A parallel heuristic for the Vehicle Routing Problem with Simultaneous Pickup and Delivery

A. Subramanian*,[a], L.M.A. Drummond[a], C. Bentes[b], L.S. Ochi[a], R. Farias[c]

[a] *Universidade Federal Fluminense, Instituto de Computação, Rua Passo da Pátria, 156 - Bloco E, 3º andar, São Domingos, Niterói-RJ, 24210-240, Brasil*
[b] *Universidade Estadual do Rio de Janeiro, Departmento de Engenharia de Sistemas, Rua São Francisco Xavier, 524 - Bloco D, 5º andar, Rio de Janeiro-RJ, 20550-900, Brasil*
[c] *Universidade Federal do Rio de Janeiro, Centro de Tecnologia - Bloco H, sala 319, Cidade Universitária, Rio de Janeiro-RJ, 21945-970, Brasil*

Abstract

This paper presents a parallel approach for solving the Vehicle Routing Problem with Simultaneous Pickup and Delivery (VRPSPD). The parallel algorithm is embedded with a multi-start heuristic which consists of a Variable Neighborhood Descent procedure, with a random neighborhood ordering (RVND), integrated in an Iterated Local Search (ILS) framework. The experiments were performed in a cluster with a multi-core architecture using up to 256 cores. The results obtained on the benchmark problems, available in the literature, show that the proposed algorithm not only improved several of the known solutions, but also presented a very satisfying scalability.

Key words:
Parallel Computing, Metaheuristic, Vehicle Routing, Pickup and Delivery

1. Introduction

The Vehicle Routing Problem with Pickup and Delivery (VRPPD) is one of the main classes of the Vehicle Routing Problem (VRP). Among the several VRPPD variants we can cite: VRP with Backhauls, VRP with mixed Pickup and Delivery; Dial-a-ride Problem; and the VRP with Simultaneous Pickup and Delivery (VRPSPD).

In this work, we deal, particularly, with the VRPSPD. This problem arises especially in the Reverse Logistics context. Companies are increasingly faced with the task of managing the reverse flow of finished goods or raw-materials. According to de Brito and Dekker [1] one can affirm that companies get involved

*Corresponding author
Email addresses: anand@ic.uff.br (A. Subramanian), lucia@ic.uff.br (L.M.A. Drummond), cristianabentes@gmail.com (C. Bentes), satoru@ic.uff.br (L.S. Ochi), rfarias@gmail.com (R. Farias)

with Reverse Logistics because they can profit from it (economics); or they are obliged (legislation); or they feel socially impelled (corporate citizenship).

The VRPSPD can be defined as follows. Let $G = (V, E)$ be a complete graph with a set of vertices $V = \{0, ..., n\}$, where the vertex 0 represents the depot ($V_0 = \{0\}$) and the remaining ones the clients. Each edge $\{i, j\} \in E$ has a non-negative cost c_{ij} and each client $i \in V - V_0$ has a demand q_i for delivery and p_i for pickup. Let $C = \{1, ..., v\}$ be the set of homogeneous vehicles with capacity Q. The VRPSPD consists in constructing a set up to v routes in such away that: (i) all the pickup and delivery demands are accomplished; (ii) the vehicle's capacity is not exceeded; (iii) a client is visited by only a single vehicle; (iv) the sum of costs is minimized.

Since the VRPSPD is a $\mathcal{NP}$-hard problem [2], heuristics methods have proved to be more suitable for dealing with such problem in terms of solution quality vs. computational cost. Even though there are very efficient heuristics for combinatorial optimization problems, they sometimes may have the drawbacks of lack of robustness and also consume considerable amount of time to obtain acceptable quality solutions for complex large-scale problems. Some ideas have been proposed to reduce these limitations, namely the incorporation of learning modules to increase the robustness and parallel versions to decrease the computational time without compromising the quality of the solutions.

In this paper, we present a parallel algorithm based on the sequential heuristic developed by Subramanian *et al.* [3]. The main features of the proposed approach are the automatic calibration of some parameters and the ability of exploring the high-level of parallelism inherent to recent multi-core clusters. The automatic calibration of parameters makes our algorithm auto-adaptive, avoiding the need of manual tuning. The ability of scaling to a large number of processors allows our algorithm to take advantage of the emerging technology of multi-core architectures. These architectures are an attractive option in high performance computing, as the aggregation of a significant number of processors stimulate the construction of "superclusters" with thousands of cores. The high-degree of parallelism of our algorithm enables us to explore these kind of clusters in order to solve large-size instances of the VRPSPD.

Our parallel algorithm was evaluated on a multi-core cluster with up to 256 cores. Our interests were to evaluate not only the quality of the solutions, but also the performance of the algorithm. In terms of the quality of the solutions, we were capable of improving the best known solution for several benchmark problems proposed in the literature. In terms of performance, we evaluated the speedup and the communication overhead. The results show that our parallel algorithm can take advantage of the increasing number of processors, especially when solving large-size instances. The communication overhead, that is usually a bottleneck in the execution of many applications running on high performance clusters, stayed bellow 15% of the execution time when the number of iterations, to be computed, is more than 8 times the number of processors.

The remainder of this paper is organized as follows. Section 2 enumerates some works related to the VRPSPD as well as those that employed parallel strategies to VRPs. In Section 3 we explain the proposed parallel algorithm.

Section 4 contains the experimental results and a comparison with the ones reported in the literature. Section 5 presents the concluding remarks.

2. Related Works

The VRPSPD was first proposed two decades ago by Min [4]. The author presented a heuristic to solve a real-life problem concerning the distribution and collection of books of a public library. The procedure used to solve this problem involved the following stages: (i) clients are first clustered in such a way that the vehicle capacity is not exceeded in each group; (ii) one vehicle is assigned to every cluster (iii) Traveling Salesman Problem (TSP) is solved for each group of clients.

Very little attention had been given to the problem during the 90's. Halse [5] proposed a two-phase heuristic based on the cluster-first-route-second concept while some insertion heuristics, also capable of solving the VRPSPD with multi-depots, were implemented by Salhi and Nagy [6]. The same authors later developed a heuristic that considers different levels of feasibility [7].

Dethloff [2] proposed an insertion heuristic based on the cheapest feasible criterion, radial surcharge and residual capacity. The author also investigates the relation between the VRPSPD and other VRP variants. Ropke and Pisinger [8] developed a Large Neighborhood Search (LNS) heuristic associated with a procedure similar to the VNS metaheuristic to solve several variants of the VRP including the VRPSPD.

Some evolutionary based metaheuristics were developed for the problem. Vural [9] proposed two Genetic Algorithms, where the first one is inspired on the Random Key Method while the second one consists in an improvement heuristic that applies Or-*opt* movements. Gokçe [10] developed an Ant Colony algorithm divided in four steps: (i) a candidate list is formed for each customer; (ii) a feasible solution is found and initial pheromone trails on each arc is calculated using it; (iii) routes are constructed and the pheromone trails are modified by both local and global pheromone update; (iv) the routes are improved using the 2-*opt* movement. Gajpal and Abad [11] also developed an Ant Colony heuristic which has two main steps: (i) the trail intensities and parameters are initialized using an initial solution obtained by means of a nearest neighborhood constructive heuristic; (ii) an ant-solution is generated for each ant using the trail intensities, followed by a local search in every ant-solution and updating of the elitist ants and trail intensities.

Most of the heuristics developed for the VRPSPD are based on pure or hybrid versions of the Tabu Search metaheuristic. Crispim and Brandao [12] presented a hybrid procedure where TS and the Variable Neighborhood Descent (VND) approach are combined. Montané and Galvão [13] proposed a TS algorithm involving the following neighborhood structures: shift, swap, cross and 2-*opt*. Chen and Wu [14] proposed a local search procedure based on the record-to-record travel approximation and tabu lists. Gribkovskaia *et al.* [15] also implemented a TS algorithm, but for the case where only one vehicle is considered. Bianchessi and Righini [16] suggested some constructive and

local search heuristics as well as a TS procedure that uses a variable neighborhood structure, in which the node-exchange-based and arc-exchange-based movements were combined. More recently, Wassan *et al.* [17] presented a reactive TS with the following neighborhood structures: rellocation of a client, exchanging two clients between two different routes and reversing the route direction (reverse). Zachariadis *et al.* [18] developed a hybrid algorithm which combines the principles of the TS and Guided Local Search metaheuristics.

Subramanian *et al.* [3] proposed an Iterated Local Search (ILS) algorithm which uses a VND approach in the local search phase, while Subramanian and Cabral [19] applied the same metaheuristic to solve the VRPSPD with time limit constraints.

Although heuristic strategies are by far the most employed to solve the VRPSPD, some exact algorithms were also explored in the literature. A branch-and-price approach was developed by Dell'Amico *et al.* [20], in which two different strategies were used to solve the subpricing problem: (i) exact dynamic programming and (ii) state space relaxation. The authors managed to find the optimum solution for instances up to 40 clients. Angelelli and Manisini [21] also developed a branch-and-price algorithm, but for the VRPSPD with time-windows constraints.

Table 1 summarizes the VRPSPD related works mentioned in this section, describing their main contributions and/or approaches.

A considerable number of works related to parallel strategies for the VRP has been developed in the last decade. However, few of them have investigated parallel approaches based on exact search methods. Ralphs *et al.* [22, 23] started to change that scenario by using the VRP as one of the testbeds in the development of their parallel branch-and-cut (price) algorithms.

As a matter of fact, due to the nature of the problem, most parallel methods are inspired in metaheuristics. Table 2 lists some of these applications that include the classical VRP, VRPPD, VRP with Time Windows (VRPTW), VRP with heterogeneous fleet (HVRP), periodical VRP (PVRP), Dynamic VRP (DVRP) and VRP with time limit (VRPTL).

