MPCOTool (the Multi-Purposes Calibration and Optimization Tool): un software libre para obtener parámetros empíricos necesarios en modelos de simulación

Autores del software: J. Burguete y B. Latorre
Autor del manual: J. Burguete

28 de noviembre de 2016

Índice general

Capítulo 1

Compilando el código fuente

El código fuente en MPCOTool está escrito en lenguaje C. El programa ha sido compilado y probado en los siguientes sistemas operativos:

- Debian Hurd, kFreeBSD y Linux 8;
- DragonFly BSD 4.6;
- Dyson Illumos;
- Fedora Linux 25;
- FreeBSD 11.0;
- Microsoft Windows 7[1], 8.1[1] y 10[1];
- NetBSD 7.0;
- OpenBSD 6.0;
- OpenIndiana Hipster;
- OpenSUSE Linux Tumbleweed;
- y Ubuntu Linux 16.10.

Es probable que también puede compilarse y funcione en otros sistemas operativos, otras distribuciones de software y otras versiones pero no ha sido probado.

Para generar el fichero ejecutable a partir de código fuente, un compilador de C (GCC [2016] o Clang [2016]), los sistemas de configuración Autoconf [2016], Automake [2016] y Pkg-config [2016], el programa de control de la creación de ejecutables GNU-Make [2016] y las siguienres librerías de software libre son necesarias:

- Libxml [2016]: Librería requerida para leer el fichero principal de entrada en formato XML.

[1] Windows 7, Windows 8.1 y Windows 10 son marcas registradas de Microsoft Corporation.

- GSL [2016]: Librería científica usada para generar los números pseudo-aleatorios usados por los algoritmos genético y de Monte-Carlo.
- GLib [2016]: Librería requerida para analizar las plantillas de los ficheros de entrada y para implementar algunos tipos de datos, funciones útiles y rutinas usadas para paralelizar la carga computacional en los diferentes procesadores de la máquina.
- JSON-GLib [2016]: Librería usada para leer el fichero principal de entrada en formato JSON.
- GTK+3 [2016]: Librería opcional usada para dibujar la interfaz gráfica interactiva.
- OpenMPI [2016] o MPICH [2016]: Librerías opcionales. Cuando están instaladas en el sistema una de ellas es usada para permitir la paralelización del cálculo en múltiples computadoras.

Las indicaciones proporcionadas en Install-UNIX [2016] pueden seguirse para instalar todas estas utilidades.

En OpenBSD 6.0, antes de generar el código, deben seleccionarse versiones adecuadas de Autoconf y Automake haciendo en un terminal:

```
> export AUTOCONF_VERSION=2.69 AUTOMAKE_VERSION=1.15
```

En sistemas Windows hay que instalar MSYS2 (http://sourceforge.net/projects/msys2) y las librerías y utilidades requeridas. Para ello se pueden seguir las instrucciones detalladas en https://github.com/jburguete/install-unix.

En Fedora Linux 25, para usar OpenMPI hay que hacer en un terminal (en la versión de 64 bits):

```
> export PATH=$PATH:/usr/lib64/openmpi/bin
```

En FreeBSD 11.0, debido en un extraño error en la versión por defecto de gcc, hay que hacer en un terminal:

```
> export CC=gcc5 (or CC=clang)
```

Una vez que todas las utilidades necesarias han sido instaladas, hay que descargar el código de Genetic. Luego puede compilarse haciendo en un terminal:

```
> git clone https://github.com/jburguete/genetic.git
> cd genetic/2.0.1
> ./build
```

El siguiente paso es descargar el código fuente de MPCOTool, enlazarlo con el de Genetic y compilar todo junto haciendo:

```
> git clone https://github.com/jburguete/mpcotool.git
> cd mpcotool/3.0.4
> ln -s ../../genetic/2.0.1 genetic
> ./build
```

En sevidores o en clústers, en los que una versión sin interfaz gráfica paralelizada con MPI es lo más conveniente, reemplácese el script *build* por:

```
> ./build_without_gui
```

Opcionalmente, si se quieren compilar los tests con las funciones analíticas de optimización estándar, hay que hacer (los ejecutables de test2, test3 y test4 usan también la librería *Genetic*):

```
> cd ../tests/test2
> ln -s ../../../genetic/2.0.1 genetic
> cd ../test3
> ln -s ../../../genetic/2.0.1 genetic
> cd ../test4
> ln -s ../../../genetic/2.0.1 genetic
> cd ../../3.0.4
> make tests
```

Finalmente podemos construir los manuales en formato PDF haciendo:

```
> make manuals
```

Capítulo 2

Interfaz

¡OJO! La representación de los números reales se hace, según el estándar internacional, separando los decimales mediante el "." decimal.

2.1. Formato en línea de comandos

En esta sección hemos marcado entre corchetes los argumentos opcionales.

- La línea de comandos en modo secuencial es (donde X es el número de hilos a ejecutar paralelamente y S una semilla para el generador de números pseudo-aleatorios):

```
> ./mpcotoolbin [-nthreads X] [-seed S] fichero_de_entrada.xml
[fichero_de_resultados] [fichero_de_variables]
```

- La línea de comandos en modo paralelizado en diferentes computadoras con MPI es (donde X es el número de hilos a ejecutar en cada nodo y S una semilla para el generador de números pseudo-aleatorios):

```
> mpirun [MPI options] ./mpcotoolbin [-nthreads X] [-seed S]
fichero_de_entrada.xml [fichero_de_resultados]
[fichero_de_variables]
```

- La sintaxis del programa simulador ha de ser:

```
> ./simulador fichero_de_entrada_1 [fichero_de_entrada_2] [...]
fichero_de_salida
```

  Hay dos opciones para el fichero de salida. Puede comenzar con un número que indique el valor de la función objetivo o puede ser un fichero de resultados que tiene que ser evaluado por un programa externo (el evaluador) comparando con un fichero de datos experimentales.

- En caso de la última opción del punto anterior, la sintaxis del programa para evaluar la función objetivo tiene que ser (donde el fichero de resultados debe comenzar con el valor de la función objetivo):

```
> ./evaluador fichero_simulado fichero_experimental
fichero_de_resultados
```

- En systemas de tipo UNIX, la aplicación interactiva puede abrirse haciendo en un terminal:

```
> ./mpcotool
```

2.2. Aplicación con interfaz gráfica de usuario interactiva

Una forma alternativa de usar el programa consiste en usar la aplicación con interfaz gráfica de usuario interactiva, llamada *MPCOTool*. En esta aplicación la paralelización en diferentes computadoras usando OpenMPI o MPICH está desactivada, esta paralelización sólo puede usarse en modo de comandos. En la figura 2.1 se muestra una figura con la ventana principal de la utilidad. Desde esta ventana podemos acceder a cada variable, coeficiente, algoritmo y programa de simulación requerido.

