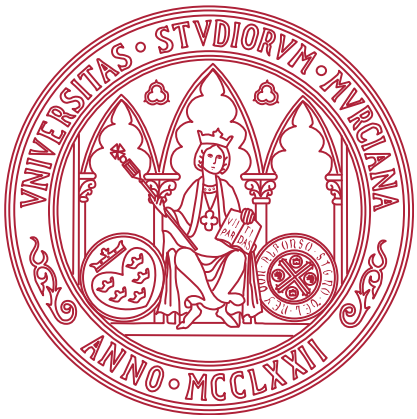
UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

TRABAJO FIN DE GRADO

ALGORITMOS PARALELOS PARA LA DETERMINACIÓN DE MODELOS DE SERIES TEMPORALES MULTIVARIANTES

JUAN ANTONIO RODRÍGUEZ LORENTE



Trabajo dirigido por

Antonio Javier Cuenca Muñoz
Domingo Giménez Cánovas

Curso 2018-2019

Declaración firmada sobre la originalidad del trabajo

D. Juan Antonio Rodríguez Lorente, con DNI 48854839X, estudiante de la titulación de Grado en Ingeniería Informática de la Universidad de Murcia y autor del TFG titulado "Algoritmos paralelos para la determinación de modelos de series temporales multivariantes".

De acuerdo con el Reglamento por el que se regulan los Trabajos Fin de Grado y de Fin de Máster en la Universidad de Murcia (aprobado C. de Gob. 30-04-2015, modificado 22-04-2016 y 28-09-2018), así como la normativa interna para la oferta, asignación, elaboración y defensa de los Trabajos Fin de Grado y Fin de Máster de las titulaciones impartidas en la Facultad de Informática de la Universidad de Murcia (aprobada en Junta de Facultad 27-11-2015).

DECLARO:

Que el Trabajo Fin de Grado presentado para su evaluación es original y de elaboración personal. Todas las fuentes utilizadas han sido debidamente citadas. Así mismo, declaro que no incumple ningún contrato de confidencialidad, ni viola ningún derecho de propiedad intelectual e industrial.

Murcia, a 24 de junio de 2019

Fdo.: Juan Antonio Rodríguez Lorente
Autor del TFG

Resumen

El propósito de este TFG es desarrollar algoritmos para intentar modelar series temporales multivariantes con resultados satisfactorios en el menor tiempo posible. Para ello, se presentará un algoritmo genético simple.

Este algoritmo presenta varios inconvenientes: ser adecuado solo para problemas pequeños y no aprovechar al máximo los recursos computacionales que tenemos disponibles en la actualidad.

Para paliar estos inconvenientes se propone hacer distintas versiones paralelas del algoritmo genético utilizando los entornos de paralelismo *OpenMP* y *MPI*. De esta forma, habrá tres versiones paralelas: una versión de memoria compartida (OpenMP), otra de paso de mensajes (MPI) y una versión híbrida (OpenMP+MPI).

La idea de estos algoritmos paralelos es dividir el trabajo utilizando los hilos / procesos disponibles. Por ejemplo, si tenemos una población de 1000 individuos y 2 hilos disponibles, cada hilo trabajará con una subpoblación de 500 individuos.

Estas versiones paralelas también presentan un inconveniente. Utilizando el ejemplo anterior, cada hilo o proceso va a trabajar con unas subpoblaciones más pequeñas que la población original. Así, es posible que el algoritmo paralelo se centre en zonas locales del espacio de búsqueda, obteniendo óptimos locales en lugar de óptimos globales ya que esas subpoblaciones están aisladas. Para corregir este problema se incluirán migraciones cada cierto tiempo para que estas subpoblaciones se combinen, es decir, una subpoblación integrará individuos de otras subpoblaciones.

Estos algoritmos tendrán varios parámetros configurables, como el tamaño de población, la probabilidad de cruce y mutación y la frecuencia de migración. Para determinar los mejores valores de estos parámetros se realizarán distintos *experimentos*. Para la realización de estos experimentos se creará un problema fijo aleatorio para posibilitar un mejor análisis de la calidad de las soluciones.

Además, estos experimentos se realizarán en dos sistemas de distintas características para obtener conclusiones generales. Lo que se busca en las versiones paralelas, además de reducir el tiempo de ejecución, es obtener mejores soluciones ejecutando el mismo tiempo en secuencial y en paralelo, por lo que estos experimentos van a incluir también análisis de tiempo, de fitness y de migraciones utilizando diferentes hilos, procesos y tamaños de población. Para una mayor fiabilidad en los resultados, se realizarán promedios de las ejecuciones realizadas.

A partir del algoritmo genético simple, se introducirán una serie de *mejoras* para intentar mejorar aún más la calidad de las soluciones y reducir el tiempo de ejecución. Los beneficios de estas mejoras se analizarán de forma experimental. También se va a incluir

un estudio experimental para estudiar y explotar la heterogeneidad del clúster Heterosolar.

Por último, y teniendo en cuenta los experimentos y las distintas mejoras, se aplicarán los métodos desarrollados a series de datos de problemas reales.

Extended Abstract

The purpose of this TFG is to develop algorithms to model multivariate time series, with satisfactory results in the shortest possible time. A simple genetic algorithm is initially presented.

This algorithm has several drawbacks: it is only suitable for small problems and does not take full advantage of the computational resources currently available.

In order to alleviate these problems, it is proposed to make different parallel versions of the genetic algorithm using the parallelism environments *OpenMP* and *MPI*. In this way, there are a total of three parallel versions: a shared memory version (OpenMP), a message passing version (MPI) and a hybrid version (OpenMP+MPI).

When designing the parallel versions, a coarse-grained approach is followed, specifically, an island model. The general idea is that each thread or process works in a smaller population than the original. This small population is called a sub-population. For example, if we have a population of 1000 individuals and 2 threads, each thread will work with a sub-population of 500 individuals.

These parallel versions also present a drawback. Using the above example, each thread or process will work with smaller sub-populations than the original population. Thus, it is possible for the parallel algorithm to focus on local areas of the search space, obtaining local optimums rather than global optimums as these sub-populations are isolated.

To correct this problem, migrations are included from time to time (taking special care in their frequency so that communications do not dominate the computation) so that these sub-populations are combined in such a way that each sub-population integrates individuals from other sub-populations. With this approach, the Speed-Up using p processors could be p, although in practice it will be lower due to the synchronizations and communications.

These algorithms have several configurable parameters, such as population size, crossing and mutation probability, and migration frequency. In order to determine the best values for these parameters, several *experiments* are carried out.

For these experiments, a random fixed problem has been created to enable a better analysis of the quality of the solutions. In addition, these experiments have been performed on two computer systems with different characteristics in order to obtain general conclusions. The parallel versions can be used both to reduce the execution time or to obtain better solutions running the same time in sequential and parallel.

Thus, the experiments include analysis of time, fitness and migrations using different threads, processes and population sizes. For greater reliability in the results, averages have

been taken of groups of executions. From the experiments, the following conclusions have been obtained:

1. The results obtained from experiments using different crossing probabilities show the tendency that the more crossings are made the better solutions are obtained. This is logical as more individuals are evaluated. On the other hand, results using different mutation probabilities show the tendency that the fewer mutations the better the solutions.

2. From time and migration experiments, several conclusions have been drawn. One conclusion is that time increases linearly with population size. The other conclusion is that the parallel versions obtain, using p processors, almost a Speed-Up of factor p. Depending on the number of cores, Speed-Up fluctuates due to migrations. In other words, when more processors are used the impact of migrations on the execution time is more visible. This is not very important since, for a fixed time, better solutions are obtained using a high migration frequency.

3. For all versions of the genetic algorithm, the results obtained from fitness experiments show that a population size between 30 and 1000 is sufficient to obtain satisfactory results. From 1000 onward the quality of the solutions worsens very quickly. It is also concluded that by using more threads / processes the quality of the solutions improves significantly in the same time compared to the sequential solution.

From the simple genetic algorithm, a series of improvements have been introduced to further improve the quality of the solutions and reduce the execution time. The benefits of these improvements have been analysed experimentally. An experimental study of the exploitation of heterogeneity in a heterogeneous cluster is also included.

Two types of modifications to the basic genetic algorithm are considered: of the functions of the simple genetic algorithm, and of the computational part of the fitness calculation. The *improvements* are as follows:

- *Crossover operator*. Three new operators have been implemented:

  Crossing by two points: two crossing points are chosen randomly and both individuals exchange their model elements between those two points.

  Arithmetic Crossing: if we call i_1 and i_2 to the individuals to be crossed, the two descendants $1/3 * i_1 + 2/3 * i_2$ and $2/3 * i_1 + 1/3 * i_2$ are considered, so that the two descendants are in the line connecting the two ascendants.

  Random Crossing: for each crossing of individuals, the crossing operator to be applied is chosen randomly.

- *Elitism*. The idea is to ensure that the quality of the solution does not diminish from one generation to the next. In this way, for each new generation, a set of individuals is formed with the individuals of the old population, the crossed individuals and the

mutated individuals. Once the fitness of all these individuals has been calculated, they are ordered to include the best individuals in the new population.

- *Selection*. Three new selection methods have been implemented:

  Selection roulette. The idea is that the best individuals are more likely to be chosen. A roulette wheel is built in which each of the portions represents an individual in proportion to their fitness. In this way, the best individuals get the best portions. This process is done in 4 steps.

  First the sum of all the fitness of each individual is calculated. We call this sum $fitnessTotal$.

  In a second step, being a minimization problem, we calculate the accumulated sum of subtracting, for each individual, $fitnessTotal$ and his fitness. We call this sum $sumaTotal$. This is important because an individual with less fitness will have to add more than another with more fitness.

  In step 3 a random number is generated between 0 and $sumaTotal$. Finally, the population is traveled accumulating, for each individual, $fitnessTotal$ minus his fitness. When the value is greater than or equal to the number generated in step 3, the individual is selected.

  Selection ranking. Individuals are ordered regarding their fitness. If we have n individuals, the individual with the worst fitness is assigned a 1 and the individual with the best fitness n. The process is very similar to the roulette selection process.

  In a first step $nTotal$ is calculated using the formula $n * (n + 1)/2$. Then a random number is generated between 0 and $nTotal$.

  Finally, the ordered population is scrolled accumulating the value of each individual. When the sum is greater than or equal to the number generated in step 2, the individual is selected.

  Random Selection. The selection function is chosen randomly for each selection to be made. In this way, you can use tournament, roulette or ranking selection. Because roulette selection malfunctions when there are large differences between fitness of individuals in the population, it will not be used in the implementation without extension (no elitism).

- *Local Search*. To try to improve the fitness even more, a hybrid method has been designed combining the genetic algorithm with local search. The local search improvement is applied to each individual when initializing a population and after forming each new generation.

  The idea is to analyze the vicinity of the elements of the model of the individual a certain number of steps. A neighborhood of two individuals is considered. In each step, the position of the individual (the entry of the model) to be updated is chosen randomly.

The first individual is obtained adding to the current value of the element a random value in the interval [0,001, $percentage * value/100$]. Thus, the value will be modified a minimum of $0{,}001$ and a maximum percentage of its value.

The second individual is obtained in the same way, but subtracting. For these two individuals the fitness is recalculated. If one of these two new individuals improves the original individual, it will be used for the next step; if not, the original individual will continue to be used. Calculating an individual's fitness is very expensive and this improvement performs many calculations.

In order to reduce the execution time of the Local Search, the function to update the fitness of an individual has been modified. The optimization is based on having to perform fewer calculations by modifying only one value of the individual. The order of the execution time in the fitness calculation reduces in that way from $O(n^3)$ to $O(n)$, which can have an important impact on the algorithm execution time in methods that include local search, especially with large neighborhoods.

- *BLAS library*. BLAS has been used to try to reduce the execution time of the fitness calculation. BLAS is a library that provides low-level routines for linear algebra operations. In this way, the dgemm routine is used for matrix multiplication instead of the typical three-loops, naive version.

  BLAS defines a general interface and functionallity, and there are available implementations optimized for high performance. Some are OpenBLAS, Intel MKL and IBM ESSL. OpenBLAS has been used in this work. Its performance is very similar to the other implementations in multiplications of dense matrices.

The following *conclusions* have been drawn from the use of the different improvements implemented:

1. The best crossing operator is the random crossing because of its tendency to get better solutions in a fixed time.

2. No significant differences have been observed between the new selection methods, so a good option is to use tournament selection.

3. Hybridization with Local Search substantially improves the quality of the solutions and, thanks to the optimization of the fitness calculation for this improvement, the order of time in the calculation of the fitness of the neighbors of an element goes from $O(n^3)$ to $O(n)$, significantly reducing the execution time of the algorithm.

4. The use of the OpenBLAS library significantly reduces the execution time needed to calculate the fitness. In addition, the larger the problem, the greater the gain.

5. The benefits of using elitism are good if Local Search enhancement is not used. When using Local Search it is best not to use elitism and to use population sizes between 30 and 100.

The conclusions drawn by comparing all versions of the genetic algorithm are as follows:

1. As mentioned above and, in a fixed time, the qualities of the solutions of the parallel versions improve those of the sequential version.
2. The OpenMP, MPI and hybrid versions are very similar in all aspects, as an island scheme is used in all cases.
3. The hybrid version can be used to exploit heterogeneity in heterogeneous clusters. Experiments in the Heterosolar cluster show that in this way it is possible to improve the solutions found for a fixed execution time.

In summary, as a general conclusion, the best option is to use:

- The hybrid version using all the available nodes in the cluster.
- Local Search (and its optimization) without elitism.
- The OpenBLAS library or similar.
- Random crossing.
- Selection by tournament.
- Population sizes between 30 and 100.
- Crossing probability of 0.9.
- Mutation probability of 0.005.

There is still a margin for improvement in this work. Some future lines of work are:

- Develop GPU or Xeon Phi implementations (or use the GPU implementation developed by the Scientific Computing and Parallel Programming Group) and combine them with the MPI+OpenMP hybrid version to exploit all the computational resources of a cluster.
- Develop other metaheuristics, analyze them in a similar way as has been done for genetic algorithms, and compare the results obtained with the different metaheuristics.
- It has been shown with some experiments that the developed software can be applied to series of data of real problems. An interesting work would be to analyze with experts in different fields the use for their problems of the software here developed.

Índice general

Capítulo 1

Introducción

Este capítulo describe el problema sobre el que se va a trabajar: modelado de series temporales multivariantes. Al ser un problema de optimización, se verá cómo las técnicas metaheurísticas nos van a ser de utilidad para intentar encontrar o aproximar su solución óptima.

Además, como en el caso de grandes volúmenes de datos las necesidades de computación pueden llegar a ser altas, se considera la aplicación de técnicas de paralelismo para reducir el tiempo de ejecución y/o mejorar la solución obtenida.

También se describe el estado del arte, el plan de trabajo que se va a seguir, el entorno que se va a utilizar para realizar pruebas y experimentos, y la estructura de este documento.

1.1. Series Temporales

Con el auge del Big Data, se dispone cada vez más de grandes cantidades de datos correspondientes a una amplia diversidad de campos, como la economía o la medicina.

En algunos casos estos datos siguen una evolución temporal dando lugar a lo que se llama series temporales. El análisis de estos datos es muy importante, pudiendo proporcionar *modelos para predecir y analizar evoluciones* bajo determinadas circunstancias, como por ejemplo, la evolución de una enfermedad [1].

Hay modelos de varios tipos. Este trabajo se centra en las series temporales multivariantes.

Se dispone de una serie de parámetros de los que se han tomado valores de forma periódica, y se quiere determinar la evolución temporal de los parámetros, teniendo en cuenta que el valor de un parámetro en un instante puede depender de los valores que tuviera en instantes anteriores y que puede haber interdependencia entre parámetros, con lo que los valores de un parámetro también pueden depender temporalmente de valores de otros parámetros.

También puede haber parámetros externos al modelo cuyo valor influye en los de los parámetros internos pero que no se ven influidos por estos [1]. Por ejemplo, cuando se están tomando valores de la glucosa (parámetro interno) y se le inyecta al paciente

insulina (parámetro externo), la cantidad de insulina influirá en la de glucosa, pero no al revés.

1.2. Series Temporales Multivariantes

Para el planteamiento formal del problema, consideramos d parámetros de los que se han tomado valores en t instantes de tiempo. Los valores para el instante k, $1 \leq k \leq t$, se almacenan en un vector $y^{(k)} = \left(y_1^{(k)}, \ldots, y_d^{(k)}\right) \in R^{1\times d}$. Los t vectores de datos se pueden organizar en una matriz $Y \in R^{t\times d}$:

$$\begin{pmatrix} y_1^{(1)} & \ldots & y_d^{(1)} \\ \vdots & \ddots & \vdots \\ y_1^{(t)} & \ldots & y_d^{(t)} \end{pmatrix} \tag{1.2.1}$$

Consideramos también dependencias temporales con i instantes de tiempo anteriores. Si tuviéramos una única variable x, las dependencias se expresarían: $x_j \approx x_{j-1}a_1 + x_{j-2}a_2 + \ldots + x_{j-i}a_i + a_0$, donde x_j representa el valor de x en el instante j, y a_l, con $0 \leq l \leq i$, son los valores a determinar para que la diferencia entre los valores experimentales de x y los obtenidos con el modelo sean mínimos.

Los a_l, $1 \leq l \leq i$, representan las dependencias con instantes anteriores, mientras a_0 representa un valor inicial. La ecuación se puede completar con una variable externa al modelo, z, que influye en los valores de x pero no se ve influenciada por ellos: $x_j \approx x_{j-1}a_1 + x_{j-2}a_2 + \ldots + x_{j-i}a_i + z_{j-1}b_1 + z_{j-2}b_2 + \ldots + z_{j-k}b_k + a_0$, donde los valores b_l habrá que determinarlos también para minimizar la diferencia de los datos experimentales con los del modelo, y el número de instantes anteriores de tiempo a los que se expande la dependencia de x del factor externo z (k) puede ser distinto del de dependencia de instantes anteriores de x (i).

Si generalizamos a varios datos tal como hemos dicho, tenemos la dependencia de vectores de datos de un instante j en función de los vectores de datos anteriores y de los factores externos en la forma:

$$\begin{aligned} y^{(j)} \approx\ & y^{(j-1)}A_1 + y^{(j-2)}A_2 + \ldots + y^{(j-i)}A_i + \\ & z^{(j-1)}B_1 + z^{(j-2)}B_2 + \ldots + z^{(j-k)}B_k + a \end{aligned} \tag{1.2.2}$$

donde A_l, $1 \leq l \leq i$, son matrices de dimensión $d \times d$ que representan la dependencia de los datos con los valores de l instantes anteriores; B_l, $1 \leq l \leq k$, son matrices de tamaño $e \times d$, con e el número de factores externos considerados; $z^{(l)}$, con $1 \leq l \leq t$, almacenan los valores de los parámetros externos para los t instantes de tiempo, y al tener e factores externos, las dimensiones de cada vector z son $1 \times e$; y a es un vector $1 \times d$.

De forma similar a la representación de los vectores y en una matriz, los t vectores z de valores de parámetros externos se pueden organizar en una matriz $Z \in R^{t \times e}$:

$$\begin{pmatrix} z_1^{(1)} & \dots & z_e^{(1)} \\ \vdots & \ddots & \vdots \\ z_1^{(t)} & \dots & z_e^{(t)} \end{pmatrix} \tag{1.2.3}$$

Para poner las dependencias temporales en forma matricial, como hay dependencias de i instantes anteriores y suponiendo que el número de dependencias temporales de parámetros externos es menor que el de internos ($k \leq i$), consideramos las dependencias de la ecuación 1.2.2 para j tal que $i+1 \leq j \leq t$, con lo que tomamos las filas de la $i+1$ a la t de la matriz Y, lo que, usando la notación clásica de Matlab [2], se representa como $Y(i+1:t,:)$ (filas de la $i+1$ a la t y todas las columnas). La ecuación queda:

$$\begin{gathered} Y(i+1:t,:) \approx \\ Y(i:t-1,:)A_1 + Y(i-1:t-2,:)A_2 + \ldots + Y(1:t-i,:)A_i + \\ Z(i:t-1,:)B_1 + Z(i-1:t-2,:)B_2 + \ldots + Z(i-k+1:t-k,:)B_k + A_0 \end{gathered} \tag{1.2.4}$$

donde A_0 es una matriz con el vector a en todas las filas. Y en forma matricial se representa:

$$\begin{gathered} Y(i+1:t,:) \approx \\ [Y(i:t-1,:)|\ldots|Y(1:t-i,:)|Z(i:t-1,:)|\ldots|Z(i-k+1:t-k,:)|1] \quad * \\ \begin{bmatrix} A_1 \\ \vdots \\ A_i \\ B_1 \\ \vdots \\ B_k \\ a \end{bmatrix} \end{gathered} \tag{1.2.5}$$

donde 1 representa un vector columna con todos sus valores a 1.

1.3. Resolución de Modelos de Series Temporales Multivariantes

El problema que se considera en este trabajo tiene las siguientes características:

- Se dispone de los datos de los t instantes de tiempo, con valores en cada instante de tiempo para los d parámetros internos y para los e parámetros externos.
- Se dispone de la longitud de las dependencias temporales de parámetros internos (i) y externos (k).

- El modelo contiene términos independientes.

