mathematics

MDPI

Article

Improved Method for Parallelization of Evolutionary Metaheuristics

Diego Díaz [1,*,†], Pablo Valledor [1,†], Borja Ena [1,†], Miguel Iglesias [1,†] and César Menéndez [2,†]

1 ArcelorMittal Global R&D Asturias, Av. Marques de Suances s/n, 33400 Aviles, Spain; pablo.valledor-pellicer@arcelormittal.com (P.V.); borja.ena@arcelormittal.com (B.E.); miguel.iglesias@arcelormittal.com (M.I.)
2 Engineering Projects, University of Oviedo, EIMEM Independencia 13, 33001 Oviedo, Spain; cesarm@uniovi.es
* Correspondence: diego.diaz@arcelormittal.com
† These authors contributed equally to this work.

Received: 3 August 2020; Accepted: 26 August 2020; Published: 1 September 2020

Abstract: This paper introduces a method for the distribution of any and all population-based metaheuristics. It improves on the naive approach, independent multiple runs, while adding negligible overhead. Existing methods that coordinate instances across a cluster typically require some compromise of more complex design, higher communication loads, and solution propagation rate, requiring more work to develop and more resources to run. The aim of the new method is not to achieve state-of-the-art results, but rather to provide a better baseline method than multiple independent runs. The main concept of the method is that one of the instances receives updates with the current best solution of all other instances. This work describes the general approach and its particularization to both genetic algorithms and ant colony optimization for solving Traveling Salesman Problems (TSPs). It also includes extensive tests on the TSPLIB benchmark problems of resulting quality of the solutions and anytime performance (solution quality versus time to reach it). These tests show that the new method yields better solutions for about two thirds of the problems and equivalent solutions in the remaining third, and consistently exhibits better anytime performance.

Keywords: metaheuristics; genetic algorithm; ant colony optimization; distributed computing

1. Introduction

Metaheuristics encompass a wide range of algorithms for optimization. Originally applied to heuristics or strategies that coordinate lower level search procedures, in time the concept extended to include methods that employ ways of escaping local optima in complex search spaces, typically several neighbourhood strategies or some extent of randomness. While these algorithms cannot compete with strict optimization methods where they are applicable, such as solving linear programs, they excel at attaining good enough solutions for problems that do not match the expression capabilities of strict methods or that span a large search space that cannot be exhaustively searched by exact methods [1].

Given their diversity, there are many different classifications of metaheuristics, according to different features. The classification most relevant to this work divides them into two classes:

1. Population-based, which at any time hold a number of solutions (the population) and create new ones by combining or modifying them; some examples of this class are Genetic Algorithms, Ant Colony Optimization, and Artificial Bee Colony.
2. Single-solution, which focus on iteratively improving just one solution; some examples of this class are Variable Neighbourhood Search, Simulated Annealing, and Iterated Local Search.

Population-based metaheuristics are more amenable to parallelization; at each step of the process, the same operations are carried out on several individuals of the population: all of them, a subset, or some pair-wise combination, depending on the method. Single-solution metaheuristics, on the other hand, are by nature sequential. This does not mean that parallelization is not possible, but it is tied to the particular algorithm, such as parallelizing the calculation or evaluation of neighbours for Local Search variants, or even the particular problem, such as exploiting the properties of the fitness function. This second parallelization approach can also benefit population-based metaheuristics, and it is currently a vibrant field of research; but the common structure of population-based metaheuristics allows for the design of high level methods that apply to all of them.

Recent trends in the evolution of computing power stress the importance of parallelization and distribution for improving the performance and scalability of algorithms [2]. Moore's Law states that the number of transistors that can be fitted into a chip roughly doubles every 18 months. For a while, this translated directly into raw power increase, and the same programs became faster just by running on newer hardware.

More recently, as the development hit certain physical limits, increases in clock speed stalled and timing constraints made it necessary to separate the processor into several cores in order to extract all its potential. Single-thread programs no longer reap the rewards of increased computing power, as they only make use of a fraction of the processor.

To be able to take advantage of the new advances, programs need to parallelize operations. There are different ways to attain this, and they can be included in different layers of the design.

The paradigm for GPU acceleration follows Single Instruction Multiple Data (SIMD): the same sequence of operations are executed on many data instances. Its main advantages come from vector- or matrix-like calculations with very large numbers of items; actual GPU speeds are usually quite a bit slower than CPU speeds, but they make up for it by performing hundreds or thousands of simultaneous operations. This form of parallelization requires programming at very low level and taking care of interleaving the computations with data transfers to and from the GPU. Because of this, the use of GPU is very problem dependent and used in the innermost loops, such as fitness function evaluation.

When using the CPU, a distinction is needed between parallelization and distribution. The former merely implies that several operations can take place at the same time, whereas the latter also considers that the separate execution threads do not share the same context, but need to explicitly communicate with each other. In both cases, the instructions are independent for each instance, as opposed to the case of the GPU.

Parallelization would then correspond to a multi-threading approach, where several threads are run at the same time, while sharing the same memory space; likewise, distribution would correspond to separate processes running in the same or even different CPUs, and all coordination is carried out by means of inter-process communication (IPC) mechanisms, such as message passing, mutexes, etc.

Multi-threading is therefore useful for exploiting multiple cores in the same CPU, while distribution can take advantage of multiple CPUs, in the same or different (networked) computers.

In the context of population-based metaheuristics, multi-threading is typically used to run the operations on the individuals of the population at the same time, rather than one after another; and distribution is typically used to run several populations across a cluster, either independently or with some level of communication, usually in an Islands model, where each population transfers good solutions to its neighbours. Here, neighbours in the sense of the network topology; the transference only takes place between nodes that are directly connected to avoid high communication overhead. The most common topology is a 2-dimensional mesh, where each node is connected to four other nodes.

The benefits of enabling a more efficient use of the available resources must be balanced against the complexity involved, both in terms of building the algorithm (coding and maintaining) and the communication required, which can in practice outweigh the gains. This paper addresses this balance by focusing on performance improvement at the baseline end of the spectrum, while adding as little extra complexity and communication as possible, narrowing the gap between simple multiple parallel runs and fully connected systems.

The method interprets metaheuristics as a form of stochastic predictors of the optimal solution. They take a specific instance and calculate, through a combination of deterministic and random operations, a value that tries to approximate the solution to the problem.

Thus, running several instances of a metaheuristic is equivalent to using a group of stochastic predictors. Even if in the case of metaheuristics it is possible to know which of the instances yields the best solution by comparing their fitness, it is possible to draw some ideas from the statistician's toolbox to extract more value from running several instances.

The group of methods, employed in statistics and machine learning, collectively known as ensemble methods or ensemble learning deal with exactly this scenario. All of them combine multiple models or predictors to obtain a better result than possible with any one of the models individually [3].

