

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

25 de noviembre de 2016

# Índice general

<b>1. Compilando el código fuente</b>	<b>2</b>
<b>2. Interfaz</b>	<b>5</b>
2.1. Formato en línea de comandos . . . . .	5
2.2. Aplicación con interfaz gráfica de usuario interactiva . . . . .	6
2.3. Ficheros de entrada . . . . .	7
2.3.1. Fichero de entrada principal . . . . .	7
2.3.2. Ficheros de plantilla . . . . .	10
2.4. Ficheros de salida . . . . .	11
2.4.1. Fichero de resultados . . . . .	11
2.4.2. Fichero de variables . . . . .	11
<b>3. Organización de MPCOTool</b>	<b>12</b>
<b>4. Métodos de optimización</b>	<b>14</b>
4.1. Método de fuerza bruta de barrido . . . . .	14
4.2. Método de Monte-Carlo . . . . .	15
4.3. Algoritmo iterativo aplicado a los métodos de fuerza bruta . . . .	16
4.4. Direction search method . . . . .	18
4.4.1. Coordinates descent . . . . .	18
4.4.2. Random . . . . .	19
4.5. Genetic method . . . . .	19
4.5.1. The genome . . . . .	21
4.5.2. Survival of the best individuals . . . . .	21
4.5.3. Mutation algorithm . . . . .	22
4.5.4. Reproduction algorithm . . . . .	22
4.5.5. Adaptation algorithm . . . . .	22
4.5.6. Parallelization . . . . .	23
<b>Referencias</b>	<b>25</b>

# 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<sup>1</sup>, 8.1<sup>1</sup> y 10<sup>1</sup>;
- 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 siguientes librerías de software libre son necesarias:

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

---

<sup>1</sup>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 servidores 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 sistemas 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.

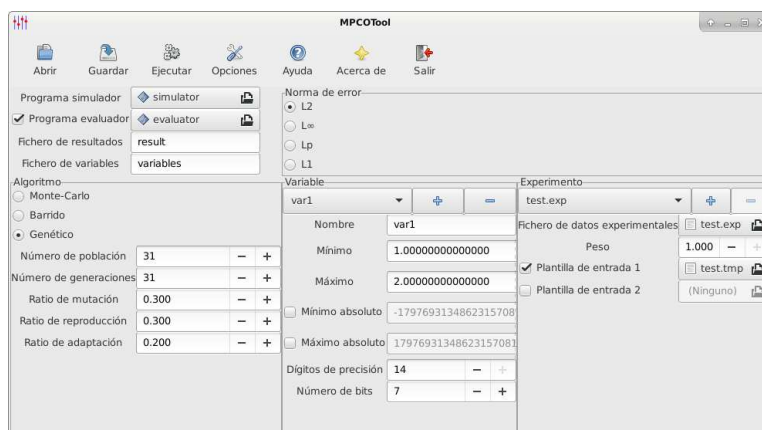


Figura 2.1: Ventana principal de la aplicación con interfaz gráfica de usuario interactiva de MPCOTool.

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

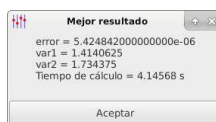


Figura 2.2: Cuadro de diálogo de los resultados óptimos de la aplicación con interfaz gráfica de usuario interactiva de MPCOTool.