De esta forma, el problema consistirá en, dadas las matrices Y y Z, encontrar el modelo (dado por la matriz formada por las matrices A_l, B_l y a) con el que se minimiza la diferencia entre los datos reales y los obtenidos con el modelo. Si llamamos $\hat{Y}$ a la matriz $Y(i+1:t,:)$ en la ecuación 1.2.5, $\hat{X}$ a la matriz formada por submatrices de Y, Z y el vector 1, y $\hat{A}$ a la matriz formada por matrices A_l, B_l y el vector a, se trata de resolver el problema de mínimos cuadrados [1]:

$$min_{\hat{A}} \left|\left| \hat{Y} - \hat{X}\hat{A} \right|\right| \tag{1.3.6}$$

La norma será la raíz cuadrada de la suma de los cuadrados de las diferencias entre elementos de $\hat{Y}$ y $\hat{X}\hat{A}$.

Este trabajo considera dos tipos de problemas que se diferencian en los posibles valores que pueden tomar los datos de $\hat{A}$:

- En una primera versión, los datos de la matriz $\hat{A}$ podrán tomar valores de un conjunto finito de números reales.
- En una segunda versión, más realista, los datos de la matriz $\hat{A}$ podrán tomar valores de un intervalo de reales.

1.4. Estado del Arte

Actualmente existen herramientas para resolver modelos de Series Temporales Multivariantes. A continuación se mencionan algunas:

- Paquete vars [3] para R.
- Librería PyFlux [4] para Python.
- Stata [5].
- EViews [6].

Estas herramientas tienen el inconveniente de ser *adecuadas solo para problemas pequeños*, ya que no son capaces de aprovechar los recursos computaciones que los sistemas de alto rendimiento actuales ofrecen.

Para intentar encontrar una solución óptima o aproximada en problemas de optimización (como es este problema) se pueden *utilizar técnicas metaheurísticas* [7].

Estas técnicas son procedimientos iterativos que exploran el espacio de búsqueda de posibles soluciones, que en este caso viene dado por los posibles valores de la matriz $\hat{A}$.

De esta forma, se van analizando posibles soluciones obteniendo su fitness o bondad.

Concretamente, el fitness de una solución será la norma, como se vio en la ecuación 1.3.6. Las soluciones se van mejorando según la técnica utilizada y, por último, se devuelve la mejor. En este problema, al ser de minimización, la mejor solución será la que tenga menor fitness.

El uso de estas técnicas metaheurísticas se puede combinar con la computación paralela [8]. La *computación paralela* persigue aprovechar al máximo los recursos computacionales para resolver problemas complejos y/o costosos de forma eficiente. Hay diversos tipos de problemas en los que la computación paralela es de gran utilidad [9]:

- En problemas de gran desafío, donde hay un alto coste computacional. Ejemplos: problemas de biología y medicina, diseño de fármacos, etc.
- En problemas de gran dimensión. Ejemplos: simulaciones del clima, búsqueda de cadenas con bases de datos muy grandes en bioinformática, etc.
- De tiempo real.

Este problema se puede considerar de gran desafío, ya que hay que explorar un espacio de busqueda de posibles soluciones enorme, y, además, el coste para calcular el fitness puede ser también de gran dimensión si el problema es muy grande.

La paralelización de la metaheurística en este trabajo persigue dos objetivos: reducir el tiempo de ejecución y obtener mejores soluciones ejecutando el mismo tiempo en secuencial y en paralelo.

En el grupo de Computación Científica y Programación Paralela de la Facultad de Informática de la Universidad de Murcia se han implementado para este problema diversas metaheurísticas paralelas utilizando memoria compartida y un enfoque de grano fino [10].

También se ha implementado una versión para GPU. En este Trabajo Fin de Grado se utiliza un modelo de islas (un enfoque de grano grueso) y, además de usar memoria compartida, se utiliza también paso de mensajes y la combinación de ambas técnicas. De esta forma se consigue combinar más elementos computacionales (nodos en un clúster) en la solución del problema, haciendo posible el abordar problemas mayores en sistemas computacionales más complejos.

1.5. Entorno de Computación

Para paralelizar la técnica metaheurística se utilizarán dos entornos de paralelismo: OpenMP y MPI.

OpenMP proporciona una interfaz simple para la programación multiproceso de memoria compartida. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran y ofrece un conjunto de directivas, rutinas y variables de entorno que influyen en el tiempo de ejecución [11].

MPI es una librería de comunicación entre los nodos que ejecutan un programa, y proporciona sincronización entre procesos y exclusión mutua. También proporciona un conjunto de rutinas que pueden ser utilizadas en programas escritos en C, C++, Fortran y Ada que garantiza programas portables y rápidos [12].

Para hacer pruebas y experimentos utilizamos el clúster heterogéneo denominado *Heterosolar* (Figura 1.1), del Grupo de Computación Científica y Programación Paralela de la Facultad de Informática de la Universidad de Murcia. Está formado por 5 nodos de cómputo más una entrada al clúster desde una máquina virtual y está ubicado en las instalaciones de ÁTICA[1]. El sistema está interconectado por una red Gigabit Ethernet y está compuesto por los siguiente nodos:

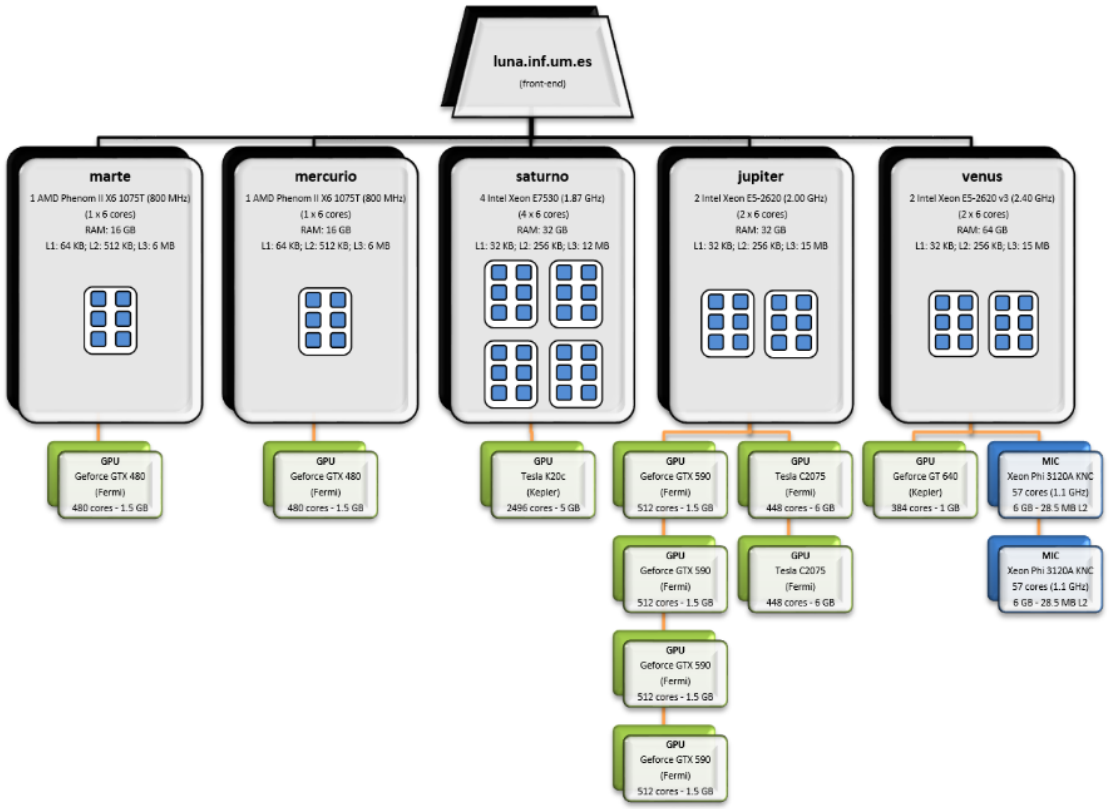


Figura 1.1: Esquema del clúster Heterosolar.

[1] Área de Tecnologías de la Información y las Comunicaciones Aplicadas de la Universidad de Murcia

- *Luna*: Máquina virtual que da acceso al sistema. La compilación de la aplicación se hará en este nodo, así como la carga de módulos. Un ejemplo de módulo es openmpi, para compilar una aplicación que incluya código MPI. Luna también dispone del gestor de recursos Torque para el envío de trabajos. Esto garantiza que el trabajo a ejecutar va a disponer de forma exclusiva de los recursos requeridos.

- *Marte*: Dispone de una CPU AMD Phenom II X6 1075T (hexa-core) a 3.00 GHz y arquitectura x86-64, 16 GB de memoria RAM, caches L1 y L2 privadas de 64 KB y 512 KB, respectivamente, y una cache L3 de 6 MB compartida por todos los cores. Incluye una GPU NVIDIA GeForce GTX 480 (Fermi) con 1536 MB de Memoria Global y 480 CUDA cores (15 Streaming Multiprocessors y 32 Streaming Processors por SM).

- *Mercurio*: Sistema idéntico a Marte.

- *Saturno*: Nodo de cómputo con 4 CPU Intel Xeon E7530 (hexa-core) a 1.87 GHz y arquitectura x86-64. Tiene el hyperthreading habilitado, por lo que cada core soporta 2 hilos de procesamiento, lo que hace un total de 48 cores lógicos. Dispone de una arquitectura de memoria tipo NUMA con 32 GB de memoria RAM, caches L1 y L2 privadas por core de 32 KB y 256 KB, respectivamente, y una cache L3 de 12 MB compartida por todos los cores de cada socket (NUMA Node). A su vez, dispone de una GPU NVIDIA Tesla K20c (Kepler) con 4800 MB de Memoria Global y 2496 CUDA Cores (13 Streaming Multiprocessors y 192 Streaming Processors por SM).

- *Jupiter*: Nodo de cómputo con 2 CPU Intel Xeon E5-2620 (hexa-core) a 2.00 GHz y arquitectura x86-64. Con hyperthreading habilitado, con lo que cada core soporta 2 hilos de procesamiento, lo que hace un total de 24 cores lógicos. Dispone de una arquitectura de memoria tipo NUMA con 32 GB de memoria RAM, caches L1 y L2 privadas por core de 32 KB y 256 KB, respectivamente, y una cache L3 de 15 MB compartida por todos los cores de cada socket (NUMA Node). Asimismo, dispone de seis GPU: dos NVIDIA Tesla C2075 (Fermi) con 5375 MB de Memoria Global y 448 CUDA cores (14 Streaming Multiprocessors y 32 Streaming Processors por SM) y cuatro GPUs agrupadas en dos tarjetas, cada una con dos NVIDIA GeForce GTX 590 (Fermi) con 1536 MB de Memoria Global y 512 CUDA cores (16 Streaming Multiprocessors y 32 Streaming Processors por SM).

- *Venus*: Nodo de cómputo con 2 CPU Intel Xeon E5-2620 (hexa-core) a 2.40 GHz y arquitectura x86-64. Dispone de una arquitectura de memoria tipo NUMA con 64 GB de memoria RAM, caches L1 y L2 privadas por core de 32 KB y 256 KB, respectivamente, y una cache L3 de 15 MB compartida por todos los cores de cada socket (NUMA-Node). A su vez, dispone de una GPU NVIDIA GeForce GT 640 (Kepler) con 1024 MB de Memoria Global y 384 CUDA cores (2 Streaming Multiprocessors y 192 Streaming Processors por SM), y dos coprocesadores Intel Xeon Phi 3120A Knights Corner con 57 cores físicos (228 lógicos) a 1.10 GHz, 6 GB de

memoria, caches L1 y L2 privadas por core de 32 KB y 28.50 MB, respectivamente, y VPUs de 512 bits.

En este trabajo no se han utilizado los coprocesadores en los distintos nodos (GPU o Xeon Phi), pero ya hemos comentado que dentro del grupo de investigación se han desarrollado implementaciones para GPU, por lo que los esquemas distribuidos que desarrollamos en este trabajo se podrían combinar con esquemas para coprocesadores, permitiendo de este modo la explotación de todos los recursos computacionales que ofrece el clúster.

1.6. Plan de Trabajo

Este trabajo tiene los siguientes *objetivos*:

- Implementar una técnica metaheurística simple para intentar encontrar una solución óptima o aproximada al problema.
- Crear versiones paralelas de la técnica metaheurística implementada para poder aprovechar al máximo los recursos computacionales que tenemos a nuestra disposición y reducir el tiempo de ejecución, sin descuidar la calidad de las soluciones.
- Analizar el comportamiento de las distintas versiones a través de experimentos utilizando diferentes configuraciones.

Para alcanzar estos objetivos se seguirá la siguiente *metodología*:

1. Se implementará un *algoritmo genético simple* por ser el método de resolución de problemas de optimización más popular [13]. La implementación se hará en C++.
2. Se implementarán las *versiones paralelas* del algoritmo genético simple usando OpenMP y MPI. Todas las versiones paralelas se implementarán en C++. De esta forma, habrá tres versiones paralelas: una versión OpenMP, una versión MPI y una versión híbrida (OpenMP + MPI).

   Se seguirá un enfoque de grano grueso. Más en concreto, se utilizará un modelo de islas (Figura 1.2).

   La idea general es que cada hilo o proceso trabaje en una población más pequeña que la original. A esta población reducida se le llamará subpoblación.

   También será necesario hacer migraciones. En las migraciones cada subpoblación integra individuos de las otras subpoblaciones. Con esto se pretende, teniendo especial cuidado en su frecuencia para que las comunicaciones no dominen a la computación, alcanzar un óptimo global y no local, ya que las subpoblaciones van a estar aisladas.

   Con este enfoque, el Speed-Up usando p procesadores podría ser de un factor p, aunque en la práctica será algo menor por las comunicaciones [14].

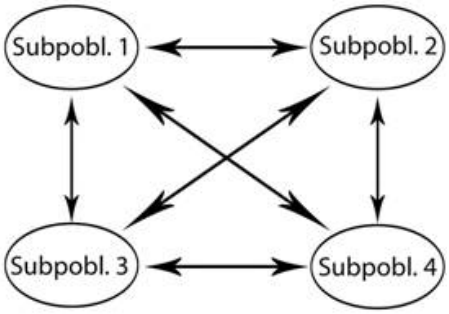


Figura 1.2: Modelo de islas.

3. Se harán *experimentos* para ver si hay alguna tendencia en las probabilidades de cruce y mutación, así como comparaciones de tiempo y fitness con distintos tamaños de población.

   También se harán experimentos para ver como influyen las migraciones en las versiones paralelas. Los experimentos se realizarán en varios sistemas computacionales de distintas características, de manera que se posibilite obtener conclusiones generales independientes del sistema.

4. A partir de las implementaciones iniciales y teniendo en cuenta los resultados de los experimentos, *se propondrán mejoras* para reducir el tiempo de ejecución o mejorar las soluciones obtenidas. Los beneficios obtenidos con las mejoras propuestas se analizarán también experimentalmente.

   También se incluirá un estudio experimental para estudiar y explotar la heterogeneidad del clúster Heterosolar.

1.7. Estructura del TFG

En los distintos capítulos del trabajo se detallará cómo se han realizado las tareas expuestas en cada uno de los puntos de la metodología.

En el capítulo 2 se describirá, en primer lugar, un *algoritmo genético secuencial* para la resolución del problema planteado y, seguidamente, se irán mostrando varias propuestas de *paralelización* de este algoritmo, usando tanto OpenMP como MPI.

En el capítulo 3 se mostrará el *estudio experimental* que se ha llevado a cabo con las diferentes propuestas descritas en el capítulo anterior.

Tras ello, en el capítulo 4 se describirán y analizarán una serie de *mejoras* para las distintas versiones del algoritmo propuesto.

En el capítulo 5 se aplicarán los métodos desarrollados a series de datos de *problemas reales*.

Finalmente, en el capítulo 6 se analizarán las *conclusiones*, estableciendo una serie de líneas de trabajo futuras.

Capítulo 2

Implementaciones de un Algoritmo Genético

Este capítulo describe un algoritmo genético base para la resolución del problema, así como distintas versiones paralelas del mismo usando OpenMP, MPI y MPI+OpenMP.

2.1. Solución Secuencial

Para implementar las distintas versiones del algoritmo genético se han utilizado las siguientes clases:

1. Util.

   Esta clase contiene los siguientes métodos y funciones para facilitar el trabajo en el resto del programa:

   - *mseconds*, calcula el tiempo que tarda en ejecutarse el programa.
   - *imprimirA*, imprime un modelo.
   - *setValor*, establece un valor de manera aleatoria desde un conjunto o intervalo, según esté configurado el programa.
   - *getRealAleatorio*, genera un valor real aleatorio entre un intervalo.
   - *getEnteroAleatorio*, genera un valor entero aleatorio entre un intervalo.
   - *setFitness*, establece un valor de fitness a un individuo. Llamamos Y a la matriz con los t vectores de datos, $Ytrabajo$ a Y a partir de los vectores iniciales, X a la matriz formada por las submatrices de Y, Z y el vector 1, $individuo$ a un modelo generado por el algoritmo genético y a *tamX* al tamaño de una fila de la matriz X *(ni*d + k*e + 1)*.

     En el algoritmo 1 se muestra cómo se realiza el cálculo. Para cada instante de tiempo i (menos los instantes iniciales) y para cada variable j de cada instante de tiempo i, se hace la multiplicación de la fila de X (comenzando en la posición que le corresponde, *i*tamX*) por la columna correspondiente del individuo (en función de la variable j que se está calculando).

     Conforme se van haciendo las multiplicaciones, se va almacenando en una variable llamada *fitness* la suma de los cuadrados de las diferencias entre elementos de Y y la multiplicación realizada. Por último, se hace la raíz cuadrada de la variable fitness y se establece el fitness del individuo.

Algorithm 1: Cálculo del fitness de un individuo

```
1  fitness = 0.0
2  for i = 0 to t − ni do
3      for j = 0 to d do
4          suma = 0.0
5          for z = 0 to tamX do
6              suma = suma + X[i ∗ tamX + z] ∗ individuo[z ∗ d + j]
7          end
8          fitness+ = (Ytrabajo[i ∗ d + j] − suma) ∗ (Ytrabajo[i ∗ d + j] − suma)
9      end
10 end
11 fitness = raiz (fitness)
```

- *multiplicarMat*, multiplica dos vectores.
- *copiar*, realiza la copia de un vector.

2. Problema.

   Se utiliza para generar problemas sintéticos con los que poder comprobar el correcto funcionamiento de los distintos algoritmos implementados. Los problemas se generan de manera aleatoria a través de un fichero de configuración. El fichero contiene los siguientes parámetros configurables:

   - d, indica la cantidad de parámetros internos.
   - e, indica la cantidad de parámetros externos.
   - t, indica los instantes de tiempo.
   - i, indica las dependencias con i instantes de tiempo anteriores.
   - k, indica las dependencias con k instantes de tiempo anteriores (para los parámetros externos).
   - $modoDatos$, para indicar si los datos van a estar en un conjunto o en un intervalo.
   - $nConjunto$, para indicar el tamaño del conjunto.
   - Parámetros para definir los valores del conjunto o del intervalo.
   - Parámetros para el algoritmo genético: el número de iteraciones que se van a realizar, el tamaño de población y las probabilidades de cruce y mutación. También se puede configurar la migración y el número de hilos y de procesos que se van a usar para las versiones paralelas.

   Para experimentar con datos sintéticos, una vez leída la configuración se crea el problema, para lo que se siguen los siguientes pasos:

1) Se genera el modelo A con tamaño *i * d * d + k * e * d + d* de forma aleatoria, según esté configurado el programa. Este modelo se utilizará para generar Y.
2) Se genera el modelo Z ($Zmodelo$) con tamaño *e * e* de forma aleatoria, según esté configurado el programa. Este modelo se utilizará para generar Z.
3) Se genera Z inicial con tamaño *k * e* de forma aleatoria, según esté configurado el programa.
4) Se genera el resto de Z. Cada nueva fila de Z se genera considerando la fila anterior. Para calcular cada fila i (a partir de las filas iniciales), se calcula para cada variable de la fila i la multiplicación de la fila anterior por la columna correspondiente de $Zmodelo$.
5) Se genera Y inicial con tamaño *i * d* de forma aleatoria, según esté configurado el programa.
6) Se genera el resto de Y a partir de los instantes dados. Para cada instante de tiempo i va creándose X e Y. Y se creará utilizando el modelo A generado anteriormente. Y es la matriz $Y(i+1:t,:)$ en la ecuación 1.2.5, X es la matriz formada por submatrices de Y, Z y el vector 1, y A es la matriz formada por matrices A_l, B_l y el vector a.

   Para formar cada fila de X hay que poner primero las filas de Y de i instantes anteriores, después las de Z de k instantes anteriores y por último el vector 1. Una vez tengamos el vector X para ese instante, para cada variable de Y, se multiplica el vector X por la columna correspondiente del modelo A.

3. Genético.

   Los parámetros del algoritmo genético son los siguientes:

   - $tamPoblacion$, indica el tamaño de la población.
   - $tamA$, indica el tamaño del modelo. Es el que se utiliza para generar el modelo A.
   - $tamIndividuo$, indica el tamaño de un individuo. Necesita el tamaño del modelo y otro más para el fitness.
   - $pCruce$, indica la probabilidad de cruce.
   - $pMut$, indica la probabilidad de mutación.

   De esta forma, para almacenar un individuo se usa un array de reales de tamaño $tamIndividuo$ y para almacenar una población se usa un array de reales de tamaño $tamPoblacion * tamIndividuo$.

   Al ser un problema de minimización, un individuo es mejor que otro cuando su fitness es menor. El esquema del algoritmo genético secuencial simple [15] se muestra en el algoritmo 2.

