Applied Soft Computing 53 (2017) 217–226

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Improving the performance of embedded systems with variable neighborhood search

Jesús Sánchez-Oro[a], Marc Sevaux[c], André Rossi[d], Rafael Martí[b], Abraham Duarte[a,*]

[a] Departamento de Informática y Estadística, Universidad Rey Juan Carlos, Móstoles, Spain
[b] Departamento Estadística e Investigación Operativa, Universidad de Valencia, Burjassot, Spain
[c] Lab-STICC, Centre de Recherche, Université de Bretagne-Sud, Lorient, France
[d] LERIA, Département d'informatique, Université d'Angers, Angers, France

ARTICLE INFO

Article history:
Received 18 December 2014
Received in revised form
25 November 2016
Accepted 19 December 2016
Available online 25 December 2016

Keywords:
Dynamic memory allocation
Variable neighborhood search
Embedded systems
Memory
Metaheuristics

ABSTRACT

Embedded systems have become an essential part of our lives, mainly due to the evolution of technology in the last years. However, the power consumption of these devices is one of their most important drawbacks. It has been proven that an efficient use of the memory of the device also improves its energy performance. This work efficiently solves the dynamic memory allocation problem, which can be formally defined as follows: given a program that has to be executed by a circuit, the objective is to fit that program in memory in such a way that the computing time required to execute it is minimized. In this work, we propose a parallel variable neighborhood search strategy to address this problem. We additionally compare this parallel procedure with the sequential version of the algorithm and with the best previous approach. Computational results show the superiority of our proposal, backed up with statistical tests.

1. Introduction

The advances in technology of the last years have been mainly focused on the development of smart portable devices (smartphones, smart-watches, etc.) and embedded systems that increase the functionality of traditional items, like on-board computers in cars (see [1,3,20], for recent works on embedded systems). However, there is still a big drawback while using them: their power consumption. In particular, these devices require a battery that needs to be frequently charged.

Designers of integrated circuits focus on improving the quality of their products in order to reduce their power consumption, which results in a longer battery duration. However, as technology evolves, those circuits are becoming more and more complex, and designers need computer tools to solve optimization problems that ensure a low cost in terms of both, silicon area and performance of the device. There exist several Computer Assisted Design (CAD) tools to generate the architecture of an integrated circuit by using its specifications. Unfortunately, these tools are not focused on reducing the energy consumption, which result in not so efficient circuits.

One of the main challenges in the design of embedded systems is the memory allocation, which is directly related to the energy consumption of a device [29]. As stated by electronic designers, if the device has a good performance in memory managing, the power consumption is also reduced [4]. In addition, it is possible to model the energy consumption of a program that is being executed on a device [17]. Parallelizing the access to data stored in the device is a common technique to reduce the execution time, and therefore the energy consumption.

Programs that are designed to be executed in an embedded device usually require of one or more data structures to store information related with the execution. The distribution of the memory during the execution is planned by the device, and it has a great effect in the time consumed by the application. This work is intended to reduce the execution time of programs on a device by finding the best memory location for the data structures used in that program.

The Dynamic Memory Allocation Problem (DMAP) has been recently studied from different perspectives. In particular, [28] proposed a mixed integer linear programming formulation and a variable neighborhood search algorithm for the static version of the problem. The mathematical formulation is only able to solve medium size instances in reasonable computational time when

* Corresponding author.
E-mail addresses: jesus.sanchezoro@urjc.es (J. Sánchez-Oro), marc.sevaux@univ-ubs.fr (M. Sevaux), andre.rossi@univ-angers.fr (A. Rossi), rafael.marti@uv.es (R. Martí), abraham.duarte@urjc.es (A. Duarte).

http://dx.doi.org/10.1016/j.asoc.2016.12.034
1568-4946/© 2016 Elsevier B.V. All rights reserved.

using commercial solvers like GNU Linear Programing Kit (GLPK) or Xpress-MP, as stated in that paper. It is proposed a Variable Neighborhood Search (VNS) algorithm hybridized with Tabu Search, which implements a tabu list to avoid repeating moves in each neighborhood exploration. The dynamic version of the problem was originally tackled in [27], where an integer linear programming formulation based on the previous one presented for the static approach and two heuristic iterative approaches were presented. The first heuristic, called Long-Term approach tries to adapt the heuristics for the static version, while the second heuristic, called Short-Term approach, tries to solve the dynamic version by including new heuristics. The computational experiments show that the Long-Term approach is consistently better than the Short-Term approach. Finally, [26] proposed a greedy randomized adaptive search procedure (GRASP) which includes ejection chains in the improvement method. The ejection chains are used to make the movement of an element dependent on a previously element moved, thus exploring a large neighborhood for improved outcomes.

Memory allocation is performed manually by the designer on those embedded systems that do not have an operating system. However, when an operating system is available (which tends to be the most frequent case), it may rely on overly-simplistic heuristics to make these decisions that are critical for embedded system performance (most of the time without any consideration for energy). Consequently, we propose to generate high-performance memory allocations for embedded system applications using VNS, in order to help the designer (or the operating system) to achieve this difficult and important task. On the one hand, it is difficult since memory allocation is an extension of the well-known bin packing problem which is a $\mathcal{NP}$-complete problem [18]. On the other hand, it is important because memory allocation has a significant impact on power consumption, and energy saving is of paramount importance since battery capacity has not increased at the same pace as mobile applications complexity and customers demand for interactive, multimedia and web oriented applications.

Designers rely on computer aided tools to build these applications. One of these tools is SoftExplorer [25]; it has been developed in Lab-STICC laboratory, and takes both the C code of an application and the features of the embedded system as input, and returns a power and energy estimation of running the C code on the aforementioned architecture. This tool performs code analysis in order to identify all the data structures of the application and their sizes, as well as the moments when they require concurrent accesses. One of its module, called MemExplorer, builds a memory allocation for the data structures of the application knowing the system architecture. But MemExplorer currently uses GLPK for solving an ILP-based model, which is convenient for small applications, but cannot meet the challenge of the ever increasing complexity of mobile applications.

In this paper, we propose different heuristic methods and embed them into a Variable Neighborhood Search (VNS) scheme to solve the Dynamic Memory Allocation Problem. More precisely, we propose a semi-greedy constructive algorithm, a randomized shake procedure, and a local search method. We additionally propose a parallel implementation of a VNS based on the Replicated-Shaking methodology. We provide an experimental comparison among our VNS variants and with the best previous methods of the literature. The experimentation reveals that our best procedures improve the state of the art in both quality and computing time. This fact is confirmed by non-parametric statistical tests. The rest of the paper is organized as follows. Section 2 formally describes the optimization problem tackled in this paper. The algorithmic approach based on variable neighborhood search is described in Section 3. The parallelization of variable neighborhood search is presented in Section 4. The computational experiments are introduced in Section 5. Finally, the conclusions derived from the research are presented in Section 6.

2. Problem description

The memory structure usually consists of m memory banks with a fixed capacity of u_j kilobytes (kB) for each $j \in \{1, \ldots, m\}$. Additionally, there exists an external memory $(m+1)$ whose capacity is considered unlimited ($u_{m+1} = \infty$). Each memory bank (including the external memory) is used to store the data structures required by the programs. In general, it is considered that the device needs q milliseconds for accessing to a memory bank, but the access to external memory is penalized by a factor p, resulting in an access time of $p \times q$ for the external memory.