## 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.

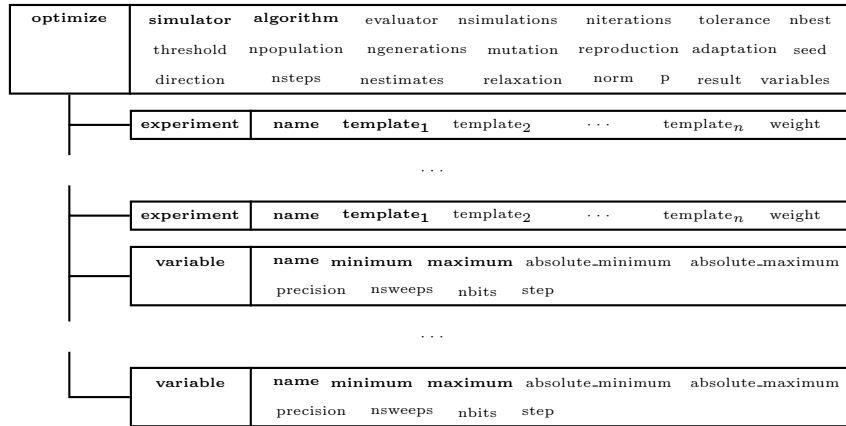


Figura 2.3: Estructura del fichero principal de entrada. Nodos y propiedades imprescindibles están en negrita. No obstante, otras propiedades también pueden ser necesarias en función del algoritmo de optimización seleccionado.