   A continuación se detallan cada una de las funciones:

Algorithm 2: Esquema del algoritmo genético secuencial

```
1  Inicializar (S,Sol)
2  while (not CondicionFin()) do
3      for tamPoblacion/2 do
4          I1 = SeleccionTorneo (S)
5          I2 = SeleccionTorneo (S)
6          Cruzar (I1,I2)
7          Mutar (I1)
8          Mutar (I2)
9          Fitness (I1)
10         Fitness (I2)
11         ActualizarSolucion (I1,Sol)
12         ActualizarSolucion (I2,Sol)
13         SS = Incluir (I1,I2)
14     end
15     S = SS
16     SS = ∅
17 end
```

- *Inicializar*: genera una solución y una población de individuos. Conforme se va generando la población se actualiza la solución si el fitness es mejor. Utiliza una función *GenerarIndividuo* para generar cada individuo.

  El modelo se forma de manera aleatoria en función de como esté configurado el programa, y se calcula su fitness.
- *CondicionFin*: se evoluciona la población un número determinado de iteraciones.
- *SeleccionTorneo*: selecciona dos individuos de forma aleatoria de una población dada y se elige el que tenga mejor fitness.
- *Cruzar*: si se da la probabilidad $pCruce$, se cruzan dos individuos. De esta forma, se elige un punto de cruce de forma aleatoria entre 0 y $tamA - 1$. Después, ambos individuos van intercambiando sus elementos del modelo desde ese punto hasta el último elemento.
- *Mutar*: muta a un individuo. Para cada elemento del modelo del individuo se comprueba si se da la probabilidad $pMut$ de forma individual. Todos los elementos seleccionados cambian su valor por otro de forma aleatoria en función de como esté configurado el programa.
- *ActualizarSolucion*: se reemplaza la solución por el individuo si su fitness es mejor que el de la solución.
- *Incluir*: copia dos nuevos individuos en la nueva población.

A continuación se detalla el funcionamiento de este algoritmo. Primero se inicializa una población de individuos y una solución aleatoria (línea 1). Después, mientras que no

se de la condición de fin, se va evolucionando la población (líneas 2 - 17). De esta forma, por un lado tenemos la antigua población, a la que en el esquema se le llama S.

Por otro lado tenemos una nueva población vacía, a la que se llama SS. Esta nueva población se va generando trabajando con pares de individuos (líneas 3 - 14). Dos nuevos individuos se generan seleccionando dos individuos de la antigua población (líneas 4 - 5). Estos dos individuos seleccionados se cruzan (línea 6) y se mutan (líneas 7 - 8). Después se calcula el fitness de los dos nuevos individuos obtenidos (líneas 9 - 10) y se actualiza la solución global Sol si es necesario (líneas 11 - 12).

Por último, estos dos nuevos individuos se incluyen en la nueva población (línea 13). Una vez generada la nueva población, se elimina la antigua población y se sustituye por esta nueva población (líneas 15 - 16). Después se vacía SS para poder evolucionar la población obtenida.

Cuando termine la ejecución, tendremos en la variable Sol el individuo con menor fitness.

Para las soluciones paralelas se incluye la función *EsquemaSecuencial*, cuyo esquema se muestra en el algoritmo 3.

El esquema es muy similar al algoritmo 2 pero con algunas diferencias. La función no tendrá la inicialización de la población (línea 1). A esta función se le tendrá que pasar el tamaño de la población y la población a evolucionar ya que en los algoritmos paralelos cada hilo o proceso trabajará en una población distinta y, además, se tendrán que hacer migraciones. Esta función tampoco tendrá el bucle principal (línea 2). Este bucle se incluirá en los algoritmos paralelos con una condición específica.

Algorithm 3: Esquema algorítmico de *EsquemaSecuencial*

```
1  for tamSubpoblacion/2 do
2      I1 = SeleccionTorneo (S,tamSubpoblacion)
3      I2 = SeleccionTorneo (S,tamSubpoblacion)
4      Cruzar (I1,I2)
5      Mutar (I1)
6      Mutar (I2)
7      Fitness (I1)
8      Fitness (I2)
9      ActualizarSolucion (I1,Sol)
10     ActualizarSolucion (I2,Sol)
11     SS = Incluir (I1,I2)
12 end
13 S = SS
14 SS = ∅
```

2.2. Solución Paralela con OpenMP

Para paralelizar la solución secuencial con OpenMP se ha seguido un esquema de islas, como se dijo en el capítulo 1.

El esquema se muestra en el algoritmo 4. A continuación se detalla el esquema. Primero se crea una solución global (línea 1). Al finalizar la ejecución tendremos la mejor solución en Sol. Después, se lanzan n hilos (línea 2). La población inicial de tamaño $tamPoblacion$ se descompone en subpoblaciones, también llamadas islas, de tamaño $tamPoblacion/n$ (en el esquema consideramos que $tamPoblacion$ es múltiplo de n por simplicidad). De esta forma, cada hilo crea su subpoblación S_i y una solución local Sol_i (línea 3).

Algorithm 4: Esquema OpenMP del algoritmo genético

```
1  GenerarIndividuo (Sol)
2  EN PARALELO en cada hilo P_i (i = 0, ..., n − 1) HACER
3  Inicializar (S_i,Sol_i,tamPoblacion/n)
4  for numGeneraciones do
5      for numIteraciones do
6          EsquemaSecuencial (S_i,Sol_i,tamPoblacion/n)
7      end
8      Barrera
9      Copiar Sol_i en el array mejores
10     Barrera
11     Integrar cada individuo (excepto Sol_i) del array mejores en S_i
12 end
13 Sincronización para actualizar Sol con Sol_i si es necesario
14 FIN PARALELO
```

Al trabajar con poblaciones más pequeñas, es posible que cada hilo busque en zonas locales del espacio de búsqueda, obteniendo mínimos locales sin acercarse al óptimo global. A esto se le llama *endogamia*, y se resuelve agrupando las iteraciones en generaciones para combinar las subpoblaciones (líneas 4 - 12). De esta forma, si, por ejemplo, hay que hacer 1000 iteraciones en total y hemos establecido en nuestra configuración que consideramos generaciones de 100 iteraciones, esto conllevará que durante la ejecución se realizarán 10 combinaciones de subpoblaciones, a razón de una cada 100 iteraciones.

Para hacer esa combinación, se ha creado un array de reales llamado $mejores$, de tamaño $n \times tamA$. La idea es que cada hilo integre en su subpoblación los mejores individuos de los otros hilos. Considerando el ejemplo anterior, cuando un hilo complete 100 iteraciones, copiará su mejor solución en el array $mejores$ (línea 9). Después, habrá una barrera, ya que un hilo no puede integrar los individuos de otros hilos hasta que estos no hayan acabado de poner su mejor solución en el array (línea 10). De esta forma, una vez que tengamos el array completo, cada hilo integra de forma aleatoria en su subpoblación

cada individuo menos el suyo (línea 11). La integración se realiza aunque el individuo a integrar sea peor ya que así se puede explorar nuevas regiones del espacio de búsqueda. También hace falta otra barrera antes de hacer la copia del mejor individuo en el array (línea 8). Esto se debe a que si un hilo hace la combinación y las iteraciones, actualizará el array $mejores$, siendo posible que algún hilo aún no haya hecho la combinación.

Por último, cada hilo realiza una *sincronización* para actualizar la solución global (línea 13). Se actualizará si su solución local tiene un fitness mejor.

2.3. Solución Paralela con MPI

El esquema general de la solución MPI es muy parecido al de la solución OpenMP.

El esquema se muestra en el algoritmo 5. Primero, se lanzan p procesos (línea 1). P_0 envía a los demás procesos los datos del problema, es decir, los tamaños de las matrices, las matrices, los intervalos, etc (línea 2). P_0 también crea una solución global (líneas 3-5).

Al finalizar la ejecución tendremos la mejor solución en la variable Sol de P_0. Una vez que cada proceso tenga los datos, comenzará la ejecución. Cada uno de los p procesos existentes genera una subpoblación S_i de tamaño $tamPoblacion/p$ y una solución local Sol_i (línea 6).

Algorithm 5: Esquema MPI del algoritmo genético

```
1  EN PARALELO en cada proceso P_i (i = 0, ..., p − 1) HACER
2  P_0 envía los datos necesarios al resto de procesos
3  if nodo == P_0 then
4      GenerarIndividuo (Sol)
5  end
6  Inicializar (S_i,Sol_i,tamPoblacion/p)
7  for numGeneraciones do
8      for numIteraciones do
9          EsquemaSecuencial (S_i,Sol_i,tamPoblacion/p)
10     end
11     MPI_Allgather (Sol_i,mejores)
12     Integrar cada individuo (excepto Sol_i) del array mejores en S_i
13 end
14 MPI_Gather (Sol_i,solucionesLocales)
15 P_0 actualiza Sol con los individuos del array solucionesLocales
16 FIN PARALELO
```

Este esquema producirá endogamia, como pasaba con la solución OpenMP. Por eso, cada proceso agrupará las iteraciones en generaciones para combinar las subpoblaciones

(líneas 7-13).

La combinación de las subpoblaciones se hace de manera parecida a la de OpenMP, de cara a que cada proceso integre en su subpoblación los mejores individuos de los otros procesos.

Para hacerlo, cada proceso crea en cada generación un array de reales llamado $mejores$, de tamaño $p \times tamA$. Después, cada proceso envía su mejor solución local a los demás procesos (línea 11). De esta forma, cada proceso tendrá en el array $mejores$ todas las soluciones locales. Con MPI_Allgather [16], es muy fácil distribuir cada solución local de cada proceso a los demás (figura 2.1). Ahora que cada proceso tiene el array con las soluciones locales, proceden a integrar de manera aleatoria cada solución local menos la suya en su subpoblación.

La integración, como pasaba con OpenMP, se realiza aunque el individuo a integrar sea peor ya que así se puede explorar nuevas regiones del espacio de búsqueda.

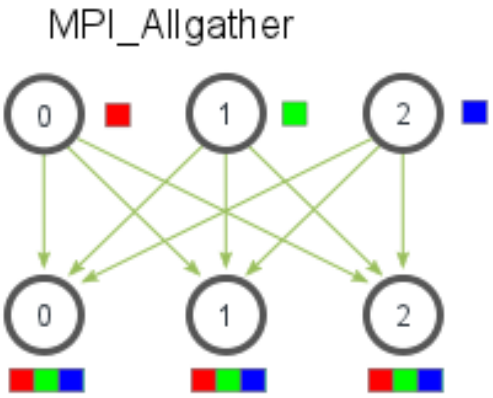


Figura 2.1: Ejemplo gráfico de como los datos se distribuyen con MPI_Allgather.

Para actualizar la solución global, P_0 debe tener las soluciones locales de los demás procesos (línea 14). Para hacerlo, se ha utilizado MPI Gather [16]. Se cogen las soluciones locales de los procesos y se envían a P_0 (figura 2.2). Las soluciones estarán disponibles en el array $solucionesLocales$. Por último, P_0 va actualizando la solución global conforme va comparando los valores de fitness.

2.4. Solución Paralela con MPI y OpenMP

La idea de esta versión es combinar MPI y OpenMP.

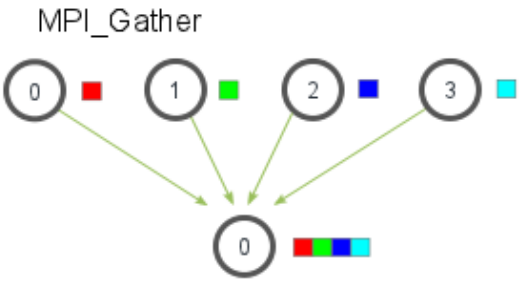


Figura 2.2: Ejemplo gráfico de como los datos se distribuyen con MPI_Gather.

Se puede decir que la solución mixta MPI+OpenMP implementa una solución por un conjunto de archipiélagos (conjunto de procesos MPI) donde cada archipiélago está formado por un conjunto de islas (hilos OpenMP). De esta forma, se lanzarán p procesos, y dentro de cada proceso n hilos.

Como en la solución MPI, el proceso 0 debe enviar al resto de procesos los datos necesarios del problema. Cada hilo tendrá una subpoblación inicial de tamaño $\frac{t}{pn}$. Estos hilos se lanzarán en cada generación a realizar. Cada uno evolucionará su subpoblación según las iteraciones establecidas y, con una sincronización, se actualizará la solución local del proceso.

Al finalizar la generación, se usará MPI_Allgather para que todos los procesos tengan todas las soluciones locales y puedan hacer la combinación. Así, antes de volver a evolucionar la población, cada hilo integrará de manera aleatoria en su subpoblación las soluciones de los demás procesos. Por último, P_0 actualizará la solución global como en la versión MPI. Con MPI_Gather, se cogerán las soluciones locales de todos los procesos y se enviarán a P_0 para que vaya comparando el fitness y actualizando si es necesario.

Capítulo 3

Resultados Experimentales

Este capítulo contiene los experimentos que se han realizado sobre las distintas versiones del algoritmo genético.

Primero se buscará un valor de cruce y de mutación que mejore el fitness en la solución secuencial, para utilizarse en el resto de experimentos. Después se realizarán experimentos sobre el tiempo y el fitness, utilizando distintos tamaños de población.

Además, se harán experimentos para ver como influyen las combinaciones de subpoblaciones en el tiempo y en el fitness en las versiones paralelas.

3.1. Generación del problema

Para la realización de las pruebas se ha utilizado un problema fijo, generado de forma aleatoria con las siguientes características:

- $d = 4$
- $e = 4$
- $t = 200$
- $i = 4$
- $k = 4$
- *intervalo* = [-0.3, 0.3]

Aunque el problema se haya generado utilizando un intervalo, se realizarán experimentos de tiempo utilizando las dos versiones del algoritmo. Para los experimentos de fitness solo se utilizará la versión más realista, es decir, intervalos de reales.

En el capítulo de mejoras se incluirán varios experimentos para analizar el comportamiento del algoritmo al variar el tamaño del problema.

3.2. Determinación de la probabilidad de cruce

Para determinar una probabilidad de cruce que intente mejorar el fitness se ha utilizado el siguiente rango de posibles valores: {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}.

De esta forma, para cada posible valor, se han realizado 10 ejecuciones para que los resultados sean más fiables y después se ha realizado el promedio del fitness obtenido. En esta prueba se utiliza una condición de fin de 20 segundos de tiempo, ya que un cruce de 0.9 tardará más tiempo en realizar las mismas iteraciones que un cruce de 0.1. En los resultados obtenidos (figura 3.1), se puede observar la *tendencia de que cuantos más cruces mejor solución*. Esto parece lógico ya que se evalúan más individuos.

En los siguientes experimentos se utilizará un valor de probabilidad de cruce de 0.9.

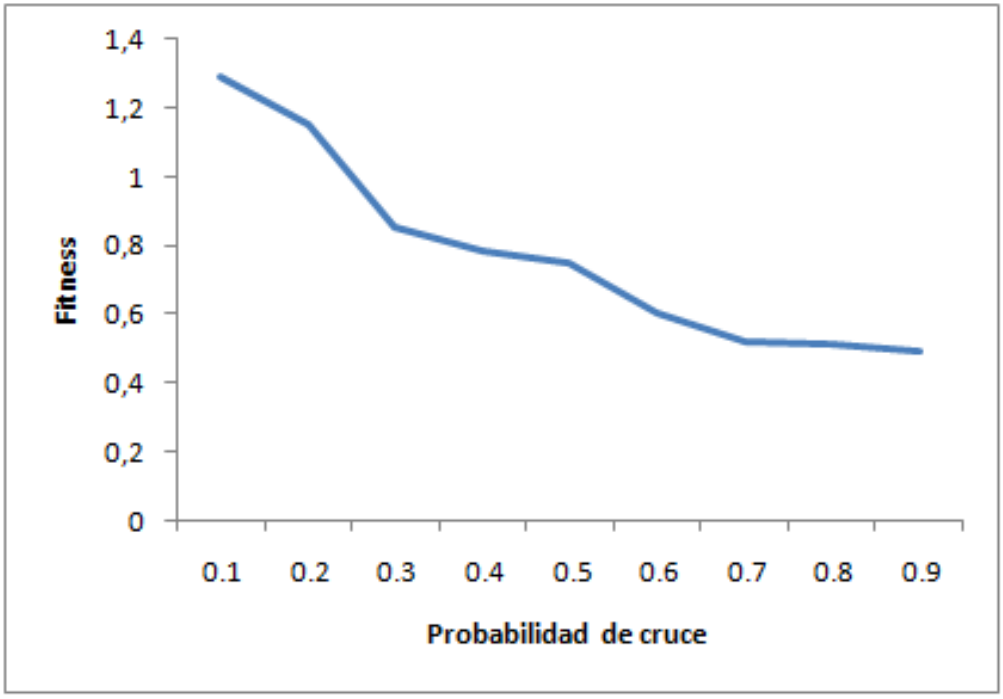


Figura 3.1: Valores de fitness obtenidos utilizando distintas probabilidades de cruce.

3.3. Determinación de la probabilidad de mutación

De forma similar a la determinación de la probabilidad de cruce, se obtiene la probabilidad de mutación. Se ha utilizado el siguiente rango de posibles valores:{0.005, 0.010, 0.015, 0.020, 0.025, 0.030, 0.035, 0.040, 0.050, 0.070, 0.100}. Para cada posible valor se han realizado 10 ejecuciones para que los resultados sean más fiables y después se ha realizado el promedio del fitness obtenido. Se ha utilizado una condición de fin de 20

segundos de tiempo para una comparación más justa. En los resultados obtenidos (figura 3.2), se puede observar la *tendencia de que cuantas menos mutaciones mejor solución*. Por tanto, se utilizará un valor de probabilidad de mutación de 0.005 en los siguientes experimentos.

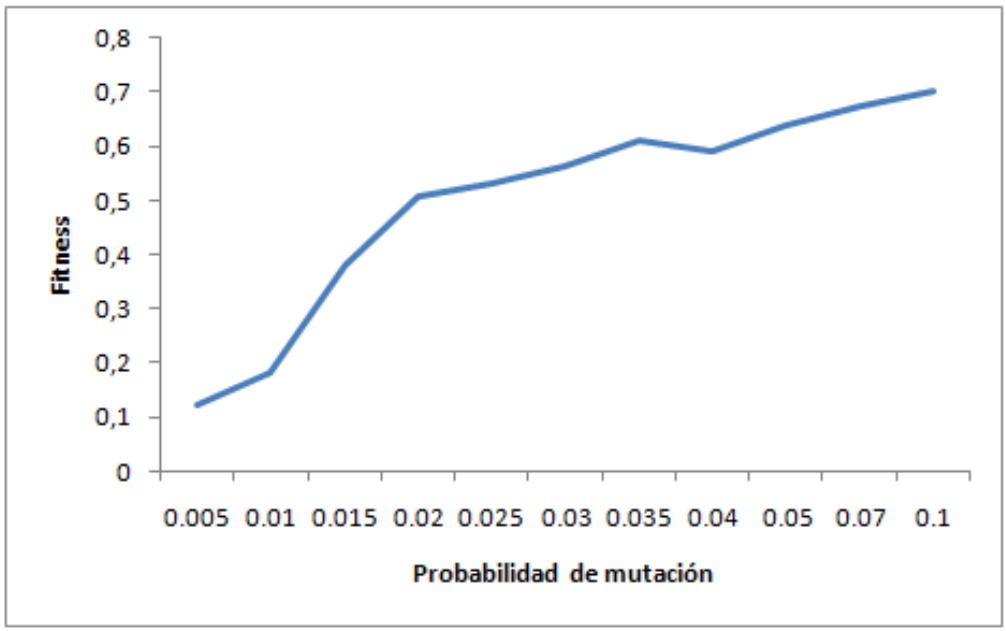


Figura 3.2: Valores de fitness obtenidos utilizando distintas probabilidades de mutación.

3.4. Experimentos de tiempo

En los siguientes experimentos se han utilizado los siguientes tamaños de población: {500, 1000, 1500, 2000}. Las ejecuciones se han realizado en Venus y Saturno, utilizando las dos versiones del problema. Además, se ha utilizado una condición de fin de 1000 iteraciones.

Los resultados obtenidos utilizando la solución secuencial y un intervalo de reales se muestran en la figura 3.3. Con cada aumento del tamaño de población se puede observar que el aumento del tiempo es similar al tiempo que tarda en hacer una ejecución de tamaño 500, lo que muestra el *comportamiento lineal en función del tamaño de la población*. También se observa que un core de Venus es aproximadamente el doble de rápido que uno de Saturno. Los resultados utilizando conjuntos han sido muy similares (figura 3.4).

