Manuscript Details

Manuscript number	SWEVO_2019_494_R2
Title	Efficient Parallel and Fast Convergence Chaotic Jaya Algorithms
Short title	Parallel Chaotic Jaya Algorithms
Article type	Full Length Article

Abstract

The Jaya algorithm is a recent heuristic approach for solving optimisation problems. It involves a random search for the global optimum, based on the generation of new individuals using both the best and the worst individuals in the population, thus moving solutions towards the optimum while avoiding the worst current solution. In addition to its performance in terms of optimisation, a lack of control parameters is another significant advantage of this algorithm. However, the number of iterations needed to reach the optimal solution, or close to it, may be very high, and the computational cost can hamper compliance with time requirements. In this work, a chaotic two-dimensional (2D) map is used to accelerate convergence, and parallel algorithms are developed to alleviate the computational cost. Coarse- and fine-grained parallel algorithms are developed, the former based on multi-populations and the latter at the individual level, and in both cases these are accelerated by an improved (computational) use of the chaos map.

Keywords	optimization;jaya algorithm;chaotic map;parallel algorithms;OpenMP
Taxonomy	Distributed Optimization (Algorithms), Parallel Algorithm, Chaos Theory, Heuristics, Optimization (Algorithms)

Corresponding Author	Hector Migallon
Corresponding Author's Institution	Miguel hernandez University
Order of Authors	Hector Migallon, Antonio Jimeno-Morenilla, Jose-Luis Sanchez-Romero, Akram Belazi

Submission Files Included in this PDF

File Name [File Type]

Response_to_reviewers (rev 2).pdf [Response to Reviewers]

SEC_PCJaya.pdf [Manuscript File]

Conflict of Interest.pdf [Conflict of Interest]

CRediT author statement.pdf [Author Statement]

To view all the submission files, including those not included in the PDF, click on the manuscript title on your EVISE Homepage, then click 'Download zip file'.

Research Data Related to this Submission

There are no linked research data sets for this submission. The following reason is given:
No data was used for the research described in the article

Efficient Parallel and Fast Convergence Chaotic Jaya Algorithms

Swarm and Evolutionary Computation

Ref: SWEVO_2019_494_R1

Please find enclosed the second revision of our paper entitled "Efficient Parallel and Fast Convergence Chaotic Jaya Algorithms", co-authored with Antonio Jimeno Morenilla, José Luis Sánchez Romero and Akram Belazi, which was submitted to *Swarm and Evolutionary Computation.*

We want to thank the reviewers for their valuable comments and suggestions, which will improve the quality of the paper. We also hope that the intense work that has been done to address all of these suggestions will be acceptable.

Reviewer 4 Concern 1: *Since the paper is aimed at comparing alternative parallel algorithms in the multi-population case, a more interesting approach would be, for example, the comparison of the Chaotic Jaya algorithm under coarse-grained multi-population (current version) and fine-grained multi-population (diffusion/cellular). I strongly encourage the authors to undertake this kind of study, as it will significantly improve the impact of the paper and its suitability for the journal.*

As suggested by the reviewer, we developed a parallel fine-grained algorithm called DGP-CJaya. This algorithm uses a diffusion grid to select a random individual to generate a new individual, and, logically, to store the population. The new parallel algorithm is explained in Section 3.3. The new algorithm is analysed experimentally, and the numerical results are given in Tables 4, 6, 7, 12 and 13. The numerical results show that this algorithm has parallel performance that is similar to that of CP-CJaya, i.e. significantly worse than the NCP-CJaya algorithm and with limited parallel scalability.

Reviewer 4 Concern 2: *The motivation for the choice of the Jaya algorithm lies in its parameter-less nature. Conceptually, only the population size and the stop criterion are needed by the baseline algorithm. This comes at the expense of requiring the search engine more a priori knowledge about the characteristics of the decision space. In spite of the previous works cited by the authors, I believe this issue might limit the applicability of the method to real-world, hard-to-process decision spaces. In order to clarify this, it seems mandatory the introduction of a real-world case study where the parallel Jaya algorithms are accordingly evaluated.*

As suggested by the reviewer, two real-world problems are solved with the proposed parallel algorithms: the welded beam problem, and the pressure vessel problem. The formulation of both problems is given in Equations (5), (6) and (7). The numerical results, including the parallel performance and the optimal solution obtained, are presented in Tables 22 and 23.

