Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

An efficient ant colony optimization framework for HPC environments

Patricia González[a,*], Roberto R. Osorio[a], Xoan C. Pardo[a], Julio R. Banga[b], Ramón Doallo[a]

[a] *CITIC, Computer Architecture Group, Universidade da Coruña, Spain*
[b] *BioProcess Engineering Group. IIM-CSIC, Spanish National Research Council, Spain*

ARTICLE INFO

Article history:
Received 29 May 2021
Received in revised form 28 September 2021
Accepted 1 November 2021
Available online 18 November 2021

Keywords:
Combinatorial optimization
Metaheuristics
Ant Colony Optimization
High performance computing
MPI
OpenMP

ABSTRACT

Combinatorial optimization problems arise in many disciplines, both in the basic sciences and in applied fields such as engineering and economics. One of the most popular combinatorial optimization methods is the Ant Colony Optimization (ACO) metaheuristic. Its parallel nature makes it especially attractive for implementation and execution in High Performance Computing (HPC) environments. Here we present a novel parallel ACO strategy making use of efficient asynchronous decentralized cooperative mechanisms. This strategy seeks to fulfill two objectives: (i) acceleration of the computations by performing the ants' solution construction in parallel; (ii) convergence improvement through the stimulation of the diversification in the search and the cooperation between different colonies. The two main features of the proposal, decentralization and desynchronization, enable a more effective and efficient response in environments where resources are highly coupled. Examples of such infrastructures include both traditional HPC clusters, and also new distributed environments, such as cloud infrastructures, or even local computer networks. The proposal has been evaluated using the popular Traveling Salesman Problem (TSP), as a well-known NP-hard problem widely used in the literature to test combinatorial optimization methods. An exhaustive evaluation has been carried out using three medium and large size instances from the TSPLIB library, and the experiments show encouraging results with superlinear speedups compared to the sequential algorithm (e.g. speedups of 18 with 16 cores), and a very good scalability (experiments were performed with up to 384 cores improving execution time even at that scale).

1. Introduction

Combinatorial optimization studies problems where the decision variables are discrete [1]. In the last decades, many research efforts have been carried out to develop new algorithms capable of outperforming existing ones, whose computational requirements to solve complex real-world problems are frequently prohibitive. Metaheuristic methods inspired by nature have arisen as robust tools to solve NP-hard optimization problems, exploiting their ability to calculate precise solutions in reasonable execution times [2].

The Ant Colony Optimization method (ACO) is a metaheuristic which, inspired by the social behavior of ant colonies, has been particularly successful. ACO applies concepts such as collaboration, adaptation, and self-organization, inspired by those displayed by ant communities, in order to efficiently solve challenging optimization problems [3,4]. The pheromone-based communication of biological ants is the basis of the ACO algorithm. While exploring the environment, real ants deposit pheromones that guide each other to resources. Similarly, artificial ants calculate the quality of their solutions and update a pheromone matrix that is used in subsequent iterations to find better solutions. For the most part, the procedures followed by each artificial ant are independent of other ants, but are guided by prior information on the success of the entire colony.

Given the parallel nature of the ACO metaheuristic and its variants, it is not surprising that many parallel proposals can be found in the literature [5] with the aim of solving large instances of difficult problems in competitive times. However, most of those proposals focus on a single problem instead of showing a methodology that can be applied to other problems. In addition, most of the proposals are based on a single programming model, using either a (1) fine-grained parallelization approach, usually shared memory models where the loops that present independent iterations can be carried out in parallel, or proposals that use accelerator devices such as GPUs to process those parallel loops; or a (2) coarse-grained parallelization approach, where ants are divided into independent colonies that may or may not collaborate with each other. Furthermore, those fine-grained and coarse-grained strategies are frequently based on synchronous communications, which take place at certain time intervals or when a certain number of iterations have been completed. This

* Corresponding author.
E-mail address: patricia.gonzalez@udc.es (P. González).

https://doi.org/10.1016/j.asoc.2021.108058

fact limits their scalability, that is, its effectiveness and efficiency when the number of computational resources increases.

In this work we propose a novel hybrid parallel ACO algorithm, introducing parallelism at different levels with the aim of improving scalability. On the one hand, a coarse-grained strategy is proposed with multiple colonies that execute the ACO algorithm in isolation but collaborate with each other by exchanging information. On the other hand, a fine-grained strategy is proposed within each colony that allows the most expensive loops to be executed in parallel. The combination of both strategies allows better use of computational resources, improving the scalability of single proposals. The main overall features of the new parallel approach proposed are:

- a *hybrid approach* using both a multicolony parallelization to increase diversity in the search by means of different colonies, and a fine-grained parallelization to accelerate the search process in each colony;
- a *fully decentralized approach*, which, compared to the more popular master–slave approaches, enables optimal use of available resources;
- an *asynchronous communication protocol* to exchange information between colonies, avoiding idle processes waiting for information exchanged from other colonies;
- *information exchange driven by quality* of the solution obtained in each colony, rather than by a elapsed time, resulting in more efficient cooperation between colonies.

The proposal is not only especially well suited for taking full advantage of traditional HPC infrastructures, but is also applicable for other distributed environments, such as cloud infrastructures or local computer networks, in which a decentralized and desynchronized approach allows to minimize the impact of unrealiable communications or high latencies.

Two of the benefits of the proposed parallel solution are especially relevant in solving large instances of difficult problems in reasonable times: (1) a more efficient use of resources, since there is no central master consuming resources, and there are no idle times in the processes waiting for communications between colonies or with the master; and (2) a fault-tolerant solution, since a failure in a colony will not interrupt the work of the rest, which can continue and finish the execution. Moreover, there is no master that represents a single point of failure (SPOF) in the execution.

Efficient resource utilization has a beneficial impact not only on the application performance in terms of execution time and consumed resources but also saves money [6], which is especially relevant in the increasingly frequent scenario, both in academia and industry, of provisioning HPC resources from public cloud providers under an on-demand, pay-per-use service model. Fault tolerance is not a feature that most algorithms take into account, generally assuming that it will be provided by the infrastructure. However, it is increasingly important for parallel applications, not only because the failure rates of massively parallel systems, especially in the exascale era, could prevent the completion of long executions [7,8], but also because of the high cost of launching and executing again failed experiments, especially for the computing facility. In the Blue Water Cray Supercomputer, the electricity cost of not using any fault tolerance mechanism in applications over a period of 261 days was estimated to be almost half a million dollars [9]. Thus, the impact of failures on applications may be significant and having an algorithm that completes its execution even in the presence of failures is valuable.

For the experimental assessment of the proposal, the popular Traveling Salesman Problem (TSP) has been used. TSP is a paradigmatic NP-hard combinatorial optimization problem. It plays a key role in ACO proposals because it was used to test the first proposal of an ACO algorithm and it has been used as test problem for most of the ACO algorithms proposed later. TSP was chosen in this work for two main reasons. First, because it is a widely-used class of hard problems that has become a standard testbed for new algorithmic ideas. Second, because it is easy to understand, so the behavior of the algorithm is not eclipsed by other technicalities unrelated to the proposed parallelization schemes. However, the techniques described here can be easily applied to other classes of problems. Further, our framework can be used as a template to guide other researchers interested in further parallel extensions of the ACO metaheuristic.

The organization of this paper is as follows. Section 2 illustrates the significance of the proposal by means of a review of recent applications in real-world case studies that would benefit from the contributions of this work. Section 3 describes the most popular approaches to parallelizing ACO metaheuristics. A brief overview of the ACO metaheuristic for solving TSP problems is presented in Section 4. Section 5 describes the hybrid MPI+OpenMP parallelization of the ACO algorithm proposed in this paper. The performance of the proposed method is evaluated in Section 6. Finally, Section 7 summarizes the main conclusions of this work.

2. Applicability of the contribution

Since its proposal thirty years ago, ACO has been used in a large number of applications. Most of them are NP-hard problems, for which computing time grows exponentially with instance size. Therefore, parallel solutions are particularly appealing for these problems in large scale domains. A recent review both of the advances that have appeared over the years in the original algorithm, and of the most notable applications, can be found in [10]. Despite the relevance of the works mentioned there, all of them were published before 2010.

To further illustrate the significance of the work presented here, we add in this section a selected review of recent papers that make use of ACO. This review focuses on works that use real-world applications as testbeds. We find ACO applied to a wide variety of problems. For instance, vehicle routing problems to solve important logistic activities within any city, such as the collection of waste. Several works have recently tackled this problem using ACO [16,23]. Robot path planning is another application of ACO that can also be found in recent papers [24, 25]. Another important category is that of scheduling problems, solved using ACO in a wide range of real-world applications, such as the placement of Virtual Machines (VMs) in data centers [26, 27], or supply chains logistics [15,20], among others. ACO has also been successfully applied in different challenging problems that arise in bioinformatics, computational biology and computational chemistry applications. Among them, we highlight, due to its significance, drug discovery [13,28]. In a recent overview of the evolution of structure-based drug discovery techniques on ligand–protein interactions [29], the use of HPC techniques is emphasized as a mean to perform simulations on a scale that otherwise would be infeasible. This list is not intended to be exhaustive, but it is representative of a wide range of real-world applications of the ACO algorithm that would profit from using the parallel techniques proposed in this paper.

Papers that describe a parallel implementation of the ACO algorithm and test it using real-world applications are summarized in Table 1. Examples of the different types of parallelization described in Section 5 and discussed throughout this paper have been selected. The description of the parallelization, the testbed used, and the most outstanding results reported, are briefed in the table. Also, since combinations of artificial ants and local search algorithms have become a method of choice for many approaches,

Table 1
Selected recent publications that test parallel implementations of ACO with real-world applications.