The overarching theme in all ensemble methods is that improvement can be obtained from the aggregation of multiple (weak) models to build a better one.

The same concept is present in the very procedure of many metaheuristics: from the partial reuse of solutions in Genetic Algorithms, to the preference for frequently traveled edges in Ant Colony Optimization, to name a few.

Path Relinking showcases this behaviour better than any other metaheuristic: its concept itself is exactly to combine good solutions to find better ones. Starting from a pool of solutions (generated randomly and/or using some heuristic), it selects the best ones and builds paths between pairs of them. Each step consists of a modification of the current point as in a local search, but always moving towards the other solution (guiding solution). As new solutions are generated, they may become new path sources or guiding solutions.

Taking all of this into account, this paper proposes a method that combines multiple instances of any population-based metaheuristic to improve its efficiency, while introducing as little additional overhead as possible. For easier reading, the method is referred to as the Multiverse method in the remainder of the paper, as opposed to multiple independent runs, which is called Multistart. The metaphor for the Multiverse being that each instance is a universe of its own, but within this method they all form a single entity, hence the Multiverse.

In the Multiverse method, one of the multiple instances has special status: the collector. It receives updates of the best solutions of the other instances. This is straightforward, as all population-based metaheuristics already step through iterations (or generations), combine the solutions in their populations to create new, improved ones, and possess mechanisms to work with multiple solutions (the population). All other instances contribute their current best solution at each iteration to the population of the collector; the same process that takes place normally, applied to this extended population, is responsible for the mixing of solutions.

The added overhead is small: injection of external solutions into the population, and one-way communication of a single solution from each instance. Furthermore, this communication scheme fits a star-like topology, such as the one provided by a standard switch, rather than the more complex and costly mesh favoured by other configurations, such as the Islands model, and more typical of super-computers than of clusters built from commodity-grade computers.

John Holland introduced the concept of Genetic Algorithms (GAs) in [4]. The field has since expanded well beyond the original idea, into a more general class of Evolutionary Algorithms that share the underlying idea of simulating Darwinian evolution and natural selection through mutation and recombination of individuals. Reeves [5] provides a comprehensive review of the history and application of Genetic Algorithms.

Ant Colony Optimization (ACO) actually encompasses a number of similar algorithms for solving discrete optimization problems and which are inspired by the behaviour of real ant colonies in nature. The first ACO system, known as Ant System (AS), was introduced in [6]. Ant Colony Optimization as such is described in [7]. And in time additional variants have surfaced, such as Ant Colony System

(ACS) and MAX-MIN Ant System (MMAS). A broad and detailed overview of this developments is available in [8].

Intensive research exists on the application of GPU to accelerate metaheuristics in general, and GA or ACO in particular. A couple of examples suffice to understand that this approach lacks the generality that this work aims at. The PhD thesis [9] addresses metaheuristics in general, and describes in depth the complications of GPU use: multiple memory management, CPU and GPU communication, partitioning the code into sequential and parallel chunks, etc. More specifically for GAs, Krömmer et al. [10] implement GPU accelerated Genetic Algorithm and Differential Evolution methods for task scheduling by offloading problem-specific computation to the GPU. For ACO specifically, Delévacq et al. [11] review GPU implementations, but the more general among them still apply only to a specific algorithm of the ACO family for a specific problem.

In the case of multiprocessing, the main approach is to partition the population, so that each processor evolves a subpopulation. When there is no communication among processors, this is equivalent to the general Multistart method described above. This is one of three possible migration strategies, as introduced in [12]; when communication is allowed, the best solutions from each subpopulation are transferred to other subpopulations, which is called migration. The other two migration strategies are allowing communication among all nodes, which incurs severe overhead, and allowing each processor communication only with a subset of all the processors, typically exploiting neighbourhood in the network topology; this corresponds to the Islands model mentioned before, and its main drawback is the limited propagation rate of good solutions.

This type of parallelization strategy improves on the sequential approach in the following aspects [13]:

- The selection of individuals is local to the subpopulation, which requires less computation compared to selecting from the whole population.
- Each subpopulation can progress asynchronously, reducing the synchronization overhead.
- The algorithm is more robust, as performance of each processor is independent from the others.

See [14] for further references and examples of applications of parallel Genetic Algorithms following this scheme.

Others approach the parallel strategies for metaheuristics in general. For instance, Salto [15] reviews multiple parallel metaheuristics applied to cutting, packing, and related problems, finding two main classes: master-slave, where the master manages the population and the slaves perform operations on the individuals or subpopulation assigned to them, such as fitness evaluation; and structured population, a generalization of the use of independent subpopulations with varying degrees of communication.

Likewise, Crainic [16] introduces a similar classification into three types, two of which match the ones described above (parallelization of low level computation within an iteration, and multiple subpopulations with varying degrees of granularity and communication); the third type involves partitioning the solution space across the components of the solution variables, with each processor working on a given subset, assuming all other components constant. This usually involves multiple iterations for a suitable exploration of the solution space, and is reminiscent of co-evolution methods. Similarly, Pedemonte et al. [17] present an ACO-specific classification of parallelization strategies, which can be easily mapped to the categories above.

For the evaluation and comparison of parallel metaheuristics, Alba [18] analyzes meaningful metrics and common pitfalls. The design of algorithm assessment methods in this work draws from this analysis to ensure that the results are relevant. In particular, the comparison is based on a predefined effort metric in order to compare solution quality, and it applies a statistical test on the distributions of solutions rather than comparing an aggregated value, such as the mean or the median. Sections 3 and 4 detail the evaluation mechanism.

For the rest of this paper, Section 2 details how to build Multiverse-enabled versions of a GA and ACO for solving the Traveling Salesman Problem (TSP) as a practical demonstration of the method. Both the GA and the ACO are used for experimentally evaluating its performance as compared to Multistart in Section 3. Section 4 provides details on the methodology followed. Finally, Section 5 draws the main conclusions of the work and presents the future work.

2. General Method and Application to a Genetic Algorithm for TSP

Population based metaheuristics follow the common procedure:

1. Create an initial population
2. Evaluate the population
3. Select the individuals that contribute to the next generation
4. Generate the new population by combining and/or modifying the selected individuals
5. Stop or go back to step 2 (according to a stopping criterion)

repeating the loop until a certain condition is met: a number of iterations, a given fitness level, some number of generations without improvement, etc.

Different algorithms define alternative strategies for each of the steps in the procedure, and may in implementation blur the distinctions between them. GA and ACO are examples of the specialization of the general procedure. They can be considered the archetypes for two broad classes of population based metaheuristics: those which build solutions atomically from previous individuals (like GAs) and those that construct new solutions by combining elements or components of previous solutions one at a time (like ACOs). As such archetypes, they are the obvious choice to test the generality of the Multiverse method.