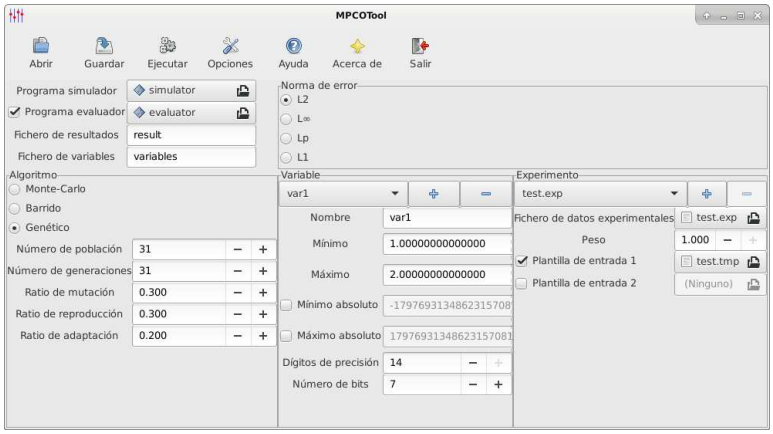

Figura 2.1: Ventana principal de la aplicación con interfaz gráfica de usuario interactiva de MPCOTool.

Los resultados óptimos se presentan finalmente en un cuadro de diálogo como el mostrado en la figura 2.2.

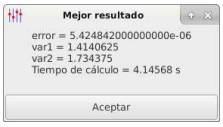

Figura 2.2: Cuadro de diálogo de los resultados óptimos de la aplicación con interfaz gráfica de usuario interactiva de MPCOTool.

2.3. Ficheros de entrada

2.3.1. Fichero de entrada principal

Este fichero puede escribirse en formato XML con una estructura arbórea como la representada en la figura 2.3.

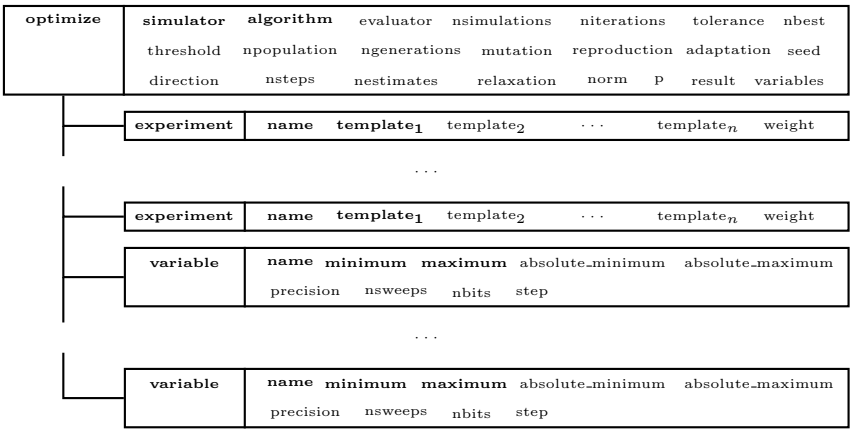

Figura 2.3: Estructura del fichero principal de entrada. Nodos y propiedades imprescindibles están en negrita. No obstante, otras propiedades también pueden ser necesarias en función del algoritmo de optimización seleccionado.

El nodo XML principal tiene que comenzar con la etiqueta "*optimize*". Las propiedades que se pueden definir son:

simulator : indica el programa simulador.

evaluator : opcional. Especifica el programa de evaluación en caso de ser requerido.

algorithm : fija el algoritmo de optimización. Actualmente tres métodos están disponibles:

sweep : algoritmo de fuerza bruta de barrido. Requiere definir en cada variable:

nsweeps : número de barridos para la variable en cada experimento.

Monte-Carlo : algoritmo de fuerza bruta de Monte-Carlo. Requiere definir en el nodo XML principal:

nsimulations : número de simulaciones a ejecutar en cada iteración y en cada experimento.

genetic : algoritmo genético. Necesita definir los siguientes parámetros en el nodo XML principal:

npopulation : número de individuos de la población.

ngenerations : número de generaciones.

mutation : ratio de mutación.

reproduction : ratio de reproducción.

adaptation : ratio de adaptación.

Además para cada variable:

nbits : número de bits para codificar la variable.

niterations : número de iteraciones (valor por defecto: 1) para realizar el algoritmo iterativo,

nbest : número de mejores simulaciones con las que calcular el siguiente intervalo de convergencia en la siguiente iteración del algoritmo iterativo (valor por defecto: 1),

tolerance : parámetro de tolerancia para relajar el intervalo de convergencia del algoritmo iterativo (valor por defecto: 0),

threshold : umbral en la función objetivo para detener la optimización,

seed : semilla del generador de números pseudo-aleatorios (valor por defecto 7007),

direction : método de buscar la dirección óptima (opcional para los algoritmos de barrido y de Monte-Carlo). Dos valores están disponibles actualmente:

coordinates : búsqueda por descenso de coordenadas,

random : búsqueda aleatoria. Requiere:

nestimates : número de pruebas aleatorias para buscar la dirección óptima.

Ambos métodos requieren además los siguientes parámetros:

nsteps : número de pasos para ejecutar el método de búsqueda de dirección,

relaxation : parámetro de relajación para el método de búsqueda de dirección,

y para cada variable:

step : tamaño de paso inicial para el método de búsqueda de dirección,

norm : selecciona la norma de error (valor por defecto: "euclidian"). Actualmente se pueden escoger cuatro tipos:

euclidian : norma de error euclidiana L_2, véase (3.1),

maximum : norma de error máximo L_∞, véase (3.2),

p : norma de error P L_p. Requiere:

p : exponente de la norma de error P, véase (3.3),

taxicab : norma de error taxicab L_1, véase (3.4),

result : define el nombre del fichero de resultados óptimos. Es opcional, si no se especifica se guarda con el nombre "*result*";

variables : define el nombre del fichero donde se guardan todas las combinaciones de variables simuladas. Es opcional, si no se especifica se guarda con el nombre "*variables*".

El primer tipo de nodos XML hijos tiene que comenzar con la etiqueta "*experiment*". Especifica los datos experimentales. Contiene las propiedades:

name : nombre del fichero de datos experimentales a calibrar.

templateX : X-ésima plantilla del fichero de datos experimentales del programa de simulación.

weight : peso (valor por defecto: 1) para aplicar en la función objetivo (véase (3.1) a (3.4)).