The execution of a program is split in T time intervals with different lengths. A time interval t is defined as the set of instructions A_t that must be executed sequentially, with no memory reallocation allowed. That is, the device can reallocate the data structures used in the program only before or after the set of instructions of each time interval is executed. The program uses n data structures that have to be stored in memory banks or external memory to be available for it. The size s_i of each data structure d_i (with $i \in \{1, \ldots, n\}$) is expressed in kB. Each instruction of the program is defined by the data structures that are required and the cost of executing it. Given the instruction a_k, where k indicates the instruction order within the program (i.e., $1 \leq k \leq \sum_{t=1}^{T} |A_t|$), it is represented as $a_k = \{d_i, d_j, c_k\}$, where d_i, d_j are the data structures that need to be accessed by the instruction, and c_k is the associated cost of executing it.

A solution S of this problem consists of determining the memory state (i.e., location of data structures in memory banks) of the device for each time interval. The evaluation of the solution can be split into two parts: the move cost and the access cost. The former refers to the cost of moving a data structure from a memory bank to another in consecutive time intervals, while the latter represents the cost for executing each instruction of the program.

The move cost of a data structure d_i in a given time interval t is computed as the time needed to move d_i from the memory bank in which it was stored in $t-1$, to the new memory bank in t. For the sake of simplicity, we define $b(d_i, t)$ as the function that finds the memory bank in which d_i is stored at t. Then, the move cost is formally defined as follows:

$$MoveCost(d_i, t) = \begin{cases} 0 & \text{if} b(d_i, t-1) = b(d_i, t) \\ l \cdot s_i & \text{if} b(d_i, t-1) \neq b(d_i, t) \neq m+1 \\ v \cdot s_i & \text{otherwise} \end{cases} \quad (1)$$

where l and v represent the device transfer rates (kB/millisecond) for moving a data structure between memory banks or between the external memory and a memory bank, respectively. This equation describes three different cases. The first one represents the situation where the data structure remains in the same memory bank during consecutive time intervals. Therefore, its associated cost is 0. The second case represents the cost of moving a given data structure, d_i, between memory banks. This cost only depends on the size of d_i, represented by s_i, and the transfer rate between memory banks (l). The last case represents a move of a data structure where the external memory is involved (either as origin or destination of the data structure). This cost also depends on the size of the data structure (s_i) and the transfer rate v. It is important to remark that moving a data structure from/to external memory is usually more time consuming than moving it between memory banks ($v \geq l$).

The access cost is evaluated for each instruction that is executed in a given program. It is worth mentioning that the device is able to access simultaneously to all memory banks in a given time interval. That is, if two data structures are stored in different memory banks in the same time interval, then the device only needs to access once to its memory, retrieving both data structures. However, if both data structures are stored in the same memory bank, then the device needs to access twice to its memory. The access cost for executing the instruction is evaluated as follows:

$$AccessCost(a_k) = \begin{cases} c_k & \text{if} b(d_i, t) \neq b(d_j, t) \neq m+1 \\ 2 \cdot c_k & \text{if} b(d_i, t) = b(d_j, t) \neq m+1 \\ 2 \cdot p \cdot c_k & \text{if} b(d_i, t) = b(d_j, t) = m+1 \\ p \cdot c_k & \text{otherwise} \end{cases} \quad (2)$$

where p is a penalty factor for accessing the external memory. This equation represents four different situations. In the first case, both data structures, d_i and d_j, are located in different memory banks. If both data structures are in the same memory bank (second case); then the cost must be multiplied by two, because the device will need to access twice its memory. The third and fourth cases are variations of the first and second but considering that one of the involved data structures is located in the external memory.

Once the move and access cost have been introduced, we can now define the complete evaluation of a solution. Specifically, given a program with A_t instructions, split in T time intervals, that uses n data structures, a solution S to the Dynamic Memory Allocation Problem, *DMAP*, is evaluated as:

$$DMAP(S) = \sum_{t \in T} \left(\sum_{i=1}^{n} MoveCost(d_i, t) + \sum_{k=1}^{|A_t|} AccessCost(a_k) \right) \quad (3)$$

Then, the objective of this optimization problem is to find the solution $S^\star$ with the minimum *DMAP*-value of all possible solutions $\mathbb{S}$:

$$S^\star = arg \min_{S \in \mathbb{S}} DMAP(S) \quad (4)$$

Let us illustrate the evaluation of a solution to the DMAP problem with an example, depicted in Fig. 1, in which the memory consists of three memory banks (b_1, b_2, b_3), and the external memory (b_{ext}).

The program executes 8 instructions, a_1, a_2, . . ., a_8, accessing to 6 data structures d_1, d_2, . . ., d_6, distributed in 3 time intervals $T = \{t_1, t_2, t_3\}$. It is important to notice that before executing the program, all the data structures are located in external memory. We model the initial configuration by considering an additional time interval, t_0, also shown in Fig. 1. The time interval t_1 consists of executing three instructions, a_1, a_2, and a_3. For the sake of simplicity, we represent each instruction by identifying the data structures involved, and the associated cost. For example, instruction $a_1 = (d_1, d_2, 100)$ indicates that a_1 uses two data structures d_1 and d_2 with a cost of 100. The second and third intervals in this example contain, respectively, instructions $\{a_4, a_5\}$ and $\{a_6, a_7, a_8\}$.

The capacity of each memory bank depends on the specific electronic device. In this example, we consider that each memory bank has a capacity of 192 kB. Let us assume that all data structures have the same size and are equal to 64 kB (i.e., $s_i = 64$ for $1 \leq i \leq 6$). Therefore, each memory bank is only able to store up to 192/64 = 3 data structures.

Table 1 breaks down the evaluation of the solution depicted in Fig. 1. The first column indicates the time interval that is being evaluated, while the second and third one present the evaluation of move and access costs, respectively. The transfer rates considered in this example are $l = 1$ and $v = 3$, and the penalty factor for accessing the external memory is $p = 4$. Let us first analyze the move costs. In the time interval t_1, the data structures d_1, d_2, d_5, and d_6 are moved from the external memory to a memory bank. So the move cost for each data structure is the transfer rate from external memory (v) multiplied by the size of the data structure (64 kB). In the second time interval, d_2 and d_4 are moved from, or to the external memory, so the move cost is the size of the data structure multiplied by the transfer rate v. However, d_6 is moved between memory banks, resulting in a move cost of the transfer rate l multiplied by the size of the structure. In the last time interval, d_1 and d_6 are moved between memory banks, using the transfer rate l to evaluate their move cost. On the contrary, d_3 and d_4 are moved from, or to external memory, using the transfer rate v. Notice that all data structures that remain in the same memory bank (or the external memory) in two consecutive time intervals do not affect the evaluation of the move cost (see for example d_3 and d_4 from t_0 to t_1)

We now analyze the access costs. In the first time interval, the cost of a_1 is directly the instruction cost because it accesses to data structures located in different memory banks. However, the cost of a_2 is multiplied by 2 because the data structures are in the same memory bank, and therefore, it needs to be accessed twice.