Problem and application	Reference	Year	LS	Parallelization	Testbed	Results
TSP applied to manufacturing time for a CNC	Montiel et al. [11]	2012	✔	Master–slave approach, master in charge of distribution of work and updating of pheromones	Visual C#, run on Intel Core i7 CPU 930 (2.80 GHz 6GB)	A 4x speedup with 6 slaves. Improvements in the cutting time of around 62% vs. a commercial CAD/CAM software solution
Feature selection applied to text categorization	Meena et al. [12]	2012	✔	Mapreduce master–worker approach to distribute the ants' solution construction	Java and Hadoop, run on 6 nodes Intel Core2Duo processor, 2 GB RAM	Parallel ACO reduces time from 90 min to 15 min
Docking problem applied to computer aided drug design	Zaidman et al. [13]	2016	✗	Single colony of 35 ants performs each round in parallel with a synchronization between them	DELL PowerEdge 2.8GHZ computer with 20 cores and 256 GB	Takes up to 24h to design a peptide of 5-15 amino acids length
Permutation flow-shop scheduling problem subject to limited machine availability	Huo et al. [14]	2016	✔	Multicolony with synchronization after a certain number of iterations	C++ with MPI, run in a SGI cluster with up to 16 nodes	Maximum execution time allowed for sequential problems was 2400 s, 15.73x speedup achieved
Job scheduling problem applied to mining supply chains	Thiruvady et al. [15]	2016	✔	Shared-memory with threads performing *n* whole generations of multiple ants in parallel	C++ with OpenMP, run on a 4 hex-core Intel Xeon processor: 1.87 GHz and 24MB	Parallel ACO reduces time from 1h to 7 min
Vehicle routing problem applied to waste collection	Grakova et al. [16]	2018	✔	Fine-grained parallelization using one single colony with 24 to 196 ants using from 2 to 24 threads	C# and .NET framework, run on one HPC node: 2.5 GHz, 128 GB, 24 cores	Execution time not reported. Speedups up to 6.89 with 24 threads and 196 ants
Training neural networks for predictions of turbine engine vibration	ElSaid et al. [17]	2018	✗	Master–slave with asynchronous communications. Master in charge of the pheromone updating	MPI with Python, run on a HPC cluster with 31 nodes of 8 cores and 64GBs RAM per node	Each run for 200 ants and 1000 iterations using 208 cores took approximately 4 days
Vehicle routing problem applied to a bike sharing system	Fan et al. [18]	2019	✔	Multicolony approach with synchronization on every iteration to update the pheromone matrix	R and Mahout, run on an Intel Core i7-7500 CPU, 2.70 GHz, 8 GB	Sequential execution of small problems lasts 300 s, a 3x speedup is achieved with the multicolony
Clustering problem applied to home care scheduling	Martin et al. [19]	2020	✗	Multicolony with 64 ants per colony synchronized to update the pheromone matrix	Google Cloud Platform with Intel Xeon with 64 cores and 60 Gb of RAM memory	For the largest problem tested and 64 cores, execution times exceed 2000 s
Scheduling problem for supply chains in a microchip producer	Dzalbs et al. [20]	2020	✗	Two shared-memory parallel algorithms: (1) IAC, colonies distributed to threads, (2) PA, ants distributed to threads	C++ and OpenMP, run on an AMD Ryzen Threadripper 1950X (16 cores, 64GB), running at 3.85 GHz	PA outperforms IAC, a 25.4x speedup was achieved with 32 threads
Scheduling problem applied to cloud virtual resources	Baniata et al. [21]	2021	✗	Multicolony approach with synchronization to update the pheromone matrix	Google Cloud Platform using a C2-standard-8 instance (8 vCPUs, 3.8 GHz, 32 GB memory)	Time to allocate 3000 tasks into 30 virtual resources exceeds 27 h using 5 cores
Clustering problem applied to fault diagnosis of rolling bearing	Wan et al. [22]	2021	✗	Mapreduce master–worker approach to distribute the ants' solution construction	Scala and Spark, run on a cluster with 1 master node Intel Xeon E3-1225 v5 and 8 workers Intel Core i7-9700k	Improvement in fault diagnosis accuracy using ACO: 5%. Parallel ACO reduces execution time from 592 min to 83 min

a column to indicate whether or not a local search is used in the proposal is also included, labeled as LS.

Several of the works in Table 1 use fine-grained parallelization [13,16,20], using a single colony and distributing the ants' solution construction between different threads. In all these cases, the scalability of the proposal is limited by the overhead to launch new threads and synchronize them at the end of the tasks. Other works use the multicolony approach [14,18–21], but all of them use some kind of synchronization. In [18] the colonies are synchronized on every iteration, so their performance is poorer. In [14,21], synchronization is less frequent, which improves performance, however, during synchronization, the global pheromone matrix is updated, which implies an overhead in the communications. Master–slave approaches are proposed in [11,12,17,22]. Most of them use synchronous communication protocols, such as those that are based on MapReduce [12, 22]. In [17] asynchronous communications between the slaves and the master are performed, however, the master is in charge of updating the pheromone matrix and of distributing new tasks to the slaves, which implicitly implies a synchronization between them. A decentralized and asynchronous scheme, like the one proposed in this work, would improve the performance of all these proposals.

Many of the papers referred above do not use high-performance programming libraries or infrastructures, including some of the works in Table 1. As the need for faster and more robust software exists, we think that most researchers do not resort to parallel codes due to the difficulties in using HPC libraries and tools, and in accessing suitable infrastructures. However, from the 12 references in the table, two of them, [19,21], make use of the Google Cloud Platform. Both are very recent papers, showing that this is an upward trend that will promote access to HPC infrastructures for many researchers. We believe that the framework presented here, together with the discussions about the different parallel strategies, its experimental assessment, and the possibility of using the available code as a template, can help future researchers interested in using and developing parallel solvers for hard combinatorial optimization problems.

3. Parallel ACO metaheuristics

Since most of the metaheuristics are conceived to face with large and complex problems, their parallelization has attracted a lot of attention in recent decades. The ACO metaheuristic has been one of those algorithms, favored by its intrinsically parallel nature.

Many works proposing parallel implementations of the ACO metaheuristic can be found in the literature. To organize them in this section we will follow the taxonomy proposed in [5], that categorizes the parallel implementation of ACO in:

- *Master–slave model*: with proposals that use a master process to manage the global information (pheromone matrix, best-so-far solution, etc.) and control the group of slaves that carry out the parallel tasks. This category includes three different models regarding the granularity of the parallel tasks: *coarse-grained*, *medium-grained* and *fine-grained* models.
- *Cellular model*: where a single colony is structured in small neighborhoods placed in a toroidal grid. Each *cell* has a local pheromone matrix and communicates only with its close neighbors using a *diffusion* model.
- *Parallel independent runs model*: where several sequential ACOs concurrently run on a set of processes in a completely independent way, without communications between them.
- *Multicolony model*: where several colonies explore the search space isolately but including cooperation steps where information is exchanged among colonies.
- *Hybrid models*: with proposals that feature characteristics from several of the previous parallel models.

The vast majority of the proposals are within the master–slave model classification. Most of these proposals use a coarse-grained pattern [30–34]. They are usually based on the distribution of the ants of a single colony among the available processors (slaves). Each slave performs a cycle or several cycles of the basic algorithm updating a local pheromone matrix, and with a certain frequency (elapsed time or number of iterations) a synchronization is carried out through the master, that manages the global information, such as the global pheromone trials. However, some solutions that implement a master–slave coarse-grained model have also tested partial asynchronous implementations [35–37], demonstrating their superiority over synchronous ones that involve waits on the slaves that slow down the algorithm. Recently, models for CPU-based SIMD architecture have been also investigated in [38]. This work demonstrates that simple vectorization strategy does not fit well on SIMD CPUs. Thus, in order to fully exploit vector-level parallelism, a variant of the algorithm based on vectors is proposed showing competitive results.

Few solutions follow a medium-grained or a fine-grained master–slave model, probably due to the fact that this model is not conceptually simple and is more difficult to implement than coarse-grained ones. The medium-grained model follows a *divide-and-conquer* approach, based on the decomposition of the problem into pieces that are distributed among the slaves that search for partial solutions for each piece and send them to the master who builds the complete solution. Some examples of medium-grained master–slave proposals are [39,40].

The fine-grained model distributes small tasks among the slaves while the master is in charge of carrying out the update of the global information and control the progress of the algorithm. The first solutions of this type showed very poor performance results due to the high frequency of communications [41]. However, more recently this model has been used in shared memory implementations [31,42,43] where critical regions are used to avoid the continuous updating of shared information and restricting it to certain and not so frequent synchronization stages, showing encouraging results.

The cellular model, which is a popular method for parallel evolutionary algorithms, has hardly any presence among the ACO parallel proposals. The best known works are those of Pedemonte and Cancela [44] that show good results in terms of computational efficiency, but a degradation of the quality of the solution compared to the sequential implementation.

The parallel independent runs model follows an embarrassingly parallel strategy, where each processor executes a sequential ACO within a colony of ants, with no communication between processes/colonies. Comparing this model with other multicolony approaches, it has been found that it shows up as a competitive approach, because although generally the models that involve communications between the colonies find better solutions, the embarrassingly parallel models may sometimes achieve better efficiency [45–48].

One of the most widely adopted model of parallel ACO implementation has been the multicolony model. This model is based on executing sequential ACO algorithms in different processors/colonies but incorporating some mechanism of cooperation between them, which normally obtain better results than the sequential algorithm. The differences between the various proposals focus in features such as the frequency with which communications are carried out between the colonies, which is usually a function of the elapsed time; the topology used for communications, such as star, hypercube, unidirectional ring, etc. (specially applied to traditional HPC supercomputers); and the information exchanged, which can be the pheromone matrix or solely the best solutions found.

Michel and Middendorf [49,50] were the first to introduce a multicolony model that has afterwards inspired many other authors [36,51–53]. One of the limitations of this model is the use of synchronous communications that limits the scalability of the proposal. Variants that use asynchronous communications and/or adjust the frequency of information exchanging can also be found [54–56]. All of these studies confirm that a trade-off between exploration within each colony and cooperation through information exchanging is desirable to achieve accurate results and performance.

Finally, there have also been a few proposals for parallel hybrid ACOs, which combine features of more than one of the previous parallel models. Some examples in this category are [42, 57–59].

Both the works referenced above, and the one proposed in this paper, are focused on traditional parallel computing environments, such as multicore multinode computers. In recent years, new types of hardware offering large amounts of parallel processing power are available. Among them, the graphics processing units (GPU) are the ones that attract most attention, also in the field of ACO metaheuristics [60–64]. These proposals can also be framed in one of the previous classifications. More recently, proposals that use new distributed programming paradigms, such as MapReduce or Spark, are also arising [65–68]. However, there is still room for new proposals in this topic.