El segundo tipo de nodos XML hijo tiene que comenzar con la etiqueta *variable*". En estos nodos se especifican los datos de las variables que se definen con las propiedades siguientes:

name : etiqueta de la variable. Para la variable X-ésima, se analizan todas las plantillas de entrada y se crean ficheros de entrada correspondientes para el programa simulador reemplazando todas las etiquetas con el formato @variableX@ por el contenido de esta propiedad.

minimum, maximum : rango de valores de las variables. El programa crea los ficheros de entrada de la simulación reemplazando todas las etiquetas @valueX@ de las plantillas de entrada por un valor en este rango para la vairable X-ésima, calculado por el algoritmo de optimización.

absolute_minimum, absolute_maximum : rango de valores permitido. En métodos iterativos, la tolerancia puede incrementar el rango inicial de valores en cada iteración. Estos valores son el rango permitido para las variables compatible con los límites del modelo.

precision : número de dígitos decimales de precisión. 0 se aplica a los números enteros.

En este formato el fichero sería de la forma:

```
<?xml version="1.0"?>
<optimize simulator="simulator_name" evaluator="evaluator_name"
    algorithm="algorithm_type" nsimulations="simulations_number"
    niterations="iterations_number" tolerance="tolerance_value"
    nbest="best_number" npopulation="population_number"
    ngenerations="generations_number" mutation="mutation_ratio"
    reproduction="reproduction_ratio" adaptation="adaptation_ratio"
    direction="direction_search_type" nsteps="steps_number"
    relaxation="relaxation_parameter" nestimates="estimates_number"
    threshold="threshold_parameter" norm="norm_type" p="p_parameter"
    seed="random_seed" result_file="result_file"
    variables_file="variables_file">
    <experiment name="data_file_1" template1="template_1_1"
        template2="template_1_2" ... weight="weight_1"/>
    ...
    <experiment name="data_file_N" template1="template_N_1"
        template2="template_N_2" ... weight="weight_N"/>
    <variable name="variable_1" minimum="min_value" maximum="max_value"
        precision="precision_digits" sweeps="sweeps_number"
        nbits="bits_number" step="step_size"/>
    ...
    <variable name="variable_M" minimum="min_value" maximum="max_value"
        precision="precision_digits" sweeps="sweeps_number"
        nbits="bits_number" step="step_size"/>
</optimize>
```

Alternativamete, este fichero también puede escribirse en formato JSON:

```
{
    "simulator": "simulator_name",
    "evaluator": "evaluator_name",
    "algorithm": "algorithm_type",
    "nsimulations": "simulations_number",
    "niterations": "iterations_number",
    "tolerance": "tolerance_value",
    "nbest": "best_number",
    "npopulation": "population_number",
    "ngenerations": "generations_number",
    "mutation": "mutation_ratio",
    "reproduction": "reproduction_ratio",
    "adaptation": "adaptation_ratio",
    "direction": "direction_search_type",
    "nsteps": "steps_number",
    "relaxation": "relaxation_parameter",
    "nestimates": "estimates_number",
    "threshold": "threshold_parameter",
    "norm": "norm_type",
    "p": "p_parameter",
    "seed": "random_seed",
    "result_file": "result_file",
    "variables_file": "variables_file",
    "experiments":
    [
        {
            "name": "data_file_1",
            "template1": "template_1_1",
            "template2": "template_1_2",
            ...
            "weight": "weight_1",
        },
        ...
        {
            "name": "data_file_N",
            "template1": "template_N_1",
            "template2": "template_N_2",
            ...
            "weight": "weight_N",
        }
    ],
    "variables":
    [
        {

            "name": "variable_1",
            "minimum": "min_value",
            "maximum": "max_value",
            "precision": "precision_digits",
            "sweeps": "sweeps_number",
            "nbits": "bits_number",
            "step": "step_size",
        },
        ...
        {
            "name": "variable_M",
            "minimum": "min_value",
            "maximum": "max_value",
            "precision": "precision_digits",
            "sweeps": "sweeps_number",
            "nbits": "bits_number",
            "step": "step_size",
        }
    ]
}
```

2.3.2. Ficheros de plantilla

Hay que generar $N_{experimentos} \times N_{entradas}$ ficheros de plantilla para reproducir cada fichero de entrada asociado a cada uno de los experimentos (véase la figura 3.1). Todos estos ficheros de plantilla son analizados sintáctimente por

MPCOTool reemplazándose las siguientes etiquetas clave para generar los ficheros de entrada del programa simulador:

@variableX@ : se reemplaza por la etiqueta asociada al X-ésima parámetro empírico definido en el fichero principal de entrada.

@valueX@ : se reemplaza por el valor asociado al X-ésima parámetro empírico calculado por el algoritmo de optimización usando los datos definidos en el fichero principal de entrada.

2.4. Ficheros de salida

2.4.1. Fichero de resultados

MPCOTool genera un fichero donde se guarda la mejor combinación de variables y su correspondiente función objetivo calculada así como el tiempo de cálculo. El nombre de este fichero puede definirse en la propiedad *result* del fichero de entrada principal. Si no se define se crea un fichero de nombre "result".

2.4.2. Fichero de variables

El programa genera también un fichero donde se guardan en columnas cada una de las combinaciones de variables probadas en la calibración, siendo además la última columna el valor de la función objetivo. El nombre de este fichero puede definirse en la propiedad *variables*. Si no se especifica esta propiedad se crea un fichero de numbre "variables".

Capítulo 3

Organización de MPCOTool

Supongamos que buscamos un conjunto de $N_{parameters}$ parámetros empíricos requeridos por un modelo de simulación que sea el que mejor ajuste el conjunto de $N_{experimentos}$ datos experimentales y que el programa simulador requiere además $N_{entradas}$ ficheros de entrada. La estructura seguida por MPCOTool se resume en el *fichero de entrada principal*, donde se especifican tanto $N_{experimentos}$ como $N_{entradas}$. También contiene los valores extremos de los parámetros empíricos y el algoritmo de optimización escogido. Entonces MPCOTool lee las corespondientes $N_{experimentos} \times N_{entradas}$ plantillas para crear los ficheros de entrada del simulador reemplazando etiquetas clave por los parámetros empíricos generados por el algoritmo de optimización. Hay dos opciones: o bien el programa simulador compara directamente los resultados de la simulación con el *fichero de datos experimentales* y genera un fichero con el valor de la función objetivo; o bien otro programa externo, definido en la propiedad *evaluator*, es ejecutado para comparar con el *fichero de datos experimentales* produciendo el valor de la función objetivo. En ambos casos el valor de la función objetivo se guarda en un *fichero de valores objetivos*. Por lo tanto, para cada experimento se obtiene un valor objetivo o_i. El valor final de la función objetivo (J) asociado al conjunto de experimentos puede calcularse de cuatro modos:

$$L_2: \quad J = \sqrt{\sum_{i=1}^{N_{experiments}} |w_i\, o_i|^2}, \tag{3.1}$$

$$L_\infty: \quad J = \max_{i=1}^{N_{experiments}} |w_i\, o_i|, \tag{3.2}$$

$$L_p: \quad J = \sqrt[p]{\sum_{i=1}^{N_{experiments}} |w_i\, o_i|^p}, \tag{3.3}$$

$$L_1: \quad J = \sum_{i=1}^{N_{experiments}} |w_i\, o_i|, \tag{3.4}$$

con w_i el peso asociado al experimento i-ésimo, que se especifica en el *fichero de entrada principal*. En la figura 3.1 puede verse un esquema de la estructura.