Rochat and Taillard [24] proposed an adaptive memory-based approach for the VRPTW. In their method, each TS process: (i) probabilistically selects one of the good solution tours from the adaptive memory; (ii) constructs an initial solution; (iii) improves it and returns the corresponding tour to the adaptive memory. From this work other ones were produced by refining/varying that original proposal, such as the one presented by Schulze and Fahle [25].

Parallel TS strategies were also applied to solve the VRPPD. Gendreau *et al.* [26] proposed a cooperative parallel TS for real-time routing and vehicle dispatching problems. The objective function minimizes the total travel time for servicing the customers plus penalties for lateness at customer locations. The dynamic problem is addressed as a series of static problems, each one defined at every time a new request arrives. A two-level parallelization scheme was proposed, where in the first level, a cooperating adaptive memory scheme was implemented, while in the second one, each TS process executes over the data produced by the other worker processes. Gendreau *et al.* [27] also treated the

Table 1: VRPSPD related works

Work	**Year**	**Contributions and/or Approach**
Min [4]	1989	First Work Case study in a public library
Halse [5]	1992	Cluster-first, route-second strategy 3-opt procedure
Salhi and Nagy [6]	1999	Insertion based heuristics
Dethloff [2]	2001	Constructive heuristic based on cheapest insertion, radial surcharge and residual capacity
Angelelli and Mansini[21]	2001	Branch-and-price for the VRPSPD with TW
Vural [9]	2002	Genetic Algorithm
Gokçe [10]	2003	Ant Colony
Ropke and Pisinger [8]	2004	Large Neighborhood Search
Nagy and Salhi [7]	2005	Heuristics with different levels of feasibility
Crispim and Brandao [12]	2005	TS + VND
Dell'amico *et al.* [20]	2005	Branch-and-price based on dynamic programming and state space relaxation
Chen and Wu [14]	2006	Record-to-record travel + Tabu Lists
Montané and Galvão [13]	2006	TS Algorithm
Gribkovskaia *et al.* [15]	2006	TS for the VRPSPD with a single vehicle
Bianchessi and Righini [16]	2007	Constructive and Local Search Heuristics TS + VNS
Wassan *et al.* [17]	2008	Reactive TS
Subramanian and Cabral [19]	2008	ILS Heuristic for the VRPSPD with Time Limit
Subramanian *et al.* [3]	2008	ILS-VND Heuristic
Zachariadis *et al.* [18]	2009	TS + Guided Local Search
Gajpal and Abad [11]	2009	Ant Colony System

deployment problem for a fleet of emergency vehicles and proposed a master-worker TS based on domain decomposition. The master manages global data structures with pre-calculated information about each vehicle and sends reallocation problems to the workers, whose execution time is controlled by fixing the number of iterations in the TS.

Attanasio *et al.* [28] addressed the multi-vehicle dial-a-ride problem and proposed two parallel TS strategies. In the former, each process runs a different TS algorithm from a same initial solution. Once a process finds a new best solution, it broadcasts it and re-initializations searches are launched. This strategy also presents a diversification procedure applied to the first half of the processes, while an intensification is run on the remaining half. In the second strategy, the same TS algorithm from different starting points is executed by each process. Periodically, such processes exchanges information in order to perform a diversification procedure. Caricato *et al.* [29] treated the VRPPD, by proposing a master-worker algorithm, where the master process manages the TS main operations while the workers perform local searches. Once a worker process finds

a new best solution, it sends it to the other processes that re-initialise their searches.

Ochi *et al.* [30, 31] proposed a fully distributed coarse-grained parallel GA for the HVRP and PVRP, where each process evolves a subpopulation and triggers individual migration asynchronously whenever a subpopulation renewal is necessary. Jozefowiez *et al.* [32] employed a coarse-grained parallel GA to address a VRP in which not only the total length of routes is to be minimized, but also the balance of route lengths, that is the difference between the maximal and minimal route lengths. Individual migrations takes place synchronously in a ring topology. Berger and Berkaoui [33] also employed a GA for solving the VRPTW. In their algorithm, two populations are used with different fitness functions. One of them attempts to minimize the total traveled distance while the other tries to minimize the violation of the time window constraints. A master-worker approach was applied, where the master coordinates the genetic operations while the workers executes the reproduction and mutation operators.

Alba and Dorronsoro [34] proposed a fine-grained, cellular parallel GA for solving a variant of VRP in which routes had to be limited by a predefined travel time. The population is arranged in a 2-dimensional toroidal grid, each individual having four neighbors. The algorithms is composed by three main steps: (i) binary tournament selection to choose parents; (ii) crossover executions; (iii) mutation and local search on the offsprings.

Hybrid techniques that combine TS and EA for solving the VRPTW were proposed by Gehring and Homberger [35] and Le Bouthillier and Crainic [36]. In [35] a parallel strategy with two phases was presented. In the first phase, an evolutionary metaheuristic is applied to minimize the number of vehicles. In the second phase, a TS is executed to minimize the total travel distance. Each different process starts at a different point using a set of particular parameters in both phases. Processes cooperates by exchanging solutions asynchronously through a master process, adopting a central-memory concept. In [36], a central memory cooperative mechanism enhanced with strategies to global search was also investigated. The algorithm is composed of TS methods, EA and post-optimization procedures executing in parallel.

Other parallel metaheuristics based on SA, VNS, and AC have also been employed for solving the VRP and its variants. Czech and Czarnas [37] proposed a master-worker SA for solving the VRPTW. The master executes the following steps: (i) the initial solution is sent to the workers; (ii) the annealing temperature schedule is controlled by collecting the best local solution from every worker for each temperature level; (iii) the global best solution is updated. Each worker runs a SA with the same parameters, cooperating with two other workers by exchanging best solutions. Polacek *et al.* [38] focused on parallel algorithms for the multi-depot VRPTW based on VNS. Two approaches were proposed. In the former, VNS processes searches in a limited number of neighborhoods and collaborates by exchanging best solutions via a central memory. When the overall best solution is improved, it is then broadcasted to all workers. In the second strategy, the VNS processes only sends their best solutions to the central memory at regular intervals.

Parallel AC based algorithms were introduced by Doerner *et al.* [39] to solve the VRP. They investigated fine and coarse-grained parallelizations with synchronous communications. In the fine-grained approach, the ant colony is partitioned into small subcolonies and a saving-based heuristic is executed on each of them. Concerning the coarse-grained method, an independent multi-colony approach and a cooperation scheme were developed.

All those works produced good results at publication time and presented experiments where the parallel version outperformed the sequential one. The reader is referred to the recent survey performed by Crainic [40] for further details concerning the parallel strategies employed.

Table 2: Parallel Metaheuristcs for the VRP

Work	**Year**	**Contributions and/or Approach**
Rochat and Taillard [24]	1995	Adaptive memory-based TS for the VRPTW
Ochi *et al.* [30]	1998	GA based on the island model for the HVRP
Gendreau *et al.* [26]	1999	Cooperative TS for the dispatching DVRP
Schulze and Fahle [25]	1999	TS for the VRPTW
Gendreau *et al.* [27]	2001	Master-worker TS for the DVRP
Drummond *et al.* [31]	2001	GA based on the island model for the PVRP
Jozefowiez *et al.* [32]	2002	GA based on the island model for the VRP with balanced route lengths
Czech and Czarnas [37]	2002	Master-worker SA for the VRPTW
Gehring and Homberger [35]	2002	Evolutionary Alg. and TS for the VRPTW
Caricato *et al.* [29]	2003	TS for the mixed VRPPD
Alba and Dorronsoro. [34]	2004	Fine-grained Cellular GA for the VRPTL
Doerner *et al.* [39]	2004	AC for the VRP
Berger and Berkaoui [33]	2004	Master-worker GA for the VRPTW
Attanasio *et al.* [28]	2004	TS for the Dial-a-ride Problem
Le Bouthillier and Crainic [36]	2005	Cooperative Evolutionary and TS algorithms for the VRPTW
Polacek *et al.* [38]	2006	VNS for the VRPTW with multi-depot

As stated in Table 2, in spite of the considerable number of parallel metaheuristics proposed we are not aware of any parallel ILS algorithm for VRPs. In addition, there are only few works that make use of parallel approaches for solving variants of the VRPPD (see [29], [28], [26]).

3. Parallel Heuristic

This section presents the proposed parallel approach. The parallel algorithm is embedded with a multi-start heuristic which consists of a Variable Neighborhood Descent procedure, with a random neighborhood ordering (RVND), integrated in an ILS framework. This heuristic (ILS-RVND) is an extension of the ILS-VND algorithm proposed in [3].

3.1. Sequential Algorithm

This subsection describes the ILS-RVND heuristic and its main steps are summarized in the Alg. 1. The multi-start heuristic executes $MaxIter$ iterations (lines 4-22), where at each iteration a solution is generated by means of a greedy procedure (line 5). The main ILS loop (lines 8-17) seeks to improve this initial solution using a RVND procedure (line 9) in the local search phase (intensification) as well as a set of perturbation mechanisms (line 15) in the diversification phase. The parameter $MaxIterILS$ represents the maximum number of perturbations allowed without improvements.

