A Parallel Algorithm for the Vehicle Routing Problem

Chris Groër
Oak Ridge National Laboratory, 1 Bethel Valley Rd, Oak Ridge, TN 37831
cgroer@gmail.com

Bruce Golden
R.H. Smith School of Business, University of Maryland, College Park, MD 20742, USA,
bgolden@rhsmith.umd.edu

Edward Wasil
Kogod School of Business, American University, Washington, DC 20016, USA,
ewasil@american.edu

The vehicle routing problem (VRP) is a difficult and well-studied combinatorial optimization problem. We develop a parallel algorithm for the VRP that combines a heuristic local search improvement procedure with integer programming. We run our parallel algorithm with as many as 129 processors and are able to quickly find high-quality solutions to standard benchmark problems. We assess the impact of parallelism by analyzing our procedure's performance under a number of different scenarios.

Key words: vehicle routing; optimization; heuristics; metaheuristics; parallel computing
History: Submitted November 2008, revised November 2009, accepted May 2010

---

1. Introduction

Since its proposal by Dantzig and Ramser (1959), the vehicle routing problem has been the subject of a great deal of research. Given customers with known locations and demands, the task is to find a minimum cost set of routes that satisfies all customer demands. Additionally, the vehicles assigned to these routes must carry no more than a fixed quantity of goods and can travel no more than a maximum distance. Although many variants of the VRP have been proposed over the last 50 years, even this simplest version of the VRP remains computationally difficult.

Because of its wide applicability and practical importance, many algorithms have been developed for the VRP. Over time, researchers have developed more powerful algorithms that find better and better solutions to both real-world VRP instances and well-studied

benchmark instances by building on previous work and by taking advantage of faster computational resources. Despite the computational difficulty of the VRP, the literature contains relatively few algorithms that use the power of parallel computing.

Parallel computing allows us to simultaneously run multiple processes or threads on several processors with the common goal of solving a particular problem instance. "Parallelism thus follows from a decomposition of the total workload and the distribution of the resulting tasks to the available processors" (Crainic, 2008). The power of parallel processing has become increasingly available with the advent of computing clusters. These clusters have (typically homogeneous) compute nodes that contain commodity hardware components running standard operating systems and are connected via special high speed networks such as Infiniband or 10-Gigabit Ethernet. Many university clusters and even the largest supercomputers in the world use this basic architecture.

Along with these clusters, another recent development in computing hardware is the multi-core processor. These are processors that contain multiple processing units (cores) that each have their own set of caches and registers that share the same memory. Many commodity laptop and desktop computers use dual- or quad-core processors. With the increasing availability of graphics processing units (GPU's) that provide even more parallelism, it is clear that future algorithms must take advantage of such readily available parallelism since failing to do so will effectively waste the majority of a processor's computing power.

In this paper, we describe an algorithm that takes advantage of a modern high-performance computing environment and multi-core processors to develop, implement, and test a parallel algorithm for the VRP. Our parallel algorithm devotes some processors to heuristic local search methods while others attempt to combine routes from different solutions into new, better solutions by solving a set covering formulation for the VRP. The resulting parallel algorithm quickly generates solutions to benchmark problems that are highly competitive with the best-known solutions reported in the literature. We analyze the parallel algorithm in several ways by measuring its performance as we vary the number of processors and the values of algorithm parameters.

This paper is organized as follows. In Section 2, we review parallel algorithms for the VRP and discuss several algorithms that combine heuristic and exact methods. We describe the details of our parallel algorithm in Section 3, present computational results in Section 4, and give our conclusions in Section 5.

2. Literature Review

There is an extensive body of literature describing algorithms for the VRP and its variants. A broad summary of recent work is given in the book by Golden et al. (2008). We refer to Cordeau et al. (2005) and Gendreau et al. (2008) for more detailed, recent reviews of work focusing on the VRP. Here, we review some parallel algorithms developed for the VRP as well as several algorithms that combine heuristic and exact integer programming methods.

2.1. Parallel Algorithms

In reviewing the literature, we were surprised to find that relatively few parallel VRP algorithms have been proposed. This view is shared by Crainic (2008) who provides a survey of exact and heuristic parallel algorithms for the VRP and its variants. We describe some of the more notable parallel algorithms developed for the VRP.

Rego (2001) presents a parallel tabu search algorithm that uses the ejection chain neighborhood as a component of the local search. His implementation uses four slave processors and a single master processor. Initial diversity is created by providing each slave with the same initial solution and then running the serial tabu search procedure with different parameters. The evaluation of some of the more complex ejection chain moves is done in parallel, and a post-optimization procedure is accelerated by determining high-quality TSP tours for each of the individual routes by sending individual routes to different slave processors.

Ralphs et al. (2003) develop a parallel branch-and-cut algorithm for solving the VRP to optimality. Much of their focus is on developing efficient methods for solving the separation problem in order to add effective cuts to the integer programming formulation. They use the SYMPHONY framework to implement their algorithm and use up to 80 processors in order to produce provably optimal solutions to problems containing up to 100 customers. They also focus on measuring the scalability of their algorithm by measuring the amount of time required by interprocess communication and the exchanging of information related to the branch-and-bound tree. We are unaware of any parallel heuristic or metaheuristic algorithms that have studied scalability issues in such detail.

Alba and Dorronsoro (2004) propose a parallel cellular genetic algorithm where genetic crossover and mutation operations are combined with a local search procedure. The algorithm allows for infeasible intermediate solutions using a penalized objective function. The population is arranged in a mesh that limits the choice of mating solutions to four neighbors.

The population is regenerated at each generation in parallel, leading to decreased computation time and better solution quality. This work is improved and extended in Dorronsoro et al. (2007) where the authors run a modified procedure on a grid platform containing 125 machines and conduct computational experiments requiring up to 75 hours of computing time. The algorithm produces very competitive solutions for the large-scale problems proposed in Li et al. (2005). Additional details on cellular genetic algorithms can be found in the book by Alba and Dorronsoro (2008).

A parallel implementation of the D-Ants algorithm is presented by Doerner et al. (2004). The authors experiment with three different ant colony optimization strategies. All of the strategies use a simple savings-based method to generate solutions that are optimized using standard local search operators. They develop a master-slave parallel algorithm where each processor constructs solutions which are sent to a master processor. The master processor then selects new solutions to broadcast out to the worker processors. Each worker updates its pheromone matrix to continue the ant-based optimizations. The authors focus only on speeding up their serial algorithm and not improving the solution quality. They present results of running their algorithm using up to 32 processors. Computational results indicate that the algorithm scales well up to eight processors, but that communication costs begin to play a more important role when increasing to 32 processors.

Finally, we mention two parallel algorithms for the VRP with Time Windows (VRPTW). Schulze and Fahle (1999) develop an effective algorithm for the VRPTW that combines a tabu search metaheuristic with a set covering formulation. In this algorithm, each processor begins with a set of solutions to the problem and then improves the solution by running tabu search. After completing the tabu search procedure, each processor broadcasts out a subset of routes discovered during tabu search and receives a set of routes sent out by other processors. Each processor then runs a heuristic algorithm to solve a set covering problem that attempts to recover a new solution that can be better than any of the individual solutions. The individual processors then update their set of solutions and repeat this process until a stopping criterion is met. The authors report very competitive results when running their algorithm with eight processors.

Le Bouthillier and Crainic (2005) develop a powerful parallel procedure for the VRPTW. They use four different methods of constructing initial solutions, four powerful metaheuristics (two based on tabu search and two based on genetic algorithms), and a post-optimization procedure that uses ejection chains along with standard local search operators. After cre-

ating an initial solution, each processor improves a solution using one of the metaheuristics and sends the best solution that it found to a central solution pool or warehouse. Post-optimization procedures are run on the solutions in this central pool, and the individual processors proceed by running one of the metaheuristics on a new solution obtained from this central pool. The algorithm produces very competitive solutions to a large number of VRPTW benchmark problems. The authors also analyze the benefit provided by the co-operative nature of their algorithm by comparing solution quality with a variant that uses the same metaheuristic solution methods without information sharing among the processors. This algorithm was subsequently improved in Le Bouthillier et al. (2005) where they extended the algorithm by considering sets of directed edges shared by the best solutions.

2.2. Combining Heuristic Methods With Integer Programming

We now present several algorithms for the VRP that combine heuristic methods with integer programming. Many of these algorithms involve the set covering formulation of the VRP. Let $\mathcal{R} = \{1, 2, \ldots, R\}$ denote the set of all feasible routes, and let c_r represent the cost of route r. The decision variable $x_r = 1$ if route r is chosen and $x_r = 0$ otherwise. Then, with $a_{ir} = 1$ if node i is contained in route r and $a_{ir} = 0$ otherwise, the set covering formulation (SC) for the VRP is given by

$$\text{Minimize} \qquad \sum_{r=1}^{R} x_r c_r \tag{1}$$

$$\text{s.t.} \qquad \sum_{r=1}^{R} x_r a_{ir} \geq 1 \text{ for all nodes } i, \tag{2}$$

$$x_r \in \{0, 1\}.$$

The objective function (1) minimizes the sum of the costs of the selected routes. Constraint (2) guarantees that every node is contained in at least one route. If the triangle inequality is not satisfied, then (2) must be replaced by an equality constraint. When the triangle inequality is satisfied, if we encounter a set of routes where some node is contained in more than one route, we modify this set of routes so that the node is contained in only one route and choose the set of routes with lowest cost.

