Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in DISCRETE MATHEMATICS

Electronic Notes in Discrete Mathematics 47 (2015) 85–92

www.elsevier.com/locate/endm

Solving dynamic memory allocation problems in embedded systems with parallel variable neighborhood search strategies

Jesús Sánchez-Oro [a,1] Marc Sevaux [b,4] André Rossi [b,5]
Rafel Martí [c,3] Abraham Duarte [a,2]

[a] *Dept. de Informática y Estadística, Universidad Rey Juan Carlos, Móstoles, Spain*

[b] *Lab-STICC, UMR6285 CNRS, Centre de Recherche, Université de Bretagne-Sud, Lorient, France*

[c] *Dept. Estadística e Investigación Operativa, Universidad de Valencia, Burjassot, Spain*

Abstract

Embedded systems have become an essential part of our lives, thanks to their evolution in the recent years, but the main drawback is their power consumption. This paper is focused on improving the memory allocation of embedded systems to reduce their power consumption. We propose a parallel variable neighborhood search algorithm for the dynamic memory allocation problem, and compare it with the state of the art. Computational results and statistical tests applied show that the proposed algorithm produces significantly better outcomes than the previous algorithm in shorter computing time.

Keywords: dynamic memory allocation problem, variable neighborhood search, parallelism, embedded systems, metaheuristics.

http://dx.doi.org/10.1016/j.endm.2014.11.012
1571-0653/© 2014 Elsevier B.V. All rights reserved.

1 Introduction

Embedded systems have become an essential part of our lives, thanks to their evolution in the recent years, but the main drawback is their power consumption. This paper is focused on improving the memory allocation of embedded systems, which is a problem that have been recently studied. A mixed linear formulation and a variable neighborhood search algorithm for the static version of the problem is proposed in [10]. In [9], the authors focused on the dynamic memory allocation problem in embedded systems. In this paper, we propose a parallel variable neighborhood search algorithm for the dynamic memory allocation problem, and compare it with the state of the art.

The memory architecture used for testing the proposal is quite similar to the one present in a TI C6201 device [6] but can be adapted to many other devices. The memory is divided into m different memory banks where each one can store up to c_j kilobytes (kB). The memory scheme also presents an external memory $(m + 1)$, whose capacity is large enough to be considered unlimited. However, the access to that external memory is slower than the access to the rest of memory banks. In particular, we need q milliseconds to access data structures placed in a memory bank, and $p \cdot q$ milliseconds to access the external memory.

Each program to be executed in the device uses a set of n data structures, each one with a different size s_i, with $1 \leq i \leq n$. The program is split in T time intervals, in such a way that for each $t \in T$ the program executes a set of operations (A_t) where each one $a \in A_t$ needs to access to one or two data structures. The processor allows simultaneous access to data structures located in different memory banks (in a specific time interval). Having defined the structure of the program and the memory architecture, we need to analyze the time needed by the processor to execute a program.

A solution for the dynamic memory allocation problem (DMAP) consists of the state of the memory in each time interval, with all the data structures placed in one of the memory banks or in the external memory. We define $b(i, t)$ as the function that finds the memory bank of interval t in which the data structure i is located. Then, the objective function value (DMAP-value) for a solution is computed as the sum of the move and access costs for all the

[1] Email: jesus.sanchezoro@urjc.es
[2] Email: abraham.duarte@urjc.es
[3] Email: rafael.marti@uv.es
[4] Email: marc.sevaux@univ-ubs.fr
[5] Email: andre.rossi@univ-ubs.fr

data structures and operations in each time interval. Specifically, the move cost for a data structure i in a time interval t is computed as follows:

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

where l denotes the milliseconds needed to move a kilobyte between memory banks and v represents the milliseconds needed to move a kilobyte between the external memory and a memory bank.