The Multiverse method injects a new step just before closing the loop. In this step, every instance of the metaheuristic sends its current best solution to the controlling node, and it relays them to the collector instance, which adds them to its own population. Due to the very design of this type of metaheuristics, everything else just needs to go on as usual, giving the new procedure:

1. Create an initial population
2. Evaluate the population
3. Select the individuals that contribute to the next generation
4. Generate the new population by combining and/or modifying the selected individuals
5. All but collector: send best solution so far. Collector: receive and add individuals to population
6. Stop or go back to step 2 (according to a stopping criterion)

Since such a generic algorithm is not amenable to testing, this work develops two specific instances of it, a GA and an ACO, both solving a TSP, and compares their performance to that of a corresponding Multistart variant with the same codebase, except for the communication of solutions to the collector. As the aim of this paper is to assess the improvements brought by the Multiverse approach, the base algorithms are far from state-of-the-art. Instead, the focus is on creating a level playing field for both versions, with as little influence from external factors as possible. For this reason, the GA relies on relatively simple crossover and mutation operators among those available in the literature and the ACO uses the standard parametrization, and neither of them performs a local search step. Similarly, there is no comparison between the results and the best known solutions in the literature; doing this would require a much greater development effort in the design and programming of the algorithms, but is irrelevant to the objective.

Listing 1 shows, in pseudocode, the operation of the Multiverse Genetic Algorithm for TSP, which closely mirrors the procedure outlined above. The condition `Collector` is true for the collector node in Multiverse mode; therefore, the code for Multistart is exactly the same, except for the two lines under this condition, as it never happens. This pseudocode corresponds to the nodes that run the GA instances; there is an additional node, the controller, that launches the execution of the instances, keeps the synchronization between them, tracks and consolidates the best solutions, and in the Multiverse case communicates the best solutions of the other instances to the collector.

Listing 1. Pseudocode for Multiverse and Multistart Genetic Algorithm for the Traveling Salesman Problem (TSP).

```
1   Population = EmptyList
2   for i = 1 to PopSize:
3       Population ← RadomPermutation(Cities)
4   Costs = EvaluateSolutions(Population)
5   BestSol, BestCost = min(Population, Costs)
6   SendToController(BestSol, BestCost)
7   while not TerminationCondition:
8       Parents = SelectParents(Population, Costs)
9       Offspring = EmptyList
10      Offspring ← ApplyCrossover(Parents)
11      Offspring ← ApplyMutation(Parents)
12      NewPopulation = EmptyList
13      NewPopulation ← EliteSolutions(Population, Costs)
14      NewPopulation ← Offspring
15      if Collector:
16          ReceiveFromController(MoreSolutions)
17          NewPopulation ← MoreSolutions
18      Population = CutToPopSize(NewPopulation, Costs)
19      Costs = EvaluateSolutions(Population)
20      BestSolIter, BestCostIter = min(Population, Costs)
21      if BestCostIter < BestCost:
22          BestSol, BestCost =  BestSolIter, BestCostIter
23      SendToController(BestSol, BestCost)
```

Lines 1–6 initialize the population with random tours and calculate the costs associated to them, informing the controller. Inside the loop, lines 8–14 generate the new population by including directly the elite individuals, selecting the ones that will act as parents, and applying the crossover and mutation operators on them. Lines 15–17 add the solutions from other instances in the collector. Finally, lines 18–23 trim the solutions down to the population size discarding the ones with worse fitness, update the best solution if needed, and send the current best solution to the controller.

The `SelectParents` function selects a fraction of individuals from the population; this fraction is given by the crossover probability parameter. The selection is random, but the probability for each individual to be selected is inversely proportional to its cost. It deterministically includes the best individual as an elitist strategy. The crossover operator is a variation of order crossover: it randomly selects a subtour of one parent and places it in the same position in the offspring; the rest of the tour is filled up with the remaining cities in the order that they appear in the other parent. The `ApplyCrossover` function repeatedly draws two individuals from the parents group and generates two offspring with them (inverting which is used as first parent). The selection at this stage is again biased by the inverse of the cost, so that better solutions are selected more often. The mutation operator switches the position of two randomly selected cities. The `ApplyMutation` function goes through all the individuals selected as parents and applies the mutation operator with probability given by the mutation probability parameter. Each time it does apply it, it adds a new individual to the population. `EliteSolutions` includes the best individuals from the original population in the new one; the number of individuals to transfer in this way is given by the elitism parameter.

Listing 2 shows the pseudocode for the Multiverse ACO for solving a TSP. The code is the same for the Multistart version, as the only difference is the two lines under the condition `Collector`, which is true only for the collector process in the Multiverse version. As in the case for GA, this pseudocode corresponds to the nodes that run the ACO instances and there is an additional controller node.

Listing 2. Pseudocode for Multiverse and Multistart Ant Colony Optimization (ACO) for TSP.

```
1   PheromMatrix ← InitialValue
2   HeurMatrix ← DistanceMatrix
3   Population = EmptyPopulation
4   while not TerminationCondition:
5     for i = 1 to NumAnts:
6       Solution ← EmptySolution
7       while CitiesLeft:
8         Solution ← GetNext(CitiesLeft, PheromMatrix, HeurMatrix)
9       Population ← Solution
10    Costs = EvaluateSolutions(Population)
11    BestSolIter, BestCostIter = min(Population, Costs)
12    if BestCostIter < BestCost:
13      BestSol, BestCost =  BestSolIter, BestCostIter
14    PheromMatrix ← Evaporate(PheromMatrix)
15    PheromMatrix ← PheromUpdate(BestSol, BestCost)
16    SendToController(BestSol, BestCost)
17    Population = EmptyPopulation
18    if Collector:
19      ReceiveFromController(MoreSolutions)
20      Population ← MoreSolutions
```

Lines 1–3 initialize the algorithm by setting the initial pheromone value and creating the heuristic matrix from the distances between nodes. The main loop, lines 4–20, runs until the stopping criterion is met; each pass of the loop is one iteration in the algorithm, where it generates a new solution for each ant (lines 5–10; the evaluation of solutions is actually performed at the time of construction), updates the current best solution if needed (lines 11–13), updates the pheromone matrix (lines 14–15), and resets the population (line 17). Lines 16, 19 and 20 manage communication with the control node.

In each iteration, each of the `NumAnts` ants builds a tour by iteratively selecting the next city to go to. The function `GetNext` does this selection by assigning a probability to choose each of the cities not yet visited calculated as:

$$\Pr(c, c') = \frac{\tau_{cc'}^{\alpha} \eta_{cc'}^{\beta}}{\sum_{k \in C} \tau_{ck}^{\alpha} \eta_{ck}^{\beta}}, \tag{1}$$

where c is the current city in the tour, c' is the candidate city for the next step, C is the set of all potential candidate cities for the next step, including c'; τ_{ij} is the pheromone level associated to including going from city i to city j in the tour; η_{ij} is the heuristic associated to including going from city i to city j in the tour, namely the inverse of the distance from i to j; and α and β are algorithm parameters.

