Parallelization Strategies for Variable Neighborhood Search

Tatjana Davidović
Teodor Gabriel Crainic

August 2013

CIRRELT-2013-47

Parallelization Strategies for Variable Neighborhood Search

Tatjana Davidović[1], Teodor Gabriel Crainic[2,*]

[1] Serbian Academy of Science and Arts, Kneza mihaila 36, P.O. Box 367, 11001 Belgrade, Serbia

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

Abstract. We analyze five parallelization strategies for the Variable Neighborhood Search (VNS) meta-heuristic. They are based on asynchronous cooperation of several search threads. We tested parallelization on various levels, from a low-level parallel neighborhood exploration, through medium-grained asynchronous execution of basic VNS steps (shaking and local search), to the coarse-grained asynchronous cooperation of various VNS algorithms. We also compared centralized and non-centralized information exchange. Parallel algorithms were implemented on multiprocessor systems containing *q* identical processors. We used two topologies: star configuration for centralized information exchange and processor ring for the non-centralized one. For the experimental evaluation, we applied these strategies to VNS-based procedures for Multiprocessor Scheduling Problem with Communication Delays (MSPCD). We compared the performance of the parallel searches with that of the sequential execution on benchmark problem instances with up to 500 tasks. We achieved not only the improvement of the solution quality but also reduction in execution time. The generality of the proposed strategies and their straightforward implementation make them easily applicable to various difficult combinatorial optimization problems.

Keywords: Asynchronous cooperation, information exchange, multiprocessor scheduling, parallel VNS.

Acknowledgements. This work has been partially supported by the Serbian Ministry of Science, Grant nos. 174010 and 174033. Funding for this work has also been provided by the Natural Sciences and Engineering Council of Canada (NSERC), through its Discovery Grant program. Computational resources were provided by Calcul Québec and Compute Canada. The authors would like to thank Mr François Guertin for his unlimited help with parallel programming and executions.

* Corresponding author: Teodor-Gabriel.Crainic@cirrelt.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec
Bibliothèque et Archives Canada, 2013

1 Introduction

Combinatorial optimization problems are concerned with the selection of the best among finitely many feasible solutions. Each problem is defined by (i) a set of objects, each with an associated contribution, (ii) an objective function computing the value of a particular subset or order of objects, and (iii) the feasibility rules specifying how subsets/orderings may be built. The best (optimal) solution is the one satisfying feasibility rules in such a way that the value of the objective function is the highest/lowest among all possible combinations.

A main difficulty in solving combinatorial optimization problems is that the number of feasible solutions usually grows exponentially with the number of objects in the initial set. Therefore, meta-heuristics are the only practical tool for addressing these problems in real-life dimensions. There are still problems, however, that cannot be treated appropriately in a reasonable amount of time, due either to their complexity or the large size of the real instances. As shown in the recent literature [4], parallelization of search procedures is a promising approach to increase the efficiency of heuristic and meta-heuristic methods. A significant amount of work has been performed in implementing and analyzing parallelization strategies for meta-heuristics, from the low level parallelization realized by distributing elementary computations among processors, up to the cooperative multi-thread parallel search [4, 29, 5, 2].

Variable Neighborhood Search (VNS) is a simple and effective meta-heuristic method [22, 16], that has been widely used to address combinatorial and global optimization problems [18]. The basic idea of VNS is the systematic change of neighborhoods both within a descent phase, to find a local optimum, and a perturbation phase to get out of the corresponding valley. There are several papers proposing parallel versions of VNS [13, 3, 25, 24, 19, 23, 30]. They proposed a very limited number of strategies, however, either independent execution of several VNS algorithms, or medium-grained parallelization of basic VNS steps. We show in this paper that more sophisticated approaches provide better performance than these straightforward parallelization strategies.

We propose various strategies for the parallelization of the VNS meta-heuristic. They are based on asynchronous cooperation of several search threads running on different processors. We developed and compared several "levels" of parallelization strategies, from a low-level parallel neighborhood exploration, through medium-grained asynchronous execution of basic VNS steps (shaking and local search), to the coarse-grained asynchronous cooperation of various VNS algorithms. Centralized and non-centralized information exchange mechanisms were also compared. Our objective is to contribute to the parallelization of the VNS method by proposing new (cooperative) parallelization strategies, and studying the influence of the parallel execution on the performance of the VNS algorithm.

Scheduling is a major issue in many fields, e.g., computer science, operation research, economy, etc. Scheduling problems are NP-complete, even in the simplest forms [14]. We illustrate our methods for constructing parallel VNS algorithms on the static problem of scheduling communicating tasks to homogeneous multiprocessor systems of arbitrary structure, referred to as Multiprocessor Scheduling Problem with Communication Delays (MSPCD) [6, 11, 28]. A mathematical programming formulation of MSPCD is given in [10], and meta-heuristic approaches for this particular variant of the problem are proposed in [11]. To the best of our knowledge, these still represent state-of-the-art results for MSPCD (recent literature focused on different versions of the problem). The results reported in [11] show that for the subset of benchmark test examples proposed in [7] (task graphs with known optimal solutions), the proposed VNS produced solutions whose deviations from the optimum are sometimes more than ten percent. Since MSPCD seems to be hard to deal with, even with meta-heuristics, we aim in this paper to improve the existing results by applying parallel VNS. To the best of our knowledge, parallel meta-heuristics have not been used for this particular variant of the scheduling problem.

The main contributions of this paper therefore are (1) three brand new strategies for the parallelization of VNS; (2) the first application of cooperative VNS to the MSPCD; (3) the performance analysis of various parallel VNS methods; (4) the improvement of the existing results for MSPCD benchmark instances. The paper is organized as follows. The next section contains a brief overview of the VNS meta-heuristic. The review of the recent literature addressing parallelization strategies for various meta-heuristic methods, including VNS, is presented in Section 3. The proposed parallelization strategies for VNS are described in Section 4, while Section 5 contains the implementation details. The experimental evaluation of the different variants of parallel VNS procedures for MSPCD is described in Section 6. We conclude in Section 7.

2 Variable Neighborhood Search

Variable Neighborhood Search (VNS) was proposed by Mladenović and Hansen [22] and has been successively applied to various optimization problems [18]. It is a single-solution neighborhood-based method whose basic building block is a Local Search (LS) procedure. VNS uses multiple neighborhoods in order to increase the efficiency of the search. VNS is based on three simple facts [16]: (1) A local optimum w.r.t. one neighborhood structure is not necessarily the optimum for another; (2) A global optimum is a local optimum w.r.t. all possible neighborhood structures; (3) For many problems, local optima w.r.t. one or several neighborhoods are relatively close to each other.

In order to describe VNS, we first introduce the following notation. Consider an optimization problem, $\min f(x)$, and its sets of *solutions* S, and *feasible solutions* $X \subseteq S$, respectively. Let $x \in X$ be an arbitrary feasible solution. Define the neighborhood $\mathcal{N}(x)$

of $x \in X$ as the set of all solutions obtained from x by the application of a predefined elementary transformation. Different neighborhoods may be obtained by changing the transformation or by applying the same transformation several times.

Let $\mathcal{N}_k$, $k = 1, \ldots, k_{max}$, be a finite set of pre-selected neighborhood structures, and $\mathcal{N}_k(x)$ the set of solutions in the k^{th} neighborhood of x. The basic VNS is given by the pseudo-code illustrated in Figure 1.

```
*Initialization.* Find an initial solution $x \in X$; Improve it with LS
              to obtain $x_{best}$; Choose stopping criterion; Set STOP = 0.

**Repeat**
  1. Set $k = 1$.
  2. **Repeat**
     (a) *Shake.* Generate a random point $x' \in N_k(x_{best})$.
     (b) *Improve.* Run LS with $x'$ as the initial solution;
         Let $x''$ be the corresponding local optimum.
     (c) *Move.* If $f(x'') < f(x_{best})$, move there ($x_{best} = x''$),
         and continue the search within $\mathcal{N}_1$ ($k = 1$);
         otherwise move to the next neighborhood ($k = k + 1$).
     (d) *Stopping criterion.* If stopping condition is met, set STOP = 1.
    **until** $k == k_{max}$ or STOP == 1.
  **until** STOP == 1.
```

Figure 1: Pseudo-code of the VNS meta-heuristic

Usually, the initial solution is determined by some constructive scheduling heuristic and then improved by LS before the beginning of the actual VNS procedure. The role of the *shake* procedure is to prevent trapping in local optima. Intensification of the search is realized by the *improve* step, which invokes the selected LS procedure with the aim to improve the current solution. The entire VNS procedure is focused on the current global best solution and, therefore, a *move* step has to ensure that this solution is always updated as soon as possible. Basic VNS has a unique parameter k_{max} – the maximum number of neighborhoods. Sometimes, but not necessarily, successive neighborhoods are nested. There are several variations and modifications of this basic VNS scheme, and many successful applications. Readers are referred to [17, 18] for further details.

