Tatjana Davidović*, Tatjana Jakšić-Krüger*, and Teodor Gabriel Crainic**

A SURVEY OF PARALLEL VARIABLE NEIGHBORHOOD SEARCH STRATEGIES

Abstract. The main focus of this study is on the survey of parallelization strategies for the Variable Neighborhood Search (VNS) metaheuristic, covering 58 publications spanning the period 2001–2025. Parallelization is a widely-used tool to enhance the efficiency of various algorithms. When stochastic algorithms such as metaheuristics are considered, the expectation is that engaging multiple processing units improves solution quality within the same or even reduced running time. However, achieving this objective is typically a demanding task. Various aspects of parallel computing must be carefully examined, including the identification of parallelizable parts, distribution granularity, load balancing, synchronization, and communication overhead. Moreover, it is important to consider the compatibility between the available hardware resources and the computational requirements. A parallel metaheuristic represents an entirely new algorithm that may significantly differ from the original sequential variant, which additionally motivates the need for a structured classification framework. We propose such a framework for the existing parallel VNS literature, organizing proposed implementations according to two complementary and non-overlapping layers. The first layer adopts the existing high-level algorithmic taxonomy (HTPM), which captures the decomposition strategy, search control, communication pattern, and solution diversity. The second layer introduces a novel implementation-level taxonomy (ITPM), proposed

here for the first time, characterizing hardware architecture, communication topology, granularity, load balancing, and parallel programming model. Neither layer alone is sufficient to interpret reported performance differences across implementations: two algorithms sharing an identical HTPM classification may perform very differently in practice, and without systematically documenting both layers such differences cannot be reliably attributed to algorithmic design choices rather than implementation-specific factors. The dual classification framework is applied consistently to all reviewed publications, accompanied by a bibliometric analysis of the field's evolution and a structured identification of open challenges specific to parallel VNS. The resulting classification provides researchers with a solid foundation to design and compare new parallel VNS implementations and to advance the parallelization of metaheuristics in general, while also laying the groundwork for a predictive understanding of parallel metaheuristic performance.

Mathematics Subject Classification (2020): Primary: 68W10; Secondary 68W20, 90C59

Keywords: metaheuristics, parallel computing, parallel algorithm design, synchronization, cooperative search, parallel metaheuristic taxonomy, GPU computing

*Mathematical Institute of the Serbian Academy of Sciences and Arts, Belgrade, Serbia
tanjad@mi.sanu.ac.rs
ORCID: 0000-0001-9561-5339
tatjana@mi.sanu.ac.rs
ORCID: 0000-0001-6766-4811
**School of Management, Université du Québec à Montréal, Department of Computer Science and Operations Research, Université de Montréal; Interuniversity Research Centre on Enterprise Networks, Logistics, and Transportation (CIRRELT), Montréal, QC, Canada
teodorgabriel.crainic@cirrelt.net
ORCID: 0000-0002-4424-0984

https://doi.org/10.64191/zr2510310104

Contents

1. Introduction 83
2. Metaheuristics 86
2.1. Metaheuristic overview 87
2.2. Variable neighborhood search 88
3. Parallel Computing 91
3.1. Standard performance evaluation measures of parallel algorithms 97
3.2. Performance measure for parallel metaheuristics 99
4. Parallelization Strategies for Metaheuristics 100
4.1. Sources of parallelism in metaheuristics 101
4.2. Taxonomy of parallel metaheuristics 102
4.3. Parallel metaheuristic implementations 109
5. Survey of Parallel VNS Approaches 112
5.1. Research Evolution 114
5.2. Shared-memory parallelization of VNS 121
5.3. Distributed memory parallelization of VNS 141
5.4. GPU-based parallelization of VNS 156
5.5. Hybrids, hierarchical and other approaches 162
6. Conclusion 163
Acknowledgments 165
References 165

1. Introduction

Combinatorial optimization problems can be found in many situations of everyday life. Determining their optimal solutions, i.e., the best among finitely many feasible solutions, often presents a very challenging task. This issue mainly results from the fact that the number of feasible solutions grows exponentially with the size of a particular problem instance, making exact methods practically unusable. Metaheuristics have become widely-used practical tools for addressing these problems in real-life cases, providing high-quality suboptimal solutions. However, even these methods may require an unreasonable amount of time, due to the large size of practical instances or the complexity of the underlying problem.

As shown in the relevant literature [6–8, 34, 42, 170], parallelization of search procedures is a promising approach to increase the efficiency of heuristic and metaheuristic methods. A significant amount of work has been conducted in designing,

implementing, and analyzing parallelization strategies for metaheuristics, ranging from the low level parallelization achieved by distributing elementary computations among processors [51, 70, 162], to the cooperative multi-thread parallel search [40, 42, 44, 53, 111, 176, 179]. Numerous survey papers have summarized developments in parallel metaheuristics [7, 8, 36, 65, 70, 128, 162, 181], yet none of them focuses specifically on VNS or provides a systematic classification of parallel VNS implementations across the full body of available literature. However, our focus is on a particular trajectory-based metaheuristic - Variable Neighborhood Search (VNS) [22, 82, 83, 89, 92, 131], and in the main part of this review, we present a survey of studies proposing parallelization of VNS. To the best of our knowledge, this survey is the only study to comprehensively address the topic of parallel VNS, covering 58 publications that appeared during the period 2001–2025, including the seminal review [133], which covered the field's founding period and predates the vast majority of the literature surveyed here. The survey [133] is included in the collection for bibliometric completeness but excluded from the classification analysis.

Parallel Simulated Annealing (SA), proposed in [112], is one of the first reports on parallelization of metaheuristic methods. This pioneering study introduced many aspects that are relevant for developing parallel metaheuristics. In addition to describing implementations of three parallel strategies for cell placement in the VLSI design layouts to be evaluated on experimental shared-memory multiprocessor system, the authors also proposed a basic taxonomy of parallelization strategies. The proposed taxonomy is based on the classification of decomposition schemes into two main classes: fine-grained and coarse-grained parallelization. Fine-grained decomposition is based on the division of individual steps into subtasks and the distribution of these subtasks on the available processing units. Coarse-grained decomposition assumes that the complete searches are performed in parallel.

Another classification of parallelization strategies is proposed in [179]. This classification identified two main groups of parallelization strategies with respect to the control of the search process: single walk and multiple walk parallelism. In the relevant literature, the authors also used the taxonomies proposed in [6] (which introduced parallelization at the algorithmic level, the iteration level, and the solution level) and [170] (distinguishing taxonomies for trajectory-based and population-based metaheuristics). To refine the existing classification schemes of parallelization strategies, one should consider communication aspects (synchronous or asynchronous) and search parameters (the same or different initial point and/or the same or different search strategies). One of the resulting classifications is described in [40]. This taxonomy is additionally extended in [35] considering also the level of parallelization (single or multiple/recursive) and decomposition aspects, i.e., decomposition type (static or dynamic) and decomposition strategies. We briefly recall this extended taxonomy later in the text and propose a complementary taxonomy that covers the implementation characteristics of parallel metaheuristic algorithms. While existing taxonomies and surveys address implementation aspects such as hardware platforms and programming environments [6, 7, 170], none provides a standardized and self-contained framework that systematically captures implementation-level characteristics alongside the algorithmic ones. As a result,

implementation characteristics are inconsistently reported across studies, making systematic cross-study comparison difficult.

VNS is a straightforward and effective metaheuristic method [84, 86, 88, 92, 131] that has been widely utilized to tackle combinatorial and global optimization problems [22, 87, 90]. The basic idea of VNS is the systematic change of neighborhoods both within a local optimization (descent/ascent) phase and in a diversification phase necessary to get out of the local optima traps.

There are numerous publications proposing parallel versions of VNS such as [10, 28, 39, 75, 106, 110, 116, 145, 163, 175, 184]. In this chapter, we identify and review 58 relevant publications addressing the parallelization of VNS. They consider various aspects, ranging from the exploitation of the available hardware architectures, over identification and granulation of parallelizable parts, toward synchronization and communication aspects. Our objective is to provide a comprehensive and technically grounded survey of the existing literature on parallel VNS by systematically organizing existing approaches with respect to their structural and implementation-level characteristics. Given the diversity of problem domains and experimental settings in the surveyed studies, the chapter focuses on classification rather than on inferring algorithmic design rationales or comparing performance trade-offs. We strongly believe that this chapter provides solid basis for future research related to parallelization of not only VNS, but metaheuristics in general. In addition to the systematic classification, we provide a bibliometric analysis of the field's evolution examining publication trends, venue distribution, and the relationship between hardware advances and research activity.

The main contributions of this chapter are: (1) the systematic application of the existing high-level (algorithmic) taxonomy (HTPM) of [35] to 57 reviewed parallel VNS publications — the first comprehensive application of HTPM to the parallel VNS literature as a whole; (2) the introduction of ITPM, an implementation-level taxonomy for parallel metaheuristics that is, to the best of our knowledge, the first standardized framework complementing HTPM at the implementation level; (3) a structured review of 57 parallel VNS implementations organized by problem domain and hardware platform, providing a unified reference for researchers and practitioners; and (4) identification of open challenges specific to parallelization of VNS (that go beyond those inherited from sequential metaheuristic research), including reproducibility, hardware-aware design, and the conditions under which parallelization yields measurable benefit. The introduction of ITPM is motivated by the observation that algorithms sharing an identical HTPM classification may exhibit substantially different performance profiles due to differences in their implementation. Without documenting both layers, performance variations across studies cannot be reliably attributed to algorithmic design choices rather than implementation-specific factors. Both taxonomies are formally introduced and discussed in Section 4.

This chapter is organized as follows. Section 2 provides a brief overview of metaheuristic methods, with particular emphasis on their stochastic nature, and on VNS as the method of focus. Section 3 presents the main characteristics and benefits of parallel computing, including models of parallel architectures, control

over module execution patterns, data-exchange mechanisms, and discusses issues specific to the parallelization of MHs. A review of the recent literature that addresses parallelization strategies for various metaheuristic methods is presented in Section 4, while the parallelization of VNS is considered in Section 5, where the existing approaches are systematically classified with respect to their structural and implementation-level characteristics. Section 6 provides concluding remarks and directions for future research.

2. Metaheuristics

Combinatorial optimization problems involve the selection of the best solution from a finite set of feasible solutions. Each problem is characterized by (i) a set of objects with associated values (contributions, utilities), (ii) an objective function that computes the value of a particular subset or ordering of these objects, and (iii) the feasibility rules specifying how subsets/orderings may be built. A solution that satisfies the feasibility constraints and attains the best (highest or lowest) objective function value among all possible combinations is referred to as the *optimal solution* (optimum). The *search space* is defined as the set of all solutions that may be considered when addressing a given optimization problem. During the search process, algorithms generate and evaluate *candidate solutions*, i.e., solutions from the search space that are considered as potential improvements, not necessarily being optimal. For a more formal definition, let us assume that one desires to minimize (linear or not) the objective function $f(x)$ over decision variables $x \in S(\subseteq \mathbb{R}^n)$, subject to $x \in X \subseteq S$. Here, S denotes the set of all *solutions*, while X represents the subset of *feasible solutions* defined by the problem constraints, i.e., the domain of f. Decision variables are generally non-negative and (in the case of combinatorial optimization) take discrete values. A (globally) *optimal solution* $x* \in X$ is the one that satisfies $f(x*) \leqslant f(x)$ for all $x \in X$.

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. As a result, metaheuristics are often the only practical tools for addressing these problems in real-life dimensions. Metaheuristics are computational techniques that optimize a problem by iteratively trying to generate or improve an existing solution relative to a specified objective. Metaheuristics are general methods, in the sense that they do not rely on prior knowledge of the problem to be optimized. They usually incorporate stochastic search strategies, making difficult to estimate the performance with respect to the solution quality and required execution time in each particular execution. In the majority of the relevant publications to compare various metaheuristics, the authors run them within the predefined stopping criterion (maximum running time, maximum number of iterations, maximum number of objective function evaluations, etc.) and report the best solution x_{best} obtained (with the corresponding value of the objective function f_{best}) and the CPU time t_{best} required to obtain this solution (the so-called best time). Sometimes, the deviations (gaps) with respect to the optimal (if available) or best-known solution are also reported.

As metaheuristics are stochastic algorithms, evaluating their performance requires statistical testing rather than point comparisons of the best found solutions. Therefore, during the development process, it is necessary to analyze the stability of the algorithm by performing controlled repetitions and statistical analysis of the obtained results [16, 96]. Non-parametric tests are commonly applied, including the Wilcoxon signed-rank test for pairwise comparisons, as well as the Friedman or Kruskal-Wallis tests for multiple algorithm comparisons. These tests assess whether observed performance differences are statistically significant across multiple independent runs. The reported performance evaluation measures typically refer to the average solution quality and running time, standard deviations, number of best-obtained solutions in all repetitions, etc. In addition, a major limitation of metaheuristics is the difficulty in guaranteeing the quality of the final solution – specifically, how far it is from the optimal one or if it is at all possible to reach the optimal solution by the application of metaheuristic rules. However, to obtain any feasible solution in the majority of real-life applications, metaheuristics are the only viable option.

2.1. Metaheuristic overview. Metaheuristics represent general approximate algorithms, applicable to a variety of optimization problems [170]. They are modern techniques defined as a universal set of rules (not involving problem characteristics) that are easily adaptable (tailored) to any problem at hand. 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. Metaheuristics serve three main purposes: finding solution faster, finding solutions for large problems, and obtaining robust algorithms. Moreover, they are very flexible and usually easy to design and implement.

Numerous metaheuristics have been developed for more than 40 years and used to find solutions to complex optimization problems. Many of them are inspired by natural metaphors (e.g., evolution of species [78, 154], annealing process [109, 136], insect colony behavior [46, 57, 66], particle swarms [74, 107], immune systems [80, 173], etc.). Others are based on mathematical concepts and 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 (TS) [76, 77], Greedy Randomized Adaptive Search Procedure (GRASP) [72, 155], Adaptive Large Neighborhood Search (ALNS) [123], and VNS [87, 88, 92, 131].

Another classification of metaheuristics is based on the number of solutions used to move from one iteration (or point in search) to the next. This distinction is particularly relevant here, as it directly affects how neighborhood-based search procedures are defined and executed. *Trajectory-based* (single-solution) metaheuristics maintain a single incumbent solution at each iteration and generate the next solution exclusively from the current one. The search therefore produces a sequence $x^{(0)}, x^{(1)}, x^{(2)}, \ldots$ where the solution $x^{(t+1)}$ is obtained by applying a transformation to $x^{(t)}$. The sequence of solutions visited in this way forms a trajectory through the solution space. Central to this class of methods is the notion of a

neighborhood. Let $x \in X$ be an arbitrary feasible solution of the optimization problem $\min_{x \in X} f(x)$. The neighborhood $\mathcal{N}(x)$ of x is the set of solutions obtained from x by applying a predefined elementary transformation. Different neighborhoods may be obtained by either changing the type of transformation or applying the same transformation multiple times. Let $\mathcal{N}_k$, $k = 1, \ldots, k_{max}$, denote a finite set of preselected neighborhood structures; then $\mathcal{N}_k(x)$ represents the set of solutions in the k^{th} neighborhood of the solution x. At each iteration, a candidate solution from the current neighborhood is selected and accepted according to a specified acceptance criterion. Depending on this criterion, the trajectory may be strictly improving or may allow non-improving moves to escape local optima. Representative trajectory-based metaheuristics include Simulated Annealing (SA) [109, 136], Tabu Search (TS) [76, 77], and Variable Neighborhood Search (VNS) [87, 88, 92, 131]. In VNS, multiple neighborhood structures are systematically explored to balance intensification and diversification. In contrast, *population-based* metaheuristics evolve a set of solutions simultaneously. New solutions are generated through interactions among population members, and the transition between iterations is determined collectively rather than by a single incumbent solution. Examples include Genetic Algorithms (GA) [78, 154], Ant Colony Optimization (ACO) [66], Bee Colony Optimization (BCO) [58, 118], and Particle Swarm Optimization (PSO) [74, 107].

Metaheuristics are also classified as constructive (if they build new solutions from the scratch during their execution), or based on improvement (in the case where they transform a given solution to obtain its descendant). The main goal of a construction/improvement is to obtain better solutions to the considered problem. Constructive metaheuristics include GRASP [72, 155], Ant Colony Optimization (ACO) [66], and Bee Colony Optimization (BCO) [58, 118–120], while Genetic Algorithms (GA) [78, 154], new variants of BCO, denoted as BCOi [55, 100, 172], Simulated Annealing (SA) [1, 109, 136], TS, and VNS represent examples of methods based on the improvement of given initial solutions. Many relevant details on the design and implementation of metaheuristic methods can be found in [170, 183].

Nowadays, metaheuristics are widely used to deal with various classes of optimization and decision making problems, even in very exotic domains such as BlockChain [122, 174]. However, although metaheuristics proved themselves very efficient in various applications, there are some "hard" problems that require more powerful tools. Even classical problems may be hard if the dimensions are too large. Therefore, the promising research direction has become the parallelization of metaheuristic methods.

2.2. Variable neighborhood search. VNS is a trajectory-based single-solution metaheuristic method originally proposed in [131]. It employs one or more neighborhood structures to explore the solution space related to a considered optimization problem [22, 84, 86–88, 90, 92]. The neighborhoods are defined in the following way. Let $x \in X$ be an arbitrary feasible solution to the optimization problem $\min f(x)$, $s \in X$. The neighborhood $\mathcal{N}(x)$ of the solution x is defined as a set of all the solutions obtained from x by applying a predefined elementary transformation. Different neighborhoods may be obtained by changing the type of transformation

or applying the same transformation multiple times. Let $\mathcal{N}_k$, $k = 1, \ldots, k_{\max}$, denote a finite set of preselected neighborhood structures, then $\mathcal{N}_k(x)$ represents the set of solutions in the k^{th} neighborhood of the solution x. The key components of VNS are the Local Search (LS) procedure used for detailed exploration of neighboring solutions and the Shaking (SH) step, which helps to avoid local optima traps. The LS phase focuses on intensifying the search around the high-quality solutions, while SH introduces the diversification (perturbation) phase, which directs the search away from the local optimum solutions. To achieve an efficient search, the basic VNS scheme performs a systematic alteration of neighborhoods during the LS phase, while in the SH phase the distance within a single neighborhood is increased. Basic VNS (BVNS) consists of three main steps: SH, LS, and Neighborhood Change (NC) as outlined in Alg. 1. During the SH step, a random perturbation of the current best solution in the given neighborhood is performed, thus providing a starting point (initial solution) for the subsequent LS phase. LS tries to improve the solution obtained in the Shaking step by exploring its neighbors within one or several neighborhoods. After LS is completed, VNS proceeds with the NC step, where it evaluates the quality of the obtained local optimum. If this local optimum is better than the current best solution, the search concentrates around it (the global best solution and the neighborhood index are updated properly). Otherwise, only the neighborhood index for the subsequent SH step is updated. These three main steps – SH, LS, and NC – repeat until a pre-specified stopping criterion is satisfied [88, 92].

Algorithm 1 Pseudo-code for BVNS method

```
procedure BVNS(Problem input data, k_max, STOP)
    x ← InitSolution()
    x_best ← x
    repeat
        k ← 1
        repeat
            x' ← RandomSolution(x_best, N_k)          ▷ Shaking
            x'' ← LS(x')                              ▷ Local Search
            if (f(x'') < f(x_best)) then              ▷ Neighborhood Change
                x_best ← x''
                k ← 1
            else
                k ← k + 1
            end if
            STOP ← StoppingCriterion()
        until (k > k_max ∨ STOP)
    until (STOP)
    return (x_best, f(x_best))
end procedure
```

The VNS search starts from a given initial solution and terminates upon the fulfillment of the pre-specified stopping criterion. Typically, the initial solution is determined at random or is built using some constructive heuristic algorithm. It is

also possible to improve it by applying LS before the actual beginning of the VNS search. As the entire VNS procedure is centered on the current global best solution x_{best}, the NC step ensures that this solution is updated as soon as possible. Therefore, VNS is known in the optimization community as the First Improvement (FI) search strategy: as soon as an improvement is made, the search immediately returns to the first neighborhood [91]. Practically, this means that the neighborhood index k is reset to 1. The BVNS algorithm has a unique parameter k_{max}, which defines the maximum number of neighborhoods. If its value depends on the problem input data (for example, the instance size), BVNS becomes the parameterless method. It is possible, though not mandatory, that successive neighborhoods are nested. In such a case, it has to be ensured that the search avoids revisiting previously explored solutions. The possible stopping criterion used to terminate VNS execution can be the maximum allowed CPU time, the maximum number of iterations, the maximum number of iterations without improvement of the current best solution, the maximum number of objective function evaluations, etc. Sometimes the authors use a combination of multiple stopping criteria [9, 105].

In addition to the main parameter of VNS, k_{max}, additional parameters are mentioned in the literature [22, 84, 86–88, 90, 92], e.g., k_{min} – which indicates that SH does not begin from $k = 1$; k_{step} – the increment for k used to skip some of the neighborhoods; δ – which controls the quality of accepted non-improving solution in a VNS variant known as Skewed VNS (SVNS), and so on.

Due to its simplicity and effectiveness [22], VNS is extensively used as an optimization tool. There are several variations and modifications of this basic VNS scheme with numerous successful applications [84, 88]. The simplest variant is Reduced VNS (RVNS), consisting only of SH and NC steps. RVNS is typically applied to challenging problems with large neighborhoods that are difficult to search efficiently or to generate a high-quality initial solution for other VNS variants. Another simple VNS variant is a deterministic VNS, known as a Variable Neighborhood Descent (VND), which excludes the SH step. In VND, the LS step is performed over a sequence of multiple pre-defined neighborhoods according to the FI rule. However, the search within each particular neighborhood can be conducted according to either FI or Best Improvement (BI) principle, as selected by an input parameter. The General VNS (GVNS) variant contains all three steps and employs VND in the LS phase, while SVNS represents a variant of BVNS that, under certain conditions, allows acceptance of non-improving solutions [92]. The variants listed here are some of the most commonly employed. For additional details, readers are referred to [22, 85, 87, 88, 92].

The applications of VNS and its variants are surveyed in [22, 88–90]. It is worth noting that the authors of this chapter have notable experience in developing VNS-based algorithms, considering Multiprocessor Scheduling Problem with Communication Delays (MSPCD) [52, 99], Vehicle Scheduling Problems (VSP) [9], Community Detection Problem [103], Vehicle Routing Problems (VRP) [125], Group Steiner Tree (GST) [126], Maximum Diversity Problem with Capacity and Budget Constraints [152], just to mention a few.

The original sequential VNS follows a strictly defined order of its fundamental steps, with each step depending on the results of the previous one. It is inherently designed as an FI search procedure, which means that the search immediately focuses on a newly found incumbent solution and resets to the first neighborhood. This sequential data dependency implies that a straightforward step-by-step parallelization of VNS is not trivial. Consequently, parallel implementations are likely to diverge from the sequential VNS basis. However, rather than being a limitation, this difference may have positive implications by enabling the generation of qualitatively new search algorithms that may significantly outperform their sequential counterparts. The resulting parallel algorithms are both problem- and hardware-specific, and a comprehensive survey of their implementations is provided in Section 5.

3. Parallel Computing

Parallel, distributed, or concurrent computing [19, 21, 32, 79, 97, 151, 153, 164] refers to the execution of multiple processes simultaneously on different processors, all working on the same problem instance with the goal of finding the best (or an acceptable) solution. Parallelism is achieved by dividing the overall computational workload into smaller tasks, which are then distributed to available processors for simultaneous execution.

The development of a parallel algorithm, whether derived from an existing sequential method or designed natively for parallel execution, typically involves two conceptually distinct phases. First, during the *algorithm design phase*, parallelism is introduced at the algorithmic level by identifying independent or cooperative components of the considered method. Second, the *parallel programming phase* implements the resulting parallel algorithm using a parallel programming model, which provides an abstraction of how such an algorithm is structured and executed on parallel computing hardware. For stochastic algorithms, these design decisions may influence not only computational efficiency but also the statistical properties of the algorithm results.

Parallel programming encompasses the development and control of software that runs multiple tasks in parallel, utilizing a variety of programming models, languages, tools, and techniques designed to exploit the capabilities of modern hardware and system architectures effectively. An important characteristic of parallel algorithm's design and implementation is the *granularity* of parallelization [14, 45, 161]. Granularity refers to the size of computational tasks that result from the decomposition of either the computational workload or the underlying problem domain, i.e., the space of solutions associated with the considered problem (search space). Granularity may also be understood as the amount of computation performed between two communication steps [44]. Parallelization is considered *fine-grained* when these tasks are relatively small in size, whereas it is termed *coarse-grained* when tasks encompass substantially large computations or broad segments of the problem domain. Thus, when the target machine's architecture supports fast communication (such as through a high-speed network or shared memory), fine-grained parallelism is well-suited for parallelization. Conversely, if communication

is slower or more costly, medium- or coarse-grained parallelism should be favored to efficiently balance computation and communication loads [44].

The decomposition may be applied to the algorithm itself, the search space, or the structure of the problem. *Functional parallelism* [140] refers to the first approach, in which the computationally intensive segments of the algorithm are partitioned into multiple tasks (processes). These tasks, operating either on the same data or on specific parts of it, are assigned to different processors and run in parallel, with the possibility of exchanging information during their execution. The second type of decomposition refers to data parallelization [68], also known as problem domain decomposition or search space decomposition. Here, the key idea is to divide the search space into subspaces and run in parallel the complete algorithm operating on these data subsets. The third way to implement parallelization is to decompose the mathematical structure of the problem into subproblems in such a way that these subproblems may be addressed by the corresponding algorithms in parallel [61]. This means that each processor creates a partial solution, while the complete solution of the considered problem is obtained by combining these partial solutions.

To effectively explore a given multiprocessor architecture, one has to address the following main issues: *load balancing*, *synchronization*, and *communication overhead* (*communication cost*). Load balancing is the process of evenly distributing the computational burden among all engaged processors, ensuring efficient utilization of processing resources between processors or cores, and preventing performance degradation due to uneven distribution of computational tasks [**?**, 144]. The communication overhead is the extra time induced by data exchanges and synchronization between processors during parallel execution. Communication overhead is a broader term that encompasses all the resources and expenses associated with communication, e.g., time and energy [124]. Communication overhead represents the machine cycles and processor resources that could otherwise be used for computation but are instead used for communication activities [79]. Synchronization is essential in shared-memory systems to coordinate access to shared resources and prevent conflicts [63]. If all computations are performed on a single processor, data transfer is not necessary and the communication cost equals zero. Parallel executions assume data transfer between processors, and this fact has as a consequence increased the total execution time: the time required to transfer data between processors is an integral part of the parallel execution time. While communication cost may sometimes be negligible, such as in distributed memory multiprocessor systems, in majority of the cases communication and synchronization overheads are significant and have to be taken into consideration.

When working with parallel computer systems, numerous challenges must be considered. These challenges are often discussed in the context of parallel computing models. These models serve as simplified, yet powerful abstractions that capture the essential characteristics of the underlying hardware and software interactions. Using parallel models as a theoretical tool allows researchers and practitioners to better understand system behavior, help predict its performance, identify bottlenecks, enable more efficient algorithm design, and better resource allocation. As

a practical tool, these models enable programmers to write parallel software that is portable, scalable, and efficient without dealing directly with low-level hardware details [19, 32].

Parallel computing models such as the PRAM, Bulk Synchronous Parallel (BSP) model, and LogP [108], provide structured abstractions that are important for performance estimation and theoretical algorithm design. Although PRAM disregards the cost of communication and synchronization, it is a relatively easy model to use [104]. However, to increase the predictability of performance, one needs to incorporate communication and synchronization costs. BSP and LogP cover these properties, whereas LogP is completely asynchronous. At the highest level, multiprocessor architectures are classified as shared-memory (Figure 1) and distributed-memory (Figure 2). Shared-memory architectures feature multiple processors accessing a single global address space, allowing implicit communication through shared variables and requiring hardware mechanisms like cache coherence to maintain consistency. Distributed-memory architectures, by contrast, consist of independent processors, each with private local memory, communicating explicitly through exchanging messages.

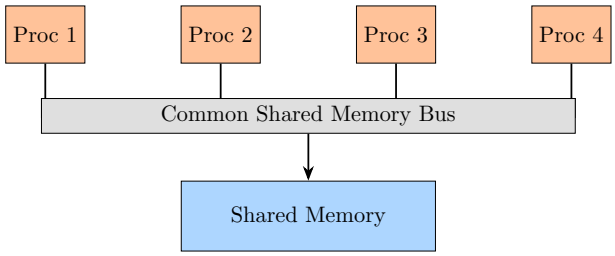


FIGURE 1. Shared-memory system

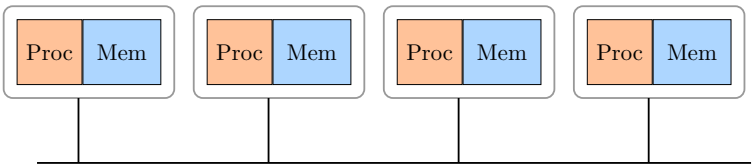


FIGURE 2. Distributed-memory system, nodes are connected via network interconnection topology

Parallel computers with shared-memory multiprocessor architecture consist of numerous processing units, known also as processing elements (PEs) or central processing units (CPUs), all connected to the common memory. It should be noted that, in the practical implementations, memory is organized into modules and each CPU within the system must be capable of accessing data by reading from or writing to any of these memory modules. If CPUs use the same mechanism to access data across all memory modules resulting in the same access speed, we declare this

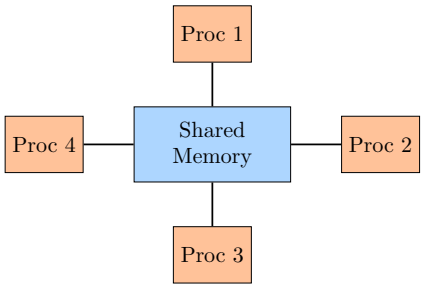


FIGURE 3. UMA Architecture: Uniform Memory Access

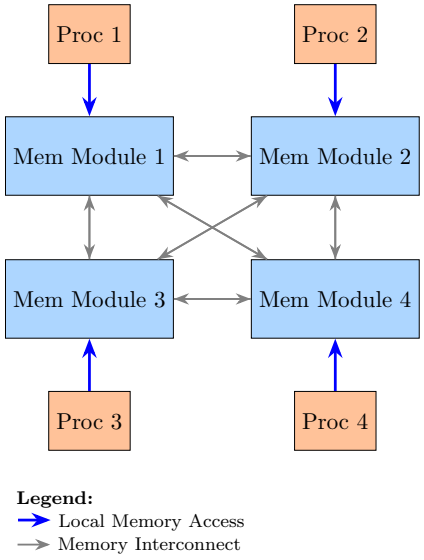


FIGURE 4. NUMA: Non-Uniform Memory Access

as a uniform memory access (UMA) model illustrated in Figure 3. However, in the majority of shared-memory topologies, is not always possible to ensure UMA. Typically, a small group of CPUs (or cores) are integrated on a chip together with their common memory module, they can access uniformly. Memory located on other chips is also accessible to these CPUs, with a slower access time resulting in a non-uniform memory access (NUMA) architecture, as illustrated in Figure 4.

In distributed-memory architectures, each CPU can access only its own local memory. To obtain data stored in the memory modules of other CPUs, it must communicate with those CPUs, typically by exchanging messages. CPU's may be either completely connected (each of them can directly connect to all others) or arranged in a network topology, where some CPUs are only indirectly connected thus the communication between non-adjacent CPUs involves all intermediate processing

units. Several examples of typical interconnection topologies are illustrated in Figure 5. Completely connected architectures ensure fast and reliable communication between CPUs, however, it may be unpractical or even impossible for realization between large number of CPUs. On the other hand, easier to implement, indirect connections can result in the increasing of the time required for communication between CPUs. In addition, intermediate CPU sometimes have to control the communication processes, which slows down their communication bandwidth.

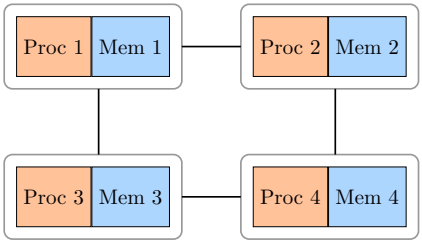


FIGURE 5. Distributed-memory System in a 2x2 mesh (grid) network

Building on the shared-memory and distributed-memory multiprocessor architectures, recent advances in parallel hardware have moved toward integrating multiple processing units directly on a single chip. In relevant literature, multi-core processors are described as integrating a small to moderate number of relatively powerful general-purpose cores (typically ranging from a few to a few dozen) on a single chip. They are optimized for both sequential and parallel execution, which makes them well-suited for problems that contain a high degree of sequential dependency and complex control flow [20]. In contrast, many-core systems integrate hundreds to thousands of simpler processing units on a chip. These are designed for high-throughput, fine-grained parallelism and are intended to perform a large number of elementary operations simultaneously over different data [12]. A prominent and now widely used example of many-core architectures is Graphics Processing Units (GPUs). GPUs have become a standard component of modern computing platforms ranging from workstations to supercomputers [135]. Figure 6 illustrates a simplified architecture diagram of a GPU. The GPU block is shown as a large rectangular area that encompasses multiple multiprocessors. Each multiprocessor contains several smaller units, denoted by $P_1, P_2, ..., P_n$, representing processing cores or processors within each multiprocessor. Device memory corresponds to the dedicated memory of GPU used to store data and instructions accessed by the GPU multiprocessors. The arrow shows communication between the GPU multiprocessors and device memory. One or more GPU blocks are usually connected to the main CPU that acts as a user-interface and server for controlling the operation of GPU processors.