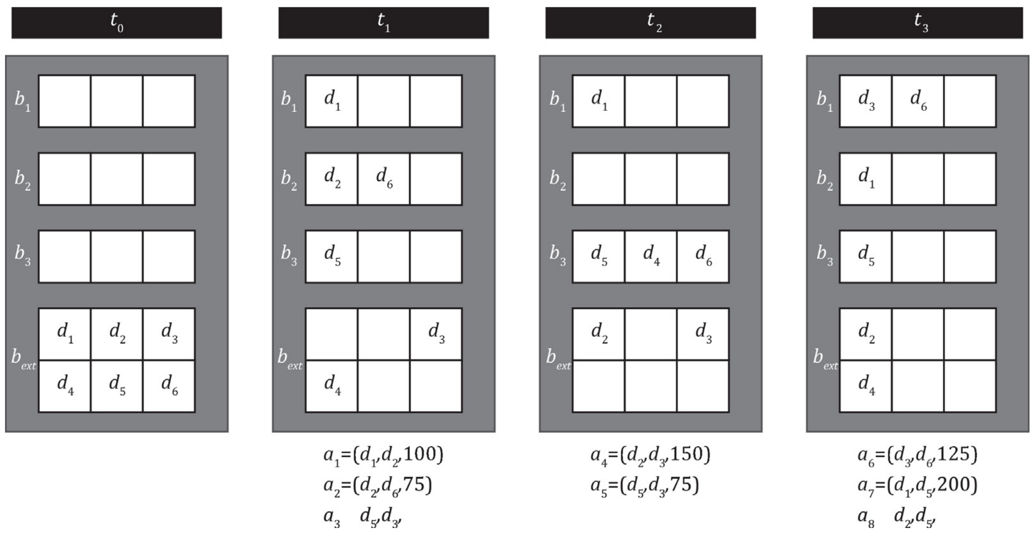

Fig. 1. Example of a possible DMAP solution for a program split in three time intervals.

Table 1
Evaluation of the solution shown in Fig. 1, where transfer rates are $l = 1$, $v = 3$ and the penalty factor for accessing external memory is $p = 4$.

Time interval	Move costs	Access costs
t_1	$MoveCost(d_1, 1) = 3 \times 64 = 192$ $MoveCost(d_2, 1) = 3 \times 64 = 192$ $MoveCost(d_5, 1) = 3 \times 64 = 192$ $MoveCost(d_6, 1) = 3 \times 64 = 192$	$AccessCost(a_1) = 100$ $AccessCost(a_2) = 2 \times 75 = 150$ $AccessCost(a_3) = 4 \times 100 = 400$
Total t_1	768	650
t_2	$MoveCost(d_2, 2) = 3 \times 64 = 192$ $MoveCost(d_4, 2) = 3 \times 64 = 192$ $MoveCost(d_6, 2) = 1 \times 64 = 64$	$AccessCost(a_1) = 2 \times 4 \times 150 = 1200$ $AccessCost(a_2) = 4 \times 75 = 300$
Total t_2	448	1500
t_3	$MoveCost(d_1, 3) = 1 \times 64 = 64$ $MoveCost(d_3, 3) = 3 \times 64 = 192$ $MoveCost(d_4, 3) = 3 \times 64 = 192$ $MoveCost(d_6, 3) = 1 \times 64 = 64$	$AccessCost(a_1) = 2 \times 100$ $AccessCost(a_2) = 200$ $AccessCost(a_3) = 4 \times 50 = 200$
Total t_3	512	500
DMAP	$(768 + 448 + 512) + (650 + 1500 + 500) = 4378$	

The access cost of a_3 is multiplied by the penalty factor p since it accesses to a data structure stored in the external memory. In the second time interval, both data structures accessed by a_4 are stored in the external memory, so the device has to access it twice, each access with its corresponding penalty p. However, only one of the structures accessed by a_5 is stored in the external memory, so penalty is only applied once. In the last time interval, a_6 accesses to two data structures stored in the same memory bank, while a_7 requires two data structures located in different memory banks. Finally, a_8 accesses to a data structure stored in the external memory, being affected by the penalty factor p.

The evaluation of the solution is calculated as the sum of all move and access costs for each time interval. Therefore, the solution S depicted in Fig. 1, presents a *DMAP*-value of $DMAP(S) = 4378$.

3. Variable Neighborhood Search

Variable Neighborhood Search (VNS) is a metaheuristic which was originally proposed by [19] as a general framework to solve hard optimization problems. This methodology is based on performing systematic changes of neighborhoods during the search space exploration. VNS has evolved in recent years, resulting in a large variety of strategies. Some of the most relevant variants are Reduced VNS (RVNS), Variable Neighborhood Descent (VND), Basic VNS (BVNS), Skewed VNS (SVNS), General VNS (GVNS) or Variable Neighborhood Decomposition Search (VNDS), among others. See [15] for a complete survey on VNS. We refer the reader to [24,23,22] to some recent and successful applications of VNS to hard optimization problems.

In this paper, we focus on the Basic VNS (BVNS) approach. BVNS algorithm combines deterministic and random changes of neighborhoods, where the deterministic component is given by an improvement procedure, and the random component is given by a perturbation of the solution. The improvement strategy in BVNS is typically a local search, while the perturbation mechanism is based on a shake procedure which alters the structure of the solution in a neighborhood given by a parameter k. Starting from an initial solution (which can be greedily or randomly constructed), the algorithm explores its closest neighborhood. If no improvement is found, BVNS jumps to a larger neighborhood in order to find better solutions which are further from the initial one. If an improvement is found, BVNS re-starts the search from the closest neighborhood of the incumbent solution. The stopping criterion for BVNS is usually determined by a fixed computing time or number of iterations.

Algorithm 1. BVNS($k_{\max}$, *iters*)

```
1:  $S \leftarrow Construct()$
2:  for $i \in 1 \ldots iters$ do
3:    $k \leftarrow 1$
4:    while $k \leq k_{\max}$ do
5:      $S' \leftarrow Shake(S, k)$
6:      $S'' \leftarrow Improve(S')$
7:      $NeighborhoodChange(S, S'', k)$
8:    end while
9:  end for
10: return $S$
```

Algorithm 1 presents the pseudo-code of the proposed method. It starts by constructing an initial solution S with the method described in Section 3.1 (step 1). The stopping criterion selected is the maximum number of iterations, determined by parameter *iters* (steps 2–9). Each iteration starts the search from the first neighborhood (step 3), and explores up to the maximum neighborhood, given by parameter $k_{\max}$ (steps 4–8). For each neighborhood, BVNS perturbs the incumbent solution with the *Shake* method (described in Section 3.2) to generate a new solution in the current neighborhood (step 5). Then, the solution is improved with the local search method proposed in Section 3.3 (step 6). Finally, if there has been an improvement, the incumbent solution is updated and the search re-starts from the first neighborhood. Otherwise, it continues with the next one (step 7).

3.1. *Constructive method*

We propose a semi-greedy procedure, called CBI (Constructive Based on Instructions), to construct solutions. It is based on initializing the state of the time interval t by coping the state of $t - 1$. The constructive method stochastically selects the instructions belonging to the current time interval. We use the classical fitness proportionate (roulette wheel) selection strategy [8,12], where the larger the cost of the instruction the larger the probability to be selected. Once an instruction is selected, the data structures involved in it are located in the best memory bank. This best memory bank, $b^\star$, for a data structure d_i in a time interval t is determined by moving d_i to all possible memory banks b_j (with $1 \leq j \leq m + 1$, where the external memory is also considered) and selecting the bank which produces the solution with the lowest *DMAP*-value. More formally:

$$b^\star \leftarrow arg \min_{b_j : 1 \leq j \leq m} DMAP(Move(d_i, b_j, t)) \tag{5}$$

We illustrate how this method works with the example shown in Fig. 2. In particular, we show how CBI constructs the first