Reviewer 4 Concern 3: *The authors claim that the NCP-CJaya algorithm offers optimal scalability. However, the study on parallel scalability has been conducted on a tiny hardware setup with only 12 cores. Furthermore, the underlying X5660 architecture is 10 years old. Since the paper strongly focuses on the parallel evaluation of the algorithms, it makes sense to examine larger system sizes, in order to confirm if* *the approaches are able to effectively take advantage of current, state-of-the-art multiprocessor systems.*

The proposed algorithms have been analysed using a more recent parallel computing platform, equipped with two Intel Xeon E5-2620 v2 processors. This platform has 12 physical cores, but we can use up to 24 logical cores. The numerical results confirm the parallel behaviour shown by the first parallel platform, and the parallel scalability is even increased. The numerical results are presented in Tables 14 to 19.

Reviewer 4 Concern 4: *Also, some discussions on how the algorithms can benefit from alternative hardware architectures should also be included, taking into account that the authors have previous experience on adapting the Jaya algorithm to many-core GPUs.*

The requested information has been added to the "Conclusions" section as future work.

Reviewer 4 Concern 5: *Some details on the experimental conditions are missing. The authors must report compilation flags, thread binding approaches, and operating system.*

The requested information has been added for the two parallel platforms used. Note that no OpenMP thread binding or affinity approaches have been used, i.e. the operating system performs these tasks.

Reviewer 4 Concern 6: *Tables 1 and 2 share the same caption. They should be named differently, something like:*
- Table 1: Benchmarks, dimensions and domains
- Table 2: Benchmarks, objective functions.

Tables 1 and 2 have been renamed in accordance with the reviewer's suggestion

Reviewer 4 Concern 7: *I believe Table 3 and 4 should be merged. In this way, it will be easier to understand the increment in computational cost introduced by each version of the algorithm.*

Tables 3 and 4 have been merged into a single table (Table 3).

We look forward to hearing from you,
Héctor Migallón

Efficient Parallel and Fast Convergence Chaotic Jaya Algorithms

H. Migallón[a,*], A. Jimeno-Morenilla[b], J.L. Sánchez-Romero[b], A. Belazi[c]

[a] *Department of Computer Engineering, University Miguel Hernández, E-03202, Elche, Alicante, Spain.*
[b] *Department of Computer Technology, University of Alicante, E-03071, Alicante, Spain.*
[c] *RISC Laboratory National Engineering School of Tunis, University of Tunis El Manar, Tunis, Tunisia*

Abstract

The Jaya algorithm is a recent heuristic approach for solving optimisation problems. It involves a random search for the global optimum, based on the generation of new individuals using both the best and the worst individuals in the population, thus moving solutions towards the optimum while avoiding the worst current solution. In addition to its performance in terms of optimisation, a lack of control parameters is another significant advantage of this algorithm. However, the number of iterations needed to reach the optimal solution, or close to it, may be very high, and the computational cost can hamper compliance with time requirements. In this work, a chaotic two-dimensional (2D) map is used to accelerate convergence, and parallel algorithms are developed to alleviate the computational cost. Coarse- and fine-grained parallel algorithms are developed, the former based on multi-populations and the latter at the individual level, and in both cases these are accelerated by an improved (computational) use of the chaos map.

Keywords: optimisation, Jaya algorithm, chaotic map, parallel algorithms, OpenMP

1. Introduction

Optimisation algorithms are used to find the optimal value, or a value as close to this as possible, for a given function called the cost function. Depending on the

*Corresponding author
Email address: hmigallon@umh.com (H. Migallón)

intrinsic characteristics of the cost function, finding this value can be a challenge, and depending on the search pattern used, the optimisation algorithm can be trapped in local optimums. Population-based algorithms, such as the one considered in this work, are also iterative algorithms, and depending on the number of iterations to be performed, the computational cost can increase dramatically.

When deterministic methods are applied to solve an optimisation problem, a sequence of points tending to the optimal value is generated based on the analytical properties of the problem to be solved. In this case, the optimisation problem becomes a problem of linear algebra, i.e. the gradient of the cost function is used in many cases to solve the optimisation problem. The deterministic methods can be used to solve the problems of optimisation of many functions [1], for large-scale problems, and especially non-differentiable, non-convex and nonlinear objective functions. However, they may be powerless to reach acceptable solutions or be invalid for use because of their high computational costs. Several heuristic methods have been proposed to overcome these drawbacks, where the solution obtained is acceptable and the computational cost is reasonable. In most cases, meta-heuristic methods employ guided search techniques in which certain random processes are used to solve the problem. Although it cannot be formally demonstrated that the optimum value obtained is the solution to the problem, they have been shown via experiment to be robust.