Each operation a for an interval t is a 3-tuple: the two data structures involved in the operation (a_1, a_2) and the cost of the operation, a_{cost}. So we can define the cost of accessing to the corresponding data structures as follows:

$$AccessCost(a) = \begin{cases} a_{cost} & \text{if } b(a_1,t) \neq b(a_2,t) \neq m+1 \\ 2 \cdot a_{cost} & \text{if } b(a_1,t) = b(a_2,t) \neq m+1 \\ 2 \cdot p \cdot a_{cost} & \text{if } b(a_1,t) = b(a_2,t) = m+1 \\ p \cdot a_{cost} & \text{otherwise} \end{cases}$$

where p is a penalization access factor for data structures placed in the external memory. The first case considers that both data structures are in different memory bank (but not in the external memory), so they can be accessed parallely . The second case identifies the situation where data structures are in the same memory bank and therefore they must be sequentially accessed (two access). The third and fourth cases are variations of the first and second one where at least one of the data structures are located in the external memory (which involves a penalization p).

With these definitions, the DMAP-value of a solution S for a program with n data structures and a set of operations A_t for each time interval $t \in T$ can be computed as: $\text{DMAP}(S) = \sum_{t \in T} \left(\sum_{s \in n} MoveCost(s) + \sum_{a \in A_t} AccessCost(a) \right)$

The rest of the paper is organized as follows. Section 2 presents our algorithmic approach based on the variable neighborhood search framework, including the description of the shake procedure (Section 2.1) and the proposed local search method (Section 2.2). Section 3 reports the computational experience performed to validate the proposed algorithm. Finally, Section 4 summarizes the main conclusions of our research.

2 Parallel Variable Neighborhood Search

Variable neighborhood search (VNS) is a metaheuristic for solving optimization problems based on systematic changes of neighborhood structures, without guaranteeing the solution's optimality. In recent years, a large variety of VNS strategies have been proposed. Some of the most relevant variants are reduced VNS, variable neighborhood descent (VND), basic VNS, skewed VNS, general VNS, or variable neighborhood decomposition search, among others (see [5] for a complete review of the methodology, and [2,7,8] for some example of successful research in VNS). The parallelization of VNS has been recently studied and successfully applied to some problems in the literature (see for instance the p-median problem [4] or the cutwidth minimization problem [3]). Authors in [4] propose several parallelizations for the VNS metaheuristic: Synchronous Parallel VNS, Replicated Parallel VNS, Replicated Shaking Parallel VNS and Cooperative Neighborhood VNS. In this paper we focus on the Synchronous Parallel VNS (SPVNS) variant, which is intended for the parallelization of the local search method in the sequential VNS.

There are different technologies to implement parallel algorithms, which are usually programming language dependent. All the algorithms in this paper have been implemented in Java, so we have selected Java threads as parallelization technology. Java threads allow us to take advantage of the architecture of the computer by executing the algorithms simultaneously in different processors, launching one or more threads (independent executions) in each processor.

Algorithm 1 shows the pseudo-code of the SPVNS method proposed. We use a multi-start scheme, since the initial solution is generated at random, so we can start exploring from different points of the solution space in each iteration. SPVNS receives two input arguments: the maximum neighborhood to be explored, k_{max}, and the maximum number of iterations for the multi-start scheme, *iters*. Firstly, the algorithm checks the available processors (step 2). Then, SPVNS performs the indicated iterations (steps 3-16). Each iteration is an independent sequential run of the simple SPVNS. The first step is the generation of a random initial solution (step 4) and the selection of the first neighborhood (step 5). Then, the method iterates over all neighborhoods until reaching the maximum neighborhood allowed (steps 6-14). For each iteration, the algorithm perturbs the solution (step 7), and then it proceeds with the parallel local search method. For each processor, SPVNS establishes the region that the local search will improve and starts the corresponding thread (step 10). Notice that steps 9-11 are executed in parallel, so the main

Algorithm 1 SPVNS(k_{max}, *iters*)

