CIRRELT

Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport

Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation

Designing Parallel Meta-Heuristic Methods

Teodor Gabriel Crainic
Tatjana Davidović
Dušan Ramljak

June 2012

CIRRELT-2012-28

Bureaux de Montréal :
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal (Québec)
Canada H3C 3J7
Téléphone : 514 343-7575
Télécopie : 514 343-7121

Bureaux de Québec :
Université Laval
2325, de la Terrasse, bureau 2642
Québec (Québec)
Canada G1V 0A6
Téléphone : 418 656-2073
Télécopie : 418 656-2624

www.cirrelt.ca

UNIVERSITÉ LAVAL UQÀM HEC MONTRÉAL ÉCOLE POLYTECHNIQUE MONTRÉAL Université de Montréal

Designing Parallel Meta-Heuristic Methods

Teodor Gabriel Crainic[1,*], Tatjana Davidović[2], Dušan Ramljak[3]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT) and Department of Management and Technology, Université du Québec à Montréal, P.O. Box 8888, Station Centre-Ville, Montréal, Canada H3C 3P8

[2] Mathematic Institute, Serbian Academy of Science and Arts, Knez Mihailova 35, 11000 Belgrade, Serbia

[3] Center for Data Analytics and Biomedical Informatics, Temple University, 1801, N. Broad Street, Philadelphia, PA 19122, USA

Abstract. Meta-heuristic methods represent very powerful tools for dealing with hard combinatorial optimization problems. However, real life instances usually cannot be treated efficiently in "reasonable" computing times. Moreover, a major issue in meta-heuristic design and calibration is to make them robust, i.e., to provide high performance solutions for a variety of problem settings. Parallel meta-heuristics aim to address both issues. The objective of this chapter is to present a state-of-the-art survey of the main parallel meta-heuristic ideas and strategies, and to discuss general design principles applicable to all meta-heuristic classes. To achieve this goal, we explain various paradigms related to parallel meta-heuristic development, where communications, synchronization and control aspects are the most relevant. We also discuss implementation issues, namely the influence of the target architecture on parallel execution of meta-heuristics, pointing out the characteristics of shared and distributed memory multiprocessor systems. All these topics are illustrated by examples from recent literature. These examples are related to the parallelization of various meta-heuristic methods, but we focus here on Variable Neighborhood Search and Bee Colony Optimization.

Keywords: Combinatorial optimization, meta-heuristics, parallelization.

Acknowledgements. This work has been partially supported by the Serbian Ministry of Science, grant nos 174010 and 174033. Partial funding for this work has also been provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canada Foundation for Innovation, and the Quebec Ministry of Education. The authors would also like to thank Mrs. Branka Mladenović and Mr. Alexey Uversky for the proofreading efforts.

Results and views expressed in this publication are the sole responsibility of the authors and do not necessarily reflect those of CIRRELT.

Les résultats et opinions contenus dans cette publication ne reflètent pas nécessairement la position du CIRRELT et n'engagent pas sa responsabilité.

---

* Corresponding author: TeodorGabriel.Crainic@cirrelt.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec
Bibliothèque et Archives Canada, 2012

© Copyright Crainic, Davidović, Ramljak and CIRRELT, 2012

1 Introduction

Meta-heuristic methods represent are widely used for solving various combinatorial optimization problems. Computing optimal solutions is intractable for many important industrial and scientific optimization problems. Therefore, meta-heuristic algorithms are used for practical applications, since, in the majority of cases, they provide high quality solutions within reasonable CPU times. However, as the problem size increases, the execution time required by a meta-heuristic to find a high quality solution may become unreasonably long. Parallelization has proven to be an efficient method for overcoming this problem.

Since meta-heuristics usually represent stochastic search processes, it is important to properly define measures to evaluate the performance of their parallelized versions, as we cannot use the standard performance measures (speedup and efficiency) for the evaluation of parallel meta-heuristics. Actually, parallelization changes the original algorithm, and consequently, the evaluation of both the *execution time* and the *quality of the final solution* is needed. Indeed, for the majority of parallelization strategies, the sequential and parallel versions of heuristic methods yield solutions that differ in value, composition, or both. Thus, an important objective when parallel heuristics are considered is to design methods that outperform their sequential counterparts in terms of solution quality and, ideally, computational efficiency. More precisely, the parallel method should not require a higher overall computation effort than the sequential method, or should justify the effort by a higher quality of the final solutions. Consequently, we select the execution time and the quality of the final solution as our performance measures.

A significant amount of work has been performed in defining, implementing, and analyzing parallelization strategies for meta-heuristics. According to the survey papers [1,2,3], several main ideas related to the parallelization of meta-heuristic methods can be recognized: starting from the low level parallelization realized by distributing neighborhoods among processors, up to the cooperative multi-thread parallel search [1,4]. Many parallelization strategies dealing with various meta-heuristic methods have been proposed in recent literature (surveyed in [1,2,3,4,5,6]). The rich collection of papers on parallel meta-heuristics [7] is devoted to both theoretical and practical aspects of this topic. Here, we briefly recall important issues, and then focus on two main classes of meta-heuristics: neighborhood-based and population-based methods. We select a representative for each class and give a survey of the existing approaches to their parallelization. As the representative for neighborhood-based meta-heuristic methods we select Variable Neighborhood Search (VNS) because of its numerous successful sequential and parallel applications. Parallelization of population-based methods is illustrated on Bee Colony Optimization (BCO), the nature-inspired meta-heuristic recently on the rise.

The rest of this chapter is organized as follows. The next section contains a brief overview of meta-heuristic methods. Review of recent literature addressing the parallelization strategies for various meta-heuristic methods is presented in Section 3. The parallelization strategies proposed in recent literature for VNS are described in Section 4, while Section 5 contains the review of the parallel BCO method. Section 6 concludes the chapter.

2 Meta-heuristics

Combinatorial optimization problems involve the selection of the best among (finitely many) feasible solutions. Each problem is defined by (i) a set of objects, with associated contributions, (ii) an objective function computing the value of a particular subset or order of objects, and (iii) the feasibility rules specifying how subsets/orderings may be built. The best solution is the one satisfying feasibility rules and yields the value of the objective function which is the highest/lowest among all possible combinations. This solution is called the *optimal solution* (*optimum*). For a more formal definition, let us assume that one desires to minimize an objective function $f(x)$, linear or not, subject to $x \in \mathcal{X} \subseteq \mathbb{R}^n$. The set $\mathcal{X}$ collects constraints on the decision variables x and defines the feasible domain. Decision variables are generally non-negative and some or all of them usually take discrete values. A globally optimal solution $x^* \in \mathcal{X}$ is the one for which holds $f(x^*) \leq f(x)$ for all $x \in \mathcal{X}$.