Algorithm 1 ILS-RVND

```
1: Procedure ILS-RVND(MaxIter, MaxIterILS, γ, v)
2: LoadData( );
3: f* := ∞;
4: for i:=1,..., MaxIter do
5:    s := GenerateInitialSolution(γ, v, seed);
6:    s' := s;
7:    iterILS := 0;
8:    while iterILS ≤ MaxIterILS do
9:       s := RVND(N(.), f(.), r, s) {r = # of neighborhoods}
10:      if f(s) < f(s') then
11:         s' := s;
12:         f(s') := f(s);
13:         iterILS := 0;
14:      end if;
15:      s := Perturb(s');
16:      iterILS := iterILS + 1;
17:   end while;
18:   if f(s') < f* then
19:      s* := s';
20:      f* := f(s');
21:   end if;
22: end for;
23: return s*;
24: end ILS-RVND.
```

3.1.1. Constructive Procedure

Let v be the number of vehicles (or routes). Each route is filled with a client k, randomly selected from a Candidate List (CL). Next, the remaining clients are evaluated according to eq. (1). Note that every time a client is inserted to the partial solution, the CL must be updated and the insertion costs of the clients re-evaluated. The greedy insertion strategy here adopted was inspired in [2].

$$g(k) = (c_{ik} + c_{kj} - c_{ij}) - \gamma (c_{0k} + c_{k0}) \quad (1)$$

Function $g(k)$ presented in eq. (1) denotes the insertion cost of the client $k \in$ CL in a given route. The value of $g(k)$ is computed by the sum of two parcels. The first computes the insertion cost of the client k between the adjacent clients i and j while the second corresponds to a surcharge used to avoid late insertions of clients located far away from the depot. The cost back and forth from the depot is weighted by a factor γ. The client k that has an insertion cost which leads to the minimum value of g is then added to the solution. The constructive procedure ends when all the clients have been added to the solution.

3.1.2. Local Search

The local search is performed by a Variable Neighborhood Descent [41] procedure which utilizes a random neighborhood ordering (RVND). Preliminary tests showed that this approach led, in average, to better results when compared to the version with deterministic neighborhood ordering adopted in [3].

Six VRP neighborhood structures involving inter-route movements were employed. Five of them are based on the λ-interchanges scheme [42], which consists of exchanging up to λ customers between two routes, while one is based on the Cross-exchange operator [43], which consists of exchanging two segments of different routes.

To limit the number of possibilities we have considered $\lambda = 2$. According to Cordeau and Laporte [44], these exchanges are better described as couples (λ_1, λ_2) (with $\lambda_1 \leq \lambda$ and $\lambda_2 \leq \lambda$) that represent an operation where λ_1 customers are transferred from route 1 to route 2 and λ_2 customers are removed from route 2 to route 1. Therefore, disregarding symmetries, the following combinations are possible in 2-exchanges: (1,0), (1,1), (2,0), (2,1), (2,2). Remark that such combinations include swap moves ((1,1), (2,1), (2,2)) and shift moves ((1,0), (2,0)).

Only feasible movements are admitted, i.e., those that do not violate the maximum load constraints. Therefore, every time an improvement occurs, the algorithm checks whether this new solution is feasible or not. The solution spaces of the six neighborhoods are explored exhaustively, that is, all possible combinations are examined, and the best improvement strategy is considered. The computational complexity of each one of these moves is $\mathcal{O}(n^2)$, where n is the number of clients. Fig. 1 illustrates an example of the effect of each one of the six neighborhood operators over a given solution. Implementation details concerning the $N^{(\eta)}$ neighborhoods ($\eta = 1 \dots 6$) are provided as follows.

Shift(1,0) – $N^{(1)}$ – A client c is transferred from a route r_1 to a route r_2. In Fig. 1.b the client 7 was moved from one route to the other one. The vehicle load is checked as follows. All clients located before the insertion's position have their loads added by q_c (delivery demand of the client c), while the ones located after have their loads added by p_c (pick-up demand of the client c). It is worth mentioning that certain mechanisms can be employed to avoid unnecessary infeasible movements. For instance, before checking the insertion of c in some certain route, a preliminary verification is performed in r_2 to evaluate the vehicle load before leaving, $\sum_{i \in r_2} q_i + q_c$, and when arriving, $\sum_{i \in r_2} p_i + p_c$,

the depot. If the load exceeds the vehicle capacity Q, then all the remaining possibilities of inserting c in this route will be always violated.

Swap(1,1) – $N^{(2)}$ – Permutation between a client c_1 from a route r_1 and a client c_2, from a route r_2. In Fig. 1.d the clients 2 and 6 were swapped. The loads of the vehicles of both routes are examined in the same way. For example, in case of r_2, all clients situated before the position that c_2 was found (now replaced by c_1), have their values added by q_{c_1} and subtracted by q_{c_2}, while the load of the clients positioned after c_1 increases by p_{c_1} and decreases by p_{c_2}.

Shift(2,0) – $N^{(3)}$ – Two adjacent clients, c_1 and c_2, are transferred from a route r_1 to a route r_2. In Fig. 1.e the adjacent clients 7 and 11 were moved from one route to the other one. The vehicle load is tested as in Shift(1,0).

Swap(2,1) – $N^{(4)}$ – Permutation of two adjacent clients, c_1 and c_2, from a route r_1 by a client c_3 from a route r_2. In Fig. 1.f the adjacent clients 6 and 7 were exchanged with client 2. The load is verified by means of an extension of the approach used in the neighborhoods Shift(1,0) and Swap(1,1).

Swap(2,2) – $N^{(5)}$ – Permutation between two adjacent clients, c_1 and c_2, from a route r_1 by another two adjacent clients c_3 and c_4, belonging to a route r_2. In Fig. 1.g the adjacent clients 6 and 7 were exchanged with the adjacent clients 1 and 2. The load is checked just as Swap(1,1).

Cross – $N^{(6)}$ – The arc between adjacent clients c_1 and c_2, belonging to a route r_1, and the one between c_3 and c_4, from a route r_2, are both removed. Later, an arc is inserted connecting c_1 and c_4 and another is inserted linking c_3 and c_2. In Fig. 1.c the arcs (7,11) and (2,8) were removed and the arcs (7,8) and (2,11) were inserted. The procedure for testing the vehicle load is more complex in comparison to the remaining neighborhood structures. At first, the initial (l_0) and final (l_f) vehicle loads of both routes are calculated. If the values of l_0 and l_f do not exceed the vehicle capacity Q then the remaining loads are verified through the following expression: $l_i = l_{i-1} + p_i - q_i$. Hence, if l_i surpasses Q, the movement is infeasible.

Four intra-route neighborhood structures were also implemented. Fig. 2 shows an example of each one of these neighborhood operators. When the function IntraRouteSearch() is called, all of the following movements are exhaustively applied in sequence with a view to further improve the quality of the routes that have been modified.

Or-*opt* – One, two or three adjacent clients are removed and inserted in another position of the route. In Fig. 2.b the adjacent clients 2 and 3 were re-inserted in another position.

2-*opt* – Two nonadjacent arcs are deleted and another two are added in such a way that a new route is generated. In Fig. 2.c The arcs (2,3) and (5,6) were deleted while the arcs (2,5) and (3,6) were inserted.

Exchange – Permutation between two clients. It is an intra-route version of the Swap(1,1). In Fig. 2.d the clients 2 and 6 were swapped.

Reverse – This movement reverses the route direction if the value of the maximum load of the corresponding route is reduced. In Fig. 2.e all the arcs

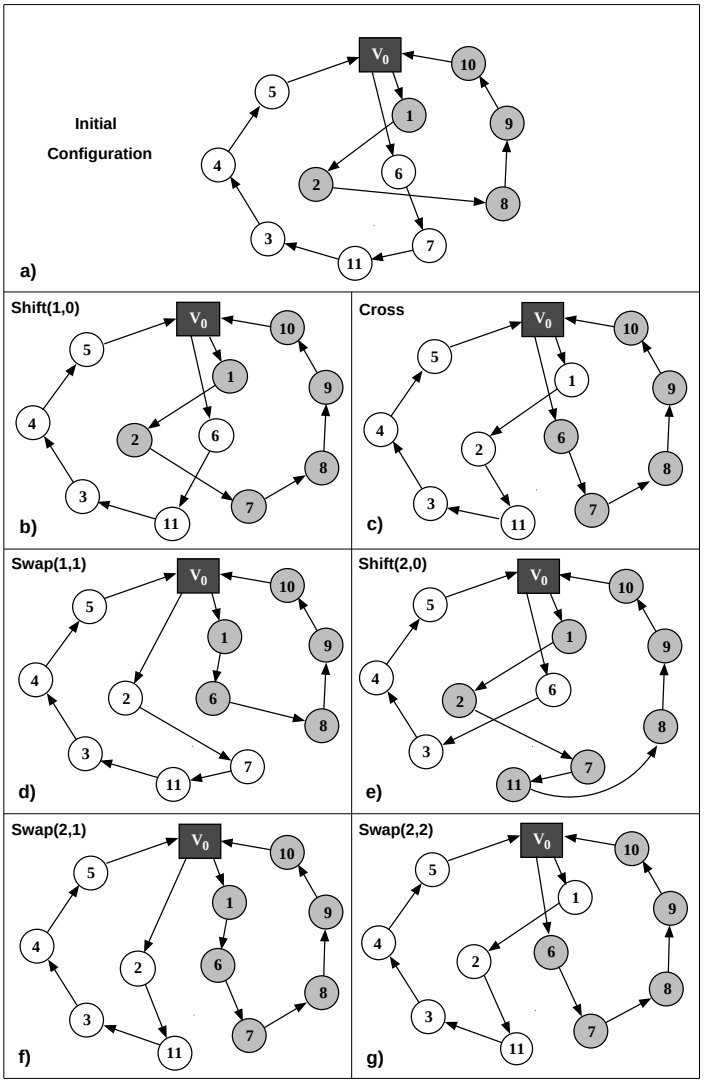

Figure 1: Inter-Route neighborhoods

had their direction reversed.

In these four neighborhood structures, the vehicle load is verified utilizing the same approach of the neighborhood Cross, but it is known in advance that l_0 and l_f never violate the maximum load allowed. The computational complexity of the neighborhoods Or-*opt*, 2-*opt* and Exchange is $\mathcal{O}(n^2)$ while the complexity of the neighborhood Reverse is $\mathcal{O}(n)$.