According to [12], key challenges spanning hardware, system software, programming models, and algorithm design (in no particular order) include: (1) power consumption limitations, (2) scaling of parallel systems to many-core chips while maintaining programmability and performance, (3) designing efficient processors,

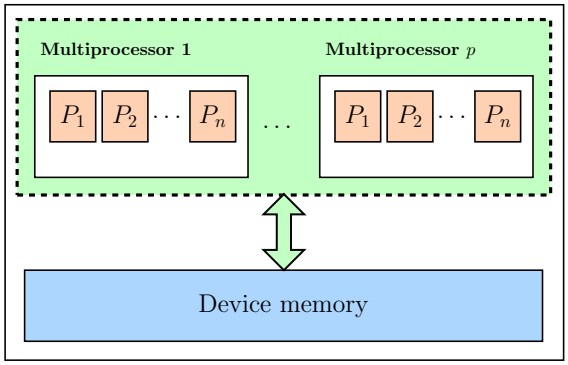


FIGURE 6. GPU-based multiprocessor system

caches, memory hierarchies, and interconnects that support high degrees of parallelism, (4) managing communication latency and coordination among processors, (5) developing programming models that are productive, portable, and scalable, (6) redesigning operating systems and runtime systems to handle massive parallelism efficiently, (7) improving energy efficiency, (8) maintaining developer productivity via accurate performance counters, debugging, and profiling tools, and (9) identifying and adapting natural parallelism in real-world application. Several of these challenges are directly related to algorithm design, particularly those concerning communication and coordination, energy-aware execution, and the identification and exploitation of natural parallelism. Among these challenges, the identification and exploitation of natural parallelism in real-world applications is primarily an algorithm-design issue. In the context of metaheuristic methods, this corresponds to identifying suitable sources of parallelism, such as independent or cooperative search processes, problem or search space decomposition, and algorithm-level parallelization strategies. These aspects are discussed in detail in Section 4.1. Parallel programming issues arise during the development of parallel algorithms, particularly when transitioning from an existing sequential implementation [127]. They can be grouped into four main categories: (1) the developer's expertise and cognitive abilities, (2) the availability and effectiveness of development tools and methodologies, (3) the particular characteristics of the software system being parallelized, and (4) various environmental factors that may impact the development process.

From this overview it is evident that measuring the performance of the parallel algorithms is not straightforward, even for classical (deterministic) computational algorithms. This complexity additionally increases when evaluating metaheuristics, whose performance assessment is challenging due to the inherent nondeterminism in their execution process. This is because the corresponding sequential version is stochastic and involves a random factor guiding the search from any given execution point, causing the variability in outcomes even under identical starting conditions. In the remainder of this section, we discuss the performance metrics commonly employed to evaluate both deterministic and stochastic parallel algorithms.

3.1. Standard performance evaluation measures of parallel algorithms. To compare different methods of decomposition, some parameters are required to measure the performance of codes that are executed on parallel machines. The primary metric for assessing the performance of a parallel algorithm has been *speedup* [15, 21, 151]. To account for the number of processors utilized, *efficiency* has been introduced as an additional performance measure.

The *parallel speedup* achieved by a parallel algorithm running on p processors is defined as the ratio of the execution time of the parallel algorithm on a single processor and the execution time of the parallel algorithm on p processors. The parallel speedup $S(n,p)$ can be expressed as,

$$S(n,p) = \frac{T(n,1)}{T(n,p)}$$

where n denotes the size of the considered problem, $T(n,1)$ and $T(n,p)$ denote the execution times of the algorithms on one and p processors respectively.

The *parallel efficiency* is equal to the speed-up divided by the number of processors p. The parallel efficiency $E(n,p)$ can be expressed as

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T(n,1)}{pT(n,p)}$$

It is important to note that parallel execution includes communication (data exchange) and synchronization parts that may consume a significant portion of running time and deteriorate the performance of the parallel algorithm.

However, the parallel speedup and efficiency equations do not provide any information about the quality of the parallel algorithm. Rather, they measure how well an algorithm has been parallelized. Therefore, we need some data that indicates the *numerical efficiency* of the parallel algorithm. This should be defined with respect to $T_{best}(n)$, the time required by a single processor within the parallel computer to execute the fastest known sequential algorithm. Combination with the definitions of parallel speedup (or efficiency) leads to the notion of *total speedup* and *total efficiency*, defined by

$$\bar{S}(n,p) = \frac{T_{best}(n)}{T(n,p)}$$

$$\bar{E}(n,p) = \frac{\bar{S}(n,p)}{p} = \frac{T_{best}(n)}{pT(n,p)}$$

The total speedup is also referred to as *absolute speedup*, while the parallel speedup is known as *relative speedup*.

Although more realistic, the latter definitions have limited usefulness in practical considerations. First of all, it is often very difficult to determine which algorithm is the best sequential one; this may depend on the size of problem n, the particular hardware used, the implementation issues, etc. Moreover, the notion of the 'best' algorithm may change over time, as better algorithms become available. In addition, a good implementation of that algorithm is not always available. In practice,

one can define the total speedup by using the execution time of a good sequential algorithm instead of $T_{best}(n)$.

These measures were used for the evaluation of parallel algorithms performance for a very long time. In the recent literature, some limitations of these measures have been identified [15, 18, 61]. The first issue arises from the different types of sequential and parallel computers. The second drawback is that these measures do not account for the scalability of the problem or the specific characteristics of the parallel processor architecture. Several alternative definitions of speedup, such as fixed-size, fixed-time, and memory-bounded, are known from the literature [18].

We can identify three possible cases for the speedup value S_p:
* $S_p < p$, defining sublinear speedup
* $S_p \sim p$, for linear speedup
* $S_p > p$, in the case of superlinear speedup.

It is reasonable to expect that an algorithm running on a multiprocessor system containing p processor can execute p times faster than on a single processor machine. This property is referred to as *linear speedup*. However, there are some overheads that can cause a significant deviation from the linear speedup, yielding $S(n,p) \ll p$ or $S(n,p) \gg p$. These overheads are enumerated as follows [61]:

(1) *Communication Overhead* - can significantly affect (decrease) the performance of parallel algorithms. Communication costs may cause processor idle times while waiting for the data necessary to perform computations. However, sometimes co-processors are added for communication, which can reduce the waiting time on the one hand and introduce additional control operations on the other. It is evident that this overhead must be carefully considered.
(2) *Load imbalance* - represents the difference between computational and communication burden among processors. When all processors are equally loaded and complete their execution almost at the same time, the multiprocessor system is ideally balanced. However, even in this case, communication overhead may take a large portion of the processor load, causing idle time on some other processors.
(3) *Synchronization overhead* - accounts for the breaks in the algorithm's execution due to data exchange or memory updates. To reduce this overhead, load balancing is of crucial importance. On the other hand, in some cases, asynchronous executions are not only allowed but also desirable. Although they allow for non-controlled communications, which are hard to define and follow during the execution, in some cases, it is possible to achieve superlinear speedup with these asynchronous executions.

The deviations from the desired linear speedup can be affected also by *Scalability* and *Amdahl's law*. Scalability means that the speedup remains close to p even in the cases where $p \to \infty$. Usually, it is hard to achieve scalability of parallel algorithms, and therefore, the careful selection of the adequate number of processors used is very important. Amdahl's law refers to the inherently sequential parts of the algorithm. In the majority of traditional sequential algorithms, there are

usually some computations that cannot be performed simultaneously due to data dependencies. Let $T(n,1) = \tau$. If we assume that f is a fraction of sequential code in the algorithm, while the remainder of the code is ideally parallelizable, then

$$T(n,p) = f\tau + \frac{(1-f)\tau}{p}.$$

Consequently,

$$S(n,p) = \frac{1}{f + (1-f)/p}.$$

In the case where $p \to \infty$, $S(n,p) \to 1/f$, indicating that the increase in speedup is limited by the constant depending on the f value. Obviously, Amdahl's law is a main limitation in achieving linear and superlinear speedup of parallel algorithms.

3.2. Performance measure for parallel metaheuristics. Having in mind all the above mentioned characteristics of metaheuristic algorithms, it is not easy to define the performance measure of their parallel counterparts. The main problems that we need to address are the following:

- Stochastic behavior of parallel execution
- Asymptotic comparison when CPU time is increasing
- Stopping criterion
- Comparison criteria

As we already explained, the results of sequential and parallel metaheuristic searches do not have to be the same for the same set of parameters. The solutions may differ in the objective function value and/or structure (composition). Therefore, the performance of parallel metaheuristics with respect to their sequential counterparts should be evaluated in terms of both solution quality and computational efficiency. More precisely, an efficient well-performing parallel implementation of metaheuristic method should not require larger computation effort than the sequential method to find solution of the same quality or should justify the effort by providing solution of higher quality.

First, it is necessary to define $T_{MH}(n,1)$ and $T_{MH}(n,p)$, which denote the execution times required to fulfill the stopping criterion by sequential and parallel (running on p processing units) metaheuristic algorithms, respectively. Next, the speedup factor should be calculated with respect to these times, i.e.,

$$S_{MH}(n,p) = \frac{T_{MH}(n,1)}{T_{MH}(n,p)}.$$

In the majority of the relevant publications, the authors compare the performance of the parallel algorithm with its sequential counterparts and report $x_{par}(p)$, the best solution obtained by the parallel algorithm running on p processing units (CPUs, cores, GPUs, etc.) that is to be evaluated with respect to x_{seq}, the best solution generated by the sequential algorithm. The stopping criterion may be the same (if we seek to improve the solution quality within the same amount of wall-clock time) or reduced proportionally to p. The best running times ($T^{best}_{MH}(n,1)$ and $T^{best}_{MH}(n,p)$) can also be considered to analyze the influence of stochastic components

within both the algorithm and the nondeterministic nature of parallel execution. Therefore, statistical analysis described in Section 2 may also apply.

An additional possibility to evaluate the performance of parallel metaheuristics is to consider *search robustness*, indicating that the solution quality does not depend heavily on the parameter settings [6, 41]. Another possibility of determining the performance measure of parallel execution is to introduce stochastic analysis as suggested by [171].

Here, we introduce a different approach. We define a new measure called the *usefulness* $\mathcal{U}_p$ of parallel execution as the product of two components: (1) a speedup factor, and (2) the ratio between the objective function value obtained by the sequential execution (f_{seq}) and the corresponding value resulting from the parallel search using p processors ($f_{par}(p)$) . More precisely,

$$\mathcal{U}_p = S_{MH}(n,p)\,\frac{f_{seq}}{f_{par}(p)}.$$

This criterion enables the speedup factor to be scaled based on adjustments in the quality of the solution. Specifically, if the solution is improved, the speedup factor is multiplied by a coefficient greater than one (which means that parallel execution is more efficient, i.e., more useful). On the other hand, if the quality of the solution is degraded, the usefulness of parallel execution decreases. If the same solution is obtained by sequential and parallel execution, the usefulness is equivalent to the speedup factor. This composite performance measure can be additionally modified by including additional factors such as resource consumption (scaling or not the stopping criterion), stability analysis (by examining the statistical characteristics of repetitive runs), etc.

4. Parallelization Strategies for Metaheuristics

As the relevant literature [6, 41, 42, 128, 176] shows, parallel metaheuristics represent a new class of optimization algorithms that aim to be more efficient (faster), capable of addressing large size real-life instances, and more robust compared to their sequential counterparts. Various parallelization strategies exist, however, only a limited number provide efficient parallel algorithms. Three levels of parallelization can be distinguished [128, 170]: high or algorithmic-level, medium or iteration-level, and low or solution-level. The highest level assumes execution of multiple metaheuristic algorithms in parallel with the main goal to improve the effectiveness of the exploration and robustness of the search. The medium level involves the generation and evaluation of a number of solutions (the whole population or neighborhood) in parallel. The parallelization of a single solution generation and evaluation is considered low level.

Based on the relevant literature, it can be concluded that most parallelization strategies significantly modify the original sequential algorithm. This observation has inspired [6], however, the database of results has since become even more extensive. In the remainder of this section, we present an overview of concepts related to the parallelization of metaheuristics. This overview identifies the main sources of parallelism, provides a detailed description and refinement of the taxonomy

proposed in [35] to classify parallelization strategies, and offers a brief overview of existing parallel metaheuristic implementations.

4.1. Sources of parallelism in metaheuristics. As metaheuristic methods contain several main steps, there are various ways to parallelize their execution [35]. The most natural source of parallelism is to execute concurrently the deepest loop, for example, to transform and evaluate neighbors in trajectory-based methods or to modify different individuals and compute their fitness values in (sub)population of solutions. This is known as an *algorithm-based decomposition* or *functional parallelism* and it requires minor changes to the original algorithm. It focuses on the computationally intensive parts of the algorithm, with the primary goal of speeding up their execution. The aim is to decompose the required computations into a number of tasks that would be executed on different processing units. The task represents the amount of algorithmic work or the size of the search space handled by a single processing unit. As the same computational operations must be performed in both sequential and parallel versions, the main algorithmic modification involves reorganizing the order of execution to enable concurrent processing. Parallelization of larger parts (modules, steps) of s metaheuristic algorithm (these that include previously mentioned deepest loops) is not straightforward because it must account for data dependencies that guide their execution. More precisely, the execution of later steps heavily depends on the intermediate results produced by previously completed steps. For that reason, parallel execution may involve additional communication and/or synchronization tasks, which introduce significant delays in the execution of the entire algorithm. Therefore, functional parallelism has been explored extensively, either as a low-level parallelization strategy or as a main component in hierarchical parallelization approaches. It is especially investigated in systems based on GPUs, to achieve significant reductions in computing times.

The second potential low-level source of parallelism involves decomposing the problem in such a way that the resulting parts could be addressed simultaneously. Decomposition means that the solution of the considered problem is divided into parts and each part needs to be processed on different processing units. In this way, only a part of the solution is allowed to be modified or a part of a solution (partial solution) is created by the corresponding processing unit. The involved processing units may use same or different solution methodologies. This type of parallelization may yield significant changes in the original sequential algorithm and requires several problems to be resolved: how to perform decomposition, how to compose a complete solution from the provided parts, how to control an overall search process, how to efficiently utilize available resources, and so forth. Various classes of parallel metaheuristic algorithms are obtained depending on the resolution of these problems, and they are categorized according to the taxonomy described in the next subsection.

Search space decomposition (SSD) represents another source of parallelism of medium- to high-level type. It enables executing in parallel large parts of the algorithm with the possibility to repeat some computations on several processing units or even to execute some parts of the code that would never be executed

by the sequential algorithm. However, this may not have negative consequences; in contrast, it may enable a more detailed exploration of the search space and finding a solution of better quality, which could not be generated otherwise. It is important to note that this type of parallelism introduces large changes to the original sequential algorithm if applied at the medium level. The order of generation and evaluation of solutions may change, which can modify the algorithm flow. At the high-level, we have multi-search (independent or cooperative) of a number of (same or different) sequential algorithms, with differences being expressed by the various sets of parameter selections. Even if the same set of parameters is selected for all algorithms the final result may not be the same, due to the stochastic nature of the metaheuristics and/or different initial solutions used for execution on different processing units. Two classes of SSD strategies are identified. Independent multi-search does not involve information exchange between metaheuristic algorithms, except the comparison of final solutions at the end of execution to determine the best among them. On the other hand, cooperative search involves communication and information-exchange between metaheuristic algorithms executed in parallel on different processing units. Various communication and cooperation strategies are available in the relevant literature, ranging from the simple exchange of the current best solutions to inclusion of learning mechanisms that, based on exchanged data, guide the search processes on all processing units.

4.2. Taxonomy of parallel metaheuristics. To implement a parallel variant of metaheuristics, it is necessary to decompose the total amount of computations required to find the high-quality solution to the considered optimization problem into smaller (and potentially simpler) computing tasks. These tasks are to be distributed among the available processing units and hopefully executed concurrently.

There are numerous possible ways to generate parallel tasks and organize (schedule) their execution. To define, describe, analyze, and classify the existing parallelization strategies for metaheuristics, several taxonomies are proposed [6, 33, 35, 40, 44, 112, 170, 179]. In the remainder of this subsection, we describe the high-level (algorithmic) taxonomy proposed in [35] and introduce a complementary implementation-level taxonomy that provides its refinement related to implementation issues.

4.2.1. High-level Taxonomy of Parallel Metaheuristics. The extended taxonomy proposed in [35] is a comprehensive framework for classifying and analyzing parallel metaheuristics based on their decomposition complexity, the source of parallelism, and the design of control-communication-diversity. This classification helps to analyze parallel metaheuristics systematically, comparing different designs, understanding coordination overheads, and identifying the nature of cooperation between parallel searches. It is broad enough to cover most metaheuristic classes and specific enough to guide detailed analysis, which is why in this survey we call it *High-Level Taxonomy for Parallel Metaheuristics* (HTPM). It should be noted that HTPM is not a new taxonomy — it is adopted directly from [35] and applied here to the parallel VNS literature. It organizes parallel metaheuristics along three main dimensions: 1) Number of levels (L), 2) Decomposition strategy (Dec), 3)

Search Strategy, which combines three components: Control (C), Communication and Learning (C&L) and Diversity (Div) (see Table 1).

TABLE 1. High-level taxonomy for parallel metaheuristics [35].

Label	Values	Description
Number of Levels (L)	1L	Single level decomposition;
	nL	Multiple (recursive) levels of decomposition
Decomposition (Dec)	A	Algorithm-level decomposition (functional parallelism)
	SE	Explicit Search Space decomposition
	SI	Implicit Search Space decomposition (Independent multi-search)
	SC	Implicit Search Space decomposition (Cooperative multi-search)
	MM	Mathematical Programming decomposition
	MA	Mathematical Attributes decomposition
Control (C)	1C	Centralized control (single process controller/master)
	pC	Decentralized control (multiple cooperating processes)
Communication & Learning (C&L)	S	Synchronous communication (Rigid synchronization, no learning)
	A	Asynchronous communication (Collegial, no learning)
	K	Knowledge-based communication (includes learning and information creation)
Diversity (Div)	SPSS	Same Point/Population, Same algorithm
	SPDS	Same Point/Population, Different algorithms
	MPSS	Multiple Points/Populations, Same algorithm
	MPDS	Multiple Points/Populations, Different algorithms

The Number of Levels indicates whether the parallel decomposition is applied at a single level (1L) or recursively across multiple levels (nL). For example, in a 1L strategy, the problem is decomposed once into parallel tasks, while in an nL strategy, tasks themselves may be hierarchically decomposed into subtasks. The Decomposition reflects what is being partitioned, i.e., identifies the structural source of parallelism, in order to create parallel tasks. In particular, what can be partitioned is: (a) Algorithm - by parallelizing internal computations or algorithm components (neighborhood evaluation, fitness computation, GPU kernels), (b) Search Space - by partitioning the solution space, (c) Mathematical Structure - by exploiting mathematical properties of the problem for decomposition. In this

case, algorithm-based decomposition always yields a single complete search trajectory. On the other hand, explicit search-space decomposition yields only partial trajectories, each restricted to a predefined subspace of the original search space. SI consists of multiple independent, complete search trajectories, whereas SC consists of multiple complete but dynamically coupled trajectories whose evolution depends on the cooperation. The Search Strategy combines the three components:

(1) Control (addresses the question of number of processes involved in the controlling of the search),
(2) Communication and Learning (specifies how information is exchanged between parallel tasks), and
(3) Diversity (defines how many different initial solutions or populations are used and whether the same or different search strategies are employed).

Each component of the Search Strategy label takes various values. In order to show all possible values, we use Table 1. In practice, the abbreviated 5-field notation (L/Dec/C/C&L/Div) is utilized to describe a category of parallel metaheuristic algorithms. Unlike sequential programs, parallel algorithms are highly dependent on the target computer architecture [44]. Nevertheless, HTPM deliberately excludes low-level details about hardware and software implementation and hence, does not explicitly address practical implementation concerns such as the type of CPU or hardware architecture being used (e.g., multi-core CPUs, many-core GPUs, clusters, cloud platforms), memory model (shared memory, distributed memory), details of parallel programming models (performance predictability, portability, management of concurrency and communication [32]), etc.

4.2.2. Implementation Taxonomy of Parallel Metaheuristics. To complement HTPM, it is necessary to introduce additional taxonomies or conceptual frameworks that categorize the above-mentioned concerns and help bridge the gap between algorithm theory and real-world software deployed in parallel computing environments. The taxonomies found in the literature that address this question are generally not standardized [6, 44, 170]. Additionally, these taxonomies either overlap or are not fully complementary to the HTPM. Moreover, two algorithms sharing an identical HTPM classification may behave very differently in practice, depending on how they are implemented. Without systematically documenting both layers, it is difficult to determine whether observed performance differences across studies arise from algorithmic design choices or from implementation-specific factors. We therefore propose an *implementation taxonomy for parallel metaheuristics* (ITPM) that is designed to be fully complementary and non-overlapping with HTPM focusing exclusively on implementation-level characteristics that cannot be derived from HTPM alone. We emphasize parameters which are commonly reported in scientific studies and are valuable for researchers and practitioners in the field. These are: hardware architecture selection (e.g., shared memory, distributed memory, GPU, or heterogeneous systems), communication topology (star, fully-connected, neighbor-based, or hierarchical structures), task granularity (fine, medium, or coarse-grained parallelization), load balancing strategy (static, dynamic, or adaptive work distribution), and parallel programming model (MPI,

OpenMP, CUDA, etc). Table 2 distinguishes these 5 primary characteristics in column "Label", each taking various different values provided in the second column "Values and Description". Inspired by the abbreviated notation of HTPM, we also introduce abbreviations for labels' values in ITPM.

Parallelism Granularity (G) refers to the size or amount of computational work assigned to parallel tasks into which the algorithm, or the overall problem, is

TABLE 2. Implementation taxonomy for parallel metaheuristics algorithms (ITPM).

Label	**Values and Description**
Parallelism Granularity (G)	*Fine-grained* (*Fg*): task involves evaluation of individual neighbor or a subset of neighbors. *Medium-grained* (*Mg*): task involves individual step or a subset of steps constituting a moderate-sized tasks. *Coarse-grained* (*Cg*): task is defined as a sequential component from *High-level* parallel execution.
Parallelization Architecture (PA)	*Shared-Memory* (*SMA*): Systems with global address space *Distributed-Memory* (*DMA*): Multi-node message-passing systems *Graphic Processing Unit* (*GPU*): Massive many-core processing units *Field-programmable gate array* (*FPGA*): Data-flow type architecture *Heterogeneous* (*HET*): Mixed processor types
Communication Topologies (CT)	*Master-slaves* (*Star*): One process communicates with all others. *Fully connected* (*FulC*): Each process directly communicates with all others *Ring/Mesh/Grid* (*Neig*): Each process communicates with immediate neighbors. *Torus*, *Trees*, *Hypercube* (*Adv*): advanced interconnects denoting hierarchical or multidimensional topologies.
Load Balancing (LB)	*Static* (*Stat*): Work distribution predetermined before execution. *Dynamic* (*Dyn*): Work distribution adjusted during execution based on processor availability. *Adaptive* (*Adap*): Work distribution based on runtime performance metrics and learning mechanisms.
Parallel Programming Model (PPM)	*Pthreads*, *OpenMP*, *CUDA*, *MPI*, *Java Threads*, *OpenCilk*, *C++ Threads*

being decomposed. It characterizes how finely or coarsely the workload is divided among processing units and directly affects the balance between the computation workload and communication overhead, system scalability, and parallel efficiency. Fine-grained (*Fg*) parallelism occurs at the level of individual move evaluations or a small subset of neighbors within the current neighborhood - for instance, simultaneously evaluating task-swap moves during the local search phase of a scheduling problem. The shortcoming of *Fg* may originate from overhead due to communication and synchronization. Medium-grained (*Mg*) parallelism corresponds to moderately sized algorithmic components, such as the execution of a neighborhood, an iteration, or a local improvement. In VNS it corresponds to one complete iteration (SH+LS) before synchronization or communication. The disadvantages of *Mg* include load imbalance (the size of *Mg* process may vary in runtime) and inefficient memory and cache management (larger processes may require more memory). Coarse-grained parallelism (*Cg*) suggests that tasks are large in terms of algorithmic work (typically representing entire search trajectories) while communication occurs only occasionally. In parallel VNS, each process executes several VNS iterations or even the whole sequential algorithm, while communication occurs only occasionally, for example, when exchanging the best solution at the end of each or a given number of iterations. In general, a parallel task is classified as *Fg*, *Mg*, or *Cg* according to whether it corresponds to a move evaluation, an algorithmic step, or a complete search trajectory, regardless of its absolute computational cost.

Parallelization Architecture (PA) refers to the underlying computational infrastructure and hardware platform used for parallel algorithm execution. It encompasses Shared-, Distributed-Memory and Graphic Processing Unit (GPU) Architecture described in detail in Section 3, as well as the reconfigurable data-flow type of devices, known as Field-Programmable Gate Arrays (FPGA) [177]. Unlike classical CPU/GPU architectures that execute instructions sequentially, FPGAs allow parallel hardware implementations of multiple parts of a metaheuristic algorithm. The heterogeneous architecture combines multiple types of processing units. This dimension fundamentally influences algorithm design, performance optimization, and implementation strategies in parallel metaheuristic development. PA should not be confused with another potential parameter, the Memory Model (MM). Memory Model represents the programming abstraction that tells how different processes may access and share data during an algorithm execution. More specifically, MM dimension models inter-process communication, data consistency mechanisms, and synchronization strategies at the software level, independent of the underlying hardware architecture. Shared Memory provides a unified global address space where all processes can directly access common data structures. Distributed Memory employs separate address spaces for each process and requires message-passing communication for data exchange and coordination. Hybrid Memory combines both paradigms and necessitates hierarchical communication patterns. However, all three types of memory models can be implemented in any of the above mentioned parallelization architectures using adequate parallel programming models. Therefore, in the proposed taxonomy, MM is not treated as an independent ITPM

parameter, as it emerges from the combination of the algorithmic structure captured by HTPM (e.g., control and communication structure) and PA.

The Communication Topologies (CT) label refers to the abstract communication pathways between parallel processes during the algorithm's execution. CT models how processes are connected and interact, independently of the underlying physical hardware [115]. Star, Fully-connected, Ring, Mesh, and Grid are known as basic communication topologies focusing on simple, local neighbor connectivity. Star Topology (St) refers to a system with one central node, connected directly to all other nodes that have no mutual connections. Communication occurs through the central node, making this communication network simple and easy to manage. The master-slaves control model is a typical example of the star communication topology. The Fully-connected (Fc) topology suggests that each node is directly connected to every other node. In contrast to the star topology, communication in a fully connected topology is peer-to-peer, allowing processes to exchange information directly without a central coordinator. Such a network provides maximum communication flexibility; however, with a large number of nodes, it becomes impractical due to the huge number of direct links. Within Ring, Mesh, or Grid topologies, each node typically connects to immediate neighbor(s). Although these structures can be generically described as network-based topologies, they are distinguished here because their neighborhood structures induce different communication scopes and information propagation patterns, which can significantly influence the behavior and performance of cooperative parallel metaheuristics. The nodes in a ring are arranged in a circular manner, representing a simple topology. Unfortunately, communication latency increases linearly with the number of nodes. Mesh/Grid topology implies that nodes are arranged in a regular two- or three-dimensional lattice, where communication is restricted to directly adjacent neighboring nodes. Such topologies are suitable for spatially decomposed problems in scientific simulations. Torus, Trees, and Hypercube topologies are considered advanced, as they have more complex structures developed to better balance scalability and communication latency. The Tree topology provides a hierarchical structure with parent and child nodes. It is suitable for efficient broadcast and reduction operations. The Torus is similar to a mesh; however, with wrap-around edges, enabling the reduction in the maximum distance between any two nodes compared to a mesh. In addition, the number of distinct paths between nodes is increased, thus improving load balancing and fault tolerance. The Hypercube suggests that the nodes are connected in the form of k-dimensional cube, where $k = \log_2 N$ (N being the number of nodes), where the dimension k also defines the diameter of the network. Due to the logarithmic relationship, N doubles while the maximum communication distance increases only by one. In practice, communication structures may also combine multiple basic topologies, forming hierarchical or clustered network organizations, often referred to as archipelago or cluster-of-clusters structures. Such configurations can be described as compositions of the basic communication topologies considered in this taxonomy.

Load balancing (LB) refers to the strategy that deals with the distribution of computational tasks among available processors to minimize execution time [182].

LB specifically refers to task allocation, which can be *static*, *dynamic*, or *adaptive*. Static load balancing (*Stat*) assigns tasks to processors before execution starts, with fixed workloads that do not change during runtime. For instance, in parallel VNS with static load balancing, if the sequential algorithm performs 1000 iterations and 4 processors are available, each processor might be assigned exactly 250 iterations to execute independently. Consequently, static load balancing is common in independent parallel runs that do not require communication between processes. Dynamic load balancing (*Dyn*) adjusts the task distribution according to processor availability and workload progress to reduce idle times. In particular, when a processor completes the assigned work, it requests a new task. For this reason, *Dyn* is suitable for heterogeneous systems or tasks with variable computation requirements. Adaptive load balancing (*Adpt*) extends *Dyn* by incorporating runtime performance metrics and learning mechanisms to guide allocation of tasks to available processors. The information involved in decision making can be solution quality improvement, convergence speed, or performance characteristics. For example, an adaptive parallel VNS might allocate more iterations to processors exploring promising neighborhoods.

The Parallel Programming Model (PPM) defines how a programmer expresses parallelism in a code and how parallel processes are managed. This model describes the organization of memory, computation, communication, and synchronization. The most common PPMs are practical programming frameworks and APIs used for implementing parallel metaheuristics such as Pthreads, OpenMP, CUDA, MPI, Java Threads, OpenCilk, and C++ Threads.

In summary, each ITPM parameter represents an implementation decision that is independent of the high-level algorithmic strategy defined by HTPM. For example, the same HTPM classification (e.g., single-level decomposition with independent implicit search space exploration and decentralized control) can be implemented across different architectures (shared memory with OpenMP or distributed memory with MPI), with different granularity (coarse-grained algorithm runs or medium-grained individual iteration steps), with different load balancing approaches (static predetermined distribution or dynamic runtime adjustment), etc. By complementing rather than duplicating HTPM, ITPM provides genuinely new information, avoiding redundancy while enabling comprehensive characterization of parallel implementations. Therefore, the value of ITPM lies not only in providing a descriptive vocabulary, but in making the field's collective design choices visible for the first time.

4.2.3. Final Remarks. The ITPM framework that we have presented depicts a foundational, yet extensible approach to characterizing parallel metaheuristic implementations. We should note that ITPM's design philosophy prioritizes *pragmatic utility* over *exhaustive completeness*. We have selected the five core parameters (G, PA, CT, LB, PPM) because they are: (1) commonly reported in parallel metaheuristic literature, (2) essential for implementation reproducibility, (3) key factors that impact both performance and scalability, and (4) relatively straightforward to identify and document. However, the taxonomy remains open to the inclusion of additional parameters that should satisfy the same standards. In particular, the

new parameters should contribute meaningfully to understanding of the considered algorithm. In addition, if there is a possibility that the existing taxonomy has not captured important aspects of how an algorithm works, this leaves the capacity for introduction of new parameters that would fill the gap. Another reason why the possible values included in Tables 1 and 2 may not be exhaustive is that parallel programming support, both hardware and software, evolves continuously. Therefore, both HTPM and ITPM should remain open to refinements and extensions as parallel computing technologies and metaheuristic strategies evolve.

Finally, we note that a complete parallel metaheuristic description requires both HTPM and ITPM frameworks. HTPM specifies *what* parallelization strategy is employed at the algorithmic level. ITPM specifies *how* that strategy is turned into executable code. Together, these complementary frameworks help to compare different parallel algorithms, contribute to implementations that are easier to reproduce, and support better design choices. For example, two algorithms classified as 1L/SC/1C/S/SPSS according to HTPM may exhibit significantly different performance in practice. However, their ITPM classifications immediately reveal which implementation-level differences are the likely explanation: hardware architecture, memory model, or parallel programming model. Researchers should provide classifications according to both taxonomies to facilitate understanding, reproduction, and fair comparison with existing approaches.

4.3. Parallel metaheuristic implementations. As we already mentioned in the introduction, metaheuristics have been parallelized since at least 1987, when [112] described three strategies for developing parallel simulated annealing (SA). The variety of technologies for hardware and software support for the development of parallel computations initiated extensive research in this field. One of the earliest surveys including parallel metaheuristics is the book chapter [95] published in 1997. The authors considered parallel variants of SA, TS, GA, and GRASP. Starting from [6, 129], numerous survey articles and book chapters have appeared in the relevant literature [7, 8, 36, 65, 70, 128, 162, 181]. A particularly interesting recent survey [168] examines the application of parallel computing to metaheuristic algorithms, describing various parallelization models, communication methods, and implementation platforms. Its most distinctive contributions are the inclusion of a specialized parallel architecture (FPGA) and several less conventional metaheuristic methods, such as the Chaos Optimization Algorithm (PCOA) and Cat Swarm Optimization (CSO).

