ANN. MULT. GPU PROG.

Modelling parallel metaheuristics and hyperheuristics for auto-tuning

José-Matías Cutillas-Lozano and Domingo Giménez*

*Correspondence: domingo@um.es
Departamento de Informática y Sistemas, Universidad de Murcia, 30071, Murcia, Spain
Full list of author information is available at the end of the article

Abstract

This paper studies the auto-tuning of parallel metaheuristics and hyperheuristics. The modelling of the shared-memory scheme is considered for both types of algorithms, and a first study of message-passing metaheuristic schemes is introduced. A theoretical model of the execution time of a parametrized metaheuristic scheme is empirically adapted for a particular metaheuristic through experimentation. The parallelization of the shared-memory scheme is achieved through the independent parallelization of the basic functions in the metaheuristic scheme. The model is used to decide at running time the number of threads to obtain a reduced execution time. The number of threads is different for the different basic functions in the scheme, and depends on the problem to be solved, the metaheuristic or hyperheuristic scheme, the implementation of the basic functions and the computational system where the problem is solved. The auto-tuning methodology for shared-memory parametrized metaheuristic schemes can in turn be applied to shared-memory hyperheuristics developed on top of them. In the case of the message-passing scheme, an island model implemented with the master-slave scheme is used, and new metaheuristic-parallelism parameters representing the migration frequency, the size of the migration and the number of processes are introduced. The applicability of the proposal is shown with a minimization of electricity consumption in exploitation of wells problem and with the problem of obtaining satisfactory metaheuristics for that problem. Experimental results with these two problems show that satisfactory execution times can be achieved in metaheuristics with auto-tuning techniques based on theoretical-empirical models of the execution time.

Keywords: parametrized metaheuristic schemes; parallel hyperheuristics; shared-memory; message-passing; auto-tuning

Introduction

A unified parametrized scheme for metaheuristics facilitates the development of metaheuristics and their application [1]. The scheme has been applied successfully in different fields, to obtain satisfactory Simultaneous Equation Models from a set of values of the variables, for a tasks-to-processes assignation problem with independent tasks and memory constraints, for the p-hub problem [2], and for the optimization of power consumption in the operation of wells [3].

Although the metaheuristic scheme has proved efficient, its use for solving large problem instances causes significant increases in the execution time. Based on the increased possibilities offered by modern hardware architectures, the application of high performance computing strategies to metaheuristics is an interesting option for reducing the execution time. There is a large number of parallel strategies that can be applied to different metaheuristics in parallel environments of different characteristics [4, 5, 6, 7].

In our work, the parallelization of different metaheuristics is tackled through a unified parametrized metaheuristic scheme, and so the different metaheuristics that fit the scheme are parallelized in a unified way. Since NUMA systems are considered, we develop a parallelization based on the shared-memory paradigm. As our long-term goal is to model the metaheuristic scheme in heterogeneous

clusters, a first study of a message-passing parametrized metaheuristic scheme is presented. In this case, the parametrized scheme is expanded with new metaheuristic-parallelism parameters, which control the intensity and frequency of information exchange between processes. The island model is used for the message-passing scheme [8, 9, 10], with an implementation with the master-slave paradigm [11, 12].

Parallelizing the scheme for shared-memory reduces the execution time, but having a parallel routine does not ensure it will be used correctly, and the execution time of the parallel routine may be far from the optimum (or even larger than the sequential time) if the number of threads used in the application of the routine is not appropriate. The same ideas are taken into account for message-passing metaheuristics, considering for them the number of processes as the main parallel parameter to be optimized.

The auto-tuning problem of sequential and parallel routines has been studied in different fields [13, 14, 15, 16], and this paper considers the application of auto-tuning methodologies to parallel parametrized metaheuristics so that the auto-tuning techniques are valid for the different metaheuristics fitting the scheme. These include some basic metaheuristics and hybridations. In [3] we studied the modelling of the basic functions of the parametrized shared-memory scheme. To our knowledge there were no previous papers in the application of auto-tuning techniques to parallel metaheuristics. The techniques used there are now applied to the whole parametrized shared-memory metaheuristic scheme, and they are extended for the application to hyperheuristics which use the same metaheuristic scheme for satisfactory selection of metaheuristics or combinations/hybridations by obtaining appropriate values of the metaheuristic parameters in the unified scheme. To tune a metaheuristic (or a hyperheuristic) for a particular problem it is necessary to conduct a large number of experiments, which means a large execution time, and a scheme with auto-tuning would be very useful for reducing the experimentation time with a good selection of the number of threads (or processes) to use in the parallel scheme.

The rest of the paper is organized as follows. The next section summarizes the ideas of the common parallel parametrized scheme for metaheuristics, which can in turn be used for the development of hyperheuristics. After describing the metaheuristic scheme, the modelling of the execution time of the different basic and combined/hybridised metaheuriscs in the scheme is analysed theoretically and experimentally. The following section describes the problem of minimization of electricity consumption in exploitation of wells, which is used as a test case to validate the auto-tuning methodology. The penultimate section shows the experimental results obtained with the optimization problem and with a hyperheuristic for obtaining satisfactory metaheuristics. The last section concludes the paper and offers some future research lines.

Parallel parametrized schemes for metaheuristics and hyperheuristics

The ideas of a unified, parametrized metaheuristic scheme are presented in [2], and those of the shared-memory version are described in [1]. The general aspects are summarized here.

The concept of representing different metaheuristics under a common scheme is not new. Vaessens et al. [17] and Raidl [18] already use this approach and present algorithmic schemes such as that shown in Algorithm 1. The scheme considers a set of basic functions (`Initialize`, `EndCondition`, `Select`, `Combine`, `Improve`, and `Include`) whose instantiation determines the particular metaheuristic that is being implemented. The arguments `S`, `SS`, `SS1`, and `SS2` correspond to the sets of solutions that the method generates and manipulates in successive iterations. The functions can be used in one method or another, and different metaheuristics can be instantiated with the same pattern. The scheme is also valid as a generic mechanism in metaheuristic hybridization when, for example, the basic functions are composed of other metaheuristics or the same element of a different metaheuristic.

Algorithm 1 General scheme for metaheuristics

```
Initialize(S)
while (not EndCondition(S)) do
  SS=Select(S)
  SS1=Combine(SS)
  SS2=Improve(SS1)
  S=Include(SS2)
end while
```

Algorithm 2 parametrized metaheuristic scheme

```
Initialize(S,ParamIni)
while ( not EndCondition(S,ParamEnd)) do
  SS=Select(S,ParamSel)
  SS1=Combine(SS,ParamCom)
  SS2=Improve(SS1,ParamImp)
  S=Include(SS2,ParamInc)
end while
```

A parametrized metaheuristic scheme

The first observation to be made is that the basic functions presented in Algorithm 1 could receive additional parameters, so becoming a unified parametrized scheme for metaheuristics (Algorithm 2) that facilitates the development of metaheuristics and their application [2]. However, selecting the appropriate values of the *metaheuristic parameters* ($ParamX$ in the algorithm) to apply a satisfactory metaheuristic to a particular problem can be difficult and is computationally demanding. The selection of these values can be made through a hyperheuristic method also developed with the parametrized metaheuristic scheme. For clarity, hereinafter we refer to the metaheuristic scheme directly applied to an optimization problem as MS, and HMS refers to a hyperheuristic based on a metaheuristic scheme for selecting the appropriate values of metaheuristic parameters.

We comment on each of the functions of the parametrized scheme, their variants and the common parameters for the basic metaheuristics considered: Greedy Randomized Adaptive Search Procedure (GRASP), Scatter Search (SS), Tabu Search (TS) and Genetic Algorithms (GA). We also consider the possibility of reusing basic functions:

- `Initialize:` Valid random elements are generated to form an initial set with $INEIni$ elements. A smaller subset with $FNEIni$ elements is selected for the iterations in Algorithm 2. In some metaheuristics (for example, SS and GRASP) some of the initial elements are improved by using, for example, a local search or a greedy approach. A parameter $PEIIni$ indicates the percentage of elements to be improved, and the improvement may be more or less intense, which is represented by an intensification parameter, $IIEIni$. The parameter $STMIni$ is used for the extension of Tabu short-term memory in the initialization improvement.
- `EndCondition:` The end condition is common to the different metaheuristics; it consists of a maximum number of iterations ($MNIEnd$) or a maximum number of iterations without improving the best solution ($NIREnd$).
- `Select:` The elements can be grouped into two sets, the best and worst according to the objective function. The number of best elements will be $NBESel$ and that of worst elements $NWESel$, and normally $NBESel + NWESel = FNEIni$.