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  $L_\infty$ , véase (3.2),

**p** : norma de error  $L_p$ . 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 variable  $X$ -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>
```

Alternativamente, 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  $N_{\text{experimentos}} \times N_{\text{entradas}}$  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 sintácticamente 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 nombre “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  $N_{experiments}$  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  $N_{experiments}$  como  $N_{entradas}$ . También contiene los valores extremos de los parámetros empíricos y el algoritmo de optimización escogido. Entonces MPCOTool lee las correspondientes  $N_{experiments} \times N_{entradas}$  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:

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

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

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

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

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.

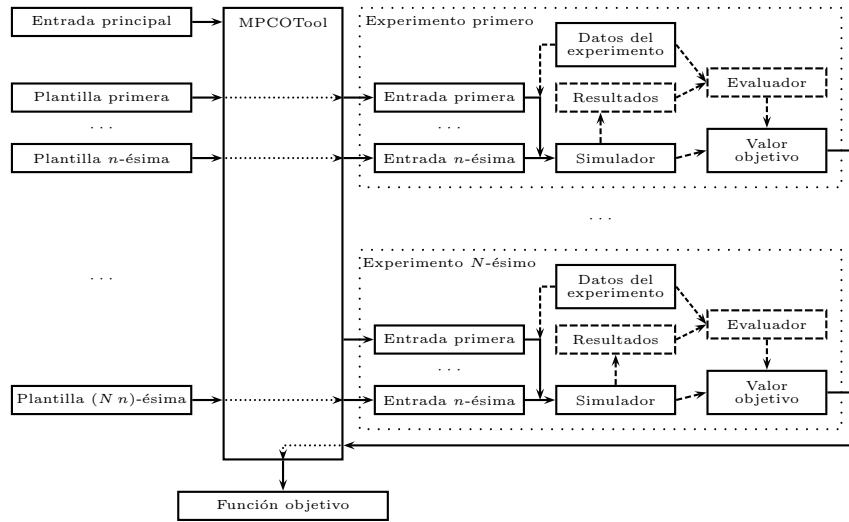


Figura 3.1: Diagrama de flujo de las interacciones entre MPCOTool, los diferentes ficheros de entrada y los programas de simulación y, en su caso, evaluación para producir el valor de la función objetivo para cada combinación de parámetros empíricos generada por el algoritmo de optimización.

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 parámetros 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:

$$N_{simulaciones} = N_{x,1} \times N_{x,2} \times \dots, \quad (4.1)$$

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  $N_{tasks}$  tareas paralelizadas cada tarea efectúa  $N_{total}/N_{tasks}$  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

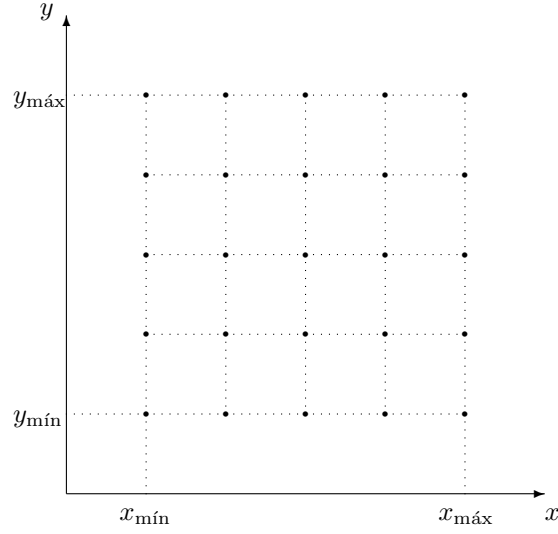


Figura 4.1: Diagrama de flujo ilustrando un ejemplo de aplicación del método de fuerza bruta de barrido con dos variables para  $N_x = N_y = 5$ .

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