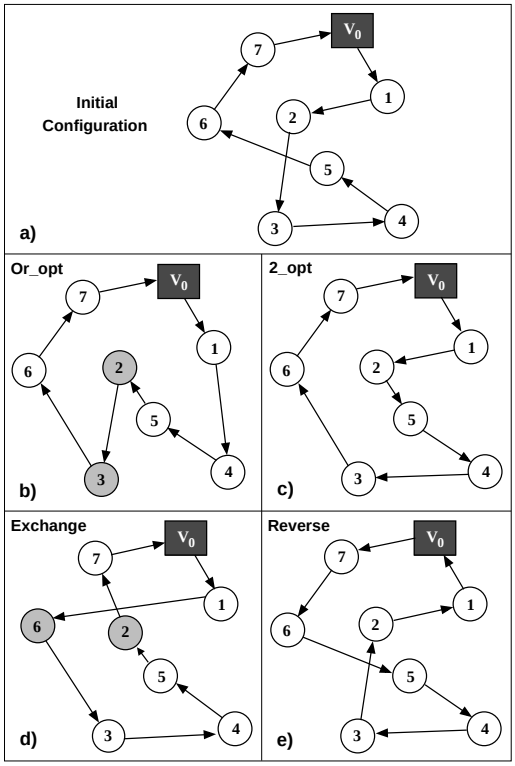

Figure 2: Intra-Route neighborhoods

The pseudocode of the RVND procedure is presented in Alg. 2. Firstly, a Neighborhood List (NL) containing the six inter-route movements is initialized (line 2). In the main loop (lines 3-13), a neighborhood $N^{(\eta)} \in$ NL is chosen at random (line 4) and then the best admissible move is determined (line 5). In case of improvement, an intra-route local search is performed and the NL is populated with all the neighborhoods (lines 6-10). Otherwise, $N^{(\eta)}$ is removed from the NL (line 11).

3.1.3. Perturbation Mechanisms

A set P of three perturbation mechanisms were adopted. Whenever the Perturb() function is called, one of the movements described below is randomly selected.

Ejection Chain – $P^{(1)}$ –This perturbation works as follows. A client from a route r_1 is transferred to a route r_2, next, a client from r_2 is transferred to a route r_3 and so on. The movement ends when one client from the last route r_v is transferred to r_1. The clients are chosen at random and the movement is applied only when there are up to 12 routes. Preliminary tests showed that the application of the Ejection Chain, in most cases, had no effect when there are

Algorithm 2 RVND

```
1: Procedure RVND(N(.), f(.), r, s)
2: Initialize the Neighborhood List (NL);
3: while NL ≠ 0 do
4:    Choose a neighborhood N^(η) ∈ NL at random;
5:    Find the best neighbor s' of s ∈ N^(η);
6:    if f(s') < f(s) then
7:       s := s';
8:       s := IntraRouteSearch(s);
9:       Update NL;
10:   else
11:      Remove N^(η) from the NL;
12:   end if;
13: end while;
14: return s;
15: end RVND.
```

more than 12 routes. If no feasible solution is generated after 50 attempts, the algorithm then tries the Double-Swap perturbation.

Double-Swap – $P^{(2)}$ – Two Swap(1,1) movements are performed in sequence randomly. There is a limit of 50 trials to obtain a feasible solution. However, in our tests, this perturbation was always capable to produce feasible solutions using less then 50 attempts.

Double-Bridge – $P^{(3)}$ – Consists in cutting four edges of a given route and inserting another four to form a new tour. Particularly, we employ an intra-route version of the neighborhood structure Swap(2,2). In principle, this movement is randomly applied in all routes. When there are more then 15 routes, each of them has a 1/3 probability of being perturbed. This prevents exaggerated perturbations which may lead to unpromising regions of the solution space. These values were empirically calibrated. For each route, a maximum of 10 attemps to produce a admissible move are performed.

3.2. Parallel Algorithm

The parallel metaheuristic developed (P-ILS-RVND) is based on the master-worker model. Its steps are described in Alg. 3.

Two learning modules were incorporated into the algorithm as a remedy to the limitation of the sequential heuristic concerning the determination of the number of vehicles and the factor γ. In [3] these parameters were manually calibrated and it was verified that they are highly dependent on each particular instance.

The load balancing strategy is crucial to the performance of P-ILS-RVND, since the time required by a processor to execute a given iteration of the ILS-RVND is not known *a priori* and may vary a lot. Our load balancing strategy is based on dynamic distribution of work by the master. The idea is that the

master performs the administrative role while the workers assume the computational roles.

Each process has an identification number (rank) from 0 to $p-1$, where p is the total number of processes. Our parallel metaheuristic is divided in three main parts, where the first two are related to the automatic calibration of the two parameters mentioned above and the third part is associated with the optimization phase.

The first part of the algorithm computes the number of vehicles (lines 4-12). Let n be the number of clients. Each one of the p process starts with $n-1$ vehicles (line 4) and then the ILS-RVND heuristic is applied aiming to reduce the number of necessary vehicles (line 5). Due to computational cost reasons, this version of the heuristic makes use of just a single perturbation without improvement ($MaxIterILS = 1$). Firstly, the constructive procedure (see Subsection 3.1) of this method enforces each route to receive a client. Next, the movements employed in the local search phase will automatically decrease the number of vehicles needed. The master process receives the number of vehicles found by each worker and broadcasts the smallest value (lines 6-12).

The second part is related to the calibration of the factor γ (lines 13-21). Each process updates the value of γ according to its rank number (line 13). A version with just pure local search, i.e, without perturbations ($MaxIterILS = 0$), of the ILS-RVND heuristic is executed 15 times (line 14). The master process receives the average costs from each worker and broadcasts the three best values of γ (lines 15-21), that is, those ones associated with the best average costs obtained.

The third part of the algorithm deals with the optimization itself (lines 22-40). Every process chooses, at random, one of the three best values for the parameter γ (line 22). The master is in charge of the load balancing by feeding the workers with iterations whenever there is a request from one worker (lines 23-39). The iteration request is sent along with the cost of the last iteration obtained by the respective worker (line 37). In case of improvement, the master updates the *bestcost* and either replies with an iteration (lines 27-28) or with a message informing that there is no more iterations available (lines 32-33). The parameter *iter* denotes the total number of iterations available in the master process and its value is defined according to the number of processes available. The algorithm terminates when the master has no more iterations to provide and all the workers are aware of it.

4. Computational Results

The experiments were performed on a cluster with 32 SMP nodes, where each node consists of 2 Intel Xeon 2.66 GHz quad-core processors that share a 16G RAM. The nodes are connected via Gigabit Ethernet network. All the 256 cores run Linux CentOS, and the communication is handled using the OpenMPI library [45].

The parallel algorithm was coded in C++ and it was tested on benchmark problems proposed by Dethloff [2], Salhi and Nagy [6] and Montané and

Algorithm 3 P-ILS-RVND

```
1: Procedure P-ILS-RVND(iter, MaxIterILS, p)
2: LoadData( );
3: γ := 0;
4: v := N − 1; {v = # of vehicles, n = # of clients}
5: ILS-RVND(1, 1, v, γ); {each process updates v}
6: if master then
7:    receive v from each worker;
8:    send the smallest v to all workers;
9: else
10:   send v to the master;
11:   receive v from the master;
12: end if;
13: γ := updategamma(rank);
14: Execute ILS-RVND(5, 0, v, γ) 15 times and compute the avg. cost;
15: if master then
16:   receive the avg. cost from each worker;
17:   send the 3 best values of γ to all workers;
18: else
19:   send the avg. cost to the master;
20:   receive the 3 best values of γ from the master;
21: end if;
22: choose a γ at random from the 3 best values of γ;
23: if master then
24:   while iter > 0 do
25:      receive an iteration request and the solution cost from any worker;
26:      update bestcost;
27:      send an iteration to the worker who has requested;
28:      iter := iter − 1;
29:   end while;
30:   receive an iteration request and the solution cost from each worker;
31:   update bestcost;
32:   send a msg to all workers informing that there is no more iterations;
33: else
34:   while iter > 0 do
35:      ILS-RVND(1, MaxIterILS, v, γ);
36:      send a msg containing the cost and a request for iteration;
37:      receive a msg informing whether there is iteration from the master;
38:   end while;
39: end if;
40: end P-ILS-RVND.
```

Galvão [13]. In [2] there are 40 instances with 50 clients; in [6] there are 14 instances involving 50 to 200 clients; and in [13] there are 18 instances involving 100 to 400 clients.

Four main points are tackled in this section, namely: parameter tuning, quality of the solutions, performance evaluation (speedup) and communication overhead.