Most parallel implementations in the literature concern population-based methods, because they exhibit inherent parallelization properties [4, 121]. In [4], the authors developed an MPI-based portfolio framework for the parallelization of metaheuristics and tested it on Artificial Bee Colony (ABC), Differential Evolution (DE), Particle Swarm Optimization (PSO), and their combinations. Different parallelization models for population-based metaheuristics, as well as their hybridization, have been studied in [121]. In addition, a review of the literature considering parallel GA and parallel PSO was provided. Numerous survey papers focus specifically on the parallelization of population-based metaheuristics [17, 24, 93, 113], to mention only a few.

We also review here the contributions to the parallelization BCO provided in [53, 54, 56]. The three parallelization strategies are proposed for a distributed-memory multiprocessor architecture under the MPI communication protocol. The independent execution of several copies of the BCO algorithm with a reduced value of the stopping criterion represents the simplest parallelization strategy. Each processor independently executes a sequential BCO, initialized with a different seed value, the obtained results are collected at the end, and the best is reported to the user. Two additional variants are also considered: in the first, the population is divided among processors, instead of stopping criterion reduction, while in the second it is also allowed to change the BCO parameter values. The second strategy assumes cooperative execution of several BCO variants. This synchronous coarse-grained parallelization allows data exchange at certain predefined execution points. The exchanged data (usually the current best solutions) are used to redirect the search. Cooperation is also implemented in an asynchronous way, to minimize the idle time of processing units and to increase the diversification of the search. Several variants are implemented, differing in parameter values, communication model, and stopping criterion. The proposed parallelization strategies have been tested on the benchmark instances of static scheduling independent tasks on identical machines problem. The presented experimental results show very good performance of all parallel BCO variants. Significant speedup is obtained in the case of independent execution, while preserving the solution quality. On the other hand, the best performance of the cooperative strategy is obtained when executed on a modest number of processors engaged in parallel BCO execution. Compared to sequential execution, better quality solutions are found within the same amount of wall-clock time. In addition, as the number of processors that execute parallel BCO increases, asynchronous strategies outperform others with respect to both solution quality and running time.

Strategies for parallelization of trajectory-based metaheuristics have also been studied. They are systematically reviewed in [8] up to 2022. The authors considered parallelization using various types of multi-core (shared-memory) and distributed architectures. The most commonly parallelized metaheuristics are Iterated Local Search (ILS), Large Neighborhood Search (LNS), SA, TS, and GRASP. Among the 127 publications analyzed in [8], 18 are devoted to parallelization of VNS. We extended this number to 58.

Parallel TS designed for the capacitated vehicle routing problem (CVRP) was proposed in [101, 102]. In [101] a cooperative parallel multi-neighborhood TS algorithm (PMNTS) was presented. Three inter-route neighborhoods are explored: reinsertion, exchange, and 2-opt. PMNTS explores a shared-memory model for synchronous communication between four threads. A pool of solutions, stored in shared memory locations, contains a sorted list of best solutions provided by all threads. The newly-found best solution is explored first, if they exist. If there is no such solution, the sorted list is scanned, and the first unexplored solution is selected by each thread. The threads differ with respect to the parameter settings and the subset of neighborhoods used. The proposed strategy is classified as 1L/SC/pC/K/SPDS according to HTPM and *Mg/SMA/FulC/Stat/C++ Threads*

according to ITPM. The experimental evaluation of PMNTS is performed on two benchmark sets containing 32 large scale instances. For each instance, PMNTS was run 10 times with different seed value, the best and average results are reported. The presented results show that PMNS is competitive with the existing methods, that it provides 4 new best solutions, and that it has a small average deviation with respect to the best-known solutions from the literature. The authors also analyzed the contribution of neighborhoods to the solution quality, as well as some connections between neighborhoods and instance characteristics.

An asynchronous cooperation of multiple parallel TS processes, named cooperative parallel metaheuristic (CPM), for the same problem is proposed in [102]. The best-found solutions are exchanged through a common solution pool. In the pool, the solutions are clustered according to the similarity defined as the relative number of common edges that each solution shares with the other solutions in the same cluster. Within each cluster, the solutions are sorted with respect to their quality. Three types of reduced inter-route neighborhoods are explored within CPM. The reduction is performed to increase the efficiency of the search. The authors designed a basic configuration of CPM that runs on eight processing units. The first process is used to manage the solution pool, the next four TS processes perform diversification steps, and the remaining three are devoted to the intensification of the search. If more processors are available, the search processes of the basic CPM can be multiplied and executed on the available processors. Having 24 processing units available, the authors used 14 diversification and 9 intensification processes in their experiments and called it the standard configuration of CPM. According to HTPM, CPM can be classified as 1L/CS/pC/K/MPDS, while it belongs to the class $Cg/DMA/Star/Stat/MPI$ in ITPM categorization. Computational experiments are conducted on the same two sets of large-scale benchmark instances as in [101]. The results reveal that CPM generated 10 (out of 32) new best results and demonstrated competitiveness with respect to the best-performing methods available in the literature.

Although covered in most of the surveys, GPU-based parallel metaheuristics are also reviewed in [70, 162]. Among the trajectory-based metaheuristics, TS is identified as the most commonly parallelized method, followed by GRASP and SA. ACO, PSO, and GA are leaders in the class of population-based methods. The GPU-accelerated GA metaheuristic for CVRP has been proposed in [3]. Hybridization between CPU and GPU has been widely explored by the Brazilian research group and applied to a variety of metaheuristics, such as GRASP [11,137], GA [29], and different variants of VNS that are reviewed in the next section, together with other GPU-based parallel VNS implementations. In [29] the hybrid evolutionary algorithm for task scheduling in the cloud environment is proposed. To efficiently address this time consuming scheduling, the authors proposed to explore massive parallelism enabled in GPUs and achieved about 98.83% reductions in the scheduling time, without the degradation in solution quality. Parallel GA (PGA) for large-scale VRP, implemented on an eight-GPU NVIDIA DGX-1 server, is described in [2]. This is actually a CPU-GPU hybrid, as coarse-grained GA-parallelization runs on CPUs and operates on subpopulations that are processed on GPU cores.

The main goals were to achieve high throughput and reduce latency for full GPU utilization. An experimental evaluation is conducted on benchmark instances of VRP containing up to 20,000 nodes. The algorithm performance (speed) is examined using different number of GPU cores, including a multi-CPU implementation. It has been concluded that, with respect to pure over the CPU run, execution time can be reduced by a factor of 1,700 when utilizing 5,120 CUDA cores.

One of the rare parallelization of metaheuristics using the FPGA architecture is provided in [186]. The authors considered GA and PSO algorithms and realized their parallel FPGA-based implementations for TSP. To evaluate the performance of the implemented parallel algorithms, a comparison was conducted with OpenMP-based and CUDA-based parallel variants. In addition, a case study that includes FPGA-based parallel SA for job shop scheduling problem (JSSP) is considered and compared with the MATLAB SA implementation. The provided experimental results show that the proposed algorithms exhibit the highest real-time performance with both solution quality and running time.

5. Survey of Parallel VNS Approaches

Our objective is to provide a comprehensive survey of the existing literature on parallel VNS that addresses some of important research questions. Ideally, our survey would include a wide range of information, covering all aspects, from the design of the parallel algorithm to its low-level implementation considerations. However, as highlighted earlier, the performance of parallel algorithms is influenced by a multitude of factors, making such an exhaustive coverage impractical. Therefore, we provide an insightful overview of parallel VNS studies, attempting to answer questions related to both conceptual design and practical implementation. Beside resolving the values of 10 HTPM and ITPM taxonomy parameters listed earlier, there is a need to answer the additional questions (presented here in no particular order):

(1) The relationship between sequential and parallel VNS
   - What type of sequential VNS algorithm serves as the basis for parallelization?
   - How many parallel variants of the VNS algorithm does the publication introduce?

(2) Performance Evaluation Metrics
   - What metrics (speedup, efficiency, scalability, etc.) are used to assess parallel performance?
   - Are statistical analyzes or multiple runs included?

(3) Parameter Settings and Adaptations
   - What VNS parameters (e.g., k_{max}, k_{min}, shaking intensity, neighborhoods, etc.) are utilized and what are their values?
   - Have the parameters been tuned or adapted for parallel execution?

(4) Search Dynamics and Robustness
   - How does parallelization affect search dynamic and solution quality evolution?

- Is search space coverage or solution diversity improved?

(5) Experimental Setup and Benchmarking
- Which problem instances or benchmarks are used?
- How comprehensive is the evaluation?
- Are comparisons made with sequential and/or other parallel methods?

(6) Novel Algorithmic Contributions
- Are new communication or cooperation methods introduced?
- What innovations distinguish the study from previous work?

The term search dynamics is used to denote the empirical evolution of solution quality during the search process, without implying theoretical convergence to a provable optimum.

It should be noted that the listed research questions are not exhaustive — they represent aspects relevant to the design of parallel VNS algorithms that are mostly reported across the literature. The research questions listed above reflect two distinct types of challenges. The first type consists of challenges inherited from sequential metaheuristic research — performance evaluation, parameter tuning, benchmarking, search dynamics, and experimental setup — where established standards exist but are inconsistently applied in the parallel VNS literature. The second type consists of challenges specific to parallel VNS, identified through the systematic classification performed in this survey: the sequential data dependency inherent in VNS that limits parallelization granularity; the mismatch between algorithmic and hardware decomposition levels in heterogeneous architectures, which prevents straightforward porting of strategies across hardware platforms; the difficulty of load balancing under stochastic variability, problem-specific constraints on neighborhood design, the design of effective cooperation strategies, and the difficulty of transferring parameter settings from sequential to parallel contexts; growing communication overhead as the number of processing units increases; and inconsistent implementation reporting across studies (which makes it impossible to reliably attribute performance differences to algorithmic rather than implementation factors). Systematically addressing these questions moves toward a long-term goal of predicting parallel algorithm performance from design factors alone, without the need for extensive experimentation. However, this goal remains out of reach due to inconsistent reporting across studies, strong problem-dependency of performance outcomes, interaction effects between design factors, and the inherently stochastic nature of parallel VNS algorithms. Nevertheless, we believe that documenting these factors as consistently as possible across the surveyed studies provides a valuable foundation for understanding design choices and their practical consequences, and constitutes a step toward a future predictive framework for parallel VNS performance.

The literature considered in this survey was collected using freely accessible academic platforms, primarily Google Scholar and ResearchGate. The search process was based on combinations of core VNS-related terms, including “variable neighborhood search”, VNS, GVNS, RVNS, and VND, with parallel computing keywords such as “parallel computing”, “parallelization”, “parallelisation”, “parallel

architecture", "parallel programming", "distributed programming", "shared memory", and "GPU". In addition to keyword-based search, both backward and forward snowballing strategies were employed: backward snowballing was used to examine the reference lists of relevant publications (including survey articles on parallel metaheuristics), while forward snowballing was applied to identify more recent studies citing foundational VNS contributions. This combined approach helped to ensure comprehensive coverage of the literature at the time of search. It should be noted that five additional parallel VNS publications that appeared in 2024 and 2025 were identified after the finalization of the literature collection [25–27,149,150], confirming that coverage of recent publications and conference proceedings is inherently limited by indexing delays in academic search platforms.

5.1. Research Evolution. In total, we reviewed 58 publications published between the years 2001 and 2025. Our survey indicates that the topic of parallel VNS has experienced uneven yet persistent research activity over the past 25-year period, reflecting continued interest within the research community. Parallel VNS represents a niche subfield at the intersection of metaheuristic optimization and parallel computing, whose activity is driven primarily by advances in parallel hardware platforms. At the same time, the field has evolved in a manner broadly consistent with established bibliometric patterns for scientific fields, progressing through characteristic stages of growth and maturation [43, 147]. This progression is captured in the annual scientific production plot (*publication trend*), presented in Figure 7 which reveals four distinct phases in the field's evolution. In particular, the publication trend shows one clearly dominant burst of activity centered around 2015–2017, driven by the maturation of GPU and many-core architectures. The burst is preceded by a foundational period of modest but sustained activity spanning 2001–2005, coinciding with the widespread availability of multi-core processors. Between these two periods, a prolonged interval of steady but uneven activity is observed. Within this publication trend, we distinguish four phases within the field's evolution:

- Foundational phase (2001–2005, 9 publications, 1.8 publications / year) - early period with pioneering work.
- Latent phase (2006–2014, 11 publications, 1.2/year) - steady but moderate activity before the next enabling hardware generation.
- Rapid growth period (2015 – 2018, 22 publications, 5.5/year) - GPU-driven explosion; the most productive period in parallel VNS research.
- Maturation phase (2020–2025, 16 publications, 2.7/year) - sustained output with increasing algorithmic sophistication.

The foundational phase has produced 9 publications, where nearly half appeared in conference proceedings, particularly the 4th Metaheuristics International Conference (MIC'2001), which hosted two seminal papers [37] and [138]. The quality of the foundational work is also evidenced by two journal papers [39] and [75] that appeared in the Journal of Heuristics, known as the premier journal for metaheuristic research. Two technical reports [139] and [38] document detailed methodological developments and computational experiments. In particular, seven of the nine

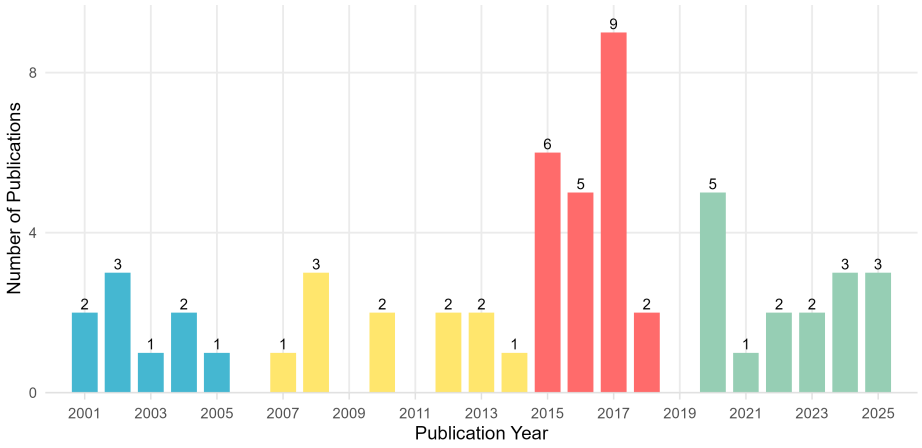


FIGURE 7. Annual distribution of published research on parallel VNS over the period 2001 – 2025, showing four distinct phases of research activity with a dominant peak in 2017: Foundational phase (blue), Latent phase (yellow), Rapid growth period (red), and Maturation phase (green).

publications originated from two research groups - the Crainic group [37–39, 47] and the Ochi group [59, 138, 139] - marking the first evidence of a stable author core in the field. The remaining two publications [75, 133] belong to the Canary Island group (Moreno-Pérez and collaborators). Finally, the first (and only to date) comprehensive survey of parallel VNS algorithms in [133] showed that parallel VNS became a recognized subdiscipline within parallel metaheuristics.

The latent phase is characterized by steady and irregular publication activity driven by the limited availability of new parallel hardware platforms, while still exhibiting methodological refinement and diversification across multiple application domains. Despite the moderate publication rate, i.e., 1.2 publications per year, with no publications in 2006, 2009 and 2011, the period produced a notable increase in high-quality journal publications. Namely, 6 out of 11 publications appeared in journals, compared to 2 of 9 in the foundational phase. The spectrum of optimization problems broadened compared to the foundational period. In addition to location and routing problems, parallel VNS was applied to scheduling problems [13, 163, 184], multi-objective network design [69], and constraint satisfaction [142]. A significant milestone was achieved with [167], which gained more than 400 citations, demonstrating the effectiveness of parallel VNS on large-scale routing problems and scaling to hundreds of cores. Following the expansion of hardware advancement, [31] is the first to explore acceleration of the graphics processing unit (GPU) for parallel VNS. Based on everything mentioned, we can conclude that the latent phase has been successful in consolidating parallel VNS as a mature and versatile optimization technique, creating the foundation for rapid growth in the following phase.

The rapid growth period (2015-2018) represents the most intensive period in the evolution of parallel VNS, producing more than 40% of all parallel VNS publications in just 4 years. The year 2017 accounts for 9 publications (15% of the total number of publications), marking the absolute peak of research activity. It is our view that phase III is not characterized by a single dominant hardware platform or a research group, but represents a confluence of three research directions: (1) maturity of VNS variants, (2) the availability of commodity multi-core processors and programmable GPU architectures, and (3) large-scale real-world instances. The algorithmic maturity direction is evidenced by the direct use of established sequential VNS variants without additional methodological development, in the sense that no new sequential VNS variant is introduced. In fact, the main research question is how to execute their computationally demanding components more efficiently. For example, decomposition-guided VNS is applied to weighted constraint satisfaction problems [143] and graphical model energy minimization [141]. Other mature VNS variants are deployed for multiprocessor scheduling with communication delays [51], cutwidth minimization [67], the traveling salesman problem [132], disjunctively constrained knapsack problems [148], circulant weighing matrices [166], and dynamic memory allocation in embedded systems [159, 160]. The second direction of Phase III reflects the increasing programmability of GPU architectures and the broader accessibility of parallel hardware platforms. GPU acceleration is represented by CUDA-based implementations [11, 60], CPU–GPU hybrid schemes [10, 30], and multi-GPU configurations [156]. Accessibility was not only a consequence of hardware availability, but of more mature programming models. In particular, CUDA Unified Memory simplified CPU–GPU memory management, and directive-based approaches such as OpenACC have enabled GPU programming without low-level expertise. These developments collectively lowered the barrier to GPU adoption for OR researchers. Beyond GPUs, the second direction of Phase III also includes parallel VNS implementations on standard multi-core and distributed platforms, addressing problems such as bus terminal location [64], multiprocessor scheduling [51], and the traveling salesman problem [132]. The third direction includes papers where problem scale and practical relevance justify parallelization, i.e., whose neighborhood structures grow rapidly with instance size. Such problems make sequential neighborhood exploration, whether exhaustive or partial, practically infeasible within reasonable time limits. Examples include large-scale urban transport [134], vehicle routing with simultaneous pickup and delivery [30], warehouse order batching [130], and vehicle routing with divisible deliveries [146]. The maturity of the field within this direction is additionally evidenced by participation in solver competitions (e.g., VeRoLog challenge), where [98] and [175] present parallel VNS approaches demonstrating competitiveness under standardized experimental conditions. Therefore, Phase III is best understood as a period of hardware–method–application alignment, in which mature VNS methods, accessible parallel hardware, and demanding applications reinforced one another. Although formally published in 2020, [94] and [71] should be treated as part of phase III due to their development timelines and because they combine several Phase III directions. The study [94], available online since 2018, presents a mature cooperative BVNS on

commodity multi-core infrastructure. [71], first received in 2018, integrates parallel GVNS as a local search component within PSO framework for job shop scheduling with assembly operations, thus combining all three Phase III directions.

Phase IV — the maturation phase (2020–2025) reflects not only the consolidation of parallel VNS as a practical optimization tool, but also a shift in how parallelization is to be utilized. The phase contains an average of 2.7 publications per year across all six active years, comparable in rate to the foundational phase. However, two papers formally published in 2020 [71, 94] represent research conducted in the Phase III context and are discussed there accordingly. The application domain expands beyond classical routing and scheduling problems toward more specialized variants. Examples include portfolio optimization [5], energy distribution network planning [158], bi-objective obnoxious p-median [169], covering salesman variants [185], pollution-aware routing [116], and long-term mine planning under geological uncertainty [149,150], multi-depot multi-compartment vehicle routing [26], subgraph optimization [27], and robust p-median [25]. Regarding the parallelization strategies themselves, a notable methodological contribution is [105], which takes the parallelization strategy itself as its primary research object. It compares a non-cooperative model with two cooperative models for the CVRP and shows that the cooperation mechanism can strongly influence solution quality, with filtered communication performing best on harder instances. [28] represents one of the most comprehensive component-level decompositions by parallelization of multiple VNS components, including initial solution generation, shaking, and local search. [180] proposes both serial and parallel version of a VNS for the order batching problem, reporting that only minor performance improvements were observed from parallelization. Moreover, [180] suggests that trivial parallelization without cooperation or a sufficient number of parallel threads is unlikely to be beneficial. The sustained publication activity through 2025, including submitted works, indicates that parallel VNS remains an active research area, with increasing attention to the conditions under which parallelization is beneficial.

5.1.1. Analysis of publication venues. This section examines the publication venues of the 58 reviewed publications, focusing on their proportional distribution by publication category and the ranking of journal venues. The reviewed studies are grouped into journal articles, proceedings and book chapters and other types of publications including technical reports and doctoral theses.

Figure 8 shows a horizontal bar chart that depicts the five most prominent journal venues among the 21 distinct journals in the corpus. Computers & Operations Research (COR) and Electronic Notes in Discrete Mathematics (ENDM) share first place with five papers each. While ENDM accumulated its five papers through its role as an ICVNS proceedings outlet, COR is a well-established Q1 journal in the Operations Research community, and its five papers reflect that a substantial part of parallel VNS research has been published in a high-ranking OR venue. The next most frequent outlet is Optimization Letters with three papers, followed by Applied Soft Computing and the *European Journal of Operational Research* with two papers each. Overall, the range of publication sources indicates that no single journal dominates the topic of parallel VNS. Instead, parallel VNS related research is

widely disseminated across many journals and publishers, reflecting the interdisciplinary publication landscape rather than being confined to a few selected journals. In addition, several identified journals are highly influential within their respective fields and represent established outlets for metaheuristic and applied optimization research. Their relevance is supported by their quartile rankings and impact factors.

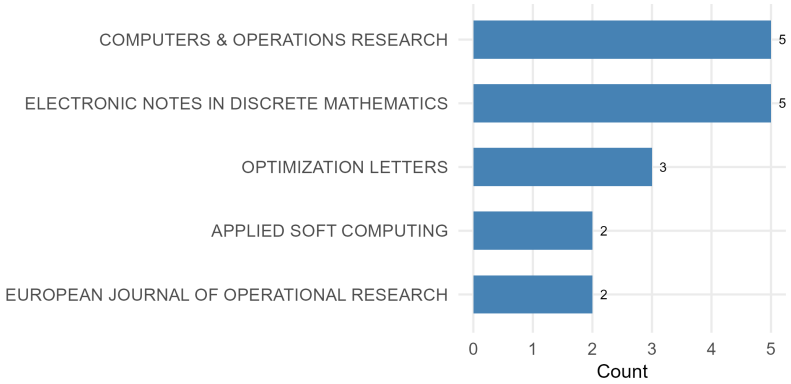


FIGURE 8. Top five primary publication outlets among the reviewed papers.

As a valuable metric, we incorporate the *impact factor quartile* to evaluate the relative influence of publication venues within their respective subject categories such as Operations Research, Computer Science, Mathematics, Engineering, Business and Management. These subjects also reflect the interdisciplinary nature of the parallel VNS research area. Quartiles categorize journals into four equal groups (Q1, Q2, Q3, Q4), the first quartile (Q1) representing the top 25% of journals ranked within their respective fields. Quartile assignments are based on the Journal Citation Reports (JCR) from Web of Science (https://wos-journal.info) using the subject category most closely aligned with the venue's main disciplinary orientation. For journals indexed in multiple categories, one category is selected per venue and applied consistently. Where we missed JCR coverage, we employed the SCImago Journal Rank (SJR, Scopus) which has affected venues such as *Businees Research* (Q1, SJR), *Journal of Heuristics* (Q2, SJR) and *Eletronic Notes in Discrete Mathematics*. The later was discontinued in 2019, thus we use its last recorded quartile Q4. For journals indexed in JCR but absent or discontinued in Scopus/SJR, such as *Assembly Automation*, the JCR quartile is retained (Q3, JCR).

Figure 9 summarizes the publication-venue distribution from two complementary perspectives. The left chart shows the distribution of the 21 ranked journal venues by quartile, while the right chart shows the distribution of all 58 publications across publication venue type. Beside ranked journal papers, the right chart includes "Proceedings & Books" (covering conference and symposium proceedings, and book chapters) and "Other" (covering technical reports, theses, not indexed journal paper). Although proceedings are not usually assigned quartiles, they may

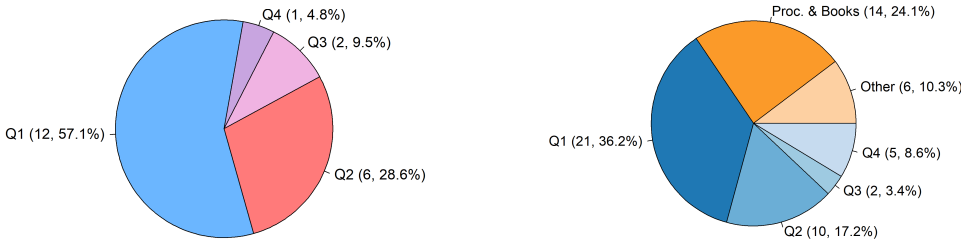


FIGURE 9. Parallel VNS publication-venue distribution (2001–2025) with respect to journals and publications themselves.

still have publication prestige within their respective research communities. The left chart in Figure 9 reveals that among the 21 ranked journal venues, 12 (57.1%) are classified as Q1, six (28.6%) as Q2, two (9.5%) as Q3, and one (4.8%) as Q4, indicating that the majority of journals in which parallel VNS research appears are highly ranked within their respective fields. The right chart shows that the largest share of the 58 publications appear in Q1 journals (21 papers, 36.2%) which are [5,25,28,30,64,67,69,98,114,134,145,146,156,158,160,163,167,175,178,180,184] followed by Q2 type journals (10 papers, 26.3%) [39,51,75,94,105,130,149,166,169, 185]. A smaller proportion of journal papers appear in Q3 and Q4, such as [71,148] in Q3-ranking journals, as well as five papers [10, 11, 49, 143, 159] published in Q4 ranking Electronic Notes in Discrete Mathematics. One paper [132] is published in a journal absent from both JCR and SJR and is excluded from the quartile count. Together, Q1 and Q2 papers account for 81.6% of the ranked journal output, reflecting a strong overall citation impact across the journal portion of the corpus. The remaining 19 publications fall outside journal venues: 14 (24.1%) appear in Proceedings & Books [13, 26, 27, 31, 37, 47, 59, 60, 106, 116, 133, 138, 141, 142], and 6 (10.3%) in the Other category [38, 50, 110, 132, 139, 150], comprising technical reports, a miscellaneous report, and doctoral theses. This diversity of high- to mid-ranked journal publication venues highlights both the maturity and ongoing research interest in parallel VNS algorithms.

5.1.2. Parallelization architecture analysis. Figure 10 summarizes the distribution of parallelization architectures across the 57 publications for which PA is identified. The largest portion comprises shared-memory architectures (SMA, 27, 47.4%) publications, leveraging multi-core CPUs within a single system for fast, synchronized parallel computation [5, 25–28, 64, 67, 69, 71, 75, 94, 98, 105, 106, 116, 130, 146, 149, 150, 158–160, 169, 175, 178, 180, 185]. The second largest portion of the pie is distributed-memory architectures (DMA, 23 publications, 40.4%), which deploy computations across multiple nodes or machines and are favored for scalability and handling large-scale problems [13, 37–39, 47, 49–51, 59, 110, 114, 134, 138, 139, 141–143, 145, 148, 163, 166, 167, 184].

GPU architectures (6 papers, 10.5%) cover both pure GPU and mixed CPU-GPU setups, reflecting growing interest in harnessing graphics processors for their high throughput on massively parallel tasks [10,11,30,31,60,156]. One paper (HET,

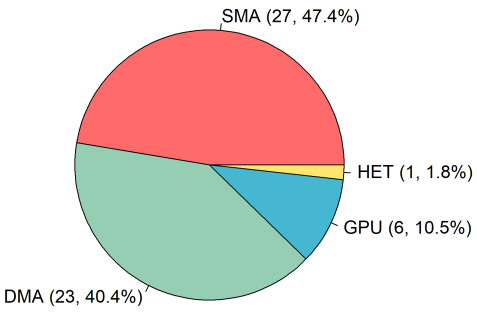


FIGURE 10. Distribution of parallelization architectures among the 57 parallel VNS publications: shared memory (SMA), distributed-memory (DMA), GPU-based (including CPU-GPU), and hybrid (HET) combining distributed and shared.

1.8%) utilizes a heterogeneous architecture combining elements of distributed- and shared-memory computing to optimize resource utilization and performance across diverse computational demands [132].

5.1.3. Optimization problem category. Figure 11 shows the distribution of the 57 publications across optimization problem categories: Routing, Scheduling, Location, Assignment & Allocation, Graph Optimization, Scientific Computing, Packing and Others. Scientific Computing category addresses optimization problems that originate outside classical operations research, arising from fields such as artificial intelligence, bioinformatics, or discrete mathematics. They are usually expressed as general discrete optimization models, such as constraint satisfaction, graphical-model energy minimization, or the search for combinatorial designs. The distribution shows that parallel VNS research is strongly focused on problems from three classical operations research domains -routing, scheduling and location - accounting for 75.4% of the total number. Routing problems form the largest category with 19 publications (33.3%) [11, 26, 30, 31, 59, 98, 105, 106, 116, 132, 134, 138, 139, 145, 146, 156, 167, 175, 185]. Scheduling problems constitute the second largest category with 14 publications accounting for 24.6% of the total number [13, 47, 49–51, 71, 110, 114, 130, 149, 150, 163, 178, 184], followed by Location problems with 10 publications (17.5%) [25, 28, 37–39, 60, 64, 75, 94, 169].

Assignment-Allocation problems appear in four papers (7.0%) [69, 158–160], while Graph optimization and Scientific Computing each contribute three papers (5.3%): Graphs [27, 67, 141] and Scientific [142, 143, 166]. Packing problems account for two papers (3.5%) [148, 180]. The remaining two papers (3.5%) are grouped under Others [5, 10], comprising portfolio optimization [5] and production/inventory planning [10], each representing a domain outside the six primary categories.

5.1.4. Final remarks. The bibliometric analysis reveals several key observations. First, the publication rate varies considerably across phases, rising from 1.2 studies per year in the Latent phase to 5.5 in the Rapid growth phase before settling

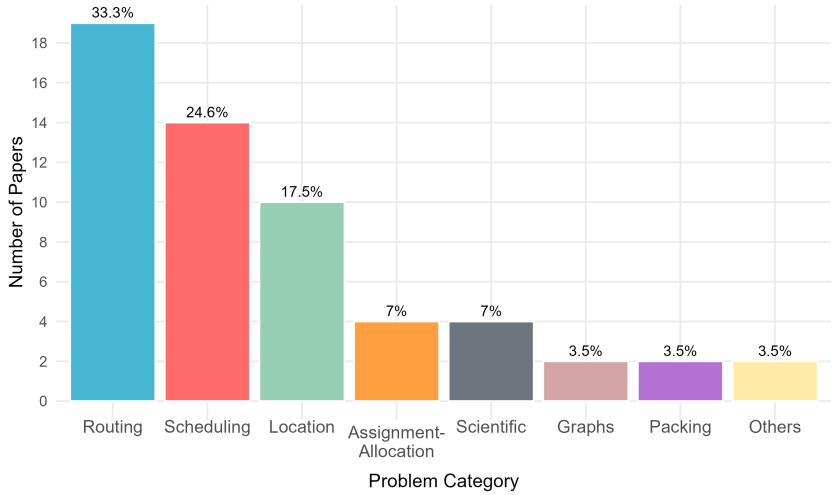


FIGURE 11. Distribution of the 57 classified parallel VNS publications across optimization problem categories.

at 2.7 in the Maturation phase, indicating that the field has moved beyond its period of rapid expansion into a more stable research activity. The causes of this growth, as discussed in the phase analysis, extend beyond the bibliometric data and include factors such as hardware availability, algorithmic maturity, and problem-driven demand. Second, among the 38 ranked journal papers, the majority appear in Q1 or Q2 venues (81.6%), reflecting a high standard of dissemination and confirms that parallel VNS research is not concentrated in a singe venue but reaches a broad academic audience. Third, shared-memory architectures account for the largest share of implementations (47.4%), followed by distributed-memory (40.4%), together accounting for the vast majority of parallel VNS implementations. GPU-based approaches represent a smaller but distinct contribution (10.3%). Finally, problems that belong to categories Routing, Scheduling, and Location together account for 75.4% of the reviewed publications, confirming the dominance of classical combinatorial OR problems in the VNS parallel literature.

We organize the following subsections according to parallelization architecture (PA), reflecting the strong influence of the underlying hardware platform on the choice of parallelization strategy. Within each PA-defined subsection, publications are grouped by optimization problem category. Each of them is assigned the category that best reflects its primary optimization focus. Such organization serves to structure the presentation of the survey, not to introduce another classification scheme.

5.2. Shared-memory parallelization of VNS. Here we review studies that implement parallel VNS on shared-memory parallelization architecture (SMA). Shared-memory systems enable low-latency communication between threads and straightforward implementation of fine-grained parallelization strategies, making them well

suited for implementations that require frequent cooperation or tight synchronization during local search.

