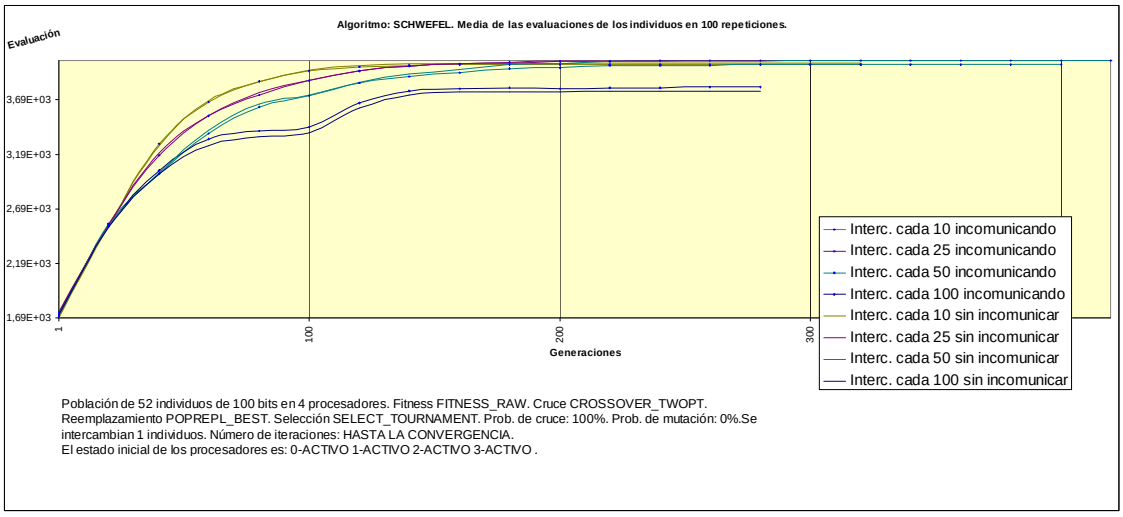
7.4.3. Fallo cada 200 generaciones



Población de 26 individuos de 100 bits en 8 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 100%. Prob. de mutación: 0%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO 4-ACTIVO 5-ACTIVO 6-ACTIVO 7-ACTIVO .

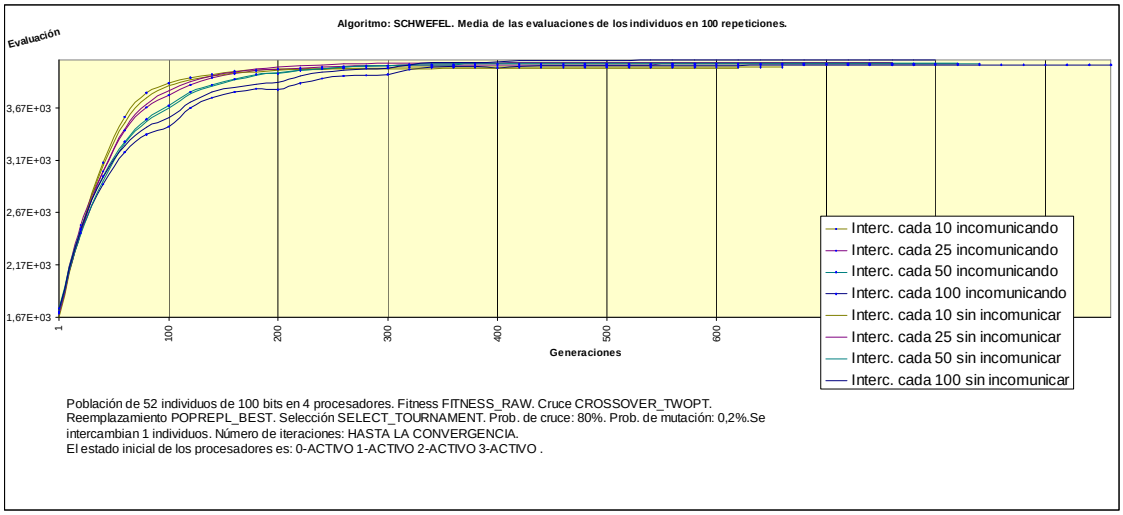


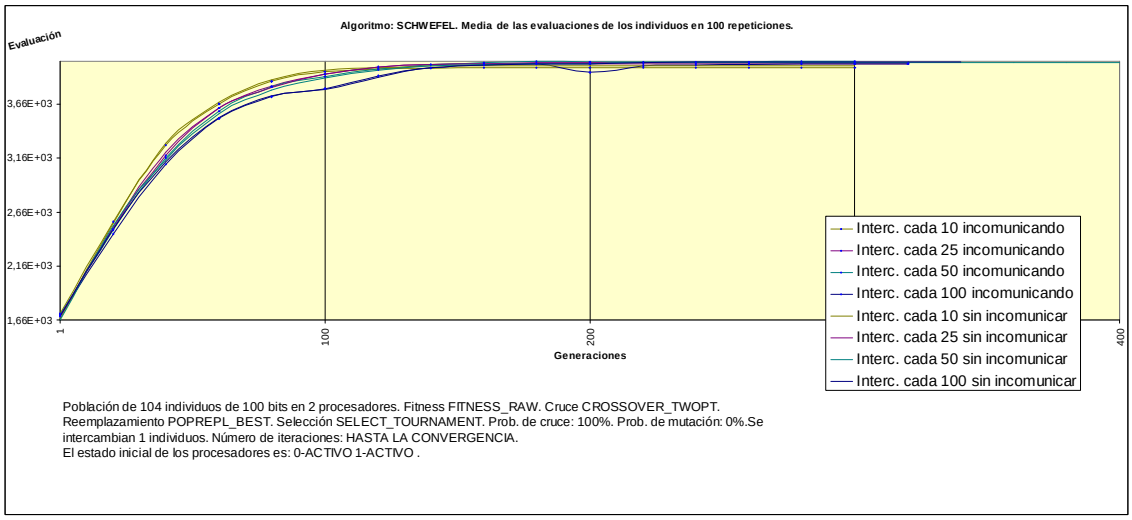
Población de 26 individuos de 100 bits en 8 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 80%. Prob. de mutación: 0,2%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO 4-ACTIVO 5-ACTIVO 6-ACTIVO 7-ACTIVO .

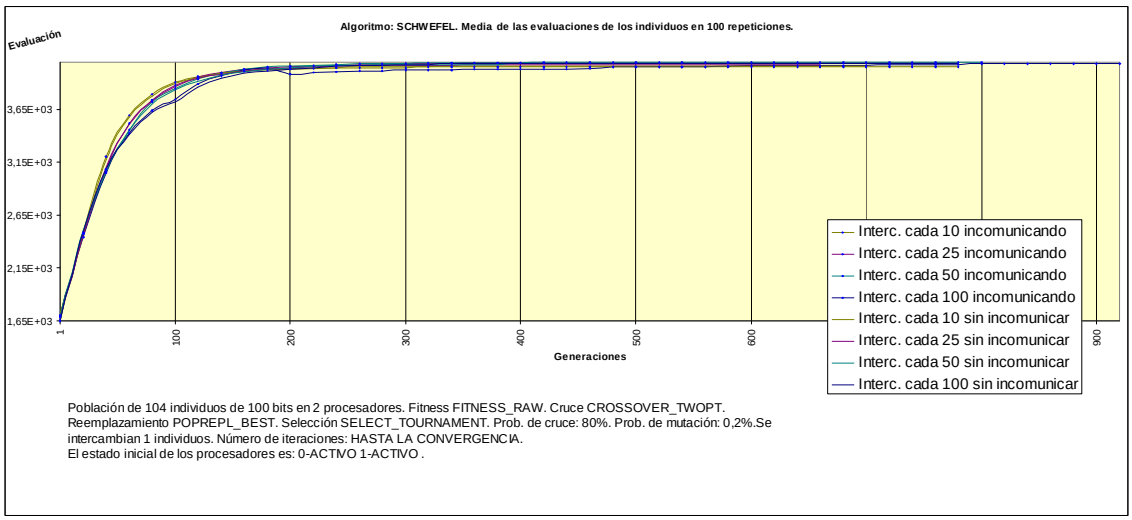


Población de 52 individuos de 100 bits en 4 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 100%. Prob. de mutación: 0%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO .

SCHWEFEL. .FIGURA 5 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 200 generaciones falla un procesador



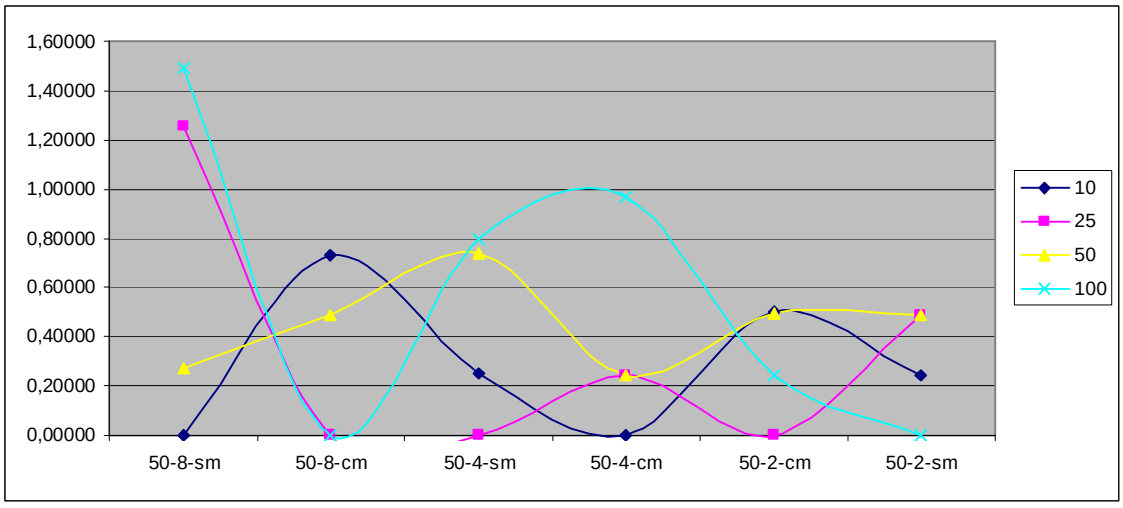




SCHWEFEL. .FIGURA 6 .4 procesadores sin mutación. 2 procesadores sin y con mutación. Cada 200 generaciones falla un procesador

experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
Schwefel 200 8 sin mutación	10	si		380			1	12,5	4050	4050	0,00000	4050	4050
	25	si	80	520	0	0,0	2	25,0	4040	3990	1,25313	3790	3670
	50	no		300			1	12,5	3720	3710	0,26954	2820	2770
	100	no	100	160	0	0,0	0	0,0	3300	3350	1,49254	2210	2240
Schwefel 200 8 con mutación	10	si	80	660	0	0,0	3	37,5	4070	4100	0,73171	4070	4100
	25	si		820			4	50,0	4130	4130	0,00000	4110	4100
	50	si		940			4	50,0	4120	4100	0,48780	4030	3950
	100	si		820			4	50,0	4030	4030	0,00000	3720	3620
Schwefel200 4 sin mutación	10	si		320			1	25,0	4010	4020	0,24876	4010	4020
	25	si		420			2	50,0	4040	4040	0,00000	4040	4040
	50	si	100	400	0	0,0	2	50,0	4010	4040	0,74257	3810	3840
	100	no	50	280	0	0,0	1	25,0	3800	3770	0,79576	3210	3190
Schwefel 200 4 con mutación	10	si		660			2	50,0	4060	4060	0,00000	4060	4050
	25	si		820			2	50,0	4090	4100	0,24390	4090	4090
	50	si		840			2	50,0	4090	4100	0,24390	4090	4090
	100	si	50	960	0	0,0	2	50,0	4090	4130	0,96852	4060	4070
Schwefel 200 2 sin mutación	10	si		300			1	50,0	3990	4010	0,49875	3990	4010
	25	si		320			1	50,0	4030	4030	0,00000	4030	4030
	50	si		300			1	50,0	4050	4030	0,49628	4040	4030
	100	si		320			1	50,0	4030	4040	0,24752	3930	3990
Schwefel 200 2 con mutación	10	si		780			1	50,0	4050	4060	0,24631	4050	4050
	25	si		780			1	50,0	4090	4070	0,49140	4080	4050
	50	si		800			1	50,0	4100	4080	0,49020	4100	4060
	100	si		920			1	50,0	4080	4080	0,00000	4030	4080

de 0,5 % a 1% de 0% a 0,5%

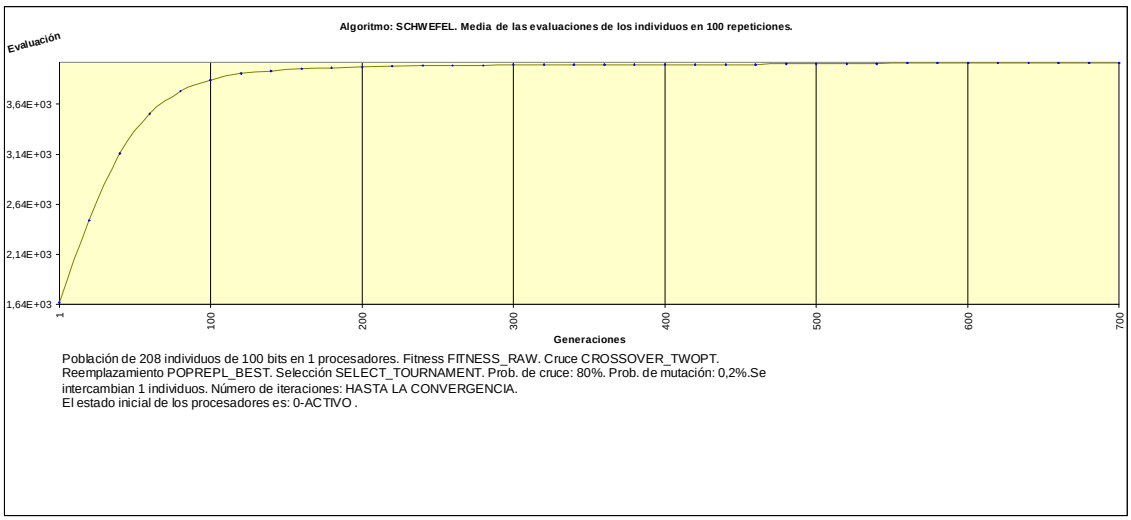


SCHWEFEL. TABLA 3. Resumen de resultados incomunicando cada 200 generaciones

Schwefel es una función bastante similar a Rastrigin, por lo que los resultados serán parecidos e intentaremos centrar en este estudio en sus diferencias.

Efectos de la mutación

Podemos obtener las mismas conclusiones que hasta ahora: el uso de la mutación ayuda a tolerar los fallos y ayuda a obtener mejores valores. Para ello puede verse las tablas mostradas mas arriba. A continuación vemos la gráfica correspondiente a un procesador, en la que vemos cómo también aquí podemos obtener buenos resultados sin utilizar el intercambio y utilizando en su lugar la mutación.



Tolerancia según el fallo de procesadores

Se vuelve a obtener las mismas conclusiones del análisis de las tablas. Conveniencia de distribuir la población en más islas, cuanto más espaciados sean los fallos obtendremos mejores resultados.

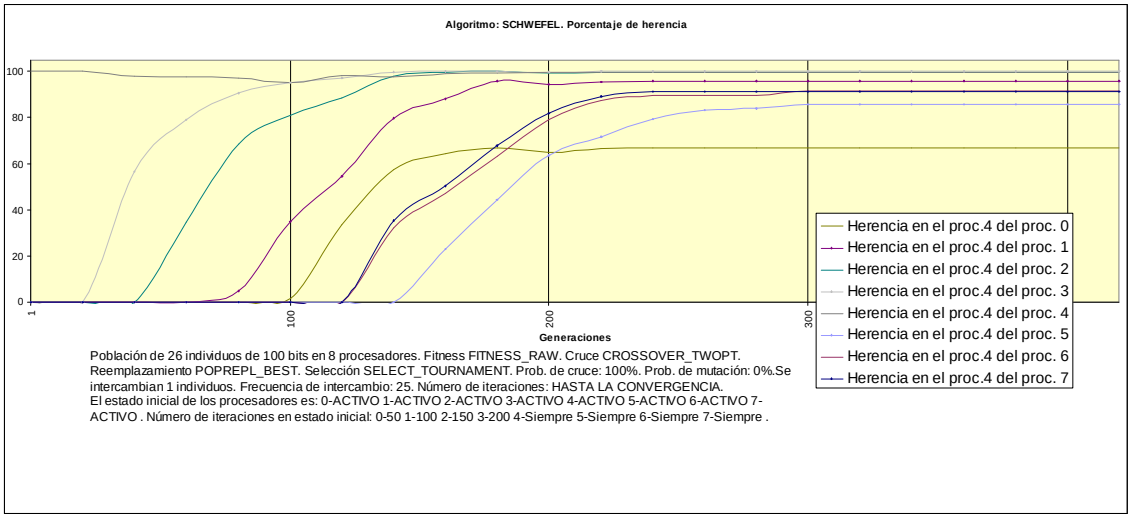
Efectos de la frecuencia de intercambio

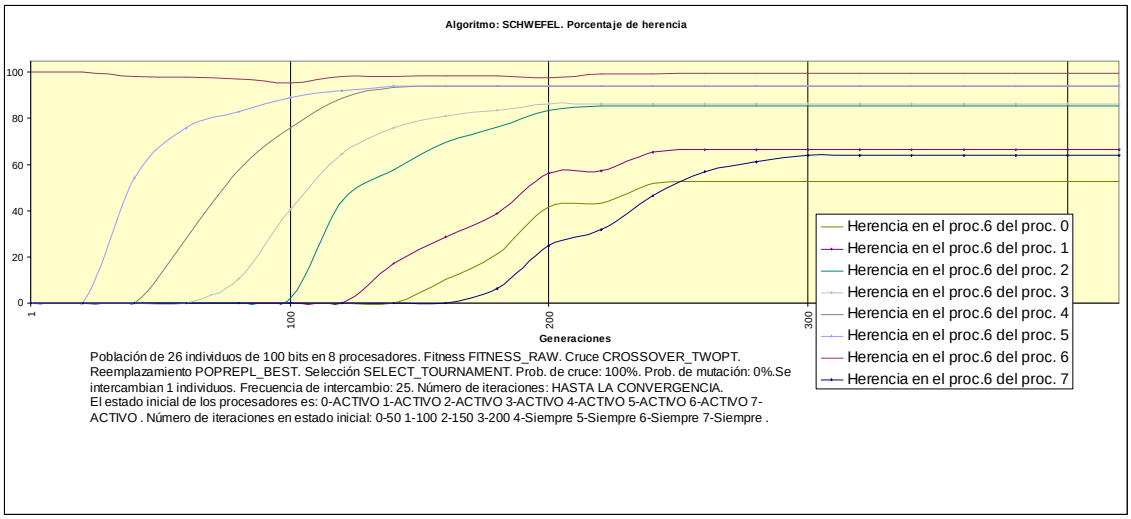
Podemos esta vez remitirnos a las gráficas para observar los efectos del intercambio, observemos la primera gráfica de la FIGURA 1. En ella vemos claramente como las líneas correspondientes a frecuencias de intercambio mayores están siempre por encima de las que se asocian con frecuencias menores, difuminándose estas diferencias con la disminución del número de islas.

Efectos de la herencia

En este punto de nuevo nos remitiremos a los experimentos realizados con la función de Rastrigin. Mostraremos a continuación algunas gráficas para justificar esta semejanza.

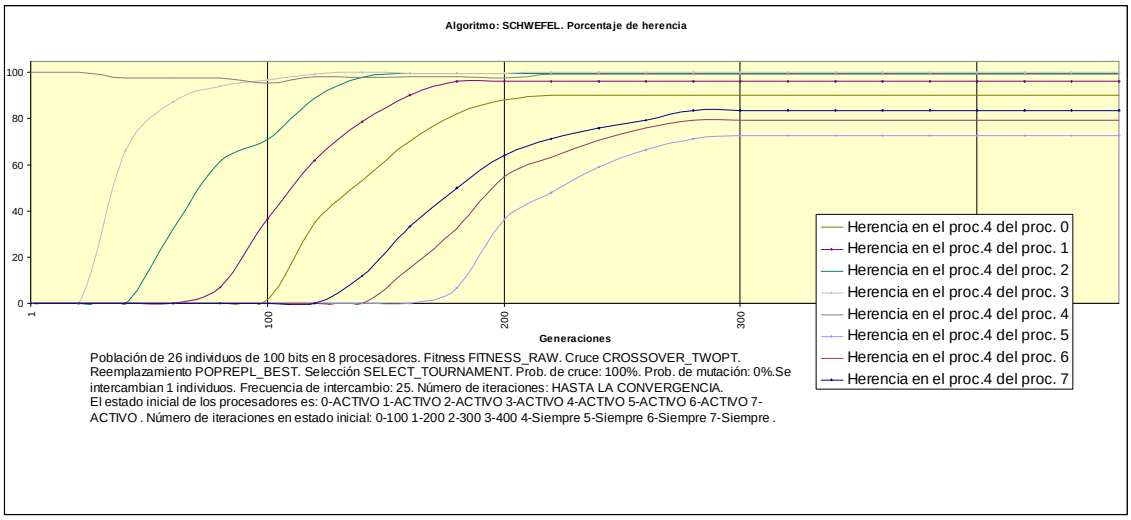
Si el fallo de un procesador es demasiado prematuro no se conseguirá distribuir correctamente la herencia al igual que pasaba en los otros ejemplos.



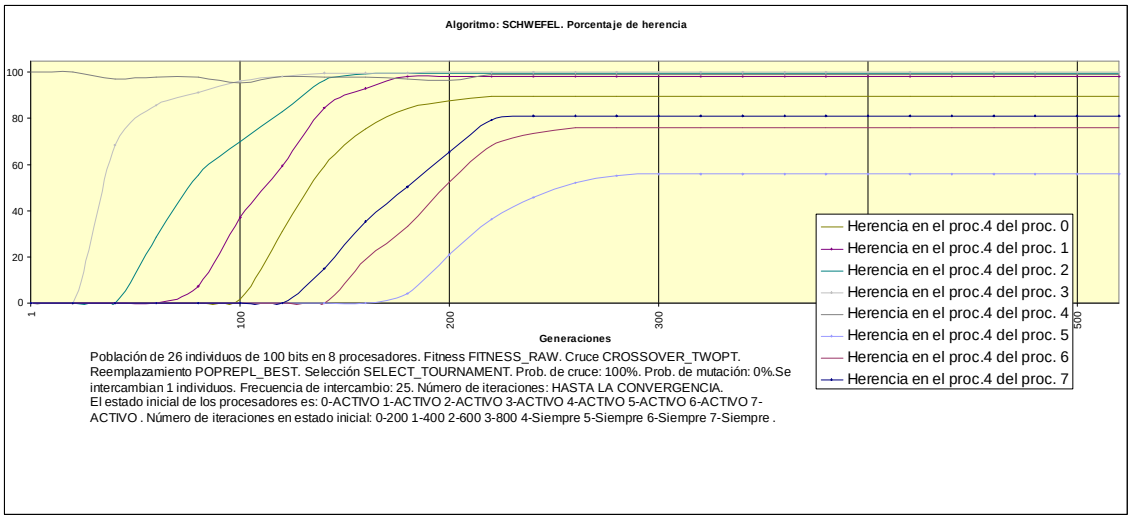


Fallos cada 50 generaciones

Observando las siguientes gráficas podemos ver también como esta situación mejora cuando provocamos muertes cada 100 y cada 200 generaciones.



Fallos cada 100 generaciones



Fallos cada 200 generaciones

7.5. Función FModal:

A continuación detallaremos un pequeño índice en el que se comentaran las pruebas realizadas para este experimento:

Fallo cada 50 generaciones: hemos provocado el fallo de un procesador cada 50 generaciones. Más propiamente dicho, incomunicación ya que ese procesador sigue ejecutándose aisladamente y seguimos recopilando datos de su ejecución.

Se ha repetido el experimento para 8 procesadores, 4 procesadores y 2 procesadores con y sin mutación.

Con mutación: Probabilidad de cruce 0,8. Probabilidad de mutación *0,002.*

Sin mutación: Probabilidad de cruce 1. Probabilidad de mutación 0.

Para todos los casos se han considerado los periodos de intercambio cada 10, 25, 50 y 100 generaciones.

Así mismo, todas estas pruebas se han repetido con las mismas condiciones sin forzar ningún fallo en los procesadores.

Se puede observar los resultados en forma de gráfica en las FIGURA 1 y 2, así como un resumen de los resultados en la TABLA1.

Fallo cada 100 generaciones: se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones, pero esta vez se han provocado los fallos cada 100 generaciones.

Se puede observar los resultados en forma de gráfica en las FIGURA 3 y 4, así como un resumen de los resultados en la TABLA 2.

Fallo cada 200 generaciones: se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones pero esta vez se han provocado los fallos cada 200 generaciones.

Se puede observar los resultados en forma de gráfica en las FIGURA 5 y 6, así como un resumen de los resultados en la TABLA 3.

Para una mejor comprensión de las tablas procederemos a continuación a la explicación de cada una de las columnas:

Experimento: resumen del experimento al que se refieren los resultados.

Intercambio: periodo de intercambio utilizado en ese experimento.

Herencia de todos: en esta columna se observará si todos los procesadores que quedan "vivos" al final de la ejecución han conseguido recibir herencia de todos los procesadores con los que se inicio la ejecución.

Separan en generación: en esta columna se muestra a partir de qué generación las líneas de la gráfica correspondientes al experimento con fallos y aquéllas correspondientes al experimento sin fallos comienzan a separarse indicando tener algún problema por la "muerte" de algún procesador.

Generacion: número de generaciones que ha ejecutado el experimento.

Proc Caidos: número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos.

%Pob Muerta: mostramos el porcentaje de población muerte en el momento en el que se empiezan a separar las líneas.

Proc Caidos Total: número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos al finalizar la ejecución.

%Pob Muerta Total: mostramos el porcentaje de población muerta en el momento en el que se empiezan a separar la líneas al finalizar la ejecución

Max Valor Matando: valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

Max Valor Sin Matar: valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

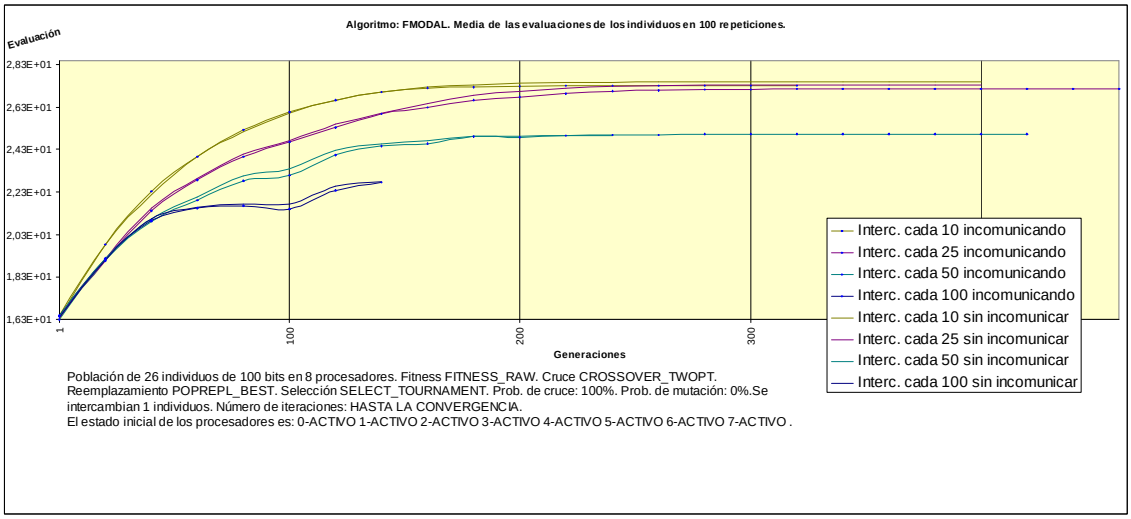
Tolerancia (% error): diferencia entre el valor alcanzado con fallos en los procesadores y la ejecución sin fallos. Expresada en porcentaje.

Min Valor Matando: valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

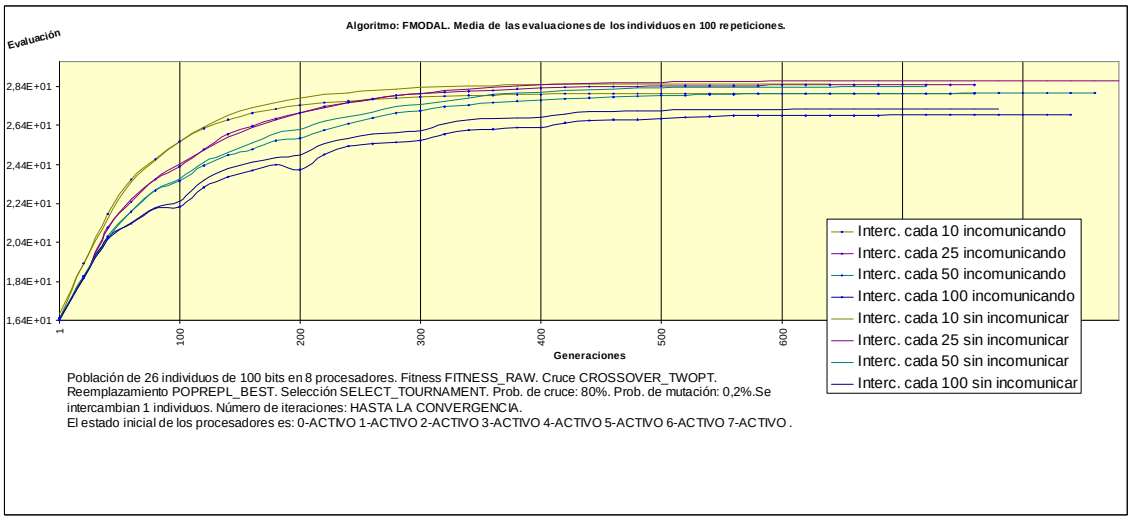
Min Valor Sin Matar: valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

A continuación de cada tabla se muestra una gráfica en la que se indica la evolución del porcentaje de tolerancia según el periodo de intercambio y si el experimento fue realizado con mutación (cm) o sin mutación (sm).

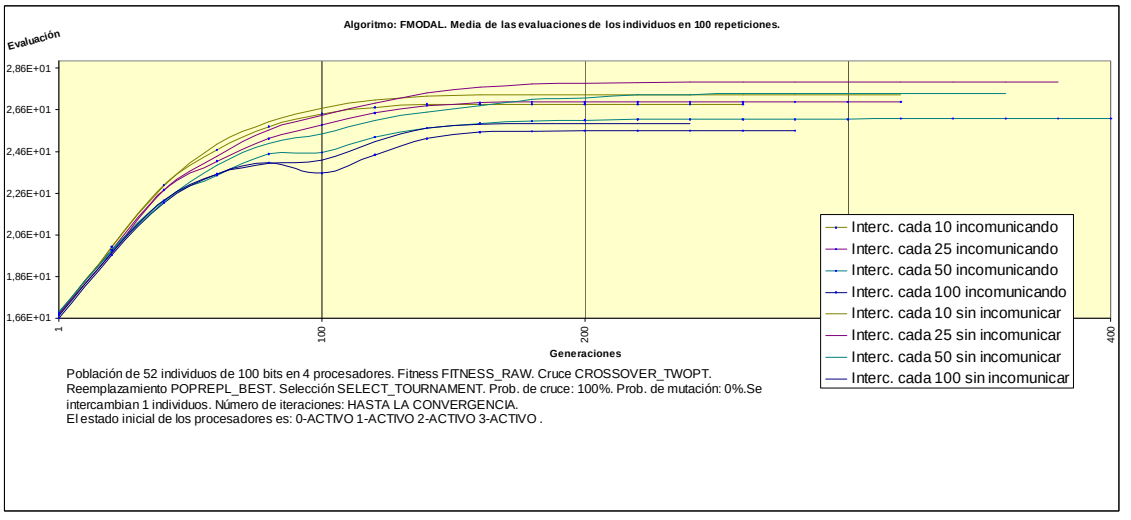
7.5.1. Fallo cada 50 generaciones



Población de 26 individuos de 100 bits en 8 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 100%. Prob. de mutación: 0%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO 4-ACTIVO 5-ACTIVO 6-ACTIVO 7-ACTIVO .

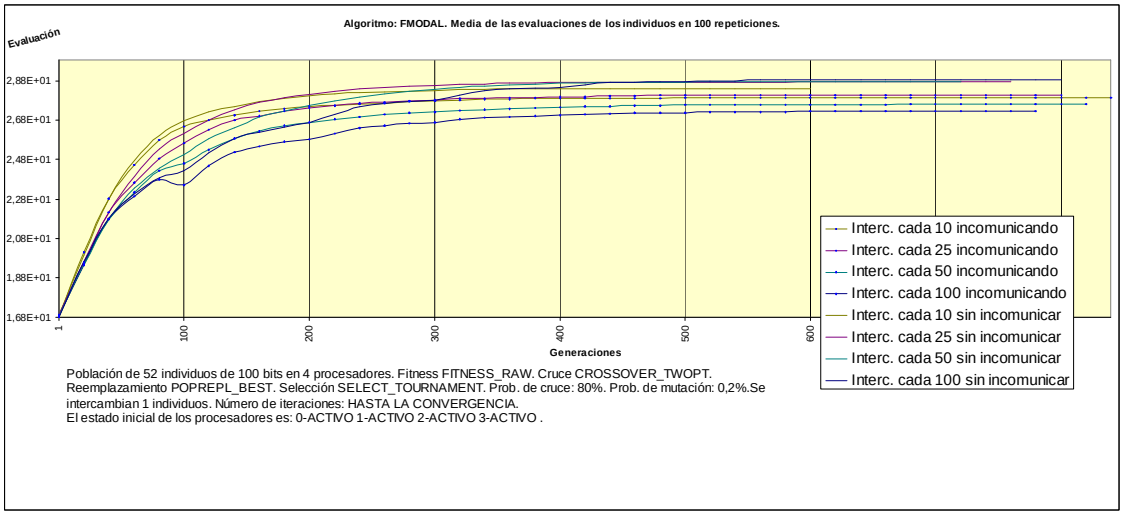


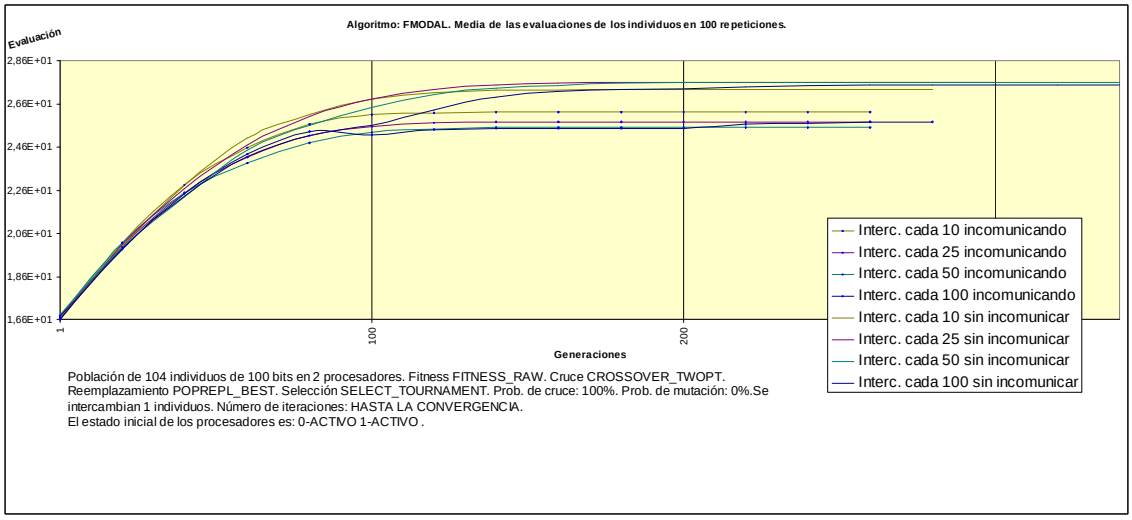
Población de 26 individuos de 100 bits en 8 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 80%. Prob. de mutación: 0,2%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO 4-ACTIVO 5-ACTIVO 6-ACTIVO 7-ACTIVO .

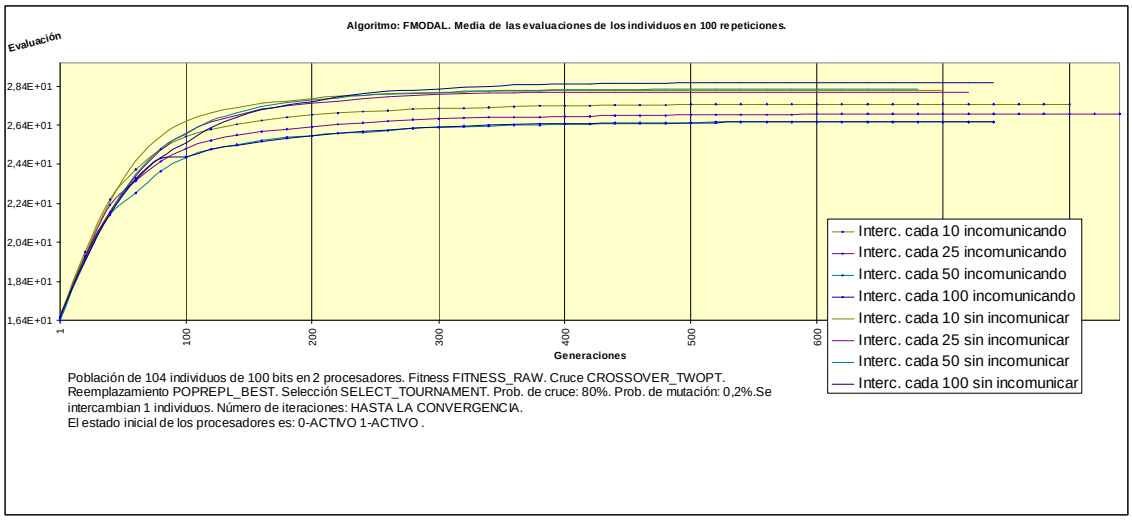


Población de 52 individuos de 100 bits en 4 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 100%. Prob. de mutación: 0%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO .

