MPCOTool: un software libre para obtener parámetros empíricos necesarios en modelos de simulación

Javier Burguete

12 de enero de 2016

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 4.2;
- Dyson Illumos;
- Fedora Linux 23;
- FreeBSD 10;
- Microsoft Windows 7[1] y 8.1[1];
- NetBSD 7.0;
- OpenBSD 5.8;
- OpenSUSE Linux 13;
- y Ubuntu Linux 12, 14 y 15.

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] and 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 y Windows 8.1 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.
- GTK+ [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 5.8, 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 23, para usar OpenMPI hay que hacer en un terminal (en la versión de 64 bits):

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

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/0.6.1
$ ./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/1.2.5
$ ln -s ../../genetic/0.6.1 genetic
$ ./build
```

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/0.6.1 genetic
$ cd ../test3
$ ln -s ../../../genetic/0.6.1 genetic
$ cd ../test4
$ ln -s ../../../genetic/0.6.1 genetic
$ cd ../../1.2.5
$ make tests
```

Finalmente podemos construir los manuales en formato PDF haciendo:

```
$ make manuals
```

Capítulo 2

Interfaz

2.1. Formato en línea de comandos

- La línea de comandos en modo secuencial es (donde X es el número de tareas a ejecutar paralelamente):

```
$ ./mpcotoolbin [-nthreads X] input_file.xml
```

- La línea de comandos en modo paralelizado en diferentes computadoras con MPI es (donde X es el número de tareas a ejecutar en cada nodo):

```
$ mpirun [MPI options] ./mpcotoolbin [-nthreads X] input_file.xml
```

- La sintaxis del programa simulador ha de ser:

```
$ ./simulator_name input_file_1 [input_file_2] [...] output_file
```

  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):

```
$ ./evaluator_name simulated_file experimental_file results_file
```

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

```
$ ./mpcotool
```

2.2. Ficheros de entrada

2.2.1. Fichero de entrada principal

Este fichero tiene que estar en formato XML con una estructura arbórea como la representada en la figura 2.1.

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

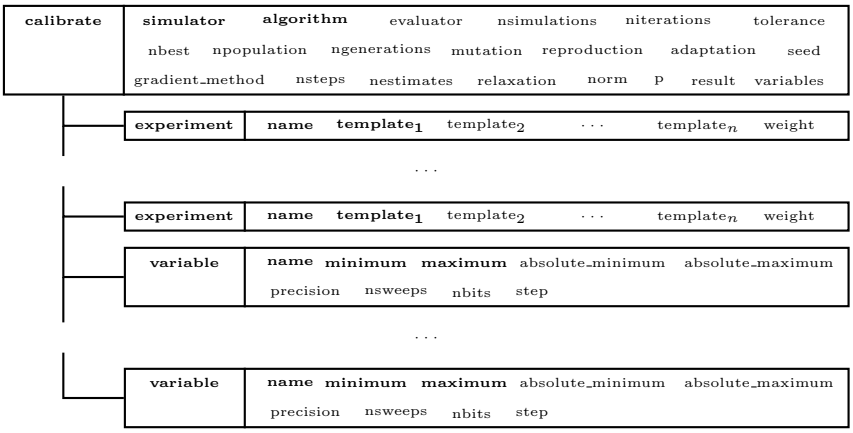

Figura 2.1: Estructura del fichero principal de entrada. Nodos y propiedades imprescindibles están en negrita. No obstante, otras propiedades también pueden ser necesarias en función del algoritmo de optimización seleccionado.

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)

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

gradient_method : método de estimar el gradiente (opcional para los algoritmos de barrido y de Monte-Carlo). Dos valores están disponibles actualmente:

coordinates : estimación por descenso de coordenadas.

random : estimación aleatoria. Requiere:

nestimates : número de pruebas aleatorias para estimar el gradiente.

Ambos métodos requieren además los siguientes parámetros:

nsteps : número de pasos para ejecutar el método basado en el gradiente,

relaxation : parámetro de relajación para el método basado en el gradiente,

y para cada variable:

step : tamaño de paso inicial para el método basado en el gradiente.

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.

2.2.2. 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.

2.3. Ficheros de salida

2.3.1. 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".

2.3.2. 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".

2.4. 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 2.2 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.