- `Combine:` The total number of elements to obtain by combination is $2(NBBCom + NBWCom + NWWCom)$, where the three parameters represent the number of combinations of the best with the best elements, the number of the best with the worst and the number of the worst with the worst.
- `Improve:` As in the improvement in the initialization, $PEIImp$, $IIEImp$ and $SMIImp$ represent the percentage of elements to be improved, the intensification of the improvement and the short-term memory in the improvement of the elements generated in the combination, and $PEDImp$, $IDEImp$ and $SMDImp$ represent the corresponding values in a diversification, which is equivalent to the mutation in the GA.
- `Include:` The $NBEInc$ best elements are maintained in the reference set, and the other $FNEIni - NBEInc$ to be included are selected from the remaining elements. $LTMInc$ is a Tabu parameter (long-term memory) that allows the tracking of individuals most frequently explored.

Thus, we have a set of 20 parameters (their meaning is summarized in table 1) with which it is possible to experiment to hybridize, mix and adapt the metaheuristics to the target problem. If other basic metaheuristics are considered, the number of metaheuristic parameters and their meaning would change. We are not interested in enumerating the possible parameters, but in the methodology for auto-tuning the shared-memory metaheuristic scheme.

Table 1 Metaheuristic parameters in the parametrized unified scheme of metaheuristics.

Initialize	$INEIni$	Initial Number of Elements
	$FNEIni$	Final Number of Elements after initialization
	$PEIIni$	Percentage of Elements to Improve in the initialization
	$IIEIni$	Intensification in the Improvement of initial Elements
	$STMIni$	Short-Term tabu Memory in the improvement of initial elements
EndCondition	$MNIEnd$	Maximum Number of Iterations
	$NIREnd$	maximun Number of Iterations with Repetition of the best solution
Select	$NBESel$	Number of Best Elements selected for combination
	$NWESel$	Number of Worst Elements selected for combination
Combine	$NBBCom$	Number of Best-Best elements combinations
	$NBWCom$	Number of Best-Worst elements combinations
	$NWWCom$	Number of Worst-Worst elements combinations
Improve	$PEIImp$	Percentage of Elements to Improve after combination
	$IIEImp$	Intensification in the Improvement of Elements after combination
	$SMIImp$	Short-term tabu Memory in the Improvement after combination
	$PEDImp$	Percentage of Elements to Diversify
	$IDEImp$	Intensification in the Diversification of Elements
	$SMDImp$	Short-term tabu Memory in the Diversification
Include	$NBEInc$	Number of Best Elements to include in the reference set
	$LTMInc$	Long-Term tabu Memory between iterations

Classification of metaheuristic combinations

Given the functional structure of the parametrized scheme and the pure metaheuristics that will be combined, the classification of combinations of metaheuristics considered using the taxonomy and nomenclature established in [6] is shown in table 2. The meaning of the design structures is $HRH(A_1 + A_2)$, which implies a high-level hybridization between the metaheuristics A_1 and A_2

which are executed in sequence without altering their internal structure, and $LRH(A_1(A_2))$ means A_2 metaheuristic embedded within A_1 with a low-level hybridization between them. In our approach we consider a different type of hybridation, $LRH(A_1, A_2)$, which represents a low-level inner join of metaheuristics A_1 and A_2. The difference between the two types of low-level hybridization is that in the second type, the metaheuristics mix their structures at the same level, while in the first, the first metaheuristic is supplemented with the structure of the second one when executed.

Table 2 Classification of the different combinations/hybridizations of metaheuristics.

Metaheuristic	Design
GR+TS	LRH(GR(TS))
GR+GA	HRH(GR+GA)
GR+SS	HRH(GR+SS)
GA+SS	LRH(GA,SS)
GA+TS	HRH(LRH(GA(TS))+TS)
SS+TS	HRH(LRH(SS(TS))+TS)
GR+GA+SS	HRH(GR+LRH(GA,SS))
GR+GA+TS	HRH(LRH(GR(TS))+LRH(GA(TS))+TS)
GA+SS+TS	HRH(LRH(LRH(GA,SS)(TS))+TS)
GR+SS+TS	HRH(LRH(GR(TS))+LRH(SS(TS))+TS)
GR+GA+SS+TS	HRH(LRH(GR(TS))+LRH(LRH(GA,SS)(TS))+TS)

To clarify how the nomenclature is applied for the basic metaheuristics and the unified scheme considered, we comment on the meaning of the last row in table 2, $HRH(LRH(GR(TS)) + LRH(LRH(GA, SS)(TS)) + TS)$. GRASP is initially applied with TS with short-term memory. After that, a combination of GA and SS is applied, with improvement of each element obtained by diversification and with the use of the short-term memory of TS. Finally, long-term memory constraints for frequent elements are considered by TS. All combinations of metaheuristics considered include various pure metaheuristics (heterogeneity), they search the entire space of solutions (are global) and all solve the same optimization problem (are general). In the previous example, the complete representation of the metaheuristic would be $HRH(LRH(GR(TS)) + LRH(LRH(GA, SS)(TS)) + TS)(het, glo, gen)$. For simplicity these terms are omitted in all the combinations.

Hyperheuristics

Hyperheuristics aim to automatically select, combine, generate or adapt several heuristics to efficiently solve computational search problems. The fundamental difference between metaheuristics and hyperheuristics is that most implementations of metaheuristics search within a search space of problem solutions, whereas hyperheuristics always search within a search space of heuristics. Thus, when using hyperheuristics, our objective is to find a method with the capacity to take good decisions in the path to finding a good metaheuristic to solve an optimization problem.

Hyperheuristics allow rapid portability to other application domains. The move to a new domain implies the implementation of low-level heuristics. If these low-level heuristics keep standard interfaces, it is not necessary to modify the hyperheuristic [19]. Therefore, our proposal can be used for hyperheuristics development, which can work by selecting appropriate values of the parameters in the parametrized metaheuristic scheme.

In the hyperheuristic, using the notation for evolutionary algorithms, an individual or element is represented by an integer vector $MetaheurParam$ that encodes the set of parameters that characterizes a metaheuristic using the scheme in Algorithm 2. The set of individuals constitutes the reference set, which means a set of metaheuristics, with each metaheuristic the combination/hybridation of basic metaheuristics given by the values in $MetaheurParam$.

Algorithm 3 parametrized shared-memory metaheuristic scheme

```
Initialize(S,ParamIni,ThreadsIni)
while ( not EndCondition(S,ParamEnd)) do
  SS=Select(S,ParamSel)
  SS1=Combine(SS,ParamCom,ThreadsCom)
  SS2=Improve(SS1,ParamImp,ThreadsImp)
  S=Include(SS2,ParamInc,ThreadsInc)
end while
```

The fitness value in the hyperheuristic for an element $MetaheurParam$ is that obtained when the metaheuristic with the parameters in $MetaheurParam$ is applied. The objective is to minimize the fitness function and so obtain the combination of the metaheuristic parameters (the prefered hibridation of basic metaheuristics) which gives the best fitness function for the problem in question.

When executing the hyperheuristic, a lot of metaheuristics are applied to different inputs, leading to a very large execution time, and parallelism is necessary. Parallel metaheuristics can be used to reduce the execution time, but it is also possible, and preferable, to use parallelism at a higher level, for which the parameterised shared-memory metaheuristic scheme is used for the hyperheuristic, and the same auto-tuning techniques are valid for the metaheuristics and the hyperheuristic.

The shared-memory scheme

In our approach, the parametrized scheme in Algorithm 2 becomes a parametrized shared-memory scheme just by independently parallelizing each basic function in the scheme (Algorithm 3) with new *parallelism parameters* ($ThreadsX$ in Algorithm 3) indicating the number of threads to use in each part of the algorithm.

A parallel parametrized scheme is used here to apply a common auto-tuning technique to select the optimum number of threads obtaining low execution times. When developing hyperheuristics with the same scheme used for metaheuristics, the same parallelization techniques for metaheuristics [4] are applicable for hyperheuristics. Two basic parallel schemes are identified in [1]:

- In the first scheme the elements of a set are treated independently, and the number of threads to work with in a loop is selected. This scheme appears, for example, when combining elements in a Genetic Algorithm or when randomly generating an initial set of elements. Thus, $ThreadsIni$ and $ThreadsCom$ contain a parallelism parameter indicating the number of threads to use in the generation of the initial set and for the combination of the selected elements. These values can be different, so giving different values of the parallelism parameters in each function.
- The second scheme has two parallelism levels and can be used to obtain fine or grained parallelism. The number of threads at each parallelism level is established. This type of parallelism appears in improvement and diversification functions, where some elements are selected (first level) and each element is improved by analysing its neighbourhood (second level).