FMODAL. .FIGURA 1 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 50 generaciones falla un procesador



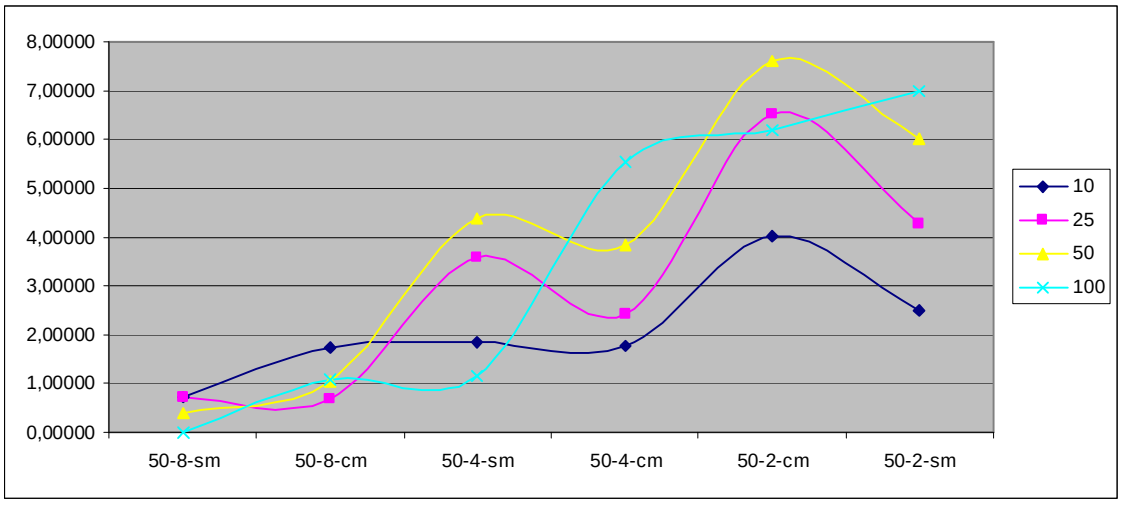




FMODAL. FIGURA 2 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 50 generaciones falla un procesador

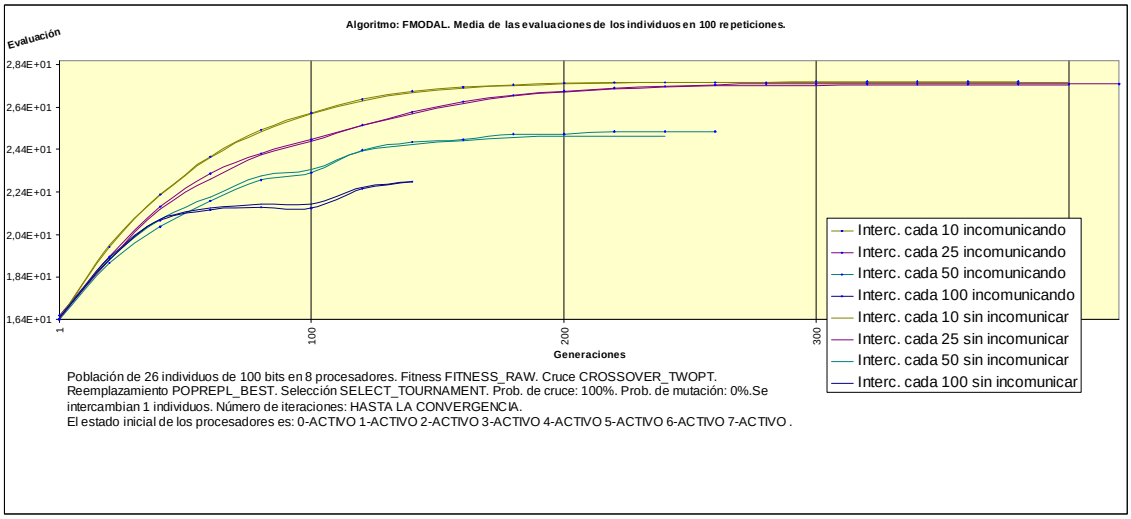
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
fmodal 50 8 sin mutación	10	si	200	320	4	50,0	4	50,0	27,3	27,5	0,72727	27,3	27,5
	25	si	140	460	2	25,0	4	50,0	27,2	27,4	0,72993	26,6	25,5
	50	no	50	420	1	12,5	4	50,0	25,1	25	0,40000	21,6	20,9
	100	no	50	140	1	12,5	2	25,0	22,8	22,8	0,00000	18,3	18
fmodal 50 8 con mutación	10	si	100	760	2	25,0	4	50,0	28,1	28,6	1,74825	28,1	28,6
	25	si	200	760	4	50,0	4	50,0	28,5	28,7	0,69686	28,5	28,3
	50	no	140	860	2	25,0	4	50,0	28,1	28,4	1,05634	27,7	26,8
	100	no	80	840	1	12,5	4	50,0	27	27,3	1,09890	25,6	24,5
fmodal 50 4 sin mutación	10	si	50	260	1	25,0	2	50,0	26,8	27,3	1,83150	26,8	27,3
	25	si	50	320	1	25,0	2	50,0	26,9	27,9	3,58423	26,9	27,9
	50	no	50	400	1	25,0	2	50,0	26,1	27,3	4,39560	26,1	25,9
	100	no	50	280	1	25,0	2	50,0	25,6	25,9	1,15830	23,8	23,1
fmodal 50 4 con mutación	10	si	60	840	1	25,0	2	50,0	27,9	28,4	1,76056	27,9	28,4
	25	si	60	800	1	25,0	2	50,0	28	28,7	2,43902	28	28,5
	50	no	80	820	1	25,0	2	50,0	27,6	28,7	3,83275	27,5	28,5
	100	no	80	780	1	25,0	2	50,0	27,2	28,8	5,55556	27,1	27,9
fmodal 50 2 sin mutación	10	si	50	260	1	50,0	1	50,0	26,2	27,3	4,02930	26,2	27,3
	25	si	50	260	1	50,0	1	50,0	25,8	27,6	6,52174	25,8	27,6
	50	no	50	260	1	50,0	1	50,0	25,5	27,6	7,60870	25,5	27,6
	100	no	50	280	1	50,0	1	50,0	25,8	27,5	6,18182	25,5	27
fmodal 50 2 con mutación	10	si	50	800	1	50,0	1	50,0	27,5	28,2	2,48227	27,5	28,2
	25	si	50	840	1	50,0	1	50,0	27	28,2	4,25532	27	28
	50	no	50	740	1	50,0	1	50,0	26,6	28,3	6,00707	26,2	28,1
	100	no	50	700	1	50,0	1	50,0	26,6	28,6	6,99301	26,6	28,4

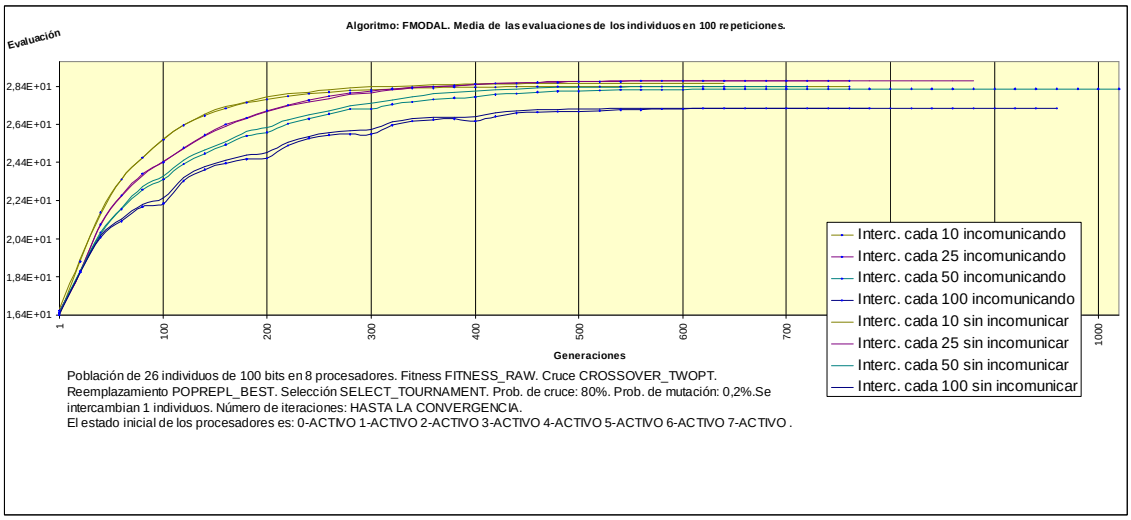
de 0,5 % a 1% de 0% a 0,5%

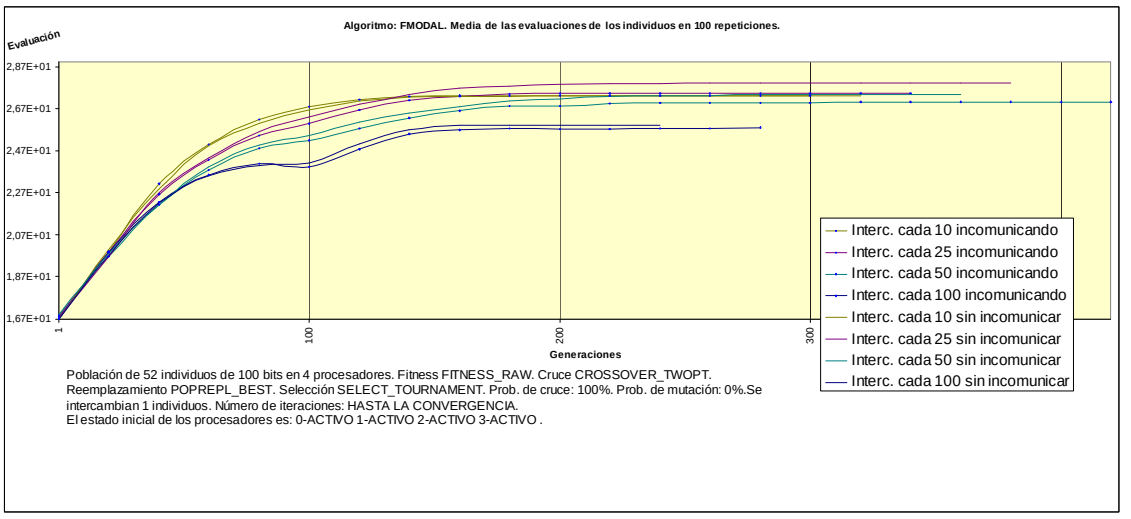


FMODAL. TABLA 1. Resumen de resultados incomunicando cada 50 generaciones

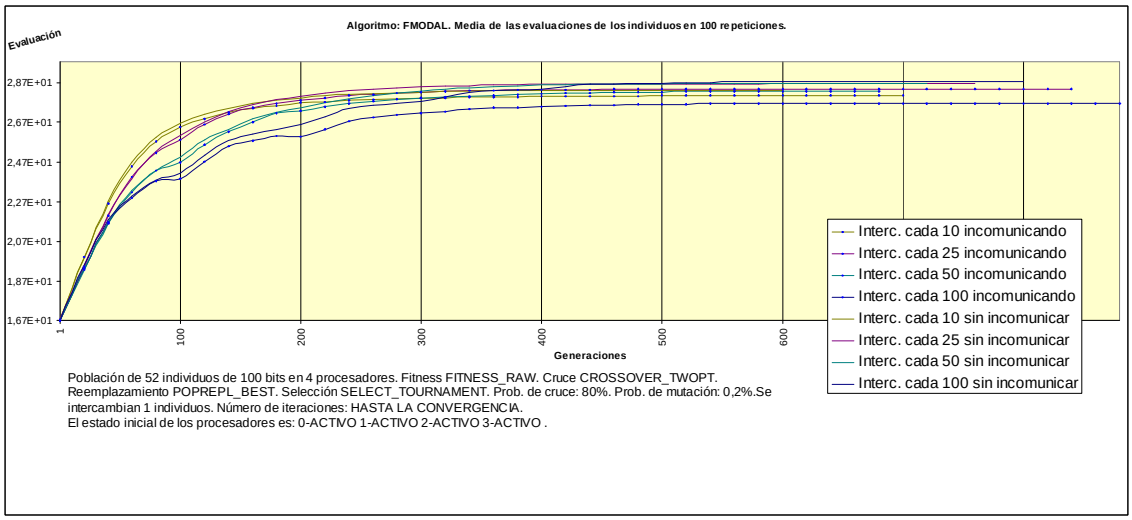
7.5.2. Fallo cada 100 generaciones



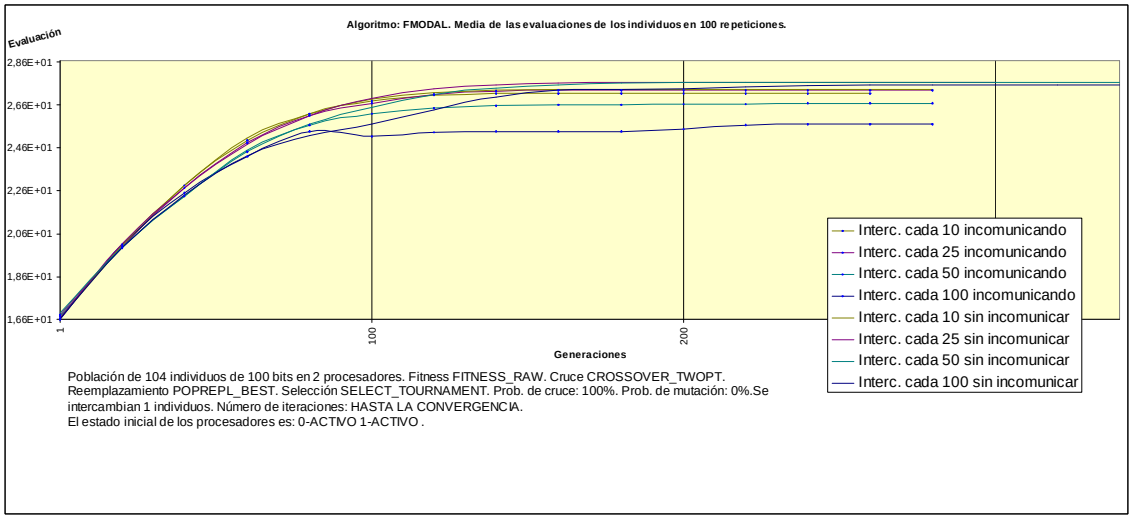




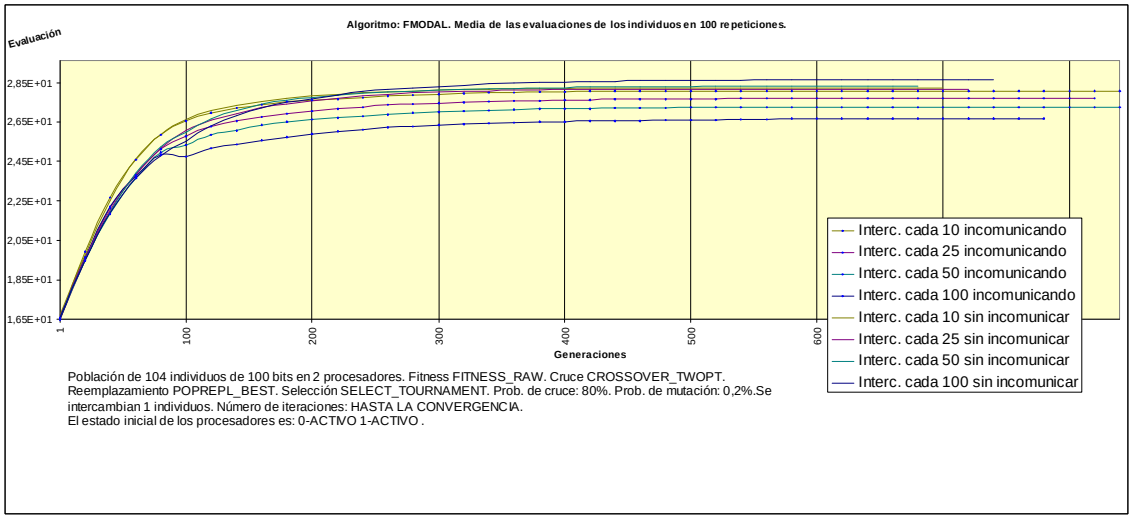
FMODAL. .FIGURA 3 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 100 generaciones falla un procesador



Población de 52 individuos de 100 bits en 4 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 80%. Prob. de mutación: 0,2%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO .



Población de 104 individuos de 100 bits en 2 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 100%. Prob. de mutación: 0%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO .

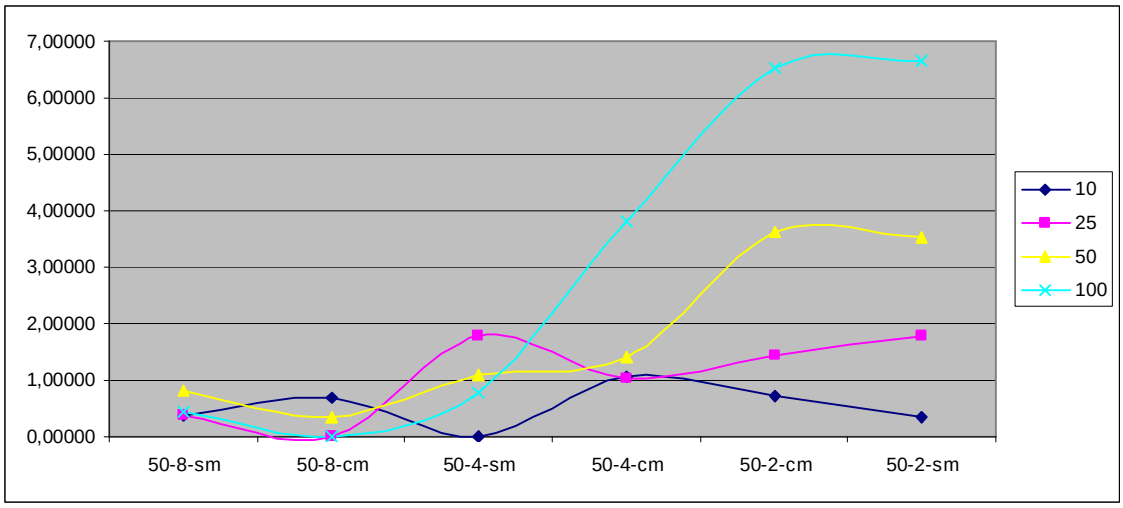


Población de 104 individuos de 100 bits en 2 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 80%. Prob. de mutación: 0,2%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO .

FMODAL. FIGURA 4 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 100 generaciones falla un procesador

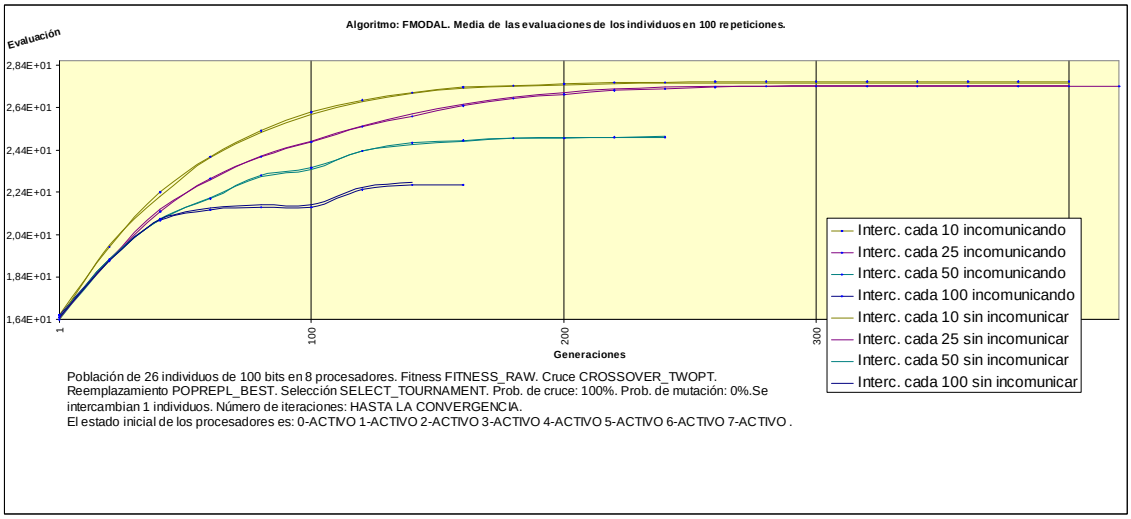
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
fmodal 100 8 sin mutación	10	si		400			4	50,0	27,6	27,5	0,36364	27,6	27,5
	25	si		400			4	50,0	27,5	27,4	0,36496	26,2	25,5
	50	no	200	260	2	25,0	2	25,0	25,2	25	0,80000	21,1	20,9
	100	no	100	140	1	12,5	1	12,5	22,9	22,8	0,43860	18,4	18
fmodal 100 8 con mutación	10	si		760			4	50,0	28,4	28,6	0,69930	28,4	28,6
	25	si		760			4	50,0	28,7	28,7	0,00000	28,7	28,3
	50	si	100	1020	1	12,5	4	50,0	28,3	28,4	0,35211	27,5	26,8
	100	no	100	960	1	12,5	4	50,0	27,3	27,3	0,00000	25,4	24,5
fmodal 100 4 sin mutación	10	si		300			2	50,0	27,3	27,3	0,00000	27,3	27,3
	25	si	100	340	1	25,0	2	50,0	27,4	27,9	1,79211	27,4	27,9
	50	si	100	420	1	25,0	2	50,0	27	27,3	1,09890	26,4	25,9
	100	no	100	280	1	25,0	2	50,0	25,7	25,9	0,77220	23,4	23,1
fmodal 100 4 con mutación	10	si	120	560	1	25,0	2	50,0	28,1	28,4	1,05634	28,1	28,4
	25	si	140	740	1	25,0	2	50,0	28,4	28,7	1,04530	28,4	28,5
	50	si	100	620	1	25,0	2	50,0	28,3	28,7	1,39373	28,2	28,5
	100	no	100	880	1	25,0	2	50,0	27,7	28,8	3,81944	27,5	27,9
fmodal 100 2 sin mutación	10	si		700			1	50,0	27,1	27,3	0,73260	27,1	27,3
	25	si	100	840	1	50,0	1	50,0	27,2	27,6	1,44928	27,2	27,6
	50	si	100	620	1	50,0	1	50,0	26,6	27,6	3,62319	26,5	27,6
	100	no	100	900	1	50,0	1	50,0	25,7	27,5	6,54545	25,3	27
fmodal 100 2 con mutación	10	si		840			1	50,0	28,1	28,2	0,35461	28,1	28,2
	25	si	120	820	1	50,0	1	50,0	27,7	28,2	1,77305	27,7	28
	50	si	100	840	1	50,0	1	50,0	27,3	28,3	3,53357	27,2	28,1
	100	no	100	780	1	50,0	1	50,0	26,7	28,6	6,64336	26,6	28,4

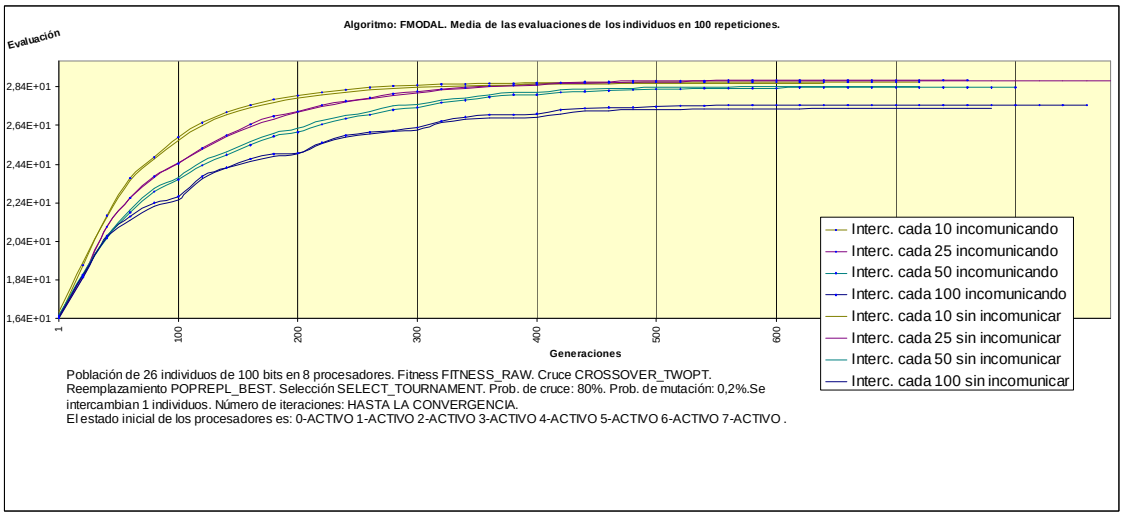
de 0,5 % a 1% de 0% a 0,5%

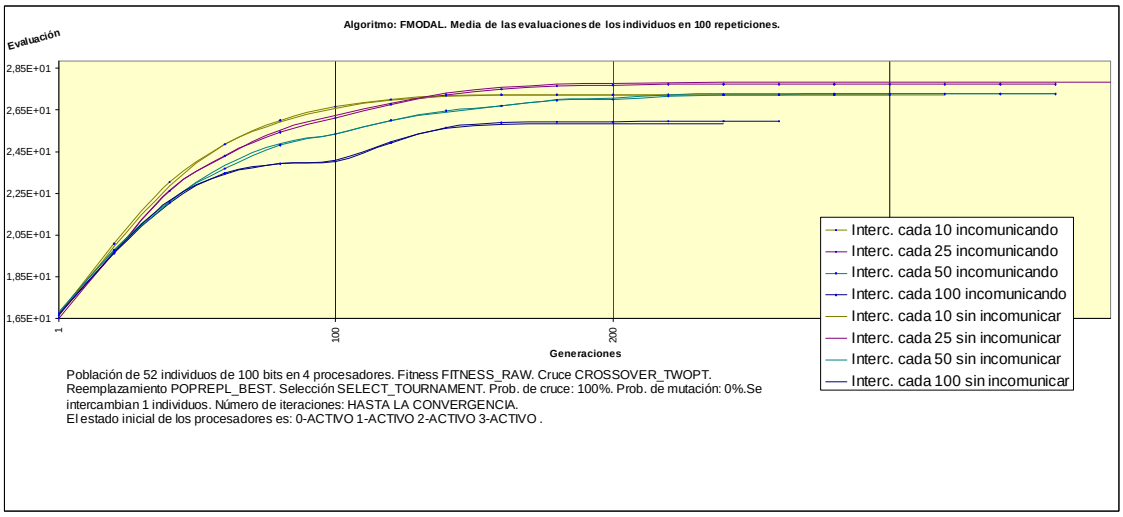


FMODAL. TABLA 2. Resumen de resultados incomunicando cada 100 generaciones

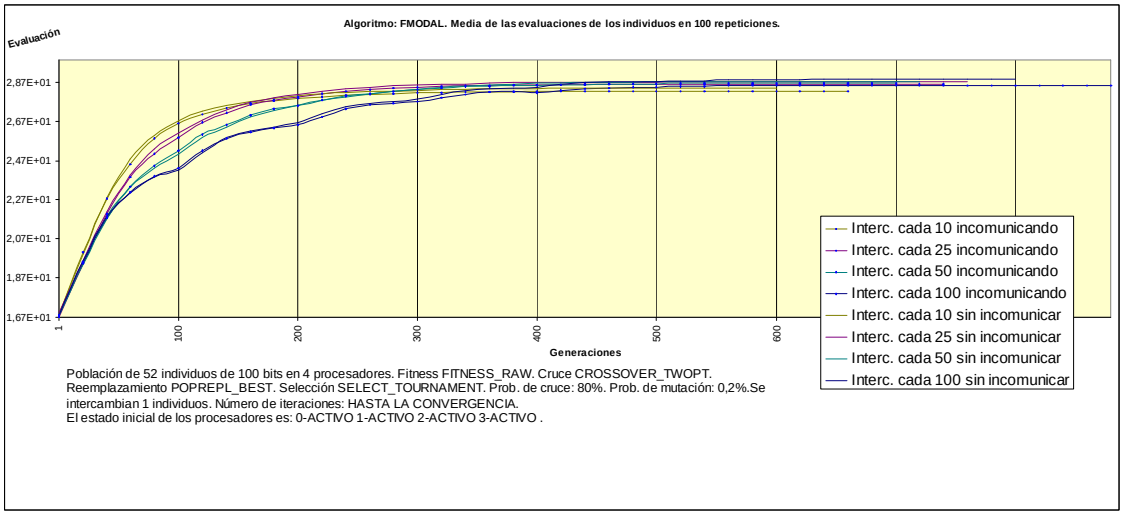
7.5.3. Fallo cada 200 generaciones

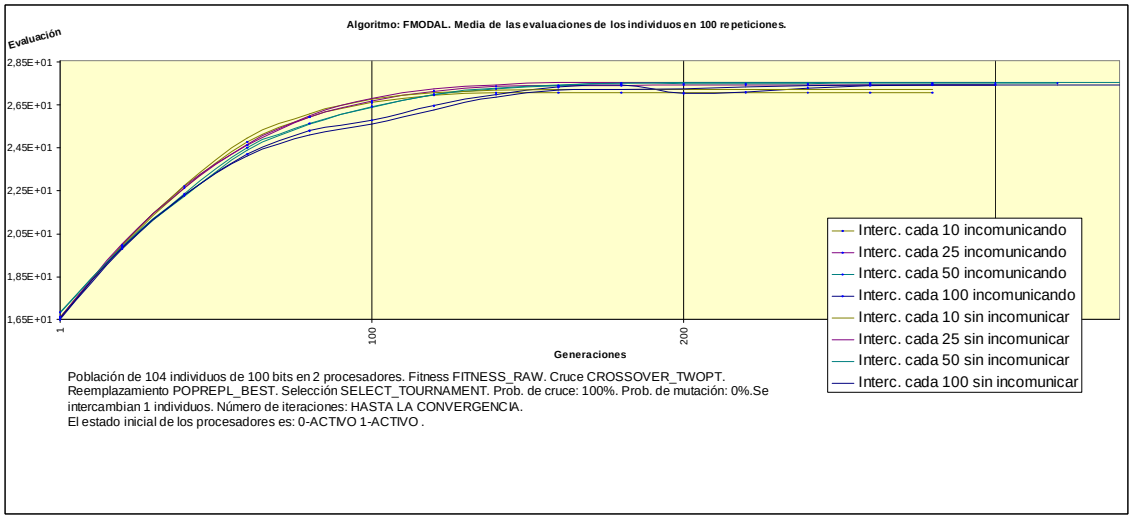


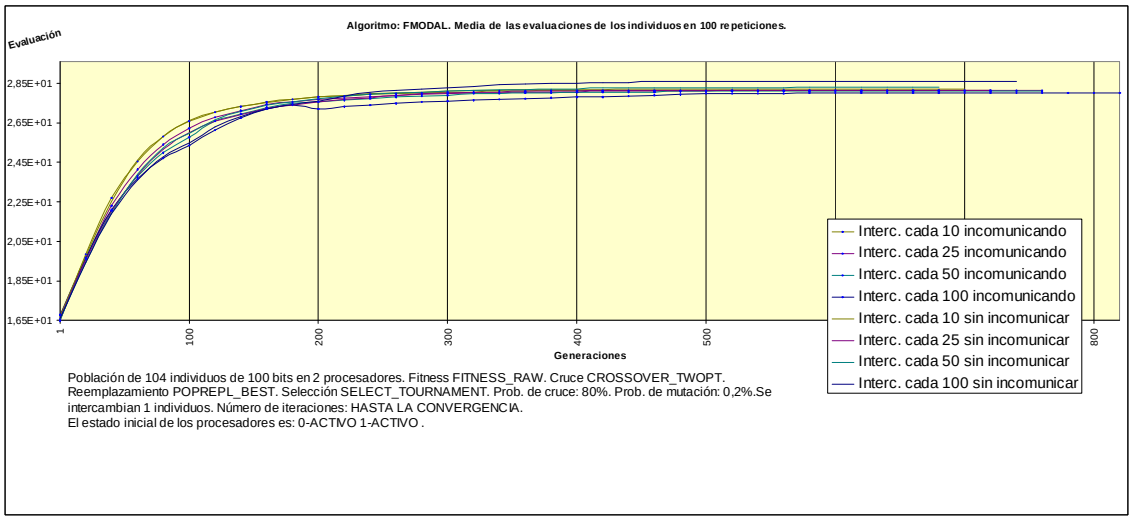




FMODAL. .FIGURA 5 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 200 generaciones falla un procesador



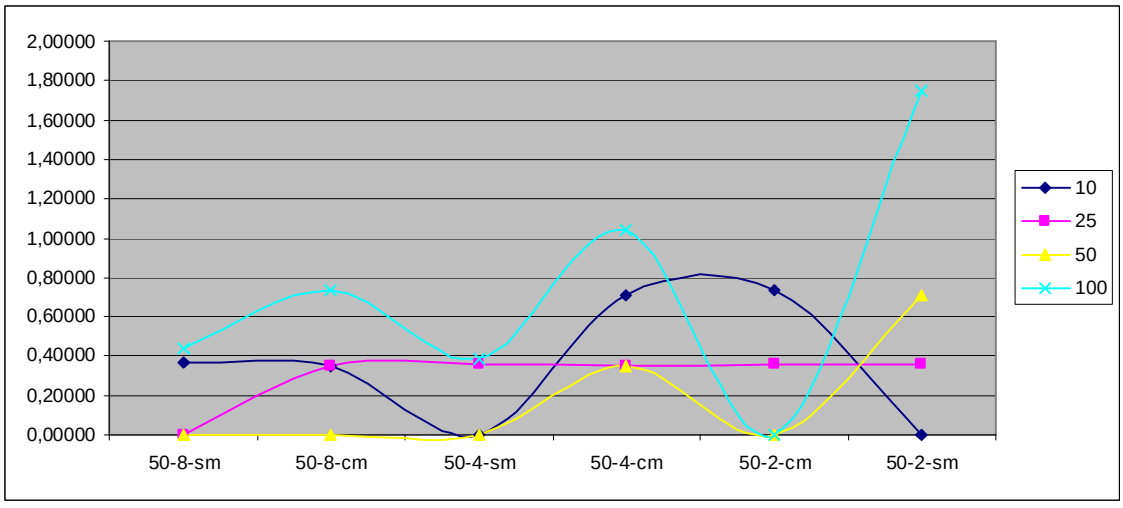




FMODAL. .FIGURA 6 .4 procesadores sin mutación. 2 procesadores sin y con mutación. Cada 200 generaciones falla un procesador

experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
fmodal 200 8 sin mutación	10	si		400			2	25,0	27,6	27,5	0,36364	27,6	27,5
	25	si		400			2	25,0	27,4	27,4	0,00000	25,7	25,5
	50	no		240			1	12,5	25	25	0,00000	20,7	20,9
	100	no		260			1	12,5	22,7	22,8	0,43860	18	18
fmodal 200 8 con mutación	10	si	200	700	1	12,5	3	37,5	28,7	28,6	0,34965	28,6	28,6
	25	si	200	760	1	12,5	3	37,5	28,8	28,7	0,34843	28,5	28,3
	50	si	200	800	1	12,5	4	50,0	28,4	28,4	0,00000	27,2	26,8
	100	si	300	860	1	12,5	4	50,0	27,5	27,3	0,73260	25	24,5
fmodal 200 4 sin mutación	10	si		300			1	25,0	27,3	27,3	0,00000	27,3	27,3
	25	si		360			1	25,0	27,8	27,9	0,35842	27,8	27,9
	50	si		360			1	25,0	27,3	27,3	0,00000	25,9	25,9
	100	no		260			1	25,0	26	25,9	0,38610	23,1	23,1
fmodal 200 4 con mutación	10	si	200	660	1	25,0	2	50,0	28,2	28,4	0,70423	28,2	28,4
	25	si	200	700	1	25,0	2	50,0	28,6	28,7	0,34843	28,5	28,5
	50	si	200	700	1	25,0	2	50,0	28,6	28,7	0,34843	28,6	28,5
	100	si	200	880	1	25,0	2	50,0	28,5	28,8	1,04167	28,3	27,9
fmodal 200 2 sin mutación	10	si		280			1	50,0	27,1	27,3	0,73260	27,1	27,3
	25	si		300			1	50,0	27,5	27,6	0,36232	27,5	27,6
	50	si		300			1	50,0	27,6	27,6	0,00000	27,5	27,6
	100	si		360			1	50,0	27,5	27,5	0,00000	26,9	27
fmodal 200 2 con mutación	10	si		700			1	50,0	28,2	28,2	0,00000	28,2	28,2
	25	si		760			1	50,0	28,1	28,2	0,35461	28,1	28
	50	si		760			1	50,0	28,1	28,3	0,70671	28,1	28,1
	100	si	200	820	1	50,0	1	50,0	28,1	28,6	1,74825	27,8	28,4

de 0,5 % a 1% de 0% a 0,5%



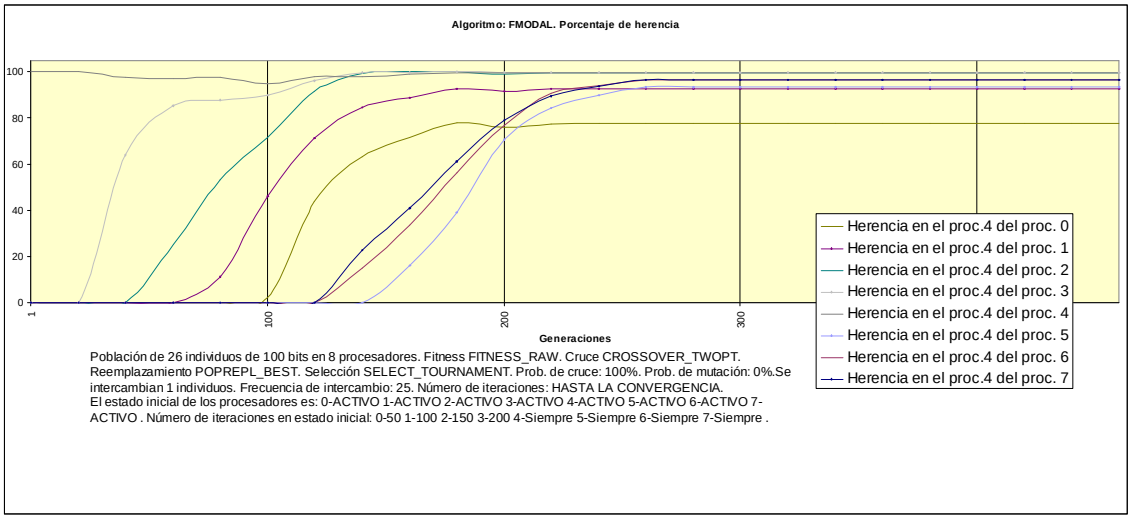
FMODAL. TABLA 3. Resumen de resultados incomunicando cada 200 generaciones

De este experimento no sacamos nuevas conclusiones, sino que podemos volver a aplicar las mismas de los experimentos anteriores, Scwhefel y Rastrigin.

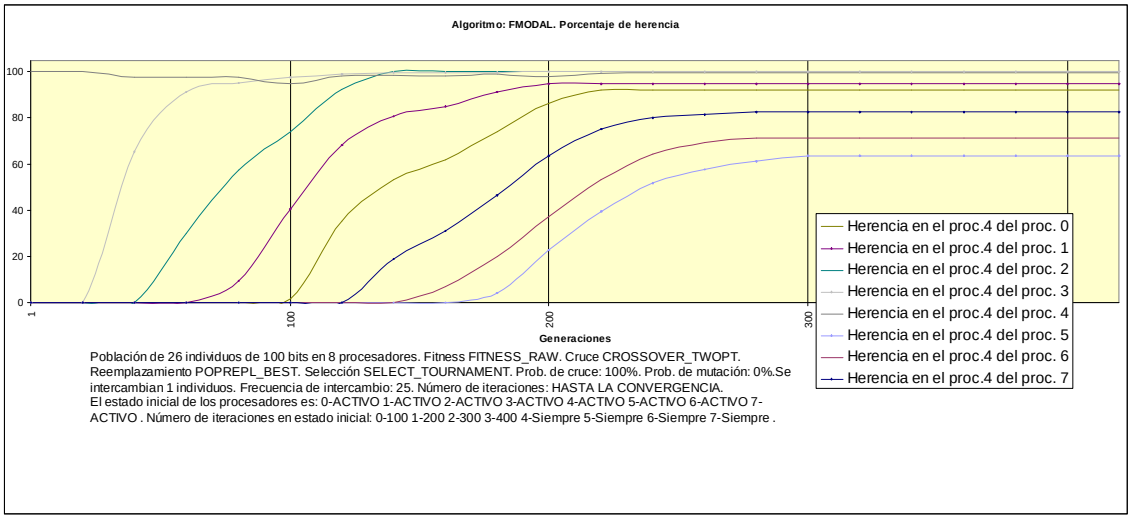
La mejor configuración para esta prueba la conseguimos cuando espaciamos la muerte de los procesadores cada 200 generaciones. Con este periodo conseguimos que en el experimento con 8 procesadores se tolere hasta la muerte del 25% la población (2 procesadores) para las frecuencias de intercambio más altas y de únicamente un 12,5% de la población para los periodos más bajos (50 y 100). En los experimentos con 4 y 2 procesadores sólo se podría permitir la pérdida de un procesador si queremos conseguir soluciones aceptables.

Debemos hacer notar, como ya hemos venido comentando, que si utilizamos la mutación podemos conseguir la tolerancia de más caídas de procesadores. Se permite la muerte de hasta 4 procesadores en los experimentos con 8 procesadores, llegando en la configuración de 4 procesadores a tolerar la muerte de hasta 2 de ellos.

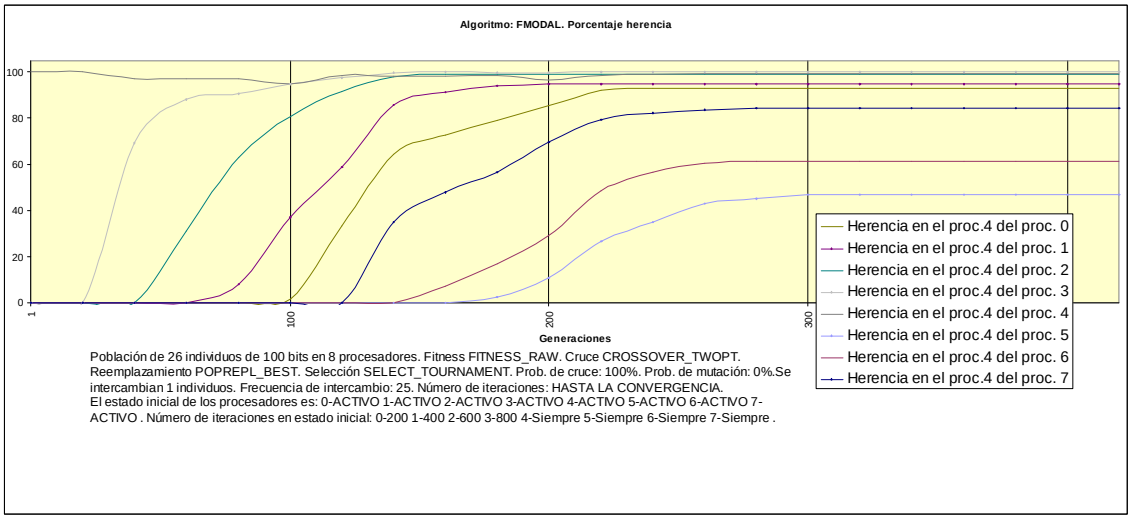
Se puede observar la misma evolución en el intercambio de la herencia. A continuación mostramos las gráficas de herencia de un procesador en el experimento con 8 procesadores en los tres experimentos: fallo cada 50, 100 y 200 generaciones respectivamente.