The update of the pheromone matrix consists of the evaporation step, calculated as $\tau_{ij} = \rho \tau_{ij}$, with the non-negative evaporation parameter $\rho < 1$, and the pheromone addition by the best ant along the tour it built, T:

$$\tau_{ij} = \tau_{ij} + \frac{1}{\sum_{i,j \in T} d_{ij}} \quad \forall i, j \in T, \tag{2}$$

where d_{ij} is the distance from i to j.

3. Experimental Results

The following analysis takes all 18 Asymmetric Traveling Salesman Problem instances in the TSPLIB benchmark library, and runs both the Multiverse and Multistart variants of each GA and ACO on them. Due to the probabilistic nature of the algorithms, it repeats each run 25 times, tracking running time and the best solution found at each iteration. It also records the random seed generated in each run so that the exact same results can be reproduced.

Tables 1 and 2 show the outcome of the analysis for the Genetic Algorithm and the Ant Colony Optimization, respectively. The fields are:

Instance	The file containing the problem data.
Δ Avg	The percent difference between the average across repetitions of final solution fitness for Multistart and Multiverse.
Δ Min	The percent difference between the best solution fitness across repetitions for Multistart and Multiverse.
RTime	The percent difference between the averages across repetitions of CPU time elapsed for Multistart and Multiverse.
WTime	The percent difference between the averages across repetitions of wall time elapsed for Multistart and Multiverse.
Δ HV	The percent difference between the averages across repetitions of the hypervolume Multistart and Multiverse.

Table 1. Results from running Multistart and Multiverse Genetic Algorithm on TSPLIB instances.

Instance	**Δ Avg**	**Δ Min**	**RTime**	**WTime**	**Δ HV**
ftv33.atsp	−4.66	−2.94	−12.3	−10.1	−3.75
rbg403.atsp	−0.67	−0.844	+4.48	−5.54	−0.628
p43.atsp	−0.173	−0.283	−10.9	−9.15	−0.207
rbg443.atsp	−0.668	−0.184	+6.3	+6.19	−0.638
ftv47.atsp	−1.49	+3.34	−10.7	−9.27	−1.95
ft70.atsp	−0.723	−0.143	−7.69	−7.06	−1.16
br17.atsp	−0.102	+0.0	−12.1	−8.51	−0.292
ftv170.atsp	−0.563	−0.304	−3.96	−3.79	−0.593
rbg358.atsp	−1.42	−1.51	+3.27	+3.27	−0.934
ftv44.atsp	−0.988	+0.274	−9.76	−8.28	−1.57
ftv55.atsp	−0.497	−4.71	−9.11	−8.04	−2.25
ftv38.atsp	−3.88	−5.44	−10.9	−9.15	−3.12
ftv35.atsp	−3.67	−3.54	−11.0	−8.68	−3.28
kro124p.atsp	−1.01	−1.78	−5.91	−5.68	−0.969
ftv64.atsp	−0.672	−2.53	−8.17	−7.48	−1.8
ftv70.atsp	−0.399	−3.82	−8.35	−7.67	−0.11
ft53.atsp	−0.733	−1.43	−10.3	−9.08	−1.69
ry48p.atsp	−2.87	−5.96	−10.1	−8.81	−2.82
rbg323.atsp	−1.02	−0.808	+3.14	+3.13	−0.671

Table 2. Results from running Multistart and Multiverse Ant Colony optimization on TSPLIB instances.

Instance	**Δ Avg**	**Δ Min**	**RTime**	**WTime**	**Δ HV**
ftv33.atsp	+0.0	+0.0	−14.3	−5.92	−0.66
rbg403.atsp	−0.0565	−0.121	+1.48	+1.46	−0.237
p43.atsp	−0.037	−0.0178	−0.572	−0.252	−0.00103
rbg443.atsp	−0.109	−0.183	−0.995	−0.954	−0.3
ftv47.atsp	−0.156	+0.0	−0.649	−0.345	−0.119
ft70.atsp	−0.211	−0.234	−2.48	−1.92	−0.0868
br17.atsp	+0.0	+0.0	−1.42	−0.157	+0.00483
ftv170.atsp	−0.932	−0.525	−2.71	−2.67	−0.394
rbg358.atsp	−0.0201	−0.252	+0.596	+0.661	−0.309
ftv44.atsp	−0.052	+0.0	−1.11	−0.564	−0.282
ftv55.atsp	−0.354	+0.0	−1.88	−1.2	−0.447
ftv38.atsp	−0.0366	+0.0	−0.558	−0.225	−0.0138
ftv35.atsp	−0.0895	+0.0	−11.3	−4.89	−0.383
kro124p.atsp	−0.373	−0.966	−2.0	−1.84	−0.244
ftv64.atsp	−0.11	+0.0	−1.14	−0.821	−0.611
ftv70.atsp	−0.364	−0.102	−4.9	−3.97	−0.367
ft53.atsp	−0.642	−0.882	+1.8	+1.13	−0.0366
ry48p.atsp	−0.255	−0.504	−5.56	−6.84	−0.306
rbg323.atsp	+0.0237	−0.0742	−3.21	−3.24	−0.125

All differences are calculated as the value for Multiverse minus value for Multistart; since all the measured items are better the lower they become, this makes negative values favorable to Multiverse. The base for the percentage is the value for Multistart for the instance.

Hypervolume is a measure used to compare solutions to multi-objective problems. In such problems, the outcome is given as a Pareto front of non-dominated solutions, that is, solutions such that there is no other solution that improves on one of the objectives without getting worse on another objective. These solutions are also known as efficient solutions. Pérez Cáceres et al. [19] apply the hypervolume measure to assess anytime performance of ACO algorithms, by considering the evolution of best solution found so far at each iteration as a bi-objective problem which aims at minimizing both the cost and the time to reach a given solution quality. This work includes the hypervolume in the same way as a measure of anytime performance.

Tables 1 and 2 show that the final solution quality is better in average using Multiverse, and that the best solution across runs for most of the instances also is attained by Multiverse. Elapsed times (CPU time and wall time) alternate between positive and negative, which is consistent with negligible overhead of Multiverse with respect to Multistart, as the random effect of synchronization is dominant; otherwise the time differences should be positive if the additional communication had a significant impact. Finally, hypervolume is in average also consistently better for Multiverse.

Instance `br17` is rather small (17 cities) and always yielded the globally optimal solution except in one run of Multistart. The analyses below do not include this instance, as it behaves as an outlier, posing problems for normalization and application of statistical tests. At this size, there is no practical difference between the two methods. For the ACO, instance `ftv33` was also problematic for the same reason in the analysis of solution quality, but had enough variability to be included in the analysis of hypervolume.