The number of threads (one value or several values) is established for each function in the parametrized shared-memory scheme. The number of parallelism parameters for each function depends on the particular implementation of the functions in the unified scheme, but the methodology is common to various metaheuristics and parallel implementations. For example, some metaheuristics include an improvement part in the initialization, and the number of threads in the two levels of this improvement are added to the number of threads for the initialization of the reference set. In the improvement function there are two subsets of parallel parameters with the same structure, one for

the intensification and another for the diversification of elements. So, we have different parallelism parameters for the different functions in the scheme:

- `Initialize`: A for loop is used to generate the initial set of elements. One-level scheme is used with a number of threads $TGEIni$. A two-level improvement is included as part of the initialization, so there are two parameters for the number of threads in the two levels, $TI1Ini$ and $TI2Ini$. These three parameters constitute the set of parallelism parameters for the initialization: $ThreadsIni = \{TGEIni, TI1Ini, TI2Ini\}$.
- `Combine`: Pairs of elements are combined in a loop, so we have $TCPCom$ threads to work with in the combination loop.
- `Improve`: As in the improvement in the initialization, two-level parallelism functions are used for the improvement of the elements. Three improvement functions are considered: for the improvement of elements in the reference set, for those obtained in the combination, and for those obtained though diversification; so there are six parallelism parameters, $TR1Imp$, $TR2Imp$, $TC1Imp$, $TC2Imp$, $TD1Imp$ and $TD2Imp$.
- `Include`: An one-level parallelization with $TIEInc$ threads is considered for the inclusion of elements.

Thus, we have eleven parallelism parameters (table 3) that can be selected to tune the shared-memory scheme to obtain reduced execution times. Of course, as with the metaheuristic parameters, different parallelizations would produce a different number of parallelism parameters, and we are not interested in an exhaustive enumeration of the possible parallelization strategies and parallelism parameters, but in the application of a methodology valid for different configurations. Furthermore, the auto-tuning methodology with which the values of the parameters are selected will be valid for the different metaheuristics and combinations obtained with different values of the metaheuristic parameters, and the optimum values of the parallelism parameters will depend on those of the metaheuristic parameters.

Table 3 Parallelism parameters in the parametrized shared-memory scheme of metaheuristics.

Initialize	$TGEIni$	number of Threads for the initial Generation of Elements
	$TI1Ini$	number of Threads in the Improvement after initialization, first level
	$TI2Ini$	number of Threads in the Improvement after initialization, second level
Combine	$TCPCom$	number of Threads for the Combination of Pairs of elements
Improve	$TR1Imp$	number of Threads in the improvement of the Reference set, first level
	$TR2Imp$	number of Threads in the improvement of the Reference set, second level
	$TC1Imp$	number of Threads in the improvement of elements obtained by Combination, first level
	$TC2Imp$	number of Threads in the improvement of elements obtained by Combination, second level
	$TD1Imp$	number of Threads in the improvement of elements obtained by Diversification, first level
	$TD2Imp$	number of Threads in the improvement of elements obtained by Diversification, second level
Include	$TIEInc$	number of Threads for the Inclusion of Elements in the reference set

The metaheuristic scheme is used at two levels: for the hyperheuristic (HMS) and for the application of the metaheuristics determined from the metaheuristic parameters (*MetaheurParam*) in each element of the reference set in the metaheuristic (MS) with which the hyperheuristic is implemented. Thus, parallelism can be applied in the hyperheuristic and the metaheuristics, with a total of four parallelism levels, but it will be preferable to parallelize at a high level, and usually parallelism is only applied in the hyperheuristic.

The parametrized message-passing metaheuristic scheme

The use of message-passing schemes allows us to have versions which can be executed in both multicore systems and clusters with multicore nodes, enabling larger problems to be solved in shorter execution times. So, we analyse the development of message-passing parametrized schemes with MPI implementations. An island model is used, with division of the population into subsets assigned to p processes, with identifiers from 0 to $p-1$, where the process P_0 acts as master in the communications, and the remaining processes are the slaves.

The parametrized scheme in Algorithm 2 is extended to obtain a parametrized message-passing scheme (Algorithm 4) with the introduction of a new function (migration) and new metaheuristic-parallelism parameters, which comprise the number of processes (p), the number of generations between migrations ($NGMPar$) and the volume of data transferred ($NEMPar$). A homogeneous data partition is considered, with assignation of the same number of elements to each process. So, subsets $\mathtt{S_i}$ are processed in parallel by processes P_i, with $|\mathtt{S_i}| = \frac{|\mathtt{S}|}{p}$, and $\mathtt{S} = \mathtt{S_0} \cup ... \cup \mathtt{S_{p-1}}$. Each process is initialized with $\frac{INEIni}{p}$ elements. Then, the metaheuristic scheme of Algorithm 5 is applied sequentially to each subset over a number of iterations. The sizes of the other sets in each process are also divided by the number of processes: $\frac{FNEIni}{p}$, $\frac{NBESel}{p}$, $\frac{NWESel}{p}$, $\frac{NBBCom}{p}$, $\frac{NBWCom}{p}$, $\frac{NWWCom}{p}$ and $\frac{NBEInc}{p}$. The end condition is established now with the number of evolution-migrations, $\frac{MNIEnd}{NGMPar}$. The master informs the slaves when the end condition is accomplished.

Algorithm 4 parametrized message-passing metaheuristic scheme. Island Model (`S,ParamMet,ParamPar`).

```
1: IN PARALLEL in each process P_i (i = 0,...,p-1) DO
2: Initialize(S_i,ParamIni)
3: while (not EndCondition(ParamEnd,NGMPar)) do
4:    Sequential_Metaheuristic_Scheme(S_i,NGMPar)
5:    Immigrate(S_i,S_0,NEMPar)
6:    In P_0 Integrate Subpopulations(S_i)
7:    Emigrate(S_0,S_i,NEMPar)
8: end while
9: END PARALLEL
10: Solution: best s_k ∈ S_0
```

Algorithm 5 Sequential Metaheuristic Scheme(`S_i,ParamMet,NGMPar`).

```
while (not EndCondition(NGMPar)) do
  SS_i=Select(S_i,ParamSel)
  SS1_i=Combine(SS_i,ParamCom)
  SS2_i=Improve(SS1_i,ParamImp)
  S_i=Include(SS2_i,ParamInc)
end while
```

The set of metaheuristic parameters is completed with the three new metaheuristic-parallelism parameters, $ParamPar = \{p, NGMPar, NEMPar\}$. There are many possibilities for the implementation of the new function (migration), but our initial goal is to analyse the advantages of using a parametrized message-passing metaheuristic scheme and to model it so that the model can be used for auto-tuning. So, we use a simple migration scheme, with immigrations from the slaves to the master (line 5 of Algorithm 4) and emigrations from the master to the slaves (line 7), and with the same number of elements in the immigration and the emigration ($NEMPar$). No exchange of elements among slaves is considered, allowing only the combination of the best elements from each subset (and subsequent improvements and diversifications) in the master process (line 6). The

percentage of migrating elements of each subset should not be very high in order to enhance only the migration of the best elements of each subset and reduce the execution time while maintaining a certain amount of native elements in each subset. Besides, high values could produce an increment in the cost of the communications. The number of generations between migrations ($NGMPar$) also affects the goodness of the solution and the execution time. High values mean less information exchange between processes, and possibly worse final solutions or more iterations to converge, but at the same time they reduce the number of communications and the execution time per iteration.

Modelling and auto-tuning methodology

To reduce the execution time it is necessary to select the values of the parallelism parameters appropriately. These parameters are the number of threads ($ThreadsIni$, $ThreadsCom$, $ThreadsImp$ and $ThreadsInc$) for each basic function in the case of shared-memory, and the number of processes and parameters controlling the amount and frequency of information exchanged between islands (p, $NEMPar$, $NGMPar$) for the message-passing paradigm.