Algorithm 1: Pseudocode of the Ant Colony Optimization metaheuristic

```
1 Set parameters
2 Initialize pheromone trails
3 while termination condition not met do
4     ConstructAntsSolutions
5     LocalSearch
6     UpdatePheromones
```

The implementation we propose in this work is initially designed to maximize efficiency in traditional HPC infrastructures such as multicore clusters. However, as it features decentralization and asynchronous communication protocol, this proposal is also suitable for cloud environments or local computer networks, where communications between nodes may not be reliable, or latency be too high.

4. Ant Colony Optimization for the TSP problem

Ant Colony Optimization (ACO) is a metaheuristic in which a *colony of ants* cooperates in finding good solutions to difficult optimization problems. An artificial ant in ACO is a stochastic procedure that gradually builds a solution by adding appropriate elements to the solution under construction. Thus, the ACO metaheuristic can be applied to combinatorial optimization problems for which a constructive heuristic can be defined.

The Traveling Salesman Problem (TSP) is one of such problems, that consists in finding the shortest possible round trip through a given set of cities. It can be represented by a graph $G = (N, A)$ with N the set of nodes/cities and A the set of arcs fully connecting the nodes. A weight d_{ij} is assigned to each arc $(i, j) \in A$, representing the distance between cities i and j. The TSP problem tries to find a minimum length Hamiltonian circuit of the graph, where a Hamiltonian circuit is a closed tour visiting each node of the graph only once.

Algorithm 1 shows the basic pseudocode of most ACO algorithms, that consists of three main procedures: `ConstructAntsSolutions`, `LocalSearch`, and `UpdatePheromones`. `ConstructAntsSolutions` manages a colony of artificial ants that incrementally build solutions to the optimization problem by means of stochastic local decisions based on pheromone trails and heuristic information. Then, a `LocalSearch` procedure improves the ants' tours with some optional local search algorithm. And finally, the `UpdatePheromone` procedure modifies the pheromone trails based both on the evaluation of the new solutions and on a pheromone evaporation mechanism.

A large number of extensions of this basic ACO algorithm can be found in the literature. They can be classified into two large groups. A first group consists of proposals that maintain the same solution construction procedure, and introduce the differences with the basic algorithm in the management of pheromone trails and the way in which the pheromone update is carried out. A second group comprises the proposals that modify the way of construction of the solutions, as well as those that more deeply modify the structure of the basic algorithm or its features. The parallel proposal presented in this work is valid and has been tested with algorithms from the first group, although the general ideas expressed here could easily be extended to algorithms from the second group as well.

In the rest of this section the three basic procedures of the ACO metaheuristic, used as a basis in the parallel implementation proposed in this work, are described in depth for its use in solving the TSP problem.

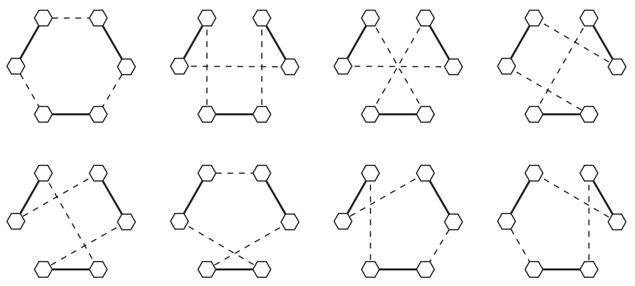

Fig. 1. Applying 3-opt local search: reconnection possibilities in a tour deleting 3 edges.

4.1. Ants' solutions

Each individual ant in the colony is randomly assigned to a city from where it travels to each new city until all cities are visited. By returning to the starting city it forms a Hamiltonian circuit. To decide the new city to visit, the artificial ant k applies the following probabilistic transition rule that depends both on pheromone and heuristic values:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{u \in N_i^k} [\tau_{iu}]^\alpha [\eta_{iu}]^\beta} \tag{1}$$

where, N_i^k is the set of cities not visited yet by the ant k, τ_{ij} represents the desirability of visiting city j just after city i, given by the pheromone trails, and η_{ij} is the heuristic information. When solving TSP problems, η_{ij} is usually set to the inverse of the distance d_{ij} between the cities. Parameters α and β are key in the solution construction. Parameter α sets the amount of pheromone on the edges that would vanish in every iteration. Parameter β sets the relative significance of pheromone versus heuristic value.

4.2. Local search

The ACO basic pseudocode includes the possibility of applying a local search routine, once ants have completed their solution construction. The coupling of solution construction with local search achieves a good trade-off between exploration and exploitation, which is at the core of modern metaheuristics [69].

Among the most popular local search procedures coupled with ACO algorithms, the one used in the implementation of the parallel proposal presented here was the so-called *3-opt*. Basically, the *3-opt* local search deletes three edges in a tour, breaking it into three paths, and then reconnects these paths in the other possible ways to select the best one. The *3-opt* goes on applying the procedure until improvements cannot be found and the tour is 3-optimal. Fig. 1 shows the 8 different reconnection possibilities deleting 3 edges.

4.3. Update pheromones

In this procedure the pheromone trails are modified, increasing their values when ants deposit pheromone on promising tours to guide other ants in constructing new solutions, or decreasing their values due to pheromone evaporation to avoid unlimited accumulation of pheromone trails and also to allow bad choices to be forgotten.

The evaporation process prevents the algorithm from premature convergence to suboptimal regions and from getting stuck in a local optimum, by decreasing τ by a constant rate ρ (the pheromone evaporation rate):

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} \tag{2}$$

Then, ants deposit pheromone on the arcs they have crossed in their tour:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{3}$$

where m is the number of ants and $\Delta\tau_{ij}^{k}$ is the amount of pheromone ant k deposits on the arcs it has visited, defined as:

$$\Delta\tau_{ij}^{k} = \begin{cases} 1/C^{k}, & \text{if } arc(i,j) \text{ belongs to } T^{k} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where C^{k} is the length of the tour T^{k} constructed by ant k.

As shown in the probabilistic transition rule, the possibility for an ant to visit a city j just after i increases with the pheromone trail. Thus, it is in the implementation of this procedure that many of the variants of the ACO algorithm differ.

5. Hybrid parallelization of the ACO metaheuristic

The parallelization of the ACO metaheuristic aims to achieve the following objectives [70,71]: (1) speed up the calculations, (2) increase the size of the problems that can be solved, and/or (3) attempt a deeper exploration of the solution space. Thus, the coupling of a multicolony proposal and a fine-grained parallelization in each single colony shows up as an appealing solution, as it can help to address these objectives simultaneously.

In addition, today's HPC systems include groups of multicore nodes that can benefit from the use of a hybrid programming model, in which a message passing model, such as MPI (Message Passing Interface), is used for inter-node communication while a shared memory programming model, such as OpenMP, is used intra-node. Though hybrid programming requires an effort from application developers, this model offers several advantages such as reducing the communication needs and memory consumption, as well as improving load balancing and numerical convergence [72].

Parallel strategies have to focus both on the efficient use of computational resources, leading towards scalable solutions, and on fault tolerance, which allows the completion of the task despite the presence of failures in part of the resources. In this paper we present a decentralized and desynchronized hybrid MPI + OpenMP parallel implementation of the ACO metaheuristic designed to address these goals.

The implementation of the sequential ACO upon which we have built our parallel solution is the one provided by Thomas Stützle in http://www.aco-metaheuristic.org/aco-code. This framework has been selected because it provides an implementation of several ACO algorithms under a single high-performance skeleton that allows us to develop a general proposal that we can easily reuse and test with different variants.

In the following subsections we describe in detail the implementation of the parallel proposal, both at the multicolony (MPI) level and at the fine-grained (OpenMP) level in each colony.

5.1. Multicolony parallelization

The multicolony parallelization proposed is based on the cooperation between parallel processes. When designing cooperative parallel metaheuristics some key issues have to be addressed, such as what information is exchanged, which processes are involved in the communication process, when and how the information exchange takes place, and how the information received from other colonies is used. The solution to these issues may impact significantly the efficiency and efficacy of the parallel proposal. For instance, too many synchronizations between processes can lead to too many idle times and poorly scalable solutions. A low frequency in communications can lead to poor collaboration schemes between colonies and a low efficiency of the parallel approach, while a too high frequency could lead to early convergence towards regions with local minima.

Algorithm 2 shows a basic pseudocode for the proposed multicolony parallelization. Regarding what information should be exchanged, most of the parallel ACO proposals that can be found in the literature follow one or both of the following approaches: (1) exchange information about the pheromone matrix (usually the entire matrix), and/or (2) exchange the best routes found in the different colonies. This information is used to update the local pheromone matrix. Based on the experience during the tests carried out in this work, exchanging the complete pheromone matrix produces encouraging results for small problems. This happens both in quality and in execution time, since the close cooperation between colonies accelerates the convergence to the optimal solution and, at the same time, the size of the matrix is not excessively large to make the cost of communications skyrocket. However, for large problems, the convergence results demonstrate that the exchange and combination of the pheromone matrices between colonies results in a loss of diversity that frequently leads to premature convergence and stagnation at local minima. Furthermore, the communications when the pheromone matrix is very large affect the efficiency of the parallel code, and there is also the problem of deciding the frequency of these communications. Therefore, in the implementation presented in this work, the exchange of information among cooperating colonies is driven by the quality of the solutions obtained. New best tours found so far in each colony are spread to the rest of the colonies. Each colony has a `promising-tour` communication buffer to asynchronously receive tours from the rest of the colonies, as well as a `global-best-tour-so-far` to track the best tour found so far among all the colonies. A new `promising-tour` substitutes the `global-best-tour-so-far` when the former outperforms the latter. Restricting the exchange of information to a single global best tour allows each colony to evolve on its own pheromone matrix while also benefiting from promising solutions from other colonies as they arrive.

The communication protocol used in our approach is designed to prevent processes from blocking waiting for messages that have not arrived during a cycle of the ACO algorithm, allowing the execution to progress in the colonies, whether or not they have received new information. Both the emission and the reception of the messages are performed using non-blocking asynchronous operations (`MPI_ISend()`, `MPI_IRecv()`, `MPI_Test()`), allowing the overlap of communications and computations and avoiding synchronization steps that harm the efficiency of the parallel approach.