The main difficulty in solving combinatorial optimization problems is that the number of feasible solutions usually grows exponentially with the number of objects in the initial set. Therefore, meta-heuristics represent the only practical tools in addressing these problems in real-life dimensions. Meta-heuristics are computational methods that optimize a problem by iteratively trying to generate or improve an existing solution with respect to a given objective. Meta-heuristics are general methods, in the sense that they do not use *a priori* knowledge about the problem to be optimized. They usually apply some form of stochastic search. However, when using meta-heuristics, it is hard to guarantee the quality of the final solution: how far it is from the optimal one or if it is at all possible to reach the optimal solution by the application of meta-heuristic rules. Nonetheless, to obtain any feasible solution in the majority of the real-life applications, meta-heuristics are the only possible choice.

Meta-heuristics represent general approximate algorithms applicable to a large variety of optimization problems [8], but should be tailored for each particular optimization problem. They deal with instances of problems that are believed to be hard in general by efficiently exploring suitably limited sub-spaces of their large solution search space. Meta-heuristics serve three main purposes: solving problems faster, solving large problems, and obtaining robust algorithms. Moreover, they are very flexible, and simple for designing and implementing.

Numerous meta-heuristics have been developed in the past twenty years. Many of them are inspired by natural metaphors (e.g., evolution of species, annealing process, insect colony behavior, particle swarms, immune systems, etc.) to solve complex optimization problems. Others are based on mathematical search principles. They include the definition of some metric to measure the distance between solutions, the neighborhoods to distinguish solutions that are close to each other, and local search principles which enable efficient searches of the solution space. Examples of these methods are Tabu Search [9,10] and Variable Neighborhood Search [11,12]. Another classification of meta-heuristics is based on the number of solutions used during the search: we distinguish single solution methods and population-based ones. Meta-heuristics are also classified as constructive (if they build new

and better solutions during their execution), or based on improvement (in the case when they transform a given solution in order to obtain an improved ancestor). Constructive meta-heuristics include Greedy Randomized Adaptive Search Procedure [13,14], Ant Colony Optimization [15] and Bee Colony Optimization [16,17,18], while Genetic Algorithms [19,20], Simulated Annealing [21,22] and Variable Neighborhood Search represent methods based on the improvement of given initial solutions. All relevant details about the design and implementation of meta-heuristic methods can be found in [8,23].

2.1. Variable Neighborhood Search

Variable Neighborhood Search (VNS) meta-heuristic was proposed by Mladenović and Hansen [11]. It is a simple and effective optimization method widely used for dealing with combinatorial and global optimization problems [12]. The basic idea behind VNS is a systematic change of neighborhoods within a descent phase to find a local optimum, as well as within a perturbation phase to get out of the corresponding valley. VNS is a single-solution neighborhood-based method whose basic building block is a Local Search (LS) procedure. It uses multiple neighborhoods in order to increase the efficiency of the search. VNS is based on three simple facts [24]:

Fact 1 A local optimum w.r.t. one neighborhood structure is not necessarily an optimum for another;

Fact 2 A global optimum is a local optimum w.r.t. all possible neighborhood structures;

Fact 3 For many problems, local optima w.r.t. one or several neighborhoods are relatively close to each other.

To describe VNS, we first introduce the following notation. For a given optimization problem, e.g., min $f(x)$, the set of *solutions* S and the set of *feasible solutions* $\mathcal{X} \subseteq S$ are defined. Let $x \in \mathcal{X}$ be an arbitrary solution, we define the neighborhood of x ($\mathcal{N}(x)$) as the set of all solutions obtained from x by the application of a predefined elementary transformation. Let, $\mathcal{N}_k$, ($k = 1, \ldots, k_{max}$), be a finite set of pre-selected neighborhood structures. Then, $\mathcal{N}_k(x)$ is the set of solutions in the kth neighborhood of x, i.e., the set of solutions obtained from x by the application of k elementary transformations. Steps in the basic VNS are illustrated in Fig. 1.

Usually, the initial solution is determined by some constructive scheduling heuristic and then improved by LS before the beginning of actual VNS procedure. The role of a *shake* procedure is to prevent trapping in a local minimum. Intensification of the search is realized by the *improvement* step involving the selected LS procedure to improve the current solution. The whole VNS procedure is concentrated around the current global best solution, and therefore, the *move* step has to ensure that this solution is always updated as soon as possible.

As a meta-heuristics, VNS runs until some predefined stopping criterion is met. The possible stopping criteria can include the maximum total number of iterations, the maximum total number of iterations without improvement of the objective function, or the maximum allowed CPU time. Once the stopping criterion is met, the global best solution is reported.

```
Initialization. Find an initial solution x ∈ X; improve it with the local search
stopping criterion; STOP = 0.

Repeat
    1. Set k = 1.
    2. Repeat
        (a) Shake. Generate a random point x'
            in the k^th neighborhood of x, (x' ∈ N_k(x)).
        (b) Improve. Apply some LS method with x' as initial solution;
            denote with x'' the obtained local optimum.
        (c) Move. If this local optimum is better than the
            current incumbent,
            then move there (x=x''), and continue search within N_1 (k=1);
            otherwise move to the next neighborhood (k = k+1).
        (d) Stopping criterion. If the stopping condition is met,
            then set STOP = 1.
    until k == k_max.
until STOP == 1.
```

Fig.1. *Pseudocode of the VNS algorithm*

VNS has a unique parameter k_{max}, the maximum number of neighborhoods. Sometimes, but not necessarily, successive neighborhoods are nested. There are several variations and modifications of this basic VNS scheme, as well as many successful applications. Readers are referred to [12,25] for more details.

2.2. Bee Colony Optimization

Lučić and Teodorović [16,17,18] were among the first to use the basic principles of collective bee intelligence in solving combinatorial optimization problems. BCO is a meta-heuristic method in which a population of *artificial bees* (consisting of B individuals) searches for the optimal solution. Every artificial bee generates a solution to the problem through a sequence of construction steps. This is done over multiple iterations, until some predefined stopping criterion is met. Each step of the BCO algorithm is composed of two alternating phases: the *forward pass* and the *backward pass*, which are repeated until all solutions (one for each bee) are completed. The best among them is used to update the global best solution, and an iteration of BCO is completed. The number of forward/backward passes (NC), as well as the number of bees (B), are the parameters of the BCO algorithm, and should be given before its execution starts. The pseudocode of the BCO algorithm is given in Fig. 2.