The original, sequential VNS, has a strictly defined order of its basic steps. Moreover, it is designed as a first-improvement procedure, i.e., as soon as a better solution is found, the search is focused on this new solution. Therefore, VNS seems not to provide a natural basis for parallelization. One should expect parallel methods to be quite different from the original VNS. On the other hand, this enables the generation of qualitatively new search algorithms as will be illustrated here by both reviewing the existing parallel VNS methods and evaluating the newly proposed ones.

3 Literature review

In this section, we review the existing results related to the parallelization of the VNS meta-heuristic. First, we point out the main issues related to the parallelization of meta-heuristic methods in general. In the second part of this section, we classify the existing parallelization strategies for VNS.

3.1 Parallelization of meta-heuristics

The main goal of parallelization is to speedup the computations needed to solve a particular problem by engaging several processors and dividing the total amount of work between them. For stochastic algorithms, meta-heuristics in particular, several goals may be achieved [27]: (i) speeding up the search (i.e., reducing the search time); (ii) improving the quality of the obtained solutions; (iii) improving the robustness; (iv) solving large-scale problems. A combination of gains may also be obtained: the parallel execution can enable an efficient search through different regions of the solution space, yielding an improvement of the final solution quality within a smaller amount of execution time.

There exists a significant amount of work concerning the parallelization of meta-heuristics. The approach can be twofold, considering theoretical aspects of parallelization, or developing practical applications of parallel meta-heuristics for different optimization problems. The survey papers [2, 4, 5, 29] summarize these results and propose an adequate taxonomy.

One of the first papers introducing a classification of parallelization strategies is [29]. This classification, based on the control of the search process (thread), resulted in two main groups of parallelization strategies: single walk and multiple walks parallelism. To refine the classification of parallelization strategies, besides the control of the search process, one has to consider communication aspects (synchronous or asynchronous) and search parameters (same or different initial point and/or same or different search strategies). The resulting classification is described in details in [2] and we briefly recall it here in order to be able to adequately classify our parallelization strategies for VNS.

The classification from [2] takes into account three main aspects of parallel execution: search control, communication control, and search differentiation. Such an approach resulted in the 3D-Taxonomy noted $\mathcal{X}/\mathcal{Y}/\mathcal{Z}$. Here, $\mathcal{X}$ is used to describe the *search control cardinality*, and can take two values: centralized ($1C$) or distributed (pC). $\mathcal{Y}$

deals with two aspects of *communication control*: synchronization and type of data to be exchanged. The four possibilities for $\mathcal{Y}$ are Rigid Synchronous (RS), Knowledge Synchronous (KS), Collegial Asynchronous (C), and Knowledge Collegial (KC). *Search differentiation* specifies the part of the search executed by each of the parallel processes. The difference is characterized by the initial point and the search strategy. Each thread can start from the same or a different initial point and can perform the same or a different search procedure. Therefore, there exist four combinations for $\mathcal{Z}$: Same initial Point–Same search Strategy (SPSS), Same initial Point–Different search Strategies (SPDS), Multiple initial Points–Same search Strategy (MPSS), Multiple initial Points–Different search Strategies (MPDS). The implementation of these strategies depends on the given multiprocessor architecture and the characteristics of the problem at hand.

3.2 Existing parallel VNS methods

The authors of [13] were among the first dealing with parallelization of VNS. They proposed and compared three strategies to treat large instances of the p-Median Problem. The first approach was to parallelize LS within a sequential VNS. This low-level parallelism may be classified as 1C/RS/SPSS. The second strategy involved independent runs of several sequential VNS procedures, with the best solution being collected at the end. It is classified as pC/RS/MPSS. The third method applied a synchronous cooperation mechanism through a classical master-slave approach. The master processor ran a sequential VNS in which the basic steps, Shake and LS (SH+LS), were performed in parallel by slave processors. The authors tested their methods using the `tsplib` problem instances with 1400 customers. Not surprisingly, the last two strategies found better solutions, with the third (1C/KS/SPSS) approach using marginally less iterations than the second one.

The p-Median Problem has also been used in [3] for the evaluation of the proposed parallelized VNS algorithms. Besides the independent run from [13], an asynchronous centrally coordinated parallelization strategy (classified as pC/C/MPSS) has been proposed. It was implemented on a master-slave multiprocessor topology, with the master processor playing the role of central (globally accessible) memory, while the slaves performed SH+LS in parallel. The proposed parallel VNS was tested on p-median benchmark problem instances of up to 1000 medians and 11948 customers. The results showed that, for a given time limit, the cooperative parallel method was able to find better solutions than the sequential VNS.

Parallel VNS algorithms for Job Shop Scheduling problems were proposed in [25]. Four parallelization strategies were taken into account: (i) synchronized cooperative strategy proposed in [13]; (ii) asynchronous centrally coordinated method from [3]; (iii) *Noncentral parallelism via unidirectional ring topology*; (iv) *Noncentral parallelism via bidirectional ring topology*. The later two strategies, classified as pC/C/MPSS, were proposed for the first time in [25]. In *Noncentral parallelism via unidirectional ring topology*, each processor was executing a single pair 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. *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 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 [1].

Two cooperation schemes based on central memory mechanism for parallelization of VNS were proposed in [24]. They were tested on the Multi Depot Vehicle Routing Problem with Time Windows. In both schemes the extension of the parallelization strategy from [3] was used. Each slave had to search through a certain number of neighborhoods. In the fine-grained cooperation scheme (pC/C/MPSS), the search in a single cycle did not necessarily include the whole set of neighborhoods. In the coarse-grained cooperation scheme, however, the number of iterations performed by each 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 the execution. This adjustment resulted in a pC/KC/MPSS classification of the coarse-grained cooperation scheme. For the experimental evaluation, the cooperative execution with up to 32 search threads was compared with the sequential procedure and with 32 independent runs. The efficiency of both cooperation schemes was verified by the runtime scalability.

Parallel VNS for the Car Sequencing Problem was developed in [19]. Time Restricted LS (TRLS) was incorporated into Randomized Variable Neighborhood Descent (VND), where randomized meant that the order of neighborhoods was not fixed. Several iterations of TRLS in different neighborhoods were performed in parallel, then the processes were synchronized, the best solution identified and propagated to the next TRLS phase. The described parallel VNS falls into the 1C/KS/SPDS class. Computational tests showed that a substantial reduction of the computation time was achieved. Since no "perfect" neighborhood ordering can be identified in advance, the parallel self-adaptive approach was proposed to obtain good solutions.

Several VNS instances running on different processors and exchanging the best solution after a number of iterations were used in [23] for tackling the Periodic Vehicle Routing Problem with Time Windows. The main aim of implementing this pC/KS/MPDS parallel VNS was to increase the quality of the final solution within the same amount of CPU time as required by the sequential VNS. In the second version of parallel VNS, an Integer Linear Programming solver was used to improve the best solution after the communication. The experimental results showed that the hybrid version improved the quality of the final solution in 80% of cases.

A parallel VNS algorithm was used in [30] to increase the exploration of the search space for the Flexible Job-Shop Problem (FJSP). The proposed algorithm was comprised of external and internal loops. The external loop controlled the stopping condition of the algorithm. In the internal loop a number of processors was used to perform a single run of SH+LS, independently, in parallel. This strategy is classified as 1C/RS/SPDS since different neighborhood structures were used. Shaking was always applied to the current best solution. The computational results on 181 benchmark problems of FJSP showed the competitiveness of the proposed algorithm to the similar methods from the relevant literature.

Table 1: Summary of the existing parallelization strategies for VNS

Ref.	Strategies	Classification	Details
[13]	PLS in seqVNS	1C/RS/SPSS	low level
	IVNS	pC/RS/MPSS	independent execution
	SH+LS in parallel	1C/KS/SPSS	for same k synchronous
[3]	IVNS	pC/RS/MPSS	independent execution
	SH+LS in parallel	pC/C/MPSS	for random k asynchronous
[25]	SH+LS in parallel	1C/KS/SPSS	for same k synchronous
	SH+LS in parallel	pC/C/MPSS	for random k asynchronous
	SH+LS non-centralized ring	pC/C/MPSS	asynchronous
	SH+LS non-centralized mesh	pC/C/MPSS	asynchronous
[24]	SH+VND in parallel	pC/C/MPSS	fine grained cooperation
	several iterations of SH+VND	pC/KC/MPSS	coarse grained cooperation
[19]	SH+RVND in parallel	1C/KS/SPDS	random neighborhood subset
[23]	several VNS in parallel	pC/KS/MPDS	synchronous VNS multisearch
[30]	SH+LS in parallel	1C/RS/SPDS	different neighborhoods for LS

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

4 Parallelization strategies for VNS

In this section, we describe in detail three parallelization strategies that produced (upon implementation) five variants of parallel VNS algorithms. Some of the strategies are modifications of existing ones in the literature, while the others are new and are used for the first time in this paper.

We considered the parallelization of the LS procedure in [8]. We proposed, developed, and compared several parallel LS variants. We incorporated the best performing parallel LS procedure into VNS. The experimental evaluation of such a fine-grained parallel VNS 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). The resulting parallel VNS, named PVNSPLS, represents the first strategy used in this paper. It is classified as 1C/C/SPSS, and it represents the sequential VNS speeded up by parallelizing the most computationally intensive part, the LS procedure. Figure 2 illustrates the block-diagram of PVNSPLS. A similar approach was used as the first strategy proposed in [13].