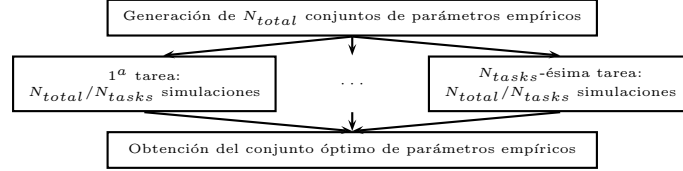


Figura 4.2: Diagrama de flujo del esquema de la paralelización en MPCOTool para métodos de fuerza bruta (barrido y Monte-Carlo).

## 4.2. Método de Monte-Carlo

Los métodos de Monte-Carlo realizan las simulaciones usando valores aleatorios de las variables 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.



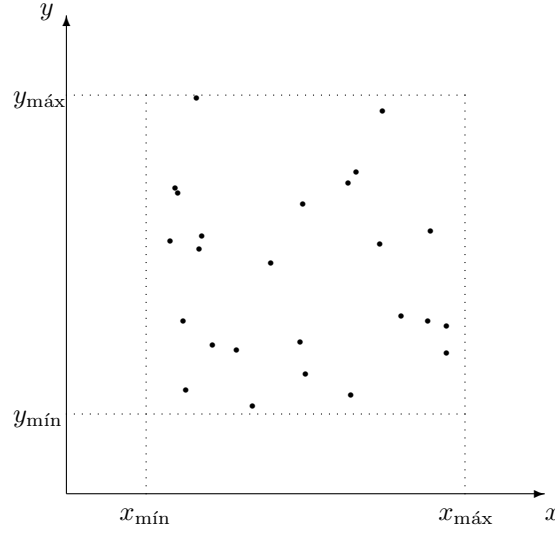


Figura 4.3: Diagrama ilustrativo de un método de fuerza bruta de Monte-Carlo con dos variables and  $N_{simulaciones} = 25$ .

### 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  $N_{best}^j$  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:

$x_{máx}^b = \max_{i \in N_{best}^j} x_i^j$  : Valor máximo de la variable  $x$  en el subconjunto de  $N_{best}^j$  mejores resultados de la iteración  $j$ -ésima.

$x_{mín}^b = \min_{i \in N_{best}^j} x_i^j$  : Valor mínimo de la variable  $x$  en el subconjunto de  $N_{best}^j$  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:

$$x_i^{j+1} \in [x_{mín}^{j+1}, x_{máx}^{j+1}], \quad (4.2)$$

con:

$$\begin{aligned} \text{Sweep} &\Rightarrow \begin{cases} x_{máx}^{j+1} = x_{máx}^b + \frac{x_{máx}^j - x_{mín}^j}{N_x - 1} tolerance, \\ x_{mín}^{j+1} = x_{mín}^b - \frac{x_{mín}^j - x_{máx}^j}{N_x - 1} tolerance, \end{cases} \\ \text{Monte - Carlo} &\Rightarrow \begin{cases} x_{máx}^{j+1} = \frac{x_{máx}^b + x_{mín}^b + (x_{máx}^b - x_{mín}^b)(1 + tolerance)}{2}, \\ x_{mín}^{j+1} = \frac{x_{máx}^b + x_{mín}^b - (x_{máx}^b - x_{mín}^b)(1 + tolerance)}{2}, \end{cases} \end{aligned} \quad (4.3)$$

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.

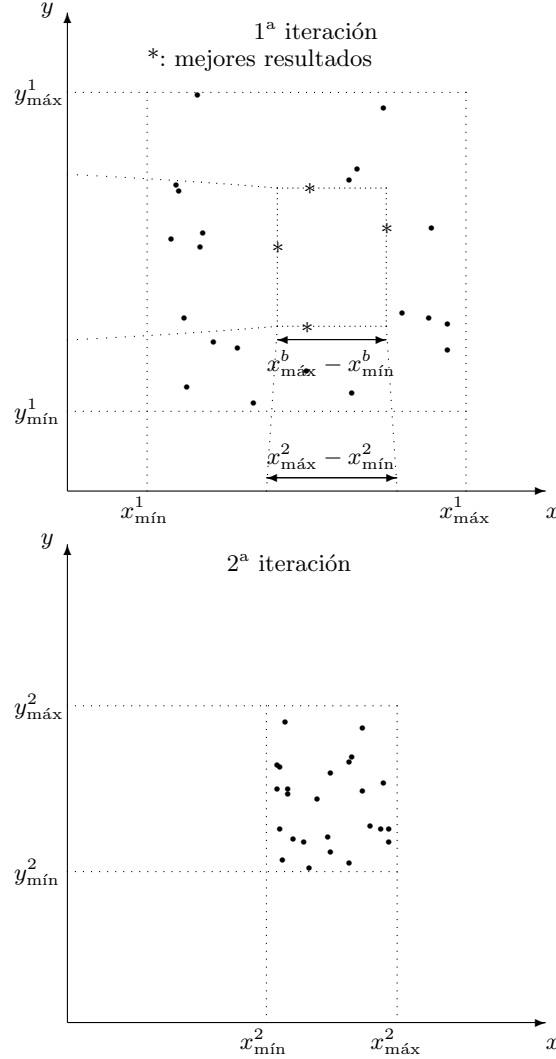


Figura 4.4: Diagrama representando un ejemplo del algoritmo iterativo aplicado al método de fuerza bruta de Monte-Carlo con dos variables para  $N_{\text{simulaciones}} = 25$ ,  $N_{\text{best}} = 4$  y dos iteraciones.

El algoritmo iterativo puede ser también fácilmente paralelizado. 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).

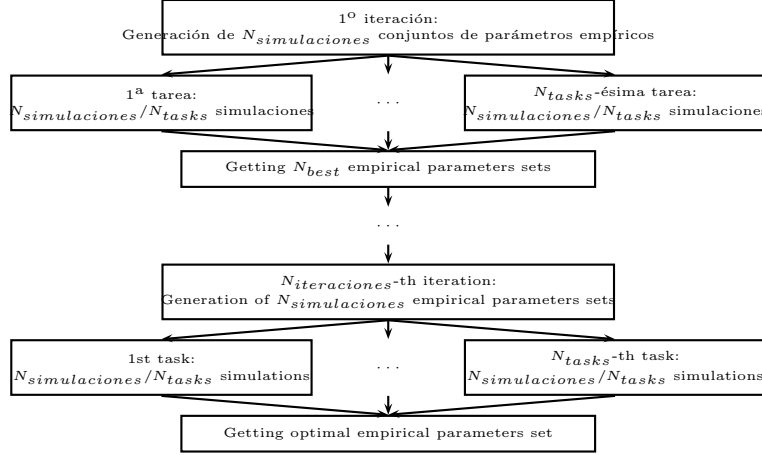


Figura 4.5: Flowchart of the parallelization scheme in MPCOTool for the iterative method.

