Asynchronous Parallel Ant-Colony Optimization Strategies: Application to the Multi-Depot Vehicle Scheduling Problem with Line Exchanges

David Semedo, Pedro Barahona, and Pedro Medeiros

NOVA-LINCS DI-FCT-UNL
Laboratory for Computer Science and Informatics
Faculdade de Ciências e Tecnologia, 2829-516 Caparica, Portugal
df.semedo@campus.fct.unl.pt
{pb,pdm}@fct.unl.pt

Abstract. The Multi-Depot Vehicle Scheduling Problem with Line Exchanges is an NP-*hard* combinatorial optimization problem arising in transit service companies.
Due to the complexity of the problem and the large size of the search space, complete search cannot be used in order to guarantee optimality. Hence, we adopted the Ant-Colony Optimization (ACO) metaheuristic. ACO can be improved using parallel computing. However, in order to maintain the semantics of the original ACO metaheuristic, parallelization strategies need to be synchronous, originating the straggler problem. We propose three shared-memory asynchronous parallelization strategies that break the original ACO algorithm semantics and differ essentially on the degree of concurrency allowed while manipulating the learned information.
The results show that two of our asynchronous strategies outperform synchronous ones in terms of the speedup achieved and also solution quality. Additionally, the speedup increases as the level of concurrency allowed in asynchronous strategies increases, without sacrificing the quality of the solutions obtained. The proposed strategies effectiveness is validated through an analysis and discussion of the ACO convergence and scalability, in terms of search depth and number of threads used in a *shared-memory* architecture, with *multi-core* processors.

Keywords: Multi-Depot Vehicle Scheduling Problem (MD-VSP), Parallel Computing, Asynchronism, Concurrency, Ant-Colony Optimization (ACO)

1 Introduction

The Multi-Depot Vehicle Scheduling Problem with Line Exchanges (MD-VSPLE), a variant of the more general Vehicle Scheduling Problem (VSP), is an NP-*hard* [1, 12] combinatorial optimization arising in transit services companies, which consists of producing a schedule for each vehicle such that each trip is

covered by only one vehicle and the operational costs are minimized, whilst satisfying a set of constraints. In this variant, more than one depot and line exchanges are considered. Additionally, we assume that the fleet of vehicles is heterogeneous (i.e. there are several types of vehicles each with different characteristics and constraints).

In a real life context, instances of this problem, specially in urban areas, have a large number of variables, which, due to the problem complexity and the additional constraints taken into account in this work, makes impractical solving it to optimality.

Metaheuristics [10] are defined as methods that orchestrate an interaction between local improvement procedures (e.g. Local Search algorithms) and higher level strategies in order to develop search strategies capable of escaping local optima and guiding the search towards promising regions of the search space, by sacrificing optimality. The Ant-Colony Optimization (ACO) metaheuristic, originally proposed by Dorigo [9], is a metaheuristic inspired in the behaviour of real ants which use pheromones as a communication medium.

A stochastic learning procedure allows the agents to build a wide variety of solutions, leading to a better (in depth) exploration of the search space, than greedy procedures. Agents search experience influences future iterations which makes ACO similar to reinforcement learning procedures.

Despite of the proven capabilities of ACO [14], for complex problems with large search spaces, as the one we address in this work, it may fail to find quality solutions in a reasonable amount of time.

Parallel computing is commonly used to improve ACO effectiveness in terms of computation time, solution quality or both. Parallel strategies [15, 6] can target either *shared-memory* (SM) or *distributed* architectures, either using CPUs, GPUs or both. Distributed architectures offer better scalability than SM ones, however, communication within nodes is more expensive due to the necessity of transferring data between disjoint address spaces. ACO agents rely on a high-level of communication in order to report their findings and improve the learned information about the search space. Therefore, in this paper we target SM architectures.

Despite of the architecture used, in order to preserve the semantics of the original ACO metaheuristics, parallel strategies must be synchronous, i.e., a synchronization point must exist after each iteration where the new information obtained by each agent must be used to update the global learned information. We propose asynchronous strategies, that break the semantics of the original metaheuristic in order to achieve better performance, and differ on the degree of concurrency allowed while manipulating the learned information. Since synchronization points after each iteration are removed, it is possible to achieve more parallelism. We propose a set of modifications to the original procedure in order to preserve convergence properties.

The proposed strategies effectiveness is validated through an analysis and discussion of the ACO convergence and scalability, in terms of search depth and

number of threads used, in a *multi*-processor SM architecture, with *multi-core* processors. We compare the results achieved with a synchronous ACO version.

We also analyse the *single*-processor architecture with *multi-core* processors due to the fact that it corresponds to common personal computers (PC) architectures. For companies that face the MD-VSPLE problem, it would be of great value to have a decision support tool capable of effectively solving this problem and which would not require expensive hardware.

The remainder of this paper is structured as follows. Section 2 describes and formalizes the MD-VSPLE as well as additional constraints taken into account. The sequential algorithm using ACO will also be described in this section. Section 3 presents a review of the literature on parallel ACO. In section 4 we present our proposed asynchronous strategies and we discuss important implementation details. In section 5 experimental results are presented and discussed. Finally, section 6 concludes the paper giving an outlook on future research.