5.2. Fine-grained parallelization in each colony

The most time-consuming task in the basic ACO algorithm described in Section 4 is usually the `LocalSearch` procedure. Among the ACO algorithms implemented in the framework, we report in Section 6 results for Elitist Ant System (EAS). In these tests 70% of the execution time to solve the `pr2392.tsp` instance from TSPLIB [73] is spent in the `LocalSearch` procedure, followed by the `ConstructAntsSolutions` procedure (21% of the execution time). The `UpdatePheromone` procedure represents less than 10% of the execution time in all the experiments reported in this work. Fortunately, both the `LocalSearch` procedure and the `ConstructAntsSolutions` can be straightforward parallelized since they consist of an iterative loop in which every iteration is independent from the rest. The procedure to update pheromone trails, on the contrary, has dependencies between different iterations of its loop, since all the ants contribute to a

Algorithm 2: Multicolony parallelization proposed

```
   // Initialize MPI environment
1  MPI_Init
2  Initialize colony parameters
3  Initialize colony pheromone trails
   // Prepare a reception buffer for asynchronous
      communications
4  MPI_IRecv(promising-tour,request)
5  while termination condition not met do
6      ConstructAntsSolutions
7      LocalSearch
       // When new local best tour is found, send it
          asynchronously
8      if local-best-tour-so-far < global-best-tour-so-far then
9          MPI_ISend(local-best-tour-so-far)
10     UpdatePheromones
       // Every cycle check the reception of foreign promising
          tours
11     repeat
           // MPI check asynchronous reception
12         MPI_Test(request, recvflag)
13         if recvflag then
14             if promising-tour < global-best-tour-so-far then
15                 global-best-tour-so-far = promising-tour
16                 UpdatePheromone_WithBestTour(global-
                    best-tour-so-far)
               // Prepare a new reception buffer for next
                  receptions
17             MPI_IRecv(promising-tour,request)
18     until recvflag
```

global shared pheromone matrix. Thus, the `UpdatePheromone` procedure remains sequential in the fine-grained parallelization proposed for each colony.

The `ConstructAntsSolutions` procedure consists of the following phases: (1) the ants' memory is emptied, by making all cities unvisited; (2) the first city of each tour is selected randomly; (3) each ant constructs a complete tour applying the ACO decision rule explained in Section 4; and (4) the ants move back to the initial city and the tour length is computed. Each of these phases can be performed in parallel for each ant, because they do not update the pheromone trails at construction time, thus, there are no communication between them during the construction of the solution. To increase the performance of the parallelization, all the parallel tasks in the construction of an ant's tour have been grouped in a single parallel loop. Algorithm 3 shows the basic pseudocode. Every time the solution construction procedure is invoked, a parallel loop is defined using the OpenMP library. The execution of the parallel loop is based on the fork-join programming model. In the parallel section, the running thread creates a group of threads, so that the ants in the colony are divided among the threads of the group and each tour construction is performed in parallel. At the end of the parallel loop, the different threads are synchronized and finally joined again into only one thread. Due to this synchronization, load imbalance in the parallel loop can cause significant delays. Though in the basic ACO implementation the time to construct each solution is more or less independent of the ant k, in other extensions of the ACO algorithm this can vary. Thus, a *guided schedule* clause is proposed so that the assignment of iterations to threads is dynamic. Iterations are handed out to threads in runtime, as they complete their previously assigned tasks.

Algorithm 3: Parallel `ConstructAntsSolutions` procedure

```
1  $$ omp parallel (guided schedule)
2  for $k = 1$ to number_of_ants do
3      ClearVisitedCities
4      step = 1
5      SelectRandomlyFirstCity
6      while step < number_of_cities do
7          ApplyACODecisionRuleToSelectNextCity
8          step++
9      CloseCircuitWithFirstCity
10     ComputeTourLength
```

Algorithm 4: Parallel `LocalSearch` procedure

```
1  $$ omp parallel (guided schedule)
2  for $k = 1$ to number_of_ants do
3      3-opt
4      ComputeTourLength
```

Table 2
Benchmarks used in the experiments reported in this section.

Problem instance	Number of Cities	Best Known Solution
`pr2392.tsp`	2392	378032
`rl5934.tsp`	5934	556045
`brd14051.tsp`	14051	469385

There exist in the literature different approaches to apply the local search to the tour solutions found by the colony. In the implementation used in this work, the local search is applied to all the solutions found in each cycle. Therefore, like the procedure for construction, the local search consists of a *for loop* that runs through each ant k, improving its tour solution, without communications with the rest of the colony. The loop, therefore, is directly parallelizable, as shown in Algorithm 4.

6. Experimental evaluation

In this section an exhaustive experimental evaluation of the proposed parallel strategy is presented. The aim of this work is not to study the efficacy of the ACO metaheuristic and its variants, nor the effect of the selection of parameters such as α, β or ρ on the performance of the algorithm. As discussed in Section 5, the parallel proposal presented in this work can be directly applied to several of the ACO algorithms in the literature and it is profitable to the same extent regardless of the ACO parameters used for the search. Therefore, in all the experiments reported in this section, the same variant of the ACO algorithm and the same execution parameters will be used as a basis, in order to make a fair comparison between the different parallel versions that are analyzed. In all cases, the Elistist Ant System (EAS) variant will be used, as well as the default parameters $\alpha = 1.00$, $\beta = 2.00$, $\rho = 0.5$ parameters, and the number of ants $m = 25$.

Benchmarks from the TSPLIB library [73] have been used to carry out the experiments. We report here results for three different popular instances summarized in Table 2. The results reported in this section come from experiments that were performed at the Galicia Supercomputing Center (CESGA) using the *FinisTerrae-II* supercomputer. Each FinisTerrae-II node is composed of two Intel Haswell E5-2680v3 CPUs running at 2.50 GHz, with 12 cores per processor (24 cores per node), and 128 GB of RAM. The nodes are connected using an InfiniBand FDR 56 Gbps interconnect using a fat-tree topology. Due to the stochastic behavior of the algorithm, 100 runs are performed independently for each experiment, and the distribution of the data is taken into account in the evaluation and discussion of the results.

To comprehensively evaluate the performance of parallel proposals, tests are performed from different perspectives. Most of the tests were performed with a quality stopping criterion. An objective value is pre-established and the effort needed to reach the target solution is evaluated. These experiments have, as a handicap, the long execution times required in some cases to complete the runs, especially if we attempt to compare with sequential executions. For this reason, other tests were carried out using a combined stopping criterion, establishing a target value and a predefined effort. The execution finishes when either of the two criteria is met.

6.1. Performance evaluation of the multicolony approach

The cooperation between colonies changes the systemic properties of the sequential algorithm and, therefore, its macroscopic behavior. Thus, in the first set of experiments the fine-grained parallelization was disabled, since it does not alter the convergence properties of the algorithm, in order to evaluate solely the impact of the multicolony approach.

Comparing the results of different parallel strategies proposed in the literature is a hard task because the results reported in the original works are seldom directly comparable. Factors such as the optimization of the sequential algorithm; the parameters settings chosen; the programming language; the infrastructure where the codes run, among others, impact performance and do not allow for a fair comparison. For this reason, in this work we have implemented different popular parallelization strategies from the literature starting from a single sequential code, in order to fairly compare the performance among them. Note again that the aim of this work is not the study of the ACO algorithm or its variants, but the parallelization strategies and their impact. In the following experiments the performance of the ACO algorithm proposed (`asyn-ACO`) is compared with the sequential version of the ACO algorithm (`seq-ACO`) and with two different parallel versions, namely: an embarrassingly-parallel non cooperative ACO (`emb-ACO`) and a synchronous cooperative version (`syn-ACO`). The `emb-ACO` consists of *nc* independent ACO runs (being *nc* the number of available cores) performed in parallel without cooperation between them and reporting the best execution time and/or best value of the *nc* runs. On its turn, the synchronous parallel version consists of exchanging the best solutions among colonies every 100 iterations.

The first experiment carried out uses an objective value as stopping criterion (VTR: *value-to-reach*), thus allowing to study the speedup of the methods to reach the target value. Table 3 shows, for each method, the average of the number of iterations performed in each experiment, the average and standard deviation of execution time (in seconds), and the speedup achieved versus the sequential algorithm (computed as $sp = T_{seq}/T_{par}$). As it can be seen, the proposed `asyn-ACO` outperforms both the embarrasingly parallel method and the synchronous strategy. In addition, it should be noted that the achieved speedups are superlinear, that is, they exceed the number of cores used. This is because, as already mentioned, the multicolony approach changes the properties of the sequential algorithm. Specifically, having different colonies attempts a deeper exploration of the solution space, and with the cooperation between the colonies the algorithm can get out of the local minima more easily.

Although the reported speedup is calculated from the average execution times of the 100 runs carried out per experiment, the large dispersion of the results makes difficult to assess the different strategies attending only to the speedup. To better illustrate the behavior of the different parallel strategies, the results obtained in the experiments in Table 3 are graphically depicted in Fig. 2, where the cumulative probability versus execution time is represented. It can be seen how the parallel versions significantly reduce the execution time as the number of colonies increases. Their curves also exhibit a steeper slope, which graphically demonstrates the robustness of the parallel algorithm compared to the sequential one. In addition, it can also be seen how asynchronous versions outperform synchronous versions in all cases.

One of the recent works that proposes a parallelization of the ACO [37], studies the behavior of the algorithm in the first 200

Table 3
Results using as stopping criterion a value to reach (VTR) of 387 500, and comparing execution times for the `pr2392.tsp` instance problem.