Figure 1 shows a boxplot of the best costs achieved in the 25 runs of each instance for Multistart and Multiverse GA, providing a deeper view of the Δ Avg and Δ Min columns in Table 1. Since the values for each instance are very different from each other, the values are normalized as

$$x_{\text{norm}} = \frac{x - \bar{x}}{\bar{x} - x_{\text{min}}}, \tag{3}$$

where x_{norm} is the normalized value for x, $\bar{x}$ is the average value for Multistart, and x_{min} is the minimum value for Multistart. This performs a translation and scaling of both the Multistart and Multiverse values so that the resulting boxes are comparable and fit well in the graph, while maintaining the relative shapes and positions unmodified within each instance. This figure corroborates the typically improved performance of the Multiverse method (with labels ending in `.mv`) over the Multistart method (with labels ending in `.ms`). There are some instances, such as `ftv55` and `ftv64` where the high variability of solutions using Multiverse makes it difficult to reach a conclusion. Running a directed Mann Whitney U test at $\alpha = 0.05$ for each instance, supports the hypothesis that there is a statistical significant difference in solution quality for 10 out of 17 instances in favor of Multiverse, and no significant difference for the other 7. Table 3 shows which instances pass the test.

Figure 2 is the equivalent of Figure 1 for ACO. The findings are similar, with 11 out of 17 significantly improving according to the directed Mann Whitney U test, as shown in Table 4.

Table 3. Instances for which the Mann Whitney U test supports that there is a significant difference between Multiverse and Multistart Genetic Algorithm in favor of Multiverse (MV) or no difference (EQ). There is no instance so that there is a difference in favor of Multistart.

MV	`ftv33`, `rbg403`, `p43`, `rbg443`, `ftv47`, `ft70`, `rbg358`, `ftv38`, `ftv35`, `ry48p`
EQ	`ftv170`, `ftv44`, `ftv55`, `kro124p`, `ftv64`, `ftv70`, `ft53`

Table 4. Instances for which the Mann Whitney U test supports that there is a significant difference in solution quality between Multiverse and Multistart Ant Colony Optimization in favor of Multiverse (MV) or no difference (EQ). There is no instance so that there is a difference in favor of Multistart.

MV	`rbg403`, `p43`, `rbg443`, `ft70`, `ftv170`, `ftv44`, `ftv55`, `ftv38`, `ftv70`, `ft53`, `ry48p`
EQ	`ftv47`, `rbg358`, `ftv35`, `kro124p`, `ftv64`, `rbg323`

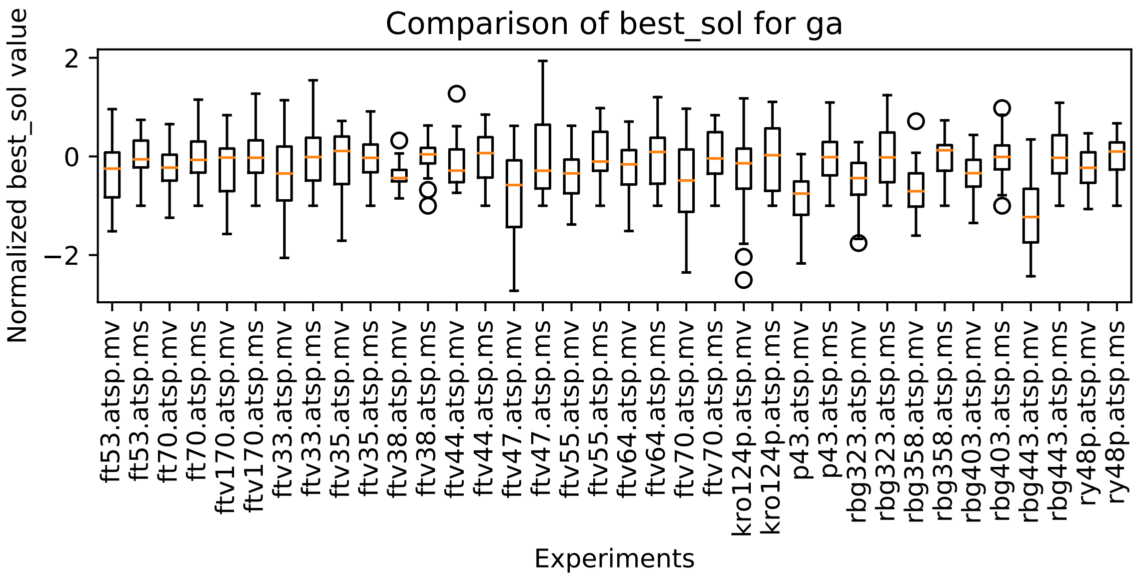

Figure 1. Boxplot of best cost achieved across the 25 runs for each instance problem using Multistart and Multiverse Genetic Algorithm. Costs are normalized to the corresponding average using Multistart so that they are comparable.

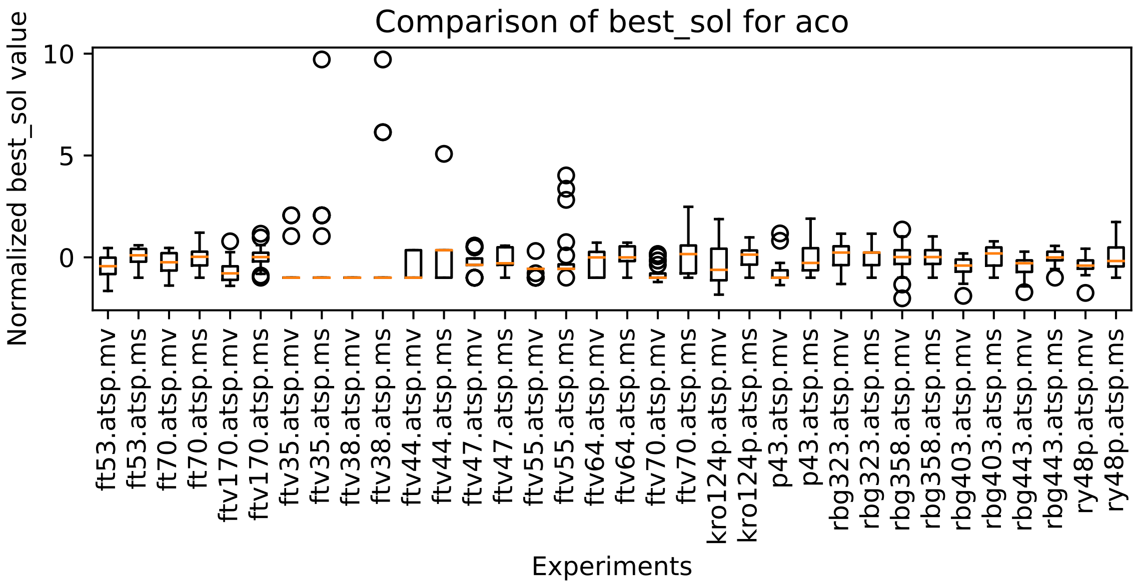

Figure 2. Boxplot of best cost achieved across the 25 runs for each instance problem using Multistart and Multiverse Ant Colony Optimization. Costs are normalized to the corresponding average using Multistart so that they are comparable.