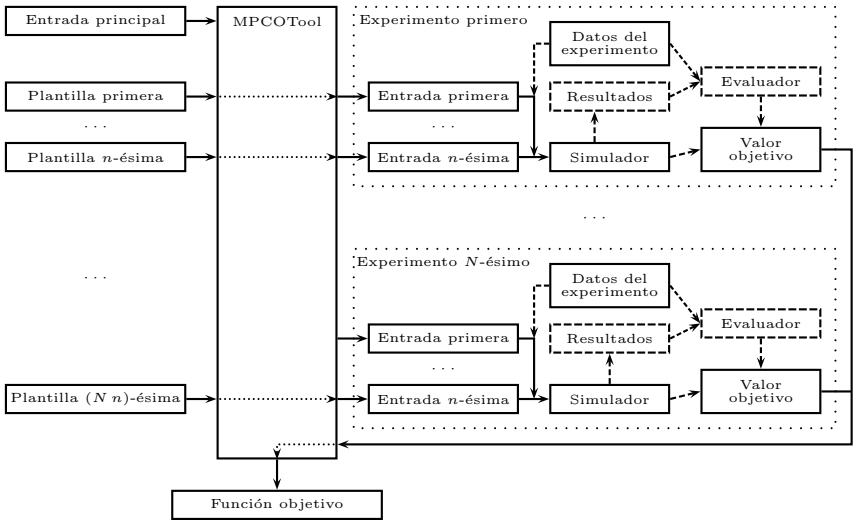

Figura 3.1: Diagrama de flujo de las interacciones entre MPCOTool, los diferentes ficheros de entrada y los programas de simulación y, en su caso, evaluación para producir el valor de la función objetivo para cada combinación de parámetros empíricos generada por el algoritmo de optimización.

El proceso completo se repite para cada combinación de parámetros empíricos generada por el algoritmo de optimización. Además, MPCOTool puede paralelizar automáticamente las simulaciones usando todos los recursos de computación disponibles en el sistema.

Capítulo 4

Métodos de optimización

A continuación se presentan los métodos de optimización implementados en MPCOTool. Se usará la siguiente notación:

$N_{simulaciones}$: número de simulaciones en cada iteración,

$N_{iteraciones}$: número de iteraciones en algoritmos iterativos,

N_{total} : número total de simulaciones.

En métodos iterativos $N_{total} = N_{simulaciones} \times N_{iteraciones}$. En métodos de fuerza bruta puros $N_{iteraciones} = 1 \Rightarrow N_{total} = N_{simulaciones}$.

4.1. Método de fuerza bruta de barrido

El método de barrido es un método de fuerza bruta que encuentra el conjunto óptimo de parámatros en una región dividiéndola en subdominios regulares. Para encontrar la solución óptima, el intervalo de dominio $x_i \in (x_{i,min}, x_{i,max})$ se define para cada variable x_i. y se subdivide en una partición regular de $N_{x,i}$ subintervalos. El número de simulaciones que se requieren es:

$$N_{simulaciones} = N_{x,1} \times N_{x,2} \times \cdots, \tag{4.1}$$

donde $N_{x,i}$ es el número de barridos en la variable x_i.

En la figura 4.1 el dominio (x, y) se define en los intervalos $x \in (x_{min}, x_{max})$ e $y \in (y_{min}, y_{max})$. Ambos intervalos x e y se dividen en 5 regiones con $N_x = N_y = 5$. El parámetro óptimo se encuentra evaluando el error de cada combinación de parámetros (x_i, y_i) requiriendo por tanto 25 evaluaciones. Nótese que el costo computacional se incrementa exponencialmente con el número de variables.

Los algoritmos de fuerza bruta presentan bajos ratios de convergencia pero son fuertemente paralelizables debido a que cada simulación es completamente independiente. Si la computadora, o el clúster de computadoras, puede ejecutar N_{tareas} tareas paralelizadas cada tarea efectúa N_{total}/N_{tareas} simulaciones, obviamente teniendo en cuenta efectos de redondeo puesto que cada tarea debe realizar un número natural de simulaciones. En la figura 4.2 se presenta un diagrama de flujo del esquema de esta paralelización. Puesto que cada tarea

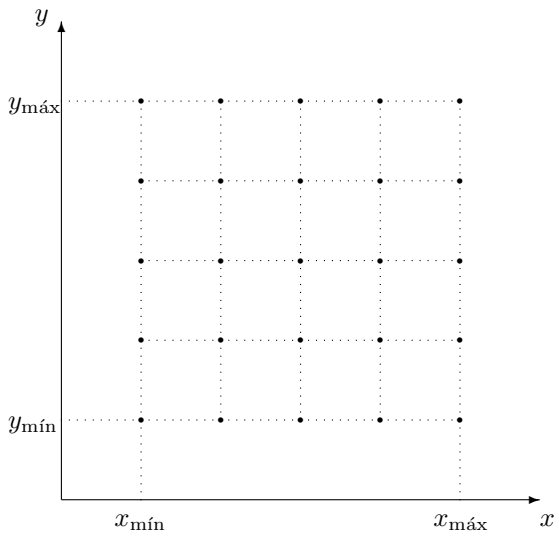

Figura 4.1: Diagrama de flujo ilustrando un ejemplo de aplicación del método de fuerza bruta de barrido con dos variables para $N_x = N_y = 5$.

es independiente, una distribución en las diferentes tareas permite explotar al máximo las capacidades computacionales de la máquina en la que se ejecuta MPCOTool.

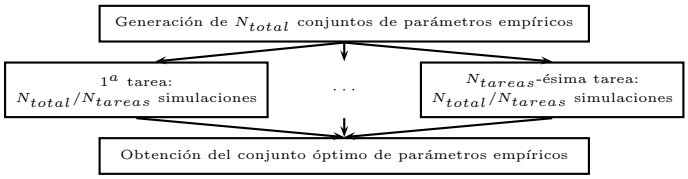

Figura 4.2: Diagrama de flujo del esquema de la paralelización en MPCOTool para métodos de fuerza bruta (barrido y Monte-Carlo).

4.2. Método de Monte-Carlo

Los métodos de Monte-Carlo realizan las simulaciones usando valores aleatorios de las varaibles suponiendo una probabilidad uniforme dentro de un intervalo permitido de valores. En la figura 4.3 se muestra la estructura de un ejemplo usando dos variables.

El método de Monte-Carlo es también fácilmente paralelizable siguiendo una estrategia como la del diagrama de flujo representado en la figura 4.2.