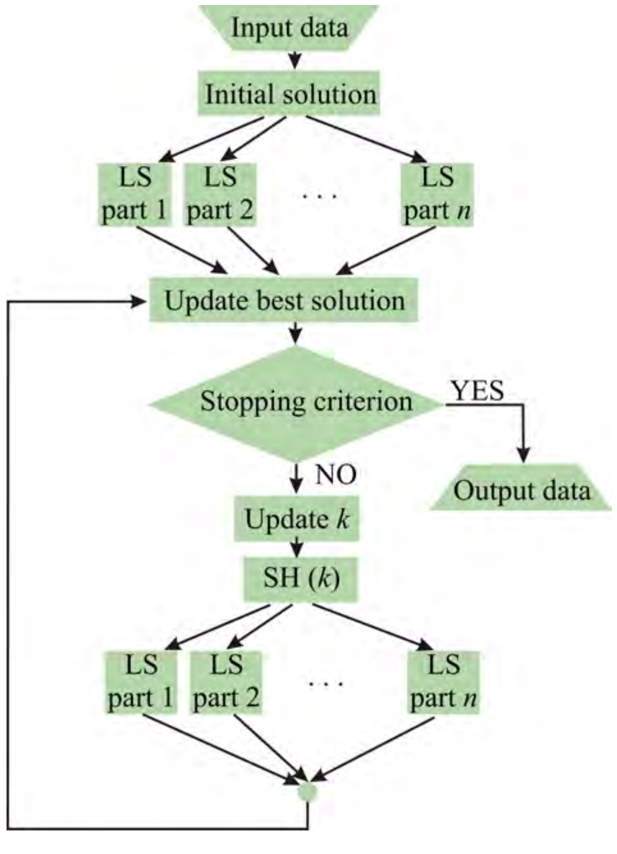

Figure 2: Low-level parallelization of VNS

The next strategy is similar to the one proposed in [3], but our method performs the search in a more systematic way. We refer to it as distributive VNS (DVNS), and classify it as 1C/C/MPSS. The main idea is to explore different neighborhoods in parallel. This is realized by performing the basic VNS steps (SH+LS) by different threads (and consecutively on different processors) at the same time (Figure 3).

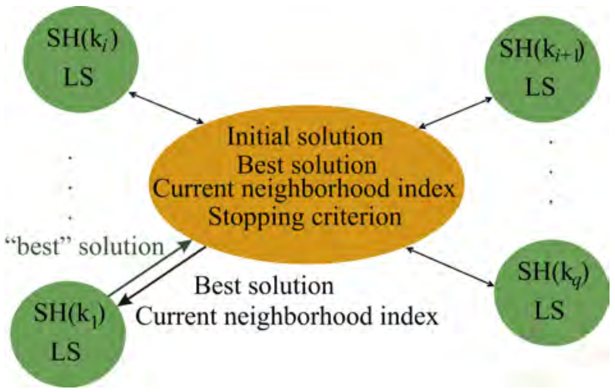

Figure 3: Distributive VNS

Actually, the sequential variant of VNS is performed with multiple executions of the basic steps. The execution of VNS is centrally controlled in order to preserve the original structure of the VNS algorithm. Each thread i shakes the current best solution x_{best} in neighborhood k_i and performs the same sequential LS procedure. Neighborhoods for shaking are assigned to the threads cyclically. At the beginning, neighborhood $\mathcal{N}_i$ is assigned to the thread i, as long as there are threads or neighborhoods available. Let us assume that DVNS is distributed among q threads. In the case $q \leq k_{max}$, the first q neighborhoods are used simultaneously for shaking. If, on the other hand, $q > k_{max}$, we assign $\mathcal{N}_1$ to thread $q+1$, $\mathcal{N}_2$ to thread $q+2$, and so on. Since shaking assumes random selection of a solution from a given neighborhood, a potential duplication of used neighborhoods does not imply redundant executions. The current global best solution x_{best} and a current index k of neighborhood for shaking are kept and updated in the central (global) memory. The access to the central memory is asynchronous and is performed after the LS procedure is completed. The following cases may occur:

1. *The current solution is improved.* A particular thread i improves the current best solution. One should check if this new solution is better than the global best x_{best} from the memory. If this is the case, the memory is updated. The new best solution is written and the current neighborhood index k is set to 1. When x_{best} from the memory is better, it is taken (together with the neighborhood index) as the new current best solution for the thread i. At the same time, the neighborhood index k in the memory is incremented (if a value larger then k_{max} is obtained, it is set to 1 again).

2. *The current solution is not improved.* The later case (taking x_{best} from the memory

and updating the neighborhood index) applies to the situation when thread i is not able to improve the current best solution.

Our main goal is to investigate the performance of the asynchronous cooperative approaches: with several (different, sequential) VNS methods running simultaneously and exchanging relevant information. Keeping in mind that shaking rules and the LS procedures may differ from processor to processor, it is obvious that we can expect significant variations in the time required to complete the basic VNS steps. Therefore, asynchronous execution represents the logical choice. Hence, the third strategy that we propose is asynchronous cooperative VNS (CVNS), illustrated in Figure 4.

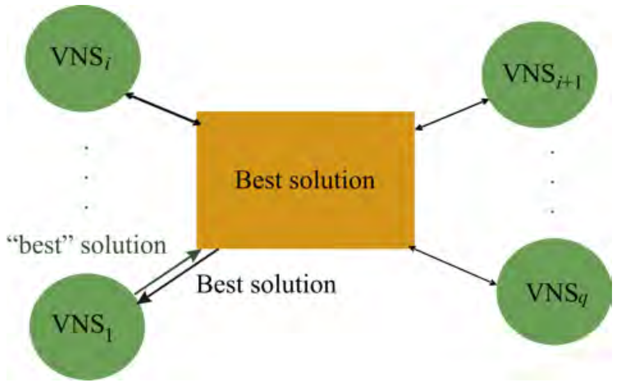

Figure 4: Cooperative VNS

The idea is that each thread runs a variant of the standard (sequential) VNS procedure [11]. At certain points during the computation, the threads communicate with the central memory in order to make the decision about the next step. The appropriate communication points have to be carefully selected through experimental comparison of several possibilities such as:

1. At the end of the "first" LS procedure, before shaking in $k = 1$ when we (possibly) have the new best solution;
2. At each improvement of the best solution;
3. When the neighborhood is to be changed (i.e., at $k++$ point);
4. When $k = k_{max}$, i.e., when the original sequential VNS would start a new iteration (going to $k = 1$ again). The CVNS also runs from $k = 1$ upon the update of the global best solution.

Cases 1 and 2 preserve the original VNS philosophy that the current global best is updated as soon as possible. The remaining two cases should control the diversification of the search performed by each individual thread. The asynchronous information exchanges could take place either through a central or distributed memory. Several variants of CVNS are obtained by varying the selections of communication points and type of information exchange. We distinguish three of them here.

The first variant represents medium-grained parallelization, where data exchange is performed after the completion of each LS procedure, i.e., Case 3. Data exchange is performed through the central memory. Regardless the quality of the obtained local optimum, the best solution already known to the corresponding thread is compared with the current global best solution recorded in the central memory. We named this variant CVNSall.

The second variant represents a mixed parallelization strategy, where for some VNS procedures data exchange is performed as in Case 3, while for the remaining sequential VNSs, access to the memory follows Case 4. This means that the combination of medium and coarse-grained parallelization is exploited. The resulting variant is referred to as CVNSallK.

Finally, the non-centralized variant of CVNSall, named CVNSring is considered. Communications are performed after the LS procedure is completed, but they are realized through several memories, common to the selected subsets of VNS threads. All relevant details about the implementation of the proposed parallel VNS algorithms are described in the next section.

In addition, we also defined the strategy involving the independent execution of various (different) VNS algorithms, named IVNS. We used IVNS for the performance evaluation of our parallel VNS strategies. Usually, parallel algorithms are compared with the best performing sequential algorithm. We went further and compared the parallel executions with the best among all independent runs within the same amount of CPU time. Moreover, if two or more algorithms generate the same solution, the fastest one is used for the comparison. In such a way parallel VNS algorithms have a very hard task to prove their dominance.

5 Implementation

The implementation of the parallel VNS algorithms was performed on a homogeneous multiprocessor system based on SGI Altix computers and the Message Passing Interface (MPI) communication protocol [15], in particular the library for C programming language. We choose a distributed memory multiprocessor system because it is more flexible. Reconfigurable clusters allow us to change the processor interconnection networks and provide a higher scalability in increasing the number of processors than the available multiple-core computers.

The distributed memory multiprocessor systems require the physical transfer of all relevant data between processors. In order to minimize slow communications based on the message-passing protocol, we have to carefully select data structures and minimize the amount of data that should be transferred. The current global best solution is the minimum data that needs to be shared. In some cases (medium level parallelization), the index of the shaking neighborhood is managed globally, and therefore, it should also be transferred between the corresponding processors.