Because the set $\mathcal{R}$ contains only feasible routes, this formulation can handle additional constraints with no modification and is the foundation for many exact solution methods for the VRP and several variants. For examples of this approach, see Agarwal et al. (1989),

Desrochers et al. (1992), Hadjiconstantinou et al. (1995), and Bixby (1998). Recent work is summarized in Bramel and Simchi-Levi (2001).

The powerful algorithm of Rochat and Taillard (1995) was one of the first to incorporate the set covering formulation into a heuristic algorithm. This procedure begins by generating a large number of initial solutions and then stores the individual routes contained in these solutions. Routes are extracted from this set in order to create new solutions that are improved by a local search procedure that uses tabu search. At each iteration, the routes are chosen probabilistically according to the quality of the solution containing the route. This allows routes from better quality solutions to be selected with a higher probability. The authors refer to this procedure as *probabilistic diversification*. Their procedure concludes with a post-optimization technique that solves a set covering problem where a single set of minimum cost routes is chosen from a large subset of all the routes found during the search. We note that our parallel algorithm shares a number of features with this algorithm as we also repeatedly apply a local search heuristic to initial solutions derived from previous runs of the heuristic. However, instead of using the set covering formulation as a post-optimization step, we include the set covering formulation as an integral part of our algorithm through parallel processing.

Taillard and Boffey (1999) combine the set covering formulation with a tabu search heuristic to produce very high-quality solutions to the heterogeneous fleet VRP. Alvarenga et al. (2007) develop a serial algorithm that combines local search and genetic crossovers to generate solutions to the VRPTW. The individual routes from these solutions are added to a large set of routes that is used as input to a set partitioning solver that is used as an integral part of their algorithm.

Recent research has attempted to incorporate more complicated integer programming formulations into heuristic methods. Franceschi et al. (2006) develop a technique for improving very good VRP solutions by extracting certain nodes and then inserting them into new positions by solving an integer program. Toth and Tramontani (2008) extend this technique by considering the column generation problem in more detail. In both cases, very good solutions are obtained, albeit with fairly long computing times.

3. Description of the Parallel Algorithm

We provide a top-down description of the parallel algorithm. We start with a general description of the algorithm and its motivation and then describe the details of the different components.

The algorithm utilizes a single master processor that coordinates the search and keeps track of the best solutions found by the other processors. The remaining processors are assigned one of two tasks, either to generate and improve solutions to the problem instance by running a metaheuristic algorithm (we call these the *heuristic solvers*), or to solve instances of a set covering problem with columns (routes) taken from solutions discovered by other processors (we call these the *set covering solvers*).

After the heuristic solvers complete a run of the metaheuristic algorithm, these processors collect a set of distinct routes discovered during the run, write a single file for each route, and send a set of full solutions to the master processor. The master processor then sends back a single solution to be improved. In the meantime, the set covering solvers read these route files and construct and solve set covering problem instances in an attempt to combine routes from different solutions into new, better solutions. After solving the current instance of the set covering problem, the set covering solvers send the solution back to the master processor which responds by sending back a set of the best solutions discovered so far. This cycle continues until a pre-defined time limit is reached at which point the master processor returns the best overall solution. The overall flow of information is shown in Figure 1. The dashed lines represent the flow of information between the heuristic solvers and the set covering solvers, and the solid lines represent information sent to and from the master processor (we omit some lines that involve the master processor to keep the diagram simple).

The algorithm shares some characteristics with the well-known master-slave architecture as we have a single processor that periodically communicates with all the other processors. However, in order to avoid a communication bottleneck involving the master processor, we designed the algorithm so that the majority of information is exchanged by reading and writing the files containing the routes discovered by the heuristic solvers. This method of information diffusion places the communication burden on the file system rather than on the interprocess communication and generally allows for a completely asynchronous exchange of information. More importantly, we observed no communication bottleneck involving the master processor in any of our computational experiments with up to 129 processors. In an

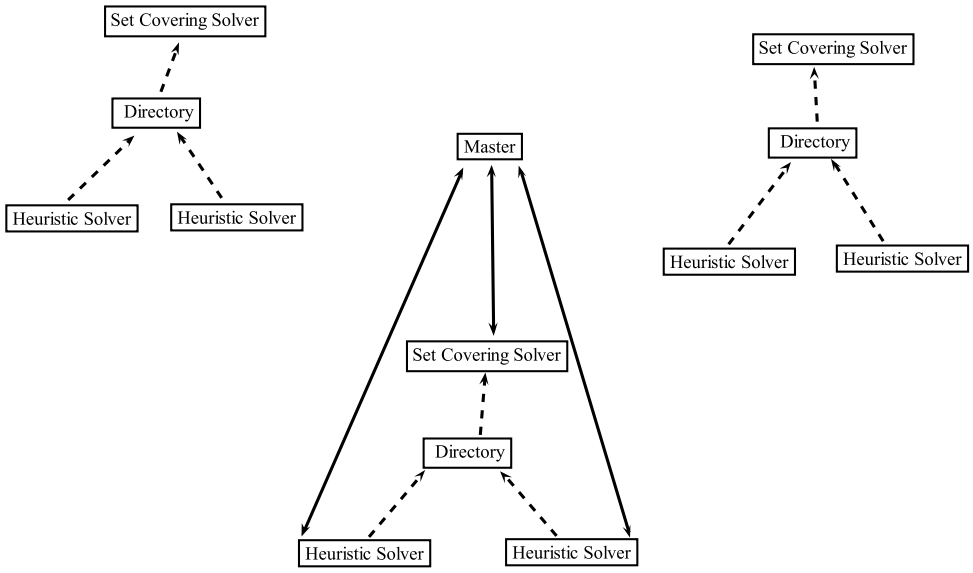


Figure 1: The parallel architecture

earlier version of our algorithm, the heuristic solvers sent their routes directly to the master processor which then broadcasted these routes to the set covering solvers. When we ran this algorithm with more than 20 or so processors, we encountered a significant communication bottleneck that forced processors to wait several seconds for the master processor to become available, clearly preventing any kind of scalability.

Before describing the components of our algorithm in more detail, we discuss its design in the context of other parallel metaheuristics by using the three dimensions suggested by Crainic et al. (2005). The first dimension is search control cardinality and measures how the search is controlled. The second dimension deals with the type and amount of information exchanged among the processors. The third dimension addresses the similarities and differences of the search strategies employed by the different processors. According to this system, our algorithm fits into the *pC/KC/MPDS* classification. The *pC* classification indicates that the global search is controlled by multiple, collaborating processors. *KC* stands for Knowledge Collegial information exchange and indicates that multiple processors exchange information asynchronously and that new solutions are created from the exchanged information (this is achieved by solving the set covering problem). Finally, our procedure fits the *MPDS* classification (Multiple Points, Different Strategies) since the heuristic solvers run a different strategy from different points in the solution space.

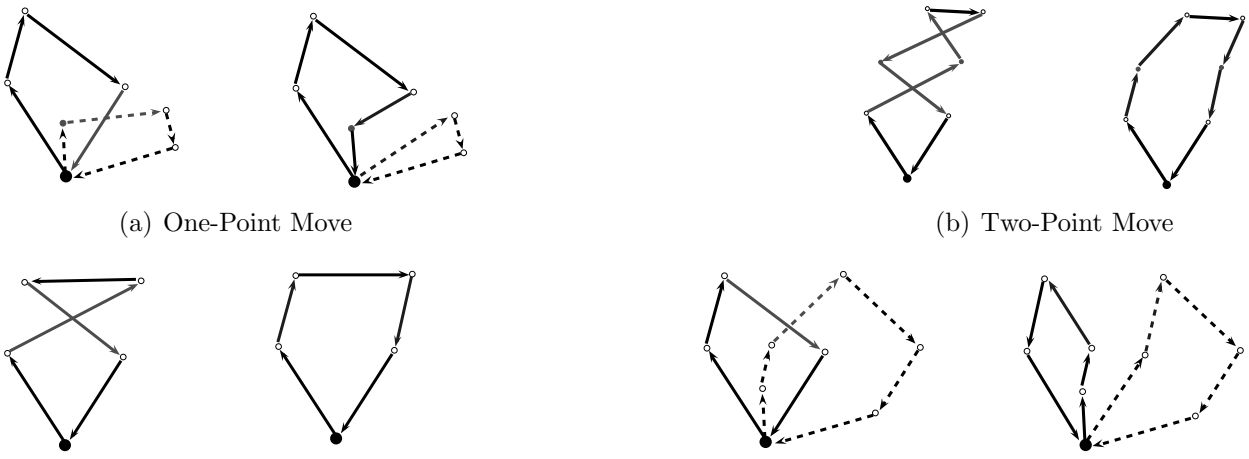
3.1. A Serial Metaheuristic Algorithm for the VRP

We now discuss the role of the heuristic solvers in more detail. The heuristic solvers begin the parallel algorithm by generating initial solutions using the well-known parameterized Clarke-Wright algorithm described in Yellow (1970) with a randomly generated shape parameter $\lambda \in (.5, 2)$. After the initial generation of solutions, these processors improve solutions by applying the record-to-record travel algorithm (RTR) used by Chao et al. (1995) for the period vehicle routing problem and by Golden et al. (1998) and Li et al. (2005) to improve solutions to the classical VRP. The underlying principle behind these algorithms is to apply different local search operators and accept all improving moves as well as some deteriorating moves. In particular, the algorithm accepts any deteriorating move that worsens the objective function value by less than a fixed amount based on the best solution found so far (i.e., the *record* solution). This fixed amount of permissible deterioration diminishes as we find better and better records so that we accept fewer deteriorating moves as the algorithm progresses. These previous RTR-based algorithms have a very simple structure, require a small number of parameters, and perform consistently well on a variety of benchmark problems. Our implementation maintains the basic structure of these previous algorithms, and enhances them by adding local search operators.