The 27 publications reviewed in this section represent 47.4% of the parallel VNS corpus, establishing shared-memory as the most frequently utilized parallelization architecture in the surveyed period. SMA implementations span several problem categories, with Routing, Location, and Scheduling being the most frequent. Figure 12 shows how publications employing shared-memory PA are distributed across optimization problem categories.

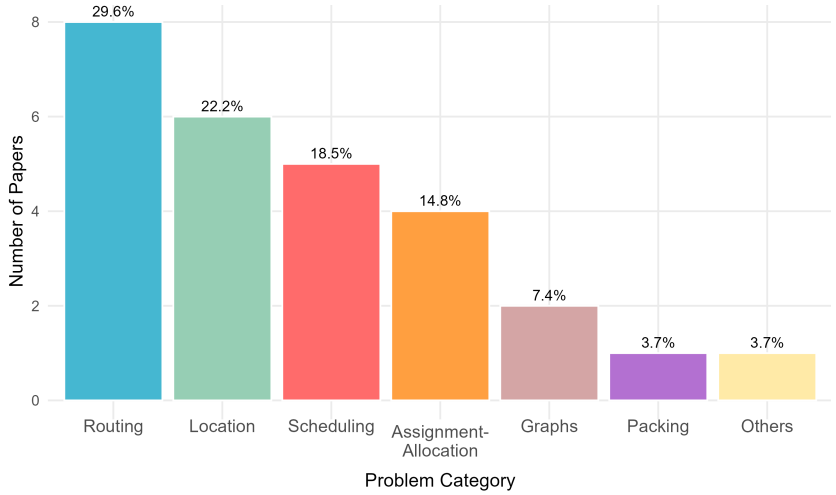


FIGURE 12. Distribution of shared-memory parallel VNS publications across optimization problem categories.

As shown, Routing problems constitute the largest category (29.6%, 8 studies), followed by Location (22.2%, 6 publications), Scheduling (18.5%, 5 publications), and Assignment-Allocation (14.8%, 4 publications). Graph optimization accounts for two papers (7.4%) [27, 67], while Packing and Others each contribute one paper (3.7% each) [5, 180]. In what follows, publications are reviewed per category in chronological order.

5.2.1. Location problems. To the best of our knowledge, the authors of [75] were among the first to deal with parallelization of VNS. Actually, the paper has been accepted in the year 2000, and completed even before, as evidenced by the acknowledgments. Therefore, we devote special attention to this study. This foundational paper introduces the three canonical parallel VNS strategies for the p-median problem, implemented in C/OpenMP on a shared-memory PA, and provides the exact algorithmic templates that became the reference point for subsequent parallel VNS publications. The first approach, named Synchronous Parallel VNS (SPVNS), was to parallelize LS within a sequential VNS. The main thread executes shaking and neighborhood change steps of VNS, while the neighborhood to be searched in LS

is decomposed and distributed among slave threads. The authors used a swap neighborhood, defined as the substitution of the center with a non-center location, indicating that the BVNS variant is used in parallelization. The swap neighborhood has a size of $p(n-p)$, where p denotes the number of selected locations, while n represents the number of available locations. Therefore, the decomposition is quite straightforward. The communication is performed after each iteration of LS to identify the best-found solution as the starting point for the next iteration. LS iterations are executed until no improvement of the starting point is obtained. The SPVNS (as well as the other two implementations) completes its execution when k reaches k_{max}. This low-level parallelism can be classified as 1L/A/1C/S/SPSS (HTPM) and *Fg/SMA/Star/Stat/OpenMP* (ITPM). In particular, the decomposition operates at a single level (1L) with fine granularity (*Fg*) because each worker thread evaluates a small subset of individual neighbor solutions. The decomposition is A, as the neighborhood is partitioned into disjoint subsets that are evaluated in parallel by the threads. There is a single control that manages the parallel local search synchronously (1C and *Star*). The communication is performed via shared-memory architecture (*SMA*) synchronously (S), because the threads must wait to synchronize solutions after exploring their neighborhoods and before proceeding with the exploration. There are no dynamic re-distributions between processes within all three strategies, thus LB=*Stat*. After the shaking phase, each thread will proceed with local search starting from the same initial point (SPSS). The second strategy, referred to as Replicated Parallel VNS (RPVNS), involves independent runs of several identical sequential VNS procedures that start from different random initial solutions. There are no communications during the execution; only the the best solutions are collected at the end and compared to identify the overall best. It is classified as 1L/SI/pC/A/MPSS by HTPM. Specifically, each thread independently runs the same VNS algorithm (SI and *Cg*) starting with its own initial solution (MPSS). It performs shaking, local search, and neighborhood change without communication during the run (pC). Communication is typically minimal and performed at the end of execution to identify the best obtained solution. Therefore, it can be considered asynchronous in the loose sense (A), as threads do not block each other by waiting for communication during the search. It's ITPM classification is *Cg/SMA/-/Stat/OpenMP*. The third variant, called Replicated Shaking Variable Neighborhood Search (RSVNS), assumes a synchronous cooperation mechanism through a classical master-slaves architecture. The main thread runs a sequential VNS in which the basic steps, Shake and Local Search (SH+LS), are performed in parallel by other threads. On each thread, shaking starts from the same best-known solution and uses the same neighborhood index k. When SH+LS is completed on all threads, the best-found solution is communicated to the main thread, which performs the neighborhood change step. Thus, HTPM is 1L/SC/1C/S/MPSS and ITPM classification takes values *Mg/SMA/Star/stat/OpenMP* as the parallelization of SH+LS steps is considered medium-grained (G=*Mg*). All variants are coded in the C programming language and executed on TEIDE (8 processors ALPHA at 466-Mhz, with 2 Gbytes and O.S. DIGITAL UNIX 4.0C) of the University of La Laguna. The authors tested their methods using the `tsplib` problem instances

with 1400 customers using different numbers of processing units (2, 4, and 8) and $k_{max} = 15$. As performance measures, running times and speedup factors are considered for SPVNS, while for RPVNS and RSVNS, the number of iterations, running times, and the best objective function values were relevant. As expected, SPVNS achieved almost linear speedup and sometimes even improved the solution quality with respect to sequential VNS. The other two strategies managed to improve the solution quality, with RSVNS being slightly faster (requiring a smaller number of iterations and shorter running times).

The authors in [64] investigated the development of parallel VNS for a bus terminal location problem (BTLP) that incorporates properties of the p-median problem and the maximal covering problem. We refer to the proposed algorithm as PVNS-BTLP. RVNS, exploring k_{rvns_max} neighborhoods, was considered to provide a high-quality initial solution. A low-level (fine-grained) parallelization strategy for BVNS was applied to two most computationally consuming parts of the PVNS-BTLP algorithm - (1) the function computing the first and second closest established terminals for each node, and (2) an evaluation of all possible neighbors within LS, required to find the best solution. LS explores the swap neighborhood, similar to approaches for the p-median problem proposed in [75, 81]. Parallelization is based on explicit search space decomposition and the proposed strategy can be classified as 1L/SE/1C/S/SPSS (HTPM). For the implementation, the C# programming language with .NET Parallel Library (shared-memory) was used on a multi-core processor IntelCore i7-860 2.8 GHz with 8GB RAM memory under the Windows 7 Professional operating system. Consequently, the fork-join pattern and central coordination of parallel loops are also implied. In addition, each iteration processes a fixed set of terminals. Therefore, the ITPM classification is *Fg/SMA/Star/Stat/.NET-TPL*. The stopping condition is set to 1000 successive iterations without improvement. The tuning of parameters k_{max} and k_{rvns_max} was performed by preliminary tests on 6 problem instances. Each given set of parameter configurations was tested by 20 runs with different seed values. After careful analysis, the following values have been selected: $k_{max} = 10$ and $k_{rvns_max} = 0$. By selecting $k_{rvns_max} = 0$, RVNS was not executed, and instead the authors used a randomly generated solution as the initial. PVNS-BTLP is evaluated on two sets of instances: 15 existing LSBTLP instances and 18 new instances derived from TSPLIB `rl` instances. Performance is measured by the average percentage gap (*agap*), the standard deviation (σ), t_{sol}, and t_{exec}. The study employs 5 seeds per instance for literature comparisons and 20 seeds for stability analysis. A controlled comparison of the parallel and non-parallel versions of PVNS-BTLP on all instances using a single seed — both producing identical solutions — yields approximately 3.4× speedup in favor of the parallel version on a quad-core processor. PVNS-BTLP is compared against the only prior work on BTLP and achieves better solution quality on every instance group, with execution time approximately 32 times smaller.

A parallel BVNS for an obnoxious p-median problem (*OpM*) was proposed in [94]. The goal in this variant of p-median is to maximize the sum of minimal distances between customers and their nearest facility. The paper progresses from

optimizing individual components (local search) and sequential BVNS (including sequential multistart BVNS) to parallel strategies (Parallel Multistart) and finally Distributed VNS (DVNS). In particular, one of the contributions in this paper is the efficient sequential BVNS algorithm for the considered problem that includes two types of neighborhood reduction. Based on theoretical results and in order to ensure convergence towards an optimal value efficiently, the authors considered a multistart variant of the proposed sequential BVNS called Multistart VNS. According to Algorithm 3 in [94], $G \cdot V$ independent BVNS algorithms could run in parallel, where G represents the number of groups, while V denotes the number of searches per group. In particular, a total of $G \cdot V$ VNS instances are generated so that a group of V BVNS instances is executed in parallel on a single multi-core processor. In their experimental study, the authors used a single multi-core processor, which means that all $G \cdot V$ BVNS instances run as concurrent threads. Instances within a group run independently without mutual communication (Dec = SC, C = pC), while the groups periodically exchange their best solutions after every R iterations. The best solution in each group is exchanged between the groups via the selected communication topologies. The newly arrived solutions are accepted by the group only if they improve at least the worst solution generated by this group, implementing selective acceptance that constitutes learning behavior (C&L = K). The groups are connected by different communication topologies: fully connected (CT = *FulC*), ring (CT = *Neig*), and master-slaves (CT = *Star*). The proposed DVNS implements a single-level cooperative multi-search approach (L = 1L, Dec = SC). A parallel task runs a complete BVNS algorithm (G = *Cg*) implemented on shared-memory architecture using OpenMP (PA = *SMA*, PPM = *OpenMP*). In addition, all BVNS instances start from different initial solutions, thus Div = *MPSS*. DVNS was coded in C++ and executed on an Intel i5 660 processor, 3.3 GHz with 8 GB of RAM using GNU/Linux. It was compared with the best previous methods, Branch-and-Cut (B&C), XTS, and GRASP, on the selected set of instances. Each algorithm is run 10 independent times per instance, and the performance measure is average success, i.e., how many times the best-known solution is reached. The drawn conclusion is that the proposed parallel DVNS outperformed previous methods, obtaining the best solution in 137 out of 144 instances, while consuming only 10% of the CPU time required by the competing algorithms. This is also confirmed by nonparametric statistical tests.

While the authors in [94] focus on the single-objective OpM, its bi-objective variant (Bi-OpM) was considered in [169]. The Bi-OpM aims to locate p facilities in such a way that two different objectives are maximized: the distance between each customer and the nearest facility center and the dispersion among facilities. This problem variant has applications in locating waste or hazardous disposal facilities, nuclear power plants, chemical factories, noisy or polluting services, etc. To address Bi-OpM, the authors proposed a Multi-objective Parallel VNS approach (Mo-PVNS) that implements the Replicated Shaking VNS (RSVNS) strategy from [75] as a parallel extension of a sequential BVNS. The Mo-PVNS employs single-level parallelization (L = *1L*). Starting from the same current Pareto front, each available processor performs the same shake and local search procedures to an incumbent

solution, creating the explicit decomposition of neighborhood exploration (Dec = *SC*). More precisely, multiple shaken solutions are generated from the same incumbent solution within the current neighborhood, and each is improved via local search before synchronization. This means that Mo-PVNS is focused on parallelizing the SH step in such a way as to explore more than one random solution in each considered neighborhood, and thus perform search on a wider portion of the solution space. The parallelization of the Mo-PVNS algorithm is realized using Java threads on an Intel Core i5-3470 (3.2 GHz), 8 GB RAM (unknown number of parallel threads). Mo-PVNS can be classified as 1L/SC/1C/S/SPSS (HTPM) and *Mg/SMA/Star/Stat/Java-Threads* (ITPM). The preliminary results are used to tune two input parameters α and $k_{\max}$ on a subset of 8 training instances. The computational results obtained on 72 instances indicate the superiority of Mo-PVNS over state-of-the-art algorithms. In particular, Mo-PVNS clearly outperforms AOLS, DBLS, NSGA-II, and SPEA2, producing on average 248 nondominated solutions per instance—more than twice the number obtained by AOLS and DBLS—and achieving a higher hypervolume. Statistical tests (Kruskal–Wallis and Wilcoxon) confirm the superiority of Mo-PVNS with respect to all considered performance metrics. We should note that Mo-PVNS is executed only once per problem instance, whereas the competing metaheuristics are run 30 times.

In [28], a parallel BVNS (BP-VNS) was developed for the α-neighbor p-center problem (αNpCP) and the α-neighbor p-median problem (αNpMP). The three main segments of the BVNS algorithm were parallelized: initial solution generation, shaking, and local search (see Fig. 1. in [28]). The initial solutions are generated by each thread in parallel and the best among them is selected for additional exploration. To balance between diversity and quality, initial solution is generated in two ways: threads with even ID-s generate completely random initial solutions, while the odd threads use a randomized greedy algorithm to generate an initial feasible solution. All these solutions undergo sequential local improvement before the threads are synchronized and the best solution identified. The parallel shaking procedure assigns r threads to independently perform k random swap operations on the current solution, selecting the best result among r generated solutions. After all threads complete their own shaking, the resulting solutions are compared, and the best is selected for local improvement by parallel LS. To perform LS procedure in parallel, the authors divided the neighborhood of the current solution into equal parts (chunks) by partitioning a set of vertices that are not included in the solution into equal subsets. Each subset is used by a single thread to substitute all active vertices (included in the current solution). These threads are assumed to be evenly balanced by the required computations and synchronized at the end to identify the best obtained neighboring solution. The LS procedure is iterated until no improved solution is found. The Neighborhood Change phase is performed by a single thread, to direct search according to the BVNS algorithm. As stopping criterion, the time limit of 30 minutes is used. The proposed parallel VNS algorithm belongs to the 1L/A/1C/S/SPSS HTPM class, as it runs a single VNS variant with parallelized computations related to the main steps: initial solution generation, SH and LS procedures. The ITPM classification is *Fg/SMA/Star/Stat/C++*. Preliminary tests

were performed on an Intel i9-13900K processor (32 threads, 128 GB RAM) while all main experiments were conducted on an AMD EPYC™ Rome 7532 cluster (24 threads, up to 96 GB RAM). The results show that for αNpCP BP-VNS improved best-known solutions in 18% of OR-library and 50% of TSPLIB instances. For αNpMP BP-VNS found optimal solutions in all 80 tested instances. The experimental evaluation revealed that parallelizing the local search yields a speedup of up to 13.48× over its serial version, with consistent gains observed up to 32 threads.

In [25], a parallel BVNS is developed for the robust capacitated p-median problem (RCPMP) under uncertainty, where customer demands are modeled using a cardinality-constrained set limiting the number of customers with high demand and a knapsack set limiting the total excess demand. To solve RCPMP, the authors develop exact (B&C and B&P) and heuristic (BVNS) approaches. Evaluating a candidate facility location solution under demand uncertainty requires solving a worst-case assignment problem, which is NP-hard. Accordingly, each candidate solution is evaluated by solving a robust generalized assignment problem (RGAP), whose high computational cost becomes a bottleneck when exploring large neighborhoods. To address this, the authors introduce parallelization of BVNS to accelerate the evaluation of multiple candidate solutions. Parallelism is explicitly applied in three VNS phases: (1) Initial solution construction: a sequential VNS-PMP runs for 5 s to produce a starting PMP solution a, after which r threads are launched to evaluate one neighbor from $\mathcal{N}_1(a)$ via a full RGAP solver in parallel; the best result is retained; neighborhood $\mathcal{N}_1$ is defined as the substitution of a single facility included in the solution with a non-included location; (2) Shaking: r threads independently sample and evaluate solutions from the neighborhood $\mathcal{N}_k$ concurrently using the simplified lower bounding procedure ($LB2_{\text{rounded}}$), which provides a fast lower bound estimate of the solution quality by rounding distances to integer values; the best solution is forwarded to local search; (3) Local search: $\mathcal{N}_1$ is explored in parallel, where threads evaluate candidate moves concurrently and the search is restarted as soon as an improving solution is identified. Threads are created once at startup and persist throughout the entire VNS loop. A shared cutoff value, updated via solver callbacks, prevents threads from expending computational effort on non-improving RGAP evaluations. The efficiency is additionally improved by a hash table that stores previous results and closed-form expressions that allow for rapid feasibility checks. The overall HTPM classification of VNS-PMP is *1L/SE/1C/S/SPSS*, where the neighborhood structure is explicitly partitioned among threads, a single master thread controls the VNS flow and selects the best result after each parallel phase, and all threads operate on the same incumbent solution. C++ threads are used for implementation. Each thread evaluates one neighbor solution (*Fg*) on a shared-memory architecture (*SMA*), with a central communication pattern (*Star*), and predominantly dynamic load balancing (*Dyn*). Computational experiments on benchmark instances with up to 400 nodes show a substantial reduction in the average optimality gap compared to the sequential configuration and enable exploration of a larger number of distinct solutions. However, no multiple independent runs with different random seeds are reported, limiting the assessment of solution variability and reproducibility.

5.2.2. Routing problems. The Swap-Body Vehicle Routing Problem (SBVRP), a generalization of the well known Vehicle Routing Problem, is considered in [175]. The goal of SBVRP is to minimize total costs, including fixed costs for the use of vehicles and the costs needed to realize the routes. The authors proposed sequential and parallel GVNS heuristics for this problem. In the proposed GVNS heuristics, three neighborhoods exploring intra-route moves and eight based on inter-route moves were used. The strategy for parallelization of GVNS is Replicated Shaking VNS ([75]) that performs four SH+VND steps simultaneously, each assigned to one of the four available CPU cores. All simultaneous shaking procedures use the same neighborhood index k as the input. A constructive heuristic based on a cluster-first, route-second approach is used to generate the initial solution, which becomes the first incumbent. In subsequent iterations, the incumbent is updated whenever a better solution is found. However, being a stochastic procedure, each SH independently generates a different perturbed solution from the current incumbent. The four solutions provided by VND are compared and the best among them is used to potentially improve the incumbent. If the incumbent solution is improved, the search process is restarted from it and the neighborhood index is updated accordingly. The CPU time limit of 600 s is used as a stopping criterion, with $k_{max} = 3$. The proposed parallel GVNS can be classified as 1L/SC/1C/S/SPSS (HTPM) and $Cg/SMA/Star/Stat/C$ (ITPM). Both sequential and parallel GVNS were executed on an Intel i7-4900MQ CPU (2.80 GHz, 16 GB RAM) using four CPU cores, each was run once within the time limit, and results were reported as single values rather than averages over multiple runs. The quality of the proposed methods is evaluated on the instances provided by the organizers of VeRolog Solver Challenge 2014. From the reported results, it follows that the proposed GVNS heuristics are highly competitive compared to the existing approaches to SBVRP. Among the two GVNS variants, the parallel GVNS finds better solutions on all but one instance and is on average approximately 40 seconds faster.

In [146], a parallel VNS for the VRP with divisible deliveries and pickups was proposed. The author implemented a cooperative parallel VNS (CVNS) that utilizes asynchronous cooperation with a centralized information exchange strategy (C&L=A). In particular, the parallelization strategy assumes multiple GVNS instances running in parallel and cooperating (Dec=SC) through a central repository. Each process starts from different initial solutions, generated heuristically with varied parameters, and runs the same VNS algorithm (Div=MPSS). The collegial data exchange is performed after the completion of all local search procedures corresponding to medium-grained parallelization. CVNS was implemented on a shared-memory multi-core workstation equipped with two Intel Xeon E5420 processors at 2.50 GHz with four cores each and 8 GB of RAM, of which six cores were used for parallelization. This is useful as the communication topology follows a star pattern, where all the slaves send their improved solutions to the central coordination mechanism (and receive improved information from it). There is no dynamic load redistribution (LB=*Stat*), while the code is implemented in MATLAB R2013b and Visual C++ 2010 on a Windows platform. When the number of non-improving tours reaches a specified value p_{max}, a perturbation of the

global best solution is applied to escape local optima; the algorithm stops after $s_{max} = 500 \cdot p_{max}$ non-improving tours. The perturbation strategy is important for achieving MPSS diversity due to the need to maintain diverse search trajectories. The resulting HTPM classification is 1L/SC/pC/A/MPSS, and for ITMP we have *Mg/SMA/Star/Stat/MATLAB-Visual-C++*. The experimental evaluation covers three CMT instance sets containing 84 instances in total, the SCA-CON set of 40 instances, and 12 MON-GAL instances; each instance was solved once, with no multiple runs reported. It shows that the CVNS approach outperforms the existing perturbation-based sequential VNS with respect to both solution quality and CPU time. For 10 out of 28 test instances, CVNS provides better solutions, while for the remaining instances, the same solutions are obtained within almost half the time.

In [98], the authors study a VNS algorithm for the Swap-Body Vehicle Routing problem (SBVRP) to determine whether the order of different neighborhood operators influences the performance of the algorithm. SBVRP is a variant of VRP that allows for a reconfiguration of vehicles with respect to the number of semi-trailers attached. Among the available solution transformations, the authors examined seven in the local search phase and identified a subset of well-performing neighborhood operators that are implemented in the VNS algorithm. They also investigate a parallel variant of VNS (Par-VNS) for the same problem. In HTPM, the parallel approach exhibits a single-level implicit cooperative decomposition (HTPM=1L/SC/1C/S/SPSS). The parallelization strategy is based on the *Star* communication model: four copies of VNS (each on a separate CPU core) with the same parameters are executed in parallel (ITPM: *LB=Stat*). Each thread runs a complete VNS instance potentially from different randomly generated initial solutions (ITPM: *G=Cg*). The central thread enforces communication at regular time-based intervals controlled by a synchronization parameter *synctime*, with the best-performing setting corresponding to an exchange every 2 seconds. After the communication is completed, the central thread collects the found solutions and determines the best (ITPM: *CT=Star*). The best solution is then propagated to all peripheral threads. Par-VNS was tested on 18 problem instances provided by the VeRoLog Challenge organizers, which represent real-world problems from the vehicle routing domain. The experimental evaluation is conducted on Intel Xeon X5650 2.66 GHz with 24.0 GB of RAM with 30 repetitions for each instance (*PA=SMA*). However, information about the programming language or the parameter tuning process is not provided. The comparison was made against four existing approaches and the authors' *Random VNS* variance using the gap with respect to the best known solution as the performance measure. The provided results show that parallel VNS outperforms other algorithms by generating four new best known solutions and equaled 11 other instances, although slightly degraded solution quality is observed for three test instances. Moreover, the analysis demonstrates that frequent synchronization and information exchange between parallel VNS instances is preferable to longer periods of independent execution.

In [105,106], the authors examined the performance of cooperation-based strategies in parallel metaheuristics with respect to non-cooperative approaches. The main goal was to investigate how cooperation between processes affects the

performance of parallel GVNS executed on a shared-memory multiprocessor system. The well-known constructive Savings Algorithm of Clarke and Wright was used for initial solution generation. In the local improvement phase, VND uses three neighborhoods that explore widely used transformations, i.e., 2-opt (Intra-route), Swap (Inter-route), and Relocate (Inter-route). Two parallel GVNS models applied to the capacitated vehicle routing problem (CVRP) were proposed in [106]. To ensure that both models have the same available resources during the execution, one thread was used as a solution manager, i.e., shared memory to which all other threads have access. The current best solution is kept and maintained by this thread. The first model, non-cooperative (NCM), involves independent execution of several identical GVNS threads until the stopping criterion is fulfilled. The second model, managed cooperative model (MCM), is based on asynchronous information exchange between GVNS threads via the solution manager - a dedicated thread (warehouse) that maintains a pool of solutions. Namely, each thread repeatedly executes SH+VND steps until an improvement of the current best solution is obtained. This solution is communicated to the warehouse and is compared with the existing solutions. If the warehouse contains a better solution, the thread adopts it to continue the search. According to HTPM, the first model can be classified as 1L/SI/pC/A/SPSS, while ITPM classification is *Cg/SMA/—/Stat/Python*. For the second model, HTPM=1L/SC/pC/A/SPSS and ITPM=*Mg/SMA/Star/Stat/Python*. Both parallel GVNS algorithms were implemented in Python 3.7 and run on an Intel Core i9 7940X CPU (3.50 GHz) and 32GB RAM at 3333MHz. As a stopping criterion, the maximum number of iterations is set. The following values for the stopping criterion were examined: 5, 10, 20, 30, 40, 50, 100, 200, and 300 iterations. Each variant of the algorithm and each stopping criterion were tested ten times on the existing benchmark instances of CVRPs. The presented results show that MCM outperforms NCM on hard instances, while NCM yields better results on easy instances.

In [105], an additional cooperative strategy (the parameterized cooperative model - PCM) for GVNS was proposed and compared with the implementations presented in [106]. The same initial solution construction method and neighborhood structures are used. The maximum number of threads was 28, one of which was playing the role of solution manager. The solution acceptance criterion is not based on the solution quality, but on the normalized distance between the new solution generated by a thread and the solution stored by the manager thread. If the distance is greater than a given parameter θ, the candidate solution is accepted by the manager, otherwise, the thread will continue search from its own best solution. In addition, the value of θ is changed during the search to ensure the intensification of cooperation toward the end of the search process. The classifications of PCM are as follows. HTPM=1L/SC/pC/K/SPSS and ITPM=*Cg/SMA/Star/Stat/Python-Manager*. The experimental evaluation is conducted using 76 benchmark instances from the CVRP library (http://vrp.atd-lab.inf.puc-rio.br), comprising instances with known optimal solutions and instances from the Golden set for which only best-known values are available. The stopping criterion is defined as the maximum number of GVNS iterations, upon reaching the known optimum where applicable,

or when the time limit of 3,600 seconds is exceeded. Notably, the experimental evaluation includes ten repeated runs per configuration, which is commendable and places this paper among the minority of surveyed works that properly account for the stochastic nature of parallel VNS. The results show that PCM achieves the lowest overall average error, with its advantage over NCM growing by 3.2% on hard instances and 14% on the Golden set. NCM outperforms MCM and PCM on easy instances, and MCM consistently under-performs both on all but the very hard instances. Algorithm dominance is closely related to instance difficulty. In addition to a description of original parallel VNS-based algorithms, [105] contains a review of the literature that considers parallel VNS implementations proposed in [39, 49, 51], and particularly the implementations developed for various types of VRP [10, 31, 145, 146].

In [185], two parallel strategies are proposed for the Covering Salesman Problem (CSP): synchronous master-slaves parallel VNS (SMPVNS) and asynchronous cooperative parallel VNS (ACPVNS). The Covering salesman problem (CSP) is a generalization of TSP that requires to find a minimum length Hamiltonian cycle in an undirected graph such that the vertices that are not in the tour must be covered by at least one visited vertex. The authors proposed a bi-level formulation of CSP (BCSP) by combining leader and follower sub-problems and used it to design parallel GVNS approaches. For the leader sub-problem, three neighborhoods (*N1*: extract and invert one node, *N2*: extract and invert two nodes, *N3*: extract and invert three nodes) are used in the local improvement phase of GVNS, while 3-opt neighborhood is explored when considering the follower sub-problem. In the shaking procedure, k vertices are selected randomly and their assignment variables are inverted. The potential infeasibility of the resulting solution is resolved by adding new vertices in a greedy way. SMPVNS exhibits a single level single control parallel decomposition (L=1L, C=1C). It divides the neighborhood into segments and assigns each segment to a peripheral process. All of them start from the same current global-best solution sent by the central process and independently explore the assigned segment (Dec=SE, Div=SPSS, G=*Cg*, LB=*Stat*, CT=*Star*). The central process waits for all others to complete before adopting the best local optimum as the current best solution, without any learning mechanism (C&L=S). The search continues until the stopping condition is fulfilled. ACPVNS represents a single-level algorithm that operates through explicit search space decomposition, with decentralized control (L=1L, C=pC). Implicit decomposition (Dec=SC) is realized using the segment identifier (seg_i), where each thread starts from the same solution (current global best), explores the assigned segment of the search space, and communicates asynchronously with other processes only if a better solution is found (C& L=A). On the implementation level, ACPVNS is a coarse-grained shared-memory implementation with static load balancing, utilizing C++. The algorithms are implemented on a 2.2 GHz Intel Core I5-5200U CPU (2 cores); the number of parallel processes is not explicitly stated. SMPVNS and ACPVNS were compared with two existing algorithms (SN and HACO) on 75 benchmark instances divided into three sets (small-, medium-, and large-sized). All stochastic algorithms were repeated 5 times on each instance. The comparison results show

that ACPVNS performs slightly better than SMPVNS. For small- and medium-sized instances, PVNS is competitive with HACO while being significantly faster. For large-sized instances, PVNS clearly dominates, finding the best solutions in 23 out of 27 instances (85.2%) compared to only 2 out of 27 for both SN and HACO.

A Smart and Parallel GVNS (SP GVNS) for the pollution-routing problem is proposed in [116]. This problem is a variant of the Green VRP, which includes costs associated with fuel, drivers, and greenhouse gas emissions. Nine neighborhood structures are explored, five of them influencing the order of customers in routes and four related to changes in speed of vehicles operating on one or more routes. The "smart" feature intensifies the shaking phase by increasing the number of consecutive perturbations within the same neighborhood structure if no improvement is obtained in a given number of iterations. The neighborhood index for shaking is incremented only when the maximum number of consecutive perturbations is exceeded. The parallelization strategy applied is Replicated Shaking VNS from [75]: starting from the incumbent solution, shaking and local search (VND) are executed in parallel. All parallel blocks start from the same incumbent solution and use the same neighborhood index k, and the best solution found across blocks is used to potentially update the incumbent (Div=SPSS, C&L=S). A single master thread controls the overall VNS execution and synchronizes the parallel blocks (C=1C, CT=*Star*). The parallelization operates at the level of SH+VND steps, corresponding to medium-grained granularity. The proposed SP GVNS can therefore be classified as 1L/SC/1C/S/SPSS (HTPM) and *Mg/SMA/Star/Stat/Java-Threads* (ITPM). SP GVNS was coded in Java version 8, executed on a PC with an Intel 8th Gen Core i7-8565 processor, 16 GB of RAM using 10 threads. Ten runs were conducted for each instance, and the stopping criterion was set at tmax=5 seconds. Comparisons were performed with GVNS, S GVNS, ALNS, and ILS-SP-SOA on instances with 10, 50, and 100 customers. The presented computational results show that SP GVNS performs best among the VNS variants and outperforms ALNS for all instance sizes. Compared to ILS-SP-SOA, SP GVNS achieves better results for small instances (10 customers) but obtains slightly higher costs for larger instances (50 and 100 customers). However, it requires more time than the other approaches.

A multi-depot multi-compartment simultaneous pickup and delivery problem with time windows and split demands (MDMCSPDTWSD) was considered in [26]. The problem combines several known optimization problems to model complex real-world requests. The multi-compartment vehicle was designed to separate goods of different types that should not be stored together. The routing of these types of vehicle (MCVRP) already represents a challenge. The multi-warehouse vehicle routing problem with time constraints (MDVRPTW) is another component of the considered problem, while the simultaneous pickup and delivery (SPD) request introduces additional complex constraints and diverse requirements. However, the joint consideration of all sub-problems enables to generate solutions that ameliorate both efficiency and quality of service. The authors provided MILP formulation of (MDMCSPDTWSD) and proposed sequential and parallel improved VNS (P-IVNS) approach. The improvement includes the introduction of a novel directed shaking method that explores nine neighborhoods used also in a VND phase. The

initial solution is generated by a newly proposed high-quality heuristic algorithm. Two stopping criteria are adopted: a maximum number of total iterations and a maximum number of non-improving iterations, both defined as a function of instance size. A shared-memory single level (L=1L) parallelization is applied to VND. The master thread runs sequential VNS, controls the whole execution C=1C, divides each neighborhood into a given number of sub-neighborhoods (Dec=SE) to be searched by other threads in parallel. Synchronization of threads is rigid (C&L=S), as master thread must wait for all others to complete their search to identify the best obtained local optimum solution. Div=SPSS, having in mind that all threads start from the same solution obtained by shaking and try to improve it in the different parts of the same neighborhood. Regarding the implementation issues and ITPM, we can conclude only that LB=*Stat*, CT=*Star* and G=*Fg*, as hardware and programming model are not specified. The experimental evaluation was conducted on 20 large-sized instances that authors generated by modifying the existing benchmarks for MDVRPTW. The reported results indicate superiority of P-IVNS over VNS and ALNS with respect to solution quality; however, the power of utilized computer resources remains unknown.