Figures 3 and 4 are the equivalent of Figures 1 and 2, respectively, but for hypervolumes instead of cost achieved. The analysis is similar, but for hypervolume the improvement obtained by Multiverse is more evident. The application of a directed Mann Whitney U test at $\alpha = 0.05$ for each instance, supports the hypothesis that there is a statistical significant difference in solution quality for Genetic Algorithm in 15 out of 17 instances in favor of Multiverse, and no significant difference for the other two: `ftv170` and `ftv70`. For Ant Colony Optimization, the advantage is found in 11 out of 18 instances (see Table 5); larger instances benefit the most from the Multiverse approach.

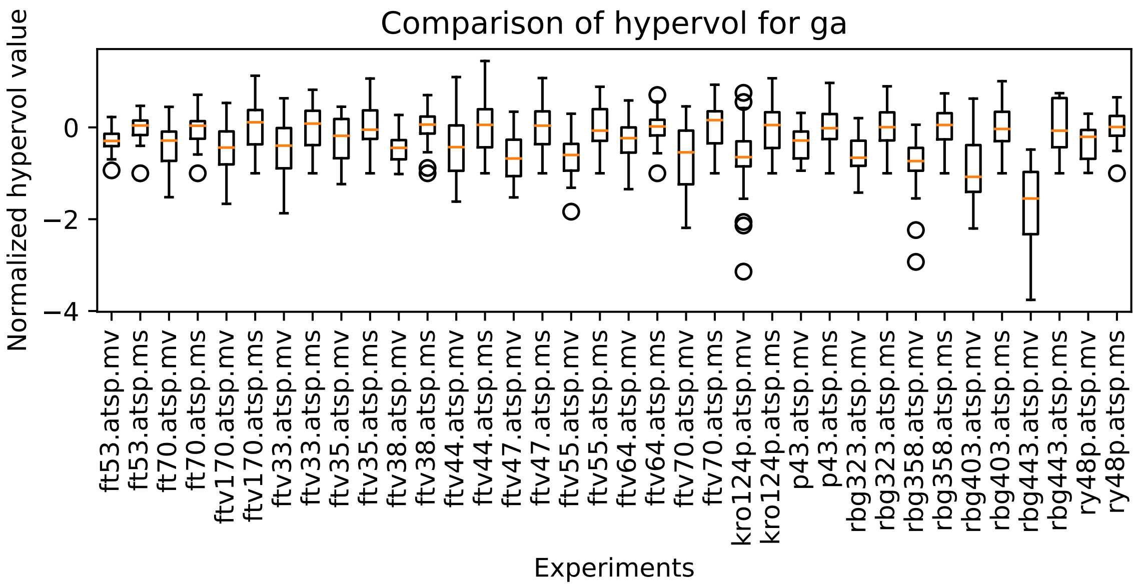

Figure 3. Boxplot of hypervolume across the 25 runs for each instance problem using Multistart and Multiverse GEnetic Algorithm methods. Hypervolumes are normalized to the corresponding average using Multistart so that they are comparable.

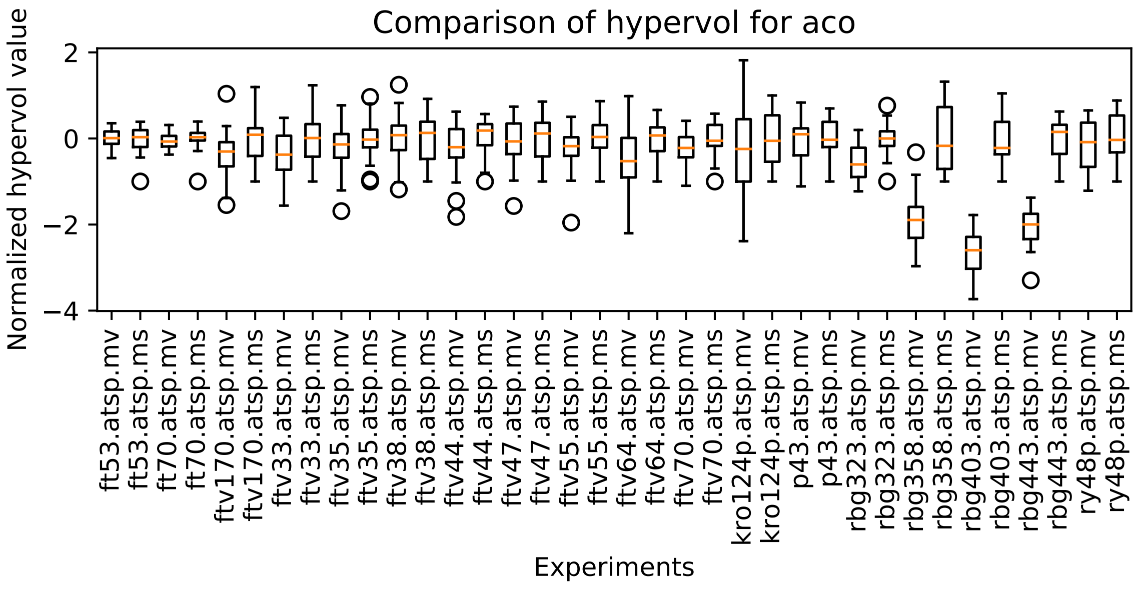

Figure 4. Boxplot of hypervolume across the 25 runs for each instance problem using Multistart and Multiverse Ant Colony Optimziation methods. Hypervolumes are normalized to the corresponding average using Multistart so that they are comparable.

Table 5. Instances for which the Mann Whitney U test supports that there is a significant difference in hypervolume between Multiverse and Multistart Ant Colony Optimization in favor of Multiverse (MV) or no difference (EQ). There is no instance so that there is a difference in favor of Multistart.

MV	`ftv33`, `rbg403`, `rbg443`, `ft70`, `ftv170`, `rbg358`, `ftv44`, `ftv55`, `ftv64`, `ftv70`, `rbg323`
EQ	`p43`, `ftv47`, `ftv38`, `ftv35`, `kro124p`, `ft53`, `ry48p`

Figures 5 and 6 show the typical graph of best cost so far versus iteration number for GA and ACO, respectively. The lines represent the average of best solution achieved in each run by the corresponding iteration for Multiverse and Multistart respectively. The examples selected show a clear gap, and others are similar with varying gap sizes.