2 Multi-Depot Vehicle Scheduling Problem with Line Exchanges

The MD-VSPLE is an NP-*hard* [1, 12] problem in which the objective is to find a schedule that minimizes the operational costs, whilst satisfying a set of constraints. Since it is our goal that the variant addressed in this work matches real life situations, we also take into account an heterogeneous fleet, in which for each type of vehicle, additional constraints are enforced. To the best of our knowledge, the variant of the MD-VSPLE addressed in this work has not been addressed yet.

2.1 Problem Formulation

The MD-VSPLE can be formulated as follows:

Let L be a set of lines to be covered and T be a set of n time-tabled departures $T_1, \ldots, T_n$, such that for each $j \in [1, n]$, s_j and e_j denote the starting and ending time, respectively, it_j and ft_j denote the initial and final terminus, respectively, and $l_j \in L$ the corresponding line of departure j. Let D be a set of m depots $D_1, \ldots, D_m$ where each depot D_k has a total capacity of r_k (with $r_k \leq n$) vehicles.

Let $\tau_{i,j}$ be the travel time for a vehicle to go from the arrival terminal of departure T_i to the terminal of departure T_j, where $i, j \in [1, n]$. A pair of trips (T_i, T_j) is said to be *compatible* iff the same vehicle can cover trips T_i and T_j in the sequence, i.e, the following inequality is verified:

$$e_i + \epsilon + \tau_{i,j} \leq s_j \tag{1}$$

where ϵ is the minimum time between an arrival and a departure, in order to allow preparation of the vehicle and/or driver exchange. A vehicle *scheduling* S_l, performed by a vehicle v_i stationed at depot D_k, consists of a sequence of tasks.

A task may correspond to performing a departure (*service*), stay idle, a trip without passengers to/from a depot, a line exchange or a maintenance episode. Departures tasks in S_l must be pair-wise compatible departures, assuming the S_l sequence order.

The following hard-constraints are enforced: each departure is covered by exactly one vehicle, each vehicle is assigned to a *schedule* with pairwise compatible departures, returning to its depot at the end of the *service*.

We also take into account additional constraints like the maximum daily working time for each vehicle type, the obligation and duration of maintenance episodes and the amount of time a vehicle can stay idle in a terminal.

Objective Function. The objective function F to be minimized takes into account the investment cost of each vehicle measured by its daily depreciation (€/day) (including the cost of maintenance episodes) and the running cost of the total distance travelled by each vehicle. Each of the costs referred change according to the vehicle type.

Let Nb be the total number of vehicles used (i.e. the total number of services in the solution). The function F of a solution *sol* is defined as follows:

$$F(sol) = \sum_{i=1}^{Nb} distTravelled(v_i) * distCatCost(v_i) + dailyCatCost(v_i) \tag{2}$$

where $distTravelled(v_i)$ is the total distance travelled by the vehicle v_i in the solution *sol*, $distCatCost(v_i)$ and $dailyCatCost(v_i)$ is the running cost and the daily depreciation, respectively, of vehicles from the category of vehicle v_i. The unused vehicles slots in each depot D_k do not contribute to the total cost.

2.2 Ant-Colony Optimization Metaheuristic

The ACO metaheuristic is a population-based metaheuristic for solving hard-combinatorial optimization problems based on a colony of agents that construct solutions using a pheromone model that corresponds to a parametrized probability distribution over the solution space, adding components to a partial solution. These components are chosen based on heuristic information of the problem and pheromone trails.

Pheromone trails are modelled as distributed numerical information of the search space graph, which varies according to the problem being solved. Agents cooperatively update pheromone trails at runtime in order to reflect their search experience. The stochastic component is essential to achieve solution diversity and avoid being stuck at a local optima.

The first ACO implementation proposed was the Ant System (AS) algorithm [8] which was applied to the Travelling Salesman Problem (TSP). Based on the AS, more sophisticated implementations emerged, such as the MAX-MIN ACO [16] and the Ant Colony System [7] (ACS) algorithms.

The base algorithm scheme, shown in algorithm 1, is roughly the same.

Algorithm 1 Ant-Colony Optimization.[10]

```
1: procedure Ant-Colony Optimization(problem) return best solution found
2:     Initialization
3:     while termination condition not met do
4:         ConstructAntSolutions
5:         UpdatePheromoneTrails
6:     end while
        return best solution
7: end procedure
```

In the *Initialization* step pheromone variables are initialized. Then, it iterates until some termination criteria is met (e.g. stop after P iterations). In the *ConstructAntSolutions* a set of N agents perform a constructive search procedure guided by heuristic information and by the pheromone trails. Lastly, in the *UpdatePheromoneTrails* step, the pheromone trails are updated such that good solutions will be more desirable. In order to avoid a fast convergence of all the agents towards sub-optimal solutions, pheromone evaporation is applied.

2.3 Sequential ACO Algorithm for the MD-VSPLE

By sacrificing optimality, efficient approximative algorithms, capable of achieving high-quality solutions in a reasonable amount of time can be developed. Approximative algorithms can be either *constructive*, in which solutions are built incrementally, or *Reparative*, in which the algorithm attempts to improve a complete solution at each iteration.