Fallos cada 50 generaciones



Fallos cada 100 generaciones



Fallos cada 200 generaciones

7.6. Función FTrap:

A continuación detallaremos un pequeño índice en el que se comentaran las pruebas realizadas para este experimento:

- **Fallo cada 50 generaciones:** hemos provocado el fallo de un procesador cada 50 generaciones. Más propiamente dicho, incomunicación ya que ese procesador sigue ejecutándose aisladamente y seguimos recopilando datos de su ejecución.

  Se ha repetido el experimento para 8 procesadores, 4 procesadores y 2 procesadores con y sin mutación.

  - *Con mutación:* Probabilidad de cruce 0,8. Probabilidad de mutación *0,002.*

  - *Sin mutación:* Probabilidad de cruce 1. Probabilidad de mutación 0.

  Para todos los casos se han considerado los periodos de intercambio cada 10, 25, 50 y 100 generaciones.

  Así mismo, todas estas pruebas se han repetido con las mismas condiciones sin forzar ningún fallo en los procesadores.

  Se puede observar los resultados en forma de gráfica en las FIGURA 1 y 2, así como un resumen de los resultados en la TABLA1.

- **Fallo cada 100 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones, pero esta vez se han provocado los fallos cada 100 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 3 y 4, así como un resumen de los resultados en la TABLA 2.

- **Fallo cada 200 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones pero esta vez se han provocado los fallos cada 200 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 5 y 6, así como un resumen de los resultados en la TABLA 3.

Para una mejor comprensión de las tablas procederemos a continuación a la explicación de cada una de las columnas:

- **Experimento:** resumen del experimento al que se refieren los resultados.

- **Intercambio:** periodo de intercambio utilizado en ese experimento.

- **Herencia de todos:** en esta columna se observará si todos los procesadores que quedan "vivos" al final de la ejecución han conseguido recibir herencia de todos los procesadores con los que se inicio la ejecución.

- **Separan en generación:** en esta columna se muestra a partir de qué generación las líneas de la gráfica correspondientes al experimento con fallos y aquéllas correspondientes al experimento sin fallos comienzan a separarse indicando tener algún problema por la "muerte" de algún procesador.

- **Generacion:** número de generaciones que ha ejecutado el experimento.

- **Proc Caidos:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos.

- **%Pob Muerta:** mostramos el porcentaje de población muerte en el momento en el que se empiezan a separar las líneas.

- **Proc Caidos Total:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos al finalizar la ejecución.

- **%Pob Muerta Total:** mostramos el porcentaje de población muerta en el momento en el que se empiezan a separar la líneas al finalizar la ejecución

- **Max Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

- **Max Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

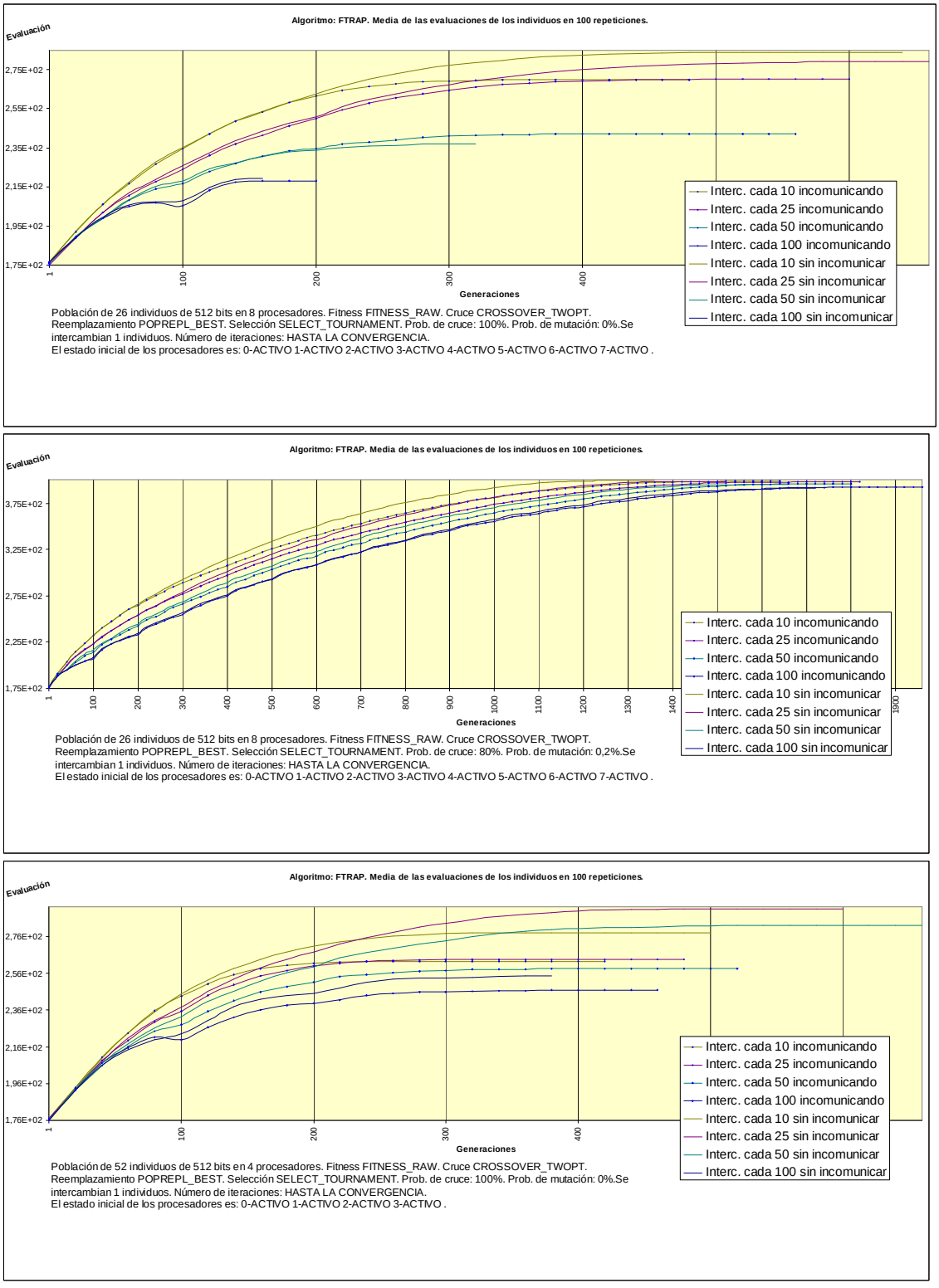
- **Tolerancia (% error):** diferencia entre el valor alcanzado con fallos en los procesadores y la ejecución sin fallos. Expresada en porcentaje.

- **Min Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

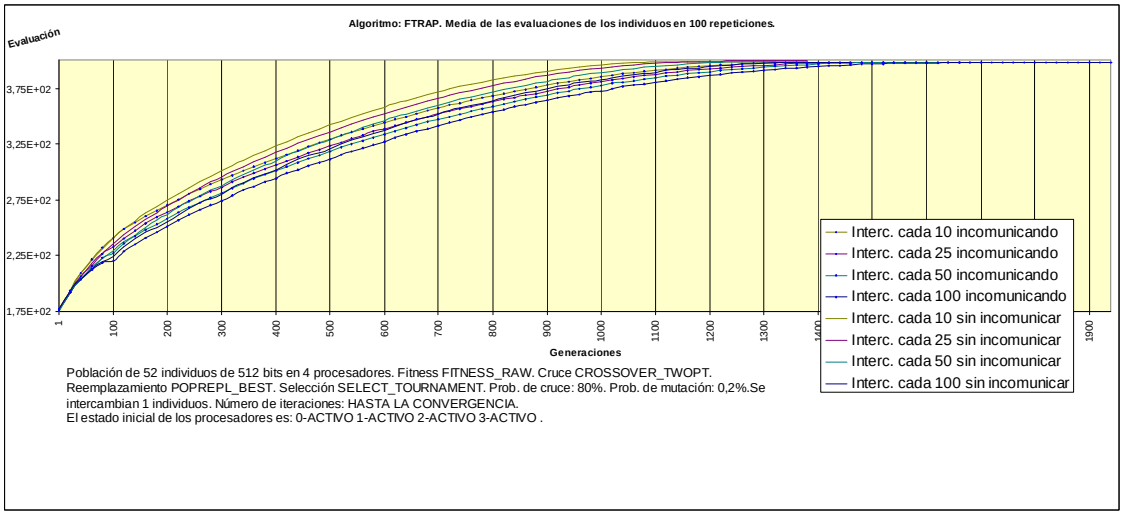
- **Min Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

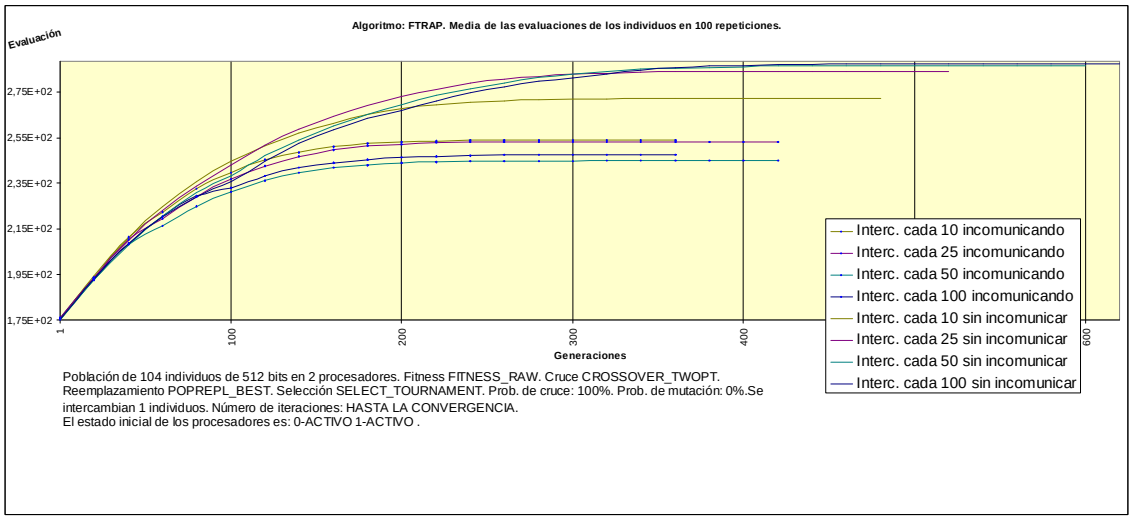
A continuación de cada tabla se muestra una gráfica en la que se indica la evolución del porcentaje de tolerancia según el periodo de intercambio y si el experimento fue realizado con mutación (cm) o sin mutación (sm).

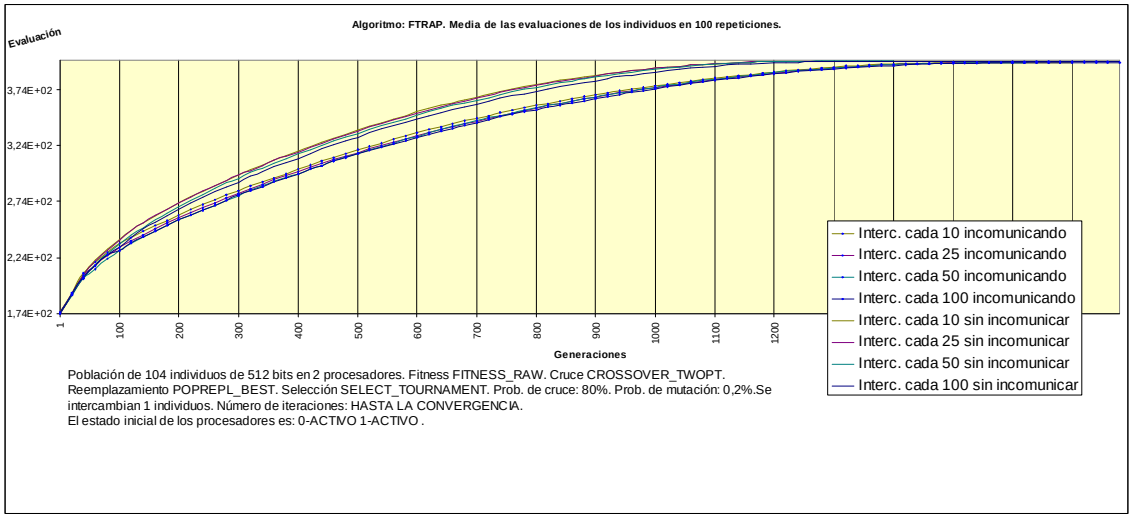
7.6.1. Fallo cada 50 generaciones



FTRAP. .FIGURA 1 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 50 generaciones falla un procesador



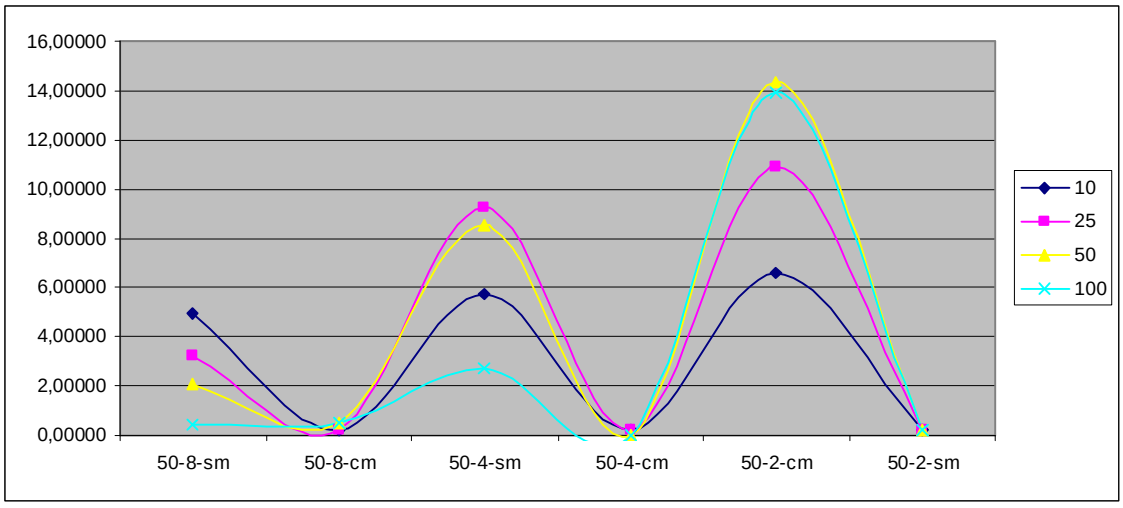




FTRAP. FIGURA 2 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 50 generaciones falla un procesador

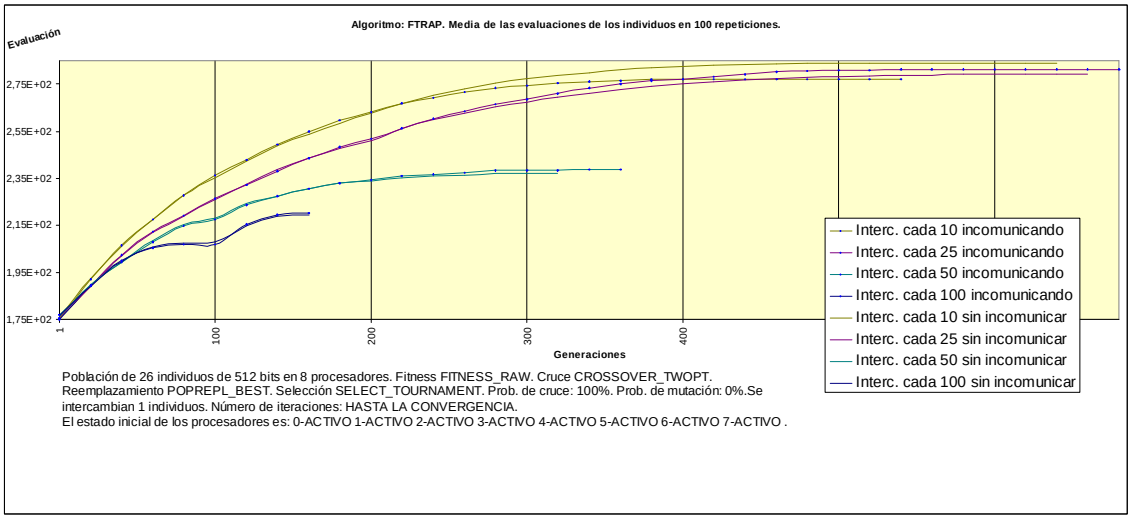
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
FTrap 50 8 sin mutación	10	si	180	480	3	37,5	4	50,0	270	284	4,92958	270	284
	25	si	100	600	2	25,0	4	50,0	270	279	3,22581	268	267
	50	no	80	560	1	12,5	4	50,0	242	237	2,10970	223	211
	100	no	80	200	1	12,5	4	50,0	218	219	0,45662	193	191
FTrap 50 8 con mutación	10	si	200	1640	4	50,0	4	50,0	399	400	0,25000	398	398
	25	si	260	1820	4	50,0	4	50,0	398	399	0,25063	396	395
	50	no	140	1800	2	25,0	4	50,0	397	395	0,50633	393	389
	100	no		1960			4	50,0	393	391	0,51151	388	384
FTrap 50 4 sin mutación	10	si	100	420	2	50,0	2	50,0	262	278	5,75540	262	278
	25	si	80	480	1	25,0	2	50,0	264	291	9,27835	264	290
	50	no	60	520	1	25,0	2	50,0	258	282	8,51064	258	271
	100	no	50	460	1	25,0	2	50,0	247	254	2,75591	240	229
FTrap 50 4 con mutación	10	si	160	1600	2	50,0	2	50,0	399	400	0,25000	398	398
	25	si	80	1660	1	25,0	2	50,0	399	400	0,25000	397	397
	50	no	80	1700	1	25,0	2	50,0	399	399	0,00000	397	396
	100	no	80	1940	1	25,0	2	50,0	399	399	0,00000	396	395
FTrap 50 2 sin mutación	10	si	50	360	1	50,0	1	50,0	254	272	6,61765	254	272
	25	si	50	420	1	50,0	1	50,0	253	284	10,91549	253	284
	50	no	50	420	1	50,0	1	50,0	245	286	14,33566	245	286
	100	no	80	360	1	50,0	1	50,0	247	287	13,93728	247	284
FTrap 50 2 con mutación	10	si	50	1700	1	50,0	1	50,0	399	400	0,25000	398	398
	25	si	50	1760	1	50,0	1	50,0	399	400	0,25000	397	397
	50	no	50	1780	1	50,0	1	50,0	399	400	0,25000	397	397
	100	no	50	1780	1	50,0	1	50,0	399	400	0,25000	397	397

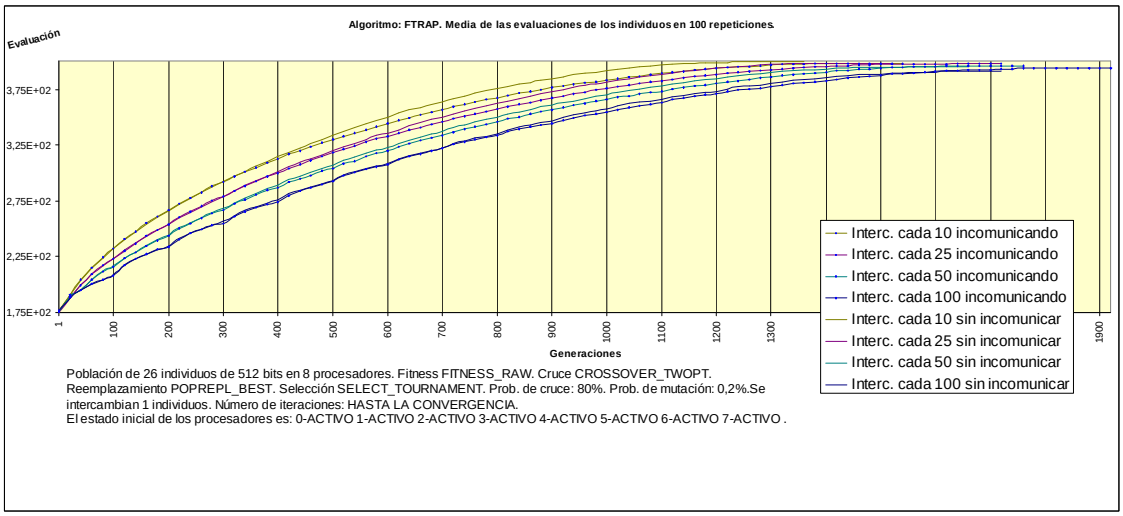
de 0,5 % a 1% de 0% a 0,5%

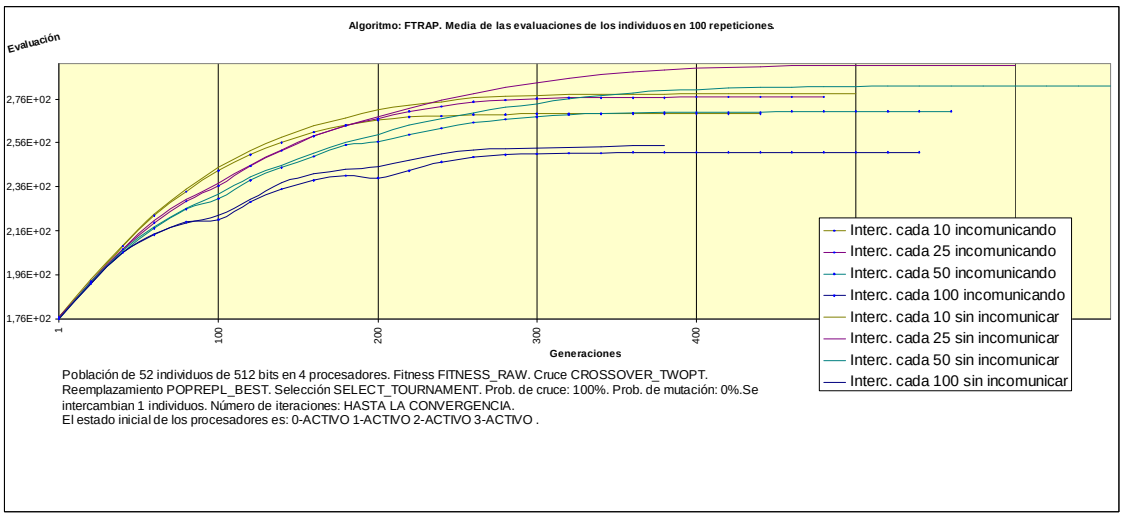


FTRAP. TABLA 1. Resumen de resultados incomunicando cada 50 generaciones

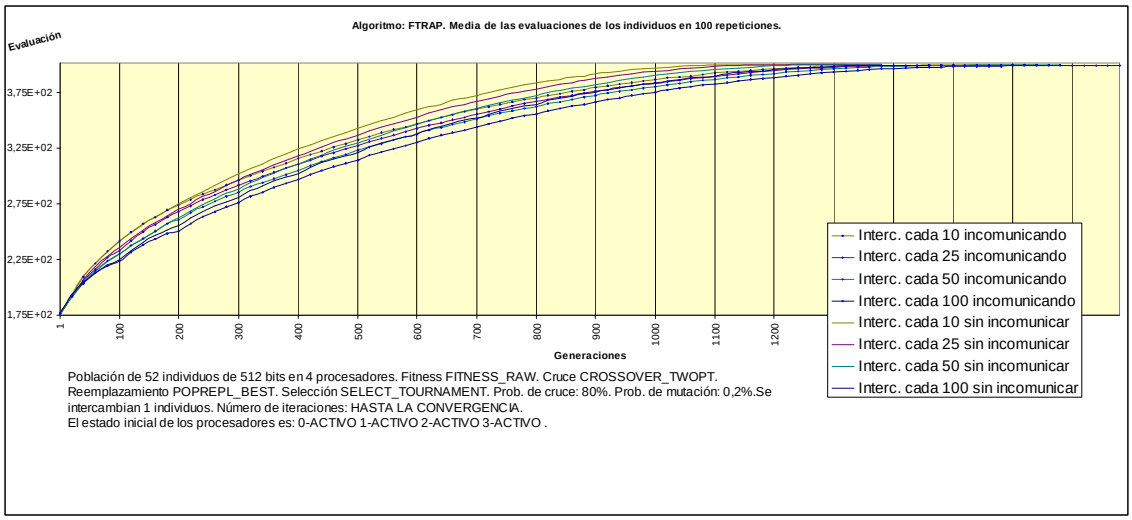
7.6.2. Fallo cada 100 generaciones

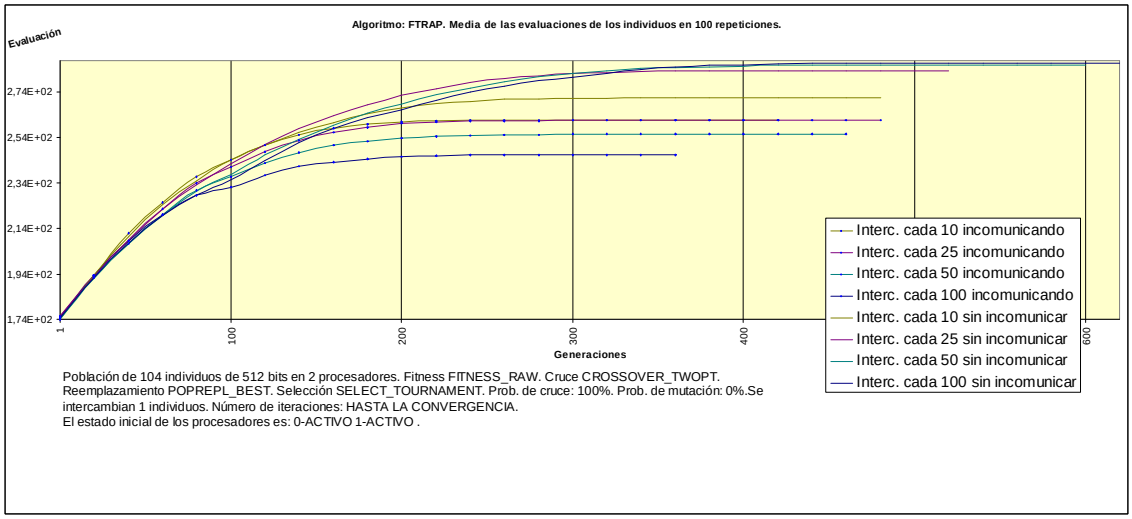


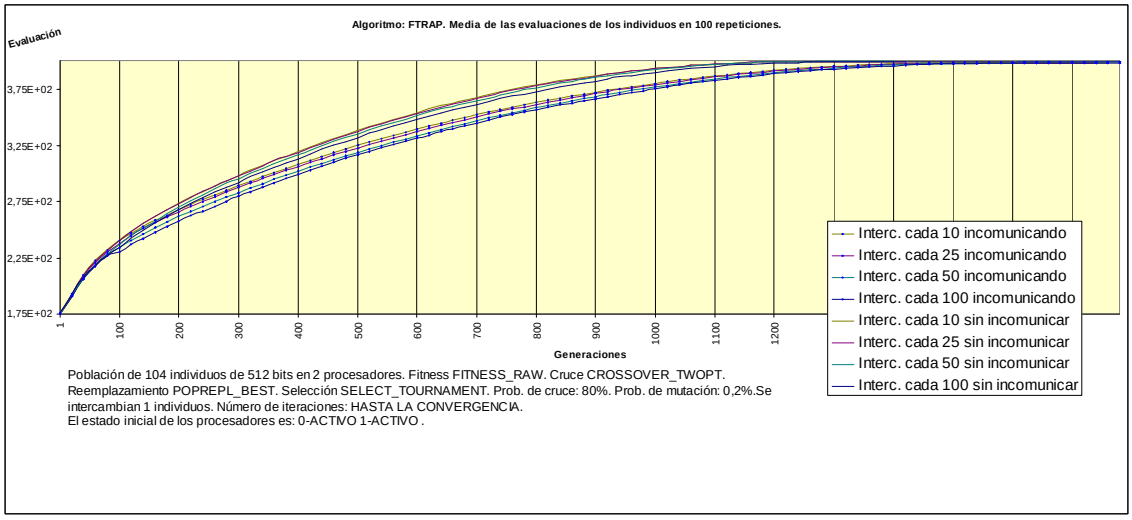




FTRAP. .FIGURA 3 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 100 generaciones falla un procesador



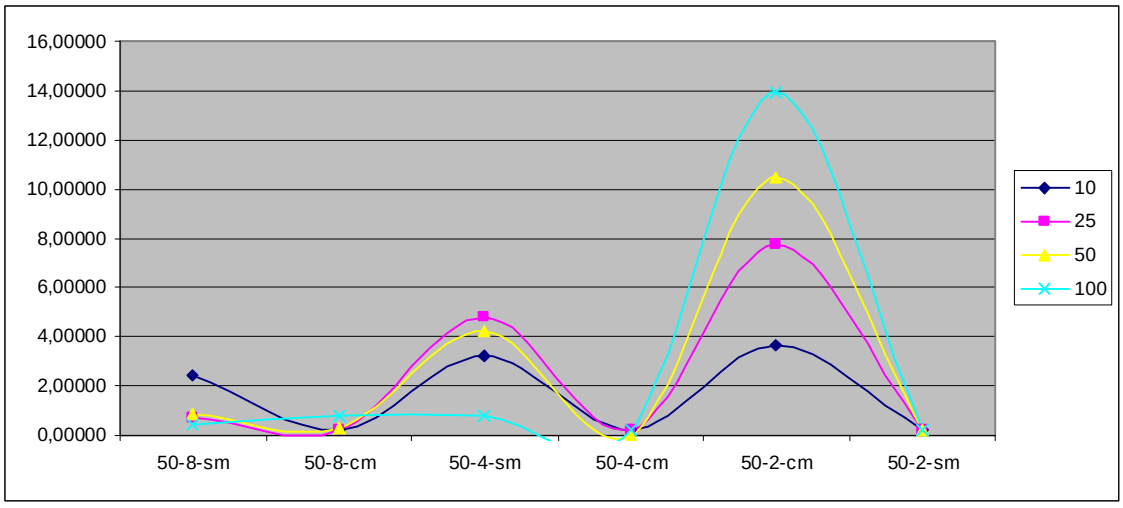




FTRAP. FIGURA 4 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 100 generaciones falla un procesador

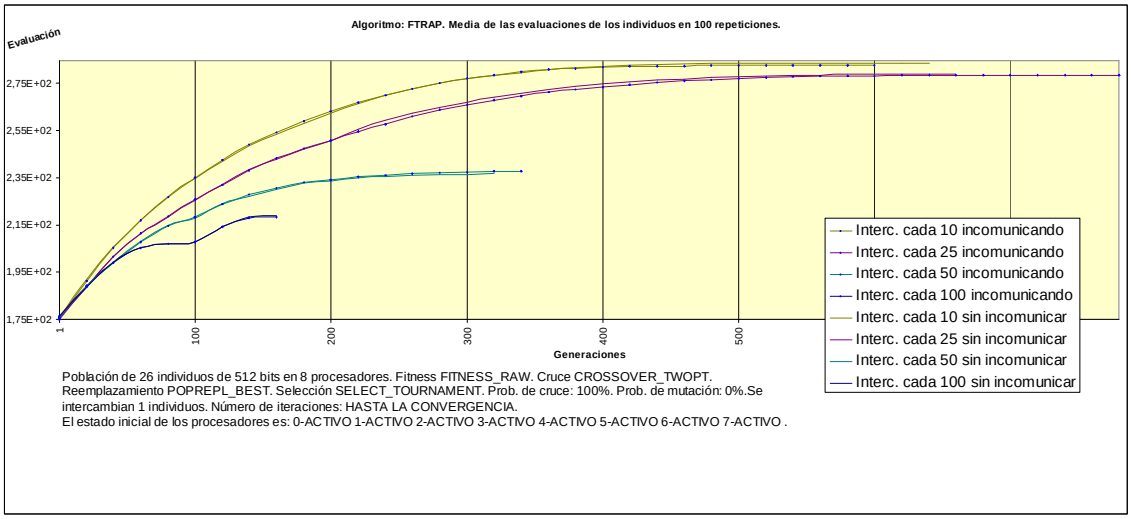
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
FTrap 100 8 sin mutación	10	si	220	540	2	25,0	4	50,0	277	284	2,46479	277	284
	25	si	180	680	1	12,5	4	50,0	281	279	0,71685	278	267
	50	si	220	360	2	25,0	3	37,5	239	237	0,84388	215	211
	100	no		160			1	12,5	220	219	0,45662	191	191
FTrap 100 8 con mutación	10	si	400	1540	4	50,0	4	50,0	399	400	0,25000	398	398
	25	si	400	1720	4	50,0	4	50,0	398	399	0,25063	396	395
	50	si	380	1760	3	37,5	4	50,0	396	395	0,25316	393	389
	100	no	800	1920	4	50,0	4	50,0	394	391	0,76726	390	384
FTrap 100 4 sin mutación	10	si	100	440	1	25,0	2	50,0	269	278	3,23741	269	278
	25	si	180	480	1	25,0	2	50,0	277	291	4,81100	277	290
	50	si	100	560	1	25,0	2	50,0	270	282	4,25532	270	271
	100	no	80	540	0	0,0	2	50,0	252	254	0,78740	240	229
FTrap 100 4 con mutación	10	si	180	1640	1	25,0	2	50,0	399	400	0,25000	398	398
	25	si	180	1580	1	25,0	2	50,0	399	400	0,25000	397	397
	50	si	180	1680	1	25,0	2	50,0	399	399	0,00000	397	396
	100	no	180	1780	1	25,0	2	50,0	398	399	0,25063	396	395
FTrap 100 2 sin mutación	10	si	140	420	1	50,0	1	50,0	262	272	3,67647	262	272
	25	si	100	480	1	50,0	1	50,0	262	284	7,74648	262	284
	50	si	100	460	1	50,0	1	50,0	256	286	10,48951	256	286
	100	no	100	360	1	50,0	1	50,0	247	287	13,93728	247	284
FTrap 100 2con mutación	10	si	100	1760	1	50,0	1	50,0	399	400	0,25000	398	398
	25	si	100	1740	1	50,0	1	50,0	399	400	0,25000	397	397
	50	si	100	1780	1	50,0	1	50,0	399	400	0,25000	397	397
	100	no	100	1780	1	50,0	1	50,0	399	400	0,25000	397	397

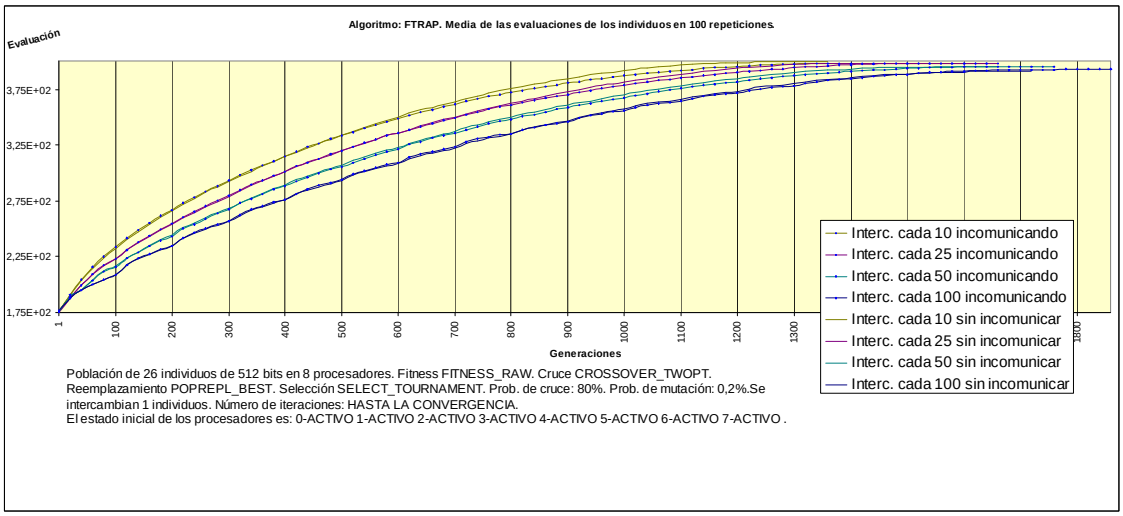
de 0,5 % a 1% de 0% a 0,5%

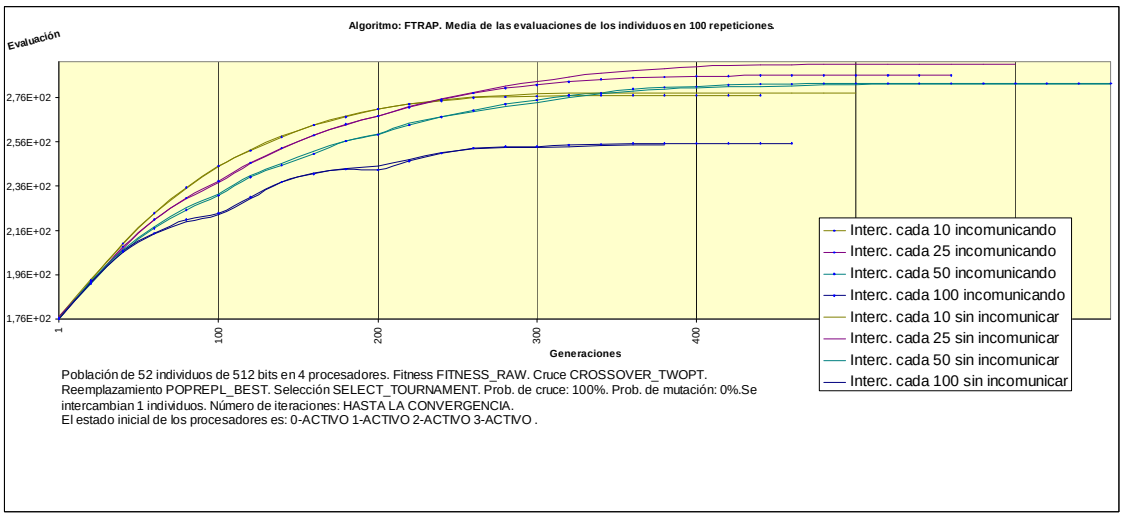


FTRAP. TABLA 2. Resumen de resultados incomunicando cada 100 generaciones

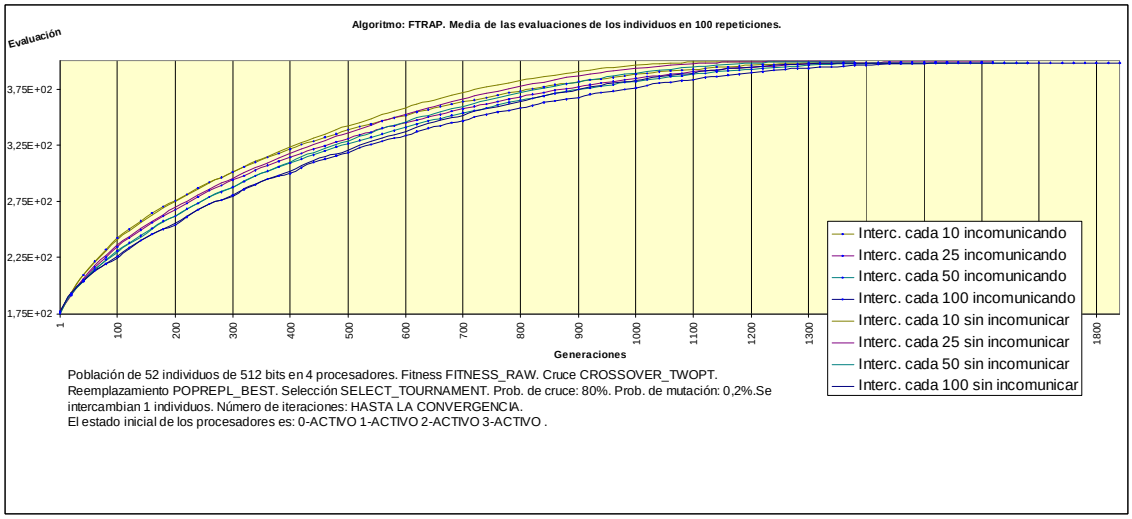
7.6.3. Fallo cada 200 generaciones



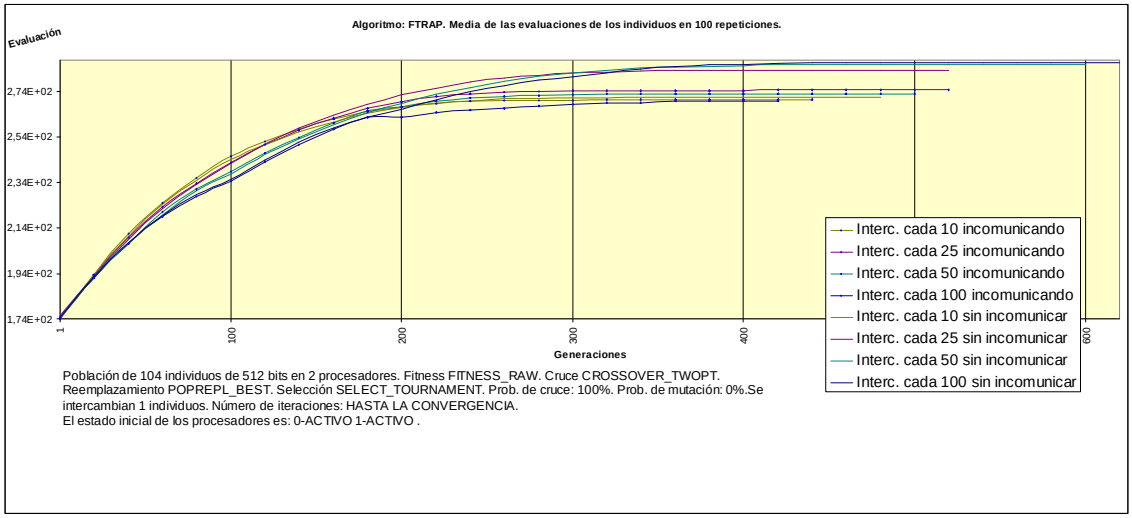




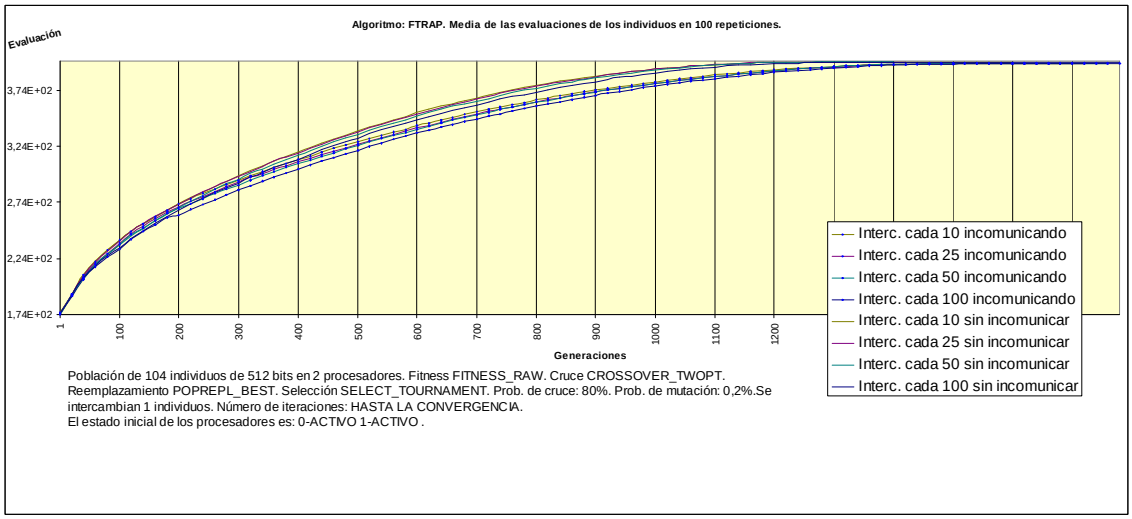
FTRAP. .FIGURA 5 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 200 generaciones falla un procesador



Población de 52 individuos de 512 bits en 4 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 80%. Prob. de mutación: 0,2%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO .



Población de 104 individuos de 512 bits en 2 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 100%. Prob. de mutación: 0%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO .

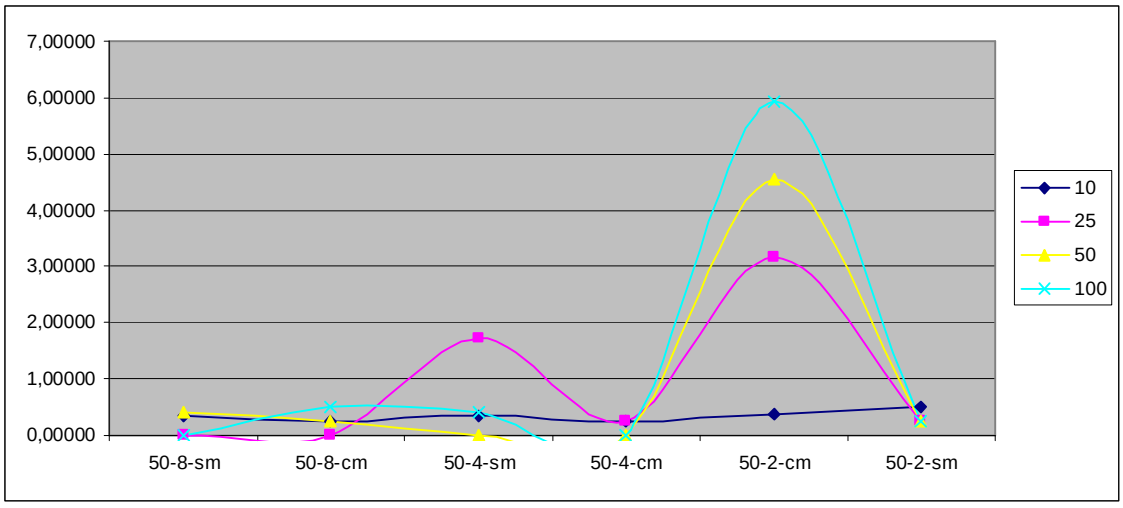


Población de 104 individuos de 512 bits en 2 procesadores. Fitness FITNESS_RAW. Cruce CROSSOVER_TWOPT. Reemplazamiento POPREPL_BEST. Selección SELECT_TOURNAMENT. Prob. de cruce: 80%. Prob. de mutación: 0,2%.Se intercambian 1 individuos. Número de iteraciones: HASTA LA CONVERGENCIA.
El estado inicial de los procesadores es: 0-ACTIVO 1-ACTIVO .

FTRAP. .FIGURA 6 .4 procesadores sin mutación. 2 procesadores sin y con mutación. Cada 200 generaciones falla un procesador

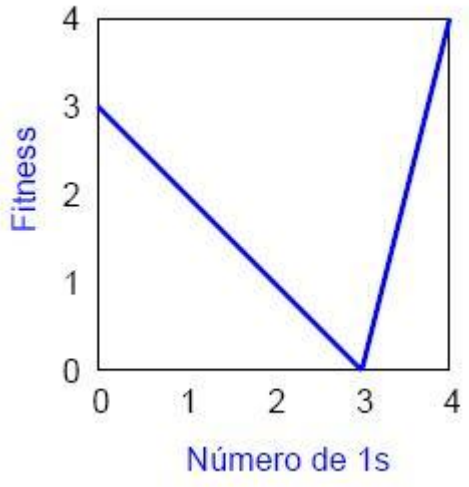
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
FTrap 200 8 sin mutación	10	si		600			3	37,5	283	284	0,35211	283	284
	25	si		780			3	37,5	279	279	0,00000	269	267
	50	no		340			1	12,5	238	237	0,42194	212	211
	100	no		160			0	0,0	219	219	0,00000	191	191
FTrap 200 8 con mutación	10	si	600	1640	3	37,5	4	50,0	399	400	0,25000	398	398
	25	si	800	1660	4	50,0	4	50,0	399	399	0,00000	397	395
	50	si	800	1760	4	50,0	4	50,0	396	395	0,25316	392	389
	100	si		1860			4	50,0	393	391	0,51151	388	384
FTrap 200 4 sin mutación	10	si	280	440	1	25,0	2	50,0	277	278	0,35971	277	278
	25	si	280	560	1	25,0	2	50,0	286	291	1,71821	286	290
	50	si		660			2	50,0	282	282	0,00000	279	271
	100	si		460			2	50,0	255	254	0,39370	233	229
FTrap 200 4 con mutación	10	si	400	1540	2	50,0	2	50,0	399	400	0,25000	398	398
	25	si	280	1620	1	25,0	2	50,0	399	400	0,25000	397	397
	50	si	440	1680	2	50,0	2	50,0	399	399	0,00000	397	396
	100	si	460	1840	2	50,0	2	50,0	399	399	0,00000	396	395
FTrap 200 2 sin mutación	10	si		440			1	50,0	271	272	0,36765	271	272
	25	si	200	520	1	50,0	1	50,0	275	284	3,16901	275	284
	50	si	200	500	1	50,0	1	50,0	273	286	4,54545	273	286
	100	si	200	420	1	50,0	1	50,0	270	287	5,92334	268	284
FTrap 200 2 con mutación	10	si	200	1720	1	50,0	1	50,0	398	400	0,50000	397	398
	25	si	200	1720	1	50,0	1	50,0	399	400	0,25000	397	397
	50	si	200	1760	1	50,0	1	50,0	399	400	0,25000	397	397
	100	si	200	1780	1	50,0	1	50,0	399	400	0,25000	397	397

de 0,5 % a 1% de 0% a 0,5%



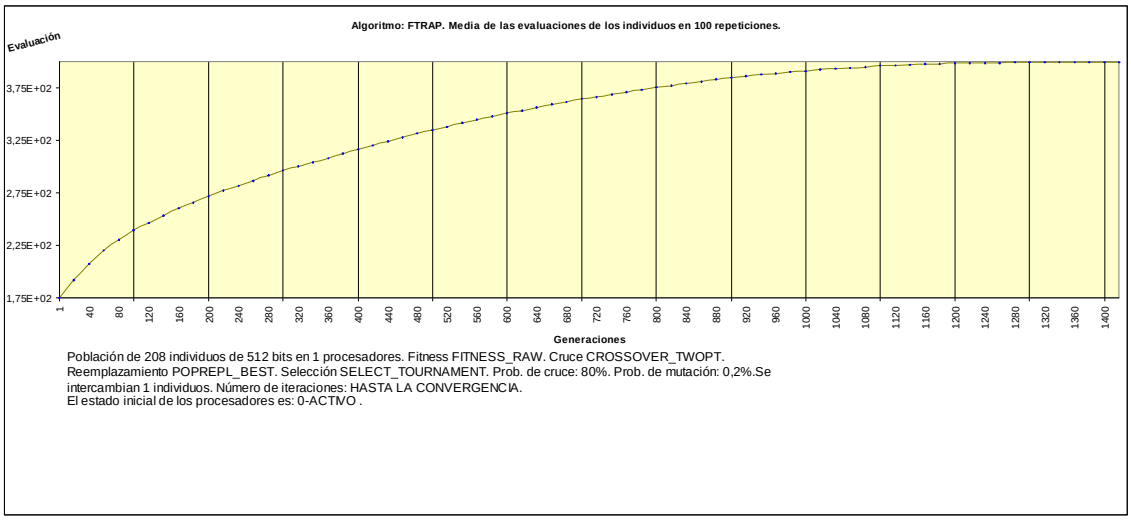
FTRAP. TABLA 3. Resumen de resultados incomunicando cada 200 generaciones

El caso de la función FTrap es diferente a todos los estudiados hasta ahora. Este tipo de problema es de los considerados como defectivos o funcionen trampa.



Hemos fijado el máximo de la función en 512 y el máximo local en 400 y en ningún experimento hemos conseguido obtener el máximo (ver tablas), siempre nos hemos quedado atrapados en el máximo local. Por lo tanto, necesitamos otra forma de acometer la resolución de este problema[1].

Podemos observar que con mutación es cuando conseguimos mejores resultados y tolerancia a fallos, teniendo en cuenta la diferencia con el experimento sin fallos en los procesadores. Sin embargo, en ninguna prueba conseguimos el valor deseado. Lo que sí podemos apreciar es que con un sólo procesador y con mutación conseguimos llegar al máximo local.



[1] J. I. Hidalgo and F. Fernandez. Balancing the Computation Effort in Genetic Algorithms. Proceedings of 2005 IEEE Congress on Evolutionary Computation. Edinburgh (United Kigdom). ISBN: 0-7803-9363-5

7.7. Función Rosenbrock:

A continuación detallaremos un pequeño índice en el que se comentaran las pruebas realizadas para este experimento:

- **Fallo cada 50 generaciones:** hemos provocado el fallo de un procesador cada 50 generaciones. Más propiamente dicho, incomunicación ya que ese procesador sigue ejecutándose aisladamente y seguimos recopilando datos de su ejecución.

  Se ha repetido el experimento para 8 procesadores, 4 procesadores y 2 procesadores con y sin mutación.

  - *Con mutación:* Probabilidad de cruce 0,8. Probabilidad de mutación *0,002.*

  - *Sin mutación:* Probabilidad de cruce 1. Probabilidad de mutación 0.

  Para todos los casos se han considerado los periodos de intercambio cada 10, 25, 50 y 100 generaciones.

  Así mismo, todas estas pruebas se han repetido con las mismas condiciones sin forzar ningún fallo en los procesadores.

  Se puede observar los resultados en forma de gráfica en las FIGURA 1 y 2, así como un resumen de los resultados en la TABLA1.

- **Fallo cada 100 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones, pero esta vez se han provocado los fallos cada 100 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 3 y 4, así como un resumen de los resultados en la TABLA 2.

- **Fallo cada 200 generaciones:** se ha repetido el mismo tipo de pruebas que en el fallo cada 50 generaciones pero esta vez se han provocado los fallos cada 200 generaciones.

  Se puede observar los resultados en forma de gráfica en las FIGURA 5 y 6, así como un resumen de los resultados en la TABLA 3.

Para una mejor comprensión de las tablas procederemos a continuación a la explicación de cada una de las columnas:

- **Experimento:** resumen del experimento al que se refieren los resultados.

- **Intercambio:** periodo de intercambio utilizado en ese experimento.

- **Herencia de todos:** en esta columna se observará si todos los procesadores que quedan "vivos" al final de la ejecución han conseguido recibir herencia de todos los procesadores con los que se inicio la ejecución.

- **Separan en generación:** en esta columna se muestra a partir de qué generación las líneas de la gráfica correspondientes al experimento con fallos y aquéllas correspondientes al experimento sin fallos comienzan a separarse indicando tener algún problema por la "muerte" de algún procesador.

- **Generacion:** número de generaciones que ha ejecutado el experimento.

- **Proc Caidos:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos.

- **%Pob Muerta:** mostramos el porcentaje de población muerte en el momento en el que se empiezan a separar las líneas.

- **Proc Caidos Total:** número de procesadores caídos en el momento en el que se aprecia una separación en las líneas de las gráficas correspondientes al experimento con fallos y al experimento sin fallos al finalizar la ejecución.

- **%Pob Muerta Total:** mostramos el porcentaje de población muerta en el momento en el que se empiezan a separar la líneas al finalizar la ejecución

- **Max Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

- **Max Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

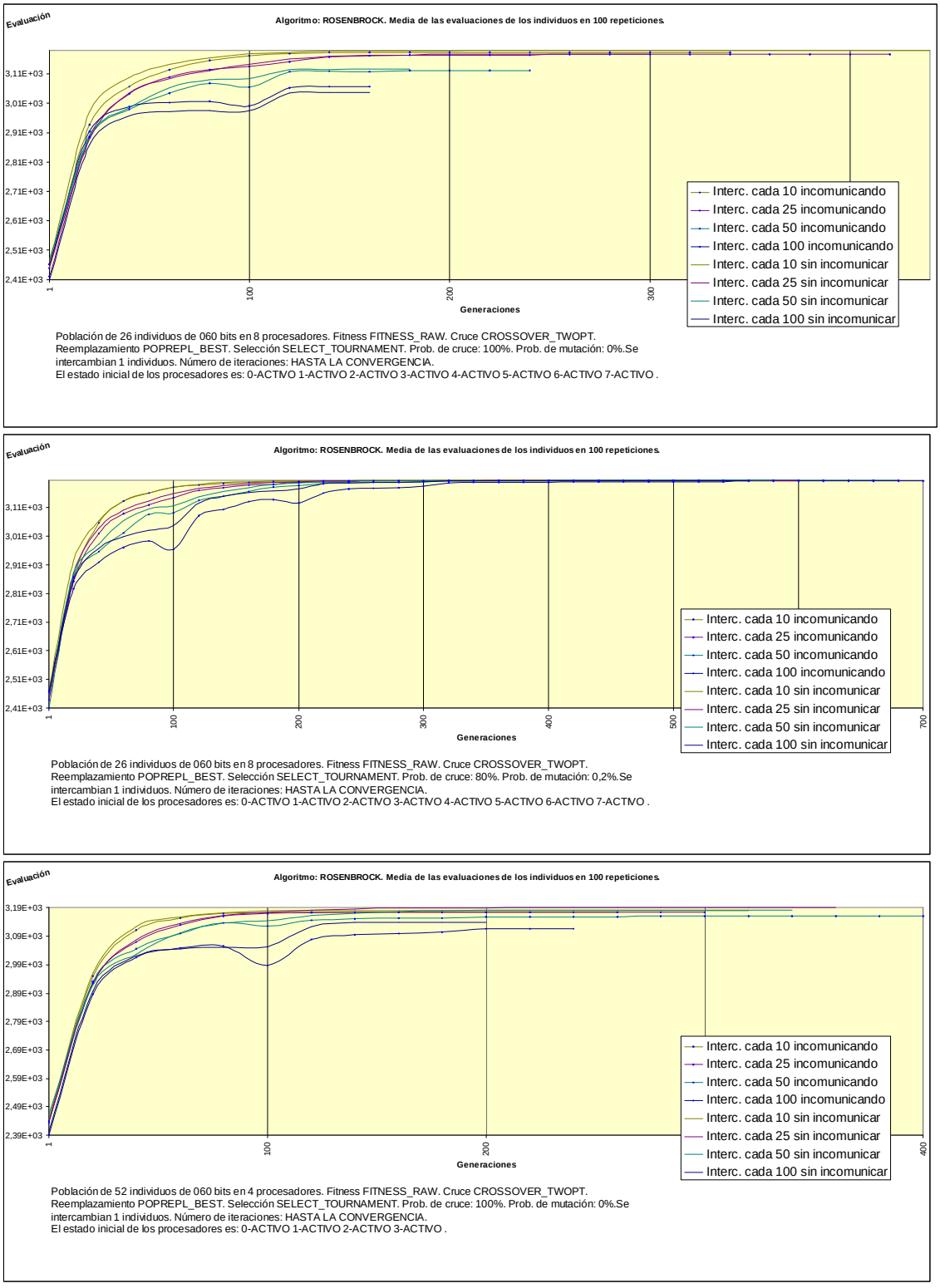
- **Tolerancia (% error):** diferencia entre el valor alcanzado con fallos en los procesadores y la ejecución sin fallos. Expresada en porcentaje.

- **Min Valor Matando:** valor máximo alcanzado en la ejecución en la que provocamos fallos en los procesadores.

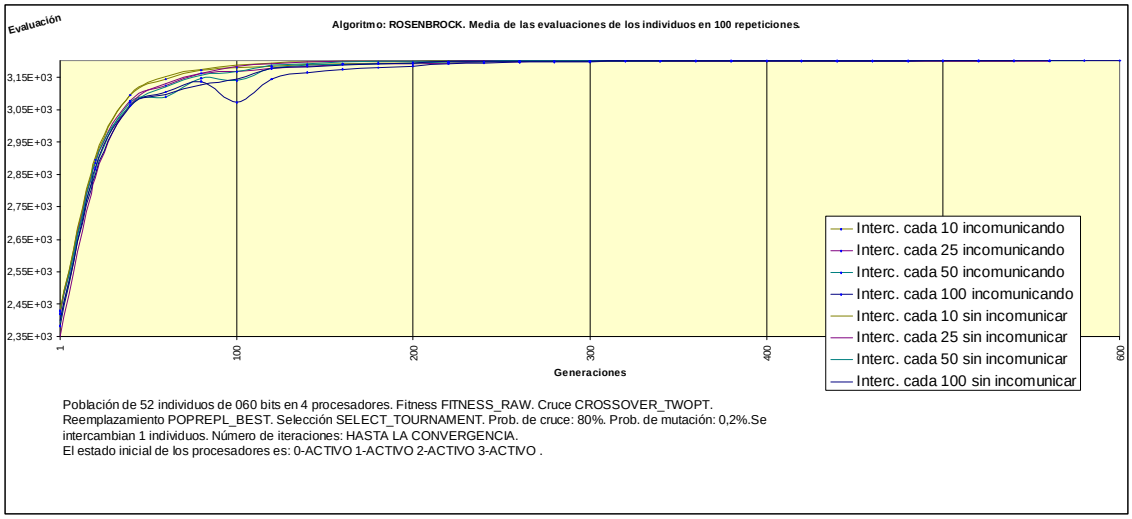
- **Min Valor Sin Matar:** valor máximo alcanzado en la ejecución en la no que provocamos fallos en los procesadores.

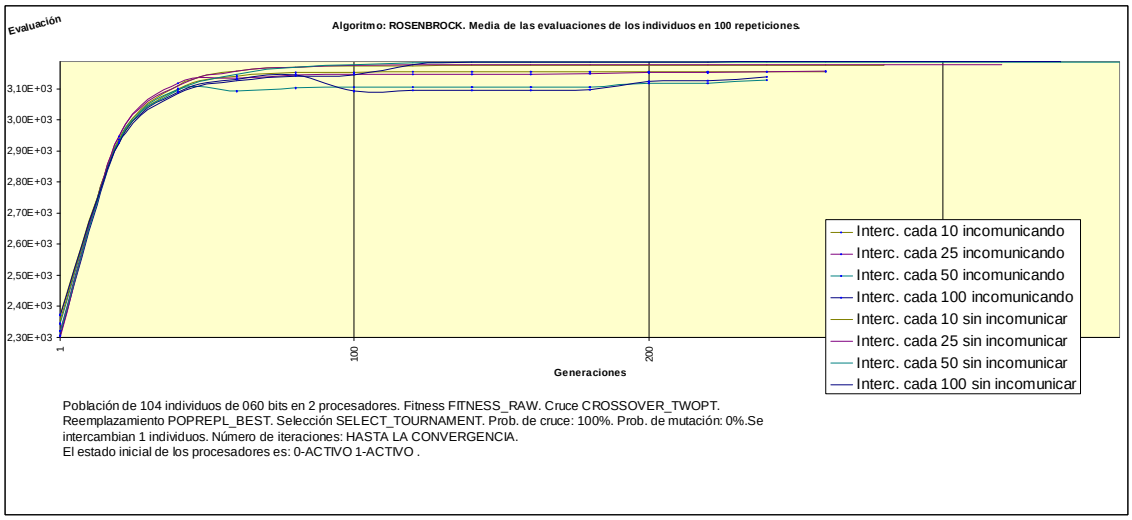
A continuación de cada tabla se muestra una gráfica en la que se indica la evolución del porcentaje de tolerancia según el periodo de intercambio y si el experimento fue realizado con mutación (cm) o sin mutación (sm).

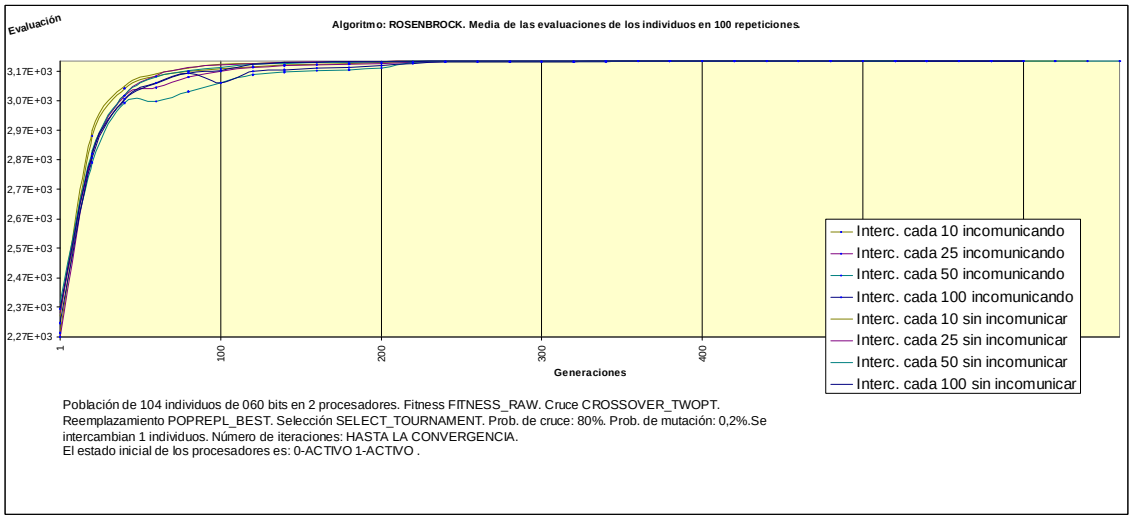
7.7.1. Fallo cada 50 generaciones



ROSENBROCK. .FIGURA 1 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 50 generaciones falla un procesador



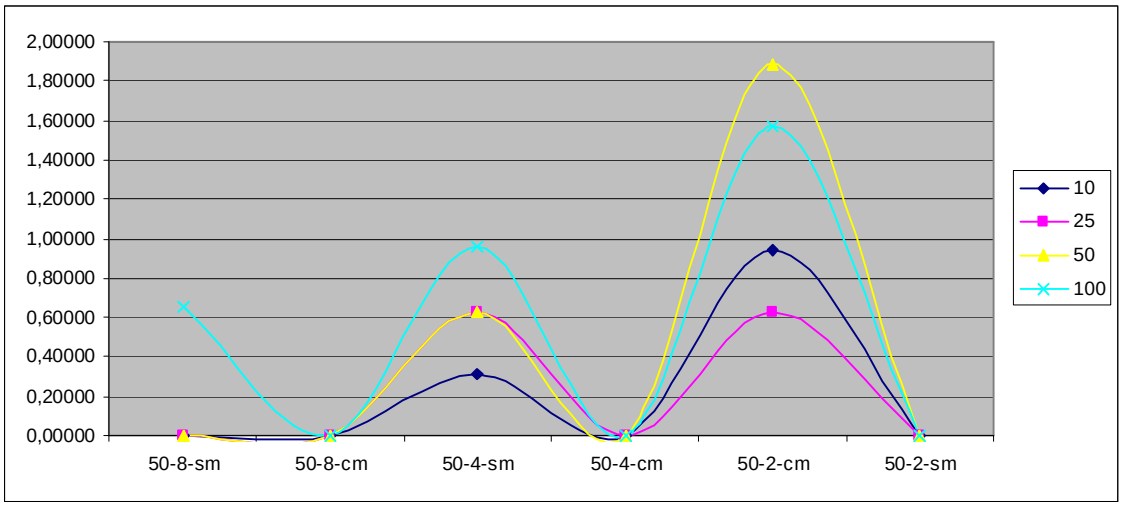




ROSENBROCK. FIGURA 2 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 50 generaciones falla un procesador

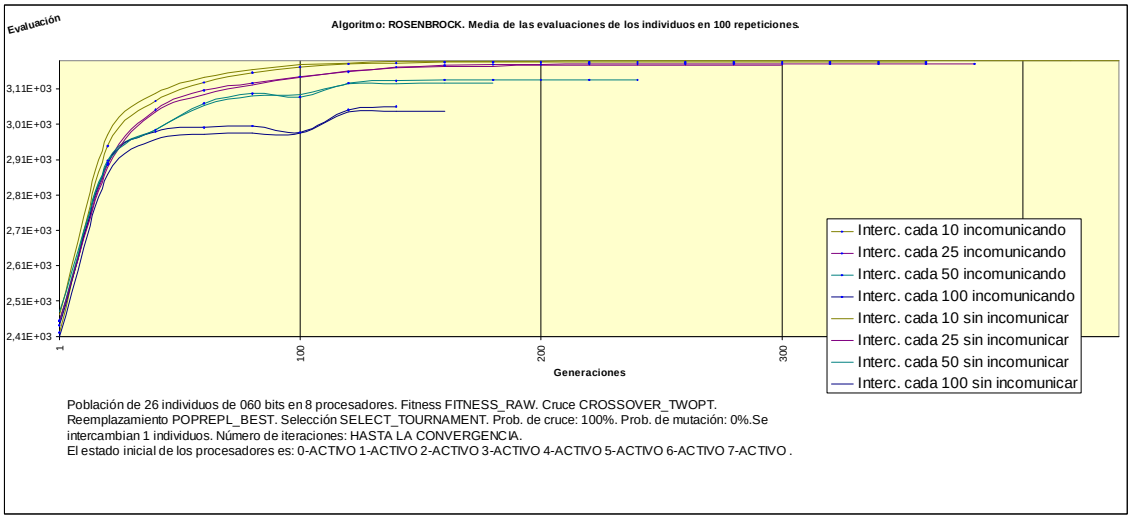
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
Rosen-brock 50 8 sin mutación	10	si	50	340	1	12,5	4	50,0	3180	3180	0,00000	3180	3180
	25	si	80	420	1	12,5	4	50,0	3170	3170	0,00000	3070	3010
	50	no	50	240	1	12,5	4	50,0	3120	3120	0,00000	2690	2650
	100	no	50	160	1	12,5	3	37,5	3060	3040	0,65789	2260	2070
Rosen-brock 50 8 con mutación	10	si		600			4	50,0	3200	3200	0,00000	3200	3200
	25	si	50	600	1	12,5	4	50,0	3200	3200	0,00000	3200	3200
	50	no	50	680	1	12,5	4	50,0	3200	3200	0,00000	3200	3190
	100	no	50	700	1	12,5	4	50,0	3200	3200	0,00000	3170	3100
Rosen-brock 50 4 sin mutación	10	si		260			2	50,0	3170	3180	0,31447	3170	3180
	25	si		300			2	50,0	3170	3190	0,62696	3170	3190
	50	no	80	400	1	25,0	2	50,0	3160	3180	0,62893	3150	3100
	100	no	80	240	1	25,0	2	50,0	3110	3140	0,95541	2980	2880
Rosen-brock 50 4 con mutación	10	si		540			2	50,0	3200	3200	0,00000	3200	3200
	25	si		560			2	50,0	3200	3200	0,00000	3200	3200
	50	no	50	580	1	25,0	2	50,0	3200	3200	0,00000	3200	3200
	100	no	50	600	1	25,0	2	50,0	3200	3200	0,00000	3200	3200
Rosen-brock 50 2 sin mutación	10	si	50	260	1	50,0	1	50,0	3150	3180	0,94340	3150	3180
	25	si	50	260	1	50,0	1	50,0	3160	3180	0,62893	3150	3180
	50	no	50	240	1	50,0	1	50,0	3130	3190	1,88088	3110	3190
	100	no	50	240	1	50,0	1	50,0	3140	3190	1,56740	3100	3180
Rosen-brock 50 2 con mutación	10	si		640			1	50,0	3200	3200	0,00000	3200	3200
	25	si	50	600	1	50,0	1	50,0	3200	3200	0,00000	3200	3200
	50	no	50	660	1	50,0	1	50,0	3200	3200	0,00000	3200	3200
	100	no	50	600	1	50,0	1	50,0	3200	3200	0,00000	3200	3200

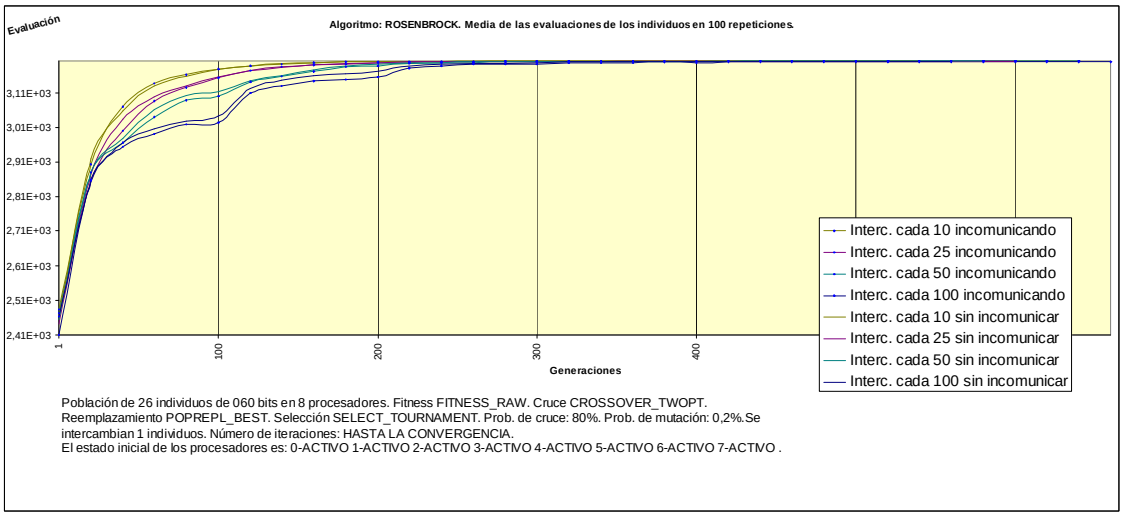
de 0,5 % a 1% de 0% a 0,5%

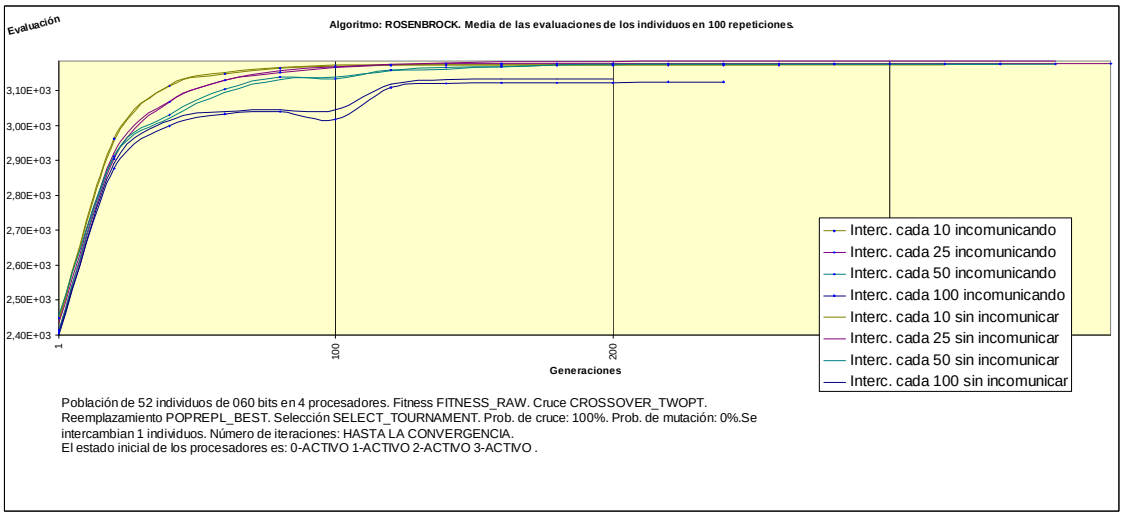


ROSENBROCK. TABLA 1. Resumen de resultados incomunicando cada 50 generaciones

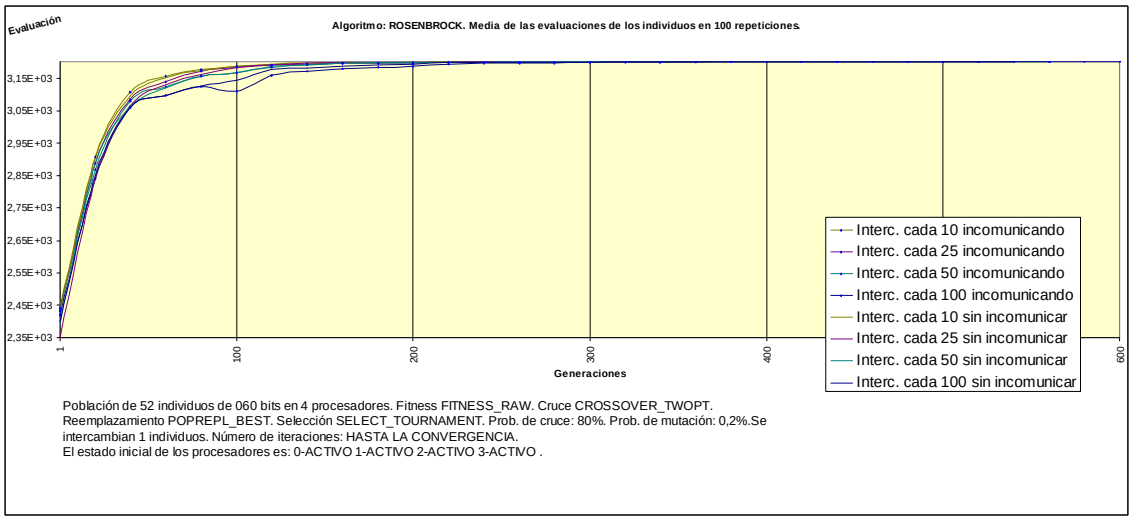
7.7.2. Fallo cada 100 generaciones

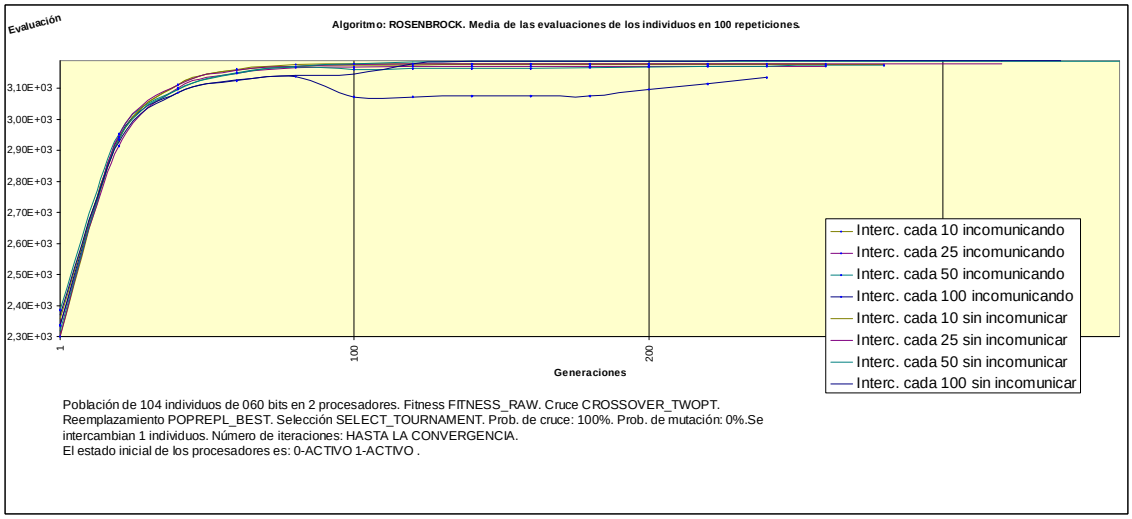


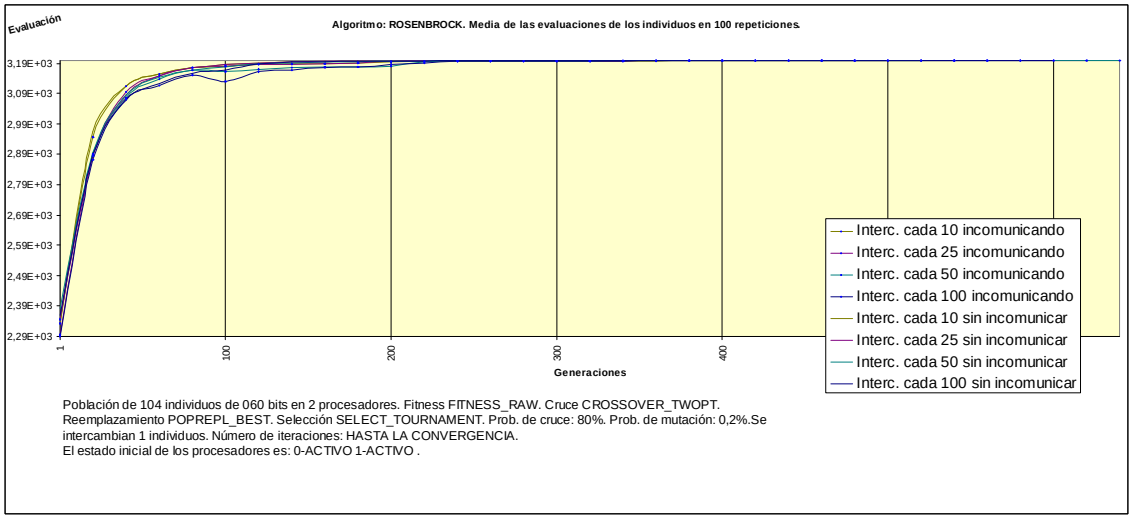




ROSENBROCK. .FIGURA 3 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 100 generaciones falla un procesador



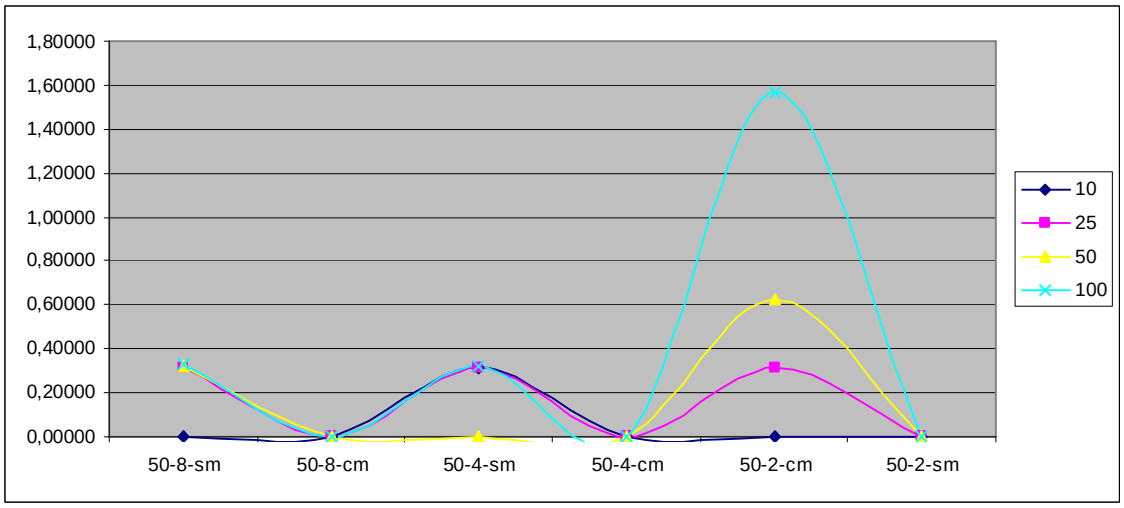




ROSENBROCK. FIGURA 4 . 4 procesadores con mutación. 2 procesadores sin y con mutación. Cada 100 generaciones falla un procesador