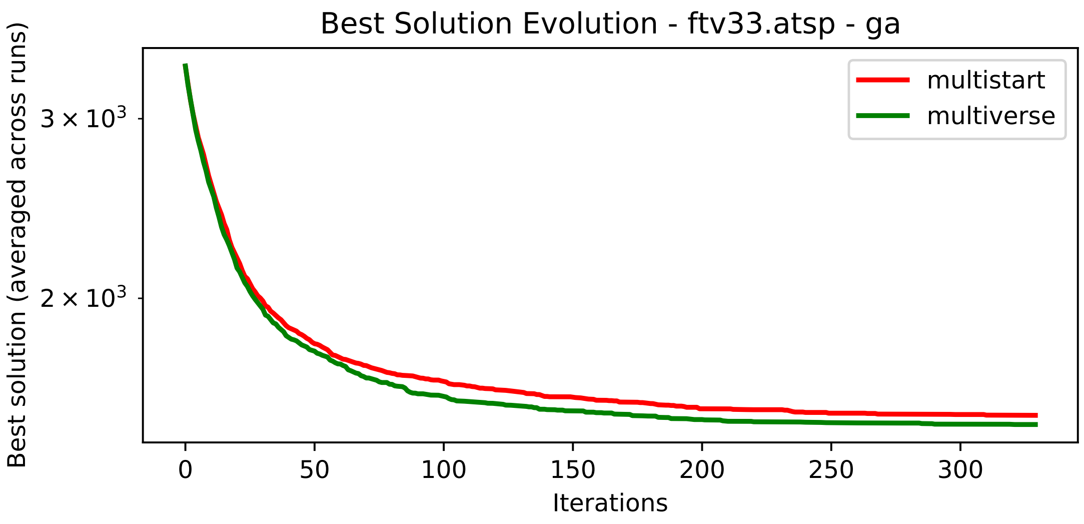

Figure 5. Typical evolution of best Genetic Algorithm solution versus iteration for Multistart and Multiverse. Each line follows the average of the best solutions across the 25 runs at the corresponding iteration for their respective method. The Y axis is shown in logarithmic scale.

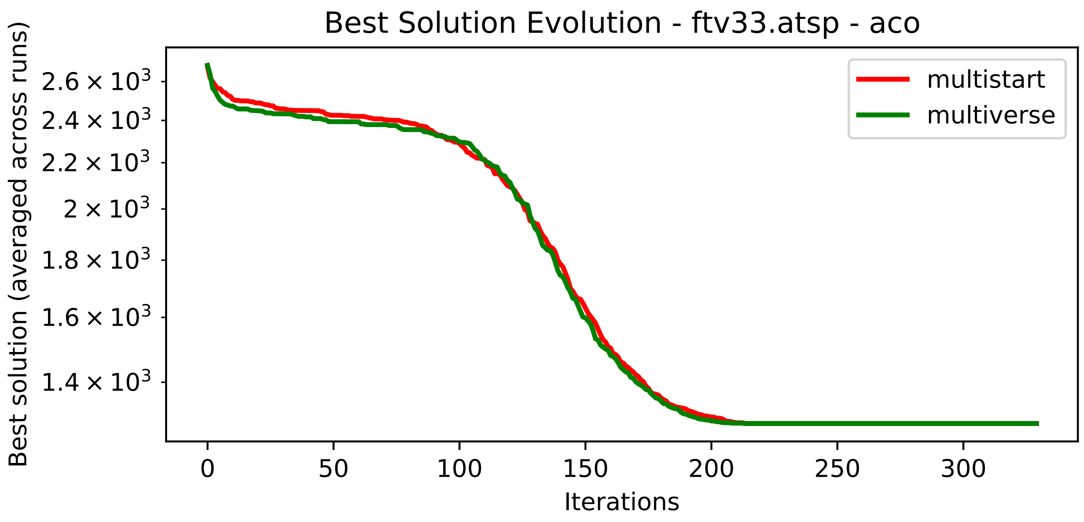

Figure 6. Typical evolution of best Ant Colony Optimization solution versus iteration for Multistart and Multiverse. Each line follows the average of the best solutions across the 25 runs at the corresponding iteration for their respective method. The Y axis is shown in logarithmic scale.

4. Materials and Methods

The hardware setup for the tests described in Section 3 consisted of a cluster of 18 virtual machines with 512 MB of RAM memory running Arch Linux on top of an Intel Xeon E5-2695 (Intel Corporation, Santa Clara, CA, USA) at 2.40 GHz.

The logs containing all the information needed for the analysis are provided as Supplementary Materials to this paper. The program for the tests was written in Python, using MPI for communication across nodes, and is available in GitHub at the URL https://github.com/valthalion/endof. The repository also included the code used to process the results and automatically generated the tables and charts used in the paper.

The overall procedure started from the 18 instances of Asymmetric Traveling Salesman Problem in the benchmark collection TSPLIB. For each of them, four sets of runs were executed: GA Multistart, GA Multiverse, ACO Multistart, and ACO Multiverse. Each set consisted of 25 runs with random seeds. Each run logged to a text file the running time, random seed, final solution, and best solution at each iteration. The logs could be recreated by re-running the experiments using the recorded seeds.

The logs were parsed and loaded into a MySQL database for easier manipulation. The repository contained the Python script for populating the database, as well as the SQL code to generate the database schema. Another script processed the database to generate the aggregated values reported in the tables across this paper, including generating $\LaTeX$ code for some of them, the statistical tests comparing corresponding Multistart and Multiverse instances, and the charts summarizing the results: boxplots of best solution and hypervolume, and the graph of evolution of the objective function with the iterations.

The algorithms used standard parameters, except for the termination condition which was set to a number of iterations equal to 10 times the number of cities. For ACO, evaporation was $\rho = 0.95$, three ants deposited pheromones out of 50 ants per iteration, a single elitist solution was kept, and $\alpha = \beta = 1$. For GA, a population of 50 solutions was used, without elitism, with 50% and 5% crossover and mutation probabilities, respectively. The crossover operator selected a chunk of the first parent given by two cut points, and placed it in the same position in the offspring; it then filled the rest of the offspring with the remaining elements in the order they appear in the second parent. This was done twice, changing the role of first and second parent, to generate two offspring. The mutation operator randomly exchanged two cities in the loop.

5. Conclusions and Future Work

The evolution of hardware leads towards parallelization of algorithms in order to extract the full performance of new systems and so tackle larger and more complex problems. This trend is already evident in the literature, and multiple approaches exist with different trade-offs of solution quality, algorithm complexity, communication overhead, and generalization capability: from problem-specific GPU acceleration to multiple independent runs.