During the forward pass, every bee adds new components to its partial solution. The number of components is calculated in such a way that a single iteration of BCO completes after NC forward/backward passes. At the end of the forward pass, the new (partial or complete) solution is generated for each bee.

```
Initialization: Read problem data, parameter values and
                stopping criterion.
Do
  (1) Assign an empty solution to each bee.
  (2) For  (i=0; i<NC; i++)  // forward pass
        (i) For (b=0; b<B; b++)
              For  (s=0; s<f(NC); s++) //count moves
                (a) Evaluate all possible moves;
                (b) Choose one move using the roulette wheel;
          // backward pass
        (ii) For (b=0; b<B; b++)
              Evaluate the (partial/complete) solution of bee b;
        (iii) For (b=0; b<B; b++)
              Loyalty decision using the roulette wheel for bee b;
        (iv) For (b=0; b<B; b++)
              If (b is uncommitted), choose a recruiter using the
                roulette wheel.
  (3) Evaluate all solutions and find the best one.
while stopping criteria is not satisfied.
```

Fig.2. *Pseudocode of the BCO algorithm*

The bees start the second phase, the so-called backward pass by sharing information about their solutions. In nature, bees perform a dancing ritual, to inform other bees about the amount of food they have found, and the proximity of the patch to the hive. In the optimization search algorithm, the values of objective functions are compared. Each bee decides, with a certain probability, whether it will stay *loyal* to its solution or not. The bees with better solutions have a higher chance of keeping and advertising them. The bees that are loyal to their partial solutions are called *recruiters*. Once a solution is abandoned, the bee becomes *uncommitted* and has to select one of the advertised solutions. This decision is made probabilistically, in such a way that better advertised solutions have higher chances to be chosen for further exploration. This way, within each backward pass, all bees are divided into two groups (R recruiters, and the remaining B-R uncommitted bees). Values for R and B-R vary from one backward pass to another.

3 Meta-Heuristics and Parallelism

The main goal of traditional parallelization is to speed up the computations needed to solve a particular problem by engaging several processors and dividing the total amount of work between them. For stochastic algorithms, meta-heuristics in particular, several goals may be achieved [8]: (i) speeding up the search (i.e., reducing the search time); (ii) improving the quality of the obtained solutions (by enabling searching through different parts of the solution space); (iii) improving the robustness (in terms of solving different optimization problems and different instances of a given problem in an effective manner; robustness may also be measured in

terms of the sensitivity of the meta-heuristic to its parameters); (iv) solving large-scale problems (i.e., solving very large instances that cannot be solved by a sequential machine). A combination of gains may also be obtained: parallel execution can enable an efficient search through different regions of the solution space, yielding an improvement of the quality of the final solution within a smaller amount of execution time.

A significant amount of work concerning the parallelization of meta-heuristics already exists. The approach can be twofold: considering theoretical aspects of parallelization, or developing practical applications of parallel meta-heuristics for different optimization problems. The survey articles [1,2,3,4,7] summarize these works and propose adequate classifications.

One of the first papers introducing classification of parallelization strategies is [3]. This classification, based on the control of the search process, resulted in two main groups of parallelization strategies: single walk and multiple walks parallelism. To refine the classification of parallelization strategies, communication aspects (synchronous or asynchronous) and search parameters (same or different initial point and/or same or different search strategies) have to be considered. The resulting classification is described in details in [4], and we briefly recall it here in order to be able to adequately classify our parallelization strategies for VNS and BCO.

The classification from [4] takes three main aspects of parallel execution into account: search control, communication control, and search differentiation. Such an approach resulted in the 3D-Taxonomy $\mathcal{X}/\mathcal{Y}/\mathcal{Z}$. Here, $\mathcal{X}$ is used to denote *search control cardinality*, which could take centralized ($1C$) or distributed (pC) values. $\mathcal{Y}$ deals with two aspects of *communication control*, synchronization and type of data to be exchanged. The four possibilities for $\mathcal{Y}$ are Rigid Synchronous (RS), Knowledge Synchronous (KS), Collegial Asynchronous (C), and Knowledge Collegial (KC). *Search differentiation* $\mathcal{Z}$ specifies the part of the search executed by each of the parallel processes. The difference is characterized by the initial point and by the search strategy. Each process can start from the same or different initial point, and it can perform the same or different search procedure. Therefore, there exist four combinations for $\mathcal{Z}$: Same initial Point-Same search Strategy (SPSS), Same initial Point-Different search Strategies (SPDS), Multiple initial Points-Same search Strategy (MPSS), Multiple initial Points-Different search Strategies (MPDS). The particular implementation of each of the described strategies may vary, depending on the given multiprocessor architecture and the characteristics of the problem at hand.

Considering implementation issues, we have to take care of the target architecture for parallel execution of meta-heuristics. When shared memory multiprocessor systems are used, the synchronization of execution steps represents the main difficulty. Namely, since all processors have access to a common (shared) memory, it is important to ensure that the relevant information is treated correctly. More precisely, it is important that no data is read before being stored to a given location, as well as that it is not overwritten before accessed by all processors. Barriers and semaphores are common control variables used for the synchronization of processors. Software resources supporting shared memory parallel implementations include openMP [26] and POSIX threads [27], with directives and library routines available for various programming languages.

In the case of distributed memory multiprocessor systems, the main problem is information exchange between various processors. Each processor needs a copy of the data relevant for its own processes to be stored in its local memory. Therefore, the physical data transfer has to be performed, causing communication delays that may significantly deteriorate the performance of parallel execution. Besides minimizing the amount and/or frequency of data exchanges, selecting of the proper multiprocessor interconnection topology can yield the minimization of communication delays. The most commonly used multiprocessor systems are illustrated in Fig. 3.