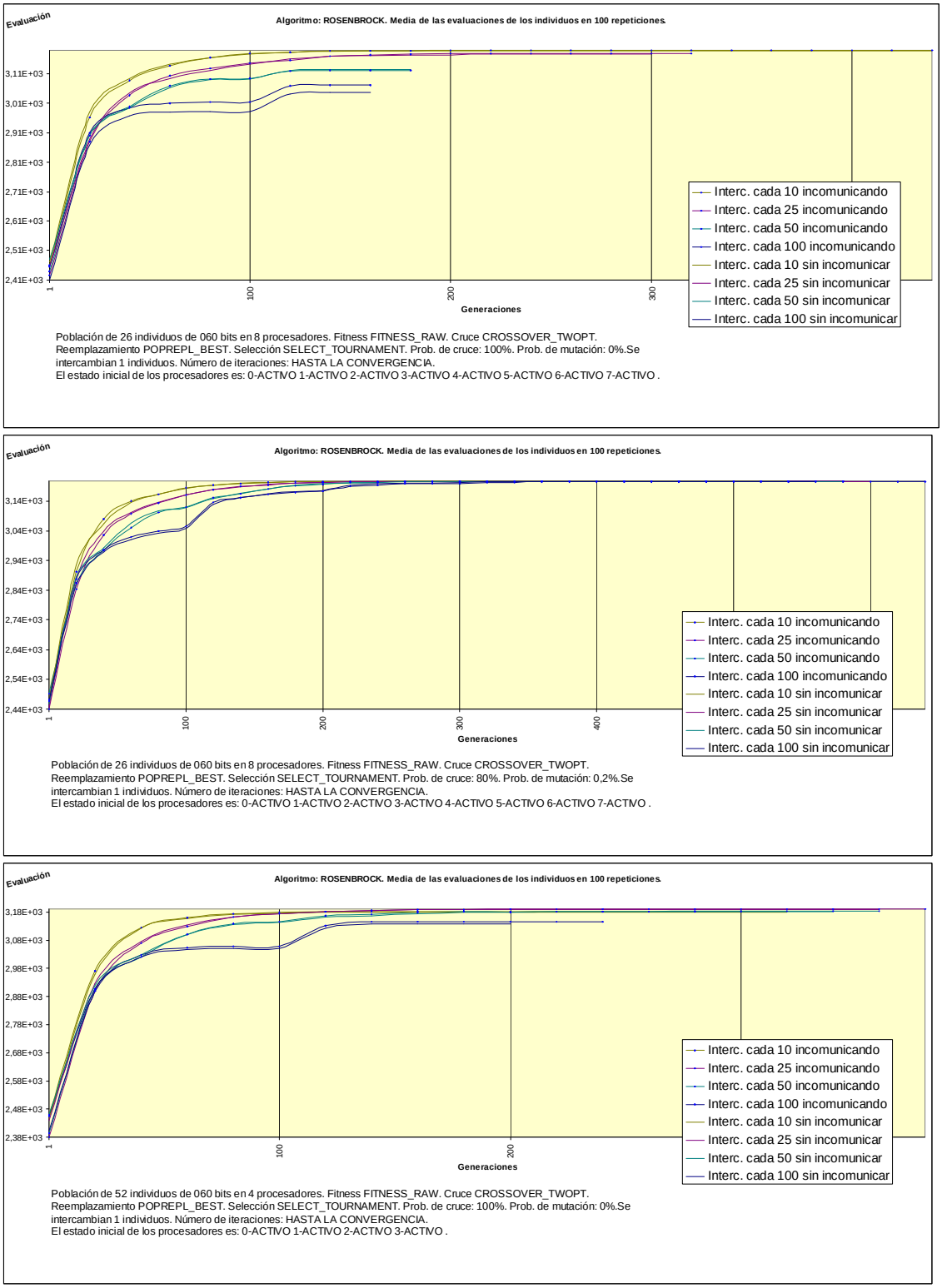
experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
Rosen-brock 100 8 sin mutación	10	si	50	360	0	0,0	3	37,5	3180	3180	0,00000	3180	3180
	25	si	50	380	0	0,0	3	37,5	3180	3170	0,31546	3040	3010
	50	no	50	240	0	0,0	2	25,0	3130	3120	0,32051	2690	2650
	100	no	50	140	0	0,0	1	12,5	3050	3040	0,32895	2200	2070
Rosen-brock 100 8 con mutación	10	si		640			4	50,0	3200	3200	0,00000	3200	3200
	25	si	50	600	0	0,0	4	50,0	3200	3200	0,00000	3200	3200
	50	si	50	640	0	0,0	4	50,0	3200	3200	0,00000	3190	3190
	100	no	80	660	0	0,0	4	50,0	3200	3200	0,00000	3130	3100
Rosen-brock 100 4 sin mutación	10	si		260			2	50,0	3170	3180	0,31447	3170	3180
	25	si		380			2	50,0	3180	3190	0,31348	3180	3190
	50	si		360			2	50,0	3180	3180	0,00000	3140	3100
	100	no	80	240	0	0,0	2	50,0	3130	3140	0,31847	2910	2880
Rosen-brock 100 4 con mutación	10	si		560			2	50,0	3200	3200	0,00000	3200	3200
	25	si		540			2	50,0	3200	3200	0,00000	3200	3200
	50	si		560			2	50,0	3200	3200	0,00000	3200	3200
	100	no	80	600	0	0,0	2	50,0	3200	3200	0,00000	3200	3200
Rosen-brock 100 2 sin mutación	10	si		260			1	50,0	3180	3180	0,00000	3180	3180
	25	si		260			1	50,0	3170	3180	0,31447	3170	3180
	50	si		280			1	50,0	3170	3190	0,62696	3160	3190
	100	no	80	240	0	0,0	1	50,0	3140	3190	1,56740	3070	3180
Rosen-brock 100 2 con mutación	10	si		540			1	50,0	3200	3200	0,00000	3200	3200
	25	si		620			1	50,0	3200	3200	0,00000	3200	3200
	50	si		640			1	50,0	3200	3200	0,00000	3200	3200
	100	no	80	600	0	0,0	1	50,0	3200	3200	0,00000	3200	3200

de 0,5 % a 1% de 0% a 0,5%

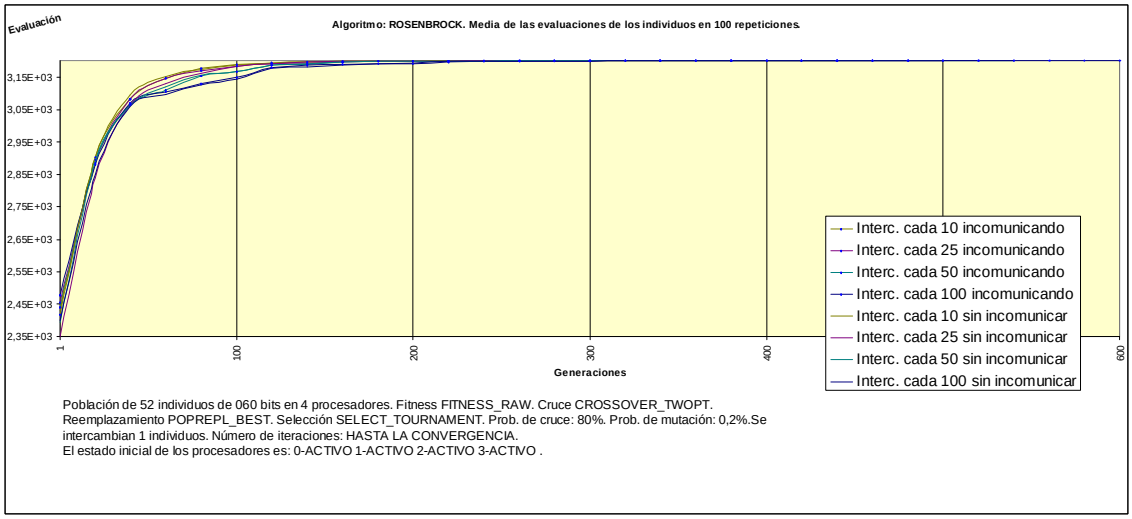


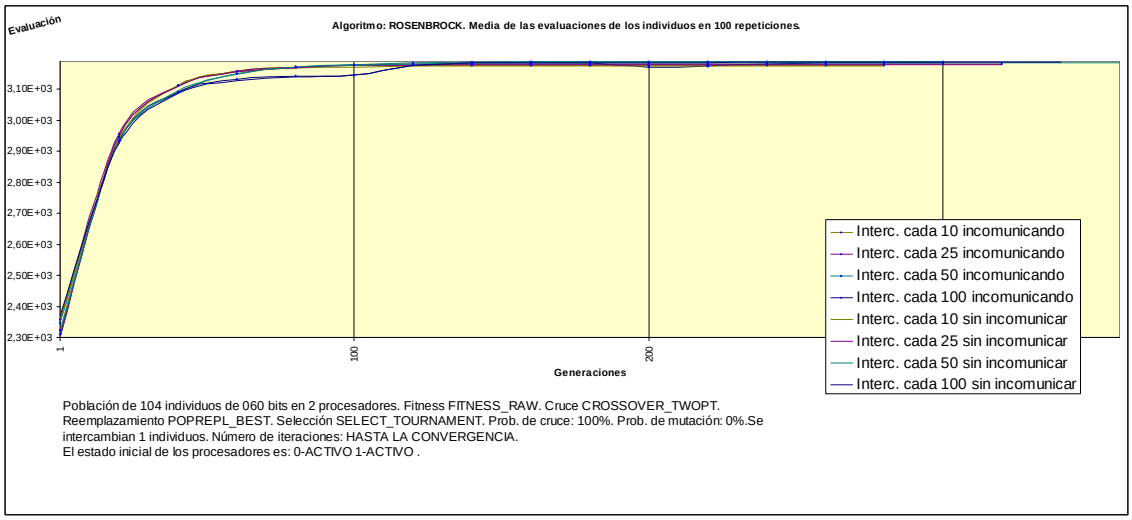
ROSENBROCK. TABLA 2. Resumen de resultados incomunicando cada 100 generaciones

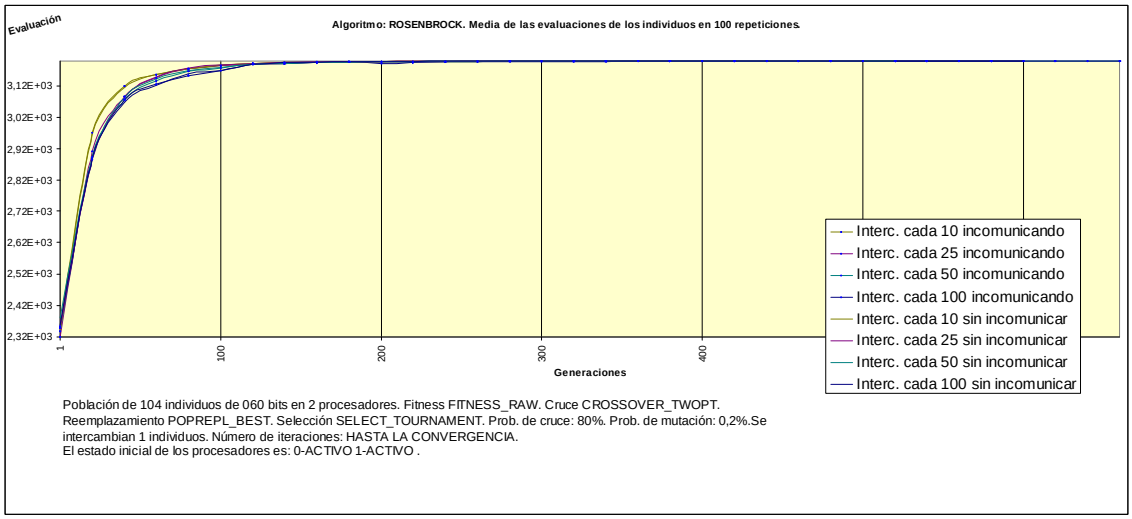
7.7.3. Fallo cada 200 generaciones



ROSENBROCK. .FIGURA 5 . 8 procesadores sin y con mutación. 4 procesadores sin mutación. Cada 200 generaciones falla un procesador



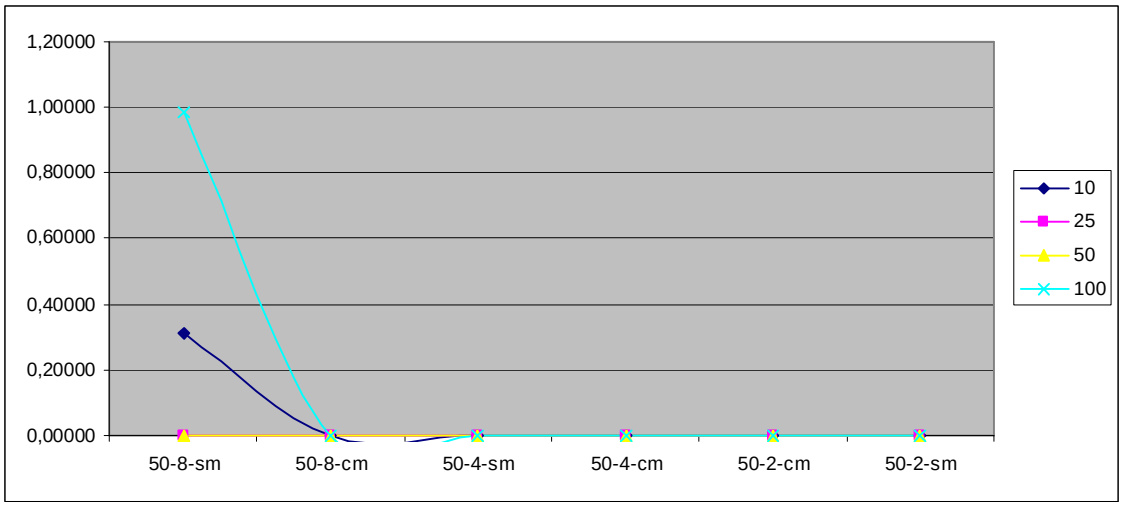




ROSENBROCK. .FIGURA 6 .4 procesadores sin mutación. 2 procesadores sin y con mutación. Cada 200 generaciones falla un procesador

experimento	intercambio	herencia de todos	separan en generacion	generación	PROC CAIDOS	%POB MUERTA	PROC CAIDOS TOT	%POB MUERTA TOT	MAX valor matando	MAX valor sin matar	tolera (error %)	MIN valor matando	Min valor sin matar
Rosen-brock 2008 sin mutación	10	si		440			2	25,0	3190	3180	0,31447	3190	3180
	25	si		320			1	12,5	3170	3170	0,00000	3020	3010
	50	no		180			0	0,0	3120	3120	0,00000	2600	2650
	100	no	50	160	0	0,0	0	0,0	3070	3040	0,98684	2150	2070
Rosen-brock 2008 con mutación	10	si		500			2	25,0	3200	3200	0,00000	3200	3200
	25	si		580			2	25,0	3200	3200	0,00000	3200	3200
	50	si		560			2	25,0	3200	3200	0,00000	3170	3190
	100	no		640			3	37,5	3200	3200	0,00000	3110	3100
Rosen-brock 2004 sin mutación	10	si		300			1	25,0	3180	3180	0,00000	3180	3180
	25	si		380			1	25,0	3190	3190	0,00000	3190	3190
	50	si		360			1	25,0	3180	3180	0,00000	3090	3100
	100	no		240			1	25,0	3140	3140	0,00000	2870	2880
Rosen-brock 2004 con mutación	10	si		460			2	50,0	3200	3200	0,00000	3200	3200
	25	si		500			2	50,0	3200	3200	0,00000	3200	3200
	50	si		540			2	50,0	3200	3200	0,00000	3200	3200
	100	si		600			2	50,0	3200	3200	0,00000	3200	3200
Rosen-brock 2002 sin mutación	10	si		280			1	50,0	3180	3180	0,00000	3180	3180
	25	si		320			1	50,0	3180	3180	0,00000	3180	3180
	50	si		300			1	50,0	3190	3190	0,00000	3190	3190
	100	si		300			1	50,0	3190	3190	0,00000	3160	3180
Rosen-brock 2002 con mutación	10	si		600			1	50,0	3200	3200	0,00000	3200	3200
	25	si		600			1	50,0	3200	3200	0,00000	3200	3200
	50	si		660			1	50,0	3200	3200	0,00000	3200	3200
	100	si		660			1	50,0	3200	3200	0,00000	3200	3200

de 0,5 % a 1% de 0% a 0,5%

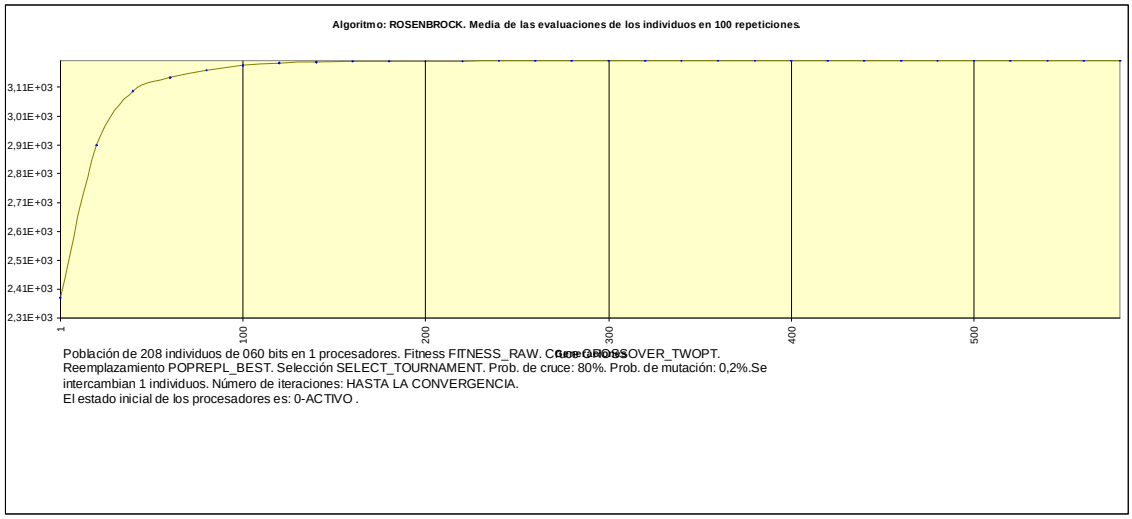


ROSENBROCK. TABLA 3. Resumen de resultados incomunicando cada 200 generaciones

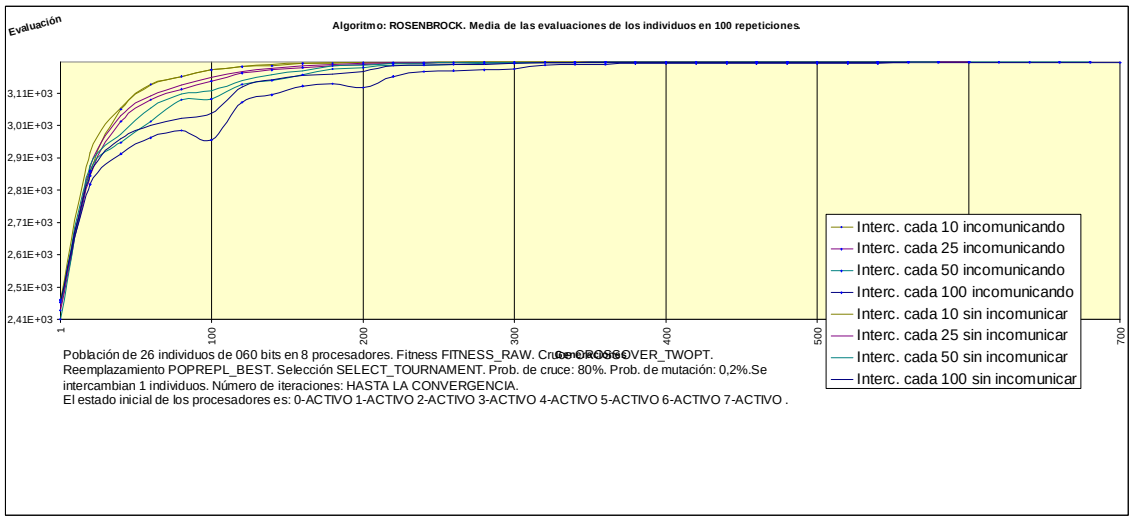
Si observamos las tablas y las gráficas vemos que esta función es con la que conseguimos una mayor tolerancia, soportando con eficiencia la muerte de hasta el 50% de los procesadores tanto con 8, como con 4 y con 2 procesadores. Este comportamiento puede ser debido a que la función de Rosenbrock permite la convivencia en las poblaciones de esquemas buenos pero diferentes. Esto nos podría remitir a un estudio futuro en el que se realice una traza de estos esquemas durante toda la ejecución del algoritmo.

Efecto de la mutación

Para esta función también observamos cómo la mutación puede sustituir al intercambio de información entre procesadores, consiguiendo mejores resultados, aunque esta vez los valores son muy cercanos.



También vemos que, gracias a la mutación, conseguimos converger al mismo valor aun habiendo perdido el 50% de la población. Puede observarse en la siguiente gráfica:



Las "líneas" que se a refieren experimentos con intercambio y mutación crecen hacia una solución mejor mas rápidamente que las de experimentos en los que se utiliza únicamente mutación, pero al final se consigue en ambas llegar al mismo valor.

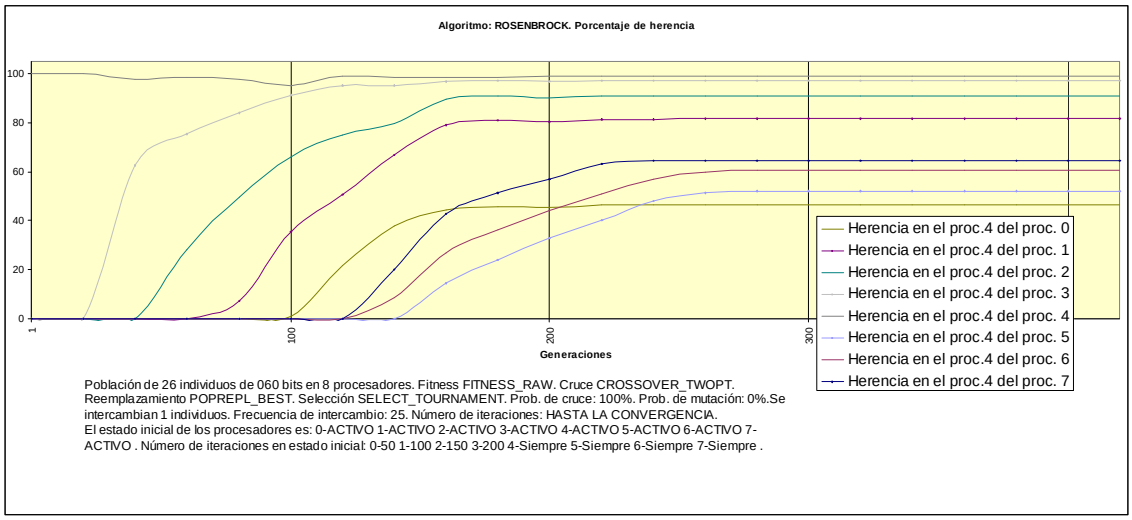
Tolerancia según el fallo de procesadores

De nuevo vemos que se consiguen mejores valores cuando la población total está más dividida, es decir, con 8 procesadores se mejora sobre 4 procesadores y a su vez sobre 2. Esto queda patente si se observan las TABLAS 1, 2 y 3.

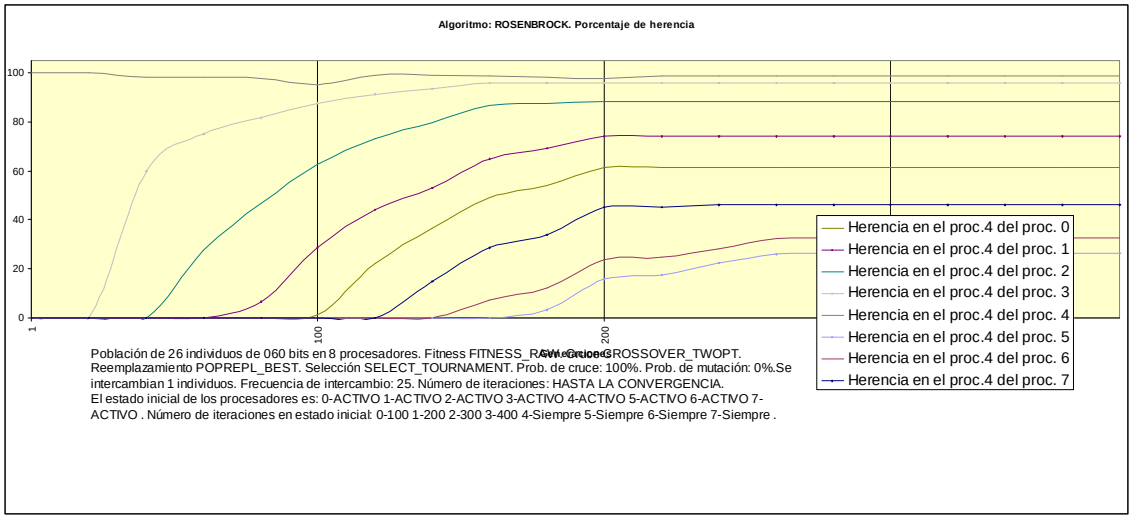
Asimismo se puede observar, al igual que en experimentos anteriores, cómo mejoran las ejecuciones cuando prolongamos durante más tiempo la vida de los procesadores sin provocarles un fallo. Si nos referimos a la TABLA 3, podemos ver que se consigue anular el error cuando únicamente se matan 2 procesadores en el caso de tener 8, 1 en el caso de tener 4 y también 1 en el caso de disponer de dos procesadores.

Efectos de la herencia

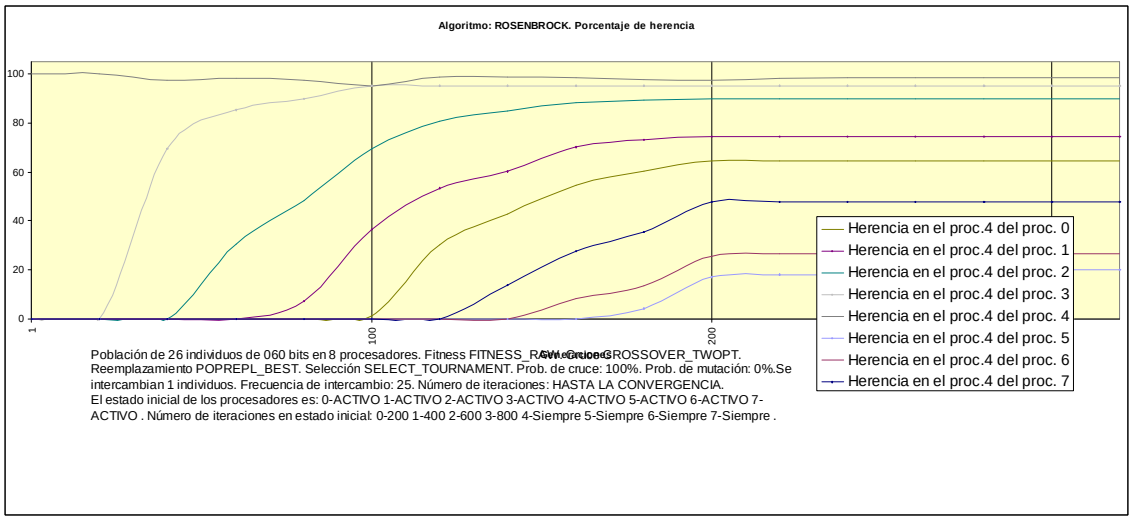
Podemos observar un comportamiento parecido a la función Rastrigin en el envío de la herencia. Se nota que los procesadores caídos no consiguen enviar suficiente información si su muerte ha sido prematura.



Fallos cada 50 generaciones

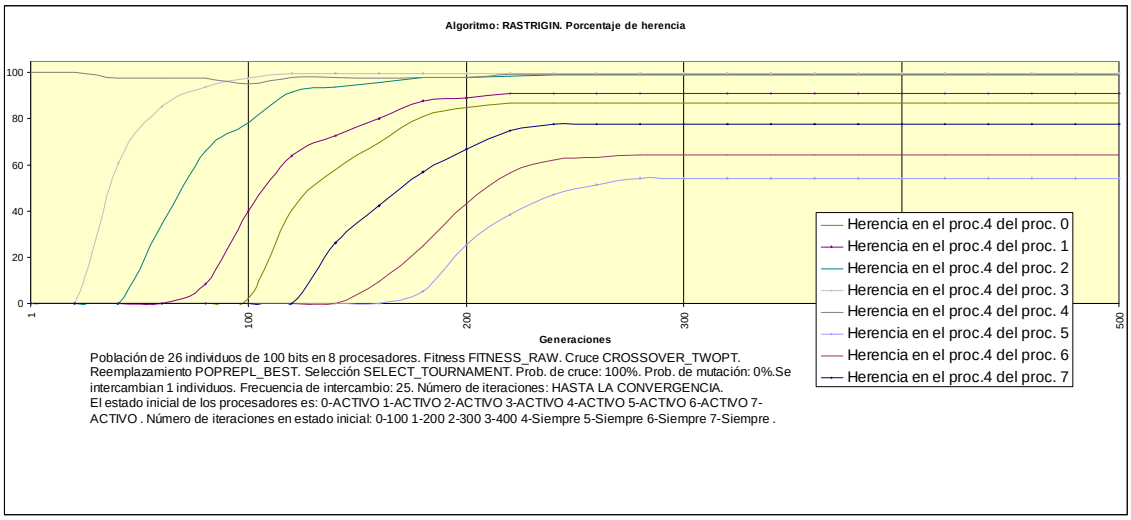


Fallos cada 100 generaciones

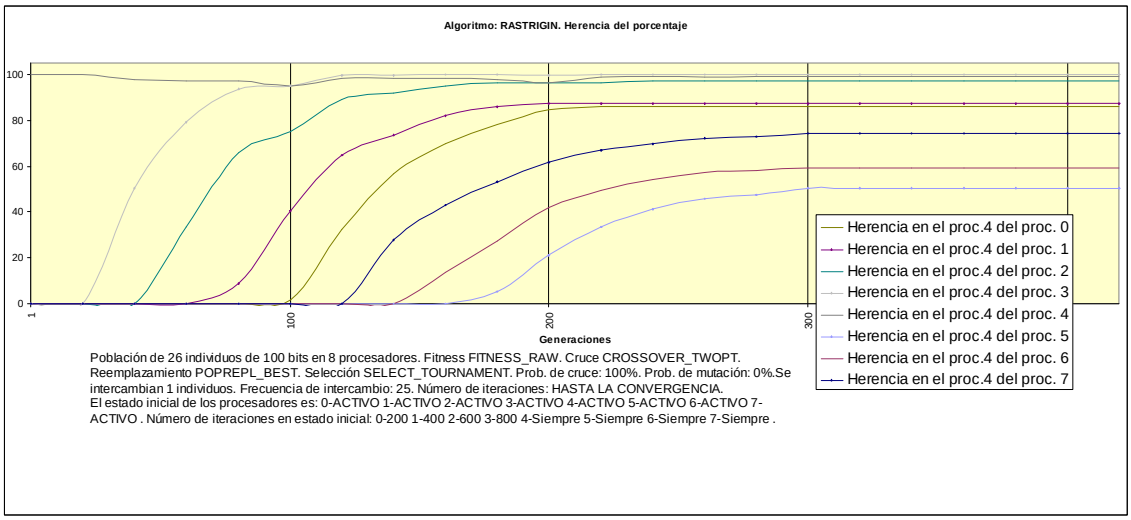


Fallos cada 200 generaciones

Podemos ver aquí cómo, aun con muerte cada 100 generaciones, se sigue percibiendo los efectos de la falta de propagación de algunos procesadores (nótese la herencia del procesador 0). Sin embargo, esto ocurre en menor medida que en el experimento Rastrigin. Obsérvense las siguientes gráficas correspondientes a éste último:



Fallos cada 100 generaciones



Fallos cada 200 generaciones

En la gráfica de herencia correspondiente a Rastrigin con fallos cada 100, se aprecia más herencia proveniente del procesador 0 (aproximadamente el 90%), mientras que en su homóloga para el experimento Rosenbrock, la herencia del procesador 0 es menor (aproximadamente 60%).

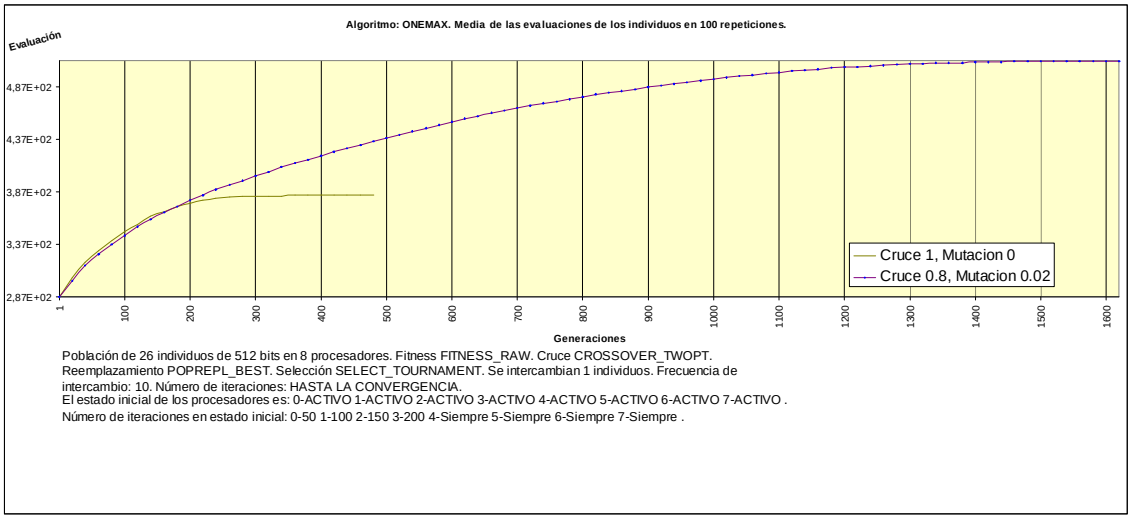
Para las gráficas homólogas con fallos cada 200 generaciones también se ve menor herencia en el caso del Rosenbrock. Esto que nos lleva a pensar en una mayor tolerancia a fallos por parte de esta función, debido a que no es tan dependiente de la herencia recibida por parte del resto de procesadores; al permitir esquemas buenos pero diferentes en las propias poblaciones, no necesita recibir “diversidad” de otros procesadores para seguir avanzando por medio del intercambio.

8. Conclusiones

A continuación enumeraremos las conclusiones obtenidas tras la realización de este estudio.

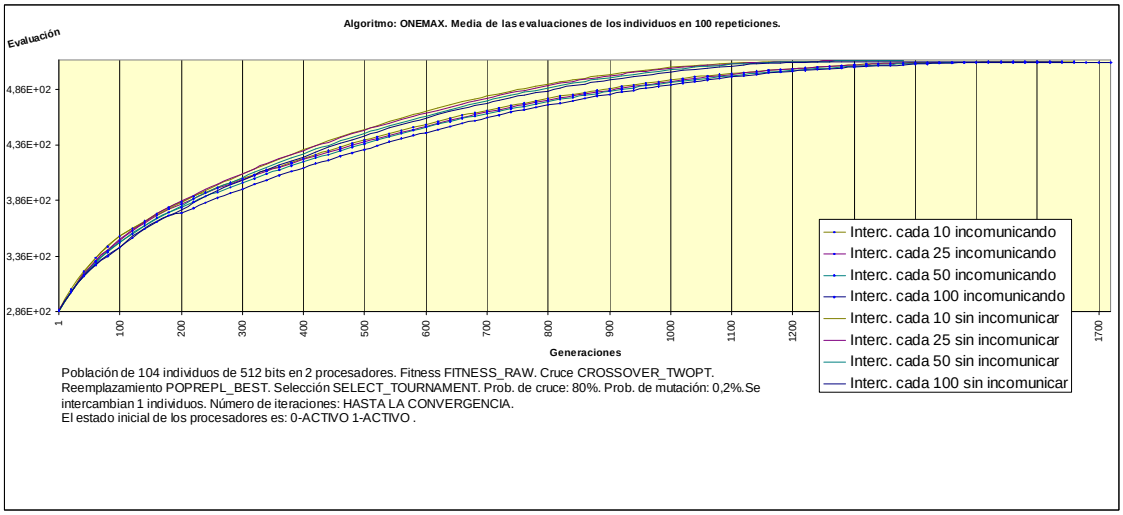
Conclusiones relativas a la mutación

Como primera conclusión podemos decir que si queremos obtener buenos resultados en todos los casos es mejor utilizar mutación. El valor que se debe establecer en la mutación no forma parte del ambito de este estudio y podría ser motivo de un estudio mas detallado.

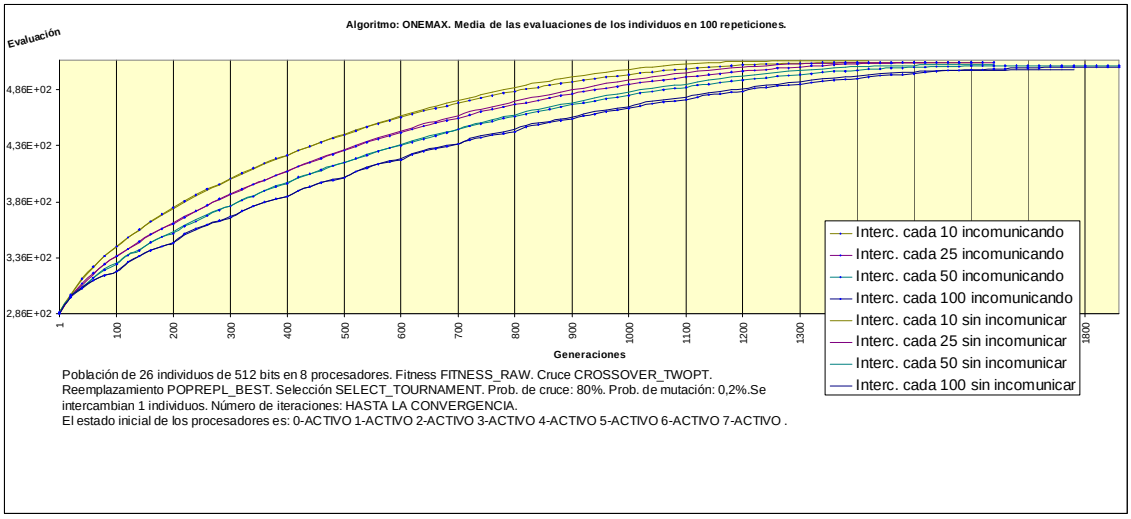


En la gráfica de arriba se ve claramente las afirmaciones anteriores, si bien debemos destacar cómo debido a la mutación se producirá un aumento en el número de generaciones necesarias para alcanzar una solución

Como segunda conclusión podemos afirmar que cuando realicemos experimentos debemos tener en cuenta cómo la mutación ayuda a la tolerancia de fallos, consiguiendo gracias a ella llegar a la tolerancia de hasta el 50% de fallos en los procesadores utilizados. Gracias a la mutación conseguimos que las ejecuciones que han experimento el fallo de algún procesador se recuperen y terminen obteniendo los mismos resultados que las ejecuciones que no han sufrido ninguna muerte.



Como tercera conclusión y siguiendo en relación al uso de la mutación, debemos comentar como en todos los casos se obtienen buenos e incluso mejores resultados utilizando la misma población en un único procesador en lugar de separarla en varios procesadores como hemos hecho en este estudio. Compárense las siguientes graficas en la primera mostramos los resultados realizando intercambio y en la segunda el mismo experimento en un solo procesador.