The RTR algorithm given in Li et al. (2005) uses the three local search operators shown in Figure 2: one-point move, two-point move, and two-opt move. The one-point move shifts a single node to a new position in the solution, the two-point move swaps the positions of two nodes in the existing solution, and the two-opt move removes two edges from the existing solution and replaces them with two new edges.

We use three additional local search operators in our implementation: Or-opt move, three-opt move, and three-point move. These operators are shown in Figure 3. In the Or-opt move (Or, 1976), we remove a string of two, three, or four customers from the solution and insert this string into a new position in the solution. The three-opt move (Lin and Kernighan, 1973) removes three edges from a single route and adds three new edges so that a feasible route is maintained. The three-point move is a special case of the λ-interchange operator (Osman, 1993) where we exchange the positions of two consecutive nodes with the position of a third node.

For each node i, we create a list $L_i(N)$ that contains i's nearest N neighbors (N is a randomly generated parameter). Then, when we apply any of the local search operators to



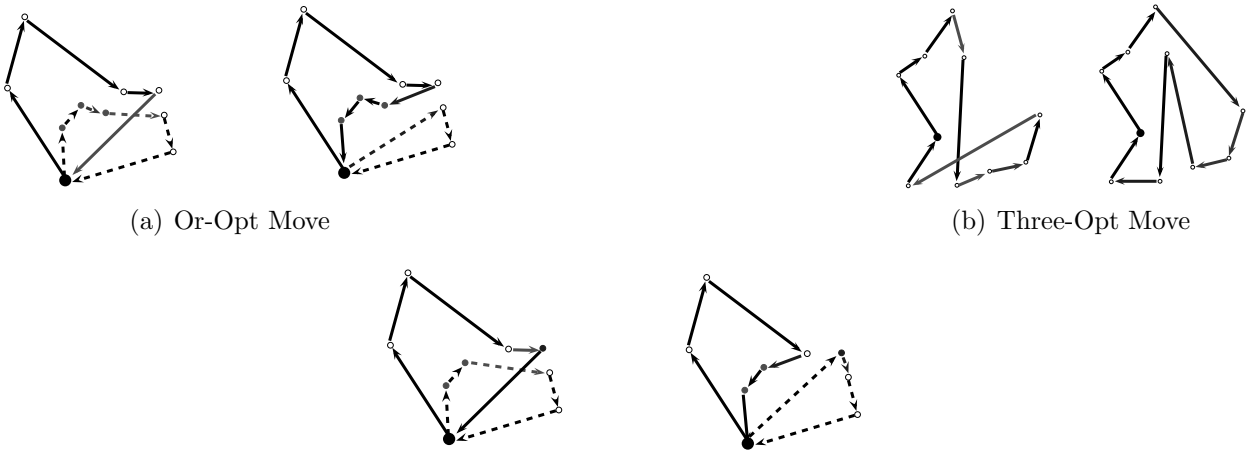
(a) One-Point Move

(b) Two-Point Move

(c) Intra-route Two-Opt Move

(d) Inter-route Two-Opt Move

Figure 2: Improvement operators used in Li et al. (2005)



(a) Or-Opt Move

(b) Three-Opt Move

(c) Three-Point Move

Figure 3: Additional improvement operators used in our serial algorithm

search for an acceptable move, we speed up the search by restricting the search to only those moves involving at least one other node taken from the list $L_i(N)$.

Our serial improvement algorithm alternates between two stages, a *diversification* phase where the record-to-record travel acceptance criterion is used, and an *improvement* phase where we seek a local minimum in the search space. In the diversification phase, for any local search operator and a given node i, we search the list $L_i(N)$ for a feasible move. If we find an improving move, then we immediately make this move. If no improving move is found, then we revisit the feasible move with the least amount of deterioration. If this

move does not increase the total route length by more than a fixed amount based on the best solution (the record), then we make the move. In the improvement phase, we use all six operators, and look for an improving move for each node i, again by searching the neighbor list $L_i(N)$. If no improving move is found, then we perform no modifications and move to the next node. The algorithm alternates between these two phases until a new solution has not been found for K iterations. The current solution is perturbed at this point using the method of Li et al. (2005) in an attempt to move to a new region of the solution space. The procedure terminates after P perturbations and the algorithm returns a list of the 50 best solutions found during the search. Finally, we note that all parameters for each run of the algorithm are generated at random within a range that we determined through preliminary experiments. We used the same range of parameters for all problem sizes, eliminating the need for parameter tuning and also adding diversity to the search since different processors use different parameters. We refer to our implementation of the algorithm as Randomized RTR (RRTR) and present the details in Algorithm 1.

In the RRTR algorithm, after generating a set of random parameters in steps 1 to 3, steps 5 to 11 attempt to diversify the solution by accepting deteriorating moves according to the record-to-record travel criterion. Steps 12 to 18 represent the improvement phase where only improving moves are accepted and we reach a local minimum in the search space. We apply all six local search operators in the improvement phase and only a randomly selected subset of these operators in the diversification phase. If the local minimum at the end of the improvement phase is a new record, then we update this value and the threshold in step 20 and reset the counter (k) that keeps track of the number of times we have been unable to escape a particular local minimum. In steps 23 to 26, we check to see if we have been unable to improve the current record after K times through the main loop of the algorithm. If there is no improvement, then we perturb the solution by removing a set of customers from the solution and reinserting them into new locations (this technique is given in Li et al. (2005)). Once we have perturbed the solution P times, we return the 50 best solutions found during the search.

In preliminary experiments on several benchmark problems, we ran the RRTR algorithm on a single processor using all six local search operators. We found that the solution quality was roughly equivalent to the results given by Li et al. (2005) where only three local search operators were used. Our motivation in using these additional operators is simple: we want

Algorithm 1 The RRTR algorithm for the VRP

Generate random parameters $I \in \{25, 75\}$, $P \in \{5, 6, \ldots, 10\}$, $N \in \{25, 26, \ldots, 75\}$, $\delta \in (0.005, 0.015)$, and $K \in \{5, 6, \ldots, 10\}$
Let $\mathcal{V}$ denote the set of six local search operators, and randomly select a subset of operators, $\mathcal{U} \subseteq \mathcal{V}$
Set the record R equal to the sum of the individual route lengths in the starting solution, set the threshold $T = (1 + \delta)R$, and set $k = p = 0$
while $p < P$ **do**
    for $i = 1$ to I **do**
        for all Operators $u \in \mathcal{U}$ **do**
            for $j = 1$ to n **do**
                Apply operator u to node j by searching the list $L_j(N)$ for feasible moves and accepting moves according to record-to-record travel
            end for
        end for
    end for
    while Improving moves can be found **do**
        for all Operators $v \in \mathcal{V}$ **do**
            for $j = 1$ to n **do**
                Apply operator v to node j by searching the list $L_j(N)$ for feasible moves, accepting only improving moves
            end for
        end for
    end while
    if The current solution is a new record **then**
        Update R and T and set $k = 0$
    end if
    $k = k + 1$
    if $k = K$ **then**
        Perturb the solution
        $p = p + 1$
    end if
end while
Return a list of the 50 best solutions found

to have each processor running a slightly different algorithm using different parameters in order to diversify the search.

Having described the details of the RRTR procedure, we now describe the role of the heuristic solvers in the parallel algorithm. After generating an initial solution with the Clarke-Wright algorithm using a randomly generated shape parameter λ as described in Yellow (1970), the heuristic solvers repeatedly perform the following steps: 1) receive a feasible solution from the master processor, 2) attempt to improve the procedure by running

the RRTR algorithm, 3) write a file containing a set of routes discovered during the search, and 4) send the 50 best unique solutions produced during the most recent run to the master processor.

As literally hundreds of thousands of solution modifications are made during a typical run of the RRTR algorithm, it is not practical to write a file for every route discovered. Thus, the heuristic solvers must write files for only a fraction of these routes. The heuristic solvers maintain a list of the 50 best unique solutions found during the search and also store the solutions that are found at the end of the *while* loop in step 18 of the algorithm since each of these solutions represents a local minimum in the search space and can contain promising routes. At the end of each run, a heuristic solver takes this set of solutions, performs the three-opt improvement on each individual route until no improvements can be found, and then creates a list of all unique routes found in this set of solutions. This list of routes is sorted by the total route length of the solution containing the route, and these routes are written to a file. By sorting the list of routes in this way, when the set covering solver reads this file and adds the routes to the integer program, routes derived from higher quality solutions are selected first.

3.2. The Set Covering Solvers

Our heuristic solvers are quite fast, completing the RRTR algorithm in 1 to 20 seconds for the problem instances we encountered (ranging from 50 to 1200 nodes). In contrast, integer programs (IPs) can be very difficult to solve, even with the best commercial solvers. In order to combine heuristic procedures and IP solvers into a single, cooperative, parallel VRP algorithm, we must ensure that the computation times for both methods are roughly equivalent. Otherwise, if the set covering problems grow too large and become difficult to solve, the IP solver may spend all of its time solving a single problem that contains routes derived from heuristic solutions generated very early in the search. In order to prevent such a situation, we must carefully manage the number of columns or routes we allow into the set covering problem.