Reparative procedures require the definition of a move operator which is used to perform local moves from one solution to another, at each iteration. Due to the complexity of our problem, more concretely to the large number of constraints involved, it is not trivial to design a move operator that keeps the solutions consistent and without constraint violations, also taking into account that the operational costs must be minimized. Therefore, *constructive* approaches should be more adequate and yield better results in our problem.

We thus adopt the ACO since it not only uses a *constructive* approach but also uses a reinforcement learning procedure that makes the ACO a superior algorithm when compared to greedy ones.

ACO for MD-VSPLE. Our algorithm implements the AS algorithm, which despite of the fact that it may be not as robust as the MAX-MIN and ACS algorithms in its sequential versio (as discussed in section 4) it is more suitable for parallel implementations.

The pheromone trails model should hold information of the desirability of each solution component. Each component added to a solution corresponds to an assignment of a vehicle to a departure. Let P be the set of possible locations from where a vehicle can be picked when assigning a vehicle to a given departure. Our model corresponds to a $|P| \times |T|$ matrix, i.e., we assign a pheromone value

τ_{ij} to each pair $\langle l_i, T_j \rangle$ that denotes the desirability of assigning vehicles that are in a location $l_i \in P$ to a departure $T_j \in T$.

At each step t each agent k uses a probability distribution to select the solution component c_i^j to be added to a partial solution s_p, i.e, from which location $l_i \in P$ should a vehicle be picked to a departure T_j. We use the most widely used probability distribution, from AS. Let $Admiss_k$ be the set of vehicles that can be assigned (do not violate any hard constraint) to a departure T_j. The probability distribution is defined as follows:

$$p_k(c_i^j|s_p) = \begin{cases} \dfrac{\tau_{ij}^{\alpha} \cdot \left[\eta(c_i^j, v_a)\right]^{\beta}}{\sum_{l \in Admiss_k} \tau_{lj}^{\alpha} \cdot \left[\eta(c_l^j, v_a)\right]^{\beta}} & , v_a \in Admissible_k \\ 0 & , otherwise \end{cases} \quad (3)$$

where $\eta(\cdot)$ is a function that assigns to each c_i^j in which a vehicle $v_a \in Admiss_k$, an heuristic value that evaluates the quality of this assignment in the context of the problem, i.e., corresponds to the utility value of a given vehicle to perform a given departure. The α and β parameters are used to control the influence of pheromone values and heuristic information, respectively, on the algorithm behaviour.

The utility $\eta(c_i^j, v_a)$ of a vehicle v_a located on a terminal tm_o, performing a departure j starting on terminal tm_j, is a function $\eta : \mathbb{R}_{\geq 0} \mapsto]0, 1]$ defined as follows:

$$\eta(c_i^j, v_a) = e^{\dfrac{-dist(tm_o, tm_j) * distCatCost(v_a) * k_1 + dTerm(tm_o, tm_j) * k_2}{k_3}} \quad (4)$$

where $dist(t_1, t_2)$ is the distance from t_1 to t_2, $dTerm(t_1, t_2)$ is a function that returns 1 if $t_1 = t_2$ or 0 otherwise, and k_w, where $w \in [1, 2, 3]$, are weights that can be parametrized.

In the end of an ACO iteration pheromone trails are evaporated and updated. Pheromone evaporation is implemented as follows:

$$\tau_{ij} \leftarrow (1 - \rho) * \tau_{ij} \quad (5)$$

where $0 < \rho \leq 1$ is a parameter that denotes the rate of evaporation. If a solution component is not used by an agent, its corresponding pheromone value τ_{ij} decreases exponentially in the number of iterations, allowing the algorithm to avoid bad solution components, since its desirability is reduced.

After performing the pheromone evaporation step, the pheromone values are updated with learned information from each agent. The update is defined as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1} \Delta\tau_{ij}^k \quad (6)$$

where $\Delta\tau_{ij}^k$ denotes the amount of pheromone deposited by agent k on the solution components that belong the solution s_k obtained:

$$\Delta\tau_{ij}^k = \begin{cases} \dfrac{\lambda}{F(s_k)} & , \; c_i^j \in s_k \\ 0 & , \; otherwise \end{cases} \tag{7}$$

where λ is a parameter used to control the amount of pheromone deposited. With this expression, the amount of pheromone deposited by each agent depends on the quality of the solutions obtained. Therefore, agents that obtain better solutions will deposit a larger amount of pheromone, favouring good solution components.

ACO Agent algorithm. As previously stated, each agent k constructs a solution incrementally by adding solution components iteratively and using the probability distribution p_k. Departures are sorted by starting minute. At each iteration of the agent algorithm, a set of admissible vehicles is built. Admissible vehicles are vehicles that do not violate any hard-constraint if assigned to the departure being addressed. While building the set of admissible vehicles, the algorithm sends to maintenance vehicles that need maintenance, i.e., vehicles that would violate the maintenance constraint if they performed the departure.

In this paper we explore parallel ACO metaheuristics, in order to improve the presented ACO algorithm performance. Concretely, we propose a new parallel ACO algorithm that explores the fact that the pheromone update and evaporation mechanisms demand a synchronization point at the end of an iteration, originating the straggler problem.