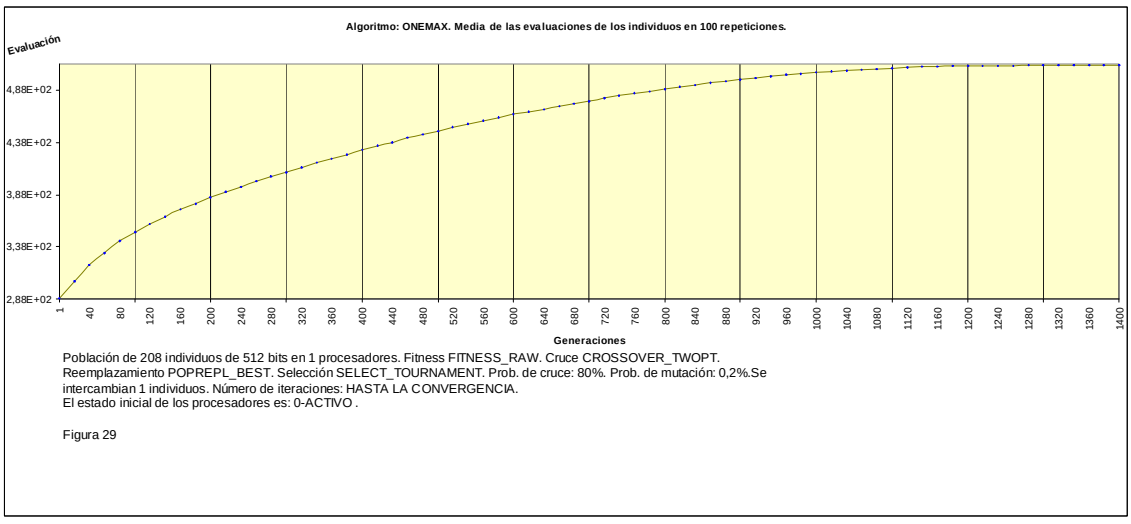
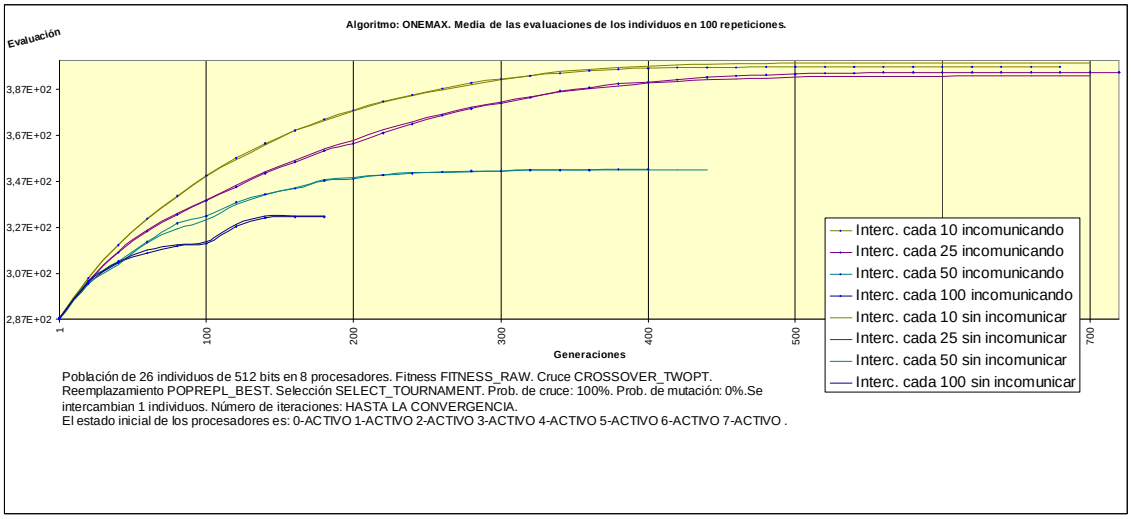


Figura 29

Cuando queramos ejecutar un algoritmo genético debemos sopesar si compensa el uso de varios procesadores en lugar de un único procesador. Se deberían mirar, aparte de los aspectos económicos, el tiempo del que disponemos para realizar el algoritmo, etc.. Y si aún nos decidimos por el uso de varios procesadores intentar utilizar la configuración de parámetros que más se adapte al problema que pretendemos resolver.

Conclusiones relativas a la frecuencia de intercambio

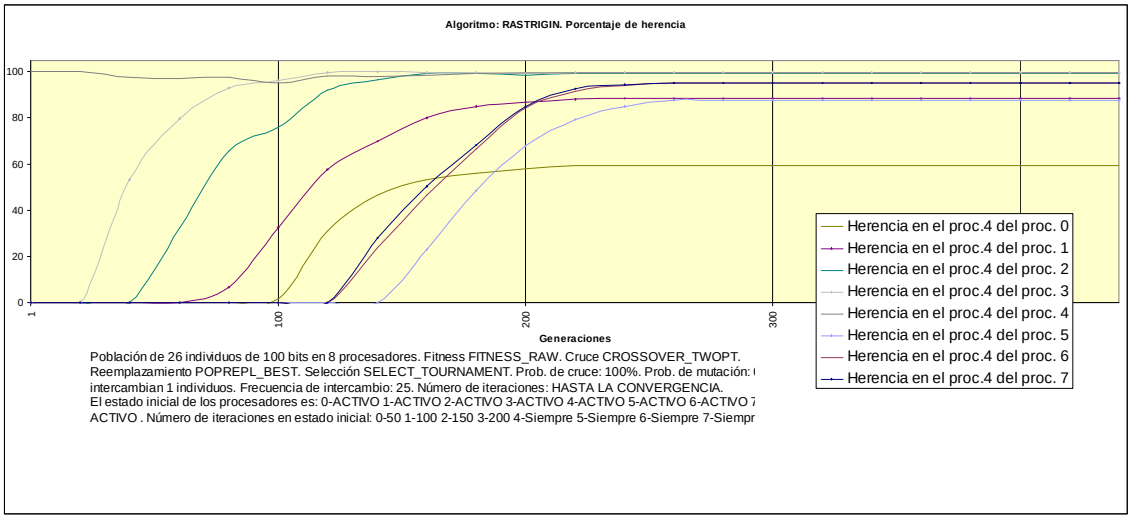
Siempre que utilicemos algoritmos genéticos paralelos debemos pensar en la frecuencia de intercambio que utilizaremos, teniendo en cuenta que a mayor frecuencia de intercambio mejores resultados obtendremos. En la siguiente gráfica podemos observar cómo se obtienen mejores valor con el menor periodo de intercambio.



Conclusiones sobre herencia

En problemas sencillos como el OneMax hemos observado la importancia de la transmisión de los buenos esquemas al inicio, ya que estos son parecen ser los esquemas determinantes para la formación de buenas soluciones. Por lo tanto, en este tipo de problemas deberíamos utilizar una alta frecuencia de intercambio sobre todo en las primeras generaciones y desechar soluciones si hemos tenido problemas con los procesadores en las primeras generaciones.

Para problemas más complejos como son Rastrigin, Schwefel y FModal, deberíamos tener en cuenta además de utilizar frecuencias elevadas de intercambio el uso de otro esquema diferente de intercambio más complejo que no sea el intercambio en anillo, de este modo puede que consigamos paliar los problemas de propagación de la herencia que presentan estos problemas. En estos problemas se observa cómo la transmisión de la herencia depende de la distancia entre los procesadores, necesitando realizar muchas iteraciones o utilizar altas frecuencias de intercambio para conseguir propagar toda la herencia necesaria para conseguir buenas soluciones siendo problemática la muerte prematura de procesadores.



En funciones como el Rosenbrock hemos observado como tolera perfectamente los fallos. Hemos achacado este comportamiento a que no es tan dependiente de la herencia recibida por parte del resto de procesadores, al permitir esquemas buenos pero diferentes en las propias poblaciones, no necesita recibir "diversidad" de otros procesadores para seguir avanzando por medio del intercambio.

Conclusiones sobre la distribución de la población.

En general, hemos observado que las funciones con menos procesadores (y la misma población total), se comportaban mejor, mostrándose más las diferencias en los casos en los que no hay mutación, ya que como hemos indicado varias veces, la mutación ayuda a tolerar los fallos. Esto es debido a que las poblaciones de cada procesador son más grandes (en el ejemplo de 2 procesadores, cada uno contiene el 50% de la población), por lo que las poblaciones evolucionan mejor.

Esto tiene un inconveniente, ya que si algún procesador (volvemos al ejemplo de 2 procesadores) falla, perdemos mucha más población que si tuviésemos los 8 procesadores funcionando (con 2 procesadores, perdemos el 50%, con 8 perdemos 12,5% de la población). Si la caída del procesador se ha producido cuando ha intercambiado gran parte de información, puede ser que no sea decisiva, pero si falla en las primeras generaciones, podría provocar la convergencia prematura hacia valores que se alejan bastante de los óptimos. Esto es debido a que la diversidad poblacional puede ser insuficiente para el experimento.

9. Futuras líneas de trabajo

Métodos propuestos para solucionar los problemas de las funciones defectivas.

Como se ha podido ver previamente, algunas funciones defectivas, tales como la Ftrap utilizada, no llegan a alcanzar el máximo global, quedándose estancada en el máximo local (en nuestro caso, x=0; y=400, cuando debería alcanzar el máximo en x=512; y=512).

Se han propuesto diferentes técnicas para afrontar los problemas defectivos. Algunas de ellas se centran en métodos de codificación. Como comentamos en apartados anteriores, puede adjuntarse a cada bit binario del cromosoma un bit etiqueta que especifica la localización en el fenotipo. Así, una cadena binaria cualquiera 11011001 se representa como una lista de pares ordenados valor-posición:

(1 1) (2 1) (3 0) (4 1) (5 1) (6 0) (7 0) (8 1)

Mediante esta codificación, el orden de los pares ordenados puede modificarse pero, a la hora de cruzar dos individuos, se alinean previamente sus cromosomas en el mismo orden.

El problema de estas aproximaciones es que implican un espacio de búsqueda adicional para encontrar la permutación correcta de ordenación de los bits, cuyo tamaño es similar al espacio de optimización de la función original. Algunos autores sugieren una reordenación de las posiciones de los genes, en lugar de las posiciones de los bits, para preservar los bloques de construcción.

Otros autores sugieren un AG de dos niveles: cada gen del nivel inferior del cromosoma se etiqueta con un gen de control de alto nivel que lo activa o desactiva. Esta aproximación es muy similar a la de genes redundantes, donde múltiples bits genotípicos deciden la activación o desactivación de un bit fenotípico.

Otra opción es mantener varias poblaciones y se permite migración entre diferentes subpoblaciones, a fin de mantener la diversidad. En la medida en que al menos una subpoblación no converja hacia el atractor defectivo, existe alguna posibilidad de alcanzar el óptimo global.

Paralelización o no del ftrap

Se podría estudiar la paralelización del ftrap (escogemos esta función porque es la que más problemas tiene para llegar a los máximos, por los puntos indicados previamenente). Como ya hemos introducido al principio, hay casos en los que algunos algoritmos no conviene paralelizarlos, ya que los valores obtenidos no son los deseados, porque el tiempo no disminuye significativamente al paralelizar el código... Por estos motivos, se podría realizar un estudio para ver si realmente es conveniente ejecutar el ftrap en varios procesadores, o se alcanzan mejores valores ejecutando secuencialmente sólo en un procesador.

Estudio de diferentes probabilidades de mutación y número de generaciones.

Otro punto a considerar en futuros trabajos sería la conveniencia o no de aumentar la mutación o el número de generaciones realizadas para aumentar la tolerancia a fallos Habría que fijar la probabilidad de mutación y el número de generaciones adecuado para la parada.

Ampliación a problemas reales

Como hemos visto, todas las funciones ejecutadas para estas prueban han sido algoritmos creados para lo mismo, sin poder darlos un uso real, así que podría ser un buen trabajo de investigación la ampliación de estas pruebas a problemas reales de optimización

Estudio detallado de la evolución de la herencia y del periodo de intercambio

Una ampliación sencilla del trabajo realizado se realizaría variando la proporcion y la forma en la que se mata a los procesadores para observar la evolución de la herencia.

También se podría profundizar un poco más en la influencia del periodo de intercambio en la herencia.

Estudio detallado de la función Rosenbrock

La función Rosenbrock parece tolerar perfectamente el fallo de hasta el 50% de los procesadores, lo que parece ser porque permite la convivencia en las poblaciones de esquemas buenos pero diferentes. Se podría realizra un estudio en el que se realice un seguimiento de estos esquemas durante toda la ejecución del algoritmo.

10. Bibliografía

H. Aguirre, K. Tonaka, and T. Sugimura. Cooperative crossover and mutation operators in genetic algorithms. In Proceedings of the 1999 Genetic and Evolutionary Computation Conference, page 772. Morgan Kaufmann, 1999.

H. Aguirre, K. Tonaka, T. Sugimura, and S. Oshita. Cooperativecompetitive model for genetic operators: Contributions of extinctive selection and paralell genetic operators. In Proceedings of the 2000 Genetic and Evolutionary Computation Conference Late Breaking Papers, pages 6-14, 2000.

T. Back, F. Ho_meister, and H.P. Schwefel. A survey of evolution strategies. In Morgan Kaufmann, editor, Proceedings of the Fourth International Conference on Genetic Algorithms, pages 2-9, San Mateo, CA, 1991.

T. Back and H.P. Schwefel. An overview of evolutionary algorithms for parameter optimization. Evolutionary Computation, 1(1):1-23, 1993.

R. Baraglia, J.I. Hidalgo, and R. Perego. A hybrid heuristic for the travelling salesman problem. IEEE Transactions on Evolutionary Computation. to appear.

R. Baraglia, J.I. Hidalgo, and R. Perego. A hybrid approach for the tsp combining genetics and the lin-kerninghan local search. Technical Report CNUCE-B4-2000-007, CNUCE - Institute of the Italian National Research Council, 2000.

R. Baraglia, J.I. Hidalgo, and R. Perego. A parallel hybrid heuristic for the tsp. In Springer Verlag, editor, Proceedings of EvoCOP2001, the First European Workshop on Evolutionary Computation in Combinatorial Optimization, page in press, Lake Como (Milan), April 2001.

A. Bertoni and M. Dorigo. Implicit paralellism in genetic algorithms. Artificial Intelligence, 61(2):307-314, Februrary 1993.

R. Bianchini and C. M. Brown. Parallel genetic algorithms on distributedmemory architectures. In S. Atkins and A. S. Wagner, editors, Transputer Research and Applications 6, pages 67-82, Amsterdam, 1993. IOS Press.

R. Bianchini and C.M. Brown. Parallel genetic algorithm on distributedmemory architectures. Technical Report TR 436, Computer Sciences Department University of Rochester, 1993.

H. Braun. On solving traveling salesman problems by genetic algorithms. In Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN , volume 496 of Lecture Notes in Computer Science, pages 129-133. Springer-Verlag, 1991.

C.A.Coello. A comprehensive survey of evolutionary-based multiobjective optimization techniques. Knowledge and Information Systems, 1(3):269-308, 1999.

C.A.Coello and A.D. Christiansen. Moses : A multiobjective optimization tool for engineering design. Engineering Optimization, 31(3):337-368,1999.

E. Cantu-Paz. A summary of research on parallel genetic algoritms. Technical Report 95007, University of Illinois at Urbana-Champaign, Genetic Algoritms Lab. (IlliGAL), http://gal4.ge.uiuc.edu/illigal.home.html, July 1995.

E. Cantu-Paz. Designing eÆcient master-slave parallel genetic algorithms. In Morgan Kaufmann, editor, Genetic Programming: Proceedings of the Third Annual Conference, page 455, San Francisco, CA, 1998.

E. Cantu-Paz. Migration policies and takeover times in parallel genetic algorithms. In Morgan Kaufmann, editor, Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99, page 775, San Francisco, CA, July 1999.

E. Cantu-Paz. Topologies, migration rates, and multi-population parallel genetic algorithms. In Morgan Kaufmann, editor, Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99, pages 91-98, San Francisco, CA, July 1999.

E. Cantu-Paz. EÆcient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Boston, MA, 2000.

E. Cantu-Paz and D.E. Goldberg. EÆcient parallel genetic algorithms: theory and practice. Computer Methods in Applied Mechanics and Engineering, -(186):221-238, 2000.

S. Cohoon, J. Hedge, S. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. IEEE Transaction on CAD, 10(4):483-491, April 1991.

G. A. Croes. A method for solving traveling salesman problem. Operations Research, 6:791-812, 1958.

L. Davis. Handbook of Genetic Algorithm. Van Nostrand Reinhold, 1991.

M. Dorigo and V. Maniezzo. Parallel genetic algorithms: Introduction and overview of current research. In Parallel Genetic Algorithms, pages 5-42. IOS Press, 1993.

F. Fernandez, M. Tomassini, W. Punch, and J.M. Sanchez. Experimental study of multipopulation parallel genetic programming. In Springer-Verlag, editor, Genetic Programming, proceedings of EuroGP2000, Lecture Notes in Computer Science,Vol. 1802, pages 1-15, 2000.

D.B. Fogel. Evolving Arti_cial Inteligence. PhD thesis, University of California, San Diego, 1992.

L.J. Fogel, A.J. Owens, and M.J. Walsh. Arti_cial Inteligence through SImulated Evolution. John Wiley, Chinchester, UK, 1966.

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. PVM: Parallel Virtual Machine. A users guide and tutorial for network parallel Computers. MIT Press Books, Massachussets, 1994.

D. E. Goldberg, K. Deb, and B. Korb. Don't worry, be messy. In Proc. Of the Fourth International Conference on Genetic Algorithms, pages 24-30, San Diego, CA, 1991.

D.E. Goldberg. Optimal initial population size for binary-coded genetic algorithms. Technical Report TCGA 85001, University of Alabama, 1985.

D.E. Goldberg. Genetic algorithms in search, optimization and machine learning. Addison Wesley, 1989.

D.E. Goldberg. Messy genetic algorithms revisited: Studies in mixed sizze and scale. Complex Systems, 4:415-444, 1990.

G. Harik, F. Lobo, and D. Goldberg. The compact genetic algorithm. Technical Report 97006, University of Illinois at Urbana-Champaign, Urbana, IL, 1997.

G. Harik, F. Lobo, and D. Goldberg. The compact genetic algorithm. IEEE Transactions on Evolutionary Computation, 3(4):287-297, 1999.

M. Herdy. Application of the evolution strategy to discrete optimization problems. In Springer Verlag, editor, PPSN; Lecture Notes in Computer Science,vol 496, pages 188-192, 1991.

J.I. Hidalgo, J. Lanchares, and R. Hermida. Partitioning and placement for multi-fpga systems using genetic algorithms. In Proceedings of the Euromicro DSD 2000, page accepted, 2000.

J. Holland. Adaptation in Natural and Arti_cial Systems. Univ. of Mitchigan Press, 1975.

High performance fortran. http://www.crpc.rice.edu/HPFF/.

D. S. Johnson and L. A. McGeoch. Local Search in Combinatorial Otimization. John Wiley and Sons, New York, 1996.

J. Juliany and M.D. Vose. The genetic algorithm fractal. Evolutionary Computation, 2(2):165-180, 1994.

J.R. Koza. Genetic Programming: On the Programming of Computers by means of Natural Selection. MIT Press, Massachussets, 1992.

J. Liening. A parallel genetic algorithm for performance-driven vlsi routing. IEEE Transactions on Evolutionary Computation, 1(1):29-39, April 1997.

S. Lin. Computer solution of the traveling salesman problem. Bell Syst. Tech. J., 44:2245-2269, 1965.

S. Lin and B. Kernighan. An e_ective heuristic for the traveling salesman problem. Oper. Res., 21:498-516, 1973.

B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In Proceedings of the Third International Conference on Genetic Algorithms, pages 428-433. Morgan Kaufmann Publishers, 1989.

Z. Michalewicz. Genetic Algorithms + data structures = Evolution Programs. Springer-Verlag, 1996.

Mpi: A message passing interface standard. http://www.mpi-forum.org.

H. Muhlenbein. Parallel genetic algorithms, population genetic and combinatorial optimization. In Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN, volume 496, pages 407-417. Lecture Notes in Computer Science, 1991.

S. Russelland P. Norving. Arti_cial Intelligence A Modern Approach. Prentice Hall, 1995.

G. Ochoa, I. Harvey, and H. Buxton. On recombination and optimal mutation rates. In Proceedings of the 1999 Genetic And Evolutionary Computation Conference, pages 488-495. Morgan Kaufmann, 1999.

G. Ochoa, I. Harvey, and H. Buxton. Optimal mutation rates and selection pressure in genetic algorithms. In Proceedings of the 2000 Genetic And Evolutionary Computation Conference, pages 315-322. Morgan Kaufmann, 2000.

The openmp application program interface. http://www.openmp.org.

I.C. Parmee, D. Cvetkovic, A.H. Watson, and C. R. Bonham. Multiobjective satisfaction within an interactive evolutionary design environment. Evolutionary Computation, 8(2):197-222, 2000.

M. Prieto. Paralelizacion de Metodos Multimalla Robustos. PhD thesis, Universidad Complutense de Madrid, 2000.

G. Rawlins. Foundations of Genetic Algorithms. Morgan Kaufmann, 1991.

G. Rudolph. Convergence analysis of canonical genetic algorithms. IEEE Transactions on Neural Networks, 5(1):96-101, Enero 1994.

J.D. Scha_er and A. Morishima. An adaptative crossover distribution mechanism for genetic algorithms. In Lawrence Erlbaum, editor, Genetic Algorithms and their applications: Proceedings of the second International Conference on Genetic Algorithmd, pages 36-40, New Jersey, 1987.

H.P. Schwefel. Evolutionsstrategie und numerische Optimierung. PhD thesis, Uechnische Universitat Berlin, 1975.

H.P. Schwefel. Evolution strategies: A family of non-linear optimization techniques based on imitating some principles of organic evolution. Annals of Operations Research, 1:165-167, 1984.

A.V. Sebald and L.J. Fogel. Proceedings of the third annual Conference on Evolutionary Programmong. World Scienti_c, San Diego, 1994.

R. Tanese. Parallel genetic algorithms for a hypercube. In Proceedings of the Second International Conference on Genetic Algorithms, pages 177-183. L. Erlbaum Associates, 1987.

R. Tanese. Distributed genetic algorithms. In Proceedings of the Third International Conference on Genetic Algorithms, pages 434-440. M. Kaufmann, 1989.

M. Tommassini. Parallel and distributed evolutionary algorithms: A review. In J. Wiley and Sons, editors, Evolutionary Algorithms in Engineering and Computer Science, pages 113-133, 1999.

M. D. Vose and G. E. Liepins. Schema disruption. In Proc. of the Fourth International Conference on Genetic Algorithms, pages 237-242, San Diego, CA, 1991.

M.D. Vose. Modeling simple genetic algorithms. Evolutionary Computation, 3(4):453-472, 1995.

M.D. Vose. The Simple Genetic Algorithm. Foundations and Theory. MIT Press, London, 1999.

M.D. Vose and A.H. Wright. Simple genetic algorithms with linear _tness. Evolutionary Computation, 2(4):347-368, 1994.

M.D. Vose and A.H. Wright. The simple genetic algorithm and the walsh transform: Part i, theory. Evolutionary Computation, 6(3):253-273, 1998.

M.D. Vose and A.H. Wright. The simple genetic algorithm and the walsh transform: Part ii, the inverse. Evolutionary Computation, 6(3):275-289, 1998.

L.D. Whitley. Foundations of Genetic Algorithms 2. Morgan Kaufmann, 1993.

L.D. Whitley and M.D. Vose. Foundations of Genetic Algorithms 3. Morgan Kaufmann, 1995.

Transparencias de la asignatura Programación Evolutiva de la profesora de la Universidad Complutense de Madrid Lourdes Araujo Serna.

Optimización de la distribución en planta de instalaciones industriales mediante algoritmos genéticos. Aportación al control de la geometría de las actividades. Tesis doctoral de la Universidad Politécnica de Valencia. http://www.dpi.upv.es/nueva/tesis/tesis_diego/diego_mas_geneticos.pdf

Transparencias de la asignatura Procesamiento Paralelo de la Universidad Complutense de Madrid del profesor Ignacio Martín Llorente.

http://mathworld.wolfram.com/RosenbrockFunction.html

http://www.applied-mathematics.net/optimization/rosenbrock.html

Users guide to the PGA-Pack Parallel Genetic Algorithm Library by David Levine, Mathematics and Computer Science Division.

11. Apéndices

11.1. Código:

A continuación mostramos el código generado para realizar los experimentos:

Cabeceras de las librerías utilizadas. PGAPACK, MPI, y las propias de c para manejo de cadenas, ficheros y tiempos.

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

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
```

Variables y funciones para la correcta ejecución del programa, el control de los parámetros, variables auxiliares, etc …

```
#define MAX_PROCESADORES 8

//variables del pgapack
PGAContext  *ctx;

//variables para contener los parámetros del algoritmo genético
int cruce;//tipo de cruce del algoritmo 1 CROSSOVER_ONEPT  2
CROSSOVER_TWOPT(Default)  3 CROSSOVER_UNIFORM

int fitness;//tipo de fitness 1 FITNESS_RAW(Default)  2 FITNESS_NORMAL  3
FITNESS_RANKING

int reemplazamiento;//tipo de reemplazamiento 1 POPREPL_BEST(Default)  2
POPREPL_RANDOM_NOREP  3 POPREPL_RANDOM_REP

int seleccion;//tipo de selección  1 SELECT_PROPORTIONAL  2 SELECT_SUS  3
SELECT_TOURNAMENT  3 SELECT_PTOURNAMENT

int experimento;//tipo de experimento 1 ONEMAX 2 FTRAP 3 Schwefel 4 Rastrigin
5 FMmodal 6 Rosenbrock

int longIndividuo;//longitud el individuo

int poblacion;//tamaño de la población

int iteraciones;//numero de iteraciones que realizara el algoritmo (-1 el
algoritmo se ejecuta hasta que converge PGA_STOP_NOCHANGE, es decir hasta
                          // que la solución no cambia)

int intercambio;//indica cada cuantas generaciones intercambiamos información

int individuosAIntercambiar;//indica cuantos individuos mandamos en cada
intercambio

double probabilidadCruce; //indica la probabilidad de cruce
```

```
double probabilidadMutacion; //indica la probabilidad de mutación

int semilla; //semilla para inicializar los números aleatorios
```

Variables auxiliares para modelar un array dinámico. Este tipo de datos nos será de gran utilidad para guardar todos los datos relativos a la ejecución de los cuales extraeremos información:

```
//Tipos, variables y funciones para definir un array dinámico
typedef struct
      {
      double *array;  //array almacén
      int ocupacion; //ocupación actual del array
      int capacidad; //capacidad máxima que tiene reservada el array
      }
arrayDinamico;

//funciones para manipular un arrayDinamico

//inicializa un array dinámico, reservando memoria
void inicializarArray(arrayDinamico *array, int _capacidad);

//libera la memoria asignada al array dinámico
void liberarArray(arrayDinamico *array);

//lo añade  en la posición que se indica
int aniadirElemento(arrayDinamico *array,double elemento,int posicion);

//añade al final del array un elemento
void aniadirElementoAlFinal(arrayDinamico *array,double elemento);

void imprimir(arrayDinamico *array); //muestra por pantalla un array por la
salida estándar

void enviarArrayDinamico(arrayDinamico* array,int destino); //envía un array
dinámico utilizando la librería MPI
void recibirArrayDinamico(arrayDinamico* array,int origen); //recibe un array
dinámico utilizando la librería MPI
```

A continuación mostramos el conjunto de variables que utilizaremos a lo largo del programa para guardar y controlar la recopilación de información que necesitamos extraer de cada ejecución.

```
//variables para las estadísticas
int convergencia; //valor en el que ha convergido el procesador

int pasoDelResumenValores; //indica cada cuantas iteraciones se guarda
información sobre los valores de los individuos

int pasoDelResumenHerencia; //indica cada cuantas iteraciones se guarda
información sobre el porcentaje de herencia

int imprimirParametros; //0 no se imprime un resumen de los parámetros, 1 si
se imprime

int repeticiones = 100;

FILE *ficheroMejores; //fichero en el que se guardara la información para
realizar las graficas, guardamos los mejores valores

FILE *ficheroPeores; //fichero en el que se guardara la información para
realizar las graficas, guardamos los peores valores
```

```
FILE *ficheroMedios; //fichero en el que se guardara la información para
realizar las graficas, guardamos los medios valores

FILE *ficheroTiempos; //fichero en el que se guardara la información para
realizar las graficas, guardamos los tiempos de ejecución

FILE *ficheroHerencia; //fichero en el que se guardara la información para
realizar las graficas, guardamos los porcentajes de herencia

char *punteroFicheroHerencia;

clock_t inicio; //variable para tomar en tiempo de inicio del algoritmo
clock_t fin ; //variable para tomar en tiempo de fin del algoritmo

double mediaEvaluacion; //media de la función de evaluación de la población
double mediaFitness; //media de la función de fitness de la población

double mediaEvaluacionFinal[MAX_PROCESADORES]; //media de la variable
mediaEvaluacion de todas las iteraciones

double mediaFitnessFinal[MAX_PROCESADORES]; //media de la variable
mediaFitness de todas las iteraciones

int mediaConverfenciaFinal[MAX_PROCESADORES]; //media de la variable
convergencia de todas las iteraciones

arrayDinamico herenciaProcesadores[MAX_PROCESADORES]; //aquí cada procesador
guardar la información de la herencia

arrayDinamico valorMejoresIndividuos;
arrayDinamico mejoresIndividuos; //aquí se guardaran los mejores individuos en
cada iteración que marque pasoDelResumenValores
arrayDinamico valorPeoresIndividuos;
arrayDinamico peoresIndividuos; //aquí se guardaran los mejores individuos en
cada iteración que marque pasoDelResumenValores
arrayDinamico valorMediosIndividuos;
arrayDinamico mediosIndividuos; //aquí se guardaran los mejores individuos en
cada iteración que marque pasoDelResumenValores
arrayDinamico aportacionesALaMedia;
arrayDinamico tiempos; //guarda los tiempos de ejecución
```

Pasamos ahora a listar las variables que hemos utilizado para controlar el intercambio de información entre los diferentes procesadores

```
//variables para realizar el intercambio de información entre los procesadores
MPI_Status  status; //variable propia de MPI

int seguir; //para el algoritmo genético si alguna población a convergido

int estadoActualProcesador[MAX_PROCESADORES]; //1 el procesador esta activo, 0
el procesador esta inactivo. Estado actual de los procesadores

int estadoActualProcesadorCopia[MAX_PROCESADORES];
//Copia del anterior para luego poder mostrar por pantalla su estado inicial

int intercambiosNormalesDelProcesador[MAX_PROCESADORES];//indica hasta que
iteración realiza intercambios normales el procesador
//que corresponda con el índice. -1 indica que nunca se quedara incomunicada
la población y => 0 indica hasta que iteración hará intercambios normales

int intercambiosNormalesDelProcesadorCopia[MAX_PROCESADORES];//Copia del
anterior para luego poder mostrar por pantalla desde cuando esta detenido

int envio[MAX_PROCESADORES];//contiene la forma en la que se envían la
información
```

```
int recepcion[MAX_PROCESADORES]; //contiene la forma en la que se recibe la
información
```

Funciones auxiliares que hemos considerado necesarias para clarificar el código, en este fragmento se muestra únicamente la cabecera de dichas funciones, pasando mas adelante a mostrar su cuerpo y explicando el funcionamiento de cada una.

```
//funciones auxiliares
int obtenerParametos(char **argv);
void descripcion();
void algoritmoGenetico(int argc, char **argv, int id); //id = identificador de
proceso
void intercambiarInformacion(); //intercambio de información durante la
ejecución
void intercambioMejores(); //mandan los mejores al final, al proceso 0
void realizarEstadisticas();
void construirArraysDeIntercambios();
void inicializaciones();
void configuracionParametros();
void comprobarConvergenciaPoblaciones();
void actualizarEstadoProcesadores();

void guardarValorMejorIndividuo();
void guardarValorMedio();
void guardarValorPeorIndividuo();
void guardarHerencia();

void procesarMejoresValores();
void procesarPeoresValores();
void procesarMediosValores();
void procesarTiempos();
void procesarHerencia();

void escribirFicheroGraficaMejores();
void escribirFicheroGraficaPeores();
void escribirFicheroGraficaTiempos();
void procesarArchivoHerencia();

void escribirParametrosEnFichero(FILE*);

void calcularMediaFitnessYEvaluacion();
void calcularHerencia(int *herencia);

void conseguirSemilla();
```

Funciones que estudiaremos OneMax, FTrap, Schwefel, Rastrigin, FModal, Rosenbrock. También mostramos aquí la función evaluar utilizada para clarificar el código que se mostrara mas detalladamente posteriormente. También mostramos en esta sección una serie de constantes utilizadas para la implementación de las funciones a estudiar.

```
double OneMax(PGAContext *, int, int);
double FTrap(PGAContext *, int, int);
double Schwefel(PGAContext *, int, int);
double Rastrigin(PGAContext *, int, int);
double FMmodal(PGAContext *, int, int);
double Rosenbrock(PGAContext *, int, int);
void evaluar(PGAContext *ctx,int population);

#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 MIN_rosenbrock  0
#define MAX_rosenbrock  2

#define N_VAR_ROS 3

#define PI 3.1416
#define A 400
#define B 512
#define Z 400
#define N_VAR 10
```

En la función principal, inicializamos el entorno MPI, llamamos al algoritmo genético que utilizaremos y posteriormente finalizamos el MPI.

```
int main( int argc, char **argv )
      {
      //inicializo los intercambios
      //realizo las preguntas por pantalla
      if(obtenerParametros(argv)==0)
            {
            return 0;
            }

      //ejecuto 4 algoritmos
      // Número de procesos  numProcesadores en pgapack.h
      // Mi dirección: 0<=yo<=(nproc-1)  que esta en
      //idProceso en pgapack.h
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &numProcesadores);
      MPI_Comm_rank(MPI_COMM_WORLD, &IdProceso);

      MPI_Barrier(MPI_COMM_WORLD);
      //realizo inicializaciones propias del programa
      inicializaciones();
      //muestro la descripción del experimento
      if (IdProceso == 0)
            descripcion();

      MPI_Barrier(MPI_COMM_WORLD);

      //ejecuto el algoritmo genético
      algoritmoGenetico(argc, argv, IdProceso);
      MPI_Barrier(MPI_COMM_WORLD);

      MPI_Finalize();     //finalizo el MPI

      return (0);
      }
```

Con esta función realizamos las operaciones necesarias para llevar a cabo el algoritmo genético con los parámetros que le hayamos asignado (función configuracionParametros() es la encargada de fijarlos).

```
void algoritmoGenetico(int argc, char **argv,int id)
	{
	//creo el contexto para que se ejecute el algoritmo
	ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, longIndividuo,
PGA_MAXIMIZE);
	configuracionParametros(); //configuro los parámetros de la ejecución
	PGASetUp(ctx);

	//tomo el valor del tiempo de inicio
	inicio = clock();

	evaluar(ctx,PGA_OLDPOP);
	//inicio del algoritmo
	PGAFitness(ctx, PGA_OLDPOP);
	while (!PGACheckStoppingConditions(ctx) && seguir==1)
		{
		PGASelect(ctx, PGA_OLDPOP);
		PGARunMutationAndCrossover(ctx,PGA_OLDPOP, PGA_NEWPOP);
		evaluar(ctx,PGA_NEWPOP);

		PGAFitness(ctx, PGA_NEWPOP);

		PGAUpdateGeneration(ctx,NULL);
		//intercambio información
		if (numProcesadores != 1)
			intercambiarInformacion();
		//guardo el valor de la evaluación de los mejores individuos
		guardarValorMejorIndividuo();
		guardarValorMedio();
		guardarValorPeorIndividuo();
		guardarHerencia();

		if (numProcesadores != 1)
			comprobarConvergenciaPoblaciones();
		}

	//tomo el valor de fin
	fin = clock();

	//guardo la iteración a la que se ha parado
	convergencia = PGAGetGAIterValue(ctx);
	//terminada la ejecución
	if (numProcesadores != 1)
		intercambioMejores();

	procesarMejoresValores();
	procesarPeoresValores();
	procesarMediosValores();
	procesarTiempos();
	procesarHerencia();
	realizarEstadisticas();

	//liberamos los arrays con los mejores individuos
	liberarArray(&valorMejoresIndividuos);
	//liberamos los arrays con los peores individuos
	liberarArray(&valorPeoresIndividuos);
	//liberamos los arrays con los mejores individuos
	liberarArray(&valorMediosIndividuos);
	//liberamos los arrays herenciaProcesadores
	int i;
	for (i=0;i<numProcesadores;i++)
		liberarArray(&(herenciaProcesadores[i]));
```

```
      if (IdProceso == 0)
            {
            liberarArray(&mejoresIndividuos);
            liberarArray(&peoresIndividuos);
            liberarArray(&mediosIndividuos);
            liberarArray(&aportacionesALaMedia);
            }

      PGADestroy(ctx);
      }
```

Función auxiliar que devuelve un valor decimal asociado a una determinada parte del genotipo del cromosoma que se indica con los parámetros de la función

```
double getX(PGAContext *ctx, int p, int pop,int ini,int len_var,int min, int
max)
      {
      double aux;
      aux= min + (max-
min)*valorDecimal(ctx,p,pop,ini,len_var)/(pow(2,len_var+1)-1);
      return aux;
      }
```

Convierte un número binario en decimal

```
double valorDecimal(PGAContext * ctx, int p, int pop,int ini,int len_var)
      {
      int i;
      double d;
      double pot;
      d=0;
      pot =1;
      for(i=ini;i<=ini+len_var;i++){
            if(PGAGetBinaryAllele(ctx,p,pop,i))
                  d+=pot;
            pot *= 2;
      }
      return d;
      }
```

A continuación listamos el código de las funciones que hemos analizado en este estudio.

```
/*****************************************************************
******************************************************************
              user defined evaluation function    oneMax
   ctx - contex variable
   p   - chromosome index in population
  pop - which population to refer to
******************************************************************
******************************************************************/

double OneMax(PGAContext *ctx, int p, int pop)
      {
      int i, nbits, stringlen;

      stringlen = PGAGetStringLength(ctx);

       nbits = 0;
       for ( i=0; i<stringlen; i++ )
            if ( PGAGetBinaryAllele(ctx, p, pop, i) )
                  nbits++;
```

```
	return((double) nbits);
	}
```

```
/*****************************************************************
******************************************************************
		user defined evaluation function    FTrap
   ctx - contex variable
  p   - chromosome index in population
  pop - which population to refer to
******************************************************************
*****************************************************************/

double FTrap(PGAContext *ctx, int p, int pop)
	{
	double x,u;
	u=OneMax(ctx,p,pop);
	if(u<=Z)
		x=A*(Z-u)/Z;
	else
		x=B*(u-Z)/(B-Z);

	return ((double)x );
	}
```

```
/*****************************************************************
******************************************************************
		user defined evaluation function   Schwefel
   ctx - contex variable
   p   - chromosome index in population
   pop - which population to refer to
******************************************************************
*****************************************************************/

double Schwefel(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;
	}
```

```
/*****************************************************************
/*****************************************************************
		user defined evaluation function    Rastrigin
   ctx - contex variable
   p   - chromosome index in population
   pop - which population to refer to
******************************************************************
*****************************************************************/
double Rastrigin(PGAContext *ctx, int p, int pop) {
	double x;
	int i,ini;
	int len_var;
	double sol;
	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_rastrigin,MAX_rastrigin);
                sol+= ((double)x*x-10*cos(2*PI*x)+10);
                ini+=len_var;
        }
        return sol;
}
```

```
/*****************************************************************
******************************************************************
                user defined evaluation function fmodal
   ctx - contex variable
   p   - chromosome index in population
:20
   pop - which population to refer to
 ******************************************************************
*****************************************************************/
double FModal(PGAContext *ctx, int p, int pop) {
        double x;
        int i,ini;
        int len_var;
        double sol;
        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_fmodal,MAX_fmodal);
                sol+=((double)-x*sin(10*x*PI)+1);
                ini+=len_var;
        }
        return sol;
}
```

```
/*****************************************************************
******************************************************************
                user defined evaluation function fmodal Rosenbrock
   ctx - contex variable
   p   - chromosome index in population
   pop - which population to refer to
******************************************************************
*****************************************************************/
double Rosenbrock(PGAContext *ctx, int p, int pop)
        {
        double x1;
        double x2;
        int i,ini;
        int len_var;
        double sol,aux1,aux2;
        sol=0;
        ini=0;
        len_var=PGAGetStringLength(ctx)/N_VAR_ROS;

        for(i=0;i<N_VAR_ROS-1;i++)
                {
                x1=getX(ctx,p,pop,ini,len_var-1,MIN_rosenbrock,MAX_rosenbrock);
                x2=getX(ctx,p,pop,len_var,(ini+(2*len_var)-
1),MIN_rosenbrock,MAX_rosenbrock);
                aux1=(double)(((x2-(x1*x1))*(x2-(x1*x1)))*100);
                aux2=((double)((1-x1)*(1-x1)));
                sol+=((double)(aux1+aux2));
                ini+=len_var;
                }
        return sol;
        }
```

Función de inicialización de todas las variables utilizadas.