We initially set two parameters: s, the maximum number of seconds allowed to solve a single set covering problem, and m, the minimum number of new columns that must be found before attempting to solve a new set covering problem. We control the total number of columns in the set covering formulation by monitoring the time required to solve the integer

program. Each set covering solver keeps track of a quantity M_j, the maximum number of columns allowed when solving integer program j. We set an initial value $M_0 = 500$ and then, after solving this initial problem, we set M_{j+1} in terms of M_j via a simple rule:

$$M_{j+1} = \begin{cases} M_j, & \text{if problem } j \text{ required between } s/2 \text{ and } s \text{ seconds,} \\ \lfloor 1.1 \times M_j \rfloor, & \text{if problem } j \text{ required less than } s/2 \text{ seconds,} \\ \lfloor .9 \times M_j \rfloor, & \text{if problem } j \text{ was not solved to optimality in } s \text{ seconds.} \end{cases} \quad (3)$$

After initially finding M_0 columns and then solving this first problem, M_1 is computed using this rule and the set covering solver starts the search for additional columns by reading the files written by the heuristic solvers. After finding m new columns, the problem is solved again. Note that recursion (3) implies that, in some cases, we remove existing columns from the integer program. In these cases, we select columns to remove at random, except that we never remove a column that was once part of an optimal solution to a set covering problem.

3.3. The Master Processor

The master processor is responsible for storing the solutions discovered by the other processors and is responsible for sending new solutions to the other processors when they complete their tasks. When the master processor receives a new solution from a set covering solver, it sends the distinct routes from the 10 best solutions to the set covering solver. This ensures that each set covering solver always has the routes from the current 10 best solutions.

When the master processor receives new solutions from the heuristic solvers, it adds them to the list of the 1000 best unique solutions (we use a hash table to increase efficiency) and sorts this list in increasing order by the total route length. After adding these solutions to its list, the master processor then determines which solution(s) to send back to the heuristic solver.

In the simplest case, the master processor selects one solution and sends it back to the heuristic solver. We experimented with two different strategies to select this solution. In the first case, the master processor simply selects the best solution found so far. In the second strategy, the master processor selects a random solution whose quality is biased in favor of better solutions as the search progresses. In particular, if, after s seconds, the master

processor has k total unique solutions and a total allowed time of t seconds, we select solution j, where j is generated by the following procedure:

- Generate $r \in (0, 1)$ uniformly at random;
- Compute $j = \lfloor k(1 - s/t)^{r+2} \rfloor$;
- Return solution j from the sorted list.

This procedure tends to send out worse solutions early in the search, and better solutions near the end. In Figure 4, we show the average value of j for $k = 1000$ and $t = 100$ as time progresses from 0 to 100.

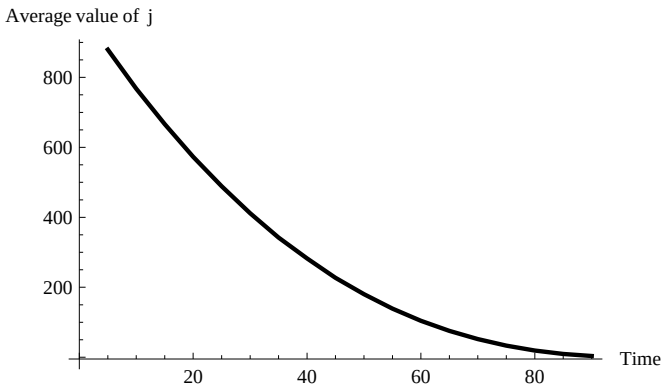
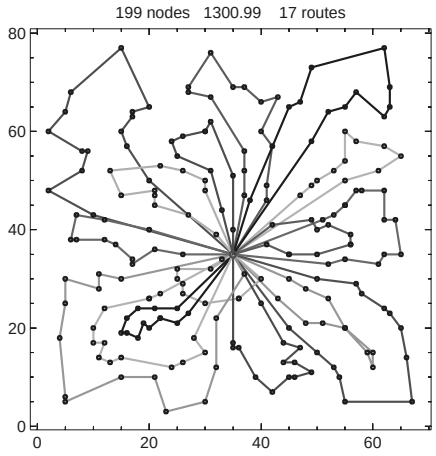


Figure 4: The average value of the randomly selected solution as time progresses

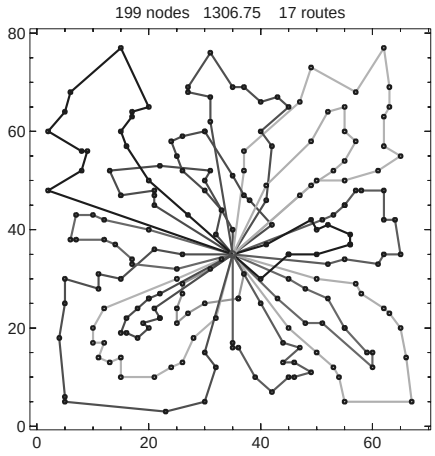
The typical metaheuristic for the VRP generates solutions by considering the entire problem instance and generating a set of routes that visit all customers. However, our parallel algorithm's use of the set covering solvers allows us to decompose the original problem instance into smaller parts. The set covering solvers simply take the feasible routes belonging to such *partial solutions* and then add them as columns. We used two strategies to divide the problem into smaller subproblems that contain fewer nodes than the original problem. By running the RRTR algorithm on these smaller problems, we hope to find additional routes that might not be discovered when running the algorithm on the entire original problem. We use these strategies in the second half of the search process after we have (hopefully) created a very high-quality set of solutions.

In the first strategy, the master processor selects one of the 10 best solutions, and a second, inferior solution that is chosen at random from the remaining pool of solutions. The master processor then attempts to create a smaller problem for the heuristic solver by taking

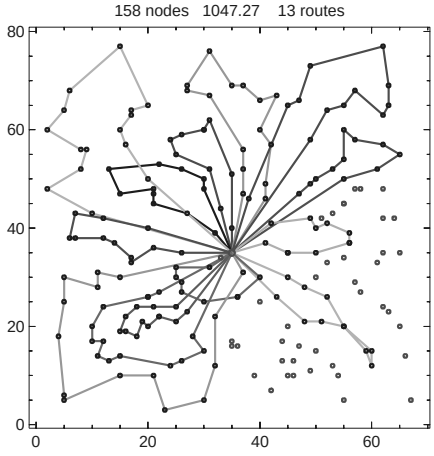
the superior solution and removing any routes that are shared with the inferior solution. The master processor sends the resulting sub-problem to the heuristic solver. The logic behind this idea is straightforward. At some point in the search, certain routes become "obvious" and are shared by many solutions. By removing routes shared by the two solutions, we can focus more intensely on the "non-obvious" parts of the problem. In Figure 5, we show an example of this route removal procedure where two solutions (17 routes, 199 nodes) have four routes in common. After removing these four routes, we have a smaller subproblem with 13 routes and 158 nodes.



(a) Solution 1



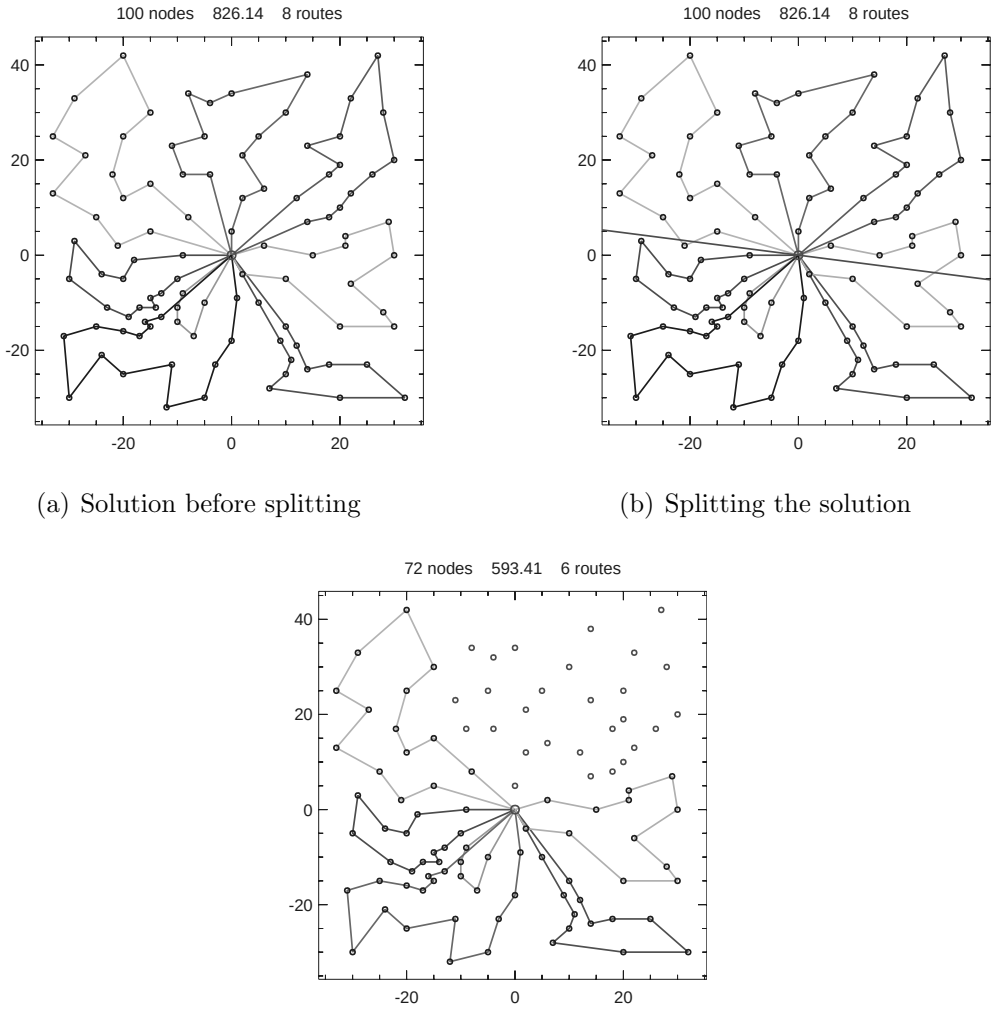
(b) Solution 2



(c) Solution after removal of routes in common

Figure 5: The route removal operation

In the second strategy, the master processor selects a single solution at random and then draws a single randomly generated line through the depot in order to split the problem into two disjoint parts. We accept this splitting if the two parts are roughly equal in size, so that each part has 40–60% of the total number of nodes. Then, in order to preserve some of the routes found in the initial solution, we create two problems by first selecting all routes that have at least one node above the line and then selecting all routes that have at least one node below this line. The master processor then sends the larger of the two resulting subproblems to the heuristic solver. This procedure is illustrated in Figure 6 where two routes are removed from the solution, creating a problem instance with 28 fewer nodes.