3 Related Work

Parallel computing has been used as a mean of improving ACO algorithms performance, therefore, several different algorithms using different strategies have been proposed [15]. The most promising models for achieving high computational efficiency are the *coarse-grain master-slave*, in which a master process that manages global information controls a group of search processes, and *multicolony*, in which each processor executes a colony.

In [5] the authors propose an SM ACO algorithm that uses the *coarse grain master-slave* model and compare SM architectures with distributed architectures. The SM version of the proposed algorithm outperformed the distributed version. The main reason is the communication overhead imposed by the network.

Delisle et al. [4] presented an SM parallel implementation using OpenMP of ACO in order to solve an industrial scheduling problem. As the authors stated, the ACO algorithm follows exactly a *fork-join* scheme. Additionally, despite the fact that with SM no explicit communications are needed, therefore omitting communications overheads, a synchronization barrier that occurs when each ant

finishes the search and the pheromone matrix needs to be updated, causes the straggler problem and prevents the algorithm from achieving better performance.

Cipar et al. [3] defined the straggler problem as the situation in which a small number of threads (the stragglers) take longer than the others to execute a given iteration. Since all the threads will eventually be synchronized, all threads will proceed at the speed of the slowest one. As stated in [2] one of the solutions for the straggler problem is to make the algorithm asynchronous. Despite of potentially making the algorithm more complex in order to maintain its semantics and properties, it eliminates completely the straggler problem, and allows the algorithms to use all the available parallelism.

In the context of ACO and in a distributed architecture, Kotsis et al.[11] proposed a *partially asynchronous* parallelization scheme on the AS algorithm to reduce the straggler problem caused by the ACO sychronization barrier. The scheme consists in creating temporary sub-colonies in computer nodes which live for a certain number of iterations. Despite reducing the amount of communication performed the straggler problem still occurs within each sub-colony.

In [17] the authors propose two ACO asynchronous models based on the *cunning* Ant System (cAS) algorithm. The difference in the two models consists in modelling the pheromone matrix as a critical section (AP-cAS) or not (RAP-cAS). The RAP-cAS model achieved the most promising results (in terms of speedup) comparing with (AP-cAS) and a synchronous model. In order to use the cAS algorithm, the problem model must target a *reparative* search scheme, which as discussed previously in section 2.3, is not suited for our problem. Additionally, asynchronism breaks the semantics of the original ACO algorithm and the convergence of the proposed algorithms was not analysed.

4 Parallel Ant-Colony Metaheuristic Models

The *MAX-MIN* and the *ACS* algorithms include mechanisms to avoid premature convergence and consequently may achieve better results than AS, in a sequential setup. However, these mechanisms affect negatively the exploitation of parallel resources: in *MAX-MIN* and *ACS* only the best solution obtained in an iteration is used to perform the update, what forces the master to always wait for all the solutions. Additionally in *ACS* search agents perform local updates while building solutions, originating concurrency on the pheromone matrix. Therefore, AS is more suitable to design an asynchronous parallel strategy. Our algorithms follow the *coarse-grain master-slave* model.

4.1 Asynchronous Model

The main modification of our asynchronous model is the following: the master generates a set of $P \times N$ tasks, where P is the total number of iterations. Each task belongs to a given iteration $p \in P$ and a given search agent $n \in N$. These tasks are executed asynchronously and the results are sent to the master.

It may now occur that a search agent that is constructing a solution views an update to the matrix. This is the situation that breaks the original semantics of the ACO algorithm and can potentially affect negatively the convergence. We show in the next section that this is not the case.

The pheromone update and evaporation steps must be reviewed in order to allow ACO to converge. The following modifications were made:

Pheromone Update - When an agent finishes the search, the master is notified and performs a partial update based on the generated solution;

Pheromone Evaporation - In the original AS algorithm the total number of evaporations performed is equal to P. Since now the agents execute asynchronously it is not trivial to design an equivalent scheme, since agents from an iteration p_1 with $p_1 > p_2$ may finish first than agents from an iteration p_2 and we do not know in each iteration how many agents will finish first, since it depends on the real-time decisions made by the operating system scheduler. In order to achieve an approximate behaviour, an evaporation step is performed after $N/2$ search agents from an iteration p, finish their execution. This assumes that some agents will be slower. With this strategy, evaporation steps may be performed sooner than they would be on the original algorithm, however this is not a problem, since there is an attempt to perform evaporation steps after an average of x agents have finished, although this is not guaranteed. Ideally an evaporation step would be performed every time x search agents finish. We believe that our proposed scheme is a good approximation of this behaviour and its contribution to the algorithm convergence is evaluated in section 5.

Despite of the new algorithm conceptual simplicity, it introduces concurrency on the pheromone matrix. The master will be regularly performing updates while agents will be reading the matrix contents.

Concurrency Models. We propose three different concurrency control models, that differ on the degree of concurrency allowed while manipulating the pheromone matrix. Each model affects the original algorithm semantics in a different manner, therefore, the algorithm convergence for each model must be assessed.