The other relevant factor is the data exchange rate. Frequent communications could yield the intensification of the searches executing on different processors around the same solution. Less frequent data exchanges enforce the autonomy of each search process and assure the exploration of different search regions. On the other hand, less frequent communication changes the original VNS algorithm more and may prevent the search to focus around promising solutions.

We used the multiprocessor networks illustrated in Figure 5 for the implementation of the parallel VNS methods. The star architecture (Figure 5(a)) is used for centrally coordinated search strategies (DVNS, PVNSPLS), cooperation with centralized information exchange (CVNSall, CVNSallK), and independent runs of different sequential VNS algorithms (IVNS). The non-centralized information exchange variant (CVNSring) is implemented on the unidirectional processor ring (Figure 5(b)).

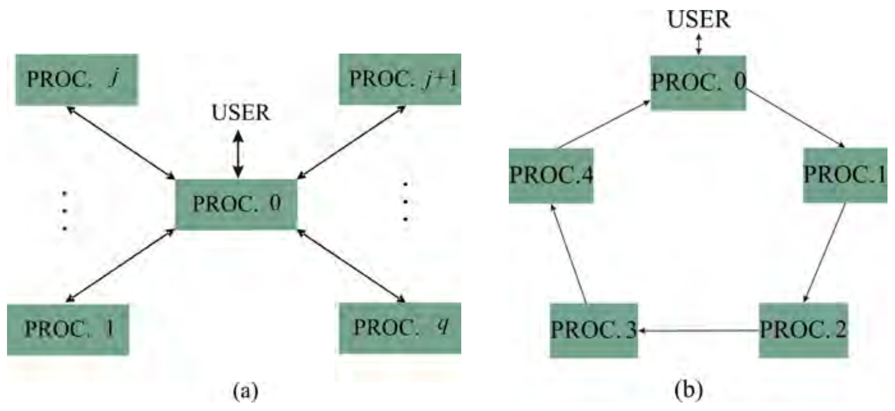

Figure 5: (a) Star architecture; (b) Unidirectional processor ring

In both cases, we distinguish the processor with the identification number zero, declared as *ui*, which is used also for communication with the user. This communication involves reading of input data, error message handling and writing the final results. The actual parallel computations cannot start if an error occurs during the initialization part of the code. If the input data are correctly read and broadcasted among all processors, the START message is sent by the *ui* to all the other processors, and the execution of parallel VNS can begin.

5.1 Low level parallelization of VNS

As already mentioned, low-level parallelization of VNS assumes the sequential execution of the VNS algorithm with the parallelized LS procedure. The star architecture is used to model the master-slave organization of processors. The *ui* processor becomes here the master processor that performs the basic steps of the sequential VNS up to the *improve* part. This part assumes the application of some LS procedure now executed in parallel by the other (slave) processors. As proposed in [8], neighborhood decomposition on all available slave processors is performed. It is executed asynchronously, although controlled by the master.

5.2 Distributive VNS - Parallel variant of single VNS

On the same multiprocessor architecture as the previous one (one master and $q-1$ slave processors), a single VNS is performed in parallel. The master processor controls the execution (manages the current global best solution and the logic of the VNS algorithm, takes care of the stopping condition, etc.). Slaves are engaged in performing SH+LS in parallel. More precisely, the master processor performs the following steps:

1. Generate the initial solution x.
2. Improve it by LS to obtain the current best solution x_{best}.
3. Send x_{best} and the corresponding neighborhood index k to all slaves.
4. Perform a loop that consists of:
   (a) Receive x'' from a slave;
   (b) **if** (x'' is better x_{best}), set $x_{best} = x''$ and $k = 1$; (*move* step),
   else increment k;
   (c) **if** (stopping condition is met), send STOP message to all slaves and exit loop;
   else send x_{best} and k to the corresponding slave.
5. Collect time measuring data from all slaves and print output.

The pseudo-code for a slave processor is:

1. Receive x_{best} and k from the master processor.

2. Perform a loop that consists of:
   (a) Shake the obtained x_{best} in the k-th neighborhood to obtain the new starting solution x';
   (b) Apply LS, starting from x' to obtain x'';
   (c) Send x'' to the master;
   (d) Receive a message from the master processor;
   (e) **if** (STOP message arrived), exit the loop;
   else read x_{best} and k.
3. Send measured time variables to the master.

The SH+LS searches are thus performed in parallel asynchronously for different neighborhoods. "Different neighborhoods" here is related mainly to the Shake step, while the LS procedures are usually the same. This also means that the starting solutions are different sometimes. Namely, as soon as the master processor detects a new global best solution, it forces the search to concentrate around this new solution. This intensification does not involve all slave processors immediately, however. Each slave is allowed to complete its current search before getting new directions. As a consequence, for some of the intermediate solutions, more neighborhoods are explored than it would be the case in the sequential execution of VNS. This makes the main difference between the sequential and parallel VNS.

5.3 Independent execution of different VNS

In the above described variants of parallel VNS, we considered low- and medium-level parallelizations of a single VNS procedure. Varying the parameters of VNS (k_{max}, Shake rules, LS procedures, etc.), we obtain different VNS procedures that can run on different processors simultaneously. The independent execution (without any communications) of different VNS is obtained if we let all VNS run in parallel until the stopping criterion is satisfied on all processors. Then, we collect the obtained results and adopt the best of them as our final solution. We use the star architecture for this execution, although the multiprocessor interconnection network is not relevant here. We declare the final result of independent executions as the best sequential result and use it for the evaluation of all parallel executions.

5.4 Cooperative VNS

As explained in the previous section, cooperation means the simultaneous execution of various sequential VNS procedures (on different processors) with the exchange of relevant data at appropriate times. The data exchange could be centralized or non-centralized. We implemented three variants of CVNS: two with centralized information exchange (CVNSall, CVNSallK) and a variant with non-centralized information exchange (CVNSring).

For the implementation of CVNSall and CVNSallK, we used the star interconnection network of processors (Figure 5(a)). Apart from the user interface role, the processor zero (ui) serves as central memory and does not perform any computations related to the parallel VNS algorithm. More precisely, in both cases, the ui processor performs the following steps:

1. Receive x'' from all processors;
2. Identify the best of them as the current best solution x_{best};
3. Send x_{best} to all processors;
4. Perform a loop that consists of:
   (a) Receive the x'' from a processor;
   (b) **if** (x'' is better than x_{best}) set $x_{best} = x''$ and send x_{best} to all processors;
   (c) **if** (stopping condition is met) send STOP message to all processors and exit loop.
5. Collect time measuring data from all processors and print output to the user.

The role of the ui processor is to perform communications with other processors, to update and store the global best value, and to take care of the stopping condition. Communications are performed asynchronously in such a way that processor ui fills in the mailboxes of the other processors with new (better) solutions and each processor picks up the last message (the one containing the current best solution) whenever it reaches the communication point. At the same time, the other processors are filling in the mailbox of processor ui, which is reading and processing these messages sequentially. After the allowed CPU time is elapsed (stopping criterion is met), ui sends the STOP message to all other processors, collects the time measurement data and reports them, together with the best found solution to the user. Neighborhood management in this variant is performed by processors executing various VNS algorithms.

The pseudo-code for the other processors depends on the CVNS variant. For medium-grained CVNSall, each processor performs the following steps:

1. Generate the initial solution x.

```
2. Improve it by LS to obtain $x''$.

3. Send $x''$ to $ui$.

4. Receive $x_{best}$ from $ui$.

5. Set $k = 1$.

6. Perform a loop that consists of:

    (a) Shake the obtained $x_{best}$ in the $k$-th neighborhood to obtain the new starting
        solution $x'$;
    (b) Apply LS, starting from $x'$ to obtain $x''$,
    (c) Send $x''$ to $ui$;
    (d) Check for a message from $ui$;
    (e) **if** (STOP message arrived) exit the loop;
        **else** read $x_{best}$ and update $k$.

7. Send measured time variables to $ui$.
```

The pseudo-code for processors executing the coarse-grained part of CVNSallK is as follows:

```
1. Generate the initial solution $x$.

2. Improve it by LS to obtain $x''$.

3. Send $x''$ to $ui$.

4. Receive $x_{best}$ from $ui$.

5. Perform a loop that consists of:

    (a) Set $k = 1$;
    (b) **do**
          i. Shake $x_{best}$ in the $k$-th neighborhood to obtain the new starting solution
             $x'$;
         ii. Apply LS, starting from the $x'$ to obtain $x''$;
        iii. **if** ($x''$ is better than $x_{best}$) set $k = 1$, $x_{best} = x''$ and send $x_{best}$ to $ui$;
             **else** increment $k$;
         iv. Check for a message from $ui$;
          v. **if** (STOP message arrived) exit the (outer) loop;
        **while** ($k < k_{max}$);
    (c) Check for a message from $ui$;
    (d) **if** (STOP message arrived) exit the loop;
        **else** read $x_{best}$;
6. Send measured time variables to $ui$.
```

Depending on the problem in hand, the initial solution x is determined either by the application of some constructive heuristics or randomly. Starting from that solution, each processor performs its variant of initial LS procedure. The initial local minimum x'' is sent to ui for the update of the global best solution x_{best}. Upon receiving x_{best} from ui, each processor starts its variant of VNS procedure and performs it until the communication point is reached.