A theoretical model of the execution time is obtained for each function and the number of threads of a loop or the number of threads in the first and the second parallelism levels is established in the case of the shared-memory scheme. Considering message-passing metaheuristics, the time model is global and includes the whole parallel scheme behaviour establishing the number of islands and the volume and frequency of the migration of elements between them.

The auto-tuning process used in [13] for linear algebra routines is adapted to the metaheuristic scheme. A problem of minimization of electricity consumption in exploitation of wells [3] is used to show the modelling and auto-tuning methodology. The metaheuristics used to show how the methodology works are Genetic Algorithm, Scatter Search, GRASP and Tabu Search, but the same methodology could be applied with other set of basic metaheuristics and implementations of the basic functions in the metaheuristic scheme. The process is divided into three phases:

- **Design:** The first phase of design is to obtain the theoretical model of the execution time for the functions in the scheme or for the whole scheme. There are various possibilities, depending on the parallel programming paradigm, so the shared-memory and message-passing approaches are explained separately.
  - In the shared-memory scheme, the routine is developed together with its theoretical execution time. A model of the execution time is obtained for each basic routine in Algorithm 3. Because two types of parallelism have been identified, two basic models can be used, one for one-level routines and another for nested parallelism. As an example for the one-level routines, the generation of the initial population in function `Initialize` with an initial number of elements in the reference set $INEIni$ can be modelled:

$$t_{one-level} = \frac{k_g \cdot INEIni}{p} + k_p \cdot p \tag{1}$$

    where k_g represents the cost of generating one individual; k_p the cost of generating one thread; and p is the number of threads.
    As an example of two-level routines, we consider the initial improvement. The improvement of a percentage $PEIIni$ of the initial elements with an intensification (extension of improvement for each neighbour) $IIEIni$ can be modelled:

$$t_{two-levels} = \frac{k_i \cdot INEIni \cdot PEIIni \cdot IIEIni}{100 \cdot p_1 \cdot p_2} + k_{p,1} \cdot p_1 + k_{p,2} \cdot p_2 \tag{2}$$

where k_i represents the cost of improving one element; $k_{p,1}$ and $k_{p,2}$ the cost of generating threads at the first and second level; and p_1 and p_2 the number of threads at each level. For each of the other basic functions, the corresponding metaheuristic parameters are determined and the model of the execution time is obtained similarly, as a function of those parameters and the parallelism parameters (the number of threads to be used in each routine and subroutine).

– For message-passing schemes, as in the case of shared-memory, the first step is to obtain a theoretical model of the execution time, which is essentially divided into one part of computing t_{cmp}, another of communications between processes t_{cmc}, and another of integration of elements in the master t_{int}. We can, in turn, divide the communication time in two parts: one for immigrations from the slaves to the master, t_{imm}, and another for emigrations from the master to the slaves, t_{emi}. Some of these times have been found experimentally to be negligible, so the equations of the model are:

$$t_{total} = t_{cmp} + t_{cmc} + t_{int}, \quad \text{with} \quad t_{int} \approx 0 \tag{3}$$

$$t_{cmp} = \sum_{i=1}^{2} k_i \cdot \frac{Param_i}{p} + \left(\sum_{j=1}^{6} k_j \cdot \frac{Param_j}{p} \right) \cdot NGMPar \tag{4}$$

$$t_{cmc} = t_{imm} + t_{emi}, \quad \text{with} \quad t_{emi} \approx 0 \tag{5}$$

$$t_{imm} = A \cdot p^3 + B \cdot p^2 + C \cdot p + D \tag{6}$$

where the parameters and constants in equation 4 have the meaning explained for table 4, and the immigration time has been modelled as a third grade polynomial (equation 6) because of its simplicity and goodness in experimental data fit. It has been found that the execution time does not vary significantly with $NEMPar$, so we can consider t_{imm} only as a function of p.

The models so obtained are very simple and do not consider some architectural aspects, like memory or threads allocation, but their simplicity facilitates their use, and satisfactory results are obtained. Furthermore, in some execution environments those system-architecture aspects can not be considered when the code runs; for example, when sending the job to a queue, the system decides the cores where the threads or processes are mapped and the data allocation.

- **Installation:** The values of the system parameters are estimated for the system where the scheme is being installed.
  – When the shared-memory parametrized scheme is installed in a particular system, the value of the parameters influenced by the system are estimated. Parameters k_g, k_i, k_p, $k_{p,1}$ and $k_{p,2}$ presented in the previous step are some of these parameters, as are the corresponding parameters for the other basic routines. The estimation can be made through experimentation with each basic function in the metaheuristic, for some values of the

metaheuristic parameters and parallelism parameters ($INEIni$ and p in equation 1; and $INEIni$, $PEIIni$ and $IIEIni$, and p_1 and p_2 in equation 2) and least square adjustment.

– As in shared-memory, the value of the parameters influenced by the system are estimated when the message-passing-parametrized scheme is installed in a particular system. The system and metaheuristic parameters in the model for each function in the scheme are presented in table 4.

Table 4 System and metaheuristic parameters considered in equation 4 for the message-passing scheme. Functions: Gen-Ini, generation of elements in the initialization; Imp-Ini, Imp-Ref and Imp-Com, improvement of elements after initialization, improvement of elements of the reference set and those obtained by combination, respectively; Com, combination; Div-Ref and Div-Com, diversification of reference and combination sets; Inc, inclusion of elements.

$func$	k_i	$Param_i$
Gen-Ini	k_g	$INEIni$
Imp-Ini	k_{ii}	$\frac{INEIni \cdot PEIIni \cdot IIEIni}{100}$

$func$	k_j	$Param_j$
Com	k_c	$2 \cdot (NMMCom + NMPCom + NPPCom)$
Imp-Ref	k_{ir}	$\frac{NFEIni \cdot PEIImp \cdot IIEImp}{100}$
Imp-Com	k_{ic}	$\frac{(NBBCom+NBWCom+NWWCom) \cdot PEIImp \cdot IIEImp}{100}$
Div-Ref	k_{dr}	$\frac{NFEIni \cdot PEDImp \cdot IDEImp}{100}$
Div-Com	k_{dc}	$\frac{(NBBCom+NBWCom+NWWCom) \cdot PEDImp \cdot IDEImp}{100}$
Inc	k_i	$NFEIni + 2 \cdot (NBBCom + NBWCom + NWWCom) - NBEInc$

The estimation can be made through experimentation with the metaheuristic scheme, for some values of the metaheuristic parameters and parallelism parameters and least square adjustment.

- **Execution:** The solution of a problem with a particular metaheuristic or hyperheuristic is obtained with the values of the algorithmic parameters (number of threads at each function or number of processes, frequency and width of migration) which provide the lowest theoretical execution time for that problem size and the combination of metaheuristic parameters, using the theoretical models developed in the design phase and with the values of the system parameters estimated in the installation phase.

  – At execution time, in shared-memory, the number of threads in each basic function is selected from the theoretical execution time (equations 1 and 2) with the values of the metaheuristic parameters being those of the metaheuristic (or hyperheuristic) we are experimenting with and the values of the system parameters estimated in the installation phase. The number of threads which gives the theoretical minimum execution time is obtained by minimizing the corresponding equation after substituting in it the values of the metaheuristic and system parameters.

  After substituting in the theoretical model the values of the constants estimated by experimentation and least-squares, the values of the parallelism parameters which give the lowest theoretical time are obtained as a function of the metaheuristic parameters, so the equations are valid for the different metaheuristics or hyperheuristics in the particular computational system where the scheme is installed.

For example, for the initial generation of the reference set of HMS and experiments carried out in a system called *Ben* with 128 cores and described below, the number of threads which provides the lowest theoretical execution time is:

$$p_{opt} = \sqrt{\frac{k_g}{k_p} \cdot INEIni} = 3.43 \cdot \sqrt{INEIni} \tag{7}$$

and for the improvement of the generated elements the optimum number of threads in the two-level function is:

$$p_{1,opt} = 0.479 \cdot \sqrt[3]{INEIni \cdot PEIIni \cdot IIEIni} \tag{8}$$

$$p_{2,opt} = 0.505 \cdot \sqrt[3]{INEIni \cdot PEIIni \cdot IIEIni} \tag{9}$$

and those are the values selected for the parallelism parameters in the initialization: $TGEIni = p_{opt}$, $TI1Ini = p_{1,opt}$ and $TI2Ini = p_{2,opt}$.