Blocking Matrix (AsyncBM) - In this model when a master is performing updates the whole matrix is blocked and none of the agents can proceed until the update is performed. With this model if an update is performed, all the agents will receive the complete update of all the solutions components;

Blocking Column (AsyncBC)- When an agent is choosing a component c_i^j to a partial solution, it will read an entire column of the pheromone matrix, i.e., selecting from which location i should a vehicle be picked to perform the departure j. In this model, instead of blocking the entire matrix, only the column that is being updated/read is blocked. This ensures that if updates/reads are going to be concurrently performed, each agent will decide

on each assignment based either on the new or the old information, but not on a mix of both;

Lock-Free (AsyncLF)- In the limit, we could allow all the concurrent updates and a search agent can possibly use both information before an update and information after an update, on the same decision. Conceptually this achieves better diversification and in practice maximizes concurrency. This model assumes that reads and writes operations are atomic.

4.2 Technical and Implementation issues

The synchronous ACO algorithm was implemented using OpenMP. The implementation is straightforward. It simply consists in adding a *pragma* annotation in the loop where search agents are launched (line 4 of Algorithm 1).

The base asynchronous algorithm design is based on the thread pool, avoiding consecutive thread creation and destruction overhead, in which the results are placed on a pool of solutions. The overall scheme corresponds to a *1-producer/N-consumers* pattern.

The tasks are placed on a pool of tasks implemented with a lock-free queue, from which threads in the thread pool will consume tasks. When a thread finishes processing a task, i.e., builds a solution, the solution is placed on a solutions pool and the master consumes it by performing the pheromone update and evaporation steps.

In order to achieve high performance, the implementation must be efficient. The proposed concurrency models are implemented as follows:

Blocking Matrix - Each time the master or one of the threads want to access the pheromone matrix they have to acquire a lock. This lock is implemented as a *spin lock* [13];

Blocking Column - In this model we have an array of *read-write* locks with T elements, where each element is the lock of a column;

Lock-Free - The lock-free model is implemented by using C++ *atomics*, ensuring that reads and writes operations are atomic.

5 Results

In this section we present and discuss the results obtained from the tests performed. We implemented a sequential version of the AS algorithm (ASSeq). This sequential implementation was parallelized using OpenMP (ASSync). Our proposed algorithms were implemented in C++.

The main tests were performed on a 4-node NUMA machine with 4 AMD(R) Opteron 6272 processors @ 2.1 GHz, each with 16 cores, and with 64GB RAM (16GB for each NUMA node). We also analysed the execution times on a PC with an Intel(R) Core i7-4700MQ CPU @ 2.40GHz supporting Hyperthreading(R) and with 32GB RAM. The instance of the MD-VSPLE addressed corresponds to a subset of 6 CARRIS[1]bus lines with ascending and descending directions,

with a total of 732 departures and 12 terminal locations. Additionally, 3 vehicle categories and 2 depots are considered, each with 100 vehicles of each category.

Due to the stochastic nature of the algorithms and to achieve more robust results, 5 runs are performed for each configuration.

5.1 Convergence Analysis

In order to assess our algorithm convergence we performed a test in which the number of iterations P is 250 and the number of search agents K is 16. The number of threads used is irrelevant since we are not analysing the speedup. For each iteration $p \in [1, 250]$ of the search, we computed the mean of the objective function of all the 16×5 generated solutions from the 16 agents and 5 runs. We plotted the mean values for each p iteration and we applied a linear regression on the data. The results can be seen in Figure 1. In this figure four plots are shown, one for the ASSync and one for each of our concurrency models. The slopes of the linear regression lines (blue lines) are shown and can be interpreted as a measure of the *intensity* of convergence. The red line shows the best solution obtained during each iteration p. The best solution objective value achieved is shown in the plot title.

We observe that our proposed asynchronous strategies are able to converge as the number of iterations increases. It is worth noting that both the AsyncLF and AsyncBC algorithms outperformed ASSync in terms of convergence intensity and solution quality. The AsyncBM algorithm was not superior in terms of solution quality but was very close to ASSync. The AsyncLF algorithm not only presents the highest convergence (slope of ~ -0.22) but also achieved the best solution in the first 100 iterations. However, it takes longer to converge than the other algorithms since only after iteration 150 the algorithm starts achieving several quality solutions. The AsyncBC and AsyncLF algorithms start achieving quality solutions in less iterations and should be preferred if one wants a good solution quickly.

The ability of achieving good diversity while generating solutions is crucial since it corresponds directly to a better exploration of the search space. As the degree of concurrency increases, search agents that are executing receive updates from agents that have already finished sooner and, this allows the executing agents to perform *better* decisions sooner, i.e., agents start exploring promising regions sooner which increases the chances of achieving quality solutions. Furthermore, since synchronization barriers where each agent receives a copy of an updated matrix are removed and search agents from the same iteration may use different numerical values for the same decision, better diversity is achieved.

5.2 Performance Analysis

In our performance experiments we target the following algorithms: ASSeq, ASSync and our three proposed parallel asynchronous algorithms.

[1] A portuguese bus service transport company.