In the past few decades, several well-known meta-heuristic optimisation algorithms based on natural phenomena have been proposed: for example, the particle swarm optimisation (PSO) algorithm [2] and its variants are based on the social behaviour of fish schooling or bird flocking; the artificial bee colony (ABC) algorithm [3] was inspired by the foraging behaviour of honey bees; the shuffled frog leaping (SFL) [4] algorithm imitates the collaborative behaviour of frogs; the ant colony optimisation (ACO) algorithm [5] imitates the foraging behaviour of ant colonies; the evolutionary strategy (ES) algorithm [6] is based on the processes of mutation and selection seen in evolution; genetic programming (GP) [7] and evolutionary programming (EP) [8] are techniques for evolving programs based on the selection of individuals for reproduction (crossover) and mutation; the firefly (FF) algorithm [9] was inspired by the flashing behaviour of fireflies; the gravitational search algorithm (GSA) [10] was based on Newtons law of gravity; the biogeography-based optimisation (BBO) algorithm [11] improves solutions stochastically and iteratively; the grenade explosion method (GEM) algorithm [12] is based on the characteristics of the explosion of a grenade; genetic algorithms (GA) [13] and their variants reflect the process of natural selection; the artificial immune algorithm (AIA) [14] is based on the behaviour of the human immune system; differential evolution (DE) [15] and its variants attempt to iteratively improve a candidate solution for a given measure of quality; the simulated annealing (SA) algorithm [16] is based on the annealing process in metallurgy; the tabu search (TS) algorithm [17] employs meta-heuristic local search methods; the teaching-learning-based optimisation (TLBO) algorithm [18] is based on the processes of teaching and learning; and the harmony search algorithm (HSA) [19] was inspired by the process of musical performance.

None of these algorithms are free of limitations in terms of their evolution process, and some of them are easily trapped in local minima. However, a key aspect of the behaviour of these algorithms is the correct adjustment of the control parameters, since the effectiveness of these algorithms depends heavily on the correct setting of these fixed control parameters [18]; for example, PSO needs the cognitive and social parameters and inertia weights to be adjusted; GA needs the crossover probability, mutation probability, selection operator, etc. to be set; the SA algorithm needs the initial annealing temperature and cooling schedule to be tuned; BBO needs the likelihood of habitat modification, mutation probability, habitat elitism parameter and population size to be set; ABC needs the number of bees and limits to be defined; HSA needs the harmony memory consideration rate, the number of improvisations, etc. to be adjusted; and BBO needs the immigration rate, emigration rate, etc. to be set. In contrast, both TLBO and the Jaya optimisation algorithm used in this work can overcome this drawback, since both algorithms only need general parameters to be established, such as the population size and number of iterations (or number of generations) or the stopping criteria.

Jaya algorithms are suitable for solving large-scale industrial problems. For example, in [20], the authors take advantage of the lack of control parameters in these algorithms to optimise the coefficients of proportional-integral controller and the filter parameters of a photovoltaic-fed distributed static compensator, showing that Jaya improves the performance of the TLBO (which also lacks control parameters). In [21], Jaya is modified to efficiently solve the maximum power point tracking problem of photovoltaic systems. In [22], the ABC algorithm is used to solve the welded beam problem, the pressure vessel problem, and the tension-compression spring problem, and is applied in the design of speed reducers and gear trains. Jaya has also been used, for example, to optimise the automatic application of carbon fiber reinforced polymer composites in various production processes [23]; to analyse the effects of the parameters of the submerged arc welding process on the geometry of the weld seam [24]; to optimise sensor placement and damage identification in laminated composite structures [25]; to design a proportional-integral-derivative controller for automatic generation control of an interconnected power system [26]; and to identify the parameters of photovoltaic models used in the simulation, evaluation and control of such systems. It can also be used to solve certain problems that commonly arise in industrial applications, for example those involving generalised sparse non-negative matrix factorisation [27] or matrix factorisation methods, which are applied in [28, 29] large-scale collaborative filtering recommender systems. Parallel optimisation algorithms have been used to optimise the tool path for numerical control machines [30]; to allocate generation and transmission resources in an electricity market [31]; and to control the flow distribution generated by the heliostat field of the receiving system of a solar power plant [32], among other applications.