On the improvement of its own best solution each processor immediately informs ui, regardless the variant of CVNS executed. This step supports the first improvement search strategy inherent to the original sequential VNS algorithm. At the same time, the corresponding processor checks for a possibly better solution in the global memory, i.e. on the ui processor.

If no improvement is made after a specified point (after LS or a entire VNS iteration, i.e. after $k = k_{max}$), the corresponding processor is looking for a new global best (that might have been registered by ui meanwhile). If the global best is not improved, it continues the search from the current solution and updates k according to the rules of the corresponding CVNS variant.

As we already described for our distributed memory environment, access to the global memory is realized by message exchange with the ui processor. The asynchronous implementation of global memory access means that processors take into account only the last message from ui, i.e. the one containing the current global best solution detected so far. On the other hand, ui is sending the improved solutions to others as soon as they are detected in the received messages. In such a way, the VNS philosophy "as soon as you find new best solution, go there and start from the beginning ($k = 1$)" is not completely preserved, Step 2 being violated. Asynchronous data exchange, however, allows both frequent exchanges of information and the individuality of each VNS method to make improvements. Stopping all the processors each time the new best solution occurs, would intensify the search around the same solution. Therefore, as in the sequential variant, only a small part of neighborhoods could be explored. In addition, the potentially promising searches that had not been completed yet would be interrupted without the chance to generate potentially good solutions.

The non-centralized data exchange is implemented on a unidirectional ring of processors (Figure 5(b)). Messages are exchanged between the two adjacent processors simulating their common memory: one of them is writing its current global best solution, while the other one is reading the last solution written. Therefore, apart from the user interface part of the processor *ui*, all of the processors perform the same pseudo-code:

```
1. Generate the initial solution $x$.
2. Improve it by LS to obtain $x'' = x_{best}$.
3. Send $x_{best}$ to the next processors.
4. Receive $x'_{best}$ from the previous processor.
5. Compare $x_{best}$ and $x'_{best}$ and set the better one to be the new $x_{best}$.
6. Set $k = 1$.
7. Perform a loop that consists of:
    (a) Shake the obtained $x_{best}$ in the $k$-th neighborhood to obtain the new starting
        solution $x'$;
    (b) Apply LS, starting from the $x'$ to obtain $x''$;
    (c) if ($x''$ is better then $x_{best}$) set $x_{best} = x''$;
    (d) Send $x_{best}$ to the next processors;
    (e) Check for a new message;
    (f) if (STOP message arrived) exit the loop;
        else read $x'_{best}$, update $x_{best}$ and $k$;
8. Send measured time variables to $ui$.
```

To summarize, we implemented and compared six variants of parallel VNS, listed in Table 2. IVNS is used only to identify the best performing sequential VNS to be parallelized into PVNSPLS and DVNS, and for obtaining the best sequential result for the performance evaluation of parallel VNS methods. The comparison results are described in the next section.

6 Experimental evaluations

In this section, we describe and analyze the results of applying the proposed cooperative strategies to the Multiprocessor Scheduling Problem with Communication Delays (MSPCD). Our first objective is to improve the state-of-the-art scheduling results (obtained by the sequential VNS meta-heuristic) for MSPCD, through the exploration of

Table 2: Summary of the tested parallelization strategies for VNS

Strategy	Classification	Details
IVNS	pC/RS/MPDS	independent execution of different VNS
PVNSPLS	1C/C/SPSS	parallel LS in sequential VNS, low level
DVNS	1C/C/MPSS	in sequential VNS, SH+LS in parallel,
		(different k, same LS)
CVNSall	pC/C/SPDS	SH+LS in parallel, (different k, different LS)
CVNSallK	pC/C/SPDS	VNS iteration in parallel, centralized
CVNS	pC/C/MPDS	SH+LS in parallel, non-centralized
		(different k, different LS)

larger part of the solution space. We also aim to reduce the computation time (to speedup the VNS execution) through a better exploration of the solution space. We tested the parallel strategies on benchmark instances provided by Davidović and Crainic [7].

We first give a brief overview of MSPCD and the sequential VNS method, then present our working environment and benchmark test instances, and finally, describe the computational results.

6.1 Problem definition and sequential method

MSPCD represents a static problem of scheduling communicating tasks to homogeneous multiprocessor systems of arbitrary structure. The tasks to be scheduled are represented by a directed acyclic graph (DAG) [6, 20, 26] called Task Graph (TG), defined by a tuple $\mathcal{G} = (T,\ L,\ E,\ C)$. Here $T = \{t_1, \ldots, t_n\}$ stands for the set of tasks; $L = \{l_1, \ldots, l_n\}$ represents the set of task computation times (execution times, lengths). $E = \{e_{ij} \mid t_i,\ t_j \in T\}$ models precedence relation between tasks by a set of communication edges. A task cannot be executed unless all of its predecessors have completed their execution and all relevant data is available. The set of edge communication costs is $C = \{c_{ij} \mid e_{ij} \in E\}$, where $c_{ij} \in C$ stands for the amount of data transferred between tasks t_i and t_j if they are executed on different processors. If both tasks are scheduled to the same processor, the communication cost equals zero. Task preemption and redundant executions are not allowed.

The multiprocessor architecture $\mathcal{M}$ is assumed to contain p identical processors (with their own local memories) that communicate by exchanging messages through bidirectional links of the same capacity. This architecture is modeled by a *distance matrix* [6, 12].

The element $(i,\ j)$ of the distance matrix $D = [d_{ij}]_{p\times p}$ is equal to the minimum distance between the nodes i and j. Here, the minimum distance is calculated as the number of links along the shortest path between two nodes. It is obvious that the distance matrix is symmetric with zero diagonal elements.

The scheduling of DAG $\mathcal{G}$ onto $\mathcal{M}$ consists of determining the index of the associated processor and starting time for each of the tasks from the task graph in such a way as to minimize some objective function. The usual objective function (that we use in this paper as well) is the completion time of the scheduled task graph (also referred to as makespan, response time or schedule length). The starting time of a task t_j depends on the completion times of its predecessors and the amount of time needed for data transferring. Depending on multiprocessor architecture, the time spent for communication between tasks t_i and t_j can be calculated in the following way

$$\gamma_{ij}^{kl} = c_{ij} \cdot d_{kl} \cdot ipc,$$

where it is assumed that task t_i will be executed on processor p_k, task t_j on processor p_l and ipc represents the Iter-Processor-Communication cost defined as the ratio between time for transferring the unit amount of data between two adjacent processors and the time spent for performing single computational operation. If $l = k$ then $d_{lk} = 0$, implying that $\gamma_{ij}^{kl} = 0$.

The MSPCD is NP-Hard and is addressed by meta-heuristic methods [11, 21]. One of the most successful meta-heuristic methods applied to MSPCD is VNS [11]. Therefore, we started from the sequential permutation-based VNS [11] and parallelized it.

The solution representation and neighborhood definitions are the same as in [11]. The solution space S is defined as a set of all permutations of tasks. According to the precedence relations, not all permutations are feasible. Therefore, the search space, the set of *feasible solutions* $X \subseteq S$ is defined as a set of all *feasible permutations*. The same solution representation was used in the exhaustive search presented in [6], the sequential GA, MLS, TS, and VNS proposed in [11], as well as for the implementation of parallel GA from the [21].

A feasible permutation defines the order in which tasks will be allocated to processors. The representation based on feasible permutations is indirect, one still needs to schedule tasks to processors, i.e., to apply some scheduling rule. In our version, the Earliest-Start (ES) scheduling rule [6] is used since it provides the best performance according to the results described in [7, 11].

By presenting a solution of MSPCD as a permutation of tasks, one can explore well-known several neighborhood structures, used in addressing the Traveling Salesman Problem, with the restriction to keep the feasibility of generated neighbors. The neighborhoods used in the sequential method are described in [11]. Here we used swap and interchange neighborhoods and combined them in the Shake and LS phase to obtain various VNS procedures to work in cooperation.

6.2 Working environment

The parallel variants of VNS were implemented in C, on a Linux operating system. We used the MPI communication library for inter-processor communications. Experiments were executed on a 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.

For the experimental evaluation of our parallel VNS methods, we used the benchmark instances proposed in [7]. There are two sets of benchmark task graphs. The first one contains completely random task graphs, while the second set consists of the task graphs with known optimal solutions for a given multiprocessor architecture. All task graphs are available at `http://www.mi.sanu.ac.rs/~tanjad/sched_results.htm`.

We first selected a subset of “hard” instances consisting of ten sparse task graphs with known optimal schedules on a 2-dimensional hypercube (4 processors). The number of tasks varies from 50 to 500 with the increment of 50, while the edge density is around 30% of the maximum allowed density (calculated from the given optimum solution). The values of the optimal schedule length for each instance are given in Table 3, as well as the maximum CPU time (in seconds) allowed for the execution of each variant of (sequential and parallel) VNS. These instances we used for the parameter adjustment described in the next subsection.