5.2.3. Scheduling problems. In [178], HGA, a hybrid algorithm incorporating GA and parallel GVNS was proposed to minimize total tardiness in the flexible job shop. The crossover and mutation operators of GA are used to adapt the structure of chromosomes to the properties of the problem. Parallel GVNS is used to enhance the best individuals within the elitist selection phase, before inserting them into the next generation. In the LS phase, to increase the solution quality, GVNS explores three neighborhoods: move an operation to an alternative machine, change the order of the two selected operations on the same machine, and change the order for two pairs of operations. The parallel GVNS algorithm was executed with 20 threads to minimize the execution time. The Java programming language was used to code the algorithm that was executed on an Intel i7 processor (2.30 GHz) with 4 GB memory, running under Windows 7 operating system. The proposed parallel GVNS can be classified as 1L/SI/1C/S/MPSS and *Cg/SMA/FulC/Stat/JavaThreads*. The experimental evaluation is conducted on 14 representative problem instances, each executed 20 times per due date type, reporting the best value, average, and coefficient of variance (CV) across runs. The results are compared with plain GA, DE algorithm from the literature, and several constructive heuristic algorithms. Provided results show that HGA demonstrates superior performance with respect to the solution quality over all compared methods, as confirmed by a paired t-test at a significance level of 95%.

The authors in [130] addressed the min–max order batching problem, which combines order-to-batch assignment with route evaluation, minimizing the maximum retrieving time across all batches. They proposed a sequential (GVNS) algorithm and its parallel variant based on the RSVNS strategy from [75]. The purpose of the parallelization strategy is to increase diversity in the search process rather than to decrease the computation time. The iterations are parallelized in such a way that all p threads independently apply the shake procedure to the same initial solution (Dec=SC, Div=SPSS). Then, each new solution is improved with the VND

procedure, and the threads are synchronized at the end of this procedure, where the single master thread determines the best among p solutions, after which it runs the NC procedure (C=1C, CT=*Star*) to decide if another parallel iteration should be performed. The workload is statically balanced as each thread executes the same fixed amount of work. The algorithm is classified as 1L/SC/1C/S/SPSS (HTPM) and *Mg/SMA/Star/Stat/Java-Threads* (ITPM), implemented in Java 7 on an Intel Core i7 with 2.5 GHz and 4 GB of RAM (*SMA*). The algorithm is run once (no repetitions) and evaluated via: CPU time, Dev% (percentage deviation from best-known), and #Best. Preliminary experiments on 14 representative instances address four stages: constructive procedure selection, LS implementation selection, improvement strategy selection (VND vs. isolated LS), and k_{max} tuning. Higher values of k_{max} consistently yield better results, but at a proportionally higher computing time, leading to $k_{max} = 5$ as a compromise. The stopping criterion, defined by t_{max}, is set to match the computing time of the previous approaches. As the number of threads increases to 8, improvement in Dev% and #Best is reported. Therefore, the RSVNS configuration with 8 threads was compared against the constructive heuristic and exact B&B in the final large-scale experiment. To compare results, Friedman's test is used. RSVNS finds 369 and 374 best solutions (out of 432) under the two routing strategies, whereas the exact branch-and-bound method achieves 141 and 172 best solutions. The constructive heuristic produces significantly inferior results.

The flexible job shop scheduling problem with assembly operations is considered in [71]. In this two-stage problem, parts are first processed in a flexible job shop and then assembled into final products, with the objective of minimizing the makespan. A new algorithm combining PSO and parallel VNS (HPSOPVNS) is proposed. In HPSOPVNS, PSO is used for global exploration of search space, while PVNS has to perform local improvements of solutions generated by PSO. In each iteration, the best-obtained solution is improved by PVNS. Five neighborhood structures are explored in PVNS: NS1 replaces a randomly selected element with its preceding neighbor, or its succeeding neighbor if no predecessor exists, NS2 moves a randomly selected operation to another machine, NS3 combines the first and the second neighborhood transformations, NS4 swaps operations between the least- and the most-loaded machines, and NS5 swaps a sequence of operations between two randomly selected parts. NS1, NS2, NS4, and NS5 are used in the VND part, while NS3 is employed in Shaking. Parallelization is realized by independent execution of several identical GVNS copies, each starting from the same current global best solution and independently performing shaking and local search, after which the master process selects the best result. The maximum number of iterations is used as the stopping criterion. HPSOPVNS is classified as 1L/SC/1C/S/SPSS (HTPM) and *Mg/SMA/Star/Stat/MATLAB* (ITPM). The Taguchi method is applied to tuning the parameters of metaheuristic algorithms. The comparison is performed with a hybrid between PSO and (sequential) VNS (HPSOVNS), PSO, and hybrid between GA and TS (HGATS) on randomly generated test instances divided into three size categories (small, medium, large), with ten independent runs per instance. The experimental results show that the HPSOPVNS algorithm provides

the same results as other algorithms within shorter execution time on small instances. For medium- and large-sized instances, HPSOPVNS outperforms others with respect to solution quality. The Least Significant Difference (LSD) statistical test is performed to confirm the superiority of the proposed approach for medium and large instances.

In [149,150], the authors address a long-term open-pit mine production scheduling under geological uncertainty with the objective of maximizing the cumulative net present value (NPV). The solution approach is a hybrid of VND and linear programming (LP). VND employs three neighborhood searches by perturbing an initial solution to find better candidates: exchange (E), which swaps two blocks between adjacent periods; shift-after (SA), which moves a block along with all its successors from one period to the next; and shift-before (SB), which transfers a block and all its predecessors from one period to the previous one. The algorithm cycles through SB→E→SA, starting from an initial solution and terminating when no improving move is found. In each neighborhood search, for every time period the algorithm evaluates all candidate blocks to check if they represent a feasible improvement without violating capacity and precedence constraints. A TEST subroutine was developed for this purpose, which evaluates the objective function value of each candidate move without permanently altering the data structures, thus enabling efficient candidate evaluation. The EXECUTE subroutine then actually applies the best improving move to update all data structures related to incumbent solution. Parallelization is applied exclusively inside the VND to accelerate the most time-consuming component - the evaluation of candidate moves. Candidate moves are partitioned among worker threads, each evaluating a subset of moves independently on the same incumbent solution (Dec=A, Div=SPSS, LB=*Stat*). A centralized master thread maintains the control (1C) over the VNS cycle (CT=*Star*). In particular, threads synchronize via a barrier before EXECUTE selects the best solution (C&L=S). At the implementation level, parallelism occurs a the level of individual move evaluations inside TEST (G=*Fg*). The architecture is shared-memory (PA=*SMA*) using native C++ multi-threading (24 threads) on Intel® Xeon® Gold 5118 processor (PPM=*C++ Threads*). Results are fully deterministic and invariant with respect to the number of threads, i.e., parallelism reduces execution time while maintaining solution quality. No stopping criterion based on time or iterations is used. Instead, the algorithm terminates when, after an exhaustive search, no neighborhood improves the solution. The parallelization yields significant speedups while preserving deterministic behavior and solution quality.

5.2.4. Optimization problems on graphs. Parallel VNS strategies for the cutwidth minimization problem (CMP) are systematically developed in [67]. The authors comprehensively explore the design space by proposing six distinct parallelization strategies. The strategies differ in communication mechanisms and in the granularity of VNS parallelization. The base algorithm is the Variable Formulation Search (VFS), a new VNS variant specifically applied to cutwidth minimization. Building upon the known literature, the parallel strategies are grouped into three templates w.r.t. which VNS stage is parallelized: (1) Replicated Parallel VNS

(RP-VNS) - indicates parallelization of the whole VNS method; RP-VNS executes multiple complete VNS instances independently from diverse initial solutions with RP-VNS1 running them fully independently until termination, while RP-VNS2 synchronizes threads after each iteration by broadcasting the best-found solution as the new starting point for all threads, (2) Replicated Shake VNS (RS-VNS) - parallelizes the shaking and local search phases under a single level single control strategy, where threads evaluate candidate solutions in parallel, (3) Cooperative Neighborhood VNS (CN-VNS) - refers to the exploration of predefined neighborhoods in parallel. For each template, two variants of the parallel algorithm are developed, which differ in communication mechanisms. RP-VNS1 represents an independent multistart where the master thread waits until all the others complete their execution to determine the best solution. Threads start from different initial solutions and perform an identical search procedure. We classify this approach as 1L/SI/-pC/A/MPSS (HTPM) and *Cg/SMA/—/Stat/Java-Threads* (ITPM). RP-VNS2 includes frequent communication between threads (after each iteration), whereas the best solution at the end of each iteration becomes a new initial point for all the threads. The HTPM classification differs significantly from the earlier variant, i.e., 1L/SC/1C/S/SPSS, and *Mg/SMA/Star/Stat/Java-Threads* (ITPM). Replicated Shaking (RS-VNS) consists of multiple threads performing shake and local search procedures in parallel, each starting from the same current solution. RS-VNS1 is implemented as a best improvement strategy, which means that the master thread waits for all other threads to finish before selecting the best solution. On the other hand, RS-VNS2 represents a first improvement strategy, i.e., the best solution is updated as soon as one thread finds an improvement (LB=*Dyn*). The corresponding HTPM classification for RS-VNS1 is 1L/SC/1C/S/SPSS and ITPM is *Mg/SMA/Star/Stat|Dyn/Java-Threds*. Finally, CN-VNS strategy focuses on cooperative exploration of different neighborhoods. The threads in CN-VNS1 explore different neighborhoods cooperatively. In particular, each thread starts from the same solution, randomly selects the neighborhood for shaking, and performs shaking and local search. These two steps (SH+LS) iterate as long as there are improvements in the solution, and then the best solution obtained is communicated to the master thread. The remaining threads wait for the master thread to identify the new overall best solution and wake up all others to continue their search. CN-VNS1 is classified as 1L/SC/1C/S/SPSS according to HTPM. In contrast, CN-VNS2 threads systematically explore k_{max} neighborhoods by evenly partitioning the whole neighborhood space. More precisely, each thread calculates the indices of the first (k_{first}) and last (k_{last}) neighborhoods, and iterates shaking and local search as long as it manages to improve the current solution. The current index for shaking (k) is updated using the values between k_{first} and k_{last} and standard VNS rules. According to HTPM, CN-VNS2 is classified as 1L/A/1C/S/SPSS, as the explicit partition of the neighborhood index range $[1, k_{max}]$ into disjoint sub-ranges constitutes algorithm-level decomposition rather than search space decomposition. Regarding ITPM, both CN-VNS1 and CN-VNS2 can be classified as *Mg/SMA/Star/Stat/OpenMP*. Parameter tuning has not been performed. It appears that each algorithm was run once per variant configuration, with metrics - average deviation and number of best

solutions - aggregated across all problem instances. The non-parametric test was used to confirm the quality of the obtained results.

The authors of [27] considered the Connected Subgraph Optimization Problem (CSOP), which consists of finding an extremal subgraph for a given objective function. The main idea was to propose a general-purpose approaches that allows to change objective functions with minimal interventions in the basic algorithm. Having in mind the complexity of problems related to the optimization on graphs, they proposed to address it by the parallel VNS. The Fractal, general-purpose graph pattern mining system [62] was used to execute independent execution of several VNS copies. Master thread is used for initiating the execution, collecting and evaluating the results. This means that the classification of this approach, that we refer to as FracVNS, is HTPM=1L/SI/1C/S/SPSS and *MC/SMA/Star/Stat/Java*. Initial solutions for different copies are determined by the stochastic heuristic, which are then explored by BVNS algorithms in parallel. The experiments were conducted on Intel(R) Core(TM) i9-14900KF 6GHz (32 cores), 64GB RAM, under Debian GNU/Linux 12 (Linux kernel 6.1.0-28-amd64). Six well-known real-world datasets were used to evaluate the performance of FracVNS. An extensive computational analysis is performed, considering the quality of solution, running time requirements, scalability (i.e., speedup with respect to the number of threads), among other performance measures.

5.2.5. Assignment-allocation problems. In [69], a parallel VNS (PMOVNS) was proposed for the multi-objective sustainable post-sales network design problem. Several copies of the VNS algorithm are executed by the corresponding number of threads to explore the solution space. Each thread performs one complete VNS iteration that consist of shaking and stochastic VND (SVND) that incorporated 9 types of neighborhood structures. The search is performed until a stopping criterion, set to 100 iterations, is reached. The resulting parallel VNS was executed using 4 threads with an additional thread performing search coordination and control. The central thread collects and selects the global best solution and redistributes it to all threads as the starting point for the next iteration. PMOVNS can be classified as 1L/A/1C/S/MPSS (HTPM) and *Mg/SMA/Star/Stat/C++* (ITPM), executed on a 5-core 2.0 GHz with 4 GB RAM. Comparisons are conducted on 7 randomly generated instances of increasing size. Two performance measures were used in the comparison: the average (over 10 repetitions) number of Pareto-optimal solutions and the average ratio of Pareto-optimal solutions. In addition, the required CPU time was analyzed. The authors reported the superiority of the proposed parallel VNS with respect to all three performance measures. It is compared with two MIP solvers: sequential LINGO and parallel Gurobi (using 4 threads), sequential multi-objective Memetic Algorithm from the literature, and sequential variant of the proposed VNS (denoted GMOVNS). PMOVNS significantly outperforms the multi-objective memetic algorithm (MOMA), achieving a much higher average Pareto ratio (78.9% vs. 38.0%) and generating more Pareto-optimal solutions on average (38.81 vs. 17.44) for less computational time. PMOVNS against GMOVNS shows a higher Pareto ratio (72.6% vs. 38.0%) with similar solution counts and lower averge runtime. In the single-objective setting PVNS also

outperforms LINGO and Gurobi on small- to medium-sized instances within substantially shorter time limits.

Sequential and parallel BVNS methods for the dynamic memory allocation problem (DMAP) were proposed in [159] and [160]. The problem is related to the control of current memory state such that the cost of moving a data structure from an external device to memory is minimized. As a parallelization strategy, the *Synchronous Parallel VNS* (SPVNS), originally proposed in [75], is used in [159]. SPVNS employs a multistart scheme in which each iteration independently starts from a randomly generated solution, with the LS phase executed in parallel. The search space is explicitly partitioned into P disjoint subsets of T/P time intervals (Dec=*SE*), with each thread executing the same LS procedure on its assigned subset. A single master thread controls the overall execution (C=*1C*, *Star*): it generates the initial solution, performs shaking, and executes the neighborhood change, while the remaining threads run the parallel LS. All threads depart from the same post-shake solution (Div=*SPSS*) and are synchronized via a barrier after each LS phase (C&L=*S*). The workload is statically balanced by construction (*Stat*), resulting in fine-grained parallelism (*Fg*) as each thread is responsible for a small portion of the search space. The proposed SPVNS is therefore classified as *1L/SE/1C/S/SPSS* (HTPM) and *Fg/SMA/Star/Stat/Java-Threads* (ITPM), implemented in Java SE 7 using Java threads on an Intel Core i7 920 CPU (2.67 GHz) with 8 GB RAM. SPVNS was compared with a variant of GRASP from the literature, and its superiority with respect to both solution quality and running time was confirmed by the Wilcoxon signed-rank test. However, no independent replications with different random seeds have been reported.

In [160], the same authors considered the same problem and proposed the *replicated shaking methodology* (RSVNS) from [75] as a parallelization strategy. This means that SH(k)+LS (for the same value of k) are executed on different CPUs in parallel. A master-slaves cooperation scheme is used to control execution. The master thread executes the main body of the sequential BVNS, while SH+LS are executed on slaves. This means that after shaking in the current neighborhood k and the succeeding LS, the master determines the best among the p solutions and completes the neighborhood change phase. Because all slave threads start from the same incumbent S_{best} the strategy is classified as *1L/SC/1C/S/SPSS* (HTPM), with each thread executing a full SH+LS cycle corresponding to medium granularity (*Mg*). RSVNS have been implemented in Java, using Java threads as parallelization technology with up to 8 threads. The experimental evaluation was conducted on the same machine as in [159]. The ITPM classification is therefore *Mg/SMA/Star/Stat/Java-Threads*. In the experiments, the same 44 instances were used. The experimental evaluation performed in [160], reported on the comparison between RSVNS, a variant of GRASP and iterative metaheuristic (IM) from the literature, and a constructive heuristic. First, the influence of the number of threads on the performance of RSVNS was examined, revealing that increasing the number of threads improves the solution quality within a negligibly longer execution time. Another conclusion is that both BVNS and RSVNS outperform the best performing methods from the literature GRSP and IM. Additionally, RSVNS outperforms

sequential BVNS according to solution quality. However, from the provided data, it is not clear whether the authors reported wall-clock or CPU times. Another interesting fact is that SPVNS and RSVNS have not been compared.

An approach to solving the medium- (MV) and low- voltage (LV) planning of large-scale distribution systems via parallel computing and decomposition techniques is proposed in [158]. The presented mathematical formulation of this complex problem considers the allocation of substations, distribution transformers, MV and LV circuits, support structures, poles, renewable energy sources (RES), and energy storage sources (ESS). In addition, the model takes into account the variable costs related to the operation of RES and ESS, power losses, distribution transformers and sub-stations, energy purchased from sub-stations, and the amount of greenhouse gas emissions. System reliability and maintenance costs of these devices are also considered in the planning. The authors proposed a Cooperative Parallel Variable Neighborhood Decomposition Search (CPVNDS). The CPVNDS algorithm follows a *Star* communication topology: the master thread executes the main body of the search process, while other threads run simultaneously and share information with the master. The master thread generates the initial solution and sends it to the others (C=*1C*, CT=*Star*), updates the incumbent, and checks the stopping criterion. The other threads run in parallel from the same starting point (Div=*SPSS*), independently generate a MV system solution using six neighborhood structures, and improve it via VND over three LV neighborhoods (Dec=*SC*, G=*Mg*). Each thread randomly applies local search decomposition to the most promising moves, balancing solution quality and search guidance (LB=*Stat*). The stopping criterion is a predetermined number of iterations. CPVNDS was coded in the C++ programming language with OpenMP to support parallelization. It was executed on an Intel Core i7-8700 processor with 16 GB of RAM. HTPM classification of CPVNDS is 1L/SC/1C/S/SPSS, while ITPM classification is *Mg*/*SMA*/*Star*/*Stat*/*OpenMP*. The performance of CPVNDS is evaluated on a large-scale distribution system with 200 nodes in MV and 1672 nodes in LV. In the algorithm comparison (Table 2), CPVNDS finds solutions 4.88% better than GVNS and 2.18% better than GVNDS in total planning cost, while also requiring the lowest average CPU time. To evaluate parallelism performance, a total simulation time of 18,000,s is fixed and simulations are performed using 1, 4, 8, and 12 threads, with 10 independent runs per configuration. Results show that configurations with 1 and 4 threads are far from the best solution found within the fixed time budget, whereas 8 and 12 threads perform equivalently well, both converging to the best solution in 80% of runs. The results are additionally validated through three planning case studies, where the most comprehensive scenario (considering all planning alternatives simultaneously) yields up to 22% improvement in system reliability and up to 18% reduction in greenhouse gas emissions costs.

5.2.6. Packing problems. In [180], the order batching problem in manual order picking systems was considered. The main goal is to collect and group the items requested by customers. Items are located in the warehouse and it is necessary to define the order of collection such that the total length of picking tours is minimized. To fulfill real-world restrictions, it is assumed that the picking devices

are heterogeneous and that it may not be possible to transport each customer order by each device. This extension of the standard OBP is called Dedicated Heterogeneous OBP (DHOBP). The authors proposed serial and parallel BVNS approaches, as well as a hybrid between BVNS and a set partitioning problem formulation (VNS-SP). The parallel variant VNS-IMS performs the sequential BVNS independently using eight threads witout any cooperation during the execution (Dec=*SI*, C=*pC*, C&L=*A*), each running a complete BVNS instance (*Cg*) on a shared-memory platform (*SMA*), all starting from the same deterministic C&W(II) initial solution (Div=*SPSS*), with statically balanced workload and no inter-thread communication (*Stat*, —), implemented in C++. When the termination criterion is met, the best-found solution is selected among the completed tasks resulting in a *1L/SI/pC/A/SPSS* (HTPM) and *Cg/SMA/—/Stat/C++* (ITPM) classification. The stopping criterion combines a maximum wall-clock time with a maximum number of iterations without improvement. The computational experiments cover OBP benchmark instances from the literature, newly proposed DHOBP instances, and real-world instances from a large-scale e-commerce warehouse. For stochastic algorithms, three independent replications are performed and average results reported. The results indicate that independent multi-start parallelization without cooperation is unlikely to yield substantial benefits.

5.2.7. Portfolio optimization. The application of parallel VNS to optimization in the financial domain is considered in [5]. More precisely, the cardinality constrained portfolio optimization problem that consists in selecting the best among the available investment alternatives in such a way to maximize the profit and minimize the risk. Cardinality constraints are introduced to limit the number of alternatives, and they transform the quadratic optimization model into the mixed-integer quadratic programming problem (QP). In [5], the authors proposed a parallel BVNS algorithm combined with the exact solver for QP (VNS+QP). The main idea is to use BVNS to select portfolio (the combination of assets) and provide this partial solution to an exact solver to calculate the proportions of assets. Parallelization is obtained by running VNS+QP independently and asynchronously on different processors (Dec=*SI*, C=*pC*, C&L=*A*), each assigned a different value of the risk-return trade-off parameter λ. As λ directly shapes the landscape of the objective function, threads not only start from different initial solutions but effectively optimize different instances of the problem, resulting in simultaneous construction of different points on the efficient frontier (Div=*MPDS*). No communication or information transfer between threads takes place (—), each thread executes a complete independent VNS+QP instance (*Cg*) with statically balanced workload (*Stat*), on a shared-memory platform (*SMA*). The HTPM classification is therefore *1L/SI/pC/A/MPDS* and ITPM is *Cg/SMA/—/Stat/MATLAB*. Performance is tested on five well-known datasets with the stopping criterion set to a maximum number of $N \times 1000$ evaluations. Parameter tuning experiments are conducted with 5 independent replications; however, the final comparative results in Tables 3–5 are aggregated across 50 different values of λ without independent replications per λ value, making it impossible to assess the variability of the reported results due to the stochastic nature of VNS. The comparison conducted

shows that VNS+QP outperforms the state-of-the-art algorithm on almost all seven performance measures and performs very well on low risk portfolios, i.e., for high values of λ.

5.2.8. Final remarks. In Table 3 we summarize a chronological overview of publications that propose SMA parallel VNS-family algorithms. In addition to the year of publication and the corresponding bibliographic number, we specify algorithms' acronyms, the corresponding base sequential and parallel algorithms, as well as the addressed optimization problems. The surveyed publications span the period from 2000 to 2025 and include 27 research studies on parallel Variable Neighborhood Search algorithms implemented for shared-memory architectures. After a pioneering study in 2000 and a decade-long dormant period, the field experienced sustained growth from 2013 onward, with particularly productive years in 2017 and 2020 (5 papers each), driven by the widespread adoption of multi-core processors. As can be seen, the surveyed publications address diverse optimization problems, with routing problems being the most numerous (8 publications, 36%), followed by location problems (5 publications, 23%) and scheduling problems (3 studies, 14%). GVNS emerges as the most common base algorithm (appearing in 12 publications), followed by BVNS (8 studies), reflecting the field's evolution toward more general and flexible VNS frameworks. Overall, within the set of shared-memory parallel VNS algorithms a clear pattern emerges: most approaches rely on a star communication topology combined with static load balancing. The choice of parallel programming model is diverse, spanning OpenMP, Java Threads, C++, Python, and MATLAB, reflecting the diversity of implementation environments in which parallel VNS has been developed.

5.3. Distributed memory parallelization of VNS. Distributed-memory parallelization played a fundamental role in the early development of parallel VNS implementations, largely due to the widespread adoption of MPI as a protable and accessible parallel programming model. This type of PA offers scalability and the ability to harness large-scale computational resources across multiple independent nodes. Unlike shared-memory approaches that are typically limited by the number of cores on a single machine, distributed-memory implementations have the potential to scale to hundreds or thousands of processing units. In Figure 13 we examine the distribution of optimization problem categories across the 23 parallel VNS publications that utilize distributed-memory parallelization architecture (PA = DMA). DMA is the second most frequently used PA appearing in 40.4% of the classified parallel VNS literature. Unlike the SMA subset where Routing and Location problems dominate, the DMA distribution reveals a different problem profile, as shown in Figure 13. The bar chart shows that DMA is most frequently used for Scheduling and Routing problems, which together account for 65.2% of the DMA publications (39.1% and 26.1% respectively). Location and Scientific Computing problems each account for 13% of the distributed parallel VNS literature. Finally, Graphs and Packing each represent only 4.5% of the publications.

In the remainder of this subsection, publications are reviewed per problem category in chronological order of publication.

TABLE 3. Chronological summary of SMA parallel VNS-based algorithms.

Year	Ref	Parallel Algorithms	Base Algorithm	Problem
2002	[75]	SPVNS, RPVNS, RSVNS	BVNS	p-median problem
2013	[69]	Parallel VNS	RVNS	Multi-objective sustainable post-sales network design
2015	[178]	HGA (Parallel GVNS)	GVNS	Flexible job shop scheduling
2015	[159]	SPVNS	BVNS	Dynamic memory allocation problem
2016	[64]	PVNS-BTLP	BVNS	Bus Terminal Location Problem (BTLP)
2016	[67]	RP-VNS1, RP-VNS2, RS-VNS1, RS-VNS2, CN-VNS1, CN-VNS2	VFS (VNS variant)	Cutwidth minimization problem (CMP)
2017	[98]	Par-GVNS	GVNS	Swap-Body Vehicle Routing Problem (SBVRP)
2017	[130]	Parallel GVNS	GVNS	Min–Max order batching problem
2017	[146]	CVNS	GVNS	VRP with divisible deliveries and pickups
2017	[160]	RSVNS	BVNS	Dynamic memory allocation problem
2017	[175]	Parallel GVNS	GVNS	SBVRP
2020	[106]	NCM, MCM	GVNS	Capacitated Vehicle Routing Problem (CVRP)
2020	[5]	VNS+QP	BVNS	Cardinality constrained portfolio optimization
2020	[71]	HPSOPVNS	GVNS	Flexible job shop with assembly operations
2020	[94]	DVNS, Parallel Multistart	BVNS, Multistart BVNS	Obnoxious p-median problem (OpM)
2020	[105]	NCM, MCM, PCM	GVNS	CVRP
2021	[158]	CPVNDS	GVNS, GVNDS	MV and LV planning of distribution systems
2022	[169]	Mo-PVNS	BVNS	Bi-objective Obnoxious p-median (Bi-OpM)
2022	[185]	SMPVNS, ACPVNS	GVNS	Bilevel Covering Salesman Problem (BCSP)
2023	[116]	SP GVNS	GVNS	Pollution-routing problem
2023	[180]	VNS-IMS	BVNS	Order batching problem
2024	[28]	Parallel BVNS	BVNS	α-neighbor p-center problem
2024	[149, 150]	Parallel VND	VND	Long-term open-pit mine production scheduling
2025	[25]	Par. VNS-PMP	BVNS	Robust capacitated p-median problem (RCPMP) under uncertainty
2025	[26]	P-IVNS	GVNS	Multi-depot multi-compartment simultaneous pickup and delivery problem with time windows and split demands (MDMCSPDTWSD)
2025	[27]	FracVNS	BVNS	Connected Subgraph Optimization Problem (CSOP)

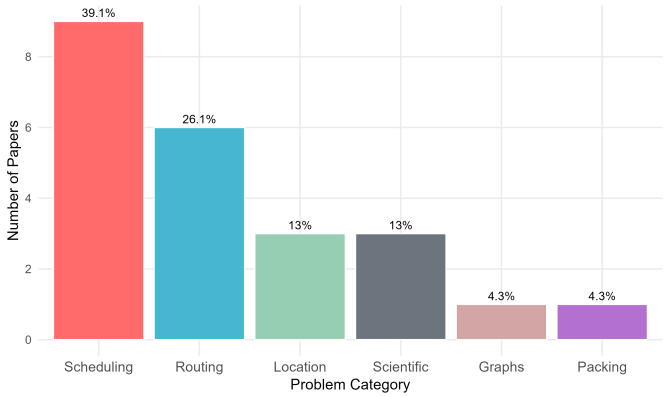


FIGURE 13. Distribution of parallel VNS publications employing distributed-memory PA across optimization problem categories.

5.3.1. Location problems. The p-median problem has been used for the evaluation of parallelized BVNS algorithms proposed in [37–39]. Besides the independent run proposed in [75], an asynchronous centrally coordinated parallelization strategy has been proposed. Several copies of the BVNS algorithm cooperate by asynchronously exchanging the locally determined best-so-far solutions. The main idea was to preserve the simplicity of the original sequential BVNS, i.e., to concentrate the search in the vicinity of the globally best-so-far solution. It was implemented by engaging a number of processors to perform BVNS with First Improvement Fast Interchange LS and shaking in the same neighborhood with $k_{max} = p$. An additional processor was used to simulate the global memory by collecting solutions from other processors, evaluating them, and distributing the global best solution to others through the star communication topology. This strategy was named Cooperative Neighborhood VNS (CNVNS) and can be classified as 1L/SC/pC/A/MPSS and *Mg/DMA/Star/Stat/MPI*. To generate initial solutions, each processor executes RVNS with $k_{max} = 2$ and the stopping criterion defined as 1000 iterations without improvement. The parameter tuning was not extensive; the values are determined by preliminary tests. Communications are initiated in one of the following two cases. In the case where an improvement of the locally best-so-far solution is obtained, it is sent to "global memory" to check if it is the new globally best-so-far solution. The second case to initiate communication is if k_{max} is reached without improvement. In both cases, the new globally best-so-far solution communicated by "global memory" is adopted by the corresponding processor. CNVNS is coded in Fortran77. Computational experiments are performed on a 64-processor SUN Enterprise 10000 with 400 MHz clock and 64 gigabytes of RAM. For the experimental evaluation, on p-median benchmark problem instances from TSPLIB with up to 1000 medians and 11948 customers. The stopping criterion was defined as the maximum allowed execution time (t_{max}) and was reduced with respect to the number of processes involved in the search. As a performance measure, the authors

adopted the deviation of solution quality obtained by CNVNS from the best known solutions, and The results showed that, for a given time limit, CNVNS was able to find better solutions than sequential BVNS. In addition, the authors compared CNVNS with some other approaches proposed in the literature. The presented results were comparable with the best existing methods, indicating that parallelization is a promising avenue for addressing large-scale optimization problems.

5.3.2. Routing problems. The early parallel VNS implementations for DMA considered a Traveling Purchaser Problem (TPP) and can be found in [59,138,139]. The authors used sequential and parallel GRASP, VNS, and their hybrids to tackle this problem. Here, we focus only on the VNS-related part of the study. The TPP represents a generalization of the Traveling Salesman Problem (TSP). A purchaser (traveler) has to visit a subset of the available markets and collect (purchase) all of the required items in such a way that the total cost of traveling and purchasing is minimal. Depending on the procedures applied to generate the initial solution and search the solution space, 48 variants of the VNS algorithm were tested. In the parallelization of these algorithms, load balancing was the main goal. Both static and dynamic load balancing strategies were considered.

In [138] multiple parallel implementations of GRASP and VNS were presented and evaluated for TPP on an IBM SP/2 cluster with 6 processors using C programming language and MPI communication library. Initially, the authors compared static and dynamic load balancing strategies, as well as *Star* and *FulC* communication topologies, and concluded that static load balancing and *Star* produced better results in the same amount of time. In addition, the results of three parallel strategies are reported: (1) ParVNSDrop - all processes execute the same VNS algorithm with DROP algorithm to generate initial solution and HybridSearch within static load balanced local search, implementing independent multi-search with decentralized control. Thus HTPM = 1L/SI/pC/A/SPSS, ITMP = $Cg/DMA/-/Stat/MPI$; (2) ParVNS - processes employ different construction method for generating the initial solution, but the same HybridSearch local search within the VNS algorithm; and (3) ParGRASPVNS - a hybrid approach in which some processes run GRASP and others a VNS variant. All three algorithms have the same ITPM, the subtle difference may be recognized in the high-level taxonomy as parVNS and ParGRASPVNS utilize different search (DS) methods, unlike ParVNSDrop. The three proposed parallel BVNS variants were compared among themselves and their sequential counterparts on two test instances within the stopping condition defined as 3000 iterations. From the presented results, the authors concluded that parallelization improves the solution quality with negligible reduction in the running time.

Building upon [138], in [139] the authors propose three more advanced VNS-based parallel algorithms to deal with TPP: VDif, VEqual and GVDif. These algorithms extend the previous work by introducing some sophisticated features related to load balancing and communication coordination. They are applied to five algorithms, of which three are related to VNS. VDif goes beyond ParVNS's approach by varying both construction and local search within different VNS variants run on different processors (HTMP = 1L/SI/pC/A/MPDS). VEqual executes identical VNS procedures on all processors (HTMP = 1L/SI/pC/A/MPSS), similar

to ParVNSDrop, but with dynamic load balancing through auxiliary control processes. GVDif represents a hybrid between GRASP construction and the VNS improvement phase on each processor (HTMP = 1L/SI/pC/A/MPDS), which fundamentally differs from ParGRASPVNS. For the implementation, the authors used the same hardware and software resources and conducted preliminary experiments on 5 randomly generated test instances. They compared the developed algorithms using 4 and 8 processors, within a stopping criterion equal to 2000 iterations and by performing 3 repetitions of each algorithm on each instance. The best results were produced by GVDif, which managed to improve the solution quality in most of the cases while achieving almost linear speedup and efficiency close to 1.