## 4.4. Direction search method

Brute force optimization methods, sweep and Monte-Carlo, can be also combined with a direction search algorithm. Defining the vector  $\vec{r}_i$  as the optime variables combination obtained in the  $i$ -th step,  $\vec{r}_1$  as the optime variables combination vector obtained by the brute force method and defining the vector  $\vec{s}_i$  as:

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

with  $\Delta \vec{r}_{i-1} = \vec{r}_i - \vec{r}_{i-1}$  and *relaxtion* the relaxation parameter, the direction search method checks  $N_{estimates}$  variable combinations and choice the optimum as:

$$\vec{r}_{i+1} = \text{optime}(\vec{r}_i, \vec{r}_i + \vec{s}_i + \vec{t}_j), \quad j = 1, \dots, N_{estimates}. \quad (4.5)$$

If the step does not improve the optimum ( $\vec{r}_i = \vec{r}_{i+1}$ ) then the direction step vectors  $\vec{t}_j$  are divided by two and  $\vec{s}_{i+1}$  is set to zero. The method is iterated  $N_{steps}$  times.

Although direction search method gets the fastest convergence, is the method in MPCOTool that obtains the least advantages of parallelization. The method is almost sequential and parallelization only can be performed for each step in the  $N_{estimates}$  simulations to estimate the optimal direction. In the figure 4.6 a flowchart of the parallelization scheme for this method is shown.

MPCOTool uses two methods to build the  $\vec{t}_j$  vectors:

### 4.4.1. Coordinates descent

This method builds the  $\vec{t}_j$  vectors by increasing or decreasing only one variable:

$$\vec{t}_1 = \begin{pmatrix} step_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \vec{t}_2 = \begin{pmatrix} -step_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \vec{t}_3 = \begin{pmatrix} 0 \\ step_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

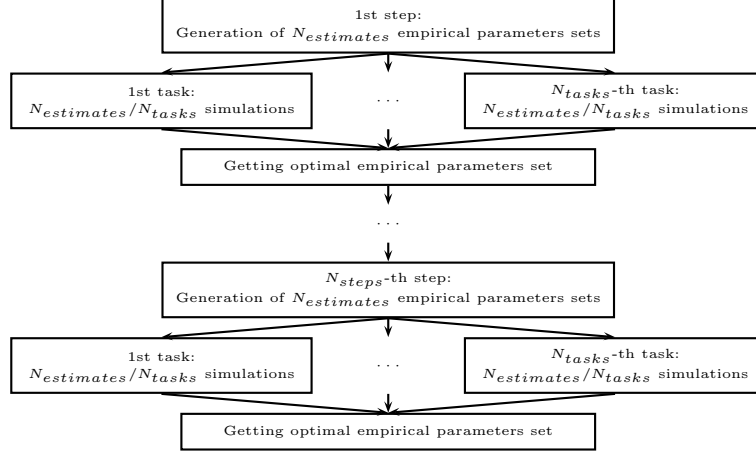


Figura 4.6: Flowchart of the parallelization scheme in MPCOTool for the direction search method.

$$\vec{t}_4 = \begin{pmatrix} 0 \\ -step_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \dots, \quad \vec{t}_{N_{estimates}} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ -step_{N_{variables}} \end{pmatrix}, \quad (4.6)$$