Utilizando la solución OpenMP se han realizado las mismas ejecuciones con 2, 4 y 8 hilos. Para medir la ganancia de velocidad obtenida se ha calculado el Speed-Up dividendo el tiempo secuencial entre el tiempo OpenMP. Los resultados obtenidos utilizando un intervalo de reales se muestran en la figura 3.5. Los resultados obtenidos utilizando un

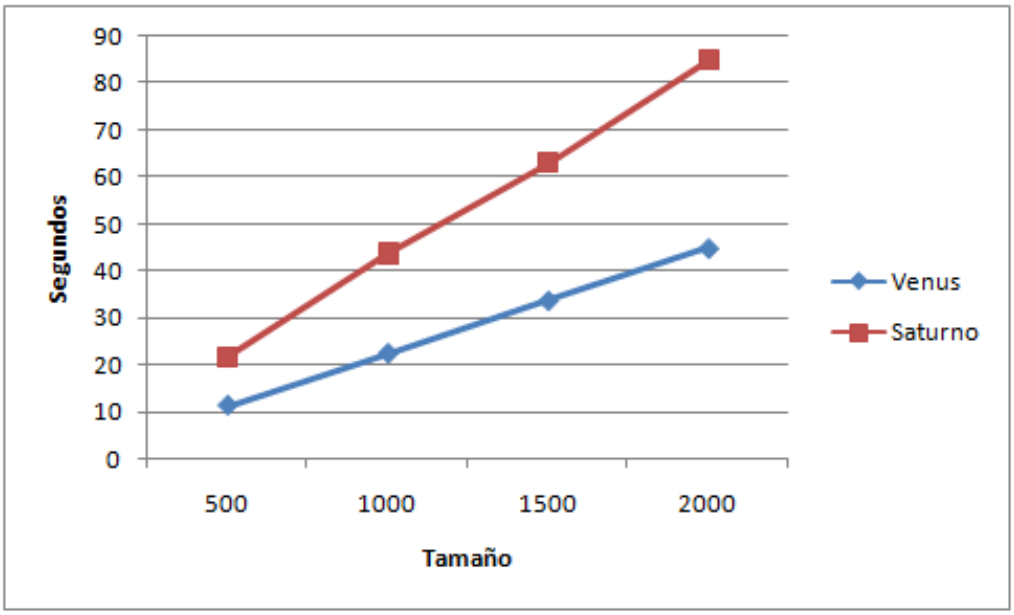


Figura 3.3: Tiempos obtenidos utilizando intervalos de reales con la solución secuencial.

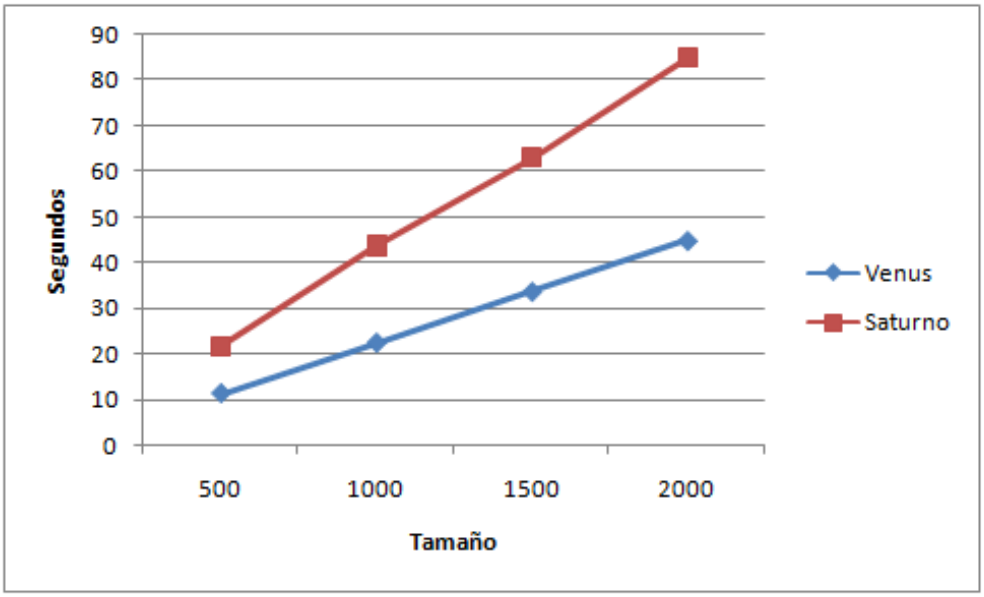


Figura 3.4: Tiempos obtenidos utilizando conjuntos de reales con la solución secuencial.

conjunto de reales se muestran en la figura 3.6. Los resultados de ambas versiones han sido muy similares, como pasaba con la solución secuencial. El Speed-Up es casi lineal hasta los 8 hilos. A partir de 8 hilos el Speed-Up fluctúa debido a las sincronizaciones y barreras utilizadas para las migraciones. También se observa que *Saturno consigue un mejor Speed-Up que Venus a partir de 8 hilos*, lo que es normal teniendo en cuenta la mayor capacidad computacional de Venus.

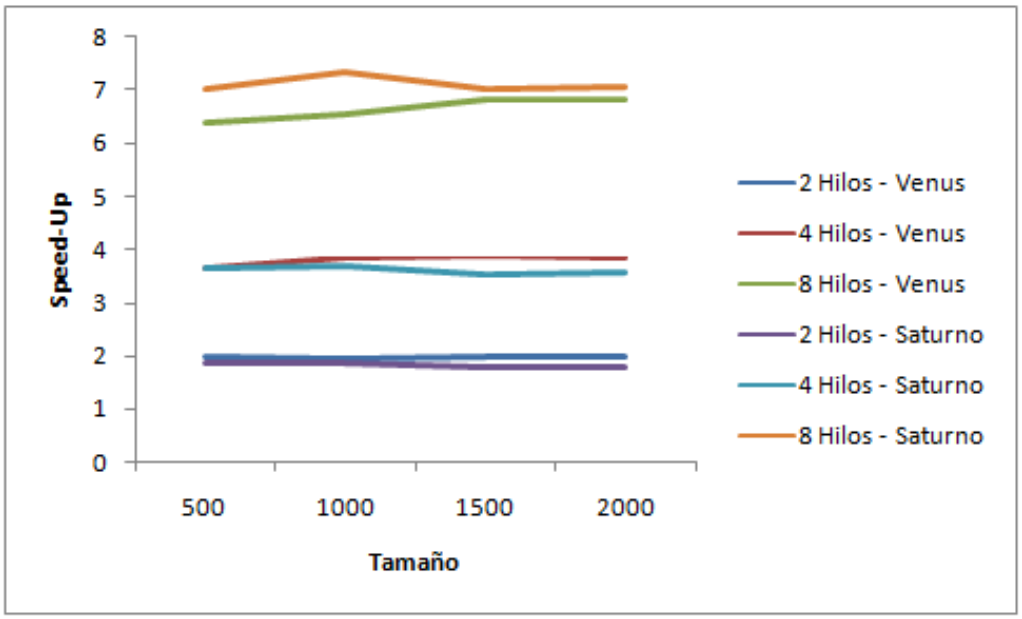


Figura 3.5: Speed-Up obtenidos utilizando intervalos de reales con la solución OpenMP.

Las conclusiones que se han obtenido utilizando la solución MPI y 2, 4 y 8 procesos han sido similares. Los resultados obtenidos utilizando un intervalo de reales se muestran en la figura 3.7, y los correspondientes a un conjunto de reales se muestran en la figura 3.8.

Las conclusiones que se han obtenido utilizando la solución híbrida también han sido las mismas. Se han utilizado 2 procesos / 1 hilo (equivalente a 2 procesos), 2 procesos / 2 hilos (equivalente a 4 procesos) y 4 procesos / 2 hilos (equivalente a 8 procesos). Los resultados obtenidos utilizando un intervalo de reales se muestran en la figura 3.9, y los de un conjunto de reales en la figura 3.10.

3.5. Experimentos de fitness

En los siguientes experimentos se han utilizado los siguientes tamaños de población:{10, 20, 30, 50, 80, 100, 150, 200, 250, 400, 500, 700, 800, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5000}. Las ejecuciones se han realizado en Venus y Saturno, utilizando intervalos de reales. Además, se han realizado 10 ejecuciones para cada tamaño y después

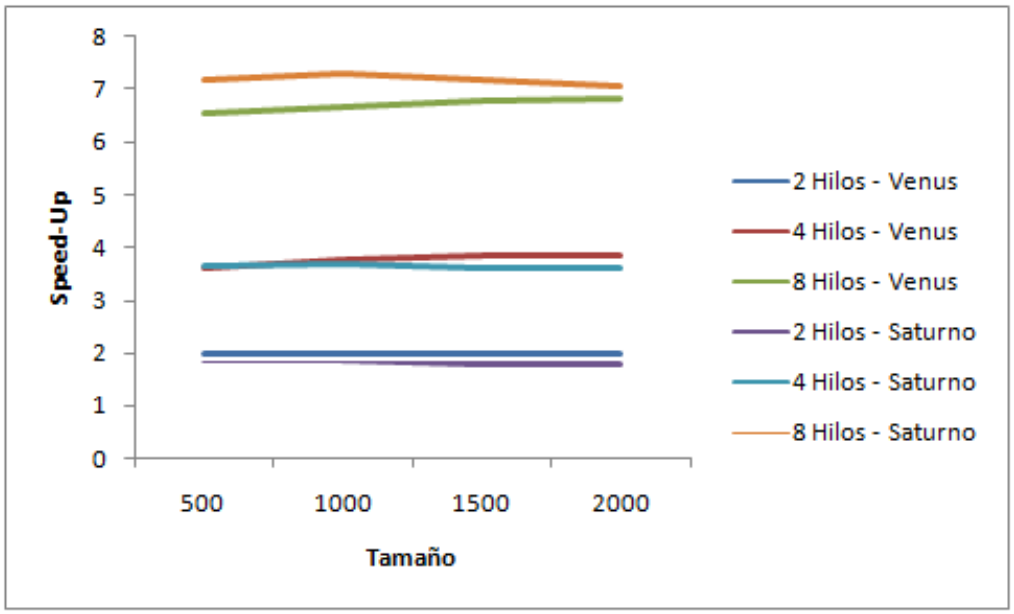


Figura 3.6: Speed-Up obtenidos utilizando conjuntos de reales con la solución OpenMP.

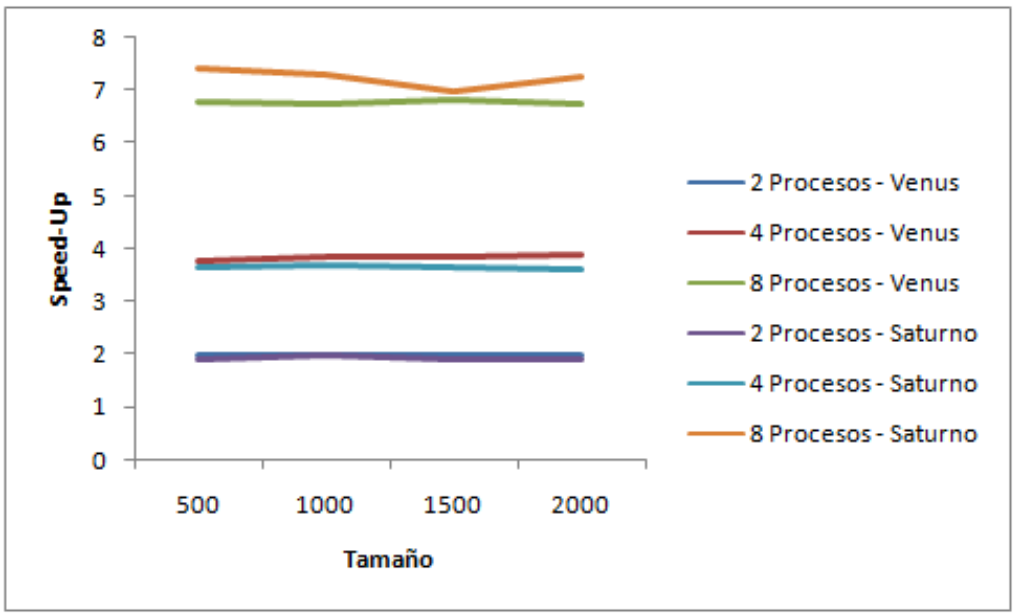


Figura 3.7: Speed-Up obtenidos utilizando intervalos de reales con la solución MPI.

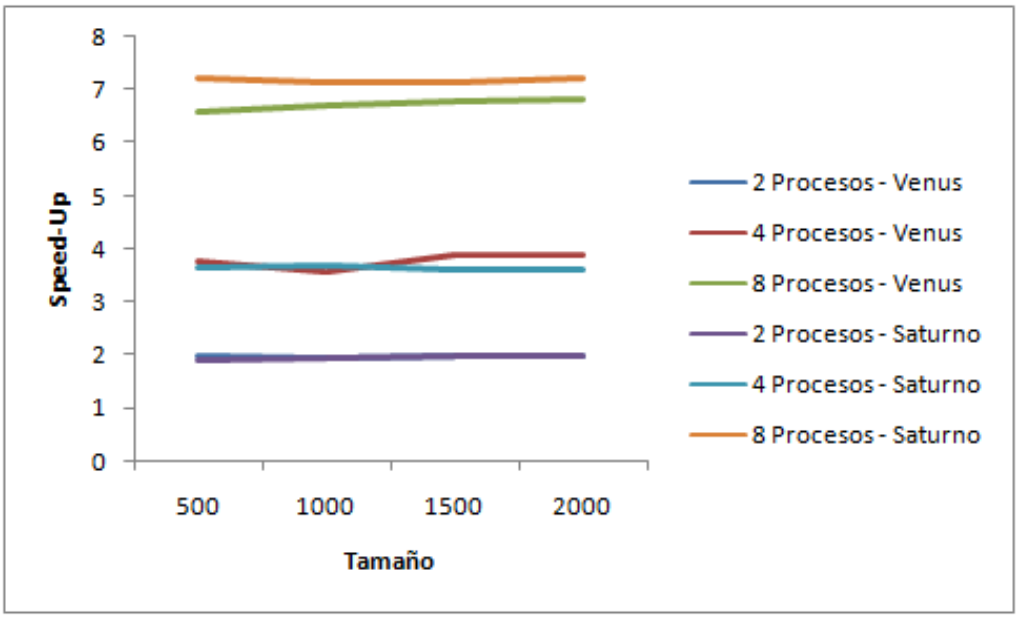


Figura 3.8: Speed-Up obtenidos utilizando conjuntos de reales con la solución MPI.

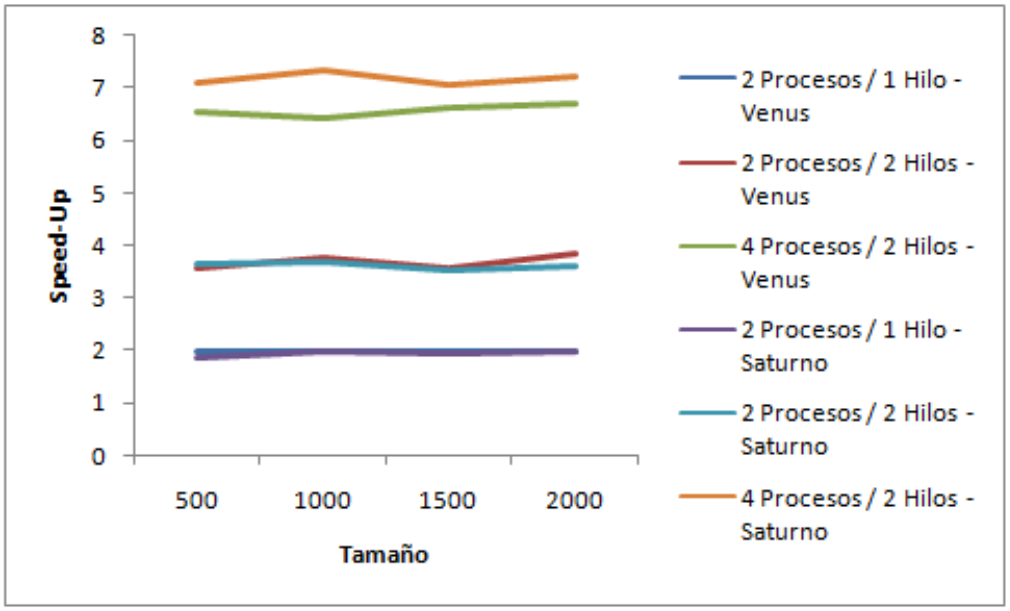


Figura 3.9: Speed-Up obtenidos utilizando intervalos de reales con la solución híbrida.

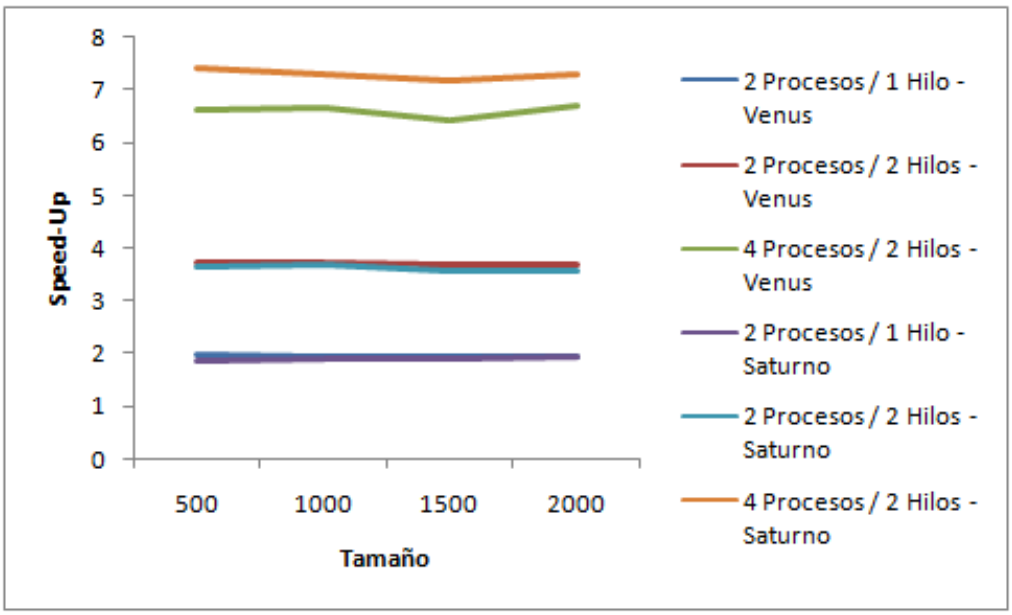


Figura 3.10: Speed-Up obtenidos utilizando conjuntos de reales con la solución híbrida.

se ha realizado el promedio del fitness obtenido.

Para experimentar con la solución secuencial y para una comparación justa, se ha utilizado una condición de fin de 12 segundos para Venus y 22 segundos para Saturno. Este tiempo es lo que tardan en ejecutarse 100 iteraciones de tamaño 5000. Los resultados obtenidos con Venus se muestran en la figura 3.11, y los obtenidos con Saturno en la figura 3.12. En ambos gráficos se puede observar que *los valores de fitness son similares* y que:

- Un tamaño de población menor de 30 obtiene peores resultados.
- Un tamaño de población entre 30 y 1000 obtiene buenos resultados.
- Un tamaño de población mayor de 1000 va empeorando los resultados muy rápidamente. Esto puede deberse por no incluir elitismo y garantizar que la calidad de la solución no disminuya de una generación a otra. En el capítulo de mejoras se hará una versión con elitismo para ver si mejora el comportamiento.

Para las soluciones paralelas se ha querido analizar si se mejora la solución en el mismo tiempo que tarda en ejecutarse el secuencial. Es decir, 12 segundos en Venus y 22 segundos en Saturno.

De esta forma, utilizando la solución OpenMP, tamaños similares y 2, 4 y 8 hilos, se han obtenido en Venus los resultados que se muestran en la figura 3.13, y en Saturno los que se muestran en la figura 3.14. En ambos gráficos se puede observar que en el mismo tiempo se obtienen mejores resultados que en secuencial, siempre que se use un tamaño de población menor de 1000. También se observa que *cuantos más hilos se utilicen, mejor*

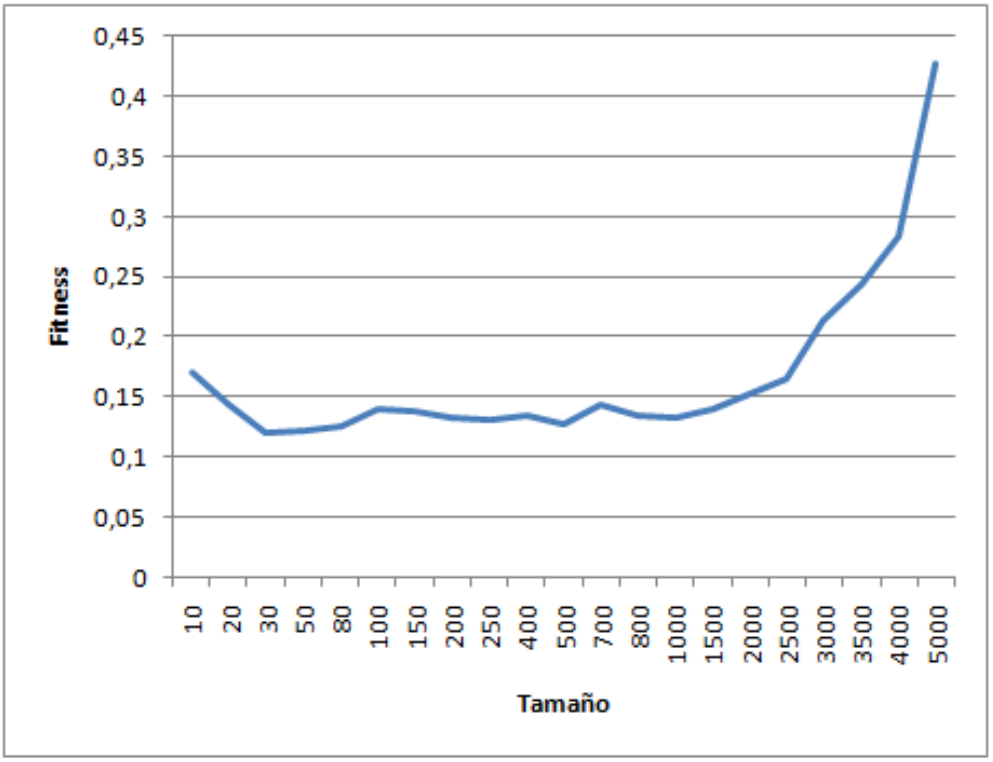


Figura 3.11: Valores de fitness obtenidos en Venus utilizando intervalos de reales con la solución secuencial.