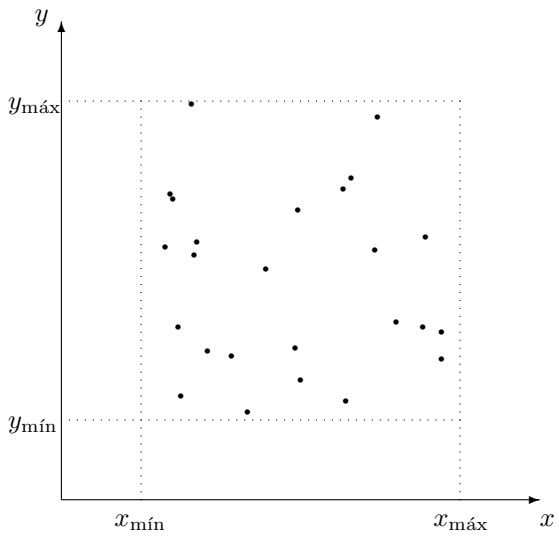

Figura 4.3: Diagrama illustrativo de un método de fuerza bruta de Monte-Carlo con dos variables and $N_{simulaciones} = 25$.

4.3. Algoritmo iterativo aplicado a los métodos de fuerza bruta

MPCOTool permite iterar ambos métodos de fuerza bruta, Monte-Carlo o barrido, para mejorar la convergencia. En este algoritmo se seleccionan un conjunto de N^j_{best} mejores resultados en cada iteración j-ésima y se usan para forzar nuevos intervalos para la búsqueda de valores óptimos en la siguiente iteración. Definiendo:

$x^b_{\text{máx}} = \underset{i \in N_{best}}{\text{máx}}\ x^j_i$: Valor máximo de la variable x en el subconjunto de N^j_{best} mejores resultados de la iteración j-ésima.

$x^b_{\text{mín}} = \underset{i \in N_{best}}{\text{máx}}\ x^j_i$: Valor mínimo de la variable x en el subconjunto de N^j_{best} mejores resultados de la iteración j-ésima.

Entonces se usará el nuevo intervalo para la variable x para buscar valores óptimos en la siguiente $(j+1)$-ésima iteración:

$$x^{j+1}_i \in \left[x^{j+1}_{\text{mín}},\ x^{j+1}_{\text{máx}}\right], \tag{4.2}$$

con:

$$\text{Sweep } \Rightarrow \begin{cases} x^{j+1}_{\text{máx}} = x^b_{\text{máx}} + \dfrac{x^j_{\text{máx}} - x^j_{\text{mín}}}{N_x - 1}\, tolerance, \\ x^{j+1}_{\text{mín}} = x^b_{\text{mín}} - \dfrac{x^j_{\text{mín}} - x^j_{\text{mín}}}{N_x - 1}\, tolerance, \end{cases}$$

$$\text{Monte} - \text{Carlo } \Rightarrow \begin{cases} x^{j+1}_{\text{máx}} = \dfrac{x^b_{\text{máx}} + x^b_{\text{mín}} + \left(x^b_{\text{máx}} - x^b_{\text{mín}}\right)(1 + tolerance)}{2}, \\ x^{j+1}_{\text{mín}} = \dfrac{x^b_{\text{máx}} + x^b_{\text{mín}} - \left(x^b_{\text{máx}} - x^b_{\text{mín}}\right)(1 + tolerance)}{2}, \end{cases} \tag{4.3}$$

siendo *tolerance* un factor que incremente el tamaño del intervalo de variables a simular en la siguiente iteración. En la figura 4.4 se muestra un esquema del procedimiento que usa el algoritmo iterativo para modificar los intervalos de variables para mejorar la convergencia.

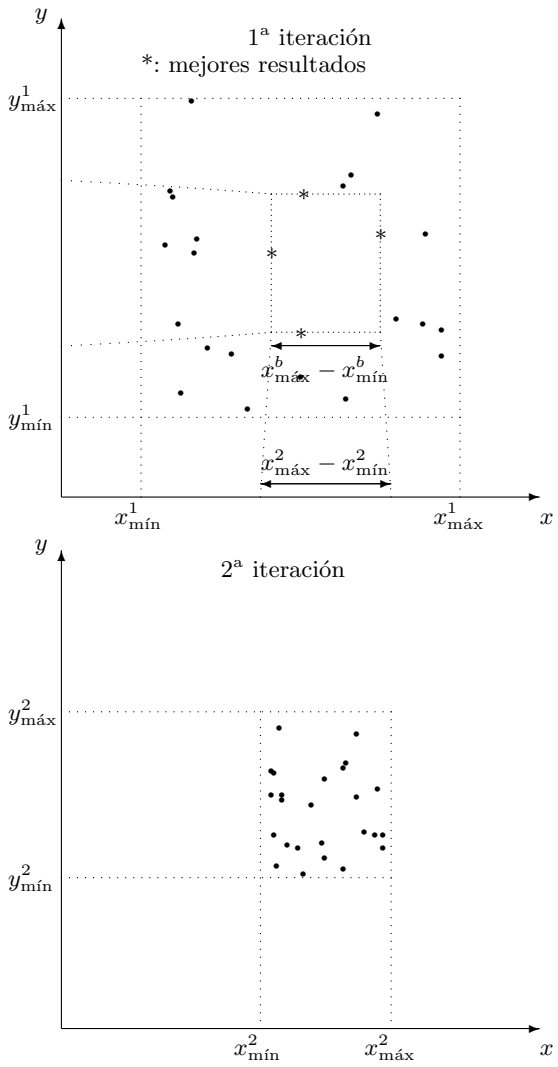

Figura 4.4: Diagrama representando un ejemplo del algoritmo iterativo aplicado al método de fuerza bruta de Monte-Carlo con dos variables para $N_{simulaciones} = 25$, $N_{best} = 4$ y dos iteraciones.

El algoritmo iterativo puede ser también fácilmente paralelizdo. No obstante, este método es menos paralelizable con los de fuerza bruta puros puesto que la paralelización debe ejecutarse en cada iteración (véase un diagrama de flujo en la figura 4.5).

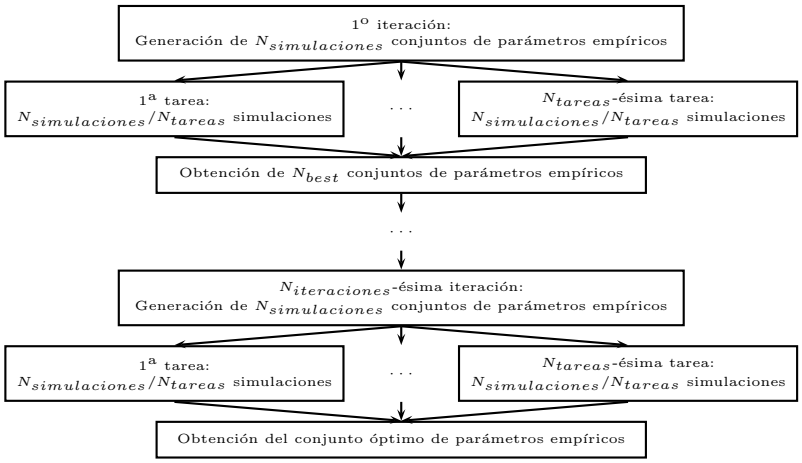

Figura 4.5: Diagrama de flujo del esquema de paralelización seguido en MPCOTool para el método iterativo.