```
void inicializaciones()
	{
	seguir = 1;

	mediaFitness = 0;
	mediaEvaluacion = 0;

	//meto 0 en los arrays a partir de numProcesadores, esa información no
	//es valida y puede traer problemas a las funciones
	int i = numProcesadores;
	for (i;i<MAX_PROCESADORES;i++)
		{
		intercambiosNormalesDelProcesador[i]=0;
		estadoActualProcesador[i]=0;
		}

	//copio intercambiosNormalesDelProcesador
	for (i=0;i<MAX_PROCESADORES;i++)
		{

	intercambiosNormalesDelProcesadorCopia[i]=intercambiosNormalesDelProcesa
dor[i];
		estadoActualProcesadorCopia[i]=estadoActualProcesador[i];
		}

	construirArraysDeIntercambios();

	//inicializo el array de los mejores
	inicializarArray(&valorMejoresIndividuos,100);
	inicializarArray(&valorPeoresIndividuos,100);
	inicializarArray(&valorMediosIndividuos,100);
	//inicializo herenciaProcesadores
	for (i=0;i<numProcesadores;i++)
		inicializarArray(&(herenciaProcesadores[i]),100);

	if (IdProceso == 0)
		{
		inicializarArray(&mejoresIndividuos,100);
		inicializarArray(&peoresIndividuos,100);
		inicializarArray(&mediosIndividuos,100);
		inicializarArray(&aportacionesALaMedia,100);
		}
	}
```

Función que lee los parámetros que se le pasan como parámetros en la llamada al programa y los asigna a sus correspondientes variables.

```
int obtenerParametros(char **argv)
	{
	//tipo de cruce ( 1 CROSSOVER_ONEPT  2 CROSSOVER_TWOPT(Default)  3
CROSSOVER_UNIFORM )
	cruce = atoi(argv[1]);
	if(cruce !=1 && cruce !=2 && cruce !=3)
		{
		printf("El tipo de cruce sólo puede tomar los valores 1, 2 ó 3.
Revise los parámetros de entrada, por favor.");
		return 0;
		}

	probabilidadCruce = atof(argv[2]);
	if(probabilidadCruce < 0)
		{
		printf("La probabilidad de cruce deber ser mayor que 0");
```

```
		return 0;
		}

	probabilidadMutacion = atof(argv[3]);
	if(probabilidadMutacion < 0)
		{
		printf("La probabilidad de mutación deber ser mayor que 0");
		return 0;
		}

	//tipo de fitness ( 1 FITNESS_RAW(Default)  2 FITNESS_NORMAL  3
FITNESS_RANKING)
	fitness = atoi(argv[4]);
	if(fitness !=1 && fitness !=2 && fitness !=3)
		{
		printf("El tipo de fitness sólo puede tomar los valores 1, 2 ó 3.
Revise los parámetros de entrada, por favor.");
		return 0;
		}

	//tipo de reemplazamiento ( 1 POPREPL_BEST(Default)  2
POPREPL_RANDOM_NOREP  3 POPREPL_RANDOM_REP)
	reemplazamiento = atoi(argv[5]);
	if(reemplazamiento !=1 && reemplazamiento !=2 && reemplazamiento !=3)
		{
		printf("El tipo de reemplazamiento sólo puede tomar los valores
1, 2 ó 3. Revise los parámetros de entrada, por favor.");
		return 0;
		}

	//tipo de seleccion ( 1 SELECT_PROPORTIONAL  2 SELECT_SUS  3
SELECT_TOURNAMENT(Default)  4 SELECT_PTOURNAMENT)
	seleccion = atoi(argv[6]);
	if(seleccion !=1 && seleccion !=2 && seleccion !=3 && seleccion !=4)
		{
		printf("El tipo de selección sólo puede tomar los valores 1, 2, 3
ó 4. Revise los parámetros de entrada, por favor.");
		return 0;
		}

	//experimento ( 1 ONEMAX  2 FTRAP 3 SCHWEFEL 4 RASTRIGIN 5 FMODAL)
	experimento = atoi(argv[7]);
	if(experimento !=1 && experimento !=2 && experimento !=3 && experimento
!=4 && experimento !=5 && experimento !=6)
		{
		printf("El tipo de experimento sólo puede tomar los valores 1, 2,
3, 4 ,6  ó  6. Revise los parámetros de entrada, por favor.");
		return 0;
		}

	//longitud del individuo
	longIndividuo = atoi(argv[8]);

	//población
	poblacion = atoi(argv[9]);

	//numero de iteraciones
	iteraciones = atoi(argv[10]);

	//intercambio
	intercambio = atoi(argv[11]);

	//numero de individuos a intercambiar
	individuosAIntercambiar = atoi(argv[12]);

	estadoActualProcesador[0]=atoi(argv[13]);
	estadoActualProcesador[1]=atoi(argv[14]);
	estadoActualProcesador[2]=atoi(argv[15]);
```

```
        estadoActualProcesador[3]=atoi(argv[16]);
        estadoActualProcesador[4]=atoi(argv[17]);
        estadoActualProcesador[5]=atoi(argv[18]);
        estadoActualProcesador[6]=atoi(argv[19]);
        estadoActualProcesador[7]=atoi(argv[20]);

        intercambiosNormalesDelProcesador[0]=atoi(argv[21]);
        intercambiosNormalesDelProcesador[1]=atoi(argv[22]);
        intercambiosNormalesDelProcesador[2]=atoi(argv[23]);
        intercambiosNormalesDelProcesador[3]=atoi(argv[24]);
        intercambiosNormalesDelProcesador[4]=atoi(argv[25]);
        intercambiosNormalesDelProcesador[5]=atoi(argv[26]);
        intercambiosNormalesDelProcesador[6]=atoi(argv[27]);
        intercambiosNormalesDelProcesador[7]=atoi(argv[28]);

        //abro el fichero en el que escribiremos los valores para la grafica
        ficheroMejores = fopen(argv[29],"a");
        ficheroMedios = fopen(argv[30],"a");
        ficheroPeores = fopen(argv[31],"a");
        ficheroTiempos = fopen(argv[32],"a");

        punteroFicheroHerencia = argv[33];
        ficheroHerencia = fopen(punteroFicheroHerencia,"a");

        pasoDelResumenHerencia = atoi(argv[34]);
        pasoDelResumenValores = atoi(argv[35]);

        imprimirParametros = atoi(argv[36]); //0 no se imprimen los parámetros
                                  //1 se imprimen los parámetros

        return 1;
        }
```

Con esta función realizamos la inicialización del algoritmo genético basándonos en las funcionalidades ofertadas por la librería PGAPACK.

```
void configuracionParametros()
        {
        conseguirSemilla();

        PGASetRandomSeed(ctx,semilla);

        //"Tipo de Cruce? ( 1 CROSSOVER_ONEPT  2 CROSSOVER_TWOPT
        //           3 CROSSOVER_UNIFORM) "
        PGASetCrossoverType(ctx,cruce);

        //configuro la probabilidad de cruce
        PGASetCrossoverProb(ctx,probabilidadCruce);

        //configuro la probabilidad de cruce
        PGASetMutationProb(ctx,probabilidadMutacion);

        //"Tipo de fitness? ( 1 FITNESS_RAW  2 FITNESS_NORMAL
        //3 FITNESS_RANKING) "
        PGASetFitnessType(ctx,fitness);

        //"Tipo de reemplazamiento? ( 1 POPREPL_BEST
        //2 POPREPL_RANDOM_NOREP  3 POPREPL_RANDOM_REP) "
        PGASetPopReplaceType(ctx,reemplazamiento);

        //"Tipo de selección? ( 1 SELECT_PROPORTIONAL
        //2 SELECT_SUS  3 SELECT_TOURNAMENT  3 SELECT_PTOURNAMENT) "
        PGASetSelectType(ctx,seleccion);

        //"tamaño de la población?
```

```
	PGASetPopSize(ctx,poblacion);

	//numero de iteraciones
	if (iteraciones == -1)
		{ //comprobamos la convergencia
		PGASetStoppingRuleType(ctx,PGA_STOP_NOCHANGE);
		PGASetMaxGAIterValue(ctx,100000);
		}
	else
		{ //ejecutamos las iteraciones que nos pasan como parámetro
		PGASetStoppingRuleType(ctx,PGA_STOP_MAXITER);
		PGASetMaxGAIterValue(ctx,iteraciones);
		}
	}
```

Cuando un procesador ha convergido debemos incomunicarlo, es decir, seguirá ejecutándose pero dejara de intercambiar información con los demás. Simulara de este modo que se ha caído. Por lo tanto debemos comprobar su convergencia. Lo haremos con la siguiente función:

```
/*****************************************************************
******************************************************************
 Compruebo si alguna población ha convergido para detener todos los algoritmos
 y dar por finalizada todas las ejecuciones. La simulación se detendrá
 cuando algún procesador haya convergido, siempre y cuando este procesador (el
cual
está ejecutando este mismo código)
 no este marcado como incomunicado
******************************************************************
*****************************************************************/

void comprobarConvergenciaPoblaciones()
	{
	int parar;
	//si es -1 controlamos la parada por el numero de iteraciones indicadas
	if ((iteraciones == -1) && (estadoActualProcesador[IdProceso]==1))
		{
		//controlo que ningún otro proceso haya convergido ya, si lo ha
hecho he de parar
		//miro si debo parar
		parar = PGACheckStoppingConditions(ctx);
		//envío parar a todos los procesadores que no estén
incomunicados, sin contarme a mi   mismo
		int i=0;
		while (i<numProcesadores)
			{
			if ((estadoActualProcesador[i]==1) && (i!=IdProceso))
				//el procesador i no esta incomunicado le mando
parar
				MPI_Send(&parar, 1, MPI_INT, i, 222,
MPI_COMM_WORLD);
			i = i+1;
			}

		//recibo el parar de todos los procesadores que me lo hayan
enviado. Esto es, de todos los no estén incomunicados
		//y exceptuándome a mi mismo
		i = 0;
		while (i<numProcesadores)
			{
			if ((estadoActualProcesador[i]==1) && (i!=IdProceso))
				{
				//el procesador i no esta incomunicado le mando
parar
				MPI_Recv(&parar, 1, MPI_INT, i, 222,
MPI_COMM_WORLD,&status);
```

```
                        //si alguien me manda parar paro mi ejecución
                        if (parar == 1)
                              seguir = 0;
                        }
                  i = i+1;
                  }
            }
      }
```

Con esta función realizamos el intercambio de información entre las diferentes poblaciones (procesadores), con una frecuencia marcada por la variable "intercambio".

```
/*********************************************************************
*********************************************************************
     Intercambia los mejores individuos durante la ejecución al finalizar
        cada generación
*********************************************************************
*********************************************************************/

void intercambiarInformacion()
      {
      int myid;
      int n_pops;
      int tag;

      tag=20; //etiqueta de los envíos
      if((PGAGetGAIterValue(ctx)%intercambio)==0)
            {
            //actualizo el estado de los procesadores
            actualizarEstadoProcesadores();

            if (estadoActualProcesador[IdProceso]==1)
                  {
                  //enviamos los  mejores individuos
(individuosAIntercambiar)
                  int i=0,j=0;
                  PGASortPop(ctx, PGA_OLDPOP);
                  for ( i=0; i < individuosAIntercambiar; i++)
                        {
                        j = PGAGetSortedPopIndex(ctx, i);
                        //los envío al siguiente
                        if (aQuienEnvio(IdProceso) != -1)

      PGASendIndividual(ctx,j,PGA_OLDPOP,aQuienEnvio(IdProceso),tag,MPI_COMM_W
ORLD);
                        }

                  //recibo todos los individuos y sustituyo los peores
                  for ( i=0; i < individuosAIntercambiar; i++)
                        {
                        j = PGAGetSortedPopIndex(ctx, (poblacion-1)-i);
//consigo los peores  para sustituirlos
                        //los recibo
                        if (deQuienRecibo(IdProceso) != -1)

      PGAReceiveIndividual(ctx,j,PGA_OLDPOP,deQuienRecibo(IdProceso),tag,MPI_C
OMM_WORLD,&status);
                        }
                  //sincronizamos todos los procesos
                  }
            }
      }
```

Realiza la actualizamos de la lista de intercambios de los procesadores, mirando cuales deben ser incomunicados y actualizando la lista para que se sigan comunicando el resto de procesadores.

```
/*********************************************************************
*********************************************************************
       Comprueba si hay que incomunicar o no a una población mirando el array
       intercambiosNormalesDelProcesador. Cuando
intercambioNormalesDelProcesador
       es menor o igual que la iteración actual se incomunica o se vuelve a
hacer accesible
       esa población, es decir, si el procesador tiene un estado 0 =
incomunicado
       pasará a un estado 1=comunicado, y viceversa, si el procesador tiene un
estado
       1 = comunicado pasara a un estado 0=incomunicado

       Si intercambiosNormalesDelProcesador[i] = -1 este procesador no cambiara
       su estado durante toda la ejecución

       Si  intercambiosNormalesDelProcesador[i] = 20  se realizan intercambios
       cada diez iteraciones, este procesador realizara intercambios en la
iteración
       0 y en la 10. Es decir a partir de intercambiosNormalesDelProcesador[i]
       cambiará el estado
*********************************************************************
*********************************************************************/

void actualizarEstadoProcesadores()
       {
       int i;
       for (i=0;i<numProcesadores;i++)
              {
              if
((PGAGetGAIterValue(ctx)>=intercambiosNormalesDelProcesador[i])
              && (intercambiosNormalesDelProcesador[i] != -1) )
                     {
                     //invierto el estado de comunicado a incomunicado o
viceversa
                     estadoActualProcesador[i] =
(estadoActualProcesador[i]+1)%2;
                     //una vez cambiado el estado no se vuelve a hacer
                     intercambiosNormalesDelProcesador[i] = -1;
                     }
              }
       //reconstruyo el array de intercambios para tener en cuenta los cambios
realizados
       construirArraysDeIntercambios();
       }
```

Al finalizar el algoritmo se intercambia la información relativa a los mejores individuos, y las estadísticas de la ejecución en cada procesador. Además mostramos por pantalla la información relativa a la ejecución.

```
/*****************************************************************
******************************************************************
      Al finalizar el algoritmo intercambiamos los mejores
      El proceso cero hace de master y recibe todos los individuos
      para imprimirlos por pantalla
******************************************************************
*****************************************************************/

void intercambioMejores()
      {
      MPI_Datatype individualtype;
      //enviamos el mejor individuo al cero
      if (IdProceso != 0)
            {
            int j;
            PGASortPop(ctx, PGA_OLDPOP);
            j = PGAGetSortedPopIndex(ctx, 0); //envío el mejor
            PGASendIndividual(ctx,j,PGA_OLDPOP,0,0,MPI_COMM_WORLD);
            }
      else if (IdProceso == 0)
            { //IdProceso==0 recibe los individuos
            //meto en la posición 0 del array el mejor individuo
            printf("\n");
            printf("****************************************");
            printf("\n");
            printf("********** MEJORES INDIVIDUOS *********");
            printf("\n");
            printf("****************************************");
            printf("\n");printf("\n");

            int j;
            PGASortPop(ctx, PGA_OLDPOP);
            j = PGAGetSortedPopIndex(ctx, 0);
            //copio el de la población 0 a la posición 0
            PGACopyIndividual(ctx, j, PGA_OLDPOP, 0, PGA_NEWPOP);
            //recibo los procesadores
            //empiezo a guardarlos en la posición 1 para conservar el mejor
del procesador 0
            int k;
            for (k=1;k<numProcesadores;k++)
                  {

      PGAReceiveIndividual(ctx,k,PGA_NEWPOP,k,0,MPI_COMM_WORLD,&status
);
                  }
            //imprimo los cuatro
            int i;
            for (i=0;i<numProcesadores;i++)
                  {
                  printf("Mejor individuo del procesador
");printf("%i",i);printf("\n");
                  PGAPrintIndividual( ctx, stdout, i, PGA_NEWPOP );
                  printf("\n\n");
                  }
            }
      }
```

Realiza las estadísticas de la ejecución a partir de los datos recibidos.

```
/*********************************************************************
*********************************************************************
                Realiza las estadísticas

             Manda todos los datos al proceso 0

*********************************************************************
*********************************************************************/

void realizarEstadisticas()
       {
       //variables auxiliares
       int i,j;
       int herencia[MAX_PROCESADORES];
       double fitnessProc[MAX_PROCESADORES];
       double evaluacionProc[MAX_PROCESADORES];
       int seguirProc[MAX_PROCESADORES];
       int convergenciaProc[MAX_PROCESADORES];

       calcularMediaFitnessYEvaluacion();
       calcularHerencia(&herencia);

       if (IdProceso == 0)
              {//recibe las estadísticas e imprime las propias
              printf("****************************************");
              printf("\n");
              printf("***ESTADISTICAS DE LOS PROCESADORES***");
              printf("\n");
              printf("****************************************");
              printf("\n");printf("\n");

              printf("ESTADISTICAS DEL PROCESADOR
");printf("%i",IdProceso);printf("\n");
              for (j=0;j<numProcesadores;j++)
                     {
                     //herencia
                     printf(" -- Herencia del procesador
");printf("%i",j);printf(": ");
                     printf("%i",herencia[j]);printf(" de
");printf("%i",PGAGetPopSize(ctx));
                     printf("\n");
                     }
              //imprimo el estado del procesador
              if (estadoActualProcesador[IdProceso] == 0) //alguien me ha
parado
                     printf("   ESTADO: Incomunicado desde la iteración
%i\n",intercambiosNormalesDelProcesadorCopia[IdProceso]);
              else
                     printf("   ESTADO: Comunicado\n");
              //imprimo la convergencia
              if (seguir == 0) //alguien me ha parado
                     printf("   CONVERGENCIA: Parado en la iteración %i
\n",convergencia);
              else
                     printf("   CONVERGENCIA: Convergencia en la iteración %i
\n",convergencia);
              //imprimo la media del fitness y de la función de evaluación
              printf("   EVALUACION (media de la población): %e
\n",mediaEvaluacion);
              printf("   FITNESS (media de la población): %e
\n\n",mediaFitness);

              //recibo información del resto de procesadores
              int j;
              for (j = 1;j<numProcesadores;j++)
                     {
```

```
                    MPI_Recv (&herencia, MAX_PROCESADORES, MPI_INT, j, 1,
MPI_COMM_WORLD, &status);
                    MPI_Recv (&seguirProc[j], 1, MPI_INT, j, 1, MPI_COMM_WORLD,
&status);
                    MPI_Recv (&convergenciaProc[j], 1, MPI_INT, j, 1,
MPI_COMM_WORLD, &status);
                    MPI_Recv (&fitnessProc[j], 1, MPI_DOUBLE, j, 1,
MPI_COMM_WORLD, &status);
                    MPI_Recv (&evaluacionProc[j], 1, MPI_DOUBLE, j, 1,
MPI_COMM_WORLD, &status);

                    //imprimo lo enviado
                    int k = 0;
                    printf("ESTADISTICAS DEL PROCESADOR
");printf("%i",j);printf("\n");
                    for (k=0;k<numProcesadores;k++)
                          {
                          //herencia
                          printf("   Herencia del procesador
");printf("%i",k);printf(": ");
                          printf("%i",herencia[k]);printf(" de
");printf("%i",PGAGetPopSize(ctx));
                          printf("\n");
                          }
                    //imprimo el estado del procesador
                    if (estadoActualProcesador[j] == 0) //alguien me ha parado
                          printf("   ESTADO: Incomunicado desde la iteración
%i\n",intercambiosNormalesDelProcesadorCopia[j]);
                    else
                          printf("   ESTADO: Comunicado\n");
                    //imprimo la convergencia
                    if (seguirProc[j] == 0) //alguien me ha parado
                          printf("   CONVERGENCIA: Parado en la iteración %i
\n",convergenciaProc[j]);
                    else
                          printf("   CONVERGENCIA: Convergencia en la
iteración %i \n",convergenciaProc[j]);
                    //imprimo la media del fitness y de la función de
evaluación
                    printf("   EVALUACION (media de la población): %e
\n",evaluacionProc[j]);
                    printf("   FITNESS (media de la población): %e
\n\n",fitnessProc[j]);
                    printf("\n");
                    }

              //guardo también la información del procesador 0 en la posición
correspondiente
              evaluacionProc[0]=mediaEvaluacion;
              fitnessProc[0]=mediaFitness;
              convergenciaProc[0]=convergencia;
              }
       else
              {//enviad las estadísticas
              //envío la herencia
              MPI_Send(&herencia, MAX_PROCESADORES, MPI_INT, 0, 1,
MPI_COMM_WORLD);
              //envío los datos de los orígenes
              MPI_Send(&seguir, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
              MPI_Send(&convergencia, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
              MPI_Send(&mediaFitness, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
              MPI_Send(&mediaEvaluacion, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);

              }
       }
```

Funciones para realizar el intercambio de los mejores individuos.

```
/*********************************************************************
*********************************************************************
      Devuelve de que procesador te corresponde recibir
      Si devuelve -1 ni recibes ni envías, el procesador se ha
      caído
      El parámetro que se le recibe es el procesador por el que
      preguntas
*********************************************************************
*********************************************************************/
int deQuienRecibo(int procesador)
      {
      return recepcion[procesador];
      }

/*********************************************************************
*********************************************************************
Devuelve a que procesador te corresponde enviar
Si devuelve -1 ni recibes ni envías, el procesador se ha Cairo
El parámetro que se le recibe es el procesador por el que preguntas
*********************************************************************
*********************************************************************/
int aQuienEnvio(int procesador)
      {
      return envio[procesador];
      }

/*********************************************************************
*********************************************************************
      envios[]  dice a que procesador envió
      recepcion[] dice de que procesador recibo
      estadoActualProcesador[] indica el estado actual del procesador
*********************************************************************
*********************************************************************/
void construirArraysDeIntercambios()
      {
      //preparo el array de los envíos
      int i = 0;
      while (i<numProcesadores)
            {
            //busco a quien debo enviar
            if (estadoActualProcesador[i]==1)
                  {
                  envio[i]=buscarSiguienteProcesadorActivo(i);
                  //pongo de quien debo recibir
                  recepcion[envio[i]]=i;
                  }
            else if (estadoActualProcesador[i]==0)
                  {
                  envio[i]=-1;
                  recepcion[i]=-1;
                  }
            i++;
            }
      }
```

```
/*****************************************************************
******************************************************************
                    Busca el siguiente procesador activo

                    empezando desde inicio si llega al final del array
                    vuelve a empezar desde el principio

******************************************************************
******************************************************************/

int buscarSiguienteProcesadorActivo(int inicio)
      {
      int resultado;
      //devolveremos esto
      int i=(inicio+1)%numProcesadores;
      int sigo = 1; //0 false 1 true
      while (sigo == 1)
            {
            if (estadoActualProcesador[i]==1)
                  {//hemos encontrado un procesador activo
                  sigo=0;
                  resultado = i;
                  }
            i = (i + 1)%numProcesadores;
            }

      return resultado;
```

Escribe por pantalla la descripción del experimento.

```
void descripcion()
      {
      printf("**************************************");
      printf("\n");
      printf("*******DESCRIPCION DEL EXPRIMENTO******");
      printf("\n");
      printf("**************************************");
      printf("\n");printf("\n");
      //cruce
      if (cruce==1)
            printf("CRUCE: CROSSOVER_ONEPT");
      else if (cruce==2)
            printf("CRUCE: CROSSOVER_TWOPT");
      else if (cruce==3)
            printf("CRUCE: CROSSOVER_UNIFORM");
      printf("\n");

      //probabilidad de cruce
      printf("PROBABILIDAD DE CRUCE: ");printf("%f",probabilidadCruce);
      printf("\n");

      //probabilidad de mutación
      printf("PROBABILIDAD DE MUTACIÓN: ");printf("%f",probabilidadMutacion);
      printf("\n");

      //fitness
      if (fitness==1)
            printf("FITNESS: FITNESS_RAW");
      else if (fitness==2)
            printf("FITNESS: FITNESS_NORMAL");
      else if (fitness==3)
            printf("FITNESS: FITNESS_RANKING");
      printf("\n");

      //reemplazamiento
      if (reemplazamiento==1)
```

```
            printf("REEMPLAZAMIENTO: POPREPL_BEST");
      else if (reemplazamiento==2)
            printf("REEMPLAZAMIENTO: POPREPL_RANDOM_NOREP");
      else if (reemplazamiento==3)
            printf("REEMPLAZAMIENTO: POPREPL_RANDOM_REP");
      printf("\n");

      //selección
      if (seleccion==1)
            printf("SELECCION: SELECT_PROPORTIONAL");
      else if (seleccion==2)
            printf("SELECCION: SELECT_SUS");
      else if (seleccion==3)
            printf("SELECCION: SELECT_TOURNAMENT");
      printf("\n");

      //experimento
      switch (experimento)
      {
            case 1:
            {printf("EXPERIENTO: ONEMAX"); break;}
            case 2:
            {printf("EXPERIENTO: FTRAP"); break;}
            case 3:
            {printf("EXPERIENTO: SCHWEFEL"); break;}
            case 4:
            {printf("EXPERIENTO: RASTRIGIN"); break;}
            case 5:
            {printf("EXPERIENTO: FMODAL"); break;}
            case 6:
            {printf("EXPERIENTO: ROSENBROCK"); break;}
      }
      printf("\n");

      //longitud del individuo
      printf("LONGITUD DEL INDIVIDUO: ");printf("%i",longIndividuo);
      printf("\n");

      //oblación
      printf("TAMAÑO DE LA POBLACION: ");printf("%i",poblacion);
      printf("\n");

      //NUMERO DE ITERACION DEL EXPERIMENTO
      if (iteraciones==-1)
            printf("ITERACIONES: HASTA LA CONVERGENCIA");
      else
            {
            printf("ITERACIONES: ");
            printf("%i",iteraciones);
            }
      printf("\n");

      //FRECUENCIA DE INTERCAMBIO
      printf("FRECUENCIA DE INTERCAMBIO: cada
");printf("%i",intercambio);printf(" generaciones");
      printf("\n");

      //NUMERO DE INDIVIDUOS INTERCAMBIADOS
      printf("NUMERO DE INDIVIDUOS INTERCAMBIADOS:
");printf("%i",individuosAIntercambiar);
      printf("\n");

      //NUMERO PROCESADORES
      printf("NUMERO DE POBLACIONES: ");printf("%i",numProcesadores);
      printf("\n");

      //ESTADO INICIAL DE LOS PROCESADORES
      printf("ESTADO INICIAL DE LOS PROCESADORES: ");
```

```
	int i;
	for (i=0;i<numProcesadores;i++)
		if (estadoActualProcesadorCopia[i]==1)
			printf("%i-ACTIVO ",i);
		else
			printf("%i-INACTIVO ",i);
	printf("\n");

	//NUMERO DE ITERACIONES EN ESTADO INICIAL
	printf("NUMERO DE ITERACIONES EN ESTADO INCIAL: ");
	for (i=0;i<numProcesadores;i++)
		if (intercambiosNormalesDelProcesadorCopia[i] == -1)
			printf("%i-Siempre ",i);
		else
			printf("%i-%i
",i,intercambiosNormalesDelProcesadorCopia[i]);
	printf("\n");

	printf("*************************************");printf("\n");
	printf("*************************************");printf("\n");
	}
```

Realiza la evaluación con la función que corresponda en cada momento.

```
/*********************************************************************
*********************************************************************
evalúa la población según el experimento que estemos ejecutando
1 Onemax
2 Ftrap
3  Schwefel
4   Rastrigin
5   FModal
6  Rosenbrock
*********************************************************************
*********************************************************************/
void evaluar(PGAContext *ctx,int population)
	{
		switch (experimento)
			{
			case 1:
				{
				PGAEvaluate(ctx, population,OneMax,NULL);//One max
				break;
				}
			case 2:
				{
				PGAEvaluate(ctx, population,FTrap,NULL);//FTrap
				break;
				}
			case 3:
				{
				PGAEvaluate(ctx,
population,Schwefel,NULL);//Schwefel
				break;
				}
			case 4:
				{
				PGAEvaluate(ctx,
population,Rastrigin,NULL);//Rastrigin
				break;
				}
			case 5:
				{
				PGAEvaluate(ctx, population,FModal,NULL);//FModal
				break;
				}
			case 6:
```

```
                        {
                        PGAEvaluate(ctx,
population,Rosenbrock,NULL);//Rosenbrock
                        break;
                        }
                }
        }
```

A continuación se muestran las funciones utilizadas para guardar los valores mejor, peor, medio, de tiempos y de herencia. La frecuencia con que se guardan estos valores viene delimitada por la variable pasoDelResumenValores.

```
/*********************************************************************
*********************************************************************
 guarda el mejor valor del individuo en cada iteración que marque la variables
pasoDelResumenValores
*********************************************************************
*********************************************************************/
void guardarValorMejorIndividuo()
        {
        if( (estadoActualProcesador[IdProceso]==1) &&
                ((PGAGetGAIterValue(ctx)%pasoDelResumenValores)==0 ||
PGAGetGAIterValue(ctx)==1) )
                {
                int mejor = PGAGetBestIndex(ctx,PGA_OLDPOP);

        aniadirElementoAlFinal(&valorMejoresIndividuos,PGAGetEvaluation(ctx,mejo
r,PGA_OLDPOP));
                }
        }

/*********************************************************************
*********************************************************************
 guarda el peor valor del individuo en cada iteración que marque la variables
pasoDelResumenValores
*********************************************************************
*********************************************************************/

void guardarValorPeorIndividuo()
        {
        if( (estadoActualProcesador[IdProceso]==1) &&
                ((PGAGetGAIterValue(ctx)%pasoDelResumenValores)==0 ||
PGAGetGAIterValue(ctx)==1) )
                {
                int peor = PGAGetWorstIndex(ctx,PGA_OLDPOP);

        aniadirElementoAlFinal(&valorPeoresIndividuos,PGAGetEvaluation(ctx,peor,
PGA_OLDPOP));
                }
        }
```

```
/*********************************************************************
*********************************************************************
 guarda el valor medio del individuo en cada iteración que marque la variables
pasoDelResumenValores
*********************************************************************
*********************************************************************/
void guardarValorMedio()
	{
	if( (estadoActualProcesador[IdProceso]==1) &&
		((PGAGetGAIterValue(ctx)%pasoDelResumenValores)==0 ||
PGAGetGAIterValue(ctx)==1) )
		{
		calcularMediaFitnessYEvaluacion();
		aniadirElementoAlFinal(&valorMediosIndividuos,mediaEvaluacion);
		}
	}


/*********************************************************************
*********************************************************************
guarda los valores de herencia
*********************************************************************
*********************************************************************/
void guardarHerencia()
	{
	if ((PGAGetGAIterValue(ctx)%pasoDelResumenHerencia)==0 ||
PGAGetGAIterValue(ctx)==1)
		{
		int herencia[MAX_PROCESADORES];
		//calculo la herencia en este momento
		calcularHerencia(&herencia);

		//guardo la información de la herencia
		int j;
		for (j=0;j<numProcesadores;j++)
			{
			//lo añado de forma de tanto por cien
	aniadirElementoAlFinal(&herenciaProcesadores[j],(double)(herencia[j]/(do
uble)PGAGetPopSize(ctx))*100);
			}
		}
	}
```

Con las siguientes funciones procesamos los valores recibidos de cada procesador y los guardamos en un array dinámico para su posterior volcado a un fichero de esto, a partir del cual, extraeremos las graficas para realizar el estudio.

```
/*********************************************************************
*********************************************************************
* El procesador 0 procesa lo mejores individuos al finalizar cada repetición
	*
*********************************************************************
*********************************************************************/

void procesarMejoresValores()
	{
		if (IdProceso != 0)
			{ //deben enviar sus array al procesador 0
			enviarArrayDinamico(&valorMejoresIndividuos,0);
			}
		else if (IdProceso == 0)
			{ //debe recibir todos los arrays y procesarlos
			//escribo los valores del procesador 0
			int j;
```

```
                    for (j=0;j<valorMejoresIndividuos.ocupacion;j++)

      aniadirElementoAlFinal(&mejoresIndividuos,valorMejoresIndividuos.array[j
]);

                    //recibo los array de los demás procesadores e imprimo
                    int i;
                    for (i = 1;i<numProcesadores;i++)
                          {
                          //recibo el array
                          arrayDinamico aux;
                          recibirArrayDinamico(&aux,i);
                          //reviso el array de mejoresTemporales actualizando
los mejores
                          int j;
                          for (j=0;j<aux.ocupacion;j++)
                                { //si el valor mejora lo sustituyo
                                if (j<mejoresIndividuos.ocupacion)
                                      { //si el valor mejora lo sustituyo
                                      if (aux.array[j] >
mejoresIndividuos.array[j])

      aniadirElemento(&mejoresIndividuos,aux.array[j],j);
                                      }
                                else //añado el valor porque no había máximos

      aniadirElementoAlFinal(&mejoresIndividuos,aux.array[j]);
                                }
                          }
                    //escribo los datos del experimento
                    escribirFicheroGraficaMejores();
                    }
      }

/*********************************************************************
*********************************************************************
* El procesador 0 procesa lo mejores individuos al finalizar cada repetición
      *
*********************************************************************
*********************************************************************/

void procesarMediosValores()
      {
      if (IdProceso != 0)
            { //deben enviar sus array al procesador 0
            enviarArrayDinamico(&valorMediosIndividuos,0);
            }
      else if (IdProceso == 0)
            { //debe recibir todos los arrays y procesarlos
            //escribo los valores del procesador 0
            int j;
            for (j=0;j<valorMediosIndividuos.ocupacion;j++)
                  {

      aniadirElementoAlFinal(&mediosIndividuos,valorMediosIndividuos.array[j])
;
                  aniadirElementoAlFinal(&aportacionesALaMedia,1.0);
                  }

            //recibo los array de los demás procesadores e imprimo
            int i;
            for (i = 1;i<numProcesadores;i++)
                  {
                  //recibo el array
                  arrayDinamico aux;
                  recibirArrayDinamico(&aux,i);
                  //reviso el array de peores Temporales actualizando los
peores
```

```
                        for (j=0;j<aux.ocupacion;j++)
                                { //lo sumo a lo que haya
                                if (j<mediosIndividuos.ocupacion)
                                        { //lo añado para la media

        aniadirElemento(&mediosIndividuos,mediosIndividuos.array[j]+aux.array[j]
,j);

        aniadirElemento(&aportacionesALaMedia,aportacionesALaMedia.array[j]+1.0,
j);
                                        }
                                else //añado el valor porque no había
                                        {

        aniadirElementoAlFinal(&mediosIndividuos,aux.array[j]);

        aniadirElementoAlFinal(&aportacionesALaMedia,1.0);
                                        }
                                }
                        }

                //hago la división entre el numero de procesadores para saber la
media
                for (j=0;j<mediosIndividuos.ocupacion;j++)
                        {
                        double media =
mediosIndividuos.array[j]/(double)aportacionesALaMedia.array[j];
                        aniadirElemento(&mediosIndividuos,media,j);
                        }

                //escribo los datos del experimento
                escribirFicheroGraficaMedios();
                }
        }


/*****************************************************************
******************************************************************
* El procesador 0 procesa lo mejores individuos al finalizar cada repetición
        *
******************************************************************
******************************************************************/

void procesarPeoresValores()
        {
                if (IdProceso != 0)
                        { //deben enviar sus array al procesador 0
                        enviarArrayDinamico(&valorPeoresIndividuos,0);
                        }
                else if (IdProceso == 0)
                        { //debe recibir todos los arrays y procesarlos
                        //escribo los valores del procesador 0
                        int j;
                        for (j=0;j<valorPeoresIndividuos.ocupacion;j++)

        aniadirElementoAlFinal(&peoresIndividuos,valorPeoresIndividuos.array[j])
;

                        //recibo los array de los demás procesadores e imprimo
                        int i;
                        for (i = 1;i<numProcesadores;i++)
                                {
                                //recibo el array
                                arrayDinamico aux;
                                recibirArrayDinamico(&aux,i);
                                //reviso el array de peores Temporales actualizando
los peores
                                int j;
```

```
                        for (j=0;j<aux.ocupacion;j++)
                              { //si el valor empeora lo sustituyo
                              if (j<peoresIndividuos.ocupacion)
                                    { //si el valor mejora lo sustituyo
                                    if (aux.array[j] <
peoresIndividuos.array[j])

      aniadirElemento(&peoresIndividuos,aux.array[j],j);
                                    }
                              else //añado el valor porque no había máximos

      aniadirElementoAlFinal(&peoresIndividuos,aux.array[j]);
                              }
                        }
                  //escribo los datos del experimento
                  escribirFicheroGraficaPeores();
                  }
      }

/*********************************************************************
*********************************************************************
Los procesos que no sean el 0 envían toda la información de tiempos de
ejecución
al proceso 0, este la recibe y la imprime
*********************************************************************
*********************************************************************/
void procesarTiempos()
      {
      //calculo el tiempo de ejecución
      double tiempo = (fin - inicio) / CLK_TCK;

      if (IdProceso == 0)
            {
            //el procesador 0 recibe la información de todos los demás, y
guarda en tiempos
            //la suya propia
            inicializarArray(&tiempos,10);
            aniadirElementoAlFinal(&tiempos,tiempo);
            //espero la recepción de la información del resto de procesadores
            int i;
            for (i=1;i<numProcesadores;i++)
                  {
                  MPI_Recv(&tiempo, 1, MPI_DOUBLE, i, 555,
MPI_COMM_WORLD,&status);
                  aniadirElementoAlFinal(&tiempos,tiempo);
                  }
            escribirFicheroGraficaTiempos();
            }
      else
            {//el resto de procesadores se limitan a enviar la variable
tiempo
            MPI_Send(&tiempo, 1, MPI_DOUBLE, 0, 555, MPI_COMM_WORLD);
            }

      }
```

Para conseguir resultados diferentes en cada ejecución de un experimento necesitábamos inicializar la veneración de la secuencia de números pseudos-aleatorios con un número verdaderamente aleatorio. Para ello utilizamos el generador de números aleatorio /dev/random del sistema operativo.

Este generador de números aleatorios recoge el ruido ambiental procedente de los manejadores de dispositivo y otras fuentes y lo guarda en un "almacén de entropía". El generador también mantiene una estimación del número de bits de ruido en el almacén de entropía. Los números aleatorios se crean a partir de este almacén.

El encargado de leer el fichero es únicamente el procesador 0 para evitar esperas innecesarias a que el sistema operativo de permiso de lectura sobre este archivo. De este modo conseguimos mayor velocidad de ejecución.

El procesador lee el archivo random lo utiliza como semilla y comienza una generación de números pseudo aleatorios que ira pasando al resto de procesadores.

```
/*****************************************************************
******************************************************************
El procesador 0 consigue una semilla leyendo en el fichero /dev/random
y le envía una semilla aleatoria al resto de procesadores
******************************************************************
*****************************************************************/
void conseguirSemilla()
	{
	if (IdProceso == 0)
		{
		//preparo una semilla inicial basándome en el archivo random
		//del sistema operativo
		int randFile;

		randFile = open("/dev/random",O_RDONLY);

		read(randFile,&semilla,sizeof(rand));
		semilla = semilla%100;
		while (semilla<=0)
			{
			read(randFile,&semilla,sizeof(rand));
			semilla = semilla%100;
			}
		close(randFile);
		//envío la semilla al resto
		if (numProcesadores != 1)
			{
			int i;
			int sucesionSemillas;
			srand(semilla);
			for (i=1;i<numProcesadores;i++)
				{
				sucesionSemillas = rand()%100;
				while (sucesionSemillas<=0)
					sucesionSemillas = rand()%100;
				MPI_Send(&sucesionSemillas, 1, MPI_INT, i, 666,
MPI_COMM_WORLD);
				}
			}
		}
	else
		{
		MPI_Recv(&semilla, 1, MPI_INT, 0, 666, MPI_COMM_WORLD,&status);
		}
	}
```

Y por ultimo mostramos las funciones utilizadas para extraer la información relacionada con el fitness y la evaluación de la población así como de su herencia.