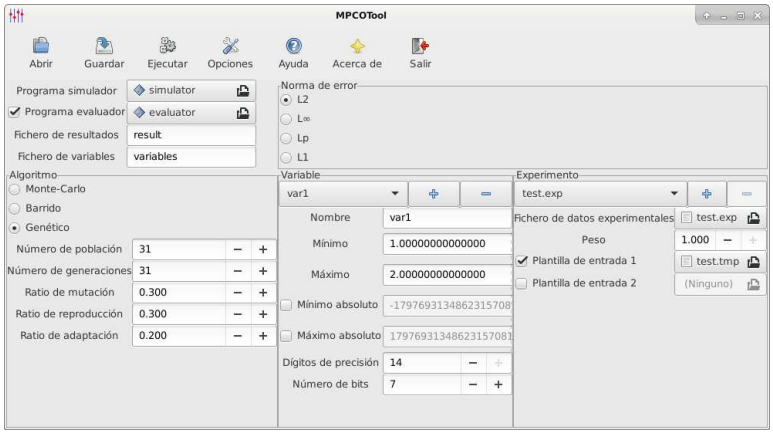

Figura 2.2: Ventana principal de la aplicación con interfaz gráfica de usuario interactiva de MPCOTool.

Los resultados óptimos se presentan finalmente en un cuadro de diálogo como el mostrado en la figura 2.3.

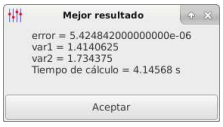

Figura 2.3: Cuadro de diálogo de los resultados óptimos de la aplicación con interfaz gráfica de usuario interactiva de MPCOTool.

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 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.

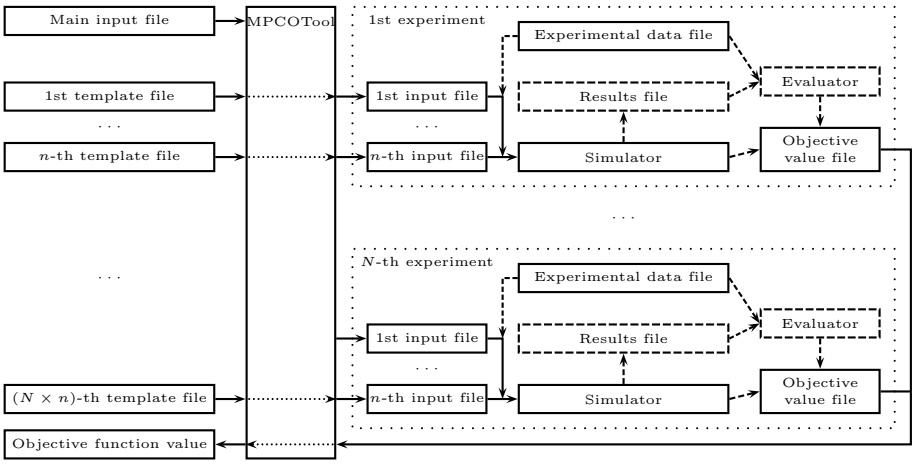

Figura 3.1: Diagrama de flujo de las interacciones entre MPCOTool, los diferentes ficheros de entrada y los programas de simulación y, en su caso, evaluación para producir el valor de la función objetivo para cada combinación de parámetros empíricos generada por el algoritmo de optimización.

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

Optimization methods

The optimization methods implemented in MPCOTool are next presented. The following notation will be used:

$N_{simulations}$: number of simulations made for each iteration.

$N_{iterations}$: number of iterations on iterative methods.

N_{total} : total number of simulations.

In iterative methods $N_{total} = N_{simulations} \times N_{iterations}$. In pure brute force methods $N_{iterations} = 1 \Rightarrow N_{total} = N_{simulations}$.

4.1. Sweep brute force method

The sweep brute force method finds the optimal set of parameters within a solution region by dividing it into regular subdomains. To find the optimal solution, the domain interval $x_i \in (x_{i,min}, x_{i,max})$ is first defined for each variable x_i. Then, a regular partition in $N_{x,i}$ subintervals is made. Taking into account this division of the solution space, the number of required simulations is:

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

where $N_{x,i}$ is the number of sweeps in the variable x_i.

In figure 4.1 the (x, y) domain is defined by the intervals $x \in (x_{min}, x_{max})$ and $y \in (y_{min}, y_{max})$. Both x and y intervals are divided into 5 regions with $N_x = N_y = 5$. The optimal will be found within the region by evaluating the error of each (x_i, y_i) set of parameters hence requiring 25 evaluations. Note that the computational cost increases strongly as the number of variables grow.