4.4. Método de búsqueda de dirección

Los métodos de fuerza bruta, barrido y Monte-Carlo, pueden combinarse también con un algoritmo de búsqueda de dirección. Definiendo el vector $\vec{r}_i$ como la combinación óptima de variables obtenida en el paso i-ésimo, $\vec{r}_1$ como la combinación óptima de variables obtenida por el método de fuerza bruta y definiendo el vector $\vec{s}_i$ como:

$$\vec{s}_1 = \vec{0}, \qquad \vec{s}_i = (1 - relaxation)\,\vec{s}_{i-1} + relaxation\,\Delta\vec{r}_{i-1}, \tag{4.4}$$

con $\Delta\vec{r}_{i-1} = \vec{r}_i + \vec{r}_{i-1}$ y $relaxtion$ el parámetro de relajación, el método de búsqueda de dirección prueba $N_{estimaciones}$ combinaciones de variables y elige el óptimo como:

$$\vec{r}_{i+1} = \text{optimo}\left(\vec{r}_i,\ \vec{r}_i + \vec{s}_i + \vec{t}_j\right),\ j = 1, \cdots, N_{estimaciones}. \tag{4.5}$$

Si en un paso no se mejora el óptimo ($\vec{r}_i = \vec{r}_{i+1}$) entonces las componentes del vector de dirección $\vec{t}_j$ se dividen por dos y las $\vec{s}_{i+1}$ se anulan. El método se itera N_{pasos} veces.

Aunque teóricamente el método de búsqueda de dirección es el de convergencia más rápida, también es el método de los implementados en MPCOTool que menores ventajas obtiene de la paralelización. El método el casi secuencial y la paralelización sólo puede hacerse en cada paso en las $N_{estimaciones}$ simulaciones necesarias para estimar la dirección óptima. El la figura 4.6 se muestra un diagrama de flujo del esquma de paralelización seguido para este método.

MPCOTool usa dos métodos para construir los vectores $\vec{t}_j$:

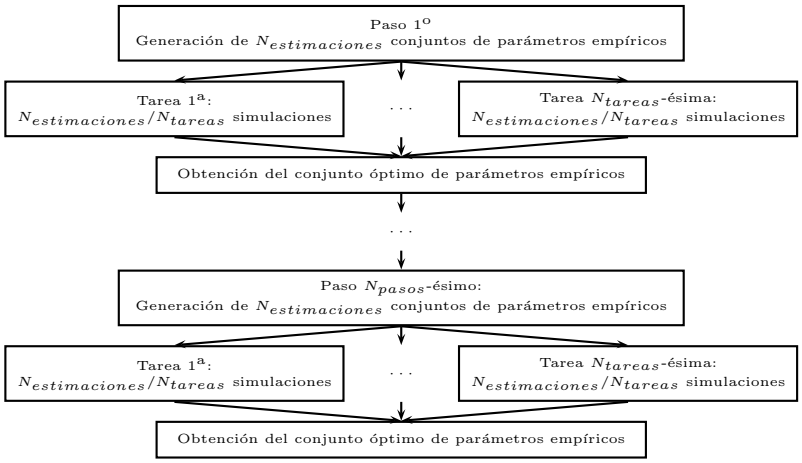

Figura 4.6: Diagrama de flujo del esquema de paralelización seguido en MPCO-Tool para el método de búsqueda de dirección.

4.4.1. Descenso de coordenadas

El método genera los vectores $\vec{t}_j$ incrementando o decreciendo una sóla variable:

$$\vec{t}_1 = \begin{pmatrix} paso_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \vec{t}_2 = \begin{pmatrix} -paso_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \vec{t}_3 = \begin{pmatrix} 0 \\ paso_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

$$\vec{t}_4 = \begin{pmatrix} 0 \\ -paso_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \cdots, \quad \vec{t}_{N_{estimaciones}} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ -paso_{N_{variables}} \end{pmatrix}, \tag{4.6}$$

siendo $paso_j$ el tamaño de paso inicial para la variable j-ésima definido por el usuario en el fichero principal de entrada. El número total de estimaciones depende del número de variables:

$$N_{estimaciones} = 2\, N_{variables} \tag{4.7}$$

4.4.2. Aleatorio

Los vectores $\vec{t}_j$ se construyen aleatoriamente:

$$\vec{t}_j = \begin{pmatrix} (1 - 2\, r_{j,1})\ paso_1 \\ \vdots \\ (1 - 2\, r_{j,k})\ paso_k \\ \vdots \\ (1 - 2\, r_{j,N_{variables}})\ paso_{N_{variables}} \end{pmatrix}, \tag{4.8}$$

con $r_{j,k} \in [0, 1)$ números aleatorios.

En la figura 4.7 se presenta un esquema para un sistema con dos variables para ilustrar el modo de funcionamiento de los algoritmos de descenso de coordenadas y aleatorio.

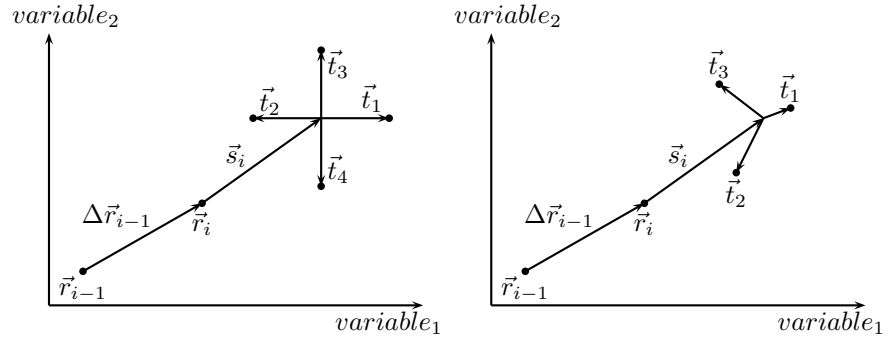

Figura 4.7: Pruebas de combinaciones de variables con los algoritmos de (izquierda) descenso de coordenadas y (derecha) aleatorio con $N_{estimaciones} = 3$ en un sistema de dos variables.

4.5. Método genético

MPCOTool también permite el uso de un método genético Genetic [2016] con sus algoritmos por defecto. Está inspirado en las ideas de GAUL [2016], pero ha sido completamente reprogramado de cero usando librerías externas modernas. El código en Genetic es también libre bajo una licencia de tipo BSD. La figure 4.8 muestra el diagrama de flujo del método genético por defecto implementado en Genetic.