(a) Solution before splitting

(b) Splitting the solution

(c) Solution after splitting

Figure 6: The solution splitting operation

4. Computational Results

We implemented our algorithm in C/C++, using the well-known Message Passing Interface (MPI) to handle the inter-processor communication and CPLEX 11.1 for the set covering problems. The serial metaheuristic was implemented using the open source VRPH software described in Groër et al. (2010) which is available for download from COIN-OR (2010). Because we use randomly generated parameters in Algorithm 1, we only had to determine parameters involved with the set covering solvers and the communication between the different processors. We set a limit of five seconds for each set covering problem as preliminary experiments demonstrated that this was a sufficient amount of time for producing an optimal solution to most set covering problems. The parameters involved in the communication of routes and solutions among the different processors were set after some initial preliminary experiments, and the parameter values were set identically for all problems. The results reported here were obtained by running the algorithm on a cluster consisting of 50 total compute nodes, each equipped with two dual-core 2.3 GHz Intel Xeon processors.

We provide computational results for our parallel algorithm's performance on 55 problems taken from four well-known sets of benchmark problems: 14 problems from Christofides and Eilon (1969) and Christofides et al. (1979), 9 problems from Taillard (1993a), 20 problems from Golden et al. (1998), and 12 problems from Li et al. (2005).

In Section 4.1, we describe our algorithm's performance on all 55 problems under several different scenarios and compare the solutions to the results reported in the literature. In Section 4.2, we generate solutions to a few selected problems with our parallel algorithm under different scenarios in order to analyze the impact of various parameters, and in Section 4.3, we vary the number of processors but use a fixed amount of total computing time in an attempt to quantify the parallel speedup of our algorithm.

4.1. Solutions to Benchmark Problems

We report the results produced by the parallel algorithm on all 55 benchmark problems using three different configurations where we vary the number of processors used in the algorithm. For each configuration, we ran the parallel algorithm with default parameters and also ran a completely non-cooperative algorithm. In this version of the algorithm, each processor is a heuristic solver that repeatedly generates a new random initial solution using

the parameterized Clarke-Wright algorithm and then improves this solution by running the RRTR algorithm. When the time limit is reached, the best overall solution is recorded. By running this alternative version of the algorithm, we attempt to measure the benefit provided by the cooperative nature of our parallel algorithm.

In the first configuration, we used 129 processors with 16 set covering solvers for a total of five minutes total wallclock time. Our goal here is to generate the best solutions we can find using the most powerful available configuration. In the remaining two configurations, we ran the algorithm on a single computer equipped with multiple computing cores, allowing an amount of wallclock time similar to that used by other serial algorithms in the literature: 100 seconds for problems with 100 nodes or less, 200 seconds for problems with 200 nodes or less, and 300 seconds for problems with more than 200 nodes. Our goal here is to compare the performance of our parallel algorithm with the most powerful serial algorithms from the literature by using a comparable amount of wallclock time.

We ran the algorithm five times on each problem with each configuration and present the results of these experiments in Tables 1–4. In each table, the first three columns describe the problem instance (problem number, number of nodes, number of routes) and the fourth column lists the previous best-known solution (BKS) along with a footnote providing the earliest reference we were able to find. The remaining columns provide the best solutions found by the cooperative and non-cooperative versions of the algorithm under the various configurations. An entry in bold indicates a new best-known solution and an entry in italics indicates a solution equal to the current best-known solution.

When run with 129 processors, the cooperative parallel algorithm generated very high-quality solutions to all 55 benchmark problems, discovering 13 new best solutions (these are provided in the Online Supplement). We found 31 solutions that equaled the best-known solutions. On the remaining 11 problems, our algorithm produced solutions that were slightly worse than the best-known solutions (an average deviation of 0.15%).

Running our algorithm with 129 processors clearly uses much more total CPU time than that required by most algorithms in the literature, and so it is difficult to make a "fair" comparison between our parallel algorithm and an algorithms that uses only a single CPU. However, multi-core processors containing four cores or more are becoming commonplace in modern computers and this trend will continue. Thus, it is worthwhile to compare the results of running our algorithm on a single multi-core processor with other serial algorithms from the literature when using a similar amount of wallclock time.

Table 1: Solutions for the problems of Christofides and Eilon (1969) and Christofides et al. (1979)

Problem Data				Number of Processors, Set Covering Solvers					
				129,16		8,1		4,1	
#	Nodes	Routes	BKS	Coop.	Non-coop.	Coop.	Non-coop.	Coop.	Non-coop.
1	50	5	524.61[a]	*524.61*	*524.61*	*524.61*	*524.61*	*524.61*	*524.61*
2	75	10	835.26[a]	*835.26*	*835.26*	*835.26*	*835.26*	*835.26*	*835.26*
3	100	8	826.14[a]	*826.14*	*826.14*	*826.14*	*826.14*	*826.14*	*826.14*
4	150	12	1028.42[a]	*1028.42*	*1028.42*	*1028.42*	*1028.42*	*1028.42*	1029.56
5	199	16	1291.29[b]	1291.45	1293.24	1291.50	1296.78	1294.25	1296.29
6	50	6	555.43[a]	*555.43*	*555.43*	*555.43*	*555.43*	*555.43*	*555.43*
7	75	11	909.68[a]	*909.68*	*909.68*	*909.68*	*909.68*	*909.68*	*909.68*
8	100	9	865.94[a]	*865.94*	*865.94*	*865.94*	*865.94*	*865.94*	*865.94*
9	150	14	1162.55[a]	*1162.55*	*1162.55*	*1162.55*	1163.66	1162.99	1163.84
10	199	18	1395.85[a]	*1395.85*	1399.93	1399.91	1400.94	1400.74	1403.70
11	120	7	1042.11[a]	*1042.11*	*1042.11*	*1042.11*	*1042.11*	*1042.11*	*1042.11*
12	100	10	819.56[a]	*819.56*	*819.56*	*819.56*	*819.56*	*819.56*	*819.56*
13	120	11	1541.14[a]	*1541.14*	1541.25	1542.36	1543.20	1542.86	1543.28
14	100	11	866.37[a]	*866.37*	*866.37*	*866.37*	*866.37*	*866.37*	*866.37*

[a]Rochat and Taillard (1995); [b]Mester and Bräysy (2007)

Table 2: Solutions for the problems of Taillard (1993b)

Problem Data				Number of Processors, Set Covering Solvers					
				129,16		8,1		4,1	
#	Nodes	Routes	BKS	Coop.	Non-coop.	Coop.	Non-coop.	Coop.	Non-coop.
100A	100	11	2041.34[a]	*2041.34*	*2041.34*	*2041.34*	*2041.34*	*2041.34*	*2041.34*
100B	100	11	1939.90[a]	*1939.90*	*1939.90*	*1939.90*	*1939.90*	*1939.90*	*1939.90*
100C	100	11	1406.20[a]	*1406.20*	*1406.20*	*1406.20*	*1406.20*	*1406.20*	*1406.20*
100D	100	11	1580.46[b]	*1580.46*	*1580.46*	*1580.46*	*1580.46*	*1580.46*	*1581.26*
150A	150	15	3055.23[a]	*3055.23*	*3055.23*	*3055.23*	3057.36	*3055.23*	3056.24
150B	150	14	2727.20[b]	*2727.20*	2728.32	2727.88	2730.88	2727.88	2730.44
150C	150	15	2341.84[c]	2358.66	2358.92	2358.66	2361.08	2358.92	2361.62
150D	150	14	2645.40[c]	*2645.40*	2646.10	*2645.40*	2647.28	2649.03	2655.06
385	385	47	24369.13[b]	**24366.69**	24462.71	24461.40	24495.50	24461.91	24531.52

[a]Mester and Bräysy (2007); [b]Nagata and Bräysy (2008);[c]Taillard (1993a)

In Table 5, we compare the performance of our parallel algorithm to the best-performing serial algorithms from the literature. The results of Prins (2009) and Mester-Bräysy are obtained via a single run with one parameter setting (note that Prins provides results only for the problems of Christofides and Golden), and the results of Nagata-Bräysy are the average of 10 runs (they do not consider the largest problems of Li). For our algorithm, we report the result of a single run using the same set of values for the parameters. Although the other algorithms generally require widely varying running times for the various problems, we ran our algorithm for 100, 200, or 300 seconds depending on the size of the problem instance.