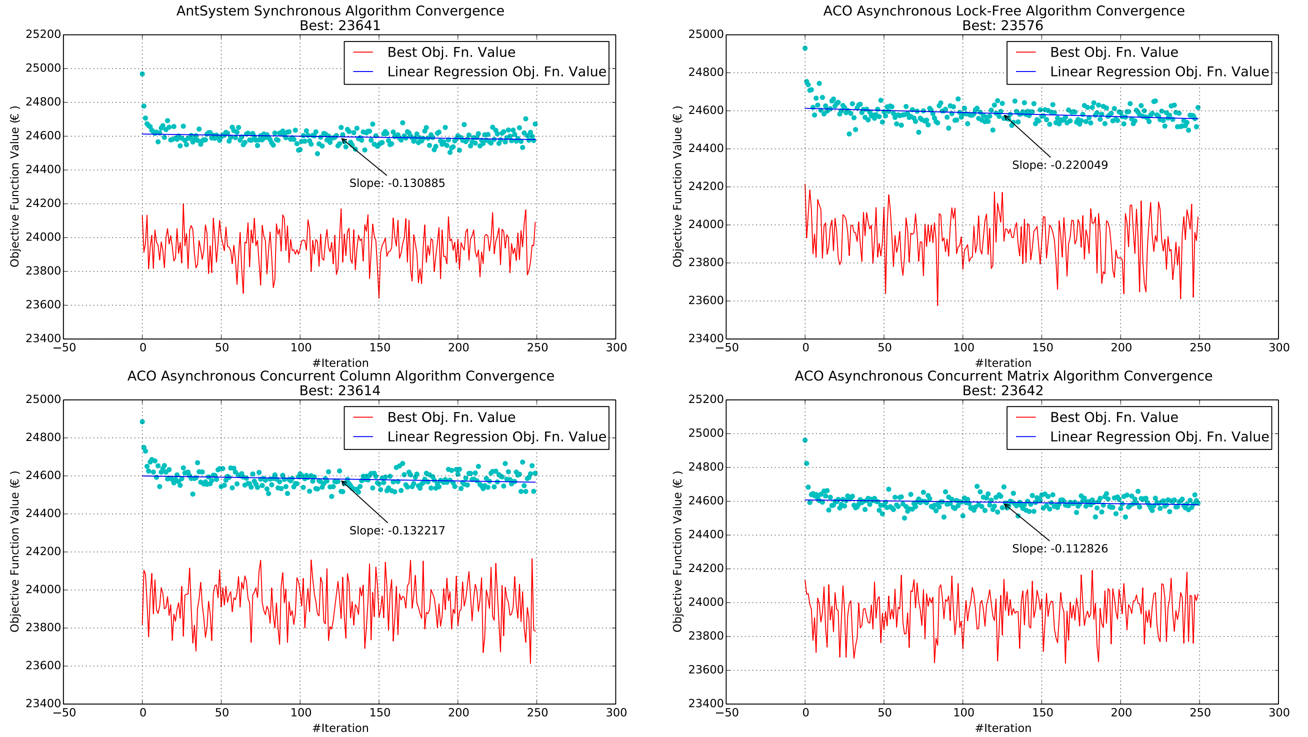

Fig. 1: Convergence Analysis of proposed algorithms with 250 iterations.

The first experiment was performed on the 64 core machine using 64 threads. With this experiment we intend to analyze how the speedups are affected when the search goes deeper (in number of iterations). We run each algorithm using $p \in [10, 50, 100, 250, 500]$ iterations. This experiment was performed with $4, 8, 16$ and 32 search agents. Figure 2 shows the result of the experiment. We observe that in all cases both AsyncLF and ASyncBC algorithms achieves better speedups than ASSync. The AsyncBM algorithm proved to be unstable yielding better speedups than ASSync with 8 and 16 agents but not with 32 and 64.

With 8, 16 and 32 search agents there are threads that were launched but are not working since the number of threads is greater than the number of tasks to be executed. With 64 agents this situation does not occur and each thread gets atleast one agent from each iteration. This explains why speedups are slightly better in general with 64 agents (except for AsyncBM). We observe that AsyncLF achieves almost always better speedups than the other asynchronous strategies. As we expected, and due to the fact that the whole matrix is blocked, the AsyncBM algorithm is the worse asynchronous algorithm. In general and as we expected, we verify that as more concurrency is allowed, the better are the speedups achieved.