```
/*********************************************************************
*********************************************************************
calcula el valor medio de la función de evaluación y del fitness
*********************************************************************
*********************************************************************/
void calcularMediaFitnessYEvaluacion()
	{
	mediaEvaluacion = 0;
	mediaFitness = 0;

	int i;
	//todos los procesadores hacen este código de recopilación de valores
	//recorro la población mirando en cada individuo de que procesador ha
recibido información
	//y contando el origen de cada individuo
	//y realizando la media de la función de evaluación  y del fitness
	for (i = 0;i<PGAGetPopSize(ctx);i++)
		{
		mediaEvaluacion = mediaEvaluacion + PGAGetEvaluation ( ctx, i,
PGA_OLDPOP );
		mediaFitness = mediaFitness + PGAGetFitness ( ctx, i, PGA_OLDPOP
);
		}
	//realizo la división entre el número de individuos de la población para
realizar la media del fitness y de la función de evaluación
	mediaEvaluacion = mediaEvaluacion / (double)PGAGetPopSize(ctx);
	mediaFitness = mediaFitness / (double)PGAGetPopSize(ctx);
	}



/*********************************************************************
*********************************************************************
Procesa los valores de herencia de todos los procesadores
*********************************************************************
*********************************************************************/
void procesarHerencia()
	{
	int i,j,k;
	typedef struct
		{
		arrayDinamico herenciaProc[MAX_PROCESADORES];
		}
	herencias;

	herencias herenciaTotal[MAX_PROCESADORES];

	//inicializo herencias
	for (i=0;i<numProcesadores;i++)
		for (j=0;j<numProcesadores;j++)
			inicializarArray(&(herenciaTotal[i].herenciaProc[j]),100);

	if (IdProceso == 0)
		{
		//copio la información del procesador 0 a herencia total
		for (i=0;i<numProcesadores;i++)
			{
			for (j=0;j<herenciaProcesadores[0].ocupacion;j++)

	aniadirElementoAlFinal(&(herenciaTotal[0].herenciaProc[i]),herenciaProce
sadores[i].array[j]);
			}
```

```
		//el procesador 0 recibe la información de todos los de más
		for (i=1;i<numProcesadores;i++)
			for (j=0;j<numProcesadores;j++)

	recibirArrayDinamico(&(herenciaTotal[i].herenciaProc[j]),i);

		//escribo la información en el fichero
		for (i=0;i<numProcesadores;i++)
			{
			int ocupacion = herenciaTotal[i].herenciaProc[0].ocupacion;
			fprintf(ficheroHerencia,"%i\n",ocupacion);
			for (j=0;j<numProcesadores;j++)
				{
				for (k=0;k<ocupacion;k++)

	fprintf(ficheroHerencia,"%e\t",herenciaTotal[i].herenciaProc[j].array[k]
);
				fprintf(ficheroHerencia,"\n");
				}
			}

		//libero las variables
		for (i=0;i<numProcesadores;i++)
			for (j=0;j<numProcesadores;j++)
				liberarArray(&(herenciaTotal[i].herenciaProc[j]));

		fclose(ficheroHerencia);
		procesarArchivoHerencia();
		}
	else
		{ //el resto de procesadores se limitan a enviar
		for (i=0;i<numProcesadores;i++)
			enviarArrayDinamico(&(herenciaProcesadores[i]),0);
		}
	}


/*********************************************************************
*********************************************************************
Calcula la herencia de cada población
*********************************************************************
*********************************************************************/
void calcularHerencia(int *herencia)
	{
	int i,j;

	//inicializacion de las variables
	for (i=0;i<MAX_PROCESADORES;i++)
		{
		herencia[i] = 0;
		}

	//todos los procesadores hacen este código de recopilación de valores
	//recorro la población mirando en cada individuo de que procesador ha
recibido información
	//y contando el origen de cada individuo
	//y realizando la media de la función de evaluación  y del fitness
	for (i = 0;i<PGAGetPopSize(ctx);i++)
		{
		for (j=0;j<numProcesadores;j++)
			{
			herencia[j]= herencia[j] + PGAGetIndividual(ctx, i,
PGA_OLDPOP)->procesador[j];
			}
		}
	}
```

11.2. Generación de gráficas

Para la creación de las gráficas se ha utilizado Visual Basic para aplicaciones, contra Excel 2003.

Se ha generado una serie de macros cuyo funcionamiento general ha sido el siguiente:

Lectura de los datos en formato texto plano, a partir de los archivos creados por el programa "proyecto.c". Esta lectura se lleva a cabo mediante la importación de datos a Excel.

Tratamiento de los datos (formato, cálculo de medias, etc).

Creación de los gráficos y formato de los mismos.

Aunque ése es el funcionamiento general, hemos hecho tres tipos de libros Excel para los disntintos datos: herencia, resultados y tiempos. A continuación describiremos un poco más a fondo cada uno de ellos.

Herencia

El libro contiene dos hojas, llamadas "Gráficos" y "Datos". En la primera es donde mostramos todos los gráficos y en la segunda es donde importamos los datos obtenidos en la ejecución del programa.

Un ejemplo de los archivos de texto de los que leemos los datos a representar es el siguiente:

1	20	40	60	80	100	120
1,00E+08	1,00E+08	1,00E+08	1,00E+08	1,00E+08	1,00E+08	1,00E+08
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1	20	40	60	80	100	120
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1,00E+08	1,00E+08	1,00E+08	1,00E+08	1,00E+08	9,62E+07	9,64E+07
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	3,85E+06	4,56E+07

1	20	40	60	80	100	120
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	3,85E+06	4,92E+07
1,00E+08	1,00E+08	1,00E+08	1,00E+08	1,00E+08	9,62E+07	9,64E+07
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1	20	40	60	80	100	120
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	3,85E+06	4,62E+07
1,00E+08	1,00E+08	1,00E+08	1,00E+08	1,00E+08	9,62E+07	9,58E+07
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1	20	40	60	80	100	120
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	3,85E+06	5,08E+07
1,00E+08	1,00E+08	1,00E+08	1,00E+08	1,00E+08	9,62E+07	9,75E+07
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1	20	40	60	80	100	120
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	3,85E+06	4,67E+07
1,00E+08	1,00E+08	1,00E+08	1,00E+08	1,00E+08	9,62E+07	9,70E+07
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1	20	40	60	80	100	120
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	3,85E+06	5,37E+07
1,00E+08	1,00E+08	1,00E+08	1,00E+08	1,00E+08	9,62E+07	9,60E+07
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1	20	40	60	80	100	120
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00

```
 0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00
 0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00
 0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00
 0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00
 0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00   0.000000e+00      3,85E+06       4,85E+07
    1,00E+08       1,00E+08       1,00E+08       1,00E+08       1,00E+08       9,62E+07       9,72E+07

CROSSOVER_TWOPT
0.800000
0.002000
FITNESS_RAW
POPREPL_BEST
SELECT_TOURNAMENT
FMODAL
          100
           26
HASTA LA CONVERGENCIA
          100
            1
            8
0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO 4-ACTIVO 5-ACTIVO 6-ACTIVO 7-ACTIVO
0-100 1-200 2-300 3-400 4-Siempre 5-Siempre 6-Siempre 7-Siempre
```

Se ha suprimido gran cantidad de columnas debido a que las dimensiones del archivo son demasiado grandes.

Una vez iniciado el programa en VBA, desde la función principal se llama a la macro “ObtenerDatosExternos”, que contiene el siguiente código:

```
'Parámetros:
'i: número de fichero de texto que estamos importando

Sub ObtenerDatosExternos(i As Integer)

    Application.Workbooks(1).Worksheets.Add
    'Le indicamos de dónde tiene que leer los datos, y dónde tiene que
colocarlos, formato, etc
    With ActiveSheet.QueryTables.Add(Connection:= _
        "TEXT;" & ActiveWorkbook.Path & "/graficaHerencia" & CStr(i) & ".txt",
_
        Destination:=Range("A1"))
        .Name = "grafica" & i
        .FieldNames = True
        .RowNumbers = False
        .FillAdjacentFormulas = False
        .PreserveFormatting = True
        .RefreshOnFileOpen = True
        .RefreshStyle = xlInsertDeleteCells
        .SavePassword = False
        .SaveData = False
        .AdjustColumnWidth = False
        .RefreshPeriod = 0
        .TextFilePromptOnRefresh = False
        .TextFilePlatform = 850
        .TextFileStartRow = 1
        .TextFileParseType = xlDelimited
        .TextFileTextQualifier = xlTextQualifierDoubleQuote
        .TextFileConsecutiveDelimiter = False
        .TextFileTabDelimiter = True
        .TextFileSemicolonDelimiter = False
```

```
        .TextFileCommaDelimiter = False
        .TextFileSpaceDelimiter = False
        .TextFileColumnDataTypes = Array(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, _
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 _
        , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, _
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
        .TextFileDecimalSeparator = "."
        .TextFileThousandsSeparator = " "
        .TextFileTrailingMinusNumbers = True
        .Refresh BackgroundQuery:=False

    End With

    'Para saber el número de procesadores que tiene el experimento que estamos
representando
    If Cells(10, 1).Value = "" Then
        numProcesadores = 8
    Else
        If Cells(6, 1).Value = "" Then
            numProcesadores = 4
        Else
            If Cells(4, 1).Value = "" Then
                numProcesadores = 2
            Else
                numProcesadores = 1
            End If
        End If
    End If
End Sub
```

No a todos los ficheros les corresponde el mismo número de gráficos. De los archivos correspondientes a experimentos con ocho procesadores, se obtienen treinta y dos gráficas, mientras de los de cuatro se obtienen dieciséis, etc. Por eso existe la variable numProcesadores, que utilizaremos en la macro principal, tras la llamada a "ObtenerDatosExternos", con el siguiente código:

```
For i = 1 To numProcesadores
'Mostramos un mensaje en la barra de estado para indicar que el programa está
en ejecución
        Application.StatusBar = "Generando gráfico " & CStr(j) & " de " &
CStr(numArchivos) & ". Espere, por favor..."

        Call GenerarGrafico(cont, ultimaLineaDatos, numGrafico)

'cont nos sirve para calcular la línea de la hoja "Datos" a partir de la cual
empiezan los datos del fichero actual
        cont = cont + numProcesadores + 2
'número o posición del gráfico que estamos creando, dentro de todos los
gráficos obtenidos de un mismo fichero de texto
        numGrafico = numGrafico + 1
    Next
```

Como se ve en el bucle, creamos un nuevo gráfico con la macro "GenerarGrafico", que mostramos a continuación:

```
'Parámetros:
'primlin: línea de la hoja de "Datos" donde comienzan los datos a representar
en el gráfico actual
'ultLin: línea de la hoja de "Datos" donde comienzan los datos a representar
en el gráfico actual
'numGraf: número o posición del gráfico que estamos creando, dentro de todos
los gráficos obtenidos de un mismo fichero de texto

Public Sub GenerarGrafico(ByVal primlin As Long, ultLin As Integer, numGraf As
Integer)
    'Variable donde almacenaremos el texto que mostramos bajo el gráfico
    Dim titGrafico As String

    'Indica el número máximo de datos que vamos a representar en el eje X
    Dim numeroDatos As Integer
    'Indican el mínimo y el máximo a representar en el eje Y
    Dim minimo As Double
    Dim maximo As Double
    'Indica el valor máximo a representar en el eje X
    Dim maximoX As Double

    'Amacenará el nombre del experimento que estamos representando
    Dim strAlgoritmo As String

    On Error GoTo Error

    'Establecemos el máximo y el mínimo del gráfico para que éste se visualice
correctamente

    minimo = 0
    maximo = 100

    'Contamos la primera línea de datos para poder asignar a numeroDatos el
valor correspondiente
    rangoacontar = primlin & ":" & primlin
    numeroDatos =
Datos.Application.WorksheetFunction.CountA(Worksheets("Datos").Range(rangoacon
tar))

    'Leemos el valor máximo del eje X, que es la última columna de la primera
fila de los datos importados
    maximoX = Sheets("Datos").Cells(primlin, numeroDatos)

    titGrafico = GenerarTitulo(ultLin, strAlgoritmo)

    Worksheets("Graficos").Activate

    'Añadimos un nuevo objeto gráfico
    Charts.Add

    'Tipo de gráfico: dispersión

    ActiveChart.ChartType = xlXYScatter

    'Establecemos la fuente de datos del gráfico

    ActiveChart.SetSourceData
Source:=Sheets("Datos").Range(Sheets("Datos").Cells(primlin, 1),
Sheets("Datos").Cells(primlin + numProcesadores, numeroDatos)), PlotBy:= _
        xlRows

    'Generamos la información que ha de mostrarse en la leyenda

    Dim procesadorActivo As Integer
```

```
    Select Case numProcesadores
        Case 8:
            procesadorActivo = 8 - ((ultLin - primlin - 14) / 10)
            ActiveChart.SeriesCollection(1).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 0"
            ActiveChart.SeriesCollection(2).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 1"
            ActiveChart.SeriesCollection(3).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 2"
            ActiveChart.SeriesCollection(4).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 3"
            ActiveChart.SeriesCollection(5).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 4"
            ActiveChart.SeriesCollection(6).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 5"
            ActiveChart.SeriesCollection(7).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 6"
            ActiveChart.SeriesCollection(8).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 7"
        Case 4:
            procesadorActivo = 4 - ((ultLin - primlin - 14) / 6)
            ActiveChart.SeriesCollection(1).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 0"
            ActiveChart.SeriesCollection(2).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 1"
            ActiveChart.SeriesCollection(3).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 2"
            ActiveChart.SeriesCollection(4).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 3"
        Case 2:
            procesadorActivo = 2 - ((ultLin - primlin - 14) / 4)
            ActiveChart.SeriesCollection(1).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 0"
            ActiveChart.SeriesCollection(2).Name = "Herencia en el proc." &
procesadorActivo & " del proc. 1"
    End Select


    'El gráfico irá en la hoja existente “Graficos” como un objeto

    ActiveChart.Location Where:=xlLocationAsObject, Name:="Graficos"

    'Damos formato al gráfico

Call FormatearGrafico(primlin, strAlgoritmo, titGrafico, minimo, maximo,
maximoX, "Graficos", True, numGraf)

    ActiveChart.ChartArea.Select

    Exit Sub

    'En caso de error, queremos que nos muestre una alerta
Error:
    MsgBox Err.Description, vbCritical

End Sub
```

Como se puede ver, existen algunas funciones auxiliaries como Generartit, que genera el texto que se muestra bajo el gráfico con los datos del experimento, FormatearGrafico, etc., cuyo código no consideramos relevante para exponerlo en este documento.

Una vez generado el primer gráfico, importamos la siguiente hoja de datos, generamos sus gráficos correspondientes y así sucesivamente. Este proceso se realiza 64 veces, una por cada fichero de herencia que tenemos.

Resultados

Hay tres tipos de libros de resultados, los de mejores, peores y medios. La única diferencia entre ellos está en la importación de datos, ya que leen de ficheros de texto con distinto nombre.

En estos documentos, tenemos varias hojas para gráficos, en las que generamos éstos combinándolos de distintas formas para ver las diferencias entre unos parámetros de ejecución y otros. Por ejemplo, diferencias entre ejecutar el experimento OneMax incomunicando procesadores y sin incomunicarlos.

En primer lugar, se generan los gráficos simples, para los que seguimos el siguiente proceso.

Leemos la información de los archivos de texto. Un ejemplo de los ficheros de texto de los que obtenemos los datos a representar es el siguiente:

1,48E+07	1,81E+07	1,93E+07	2,00E+07	2,13E+07	2,40E+07	2,40E+07
1	20	40	60	80	100	120
1,68E+07	1,87E+07	2,18E+07	2,34E+07	2,34E+07	2,35E+07	2,43E+07
1	20	40	60	80	100	120
1,64E+07	1,74E+07	1,92E+07	2,19E+07	2,41E+07	2,52E+07	2,58E+07
1	20	40	60	80	100	120
1,66E+07	1,91E+07	2,22E+07	2,44E+07	2,59E+07	2,69E+07	2,73E+07
1	20	40	60	80	100	120
1,70E+07	1,91E+07	2,23E+07	2,27E+07	2,35E+07	2,35E+07	2,45E+07
1	20	40	60	80	100	120
1,62E+07	1,83E+07	2,19E+07	2,19E+07	2,27E+07	2,28E+07	2,31E+07
1	20	40	60	80	100	120
1,83E+07	1,93E+07	2,04E+07	2,33E+07	2,41E+07	2,48E+07	2,54E+07
1	20	40	60	80	100	120
1,63E+07	1,91E+07	2,10E+07	2,25E+07	2,33E+07	2,37E+07	2,46E+07
1	20	40	60	80	100	120
1,61E+07	1,84E+07	2,08E+07	2,14E+07	2,28E+07	2,36E+07	2,43E+07
1	20	40	60	80	100	120
1,60E+07	1,81E+07	1,98E+07	2,25E+07	2,33E+07	2,37E+07	2,43E+07
1	20	40	60	80	100	120
1,66E+07	1,83E+07	2,06E+07	2,32E+07	2,39E+07	2,50E+07	2,53E+07
1	20	40	60	80	100	120
1,54E+07	1,74E+07	2,01E+07	2,15E+07	2,36E+07	2,55E+07	2,59E+07
1	20	40	60	80	100	120
1,60E+07	1,81E+07	1,98E+07	2,25E+07	2,33E+07	2,37E+07	2,43E+07
1	20	40	60	80	100	120
1,64E+07	1,90E+07	2,05E+07	2,26E+07	2,33E+07	2,37E+07	2,41E+07
1	20	40	60	80	100	120
1,63E+07	1,91E+07	2,10E+07	2,25E+07	2,33E+07	2,37E+07	2,46E+07
1	20	40	60	80	100	120
1,59E+07	1,80E+07	2,16E+07	2,36E+07	2,58E+07	2,58E+07	2,65E+07

1	20	40	60	80	100	120
1,62E+07	1,83E+07	2,19E+07	2,19E+07	2,27E+07	2,28E+07	2,31E+07
1	20	40	60	80	100	120
1,64E+07	1,80E+07	2,12E+07	2,20E+07	2,28E+07	2,42E+07	2,49E+07
1	20	40	60	80	100	120
1,57E+07	1,89E+07	1,99E+07	2,12E+07	2,23E+07	2,33E+07	2,46E+07
1	20	40	60	80	100	120
1,51E+07	1,71E+07	1,94E+07	2,03E+07	2,17E+07	2,27E+07	2,37E+07
1	20	40	60	80	100	120
1,62E+07	1,86E+07	2,24E+07	2,28E+07	2,33E+07	2,51E+07	2,59E+07
1	20	40	60	80	100	120
1,70E+07	1,93E+07	2,05E+07	2,18E+07	2,34E+07	2,34E+07	2,42E+07
1	20	40	60	80	100	120
1,66E+07	1,89E+07	2,08E+07	2,35E+07	2,40E+07	2,47E+07	2,55E+07
1	20	40	60	80	100	120
1,66E+07	1,93E+07	2,06E+07	2,13E+07	2,31E+07	2,44E+07	2,50E+07
1	20	40	60	80	100	120
1,68E+07	1,81E+07	2,05E+07	2,16E+07	2,28E+07	2,34E+07	2,45E+07
1	20	40	60	80	100	120
1,71E+07	1,89E+07	2,12E+07	2,29E+07	2,52E+07	2,58E+07	2,61E+07
1	20	40	60	80	100	120
1,58E+07	1,88E+07	2,17E+07	2,26E+07	2,52E+07	2,56E+07	2,64E+07
1	20	40	60	80	100	120
1,83E+07	1,93E+07	2,12E+07	2,21E+07	2,41E+07	2,49E+07	2,53E+07
CROSSOVER_TWOPT						
0.800000						
0.002000						
FITNESS_RAW						
POPREPL_BEST						
SELECT_TOURNAMENT						
FMODAL						
100						
26						
HASTA LA CONVERGENCIA						
25						
1						
8						
0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO 4-ACTIVO 5-ACTIVO 6-ACTIVO 7-ACTIVO						
0-Siempre 1-Siempre 2-Siempre 3-Siempre 4-Siempre 5-Siempre 6-Siempre 7-Siempre						

En el caso de los libros de resultados, el procedimiento que seguimos es diferente a los de herencia. En primer lugar importamos todos los datos, después los tratamos para poder acceder a ellos de forma más cómoda a la hora de crear los gráficos y, finalmente, generamos éstos.

El código para obtener los datos es muy similar al que utilizábamos en herencia, con la diferencia de que no necesitamos saber el número de procesadores y en este caso, una vez importada cada hoja, llamamos la función "ConvertirFichero", que se encargará del tratamiento de los datos que acabamos de importar. El código de esta última es el que sigue:

```
'parámetros:
' hoja: hoja donde están los datos inicialmente, al ser importados. Es una
hoja provisional, después se elimina
' primlinDatos: línea de la hoja "Datos" en la que comenzarán los datos del
fichero de texto actual
' diferenciaGeneraciones: diferencia de generaciones entre los datos que
mostramos(en estos experimentos, 20)
Public Sub ConvertirFichero(ByVal hoja As String, ByVal primlinDatos As
Integer, diferenciaGeneraciones As Integer)
    colMax = 0
    Cells.Select
    Selection.NumberFormat = "0.00E+00"
    Rows("1:1").Select
    Selection.NumberFormat = "General"

    'Tratamos la hoja inicial
    difGeneraciones = Cells(numIteraciones * 2 + 11, 1).Value
    Dim valores() As Double
    cont = 1
    a = numIteraciones

'En el siguiente bucle miramos cuál es la longitud mayor entre todas las filas
de datos
bucle:    For i = 1 To a - 1
        rangoacontar = cont & ":" & cont
        numeroDatos =
Datos.Application.WorksheetFunction.CountA(Worksheets(hoja).Range(rangoacontar
))

        If numeroDatos > colMax Then
            colMax = numeroDatos
        End If
        If (cont = 1) Then
            cont = cont + 2
            GoTo bucle
        End If
        Rows(CStr(cont) & ":" & CStr(cont)).Select
        Selection.Delete Shift:=xlUp
        cont = cont + 1
    Next

    'Rellenamos los datos que faltan en cada línea hasta llegar al final
(maxCol), con los máximos
    cont = 1
    Dim formulaMax As String
    For i = 1 To a
        rangoacontar = cont & ":" & cont
        numeroDatos =
Datos.Application.WorksheetFunction.CountA(Worksheets(hoja).Range(rangoacontar
))

        If numeroDatos < colMax Then

            Dim col As Integer
            If (cont = 1) Then
                For col = numeroDatos + 1 To colMax
                    formulaMax = "=MAX(RC[-" & CStr(col - 1) & "]:RC[-1])"

                        Worksheets(hoja).Cells(cont, col).FormulaR1C1 =
Worksheets(hoja).Cells(cont, col - 1).FormulaR1C1 + diferenciaGeneraciones
```

```
                        Worksheets(hoja).Cells(cont + 1, col).FormulaR1C1 =
formulaMax
                        If (Worksheets(hoja).Cells(cont + 1, col).Value = 0)
Then
                            Worksheets(hoja).Cells(cont + 1, col) =
Worksheets(hoja).Cells(cont + 1, col - 1)
                        End If
                Next
                cont = cont + 2
            Else
                For col = numeroDatos + 1 To colMax
                    formulaMax = "=MAX(RC[-" & CStr(col - 1) & "]:RC[-1])"
                    Worksheets(hoja).Cells(cont, col).FormulaR1C1 = formulaMax
                    If (Worksheets(hoja).Cells(cont, col).Value = 0) Then
                            Worksheets(hoja).Cells(cont, col) =
Worksheets(hoja).Cells(cont, col - 1)
                        End If
                Next
                cont = cont + 1
            End If
        Else
            cont = cont + 1
        End If


    Next

    'Pasamos los datos de la hoja inicial a la hoja “Datos”

    Worksheets(hoja).Rows("1:1").Select
    Selection.Copy
    Sheets("Datos").Select
    Rows(CStr(primlinDatos) & ":" & CStr(primlinDatos)).Select
    ActiveSheet.Paste

    'Calculamos las medias de todas las columnas que hemos creado. Esta será
la información para la hoja “Datos”
    Dim filaAct As Integer
    Dim rangoVertical As String
    filaAct = primlinDatos + 1
    Cells(filaAct, 1).Select
    rangoVertical = "A2:A" & numIteraciones + 1
    ActiveCell.Formula = "=AVERAGE(" & hoja & "!" & rangoVertical & ")"

    Selection.AutoFill Destination:=Range(Selection,
Selection.End(xlToRight)), Type:=xlFillDefault

On Error GoTo err
    Cells(filaAct, colMax + 1).Select
    Range(Selection, Selection.End(xlToRight)).ClearContents
err:
    'Copiamos el resto de filas, donde están los parámetros del experimento
    Cells(filaAct + 1, 1).Select
    Cells(filaAct + 1, 1).Formula = "=" & hoja & "!A" & numIteraciones + 2
    Cells(filaAct + 1, 1).Select
    Selection.AutoFill Destination:=Range(Cells(filaQueToca + 1, 1),
Cells(filaAct + 15, 1)), Type:=xlFillDefault

    'Calculamos el mínimo y el máximo de los datos
    Cells(filaAct + 16, 1).FormulaR1C1 = "=MAX(R[-16]C:R[-16]C[" & colMax - 1
& "])"
    Cells(filaAct + 17, 1).FormulaR1C1 = "=MIN(R[-17]C:R[-17]C[" & colMax - 1
& "])"

    End Sub
```

Una vez hecho esto, se generan los gráficos correspondientes con la macro "GenerarGrafico". Al ser también muy similar a la utilizada en herencia, no vamos a exponer el código.

Cuando los datos están importados, ya podemos generar el resto de gráficos, los compuestos. Para eso tenemos distintas funciones que toman la información que necesitan de la hoja "Datos", la copian en su hoja correspondiente con el formato y orden necesario, y crean las nuevas gráficas. Un ejemplo de este tipo de macros es la que genera las combinaciones de gráficos con y sin incomunicación de procesadores. El código es el siguiente:

```
Public Sub GenerarCombinado()

    Dim a As Long
    Dim cont As Long
    Dim myRange As Range

    'Nombre de las hojas donde irán los gráficos y los datos
    nombreHojaGraficos = "CombiGrafConSinIncom"
    nombreHojaDatos = "CombiConSinIncom"

    cont = 1
    numIteraciones = 100
    numArchivos = 64

    Application.DisplayFormulaBar = False
    oldStatusBar = Application.DisplayStatusBar
    Application.ScreenUpdating = False

    'Contamos el número de gráficos que vamos a sacar
    a =
Datos.Application.WorksheetFunction.CountA(Worksheets("Datos").Range("A:A")) /
19

    Dim copiar As Boolean
    copiar = False

    'Si existen las hojas donde vamos a crear los gráficos, las eliminamos
    On Error GoTo NoExiste
    Application.DisplayAlerts = False
    Worksheets(nombreHojaDatos).Delete
    Worksheets(nombreHojaGraficos).Delete
    Application.DisplayAlerts = True

NoExiste:

    'y las volvemos a añadir
    Sheets.Add.Activate
    ActiveSheet.Name = nombreHojaDatos
    Sheets.Add.Activate
    ActiveSheet.Name = nombreHojaGraficos

    With ActiveWindow
        .DisplayHeadings = False
        .DisplayWorkbookTabs = False
        .DisplayGridlines = False
        .Zoom = 70
    End With

    For i = 1 To a
        Application.StatusBar = "Generando gráficos. Espere, por favor..."
        If (i < numArchivos / 2 + 1) Then
            'Llamamos a la función que generará los gráficos
            Call CombinarConSinInterc(cont)
```

```
        End If
        cont = cont + numIteraciones - 1
    Next

    Application.StatusBar = False
    Application.DisplayStatusBar = oldStatusBar

    Graficos.Activate
    Sheets(nombreHojaGraficos).Activate

    Application.ScreenUpdating = True

End Sub

'parámetros
'primlin: línea donde comienzan los datos de cada gráfico en la hoja “Datos”
Private Sub CombinarConSinInterc(ByVal primlin As Integer)
    Dim strAlgoritmo As String
    Dim minimo As Double
    minimo = ObtenerMinimo(primlin)
    Dim maximo As Double
    maximo = ObtenerMaximo(primlin)

    titGrafico = GenerarTitulo(primlin, strAlgoritmo)

    Worksheets(nombreHojaGraficos).Activate
    Charts.Add.Activate

    'distancia en líneas entre gráficos
    distancia = numIteraciones - 1

    'Tipo de gráfico: dispersión
    ActiveChart.ChartType = xlXYScatter
    ActiveChart.Location Where:=xlLocationAsObject, Name:=nombreHojaGraficos
    Dim s As String
    s = ActiveChart.Name
    s = Replace(s, nombreHojaGraficos & " Gráfico ", "")

'aqui vemos cual es la fila mas larga entre las dos que vamos a comparar.
Están separadas por 32 gráficos,  según el orden en que importamos a la hoja
“Datos” al comienzo del proceso
    Dim rangoMasLargo As String
    rangoacontar = primlin & ":" & primlin
    numeroDatos =
Datos.Application.WorksheetFunction.CountA(Worksheets("Datos").Range(rangoacon
tar))
    rangoMasLargo = rangoacontar
    rangoacontar = primlin + distancia * 32 & ":" & primlin + distancia * 32
    If
(Datos.Application.WorksheetFunction.CountA(Worksheets("Datos").Range(rangoaco
ntar)) > numeroDatos) Then
        numeroDatos =
Datos.Application.WorksheetFunction.CountA(Worksheets("Datos").Range(rangoacon
tar))
        rangoMasLargo = rangoacontar
    End If
    Dim rango As String

    rango = rangoMasLargo
    Sheets("Datos").Activate
    Range(rango).Select
    Selection.Copy

    'Pegamos en la hoja de datos la línea de cabecera más larga
    ActiveSheet.Paste Destination:=Worksheets(nombreHojaDatos).Range("A" &
primlin)

    'Seguidamente pegamos la primera línea de datos
```

```
    rango = primlin + 1 & ":" & primlin + 1
    Sheets("Datos").Activate
    Range(rango).Select
    Selection.Copy
    ActiveSheet.Paste Destination:=Worksheets(nombreHojaDatos).Range("A" &
primlin + 1)

    'A continuación, pegamos la otra línea de datos
    rango = primlin + (distancia * 32 + 1) & ":" & primlin + (distancia * 32 +
1)

    Sheets("Datos").Activate
    Range(rango).Select
    Selection.Copy
    ActiveSheet.Paste Destination:=Worksheets(nombreHojaDatos).Range("A" &
primlin + 2)

    rangoacontar = primlin & ":" & primlin
    numeroDatos =
Datos.Application.WorksheetFunction.CountA(Worksheets(nombreHojaDatos).Range(r
angoacontar))
    rangoacontar = primlin + 2 & ":" & primlin + 2
    If
(Datos.Application.WorksheetFunction.CountA(Worksheets(nombreHojaDatos).Range(
rangoacontar)) > numeroDatos) Then
        numeroDatos =
Datos.Application.WorksheetFunction.CountA(Worksheets(nombreHojaDatos).Range(r
angoacontar))
    End If

    maximoX = Sheets(nombreHojaDatos).Cells(primlin, numeroDatos)

    Sheets(nombreHojaGraficos).ChartObjects(s).Activate

    'Establecemos la fuente de datos del gráfico
    ActiveChart.SetSourceData
Source:=Sheets(nombreHojaDatos).Range(Sheets(nombreHojaDatos).Cells(primlin,
1), _
        Sheets(nombreHojaDatos).Cells(primlin + 2, numeroDatos)), PlotBy:= _
            xlRows

    ActiveChart.SeriesCollection(1).Name = "Sin Incomunicación"
    ActiveChart.SeriesCollection(2).Name = "Con Incomunicación"

    Call FormatearGrafico(primlin, strAlgoritmo, titGrafico, minimo, maximo,
maximoX, nombreHojaGraficos, True)

    ActiveChart.ChartArea.Select

End Sub
```

En los libros de resultados tenemos macros comunes como, por ejemplo, “FormatearGrafico”, que se llaman desde las funciones que generan los distintos tipos de gráficos.

Tiempos

En este tipo de documentos, se importan los 64 archivos con la información de los tiempos de forma análoga a los anteriores. Estos archivos son de la siguiente forma:

7,00E+08	1,10E+09	1,10E+09	1,10E+09	1,10E+09	1,00E+09	1,10E+09	1,00E+09
7,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08
7,00E+08	1,00E+09	1,10E+09	1,10E+09	1,00E+09	1,10E+09	1,00E+09	1,00E+09
7,00E+08	1,10E+09	1,00E+09	1,10E+09	1,00E+09	1,10E+09	1,00E+09	1,00E+09
7,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08
6,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08
6,00E+08	1,10E+09	1,00E+09	1,00E+09	9,00E+08	9,00E+08	1,00E+09	8,00E+08
7,00E+08	8,00E+08	7,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	9,00E+08
6,00E+08	7,00E+08	7,00E+08	7,00E+08	7,00E+08	7,00E+08	7,00E+08	7,00E+08
7,00E+08	8,00E+08	9,00E+08	8,00E+08	9,00E+08	8,00E+08	8,00E+08	8,00E+08
7,00E+08	8,00E+08	7,00E+08	7,00E+08	8,00E+08	7,00E+08	8,00E+08	7,00E+08
6,00E+08	9,00E+08	1,00E+09	1,00E+09	9,00E+08	9,00E+08	9,00E+08	9,00E+08
7,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08	8,00E+08	9,00E+08	9,00E+08
7,00E+08	1,00E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09	9,00E+08
7,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08
7,00E+08	9,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08	9,00E+08	9,00E+08
7,00E+08	7,00E+08	8,00E+08	8,00E+08	7,00E+08	8,00E+08	7,00E+08	8,00E+08
7,00E+08	1,00E+09	1,10E+09	1,00E+09	1,00E+09	1,10E+09	1,00E+09	1,00E+09
6,00E+08	7,00E+08	6,00E+08	7,00E+08	6,00E+08	7,00E+08	8,00E+08	6,00E+08
7,00E+08	1,10E+09	1,10E+09	1,10E+09	1,10E+09	1,00E+09	1,10E+09	1,00E+09
6,00E+08	1,10E+09	1,00E+09	1,10E+09	1,00E+09	1,10E+09	1,10E+09	1,00E+09
6,00E+08	7,00E+08	7,00E+08	6,00E+08	6,00E+08	6,00E+08	6,00E+08	6,00E+08
7,00E+08	9,00E+08	7,00E+08	8,00E+08	9,00E+08	8,00E+08	8,00E+08	7,00E+08
7,00E+08	1,00E+09	1,00E+09	9,00E+08	1,00E+09	8,00E+08	9,00E+08	9,00E+08
6,00E+08	8,00E+08	8,00E+08	7,00E+08	7,00E+08	6,00E+08	7,00E+08	7,00E+08
7,00E+08	1,10E+09	1,10E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09
7,00E+08	1,00E+09	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	1,00E+09
6,00E+08	1,10E+09	9,00E+08	1,00E+09	1,10E+09	1,00E+09	1,00E+09	1,00E+09
6,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08
6,00E+08	8,00E+08	8,00E+08	7,00E+08	8,00E+08	9,00E+08	9,00E+08	8,00E+08
7,00E+08	1,00E+09	9,00E+08	1,00E+09	9,00E+08	1,00E+09	1,00E+09	1,00E+09
7,00E+08	9,00E+08	8,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08
7,00E+08	9,00E+08	9,00E+08	8,00E+08	8,00E+08	7,00E+08	9,00E+08	8,00E+08
7,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08	1,00E+09	9,00E+08	1,00E+09
7,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08
7,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08	9,00E+08	1,00E+09	8,00E+08
6,00E+08	8,00E+08	9,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08
6,00E+08	8,00E+08	8,00E+08	9,00E+08	8,00E+08	8,00E+08	9,00E+08	8,00E+08
7,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08	9,00E+08	8,00E+08
7,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	7,00E+08	7,00E+08	7,00E+08
7,00E+08	8,00E+08	9,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	9,00E+08
7,00E+08	8,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	8,00E+08	8,00E+08
6,00E+08	9,00E+08	1,00E+09	1,00E+09	9,00E+08	1,00E+09	9,00E+08	9,00E+08
7,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08
7,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08	8,00E+08	8,00E+08
7,00E+08	1,00E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09
7,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	7,00E+08	8,00E+08
6,00E+08	7,00E+08	8,00E+08	7,00E+08	8,00E+08	7,00E+08	7,00E+08	7,00E+08

6,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08
6,00E+08	1,00E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09	1,00E+09
7,00E+08	8,00E+08	8,00E+08	8,00E+08	9,00E+08	8,00E+08	7,00E+08	7,00E+08
6,00E+08	7,00E+08	7,00E+08	8,00E+08	7,00E+08	7,00E+08	7,00E+08	7,00E+08
7,00E+08	9,00E+08	1,00E+09	1,00E+09	1,00E+09	1,00E+09	9,00E+08	9,00E+08
6,00E+08	7,00E+08	7,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08
6,00E+08	1,00E+09	8,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08	8,00E+08
6,00E+08	9,00E+08	9,00E+08	9,00E+08	1,00E+09	9,00E+08	8,00E+08	9,00E+08
8,00E+08	1,10E+09	1,10E+09	1,10E+09	1,00E+09	1,00E+09	1,10E+09	9,00E+08
7,00E+08	7,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	7,00E+08
8,00E+08	1,00E+09	1,10E+09	1,10E+09	1,00E+09	1,00E+09	1,00E+09	9,00E+08
7,00E+08	1,00E+09	1,00E+09	1,00E+09	1,00E+09	9,00E+08	9,00E+08	9,00E+08
6,00E+08	9,00E+08	9,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	7,00E+08
7,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08
6,00E+08	1,00E+09	1,00E+09	1,10E+09	1,00E+09	1,00E+09	1,00E+09	9,00E+08
6,00E+08	1,00E+09	1,00E+09	1,00E+09	1,00E+09	9,00E+08	1,00E+09	1,00E+09
7,00E+08	9,00E+08	1,10E+09	1,10E+09	1,00E+09	1,00E+09	1,00E+09	9,00E+08
8,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	7,00E+08	8,00E+08	7,00E+08
7,00E+08	1,00E+09	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08
7,00E+08	9,00E+08	8,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08
7,00E+08	1,00E+09	1,10E+09	1,00E+09	1,00E+09	8,00E+08	9,00E+08	9,00E+08
6,00E+08	1,00E+09	1,10E+09	1,00E+09	1,00E+09	9,00E+08	1,00E+09	1,00E+09
7,00E+08	8,00E+08	9,00E+08	8,00E+08	9,00E+08	9,00E+08	8,00E+08	8,00E+08
6,00E+08	1,10E+09	9,00E+08	9,00E+08	8,00E+08	9,00E+08	9,00E+08	9,00E+08
6,00E+08	7,00E+08	8,00E+08	8,00E+08	7,00E+08	8,00E+08	7,00E+08	8,00E+08
6,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08
7,00E+08	8,00E+08	9,00E+08	8,00E+08	8,00E+08	7,00E+08	8,00E+08	8,00E+08
6,00E+08	9,00E+08	1,00E+09	1,00E+09	8,00E+08	9,00E+08	7,00E+08	8,00E+08
7,00E+08	1,00E+09	1,00E+09	1,10E+09	1,10E+09	1,10E+09	1,00E+09	1,00E+09
7,00E+08	8,00E+08	9,00E+08	9,00E+08	8,00E+08	8,00E+08	9,00E+08	8,00E+08
7,00E+08	8,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	1,00E+09
6,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08	8,00E+08	8,00E+08	8,00E+08
7,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08
7,00E+08	9,00E+08	1,10E+09	9,00E+08	9,00E+08	9,00E+08	9,00E+08	1,00E+09
7,00E+08	1,10E+09	1,00E+09	1,10E+09	9,00E+08	1,00E+09	1,00E+09	1,00E+09
7,00E+08	9,00E+08	9,00E+08	9,00E+08	1,00E+09	8,00E+08	9,00E+08	9,00E+08
6,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	9,00E+08	9,00E+08	8,00E+08
7,00E+08	1,00E+09	9,00E+08	9,00E+08	1,00E+09	1,00E+09	9,00E+08	9,00E+08
6,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08	7,00E+08	8,00E+08	7,00E+08
7,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	1,00E+09	9,00E+08
6,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08
6,00E+08	8,00E+08	9,00E+08	9,00E+08	9,00E+08	9,00E+08	1,00E+09	9,00E+08
6,00E+08	7,00E+08	7,00E+08	7,00E+08	8,00E+08	6,00E+08	6,00E+08	8,00E+08
7,00E+08	9,00E+08	9,00E+08	9,00E+08	8,00E+08	8,00E+08	8,00E+08	8,00E+08
7,00E+08	9,00E+08	9,00E+08	9,00E+08	1,00E+09	9,00E+08	9,00E+08	9,00E+08
6,00E+08	1,00E+09	9,00E+08	1,00E+09	1,00E+09	1,00E+09	9,00E+08	1,00E+09
6,00E+08	9,00E+08	1,00E+09	1,00E+09	9,00E+08	1,00E+09	9,00E+08	9,00E+08
7,00E+08	9,00E+08	8,00E+08	8,00E+08	9,00E+08	8,00E+08	9,00E+08	8,00E+08
6,00E+08	1,10E+09	1,00E+09	1,10E+09	1,10E+09	1,00E+09	1,00E+09	9,00E+08
6,00E+08	7,00E+08	6,00E+08	6,00E+08	6,00E+08	6,00E+08	6,00E+08	6,00E+08
7,00E+08	8,00E+08	8,00E+08	9,00E+08	8,00E+08	8,00E+08	8,00E+08	7,00E+08
6,00E+08	7,00E+08	7,00E+08	7,00E+08	7,00E+08	6,00E+08	6,00E+08	6,00E+08

CROSSOVER_TWOPT

1.000.000

```
0.000000
FITNESS_RAW
POPREPL_BEST
SELECT_TOURNAMENT
SCHWEFEL
                          100
                           26
HASTA LA
CONVERGENCIA
                           10
                            1
                            8
0-ACTIVO 1-ACTIVO 2-ACTIVO 3-ACTIVO 4-ACTIVO 5-ACTIVO 6-ACTIVO 7-ACTIVO
0-200 1-400 2-600 3-800 4-Siempre 5-Siempre 6-Siempre 7-Siempre
```

Tras importar esta información, para cada fichero, la tratamos de la forma siguiente:

```
    'En la celda A116, calculamos la media de las 100 filas de la primera
columna
    Cells(116, 1).Activate
    ActiveCell.FormulaR1C1 = "=AVERAGE(R[-115]C:R[-16]C)"
    Range("A116").Select
    'rellenamos hasta la última columna con datos con las medias de sus filas
correspondientes
    Selection.AutoFill Destination:=Range("A116:H116"), Type:=xlFillDefault
    Range("A116:H116").Select
    Selection.Copy
    Sheets("Hoja2").Activate
    'En la segunda hoja, en la fila correspondiente al archivo actual (en
número) pegamos los valores
    Cells(i, 1).Select
    Selection.PasteSpecial Paste:=xlPasteValues, Operation:=xlNone, SkipBlanks
_
        :=False, Transpose:=False
    Hoja1.Cells.Clear
```