– In the case of the message-passing scheme, the parallelism parameters are also selected from the theoretical execution time, as a function of the metaheuristic parameters and the values of the system parameters estimated in the installation phase. As in shared-memory, the optimum parallelism parameters (number of processes in this case) are obtained by minimizing the corresponding equation after substituting the values of the metaheuristic and system parameters.
For example, in one node of the cluster described below (*Saturno*, with 24 cores), we have:

$$\frac{-K_{cmp}}{p_{opt}^2} + 3 \cdot \ -4 \cdot 10^{-6} \cdot p_{opt}^2 + 2 \cdot 0.0003 \cdot p_{opt} - 0.006 = 0 \tag{10}$$

where K_{cmp} is the sum of all constants and metaheuristic parameters in the computing term specified in equation 4:

$$K_{cmp} = \sum_{i=1}^{2} k_i \cdot Param_i + \left(\sum_{j=1}^{6} k_j \cdot Param_j \right) \cdot NGMPar \tag{11}$$

and where the constants for each basic routine of the node *Saturno*, k_i and k_j, are presented in table 19 in the results section.

A problem of electricity consumption in exploitation of wells

The cost-minimization problem [3] used as a test case is briefly explained. We consider a water system consisting of a series of pumps (B) of known power, located in wells, that draw water flow along a daily time range R. The total flow is the sum of the flows contributed by each well. The pumps may

be running or idle at a given time. The pumps operate electrically and the electricity has a daily cost which should be minimal:

$$C_e = \sum_{i=1}^{R} \sum_{j=1}^{B} T_i P_j N_i x_{ij} \tag{12}$$

subject to

$$H \cdot \sum_{i=1}^{R} \sum_{j=1}^{B} Q_j x_{ij} = V_{td} \tag{13}$$

$$\forall i, \quad \sum_{j=1}^{B} Q_j x_{ij} \geq Q_{min.} \tag{14}$$

$$\forall j, \quad \frac{24}{R} Q_j \sum_{i=1}^{R} x_{ij} \leq V_{g,j} \tag{15}$$

$$\forall i, \quad \frac{\sum_{j=1}^{B} Q_j \sigma_j x_{ij}}{\sum_{j=1}^{B} Q_j x_{ij}} \leq \sigma_{lim.} \tag{16}$$

$$\exists j \mid DD_j \leq MD_j \quad \Longrightarrow \quad \forall i, \quad x_{ij} = 0 \tag{17}$$

where C_e represents the cost of the electricity consumed by the combination of pumps selected in a day; T_i is the cost of the electricity in the range i; N_i is the number of hours of pump operation in the time slot i; P_j is the electric power consumed by the pump j; and x_{ij} represents a binary element of a matrix with values 1 or 0 for pump on or off in equation 12. In equation 13, H represents the number of hours of each time slot (the same for all the slots); Q_j is the flow extracted from the well j, constant in all the time intervals in which the well is operating; and V_{td} is the total daily volume demanded. In equation 14, $Q_{min.}$ is the total minimum flow in the pipeline for each time slot. $V_{g,j}$ is the operating volume daily granted to the well j in equation 15. In equation 16, σ_j represents the conductivity of each well and $\sigma_{lim.}$ is the conductivity limit of the water mix. And, finally, in equation 17, DD_j is the depth (m) of the dynamic level of well j and MD_j is the maximum depth (m) of the dynamic level of well j.

An individual is represented by a binary vector of size $B \cdot R$ that encodes the set of pumps distributed in different time slots. The set of individuals constitutes a population. Not all possible combinations result in feasible individuals, and each time an individual is generated or modified five constraints are evaluated:

- Demand satisfaction (equation 13). This restriction arises from the condition that the sum of volumes provided in the range of hours established correspond to the programmed demand at the beginning of each day.
- Minimum flow maintenance (equation 14). We sought to establish a minimum flow in the pipeline in all time slots of operation.
- Compliance with maximum exploitation volumes for each well (equation 15). In practice, the accumulated volume granted for each well is updated daily and is inserted as a parameter of the problem.

- The conductivity of the total flow is calculated as a weighted mean of the conductivity contributed by each well, and the mean conductivity is maintained below the limit established (equation 16).
- Compliance with maximum depths of dynamic levels (equation 17). Each well must have a minimum water level to exploit to be considered operative. Below this, the well will be classified as temporarily unavailable.

This problem formulation means in some cases that obtaining a new individual is very time-consuming. Furthermore, for large exploitation systems the number of wells and of time ranges may be large, resulting in large computation times.

Experimental results

The results of applying the auto-tuning technique to the parallel parametrized schemes of metaheuristics are presented below. Both metaheuristics and hyperheuristics are considered in shared-memory, but only metaheuristics are applied through message-passing schemes. Experiments are carried out in two computational systems whose characteristics are presented below.

Firstly, we have a *HP Integrity Superdome SX2000* with 128 cores of *Intel Itanium-2 dual-core Montvale* with shared-memory called *Ben*.

Secondly, *Heterosolar*, a cluster with five nodes connected through a Gigabit Ethernet working at 1 Gbit/s and comprising the following multicore systems:

- *Saturno* is a NUMA system with 4 Intel hexa-core NEHALEM-EX EC E7530 nodes, with a total of 24 cores, 1.87 GHz, 32 GB of shared-memory.
- *Marte* and *Mercurio* are AMD Phenom II X6 1075T (hexa-core), 3 GHz, 15 GB (Marte) and 8 GB (Mercurio), each with private L1 and L2 caches of 64 KB and 512 KB, and L3 of 6 MB shared by all the cores.
- *Jupiter* comprises two hexa-cores (12 cores) Intel Xeon E5-2620, 2.00GHz, and 32 GB of RAM.
- *Luna* is a quad-core processor Intel Core 2 Quad Q6600, 2.4 GHz, with 4 GB of shared memory.

Shared-memory results

To validate the auto-tuning methodology, the optimum number of threads and the maximum speed-up achieved are calculated from the models for different metaheuristic parameters using the system parameters obtained in the installation. These system parameters are calculated using small values of the metaheuristic parameters to reduce the installation time.

Experiments in shared-memory are carried out in *Ben* and *Saturno*. The main differences between them are their size and the structure of the memory hierarchy, which produces differences in the shared-memory access latencies.

Firstly, as an example, we summarize the values of the system parameters obtained in the installation phase for two basic functions of the HMS in *Ben*. The optimum number of threads varies with the number of elements, and we are interested in selecting a number of threads close to the optimum from a small number of elements (for low installation time). As an example for one-level routines, in the initialization in a hyperheuristic the model in equation 1 is used, and parameters k_g and k_p in the model are obtained by least-squares with $INEIni = 5$. The values obtained are $k_g = 0.577$ and $k_p = 0.0491$ (both in seconds). For a two-level routine, like the routine to improve elements after the initial generation or after combination or diversification, the values of the parallelism parameters are obtained by least-squares with experiments with parameters for the hyperheuristic $INEIni = 10$, $PEIIni = 100$ and $IIEIni = 1$. The results are $k_i = 1.21$, $k_{p,1} = 0.104$ and $k_{p,2} = 0.0989$ (all in seconds). By substituting these values in the theoretical model of the execution time (equation 2),

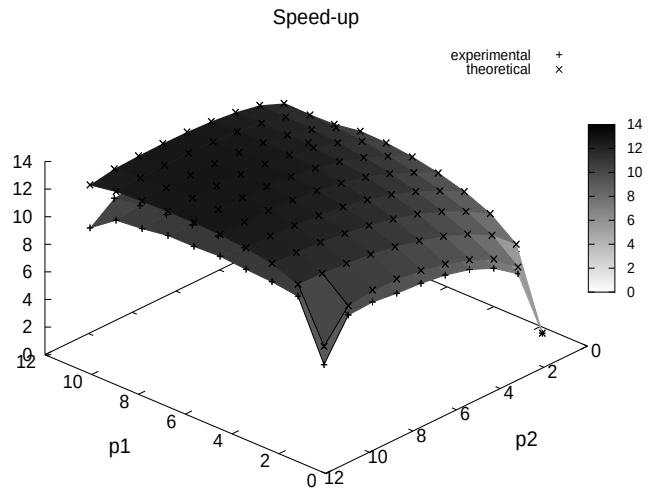

Figure 1 Theoretical and experimental speed-ups when varying the number of threads of the first and second level of parallelism in a two-level parallel routine when applying the HMS in shared-memory in *Ben*.

the behaviour of the routine in the system is well predicted, as can be seen in Figure 1, where the theoretical and experimental speed-ups in the improvement of the initial population are represented for the hyperheuristic parameter combination $INEIni = 50$, $PEIIni = 50$ and $IIEIni = 1$.