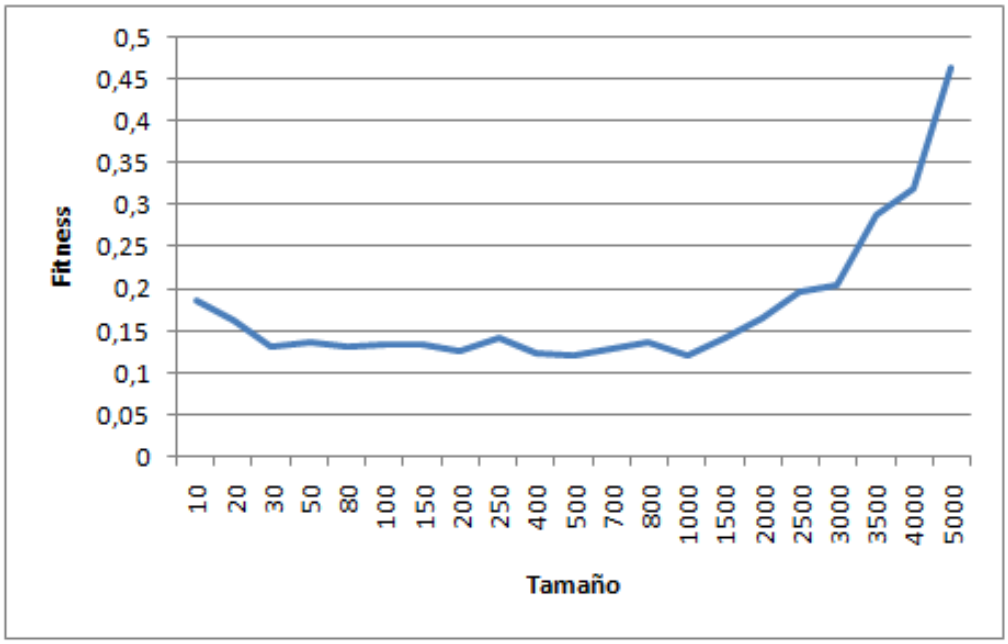


Figura 3.12: Valores de fitness obtenidos en Saturno utilizando intervalos de reales con la solución secuencial.

es la solución obtenida.

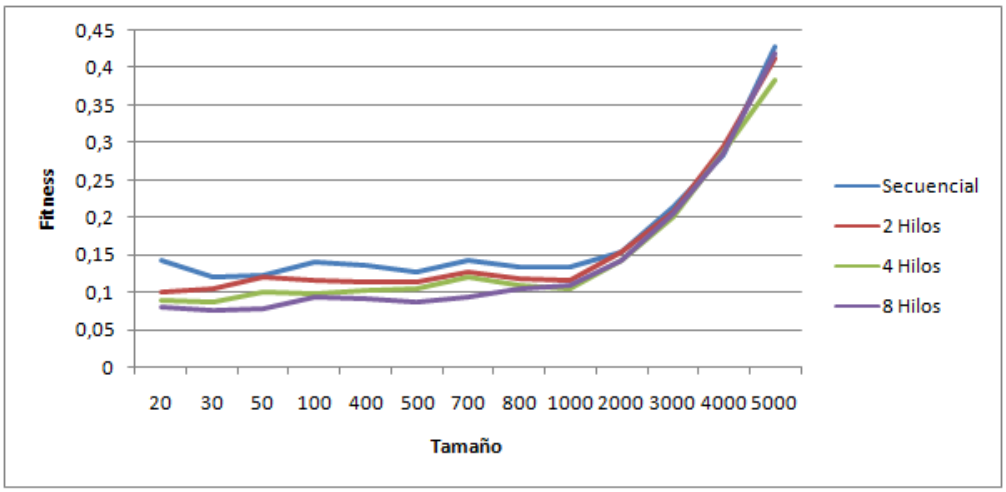


Figura 3.13: Valores de fitness obtenidos en Venus utilizando intervalos de reales con la solución OpenMP.

Las conclusiones que se han obtenido utilizando la solución MPI y 2, 4 y 8 procesos han sido las mismas. Los resultados de Venus se muestran en la figura 3.15, y los de Saturno se muestran en la figura 3.16.

Las conclusiones que se han obtenido utilizando la solución híbrida también han sido las mismas. Se han utilizado 2 procesos / 1 hilo (equivalente a 2 procesos), 2 procesos / 2 hilos (equivalente a 4 procesos) y 4 procesos / 2 hilos (equivalente a 8 procesos). Los resultados de Venus se muestran en la figura 3.17, y los de Saturno se muestran en la figura 3.18.

3.6. Experimentos de migraciones

Para ver cómo influyen las migraciones se ha obtenido el promedio de múltiples ejecuciones con un número de iteraciones fijas con distintas iteraciones por generación: {10, 100, 600}.

En Venus se han utilizado 6 y 12 hilos / procesos y en Saturno 12 y 48 hilos / procesos.

Con esto se pretende ver la influencia de las migraciones en el tiempo al utilizar muchos más hilos / procesos y distintas frecuencias de migración. También se pretender observar si el fitness sigue mejorando al utilizar muchos más hilos / procesos como pasaba en los experimentos de fitness anteriores y si merece la pena utilizar frecuencias de

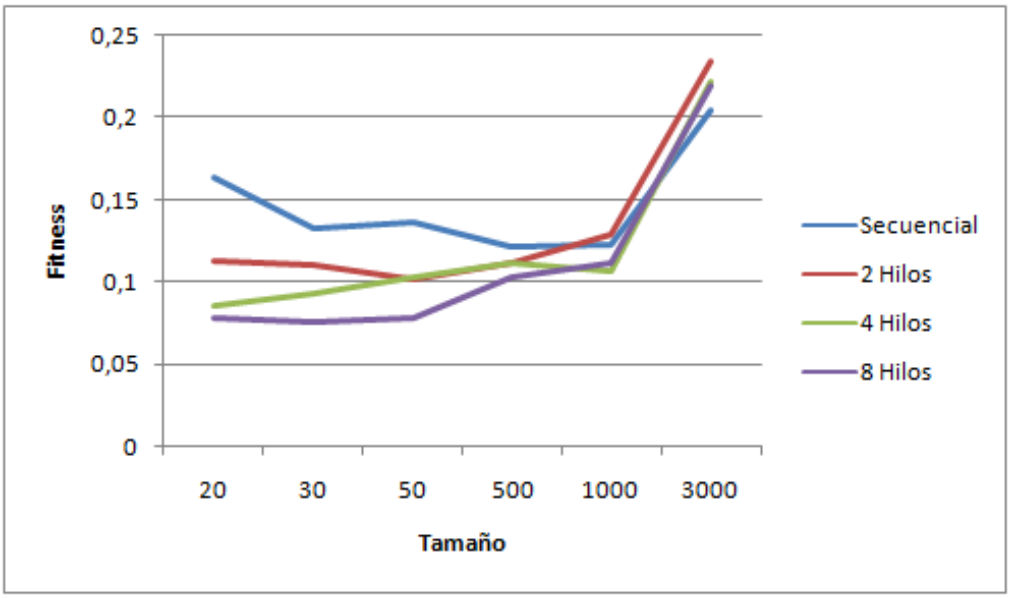


Figura 3.14: Valores de fitness obtenidos en Saturno utilizando intervalos de reales con la solución OpenMP.

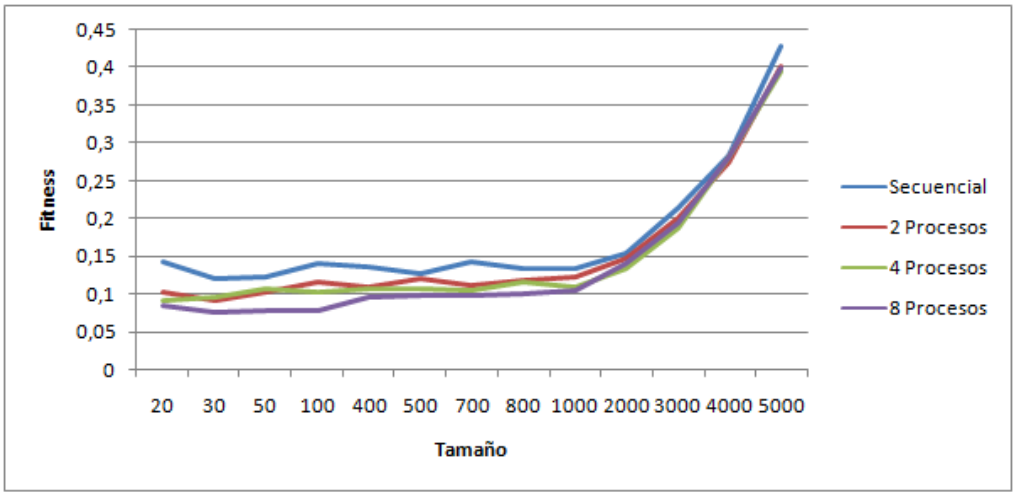


Figura 3.15: Valores de fitness obtenidos en Venus utilizando intervalos de reales con la solución MPI.

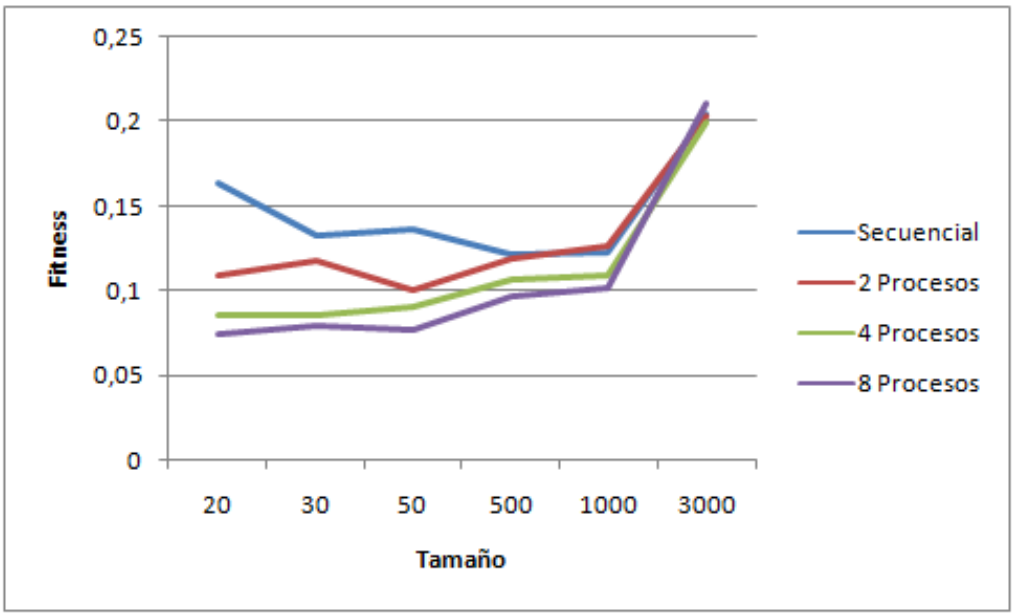


Figura 3.16: Valores de fitness obtenidos en Saturno utilizando intervalos de reales con la solución MPI.

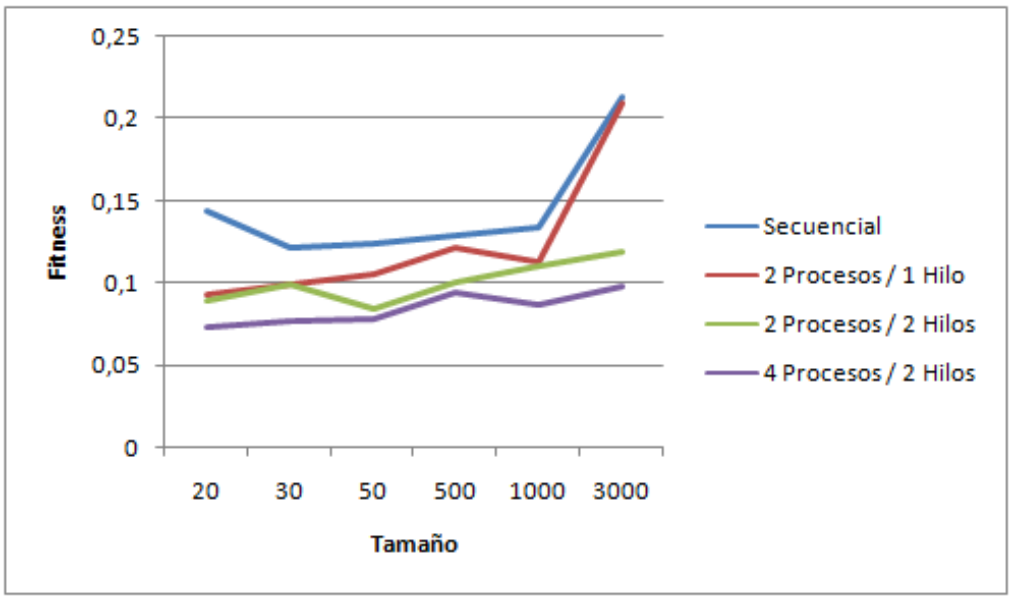


Figura 3.17: Valores de fitness obtenidos en Venus utilizando intervalos de reales con la solución híbrida.

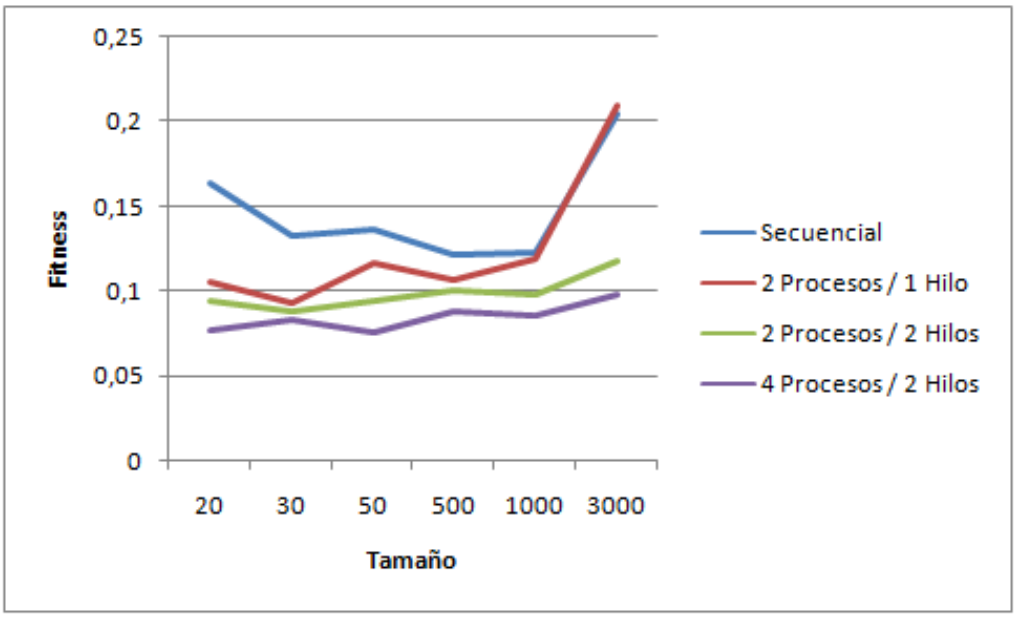


Figura 3.18: Valores de fitness obtenidos en Saturno utilizando intervalos de reales con la solución híbrida.

migración altas a pesar de que puedan tardar más tiempo.

Los resultados de tiempo que se han obtenido en Venus se muestran en la figura 3.19. Los resultados de tiempo que se han obtenido en Saturno se muestran en la figura 3.20.

Como se puede ver en ambos resultados, el tiempo disminuye si se realizan menos migraciones. Esto es lógico ya que se realizan menos comunicaciones / sincronizaciones. Al utilizar el doble de hilos / procesos en Venus, no se aprecia demasiado el tiempo de las migraciones. En Saturno se puede ver más claramente la influencia de las migraciones con el uso de muchos más hilos / procesos.

Para analizar el fitness se ha obtenido el promedio de múltiples ejecuciones utilizando los mismos parámetros pero utilizando un tiempo fijo en segundos. Los resultados de fitness que se han obtenido en Venus se muestran en la figura 3.21. Los resultados de fitness que se han obtenido en Saturno se muestran en la figura 3.22.

Se puede observar que utilizar una *frecuencia de migración alta* merece la pena ya que se obtienen mejores resultados en el mismo tiempo. También se puede observar que con *el uso de muchos más hilos / procesos se siguen mejorando las calidades de las soluciones*.

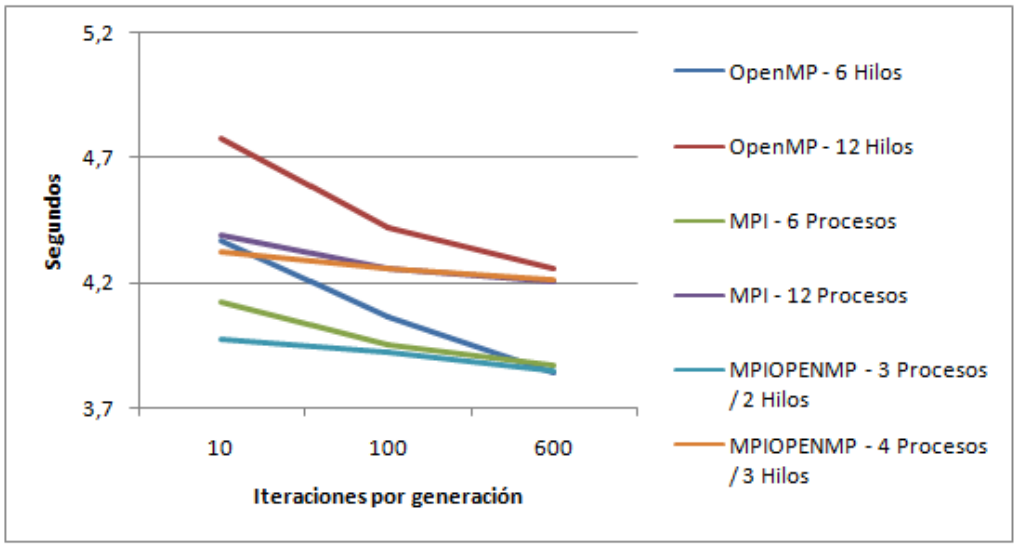


Figura 3.19: Tiempos obtenidos en Venus utilizando todas las soluciones con varios hilos / procesos, distintas frecuencias de migración y un número fijo de iteraciones.

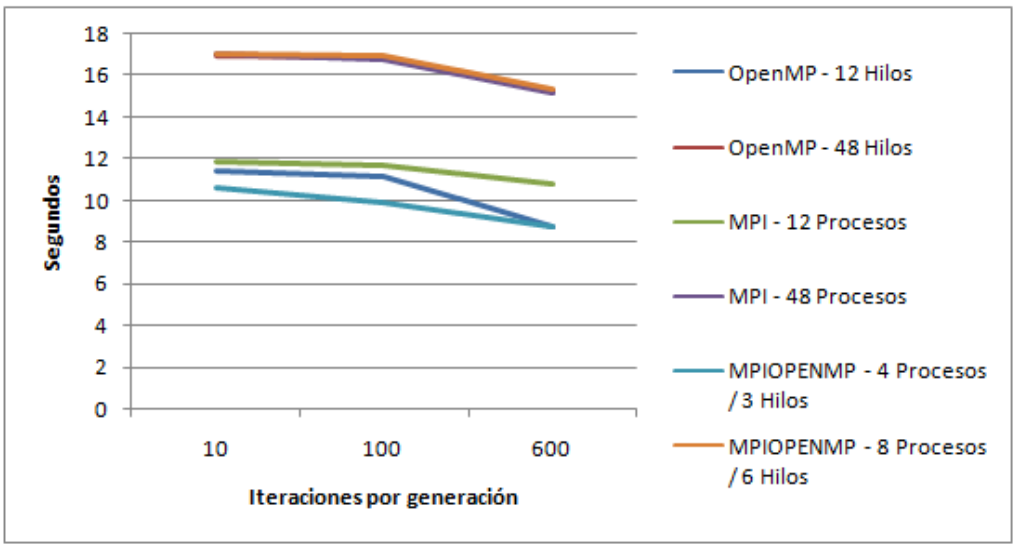


Figura 3.20: Tiempos obtenidos en Saturno utilizando todas las soluciones con varios hilos / procesos, distintas frecuencias de migración y un número fijo de iteraciones.