Table 3: Optimal schedule lengths and time limits

n	50	100	150	200	250	300	350	400	450	500	av.
SL_{opt}	600	800	1000	1200	1400	1600	1800	2000	2200	2400	1500.0
CPU_{time}	6	70	400	600	1000	2000	4000	6000	10000	16000	4007.6

6.3 Preliminary results

We first describe the preliminary results of applying the parallel strategies we developed to MSPCD. For all experiments, the stopping criterion was the maximum allowed CPU time, the initial solution was obtained by the CPES constructive heuristic [11], and the maximum number of neighborhoods was set to $k_{max} = n/2$.

We identified 5 best performing variants of the sequential VNS and used them for the parallel methods. The results of the independent execution of these 5 sequential VNS procedures are given in Table 4. Two rows are devoted to each instance, the first one containing the best obtained schedule length, while the minimum wall-clock time required to reach this value is given in the second row. The best obtained results are displayed in **bold** characters and repeated in the last two columns of Table 4. The last row of the table contains average values for the 10 instances.

Table 4: Independent execution of best 5 sequential VNS algorithms

no.	SL					best	
tasks	t_{min}					SL	t_{min}
n	VNS1	VNS2	VNS3	VNS4	VNS5		
50	697	**650**	683	695	**650**	650	3.952
	5.988	**3.952**	5.74	4.996	4.736		
100	1036	912	1008	**910**	1057	910	11.28
	38.124	68.616	46.476	**11.28**	57.564		
150	1654	1103	1214	1077	**1062**	1062	129.112
	162.032	76.316	75.988	299.48	**129.112**		
200	2005	**1334**	1685	1532	1542	1334	167.388
	337.736	**167.388**	374.576	255.876	485.476		
250	2407	**1612**	1856	1663	1959	1612	430.216
	64.496	**430.216**	985.208	935.352	370.408		
300	2769	1767	1944	**1764**	2240	1764	1816.436
	1371.864	1651.352	1961.608	**1816.436**	2003.872		
350	3331	2337	3300	**2274**	3301	2274	2152.532
	504.956	2745.068	2362.74	**2152.532**	3087.12		
400	3687	**2435**	2509	2715	2638	2435	2822.912
	5751.9	**2822.912**	3241.008	5926.492	1879.868		
450	2982	**2367**	4147	2757	2433	2367	8004.596
	6927.312	**8004.596**	2938.824	9473.8	6676.644		
500	3039	3066	3160	**2722**	2920	2722	14412.668
	4089.592	5982.76	9597.124	**14412.668**	9410.072		
av. SL	2360.7	1758.3	2150.6	1810.9	1980.2	1713	2995.11

We used up to $q = 20$ processors for the parallel execution of our VNS algorithms. DVNS and PVNSPLS algorithms executed the best performing sequential VNS variant (VNS2). For the CVNS variants, the selected five VNS variants were added cyclically with different seeds each time we use more than five processors were used.

The preliminary results are displayed in Table 5. The first row of Table 5 contains the average (over the ten benchmark instances) best schedule lengths obtained by the

Table 5: Comparison of different cooperation strategies

	SL_{av}				
q	CVNSall	CVNSallK	CVNSring	DVNS	PVNSPLS
1	1713.0	1713.0	1713.0	1713.0	1713.0
t_{min}	2995.11	2995.11	2995.11	2995.11	2995.11
5	1680.1	1733.9	1616.5	1670.1	1746.3
6	1696.2	1669.0	1641.5	1719.3	1669.9
7	1661.6	1706.5	1621.5	1645.7	1709.1
8	1642.9	1666.6	1618.2	1665.2	1686.9
9	1609.5	1670.1	1648.6	1667.4	1701.3
10	1559.2	1618.9	1604.4	1666.7	1705.6
11	1603.6	1568.8	1603.6	1655.6	1710.3
12	1606.6	1632.6	1628.2	1623.4	1671.4
13	1601.8	1643.1	1620.1	1662.4	1701.7
14	1645.5	1671.4	1647.5	1637.3	1667.4
15	1653.7	1677.6	1610.1	1632.1	1635.0
16	1608.9	1621.1	1602.6	1602.9	1703.1
17	1581.2	1611.3	1591.9	1684.5	1676.8
18	1637.5	1606.0	1627.1	1654.4	1693.0
av. SL	1627.74	1649.78	**1620.13**	1656.2	1691.27
av. t_{min}	**2016.87**	2597.20	2455.67	3018.25	3259.20

independent execution of the five best-performing variants of the sequential VNS (IVNS). The IVNS result is therefore better than the best performing sequential algorithm since we always take the best schedule for each instance (regardless of the variant of VNS that produced it). Taking IVNS as the referent sequential result allows us to fully demonstrate the benefits of parallelization. The corresponding average CPU times required to obtain the best solutions, called *minimum time* (t_{min}), are given in the second row. The average (over the ten instances) schedule lengths are presented in the next 14 rows for different numbers of processors. The last two rows contain the average (over the values for q) schedule lengths and the corresponding average t_{min}. These results were presented for the first time in [9].

As can be seen from these results, CVNSring is the best performing parallel VNS. Moreover, within the same amount of wall-clock time, all parallel methods outperform the independent IVNS (and thus, the sequential VNS) with respect to solution quality. In addition, all cooperative methods obtain their best results faster than IVNS. The best results (minimum values for schedule length) are obtained when the parallel VNS is executed on a modest number of processors ($q \approx 10$). This coincides with the conclusion regarding the parallelization of the LS procedure described in [8]. Since our multiprocessor system is different from the one used in [8], it is worth to note that for PVNSPLS the communication time is not negligible. The data transfer between $q = 5$ processors requires 2.5% of the total running time, while in the case when $q = 20$, communication time takes 15% of the total time. In all other cases, the amount of data that is transferred is small and therefore, the communication time becomes negligible.

6.4 General experimentation results

For the rest of the paper, we will focus on CVNSs only, since the preliminary results showed that the coarse-grained cooperative methods perform better than the fine-grained parallel VNS methods.

Two extensive sets of benchmark instances were used for this experimentation. First, a subset of the random test graphs consisting of 36 sparse task graphs ($\rho = 20\%$), 6 graphs for each number of tasks $n = 50, 100, 200, 300, 400, 500$ citeDC04. The second set consisted of 90 graphs with known optimal solutions for a 2D-hypercube, 9 graphs for each number of tasks $n = 50, 100, 150, 200, \ldots, 500$. The time limits for the cooperative VNS methods were set to the values indicated in Table 3 for the graphs of same size.

Table 6: Scheduling results for random task graphs on p=2

	av. t_{min}	av. t_{min} for $q = 5, 6, ..., 20$		
n	seqVNS	CVNSall	CVNSallK	CVNSring
50	0.075	0.046	0.085	0.052
100	5.362	1.296	2.929	2.447
200	48.787	44.126*	62.631**	67.215**
300	410.913	213.259*	339.494**	265.110*
400	1422.706	686.062*	1137.353*	613.302**
500	4236.249	2401.833*	3407.779	3365.006**
av.	1020.682	557.77	825.045	718.855

$*$ - this execution produces results better than the sequential run
$**$ - some of the results are better than the best ones produced by CVNSall

The results of the three cooperative VNS methods to the subset of random task graphs scheduled on $p = 2$ processors are presented in Table 6. We compare the average values of cooperative methods on $q = 5, 6, \ldots, 20$ processors with those of the independent search (IVNS). Since the quality of the final schedule obtained by all cooperative VNS algorithms is always better than or at least equal to the best schedule length obtained by IVNS, Table 6 presents only the average time required to find the best solution, i.e., t_{min}. As can be seen from Table 6, cooperative executions are able to obtain better results faster than the sequential run (i.e., within smaller amount of wall-clock time) in most of the cases. However, the gain in the quality of the final solution (0.01%) does not always justify the number of processors used. The best performance is obtained for $q \leq 10$, while using $q > 15$ processors seems to be pointless.

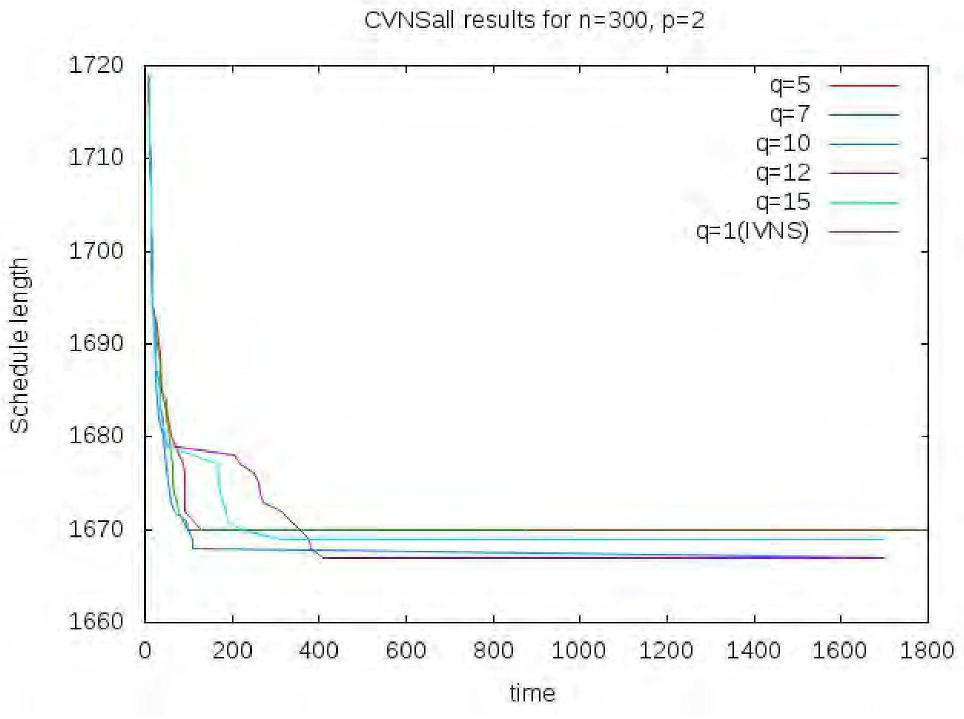

Figure 6: Evolution of the best solution – CVNSall on various numbers of processors