1: $S^\star \leftarrow \emptyset$
2: $P \leftarrow AvailableProcessors()$
3: **for** $i \in iters$ **do**
4: $S \leftarrow RandomSolution()$
5: $k \leftarrow 1$
6: **while** $k < k_{max}$ **do**
7: $S' \leftarrow Shake(S)$
8: $S'' \leftarrow S'$
9: **for** $p \in P$ **do**
10: $LS_{parallel}(S'', p)$
11: **end for**
12: $Synchronize()$
13: $k \leftarrow NeighborhoodChange(S, S'', k)$
14: **end while**
15: $Update(S'', S)$
16: **end for**
17: **return** $S^\star$

algorithm waits in step 12 until all the processors have finished. Finally, the algorithm updates the best solution found if necessary (step 15). Once all the iterations have finished, the algorithm ends, returning the best solution found (step 17).

2.1 Shake

The main aim of a shake procedure in the VNS context is the diversification of the search to escape from local optima. The shake procedure proposed in this paper traverses the memory through each time interval, generating a new solution in a neighborhood k by perturbing the original solution. We define the move for a data structure i in a time interval t as $Move(i, t, j)$, where $1 \leq j \leq m + 1$ indicates the bank where the data structure will be placed. Note that the method will only accept feasible moves. In particular, a data structure will only be moved to a memory bank with enough free space for storing it. The proposed shake method iterates over all time intervals $t \in T$ selecting k data structures at random and moving them to a random memory bank in which there is enough free space for the data structure.

2.2 Parallel local search

The parallel local search proposed in this paper is designed to make the most of the computer architecture, by balancing load among the available processors in the computer. This is conducted by following a divide and conquer strategy. In particular, if the computer has P available processors and the instance that we are solving has T time intervals, then each processor is focused on improving a total of T/P periods. Then, the first processor will improve the periods in the range $[1, T/P]$, the second one periods in range $[T/P + 1, 2 * (T/P)]$, and so on. In general, the processor $p \in \{1, \ldots, P\}$ will improve periods in the range $[(p-1) * (T/P), p * (T/P)]$. Then, the corresponding partial local search improves each time period of its designated interval as follows. The first step in each time interval consists of ordering the data structures in descending order according to its contribution to the objective function value. The contribution $\Delta_{\text{DMAP}}(S, i, t)$ of a data structure i in a time interval t in a solution S can be defined as the sum of the move cost for that data structure plus the access cost of each operation in which the data structure is involved in that time interval. More formally,

$$\Delta_{\text{DMAP}}(S, i, t) = MoveCost() + \sum_{\substack{a \in A_t \\ a_1 \vee a_2 = i}} AccessCost(a)$$

Then, each partial local search traverses the ordered data structures finding the best bank in which it can be moved. The selection of the best bank is performed by checking the change in the objective function value for moving the data structure to each bank $j \in \{1, \ldots, m+1\}$. The improvement moves produce solutions with lower objective function value, so each data structure is moved to the bank which produces the lowest DMAP-value (as described in Section 2.1). The method stops when no improvement move is found after traversing all data structures in each time interval.

3 Computational experiments

This section reports the computational experiments that we have performed for testing the effectiveness and efficiency of the proposed SPVNS algorithm compared to the best algorithm found in the state of the art. The algorithms have been developed in Java SE 7 and all the experiments were conducted on an Intel Core i7 920 CPU (2.67 GHz) and 8GB RAM. The testbed used for testing the algorithm is the same set of 44 instances used in the algorithms of the state of the art (see [9] for a detailed description of the instances).

Firstly, we have conducted a preliminar experimentation with 8 of the most representative instances to find the best k_{max} parameter value. Specifically, we have tested the following values: $0.10n$, $0.25n$, and $0.50n$, n being the number of data structures. The best k_{max} found have been $0.50n$, so it has been used for the final comparison.