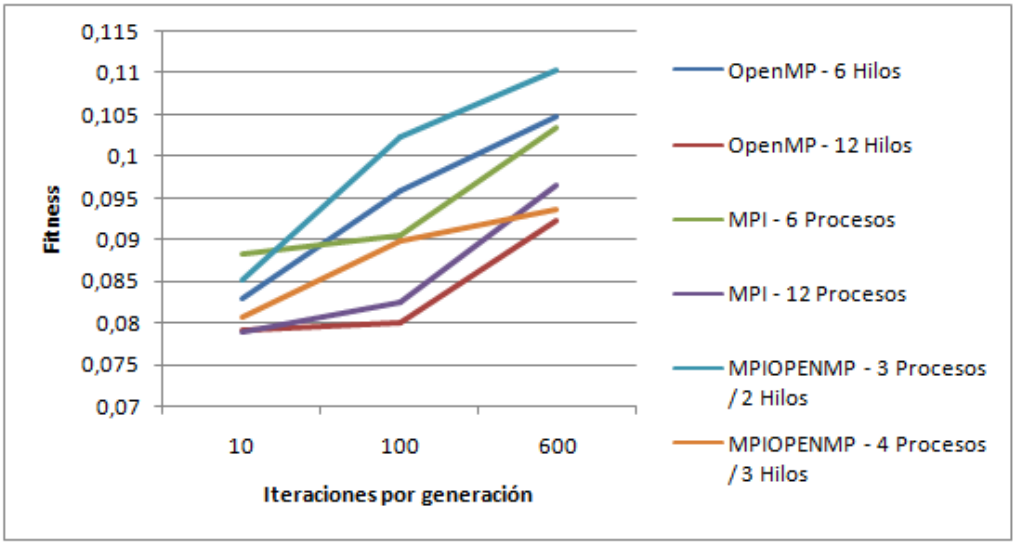


Figura 3.21: Valores de fitness obtenidos en Venus utilizando todas las soluciones con varios hilos / procesos, distintas frecuencias de migración y un tiempo fijo.

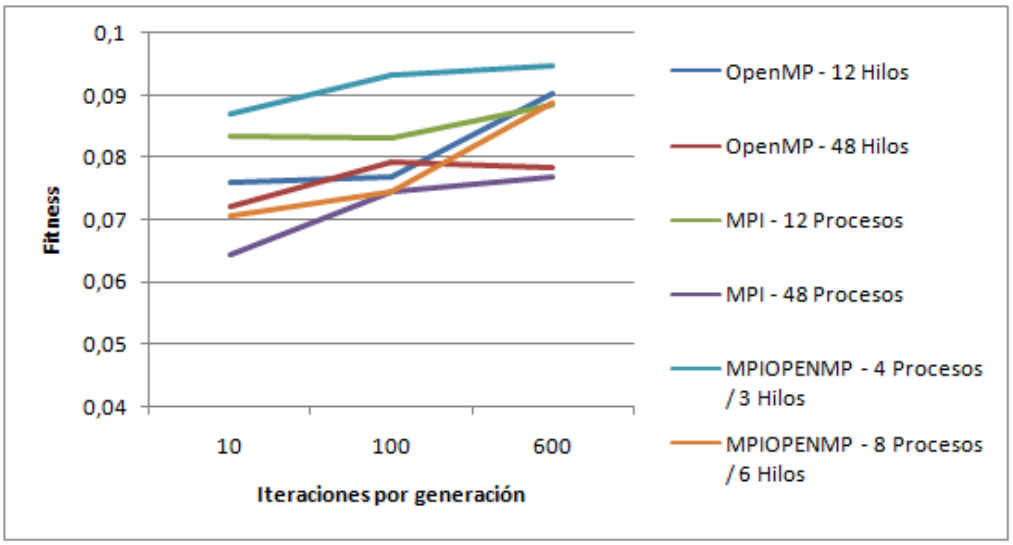


Figura 3.22: Valores de fitness obtenidos en Saturno utilizando todas las soluciones con varios hilos / procesos, distintas frecuencias de migración y un tiempo fijo.

Capítulo 4

Mejoras

En este capítulo se describen algunas modificaciones al esquema básico para reducir el tiempo de ejecución y mejorar la calidad de las soluciones, así como distintos experimentos para analizar los beneficios obtenidos y un análisis experimental para estudiar y explotar la heterogeneidad del clúster Heterosolar.

Se consideran dos tipos de modificaciones: de las funciones del algoritmo genético básico, y de la parte computacional del cálculo del fitness.

4.1. Modificaciones en las funciones del Algoritmo Genético

4.1.1. Mejora del operador de cruce

Hasta este momento se ha estado utilizando un operador de cruce sencillo, llamado cruce por un punto. Para intentar mejorar el fitness se han implementado tres nuevos operadores de cruce:

- *Cruce por dos puntos*: se eligen dos puntos de cruce de forma aleatoria entre 0 y $tamA - 1$. Después, ambos individuos van intercambiando sus elementos del modelo que se encuentren entre esos dos puntos.

- *Cruce aritmético*: llamamos i_1 y i_2 a los individuos a cruzar. Se toma como primer descendiente a $1/3{*}i_1{+}2/3{*}i_2$ y como segundo descendiente a $2/3{*}i_1{+}1/3{*}i_2$. De esta forma, en lugar de cambiar elementos de ambos individuos, pudiendo perder todo el significado que pudiera tener la relación entre ellos, se toman elementos que están aritméticamente entre los elementos de los dos.

- *Cruce aleatorio*: para cada cruce de individuos, se elige de forma aleatoria el operador de cruce que se va a aplicar. De esta forma, se podrá aplicar cruce aritmético, cruce por un punto y cruce por dos puntos.

4.1.2. Extensión del algoritmo genético

Para intentar mejorar el fitness se ha implementado una extensión en el algoritmo genético para *añadir elitismo*.

Esta mejora consiste en *garantizar que la calidad de la solución no disminuya de una generación a otra*. Así, para cada nueva generación, se forma un conjunto de individuos con los individuos de la antigua población, los individuos cruzados y los individuos mutados. Una vez que se hayan calculado los fitness de todos estos individuos se ordenan para incluir en la nueva población a los mejores individuos.

4.1.3. Mejora de la selección

Para intentar mejorar el fitness se han implementado tres nuevas funciones de selección [17]:

- *Selección ruleta*: la idea es que los mejores individuos tengan mayores probabilidades de ser elegidos.

  Se construye una ruleta en la que cada una de las porciones representa a un individuo de forma proporcional a su fitness. De esta forma, los mejores individuos se llevan las mejores porciones.

  Este proceso se realiza en 4 pasos.

  Primero se calcula la suma de todos los fitness de cada individuo. A esta suma le llamamos $fitnessTotal$.

  En un segundo paso, al ser un problema de minimización, se calcula la suma acumulada de restar, para cada individuo, $fitnessTotal$ y su fitness. A esta suma le llamamos $sumaTotal$. Esto es importante porque un individuo con menor fitness deberá sumar más que uno con mayor fitness.

  En el paso 3 se genera un número aleatorio entre 0 y $sumaTotal$. Por último, se recorre la población acumulando, para cada individuo, $fitnessTotal$ menos su fitness. Cuando la suma que se lleve sea mayor o igual al número generado en el paso 3, se selecciona el individuo.

- *Selección ranking*: los individuos son ordenados respecto a su fitness.

  Si tenemos n individuos, al individuo con peor fitness se le asigna un 1 y al individuo con mejor fitness la n. El proceso es muy similar al proceso de la selección por ruleta.

  En un primer paso se calcula $nTotal$ mediante la fórmula $n * (n + 1)/2$.

  Después se genera un número aleatorio entre 0 y $nTotal$.

  Por último, se recorre la población ordenada acumulando el valor de cada individuo. Cuando la suma que se lleve sea mayor o igual al número generado en el paso 2, se selecciona el individuo.

- *Selección aleatoria*: para cada selección a realizar se elige de forma aleatoria la función de selección.

  De esta forma, se podrá utilizar selección por torneo, ruleta o ranking. Debido a que la selección por ruleta funciona mal cuando existen grandes diferencias entre los

fitness (si un individuo ocupa gran parte de la ruleta los demás individuos van a tener muy pocas posibilidades de resultar elegidos) de los individuos de la población, no se utilizará en la implementación sin extensión.

4.1.4. Búsqueda Local

Para intentar mejorar aún más el fitness se ha diseñado un *método híbrido combinando el algoritmo genético con búsqueda local*.

La mejora por búsqueda local se aplica a cada individuo al inicializar una población y tras formar cada nueva generación.

Consiste en ir analizando la vecindad de los elementos del modelo del individuo un número determinado de pasos. La vecindad es de dos individuos.

En cada paso se elige de forma aleatoria el valor a actualizar.

El primer individuo se obtiene sumando al valor actual del elemento un valor aleatorio en el intervalo [$0{,}001$, $porcentaje * valor/100$]. Así, el valor se modificará un mínimo de $0{,}001$ y un porcentaje máximo de su valor.

El segundo individuo se obtiene de la misma manera, pero restando. Para estos dos individuos se vuelve a calcular el fitness. Si alguno de estos dos nuevos individuos mejora al individuo original, se utilizará para los siguientes pasos; si no, se seguirá utilizando el individuo original.

Calcular el fitness de un individuo es muy costoso y esta mejora realiza muchos cálculos. Para paliar este problema se ha limitado el número de pasos.

4.2. Mejoras computacionales en el cálculo del fitness

4.2.1. Optimización del cálculo del fitness para la mejora de Búsqueda Local

Para reducir el tiempo de ejecución tras la implementación de la mejora de Búsqueda Local se propone la *modificación de la función de calcular el fitness de un individuo*.

Esta nueva función de calcular fitness se utilizará sólo para la Búsqueda Local.

En el algoritmo 6 se muestra cómo se realiza el cálculo.

Primero, se guarda la matriz $\hat{Y}$ y el fitness del individuo que se va a mejorar (sin haber realizado la raíz cuadrada). Al modificar solo un elemento del modelo del individuo,

únicamente hay que recalcular una columna de la matriz $\hat{Y}$. Para cada instante de tiempo i (menos los instantes iniciales) y sólo para la variable afectada de cada instante de tiempo i, se hace la multiplicación de la fila de $\hat{X}$ correspondiente por la columna correspondiente del individuo. Esta multiplicación tampoco hay que hacerla entera ya que, como hemos guardado $\hat{Y}$, tenemos el cálculo. Si llamamos $posicionEle$ a la posición que se modifica en el individuo, $ovalor$ al valor del elemento antes de cambiarlo , $nvalor$ al nuevo valor y $xafectado$ al valor de la posición $posicionEle/d$ de la fila correspondiente de $\hat{X}$, es suficiente con restar al cálculo que ya teníamos $ovalor * xafectado$ y sumarle $nvalor * xafectado$.

Una vez que tenemos el nuevo valor calculado de ese instante de tiempo i, se recalcula el fitness restando la suma de los cuadrados de las diferencias entre el elemento afectado de la matriz $\hat{Y}$ del problema original y el elemento afectado de la matriz $\hat{Y}$ guardada inicialmente y sumando la suma de los cuadrados de las diferencias entre el elemento afectado de la matriz $\hat{Y}$ del problema original y el nuevo valor.

El orden del tiempo de ejecución en el cálculo del fitness pasa de $O(n^3)$ a $O(n)$, lo que puede repercutir de forma importante en el tiempo de ejecución del algoritmo en métodos que incluyan búsqueda local, especialmente con vecindades grandes.

Algorithm 6: Cálculo del fitness de un individuo para la mejora de Búsqueda Local

```
1  columna = posicionEle %d
2  fitness = fitnessIndividuo
3  for i = 0 to t − ni do
4      xindex = i * tamX + posicionEle/d
5      iindex = i * d + columna
6      vYi = Y[iindex]
7      vYt = Ytrabajo[iindex]
8      vY = vYi − ovalor * X[xindex] + nvalor * X[xindex]
9      fitness = fitness − (vYt − vYi) * (vYt − vYi)
10     fitness = fitness + (vYt − vY) * (vYt − vY)
11     Y[iindex] = vY
12 end
```

4.2.2. Uso de la librería BLAS

Para intentar reducir el tiempo de ejecución de la función de calcular fitness se ha utilizado BLAS.

BLAS es una especificación que *proporciona rutinas de bajo nivel para realizar operaciones de álgebra lineal* [18]. De esta forma, se utiliza la rutina dgemm para la multiplicación de matrices en lugar de utilizar los bucles del algoritmo 1. La forma de calcular el fitness es muy parecida. El resultado de multiplicar la matriz X y la matriz individuo

se guarda en una matriz auxiliar llamada aux. Para calcular el fitness se va almacenando en la variable *fitness* la suma de los cuadrados de las diferencias entre elementos de Y y aux. Por último, se hace la raíz cuadrada de la variable fitness y se establece el fitness del individuo.

La implementación de BLAS es general. Hay implementaciones optimizadas para conseguir un mayor rendimiento. Algunas son OpenBLAS [19], Intel MKL [20] y IBM ESSL [21]. Se experimentará con OpenBLAS. Su rendimiento es muy similar a las otras implementaciones en multiplicaciones de matrices densas [22].

4.3. Análisis experimental de las mejoras propuestas

Para determinar el mejor operador de cruce se ha obtenido el promedio utilizando la solución secuencial con distintos tamaños de población, 10 ejecuciones por tamaño y un tiempo fijo de 20 segundos.

Los resultados obtenidos se muestran en la figura 4.1. Se puede observar que el *mejor operador de cruce es el Aleatorio* por su tendencia a obtener mejores resultados en un tiempo fijo. El operador Aritmético, que inicialmente introducimos para tomar descendientes en la línea que une los dos ascendientes, es el que da peor resultado.

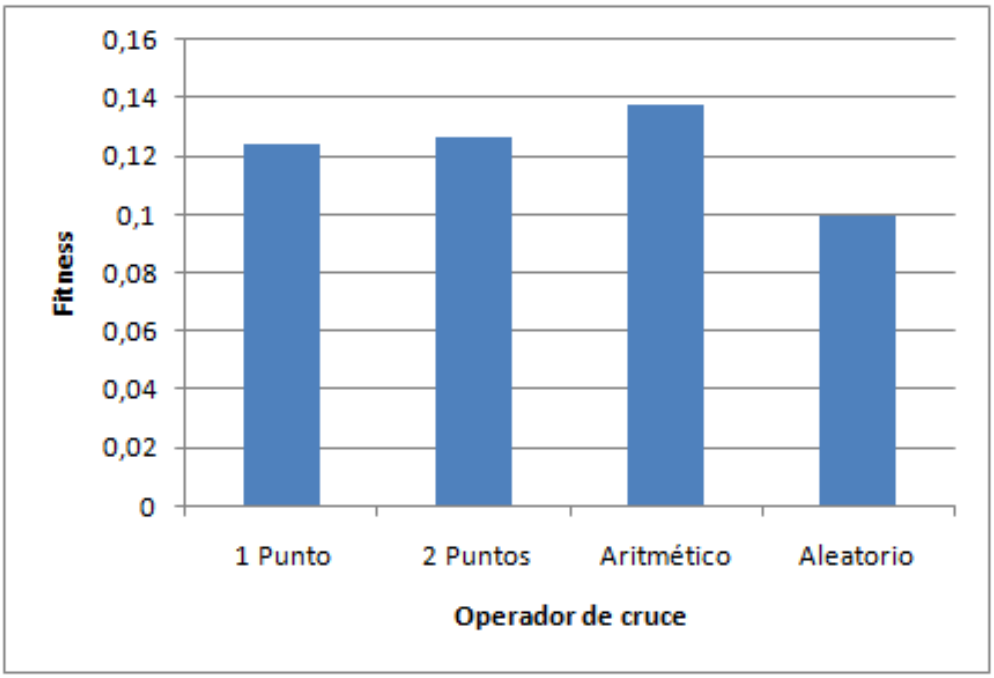


Figura 4.1: Valores de fitness obtenidos con la solución secuencial utilizando intervalos de reales, distintos operadores de cruce y un tiempo fijo.

Para determinar la mejor función de selección se ha obtenido el promedio utilizando la solución secuencial con y sin extensión, distintos tamaños de población, 10 ejecuciones por tamaño y un tiempo fijo de 20 segundos. Los resultados obtenidos se muestran en la figura 4.2.

No se observan diferencias significativas entre los distintos métodos de selección, y en lo sucesivo se utilizará selección por torneo. Sí se aprecia que considerando elitismo se obtienen mejores resultados.

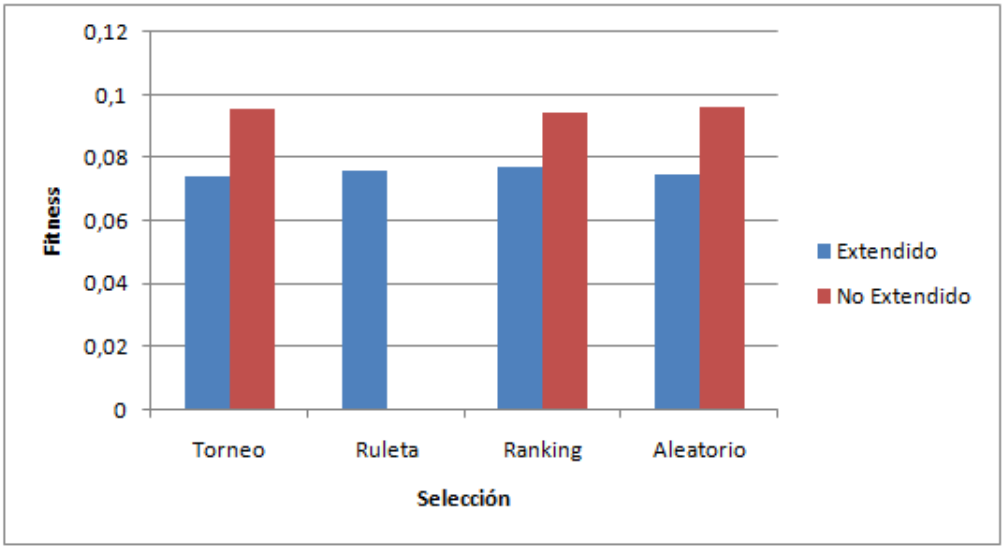


Figura 4.2: Valores de fitness obtenidos con la solución secuencial con y sin extensión utilizando intervalos de reales, distintas funciones de selección y un tiempo fijo.

Para analizar la mejora de Búsqueda Local se tiene que determinar cuánto puede cambiar un valor de un elemento como máximo y el número de pasos a realizar.

El primer parámetro, el porcentaje máximo a sumar o restar de un valor, se ha obtenido analizando el promedio de múltiples ejecuciones con distintos porcentajes y un tiempo fijo. Los resultados obtenidos se muestran en la figura 4.3. Se puede observar que un porcentaje de 10 obtiene buenos resultados.

El segundo parámetro, el número de pasos, se ha obtenido analizando el promedio de múltiples ejecuciones con distintos números de pasos y un tiempo fijo de 19 segundos (el tiempo de ejecución utilizando 10000 pasos). Los resultados obtenidos se muestran en la figura 4.4.

Se puede observar la tendencia de que *cuantos más pasos, mejor fitness*. El número de pasos óptimo es de 2000 ya que a partir de 2000 pasos el fitness va empeorando.

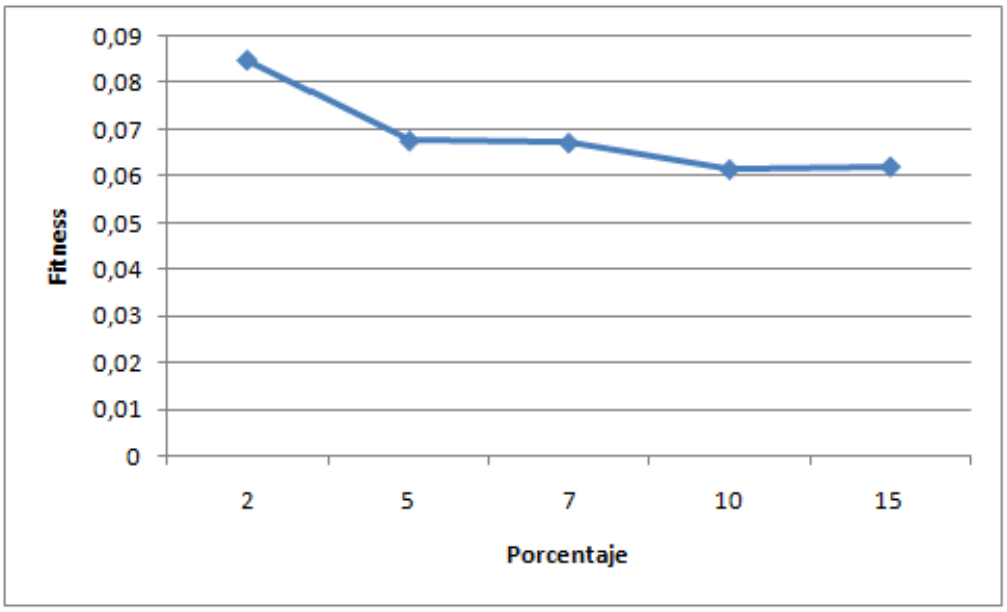


Figura 4.3: Valores de fitness obtenidos con la solución secuencial utilizando intervalos de reales, distintos porcentajes y un tiempo fijo.