4.1. Parameter Tuning

The main parameters of the P-ILS-RVND algorithm are $iter = K \times (p-1)$, where K is a constant, and $MaxIterILS$. Experiments were performed in order to verify which values of these parameters appear to be more effective. Thus, we have chosen 7 instances with varying sizes (100 to 400 clients), giving preference to those considered "hard" for the P-ILS-RVND, in order to evaluate different configurations and run 20 executions of the algorithm using $p = 256$ processes. Table 3 shows the average gap between the solutions obtained by each configuration and the respective best solution found in the literature, while Table 4 shows the average time, in seconds, of the 20 executions. The gap between the P-ILS-RVND solutions and the literature solutions were computed using eq. (2).

$$gap = \frac{\text{P-ILS-RVND}\_solution - literature\_solution}{literature\_solution} \times 100 \qquad (2)$$

Table 3: Average gap between the solutions obtained by each parameter configuration

K / $MaxIterILS$	Instance							
	r101	sn11x	r1_2_1	rc1_2_1	r1_4_1	c1_4_1	rc1_4_1	Avg.
1.0 / 150	0.00	0.75	0.02	-0.13	-0.25	0.09	-0.18	0.04
1.0 / 200	0.07	0.94	0.11	-0.23	-0.35	0.01	-0.25	0.04
1.0 / 250	0.01	0.75	0.12	-0.25	-0.46	-0.01	-0.34	-0.03
1.0 / 300	0.04	0.66	-0.04	-0.24	-0.48	-0.05	-0.43	-0.08
2.0 / 150	0.04	0.45	0.09	-0.17	-0.31	0.00	-0.18	-0.01
2.0 / 200	0.05	0.62	0.01	-0.22	-0.34	-0.03	-0.30	-0.03
2.0 / 250	-0.04	0.66	-0.06	-0.36	-0.46	-0.04	-0.46	-0.11
2.0 / 300	-0.04	0.67	-0.01	-0.36	-0.54	-0.10	-0.47	-0.12
3.0 / 150	-0.03	0.71	-0.02	-0.26	-0.34	-0.02	-0.31	-0.04
3.0 / 200	-0.04	0.38	-0.03	-0.32	-0.44	-0.03	-0.36	-0.12
3.0 / 250	-0.05	0.46	-0.08	-0.33	-0.57	-0.08	-0.42	-0.15
3.0 / 300	-0.08	0.53	-0.12	-0.34	-0.59	-0.12	-0.46	-0.17

From Table 3 it can be observed that the P-ILS-RVND algorithm had produced, on average, good quality solutions when compared to the literature in the 12 different configurations tested. As expected the configuration that led to better results was the 3.0 / 300, but at the same time it was the one that demanded by far the highest computational effort, as shown in Table 4. We have chosen to adopt the configuration 2.0 / 250 since it was capable of producing, on average, satisfactory solutions in much less computational time.

Table 4: Average time in seconds between the solutions obtained by each parameter configuration

K / $MaxIterILS$	Instance							
	r101	sn11x	r1_2_1	rc1_2_1	r1_4_1	c1_4_1	rc1_4_1	Avg.
1.0 / 150	8.42	11.01	37.10	39.42	243.76	275.30	264.21	125.60
1.0 / 200	11.09	12.92	44.78	47.03	301.83	350.08	335.78	157.65
1.0 / 250	12.25	14.72	50.43	57.49	367.20	428.24	385.89	188.03
1.0 / 300	14.74	16.70	57.58	64.23	417.12	508.75	447.44	218.08
2.0 / 150	11.55	14.89	46.06	51.19	315.51	345.20	345.24	161.38
2.0 / 200	14.29	16.35	56.09	61.42	381.05	457.44	429.77	202.34
2.0 / 250	16.48	20.65	67.86	74.80	463.39	558.61	534.41	248.03
2.0 / 300	18.83	21.69	78.00	86.76	563.83	673.80	618.29	294.46
3.0 / 150	13.78	16.70	56.81	62.56	373.95	440.61	409.95	196.34
3.0 / 200	17.00	20.26	70.74	77.94	483.47	568.87	532.07	252.91
3.0 / 250	19.79	24.31	83.25	92.68	582.44	689.89	636.64	304.14
3.0 / 300	23.90	29.35	99.00	109.81	703.57	821.90	788.44	368.00

4.2. Quality of the Solutions

In this subsection we analyze the quality of the solutions generated by P-ILS-RVND when 128 and 256 cores are used for the computation. As stated in Subsection 4.1, the maximum number of perturbations without improvement ($MaxIterILS$) was 250 while the number of iterations ($iter$) was $iter = 2 \times (p - 1)$. We have executed the algorithm 50 times for each instance.

Although we are not directly interested in the performance of the algorithm in this subsection, one can observe in Tables 5, 6 and 8 that the average execution time (Avg. t) in all instances was higher when 256 cores were considered. This was expected since in the number of iterations to be computed increase with the number of processors used.

Table 5 shows the results obtained in Dethloff's instances. For all cases the P-ILS-RVND, with both 128 and 256 cores, have equaled the best result found in the literature. It is relevant to mention that except for the instances SCA3-0, SCA8-2, SCA8-7 and CON3-2 the average solution (Avg. Sol.) of each instance coincides with the best solution (Best Sol.). The average gap between the Avg. Sols. and the literature solutions for 128 and 256 cores was respectively 0.004% and 0.002%.

The results obtained in Salhi and Nagy's 14 instances are presented in Table 6. When executed in both 128 and 256 cores, the P-ILS-RVND improved 3 results and equaled another 5. The average gap between the Avg. Sols. and the ones found in the literature was 0.80% and 0.75% for 128 and 256 cores respectively. The P-ILS-RVND failed to improve/equal the literature solution of the instances CMT2X, CMT2Y, CMT12X, CMT12Y, CMT11Y and CMT4Y. The average gap between the Best Sols. and the literature solutions for 128 and 256 cores was respectively 0.59% and 0.60%.

Table 7 shows a comparison between the P-ILS-RVND and the algorithms that obtained the best results in Salhi and Nagy's instances, namely those proposed by Chen and Wu (CW) [14], Wassan et al (W). [17], Zachariadis et al (Z). [18] and Subramanian *et al.* (S) [3]. When individually comparing the P-ILS-RVND with each one of these algorithms, one can verify that the P-ILS-

Table 5: Results obtained in Dethloff's instances, showing the number of clients (Clients), the number of vehicles ($\#v$), the best solution found in the literature (Literature), the best solution (Best Sol.), the average solution (Avg. Sol.), and the average execution time in seconds (Avg. t)

Instance	Clients	$\#v$	Literature	P-ILS-RVND (128 cores)			P-ILS-RVND (256 cores)		
				Best Sol.	Avg. Sol.	Avg. t	Best Sol.	Avg. Sol.	Avg. t
SCA3-0	50	4	**635.62**	**635.62**	636.00	2.19	**635.62**	635.92	2.31
SCA3-1	50	4	**697.84**	**697.84**	697.84	2.14	**697.84**	697.84	2.28
SCA3-2	50	4	**659.34**	**659.34**	659.34	2.09	**659.34**	659.34	2.14
SCA3-3	50	4	**680.04**	**680.04**	680.04	2.26	**680.04**	680.04	2.49
SCA3-4	50	4	**690.50**	**690.50**	690.50	2.03	**690.50**	690.50	2.18
SCA3-5	50	4	**659.90**	**659.90**	659.90	2.13	**659.90**	659.90	2.23
SCA3-6	50	4	**651.09**	**651.09**	651.09	2.36	**651.09**	651.09	2.51
SCA3-7	50	4	**659.17**	**659.17**	659.17	2.36	**659.17**	659.17	2.49
SCA3-8	50	4	**719.48**	**719.48**	719.48	2.19	**719.48**	719.48	2.26
SCA3-9	50	4	**681.00**	**681.00**	681.00	1.76	**681.00**	681.00	1.90
SCA8-0	50	9	**961.50**	**961.50**	961.50	3.05	**961.50**	961.50	3.37
SCA8-1	50	9	**1049.65**	**1049.65**	1049.65	2.70	**1049.65**	1049.65	2.89
SCA8-2	50	9	**1039.64**	**1039.64**	1040.03	2.32	**1039.64**	1039.69	2.38
SCA8-3	50	9	**983.34**	**983.34**	983.34	2.83	**983.34**	983.34	2.98
SCA8-4	50	9	**1065.49**	**1065.49**	1065.49	2.69	**1065.49**	1065.49	2.81
SCA8-5	50	9	**1027.08**	**1027.08**	1027.08	3.04	**1027.08**	1027.08	3.31
SCA8-6	50	9	**971.82**	**971.82**	971.82	3.20	**971.82**	971.82	3.51
SCA8-7	50	10	**1051.28**	**1051.28**	1051.49	2.99	**1051.28**	1051.39	3.12
SCA8-8	50	9	**1071.18**	**1071.18**	1071.18	2.79	**1071.18**	1071.18	2.92
SCA8-9	50	9	**1060.50**	**1060.50**	1060.50	2.11	**1060.50**	1060.50	2.18
CON3-0	50	4	**616.52**	**616.52**	616.52	2.96	**616.52**	616.52	3.12
CON3-1	50	4	**554.47**	**554.47**	554.47	2.80	**554.47**	554.47	2.83
CON3-2	50	4	**518.00**	**518.00**	518.23	2.60	**518.00**	518.03	2.77
CON3-3	50	4	**591.19**	**591.19**	591.19	2.31	**591.19**	591.19	2.34
CON3-4	50	4	**588.79**	**588.79**	588.79	2.50	**588.79**	588.79	2.63
CON3-5	50	4	**563.70**	**563.70**	563.70	2.44	**563.70**	563.70	2.69
CON3-6	50	4	**499.05**	**499.05**	499.05	2.55	**499.05**	499.05	2.75
CON3-7	50	4	**576.48**	**576.48**	576.48	2.64	**576.48**	576.48	2.75
CON3-8	50	4	**523.05**	**523.05**	523.05	2.37	**523.05**	523.05	2.46
CON3-9	50	4	**578.25**	**578.25**	578.25	3.10	**578.25**	578.25	3.37
CON8-0	50	9	**857.17**	**857.17**	857.17	3.40	**857.17**	857.17	3.65
CON8-1	50	9	**740.85**	**740.85**	740.85	2.93	**740.85**	740.85	3.02
CON8-2	50	9	**712.89**	**712.89**	712.89	3.02	**712.89**	712.89	3.08
CON8-3	50	9	**811.07**	**811.07**	811.07	3.86	**811.07**	811.07	3.99
CON8-4	50	9	**772.25**	**772.25**	772.25	3.33	**772.25**	772.25	3.69
CON8-5	50	9	**754.88**	**754.88**	754.88	3.88	**754.88**	754.88	4.18
CON8-6	50	9	**678.92**	**678.92**	678.92	3.75	**678.92**	678.92	4.09
CON8-7	50	9	**811.96**	**811.96**	811.96	3.79	**811.96**	811.96	4.03
CON8-8	50	9	**767.53**	**767.53**	767.53	3.32	**767.53**	767.53	3.42
CON8-9	50	9	**809.00**	**809.00**	809.00	3.29	**809.00**	809.00	3.48

RVND produced, on average, superior results.