being  $step_j$  the initial step size for the  $j$ -th variable defined by the user in the main input file. The number of estimates in this method depends on the variables number:

$$N_{estimates} = 2 N_{variables} \quad (4.7)$$

#### 4.4.2. Random

The vectors  $\vec{t}_j$  are built randomly as:

$$\vec{t}_j = \begin{pmatrix} (1 - 2r_{j,1}) step_1 \\ \vdots \\ (1 - 2r_{j,k}) step_k \\ \vdots \\ (1 - 2r_{j,N_{variables}}) step_{N_{variables}} \end{pmatrix}, \quad (4.8)$$

with  $r_{j,k} \in [0, 1)$  random numbers.

In the figure 4.7 a sketch for a system with two variables is presented to illustrate the working mode of coordinates descent and random algorithms.

### 4.5. Genetic method

MPCOTool also offers the use of a genetic method Genetic Genetic [2016] with its default algorithms. It is inspired on the ideas in GAUL [2016], but it has been fully reprogrammed involving more modern external libraries. The code in

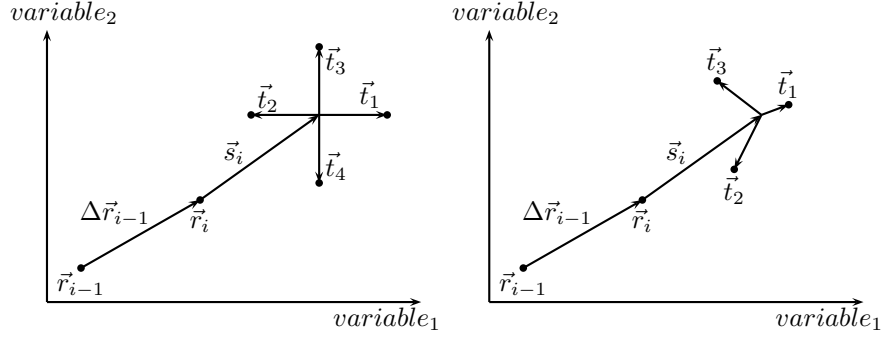


Figure 4.7: (Left) coordinates descent and (right) random with  $N_{estimates} = 3$  direction search method checks of variables combination in a system with two variables.

Genetic is also open source under BSD license. Figure 4.8 shows the flowchart of the genetic method implemented in Genetic.

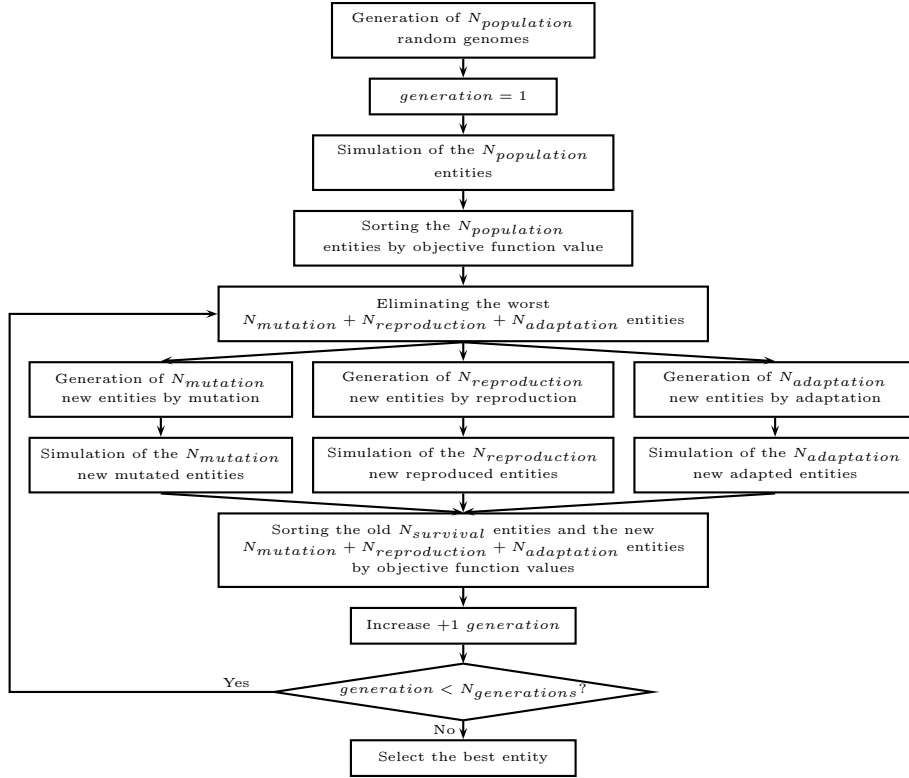


Figure 4.8: Flow diagram of the genetic method implemented in Genetic.

### 4.5.1. The genome

The variables to calibrate/optimize are coded in Genetic using a bit chain: the genome. The larger the number of bits assigned to a variable the higher the resolution. The number of bits assigned to each variable, and therefore the genome size, is fixed and the same for all the simulations. Figure 4.9 shows an example for the coding of three variables. The value assigned to a variable  $x$  is determined by the allowed extreme values  $x_{\min}$  and  $x_{\max}$ , the binary number assigned in the genome to variable  $I_x$  and by the number of bits assigned to variable  $N_x$  according to the following formula:

$$x = x_{\min} + \frac{I_x}{2^{N_x} - 1} (x_{\max} - x_{\min}). \quad (4.9)$$