Colonies	Method	Iterations	Time (s)	Speedup
1	seq-ACO	2949	146.36 ± 165.88	–
2	emb-ACO	5701	148.90 ± 150.14	0.98
	syn-ACO	2086	54.37 ± 50.14	2.69
	asyn-ACO	1821	47.45 ± 56.76	3.08
4	emb-ACO	11957	155.89 ± 167.89	0.94
	syn-ACO	1574	20.97 ± 13.30	6.98
	asyn-ACO	1402	18.41 ± 13.97	7.95
8	emb-ACO	19935	141.95 ± 133.90	1.03
	syn-ACO	2339	16.96 ± 10.78	8.63
	asyn-ACO	1909	14.00 ± 12.21	10.45
16	emb-ACO	38707	143.19 ± 133.97	1.02
	syn-ACO	2816	11.16 ± 7.46	13.12
	asyn-ACO	2122	8.25 ± 4.20	17.74

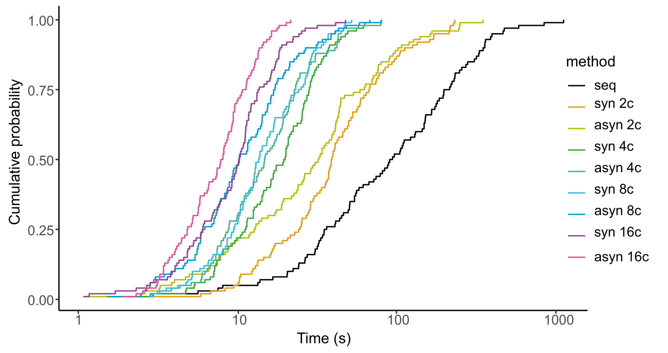

Fig. 2. Cumulative probability of reaching the VTR versus elapsed time for the `pr2392.tsp` instance problem.

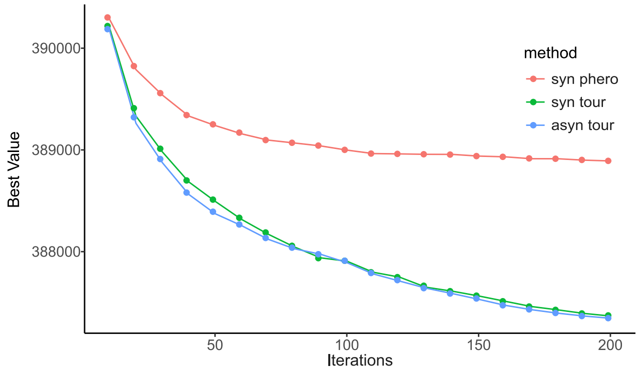

Fig. 3. Best value over iteration for the first 200 iterations in the `pr2392.tsp` instance problem. Comparing a synchronous strategy exchanging the pheromone matrix or the best found tour in each colony every 10 iterations with the asynchronous strategy.

Table 4
Average iteration times (s) for different parallel strategies in the `pr2392.tsp` instance problem.

Colonies	Synchronous strategy		Asynchronous strategy
	Pheromones	Best tours	
2	0.065	0.053	0.054
4	0.083	0.054	0.055
8	0.123	0.059	0.058
16	0.218	0.062	0.060

iterations. We have carried out similar experiments to be able to graphically confirm the behavior of the different parallelization strategies. Fig. 3 shows the best tour length achieved for the first 200 iterations, for two different synchronous strategies: one in which the exchanged information is the complete pheromone matrix, and another one where it consists of the best tours found. In both cases, a high exchange frequency has been chosen (every 10 iterations a communication between colonies is performed). The convergence curve for the asynchronous strategy proposed in this work, where the exchange of information is driven by the quality of the solutions found and not by the time elapsed (or the number of iterations), is also included in the graph. These results correspond to executions using 8 colonies. The figure shows that better results (in terms of convergence) are obtained by exchanging only the best tours (which will subsequently participate in the pheromones update) than by sharing the entire matrix. In addition, exchanging the pheromone matrix has other disadvantages, such as that the exchange has to take place at predetermined times forcing at least partial synchronization protocols among the colonies; also the size of the exchanged message can be huge for very large problems, even turning it unfeasible, and always penalizing performance. Table 4 shows the times per iteration calculated for the three strategies: synchronous sharing a pheromone matrix every 10 iterations, synchronous exchanging best solutions every 10 iterations, and asynchronous exchanging the best solutions as they are found. As it can be seen, the cost of the solution where the complete pheromone matrix is shared increases considerably with the number of colonies, and in all cases it is larger than the overhead of the other strategies. On the other hand, the synchronous and asynchronous strategies that exchange best tours present similar results in terms of iteration time. But, although for a small number of colonies the synchronous one obtains slightly better results because the asynchronous version has a small overhead due to the continuous checks for the arriving of new messages in each iteration, as the number of colonies increases the cost of the synchronous is slightly larger. Besides, the results are similar because these TSP problems have an almost constant cost per iteration throughout the entire execution. In other problems where the time per iteration is not as balanced as in this one, the asynchronous strategy has clear advantages over the synchronous one. This question is briefly addressed at the end of Section 6.3, where the possibility of configuring different colonies with different parameters is proposed.

Focusing on the synchronous strategy that exchanges the best tours, Fig. 4 shows the impact of the information exchange frequency in the convergence of the algorithm. The more frequently the information is exchanged the better. However, frequent synchronizations will harm the overall execution time.

Finally, Fig. 5 demonstrates the improvement in the `asyn-ACO` proposal when the number of colonies increases.

Most of the papers that can be found in the literature, report results for experiments that combine stopping criterion using quality and predefined effort. Thus, another experiment was carried out using a predefined effort as stopping criterion along with

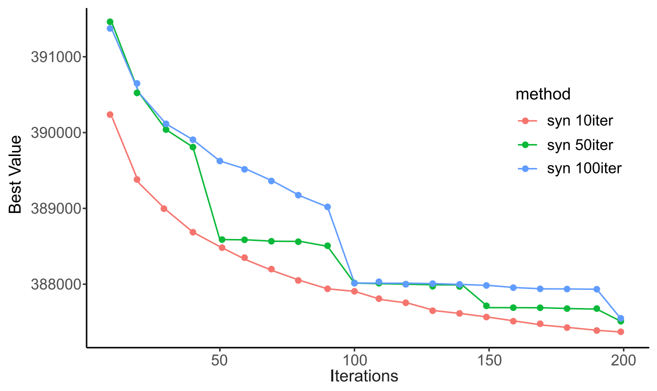

Fig. 4. Best value over iteration for the first 200 iterations in the `pr2392.tsp` instance problem. Comparing a synchronous strategy using 8 colonies and different communication frequency (10, 50 and 100 iterations).

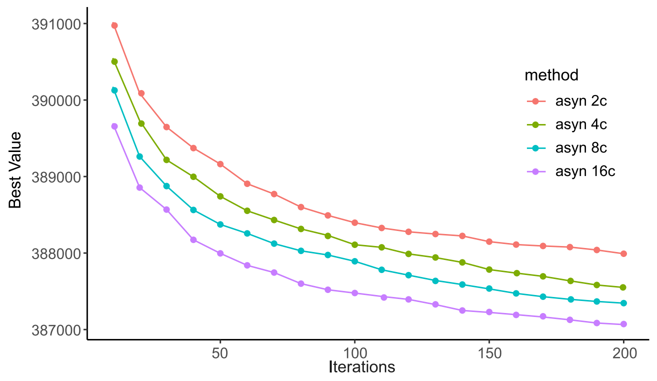

Fig. 5. Best value over iteration for the first 200 iterations in the `pr2392.tsp` instance problem. Comparing results for `asyn-ACO` increasing the number of colonies.

Table 5
Results using as stopping criterion a predefined effort (maximum time of 1000 s), reporting best, worst and average tour lengths, and comparing runs that achieve a VTR = 387 500 (%hits) for the `pr2392.tsp` instance problem.

Colonies	Method	Best	Worst	Average	%hits
1	seq-ACO	385410	387657	386559 ± 443	0%
2	emb-ACO	385174	387238	386288 ± 367	4%
	syn-ACO	384879	386556	385911 ± 433	20%
	asyn-ACO	384669	387067	385959 ± 442	19%
4	emb-ACO	384791	386963	386088 ± 385	7%
	syn-ACO	384114	386585	385654 ± 449	34%
	asyn-ACO	384675	386376	385572 ± 401	45%
8	emb-ACO	385071	386593	385878 ± 331	14%
	syn-ACO	384162	386123	385354 ± 424	72%
	asyn-ACO	383427	386408	385369 ± 417	69%
16	emb-ACO	384176	386334	385650 ± 345	24%
	syn-ACO	383546	386034	385169 ± 519	82%
	asyn-ACO	383547	385958	385093 ± 416	93%

a VTR, and reporting the percentage of hits achieved for each method. Table 5 shows, for each method, the best, the worst and the average of final tour length in those runs, and the percentage of hits.

To better illustrate the results of Table 5, Fig. 6 shows beanplots with the distribution of the achieved solutions, comparing the synchronous and asynchronous solutions. As it can be seen, the results achieved by both versions in these problems are

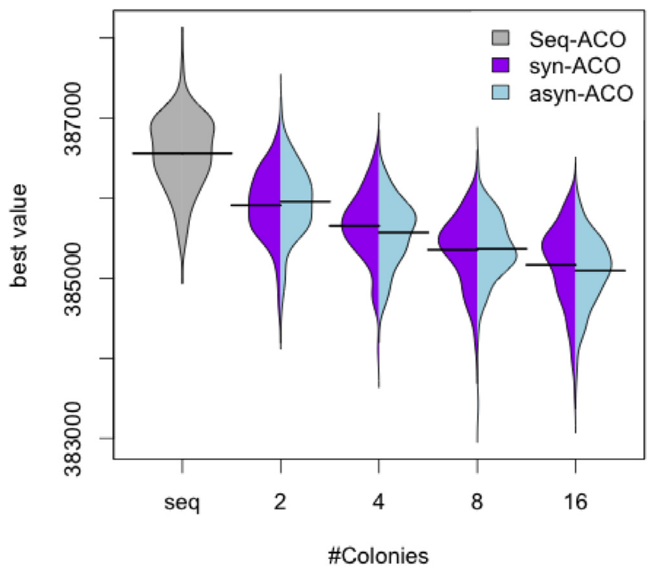

Fig. 6. Beanplots for the results reported in Table 5.