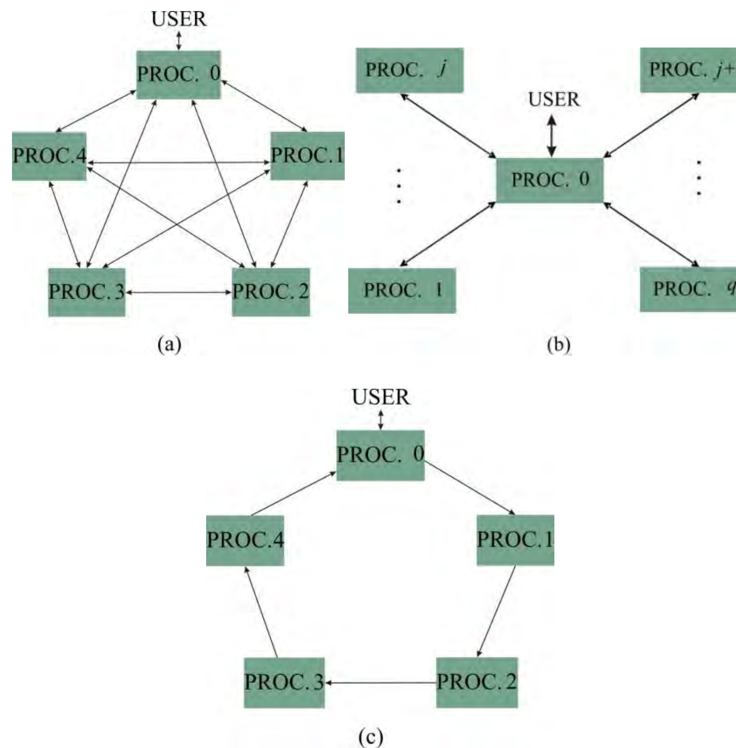

Fig.3. *(a) Complete interconnection network of 5 processors; (b) Star architecture; (c) Unidirectional processor ring*

The completely connected architecture (Fig. 3(a)) provides the minimal communication delay since each processor can directly exchange data with any other processor. On the other hand, it becomes hard for implementation when the number of processors increases. Star architecture (Fig. 3(b)) is the easiest for implementation, and the communication delay is not too large since the distance between any two processors is at most two. It is used to implement centrally coordinated parallel applications or asynchronous cooperative methods that require global memory. However, this architecture is highly fault sensitive: if the central processor crashes, the resulting system becomes unconnected. Processor rings (Fig. 3(c)) are also quite popular and easy for implementation. They provide platforms suitable for autonomous, non-centrally coordinated parallel applications. Basic advantages and disadvantages of various multiprocessor topologies are analyzed in [28] using the theory of graph spectra.

Usual way to realize distributive memory parallel applications is to use Message Passing Interface (MPI) communication protocol (library for C, FORTAN, JAVA, and some other programming languages) [29,30]. As we could notice, distributed memory implementations dominate in recent literature, probably due to the available hardware resources and/or straightforward application of the MPI routines. The implementations described in this chapter are also based on the distributed memory multiprocessor systems.

4 Parallelization of Variable Neighborhood Search

The first parallelization strategies for VNS were tested with benchmark instances of the *p*-Median Problem [31]. The authors in [31] proposed and compared three strategies. The first approach involved low-level parallelism, and may be classified as 1C/RS/SPSS: it attempted to cut computation time by parallelizing the local search phase within a sequential VNS. The second one implemented an independent search strategy that ran a sequential VNS procedure on each processor, the best solution being collected at the end. We can classify it as pC/RS/MPSS. The third method applied a synchronous cooperation mechanism through a classical master-slave approach. The master processor ran a sequential VNS. The current best solution was sent to each slave processor that had to shake it to obtain an initial solution from which a local search would be started. The solutions were passed on to the master to select the best among them and continue the algorithm. The authors tested their methods using the tsplib problem instances with 1400 customers. Not surprisingly, the last two strategies found better solutions compared to the first one. In addition, the third (1C/KS/SPSS) approach used marginally less iterations than the second one.

The *p*-Median Problem has also been used in [32] for the evaluation of the proposed parallelized VNS algorithms. Besides the independent run from [31], an asynchronous centrally coordinated parallelization strategy (classified as pC/C/MPSS) has been proposed. It was implemented on master-slave multiprocessor topology, with the master processor playing the role of central (globally accessible) memory, and the slaves performing basic VNS steps (shake and local search, SH+LS) in parallel. The proposed parallel VNS has been extensively tested on *p*-median benchmark problem instances of up to 1000 medians and 11948 customers. The results of comparison between parallel and sequential VNS indicated that the cooperative strategy yields significant gains in terms of computation time without losing solution quality. The results also showed that, for a given time limit, the cooperative parallel method was able to find better solutions than the sequential strategy. The parallel VNS allowed solving large-scale problem instances, and the quality of the obtained solutions was comparable to the quality of the best results in literature (when available).

Parallel VNS algorithms for Job Shop Scheduling problems were proposed in [33]. Four parallelization strategies were taken into account: (i) synchronized cooperative strategy proposed in [31]; (ii) asynchronous centrally coordinated method from [32]; (iii) *Non-centralized parallelism via unidirectional-ring topology*; (iv)

Non-centralized parallelism via bidirectional-ring topology. The last two strategies were applied to VNS for the first time in [33]. *Non-centralized parallelism via unidirectional-ring topology* can be classified as pC/C/MPSS. It implies that the processors are organized in a unidirectional-ring topology, in which the processors with succeeding indices are adjacent to each other and the first processor is adopted to be the adjacent to the last one. The idea in [33] was to feed a particular processor for the next generation with the outcome of the previous processor while the first processor was fed by the last processor. A particular processor was executing a single set of basic VNS steps (SH+LS), sending the obtained result to the succeeding one, and collecting the result generated by the preceding processor. The newly arrived solution became an initial point for the next execution cycle. This method provided a number of different ongoing runs with different initial solutions and diversifying with exchanging intermediate states. *Non-centralized parallelism via mesh* (*bidirectional-ring topology*) was another proposed peer-to-peer organization of processors. Each particular processor received three different solutions: one obtained by the processor itself, and the other two from the previous and the next adjacent processors. The processor selected the best of these solutions for the next execution cycle. The experimental investigation revealed that central coordination was less efficient than non-central coordination. Since the synchronous policy offers a rigorous simultaneous search within a particular region of the space, the weakest performance occurred within this approach. On the other hand, unidirectional-ring topology proposed a concurrent search in various regions of the space and outperformed the others with respect to the solution quality. The comparison between parallel and sequential VNS implementation was performed in [34].

Two cooperation schemes, based on a central memory mechanism, for parallelization of VNS were proposed in [35]. They were tested on the Multi Depot Vehicle Routing Problem with Time Windows. The used parallelization strategy was an extension of the one implemented in [32]. Each worker had to search through a certain number of neighborhoods. In the fine-grained cooperation scheme (pC/C/MPSS), the search in a single cycle did not necessarily include the whole set of neighborhoods. In the coarse-grained cooperation scheme, however, the number of iterations performed by each worker before the information exchange was significantly higher than the number of neighborhoods. This resulted in a more independent search of the individual processes. The authors proposed making the cooperative scheme adaptive by adjusting search parameters during the execution. This adjustment resulted in a pC/KC/MPSS classification of the coarse-grained cooperation scheme. For the experimental evaluation cooperative execution with up to 32 search threads was compared to the sequential procedure, and to 32 independent runs. The fine-grained cooperation scheme performed better for cases where the characteristics of the problem instances were known in advance and the appropriate parameter settings could be made. Both cooperation schemes showed high efficiency, resulting in excellent runtime scalability.