The same authors extended their study in [59]. The focus of this study is the exploration of dynamic load balancing. As in the previous studies, two types of CT were considered, *Star* and *FulC*; however, the processors are assumed to be heterogeneous. The idea is to assign more computations to the faster processors. Dynamic load balancing is achieved by iterative assignments. In the beginning of the search process, only a portion of the computations (a given number of BVNS iterations) is assigned to all processors equally. In the *Star* model, communication is initiated upon the completion of the assigned number of iterations. Dynamic load balancing assumes that the distribution of computational load is performed during the search. More precisely, when the assigned load is completed, the central processor sends a new portion if available. In the *FulC* model, each processor executed two processes, the main (p_i) that performs computations related to GRASP and/or BVNS and the associated process (q_i) that is responsible for controlling the number of iterations assigned to p_i and the termination condition of the entire distributed algorithm. The computational load of q_i is quite small and does not influence the CPU time required by p_i. The dynamic load balancing is implemented in the cooperative way by using q-processes to redistribute the computations among processors. When a process p_i completes its computations, the corresponding q_i process selects a process q_j randomly and asks for more iterations to be assigned to p_i. If this selection fails, other q-processes are communicated iteratively. In the case where there are no iterations to be completed, q_i sends a termination indicator to p_i. Three parallel models were compared for both GRASP and BVNS: (1) ParInd - the distributed algorithm exploring static load balancing. Classifications for this model are HTMP = 1L/SI/pC/A/SPSS, ITMP = $Cg/DMA/—/Stat/MPI$; (2) ParDist - the distributed algorithm with dynamic load balancing to be executed on a fully connected PA. This model is classified as 1L/SI/pC/A/SPSS for HTPM and $Mg/DMA/FulC/Dyn/MPI$ for ITPM; and (3) ParMW - dynamic load balancing on a star PA, classified as 1L/SI/1C/A/SPSS for HTPM and $Mg/DMA/Star/Dyn/MPI$ for ITPM. For each model, two variants of GRASP, two variants of BVNS, and a hybrid GRASP+BVNS variant were implemented. GDiff and VDiff assume that each processor executes different variants of sequential GRASP (BVNS), while in GEqual and VEqual, all processors execute the same variant of GRASP (BVNS). Finally, in the GVDiff implementation, each processor executes different variants of GRASP+BVNS. The experimental evaluation was conducted on the same hardware as in [138, 139] using 4 and 8 processors.

However, the results were not in favor of dynamic load balancing because the multiprocessor system used contains homogeneous processors.

Two cooperation schemes, based on the central memory mechanism for Skewed GVNS (SGVNS) parallelization, were proposed in [145]. They were tested on the Multi Depot Vehicle Routing Problem with Time Windows. The set of 12 neighborhoods is used and divided into two parts, the first 6 neighborhoods consider routes from a single depot, while the remaining neighborhoods consider routes starting in different depots. SGVNS allows the acceptance of non-improving solutions with a quality defined by an additional parameter. In both cooperation schemes, the extension of the parallelization strategy proposed in [39] was used. Each slave had to search through a certain number of neighborhoods. In the fine-grained cooperation scheme (HTPM=1L/SC/pC/A/MPSS), the search in a single cycle did not necessarily include the entire set of neighborhoods. The resulting algorithm is denoted FG-CSGVNS. However, in the coarse-grained cooperation scheme (CG-CSGVNS), the number of iterations performed by each slave before the information exchange was significantly higher than the number of neighborhoods. The authors proposed to make the cooperative scheme adaptive by adjusting search parameters during execution. This adjustment resulted in a 1L/SC/pC/K/MPSS HTPM classification of CG-CSGVNS. For the experimental evaluation, both cooperative SGVNS were executed on up to 32 search threads and compared with 32 independent runs of the sequential procedure. The efficiency of both cooperation schemes was verified by the runtime scalability.

In [134], the dial-a-ride problem (DARP) is considered with an attempt to design an efficient algorithm for the large-sized instances. DARP consists of n requests for transportation that have to be served by m vehicles of a given capacity Q. For each request, the source and destination locations are given, as well as the soft time windows of the pickup and delivery. The goal is to minimize the total transportation cost defined as the sum of costs of all routes and penalties for violating the pickup times, ride durations, and capacity. The authors proposed a distributed and parallel BVNS algorithm (dVNS) that allows each search process to run on a partition of the search space. An initial solution is defined by the set of initial requests that are clustered into k (the number of processing units) disjoint parts by the k-medoid algorithm. Based on a *Star* communication pattern, these parts are distributed over all the available processing units. The processing units generate routes that fulfill the assigned subset of requests and improve them by the sequential VNS algorithm. At the pre-defined synchronization points, the processing units send generated routes to the central processor, which joins them to complete a solution and calculates the objective function value. Re-clustering of routes is performed by the central processor according to the proximity criterion, and the new set of clusters is distributed to the slaves for additional improvement cycle. The algorithm runs until a stopping criterion is fulfilled. dVNS was implemented in the C++ programming language with the OpenMPI communication library. In the experimental evaluation, dVNS was executed on 4, 8 and 16 processing units and compared with sequential VNS on several sets of large-sized synthetic test instances, generated to simulate real demands in the city of San Francisco. Parameter tuning

was performed using the Taguchi method. Based on the experimental results, the authors concluded that dVNS exhibits superior performance compared to its sequential counterpart.

The parallel algorithm P-ILS-RVND, developed to effectively explore the solution space for vehicle routing problems with simultaneous pickup and delivery, is presented in [167]. It combines Iterated Local Search (ILS) with the newly proposed Random Variable Neighborhood Descent (RVND) in which the order of the neighborhoods is selected at random in each run. ILS is used for solution perturbation to enable escape from local optima, while RVND applies 6 inter-route neighborhood operators randomly to refine solutions. The study focuses on distributed-memory parallelization. In particular, a *Star* communication model is explored on a high-performance computing cluster with 256 CPU cores. P-ILS-RVND is divided into three main parts, the first two related to parameter tuning of the number of vehicles and the cost weighting factor gamma. The third part is focused on the optimization phase. The central node manages the overall search process, including initial solution generation, solution evaluation, and coordination. Other processes are responsible for the parallel execution of local search and neighborhood evaluation tasks. Communication among CPU nodes is handled via MPI. The first and second phases execute rapidly, and each involves a single message being transmitted to the central processor. The third phase is the most critical. The communication overhead here is influenced by three factors: the number of perturbations applied, the total number of iterations performed, and the number of cores used in the computation. The number of cores affects both the number of iterations and the volume of messages exchanged. As the core count increases, the central process receives an increasing number of requests. P-ILS-RVND was coded in C++ and executed on a cluster with 32 SMP nodes, where each node consists of two Intel Xeon 2.66 GHz quad-core processors that share 16 GB of RAM and run under Linux CentOS. The nodes are connected via the Gigabit Ethernet network. The P-ILS-RVND algorithm was tested on 72 problem instances, comparing performance when using 128 and 256 processing units. The computational results show that P-ILS-RVND improved the best-known solution in 15 cases and matched the best results in 51 instances, demonstrating strong competitiveness, particularly for large-scale problems. With respect to running time, an increase in speedup was noted with doubling the number of processing units. The authors analyzed also the communication overhead, reporting its value below 15% of the total execution time.

5.3.3. Scheduling problems. Parallel GVNS algorithms for Job Shop Scheduling problems were proposed in [13, 163]. Local improvement phase involves search of two neighborhoods, swap and insert. For the parallelization, identical copies of GVNS are run on different processors. Four parallelization strategies were taken into account: (1) synchronized cooperative strategy as proposed in [75]; (2) asynchronous centrally coordinated method from [39]; (3) Noncentral parallelism via unidirectional ring topology; (4) Noncentral parallelism via bidirectional ring topology. The later two strategies, classified by HTPM as 1L/SC/pC/A/MPSS, were proposed for the first time in [163]. In Noncentral parallelism via unidirectional ring topology, each processor was executing a single pair of SH+LS, sending the

obtained result to the succeeding processor, and collecting the result generated by the preceding processor. The newly arrived solution become an initial point for the next execution cycle regardless of its quality. This strategy increases the diversity of the search process. However, the overall best solution is always kept in the memory of each processor. Noncentral parallelism via bidirectional ring topology differs from the previous strategy in the following: as the initial point for the next execution cycle, each processor selected the best out of three solutions (the one resulted from its own computations and two solutions received from the adjacent processors). The parallel VNS algorithms were coded in Java SDK 1.4 and the communications are conducted via TCP/IP protocol. According to ITPM, the third parallel VNS variant can be classified as *Mg/DMA/Neig/Stat/TCP/IP*, while the fourth variants differs only in the CT field, which takes the value *Mesh*. For an experimental evaluation, a local area network of PCs equipped with Intel Pentium IV 2.6 GHz processor and 256 MB memory was used. The benchmark problem instances were selected from OR Library. At the beginning of the experiment, the best out of 7 VNS variants was identified to be used in parallel implementations using 2, 5, 10 and 20 processors. Four parallel variants are referred to as Syn, Asyn, Ring, and Mesh. The experimental investigation showed that unidirectional ring topology outperformed the others with respect to solution quality. The comparison between parallel and sequential VNS is described in [13].

Parallel VNS for the Car Sequencing Problem was developed in [110]. The algorithm was named Parallel Efficiency Guided VNS (PEGVNS) and represents parallel GVNS with Time Restricted LS (TRLS) incorporated into Randomized VND. Here, randomized means that the order of the neighborhoods was not fixed. The authors used 5 different neighborhoods and experimentally evaluated their efficiency with respect to both solution quality and running time. These efficiencies are used in the randomization process. Several iterations of TRLS in different neighborhoods were performed in parallel, then the processes were synchronized, the best solution was identified and propagated to the next TRLS phase. The described parallel VNS falls into the 1L/SC/1C/K/SPDS HTPM class. PEGVNS was implemented in C++ using EAlib 2.0, also implemented in C++. EAlib 2.0 is a generic library for metaheuristics developed at the Institute of Computer Graphics and Algorithms at the Vienna University of Technology. For parallelization, the MPI library was used on a *Star* communication model. All tests were performed on two Dual Core AMD Opteron(tm) 270 Processors with 2 GHz and 8 GB RAM in total. Therefore, ITPM=*Mg/DMA/Star/Dyn/MPI*. Computational tests showed that a substantial reduction in computation time was achieved. Because no "perfect" neighborhood ordering can be identified in advance, the parallel self-adaptive approach was proposed to obtain good solutions.

A parallel GVNS algorithm (PGVNS-FJSP) was used in [184] to increase exploration of the search space for the Flexible Job Shop Problem (FJSP). This problem is an extension of the classical job shop scheduling problem and consists of two main steps. The first is to assign each operation to one of the available and compatible machines, while the second needs to find the best ordering (sequencing) of operations assigned to each machine to minimize the makespan. The proposed algorithm

consists of external and internal loops. The external loop controls the stopping condition of the algorithm. In the internal loop, a number of processors was used to perform a single run of SH+LS in different neighborhoods, independently, in parallel. Six neighborhood structures with different parameters have been explored in PGVNS-FJSP, such that 9 possible transformations were used in the local improvement phase. According to HTPM, this strategy is classified as 1L/SC/1C/S/SPDS. Due to synchronized parallelization, shaking was always applied to the current best solution. The PGVNS-FJSP algorithm was implemented in C++ programming language and run on a PC with 2.0 GHz and 1.0 GB of RAM memory exploring 5 processors. The parameters were tuned by preliminary tests and set to the following values: $k_{max} = 4$, the number of VND iterations $n_{max} = 350$, and the stopping criterion was defined as 1000 GVNS iterations. Comparisons were made with TS, GA, and hybrid GA (hGA) from the literature. The computational results on 181 FJSP benchmark instances show the competitiveness of the proposed algorithm to similar methods from the relevant literature. It was also noted that the parallel search enabled an improvement in the solution quality for some of the instances.

In [114], the parallel neighborhood search (PNS) strategy executed on two processors is applied to the problem of order acceptance and scheduling (OAS) in the flow shop environment. The authors considered a multi-objective variant where the goal is to simultaneously minimize makespan and maximize total net revenue. The proposed PNS for OAS starts from two randomly generated solutions, the first of which is improved through three types of neighborhood: swap, insert, and exchange, while for the second, the exchange and swap neighborhoods are explored, both in that particular order. This actually means that two different VND procedures (searched according to the BI strategy) are executed on different processors independently for a given number of iterations. After a predefined number of iterations has been completed, the incumbents of two search procedures are exchanged. The stopping criterion is defined as the maximum number of iterations. All non-dominated solutions are stored in the set Ω that represents the Pareto front, used for the comparison of PNS with existing approaches from the literature, based on the VNS and TS heuristics. PNS is coded in Microsoft Visual C++ 7.0 and the experiments are conducted on CPU Pentium PC with 2G RAM 4.0 G. The three proposed neighborhoods have been incorporated into the sequential GVNS for OAS, used for comparison with PNS. In addition, PNS is compared with sequential TS from the literature. The experimental evaluation was performed on randomly generated test instances with $n = 20, 30, 50, 100, 200$ jobs and 5 or 10 machines. For evaluating the stability, PNS is run 30 times with different seed values. Parameter tuning is performed by the preliminary test on a subset of instances involving 27 combinations of parameter values. Three performance metrics have been applied to compare the results of Pareto fronts generated by different algorithms: the distance metric, the SP metric, and the cardinality of Pareto front Ω. The reported computational results show that PNS provides a better Pareto front than GVNS and TS on most of the instances and requires less computation time.

In [51], the authors considered parallelization of the LS procedure to be executed on distributed-memory multiprocessor systems. Several parallel LS variants

were proposed, developed, and compared for a Multiprocessor Scheduling Problem with Communication Delays (MSPCD). The parallel LS procedure with the best performance identified in [51] was incorporated into the BVNS framework, resulting in a parallel BVNS variant named PVNSPLS. This variant is classified as 1L/SE/1C/A/SPSS and *Fg/DMA/Star/Stat/MPI*, and represents the sequential VNS in which a speed up was obtained by parallelizing the most computationally intensive part, the LS procedure. A similar approach was used as the first strategy proposed in [75]. The PVNSPLS was implemented in C programming language with MPI communication library and executed on a SUN Enterprise 10000 multiprocessor system containing 64 processors, each with 400MHz clock and 64Gb of RAM. For parallelization of LS, up to 20 processors were used. The experimental evaluation of such a fine-grained parallel BVNS showed that both the quality of the final solution and the running time were improved when parallel LS was executed on a modest number of processors (up to 10). Increasing the number of processors involves larger communications delays and yields the degradation in the performance of parallel VNS.

Various strategies for parallelization of VNS for distributed-memory multiprocessor architectures are proposed in [47,49,50]. These strategies are based on the asynchronous cooperation of several search processes. Various levels of parallelization were tested, from low level parallel neighborhood exploration [51], through medium level asynchronous execution of basic VNS steps (SH+LS) [47], to high level asynchronous cooperation of various sequential VNS algorithms [49,50], the variants of BVNS and GVNS in particular. In addition, centralized and non-centralized information exchange were compared. Parallel algorithms were implemented on multiprocessor systems consisting of a given number of identical processors connected via the following two topologies: star configuration for centralized information exchange and processor ring for the non-centralized communication. The proposed strategies were implemented in C programming language with MPI communication library for MSPCD. The following variants of parallel VNS were implemented:

- IVNS - independent execution of various sequential BVNS and GVNS algorithms. The HTPM and ITPM classifications for this strategy are 1L/SI/pC/A/MPDS and *Cg/DMA/—/Stat/MPI*
- DVNS - a strategy similar to RSVNS from [75], except that shaking is performed in parallel for different values of k. This strategy is classified as 1L/SE/1C/A/MPSS and *Mg/DMA/Star/Stat/MPI*
- CVNSall - asynchronous cooperative execution of different VNS variants with centralized information exchanges where communications are initiated by slaves after each SH+LS phase is completed. The classifications are 1L/SE/pC/A/SPDS and *Cg/DMA/Star/Stat/MPI*
- CVNSallK - the same as previous strategy but with communications initiated after the whole VNS iteration is completed, i.e., at the points when k_{max} is reached. Classifications are the same as for CVNSall
- CVNSring - asynchronous cooperative execution of different VNS variants with non-centralized information exchanges initiated after each SH+LS phase is completed. A processor sends its current best solution to its next

pear, then collects the current best solutions received from the pervious peer, and continues the search from the better of two "best" solutions. This strategy is classified as 1L/SE/pC/A/MPDS and *Cg/DMA/Neig/Stat/MPI*.

Experiments in [49,50]were conducted on an SGI Altix system containing 384 Intel Itanium2 dual core processors running at 1.6 GHz for a total of 768 cores. The system also has 1.5 TB of memory, 2 GB per core. The performance of parallel searches on up to 20 processors was compared with sequential execution on benchmark problem instances from [48] containing up to 500 tasks. The authors concluded that all cooperative variants outperformed IVNS, i.e., each tested sequential VNS, with respect to both solution quality and computational efficiency. On average, CVNSring offers the best performance among the cooperative methods for the problem considered.

5.3.4. Packing problems. The authors in [148] investigated the disjunctively constrained knapsack problem (DCKP). In this variant of the problem, there exists a set of incompatibilities that prevent some pairs of items from being packed together. The aim of DCKP is to maximize the total profit of the items packed in the knapsack such that the capacity is not exceeded and that all the selected items are compatible. The authors proposed a PNS strategy with the aim to benefit from multiple neighborhood exploration: instead performing sequential BI VND, different neighborhoods are explored in parallel on different processors within PNS. The search control is coordinated by the central thread (manager), while the others (team members) perform the search in the assigned neighborhood. The information exchange is similar to that proposed in [166]. The manager keeps the pool of best solutions collected from the other threads. In an asynchronous way, team members communicate to the manager their best solution and receive the current best solution for additional local exploration. PNS was coded in C++ enriched with the MPI library. All testes were conducted on Intel Xeon, 2× 3.06 Ghz with 6-Core. The performance of the proposed method was evaluated on the set of standard benchmark instances. For small-sized instances, the results were compared with exact solvers, while for large-sized instances, the deviations from the best-known solutions were used as the performance measures. It has been concluded that PNS provides high quality solutions within reasonable running time. In addition, the performance scalability with respect to the number of threads is reported for up to 400 threads.

5.3.5. Scientific computing. A Cooperative Parallel Decomposition Guided VNS (CPDGVNS) was proposed for the weighted Constraint Satisfaction Problem (CSP) in [142]. A sequential Decomposition Guided VNS (DGVNS) [73] has been designed for problems that can be decomposed into an acyclic graph of clusters by constructing a tree decomposition [157]. The decision variables connected to the currently explored clusters are released and are the subject of the main VNS steps. The remainder of the decision variables are fixed to their current values. In such a way, DGVNS operates iteratively on different subproblems defined by the graph of clusters. CPDGVNS represents a straightforward parallelization of DGVNS and assumes that each cluster is explored by a separate processor. In a

Star communication topology, CPDGVNS requires the number of peripheral processes to be equal to the number of available clusters. The central process generates the initial solution and declares it as the current best, sends the current best solution and the corresponding cluster to be explored to each peripheral process, communicates (asynchronously) with other processes and updates the current best solution, and verifies the stopping condition to determine the content of the next communication. If the stopping condition is satisfied, the central process informs the others to finish execution. Otherwise, the new data is sent to continue the search. Upon communicating the results of its search, each peripheral process waits for a message from the central process. If the message does not contain an information that the stopping condition is satisfied, another iteration of DGVNS on the corresponding cluster is executed and the locally-found best solution is sent to the central process. The DGVNS iteration means that all neighborhoods from k_{min} to k_{max} are explored at least once. It should be noted that the structure (type of neighborhood) is unique, only distances between solutions change during the search. On the other hand, each process is allowed to make changes only on the assigned cluster, which means that an object of parallelization is the VNDS variant of VNS. This parallelization strategy can be classified as 1L/SC/1C/A/MPDS according to HTPM. CPDGVNS was implemented in C++ using the toulbar2 and MPI libraries. Therefore, ITPM=*Cg/DMA/Star/Stat/MPI*. Experimental evaluation was conducted on benchmark instances of three optimization problems: Radio Link Frequency Assignment Problem (RLFAP), Remote sensing image classification problem (SPOT5) involving composing packages of photos to download from satellite, and Single Nucleotide Polymorphism tagging in genetics (tagSNP). The comparison has been made between CPDGVNS and DGVNS with stopping condition set to 1 hour of CPU time and 50 repetitions for each instance. For each instance, reported are the number of successful runs (found optimal solutions), the average CPU time (in seconds) for the successful runs, and the average cost of the final best solution over the 50 runs. The authors concluded that parallelization enables to increase the number of successful runs and to reduce the required CPU times.

Two parallelization strategies for DGVNS were proposed in [143]. The authors noted the main limitation of CPDGVNS proposed in [142], which is a quite rare communication. The workload of peripheral processes is large as they have to perform a number of SH+LS steps before sharing their best solutions with the central process. This fact increases the diversification of the overall search process; however, it intensifies the search performed by each process. Therefore, in [143] the authors proposed replicated synchronous and asynchronous DGVNS and named the resulting parallel algorithms Replicated Synchronous DGVNS (RSDGVNS) and Replicated Asynchronous DGVNS (RADGVNS). The first difference between these algorithms and CPDGVNS is the communication that is performed after each SL+SL phase. The second difference includes *the intensified shaking procedure* that incorporates the release of randomly chosen k variables in the solution that are then determined in the best possible way. In RADGVNS, the communication between the central and peripheral processes is asynchronous, the same as in CPDGVNS,

it is just more frequent. On the contrary, during the execution of RSDGVNS the central process waits for all others to complete SH+LS, collects all solutions, determines the best among them, compares it with the current best, performs an update (if necessary), and broadcasts the current best solution to other processes. The authors employ a Linux cluster consisting of eight nodes, each equipped with dual Intel Xeon E5-2650 processors (16 cores per node), 64 GB of RAM, and a clock speed of 2 GHz. The VNS parameter values were not tuned for the particular set of benchmark instances. The classifications of these two algorithms differ from CPDGVNS, in the following fields: Div=SPDS, G=*Mg* for both algorithms, C&L field for RSDGVNS takes the value S. The authors tested new algorithms on instances from [142], and the presented results show that RADGVNS is more efficient and effective, providing better speed-ups and solution improvements. RADGVNS's asynchronous communication reduces overhead and better exploits parallelism, making it more scalable with increasing processors. RSDGVNS shows improvements over purely sequential implementations, however, is to an extent limited by waiting times at synchronization points.

The parallelization strategy proposed in [166] deals with the execution of various metaheuristic algorithms (TS, VNS, DE, and PSO) on different CPU nodes utilizing the Algorithm Portfolio (AP) approach and Solution Trading [165] via MPI. In particular, the AP approach is a highly adaptive framework in which multiple metaheuristic algorithms may be executed in parallel. The single process coordinates execution via *solution trading*. Solution trading is an adaptive trading scheme where coordinated processes periodically send their best-found ('elite') solutions to a controller process. When any of them stalls, the controller sends elite solutions to it with the aim to kick-start its potential progress. The processes independently explore their part of the search space using the assigned metaheuristic and periodically report improvements. Due to dynamic allocation of runtime, the AP approach is balancing exploitation of strong performers and exploration by weaker or diverse strategies. In addition, it is robust to changes in metaheuristic parameters. As a result, AP outperforms single-algorithm approaches in both running time and success rate. In this particular case, the authors of [166] considered several optimization problems related to Circulant Weighing Matrices (CWMs). Parallelization of these algorithms in the AP approach is straightforward, it explores a multi-trajectory or multi-population framework. In addition, it allows for the cooperation of different search processes by the information exchange to increase efficiency. A *Star* communication model is adopted, such that each process executes a sequential algorithm and these algorithms communicate asynchronously via the controller. The controller keeps a pool of M elite solutions, M being the number of controlled processes. If an improved solution is generated, the solution in the pool corresponding to that process is replaced. At some predefined search points (1000 non-improving iterations for BVNS), the process requests the overall best solution from the pool to diversify its search. All experiments were conducted on the glacier cluster of the WestGrid consortium (http://www.westgrid.ca), using 8 and 16 nodes. The implementation was based on the OpenMPI project (http://www.open-mpi.org/).

Based on the provided comparison results, AP-VNS did not exhibit a good performance, although the winning method happens to be the other trajectory-based metaheuristic, TS.

In [141], the problem of probabilistic graphical model design is considered. This problem is NP-complete, and therefore, the sequential and parallel VNS-based approaches were proposed. Actually, the methodology that the authors used for parallelization is VNS/LDS+CP [117], a hybrid between VNS and Limited Discrepancy Search with Constraint Propagation (LDS+CP), which is a partial tree search method. The variant of VNS explored is VNDS, where the decomposition is implemented through the definition of neighborhoods. The neighborhood $\mathcal{N}_k$ contains all subsets of k decision variables, while the neighbor of a solution x in $\mathcal{N}_k$ contains all solutions with $n-k$ variables with the same assigned values and k variables with different assigned values. The search in neighborhood $\mathcal{N}_k$ consists of un-assigning values to k variables and then restoring them in all possible ways searching for an improvement. LDS+CP with and without discrepancy limit is used for solution reconstruction. Without discrepancy limit LDS+CP becomes an exact search. The authors proposed unified decomposition guided VNS (UGDVNS) and its parallel variant UPDGVNS that we are interested in. The prefix "unified" denotes various combinations of two complete and incomplete search methods. UPDGVNS represents an asynchronous cooperation of several UDGVNS processes with centralized communication controlled by one of the processes. During the communication, the global best solution is exchanged among processes. The experimental evaluation was conducted on a cluster of 48-core AMD Opteron 6176 at 2.3 GHz and 384 GB of RAM within CPU time limit of one hour. UPDGVNS can be classified as follows: HTPM=1L/SC/1C/A/SPSS and ITPM=*Mg*/*DMA*/*Star*/*Dyn*/*MPI*. A subset of hard test instances not being solved to optimality with the existing exact and heuristic approaches is selected and used to compare the results of UPDGVNS with several sequential and parallel algorithms from literature. For the parallel execution, 10 and 30 processing units were used. The provided comparison results show that UPDGVNS outperforms its sequential counterpart (which is already significantly better than the approaches from literature) with respect both solution quality and wall-clock running time. Increasing the number of processors does not significantly influence the performance. In addition, a computational protein design problem has been taken as an interesting case study and experimental results show superiority of UPDGVNS over the existing methods.

5.3.6. Final remarks. Table 4 presents a chronological overview of 23 parallel VNS publications implemented on distributed-memory architectures, spanning from 2001 to 2017. The research timeline reveals an early burst of activity (2001-2004) with 6 publications [38, 39, 47, 59, 138, 139] establishing foundational parallelization strategies, predominantly based on *Star* topology, utilizing MPI for communication and C/Fortran as programming languages. In fact, the *Star* topology remains the dominant communication pattern among the DMA studies. Among the problem-specific contributions, the MSPCD problem appears in four publications spanning 11 years [47, 49–51] making the MSPCD group the longest sustained research effort in parallel VNS history. Similarly, the p-median problem is addressed

TABLE 4. Chronological summary of DMA parallelization strategies for VNS.

Year	Ref	Parallel Algorithms	Base Algorithm	Problem
2001	[138]	ParVNSDrop, ParVNS, ParGRASPVNS	BVNS	Traveling Purchaser Problem (TPP)
2001	[37]	CNVNS	BVNS	p-median problem
2002	[139]	VDif, VEqual, GVDif	BVNS	TPP
2002	[59]	ParInd, ParDist, ParMW	BVNS	TPP
2003	[38]	CNVNS	BVNS	p-median problem
2004	[39]	CNVNS	BVNS	p-median problem
2004	[47]	DVNS	BVNS	Multiprocessor Sched. Prob. with Comm. Delays (MSPCD)
2007	[163]	Syn, Asyn, Ring, Mesh	GVNS	Job Shop Scheduling problem
2008	[145]	FG-CSGVNS, CG-CSGVNS	SGVNS	Multi Depot Vehicle Routing Problem with Time Windows
2008	[110]	PEGVNS	GVNS	Car Sequencing Problem
2008	[13]	Syn, Asyn, Ring, Mesh	GVNS	Job Shop Scheduling problem
2010	[167]	P-ILS-RVND	RVND	Vehicle Routing Problem with simultaneous pickup and delivery
2010	[184]	PGVNS-FJSP	GVNS	Flexible Job Shop Problem (FJSP)
2012	[49]	PVNSPLS, DVNS, CVNSall, CVNSallK, CVNSring	BVNS	MSPCD
2013	[50]	IVNS, DVNS, CVNSall, CVNSallK, CVNSring	BVNS, GVNS	MSPCD
2014	[142]	CPDGVNS	DGVNS	Weighted Constraint Satisfaction Problem (CSP)
2015	[51]	PVNSPLS	BVNS	MSPCD
2015	[143]	RSDGVNS, RADGVNS	DGVNS	CSP
2015	[114]	PNS	VND	Order acceptance and scheduling (OAS) in flow shop
2015	[134]	dVNS	BVNS	Dial-a-ride problem (DARP)
2016	[166]	AP-VNS	BVNS	Circulant Weighing Matrices (CWMs)
2017	[148]	PNS	VND	Disjunctively constrained knapsack problem (DCKP)
2017	[141]	UPDGVNS	VNDS	Probabilistic graphical model design

in three publications from 2001-2004 ([37–39]). Notably, BVNS emerges as the dominant base algorithm in the DMA subset (appearing in 12 publications). GVNS appears in 8 publications, VND is a subject of DMA parallelization in three studies, while we have identified only one parallel implementation of VNDS.

5.4. GPU-based parallelization of VNS. GPUs offer massive parallelism through thousands of lightweight cores designed for many simultaneous computations. This makes them a natural candidate for accelerating the computationally intensive phases of VNS. Despite this natural fit, GPU-based parallelization is the least represented architecture in the parallel VNS literature. We have identified six publications that explicitly investigate GPU-based implementations of VNS. Most GPU-based approaches employ algorithm-level decomposition (Dec=A), in which the computationally intensive neighborhood evaluations are executed concurrently across GPU threads, while higher-level search is controlled by CPU. Despite this shared property, the six publications deal with different optimization problems, utilize different base VNS algorithms, and differ by the extent to which GPU computation is integrated into the VNS search. We review them in chronological order of publication.

In [31], the authors propose a Hybrid General Variable Neighborhood Search (HGVNS) and its parallel variant (HP-HGVNS) for the Single VRP with Deliveries and Selective Pickups (SVRPDSP). In this problem, a single vehicle delivers products to all customers while optionally collecting pickups for revenue, as long as the capacity of the vehicle is not exceeded. The main idea was to show the advantage of combining classical metaheuristics with modern GPU parallelism for solving challenging vehicle routing problems. The novel HGVNS algorithm, built on the GVNS framework, outperforms state-of-the-art results from the literature. HP-HGVNS is proposed for large-scale problems and incorporates memory optimization aimed at improving data management and access efficiency between CPU and GPU. The parallelelization strategy employs algorithm-level decomposition in a heterogeneous CPU-GPU environment (DEC=A, PA=*HET*). In particular, the CPU is responsible for controlling the search process by orchestrating the complete metaheuristic workflow including shaking, neighborhood selection and solution acceptance. The GPU is responsible for the computationally intensive Best Improvement local search within the VND. For each neighborhood, HP-HGVNS launches a GPU kernel in which CUDA threads evaluate $O(n^2)$ candidate move pairs in parallel, one pair (i, j) per thread. This corresponds to fine-grained parallelism (G=*Fg*). After each kernel execution, GPU stores the move costs in the result matrix, which is copied back to CPU. CPU then selects the best move sequentially, yielding synchronous CPU–GPU coordination with CPU as the central controller (C=1C, C&L=S, CT=*Star*). All GPU threads start from the same incumbent solution and execute the same evaluation procedure (Div=SPSS). Load balancing is static as all $O(n^2)$ thread configurations are fixed at kernel launch with no runtime redistribution (LB=*Stat*). The implementation uses C++ and CUDA SDK 4.0 on an Intel i7 2.67 GHz CPU (one core used) with a GeForce GTX 560 Ti GPU featuring 384 cores and 1 GB of memory (PPM=*CUDA*). Experimental evaluation was conducted on 68 benchmark instances, with each configuration executed 10 times

to assess algorithm stability. The results confirm that HGVNS improves the best known solutions from the literature. In case of HP-HGVNS parallel performance, the 1-Or-opt local search achieved an average speedup of 27.26, while 2-Or-opt reached a maximum speedup of 43.75. The Swap local search obtained the lowest speedup values, ranging from 1.91 to 14.36, which limited the overall gains of HP-HGVNS. The complete algorithm achieves average speedups ranging from 2.73 on small instances to 16.23 on large instances, confirming the advantage of GPU parallelism for large-scale routing problems. Complete Classification: HTPM = 1L/A/1C/S/SPSS, ITPM = *Fg*/*HET*/*Star*/*Stat*/*OpenMP+CUDA*.