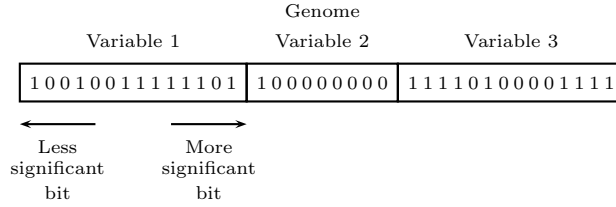


Figura 4.9: Example coding three variables to optimize into a genome. The first and third variables have been coded with 14 bits, and the second variable has been coded with 9 bits.

### 4.5.2. Survival of the best individuals

In a population with  $N_{population}$  individuals, in the first generation all the cases are simulated. The input variables are taken from the genome of each individual. Next, in every generation,  $N_{population} \times R_{mutation}$  individuals are generated by mutation,  $N_{population} \times R_{reproduction}$  individuals are generated by reproduction and  $N_{population} \times R_{adaptation}$  individuals are generated by adaptation, obviously taking into account rounding. On second and further generations only simulations associated to this new individuals ( $N_{new}$ ) have to be run:

$$N_{new} = N_{population} \times (R_{mutation} + R_{reproduction} + R_{adaptation}). \quad (4.10)$$

Then, total number of simulations performed by the genetic algorithm is:

$$N_{total} = N_{population} + (N_{generations} - 1) \times N_{new}, \quad (4.11)$$

with  $N_{generations}$  the number of generations of new entities. The individuals of the former population that obtained lower values in the evaluation function are replaced so that the best  $N_{survival}$  individuals survive:

$$N_{survival} = N_{population} - N_{new}. \quad (4.12)$$

Furthermore, the ancestors to generate new individuals are chosen among the surviving population. Obviously, to have survival population, the following condition has to be enforced:

$$R_{mutation} + R_{reproduction} + R_{adaptation} < 1 \quad (4.13)$$

MPCOTool uses a default aleatory criterion in Genetic, with a probability linearly decreasing with the ordinal in the ordered set of surviving individuals (see figure 4.10).

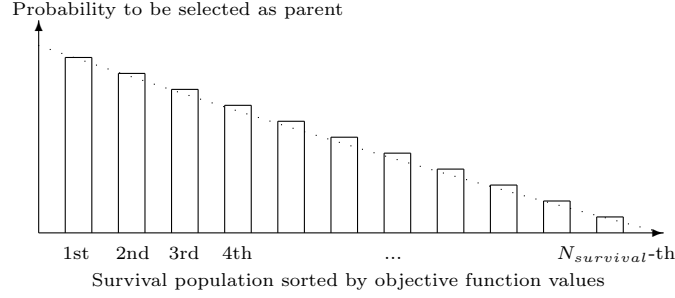


Figure 4.10: Probability of a survival entity to be selected as parent of the new entities generated by mutation, reproduction or adaptation algorithms.