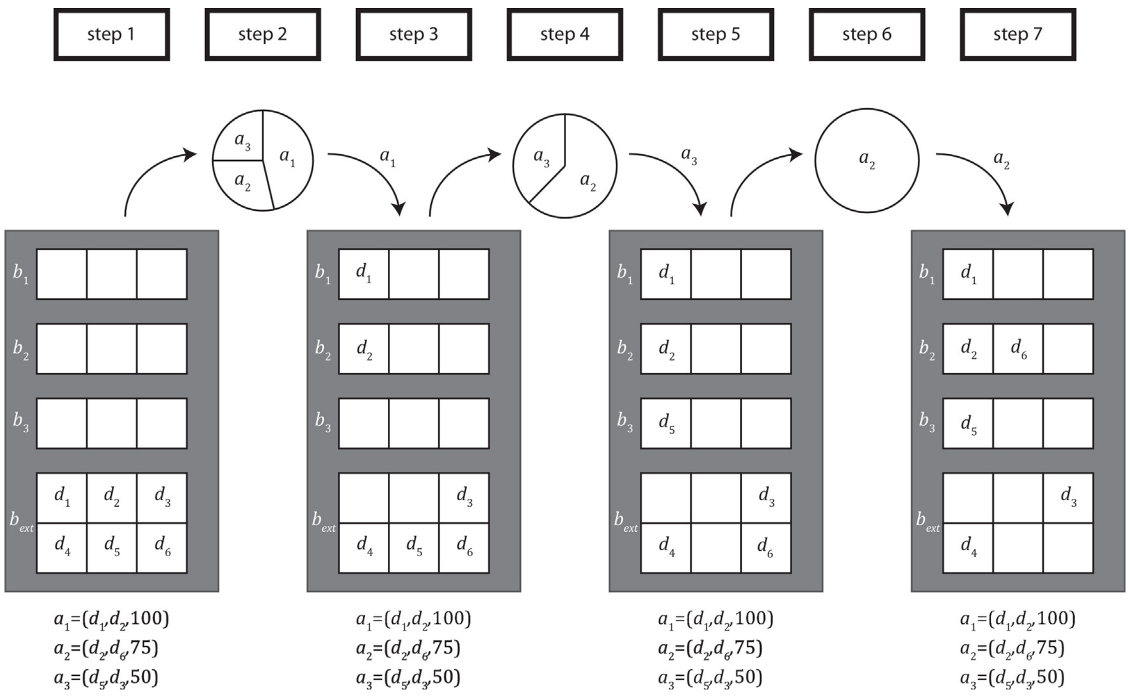

Fig. 2. Constructive procedure based on the roulette wheel strategy. Instructions that can be selected are highlighted in bold font.

time interval. Then, the first step consists of copying the state of the memory from the previous time interval (i.e., time interval t_0, where all data structures are located in the external memory). The second step consists of determining the probability of selecting an instruction. In this example, there are tree different instructions a_1, a_2, and a_3 whose costs are 100, 75, and 50, respectively. Therefore, the probability of selecting the first instruction is $100/(100+75+50)=0.45$. Similarly, the probabilities of selecting a_2 and a_3 are 0.33 and 0.22, respectively. The roulette wheel selection strategy stochastically selects one of these instructions according to their probabilities. Let us assume that the selected instruction is a_1. Then, in the third step, the data structures involved (i.e., d_1 and d_2) are located in the best bank. The fourth step now proceeds in a similar way, by computing the probabilities of selecting the remaining instructions (i.e., a_2 and a_3 with probabilities 0.60 and 0.40). Let us assume that now the roulette strategy randomly selects a_2. Then, the involved data structures (d_3 and d_5) are located in the best memory bank (fifth step). The sixth step selects the non-used instruction (a_2) and the last step locates the corresponding data structures (d_2 and d_6) in the best bank. At this point, all the data structures required in the time interval t_1 have been assigned to a memory bank, and the method continues constructing the next time interval by following the same approach.

The method ends when the data structures for each instruction in all time intervals have been moved to the best bank. Considering that the proposed method is a semi-greedy procedure, different executions may produce different solutions. For that reason, the computational experiments described in Section 5 discuss, in terms of quality and computing time, the results obtained when performing different number of constructions.

3.2. Shake

The shake procedure is a key step in the Variable Neighborhood Search framework, to generate a solution relatively different to the current one. The main objective of the shake procedure is to diversify the search, so the new solution is usually generated by randomly perturbing the incumbent solution. In order to design this procedure, it is necessary to define a move operator that conforms the corresponding neighborhood. Recent works have studied more advanced shake techniques that balances the diversification and intensification of the search. See for instance, [13,24].

Given a solution S, the proposed move operator basically removes a data structure from its current memory bank randomly selected (including the external memory) and includes it in a different memory bank (or the external memory) producing a new solution S'. For the sake of simplicity, we represent this operation as $S' = Move(S, d_i, b_j, t)$, which moves the data structure d_i from its current bank to b_j in time interval t. It is important to remark that the *Move*-operator only produces feasible solutions. Therefore, d_i is included in b_j if and only if b_j has enough free space to store d_i.

We now define a compound move operator, denoted as $Move_k$, which applies k times *Move* in each time interval $t \in T$. More formally,

$$Move_k(S,t) = \begin{cases} Move_{k-1}(Move(S, d_i, b_j, t), t) & \text{if} k > 1 \\ Move(S, d_i, b_j, t) & \text{if} k = 1 \end{cases} \quad (6)$$

where the data structure d_i is randomly selected in the range $1 \le i \le n$ and the bank b_j is randomly selected in the range $1 \le j \le m$. Note that this new move is based on *Move* and, therefore, it only produces feasible solutions.

In this work, we define the neighborhood N_k of a solution S as the set of solutions that can be generated by moving k data structures in each time interval $t \in T$. We have:

$$N_k(S) = \{S' : Move_k(S,t),\ \forall t \in T\} \quad (7)$$

Given a solution S, our shake procedure generates a random solution in the neighborhood $N_k(S)$. Specifically, the method traverses each time interval selecting at random k data structures in each one. Then, each data structure selected is moved to a random memory bank (including external memory) with enough free space to store it. The shake procedure finishes when the operator $Move_k$ has been applied to each time interval. We have empirically found that the random component of this compound move, induces a diversity pattern in the search that results, over the long term, in improved outcomes.

3.3. Local search

The local search method corresponds to the intensification part of the VNS framework. Its main objective is to find a local optimum in the current neighborhood. Obviously, there is a compromise between the number of iterations required to find a local optimum and the associated computing time. In general, iterations performed in the first improvement strategy are more efficient than those in the best improvement one, since the former only evaluates part of the neighborhood, while the latter explores it completely. On the other hand, the improvement obtained in the first improvement strategy is typically smaller than the one achieved by the best improvement strategy, requiring in general more iterations to obtain the local optimum. Additionally, the best improvement strategy is usually more adequate to perform efficient catching and updating mechanisms, which allows the search to efficiently explores the neighborhood (see [16] for further details).

In [14], an empirical study on traveling salesman problem was conducted to compare the first and the best improvement strategy within 2-opt neighborhood structure. It appeared that the quality of the final solution depends on the quality of the initial solution: (i) if random initial solution is chosen, the better and faster is the first improvement strategy; (ii) the opposite holds if the greedy solution is taken as initial one.