Brute force algorithms present low convergence rates but they are strongly parallelizable because every simulation is completely independent. If the computer, or the computers cluster, can execute N_{tasks} parallel tasks every task do N_{total}/N_{tasks} simulations, obviously taking into account rounding effects (every task has to perform a natural number of simulations). In figure 4.2 a flowchart of this parallelization scheme is represented. Being independent each task, a distribution on different execution threads may be performed exploding the full parallel capabilities of the machine where MPCOTool is run.

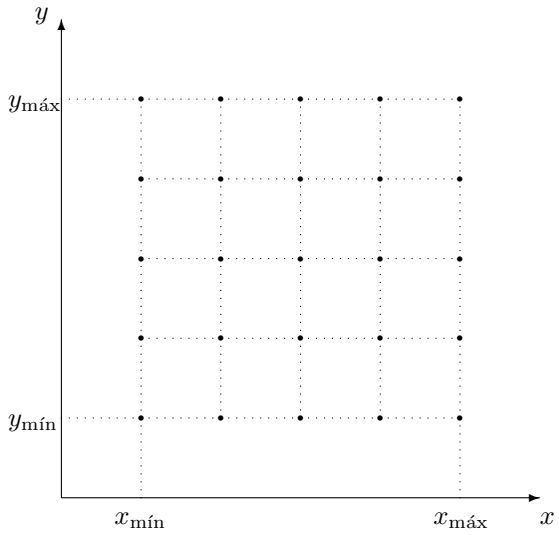

Figura 4.1: Diagram showing an example of application of the sweep brute force method with two variables for $N_x = N_y = 5$.

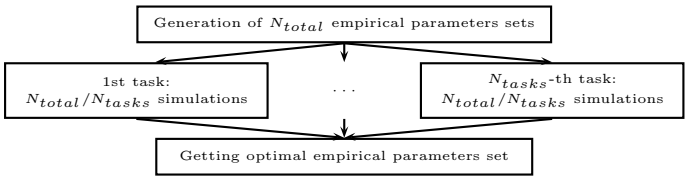

Figura 4.2: Flowchart of the parallelization scheme in MPCOTool for brute force methods (sweep and Monte-Carlo).

4.2. Monte-Carlo method

Monte-Carlo based methods run simulations using aleatory values of the variables assuming uniform probability within the extreme values range. Figure 4.3 shows the structure of an example using two variables.

Monte-Carlo method is also easily parallelizable following a strategy as the flowchart represented in the figure 4.2.

4.3. Iterative algorithm applied to brute force methods

MPCOTool allows to iterate both sweep or Monte-Carlo brute force methods in order to seek convergence. In this case, the best results from the previous iteration are used to force new intervals in the variables for the following iteration. Then for N_{best}^j, the subset of the best simulation results in the j-th iteration, the following quantities are defined:

$x_{\text{máx}}^b = \max_{i \in N_{best}} x_i^j$: Maximum value of variable x in the subset of the best simulation results from the j-th iteration.

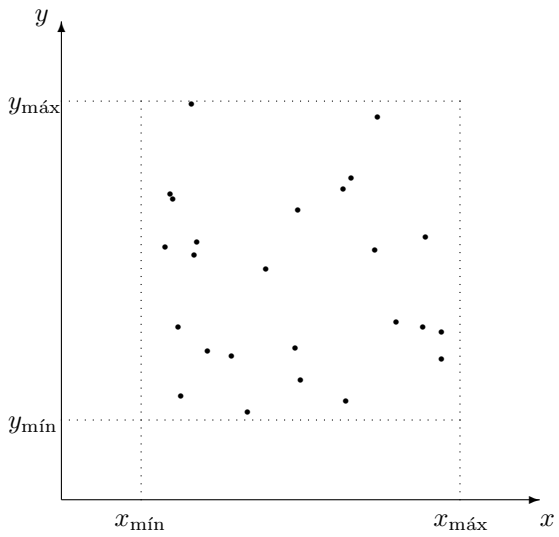

Figura 4.3: Diagram illustrating a Monte-Carlo brute force method with two variables and $N_{simulations} = 25$.

$x^b_{\text{mín}} = \underset{i \in N_{best}}{\text{máx}}\ x^j_i$: Minimum value of variable x in the subset of the best simulation results from the j-th iteration.