Parallel VNS for the Car Sequencing Problem was developed in [36]. Using Time Restricted LS (TRLS), in which the local search procedure was allowed to run until a predefined amount of CPU time, was proposed.

TRLS was incorporated into Randomized Variable Neighborhood Descent (VND), where “randomized” means that the order of neighborhoods is not fixed. Several iterations of TRLS in different neighborhoods were performed in parallel, the processes were synchronized, and the best solution identified and propagated to the next TRLS phase. The described parallel VNS falls into the 1C/KS/SPDS class. Computational tests were twofold: identifying the most appropriate order of neighborhoods and increasing the efficiency by parallelization. It was shown that a substantial reduction of the computation time is possible. Further, the tests revealed that no "perfect" neighborhood ordering could be identified, which implies that such a parallel self-adaptive approach is valuable and necessary for obtaining solutions of good qualities.

Several VNS instances running on different processors and exchanging the best solution after several iterations have been used in [37] for tackling the Periodic Vehicle Routing Problem with Time Windows. The number of iterations between two communications was determined with respect to the number of VNS instances, i.e., the number of available processors. The main aim of implementing this pC/KS/MPDS parallel VNS was to increase the quality of the final solution within the same amount of CPU time as required by the sequential VNS. In the second version of parallel VNS, the authors combined a heuristic search with an Integer Linear Programming solver used to improve the best solution after communication. The experimental results showed that the hybrid version yielded an improvement in the quality of the final solution for 80% of used test instances.

Multiple independent runs of various VNS algorithms were used in [38] in order to increase exploration of the search space for the Flexible Job-Shop Problem (FJSP). The main part of the proposed algorithm was comprised of internal and external loops. The internal loop was responsible for searching the solution space, whereas the external loop controlled the stopping condition of the algorithm. A number of processors in the internal loop were used in the search process to perform a single run of shake and local search procedures, independently, in parallel. These procedures used different neighborhood structures, and this strategy is therefore classified as 1C/RS/SPDS. Shaking was always applied to the current best solution. The performance of the presented algorithm was evaluated on 181 benchmark problems of FJSP. The computational results showed that the proposed algorithm was competitive to similar methods from relevant literature.

Six strategies for the parallelization of VNS applied to the multiprocessor scheduling problem with communication delays were considered in [39]. The first strategy involved independent execution of various (different) VNS algorithms. It was named IVNS, classified as pC/RS/MPDS, and used as a referent sequential execution result: parallel executions were compared to the best sequential one (the best among all independent executions) within the same amount of CPU time. In [40] the parallelization of LS procedure was considered. The best-performing parallel LS procedure was incorporated into VNS. The reported experimental evaluation of such a fine-grained parallel VNS showed that both, the quality of final solution and the running time were improved when parallel LS was executed on a modest number of processors (up to 10). The resulting parallel VNS, named PVNSPLS, was the second strategy used in [39]. It was classified as 1C/C/SPSS, as it represented the sequential

VNS speeded up by parallelizing the most computationally intensive part, namely the local search procedure. The third strategy was distributive VNS (DVNS), classified as 1C/C/MPSS, based on the main idea to explore different neighborhoods in parallel. This was realized by performing basic VNS steps (SH+LS) on different processors at the same time. This strategy is similar to the one proposed in [32], but it performs search in more systematic way. Centralized, medium-grained asynchronous cooperation of different VNS algorithms represented the fourth strategy proposed in [39]. It was implemented on star multiprocessor architecture with the central processor playing the role of global memory and the others executing various VNS algorithms. The central processor was updating the current global best solution and sending it to the others upon request. Each out the remaining processors executed basic VNS steps (SH+LS) and referred to the global memory. The resulting solution was sent to the central processor who replied with the actual global best solution. The better among these two solutions served as the new reference point for further search. This strategy was named CVNSall and classified as pC/C/SPDS, as well as the fifth one, coarse-grained centralized asynchronous cooperation named CVNSkkALL. The main difference between these two was in the amount of work performed between two communications with the global memory. In the latter case each processor performed the whole VNS iteration (until $k == k_{\max}$) before referring to the global memory. As the last strategy, the medium-grained non-centralized asynchronous cooperation on unidirectional processor ring (pC/C/MPDS) was implemented and named CVNSring. The main difference between this and the corresponding strategy proposed in [33] was that CVNSring always took the better solution as the new reference point. The experimental evaluation reported in [39] showed that CVNSring was the best performing parallel VNS. Moreover, all parallel methods (except DVNS) outperformed sequential VNS within the same amount of wall clock time.

We summarize the characteristics of the described methods in Table 1. As can be seen from this table, both synchronous and asynchronous strategies have been used. However, in the majority of papers better performance of the asynchronous parallelization has been reported. On the other hand, cooperative execution dominates centrally coordinated not only with respect to the performance but also regarding the frequency of the usage.

Table.1. *Summary of the parallelization strategies for VNS*

Reference	Strategies	Classification	Details
García-Lopez *et*	PLS in seq. VNS	1C/RS/SPSS	Low level
	IVNS	pC/RS/MPSS	independent execution
	SH+LS in parallel	1C/KS/SPSS	same *k* synchronous
Crainic *et al.* [32	IVNS	pC/RS/MPSS	independent execution
	SH+LS in parallel	pC/KS/MPSS	random *k* asynchronous
Sevkli, Aydin [3	SH+LS in parallel	1C/KS/SPSS	same *k* synchronous
	SH+LS in parallel	pC/C/MPSS	random *k* asynchronous
	SH+LS ring	pC/C/MPSS	systematic asynchronous
	SH+LS mesh	pC/C/MPSS	systematic asynchronous
Polacek *et al.* [3	SH+VND in paralle	pC/C/MPSS	fine grained cooperation
	multiple SH+VND	pC/KC/MPSS	coarse grained cooperatio
Knausz [36]	SH+RVND in parall	1C/KS/SPDS	random neighborhood su
Pirkwieser, Raid	several VNS in para	pC/KS/MPDS	synchronous VNS multi-
Yazdani *et al.* [3	SH+LS in parallel	1C/RS/SPDS	different neighborhoods
Davidović, Crai	IVNS (different VN	pC/RS/MPDS	independent execution
	PLS in seq. VNS	1C/C/SPSS	low level
	SH+LS in parallel	1C/C/MPSS	different *k*-same LS
	SH+LS in parallel	pC/C/SPDS	different *k* and different l
	VNS iteration in par	pC/C/SPDS	coarse centralized asynch
	SH+LS in parallel	pC/C/MPDS	different *k*-different LS, r centralized

5 Parallelization of Bee Colony Optimization

The BCO algorithm is created as a multi-agent system which inherently provides a good basis for parallelization on different levels. High-level parallelization assumes a coarse granulation of tasks, and can be applied to the iterations of BCO. Smaller parts of the BCO algorithm (the forward and backward passes within a single iteration) are suitable for low-level parallelization because they contain a lot of independent executions. To the best of the authors' knowledge, parallel execution of BCO was treated only in [41,42]. However, there are some papers in recent literature describing parallelization techniques for another bees-inspired meta-heuristic, the Artificial Bee Colony (ABC) method [43,44]. In this survey, we cover both ABC and BCO meta-heuristics.