It is important to point out that Wassan et al. *et al.* [17] may have used another approach to generate the instance CMT1Y. We have found the optimum solution of this instance (466.77) by means of the mathematical formulation presented in [20]. This value is greater than the one obtained by Wassan *et al.* [17](458.96). Note that the optimum solution coincides with the solution found in [3] and by the P-ILS-RVND.

In Montané and Galvão's instances (Table 8), the P-ILS-RVND with either 128 or 256 cores improved the results of 12 out of 18 instances and equaled the other 6. The best solutions found by the version with 256 cores were slightly better than the one with 128. The average solutions follow the same behavior,

Table 6: Results obtained in Salhi and Nagy's instances, showing the number of clients (Clients), the number of vehicles ($\#v$), the best solution found in the literature (Literature), the best solution (Best Sol.), the average solution (Avg. Sol.), and the average execution time in seconds (Avg. t)

Instance	Clients	$\#v$	Literature	P-ILS-RVND (128 cores)			P-ILS-RVND (256 cores)		
				Best Sol.	Avg. Sol.	Avg. t	Best Sol.	Avg. Sol.	Avg. t
CMT1X	50	3	**466.77**	**466.77**	466.77	2.24	**466.77**	466.77	2.28
CMT1Y	50	3	**466.77**	**466.77**	466.77	2.17	**466.77**	466.77	2.27
CMT2X	75	6	**668.77**	684.21	684.54	6.07	684.21	684.49	6.44
CMT2Y	75	6	**663.25**	684.21	684.55	6.19	684.21	684.43	6.41
CMT3X	100	5	**721.27**	**721.27**	721.29	11.81	**721.27**	721.27	12.10
CMT3Y	100	5	**721.27**	**721.27**	721.28	11.77	**721.27**	721.27	12.28
CMT12X	100	5	**644.70**	662.22	662.25	9.87	662.22	662.22	10.29
CMT12Y	100	5*	**659.52**	662.22	662.30	10.31	662.22	662.25	10.76
CMT11X	120	4	838.66	835.26	845.13	17.83	**833.92**	842.78	18.87
CMT11Y	120	4	**830.39**	833.92	844.68	17.61	833.92	842.78	19.03
CMT4X	150	7	**852.46**	**852.46**	852.47	29.62	**852.46**	852.46	30.89
CMT4Y	150	7	**852.35**	852.46	852.46	30.44	852.46	852.46	31.61
CMT5X	199	10	1030.55	**1029.25**	1030.04	67.76	**1029.25**	1029.66	71.50
CMT5Y	199	10	1030.55	**1029.25**	1030.35	65.08	**1029.25**	1029.71	69.58

(*) The best known solution was found with 6 vehicles.

Table 7: Comparison between P-ILS-RVND (256 cores) and literature results in Salhi and Nagy's instances

Instance	CW [14]		W [17]		Z [18]		S [3]		P-ILS-RVND	
	Best Sol.	$\#v$	Best Sol..	$\#v$	Best Sol.	$\#v$	Best Sol.	$\#v$	Best Sol.	$\#v$
CMT1X	478.52	3	468.30	3	469.80	3	**466.77**	3	**466.77**	3
CMT1Y	480.78	3	458.96	3	469.80	3	**466.77**	3	**466.77**	3
CMT2X	688.51	6	**668.77**	6	684.21	6	684.21	6	684.21	6
CMT2Y	679.44	6	**663.25**	6	684.21	6	684.21	6	684.21	6
CMT3X	744.77	5	729.63	5	**721.27**	5	721.40	5	**721.27**	5
CMT3Y	723.88	5	745.46	5	**721.27**	5	721.40	5	**721.27**	5
CMT12X	678.46	6	**644.70**	5	662.22	5	662.22	5	662.22	5
CMT12Y	676.23	6	**659.52**	6	662.22	5	662.22	5	662.22	5
CMT11X	858.57	4	861.97	4	838.66	4	839.39	4	**833.92**	4
CMT11Y	859.77	5	**830.39**	4	837.08	4	841.88	4	833.92	4
CMT4X	887.00	7	876.50	7	**852.46**	7	852.83	7	**852.46**	7
CMT4Y	**852.35**	7	870.44	7	852.46	7	852.46	7	852.46	7
CMT5X	1089.22	10	1044.51	9	1030.55	10	1030.55	10	**1029.25**	10
CMT5Y	1084.27	10	1054.46	9	1030.55	10	1031.17	10	**1029.25**	10

but one can verify that the difference tends to increase with the number of clients. The average gap between the values of the Avg. Sols. and those reported in the literature was -0.12% and -0.16% for 128 and 256 cores respectively. These results are quite impressive since it illustrates that even the Avg. Sols. found by the P-ILS-RVND are, in most cases, better than the best known solutions. The average gap between the Best Sols. and the literature solutions for 128 and 256 cores was respectively -0.26% and -0.28%.

In order to reinforce the importance of the learning phase and the load balancing scheme of P-ILS-RVND, we compared it with a simplified parallel algorithm, called Brute-Force, that does not include neither learning nor load balancing. In the Brute-Force algorithm, each core independently computes the sequential ILS-RVND algorithm for a different value of γ, and sends its solution

Table 8: Results obtained in Montané and Galvão's instances, showing the number of clients (Clients), the number of vehicles ($\#v$), the best solution found in the literature (Literature), the best solution (Best Sol.), the average solution (Avg. Sol.), and the average execution time in seconds (Avg. t)

Instance	Clients	$\#v$	Literature	P-ILS-RVND (128 cores)			P-ILS-RVND (256 cores)		
				Best Sol.	Avg. Sol.	Avg. t	Best Sol.	Avg. Sol.	Avg. t
r101	100	12	1010.90	**1009.95**	1011.17	15.60	**1009.95**	1010.54	15.81
r201	100	3	**666.20**	**666.20**	666.20	15.13	**666.20**	666.20	15.95
c101	100	16	1220.26	**1220.18**	1220.84	9.73	**1220.18**	1220.64	10.39
c201	100	5	**662.07**	**662.07**	662.07	8.30	**662.07**	662.07	8.83
rc101	100	10	1059.32	**1059.32**	1059.32	10.15	**1059.32**	1059.32	11.07
rc201	100	3	**672.92**	**672.92**	672.92	6.95	**672.92**	672.92	7.28
r1_2_1	200	23	3371.29	3360.40	3373.25	61.46	**3360.02**	3369.93	66.21
r2_2_1	200	5	**1665.58**	**1665.58**	1665.58	44.88	**1665.58**	1665.58	45.30
c1_2_1	200	28	3640.20	3630.20	3636.86	82.16	**3629.89**	3635.87	87.38
c2_2_1	200	9	1728.14	**1726.59**	1726.59	63.19	**1726.59**	1726.59	65.01
rc1_2_1	200	23	3327.98	3311.11	3320.84	68.71	**3306.00**	3317.51	71.71
rc2_2_1	200	5	**1560.00**	**1560.00**	1560.00	43.33	**1560.00**	1560.00	44.71
r1_4_1	400	54	9695.77	**9605.75**	9653.75	458.42	9618.97	9647.24	481.61
r2_4_1	400	10	3574.86	3551.53	3559.61	439.43	**3551.38**	3557.43	459.15
c1_4_1	400	63	11124.29	11102.87	11123.57	530.74	**11099.54**	11118.98	546.22
c2_4_1	400	15	3575.63	3552.09	3561.61	456.70	**3546.10**	3558.92	488.56
rc1_4_1	400	52	9602.53	9544.61	9569.99	495.10	**9536.77**	9564.86	513.38
rc2_4_1	400	11	3416.61	**3403.70**	3405.45	416.96	**3403.70**	3404.62	422.61

to the master. After receiving the results from all cores, the master chooses the best solution and finishes the computation. This algorithm clearly involves less communication overhead, however its results are sensitive to the initial choices of the parameter γ.

In Tables 9 and 10, we show a comparison between the P-ILS-RVND and Brute-Force results running on 256 cores for two different ways of generating the initial choices of gamma. Table 9 shows the results when the values of γ are generated sequentially, from 0.00, up to 5.10, incremented in a step of 0.02, while Table 10 shows the results when the values of γ are generated randomly in the range of 0.00 to 5.10 with a precision of two decimal digits. The tests were performed in the same set of instances utilized in Subsection 4.1. As can be observed in these two tables, when γ was generated sequentially, P-ILS-RVND obtained the best results in 5 of the 7 instances and equaled in one instance, and when γ was generated randomly, P-ILS-RVND found the best results in 3 instances and equaled in one. Nevertheless, when comparing the timing results, the Brute-Force algorithm was not always faster than P-ILS-RVND, even without the learning phase, which confirms the importance of the load balancing scheme used. In summary, the results of both approaches are quite similar in terms of the best solutions and timing results. However, they corroborate the fact that the Brute-Force algorithm is more susceptible to the initial choices of γ. Moreover, when less processors are available, the auto-tuning phase of γ highly increases the probability of achieving better solutions, in much shorter times.

Table 9: P-ILS-RVND versus Brute-Force for sequential γ generation.

Instance	Brute-Force		P-ILS-RVND	
	Best Sol.	Avg. t	Best Sol.	Avg. t
r101	**1009.95**	16.04	**1009.95**	15.41
sn11x	846.23	18.67	**836.04**	18.84
r1_2_1	3362.87	64.78	**3357.64**	63.53
rc1_2_1	3313.70	78.09	**3310.01**	69.49
r1_4_1	9632.18	428.74	**9620.52**	466.88
c1_4_1	**11106.27**	517.79	11109.40	554.38
rc1_4_1	9547.16	481.61	**9535.03**	504.27

Table 10: P-ILS-RVND versus Brute-Force for random γ generation.