A new interval in the variable x is defined to build the optimization values in the next $(j+1)$ iteration so that:

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

with:

$$x^{j+1}_{\text{máx}} = \frac{x^b_{\text{máx}} + x^b_{\text{mín}} + \left(x^b_{\text{máx}} - x^b_{\text{mín}}\right)(1 + tolerance)}{2},$$
$$x^{j+1}_{\text{mín}} = \frac{x^b_{\text{máx}} + x^b_{\text{mín}} - \left(x^b_{\text{máx}} - x^b_{\text{mín}}\right)(1 + tolerance)}{2},$$

being *tolerance* a factor increasing the size of the variable intervals to simulate the next iteration. Figure 4.4 contains a sketch of the procedure used by the the iterative algorithm to modify the variables intervals in order to enforce convergence.

The iterative algorithm can be also easily parallelized. However, this method is less parallelizable than pure brute force methods because the parallelization has to be performed for each iteration (see a flowchart in the figure 4.5).

4.4. 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 Genetic is also open source under BSD license. Figure 4.6 shows the flowchart of the genetic method implemented in Genetic.

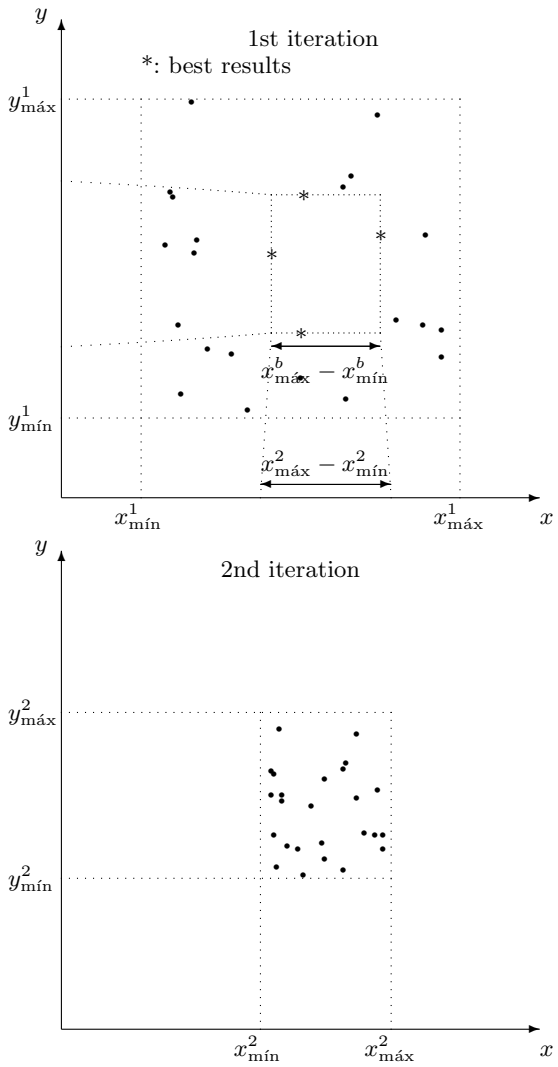

Figura 4.4: Diagram representing an example of the iterative algorithm applied to a Monte-Carlo brute force method with two variables for $N_{simulations} = 25$, $N_{best} = 4$ and two iterations.

4.4.1. 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 4.7 shows an example for the coding of three variables. The value assigned to a variable x is determined by the allowed extreme values $x_{\text{mín}}$ and $x_{\text{máx}}$, the binary number assigned in the genome to variable I_x and by the number of bits assigned to

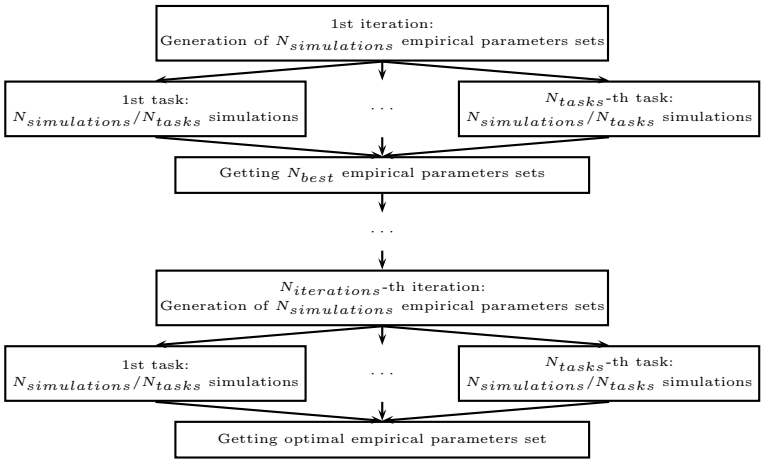

Figura 4.5: Flowchart of the parallelization scheme in MPCOTool for the iterative method.