Two of the most widely used techniques for improving these algorithms are hybridisation and the use of chaos theory. Hybridisation involves the use of more than one search technique to enhance the behaviour of the optimisation process (see, for example, [33, 34, 35, 36, 37]). This hybridisation, as it is logical, usually improves the quality of the solution obtained. However, it may make it difficult to adjust the parameters correctly and can also increase the computational cost. Chaos theory concerns the study of chaotic dynamical systems, which can be defined as nonlinear dynamical systems that are characterised by a high sensitivity to their initial conditions [38, 39]. Many of the algorithms mentioned here make use of randomness in their search patterns, and this randomness can be totally or partially replaced by the use of chaos theory, which offers a powerful technique for hybridisation. Chaos has been used in meta-heuristic algorithms to: (a) replace random number sequences with sequences generated by chaotic maps; (b) perform a local search by means of a chaotic map function; and (c) to generate the control parameters chaotically [40].

Some of the works in which chaos has been successfully used to improve meta-heuristic optimisation algorithms are reported in [41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52].

The main objective of our work is to develop efficient parallel algorithms based on the chaotic C-Jaya algorithm. The aim is for these algorithms to be efficient at the levels of both optimisation behaviour and computational cost. We developed parallel coarse-grained algorithms based on multi-population, a technique previously used in several sequential and parallel proposals (see, for instance, [53, 54, 55, 56]) and parallel fine-grained algorithms based on a diffusion grid (see, for example, [57, 58]). In all cases, the 2D chaotic map used in [59] was applied. These parallel algorithms were computationally improved and adapted to the use of the 2D chaotic map.

The main contributions of this paper are summarised below. Firstly, we analyse the use of the 2D cross chaotic map proposed in the Jaya algorithm, both at the computational level and at optimisation behaviour level. The results show a significant improvement in optimisation performance, but at the expense of a huge increase in the computational complexity during the optimisation process. Secondly, to reduce the global computing time, we develop efficient parallel coarse- and fine-grained algorithms, in which the increase in computational complexity introduced by the use of the 2D chaotic map reduces the intrinsic parallelism that can be exploited, thus reducing parallel scalability. Finally, we modify the pattern of use of the chaotic map to develop more efficient and scalable parallel algorithms that maintain this remarkable improvement in optimisation behaviour.

The remainder of this paper is organised as follows: Section 2 presents a brief description of the Jaya algorithm and the 2D chaotic map used here. In Section 3, the parallel algorithms and the improvements we introduce are explained in detail. In Section 4, we analyse the performance of the proposed parallel algorithms, and finally, in Section 5, concludes the paper.

2. Preliminaries

In this section, we present a description of the Jaya algorithm, and we give the mathematical formula of the 2D chaotic along with its advantages for optimization algorithms.

2.1. Jaya algorithm

The Jaya algorithm was presented in [60], in which the results of optimising both constrained and unconstrained functions are reported. These results show that the Jaya algorithm behaves better than the most common reference algorithms, and a more detailed comparative analysis is presented in [61]. The Jaya algorithm is a population-based algorithm in which the evolution does not depend on function-specific adjustment parameters, and only the size of the population and the maximum number of evaluations needed to be set.

Once the best and worst individuals of the current population have been identified, the basic operation of the Jaya algorithm consists of searching the global optimum, which is achieved by moving toward the best individual and avoiding the worst one. This strategy is implemented by obtaining a new individual following Eq. (1), in which $r_{1,j}$ and $r_{2,j}$ are uniformly distributed random numbers, and j refers to the design variable of the specific cost function to be optimised.

$$x_j^{'} = x_j + r_{1,j}\left(x_{j,best} - |x_j|\right) - r_{2,j}\left(x_{j,worst} - |x_j|\right) \tag{1}$$

Jaya is an iterative algorithm in which Eq. (1) is applied at each iteration to each individual in the current population. If the new individual is better than the old one, it is exchanged; otherwise, it is discarded. Algorithm 1 describes this process. Different executions are performed in this type of algorithm, since the intrinsic randomness of these algorithms may cause poor execution (see line 2). A random initial population is computed (lines 3–8). At each iteration (line 9), after searching for the best and worst individuals in the population (line 10), new individuals are computed. They are inserted into the population as replacements if they are better than the previous ones. The value of each variable for the new individual is trimmed if necessary (lines 16–21).