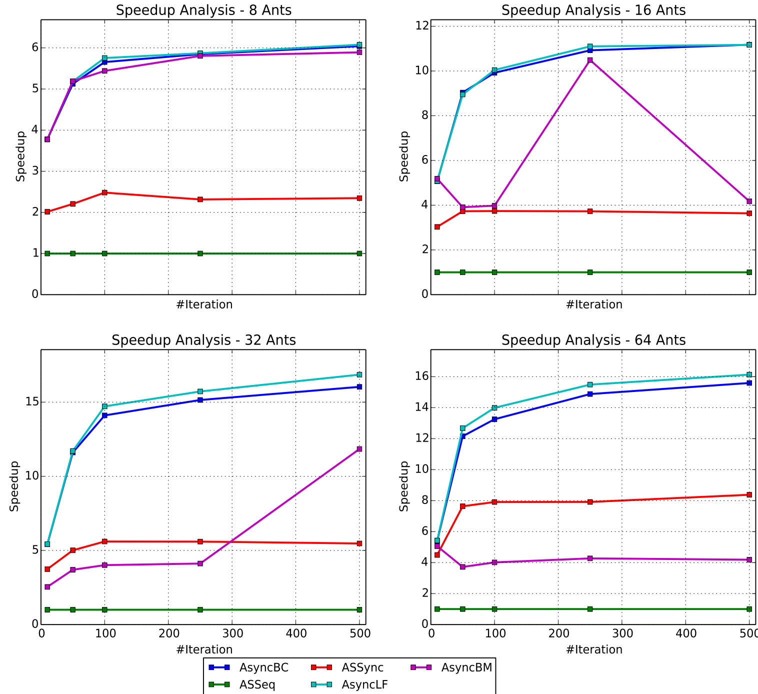

Fig. 2: Speedup Results with 64 threads.

For the second experiment, also performed on the 64 core machine, the number of iterations was fixed to 1000. This experiment aims to evaluate the scalability of our algorithms as the number of threads increases. We run each algorithm using $th \in [2, 4, 8, 16, 32, 64]$ threads and, for each number of threads th we measured the speedup achieved. Like in the previous experiment, we tested with $4, 8, 16$ and 32 search agents. Figure 3 shows the result of the experiment.

We observe a speedup increase as the number of threads increases. In all configurations, the overhead of concurrency mechanisms is only observed when more than 8 threads are used. Like in the previous experiment, the best results were achieved with 64 search agents per iteration. Both results indicate that as we increase the amount of work in each iteration, better speedups are achieved, i.e., parallel resources exploitation is more effective.

Despite of the overhead introduced by read/write locks, the AsyncBC performance is similar to the AsyncLF algorithm, with the last being slightly better. We observe that both AsyncBC and AsyncLF algorithms are able to scale in terms of workload and number of threads. Furthermore, both algorithms scale almost logarithmically with 32 and 64 agents as the number of threads increases, suggesting that better results would be achieved on a machine with more cores and using more threads.

The AsyncBM algorithm is always worse than the other asynchronous models and sometimes even worse than SyncAS, revealing that blocking the entire

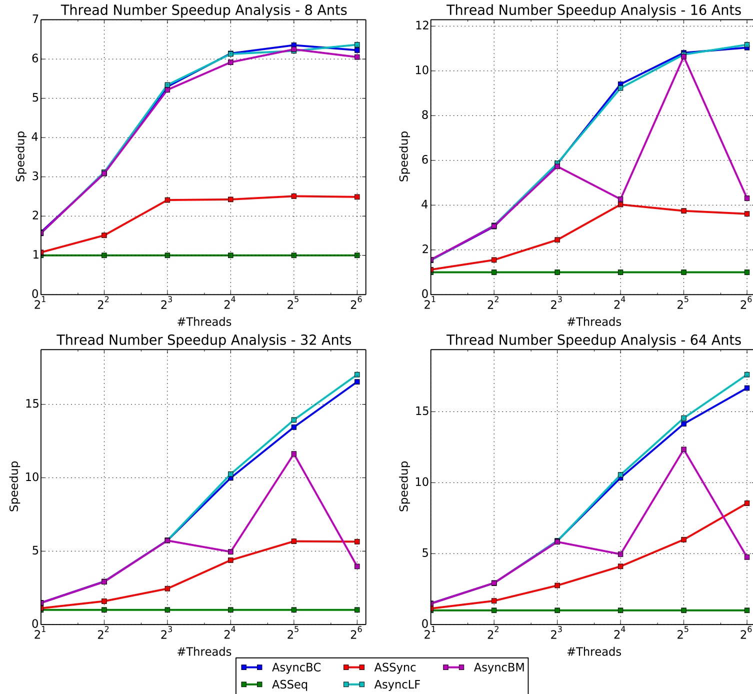

Fig. 3: Speedup Results with 1000 iterations.

matrix has a significant negative impact on the performance. The maximum speedup achieved was ~ 17.6x with 64 threads and 64 agents on a 64 core machine, what gives an efficiency of ~ 0.364. Even for the best configuration (64 agents) efficiency decreases as the number of threads increases. A strict comparison with other proposed algorithms is not possible to stablish since no ACO parallel algorithm was developed and applied to the MD-VSPLE.

It is worth noting that in our first implementation we were only able to achieve speedups smaller than 5.5x with CPU cores usage being very low. We performed some experiments and we found out that concurrent memory allocations have a great impact on the performance ($malloc(3)$ contention). In our implementations we attempted to minimize the number of memory allocations on parallel regions, however, the remaining allocations still affect the performance. Still, the improvements were significant.

Our last experiment intends to evaluate the capability of our best algorithm (AsyncLF), using 8 search agents and 8 threads, of performing a moderate/high depth search in a reasonable amount of time on a PC. Table 1 presents the results. We verify that our algorithm is able to achieve more than 4x speedup for moderate/high search depth (500/1000 iterations) on a processor with 4 cores, with Hyperthreading(R), and taking only a few seconds to execute.

Table 1: Results of executing the AsyncLF algorithm on a PC, with 8 search agents and using 8 threads.