Table 3: Solutions for the problems of Golden et al. (1998)

Problem Data				Number of Processors, Set Covering Solvers					
				129,16		8,1		4,1	
#	Nodes	Routes	BKS	Coop.	Non-coop.	Coop.	Non-coop.	Coop.	Non-coop.
1	240	9	5627.54[a]	**5623.47**	5628.95	5636.96	5637.83	5644.44	5645.90
2	320	10	8431.66[e]	*8447.92*	8435.00	*8447.92*	8450.32	*8447.92*	8451.52
3	400	10	11036.22[e]	*11036.22*	11037.42	*11036.22*	11039.82	*11036.22*	11039.82
4	480	10	13592.88[e]	*13624.52*	13625.72	*13624.52*	13628.12	*13624.52*	13629.32
5	200	5	6460.98[e]	*6460.98*	*6460.98*	*6460.98*	*6460.98*	*6460.98*	*6460.98*
6	280	7	8404.26[e]	8412.90	8412.90	8412.90	8413.36	8412.90	8412.90
7	360	9	10156.58[e]	10195.59	10195.59	10195.59	10195.59	10195.59	10195.59
8	440	10	11643.90[b]	*11663.55*	11649.89	11691.76	11681.22	11691.76	11680.31
9	255	14	580.02[d]	**579.71**	582.30	581.92	584.26	583.37	585.26
10	323	16	738.44[d]	**737.28**	740.24	739.82	744.92	742.73	743.47
11	399	18	914.03[d]	**913.35**	918.63	916.14	922.08	917.91	920.69
12	483	19	1104.84[d]	**1102.76**	1112.02	1112.73	1115.96	1118.14	1117.05
13	252	26	857.19[d]	*857.19*	859.44	858.45	861.06	858.89	863.00
14	320	30	1080.55[d]	*1080.55*	1083.43	1080.55	1082.96	1081.24	1086.50
15	396	33	1340.24[d]	**1338.19**	1342.89	1341.41	1350.28	1346.45	1352.82
16	480	37	1616.33[d]	**1613.66**	1622.74	1619.45	1628.79	1624.42	1628.34
17	240	22	707.76[d]	*707.76*	707.96	707.79	708.63	707.79	708.75
18	300	27	995.13[d]	*995.13*	1000.27	997.25	1003.62	998.66	1001.89
19	360	33	1365.97[e]	**1365.60**	1369.39	1366.26	1371.65	1369.34	1372.87
20	420	38	1819.99[e]	**1818.25**	1826.47	1820.88	1834.35	1824.98	1833.70

[a]Mester and Bräysy (2007); [b]Prins (2009); [c]Pisinger and Røpke (2007); [d]Nagata and Bräysy (2008);[e]Nagata and Bräysy (2009)

Table 4: Solutions for the problems of Li et al. (2005)

Problem Data				Number of Processors, Set Covering Solvers					
				129,16		8,1		4,1	
#	Nodes	Routes	BKS	Coop.	Non-coop.	Coop.	Non-coop.	Coop.	Non-coop.
21	560	10	16212.74[a]	16212.83	16214.03	16212.83	16216.42	16212.83	16216.42
22	600	15	14597.18[a]	**14584.42**	14611.10	14621.40	14636.25	14631.73	14632.08
23	640	10	18801.12[a]	18801.13	18802.33	18801.13	18805.92	18801.13	18804.72
24	720	10	21389.33[a]	21389.43	21391.83	21389.43	21395.42	21390.63	21395.42
25	760	19	16902.16[b]	**16763.72**	16835.12	16832.77	16936.83	17089.62	17043.02
26	800	10	23971.74[a]	23977.73	23980.13	23977.73	23981.33	23977.73	23980.13
27	840	20	17488.74[a]	**17433.69**	17528.58	17563.24	17592.48	17589.05	17611.23
28	880	10	26565.92[a]	26566.03	26568.43	26568.43	26585.58	26567.23	26586.49
29	960	10	29154.34[c]	*29154.34*	29155.54	29154.34	29160.33	29155.54	29160.33
30	1040	10	31742.51[a]	31742.64	31745.04	31742.64	31751.97	31743.84	31925.38
31	1120	10	34330.84[a]	34330.94	34333.34	34330.94	34335.73	34333.37	34335.73
32	1200	11	36919.24[c]	37185.55	37256.25	37294.28	37337.51	37285.90	37303.05

[a]Mester and Bräysy (2007); [b]Pisinger and Røpke (2007); [c]Estimated solution from Li et al. (2005)

The results for the seven algorithms in Table 5 are quite similar and all algorithms are within about 0.5% of the best-known solution on the four problem sets. The memetic algorithm of Nagata and Bräysy (2009) produces the best solutions on average although the computations are generally longer than for the other algorithms. Our algorithm does

Table 5: The average percent above the best-known solution for the top-performing algorithms in the literature

	Problem Set			
Algorithm	Christofides	Taillard	Golden	Li
Prins (2009)	0.071	N/A	0.525	N/A
Mester-Bräysy (2005, 2007)	0.027	0.236	0.263	0.202
Nagata-Bräysy (2009)	0.030	0.096	0.210	N/A
4 cores, cooperative	0.115	0.197	0.468	0.372
4 cores, non-cooperative	0.144	0.320	0.814	0.489
8 cores, cooperative	0.085	0.131	0.411	0.299
8 cores, non-cooperative	0.096	0.300	0.650	0.458

not perform as well as the others on the small problems of Christofides, but it compares favorably on the other three problem sets. By comparing the average solution quality of the cooperative and non-cooperative configurations using four and eight cores, we see that cooperation generally leads to better solutions with an average improvement of less than 0.3%. Although this is a modest improvement, this amount is not as negligible as it may seem at first glance. The non-cooperative version of our algorithm finishes behind all other algorithms from the literature on the four problem sets. The results in Table 5 show that adding cooperation to our algorithm provides a critical boost to create solutions of the highest quality.

4.2. Analyzing the Effect of Parameters on Performance

In order to study the importance of certain parameters in our parallel algorithm, we ran our procedure with different settings on three benchmark problems: problem 15 from Golden et al. (1998), problem 385 from Taillard (1993b), and problem 25 from Li et al. (2005) (we refer to these problems as G15, T385, and L25). These three problems are dissimilar in problem size, spatial construction, and constraints. They also appear to be *difficult* in the sense that there is considerable variation in the solutions reported by the best-performing algorithms from the literature.

We ran our algorithm 10 times on each problem, varying the following parameters:

- Number of processors (8, 16, 32, or 64);

- Number of set covering solvers (up to 8);
- Total time allowed (200 or 400 seconds);
- Strategy used by the master processor to distribute solutions (randomly selected solutions, best solution).

We summarize the results of this experiment in Table 6 where each row in Table 6 corresponds to a particular parameter setting, and each entry in the table contains two values. The first value is the overall average solution. The second value is the average best solution using the results of all runs with a particular parameter setting. For example, the first row shows the overall average and average best solution values found when using 8 processors and varying the remaining parameters over their possible settings.

In the first four rows of Table 6, we see that, for all three problems, using more processors usually leads to better solutions, both in terms of the overall average solution and the average best solution. The next two rows indicate that while doubling the computing time from 200 to 400 seconds leads to better solutions, the difference is very slight as the average improvement is less than 0.1%.

The strategy used by the master processor in distributing solutions appears to have little effect on the average solution value and the average best solution value. For all three problems, the average solutions generated by the two strategies are within 0.04% of each other. The average best solutions of the two strategies differ by at most 0.05%. It appears that there is little to be gained by trying to inject some diversification into the procedure by having the master processor distribute randomly selected solutions, since the strategy of sending the best solution appears to work equally well.

In the last five rows of Table 6, we present the results when the algorithm is run with 64 processors and the number of nodes assigned to solving the set covering problem is varied. The results suggest that using more set covering solvers leads to better solutions. In particular, the configuration with 8 set covering solvers generates the best results for all three problems based on average solution value and average best solution value.

4.3. Measuring the Parallel Speedup

For exact algorithms that obtain the same, optimal solution on multiple runs, the impact of parallelism can be determined by studying how the time required to complete the computa-

Table 6: Summary of the effect of various parameters

	Problem					
	G15		T385		L25	
Parameter Setting	Overall Average	Average Best	Overall Average	Average Best	Overall Average	Average Best
8 processors	1344.8	1342.7	24443.3	24406.6	16966.2	16864.8
16 processors	1344.0	1341.8	24440.2	24400.2	16950.0	16821.9
32 processors	1342.8	1340.7	24425.4	24390.3	16889.6	16800.3
64 processors	1342.6	1340.4	24415.0	24383.1	16839.8	16790.1
200 seconds	1343.9	1341.8	24434.3	24396.3	16920.1	16822.4
400 seconds	1343.0	1340.8	24424.6	24391.4	16879.9	16801.8
Random solution	1343.6	1341.4	24431.6	24394.8	16905.8	16808.7
Best solution	1343.3	1341.2	24427.3	24392.9	16894.2	16815.4
64 processors, 0 IP solvers	1343.4	1341.4	24424.4	24386.8	16847.8	16794.1
64 processors, 1 IP solvers	1343.0	1341.3	24415.8	24381.6	16836.0	16797.8
64 processors, 2 IP solvers	1342.2	1339.9	24411.7	24385.7	16828.3	16788.5
64 processors, 4 IP solvers	1341.7	1339.7	24408.1	24378.5	16847.1	16780.0
64 processors, 8 IP solvers	1341.6	1339.7	24401.5	24372.4	16825.4	16773.7

tion as the number of processors is varied. For metaheuristics where multiple runs typically produce different solutions, it is more difficult to assess the effectiveness of parallelism. Such difficulties are addressed in Alba and Luque (2005) where they discuss different ways of measuring the speedup obtained via parallelism.