For the application of the MS, the values used in the installation were $INEIni = 20$ for the initialization and $INEIni = 20$, $PEIIni = 50$, $IIEIni = 20$ for the improvement in the initialization. For the HMS those values were $INEIni = 5$, $INEIni = 10$, $PEIIni = 100$ and $IIEIni = 1$. The differences between the values of the parameters of MS and HMS are explained by the higher execution time of the hyperheuristic, which makes it necessary to have lower values for a low installation time.

Once the scheme is installed, the next step is to determine the optimum parallel parameters and the corresponding speed-up for the metaheuristics and hyperheuristics considered. Firstly, the results of auto-tuning two basic routines in *Ben* are presented, and then the auto-tuning of the whole scheme is analysed in *Saturno*.

Metaheuristics and hyperheuristics for the minimization of power consumption in wells exploitation are considered, so experiments consider implementations of the basic functions with the metaheuristic parameters with different computational costs.

Tables 5 and 6 compare the results for the initial generation of the reference set and for the improvement of elements for two parameter combinations using the MS in *Ben*. The number of threads selected with the auto-tuning methodology is not far from the best values obtained experimentally and, as a consequence, the speed-up achieved with auto-tuning is not far from the experimental maximum and the auto-tuning methodology is useful for the reduction of the execution time of metaheuristics.

We can compare the results obtained when directly applying individual metaheuristics to a problem of optimization of electrical costs with those for the hyperheuristic using the auto-tuning methodology. Since the metaheuristic scheme is the same, similar results would be expected in both cases, although there may be differences due to different implementations. For example, in the improvement function of the MS, the second level was used to start more threads to work on the improvement of the fitness

Table 5 Comparison of the highest experimental speed-up (exp), the modelled speed-up (mod) and that obtained in the experiments when using the auto-tuning methodology (exp-auto), for $INEIni = 100$ and 500 in the one-level parallel routine of `Initialize`, when applying the MS in *Ben*.

	threads		speed-up		
$INEIni$	exp	mod	exp	mod	exp-auto
100	48	55	27	27	25
500	121	122	77	61	75

Table 6 Comparison of the highest experimental speed-up (exp), the modelled speed-up (mod) and that obtained in the experiments when using the auto-tuning methodology (exp-auto), for other parameter combinations in the two-level parallel routine of `Initialize`, when applying the MS in *Ben*.

			threads		speed-up		
$INEIni$	$PEIIni$	$IIEIni$	exp	mod	exp	mod	exp-auto
100	50	10	89	67	35	17	21
500	100	5	128	150	78	51	78

function (more neighbours are analysed) but not to reduce the execution time, from which the number of threads of second level could be taken as constant. So, in this case the model is slightly different. In the function of the initial generation of elements there are no differences in the implementation. The behaviour of the one-level routine when applying the MS was well predicted, as can be seen in Figure 2, where the theoretical and experimental speed-ups are represented.

Tables 7 and 8 compare the results for the initial generation of the reference set and for the improvement of elements for two parameter combinations using the HMS in *Ben*. As in the case of the MS, the number of threads and the speed-up selected with the auto-tuning methodology were not very different from the best values obtained experimentally, and so the auto-tuning methodology proves useful for the reduction of the execution time of hyperheuristics, which have a high cost caused by the application of a large number of metaheuristics. It can be seen that the technique applied for MS is also valid for HMS.

Table 7 Comparison of the highest experimental speed-up (exp), the modelled speed-up (mod) and that obtained in the experiments when using the auto-tuning methodology (exp-auto), for the parameters $INEIni = 20$ and 100 in the one-level parallel routine of `Initialize`, when applying the HMS in *Ben*.

	threads		speed-up		
$INEIni$	exp	mod	exp	mod	exp-auto
20	22	15	11	8	8
100	24	34	12	17	12

The advantages of using auto-tuning can be seen more clearly when comparing the speed-up obtained by launching the maximum number of threads available in the system and the half of this maximum (as a first approximation to a threads selection), with the results achieved when selecting the optimal number of threads at each level with our auto-tuning technique. Results are presented in tables 9 to 12. Both in the case of directly applying the MS to the optimization problem and for the HMS, the speed-up achieved with the model is nearly always better than that obtained when running roughly the maximum number of threads available or its half, and gets close to the experimental optimum value in most cases.

Table 8 Comparison of the highest experimental speed-up (exp), the modelled speed-up (mod) and that obtained in the experiments when using the auto-tuning methodology (exp-auto), for two combinations of the parameters $INEIni$, $PEIIni$ and $IIEIni$ (param1: 50,50,1; param2: 100,50,1) in the two-level parallel routine of `Initialize`, when applying the HMS in *Ben*.

	threads one-level		threads two-levels		speed-up		
parameters	exp	mod	exp	mod	exp	mod	exp-auto
param1	9	6	8	7	14	15	11
param2	9	8	4	9	15	24	14

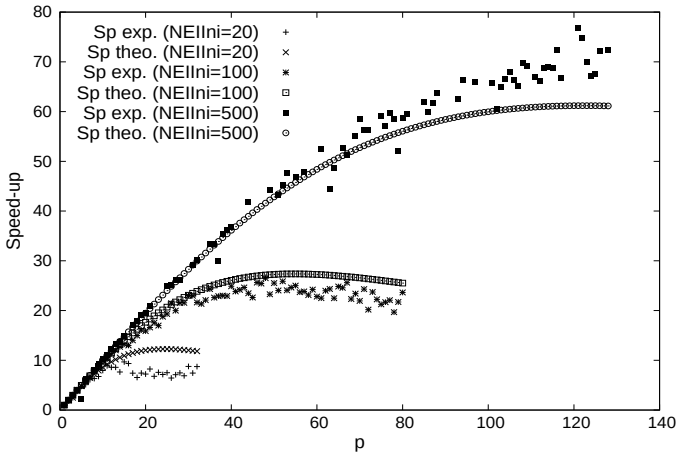

Figure 2 Theoretical and experimental speed-up when varying the number of threads for three parameters in a one-level parallel routine when applying the MS in *Ben*.

Table 9 Comparison of the speed-up obtained with the maximum number of threads available (max), with half of the maximum number of threads available (max/2), the highest experimental speed-up (exp) and that obtained with our auto-tuning methodology (auto), for $INEIni = 100$ and 500 in the one-level parallel routine when applying the MS in *Ben*.

	threads				speed-up			
$INEIni$	max	max / 2	exp	auto	max	max / 2	exp	auto
100	128	64	48	55	20	23	27	25
500	128	64	121	122	73	49	77	75

Table 10 Comparison of the speed-up obtained with the maximum number of threads available (max), with half of the maximum number of threads available (max/2), the highest experimental speed-up (exp) and that obtained with our auto-tuning methodology (auto), for two combinations of the parameters $INEIni$, $PEIIni$ and $IIEIni$ (param1: 100,50,10; param2: 500,100,5) in the two-level parallel routine when applying the MS in *Ben*.

	threads				speed-up			
parameters	max	max / 2	exp	auto	max	max / 2	exp	auto
param1	128	64	89	67	27	15	35	21
param2	128	64	128	150	78	52	78	78

So far we have checked the validity of the auto-tuning methodology in a large system as *Ben*, where the access to the shared-memory can suppose an additional delay in the execution time. For more general conclusions, the complete auto-tuning process has been analysed on *Saturno*. Tables 13 and

Table 11 Comparison of the speed-up obtained with the maximum number of threads available (max), with half of the maximum number of threads available (max/2), the highest experimental speed-up (exp) and that obtained with our auto-tuning methodology (auto), for $INEIni = 20$ and 100 in the one-level parallel routine when applying the HMS in *Ben*.

	threads				speed-up			
$INEIni$	max	max / 2	exp	auto	max	max / 2	exp	auto
20	128	64	22	15	11	11	11	8
100	128	64	24	34	11	11	12	12

Table 12 Comparison of the speed-up obtained with the maximum number of threads available (max), with half of the maximum number of threads available (max/2), the highest experimental speed-up (exp) and that obtained with our auto-tuning methodology (auto), for two combinations of the parameters $INEIni$, $PEIIni$ and $IIEIni$ (param1: 50,50,1; param2: 100,50,1) in the two-level parallel routine when applying the HMS in *Ben*.

	threads one-level				threads two-levels			
parameters	$\sqrt{max}$	$\sqrt{max/2}$	exp	auto	$\sqrt{max}$	$\sqrt{max/2}$	exp	auto
param1	11	8	9	6	11	8	8	7
param2	11	8	9	8	11	8	4	9

	speed-up			
parameters	$\sqrt{max}$	$\sqrt{max/2}$	exp	auto
param1	9	11	14	11
param2	11	12	15	14

14 give the values of the constants of the model obtained in the installation phase in *Saturno* for the whole MS and HMS.