The parallelization of BCO reported in [42] considered independent multiple executions of different BCO algorithms using distributed memory processors. The authors reported a significant speedup, while preserving solution quality. They also evaluated fine-grained (low-level) parallelization and showed that it was not suitable for these multiprocessor systems. On the other hand, in [45] it was shown that this strategy was hard to implement efficiently for ABC even on shared-memory multiprocessor systems. The conclusion was that the portion of work is too small, and thus the extensive use of CPU time for creating threads and their synchronization outweighs the benefits of parallel execution.

In [42], various coarse grain strategies for the parallelization of BCO using distributed memory multiprocessor systems were considered. These included two synchronous strategies, *distributed* BCO (DBCO) and *cooperative*

BCO (CBCO), and an asynchronous strategy named *general* BCO (GBCO). DBCO assumed that the total amount of computation was (equally) distributed among available processors. CBCO involved a knowledge exchange between various BCO algorithms executed on different processors. GBCO implemented asynchronous cooperation of various BCO algorithms as the most general parallelization concept. All strategies were implemented in several different ways, and compared with each other and with the sequential BCO execution. Scheduling of independent tasks to identical machines was used as test problem. Here, we present some details about these strategies, and point out the analogy with the corresponding parallelization of ABC when applicable.

The independent execution of all necessary computations on different processors represents the simplest form of coarse-grained parallelization strategies. For BCO it was implemented in three different ways [41]. All the calculations were equally distributed among the available processors by reducing the stopping criterion (DBCO), the number of bees (BBCO), or both (MBCO). In all cases, each processor independently performed a sequential variant of BCO, with a different seed or different parameter values.

The main aim of the DBCO approach was to speed up the execution of BCO by dividing the total workload among several processors, and therefore it could be classified as pC/RS/MPSS. DBCO is similar to the second approach proposed in [46] for the ABC algorithm, while for BCO it was proposed for the first time in [42]. BBCO was also considered for the first time in [42] and it is similar to the first approach proposed for ABC in [45]. It was assumed that the BCO parameters (the number of bees B and the number of forward/backward passes NC) were the same for all BCO processes executing on different processors, in order to ensure load balancing between all processors. Therefore, this approach was also classified as pC/RS/MPSS.

Combining these two approaches, it was possible to vary the values of the BCO parameters and change the stopping criterion at the same time. This approach was referred to as MBCO (Multiple BCO) and classified as pC/RS/MPDS. It had the best performance among the independent executions since it introduced more diversification into the search process.

A more sophisticated way to achieve coarse-grained parallelization in [41] was through cooperative execution of several BCO processes. At certain predefined execution points, all processes were exchanging the relevant data (usually the current best solutions). These data were used to guide further searches. The synchronous cooperative strategy, named CBCO, was classified as pC/KS/MPSS if all BCO processes had the same values of the parameters B and NC, or as pC/KS/MPDS otherwise. Similar approaches were used for the parallelization of the ABC meta-heuristic in [45,47,48].

The communication points were determined in two different ways: fixed and processor-dependent. In the first case, the best solution was exchanged 10 times during the parallel BCO execution, regardless of the number of processors engaged. The processor-dependent communication frequency was defined using the following rule: the current global best solution was exchanged each $runtime/(10*q)$ iterations where *runtime* represented the maximum allowed CPU time.

The experimental results were performed on completely connected multiprocessor architectures consisting of 2 to 20 processors. The authors showed that in both cases the quality of the final solution obtained by CBCO was either improved, or at least preserved, for a modest number of processors ($q \leq 12$), with respect to the sequential execution of the best performing BCO. At the same time, in the majority of cases, the CPU time was reduced. The CBCO variant with less frequent communications showed a slightly better performance with respect to both solution quality and minimum CPU time.

To decrease the communication and synchronization overhead during the cooperative execution of different BCO algorithms, the asynchronous execution strategy was proposed as the third approach in [41]. This strategy was named general (GBCO) and was implemented in two different ways. The first implementation involved a centrally coordinated knowledge exchange, while the second one utilized non-centralized parallelism. Each processor executed a particular sequential variant of BCO until some predefined communication condition was satisfied. It then informed others about its search status, collected the current global best, and continued its execution. As its main characteristic, this strategy did not require all of the processors to participate in the communication at the same time. Each processor would send its results, and collect the results from other processors, when it reached its own communication condition.

The first approach assumed the existence of a *central blackboard* (a global memory) [1] to which each processor had access to. The communication condition was defined as *the improvement of the current best solution or the execution of* 5 *iterations without improvement*. Each improvement of the current best solution was immediately noted on the blackboard. On the other hand, if improvement did not occur after 5 iterations, the corresponding processor referred to the blackboard for the improvements generated by other processors. If some other processor reported an improved solution, that new solution was used as the reference point for further search. When an improvement had not been announced by others, the execution continued with the previous best as the reference point. This strategy was classified as pC/C/MPSS when the BCO parameters were the same on all processors (only the seed differs), and as pC/C/MPDS otherwise. The implementation used the master-slave multiprocessor system, with the master processor playing a role of central blackboard and slaves executing the cooperating BCO algorithms.

Non-centralized asynchronous parallel BCO execution was implemented on a unidirectional ring of processors. Each processor was communicating only with its neighbors. More precisely, it was allowed to write (put new best solutions) to the blackboard of its predecessor and read from the blackboard associated to its successor. The communications were performed after a single iteration of the corresponding BCO was completed. This strategy was also classified as pC/C/MPSS or pC/C/MPDS, depending on the search parameters. Analysis of the computational results showed that non-centralized asynchronous execution outperformed all other parallel variants in the majority of the cases, with respect to both the solution quality and the running time. The strategies proposed in [41] are summarized in Table 2.