One of the most widely used measures of a parallel algorithm's effectiveness is its parallel speedup, defined to be the ratio t_s/t_p, where t_s is the amount of time required by a single sequential computer and t_p represents the amount of time required for the parallel computation when using p processors. For many computational tasks, this quantity is easy to measure and a linear parallel speedup of p is typically the goal. However, in our experiments, this measure does not apply directly since we typically obtain different final solutions from different runs of the algorithm. Therefore, in our study of the parallel speedup, we fix a total amount of computing time (i.e., the total allowed wallclock time multiplied by the number of processors), and then compare the average solution value and best solution value when using a different number of processors and a fixed amount of computing time.

In this experiment, we again ran our cooperative parallel algorithm on three problems (G15, T385, and L25). For each problem, we ran the algorithm 20 times with four different configurations: 8 processors, 1 set covering solver, 800 seconds; 16 processors, 2 set covering

Table 7: Analyzing the parallel speedup for three problems

Problem	Number of Processors	Total Time (seconds)	Average Solution	Best Solution
G15	8	800	1342.9	1340.0
	16	400	1342.2	1339.4
	32	200	1342.0	1339.3
	64	100	1342.4	1338.8
T385	8	800	24428.4	24384.5
	16	400	24408.6	24371.2
	32	200	24409.1	24371.2
	64	100	24405.8	24370.6
L25	8	800	16940.0	16809.3
	16	400	16906.5	16782.0
	32	200	16905.2	16797.2
	64	100	16920.4	16796.9

solvers, 400 seconds; 32 processors, 4 set covering solvers, 200 seconds; 64 processors, 8 set covering solvers, 100 seconds. In each case, we have a total computing time of 6400 seconds, and we use one set covering solver for every eight heuristic solvers.

The results of this experiment are summarized in Table 7. For all three problems, we generate solutions of roughly the same quality when using the configurations with 16, 32, or 64 processors, while the solutions generated with 8 processors for 800 seconds are generally the worst. The configuration with 64 processors finds the best overall solution to two problems, while the configuration with 16 processors finds the best solution to the largest problem, L25. These results suggest that the parallel speedup of our algorithm is quite satisfactory as we are able to obtain solutions of equivalent (or better) quality by cutting the computation time in half and doubling the number of processors.

We analyze the parallel speedup in more detail by studying the solution trajectories under the different configurations. For all 20 runs on the three problems, we had the master processor record the best solution found every five seconds. We then calculated the average solution value at each time interval for each of the configurations. The average solution trajectories are given in Figures 7, 8, and 9. In each figure, there are two key observations. A steep slope indicates more rapid convergence towards a minimum. The final ending point of the trajectory represents the average final solution that was generated in the allotted

time limit. These plots clearly show that, by using more processors, our algorithm finds better solutions more quickly. In addition, it is interesting to note that when running the algorithm with 8 processors, we typically reach a point of diminishing returns with respect to time where the search trajectory flattens out. In contrast, for the 64 processor runs of 100 seconds, we observe that the solution value decreases almost linearly with time.

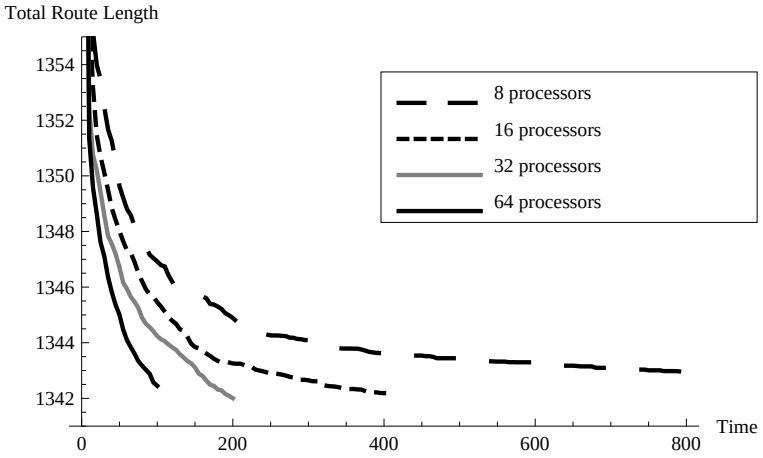


Figure 7: Average solution trajectories for problem G15

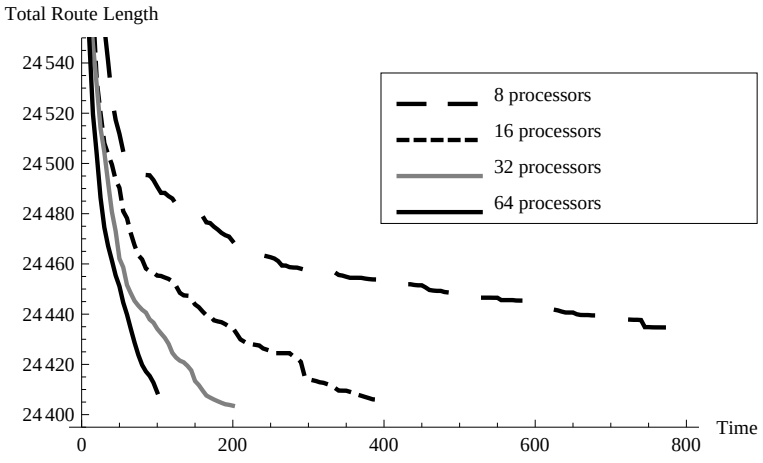


Figure 8: Average solution trajectories for problem T385

These experiments indicate that, for up to 64 processors, doubling the number of processors generally allows us to discover solutions of roughly equivalent quality in about half the time. Furthermore, the solution trajectories in Figures 7–9 indicate that increasing the number of processors allows the algorithm to improve solutions at a much higher rate. However,

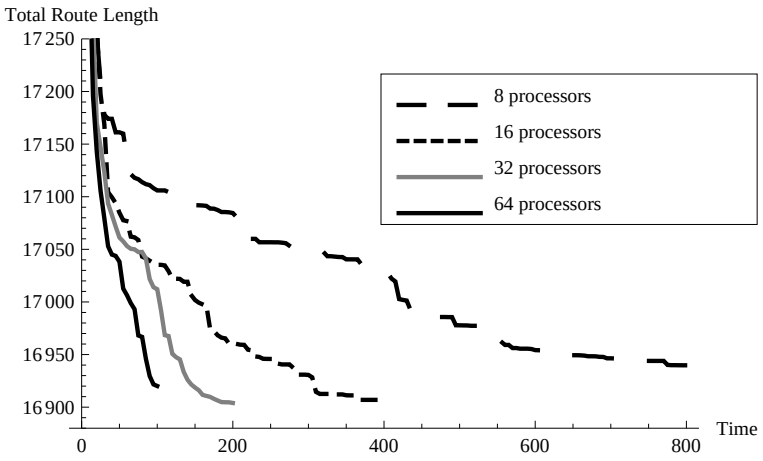


Figure 9: Average solution trajectories for problem L25

it is important to note that this behavior cannot continue indefinitely. First, the set covering solvers do not play a useful role in the algorithm until after the heuristic solvers have written routes to the directories in the course of running Algorithm 1. Second, since reading and writing to disk in a high performance cluster requires the data to travel over a network, our file-based method of distributing information to the set covering solvers will eventually reach a scalability limit as well. This limit is likely very dependent on the specifics of the cluster's network and file system, and we expect that at some point the performance of the algorithm will actually begin to worsen as we add additional processors due to contention over these shared resources. Nevertheless, running a 64 processor configuration for a hundred seconds generally produces very good solutions and the solution quality improves rapidly with time.

5. Conclusions

Developing and implementing effective parallel metaheuristics for the VRP and other difficult combinatorial optimization problems is a challenging task. In this section, we discuss some of the lessons we learned during the course of our work and present our conclusions.

One very practical issue we encountered early in the development of our algorithm is related to the comparison of results produced by multiple runs on a high performance computing (HPC) cluster. Many HPC clusters are heterogeneous in nature, with different compute nodes containing different types of processors and varying amounts of memory per compute

node. Additionally, some clusters allow multiple jobs to run on a single compute node so that one user's code can compete with another (potentially unknown) user for resources such as memory and disk access. In early experiments, we were unaware of these issues when testing and timing various parts of our procedure using a small number of processors. When performance and solution quality decreased, we did not consider the possibility that it was due to a hardware change and instead tried to find algorithmic and parameter changes that could be responsible for the observed decreases. Although we eventually discovered that the majority of these fluctuations in performance were due to the job running on different types of compute nodes, we spent a considerable amount of time and effort looking elsewhere for the sources of the variations. We note that this issue can be avoided on many HPC systems by specifying specific hardware requirements for a job when submitting it to the scheduler. However, this can lead to the job sitting in a queue for a period of time due to the additional restrictions this imposes.

Our algorithm conforms to the master-slave paradigm which often results in unacceptable scalability issues due to communication bottlenecks involving the master processor. When developing our algorithm, we used various profiling tools in order to determine the most time-consuming portions of the parallel algorithm, in terms of both computation and communication. These tools allowed us to quickly observe communication bottlenecks, eventually leading us to create the file-based approach for information diffusion. By using the file system rather than processor-to-processor communication over the network, we were able to eliminate any communication bottlenecks when using up to 128 slave processors. This method of information diffusion also allowed our algorithm to exhibit a satisfactory parallel speedup, and for the configurations we experimented with, the observed solution trajectories indicate that solutions of similar quality can be achieved in much less time by increasing the number of processors.