The authors in [30] also address SVRPDSP and present the Four-Neighborhood VNS (FN-VNS), based on exploration of four neighborhoods (Swap, 2-Opt, 1-OrOpt and 2-OrOpt). CPU controls the FN-VNS search flow, i.e., perturbation, neighborhood selection, and acceptance, while the four local-search procedures are executed on GPU (C=1C). Therefore, the parallelization strategy follows a single-level algorithm-level decomposition (L=1L, Dec=A). For each neighborhood, the GPU kernels evaluate candidate moves in parallel, where each thread evaluates a move indexed by customer positions. This corresponds to fine-grained parallelism (G=*Fg*) over a single incumbent (Div=SPSS). CPU dispatches each kernel and collects the result vector after the best-improvement LS step is applied, establishing a synchronous star communication topology with GPU as the logical central controller (C&L=S, CT=*Star*). Work is distributed statically across 256 threads per block at kernel launch (LB=*Stat*). The implementation uses C++ and CUDA 4.0 on a CPU–GPU platform composed of an Intel i7 2.67 GHz CPU, with only one CPU core used, and a GeForce GTX 560 Ti GPU with 384 cores and 1 GB of memory (PA=*HET*, PPM=*CUDA*). Experiments on 68 benchmark instances (each run repeated 10 times) show that FN-VNS improves solution quality for 51 out of 68 instances compared to the evolutionary algorithm. On a new set of 32 large-size instances, the maximum overall speedup over the sequential version is 14.49.

A hybrid CPU-GPU parallelization scheme of VNS for the multi-product dynamic lot sizing problem with product returns and recovery was proposed in [10]. The study investigates several parallelization strategies for the GVNS metaheuristic on a CPU-GPU platform. The implementations rely on OpenMP and OpenACC, as well as their hybrid combination. Due to strong data dependencies in the neighborhood search procedure, the authors conclude that it was not possible to parallelize the neighborhood search itself. Instead, parallelism was applied to the evaluation of the objective function by exploiting the independence among products in the uncapacitated problem setting. Three parallel GVNS implementations are evaluated: an OpenMP-based version, an OpenACC-based version, and a hybrid OpenMP-OpenACC version. They are tested on 108 benchmark instances within stopping criterion of 60 seconds. The first two parallel variants are classified as single-level algorithm (L=1L), while the third employs multiple levels of decomposition. In the OpenMP GVNS implementation, parallelism is introduced exclusively at the CPU level via OpenMP directives on a shared-memory architecture (PA=*SMA*). The GVNS logic remains sequential, while objective-function evaluations are parallelized across products using OpenMP directives enforcing implicit barriers (L=1L,

Dec=A, C&L=S). A singe CPU thread controls the entire GVNS logic as worker threads operate on the same solution by executing parallel loops (C=1C). Note that due to the PGI University Developer License constraints, the maximum number of available OpenMP threads was limited to four. The OpenACC parallel strategy (VNDOpenACC), detailed in Algorithm 1 of [10], serves as the representative GPU-based approach. The VNDOpenACC algorithm uses a single-level, algorithm-level parallelization where a centralized CPU-controlled VND coordinates product-parallel computations on a single GPU (HTPM: L=1L, Dec=A, C=1C). Communication is strictly synchronous, as each iteration launches a GPU kernel, waits for all thread groups (gangs) to finish, and then updates the incumbent solution without any learning or adaptive behavior (C&L=S). All GPU gangs operate on the same incumbent solution with the same VND logic (Div=SPSS). At implementation level, VNDOpenACC employs fine-grained product-level parallelism on a heterogeneous CPU–GPU platform in Fortan (ITPM: G=*Fg*, PA=*HET*). The CPU acts as a central hub launching kernels and collecting results, while GPU gangs do not communicate directly, which corresponds to a star communication topology (CT=*Star*). Work is distributed statically; at kernel launch—product-to-gang assignments do not change during execution (LB=*Stat*). The entire GPU offloading and parallelization is implemented through OpenACC directives (data, parallel loop, gang vector, reduction), making PPM=*OpenACC*. Finally, HybridGVNS follows a two-level decomposition (L=2L) combining two GPU devices: an NVIDIA GeForce GTX 980 (4 GB, 2048 CUDA cores) and an NVIDIA GeForce GTX 960 (2 GB, 1024 CUDA cores). At Level 1, GVNS is parallelized at the CPU level using OpenMP, while at Level 2 OpenACC accelerates product-level computations inside of VND on the GPU. At level 1, OpenMP threads are each assigned to one GPU device and execute the shaking and VND phases within a critical section, starting from the same initial solution (L=1L, Div=SPSS, G=*Cg*). OpenMP threads perform a complete search where, at the end of each GVNS iteration, the master thread selects the best solution among these generated by two GPUs, which also represents a synchronization point (C=1C, C&L=S). Within each GPU, the VND procedure is run using OpenACC in the same manner as in VNDOpenACC strategy. Results show that all parallel implementations improved the solution quality w.r.t. the sequential VND. Between the three strategies, the most successful was HybridGVNS, followed by VNDOpenACC.

In [60], the authors present three parallel VNS variants (VNSPLS, VNSPIR and VNSHYB) for the Inverse Lighting Problem (ILP) and compare them to the original sequential VNS (VNSPW). The base algorithm VNSPW follows an RVNS structure. Each parallel variant is implemented in both CPU and heterogenous CPU+GPU variants. All implementations are developed in Matlab R2011b using MEX files, and the CPU+GPU implementations are evaluated on two platforms: Platform I (Intel i7 2600 at 3.4 GHz + NVIDIA Tesla C2070) and Platform II (Intel i7 3770 at 3.4 GHz + NVIDIA GeForce GTX 680) (PA=*HET*, PPM=*CUDA*). The key parallelizable component is the simultaneous evaluation of multiple candidate solutions. Instead of performing several independent matrix–vector products, the authors replace them with a single matrix–matrix multiplication, which

is particularly well suited for GPU. The stopping criterion for all variants is a fixed budget of 30,000 function evaluations. The first variant, VNSPLS, randomly samples at least 500 feasible candidate solutions from the current neighborhood $N_k(x)$ of a single VNS trajectory. On CPU, solutions are evaluated simultaneously through a matrix–matrix multiplication, while in the CPU–GPU implementation, this evaluation is offloaded to GPU. VNSPLS exhibits single-level explicit search space decomposition with centralized control where the candidate solutions are partitioned among the parallel computation units for simultaneous evaluation (HTPM: 1L/SE/1C/S/SPSS, G=*Fg*). Work is distributed statically (LB=*Stat*), and CPU acts as the central hub collecting the result matrix (CT=*Star*). In the second strategy, VNSPIR, the authors conceptually introduce several independent trajectories, so the algorithmic intention corresponds to independent implicit search-space decomposition and multiple incumbent solutions (HTPM = 1L/SI/1C/S/MPSS). However, the paper does not specify how these trajectories are managed concurrently at the CPU level. The third strategy, VNSHYB, represents a balanced approach between exploitation and exploration, as it combines multiple candidates per VNS (50 each) with multiple VNS runs (10 total). All VNS instances are managed sequentially, building input matrix with 500 solutions, which are explicitly partitioned among GPU threads (Dec=A, PA=*GPU*). All GPU threads start with the same input matrix and apply identical evaluation procedures without any GPU thread acting as a master (Div=SPSS, C=pC). Throughout GPU computation phase, the threads must synchronize (C&L=S), and at the end, all contributions are collected centrally (CT=*Star*, LB=*Stat*). The results show that VNSPLS outperforms both VNSPIR and VNSHYB, particularly for large problem instances where the evaluation phase is time demanding.

In [11], the authors address the Traveling Thief Problem (TTP) by proposing a List-Constrained Randomized VND (LCRVND) hybridized with GRASP. The method combines GRASP with LCRVND as a local search procedure for simultaneous exploration of multiple neighborhood structures. The LCRVND procedure maintains an ordered list of L_{max} best candidate solutions. In each iteration, LCRVND selects the best unmarked solution, marks it, enumerates its neighbors for each neighborhood structure, merges the generated solutions into the list, and retains only the best *listSize* solutions. The process continues until no unmarked solution remains or an iteration limit is reached. Thus, the hybrid algorithm operates in two phases: in the first, GRASP invokes a limited version of LCRVND with small list size to preform few iterations in few neighborhoods; in the second, the best solution obtained is submitted to a wider LCRVND search. The algorithm is implemented on a heterogeneous CPU-GPU platform, using C++ with OpenMP and CUDA on Intel Core i7-4820K 3.70 GHz with an NVIDIA GeForce GTX 780. The parallel GRASP+LCRVND algorithm is best classified as a hierarchical functional decomposition (L=nL), each classified separately. Namely, GRASP phase exploits CPU-parallelism through OpenMP threads, where each thread runs on its own CUDA stream and independently constructs randomized TTP solutions before applying LCRVND. At Level 1 the GRASP algorithm independently constructs its own complete heuristic solution and local search

trajectory, without coordination. Thus, the classification is HTPM: (1L/SI/-/-/MPSS), ITPM: (*Cg*/*SMA*/-/-/*OpenMP*). At Level 2, the LCRVND procedure is executed on GPU. Starting from a solution (x', y') selected from the list of size L_{max}, it marks this solution and explores the neighborhood $\lambda \in \Lambda$ (as it is shown by Algorithm 2 in [11]). In line 6, L' is generated as the list of solutions from the neighborhood $\lambda(x', y')$, which is implemented as a CUDA kernel call whose launch configuration depends on the neighborhood size. As different neighborhoods may have different sizes, dynamic parallelism is used to improve GPU utilization. In our taxonomy, this level is classified as 1L/A/1C/S/SPSS (HTPM) and *Fg*/*GPU*/*Star*/*Dyn*/*CUDA*. The GRASP + LCRVND hybrid delivers notable speedups on 60 TTP instances, reducing average runtime from about 10 minutes to 3.84 minutes while outperforming baseline methods in 54 cases. The results indicate that the list size strongly influences this efficiency, which directs the future implementations towards the automated or adaptive tuning of this parameter.

In [156], the authors introduce Distributed VND (DVND), a novel parallel local search procedure designed for heterogeneous CPU–multi-GPU environments. DVND addresses two key challenges of parallelizing VND: the sensitivity of solution quality to neighborhood processing order, resolved naturally by GPU task scheduling and processing time differences playing a role similar to RVND's randomized ordering; and the dependency on the current best solution, resolved by a shared history structure H. A history is an auxiliary data structure that stores a global best solution (to be shared between all processes) and the current best solution for each neighborhood. Another contribution is Multi Improvement (MI) strategy for selecting neighbors, in which several mutually independent improving moves may be applied simultaneously. Six parallel configurations are developed by combining the local search type (DVND or RVND), the GPU configuration (single-GPU or multi-GPU), and the move-selection strategy (BI or MI), yielding RVND-SG-BI, RVND-SG-MI, DVND-SG-BI, DVND-SG-MI, DVND-MG-BI, and DVND-MG-MI variants that are evaluated. To avoid trapping LS methods in local optima, the authors integrated them into a hybrid GRASP + ILS metaheuristic framework (GILS), tested on the Minimum Latency Problem (MLP). The most representative complete variant is GILS-DVND-MG-MI. Selected configurations are embedded into a hybrid GRASP + Iterated Local Search framework (GILS), where GRASP generates high-quality initial tours and DVND provides an intensive refinement phase. We distinguish GILS-DVND-MG-MI as it represents the authors' complete vision. The algorithm involves a 3-level parallel decomposition, where multiple CPU threads coordinate GPU threads (L=nL). Each level exhibits distinct HTPM and ITPM characteristics and is therefore classified independently. At level 1, g POSIX threads are lunched from the same solution and with the same shared history H, each assigned to a different GPU device on which *dvnd_task* procedure is executed (Alg. 3, line 7 in [156]). Each CPU thread independently handles one GPU-level *dvnd_task* on a subset of neighborhoods, while cooperation among tasks is mediated through H, which stores the best solutions found during the search. The CPU threads periodically (after each kernel completion) check the history H to exchange information about the best solutions. Therefore, this

level is classified as 1L/SC/pC/A/SPSS (HTPM) and *Cg/HET/-/Pthreads+CUDA* (ITPM). Because the neighborhood set is partitioned, load balance is static while PPM=*pthreads+CUDA*. The granularity is coarse because each CPU thread manages a complete DVND procedure on one GPU. At Level 2, inside each *dvnd_task*, the CPU thread coordinates the CUDA kernels launched for its assigned subset of neighborhoods on its assigned GPU. Each kernel corresponds to one neighborhood search and is launched asynchronously. When a kernel finishes, the CPU thread retrieves its result, updates the shared history H if an improvement is found, as well as the best solution currently stored in H, and relaunches the corresponding kernel only if this solution improves the local optimum associated with that neighborhood. Otherwise, the neighborhood is stored in a locally optima history and is revisited only if the best solution in H later improves its associated local optimum. This level is classified as 1L/A/1C/A/SPSS in HTPM and *Mg/GPU/Star/Dyn/CUDA* in ITPM. The decomposition is algorithm-level rather than independent multi-search, because kernels are neighborhood-search components of the same DVND procedure; control is centralized in the CPU thread; communication is asynchronous; and dynamic behavior arises from runtime kernel completion and conditional relaunching, not from redistribution of neighborhoods among GPUs. Level 3 represents the finest granularity of parallelization where individual move evaluations are explicitly distributed across many GPU threads within each neighborhood kernel (G=*Fg*, Dec=A, PPM=*CUDA*). All threads evaluate moves from the same base solution, but each thread computes one candidate move rather than contributing to the evaluation of a single move. As no individual GPU thread controls the search and the best move is obtained through a collective reduction, this level is classified as C=pC, Div=SPSS and C&L=S. Our ITPM classifies CT as *Star* due to the central aggregation point. Because the move-to-thread assignment is fixed within a kernel launch, LB is static. The results show the effectiveness of the proposed parallel strategies in terms of solution quality, performance, and scalability. In particular, GILS-DVND-MG-MI represents the state-of-the-art for solving MLP, achieving substantial speedups (10.62 in average) while maintaining or improving solution quality.

Final remarks. Table 5 illustrates the chronological listing of the parallel VNS algorithms proposed for CPU-GPU platforms. All papers are published within the period 2012-2018, indicating a more recent research direction compared to shared-memory and distributed-memory strategies. In addition, all papers utilize heterogeneous CPU-GPU (*HET*) platforms, where CPU controls the workflow while GPU handles computationally intensive neighborhood evaluations.

CPU–GPU systems naturally exhibit a hierarchical hardware organization. However, in our taxonomy, the Level parameter (L) denotes the levels of VNS algorithmic decomposition, rather than the hardware levels. Papers such as [31] and [30], in which the CPU sequentially controls the VNS while the GPU performs parallel neighborhood evaluations, are classified as 1L with Dec = A and PA = *HET*. In contrast, [10], [11], and [156] are classified as nL because they exhibit multiple levels of VNS algorithmic decomposition: (1) parallel VNS instances executed at the CPU-level, and (2) GPU-based acceleration of neighborhood evaluations within

TABLE 5. Chronological summary of parallel VNS algorithms with GPU-based implementations.

Year	Ref	Parallel Algorithms	Base Algorithm	Problem
2012	[31]	HP-HGVNS	HGVNS	Single Vehicle Routing Problem with Deliveries and Selective Pickups (SVRPDSP)
2016	[30]	FN-VNS	GVNS	Single Vehicle Routing Problem with deliveries and selective pickups
2017	[10]	OpenMP GVNS, VNDOpenACC, HybridGVNS	GVNS	Multi-product dynamic lot sizing problem with product returns and recovery
2017	[60]	VNSPLS, VNSPIR, VNSHYB	VNS	Inverse Lighting Problem (ILP)
2018	[11]	GRASP+LCRVND	LCRVND	Traveling Thief Problem (TTP)
2018	[156]	RVND-SG-BI, RVND-SG-MI, DVND-SG-BI, DVND-SG-MI, DVND-MG-BI, DVND-MG-MI, GILS-DVND-MG-MI	DVND, RVND	Minimum Latency Problem (MLP)

each instance. In addition, [156] extends the decomposition to three levels and employs multi-GPU execution and cooperative search.

It is important to note that while the GPU hardware hierarchy strongly influences which parallelization patterns are feasible in practice, the proposed taxonomies classify abstract algorithmic structures rather than hardware implementation details. Although abstract patterns often reflect hardware capabilities, the classification is determined by the logical communication and control structure of the algorithm itself, not the underlying hardware organization. Nevertheless, these mismatches suggest that future parallel metaheuristic taxonomies should treat hardware awareness as an explicit complementary dimension.

5.5. Hybrids, hierarchical and other approaches. A parallel VNS-based algorithm (pVNS) for TSP was proposed in [132]. The sequential The sequential baseline is a VNS method that explores four types of neighborhood structures: one adjacent pair, one random pair, two adjacent pairs, and two random pairs. Upon analysis of the execution time profile, the authors concluded that 99% of the execution

is spent on the evaluation of candidate solutions. Therefore, they proposed to perform this part in parallel and developed pVNS. The main idea of parallelization was to explore more than one solution at a time. pVNS combines two types of parallelization (shared- and distributed-memory) hierarchically, implemented using MPI and OpenMP, respectively. The experimental platform is the Tarantula minigrid, a multi-cluster architecture composed of two clusters: the ITVer cluster (1 master and 14 slave nodes) and the UAEM cluster (1 master and 18 slave nodes). The parallelization is structured across three levels. At the first (inter-cluster) level, the two master nodes — one per cluster — periodically exchange their best solutions with each other via MPI before each new round of intra-cluster evaluation. This is classified as 1L/SC/pC/A/MPSS (HTPM) and *Cg/DMA/FulC/Stat/MPI* (ITPM). The two clusters cooperate by sharing best solutions; control is decentralized as no master exists between the two clusters; communication is asynchronous as the exchange occurs independently of the other cluster's internal state; granularity is coarse as each entity is a complete cluster-level search; topology is fully connected as both masters communicate directly with each other; and load balancing is static. At the second (intra-cluster) level, a VNS procedure is executed on a master node. The master node maintains a group of candidate solutions, distributes them to slave nodes for evaluation via MPI, and collects the returned path costs. The group is then updated by retaining only the best solutions. This level is classified as nL/SI/1C/S/MPSS (HTPM) and *Cg/DMA/Star/Stat/MPI* (ITPM). Multiple candidate solutions are explored independently by working nodes starting from different points using the same VNS algorithm; control is centralized in the master node; communication is synchronous as the master waits for all path costs before updating the candidate set; granularity is coarse as each worker executes a complete solution evaluation; architecture is distributed-memory; topology is star with the master as hub; and load balancing is static as each slave receives one solution per round. At the third level, each worker node receives a complete solution s_i, divides it into sub-paths and assigns these segments to several threads in shared-memory parallelization manner to determine its partial path cost. When all threads complete, the worker node aggregates the partial values to obtain the total tour length $f(s_i)$, which is then returned to the master node. Thus, this level is classified as 1L/A/1C/S/SPSS (HTPM) and *Fg/SMA/Star/Stat/OpenMP* (ITPM). pVNS was evaluated on one randomly generated TSP instance with 4000 cities, and the average results of 30 repetitions evaluating a group of 60 candidate solutions were reported. Different configurations of clusters were considered due to the available heterogeneous multi-cluster architectures. Parallel configurations with 10, 20, and 30 processes were tested, alongside two sequential baselines on one quad-core and one dual-core node. As performance measures, execution time, speedup, and solution quality (tour length) are considered.

6. Conclusion

This study presents a comprehensive survey of strategies and implementations for parallel Variable Neighborhood Search (VNS) algorithms. We identified and reviewed 58 relevant publications that consider various aspects of parallelization,

ranging from the exploration of available hardware architecture, identification and granulation of parallelizable parts, to synchronization and communication aspects. Although metaheuristics contain numerous sources of parallelism, their stochastic nature introduces new perspectives and expectations when parallelized. In particular, the evaluation of parallel metaheuristics requires consideration of both solution quality and computational efficiency. As stochastic algorithms, parallel metaheuristics should be evaluated by the same standards that apply to sequential metaheuristics — standards that are well established in the literature and cover multiple independent runs, statistical validation, transparent parameter tuning, and consistent stopping criteria. The survey reveals that these standards are inconsistently applied across the parallel VNS literature, which limits the reproducibility and comparability of results across studies.

We examined both theoretical and practical (implementation-related) aspects of parallelization and provided taxonomies that enable detailed classification of existing parallel metaheuristic approaches, VNS in particular. Our statistical analysis considers the temporal evolution of research approaches to parallelization of VNS, the types of publications used to disseminate results, the variants of the VNS algorithm that have been parallelized, and the optimization problems addressed. The analysis reveals four distinct phases in the field's evolution: the Foundational phase (2001–2005, 9 publications, 1.8/year), characterized by pioneering work from two dominant research groups; the Latent phase (2006–2014, 11 publications, 1.2/year), marked by steady but moderate activity before the next enabling hardware generation; the Rapid growth period (2015–2018, 22 publications, 5.5/year), driven by the maturation of GPU and many-core architectures and representing the most productive period in parallel VNS history; and the Maturation phase (2020–2025, 16 publications, 2.7/year), characterized by sustained output with increasing algorithmic sophistication. The bibliometric analysis additionally reveals that shared-memory architectures account for the largest share of implementations (47.4%), followed by distributed-memory (40.4%), with GPU-based approaches representing a smaller but distinct contribution (10.5%). Among the 38 ranked journal papers, 81.6% appear in Q1 or Q2 venues, reflecting a high standard of dissemination. Routing, Scheduling, and Location problems together account for 75.4% of the reviewed publications, confirming the dominance of classical combinatorial OR problems in the parallel VNS literature.

We also defined research questions that should be considered when evaluating the description and performance of each parallel VNS implementation. They include (1) The relationship between sequential and parallel VNS; (2) Performance Evaluation Metrics; (3) Parameter Settings and Adaptations; (4) Search Dynamics and Robustness; (5) Experimental Setup and Benchmarking; (6) Novel Algorithmic Contributions. However, we did not completely and systematically classify all parallel strategies or answer all research questions, as some of the reviewed papers are more than 20 years old and exhibit variability and incompleteness in the data provided. Nonetheless, we strongly believe that this chapter provides a solid basis and useful material for future research related to parallelization of not only VNS but metaheuristics in general.

Despite the lack of necessary data in some publications, our study can be extended to extract and analyze additional characteristics of existing parallelization strategies. For example, future work could involve deeper taxonomy-driven statistical analysis, including the distribution of HTPM and ITPM parameters, such as: decomposition choices, control schemes, communication patterns, architectural models, and task granularity levels. Their relation to reported performance, scalability, and solution quality could also be studied. The proposed classification clearly highlights that certain parallelization strategies remain underrepresented in the parallel VNS literature. In particular, multi-level decomposition is rare and appears primarily in GPU-based implementations, with the notable exception of hybrid MPI+OpenMP approaches. Knowledge-based communication appears in only a handful of papers, and dynamic or adaptive load balancing remains largely unreported — partly reflecting the inconsistencies in experimental reporting noted earlier. These observations point to concrete directions for future research. Additional investigations may examine which strategies proved the most popular, assess how many papers analyzed the scalability of the performance with respect to the number of processors, and similar quantitative characterizations. All these analyses may provide useful guidance for new developments in parallel VNS algorithms.

Acknowledgments. This work has been partially supported by the Serbian Ministry of Science, Technological Development, and Innovation, Agreement No. 451-03-136/2025-03/200029.

References

[1] E. H. Aarts, H. M. Korst, P. J. van Laarhoven, *Simulated annealing*, In: E. Aarts, J. K. Lenstra (eds.), *Local Search in Combinatorial Optimization*, 121–136, Wiley, Chichester, 1997.

[2] M. Abdelatti, M. Sodhi, R. Sendag, *A multi-GPU parallel genetic algorithm for large-scale vehicle routing problems*, In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, New York, USA, 2022, 1–8.

[3] M. F. Abdelatti, M. S. Sodhi, *An improved GPU-accelerated heuristic technique applied to the capacitated vehicle routing problem*, In: C. A. Coello Coello (ed.), *GECCO '20: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 663–671, Cancún, Mexico, July 8–12, 2020, Association for Computing Machinery, New York, USA, 2020.

[4] R. Akay, A. Basturk, A. Kalinli, X. Yao, *Parallel population-based algorithm portfolios: An empirical study*, Neurocomputing **247** (2017), 115–125.

[5] M. A. Akbay, C. B. Kalayci, O. Polat, *A parallel variable neighborhood search algorithm with quadratic programming for cardinality constrained portfolio optimization*, Knowl.-Based Syst. **198** (2020), 105944.

[6] E. Alba, *Parallel Metaheuristics: A New Class of Algorithms*, Wiley-Interscience, Hoboken, NJ, USA, 2005.

[7] E. Alba, G. Luque, S. Nesmachnow, *Parallel metaheuristics: Recent advances and new trends*, Int. Trans. Oper. Res. **20**(1) (2013), 1–48.

[8] A. L. B. Almeida, J. C. Lima, M. A. M. Carvalho, *Systematic literature review on parallel trajectory-based metaheuristics*, ACM Comput. Surv. **55**(8) (2022), 171.

[9] A. Anokić, Z. Stanimirović, T. Davidović, Đ. Stakić, *Variable neighborhood search based approaches to a vehicle scheduling problem in agriculture*, Int. Trans. Oper. Res. **27**(1) (2020), 26–56.

[10] N. Antoniadis, A. Sifaleras, *A hybrid CPU-GPU parallelization scheme of variable neighborhood search for inventory optimization problems*, Electron. Notes Discrete Math. **58** (2017), 47–54.

[11] R. P. Araujo, E. Rios, I. M. Coelho, L. A. J. Marzulo, M. C. Castro, *A novel list-constrained randomized vnd approach in GPU for the traveling thief problem*, Electron. Notes Discrete Math. **66** (2018), 183–190.

[12] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, *A view of the parallel computing landscape*, Commun. ACM, **52**(10) (2009), 56–67.

[13] M. Aydin, M. Sevkli, *Sequential and parallel variable neighborhood search algorithms for job shop scheduling*, In: F. Xhafa, A. Abraham (eds.), *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, 125–144, Springer, New York, USA, 2008.

[14] M. Ayed, J-L Gaudiot, *An efficient heuristic for code partitioning*, Parallel Comput. **26**(4) (2000), 399–426.

[15] 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.

[16] T. Bartz-Beielstein, C. Doerr, D. van den Berg, J. Bossek, S. Chandrasekaran, T. Eftimov, A. Fischbach, P. Kerschke, W. La Cava, M. Lopez-Ibanez, K. M. Malan, J. H. Moore, B. Naujoks, P. Orzechowski, V. Volz, M. Wagner, T. Weise, *Benchmarking in optimization: Best practice and open issues*, arXiv:2007.03488, 2020.

[17] Z. Beheshti, S. M. H Shamsuddin, *A review of population-based meta-heuristic algorithms*, Int. J. Adv. Soft Comput. Appl **5**(1) (2013), 1–35.

[18] M. P. Bekakos, K. D. Antonakopoulos, *Fairness in performance evaluation parameters*, In *Proceedings of 7th Hellenic–European Conference on Computer Mathematics and its Applications*, Athens University of Economics and Business, Athens, Greece, Sept. 22-24, 2005.

[19] B. Blaise, *Introduction to parallel computing*, https://computing.llnl.gov/tutorials/parallel_comp/, High-Performance Computing at Lawrence Livermore National Laboratory, 2012.

[20] G. Blake, R. G. Dreslinski, T. Mudge, *A survey of multicore processors*, IEEE Signal Proc. Mag. **26**(6) (2009), 26–37.

[21] S. Brawer, *Introduction to Parallel Programming*, Academic Press, Inc., San Diego, CA, USA, 1989.

[22] J. Brimberg, S. Salhi, R. Todosijević, D. Urošević, *Variable neighborhood search: The power of change and simplicity*, Comput. Oper. Res. **155** (2023), 106221.

[23] B. P. Bruck, A. G. dos Santos, J. E. C. Arroyo, *Hybrid metaheuristic for the single vehicle routing problem with deliveries and selective pickups*, In: *2012 IEEE Congress on Evolutionary Computation*, IEEE, 2012, New York, USA, 1–8.

[24] P. R. Calégari, *Parallelization of population-based evolutionary algorithms for combinatorial optimization problems*, PhD thesis, Verlag nicht ermittelbar, 1999.

[25] R. A. Campos, G. O. Chagas, L. C. Coelho, P. Munari, *Exact methods and a variable neighborhood search for the robust capacitated p-median problem*, Comput. Oper. Res. **173** (2025), 106851.

[26] J. Cao, L. Lin, *A parallel improved variable neighborhood search algorithm for multidepot multicompartment simultaneous pickup and delivery problem with time windows and split demands*, In: H. Chen, W. Shangguan (eds.), *Fourth International Conference on Intelligent Traffic Systems and Smart City* (*ITSSC 2024*), Proc. SPIE **13422**, 253–266, SPIE, Bellingham, WA, USA, 2025.

[27] D. Carvalho, M. Moreira, V. Dias, *An experimental study of variable neighborhood search for general-purpose subgraph optimization in parallel systems*, In: *Proceedings of the 26th Brazilian Symposium on High Performance Computing Systems*, 85–96, Bonito, MS, Brasil, October 28–31, 2025, SBC, Porto Alegre, RS, Brasil, 2025.

[28] G. O. Chagas, L. A. N. Lorena, R. D. C. dos Santos, J. Renaud, L. C. Coelho, *A parallel variable neighborhood search for α-neighbor facility location problems*, Comput. Oper. Res. **165** (2024), 106589.

[29] E. Cieza, L. Teylo, Y. Frota, C. Bentes, L. M. A. Drummond, *A GPU-based metaheuristic for workflow scheduling on clouds*, In: H. Senger, O. Marques, R. Garcia, T. P. de Brito, R. Iope, S. Stanzani, V. Gil-Costa (eds.), *High Performance Computing for Computational Science – VECPAR 2018*, Lect. Notes Comput. Sci. **11333**, 62–76, Springer, New York, USA, 2019.

[30] I. M. Coelho, P. L. A. Munhoz, L. S. Ochi, M. J. F. Souza, C. Bentes, R. Farias, *An integrated CPU–GPU heuristic inspired on variable neighbourhood search for the single vehicle routing problem with deliveries and selective pickups*, Int. J. Prod. Res. **54**(4) (2016), 945–962.

[31] I. M. Coelho, L. S. Ochi, P. L. A. Munhoz, M. J. F. Souza, R. Farias, C. Bentes, *The single vehicle routing problem with deliveries and selective pickups in a CPU–GPU heterogeneous environment* In: G. Min, L. Lefevre, J. Hu, L. Liu, L. T. Yang, S. Seelam, T. J. Watson (eds.), *HPCC '12: Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, 1606–1611, Liverpool, United Kingdom, June 25–27, 2012, IEEE, Los Alamitos, California, USA, 2012.

[32] R. Corrêa, I. Dutra, M. Fiallos, F. Gomes (eds.), *Models for Parallel and Distributed Vomputation: Theory, Algorithmic Techniques and Applications*. Kluwer Academic Publishers, Dordrecht, Netherlands, 2002.

[33] T. G. Crainic, *Parallel metaheuristic search*, In: R. Martí, P. M. Pardalos, M. G. C. Resende (eds.), *Handbook of Heuristics*, 809–847, Springer, New York, USA, 2018.

[34] T. G. Crainic, *Parallel meta-heuristics and cooperative search*, In: M. Gendreau, J.-Y. Potvin (eds.), *Handbook of Metaheuristics*, International Series in Operations Research & Management Science **272**, 419–451, Springer, Cham, 2019.

[35] T. G. Crainic, *Parallel meta-heuristic search*, In R. Martí, P. M. Pardalos, M. G. C. Resende (eds.), *Handbook of Heuristics*, 2$^{\text{nd}}$ ed., 1087–1130, Springer, New York, USA, 2025.

[36] T. G. Crainic, T. Davidović, D. Ramljak, *Designing parallel meta-heuristic methods*, In: M. Despotović-Zrakić, V. Milutinović, A. Belić (eds,), *High Performance and Cloud Computing in Science and Education*, 260–280, IGI-Global, Hershey, PA, USA, 2014.

[37] T. G. Crainic, M. Gendreau, P. Hansen, N. Hoeb, N. Mladenović, *Parallel VNS for the p-median*, In: J. Sousa (ed.), *Proceedings of 4th Metaheuristics International Conference, MIC'2001*, 595–599, Porto, Portugal, July 16–20, 2001.

[38] T. G. Crainic, M. Gendreau, P. Hansen, N. Mladenović, *Parallel Variable Neighborhood Search for the p-median*, Technical report, GERAD report G-2003-04, Montreal, Canada, 2003.

[39] 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.

[40] T. G. Crainic, N. Hail, *Parallel meta-heuristics applications*, In: E. Alba (ed.), *Parallel Metaheuristics*, 447–494, John Wiley & Sons, Hoboken, NJ, USA, 2005.

[41] T. G. Crainic, M. Toulouse, *Parallel strategies for meta-heuristics*, In: F. Glover, G. Kochenberger (eds.), *Handbook in Metaheuristics*, 475–513, Kluwer Academic Publishers, Dordrecht, Netherlands, 2003.

[42] T. G. Crainic, M. Toulouse, *Parallel Meta-heuristics*, In: M. Gendreau, J. Y. Potvin (eds.), *Handbook of Metaheuristics*, 497–541, Springer, New York, USA, 2010.

[43] D. Crane, *Invisible Colleges: Diffusion of Knowledge in Scientific Communities*, University of Chicago Press, Chicago, IL, USA, 1972.