4.5.1. El genoma

Las varaibles a calibrar/optimizar son codificadas en Genetic usando una cadena de bits: el genoma. Cuanto mayor es el número de bits asignado a una variable mayor es su resolución. El número de bits asignado a cada variable, y de ahí el tamaño del genoma, son fijos e iguales para todas las simulaciones. En la figura 4.9 se muestra un ejemplo codificando tres variables. El valor asignado a cada variable x está determinado por los valores extremos permitidos $x_{\text{mín}}$ and $x_{\text{máx}}$, el número binario asignado a la variable en la región correspondiente del genoma I_x y por el número de bits asignado a la variable N_x según la siguiente fórmula:

$$x = x_{\text{mín}} + \frac{I_x}{2^{N_x} - 1} \left(x_{\text{máx}} - x_{\text{mín}}\right). \tag{4.9}$$

4.5.2. Supervivencia de los mejores individuos

En una población con $N_{poblacion}$ individuos, en la primera generación se simulan todos los casos. Las variables de entrada se obtienen del genoma de cada individuo. Despueés, en cada generación, $N_{poblacion} \times R_{mutacion}$ individuos se

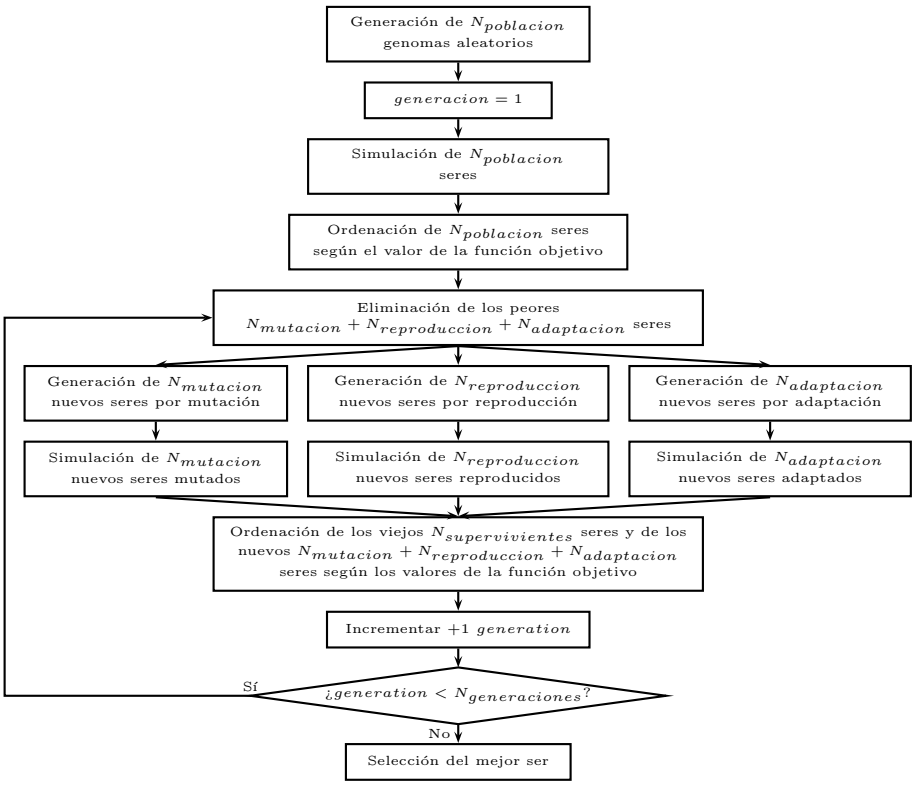

Figura 4.8: Diagrama de flujo del método genético por defecto implementado en Genetic.

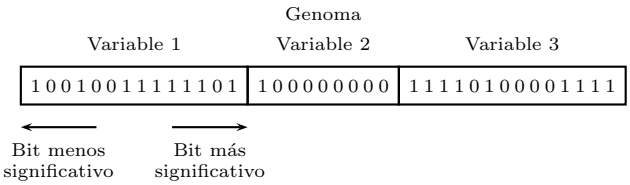

Figura 4.9: Ejemplo codificando tres variables a optimizar en un genoma. Primera y tercera variables han sido codificadas con 14 bits y la segunda con 9 bits en este ejemplo.

generan por mutation, $N_{poblacion} \times R_{reproduccion}$ por reproducción y $N_{poblacion} \times R_{adaptacion}$ por adaptación, obviamente teniendo en cuenta redondeos. A partir de la segunda generación sólo se se simulan los casos asociados a estos nuevos individuos (N_{nuevos}):

$$N_{nuevos} = N_{poblacion} \times (R_{mutacion} + R_{reproduccion} + R_{adaptacion}) \,. \tag{4.10}$$

Por lo tanto, el número total de simulaciones ejecutadas por el algoritmo genético es:

$$N_{total} = N_{poblacion} + (N_{generaciones} - 1) \times N_{nuevos}, \tag{4.11}$$

con $N_{generaciones}$ el número de generaciones de nuevos individuos. Los individuos de la anterior población que han obtenido peores valores en la función objetivo son son reemplazados así que sólo los mejores $N_{supervivientes}$ individuos sobreviven:

$$N_{supervivientes} = N_{poblacion} - N_{nuevos}. \tag{4.12}$$

Además, los ancestros que generan los nuevos individuos son elegidos entre los supervivientes. Obviamente, para tener población superviviente con la que generar nuevos individuos se debe cumplir:

$$R_{mutacion} + R_{reproduccion} + R_{adaptacion} < 1 \tag{4.13}$$

MPCOTool usa el criterio por defecto en Genetic, con una probabilidad linealmente decreciente con el ordinal del conjunto ordenado de individuos supervivientes (véase la figura 4.10).

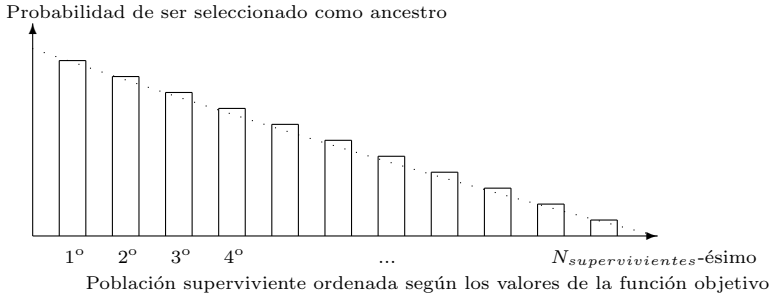

Figura 4.10: Probabilidad de un superviviente de ser seleccionado como ancestro de nuevos individuos generados por los algoritmos de mutación, reproducción o adaptación.

4.5.3. Algortimo de mutación

En este algoritmo se hace una copia idéntica del genoma del ancestro excepto en que se invierte un bit, elegido aleatoriamente con probabilidad uniforme. La figura 4.11 muestra un ejemplo.