Table.2. *Summary of the parallelization strategies for BCO*

Name	Description	Classification	Details
DBCO	same BCO, diffe	pC/RS/MPSS	reduced stopping criteria
BBCO	same BCO, differ	pC/RS/MPSS	reduced number of bees
MBCO	different BCO, independent	pC/RS/MPDS	different B and NC
CBCO1	cooperative synch	pC/KS/MPDS	fixed number of communications
CBCO2	cooperative synch	pC/KS/MPDS	variable number of communications
GBCO1	cooperative async	pC/C/MPDS	centralized communications
GBCO2	cooperative async	pC/C/MPDS	non-centralized communications

6 Conclusion

Parallel meta-heuristics represent powerful tools for dealing with hard combinatorial optimization problems, especially for large size real-life instances. Therefore, a systematic approach to the design and implementation of parallel meta-heuristic methods is of great importance. The main objective of this chapter was to present a state-of-the-art survey of the ideas and strategies for parallel meta-heuristic, and to discuss general design and implementation principles applicable to all meta-heuristic classes, neighborhood- and population-based, in particular. We explained various paradigms related to the parallel meta-heuristic development. We recalled the corresponding taxonomy and used it for the classification of the described strategies. We also discussed implementation issues, namely the influence of the target architecture on parallel execution of meta-heuristics. The characteristics of shared and distributed memory multiprocessor systems were pointed out. These topics were illustrated through examples from recent literature. These examples are related to the parallelization of two meta-heuristic methods, population-based Bee Colony Optimization and neighborhood-based Variable Neighborhood Search. The common conclusion for both methods is that non-centralized asynchronous parallelization performs the best. The extensive literature and practical experience provided in this overview should help researchers in designing efficient parallel optimization algorithms.

Acknowledgments

This work has been partially supported by the Serbian Ministry of Science, Grant nos. 174010 and 174033. Partial funding for this work has also been provided by the Natural Sciences and Engineering Research Council of Canada, the Canada Foundation for Innovation, and the Quebec Ministry of Education. The authors would also like to thank Mrs. Branka Mladenović and Mr. Alexey Uversky for the proofreading efforts.

7 References

[1] Crainic, T. G.; Toulouse, M.; (2010), *Parallel Meta-heuristics*, Springer, New York Dordrecht Heidelberg London, Gendreau, M.; Potvin, J.Y. (eds), Handbook of metaheuristics, pp. 497-541.

[2] Cung, V.-D.; Martins, S.L.; Ribeiro, C. C.; Roucairol, C.; (2002), *Strategies for the parallel implementations of metaheuristics*, Kluwer Academic Publishers, Norwell, MA, Ribeiro, C. C.; Hansen, P. (eds), Essays and Surveys in Metaheuristics, pp. 263-308.

[3] Verhoeven, M. G. A.; Aarts, E. H. L.; (1995), *Parallel local search*, J. Heur., Vol. 1, pp. 43-65.

[4] Crainic, T. G.; Hail, N.; (2005), *Parallel meta-heuristics applications*, John Wiley & Sons, Hoboken, NJ, Alba, E. (ed), Parallel Metaheuristics, pages pp. 447-494.

[5] Ferreira, A.; Morvan, M.; (1997), *Models for parallel algorithm design: An introduction*, Kluwer Academic Publishers, Dordrecht/Boston/London, Migdalas, A.; Pardalos, P.; Storøy, S. (eds), Parallel Computing in Optimization, pp. 1-26.

[6] Toulouse, M.; Crainic, T. G.; Gendreau, M.; (1996), *Communication issues in designing cooperative multi thread parallel searches*, Kluwer Academic Publishers, Osman, I. H.; Kelly, J. P. (eds), Meta-heuristics 98: Theory & Applications, pp. 501-522.

[7] Alba E., editor; (2005), Wiley-Interscience, Parallel metaheuristics: a new class of algorithms.

[8] Talbi, E.-G.; (2009), John Wiley & Sons, Inc., Hoboken, New Jersey, *Metaheuristics: From Design to Implementation*.

[9] Glover, F.; Laguna, M.; (1997), Kluwer Academic Publishers, *Tabu search*.

[10] Gendreau, M.; Potvin, J-Y.; (2010), *Tabu search*, Springer, New York Dordrecht Heidelberg London, Gendreau, M.; Potvin, J-Y. (eds), Handbook of Metaheuristics, (second edition), pp. 41-59.

[11] Mladenović, N.; Hansen, P.; (1997), *Variable neighborhood search*, Comput. & OR, Vol. 24, No. 11, pp. 1097-1100.

[12] Hansen, P.; Mladenović, N.; Brimberg, J.; Moreno-Pérez, J. A.; (2010), *Variable neighbourhood search*, Springer, New York Dordrecht Heidelberg London, Gendreau, M.; Potvin, J-Y. (eds), Handbook of Metaheuristics, (second edition), pp. 61-86.

[13] Feo, T. A.; Resende, M. G. C.; (1995), *Greedy randomized adaptive search procedures,* Journal of Global Optimization, Vol. 6 pp. 109-133.

[14] Resende, M. G. C.; Ribeiro, C. C..; (2010), *Greedy Randomized Adaptive Search Procedures: Advances, Hybridizations, and Applications,* Springer, New York Dordrecht Heidelberg London, Gendreau, M.; Potvin, J-Y. (eds), Handbook of Metaheuristics, (second edition), pp. 283-319.

[15] Dorigo, M.; Stützle, T.; (2010), *Ant Colony Optimization: Overview and Recent Advances,* Springer, New York Dordrecht Heidelberg London, Gendreau, M.; Potvin, J-Y. (eds), Handbook of Metaheuristics, (second edition), pp. 227-263.

[16] Lučić, P.; Teodorović, D.; (2001), *Bee system: modeling combinatorial optimization transportation engineering problems by swarm intelligence*, Preprints of the TRISTAN IV Triennial Symposium on Transportation Analysis, Sao Miguel, Azores Islands, pp. 441-445.

[17] Lučić, P.; Teodorović, D.; (2003), *Computing with bees: attacking complex transportation engineering problems*, Int. J. Artificial Intelligence Tools, Vol. 12, pp. 375-394.

[18] Lučić, P.; Teodorović, D.; (2003), *Vehicle routing problem with uncertain demand at nodes: the bee system and fuzzy logic approach*, Physica Verlag: Berlin Heidelberg, Verdegay, J. L. (ed), Fuzzy Sets based Heuristics for Optimization, pp. 67-82.

[19] Goldberg, D. E.; (1989), Addison-Wesley Publ. Comp. Inc., *Genetic algorithms in search, optimization, and machine learning*.