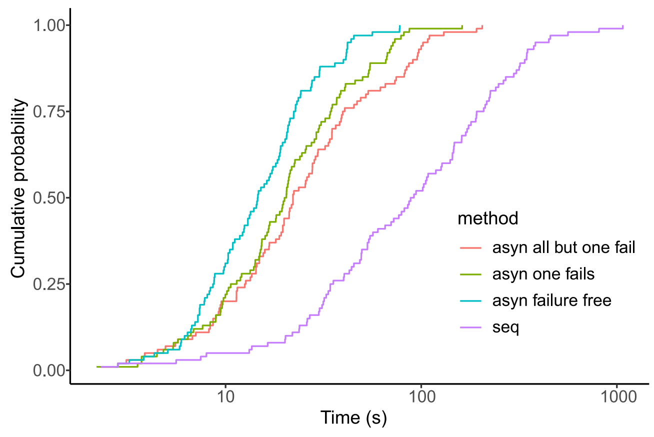

Fig. 7. Cumulative probability of reaching the VTR versus elapsed time. Comparing a sequential and 3 parallel asynchronous executions with 4 colonies: a fault-free execution, an execution that experiments a fault in all the colonies but one (3-faults), and an execution that experiments a fault in a single colony (1-fault).

similar, since the cooperation strategy is also similar (exchanging the best tours) and the time per iteration is almost constant, which keeps the colonies naturally synchronized. Nevertheless, the asynchronous strategy has other benefits. On the one hand, cooperation is more effective, since good solutions are shared as soon as they are found, thus eliminating the need to introduce the frequency of information exchange as an additional parameter in the algorithm. Moreover, this feature would especially impact other problems where the colonies are not naturally synchronized. On the other hand, the decentralized and asynchronous strategy makes the implementation fault-tolerant, since the colonies will not wait for messages that have not yet arrived from neighboring colonies. Thus, delays in communications, or even permanent failures in nodes/colonies would not affect the completion of the executions.

To demonstrate the benefit of the fault tolerance feature in the proposed strategy we have carried out two additional experiments injecting faults into executions using 4 initial colonies. In one of the experiments every 100 iterations a fault is injected in a running colony until only one remains alive. In the second experiment a fault is injected only in one colony at iteration 100, while the rest of the colonies remain alive. Fig. 7 shows the cumulative probability curve versus execution time comparing both experiments with the sequential execution and with the execution of the fault-free `asyn-ACO` with 4 colonies. As it can be seen, the results are impacted by the faults, being worse than the fault-free execution, however, they still outperform the sequential execution. Note that not all runs performed in the experiments required more than 100 iterations to reach the VTR, so they were not affected by fault injection.

6.2. Performance evaluation of the fine-grained parallelization

The goal of the fine-grained parallelization is to perform the construction of the solution for each ant using parallel threads, thus, accelerating the execution without altering the properties of the algorithm. As commented in Section 5, the OpenMP parallel loops are located in the construction of the solution and the local search procedures. Fig. 8 shows the speedup obtained in these procedures isolated for the `pr2392.tsp` instance problem. As it can be seen, the parallelization of the local search procedure achieves a good performance, but the results for the construction

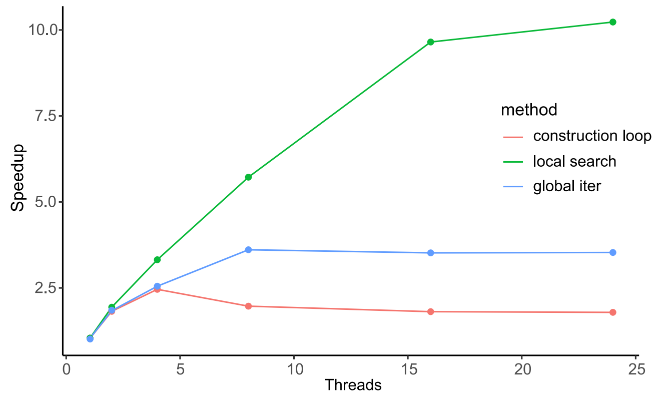

Fig. 8. Speedup of the OpenMP parallelization for the pr2392.tsp instance problem. Results for the parallelization of the construction loop and local search isolated and for the global iteration.

of the solutions are more limited. This is due to the overhead associated with the creation and final synchronization of the parallel threads. Note that for a single ant the construction of one solution took less than 20 ms, while the local search took almost 50 ms. When the number of threads is small, more ants are assigned to each thread, but as the number of threads increases, the number of ants per thread is reduced to the lower bound of one ant per thread. Whereas procedures with high computational cost, such as local search, largely make up for that overhead, lighter procedures do not.

The speedup for the global iteration is also shown in Fig. 8. It stagnates over 8 threads and does not exceed a value of 4. This complies with Amdahl's law, which allows us to calculate the maximum acceleration that we will achieve in a system if we improve only a part of its execution. Given that 70% of the execution takes place in the local search (which is the part that clearly benefits from this parallelization), the maximum speedup that, according to Amdahl's law, we could reach would be 3.

6.3. Performance evaluation of the hybrid parallel method

To graphically illustrate the performance of the hybrid method, Figs. 9 and 10 show, for the `pr2392.tsp` instance problem, the

Colonies	Threads					
MPI	1	2	4	8	16	24
1	97,65%	97,87%	97,93%	97,93%	97,94%	97,94%
2	97,65%	98,03%	98,05%	98,08%	98,10%	98,12%
4	97,65%	98,11%	98,19%	98,19%	98,18%	98,20%
8	97,66%	98,16%	98,22%	98,23%	98,29%	98,28%
16	97,66%	98,22%	98,30%	98,30%	98,32%	98,33%

Fig. 9. Results of best tours encountered for a predefined effort of 1000 s, expressed as a percentage of the proximity to the optimal solution, for different configurations of the number of colonies (MPI processes) and the number of OpenMP threads in each colony. Each entry is an average of 30 individual runs. Color coded from worse in red, to best in green.

Colonies	Threads					
MPI	1	2	4	8	16	24
1	146,36	68,92	49,98	49,09	35,12	33,26
2	47,45	23,89	11,22	10,01	11,11	13,16
4	18,41	12,26	6,87	4,90	5,07	4,06
8	14,00	8,99	4,64	3,39	2,92	2,97
16	8,25	4,54	3,11	3,14	2,02	1,64

Fig. 10. Results of execution time needed to reach a VTR = 387 500 for different configurations of the number of colonies (MPI processes) and the number of OpenMP threads in each colony. Each entry is an average of 30 individual runs. Color coded from worse in red, to best in green.

Colonies	Threads					
MPI	1	2	4	8	16	24
1	0,04	0,09	0,12	0,12	0,17	0,18
2	0,13	0,26	0,54	0,61	0,27	0,23
4	0,33	0,50	0,89	0,62	0,30	0,38
8	0,44	0,68	0,66	0,60	0,26	0,26
16	0,74	0,67	0,65	0,49	0,19	0,23

Fig. 11. Efficiency attending to results in Fig. 10 computed as efficiency=time/#cores, where #cores includes all the cores reserved for each experiment in the Finisterrae-II. Color coded from worse in red, to best in green.

results obtained for different combinations of MPI colonies and OpenMP threads both in an experiment with a predefined effort (Fig. 9), and in an experiment with quality stopping criterion (Fig. 10). At the sight of these results we can conclude that it is necessary to find a good balance between the number of colonies in the coarse-grained parallelization and the number of threads in the fine-grained strategy in order to maximize the efficiency of the parallel algorithm. A popular metric in the HPC field is efficiency (e), which is calculated as e=speedup/cores. To decide which combinations of MPI vs. OpenMP are the most advantageous, both from a performance and computational efficiency point of view, it is necessary to take into account the total number of cores allocated to each experiment. In our case, since the Finisterrae-II supercomputer allocates full nodes to users, this means that all the cores of a node (24) must be accounted for even if fewer are actually used. Fig. 11 shows the efficiency of the results shown in Fig. 10 taking into account the total number of nodes/cores reserved for each experiment, computed as

$$e = \frac{Time_{seq}}{Time_{parallel} * cores_{reserved}} \tag{5}$$

The performance results must be analyzed taking into account both quality and execution time criteria, as well as efficiency. In general terms, the choice would be to run a colony in each computing node and as many threads as cores per node. However, if the number of cores per node is large, and the work to be distributed among the threads does not compensate for the use of so many cores, as in this case, the number of threads per colony should be reduced to locate more than one colony per node. In view of the previous results, we can conclude that the best combinations for these problems and the supercomputer that we are using in this work is between 4 and 8 threads per colony (which results in 3 to 6 colonies per node).

Table 6 shows results for problems `pr2392.tsp`, `rl5934.tsp` and `brd14051.tsp` tested using from 48 to 384 cores with different configurations of colonies and 8 threads per colony. The stopping criterion used was a predefined effort; and best, worst, and average distance tours are reported, as well as percentage of hits that achieved a VTR. As it can be seen, the quality of the solution continues to improve as the number of cores increases and the percentage of hits increases considerably even for high numbers of processes, which demonstrates the good scalability of the proposal.

Finally, it has already been commented that an interesting application of combining multicolony strategy, an asynchronous

Table 6
Results using a predefined effort of 1000 s as stopping criterion, and comparing execution times for the hybrid proposal. All configurations use 8 threads per colony for the fine-grained parallelization except for the first experiment that corresponds to the sequential execution. Different configurations are tested using different number of colonies (cores/8) and nodes (cores/24).

Problem	Cores	Colonies	Best	Worst	Average	VTR	%hits
`pr2392.tsp`	1	1	385410	387657	385659 ± 443	384000	0%
	48	6	384443	385567	384908 ± 301	384000	0%
	96	12	383634	385325	384531 ± 438	384000	13%
	192	24	383740	385065	384382 ± 360	384000	17%
	384	48	382878	384673	384017 ± 404	384000	50%
`rl5934.tsp`	1	1	568474	572589	570858 ± 1104	568000	0%
	48	6	567371	569832	568666 ± 682	568000	17%
	96	12	566952	569138	568227 ± 488	568000	33%
	192	24	566811	568548	567941 ± 522	568000	60%
	384	48	566210	568548	567499 ± 551	568000	87%
`brd14051.tsp`	1	1	485722	486596	486226 ± 249	485400	0%
	48	6	484896	485991	485598 ± 235	485400	17%
	96	12	484932	485788	485472 ± 234	485400	30%
	192	24	485073	485764	485400 ± 174	485400	53%
	384	48	484748	485549	485184 ± 221	485400	87%

Table 7
Configuration parameters for the heterogeneous multicolony experiment reported in Table 8.