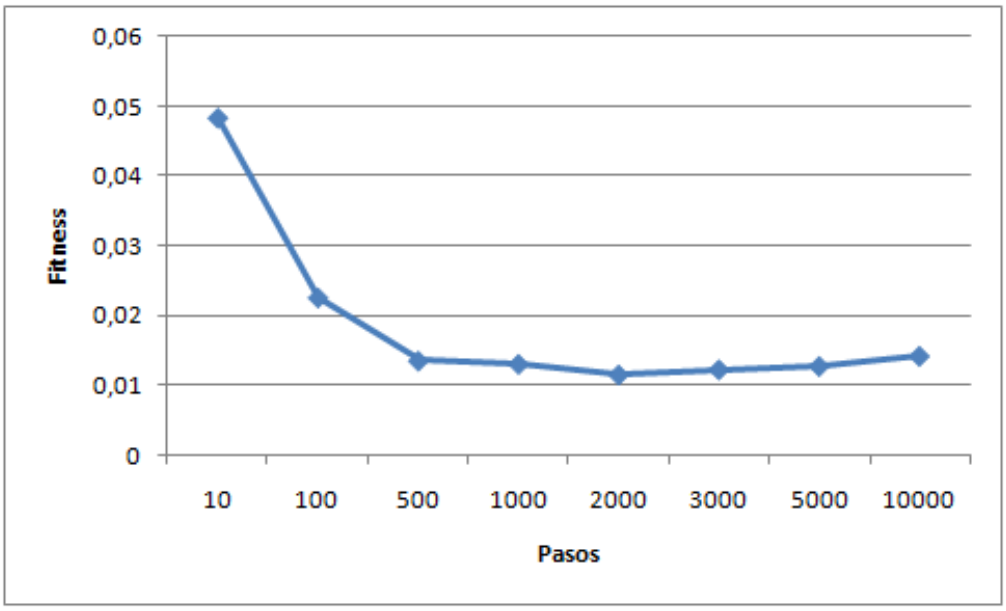


Figura 4.4: Valores de fitness obtenidos con la solución secuencial utilizando intervalos de reales, distintos números de pasos y un tiempo fijo.

Respecto a la optimización del cálculo del fitness para la mejora de Búsqueda Local, se ha obtenido el promedio del tiempo de múltiples ejecuciones de la función que hace la Búsqueda Local con distintos números de pasos en Venus y en Saturno. Los resultados obtenidos se muestran en la figura 4.5.

Se puede ver que el *tiempo es lineal en Venus y Saturno, tanto si se utiliza esta optimización como si no se utiliza*. Se puede observar mejor el tiempo lineal utilizando la optimización en la figura 4.6.

Con la optimización, Venus hace la Búsqueda Local casi 20 veces más rápido. Saturno hace la Búsqueda Local casi 17 veces más rápido.

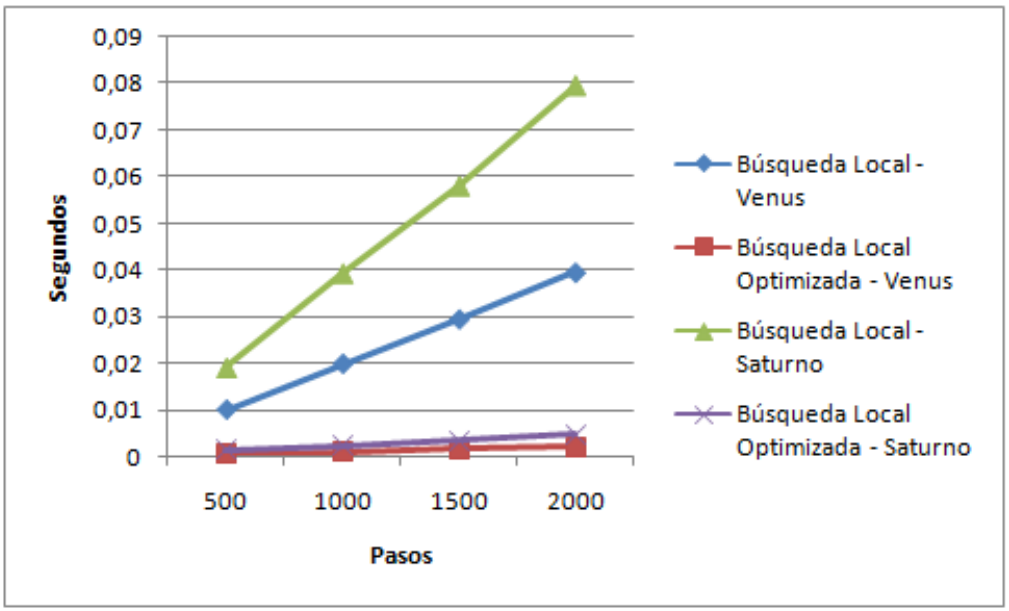


Figura 4.5: Tiempos obtenidos con la función Búsqueda Local en Venus y Saturno utilizando en el cálculo del fitness la implementación optimizada y la no optimizada, con distintos números de pasos.

Para analizar la mejora *OpenBLAS* se ha obtenido el promedio del tiempo de múltiples ejecuciones de la función de calcular fitness en Venus y Saturno utilizando problemas de distintos tamaños. Los resultados obtenidos se muestran en la tabla 4.1. Se puede observar que *la ganancia aumenta conforme lo hace el tamaño del problema*. Para el problema que se está utilizando en los experimentos, de tamaño 200, va *el doble de rápido tanto en Venus como en Saturno*.

A continuación se obtienen promedios de fitness con la solución secuencial, distintos tamaños de población, 10 ejecuciones por tamaño, un tiempo fijo y todas las mejoras. Se utiliza la versión secuencial con extensión, sin extensión, con búsqueda local y sin bús-

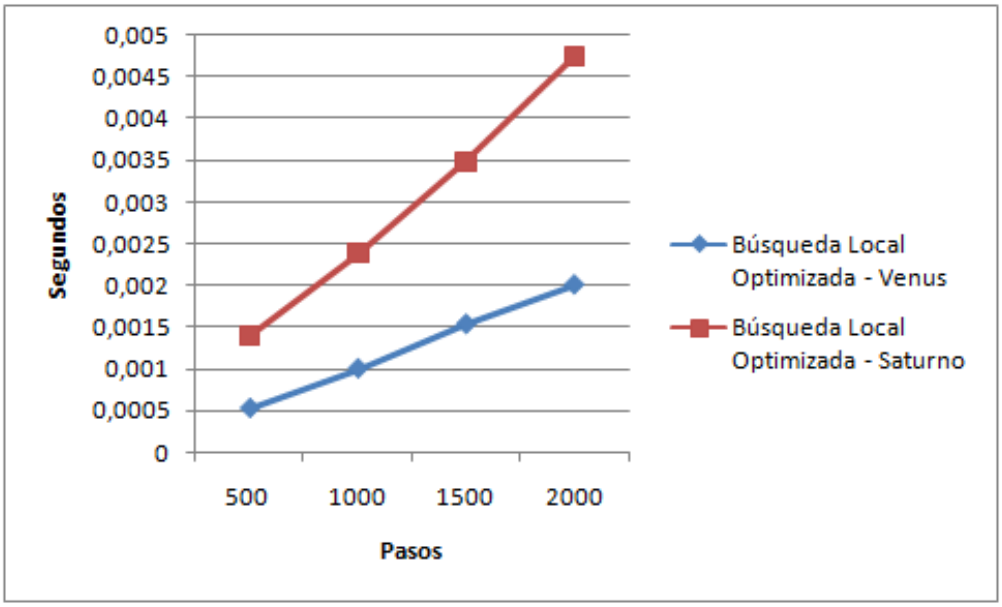


Figura 4.6: Tiempos obtenidos de la función Búsqueda Local en Venus y Saturno utilizando la implementación optimizada con distintos números de pasos.

Tamaño:	20	200	1000	5000
Venus	1.397	2.021	2.104	3.066
Saturno	1.514	2.014	2.389	3.585

Tabla 4.1: Ganancia obtenida en Venus y Saturno utilizando OpenBLAS con problemas de distintos tamaños.

queda local. Los resultados obtenidos se muestran en la figura 4.7.

Como se puede observar, el comportamiento del algoritmo genético con la extensión, sin utilizar búsqueda local, mejora bastante a la implementación sin extensión. Al utilizar búsqueda local, se obtienen mejores resultados si no se utiliza la extensión hasta un tamaño de población de entre 300 y 500. A partir de un tamaño de población de 500, se obtienen mejores resultados con la extensión. También se puede observar que al utilizar búsqueda local se mejora mucho la calidad de las soluciones.

Para analizar el comportamiento del algoritmo genético al variar el número de instantes de tiempo y el número de variables se han obtenido promedios de fitness utilizando problemas aleatorios, distintos tamaños de población, 10 ejecuciones por tamaño, un tiempo fijo y todas las mejoras. Se ha utilizado Búsqueda Local sin extensión. Los resultados, utilizando distintas longitudes de instantes de tiempo, se muestran en la figura 4.8. Los resultados, variando el número de variables, se muestran en la figura 4.9.

Se puede observar que cuantas más variables y/o instantes de tiempo se utilicen, más tiempo se necesita para obtener buenos resultados. También se sigue observando que es mejor utilizar tamaños pequeños de población.

Como conclusión, *se debe utilizar la implementación sin extensión, con búsqueda local y tamaños pequeños de población*.

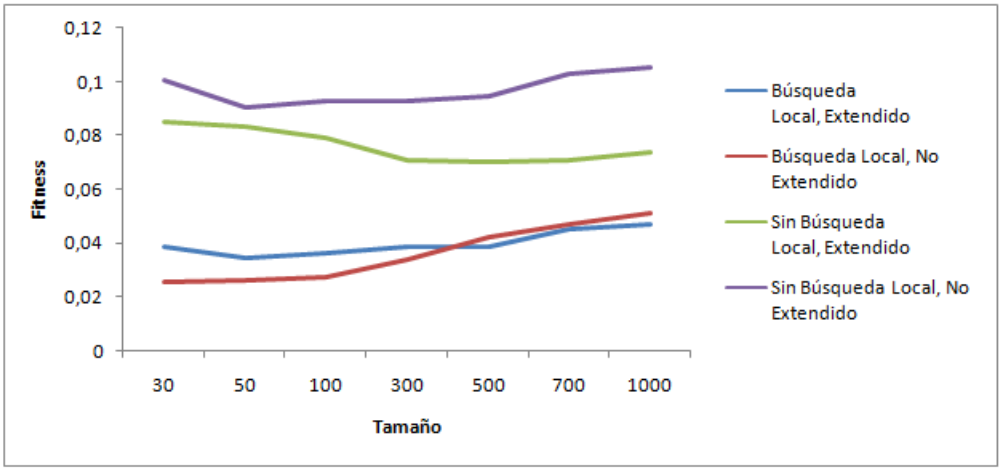


Figura 4.7: Valores de fitness obtenidos con la solución secuencial utilizando todas las mejoras, intervalos de reales y un tiempo fijo.

4.4. Explotación de la Heterogeneidad

Teniendo en cuenta que disponemos de implementaciones MPI e híbridas MPI+OpenMP y de un clúster heterogéneo (Heterosolar), analizamos la explotación de

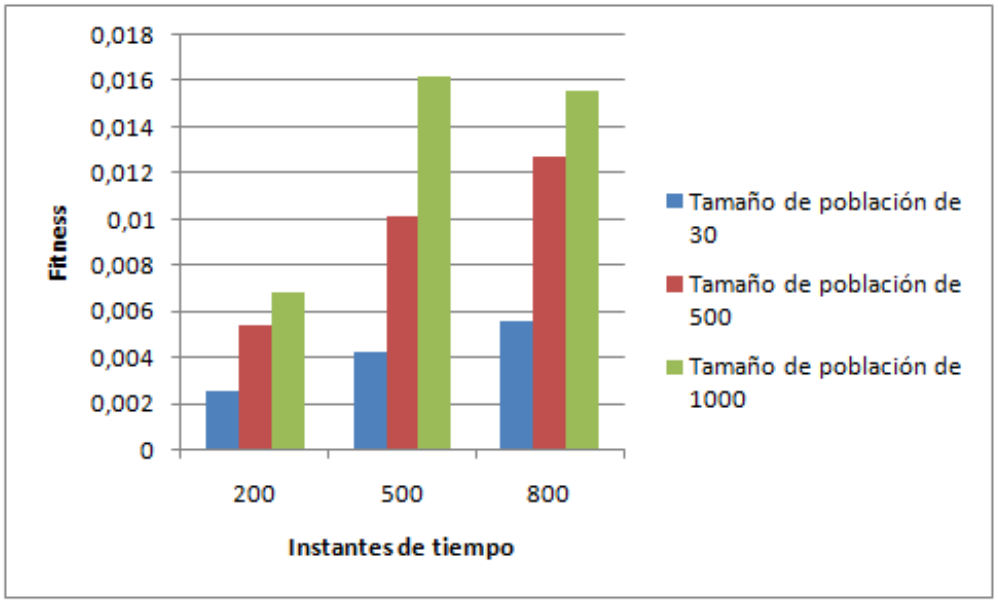


Figura 4.8: Valores de fitness obtenidos utilizando problemas aleatorios, intervalos de reales, distintas longitudes de instantes de tiempo, distintos tamaños de población y un tiempo fijo.

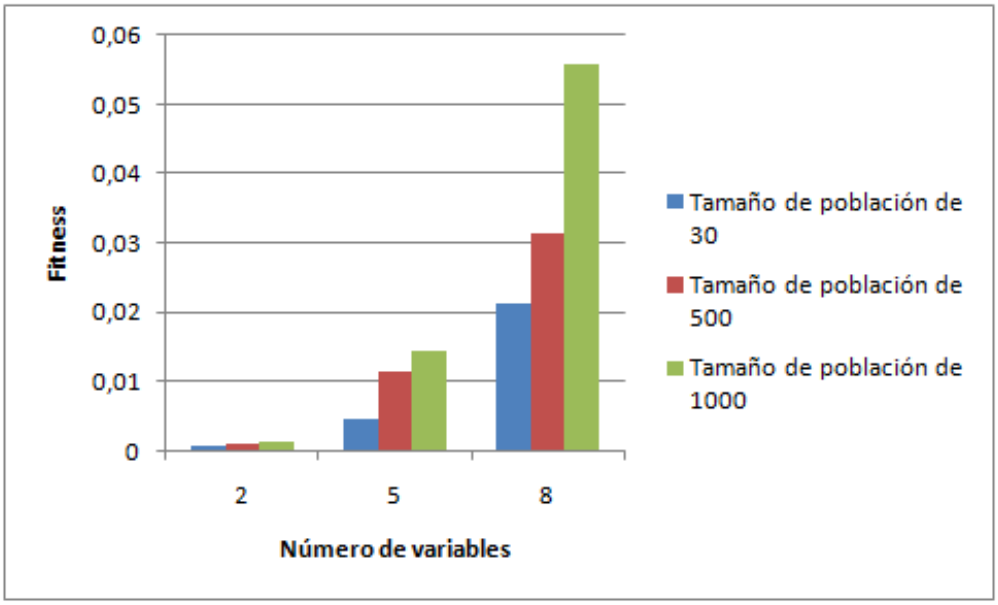


Figura 4.9: Valores de fitness obtenidos utilizando problemas aleatorios, intervalos de reales, distintas variables, distintos tamaños de población y un tiempo fijo.

la heterogeneidad del sistema computacional. Así, para finalizar, y con el análisis de las mejoras realizadas, se quiere analizar de nuevo todas las versiones. Es decir, la solución secuencial, la solución OpenMP, la solución MPI y la solución Híbrida. Además, se incluye una solución Híbrida utilizando todos los nodos (Marte, Mercurio, Saturno, Júpiter y Venus) para estudiar y explotar la heterogeneidad del clúster.

Para hacer una buena distribución de trabajo entre todos los nodos, se ha obtenido el tiempo que tarda cada nodo en realizar 100 iteraciones (utilizando todos los hilos en cada nodo).

En cada nodo se ha utilizado un tamaño de población diferente, siendo $30 * nHilosNodo$. Marte y Mercurio tardan 744 ms, Saturno 2028 ms, Júpiter 1303 ms y Venus 498 ms. Al ser Venus el que menos tarda, los tamaños se van a calcular en función del tiempo de Venus. Las proporciones quedan de la siguiente manera: 1 para Venus, $2,61$ para Júpiter $(1303/498)$, $4,07$ para Saturno $(2028/498)$ y $1,49$ para Marte y Mercurio $(744/498)$. De esta manera, si se quiere que cada hilo / proceso de Venus utilice un tamaño de población de 30, se hace la distribución de la siguiente forma: tamaño de población de 360 para Venus (30×12), 275 para Júpiter $(30 \times 24/2,61)$, 353 para Saturno $(30 \times 48/4,07)$ y 120 para Marte y Mercurio $(30 \times 6/1,49)$.

Para ver cómo influyen las migraciones se ha obtenido el promedio de múltiples ejecuciones con un número de iteraciones fijas con distintas iteraciones por generación: {10, 50, 100, 300, 600}. Los resultados obtenidos se muestran en la figura 4.10.

Como se puede observar *las migraciones no influyen mucho en el tiempo*, ya que se ha hecho una *buena distribución de trabajo* y hay solo 5 nodos. Los resultados obtenidos como el promedio del fitness de múltiples ejecuciones en un tiempo fijo y distintas iteraciones por generación, se muestran en la figura 4.11.

Se puede observar, como en el capítulo anterior, que las *soluciones son mejores al utilizar una frecuencia de migración alta*. Por lo tanto, se realizarán 10 iteraciones por generación.

De esta forma, los resultados que se han obtenido utilizando todas las versiones, dos tamaños de población, 10 ejecuciones por tamaño para obtener promedios y un tiempo fijo de 30 segundos, se muestran en la figura 4.12.

Las soluciones (secuencial, OpenMP, MPI e híbrida) se han ejecutado en Venus, por ser el nodo que mejor fitness obtiene en un tiempo fijo. Se puede observar, como en el capítulo anterior, que *las versiones paralelas obtienen soluciones bastante mejores* que la versión secuencial en el mismo tiempo. También se observa que la versión *heterogénea* obtiene resultados notablemente mejores que las demás versiones.

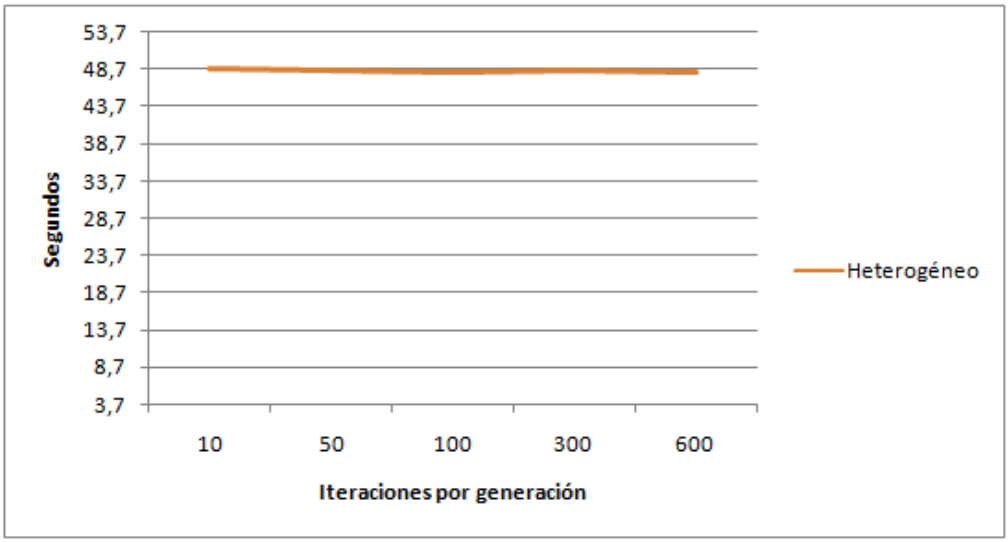


Figura 4.10: Tiempos obtenidos con todos los nodos utilizando distintas frecuencias de migración y un número fijo de iteraciones.

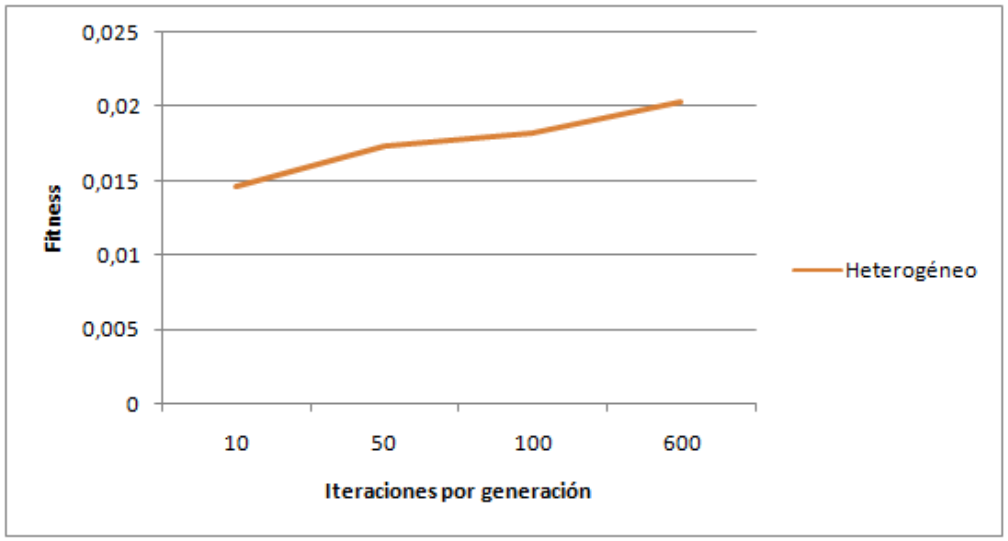


Figura 4.11: Valores de fitness obtenidos con todos los nodos utilizando distintas frecuencias de migración y un tiempo fijo.