[44] V.-D. Cung, S. L. Martins, C. C. Ribeiro, C. Roucairol, *Strategies for the parallel implementations of metaheuristics*, In: C. C. Ribeiro, P. Hansen (eds.), *Essays and Surveys in Metaheuristics*, **15**, 263–308, Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[45] Z. Cvetanovic, *The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems*, IEEE Trans. Comput. **100**(4) (1987), 421–432.

[46] T. Davidović, *Bee colony optimization: Recent developments and applications*, In *Proceedings of XI Balkan Conference on Operational Research, BALCOR 2015*, 225–235,

Constanța, Romania, September 9–12, 2015, "Mircea cel Batran" Naval Academy Press, Constanța, Romania, 2015.

[47] T. Davidović, T. G. Crainic, *A strategy for parallelization of VNS for multiprocessor scheduling problem*, In: S. Vujić (ed.),*Zbornik radova*, 221–224, XXXVII Symposium on Operations Research, SYM-OP-IS 2004, Fruška Gora, Serbia, September 14–17, 2004, Faculty of Mining and Geology, Department of Computer Application, Beograd, 2004.

[48] T. Davidović, T. G. Crainic, *Benchmark problem instances for static task scheduling of task graphs with communication delays on homogeneous multiprocessor systems*, Comput. Oper. Res. **33**(8) (2006), 2155–2177.

[49] T. Davidović, T. G. Crainic, *MPI parallelization of variable neighborhood search*, In: A. Sifaleras, D. Urošević, N. Mladenović (eds.), *Proceedings of EURO mini conference XXVIII dedicated to variable neighborhood search, EUROmC-XVIII-VNS*, 241–248, Herceg-Novi, Montenegro, October 4-7, 2012, Elsevier B.V., Amsterdam, Netherlands, 2012.

[50] T. Davidović, T. G. Crainic, *Parallelization strategies for variable neighborhood search*, Technical Report CIRRELT-2013-47, CIRRELT, Centre interuniversitaire de recherche sur les réseaux d'entreprise systems, Montreal, Canada, 2013.

[51] T. Davidović, T. G. Crainic, *Parallel local search to schedule communicating tasks on identical processors*, Parallel Comput. **48** (2015), 1–14.

[52] T. Davidović, P. Hansen, N. Mladenović, *Permutation based genetic, tabu and variable neighborhood search heuristics for multiprocessor scheduling with communication delays*, Asia-Pac. J. Oper. Res. **22**(3) (2005), 297–326.

[53] T. Davidović, T. Jakšić, D. Ramljak, M. Šelmić, D. Teodorović, *Parallelization strategies for bee colony optimization based on message passing communication protocol*, Optimization **62**(8) (2013), 1113–1142.

[54] T. Davidović, D. Ramljak, M. Šelmić, D. Teodorović, *Parallel bee colony optimization for scheduling independent tasks to identical machines*, In: B. Forca, M. Kovač, O. Čabarkapa, D. Petrović (eds.), *Zbornik radova*, 389–392, XXXVII Symposium on Operations Research, SYM-OP-IS 2010, Tara, Serbia, October 21–24, 2010, Media center "Odbrana", Beograd, 2010.

[55] T. Davidović, D. Ramljak, M. Šelmić, D. Teodorović, *Bee colony optimization for the p-center problem*, Comput. Oper. Res. **38**(10) (2011), 1367–1376.

[56] T. Davidović, D. Ramljak, M. Šelmić, D. Teodorović, *MPI parallelization of bee colony optimization*, In: V. Kostoglou, G. Arabatzis, L. Karamitopoulos (eds.),*Procidings of the 1st International Symposium & 10th Balkan Conference on Operational Research* **2**, 193–200, Thessaloniki, Greece, September 22–24, 2011, University of Macedonia, Thessaloniki , 2011.

[57] T. Davidović, D. Teodorović, M. Šelmić, *Bee colony optimization Part I: The algorithm overview*, YUJOR **25**(1) (2015), 33–56.

[58] T. Davidović, M. Šelmić, D. Teodorović, D. Ramljak, *Bee colony optimization for scheduling independent tasks to identical processors*, J. Heuristics **18**(4) (2012), 549–569.

[59] L. M. de Assumpç ao Drummond, L. S. Vianna, M. B. da Silva, L. S. Ochi, *Distributed parallel metaheuristics based on GRASP and VNS for solving the traveling purchaser problem*, In *ICPADS '02: Proceedings of the 9th International Conference on Parallel and Distributed Systems*, 257–263, Taiwan, ROC, December 17–20, 2002, IEEE, New York, USA, 2002.

[60] I. Decia, R. Leira, M. Pedemonte, E. Fernández, P. Ezzatti, *A VNS with parallel evaluation of solutions for the inverse lighting problem*, In: G. Squillero, K. Sim (eds.), *Applications of Evolutionary Computation*, 20$^{\text{th}}$ European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Lect. Notes Comput. Sci. **10199**, 741–756, Springer, Cham, Switzerland, 2017.

[61] Y. Deng, *Applied Parallel Computing*, World Scientific, River Edge, NJ, USA 2013.

[62] V. Dias, C. H. C. Teixeira, D. Guedes, W. Meira, S. Parthasarathy, *Fractal: A general-purpose graph pattern mining system*, In: P. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, T. Kraska (eds.), *SIGMOD '19: Proceedings of the 2019 International Conference*

on Management of Data, 1357–1374, Amsterdam, The Netherlands, June 30 – July 5, 2019, ACM, New York, USA, 2019.

[63] E. W. Dijkstra, *Solution of a problem in concurrent programming control*, Commun. ACM **8**(9) (1965), 569.

[64] A. Djenić, N. Radojičić, M. Marić, M. Mladenović, *Parallel VMS for bus terminal location problem*, Appl. Soft Comput. **42** (2016), 448–458.

[65] T. Dokeroglu, E. Sevinc, T. Kucukyilmaz, A. Cosar, *A survey on new generation metaheuristic algorithms* Comput. Oper. Res. **137** (2019), 106040.

[66] M. Dorigo, T. Stützle, *Ant colony optimization: Overview and recent advances*, In: M. Gendreau, J-Y. Potvin (eds.), *Handbook of Metaheuristics*, 2nd ed., 227–263, Springer, New York, Dordrecht, Heidelberg, London, 2010.

[67] A. Duarte, J. J. Pantrigo, E. G. Pardo, J. Sánchez-Oro, *Parallel variable neighbourhood search strategies for the cutwidth minimization problem*, IMA J. Manag. Math. **27**(1) (2016), 55–73.

[68] V. Eijkhout, *Introduction to High Performance Scientific Computing*, Lulu.com, Morrisville, NC, USA, 2010; 2nd ed., 2014.

[69] M. Eskandarpour, S. H. Zegordi, E. Nikbakhsh, *A parallel variable neighborhood search for the multi-objective sustainable post-sales network design problem*, Int. J Prod. Econ. **145**(1) (2013), 117–131.

[70] M. Essaid, L. Idoumghar, J. Lepagnot, M. Brévilliers, *GPU parallelization strategies for metaheuristics: A survey*, Int. J. Parallel Emergent Distrib. Syst. **34**(5) (2019), 497–522.

[71] P. Fattahi, N. Bagheri Rad, F. Daneshamooz, S. Ahmadi, *A new hybrid particle swarm optimization and parallel variable neighborhood search algorithm for flexible job shop scheduling with assembly process*, Assembly Autom. **40**(3) 2020, 419–432.

[72] T. A. Feo, M. G. C. Resende, *Greedy randomized adaptive search procedures*, J. Glob. Optim. **6**(2) (1995), 109–133.

[73] M. Fontaine, S. Loudni, P. Boizumault, *Exploiting tree decomposition for guiding neighborhoods exploration for VNS*, RAIRO, Oper. Res. **47**(2) (2013), 91–123.

[74] E. Garcia-Gonzalo, J. L. Fernandez-Martinez, *A brief historical review of particle swarm optimization (PSO)*, J. Bioinform. Intell. Control **1**(1) (2012), 3–16.

[75] F. García-López, B. Melián-Batista, J. A. Moreno-Pérez, J. M. Moreno-Vega, *The parallel variable neighborhood search for the p-median problem*, J. Heuristics **8**(3) (2002), 375–388.

[76] M. Gendreau, J-Y. Potvin, *Tabu search*, In: M. Gendreau, J-Y. Potvin (eds.), *Handbook of Metaheuristics*, 2nd ed., 41–59, Springer, New York, Dordrecht, Heidelberg, London, 2010.

[77] F. Glover, M. Laguna, *Tabu Search*, Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[78] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[79] A. Grama, A. Gupta, G. Karypis, V. Kumar, *Introduction to Parallel Computing*, 2nd ed., Addison Wesley, Boston, MA, USA, 2003.

[80] J. Greensmith, A. Whitbrook, U. Aickelin, *Artificial immune systems*, In: M. Gendreau, J-Y. Potvin (eds.), *Handbook of Metaheuristics*, 2nd ed., 421–448, Springer, New York, Dordrecht, Heidelberg, London, 2010.

[81] P. Hansen, N. Mladenović, *Variable neighborhood search for the p-median*, Locat. Sci. **5**(4) (1997), 207–226.

[82] P. Hansen, N. Mladenović, *An introduction to variable neighborhood search*, In: S. Voss, S. Martello, I. H. Osman, C. Roucairol (eds.), *Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization*, 433–458, Kluwer, Dordrecht, Netherlands, 1999.

[83] P. Hansen, N. Mladenović, *Variable neighborhood search: Principles and applications*, Europ. J. Oper. Res. **130** (2001), 449–467.

[84] P. Hansen, N. Mladenović, *Variable neighbourhood search*, In: F. Glover, G. Kochenagen (eds.), *Handbook of Metaheuristics*, 145–184, Kluwer Academic Publishers, Dordrecht, Netherlands, 2003.

[85] P. Hansen, N. Mladenović, *Variable neighbourhood search*, In: E. K. Burke, G. Kendall (eds.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, 211–238, Springer, New York, Dordrecht, Heidelberg, London, 2005.

[86] P. Hansen, N. Mladenović, *Variable neighborhood search*, In: E. K. Burke, G. Kendall (eds.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, 313–337, Springer-Verlag, New York, USA, 2014.

[87] P. Hansen, N. Mladenović, J. Brimberg, J. A. Moreno Pérez, *Variable neighborhood search*, In: M. Gendreau, J-Y. Potvin (eds.), *Handbook of Metaheuristics*, 2nd ed., 61–86, Springer, New York, Dordrecht, Heidelberg, London, 2010.

[88] P. Hansen, N. Mladenović, J. Brimberg, J. A. Moreno Pérez, *Variable neighborhood search*, In: M. Gendreau, J-Y. Potvin (eds.), *Handbook of metaheuristics*, 3rd ed., 57–97, Springer, New York, Dordrecht, Heidelberg, London, 2019.

[89] P. Hansen, N. Mladenović, J. A. Moreno-Perez, *Variable neighbourhood search: methods and applications*, 4OR **6**(4) (2008), 319–360.

[90] P. Hansen, N. Mladenović, J. A. Moreno Pérez, *Variable neighbourhood search: methods and applications*, Ann. Oper. Res. **175** (2010), 367–407.

[91] P. Hansen, N. Mladenović, D. Perez-Britos, *Variable neighborhood decomposition search*, J. Heuristics **7**(4) 2001, 335–350.

[92] P. Hansen, N. Mladenović, R. Todosijević, S. Hanafi, *Variable neighborhood search: Basics and variants*, EURO J. Comput. Optim. **5**(3) (2017), 423–454.

[93] T. Harada, E. Alba, *Parallel genetic algorithms: a useful survey*, ACM Comput. Surv. **53**(4) (2020), 1–39.

[94] A. Herrán, J. M. Colmenar, R. Martí, A. Duarte, *A parallel variable neighborhood search approach for the obnoxious p-median problem*, Int. Trans. Oper. Res. **27**(1) (2020), 336–360.

[95] K. Holmqvist, A. Migdalas, P. M. Pardalos, *Parallelized heuristics for combinatorial search*, In: A. Migdalas, P. M. Pardalos, S. Storøy (eds.), *Parallel Computing in Optimization*, 269–294, Springer, Boston, MA, USA, 1997.

[96] H. H. Hoos, T. Stützle, *Empirical analysis of randomised algorithms*, In: T. Gonzalez (ed.), *Handbook of Approximation Algorithms and Metaheuristics*, 225–242, Chapman and Hall/CRC, Boca Raton, FL, USA, 2018.

[97] T. C. Hu, *Parallel sequencing and assembly line problems*, Oper. Res **9**(6) (1961), 841–848.

[98] S. Huber, M. J. Geiger, *Order matters–a variable neighborhood search for the swap-body vehicle routing problem*, Eur. J. Oper. Res. **263**(2) (2017), 419–445.

[99] T. Jakšić Kruger, T. Davidović, V. Jelisavčić, *Parameter analysis of variable neighborhood search applied to multiprocessor scheduling with communication delays*, In: Y¿ Kochetov, A. Eremeev, O. Khamisov, A. Rettieva (eds), *Mathematical Optimization Theory and Operations Research: Recent Trends, MOTOR 2022*, Petrozavodsk, Russia, July 2–6, 2022, Commun. Comput. Inf. Sci. **1661**, 104–118, Springer, Cham, 2022.

[100] T. Jakšić Krüger, T. Davidović, D. Teodorović, M. Šelmić, *The bee colony optimization algorithm and its convergence*, Int. J. Bio-Inspired Comput. **8**(6) (2016), 340–354.

[101] J. Jin, T. G. Crainic, A. Løkketangen, *A parallel multi-neighborhood cooperative tabu search for capacitated vehicle routing problems*, Eur. J. Oper. Res. **222**(3) (2012), 441–451.

[102] J. Jin, T. G. Crainic, A. Løkketangen, *A cooperative parallel metaheuristic for the capacitated vehicle routing problem*, Comput. Oper. Res. **44** (2014), 33–41.

[103] Đ. Jovanović, T. Davidović, D. Urošević, T. Jakšić-Krüger, D. Ramljak, *Variable neighborhood search approach to community detection problem*, In: I. Georgiev, M. Datcheva, K. Georgiev, G. Nikolov (eds), *Numerical Methods and Applications, NMA 2022*, Borovets, Bulgaria, August 22–26, 2022, Lect. Notes Comput. Sci. **13858**, 188–199, Springer, Cham, Switzerland, 2023.

[104] B. H. H. Juurlink, H. A. G. Wijshoff, *A quantitative comparison of parallel computation models*, ACM Trans. Comput. Syst. **16**(3) (1998), 271–318.

[105] P. Kalatzantonakis, A. Sifaleras, N. Samaras, *Cooperative versus non-cooperative parallel variable neighborhood search strategies: A case study on the capacitated vehicle routing problem*, J. Glob. Optim. **78**(2) (2020), 327–348.

[106] P. Kalatzantonakis, A. Sifaleras, N. Samaras, *On a cooperative VNS parallelization strategy for the capacitated vehicle routing problem*, In *International conference on learning and intelligent optimization* (*LION 2019*), Chania, Crete, Greece, May 27–31, 2019, Lect. Notes Comput. Sci. **11968**, 231–239, Springer, Cham, 2020.

[107] J. Kennedy, R. Eberhart, *Particle swarm optimization*, In: *Proceedings of ICNN'95 - International Conference on Neural Networks*, **4**, 1942–1948, Perth, WA, Australia, 1995.

[108] C. Kessler, J. Keller, *Models for parallel computing: Review and perspectives*, PARS-Mitteilungen **24** (2007), 13–29.

[109] S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, *Optimization by simulated annealing*, Science **220**(4598) (1983), 671–680.

[110] M. Knausz, *Parallel Variable Neighbourhood Search for the Car Sequencing Problem*, Technical report, Fakultät für Informatik der Technischen Universität Wien, 2008.

[111] Z. Kokosiński, M. Pijanowski, *Application of parallel and hybrid metaheuristics for graph partitioning problem*, In: G. Nikolov, N. Kolkovska, K. Georgiev (eds.), *Numerical Methods and Applications, NMA 2018*, Lect. Notes Comput. Sci. **11189**, 125–132, Springer, Cham, 2019.

[112] S. A. Kravitz, R. A. Rutenbar, *Placement by simulated annealing on a multiprocessor*, IEEE T. Comput. Aid. D. **6**(4) (1987), 534–549.

[113] L. Kumar, M. Pandey, M. K. Ahirwal, *A survey on parallel nature inspired algorithms*, Wireless Pers. Commun. **138**(3) (2024), 1893–1918.

[114] D. Lei, X. Guo, *A parallel neighborhood search for order acceptance and scheduling in flow shop environment*, Int. J. Prod. Econ. **165** (2015), 12–18.

[115] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures, Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA, USA, 1992.

[116] M. M. S. Leite, T. Pinto, C. Alves, *Smart and parallel general variable neighborhood search for the pollution-routing problem*, In: Y. Dessouky, A. Shamayleh (eds.), *50th International conference on computers & industrial engineering CIE50: Sustainable digital transformation*, 1503-1512, Sarajah, UAE, October 30 – November 2, 2023, Computers & Industrial Engineering, Los Angeles, CA, USA, 2023.

[117] S. Loudni, P. Boizumault, *Solving constraint optimization problems in anytime contexts*, In: G, Gottlob, T. Walsh (eds.), *Proceedings of the 18th International joint conference on artificial intelligence, IJCAI-03*, 251–256, Acapulco, Mexico, August 9–15, 2003, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[118] P. Lučić, D. Teodorović, *Bee system: modeling combinatorial optimization transportation engineering problems by swarm intelligence*, In: *Preprints of the TRISTAN IV Triennial symposium on transportation analysis*, 441–445, June 13–19, 2001, Sao Miguel, Azores Islands, 2001.

[119] P. Lučić, D. Teodorović, *Computing with bees: Attacking complex transportation engineering problems*, Int. J. Artif. Intell. T. **12**(3) (2003), 375–394.

[120] P. Lučić, D. Teodorović, *Vehicle routing problem with uncertain demand at nodes: The bee system and fuzzy logic approach*, In: J. L. Verdegay (ed.), *Fuzzy Sets Based Heuristics for Optimization*, 67–82, Physica Verlag, Berlin, Heidelberg, 2003.

[121] Madhuri, K. Deep, *A state-of-the-art review of population-based parallel meta-heuristics*, In: *2009 World Congress on Nature & Biologically Inspired Computing* (*NaBIC*), 1604–1607, Coimbatore, India, December 09–11, 2009, IEEE, New York, USA, 2009.

[122] U. Maleš, D. Ramljak, T. Jakšić-Krüger, T. Davidović, D. Ostojić, A. Haridas, *Controlling the difficulty of combinatorial optimization problems for fair proof-of-useful-work-based blockchain consensus protocol*, Symmetry **15**(1) (2023), 140.

[123] S. T. W. Mara, R. Norcahyo, P. Jodiawan, L. Lusiantoro, A. P. Rifai, *A survey of adaptive large neighborhood search algorithms and applications*, Comput. Oper. Res. **146** 2022, 105903.

[124] R. P. Martin, A. M. Vahdat, D. E. Culler, T. E. Anderson, *Effects of communication latency, overhead, and bandwidth in a cluster architecture*, SIGARCH Comput. Archit. News **25**(2) (1997), 85–97.

[125] L. Matijević, V. Ilin, T. Davidović, T. Jakšić-Krüger, P. M. Pardalos, *General VNS for asymmetric vehicle routing problem with time and capacity constraints*, Comput. Oper. Res. **167** (2024), 106630.

[126] L. Matijević, S. Jelić, T. Davidović, *General variable neighborhood search approach to group steiner tree problem*, Optim. Lett. **17**(9) (2023), 2087–2111.

[127] A. Meade, J. Buckley, J. J. Collins, *Challenges of evolving sequential to parallel code: An exploratory review*, In: *Proceedings of the 12th International workshop on principles of software evolution and the 7th Annual ERCIM workshop on software evolution*, Szeged, Hungary, 1–5, September 5–6, 2011, ACM, New York, USA, 2011.

[128] N. Melab, M. Mezmaz, *Multi and many-core computing for parallel metaheuristics: Editorial*, Concurrency Comput. Pract. Exp. **29**(9) (2017), e4116.

[129] N. Melab, E.-G. Talbi, S. Cahon, E. Alba, G. Luque, *Parallel metaheuristics: Models and frameworks*, In: E.-G. Talbi (ed.), *Parallel Combinatorial Optimization*, 149–162, Wiley Online Library, Hoboken, NJ, USA, 2006.

[130] B.Menendez, E. G. Pardo, J. Sánchez-Oro, A. Duarte, *Parallel variable neighborhood search for the min–max order batching problem*, Int. Trans. Oper. Res. **24**(3) (2017), 635–662.

[131] N. Mladenović, P. Hansen, *Variable neighborhood search*, Comput. Oper. Res. **24**(11) (1997), 1097–1100.

[132] R. E. Morales-Navarro, R. Rivera-Lopez, A. Rodriguez-Leon, M. A. Cruz-Chavez, A. Martinez-Oropeza, *A parallel variable neighborhood search approach for solving the traveling salesman problem*, Int. J. Comput. **1** (2016), 278–283.

[133] J. A. Moreno Pérez, P. Hansen, N. Mladenović, *Parallel variable neighborhood search*, In: E. Alba (ed.), *Parallel Metaheuristics: A New Class of Algorithms*, 247–266, John Wiley & Sons, Hoboken, NJ, USA, 2005.

[134] S. Muelas, A. LaTorre, J.-M. Pena, *A distributed VNS algorithm for optimizing dial-a-ride problems in large-scale scenarios*, Transport Res. C-Emer. **54** (2015), 110–130.

[135] J. Nickolls, W. J. Dally, *The GPU computing era*, IEEE Micro **30**(2) (2010), 56–69.

[136] A. G. Nikolaev, S. H. Jacobson, *Simulated annealing*, In: M. Gendreau, J-Y. Potvin (eds.), *Handbook of Metaheuristics*, 2nd ed., 1–39, Springer, New York, Dordrecht, Heidelberg, London, 2010.

[137] B. Nogueira, E. Tavares, J. Araujo, G. Callou, *Accelerating continuous GRASP with a GPU*, J. Supercomput. **75**(9) (2019), 5741–5759.

[138] L. S. Ochi, M. B. Silva, L. Drummond, *Metaheuristics based on GRASP and VNS for solving traveling purchaser problem*, In: *Proceedings of the 4th Metaheuristics international conference, (MIC 2001)*, 617–621, July 16–20, 2001, Porto, Portugal, 2001.

[139] L. S. Ochi, L. S. Vianna, M. B. da Silva, L. M. de Assumpçao Drummond, *Sequential and parallel metaheuristics based on GRASP and VNS for solving the traveling purchaser problem*, https://www.researchgate.net/profile/Luiz-Ochi/publication/255013935_Sequential_and_Parallel_Metaheuristics_based_on_GRASP_and_VNS_for_Solving_the_Traveling_Purchaser_Problem/links/58dcf6d992851c611d4e7a43/Sequential-and-Parallel-Metaheuristics-based-on-GRASP-and-VNS-for-Solving-the-Traveling-Purchaser-Problem.pdf, 2002. [Accessed: 2025-08-29]

[140] J. L. Ortega-Arjona, *The parallel layers pattern. A functional parallelism architectural pattern for parallel programming*, In: *SugarLoafPLoP2007: Proceedings of the 6th Latin American conference on pattern languages of programming and computing*, May 27–30, 2007, C3, Porto de Galinhas, Pernambuco, Brasil, 2007.

[141] A. Ouali, D. Allouche, S. de Givry, S. Loudni, Y. Lebbah, F. Eckhardt, L. Loukil, *Iterative decomposition guided variable neighborhood search for graphical model energy minimization*, Conference on Uncertainty in Artificial Intelligence, UAI 2017, August 11–15, 2017, 197, Sydney, Australia, 2017.

[142] A. Ouali, S. Loudni, L. Loukil, P. Boizumault, Y. Lebbah, *Cooperative parallel decomposition guided VNS for solving weighted SCP*, In: *Hybrid Metaheuristics*, 9$^{\text{th}}$ International Workshop, HM 2014, Hamburg, Germany, June 11–13, 2014, Lect. Notes Comput. Sci. **8457**, 100–114, Springer, Cham, Switzerland, 2014.

[143] A. Ouali, S. Loudni, L. Loukil, P. Boizumault, Y. Lebbah, *Replicated parallel strategies for decomposition guided VNS*, Electron. Notes Discrete Math. **47** (2015), 93–100.

[144] P. S. Pacheco, M. Malensek, *An introduction to parallel programming*, 2$^{\text{nd}}$ ed., Morgan Kaufmann, San Francisco, CA, USA, 2021.

[145] M. Polacek, S. Benkner, K. F. Doerner, R. F. Hartl, *A cooperative and adaptive variable neighborhood search for the multi depot vehicle routing problem with time windows*, Bus. Res. **1**(2) (2008), 1–12.

[146] O. Polat, *A parallel variable neighborhood search for the vehicle routing problem with divisible deliveries and pickups*, Comput. Oper. Res. **85** (2017), 71–86.

[147] D. J. S. Price, *Little Science, Big Science*, Columbia University Press, New York, NY, 1963.

[148] Z. Quan, L. Wu, *Design and evaluation of a parallel neighbor algorithm for the disjunctively constrained knapsack problem*, Concurrency Comput. Pract. Exp. **29**(20) (2017), e3848.

[149] A. Quelopana, A. Navarra, *Incorporating operational modes into long-term open-pit mine planning under geological uncertainty: An optimization combining variable neighborhood descent with linear programming*, Min. Metall. Explor. **41**(5) 2024, 2769–2782.

[150] A. Quelopana Retamal, *Variable neighborhood descent algorithm to assess technology implementation in open-pit mining systems*, PhD thesis, McGill University, Montreal, QC, Canada, 2024.

[151] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, USA, 1987.

[152] L. Radanović, A. Mijović, D. Urošević, T. Davidović, R. Jovanovic, *General variable neighborhood search for maximum diversity problem with capacity and budget constraints*, Expert Syst. Appl. **267** (2025), 126188.

[153] M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*, Springer, New York, USA, 2012.

[154] C. R. Reeves, *Genetic algorithms*, In: M. Gendreau, J-Y. Potvin (eds.), *Handbook of Metaheuristics*, 2$^{\text{nd}}$ ed., 109–139, Springer, New York, Dordrecht, Heidelberg, London, 2010.

[155] M. G. C. Resende, C. C. Ribeiro, *Greedy randomized adaptive search procedures: Advances, hybridizations, and applications*, In: M. Gendreau, J-Y. Potvin (eds.), *Handbook of Metaheuristics*, 2$^{\text{nd}}$ ed., 283–319, Springer, New York, Dordrecht, Heidelberg, London, 2010.

[156] E. Rios, L. S. Ochi, C. Boeres, V. N. Coelho, I. M. Coelho, R. Farias, *Exploring parallel multi-GPU local search strategies in a metaheuristic framework*, J. Parallel Distrib. Comput. **111** (2018), 39–55.

[157] N. Robertson, P. D. Seymour, *Graph minors. II. Algorithmic aspects of tree-width*, J. Algorithms **7**(3) (1986), 309–322.

[158] D. Rupolo, B. R. P. Junior, J. Contreras, J. R. S. Mantovani, *A new parallel and decomposition approach to solve the medium-and low-voltage planning of large-scale power distribution systems*, Int. J. Elec. Power **132** (2021), 107191.

[159] J. Sánchez-Oro, M. Sevaux, A. Rossi, R. Martí, A. Duarte, *Solving dynamic memory allocation problems in embedded systems with parallel variable neighborhood search strategies*, Electron. Notes Discrete Math. **47** (2015), 85–92.

[160] J. Sanchez-Oro, M. Sevaux, A. Rossi, R. Marti, A. Duarte, *Improving the performance of embedded systems with variable neighborhood search*, Appl. Soft Comput. **53** (2017), 217–226.

[161] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, 1989.
[162] G. Schryen, *Parallel computational optimization in operations research: A new integrative framework, literature review and research directions*, Eur. J. Oper. Res. **287**(1) (2020), 1–18.
[163] M. Sevkli, M. E. Aydin, *Parallel variable neighbourhood search algorithms for job shop scheduling problems*, IMA J. Manag. Math. **18**(2) (2007), 117–133.
[164] A. Udaya Shankar, *Distributed Programming: Theory and Practice*, Springer, New York, USA, 2012.
[165] D. Souravlias, K. E. Parsopoulos, E. Alba, *Parallel algorithm portfolio with market trading-based time allocation*, In: *Operations Research Proceedings 2014: Selected Papers of the Annual International Conference of the German Operations Research Society* (*GOR*), 567–574, RWTH Aachen University, Germany, September 2–5, 2014, Springer, Basel, 2016.
[166] D. Souravlias, K. E. Parsopoulos, I. S. Kotsireas, *Circulant weighing matrices: A demanding challenge for parallel optimization metaheuristics*, Optim. Lett. **10**(6) (2016), 1303–1314.
[167] A. Subramanian, L. M. A. Drummond, C. Bentes, L. S. Ochi, R. Farias, *A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery*, Comput. Oper. Res. **37**(11) (2010), 1899–1911.
[168] Y. Sun, S.-C. Chu, P. Hu, J. Watada, M. Si, J.-S. Pan, *Overview of parallel computing for meta-heuristic algorithms*, J. Netw. Intell. **7**(3) (2022), 656–684.
[169] J. Sánchez-Oro, A. D. López-Sánchez, J. M. Colmenar, *A multi-objective parallel variable neighborhood search for the bi-objective obnoxious p-median problem*, Optim. Lett. **16**(1) (2022), 301–331.
[170] E.-G. Talbi, *Metaheuristics: From Design to Implementation*, John Wiley & Sons, Inc., Hoboken, NJ, USA, 2009.
[171] H. M. M. ten Eikelder, B. J. M. Aarts, M. G. A. Verhoeven, E. H. L. Aarts, *Sequential and parallel local search for job shop scheduling*, In: S. Voss, S. Martello, C. Roucairol, I. H. Osman (eds.), *Meta-heuristics 98: Theory & Applications*, 359–371, Kluwer Academic Publishers, Norwell, MA, USA, 1999.
[172] D. Teodorović, T. Davidović, M. Šelmić, M. Nikolić, *Bee colony optimization and its applications*, In: A. J. Kulkarni, P. Siarry (eds), *Handbook of AI-based Metaheuristics*, 301–322, CRC Press, Boca Raton, FL, USA, 2021.
[173] J. Timmis, P. Andrews, E. Hart, *On artificial immune systems and swarm intelligence*, Swarm Intell. **4**(4) (2010), 247–273.
[174] M. Todorović, L. Matijević, D. Ramljak, T. Davidović, D. Urošvić, T. Jakšić-Krüger, Đ. Jovanović, *Proof-of-useful-work: Blockchain mining by solving real-life optimization problems*, Symmetry **14**(9) (2022), 1831.
[175] R. Todosijević, S. Hanafi, D. Urošević, B. Jarboui, B. Gendron, *A general variable neighborhood search for the swap-body vehicle routing problem*, Comput. Oper. Res. **78** (2017), 468–479.
[176] M. Toulouse, T. G. Crainic, M. Gendreau, *Communication issues in designing cooperative multi thread parallel searches*, In: I. H. Osman, J. P. Kelly (eds.), *Meta-heuristics 98: Theory & Applications*, 501–522, Kluwer Academic Publishers, Dordrecht, Netherlands, 1996.
[177] N. Trifunovic, V. Milutinovic, J. Salom, A. Kos, *Paradigm shift in big data supercomputing: Dataflow vs. controlflow*, J. Big Data **2**(1) (2015), 4.
[178] A. Türkyılmaz, S. Bulkan, *A hybrid algorithm for total tardiness minimisation in flexible job shop: Genetic algorithm with parallel VNS execution* Int. J. Prod. Res. **53**(6) (2015), 1832–1848.
[179] M. G. A. Verhoeven, E. H. L. Aarts, *Parallel local search*, J. Heur. **1** (1995), 43–65.
[180] S. Wagner, L. Mönch, *A variable neighborhood search approach to solve the order batching problem with heterogeneous pick devices*, Eur. J. Oper. Res. **304**(2) (2023), 461–475.
[181] N. Xiong, D. Molina, M. L. Ortiz, F. Herrera, *A walk into metaheuristics for engineering optimization: Principles, methods and recent trends*, Int. J. Comput. Int. Sys. **8**(4) (2015), 606–636.

[182] C. Xu, F. C. M. Lau, *Load Balancing in Parallel Computers: Theory and Practice*, SECS **381**, Springer, New York, USA, 2007.
[183] X.-S. Yang, *Nature-inspired optimization algorithms*, Elsevier, New York, USA, 2014.
[184] M. Yazdani, M. Amiri, M. Zandieh, *Flexible job-shop scheduling with parallel variable neighborhood search algorithm*, Expert Syst. Appl. **37**(1) (2010), 678–687.
[185] X. Zang, L. Jiang, M. Ratli, B. Dingm, *A parallel variable neighborhood search for solving covering salesman problem*, Optim. Lett. **16**(1) (2022), 175–190.
[186] X. Zou, L. Wang, Y. Tang, Y. Liu, S. Zhan, F. Tao, *Parallel design of intelligent optimization algorithm based on FPGA*, Int. J. Adv. Manuf. Tech. **94**(9) (2018), 3399–3412.