Colony	α	β	ρ	#ants
A	1	2	0.5	25
B	1	5	0.2	32
C	1	10	0.8	16
D	1	15	0.6	40

communication protocol, and sharing only the best found solutions instead of the pheromone matrix, is to develop optimizers where each colony uses different parameters, or even different variants of the ACO. To demonstrate this point, as a proof of concept, the largest problem (`brd14051.tsp`) has been executed with 4 different variants of colonies each using a different parameter combination, specified in Table 7. The results obtained are shown in Table 8, where it can be seen that they improve the results obtained with the homogeneous configuration used in the experiments reported in Table 6. Actually, the heterogeneous configuration will not improve the results of the best homogeneous configuration, but given that the appropriate values of the ACO parameters for a new problem are not usually known in advance, the execution in HPC environments with strategies such as the one presented in this work would allow to collaboratively explore different variants, taking advantage of the most successful ones.

7. Conclusions

This paper presents a hybrid MPI+OpenMP parallel implementation of the well-known Ant Colony Optimization (ACO) metaheuristic. It combines multicolony parallelization, focused on stimulating the diversification in the search and the cooperation between different colonies, with fine-grained parallelization, aimed at accelerating the computation by means of performing the construction of the solutions in parallel. This approach improves the algorithm efficiency and leads to a better use of computational resources through a balance between diversification and intensification.

The main contributions of this proposal are: (1) a completely decentralized approach, in contrast to the classical centralized master–slave approaches, which improves the efficient use of available resources; (2) the exchange of information driven by quality of the solutions, rather than by time elapsed or a fix number of iterations, in order to achieve a more effective cooperation between colonies; and (3) an asynchronous communication protocol between colonies, in order to avoid idle processes while waiting for information exchanged from other colonies.

A comprehensive evaluation has been carried out using three medium and large instances of the TSPLIB library, obtaining encouraging results. The main guidelines that can be extracted from these experiments are:

- For large problems, or those that easily suffer from stagnation at local minima, it is better to prioritize the number of colonies over the number of threads per colony, since in large problems stimulating diversification accelerates the convergence of problems and the cooperation between colonies has demonstrated its effectiveness to get out of local minima.
- The performance, taking the number of threads per colony into consideration, is limited to the number of ants and the execution time of the construction and evaluation of the solution. Either a small number of ants, or an effortless problem where the construction and evaluation of the solution were too fast, would cause the distribution of work between the threads not to pay for the overhead of the fork-join loops.

Therefore, it is important to reach a compromise between the number of colonies and the number of threads per colony, being convenient to use as many colonies as available computing nodes and as many threads as cores per node, decreasing the number of threads per colony to locate more than one colony per node in case the number of cores per node was large enough and diversification was favored over intensification.

The proposed implementation is initially designed to maximize efficiency in traditional HPC infrastructures such as multicore clusters. However, its key features, i.e. decentralization and an asynchronous communication protocol, make the proposal particularly suitable for cloud environments and local computer networks, where communications between nodes may not be reliable or latency be too high. Thus, a communication delay from one colony (or even its absence, in case of failure) would not cause the rest to stall, and the algorithm could continue until the task was completed.

The proposal presented in this work lays the foundations for future developments where the colonies can execute different variants of the algorithm or with different parameters, cooperating with each other. Moreover, an adaptive method could be developed that would allow colonies to self-reconfigure at runtime based on the best tours found by other colonies with different configurations.

The source code is made publicly available at DOI: https://doi.org/10.5281/zenodo.5711353.

Table 8
Results of the heterogeneous multicolony experiment for the `brd14051.tsp` problem using the variants described in Table 7 and a predefined effort of 1000 s and a VTR = 484 500 as stopping criteria. All configurations use 8 threads per colony for the fine-grained parallelization except for the first experiment that corresponds to the sequential execution.

Variant	Cores	Colonies	Best	Worst	Average	VTR	%hits
Colony A	1	1	485722	486596	486226 ± 249	484500	0%
Colony B	1	1	484437	485424	485177 ± 297	484500	10%
Colony C	1	1	484611	485543	485253 ± 265	484500	0%
Colony D	1	1	484916	485565	485286 ± 162	484500	0%
Heterogeneous multicolony	48	6	484118	484993	484653 ± 188	484500	20%
	96	12	484085	484698	484432 ± 191	484500	60%
	192	24	483775	484574	484287 ± 225	484500	80%
	384	48	483672	484567	484256 ± 196	484500	93%

CRediT authorship contribution statement

Patricia González: Conceptualization, Methodology, Software, Validation, Writing – original draft, Supervision. **Roberto R. Osorio:** Software, Visualization, Writing – original draft. **Xoan C. Pardo:** Software, Writing – review & editing. **Julio R. Banga:** Conceptualization, Writing – review & editing, Funding acquisition. **Ramón Doallo:** Conceptualization, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00 / AEI / 10.13039/501100011033), and by Xunta de Galicia and FEDER funds of the EU (Centro de Investigación de Galicia accreditation 2019–2022, ref. ED431G 2019/01; Consolidation Program of Competitive Reference Groups, ref. ED431C 2021/30). JRB acknowledges funding from the Ministry of Science and Innovation of Spain MCIN / AEI / 10.13039/501100011033 through grant PID2020-117271RB-C22 (BIODYNAMICS), and from MCIN / AEI / 10.13039/501100011033 and "ERDF A way of making Europe" through grant DPI2017-82896-C2-2-R (SYNBIOCONTROL). Authors also acknowledge the Galician Supercomputing Center (CESGA) for the access to its facilities. Funding for open access charge: Universidade da Coruña/CISUG.

References

[1] P.M. Pardalos, D.-Z. Du, R.L. Graham, Handbook of Combinatorial Optimization, Springer, 2013.
[2] F. Glover, G. Kochenberger (Eds.), Handbook of Metaheuristics, Kluwer Academic Publisher, 2006.
[3] M. Dorigo, Optimization, learning and natural algorithms, (Ph.D. thesis), Dipartamento di Electronica, Politecnico di Milano, Italy, 1992.
[4] M. Dorigo, V. Maniezzo, A. Colorni, Ant system: optimization by a colony of cooperating agents, IEEE Trans. Syst. Man Cybern. B 26 (1) (1996) 29–41.
[5] M. Pedemonte, S. Nesmachnow, H. Cancela, A survey on parallel ant colony optimization, Appl. Soft Comput. 11 (2011) 5181–5197.
[6] M.A.S. Netto, R.N. Calheiros, E.R. Rodrigues, R.L.F. Cunha, R. Buyya, HPC Cloud for scientific and business applications: Taxonomy, vision, and research challenges, ACM Comput. Surv. 51 (1) (2018).
[7] J. Dongarra, T. Herault, Y. Robert, Fault tolerance techniques for high-performance computing, in: Fault-Tolerance Techniques for High-Performance Computing, 2015, pp. 3–85.
[8] N. Losada, P. González, M.J. Martín, G. Bosilca, A. Bouteiller, K. Teranishi, Fault tolerance of MPI applications in exascale systems: The ulfm solution, Future Gener. Comput. Syst. 106 (2020) 467–481.
[9] C. Di Martino, Z. Kalbarczyk, R. Iyer, Measuring the resiliency of extreme-scale computing environments, in: Principles of Performance and Reliability Modeling and Evaluation, 2016, pp. 609–655.
[10] M. Dorigo, T. Stützle, Ant colony optimization: Overview and recent advances, in: Handbook of Metaheuristics, Springer International Publishing, 2019, pp. 311–351.
[11] O. Montiel-Ross, N. Medina-Rodriguez, R. Sepulveda, P. Melin, Methodology to optimize manufacturing time for a CNC using a high performance implementation of ACO, Int. J. Adv. Robot. Syst. 9 (4) (2012) 121.
[12] M.J. Meena, K. Chandran, A. Karthik, A.V. Samuel, An enhanced ACO algorithm to select features for text categorization and its parallelization, Expert Syst. Appl. 39 (5) (2012) 5861–5871.
[13] D. Zaidman, H.J. Wolfson, Pinacolada: peptide–inhibitor ant colony ad-hoc design algorithm, Bioinformatics 32 (15) (2016) 2289–2296.
[14] Y. Huo, J.X. Huang, Parallel ant colony optimization for flow shop scheduling subject to limited machine availability, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, 2016, pp. 756–765.
[15] D. Thiruvady, A.T. Ernst, G. Singh, Parallel ant colony optimization for resource constrained job scheduling, Ann. Oper. Res. 242 (2) (2016) 355–372.
[16] E. Grakova, K. Slaninová, J. Martinovič, J. Křenek, J. Hanzelka, V. Svatoň, Waste collection vehicle routing problem on HPC Infrastructure, in: IFIP International Conference on Computer Information Systems and Industrial Management, 2018, pp. 266–278.
[17] A. ElSaid, F. El Jamiy, J. Higgins, B. Wild, T. Desell, Optimizing long short-term memory recurrent neural networks using ant colony optimization to predict turbine engine vibration, Appl. Soft Comput. 73 (2018) 969–991.
[18] Y. Fan, G. Wang, X. Lu, G. Wang, Distributed forecasting and ant colony optimization for the bike-sharing rebalancing problem with unserved demands, PLoS One 14 (12) (2019) e0226204.
[19] E. Martin, A. Cervantes, Y. Saez, P. Isasi, IACS-Hcsp: Improved ant colony optimization for large-scale home care scheduling problems, Expert Syst. Appl. 142 (2020) 112994.
[20] I. Dzalbs, T. Kalganova, Accelerating supply chains with ant colony optimization across a range of hardware solutions, Comput. Ind. Eng. 147 (2020) 106610.
[21] H. Baniata, A. Anaqreh, A. Kertesz, PF-BTS: A Privacy-aware fog-enhanced blockchain-assisted task scheduling, Inf. Process. Manage. 58 (1) (2021) 102393.
[22] L. Wan, G. Zhang, H. Li, C. Li, A novel bearing fault diagnosis method using spark-based parallel ACO-K-Means clustering algorithm, IEEE Access 9 (2021) 28753–28768.
[23] E.B. Tirkolaee, M. Alinaghian, A.A.R. Hosseinabadi, M.B. Sasi, A.K. Sangaiah, An improved ant colony optimization for the multi-trip capacitated arc routing problem, Comput. Electr. Eng. 77 (2019) 457–470.
[24] J. Liu, J. Yang, H. Liu, X. Tian, M. Gao, An improved ant colony algorithm for robot path planning, Soft Comput. 21 (19) (2017) 5829–5839.
[25] D. Zhang, X. You, S. Liu, H. Pan, Dynamic multi-role adaptive collaborative ant colony optimization for robot path planning, IEEE Access 8 (2020) 129958–129974.
[26] X.-F. Liu, Z.-H. Zhan, K.-J. Du, W.-N. Chen, Energy aware virtual machine placement scheduling in cloud computing based on ant colony optimization approach, in: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, 2014, pp. 41–48.
[27] F. Alharbi, Y.-C. Tian, M. Tang, W.-Z. Zhang, C. Peng, M. Fei, An ant colony system for energy-efficient dynamic virtual machine placement in data centers, Expert Syst. Appl. 120 (2019) 228–238.
[28] R. Kleinkauf, M. Mann, R. Backofen, Antarna: ant colony-based RNA sequence design, Bioinformatics 31 (19) (2015) 3114–3121.
[29] V. Salmaso, S. Moro, Bridging molecular docking to molecular dynamics in exploring ligand-protein recognition process: an overview, Front. Pharmacol. 9 (2018) 923.
[30] E.-G. Talbi, O. Roux, C. Fonlupt, D. Robillard, Parallel ant colonies for the quadratic assigment problem, Future Gener. Comput. Syst. 17 (4) (2001) 441–449.