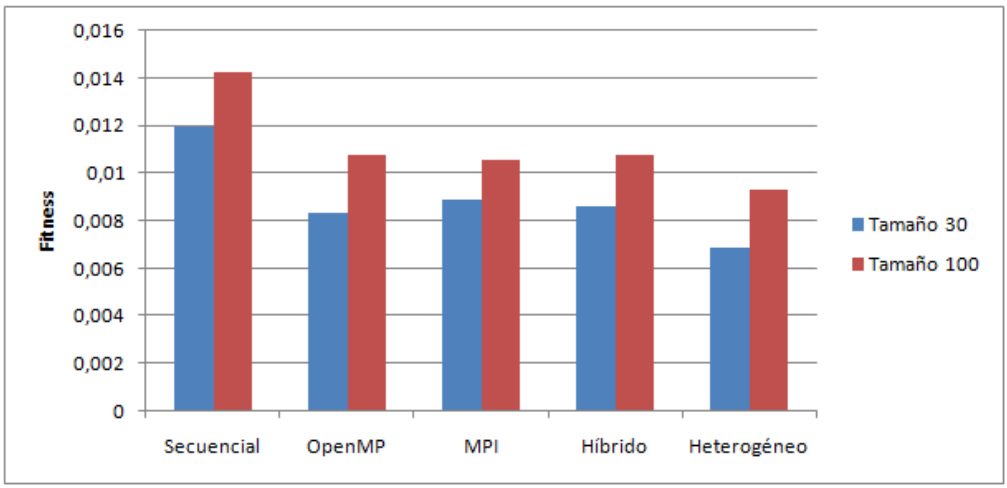


Figura 4.12: Valores de fitness obtenidos utilizando todas las versiones en un tiempo fijo de 30 segundos.

Capítulo 5

Resolución de problemas reales

En este capítulo se ha experimentado con cuatro problemas reales.

Los tres primeros problemas son conjuntos de datos de economía.

Los datos del cuarto problema son de un registro cerebral de una persona adulta.

Estos problemas solo contienen variables internas. Para todos estos problemas se ha determinado la longitud de las dependencias temporales de parámetros internos de forma experimental. Para una comparación justa, se han obtenido los modelos utilizando solo la primera mitad de la serie. De esta forma, si con ese modelo se obtiene una buena o aceptable segunda mitad de la serie, podremos decir que es un buen modelo.

Para todos los problemas se han realizado varias ejecuciones con distintas longitudes de dependencias temporales de parámetros internos y se ha calculado, para cada modelo obtenido, el fitness de la primera mitad de la serie, el fitness de la segunda mitad de la serie y el fitness total. Estas ejecuciones se han realizado con la solución heterogénea.

Se ha utilizado la condición de fin de no parar hasta que la solución deje de mejorar de forma significativa tras cinco generaciones seguidas.

5.1. Problema 1

Este problema corresponde con una serie que relaciona productividad con horas de trabajo. Tiene un total de 135 instantes de tiempo y 2 variables internas. Los resultados de las ejecuciones se muestran en la figura 5.1. Como se puede ver, lo mejor es utilizar una longitud de dependencia temporal de uno ya que el mejor modelo obtenido produce una buena segunda mitad de la serie. Con el resto de dependencias temporales se obtienen modelos que producen una segunda mitad cada vez peor.

El mejor fitness obtenido es de 0.4437 para $ni = 1$.

5.2. Problema 2

En este problema intervienen los tipos de interés a 12 meses y a 3 meses para EEUU, que se suele utilizar como indicador adelantado de una crisis económica. Tiene un total

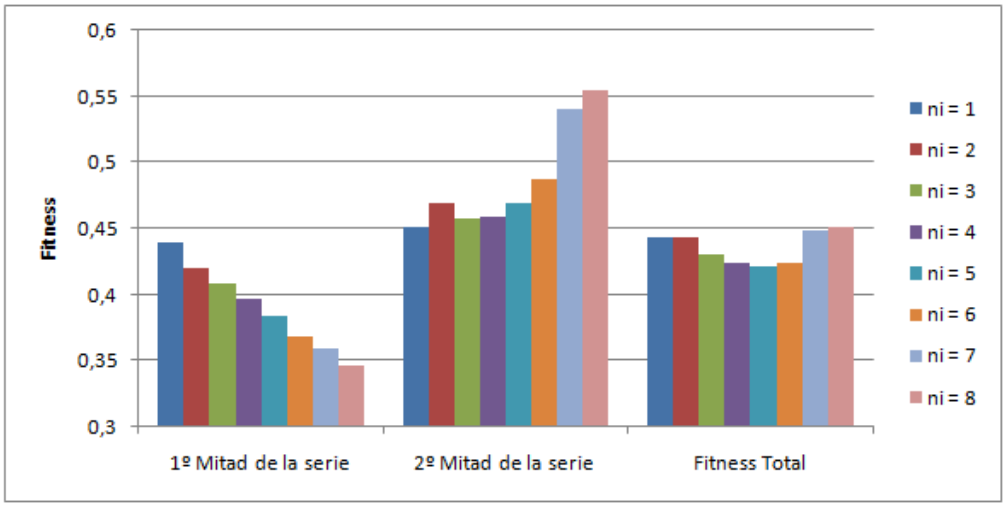


Figura 5.1: Valores de fitness obtenidos con el problema 1 y la solución heterogénea con la condición de fin de no parar hasta que la solución deje de mejorar y utilizando varias longitudes de dependencias temporales de parámetros internos.

de 628 instantes de tiempo y 2 variables internas. Los resultados de las ejecuciones se muestran en la figura 5.2. Los mejores modelos se obtienen con $ni = 12$.

El mejor fitness obtenido es de 0.4060 para $ni = 12$.

5.3. Problema 3

Este problema se utiliza para predecir el PIB. Tiene un total de 464 instantes de tiempo y 4 variables internas. Los resultados de las ejecuciones se muestran en la figura 5.3. Los mejores modelos se obtienen con $ni = 10$.

El mejor fitness obtenido es de 0.2259 para $ni = 10$.

5.4. Problema 4

Este problema, como se ha comentado, corresponde con datos de un registro cerebral de una persona adulta. Tiene un total de 4001 instantes de tiempo y 62 variables internas. Los resultados de las ejecuciones se muestran en la figura 5.4. Los mejores modelos se obtienen con $ni = 1$.

El mejor fitness obtenido es de 0.8744 para $ni = 1$. Con el resto de dependencias temporales se obtienen modelos cada vez peores.

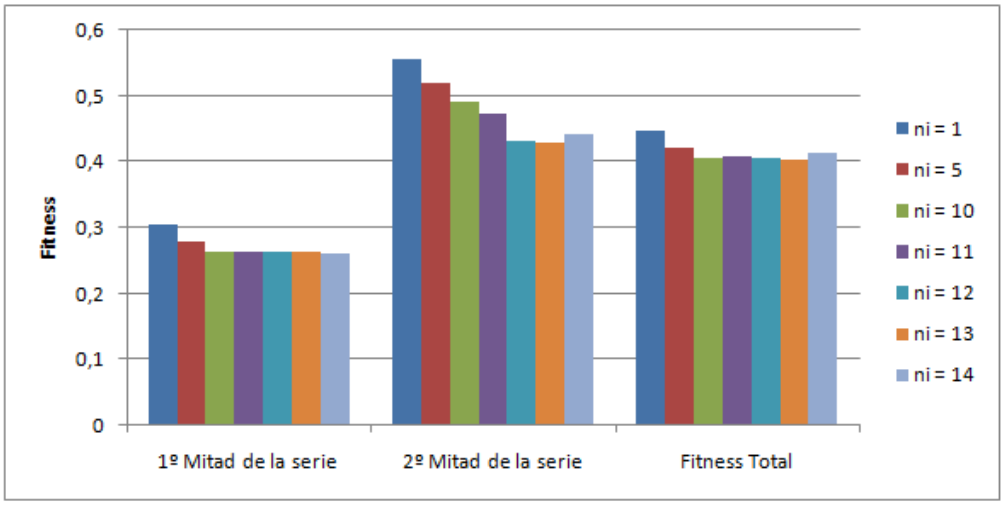


Figura 5.2: Valores de fitness obtenidos con el problema 2 y la solución heterogénea con la condición de fin de no parar hasta que la solución deje de mejorar y utilizando varias longitudes de dependencias temporales de parámetros internos.

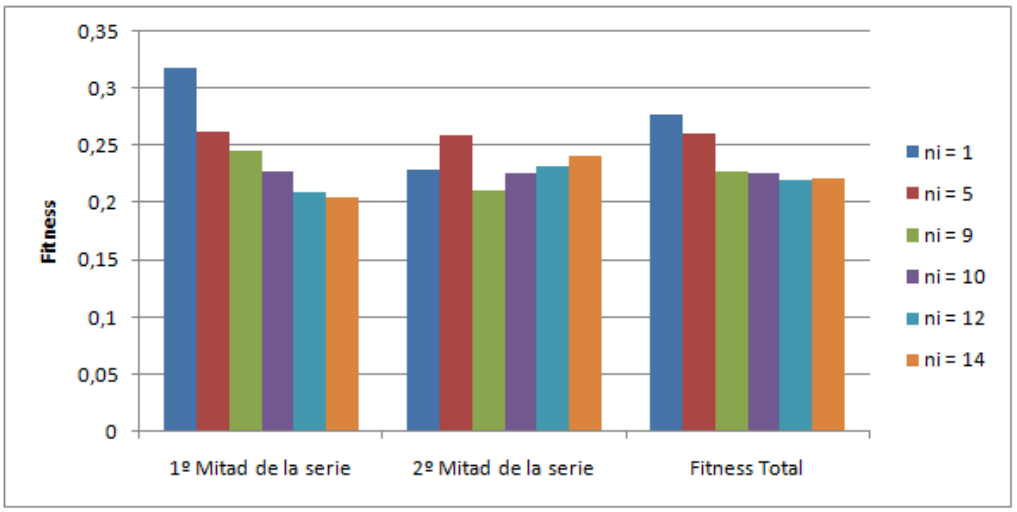


Figura 5.3: Valores de fitness obtenidos con el problema 3 y la solución heterogénea con la condición de fin de no parar hasta que la solución deje de mejorar y utilizando varias longitudes de dependencias temporales de parámetros internos.

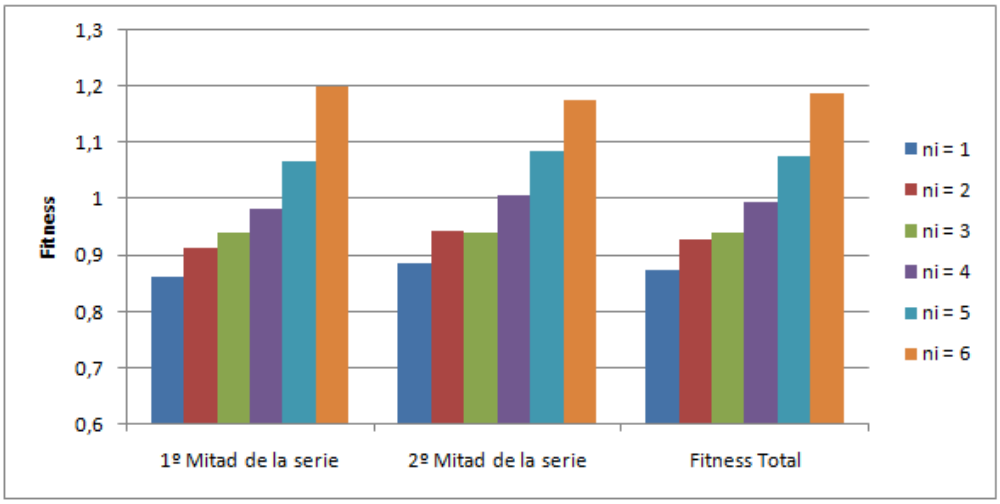


Figura 5.4: Valores de fitness obtenidos con el problema 4 y la solución heterogénea con la condición de fin de no parar hasta que la solución deje de mejorar y utilizando varias longitudes de dependencias temporales de parámetros internos.

5.5. Conclusiones generales

Los experimentos en este capítulo muestran que los programas desarrollados se pueden aplicar para analizar, para una serie de datos dada, si ésta sigue un modelo multivariante lineal.

Los programas se pueden utilizar para distintos tipos de análisis. Otras posibles aplicaciones del software serían:

- *Determinación de variables con menor influencia*. Por ejemplo, en el problema de datos neurológicos, con un número grande de variables, se puede analizar qué variable influye más negativamente en los resultados obtenidos, e ir excluyendo variables con una especie de búsqueda local mientras se obtengan mejoras significativas.

- Se pueden *considerar series cuadráticas* (o en general de otros órdenes) considerando composiciones de datos. Por ejemplo, si tenemos datos de dos variables x_1 y x_2, se pueden generar los datos de las variables $x_{11} = x_1 * x_1$, $x_{12} = x_1 * x_2$ y $x_{22} = x_2 * x_2$, obteniendo una serie con cinco variables para la que se obtendría un modelo lineal.

- Se podrían *comparar los modelos obtenidos con los usados tradicionalmente* en cada campo, y también incluir estos en la población inicial con la que trabajan los algoritmos.

En cualquier caso, el tipo de análisis a realizar y el que los resultados sean satisfactorios o no para el problema con el que se trabaja es algo que tendrían que decidir los

expertos en el campo concreto.

Los problemas del campo de economía han sido proporcionados por el profesor Máximo Cosme Camacho Alonso, del Departamento de Métodos Cuantitativos para la Economía y la Empresa de la Universidad de Murcia, y por Alberto Pérez Bernabeu, del Centro de Investigación Operativa de la Universidad Miguel Hernández de Elche. Los datos del registro cerebral han sido facilitados por Eduardo Fernández Jover, director del grupo NBIO del Instituto de Ingeniería de la Universidad Miguel Hernández de Elche, y se han conseguido a través de Jose Juan López Espín, de la misma universidad, y con el que colabora el Grupo de Computación Científica y Programación Paralela en la aplicación de técnicas computacionales para el tratamiento de datos médicos.

Capítulo 6

Conclusiones y Trabajo Futuro

Este trabajo comenzó presentando un algoritmo genético simple para modelar series temporales multivariantes. Con la implementación de sus tres versiones paralelas y, a partir de los experimentos realizados en el capítulo 3 (sin las mejoras), se han obtenido las *conclusiones* que se muestran a continuación:

1. Los resultados obtenidos de los experimentos utilizando diferentes probabilidades de cruce muestran la tendencia de que *cuantos más cruces se realicen mejores soluciones se obtienen*. Esto es lógico ya que se evalúan más individuos.

   Por otro lado, los resultados utilizando diferentes probabilidades de mutación muestran la tendencia de que *cuantas menos mutaciones se realicen mejor serán las soluciones*.

2. A partir de los experimentos de tiempo y de migraciones, se han obtenido varias conclusiones.

   Una conclusión es que *el tiempo aumenta de forma lineal en función del tamaño de la población*.

   La otra conclusión es que las versiones paralelas obtienen, utilizando p procesadores, casi un *Speed-Up de factor* p.

   Dependiendo del número de cores, el Speed-Up fluctúa debido a las migraciones. Es decir, cuantos más procesadores se utilicen, más se notará el impacto de las migraciones en el tiempo. Esto no es muy importante ya que, para un tiempo fijo, se obtienen mejores soluciones utilizando una frecuencia de migración alta.

3. Para todas las versiones del algoritmo genético, los resultados obtenidos de los experimentos de fitness muestran que *un tamaño de población entre 30 y 1000 es suficiente para obtener buenos resultados*. A partir de 1000 la calidad de las soluciones empeora muy rápidamente.

   También se obtiene la conclusión de que al *utilizar más hilos / procesos la calidad de las soluciones mejora* notablemente en el mismo tiempo respecto a la solución secuencial.

Las *conclusiones* que se han obtenido al utilizar las *diferentes mejoras* implementadas han sido las siguientes:

1. El *mejor operador de cruce es el cruce aleatorio* por su tendencia a obtener mejores soluciones en un tiempo fijo.

2. *No se han observado diferencias significativas entre los nuevos métodos de selección*, por lo que una buena opción es utilizar selección por torneo.

3. La *hibridación con Búsqueda Local mejora de manera sustancial las calidades de las soluciones* y, gracias a la *optimización del cálculo del fitness* para esta mejora, el orden del tiempo en el cálculo del fitness de los vecinos de un elemento pasa de $O(n^3)$ a $O(n)$, reduciendo el tiempo de ejecución del algoritmo de manera importante.

4. El uso de la librería *OpenBLAS reduce de manera notable el tiempo de ejecución* necesario para calcular el fitness. Además, cuanto mayor sea el tamaño del problema, mayor será la ganancia obtenida.

5. Los beneficios que se obtienen al utilizar elitismo son buenos si no se utiliza la mejora de Búsqueda Local. Al utilizar Búsqueda Local es mejor *no utilizar elitismo y usar tamaños de poblaciones entre 30 y 100*.

Las *conclusiones* que se han obtenido al *comparar todas las versiones* del algoritmo genético han sido las siguientes:

1. Como se ha comentado anteriormente y, en un tiempo fijo, las calidades de las soluciones de las *versiones paralelas mejoran* a las calidades de las soluciones de la versión secuencial de manera importante.

2. Las versiones *OpenMP, MPI e híbrida son muy parecidas en todos los aspectos*, al usarse en todos los casos un esquema de islas.

3. La *versión híbrida se puede utilizar para explotar la heterogeneidad* en clústers heterogéneos. Experimentos en el clúster Heterosolar muestran que de esta manera se pueden mejorar las soluciones que se encuentran para un tiempo de ejecución fijo.

En resumen, como *conclusión general*, la mejor opción es utilizar:

- La versión híbrida utilizando todos los nodos disponibles del clúster.
- Búsqueda Local (y su optimización) sin elitismo.
- La librería OpenBLAS o similar.
- Cruce aleatorio y selección por torneo.
- Tamaños de población entre 30 y 100.
- Una probabilidad de cruce de 0.9.
- Una probabilidad de mutación de 0.005.

Este trabajo tiene aún *margen de mejora*. A continuación se muestran algunas líneas de trabajo futuras:

- Desarrollar implementaciones *GPU* o *Xeon Phi* (o utilizar la implementación GPU que ha desarrollado el Grupo de Computación Científica y Programación Paralela) y combinarlas con la versión híbrida MPI+OpenMP para la explotación de todos los recursos computacionales de un clúster.
- Desarrollar *otras metaheurísticas*, analizarlas de forma similar a como se ha hecho para algoritmos genéticos, y comparar los resultados obtenidos con las distintas metaheurísticas.
- Se ha mostrado con algunos experimentos que el software desarrollado se puede aplicar a series de datos de problemas reales. Un trabajo interesante sería analizar con expertos en diferentes casos la utilización de este software en sus problemas.

Bibliografía

[1] Domingo Giménez. Perspectivas en la aplicación de métodos computacionales a la resolución y obtención de modelos de series temporales multivariantes. Comunicación personal, no publicado.

[2] G. Golub and C. F. Van Loan. *Matrix Computations*. The John Hopkins University Press, fourth edition, 2013.

[3] Bernhard Pfaff. VAR, SVAR and SVEC models: Implementation within R package vars. `https://cran.r-project.org/web/packages/vars/vignettes/vars.pdf`.

[4] Ross Taylor. PyFlux: An open source time series library for Python. `https://github.com/rjt1990/pyflux`, 2016.

[5] Stata: Software for statistics and data science. `https://www.stata.com/`.

[6] EViews. `https://www.eviews.com/home.html`.

[7] G. R. Raidl. A unified view on hybrid metaheuristics. In *Hybrid Metaheuristics, Third International Workshop, LNCS*, volume 4030, pages 1–12, October 2006.

[8] Enrique Alba. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, New York, 2005.

[9] Domingo Giménez. Aplicaciones científicas de la computación paralela. `http://dis.um.es/~domingo/apuntes/PPCAP/1617/Aplicaciones.pdf`.

[10] Alfonso L. Castaño, Javier Cuenca, Jose Matias Cutillas Lozano, Domingo Giménez, Jose-Juan López-Espín, and Alberto Pérez-Bernabeu. Parallelism on hybrid metaheuristics for vector autoregression models. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*, pages 828–835, 2018.

[11] OpenMP. `https://www.openmp.org/`.

[12] OpenMPI: Open source high performance computing. `https://www.open-mpi.org/`.

[13] Mervyn Márquez Gómez. Las metaheurísticas: tendencias actuales y su aplicabilidad en la ergonomía. `https://www.redalyc.org/html/2150/215037911009/`.

[14] Domingo Giménez. Algoritmos genéticos paralelos. `http://dis.um.es/~domingo/apuntes/ProgPar/trasparencias/T7.Alg.AlgoGene.ppt`.

[15] Algoritmos genéticos. http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf.

[16] MPI Tutorial. MPI Scatter, Gather, and Allgather. http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/.

[17] Centro de inteligencia artificial. http://www.aic.uniovi.es/ssii/tutorial/seleccion.htm.

[18] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/.

[19] OpenBLAS. https://www.openblas.net/.

[20] Intel® Math Kernel Library. https://software.intel.com/en-us/mkl.

[21] Engineering and scientific subroutine library. https://www.ibm.com/support/knowledgecenter/en/SSFHY8/essl_welcome.html.

[22] Siddharth Gopal. Fast Matrix Multiply and ML. http://gcdart.blogspot.com/2013/06/fast-matrix-multiply-and-ml.html.