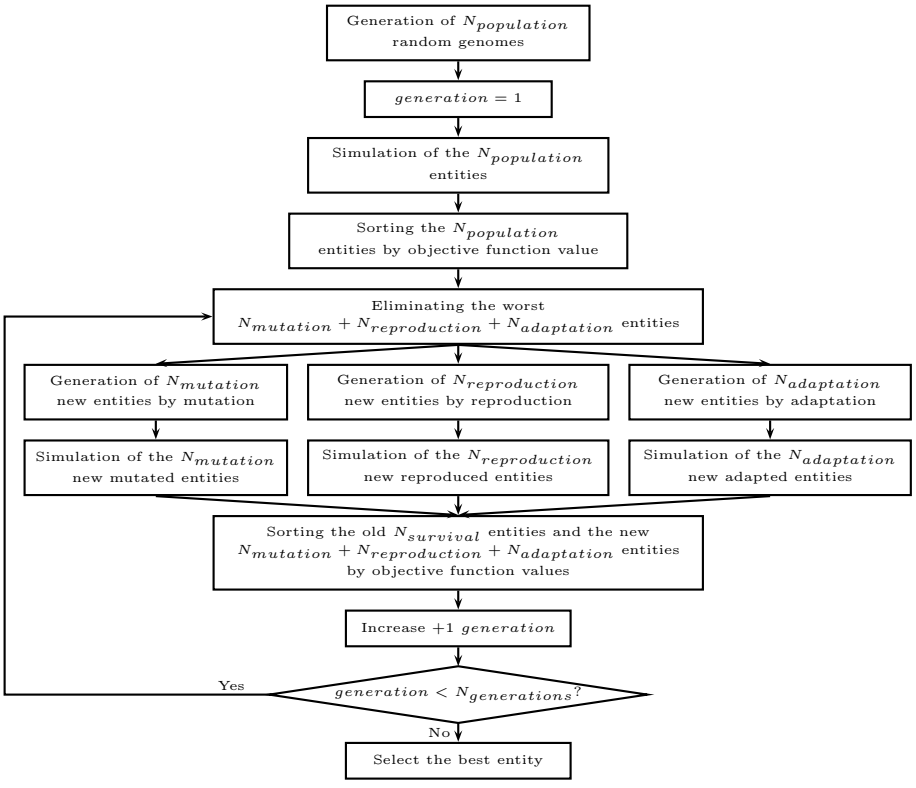

Figura 4.6: Flow diagram of the genetic method implemented in Genetic.

variable N_x according to the following formula:

$$x = x_{\text{mín}} + \frac{I_x}{2^{N_x}} \left(x_{\text{máx}} - x_{\text{mín}}\right). \tag{4.3}$$

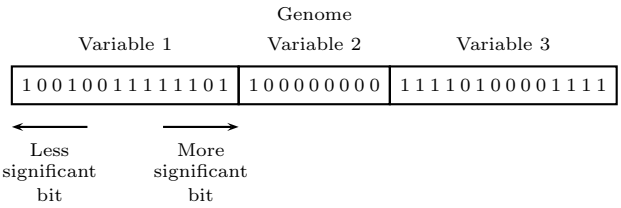

Figura 4.7: Example coding three variables to optimize into a genome. The first and third variables have been coded with 14 bits, and the second variable has been coded with 9 bits.

4.4.2. 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:

$$N_{new} = N_{population} \times (R_{mutation} + R_{reproduction} + R_{adaptation}) . \tag{4.4}$$

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

$$N_{total} = N_{population} + (N_{generations} - 1) \times N_{new}, \tag{4.5}$$

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:

$$N_{survival} = N_{population} - N_{new}. \tag{4.6}$$

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:

$$R_{mutation} + R_{reproduction} + R_{adaptation} < 1 \tag{4.7}$$

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 4.8).

4.4.3. 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 4.9 shows an example of the procedure.

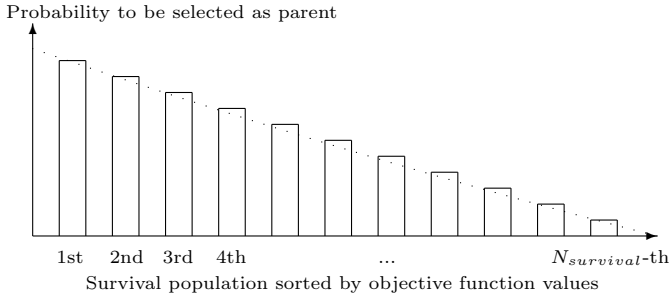

Figura 4.8: Probability of a survival entity to be selected as parent of the new entities generated by mutation, reproduction or adaptation algorithms.