To illustrate this observation, we present in Figures 6-9 the improvement evolution of the best solution for a selected task graph instance for the three cooperative methods with several numbers of processors. Because the smaller task graphs (with up to 200 tasks) are too easy, we selected one of the instances with 300 tasks. The horizontal time line starting at $t_{min} = 883.72$ seconds and ending at 1800.00 seconds displays the referent sequential run (the best out of five independent VNS schedule lengths obtained at the t_{min}). As can be seen from Figures 6-8, all CVNS executions produced solutions of the same quality or better than IVNS within a smaller amount of wall-clock time, approximately 4 times faster regardless the number of processors used. In most of the cases, the best results were obtained for $q = 10$ and $q = 12$.

Figure 9 illustrates the comparison between the three CVNS methods on the same number of processors $q = 10$. We can conclude that, at the beginning of the execution, CVNSall improves the current best solution faster than the other two variants. This is due to the intensive information exchange, i.e., the adoption of the current global best solution after the completion of a single LS procedure. The consequence of this fact, however, is that later on, all search procedures are working around the same current global best solution, which has already been fairly explored. On the other hand, less frequent information exchanges within CVNSallK results in the slowest improvement rate and, sometimes, the chaotic (random-walk like) search of the solution space. In most of the cases, decentralized communication shows the most desirable behavior: smooth descent and the best solution at the end.

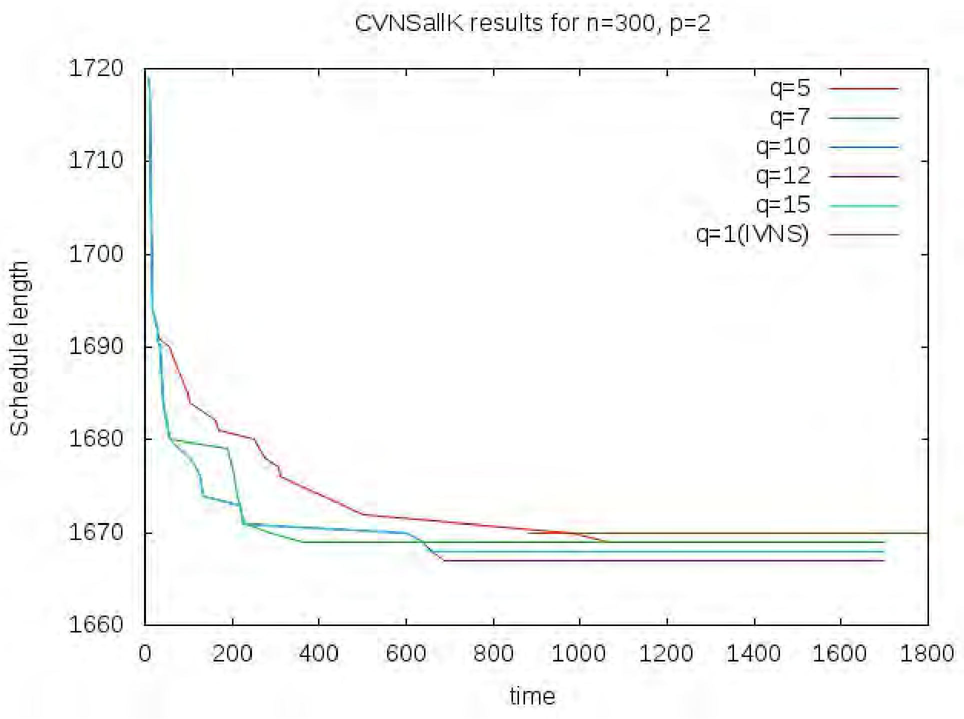

Figure 7: Evolution of the current best solution – CVNSallK on various numbers of processors

For the next experimental step, we increase the problem size by setting $p = 8$. Moreover, we consider an architecture with incomplete connection of processors, namely a 3-dimensional hypercube. These changes make the problem harder, at least for the sequential VNS. The results obtained by cooperative VNS algorithms are presented in Table 7.

The results presented in Table 7 show that cooperation preserves or even improves the solution quality and reduces the required CPU time. The method with more frequent centralized information exchanges (CVNSall) performs better than the variant based on lees frequent communication (CVNSallK) with respect to both solution quality and execution time. For the problem considered, the non-centralized information exchange mechanism outperforms both variants based on centralized communications, generating better solutions very fast.

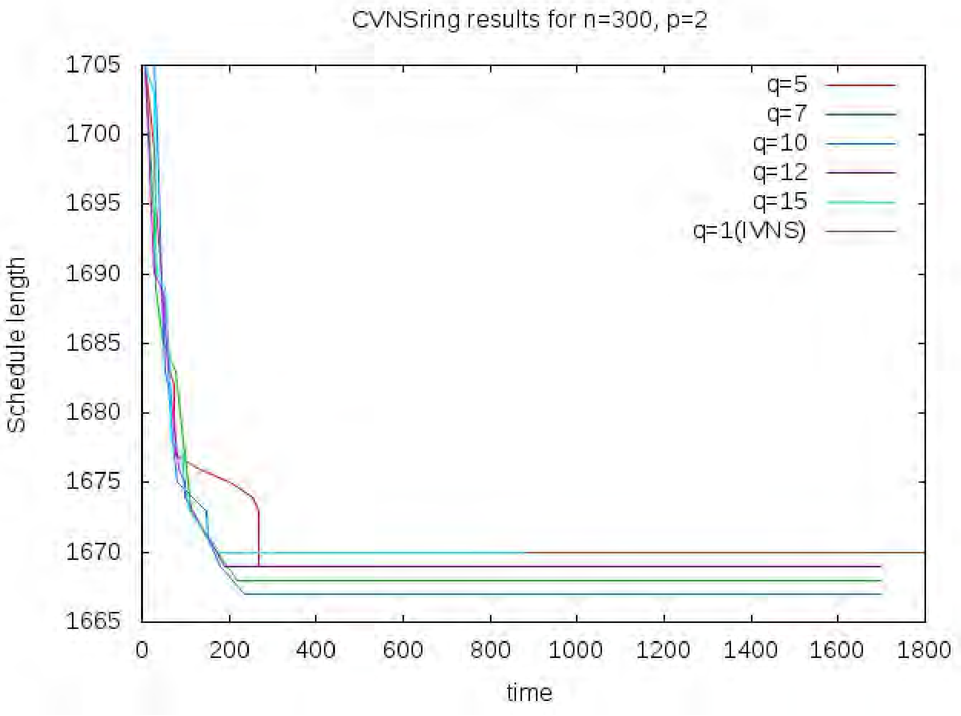

Figure 8: Evolution of the current best solution – CVNSring on various numbers of processors

Table 7: Scheduling results for random task graphs on 3D hypercube, p=8 processors

	av. t_{min}	av. t_{min} for $q = 5, 6, ..., 20$		
n	seqVNS	CVNSall	CVNSallK	CVNSring
50	0.131	0.087	0.112	0.100
100	22.308	8.266	12.781	13.714**
200	186.349	180.002**	160.302*	161.466**
300	558.849	536.169**	530.364*	570.613**
400	2716.662	1821.536*	2173.294*	1863.706**
500	6702.583	3826.659*	7315.885*	3859.508**
av.	1697.814	1062.120	1698.79	1078.184

$*$ - this execution produces results better than the sequential run
$**$ - some of the results are better than the best ones produced by other CVNSs

The final challenge for our CVNS algorithms was to address the set of test instances with known optimal solutions. We limited the experimental evaluation to $q \leq 12$ processors since we already concluded that CVNS algorithms perform the best in this case.