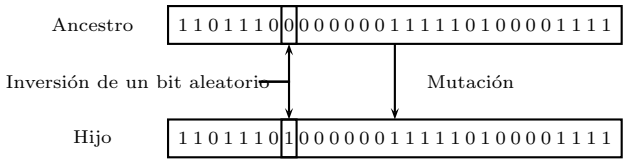

Figura 4.11: Diagrama ilustrando un ejemplo de la generación de un nuevo individuo por mutación.

4.5.4. Algoritmo de reproducción

El algoritmo de reproducción por defecto en Genetic selecciona dos ancestros diferentes entre la población superviviente y genera un nuevo individuo con los

mismos bits que los que resultan iguales en los genomas de ambos ancestros y con bits aleatorios en los demás. El nuevo hijo tiene el mismo número de bits que los antecesores pero distinto genoma. En la figura 4.12 se ilustra un diagrama del algoritmo.

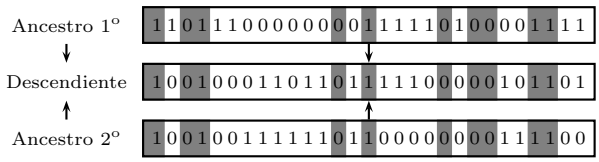

Figura 4.12: Ejemplo da la generación de un nuevo individuo con el algoritmo por defecto de Genetic. Nótese como los bits idénticos entre ambos antecesores (en gris) están también presentes en el descendiente. El resto de los bits es aleatorio.

4.5.5. Algoritmo de adaptación

Otro algoritmo se incluye en Genetic denominado "adaptation" aunque, en sentido biológico, es más bien una mutación suave. Primero, se selecciona aleatoriamente una variable codificada en el genoma con probabilidad uniforme. Luego se invierte un bit elegido aleatoriamente pero con una probabilidad decreciente según sea el bit más significativo. En la figura 4.13 puede verse un ejemplo.

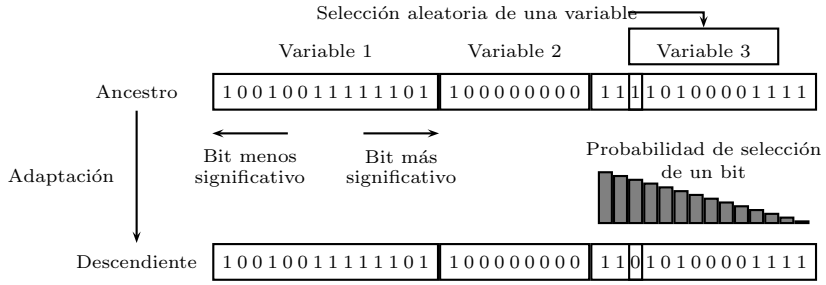

Figura 4.13: Ejemplo de generación de un nuevo individuo a partir de un ancestro por el algoritmo de adaptación.

Este algoritmo es similar al de mutación descrito previamente pero, puesto que la probabilidad de afectar bit menos significativos es mayor, también lo es la probabilidad de producir cambios pequeños.

4.5.6. Paralelización

El método genético es también fácilmente paralelizable siguiendo un esquema similar al del algoritmo iterativo, como puede verse en la figura 4.14.

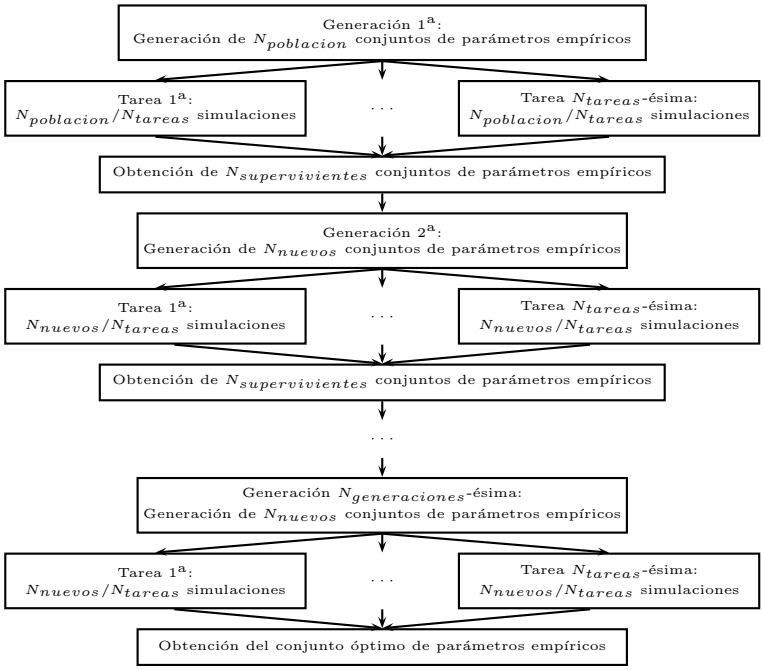

Figura 4.14: Diagrama de flujo del esquema de paralelización seguido en Genetic para el método genético.

Referencias

Autoconf. Autoconf is an extensible package of M4 macros that produce shell scripts to automatically configure software source code packages. https://www.gnu.org/software/autoconf, 2016.

Automake. Automake is a tool for automatically generating Makefile.in files compliant with the GNU coding standards. https://www.gnu.org/software/automake, 2016.

Clang. A C language family frontend for LLVM. http://clang.llvm.org, 2016.

GAUL. The genetic algorithm utility library. http://gaul.sourceforge.net, 2016.

GCC. The GNU compiler collection. https://gcc.gnu.org, 2016.

Genetic. Genetic: a simple genetic algorithm. https://github.com/jburguete/genetic, 2016.

GLib. A general-purpose utility library, which provides many useful data types, macros, type conversions, string utilities, file utilities, a mainloop abstraction, and so on. https://developer.gnome.org/glib, 2016.

GNU-Make. GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. http://www.gnu.org/software/make, 2016.

GSL. GNU scientific library. http://www.gnu.org/software/gsl, 2016.

GTK+3. GTK+, or the GIMP toolkit, is a multi-platform toolkit for creating graphical user interfaces. http://www.gtk.org, 2016.

Install-UNIX. A set of Makefiles to install some useful applications on different UNIX type operative systems. https://github.com/jburguete/install-unix, 2016.

JSON-GLib. A JSON reader and writer library using GLib and GObject. https://github.com/ebassi/json-glib, 2016.

Libxml. The XML C parser and toolkit of GNOME. http://xmlsoft.org, 2016.

MPICH. High-performance portable MPI. http://www.mpich.org, 2016.

OpenMPI. Open source high performance computing. http://www.open-mpi.org, 2016.

Pkg-config. Pkg-config is a helper tool used when compiling applications and libraries. http://www.freedesktop.org/wiki/Software/pkg-config, 2016.