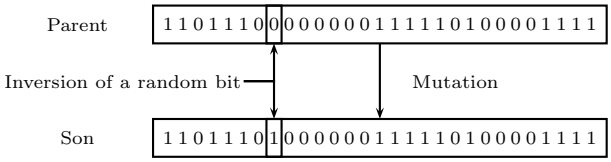

Figura 4.9: Diagram showing an example of the generation of a new entity by mutation.

4.4.4. 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 4.10 shows a sketch of the algorithm.

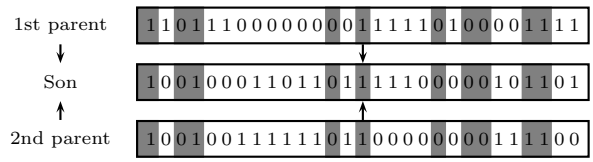

Figura 4.10: Example of the generation of a new entity by reproduction in the Genetic default algorithm. Note that the identical bits in both parents (in grey) are also present in their son. The rest of the bits are random.

4.4.5. 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 4.11 contains an example.

This algorithm is rather similar to the mutation algorithm previously des-

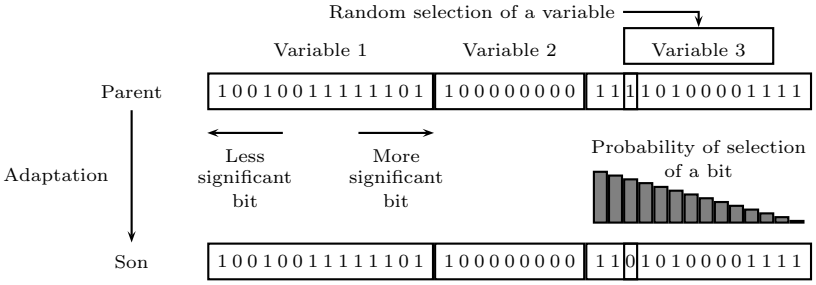

Figura 4.11: Example of the generation of a new individual from a parent by adaptation.

cribed but, since the probability to affect bits less significant is larger, so is the probability to produce smaller changes.

4.4.6. Parallelization

This method is also easily parallelizable following a similar scheme to the iterative algorithm, as it can be seen in the figure 4.12.

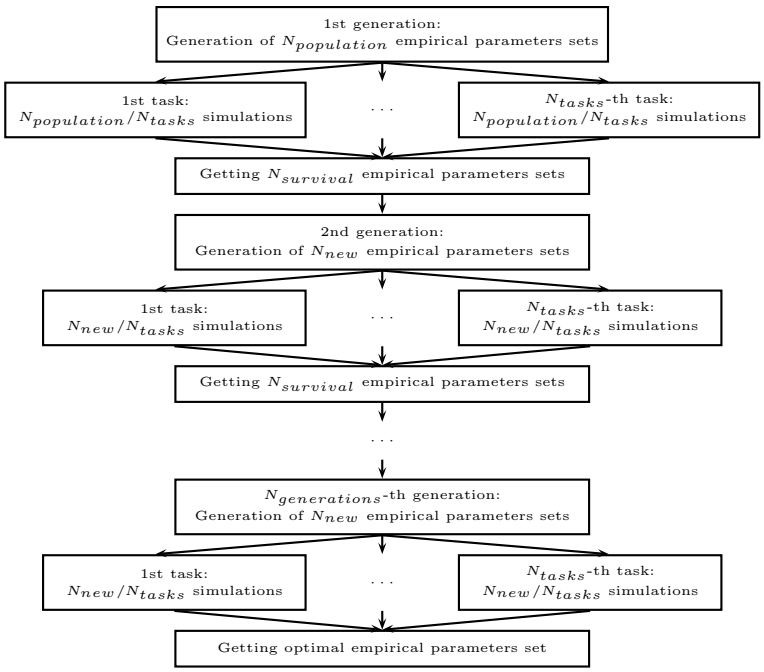

Figura 4.12: Flowchart of the parallelization scheme implemented in Genetic for the genetic method.

Bibliografía

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

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