Instance	Brute-Force		P-ILS-RVND	
	Best Sol.	Avg. t	Best Sol.	Avg. t
r101	**1009.95**	15.76	**1009.95**	15.40
sn11x	836.22	18.72	**833.92**	19.04
r1_2_1	3360.87	66.85	**3360.02**	63.65
rc1_2_1	**3310.78**	76.10	3312.03	72.01
r1_4_1	**9609.83**	428.67	9618.97	456.35
c1_4_1	11109.13	517.50	**11099.54**	563.62
rc1_4_1	**9542.51**	481.64	9550.86	500.55

4.3. Performance Evaluation

In this section we evaluate the performance of our parallel algorithm. We are interested in analyzing the speedup that P-ILS-RVND achieves when more processors are used. In this experiment, we used a different approach for the workload distribution. Instead of increasing the number of iterations as the number of processors increases, we keep the number of iterations fixed, and increase the number of processors to compute them. This way, we can evaluate the benefits of using more processors on the algorithm performance.

To perform this experiment, we used the instances sn1x, sn3x, r2_2_1, and r2_4_1, with $iter = 510$, $MaxIterILS = 250$. The instances used represent different problem sizes: sn1x (50 clients) and sn3x (100 clients) represent medium-size problems, and r2_2_1 (200 clients) and r2_4_1 (400 clients) represent large-size problems. The speedup computed is defined as the ratio between the time taken by the sequential code and that of the parallel implementation, as given by eq. (3). Parallel and sequential elapsed times used for the speedup calculation are the average of 20 consecutive runs on a dedicated machine.

$$Speedup = \frac{Sequential\_Time(\text{ILS-RVND})}{Parallel\_Time(\text{P-ILS-RVND})} \tag{3}$$

In Fig. **??** we show the speedups of P-ILS-RVND for solving sn1x, sn3x, r2_2_1, and r2_4_1, on 2, 4, 8, 16, 32, 64, 128 and 256 cores in log scale. As we can observe on these figures, P-ILS-RVND achieve increasing speedups for all the four instances. We can also observe that the speedups obtained for up to 32 processors are nearly the ideal. For more than 32 processors, however, the communication overhead slows down the computation, making our gains

not so prominent. Nevertheless, at some point, the increasing in the number of processors is not profit, since the number of iterations to compute is fixed.

When comparing the execution for medium and large instances, we find interesting results. The speedups are greater for medium instances. This is due to the greater probability of occurring load imbalance in the computation of the larger instances, for a large number of cores. The overall computational effort for each core depends upon the number of local searches performed, while the number of local searches performed varies randomly. The larger the instance, the greater the probability of one core having to compute more local searches than the other. Medium instances, on the other hand, tend to present faster convergence towards a good final solution, resulting in more homogeneous execution times for each core. This behavior can be observed by analyzing the standard deviation of the execution times obtained for each instance.

The speedup curves and the large number of processors involved in the computation show that our parallel implementation can successfully take advantage of th :-
more

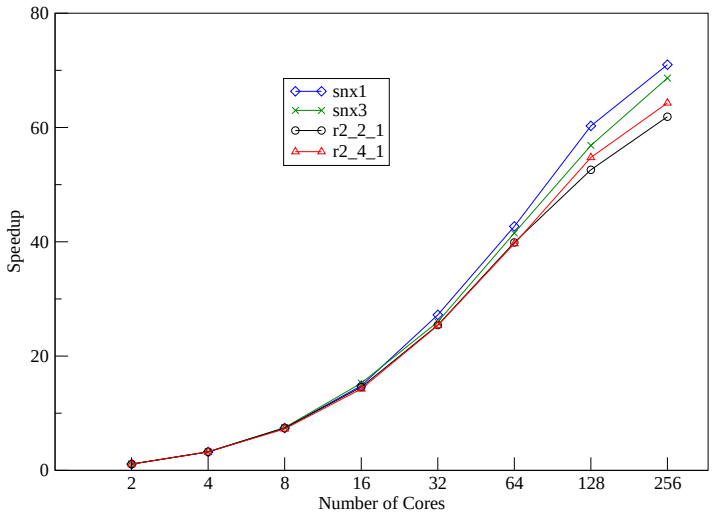

Figure 3: Speedup of P-ILS-RVND for the four instances

4.4. Communication Overhead

Communication is one of the main factors that limit the speedup of a parallel application. In this section, we investigate the communication overhead generated by P-ILS-RVND for different scenarios.

In P-ILS-RVND, the communication occurs in the three computational phases. After the first phase, each process sends the computed number of vehicles to the master. The master, then, broadcasts the smallest number found. In the

second phase, each process executes ILS-RVND for different γ values and sends the average cost to the master. In the third phase, the master distributes the ILS-RVND iterations to the workers.

The first and second phases are fast to compute and involve only one message from each worker to the master. The time spent in these phases, including computation and communication, is negligible in the overall execution time. The third phase is the one that really matters. The communication overhead in this phase depends on three parameters: the number of perturbations applied, the number of iterations to be computed, and the number of cores involved in the computation. The number of perturbations of the workers, $MaxIterILS$, controls the granularity of the work performed in each iteration. The coarser the granularity (the greater number of perturbations), the greater the execution time of each iteration. The number of iterations to be computed controls the amount of ILS-VND iterations the master distributes to the workers. The more iterations the algorithm computes, the greater the chance to find a best solution. The number of cores not only influences the number of iterations, but also the number of messages exchanged. As the number of core increases, more requests are made to the master.

In Tables 11, 12, and 13 we show the communication overhead and the number of messages generated by P-ILS-RVND for the medium and large instances of the speedup experiment: sn1x, sn3x, r2_2_1 and r2_4_1. In these tables, we varied the number of cores, the parameter k (that influences the number of iterations), and the granularity of each iteration, $MaxIterILS$. The communication overhead was computed as the percentage of the time the workers spend with communication, i.e., the percentage of the time spent on sending and receiving operations. We do not consider the master communication time in the communication overhead, as the master's main task is to communicate with the workers in order to distribute the work.

In Table 11, we show the communication overhead and the number of messages exchanged when k is fixed as 2, $MaxIterILS$ is fixed as 250, and the number of cores goes from 16 to 256. The number of messages exchanged is the same for all the instances. As expected, with increase in the number of cores, the communication overhead and the number of messages also increase, since more workers are involved in the computation. We can also observe in this table that the medium instances have slightly greater communication overhead. This occurs because they have less work to do, and so, the communication overhead becomes a little more prominent.

In Table 12, we show the communication overhead and the number of messages exchanged for 256 cores, and $MaxIterILS = 250$. We vary K from 2 to 12, generating scenarios where the number of iterations to be computed varies from 512 to 3072. As we can observe in these tables, as the value of k increases, the communication overhead decreases. These results are interesting, since we can also observe that the number of messages exchanged grows with the increase of K. The reason for the reduction in the communication overhead with the increase of K is that the load imbalance is reduced. The load imbalance has direct influence in the communication overhead, since the cores wait for the

Table 11: Communication overhead and number of messages for $K = 2$ and $MaxIterILS = 250$

# procs	Communication Overhead				# Messages
	sn1x	sn3x	r2_2_1	r2_4_1	
16	28%	22%	24%	23%	105
32	25%	35%	24%	22%	217
64	30%	31%	29%	32%	441
128	41%	35%	33%	35%	889
256	42%	34%	36%	38%	1785

others to complete a receive operation. Higher values of K increase the number of iterations to be distributed, augmenting the chance for a good load balancing between the cores.

Table 12: Communication overhead and number of messages for 256 processors and $MaxIterILS = 250$

K	Communication Overhead				# Messages
	sn1x	sn3x	r2_2_1	r2_4_1	
2	42%	34%	36%	38%	1785
4	26%	28%	27%	25%	2295
6	20%	20%	21%	18%	2805
8	18%	14%	16%	15%	3315
10	12%	13%	13%	12%	3825
12	12%	13%	11%	12%	4335

Table 13: Communication overhead for 256 processors and $K = 2$

$MaxIterILS$	Communication Overhead			
	sn1x	sn3x	r2_2_1	r2_4_1
50	42%	46%	33%	34%
150	39%	44%	30%	27%
250	42%	34%	36%	38%
350	40%	39%	34%	40%
450	37%	37%	35%	40%
550	39%	37%	34%	39%

In Table 13, we show the communication overhead for 256 cores, and $K = 2$. We vary $MaxIterILS$ from 50 to 550, generating scenarios from finer to coarser granularity of work. The number of messages exchanged is not shown, since they do not change with the growth of $MaxIterILS$. As we can observe in this table,

for large instances, as $MaxIterILS$ increases, the communication overhead also increases. This is explained by the greater load imbalance generated. Each iteration to be computed present a different computational load, as a result of the randomness of the ILS-RVND. Higher values of $MaxIterILS$ produces iterations with even more different loads, hardening the load balancing task of the master. Medium instances, on the other hand, have fewer work for 256 cores, so the increase in $MaxIterILS$ augments the amount of work to be done, reducing the weight of the communication in the overall execution time.

5. Concluding Remarks

In this paper, we have proposed a parallel metaheuristic for the Vehicle Routing Problem with Simultaneous Pickup and Delivery. The developed algorithm (P-ILS-RVND) is based on the master-worker model and has three main parts. The first one estimates the number of vehicles; the second part corresponds to the automatic calibration of the parameter γ, which is a factor that controls the bonus of inserting clients remotely located from the depot; and the third is the optimization phase.

The algorithm P-ILS-RVND was tested in 72 benchmark problems with 50-400 clients and it was found capable of improving the results of 15 instances and equaling those of another 51. In terms of quality of the solutions, our heuristic has proven to be highly competitive, especially in large-size instances.