According to previous results in the related literature and considering the nature of the DMAP, we propose an algorithm that systematically explores the local neighborhood of the incumbent solution by following the best improving strategy. According to the solution representation described above, the size of this neighborhood is bounded by $n \times m$, where n is the number of data structures and m is the number of available memory banks. In order to increase the effectiveness and efficiency of the local search, it first determines the candidate data structure to be moved by using an estimation of the contribution of each data structure, d_i, in the time interval, t, to the value of the objective function. We denote it as $\Delta(d_i, t)$ and it is formally defined as:

$$\Delta(d_i, t) = MoveCost(d_i, t) + \sum_{a_k \in A(d_i, t)} AccessCost(a_k) \quad (8)$$

where $A(d_i, t)$ represents the set of instructions that involves d_i in a time interval t. Hence:

$$A(d_i, t) = \{a_k = (d_p, d_1, c_k) : a_k \in A_t \wedge (p = i \vee q = i)\} \quad (9)$$

The local search scans all the data structures in descending order of Δ-values. The rationale behind this strategy is to first move those data structures with large contribution to the objective function, since DMAP is a minimization problem.

Once we have identified the data structure to be moved, the second main component in the local search is to determine the destination of such data structure. In order to do so, we select the best bank for each data structure by using the equation introduced in Section 3.1.

The local search iterates over all time intervals sequentially, where the first time interval is examined at the beginning. We proceed in this way since in our estimation, a time interval is only affected by the previous one. Therefore, the estimation of the first time interval is only affected by the initial state t_0, which remains constant during the search (i.e., all data structures in the external memory). The same logic is applied for the rest of time intervals. The method ends when no improvement is found in any time interval.

4. Parallel VNS

The constant evolution of the technology has lead us to use computers with several processors, which are able to simultaneously execute different programs. The use of multi-processors can substantially increase the performance of the algorithms. Note that, in general, the methods must be re-designed to be executed in parallel in order to take advantages of the multi-processor architecture.

Nowadays there exist several parallelization technologies that can be used to implement parallel algorithms (OpenMP, threads, CUDA, etc.). We refer the reader to [21], [10], and [5] for some tutorials on parallel programing. In this work, we focus on the use of threads. A thread is defined as a fragment of code that is independently executed in a processor. It is important to remark that there exist two types of processors: physical and logical. The former refers to the real processors that are installed in the computer. The latter, however, refers to the number of threads that can be independently executed in a computer. In the remaining of the paper, we only refer to logical processors since parallel programming tools have only access to them.

The parallelization of a metaheuristic has the following two goals: the reduction of the computing time of the corresponding algorithm, or the exploration of a wider portion of the search space [11]. Furthermore, it is essential to know which parts of the algorithm can be re-designed to be executed in a parallel way. This section proposes a parallel version of the Basic VNS described in Section 3. See [11,7,9] for some successful applications of parallel VNS.

The parallelization of VNS was originally introduced in [11], where three parallel approaches are proposed. The first one is called Synchronous Parallel VNS (SPVNS) and it is focused on parallelizing the local search of the sequential Basic VNS. The main objective is to reduce the computational time, since the local search is usually the most time-consuming part of the VNS algorithm. Then, SPVNS splits the local search in order to execute it in different threads simultaneously. The second approach, called Replicated Parallel VNS (RPVNS) is based on the exploration of a wider portion of the solution space by using multi-start strategies. In particular, the algorithm executes simultaneously several Basic VNS methods, each one in a different processor. The third variant, called Replicated Shaking VNS (RSVNS), is based on a classical master-slave approach. In particular, the master processor (the main program) executes a sequential Basic VNS, and each slave processor executes an independent shake and local search method.

Cooperative Neighborhood VNS (CNVNS) is proposed in [7], where several independent Basic VNS methods are launched in each processor. Each independent Basic VNS communicates to the main processor its local best solution. When the global best solution is updated, the main processor communicates it to the processors executing the Basic VNS to continue the search from the new best solution.

In order to select a VNS parallel strategy we must analyze how each one fits to the Dynamic Memory Allocation Problem. In particular, we believe that SPVNS is not well-suited for this problem, since the local search is eminently sequential (i.e., the current iteration strongly depends on the previous one). The RPVNS follows a multi-start strategy, which is useful when using constructive procedures that generates highly diverse solutions. In our case, the semi-greedy constructive method described in Section 3.1 produces different but not diverse solutions. The CNVNS is basically conceived for optimization problems where different type of move are defined, which determine different topologies of the search space. The moves associated to the DMAP are always based on moving a data structure from one bank to another.

This paper therefore focuses on the Replicated Shaking VNS (RSVNS). This methodology allows the search to explore simultaneously p solutions (where p is the number of threads) in the current neighborhood. This algorithm controls the cooperation among threads with a classical master-slave approach. Specifically, the master thread executes a sequential VNS, while the shake and

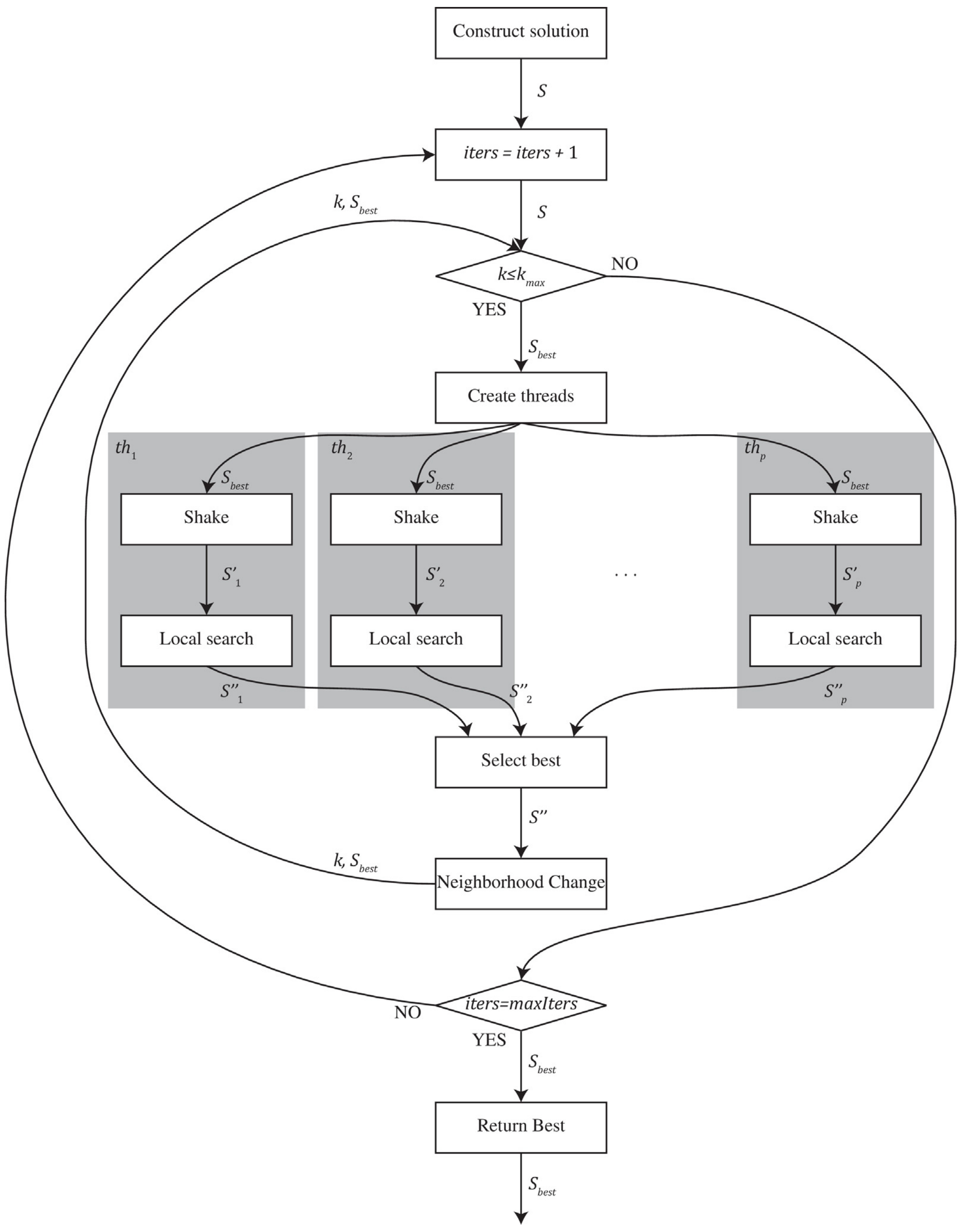

Fig. 3. Flow diagram of the Replicated Shaking Variable Neighborhood Search.