#Iterations	Execution Time (s)	Speedup	Best Obj. Function Value (€)
10	2.312	1.47	24154
50	3.542	2.42	23832
100	4.897	3.16	23720
250	9.226	3.91	23653
500	16.456	4.31	23606
1000	30.801	4.55	23540

6 Conclusions and Future Work

In this work we presented a parallel asynchronous ACO algorithm for the MD-VSPLE (taking into account additional real world constraints), based on the AS algorithm and three different concurrency models, mitigating the straggler problem. Additionally, as asynchronism modifies the semantics of the original algorithm, we proposed a set of modifications in order to ensure convergence and effectively explore the search space.

Our experiments show that our proposed asynchronous algorithms not only are able to converge but also achieved better solutions than ASSync. The AsyncBC and AsyncLF algorithms outperform ASSync in terms of speedups achieved and scalability, with the AsyncLF algorithm yielding the best results.

Experiments performed on a PC shown that AsyncLF is able to perform a moderate/high depth search in a reasonable amount of time.

The next step in order to improve our algorithms performance is to use a different *malloc* implementation (e.g. *TCMalloc* or *jemalloc*) instead of the *glib* implementation. Additionally we also intend to perform additional tests in order to identify other aspects that affect negatively the performance. It is also our intention to extend the pheromone trails model and instead of having a row for each location, we want to have the set of all possible vehicles. This model significantly increases the pheromone matrix dimensions and, the computations performed by each search agent become very expensive. We aim to add a second level of parallelism by using GPUs to compute the ACO probability distribution values for each vehicle, at each decision point.

Acknowledgements
This research was partly supported by project "RtP - Restrict to Plan", funded by FEDER (*Fundo Europeu de Desenvolvimento Regional*), through programme COMPETE - POFC (*Operacional Factores de Competitividade*) with reference 34091.

References

[1] Bertossi, A.A., Carraresi, P., Gallo, G.: On some matching problems arising in vehicle scheduling models. Networks 17(3), 271–281 (1987)

[2] Cipar, J., Ho, Q., Kim, J.K., Lee, S., Ganger, G.R., Gibson, G., Keeton, K., Xing, E.: Solving the straggler problem with bounded staleness. In: Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems. pp. 22–22. HotOS'13, USENIX Association, Berkeley, CA, USA (2013)

[3] Cipar, J., Ho, Q., Kim, J.K., Lee, S., Ganger, G.R., Gibson, G., Keeton, K., Xing, E.P.: Solving the straggler problem with bounded staleness. In: HotOS'13. pp. –1–1 (2013)

[4] Delisle, P., Krajecki, M., Gravel, M., Gagné, C.: Parallel implementation of an ant colony optimization metaheuristic with OpenMP. In: International Conference on Parallel Architectures and Compilation Techniques (2001)

[5] Delisle, P., Gravel, M., Krajecki, M., Gagné, C., Price, W.L.: Comparing parallelization of an aco: Message passing vs. shared memory. In: Proceedings of the Second International Conference on Hybrid Metaheuristics. pp. 1–11. HM'05, Springer-Verlag, Berlin, Heidelberg (2005)

[6] Delévacq, A., Delisle, P., Gravel, M., Krajecki, M.: Parallel ant colony optimization on graphics processing units. Journal of Parallel and Distributed Computing 73(1), 52 – 61 (2013)

[7] Dorigo, M., Gambardella, L.M.: Ant colony system: A cooperative learning approach to the traveling salesman problem. Trans. Evol. Comp 1(1), 53–66 (Apr 1997)

[8] Dorigo, M., Maniezzo, V., Colorni, A.: Ant system: Optimization by a colony of cooperating agents. Trans. Sys. Man Cyber. Part B 26(1), 29–41 (Feb 1996)

[9] Dorigo, M.: Optimization, Learning and Natural Algorithms. Ph.D. thesis, Politecnico di Milano, Italy (1992)

[10] Gendreau, M., Potvin, J.Y.: Handbook of Metaheuristics. Springer Publishing Company, Incorporated, 2nd edn. (2010)

[11] Kotsis, G., Bullnheimer, B., Strauss, C.: Parallelization strategies for the ant system. Technical report (October 1997)

[12] Lenstra, J.K., Kan, A.H.G.R.: Complexity of vehicle routing and scheduling problems. Networks 11(2), 221–227 (1981)

[13] Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. 9(1), 21–65 (Feb 1991)

[14] Mohan, B.C., Baskaran, R.: A survey: Ant colony optimization based recent research and implementation on several engineering domain. Expert Systems with Applications 39(4), 4618 – 4627 (2012)

[15] Pedemonte, M., Nesmachnow, S., Cancela, H.: A survey on parallel ant colony optimization. Applied Soft Computing 11(8), 5181 – 5197 (2011)

[16] Stützle, T., Hoos, H.H.: Max-min ant system. Future Gener. Comput. Syst. 16(9), 889–914 (Jun 2000)

[17] Tsutsui, S., Fujimoto, N.: Parallel ant colony optimization algorithm on a multi-core processor. In: Proceedings of the 7th International Conference on Swarm Intelligence. pp. 488–495. ANTS'10, Springer-Verlag, Berlin, Heidelberg (2010)