Multiple independent runs, or Multistart, sits at one end of the trade-off spectrum: fully generic, with minimum complexity and overhead, and is typically considered the baseline against which to compare other methods. This work introduces the Multiverse method as an alternative to multiple independent runs. The Multiverse method is also fully generic, in the sense that it is directly applicable to any population-based metaheuristic, and incurs negligible overhead both in terms of added complexity and communication.

In order to test these claims, and to evaluate its performance in solution quality with respect to Multistart, this paper describes concrete instantiations of the method for solving the Traveling Salesman Problem using GA and ACO as representatives of the two broad classes of population-based metaheuristics. The tests run on the asymmetric TSP instances in the benchmark library TSPLIB. The Multiverse variant is often superior in solution quality and never worse, and consistently outperforms the Multistart approach in anytime behaviour, reaching better solutions faster. The tests show statistical significance for these findings, and confirm that the added overhead is smaller than the intrinsic variability in running time for the problems.

These results support the Multiverse method as a potential candidate to replace multiple independent runs as the baseline for distributed algorithm comparison. To further consolidate this outcome, the next step in its development is to test whether these benefits hold for a real-world problem. In particular we will apply it to Ant Colony Optimization solving a line scheduling problem in the steel industry, as described in [20].

Supplementary Materials: The logs of the tests used to generate the results and charts in this paper are available online at http://www.mdpi.com/2227-7390/8/9/1476/s1.

Author Contributions: Conceptualization, D.D. and P.V.; data curation, B.E. and M.I.; investigation, D.D. and P.V.; methodology, D.D. and C.M.; software, D.D.; supervision, D.D.; validation, B.E., M.I. and C.M.; writing—original draft, D.D.; writing—review and editing, B.E., M.I. and C.M. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the Science, Technology and Innovation Plan of the Principality of Asturias (Spain) Ref: FC-GRUPIN-IDI/2018/000225, which is part-funded by the European Regional Development Fund (ERDF).

Acknowledgments: The authors would like to thank ArcelorMittal Global R&D for supplying the equipment for running the experiments, and colleagues Manuel García and Alex Abol for their support in configuring the experimental setup.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gendreau, M.; Potvin, J.Y. (Eds.) *Handbook of Metaheuristics*, 3rd ed.; Number 978-1-4419-1665-5 in International Series in Operations Research and Management Science; Springer Nature Switzerland AG: Cham, Switzerland, 2019. [CrossRef]
2. Sutter, H. *Software and the Concurrency Revolution*; ACM: New York, NY, USA, 2005.
3. Opitz, D.; Maclin, R. Popular ensemble Methods: An Empirical Study. *J. Artif. Intell. Res.* **1999**, *11*, 169–198. [CrossRef]
4. Holland, J.H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*; re-issued 1992; MIT Press: Cambridge, MA, USA, 1975.
5. Reeves, C. Genetic Algorithms. In *Handbook of Metaheuristics*, 2nd ed.; Gendreau, M., Potvin, J.Y., Eds.; Springer Publishing Company, Inc.: Berlin/Heidelberg, Germany, 2010; Chapter 3, pp. 55–82.
6. Dorigo, M. Optimization, Learning and Natural Algorithms. Ph.D. Thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1992. (In Italian)
7. Dorigo, M.; Di Caro, G.; Gambardella, L.M. Ant Algorithms for Discrete Optimization. *Artif. Life* **1999**, *5*, 137–172. [CrossRef] [PubMed]
8. Dorigo, M.; Stützle, T. Ant Colony Optimization: Overview and Recent Advances. In *Handbook of Metaheuristics*; Springer: Cham, Switzerland, 2019; pp. 311–351._10. [CrossRef]
9. Van Luong, T. Parallel Metaheuristics on GPU. Ph.D. Thesis, Université Lille1, Villeneuve-d'Ascq, France, 2011.
10. Krömer, P.; Platos, J.; Snásel, V.; Abraham, A. A comparison of many-threaded differential evolution and genetic algorithms on CUDA. In Proceedings of the 2011 Third World Congress on Nature and Biologically Inspired Computing, Salamanca, Spain, 19–21 October 2011; pp. 509–514.
11. Delévacq, A.; Delisle, P.; Gravel, M.; Krajecki, M. Parallel Ant Colony Optimization on Graphics Processing Units. *J. Parallel Distrib. Comput.* **2013**, *73*, 52–61. [CrossRef]
12. Muhlenbein, H.; Schomisch, M.; Born, J. The Parallel Genetic Algorithm as Function Optimizer. In Proceedings of the 4th International Conference on Genetic Algorithms, San Diego, CA, USA, 13–16 July 1991.
13. Jog, P.; Suh, J.Y.; Van Gucht, D. Parallel Genetic Algorithms Applied to the Traveling Salesman Problem. *SIAM J. Optim.* **1991**, *1*, 515–529. [CrossRef]

14. Ekscanioglu, S.D.; Pardalos, P.M.; Resende, M.G. Parallel Metaheuristics for Combinatorial Optimization. In *Models for Parallel and Distributed Computation*; Corrêa, R., Dutra, I., Fiallos, M., Gomes, F., Eds.; Applied Optimization; Springer: Boston, MA, USA, 2002; Volume 67, pp. 179–206. [CrossRef]
15. Salto, C. Metaheurísticas Híbridas Paralelas para Problemas de Corte, Empaquetado y Otros Relacionados. Ph.D. Thesis, Universidad Nacional de San Luis, San Luis, Argentina, 2009.
16. Crainic, T.; Toulouse, M. Parallel Strategies for Meta-heuristics. In *Handbook of Metaheuristics*; Springer: Boston, MA, USA, 2003; pp. 475–513.
17. Pedemonte, M.; Nesmachnow, S.; Cancela, H. A survey on parallel ant colony optimization. *Appl. Soft Comput.* **2011**, *11*, 5181–5197. [CrossRef]
18. Alba, E.; Luque, G. Evaluation of Parallel Metaheuristics. In Proceedings of the Empirical Methods for the Analysis of Algorithms, Workshop EMAA 2006, Reykjavik, Iceland, 9 September 2006; Paquete, L., Chiarandini, M., Basso, D., Eds. Available online: https://imada.sdu.dk/~marco/EMAA/ (accessed on 31 August 2020).
19. Pérez Cáceres, L.; López-Ibáñez, M.; Stützle, T. Ant Colony Optimization on a Budget of 1000. In Proceedings of the Swarm Intelligence—9th International Conference, ANTS 2014, Brussels, Belgium, 10–12 September 2014; pp. 50–61. [CrossRef]
20. Fernandez, S.; Alvarez, S.; Díaz, D.; Iglesias, M.; Ena, B. Scheduling a Galvanizing Line by Ant Colony Optimization. In Proceedings of the Swarm Intelligence—9th International Conference, ANTS 2014, Brussels, Belgium, 10–12 September 2014; pp. 146–157. [CrossRef]

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).