local search methods are executed in the slave threads. Each slave thread executes the shake and local search methods independently, returning the best solution found to the master thread. Then, the master thread updates (if necessary) the incumbent solution with the best solution recovered from the slave threads. Therefore, in a single iteration, this variant is able to explore p solutions while the sequential version explores only one. Notice that if the number of processors is larger than or equal to p, the computing time of the parallel and sequential versions should be similar. Otherwise, the parallel version could be even slower than the sequential one.

Fig. 3 shows a flow diagram of the RSVNS. The algorithm starts by constructing a solution with the method described in Section 3.1, becoming the current best solution, S_{best}. After that, the method starts the search in the first neighborhood ($k = 1$). The parallel method creates p different threads, where each one executes shake and local search methods, independently.

The threads perform a parallel exploration of the search space, in such a way that each thread th_i (with $1 \leq i \leq p$) starts from S_{best}. Each shake procedure perturbs S_{best}, producing a solution S'_i in the current neighborhood N_k. Finally, each thread is in charge of improving its corresponding perturbed solution, generating a local optimum S''_i. We use a barrier synchronization method to wait until all threads have finished their task. Then, the parallel method selects the best solution S'', among the solutions S''_1 to S''_p.

The neighborhood-change method determines whether S'' improves upon the current best solution (S_{best}), or not. If so, the method updates S_{best} and resorts to the first neighborhood ($k = 1$); otherwise, it continues the search by exploring the next neighborhood (i.e., $k = k + 1$).

Table 2
Results obtained by CBI when performing 1, 10, 50, 100, and 1000 constructions.

	Avg.	Dev(%)	#Best	Time (s)
CBI(1)	674714.53	1.71	3	0.03
CBI(10)	670925.23	0.78	3	0.31
CBI(50)	670435.33	0.51	3	1.54
CBI(100)	669448.43	0.36	4	3.07
CBI(1000)	667966.93	0.00	10	31.74

Table 3
Comparison of the BVNS performance varying the $k_{\max}$ parameter.

	Avg.	Dev(%)	#Best	Time (s)
BVNS(0.10n)	624864.93	5.68	4	17.94
BVNS(0.25n)	620552.73	0.22	6	25.72
BVNS(0.50n)	620315.03	0.00	9	29.80
CBI(1000)	667966.93	20.23	0	31.74

The current execution of the parallel VNS ends when the largest neighborhood (k_{max}) is reached. If the maximum number of iterations is not exceeded, the method performs the next repetition. Otherwise, it returns the best solution found.

5. Computational experiments

This section discusses the results obtained by the sequential and parallel VNS algorithms proposed, and then compares it with the best previous algorithms for the DMAP problem. All the algorithms were coded in Java 8 and the experiments were conducted in an Intel Core i7 920 CPU (2.67 GHz) with 8 GB RAM. The test bed used in the experiments is the same set of 44 instances used in previous works. Fifteen of these instances were directly obtained from real design problems addressed in the Lab-STICC laboratory, while the remaining 29 were derived from the DIMACS[1] set, adding them random values to: instructions costs, number of memory banks, capacity of the memory banks, number of data structures, and sizes of data structures. We consider standard values for the transfer rates ($l = 1$ ms/kB and $v = 4$ ms/kB), and the penalty factor for accessing to external memory ($p = 16$).

The experiments are separated into two parts. The preliminary experimentation is intended to fulfill two goals. On the one hand, to find the best parameters for our algorithms ($k_{\max}$ and *iters*). On the other hand, to evaluate the impact of the proposed strategies. The final experimentation is oriented to compare our best variant with the best previous algorithms found in the related literature.

The preliminary experimentation is conducted with a subset of 10 representative instances of the whole set to avoid over-training in the final comparison. All these experiments report the following statistics: the average value of the objective function, Avg.; the average deviation with respect to the best solution found in the experiment, Dev (%); the number of best solutions found in the experiment, #Best; and the average computing time in seconds, Time (s).

The first experiment evaluates the performance of the constructive method CBI proposed in Section 3.1. In particular, this experiment compares the results obtained by this method when performing different number of constructions. Table 2 shows the results obtained, where the number of constructions is specified in parenthesis.

As expected, the larger the number of constructions, the better the quality and, naturally, the larger the computing time. We can observe that Time (s) grows linearly with the number of constructions. However, the quality seems to be stalling after 100 constructions. The average computing time of CBI(100) is about 3 seconds (with a deviation of 0.36%), while CBI(1000) needs about 31 seconds (with a deviation of 0.00%). We therefore select CBI(100) since it exhibits a trade-off between quality and efficiency, suitable to become part of the VNS method.

The second experiment is performed to study how the largest neighborhood explored affects the VNS method. Specifically, we test three different values for the $k_{\max}$ parameter: 0.10n, 0.25n, and 0.50n, where n is the number of data structures. Table 3 shows the results obtained with the Basic VNS algorithm when using these values (indicated in parenthesis). We include CBI(1000) as a baseline method.

The first conclusion that can be drawn from the results in Table 3 is that the constructive method is not competitive when comparing it with advanced metaheuristics. In particular, this algorithm presents a deviation of 20.23%, while the worst variant tested of Basic VNS has a deviation of 5.65% (requiring similar computing times). The best results are obtained with $k_{\max} = 0.50n$, i.e., with the variant that explores the largest number of neighborhoods. It obtains the best results in 9 (out of 10) instances. In addition, the computing time of this variant is 29.80 s, which although is larger than the other VNS variants in this table, it is still moderate.

We do not test larger values of $k_{\max}$ since the method would be exploring solutions located in a rather far neighborhood, which is conflicting with the basic principles of VNS. In fact, it would become essentially a multi-start procedure. The VNS methodology recommends to perform several iterations in order to continue the search (see Algorithm 1).

The next experiment undertakes to study how the number of iterations affects the performance of the Basic VNS in terms of quality and computing time. In line with the objective of heuristic methods of obtaining good solutions in short time, we limit the maximum execution time to 100 s per instance, stopping the execution at that time, and returning the best solution found so far. Fig. 4 shows the time evolution of the Basic VNS and Reduced VNS. In particular, we report the average deviation in 1, 10, 100, and 1000 iterations. The number close to each point marker in the profile indicates the computing time.

This experiment shows a typical behavior of a good heuristic method. Specifically, the average deviation is drastically reduced at the beginning of the search but, at a "critical point", the algorithm needs large computing times to marginally improve the solution quality. In other words, it exhibits an aggressive search pattern very well suited for a real application. According to the results presented in Fig. 4, this critical point seems to be close to 100 iterations. Therefore, we select this value for the next experiments.