The high-degree of parallelism inherent in the P-ILS-RVND allowed us to evaluate its performance on a cluster with up to 256 cores. In terms of speedup, we obtained increasing values, confirming the scalability of our algorithm for big clusters. We also evaluated the communication overhead generated by the algorithm for different number of processors; number of iterations; and granularity of the work.Our results show that the communication overhead stays in the average bellow 15% of the execution time when the number of iterations to be computed is more than 8 times the number of processors.

As future work, we intend to further improve P-ILS-RVND to make better use of the hybrid communication environment provided by multi-core clusters, taking advantage of the shared memory on each computing node. We also intend to reduce the communication overhead in order to improve the scalability of our algorithm on bigger clusters.

References

[1] M. P. de Brito, R. Dekker, Reverse Logistics - Quantitative Models for Closed-Loop Supply Chains, Springer, 2004, Ch. Reverse Logistics: a framework, pp. 3–27.

[2] J. Dethloff, Vehicle routing and reverse logistics: the vehicle routing problem with simultaneous delivery and pick-up, OR Spektrum 23 (1) (2001) 79–96.

[3] A. Subramanian, L. A. F. Cabral, L. S. Ochi, An efficient ILS heuristic for the vehicle routing problem with simultaneous pickup and delivery, Tech. rep., Universidade Federal Fluminense, available at http://www.ic.uff.br/~satoru/index.php?id=2 (2008).

[4] H. Min, The multiple vehicle routing problem with simultaneous delivery and pick-up points, Transportation Research 23 (5) (1989) 377–386.

[5] K. Halse, Modeling and solving complex vehicle routing problems, Ph.D. thesis, Institute of Mathematical Statistics and Operations Research, Technical University of Denmark, Denmark (1992).

[6] S. Salhi, G. Nagy, A cluster insertion heuristic for single and multiple depot vehicle routing problems with backhauling, Journal of the Operational Research Society 50 (10) (1999) 1034–1042.

[7] G. Nagy, S. Salhi, Heuristic algorithms for single and multiple depot vehicle routing problems with pickups and deliveries, European Journal of Operational Research 162 (2005) 126–141.

[8] S. Röpke, D. Pisinger, A unified heuristic for a large class of vehicle routing problems with backhauls, Tech. Rep. 2004/14, University of Copenhagen (2004).

[9] A. V. Vural, A GA based meta-heuristic for capacited vehicle routing problem with simultaneous pick-up and deliveries, Master's thesis, Graduate School of Engineering and Natural Sciences, Sabanci University (2003).

[10] E. I. Gökçe, A revised ant colony system approach to vehicle routing problems, Master's thesis, Graduate School of Engineering and Natural Sciences, Sabanci University (2004).

[11] Y. Gajpal, P. Abad, An ant colony system (acs) for vehicle routing problem with simultaneous delivery and pickup, Computers & Operations Research in press.

[12] J. Crispim, J. Brandao, Metaheuristics applied to mixed and simultaneous extensions of vehicle routing problems with backhauls, Journal of the Operational Research Society 56 (7) (2005) 1296–1302.

[13] F. A. T. Montané, R. D. Galvão, A tabu search algorithm for the vehicle routing problem with simultaneous pick-up and delivery service, Computers & Operations Research 33 (3) (2006) 595–619.

[14] J. F. Chen, T. H. Wu, Vehicle routing problem with simultaneous deliveries and pickups, Journal of the Operational Research Society 57 (5) (2006) 579–587.

[15] I. Gribkovskaia, Ø. Halskau, G. Laporte, M. Vlcek, General solutions to the single vehicle routing problem with pickups and deliveries, European Journal of Operational Research 180 (2007) 568–584.

[16] N. Bianchessi, G. Righini, Heuristic algorithms for the vehicle routing problem with simultaneous pick-up and delivery, Computers & Operations Research 34 (2) (2007) 578–594.

[17] N. A. Wassan, A. H. Wassan, G. Nagy, A reactive tabu search algorithm for the vehicle routing problem with simultaneous pickups and deliveries, Journal of Combinatorial Optimization 15 (4) (2008) 368–386.

[18] E. E. Zachariadis, C. D. Tarantilis, C. T. Kiranoudis, A hybrid metaheuristic algorithm for the vehicle routing problem with simultaneous delivery and pick-up service, Expert Systems with Applications 36 (2) (2009) 1070–1081.

[19] A. Subramanian, L. A. F. Cabral, An ILS based heuristic for the vehicle routing problem with simultaneous pickup and delivery, Proceedings of the Eighth European Conference on Evolutionary Computation in Combinatorial Optimisation, Lecture Notes in Computer Science 4972 (2008) 135–146.

[20] M. Dell'Amico, G. Righini, M. Salanim, A branch-and-price approach to the vehicle routing problem with simultaneous distribution and collection, Transportation Science 40 (2) (2006) 235–247.

[21] E. Angelelli, R. Mansini, Quantitative Approaches to Distribution Logistics and Supply Chain Management, Springer, Berlin-Heidelberg, 2002, Ch. A branch-and-price algorithm for a simultaneous pick-up and delivery problem, pp. 249–267.

[22] T. K. Ralphs, L. Ladányi, M. J. Saltzman, Parallel branch, cut, and price for large-scale discrete optimization, Mathematical Programming 98 (2003) 253–280.

[23] T. K. Ralphs, L. Ladányi, M. J. Saltzman, A library hierarchy for implementing scalable parallel search algorithms, The Journal of Supercomputing 28 (2004) 215–234.

[24] Y. Rochat, R. D. Taillard, Probabilistic diversification and intensification in local search for vehicle routing, Journal of Heuristics 1 (1995) 147–167.

[25] J. Schulze, T. Fahle, A parallel algorithm for the vehicle routing problem with time window constraints, Annals of Operations Research 86.

[26] M. Gendreau, F. Guertin, J.-Y. Potvin, E. Taillard, Parallel tabu search for real-time vehicle routing and dispatching, Transportation Science 33 (4) (1999) 381–390.

[27] M. Gendreau, G. Laporte, F. Semet, A dynamic model and parallel tabu search heuristic for real-time ambulance relocation, Parallel Computing 27 (12) (2001) 1641–1653.

[28] A. Attanasio, J.-F. Cordeau, G. Ghiani, G. Laporte, Parallel tabu search heuristics for the dynamic multi-vehicle dial-a-ride problem, Parallel Computing 30 (3) (2004) 377–387.

[29] P. Caricato, G. Ghiani, A. Grieco, E. Guerriero, Parallel tabu search for a pickup and delivery problem under track contention, Parallel Computing 29 (5) (2003) 631–639.

[30] L. S. Ochi, D. S. Vianna, L. M. A. Drummond, A. O. Victor, A parallel evolutionary algorithm for the vehicle routing problem with heterogeneous fleet, Future Generation Computer Systems 14 (5-6) (1998) 285–292.

[31] L. M. A. Drummond, L. S. Ochi, D. S. Vianna, An asynchronous parallel metaheuristic for the period vehicle routing problem, Future Generation Computer Systems 17 (4) (2001) 379–386.

[32] N. Jozefowiez, F. Semet, E.-G. Talbi, Parallel and hybrid models for multi-objective optimization: Application to the vehicle routing problem, in: PPSN VII: Proceedings of the 7th International Conference on Parallel Problem Solving from Nature, Springer-Verlag, London, UK, 2002, pp. 271–280.

[33] J. Berger, M. Barkaoui, A parallel hybrid genetic algorithm for the vehicle routing problem with time windows, Computers & Operations Research 31 (12) (2004) 2037–2053.

[34] E. Alba, B. Dorronsoro, Solving the vehicle routing problem by using cellular genetic algorithms., in: J. Gottlieb, G. R. Raidl (Eds.), EvoCOP, Vol. 3004 of Lecture Notes in Computer Science, Springer, 2004, pp. 11–20.

[35] H. Gehring, J. Homberger, Parallelization of a two-phase metaheuristic for routing problems with time windows, Journal of Heuristics 8 (3) (2002) 251–276.

[36] A. L. Bouthillier, T. G. Crainic, A cooperative parallel meta-heuristic for the vehicle routing problem with time windows, Computers & Operations Research 32 (7) (2005) 1685–1708.

[37] P. Czarnas, Parallel simulated annealing for the vehicle routing problem with time windows, in: 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, Canary Islands Spain, 2002, pp. 376–383.

[38] 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, BuR - Business Research 1 (2) (2008) 207–218.

[39] K. Doerner, R. F. Hartl, G. Kiechle, M. Lucká, M. Reimann, Parallel ant systems for the capacitated vehicle routing problem, in: J. Gottlieb, G. R. Raidl (Eds.), EvoCOP, Vol. 3004 of Lecture Notes in Computer Science, Springer, 2004, pp. 72–83.

[40] T. G. Crainic, The Vehicle Routing Problem: Latest advances and new challenges, Springer, 2008, Ch. Parallel Solution Methods for Vehicle Routing Problems, pp. 171–198.

[41] N. Mladenović, P. Hansen, Variable neighborhood search, Computers & Operations Research 24 (11) (1997) 1097–1100.

[42] I. H. Osman, Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem, Annals of Operations Research 41 (1-4) (1993) 421–451.

[43] E. Taillard, P. Badeau, M. Gendreau, F. Guertin, J. Y. Potvin, A tabu search heuristic for the vehicle routing problem with soft time windows, Transportation Science 31 (1997) 170–186.

[44] J.-F. Cordeau, G. Laporte, Metaheuristic Optimization via Memory and Evolution: Tabu Search and Scatter Search, Kluwer, 2005, Ch. Tabu search heuristics for the vehicle routing problem, pp. 145–163.

[45] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97–104.