Table 13 Values of the constants of the model (in seconds) for all the functions when applying the MS in *Saturno*.

	One-level parallel routines			Two-level parallel routines			
	Ini	Com	Inc	Imp-Ini	Imp-Ref	Imp-Com	Div
$k_s \cdot 10^4$	4.43	5.69	1.44	6.05	6.01	1.20	107
$k_{p,1} \cdot 10^3$	3.96	2.61	58.3	3.08	1.91	3.38	63.8
$k_{p,2} \cdot 10^3$	-	-	-	1.56	8.38	23.2	0.915

Table 14 Values of the constants of the model (in seconds) for all the functions when applying the HMS in *Saturno*.

	One-level parallel routines			Two-level parallel routines			
	Ini	Com	Inc	Imp-Ini	Imp-Ref	Imp-Com	Div
$k_s \cdot 10^2$	1.45	2.91	0.296	15.7	25.5	52.3	26.2
$k_{p,1} \cdot 10^2$	0.541	0.679	2.44	5.66	2.55	8.86	2.22
$k_{p,2} \cdot 10^2$	-	-	-	7.09	3.25	34.8	4.84

Taking into account the values of these constants, we can verify the validity of the methodology of auto-tuning running all the functions of the metaheuristic scheme. Table 15 gives the values of typical metaheuristic parameters used in the MS and in the HMS. We must consider that there are differences in the execution time between metaheuristics and hyperheuristics. While the metaheuristic executes

problem instances directly, the hyperheuristic is more time-consuming, since it executes different metaheuristics at the same time to optimize the resolution of the problem. So, these parameter values have been chosen because they produce metaheuristics and hyperheuristics of intermediate size, so allowing for a comprehensive study in a relatively reduced time.

Table 15 Values of the metaheuristic parameters used for the auto-tuning experiments when applying the MS (m1 and m2) and the HMS (h1 and h2) in *Saturno*.

	INEIni	FNEIni	PEIIni	IIEIni	STMIni	NBESel	NWESel	NBBCom	NBWCom
m1	20	20	50	20	-	10	10	20	5
m2	100	50	100	10	-	25	25	100	20
h1	10	10	100	1	7	5	5	20	5
h2	20	20	50	3	7	10	10	50	10

	NWWCom	PEIImp	IIEImp	SMIImp	PEDImp	IDEImp	SMDImp	NBEInc	LTMInc
m1	10	100	20	-	50	20	-	10	-
m2	5	50	10	-	100	20	-	25	-
h1	10	100	1	7	100	1	7	10	7
h2	5	20	3	7	20	3	7	10	7

Table 16 shows the optimum number of threads of the first and second level of parallelism obtained with auto-tuning for all the functions in the scheme for four metaheuristic parameter combinations in *Saturno*. Results are presented for the application of the MS and the HMS. The value of the second level parameter was fixed to 1 in the case of the MS because the second level was used to start more threads to work on the improvement of the fitness function but not to reduce the execution time.

Our goal is not to optimize the code but, given a parametrized code, to find satisfactory values for the parallelism parameters so that low execution times can be achieved automatically. Table 17 compares the speed-up achieved with the optimum number of threads given by the model with those achieved when the values of the parallelism parameters are selected without an auto-tuning method. Numbers of threads equal to the maximum number of cores available and to half this number are considered. Results are shown for basic and optimized implementations.

Table 16 Values of the parallelism parameters for four metaheuristic parameter combinations in *Saturno*. Optimum number of threads of the first level of parallelism for all the functions when applying the MS (m1 and m2). Optimum number of threads (levels one and two of parallelism) for all the functions when applying the HMS (h1 and h2).

	One-level parallel routines			Two-level parallel routines			
	TGEIni	TCPCom	TIEInc	TI1Ini	TR1Imp	TC1Imp	TD1Imp
m1	2	3	1	6	11	5	6
m2	3	5	1	14	9	5	13

		One-level parallel routines			Two-level parallel routines			
	level	TGEIni	TCPCom	TIEInc	TI_Ini	TR_Imp	TC_Imp	TD_Imp
h1	p_1	5	12	3	3	5	9	6
	p_2	-	-	-	3	4	2	3
h2	p_1	7	17	4	5	5	10	7
	p_2	-	-	-	4	4	2	3

Table 17 Speed-ups for various metaheuristic parameter combinations when applying all the functions in the MS (m1 and m2 in tables 15 and 16) and all the functions in the HMS (h1 and h2 in tables 15 and 16). Experimental values obtained with a number of threads equal to the number of cores available (max), and half the number of cores (max / 2), and values obtained with auto-tuning (auto). In *Saturno* and for basic and optimized implementations.

	speed-up					
	Basic			Optimized		
	max	max / 2	auto	max	max / 2	auto
m1	1	2	2	10	8	10
m2	5	6	6	14	10	16
h1	3	4	5	6	5	6
h2	4	5	5	6	6	7

In all cases it is observed that the speed-up obtained with the auto-tuning methodology improves or equals the values obtained with a number of threads selected in a non-optimal manner. This result is also observed in each function separately, which gives us an idea of the validity of the methodology studied.

Message-passing results

For the message-passing scheme, experiments were carried out for the different metaheuristic combinations shown in table 18. The values of the metaheuristic parameters have been selected to have metaheuristics with a wide variety of set sizes (between 20 and 500 elements) and with reasonable values for the parallelism parameters ($NGMPar$ equal to 5, and $NEMPar$ between 5 and 20). The number of iterations was fixed to 100 to compare the execution times between subsets of different sizes, and because it is sufficiently high to get good fitness results.

Table 18 Values of the metaheuristic parameters used in the message-passing experiments.

	INEIni	NFEIni	PEIIni	IIEIni	STMIni	NGMPar	NEMPar
m1	20	20	100	20	12	5	5
m2	100	100	50	10	4	5	15
m3	500	500	25	5	2	5	20

	NBESel	NWESel	NBBCom	NBWCom	NWWCom	PEIImp	IIEImp
m1	10	10	15	20	15	100	5
m2	50	50	90	100	90	50	5
m3	250	250	450	500	450	25	5

	SMIImp	PEDImp	IDEImp	SMDImp	NBEInc	LTMInc
m1	4	10	5	2	10	12
m2	4	10	5	2	50	12
m3	4	10	5	2	250	15

Similarly to shared-memory, to validate the auto-tuning methodology with the message-passing paradigm, the optimum parallelism parameters (p, $NEMPar$ and $NGMPar$) and the maximum speed-up achieved are calculated from the models for different metaheuristics using the system parameters obtained in the installation for different configurations of the computational system. These system parameters were calculated using small values of the metaheuristic parameters to reduce the installation time. It has been determined experimentally, for all the systems studied, that only the computation and immigration times contribute significantly to the total time modelled.

A first study of the methodology is carried out in *Saturno*, which is a shared-memory homogeneous system, in which we use the message-passing scheme, considering it as a distributed homogeneous

system. In table 19 the specific values of the constants in equation 4 for the computation time are presented. The values for the constants in equation 6, which models the immigration time, t_{imm}, are $A = -4$, $B = 0.0003$, $C = -0.006$ and $D = 0.0549$. Table 20 shows the speed-up and corresponding optimal number of processes when executing metaheuristics m1 and m2. We can see that the number of processes predicted by the model is quite close to the optimum obtained experimentally, with the speed-ups achieved with the auto-tuning methodology (exp-auto) close to the best obtained experimentally (exp).

Table 19 Values of the system and metaheuristic parameters considered in equation 4 in *Saturno*.