Our work in this paper focused on generating solutions for the VRP. However, it should be clear that our procedure can handle more complex variants of the VRP by modifying the RRTR algorithm to account for additional constraints since many VRP variants can be modeled as a set covering or set partitioning problem. Furthermore, our algorithm's use of both metaheuristic solvers and set covering solvers could be effective for other difficult combinatorial optimization problems that can be formulated as a relatively simple integer program that permits one to quickly generate new columns via heuristics. For example, crew scheduling and certain facility location problems share this structure and have been

the subject of a large amount of research involving both heuristic and exact approaches. For these problems, it would also be possible to generate columns for the underlying set covering problem by running a heuristic on a smaller problem instance as we did for the VRP. This approach could be particularly effective for larger problem sizes where running a heuristic on the entire problem instance is computationally expensive. Although combining metaheuristic and integer programming methods for the VRP led to modest improvements over a non-cooperative version involving only the metaheuristic algorithm, the gain could be larger for other problems where metaheuristics are less effective than for the VRP.

Finally, we emphasize the fact that as the major producers of computing hardware move towards multi-core and so-called "many core" processors, parallel processing will be available in nearly every laptop and desktop computer, providing an extremely inexpensive platform for shared-memory parallel computing. Our experiments illustrate that running an algorithm on a single computer with four or eight cores leads to very good solutions in a small amount of time. We hope that the algorithmic and implementation details presented here will allow others to take advantage of this increasingly ubiquitous computational resource.

Notes

The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. Edward Wasil was supported in part by a Kogod Research Professorship at American University.

References

Agarwal, Y., K. Mathur, H.M. Salkin. 1989. A set-partitioning-based exact algorithm for the Vehicle Routing Problem. *Networks* **19** 731–749.

Alba, E., B. Dorronsoro. 2004. Solving the vehicle routing problem by using cellular genetic algorithms. J. Gottlieb, G. Raidl, eds., *EvoCOP*, *Lecture Notes in Computer Science*, vol. 3004. Springer, Berlin, 11–20.

Alba, E., B. Dorronsoro. 2008. *Cellular Genetic Algorithms*. Springer, New York.

Alba, E., G. Luque. 2005. Measuring the performance of parallel metaheuristics. E. Alba, ed., *Parallel Metaheuristics*. Wiley-Interscience, 43–62.

Alvarenga, G.B., G.R. Mateus, G. de Tomi. 2007. A genetic and set partitioning two-phase approach for the vehicle routing problem with time windows. *Computers & Operations Research* **34** 1561–1584.

Bixby, A. 1998. Polyhedral analysis and effective algorithms for the capacitated vehicle routing problem. Ph.D. thesis, Northwestern University, Evanston, IL.

Bramel, J., D. Simchi-Levi. 2001. Set-covering-based algorithms for the capacitated VRP. P. Toth, D. Vigo, eds., *The Vehicle Routing Problem*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 85–108.

Chao, I-M., B. Golden, E. Wasil. 1995. An improved heuristic for the period vehicle routing problem. *Networks* **26** 22–44.

Christofides, N., S. Eilon. 1969. An algorithm for the vehicle dispatching problem. *Operations Research Quarterly* **20** 309–318.

Christofides, N., A. Mingozzi, P. Toth. 1979. The vehicle routing problem. N. Christofides, A. Mingozzi, P. Toth, C. Sandi, eds., *Combinatorial Optimization*. John Wiley, 315–338.

COIN-OR. 2010. COmputational INfrastructure for Operations Research. `http://www.coin-or.org/projects/VRPH.xml`.

Cordeau, J.-F., M. Gendreau, A. Hertz, G. Laporte, J.-S. Sormany. 2005. New heuristics for the vehicle routing problem. A. Langevin, D. Riopel, eds., *Logistics Systems: Design and Optimization*. Springer, 270–297.

Crainic, T., , H. Nourredine. 2005. Parallel meta-heuristics applications. E. Alba et al., ed., *Parallel Metaheuristics*. John Wiley and Sons, Hoboken, NJ, 447–494.

Crainic, T. 2008. Parallel solution methods for vehicle routing problems. B. Golden, S. Raghavan, E. Wasil, eds., *The Vehicle Routing Problem: Latest Advances and New Challenges*. Springer, New York, 171–198.

Dantzig, G., J. Ramser. 1959. The truck dispatching problem. *Management Science* **6** 80–91.

Desrochers, M., J. Desrosiers, M.M. Solomon. 1992. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research* **40** 342–354.

Doerner, Karl, Richard F. Hartl, Guenter Kiechle, Mária Lucká, Marc Reimann. 2004. Parallel ant systems for the capacitated vehicle routing problem. Jens Gottlieb, Günther R. Raidl, eds., *EvoCOP*, *Lecture Notes in Computer Science*, vol. 3004. Springer, 72–83.

Dorronsoro, B., D. Arias, F. Luna, A.J. Nebro, E. Alba. 2007. A grid-based hybrid cellular genetic algorithm for very large scale instances of the CVRP. W. Smari, ed., *2007 High Performance Computing & Simulation Conference (HPCS 2007)*. Czech Republic, 759–765.

Franceschi, R. De, M. Fischetti, P. Toth. 2006. A new ILP-based refinement heuristic for vehicle routing problems. *Mathematical Programming* **105** 471–499.

Gendreau, M., Y. Potvin, O. Bräysy, G. Hasle, A. Løkketangen. 2008. Metaheuristics for the vehicle routing problem and its extensions: A categorized bibliography. B. Golden, S. Raghavan, E. Wasil, eds., *The Vehicle Routing Problem: Latest Advances and New Challenges*. Springer, New York, 143–169.

Golden, B., S. Raghavan, E. Wasil. 2008. *The Vehicle Routing Problem: Latest Advances and New Challenges*. Springer, New York.

Golden, B., E. Wasil, J. Kelly, I-M. Chao. 1998. The impact of metaheuristics on solving the vehicle routing problem: Algorithms, problem sets, and computational results. T. Crainic, G. Laporte, eds., *Fleet Management and Logistics*. Kluwer, Boston, 33–56.

Groër, C., B. Golden, E. Wasil. 2010. A library of local search heuristics for the vehicle routing problem. *Mathematical Programming Computation* , forthcoming.

Hadjiconstantinou, E., N. Christofides, A. Mingozzi. 1995. A new exact algorithm for the vehicle routing problem based on q-paths and k-shortest paths relaxation. *Annals of Operations Research* **61** 21–43.

Le Bouthillier, A., T. Crainic. 2005. A cooperative parallel meta-heuristic for the vehicle routing problem with time windows. *Computers & Operations Research* **32** 1685–1708.

Le Bouthillier, A., T. Crainic, P. Kropf. 2005. Towards a guided cooperative search. Tech. Rep. CRT-05-09, Centre de recherche sur les transports, Université de Montréal, Montréal, Canada.

Li, F., B. Golden, E. Wasil. 2005. Very large-scale vehicle routing: New test problems, algorithms, and results. *Computers & Operations Research* **32** 1165–1179.

Lin, S., B.W. Kernighan. 1973. An effective heuristic algorithm for the traveling salesman problem. *Operations Research* **21** 2245–2269.

Mester, D., O. Bräysy. 2007. Active-guided evolution strategies for large-scale vehicle routing problems. *Computers & Operations Research* **34** 2964–2975.

Nagata, Y., O. Bräysy. 2008. Efficient local search limitation strategies for vehicle routing problems. J. Hemert, C. Cotta, eds., *EvoCOP*, *Lecture Notes in Computer Science*, vol. 4972. Springer, Berlin, 48–60.

Nagata, Y., O. Bräysy. 2009. Edge assembly-based memetic algorithm for the capacitated vehicle routing problem. *Networks* **0** 0—0.

Or, I. 1976. Traveling salesman-type combinatorial problems and their relation to the logistics of blood banking. Ph.D. thesis, Northwestern University, Evanston, IL.

Osman, I. 1993. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research* **41** 421–451.

Pisinger, D., S. Røpke. 2007. A general heuristic for vehicle routing problems. *Computers & Operations Research* **34** 2403–2435.

Prins, Christian. 2009. A grasp $\times$ evolutionary local search hybrid for the vehicle routing problem. Francisco Baptista Pereira, Jorge Tavares, eds., *Bio-inspired Algorithms for the Vehicle Routing Problem*, *Studies in Computational Intelligence*, vol. 161. Springer, 35–53.

Ralphs, T., L. Kopman, W. Pulleyblank, L. Trotter. 2003. On the capacitated vehicle routing problem. *Mathematical Programming* **94** 343–359.

Rego, C. 2001. Node-ejection chains for the vehicle routing problem: Sequential and parallel algorithms. *Parallel Computing* **27** 201–222.

Rochat, Y., E. Taillard. 1995. Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics* **1** 147–167.

Schulze, J., T. Fahle. 1999. A parallel algorithm for the vehicle routing problem with time window constraints. *Annals of Operations Research* **86** 585–607.

Taillard, E. 1993a. Parallel iterative search methods for vehicle routing problems. *Networks* **23** 661–676.

Taillard, E. 1993b. VRP benchmarks. `http://mistic.heig-vd.ch/taillard/problemes.dir/vrp.dir/vrp.html`.

Taillard, E., B. Boffey. 1999. A heuristic column generation method for the heterogeneous fleet VRP. *RAIRO* **33** 1–14.

Toth, P., A. Tramontani. 2008. An integer linear programming local search for capacitated vehicle routing problems. B. Golden, S. Raghavan, E. Wasil, eds., *The Vehicle Routing Problem: Latest Advances and New Challenges*. Springer, New York, 275–295.

Yellow, P.C. 1970. A computational modification to the savings method of vehicle scheduling. *Operations Research Quarterly* **21** 281–293.