According to [6] the classical performance measure for parallel algorithms, i.e., speedup described by [2], is not adequate to evaluate the performance of parallel metaheuristics since asynchronous interactions between threads generally induce significant differences in search behavior, not only for the global parallel method, but also for each search process participating to the cooperation. Therefore, the sequential and parallel methods may then be viewed as different metaheuristics, requiring a redefinition of speedup and other performance measures. This situation is further aggravated by the randomness embedded in the VNS methods considered in this paper. However, it is important to notice that a parallelization strategy should speed up the search or produce better results than the sequential method. Consequently, we compare the quality of the solutions obtained by the sequential and parallel VNS methods to evaluate the quality of the parallel design strategy. In particular, the next experiment compares the performance of BVNS and RSVNS (described in Section 4) considering different number of threads: 1 (i.e., sequential version), 2, 4, and 8. Table 4 reports the associated

[1] http://www.info.univ-angers.fr/pub/porumbel/graphs/

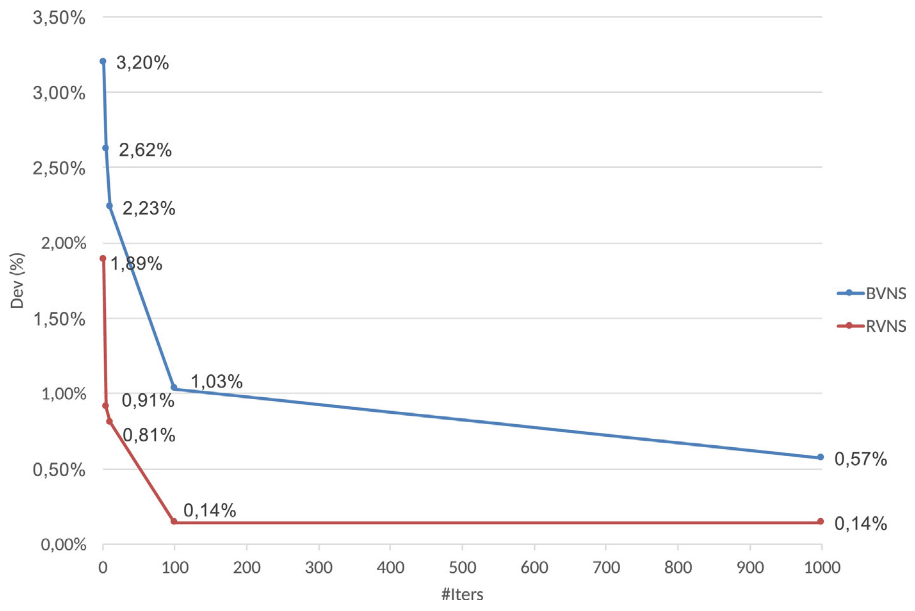

Fig. 4. Average deviation vs. number of iterations of the Basic VNS method.

Table 4
Performance of RSVNS.

	Avg.	Dev (%)	#Best	Time (s)
BVNS	615136.32	1.01%	2	45.75
RSVNS(2)	610898.42	0.42%	5	46.62
RSVNS(4)	610151.82	0.07%	7	51.36
RSVNS(8)	610061.22	0.06%	7	50.99

results, where the parameter in parenthesis indicates the number of threads.

Table 4 shows that all parallel versions outperform the sequential VNS method according to the statistics in this table. These results can be partially explained by the fact that the proposed parallel methods explore larger portions of the search space. We also observe that the best outcomes are obtained with 8 threads. It presents an average deviation of 0.06% and 7 best found solutions (out of 10), which compares favorably with the sequential version (1.01% and 2). Note that the computing time of all methods is similar (about 50 s on average). We conduct an additional experiment where the Basic BVNS is executed for 400 seconds ($p = 8$ and $Time(s) = 50$) to test whether it improves upon RSVNS when running for that computing time. The average deviation of BVNS is 0.09% in 400 s, confirming the superiority of RSVNS (0.06% in 50 s).

Once we have identified the best parameters for Basic VNS and RSVNS, the final experiment is intended to compare the performance of both methods with the 2 best procedures identified in the state of the art. In particular, a GRASP algorithm combined with Ejection Chains (GRASP+EC) and an Iterative Metaheuristic (IM). For this final comparison we consider the full set of 44 instances. Table 5 reports the associated results.

The first conclusion that we can extract by analyzing these results is that the new proposed procedures clearly outperform previous methods in both, quality and computing time. In particular, the IM method presents a large deviation (74.69%) but it is able to find the best known solution in 10 instances, while the GRASP+EC obtains lower deviation (12.80%) than IM, but only finds the best known solution in 4 instances. The RSVNS clearly outperforms all its competitors. In particular, it presents an average deviation of 0.81%, finding the best known solution in 36 instances (out of 44). The results obtained by the Basic VNS are also remarkable since it obtains considerably better results than the previous methods in shorter computing time.

Although RSVNS is a parallel method it has similar computing time (35.62 s) than the Basic VNS (34.08 s). This is mainly because the objective of RSVNS is not the reduction of the computing time, but the exploration of a larger portion of the search space. Attending to the results presented in Table 5, we conclude that exploring a large portion of the solution space results in a successful strategy.

We have performed the Friedman non-parametric statistical test with all the individual values obtained in the experiment summarized in Table 5 in order to confirm the differences among the presented algorithms. The Friedman test ranks each algorithm in all instances according to the quality of the solution obtained, giving rank 1 to the best algorithm, 2 to the second one, and so on. If the averages differ greatly, the associated p-value or significance will be small. The resulting p-value (lower than 0.001) obtained in this experiment clearly indicates that there are statistically significant differences among the 4 methods tested. Additionally, the average rank values produced by this test are 1.40 (RSVNS), 2.11 (BVNS), 3.22 (GRASP+EC), and 3.27 (IM).

We conduct an additional statistical test to perform pair-wise comparisons between our methods (RSVNS and BVNS) and the previous algorithms (GRASP + EC and IM). Table 6 presents the results

Table 5
Final comparison.

	Avg.	Dev (%)	#Best	Time (s)
BVNS	1006790.25	1.60%	11	34.08
RSVNS	1003751.30	0.81%	36	35.62
GRASP + EC	1060597.74	12.80%	4	130.55
IM	1378609.24	74.69%	10	300.72

Table 6
Results of the Wilcoxon signed rank test run over each pair of algorithms of the final experiment.

$A1$	$A2$	$A1 < A2$	$A1 > A2$	$A1 = A2$	p-value
RSVNS	BVNS	33	3	8	≤ 0.001
RSVNS	GRASP + EC	39	1	4	≤ 0.001
RSVNS	IM	34	5	5	≤ 0.001
BSVNS	GRASP + EC	39	1	4	≤ 0.001
BSVNS	IM	37	7	4	≤ 0.001

obtained with the Wilcoxon signed rank test for these five pairs. We report the algorithm being compared ($A1$ and $A2$), the number of times that $A1$ outperforms $A2$ ($A1 < A2$), the number of times that $A2$ outperforms $A1$ ($A1 > A2$), the number of ties ($A1 = A2$), and the associated p-value.

Table 6 shows that the p-value obtained in the five tests is lower than 0.001, which means that there are statistical significant differences between the compared algorithms. As it can be seen, both BVNS and RSVNS outperform the results obtained by GRASP+EC. When comparing the algorithms proposed in this paper, RSVNS and BVNS, we can confirm that the parallel version (RSVNS) emerges as the best algorithm overall, therefore becoming the state-of-the-art method for the Dynamic Memory Allocation Problem.