	Avg.	Time (s)	Dev. (%)	#Best
GRASP+EC	1060597.74	130.55	5.58	12
SPVNS	1015325.49	32.32	3.49	36

Table 1
Comparison of GRASP+EC and SPVNS

Table 1 shows the comparison of the proposed algorithm, SPVNS, and the best previous algorithm found in the literature, GRASP+EC [9], reporting the following statistics: Avg., the average objective function value; Time (s), the average computing time measured in seconds; Dev. (%), the average percentage deviation with respect to the best solution found; and #Best, the number of times that a method matches the best known solution. The results show that the proposed algorithm produces better outcomes in all the metrics analyzed. Specifically, SPVNS obtains a lower average objective function value and a deviation 5% lower than the previous method by requiring three times less computing time. In addition, SPVNS obtains 36 out of 44 best solutions, while GRASP+EC is only able to reach 12 best solutions. These results are confirmed with the Wilcoxon signed statistical test, which presents an associated p-value lower than 0.001.

4 Conclusions

In this work we have proposed a parallel variable neighborhood search algorithm for solving the dynamic memory allocation problem (DMAP). The parallel strategy used has been the synchronous parallel VNS (SPVNS), which is focused on the parallelization of the local search method. We also include a shake procedure that perturbs a solution of the DMAP problem randomly to escape from local optima. In addition, we have proposed a new parallel local search for the DMAP that reduces the computing time needed to improve a solution by concurrently improving different regions of a solution. The experiments, supported by the Wilcoxon signed test, show that SPVNS clearly outperforms the best algorithm found in the literature, becoming the state of

the art for the DMAP.

Acknowledgment

This research has been partially supported by the Spanish Ministry of "Economía y Competitividad", grants ref. TIN2012-35632-C02.

References

[1] Chimientia, A., Fanucci, L., Locatellic, R. and Saponarac S., *VLSI architecture for a low-power video codec system*, Microelectronics Journal **33** (2002), pp. 417–427.

[2] Duarte, A., L. F. Escudero, R. Martí, N. Mladenović, J. J. Pantrigo and J. Sánchez-Oro, *Variable neighborhood search for the Vertex Separation Problem*, Computers & Operations Research, **39** (2012), pp. 3247–3255.

[3] Duarte, A., J. J. Pantrigo, E. G. Pardo and J. Sánchez-Oro, *Parallel variable neighbourhood search strategies for the cutwidth minimization problem*, to appear in IMA Journal of Management Mathematics, DOI:10.1093/imaman/dpt026, (2013).

[4] García López, F., B. Melián-Batista, J. A. Moreno-Pérez and J. M. Moreno-Vega, *The Parallel Variable Neighborhood Search for the p-Median Problem*, Journal of Heuristics **8** (2002), pp. 375–388.

[5] Hansen P., N. Mladenović and J. A. Moreno-Pérez, *Variable neighbourhood search: methods and applications*, Annals of Operations Research, **175** (2010), pp. 367–407.

[6] Julien, N., J. Laurent, E. Senn and E. Martin, *Power consumption modeling and characterization of the TI C6201*, IEEE Micro **23** (2003), pp. 40–49.

[7] Pardo, E. G., N. Mladenović, J. J. Pantrigo and A. Duarte, *Variable Formulation Search for the Cutwidth Minimization Problem*, Applied Soft Computing, **13** (2013), pp. 2242–2252.

[8] Sánchez-Oro, J., J. J. Pantrigo, A. Duarte, *Combining intensification and diversification strategies in VNS. An application to the Vertex Separation Problem*, Computers & Operations Research, **52** (2014), pp. 209–219.

[9] Sevaux, M., A. Rossi, M. Soto, A. Duarte and R. Martí, *GRASP with Ejection Chains for the dynamic memory allocation in embedded systems*, Soft Computing **18** (2014), pp. 1515–1527.

[10] Soto, M., A. Rossi and M. Sevaux, *A mathematical model and a metaheuristic approach for a memory allocation problem*, Journal of Heuristics **18** (2011), pp. 149–167.