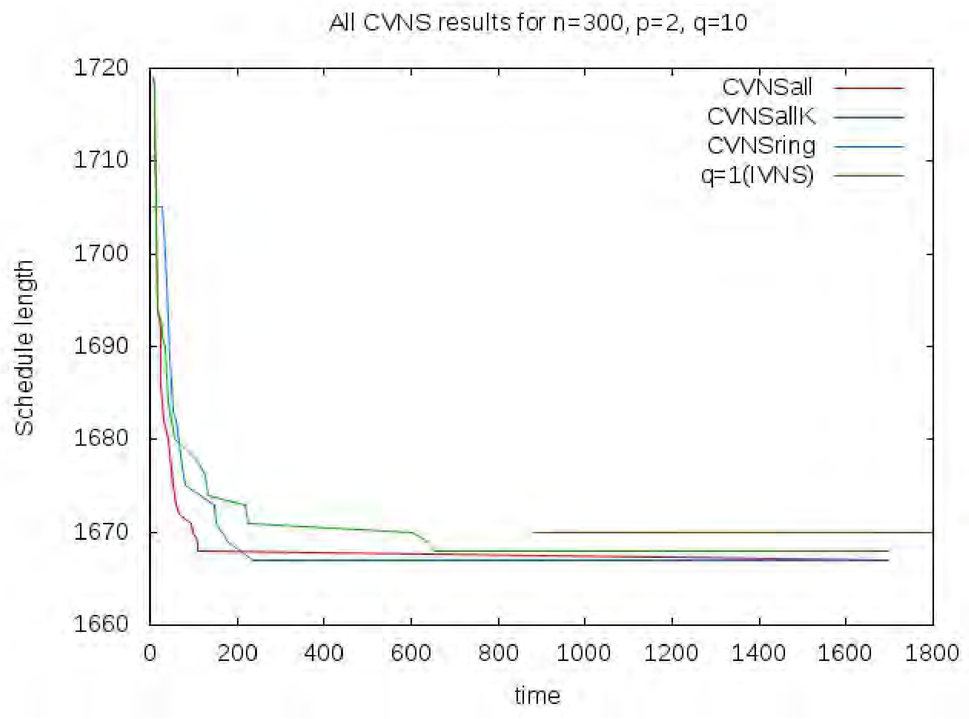

Figure 9: Evolution of the current best solution – all CVNS methods

The results are presented in Table 8 displaying average schedule lengths and average minimum CPU times, for IVNS and the three CVNS methods, over the 9 instances of each size. The figures clearly show that all versions of cooperative VNS obtain better schedules faster than IVNS (and sequential VNS). The best performing CVNS method is again non-centralized CVNS (CVNSring) with respect to both criteria, solution quality and execution time. Regarding the centralized variants, the one with more frequent communication preforms slightly better.

7 Conclusion

Parallel meta-heuristics represent powerful tools for dealing with hard combinatorial optimization problems, especially for large size real-life instances. Therefore, a systematic approach to the design and implementation of parallel meta-heuristic methods is of great importance. The main objective of this work was to propose and evaluate parallelization strategies for the Variable Neighborhood Search meta-heuristic and evaluate them on a very hard combinatorial optimization problem, the Multiprocessor Scheduling Problem with Communication Delays.

Table 8: Scheduling results for task graphs with known optimal solutions on hypercube with p=4 processors and $q = 12$

	IVNS		CVNSall		CVNSallK		CVNSring	
n	av. SL	av. t_{min}	av. SL	av. t_{min}	av. SL	av. t_{min}	av. SL	av. t_{min}
50	618.00	1.98	607.28	1.14	609.83	1.33	609.19	1.11
100	850.89	16.15	846.21	16.93	842.44	13.65	843.94	16.84
150	1066.57	146.81	1064.26	82.70	1063.90	118.98	1057.78	88.49
200	1326.11	101.53	1298.13	175.75	1299.56	158.37	1293.15	168.84
250	1644.22	620.34	1581.97	333.43	1597.19	492.85	1582.89	374.77
300	1821.33	835.83	1796.11	838.31	1805.99	935.34	1782.24	835.70
350	2123.00	2321.68	2071.80	1416.29	2069.69	1854.71	2051.50	1539.31
400	2406.44	3673.86	2350.46	2103.40	2395.93	2648.03	2282.15	2216.28
450	2616.33	6638.79	2490.85	3746.94	2553.83	4996.02	2477.82	3302.23
500	2927.33	7786.67	2798.89	4902.67	2835.21	6752.01	2748.50	4668.28
av.	1740.02	2214.36	1690.60	1361.76	1707.36	1797.13	1672.92	1321.19

We proposed and analyzed several methods based on the distributed memory approach and with centralized and non-centralized information exchange. According to the results of a comprehensive computational study, all cooperative methods outperformed the independent-search method and, thus, the sequential VNS both on solution quality and computational efficiency. New best-know solutions for the test instances considered were thus achieved. The non-centralized asynchronous parallel VNS offers the best performance among the cooperative methods for the problem considered. It provides the largest improvement of the sequential results within the smallest amount of wall-clock time. The shared memory approaches are an interesting avenue for further research.

Acknowledgments

This work has been partially supported by the Serbian Ministry of Science, Grant nos. 174010 and 174033. Funding for this work has also been provided by the Natural Sciences and Engineering Council of Canada, through its Discovery Grant program. Computational resources were provided by Calcul Québec and Compute Canada. The authors would like to thank Mr François Guertin for his unlimited help with parallel programming and executions.

References

[1] M. Aydin and M. Sevkli. Sequential and parallel variable neighborhood search algorithms for job shop scheduling. In F. Xhafa and A. Abraham, editors, *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, pages 125–144. Springer, 2008.

[2] T. G. Crainic and N. Hail. Parallel meta-heuristics applications. In E. Alba, editor, *Parallel Metaheuristics*, pages 447–494. John Wiley & Sons, Hoboken, NJ., 2005.

[3] T. G. Crainic, M. Gendreau, P. Hansen, and N. Mladenović. Cooperative parallel variable neighborhood search for the p-median. *J. Heur.*, 10(3):293–314, 2004.

[4] T.G. Crainic and M. Toulouse. Parallel Meta-heuristics. In M. Gendreau and J.Y. Potvin, editors, *Handbook of metaheuristics*, pages 497–541. Springer, 2010.

[5] V.-D. Cung, S.L. Martins, C.C. Ribeiro, and C. Roucairol. Strategies for the parallel implementations of metaheuristics. In C.C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 263–308. Kluwer Academic Publishers, Norwell, MA, 2002.

[6] T. Davidović. Exhaustive list–scheduling heuristic for dense task graphs. *YUJOR*, 10(1):123–136, 2000.

[7] T. Davidović and 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):2155–2177, Aug. 2006.

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

[9] T. Davidović and T. G. Crainic. MPI Parallelization of Variable Neighborhood Search. In *EURO Mini Conference XXVIII dedicated to Variable Neighborhood Search, (EUROmC-XVIII-VNS)*, pages 241–248, Herceg-Novi, Montenegro, 2012.

[10] T. Davidović, L. Liberti, N. Maculan, and N. Mladenović. Towards the optimal solution of the multiprocessor scheduling problem with communication delays. In *Proc. 3rd Multidisciplinary Int. Conf. on Scheduling: Theory and Application*, Paris, France.

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

[12] G. Djordjević and M. Tošić. A compile-time scheduling heuristic for multiprocessor architectures. *The Computer Journal*, 39(8):663–674, 1996.

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

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completness*. W. H. Freeman and Company, 1979.

[15] W. Gropp and E. Lusk. *Users Guide for* `mpich` *a Portable Implementation of MPI*. University of Chicago, Argonne National Laboratory, 1996.

[16] P. Hansen and N. Mladenović. Variable neighbourhood search. In F. Glover and G. Kochenagen, editors, *Handbook of Metaheuristics*, pages 145–184. Kluwer Academic Publishers, Dordrecht, 2003.

[17] P. Hansen and N. Mladenović. Variable neighbourhood search. In E. K. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 211–238. Springer, 2005.

[18] P. Hansen, N. Mladenović, J. Brimberg, and J. A. Moreno Pérez. Variable neighbourhood search. In M. Gendreau and J-Y. Potvin, editors, *Handbook of Metaheuristics*, pages 61–86. (second edition) Springer, New York Dordrecht Heidelberg London, 2010.

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

[20] Y.-K. Kwok and I. Ahmad. Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *Proc. 7th IEEE Symposium of Parallel and Distributed Processing (SPDP'95)*, pages 36–43, Dallas, Texas, USA, Oct. 1995.

[21] Y.-K. Kwok and I. Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *J. Parallel and Distributed Computing*, 47:58–77, 1997.

[22] N. Mladenović and P. Hansen. Variable neighborhood search. *Comput. & OR*, 24 (11):1097–1100, 1997.

[23] S. Pirkwieser and G. Raidl. Multiple variable neighborhood search enriched with ilp techniques for the periodic vehicle routing problem with time windows. *Hybrid Metaheuristics*, pages 45–59, 2009.

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

[25] M. Sevkli and M.E. Aydin. Parallel variable neighbourhood search algorithms for job shop scheduling problems. *IMA Journal of Management Mathematics*, 18(2): 117–133, 2007.

[26] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel and Distributed Systems*, 4(2):175–187, February 1993.

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

[28] B. Veltman, B. J. Lageweg, and J. K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.

[29] M. G. A. Verhoeven and E. H. L. Aarts. Parallel local search. *J. Heur.*, 1:43–65, 1995.

[30] M. Yazdani, M. Amiri, and M. Zandieh. Flexible job-shop scheduling with parallel variable neighborhood search algorithm. *Expert Systems with Applications*, 37(1): 678–687, 2010.