#### 4.5.3. Mutation algorithm

In the mutation algorithm an identical copy of the parent genome is made except for a bit, randomly chosen with uniform probability, which is inverted. Figure 4.11 shows an example of the procedure.

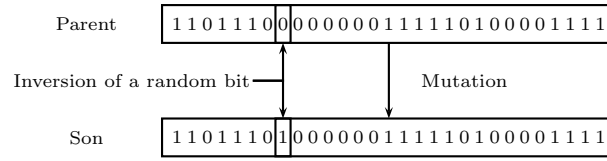


Figure 4.11: Diagram showing an example of the generation of a new entity by mutation.

#### 4.5.4. Reproduction algorithm

The default algorithm in Genetic selects two different parents with one of the least errors after the complete simulation of one generation. A new individual is then generated by sharing the common bits of both parents and a random choice in the others. The new child has the same number of bits as the parents and different genome. Figure 4.12 shows a sketch of the algorithm.

#### 4.5.5. Adaptation algorithm

Another algorithm is included in Genetic called 'adaptation'. Although, in the biological sense, it would be rather be a smooth mutation. First, one of the variables codified in the genome is randomly selected with uniform probability. Then, a bit is randomly chosen assuming a probability linearly decreasing with the significance of the bit. The new individual receives a copy of the parents genome with the selected bit inverted. Figure 4.13 contains an example.

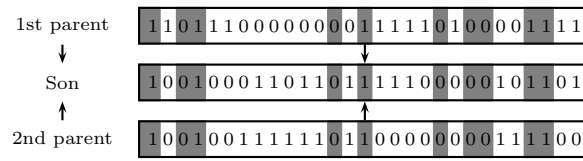


Figura 4.12: Example of the generation of a new entity by reproduction in the Genetic default algorithm. Note that the identical bits in both parents (in grey) are also present in their son. The rest of the bits are random.

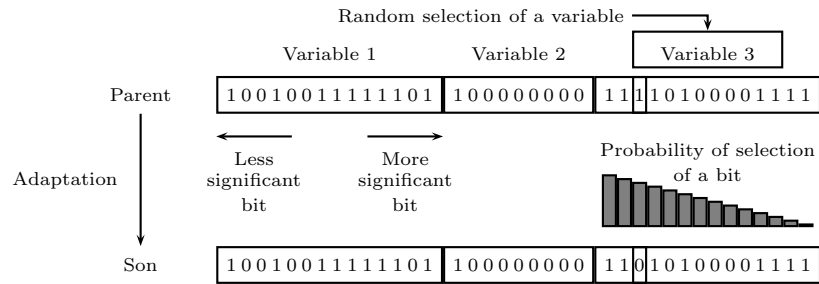


Figura 4.13: Example of the generation of a new individual from a parent by adaptation.

This algorithm is rather similar to the mutation algorithm previously described but, since the probability to affect bits less significant is larger, so is the probability to produce smaller changes.

#### 4.5.6. Parallelization

This method is also easily parallelizable following a similar scheme to the iterative algorithm, as it can be seen in the figure 4.14.



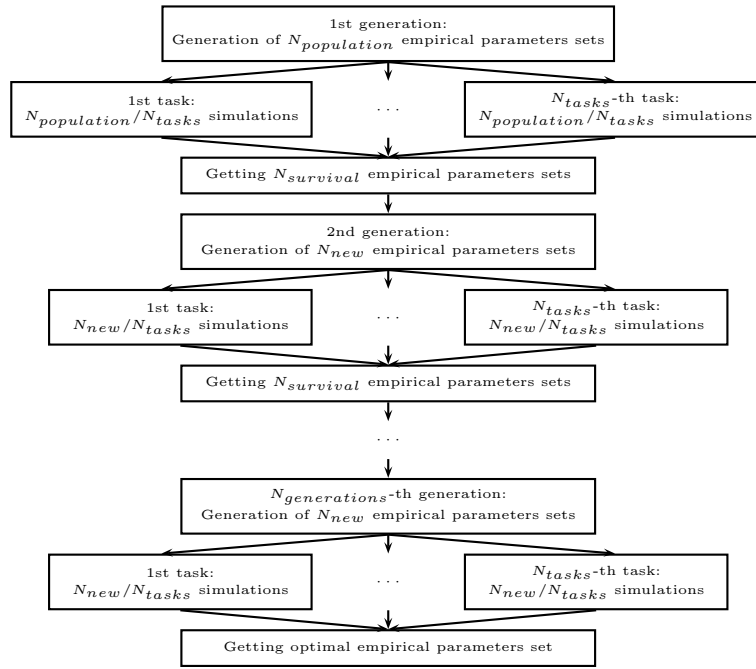


Figura 4.14: Flowchart of the parallelization scheme implemented in Genetic for the genetic method.

# 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.