6. Conclusions

This paper presents two variable neighborhood search algorithms for solving the Dynamic Memory Allocations Problem (DMAP). The first one is a sequential Basic Variable Neighborhood Search (Basic VNS), while the second one is a parallel VNS called Replicated Shaking VNS (RSVNS). Both methods are composed among other search elements, by a semi-greedy method to construct high quality solutions, as well as a fast local search procedure which finds local optima in short computing time. The extensive experimental comparison performed, backed up by statistical tests, shows that the parallel version of the algorithm, RSVNS, clearly outperforms the sequential version, BVNS. We also compare the results obtained with the best previous approaches for the DMAP, which are a GRASP algorithm combined with ejection chain (GRASP+EC) and an Iterative Metaheuristic (IM). Note that these are recent methods based on metaheuristic methodologies and outperform them constitutes a challenge for a new method. Both methods, RSVNS and BVNS, are able to obtain better results than the previous state-of-the-art algorithms in considerably shorter computing time, emerging RSVNS as the new state-of-the-art algorithm for the DMAP. The results of this research are in line with other papers that propose VNS methods for other problems [22,24]: in short, the VNS methodology is able to provide better solutions than other methodologies. The interplay between intensification and diversification strategies is implemented in VNS in a way that turns out to be simple and very efficient. The future lines of research include new problem variants, such as robustness versus memory failures, memory type in terms of speeds and size, etc. The adaptation of this algorithm to new variants will show the scalability of the proposal. Finally, the quality of the obtained results indicates that the proposed algorithm can be embedded in Computer Aided Design (CAD) tools in order to increase the efficiency of electronic circuits.

Acknowledgements

This research has been partially supported by the Spanish Ministry of "Economía y Competitividad", and "Comunidad de Madrid" with grants ref. TIN2015-65460-C2 and S2013/ICE-2894, respectively.

References

[1] G. Ascia, V. Catania, A.G. Di Nuovo, M. Palesi, Patti, Performance evaluation of efficient multi-objective evolutionary algorithms for design space exploration of embedded computer systems, Appl. Soft Comput. 11 (1) (2011) 382–398, ISSN 1568-4946.
[2] R.S. Barr, B.L. Hickman, Reporting computational experiments with parallel algorithms: issues, measures, and experts opinions, ORSA J. Comput. 5 (1) (1993) 2–18.
[3] I. Baturone, A. Gersnoviez, ç. Barriga, Neuro-fuzzy techniques to optimize an FPGA embedded controller for robot navigation, Appl. Soft Comput. 21 (2014) 95–106, ISSN 1568-4946.
[4] A. Chimienti, L. Fanucci, R. Locatelli, S. Saponara, VLSI architecture for a low-power video codec system, Microelectron. J. 33 (5) (2002) 417–427.
[5] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs (Applications of GPU Computing), 1st ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.
[6] T.G. Crainic, T. Michel, Parallel strategies for meta-heuristics, in: F. Glover, G.A. Kochenberger (Eds.), Handbook of Metaheuristics, volume 57 of International Series in Operations Research & Management Science, Springer US, 2003, pp. 475–513.
[7] T.G. Crainic, M. Gendreau, P. Hansen, N. Mladenović, Cooperative parallel variable neighborhood search for the p-median, J. Heuristics 10 (3) (2004) 293–314.
[8] K.A. De Jong, An analysis of the behavior of a class of genetic adaptive systems (PhD thesis), University of Michigan, 1975.
[9] A. Duarte, J.J. Pantrigo, E.G. Pardo, J. Sánchez-Oro, Parallel variable neighbourhood search strategies for the cut width minimization problem., IMA J. Manag. Math. 27 (1) (2016) 55–73.
[10] G.R. Gao, S. Mitsuhisa, E. Ayguad, Special issue on parallel programming with OpenMP, Int. J. Parallel Program. 36 (3) (2008).
[11] F. García-López, B. Melián-Batista, J. Moreno-Pérez, J.M. Moreno-Vega, The parallel variable neighborhood search for the p-median problem, J. Heuristics 8 (3) (2002) 375–388.
[12] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1989.
[13] P. Hansen, N. Mladenović, Variable Neighborhood Search, Springer US, Boston, MA, 2005, pp. 211–238.
[14] P. Hansen, N. Mladenović, First vs. best improvement: an empirical study, Discrete Appl. Math. 154 (5) (2006) 802–817.
[15] P. Hansen, N. Mladenović, J.A. Moreno-Pérez, Variable neighborhood search: methods and applications, Ann. Oper. Res. 175 (1) (2010) 367–407.
[16] H. Hoos, T. Sützzle, Stochastic Local Search: Foundations & Applications, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004, ISBN 1558608729.
[17] N. Julien, J. Laurent, E. Senn, E. Martin, Power consumption modeling and characterization of the TI C6201, IEEE Micro 23 (5) (2003) 40–49.
[18] R.E. Korf, A new algorithm for optimal bin packing, in: Eighteenth National Conference on Artificial Intelligence, Menlo Park, CA, USA, American Association for Artificial Intelligence, 2002, pp. 731–736.
[19] N. Mladenović, P. Hansen, Variable neighborhood search, Comput. Oper. Res. 24 (1997) 1097–1100.
[20] P.K. Nath, Datta, Multi-objective hardware software partitioning of embedded systems: a case study of {JPEG} encoder, Appl. Soft Comput. 15 (2014) 30–41, ISSN 1568-4946.
[21] S. Oaks, H. Wong, Java Threads, O'Reilly Media, 2004.
[22] E.G. Pardo, N. Mladenović, J.J. Pantrigo, A. Duarte, Variable formulation search for the cut width minimization problem, Appl. Soft Comput. 13 (5) (2013) 2242–2252.
[23] J. Sánchez-Oro, N. Mladenović, A. Duarte, General variable neighborhood search for computing graph separators, Optim. Letters (2014), http://dx.doi.org/10.1007/s11590-014-0793-z.
[24] J. Sánchez-Oro, J.J. Pantrigo, A. Duarte, Combining intensification and diversification strategies in VNS. An application to the vertex separation problem, Comput. Oper. Res. 52 (B) (2014) 209–219.
[25] E. Senn, J. Laurent, N. Julien, E. Martin, Softexplorer: estimation, characterization, and optimization of the power and energy consumption at the algorithmic level, in: Enrico Macii, Vassilis Paliouras, Odysseas Koufopavlou (Eds.), Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation, volume 3254 of Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 2004, pp. 342–351, ISBN 978-3-540-23095-3.
[26] M. Sevaux, A. Rossi, M. Soto, A. Duarte, R. Martí, GRASP with ejection chains for the dynamic memory allocation in embedded systems, Soft Comput. 18 (8) (2014) 1515–1527.
[27] M. Soto, A. Rossi, M. Sevaux, Two iterative metaheuristic approaches to dynamic memory allocation for embedded systems, in: Evolutionary Computation in Combinatorial Optimization, volume 6622 of Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 2011, pp. 250–261.
[28] M. Soto, A. Rossi, M. Sevaux, A mathematical model and a metaheuristic approach for a memory allocation problem, J. Heuristics 18 (1) (2012) 149–167.
[29] S. Wuytack, F. Catthoor, L. Nachtergaele, H. De Man, Power exploration for data dominated video applications., in: Proceedings of the 1996 International Symposium on Low Power Electronics and Design, Piscataway, NJ, USA, 1996, pp. 359–364.