[31] P. Delisle, M. Krajecki, M. Gravel, C. Cagné, Parallel implementation of an ant colony optimization with OpenMP, in: International Conference of Parallel Architectures and Complication Techniques, Proceedings of the Third European Workshop on OpenMP, 2001, pp. 8–12.
[32] M. Craus, L. Rudeanu, Parallel framework for ant-like algorithms, in: Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2001, pp. 36–41.
[33] Q. Xia, P. Quian, A parallel ACO approach based on one pheromone matrix, in: Proceedings of the 5th International Workshop on Ant Colony Optimization and Swarm Intelligence, in: Lecture Notes in Computer Science, vol. 4150, 2006, pp. 332–339.
[34] S. Tsutsui, N. Fujimoto, Parallel ant colony optimization algorithm on a multicore processor, in: International Conference on Swarm Intelligence, in: Lecture Notes in Computer Science, vol. 6234, 2010, pp. 488–495.
[35] B. Bullnheimer, G. Kotsis, C. Strauss, High performance algorithms and software in nonlinear optimization, Kluwer Academic Publishers, 1998, pp. 87–100.
[36] X. Jie, L. CaiYun, C. Zhong, A new parallel ant colony optimization algorithm based on message passing interface, in: 2008 IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application, 2008, vol. 2, pp. 178–182.
[37] M. Starzec, G. Starzec, A. Byrski, W. Turek, K. Piętak, Desynchronization in distributed ant colony optimization in hpc environment, Future Gener. Comput. Syst. 109 (2020) 125–133.
[38] Y. Zhou, F. He, N. Hou, Y. Qiu, Parallel ant colony optimization on multi-core SIMD CPUs, Future Gener. Comput. Syst. 79 (2018) 473–487.
[39] J.A. Mocholi, J. Jaen, J.H. Canos, A grid ant colony algorithm for the orienteering problem, in: 2005 IEEE Congress on Evolutionary Computation, vol. 1, 2005, pp. 942–949.
[40] K. Doerner, R. Hartl, S. Benkner, M. Luckà, Parallel cooperative savings based ant colony optimization – multiple search and decomposition approaches, Parallel Process. Lett. 16 (3) (2006) 351–370.
[41] M. Randall, A. Lewis, A parallel implementation of ant colony optimization, J. Parallel Distrib. Comput. 62 (9) (2002) 1421–1432.
[42] P. Delisle, M. Gravel, M. Krajecki, C. Cagné, Comparing parallelization of an ACO: Message passing vs. Shared memory, in: Proceedings of the 2nd International Workshop on Hybrid Metaheuristics, in: Lecture Notes in Computer Science, vol. 3636, 2005, pp. 1–11.
[43] A. Hadian, S. Shahrivari, B. Minaei-Bidgoli, A fine-grained parallel ant colony system for shared-memory architectures, Int. J. Comput. Appl. 53 (8) (2012).
[44] M. Pedemonte, H. Cancela, A cellular ant colony optimisation for the generalised steiner problem, Int. J. Innovative Comput. Appl. 2 (3) (2010) 188–201.
[45] T. Stützle, Parallelization strategies for ant colony optimization, in: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, in: Lecture Notes in Computer Science., vol. 1498, 1998, pp. 722–731.
[46] M. Rahoual, R. Hadji, V. Bachelet, Parallel ant system for the set covering problem, in: Proceedings of the 3rd International Workshop on Ant Algorithms, in: Lecture Notes in Computer Science, 2463, 2002, pp. 262–267.
[47] E. Alba, G. Leguizamón, G. Ordoñez, Two models of parallel ACO algorithms for the minimum tardy task problem, Int. J. High Perfom. Syst. Archit. 1 (1) (2007) 50–59.
[48] H. Bai, D. OuYang, X. Li, L. He, H. Yu, MAX-MIN Ant System on GPU with CUDA, in: 2009 Fourth International Conference on Innovative Computing, Information and Control, ICICIC, 2009, pp. 801–804.
[49] R. Michel, M. Middendorf, An island model based ant system with lookahead for the shortest supersequence problem, in: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, in: Lecture Notes in Computer Science, 1498, 1998, pp. 692–701.
[50] R. Michel, M. Middendorf, An aco algorithm for the shortest common supersequence problem, in: New Ideas in Optimization, McGraw-Hill, 1999, pp. 51–62.
[51] D.A.L. Piriyakumar, P. Levi, A new approach to exploiting parallelism in ant colony optimization, in: Proceedings of 2002 International Symposium on Micromechatronics and Human Science, 2002, pp. 237–243.
[52] S.-C. Chu, J.F. Roddick, J.-S. Pan, Ant colony system with communication strategies, Inform. Sci. 167 (1) (2004) 63–76.
[53] C. Twomey, T. Stützle, M. Dorigo, M. Manfrin, M. Birattari, An analysis of communication policies for homogeneous multi-colony aco algorithms, Inform. Sci. 180 (12) (2010) 2390–2404.
[54] I. Ellabib, P. Calamai, O. Basir, Exchange strategies for multiple ant colony system, Inform. Sci. 177 (5) (2007) 1248–1264.
[55] L. Chen, H.-Y. Sun, S. Wang, Parallel implementation of ant colony optimization on MPP, in: 2008 International Conference on Machine Learning and Cybernetics, vol. 2, 2008, pp. 981–986.
[56] L. Chen, H.-Y. Sun, S. Wang, A parallel ant colony algorithm on massively parallel processors and its convergence analysis for the travelling salesman problem, Inform. Sci. 199 (2012) 31–42.
[57] H. Liu, P. Li, Y. Wen, Parallel Ant Colony Optimization Algorithm, in: 2006 6th World Congress on Intelligent Control and Automation, vol. 1, pp. 3222–3226.
[58] C. Liu, L. Li, Y. Xiang, Research of multi-path routing protocol based on parallel ant colony algorithm optimization in mobile ad hoc networks, in: Fifth International Conference on Information Technology: New Generations, Itng 2008, 2008, pp. 1006–1010.
[59] J. Hajewski, S. Oliveira, D.E. Stewart, L. Weiler, Exploring Trade-offs in Parallel Beam-ACO, in: 2021 IEEE 11th Annual Computing and Communication Workshop and Conference, CCWC, 2021, pp. 1525–1534.
[60] J.M. Cecilia, J.M. García, A. Nisbet, M. Amos, M. Ujaldón, Enhancing data parallelism for ant colony optimization on GPUs, J. Parallel Distrib. Comput. 73 (1) (2013) 42–51.
[61] A. Delévacq, P. Delisle, M. Gravel, M.A. l Krajecki, Parallel ant colony optimization on graphics processing units, J. Parallel Distrib. Comput. 73 (1) (2013) 52–61.
[62] R. Skinderowicz, The GPU-based parallel ant colony system, J. Parallel Distrib. Comput. 98 (2016) 48–60.
[63] Y. Zhou, F. He, Y. Qiu, Dynamic strategy based parallel ant colony optimization on GPUs for TSPs, Sci. China Inf. Sci. 60 (2017).
[64] J.M. Cecilia, A. Llanes, J.L. Abellán, J. Gómez-Luna, L.-W. Chang, W.-M.W. Hwu, High-throughput ant colony optimization on graphics processing units, J. Parallel Distrib. Comput. 113 (2018) 261–274.
[65] B. Wu, G. Wu, M. Yang, A MapReduce based Ant Colony Optimization approach to combinatorial optimization problems, in: 2012 8th International Conference on Natural Computation, 2012, pp. 728–732.
[66] A. Mohan, G. Remya, A parallel implementation of ant colony optimization for TSP based on MapReduce framework, Int. J. Comput. Appl. 88 (8) (2014).
[67] A. Banharnsakun, A MapReduce-based artificial bee colony for large-scale data clustering, Pattern Recognit. Lett. 93 (2017) 78–84, Pattern Recognition Techniques in Data Mining.
[68] D. Gaifang, F. Xueliang, L. Honghui, X. Pengfei, Cooperative ant colony-genetic algorithm based on spark, Comput. Electr. Eng. 60 (2017) 66–75.
[69] M. Črepinšek, S.-H. Liu, M. Mernik, Exploration and exploitation in evolutionary algorithms: A survey, ACM Comput. Surv. 45 (3) (2013) 35:1–35:33.
[70] T. Crainic, M. Toulouse, Parallel strategies for meta-heuristics, in: Handbook of Metaheuristics, in: International Series in Operations Research & Management Science, vol. 57, 2003, pp. 475–513.
[71] E. Alba, Parallel Metaheuristics: A New Class of Algorithms, vol. 47, Wiley-Interscience, 2005.
[72] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, B. Chapman, High performance computing using MPI and openmp on multi-core parallel systems, Parallel Comput. 37 (9) (2011) 562–575.
[73] G. Reinelt, TSPLIB- a traveling salesman problem library, ORSA J. Comput. 3 (1991) 376–384.