$func$	k_i	$Param_i$
Gen-Ini	$4.70 \cdot 10^{-3}$	50
Imp-Ini	$3.36 \cdot 10^{-4}$	$\frac{50 \cdot 75 \cdot 15}{100}$

$func$	k_j	$Param_j$
Com	$3.98 \cdot 10^{-5}$	$2 \cdot (45 + 50 + 45)$
Imp-Ref	$3.36 \cdot 10^{-4}$	$\frac{50 \cdot 75 \cdot 5}{100}$
Imp-Com	$6.72 \cdot 10^{-4}$	$\frac{(45+50+45) \cdot 75 \cdot 5}{100}$
Div-Ref	$3.53 \cdot 10^{-4}$	$\frac{50 \cdot 10 \cdot 5}{100}$
Div-Com	$7.06 \cdot 10^{-4}$	$\frac{(45+50+45) \cdot 10 \cdot 5}{100}$
Inc	$1.50 \cdot 10^{-5}$	$50 + 2 \cdot (45 + 50 + 45) - 25$

Table 20 Comparison of the highest experimental speed-up (exp), the modelled speed-up (mod) and that obtained experimentally when using the number of processes given by the auto-tuning methodology (exp-auto), for the metaheuristics m1 and m2 in table 18, when applying the metaheuristic scheme to PECEW in *Saturno*.

	MPI Processes		Speed-up		
metaheuristic	exp	mod	exp	mod	exp-auto
m1	16	20	15	11	12
m2	24	24	17	16	17

The same procedure was carried out in the small homogeneous cluster *Marte* + *Mercurio*, with different values for the installation constants of the model. As in the case of *Saturno*, the number of processes given by the model are close to the experimental ones for the metaheuristics studied, with the speed-ups achieved close to the best obtained experimentally (table 21).

Table 21 Comparison of the highest experimental speed-up (exp), the modelled speed-up (mod) and that obtained experimentally when using the number of processes given by the auto-tuning methodology (exp-auto), for the metaheuristics m1 and m2 in table 18, when applying the metaheuristic scheme to PECEW in *Marte* + *Mercurio*.

	MPI Processes		Speed-up		
metaheuristic	exp	mod	exp	mod	exp-auto
m1	11	12	10	7	9
m2	12	12	8	9	8

Finally, the auto-tuning technique was applied in the whole cluster *Jupiter* + *Luna* + *Saturno* + *Marte* + *Mercurio*. There were not big differences in the speed of the nodes when applying the sequential scheme, so the methodology for homogeneous systems was considered, with the execution time limited by the islands assigned to the slowest cores.

We can see the results of speed-up and optimal number of processes for each metaheuristic in table 22. The experimental and modelled values for the number of processes and speed-up are close, especially in m1 and m3. Furthermore, for the population sizes considered (up to a value of $NFEIni$=500), the results obtained with the process mapping given by the model suggest that the number of cores available in the cluster is sufficient to minimize the execution time effectively without the requirement of more processors.

Table 22 Comparison of the highest experimental speed-up (exp), the modelled speed-up (mod) and that obtained experimentally when using the number of processes given by the auto-tuning methodology (exp-auto), for the metaheuristics m1, m2 and m3 in table 18, when applying the metaheuristic scheme to PECEW in *Jupiter* + *Luna* + *Saturno* + *Marte* + *Mercurio*.

metaheuristic	MPI Processes		Speed-up		
	exp	mod	exp	mod	exp-auto
m1	20	18	10	7	9
m2	38	26	18	13	11
m3	40	37	24	23	20

Conclusions and future work

An auto-tuning methodology for parametrized shared-memory metaheuristic schemes that can in turn be applied to hyperheuristics based on metaheuristic schemes has been shown. The technique has been also introduced and analysed for the message-passing scheme. A problem of minimization of electricity consumption in exploitation of wells has been used as a test case, but similar results can be obtained with other problems. As far as we know, this is the first time that auto-tuning techniques are applied to parallel metaheuristics. The methodology provides satisfactory values for the number of threads to use in the application of the parallel metaheuristics and hyperheuristics in NUMA systems. The parallel parameters are also optimized for the message-passing scheme in a cluster with multicore nodes.

One possibility to improve the application of the hyperheuristic is to determine search ranges for each metaheuristic parameter, so reducing the possible values of the elements in the metaheuristic with which the hyperheuristic is implemented. For this, statistical analysis like those in [2] can be used.

As future research lines, the parametrized schemes and the auto-tuning methodology could be applied to other optimization problems. At present we are working with problems in the fields of data envelopment analysis, determination of chemical components of polymers and drug design. The inclusion of new basic metaheuristics, for example, Ant Colony Optimization or Particle Swarm Optimization, would generate new parametrized schemes with a larger number of metaheuristic parameters. Similar parametrized, parallel metaheuristic schemes, together with the corresponding auto-tuning methodologies, should be developed for GPU or manycore systems (Xeon Phi) and in heterogeneous clusters comprising nodes of multicores + multiple GPU or manycores. The use of large, heterogeneous clusters would be of especial interest for the application of hyperheuristics with large reference sets or with a high fitness function cost.

Competing interests
The authors declare that they have no competing interests.

Acknowledgements
This work was supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2012-38341-C04-03.

References

1. Almeida, F., Giménez, D., López-Espín, J.-J.: A parameterized shared-memory scheme for parameterized metaheuristics. The Journal of Supercomputing **58**(3), 292–301 (2011)
2. Almeida, F., Giménez, D., López-Espín, J.-J., Pérez-Pérez, M.: Parameterised schemes of metaheuristics: basic ideas and applications with Genetic algorithms, Scatter Search and GRASP. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans **43**(3), 570–586 (2013)
3. Cutillas-Lozano, L.-G., Cutillas-Lozano, J.-M., Giménez, D.: Modeling shared-memory metaheuristic schemes for electricity consumption. In: Distributed Computing and Artificial Intelligence - 9th International Conference, pp. 33–40 (2012)
4. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. Wiley-Interscience, New York (2005)
5. Luque, G., Alba, E.: Parallel Genetic Algorithms: Theory and Real World Applications. Springer, Berlin (2011)
6. Talbi, E.-G.: Metaheuristics - From Design to Implementation. Wiley, New York (2009)
7. Talbi, E.-G., Hasle, G.: Metaheuristics on GPUs. J. Parallel Distrib. Comput. **73**(1), 1–3 (2013)

8. Luong, T.V., Melab, N., Talbi, E.-G.: GPU-based island model for evolutionary algorithms. In: GECCO, pp. 1089–1096 (2010)
9. Mezmaz, M.-S., Kessaci, Y., Lee, Y.C., Melab, N., Talbi, E.-G., Zomaya, A.Y., Tuyttens, D.: A parallel island-based hybrid genetic algorithm for precedence-constrained applications to minimize energy consumption and makespan. In: GRID, pp. 274–281 (2010)
10. Mezmaz, M.-S., Melab, N., Talbi, E.-G.: Using the multi-start and island models for parallel multi-objective optimization on the computational grid. In: e-Science, p. 112 (2006)
11. Aida, K., Natsume, W., Futakata, Y.: Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In: Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), pp. 156–163 (2003)
12. Bendjoudi, A., Melab, N., Talbi, E.-G.: An adaptive hierarchical master-worker (ahmw) framework for grids - Application to B&B algorithms. J. Parallel Distrib. Comput. **72**(2), 120–131 (2012)
13. Cuenca, J., Giménez, D., González, J.: Architecture of an automatic tuned linear algebra library. Parallel Computing **30**(2), 187–220 (2004)
14. Frigo, M., Johnson, S.G.: FFTW: An Adaptive Software Architecture for the FFT. In: IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3, pp. 1381–1384 (1998)
15. Katagiri, T., Kise, K., Honda, H.: Effect of auto-tuning with user's knowledge for numerical software. In: S. Vassiliadis, J.L.G., Piuri, V. (eds.) Proceedings of the First Conference on Computing Frontiers, pp. 12–25 (2004)
16. Whaley, R.C., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing **27**(1-2), 3–35 (2001)
17. Vaessens, R., Aarts, E., Lenstra, J.: A local search template. In: Manner, R., Manderick, B. (eds.) Parallel Problem Solving from Nature, pp. 67–76. Elsevier, Amsterdam (1992)
18. Raidl, G.R.: A unified view on hybrid metaheuristics. In: Hybrid Metaheuristics, Third International Workshop, LNCS, vol. 4030, pp. 1–12 (2006)
19. Burke, E., Hart, E., Kendall, G., Newall, J., Ross, P., Schulenburg, S.: Hyper-heuristics: An Emerging Direction in Modern Search Technology, pp. 457–474. Kluwer, ??? (2003). Chap. 16. In Fred Glover and Gary Kochenberger, editors, Handbook of Meta-heuristics