[20] Reeves, C. R.; (2010), *Genetic algorithms*, Springer, New York Dordrecht Heidelberg London, Gendreau, M.; Potvin, J-Y. (eds), Handbook of Metaheuristics, (second edition), pp. 109-139.

[21] Kirkpatrick, S.; Gelatt Jr, C. D.; Vecchi, M. P.; (1983), *Optimization by simulated annealing*, Science, Vol. 220, No. 4598, pp. 671-680.

[22] Nikolaev, A. G.; Jacobson, S. H.; (2010), *Simulated annealing*, Springer, New York Dordrecht Heidelberg London, Gendreau, M.; Potvin, J-Y. (eds), Handbook of Metaheuristics, (second edition), pp. 1-39.

[23] Gendreau, M.; Potvin, J-Y. (eds.); (2010), Springer, New York Dordrecht Heidelberg London, *Handbook of Metaheuristics* (second edition).

[24] Hansen, P.; Mladenović, N.; (2003), *Variable neighbourhood search*. Kluwer Academic Publishers, Dordrecht, Glover, F.; Kochenagen, G. (eds), Handbook of Metaheuristics, pp. 145-184.

[25] Hansen, P.; Mladenović, N.; (2005), *Variable neighbourhood search*, Springer, Burke, E. K.; Kendall, G. (eds), Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, pp. 211-238.

[26] Dagum, L.; Menon, R.; (1998), *OpenMP: an industry standard api for shared-memory programming*, Computational Science & Engineering, IEEE, Vol. 5, No. 1, pp. 46-55.

[27] Butenhof, D.R.; (1997), Addison-Wesley Professional, *Programming with POSIX threads*.

[28] Cvetković, D.; Davidović, T.; (2011), *Multiprocessor Interconnection Networks*, Mathematical Institute SANU, Cvetković, D.; Gutman, I. (eds), Zbornik radova, Selected Topics on Applications of Graph Spectra, second edition 14(22), pp. 35-62.

[29] Gropp, W.; Lusk, E.; Skjellum. A.; (1994), The MIT Press, *Using MPI: Portable Parallel Programming with theMessage-Passing Interface.*

[30] Gropp, W.; Lusk, E.; (1996), University of Chicago, Argonne National Laboratory, *Users Guide for* mpich *a Portable Implementation of MPI.*

[31] García-López, F.; Melión-Batista, B.; Moreno-Pérez, J. A.; Moreno-Vega, J. M.; (2002), *The parallel variable neighborhood search for the p-median problem*, J. Heur., Vol. 8, No. 3, pp. 375-388.

[32] Crainic, T. G.; Gendreau, M.; Hansen, P.; Mladenović, N.; (2004), *Cooperative parallel variable neighborhood search for the p-median*, J. Heur., Vol. 10, No. 3, pp. 293-314.

[33] Sevkli, M.; Aydin, M. E.; (2007), *Parallel variable neighbourhood search algorithms for job shop scheduling problems*, IMA Journal of Management Mathematics, Vol. 18, No. 2, pp. 117-133.

[34] Aydin M.; Sevkli M; (2008), *Sequential and parallel variable neighborhood search algorithms for job shop scheduling*, Springer, Xhafa F.; Abraham A. (eds), Metaheuristics for Scheduling in Industrial and Manufacturing Applications, pp. 125-144.

[35] Polacek, M,; Benkner, S.; Doerner, K. F.; Hartl, R. F.; (2008), *A cooperative and adaptive variable neighborhood search for the multi depot vehicle routing problem with time windows*, Business Research, Vol. 1, No. 2, pp. 1-12.

[36] Knausz, M.; (2008), *Parallel variable neighbourhood search for the car sequencing problem*, Technical report, Fakultät für Informatik der Technischen Universität Wien.

[37] Pirkwieser, S.; Raidl, G.; (2009), *Multiple variable neighborhood search enriched with ilp techniques for the periodic vehicle routing problem with time windows*, Hybrid Metaheuristics, pp. 45-59.

[38] Yazdani, M.; Amiri, M.; Zandieh, M.; (2010), *Flexible job-shop scheduling with parallel variable neighborhood search algorithm*, Expert Systems with Applications, Vol. 37, No. 1, pp. 678-687.

[39] Davidović, T.; Crainic, T. G.; (2011), *Parallelization strategies for VNS*, (submitted for publication).

[40] Davidović, T.; Crainic, T. G.; (2011), *Parallel local search to schedule communicating tasks on identical processors*, (submitted for publication).

[41] Davidović, T.; Jakšić, T,; Ramljak, D.; Šelmić, M.; Teodorović, D.; (2012), *MPI parallelization strategies for bee colony optimization*, (submitted for publication).

[42] Davidović, T.; Ramljak, D.; Šelmić, M.; Teodorović, D.; (2011), *MPI parallelization of bee colony optimization*, Proc. 1st International Symposium & 10th Balkan Conference on Operational Research, Vol. 2, pp. 193-200, Thessaloniki, Greece.

[43] Karaboga, D.; (2005), *An idea based on honey bee swarm for numerical optimization*, Technical report, Erciyes University, Engineering Faculty Computer Engineering Department Kayseri/Turkiye.

[44] Karaboga, D.; Basturk Akay, B.; Ozturk, C.; (2007), *Artificial bee colony (ABC) optimization algorithm for training feed-forward neural networks*, LNCS: Modeling Decisions for Artificial Intelligence, Vol. 4617, pp. 318-319.

[45] Subotić, M.; Tuba, M.; Stanarević, N.; (2011), *Different approaches in parallelization of the artificial bee colony algorithm*, Int. J.Mathematical Models and Methods in Applied Sciences, Vol. 5, No. 4 pp. 755-762.

[46] Parpinelli, R. S.; Benitez, C. M. V.; Lopes, H. S. (2011), *Parallel approaches for the artificial bee colony algorithm*, Springer, Berlin, Germany, Panigrahi, B. K.; Shi, Y.; Lim, M-H. (eds), Handbook of Swarm Intelligence: Concepts, Principles and Applications, volume Series: Adaptation, Learning, and Optimization, Vol. 8, pp. 329-346.

[47] Banharnsakun, A.; Achalakul T.; Sirinaovakul B.; (2010), *Artificial bee colony algorithm on distributed environments*, Proc. Second World Congress on Nature and Biologically Inspired Computing (NaBIC'10), Fukuoka.

[48] Narasimhan, H.; (2009), *Parallel artificial bee colony (PABC) algorithm*, Proc. VIII International Conference on Computer Information Systems and Industrial Management (CISIM, 2009), World Congress on Nature and Biologically Inspired Computing (NaBIC'09), Coimbatore.