The use of chaotic maps is proposed in order to increase the diversity of the population, and thus avoid a possible premature convergence at a local minimum and accelerate the convergence. When a chaotic map is employed, the randomness of Eq. (1) (provided by the two random numbers) is replaced by a sequence of numbers that can be generated from a random starting point. In this work, we use the chaotic map reported in [59], and this is briefly described below.

2.2. 2D chaotic map

The 2D chaotic map used here [59] balances the exploitation and exploration phases that are characteristic of heuristic optimisation algorithms. Note that the exploitation phase is related to the convergence ratio, while the exploration phase is related to the algorithms ability to explore different regions in a search space. To balance both phases, the new individuals (cf. Algorithm 1) are obtained by using a random individual (x^{rand}) from the current population, along with the current best and current worst individuals. Moreover, the new individual can be obtained in three different ways, using Eqs. (2), (3) and (4). In these equations, as described above, x^{rand} is a random individual, and $x_{\_,best}$ and $x_{\_,worst}$ are the current best and worst individuals, respectively, of the population. Each $ch_{\_,j}$ is the absolute value of a chaotic variable from the 2D cross chaotic map, and S_F is a scaling factor that takes a value of one or two.

$$x_j^{'} = ch_{1,j} x_j^{rand} + ch_{2,j} \left(x_j - ch_{3,j} x_j^{rand} \right) + ch_{4,j} \left(x_{j,best} - ch_{5,j} x_j^{rand} \right) \tag{2}$$

$$x_j^{'} = ch_{1,j} x_j^{rand} + ch_{2,j} \left(x_j - ch_{3,j} x_j^{rand} \right) + ch_{4,j} \left(x_{j,worst} - ch_{5,j} x_j^{rand} \right) \tag{3}$$

$$x_j^{'} = ch_{1,j} x_{j,best} + ch_{2,j} \left(x_j^{rand} - S_F x_{j,best} \right) \tag{4}$$

The 2D chaotic map can be formulated as shown in Algorithm 2. In the experiments performed here, the initial conditions for generating the 2D chaotic map are $x_1 = 0.2$,

Algorithm 1 Jaya algorithm

Set parameters ($Iterations$ and $PopulationSize$) and define cost function
for $l = 1$ to $Runs$ **do**
  for $i = 1$ to $PopulationSize$ {Create Initial Population X:} **do**
    for $j = 1$ to $VARS$ **do**
      $x_j^i = MinValue + (MaxValue - MinValue) * rand_{[0,1]}$
    end for
    Compute and store $F(x_j^i)$ {Function evaluation}
  end for
  for $l = 1$ to $Iterations$ **do**
    Search for $best$ and $worst$ individuals
    for $i = 1$ to $PopulationSize$ {Create New Population X':} **do**
      for $j = 1$ to $VARS$ **do**
        Obtain 2 random numbers ($rand_{1,2_{[0,1]}}$)
        $x_j^{'i} = x_j^i + rand_{1,j}\left(x_{j,best}^i - \left|x_j^i\right|\right) - rand_{2,j}\left(x_{j,worst}^i - \left|x_j^i\right|\right)$
        { Check the bounds of $x_j^{'i}$ }
        if $x_j^{'i} < MinValue$ **then**
          $x_j^{'i} = MinValue$
        end if
        if $x_j^{'i} > MaxValue$ **then**
          $x_j^{'i} = MaxValue$
        end if
      end for
      Compute $F(x^{'i})$ {Function evaluation}
      if $F(x^{'i}) < F(x^i)$ **then**
        Replace solution in population
      end if
    end for
  end for
end for
Obtain Best Solution and Statistical Data

$y_1 = 0.3$, $k = i$ and $maxDimMap = 500$. The computed values x_i and y_i are in $[-1, 1]$.

Algorithm 2 2D Chaotic map

```
Initialize x_1
Initialize y_1
Initialize maxDimMap
for i = 1 to maxDimMap do
    x_{i+1} = cos(k * arccos(y_i))
    y_{i+1} = 16 x_i^5 - 20 x_i^3 + 5 x_i
end for
```

At each iteration, the chaotic numbers are generated through one of the previous equations, i.e. Eqs. (2), (3) and (4)).

Algorithm 3 Selection of the population update option

Obtain two ordered integer random numbers and one chaotic number:

$a = min$(rnd1,rnd2)

$b = max$(rnd1,rnd2)

ch_j are randomly selected chaotic values

Select between Eqs. (2), (3) or (4):

if $ch_j < a$ **then**

$\quad x_j^{'} = ch_{1,j} x_j^{rand} + ch_{2,j} \left(x_j - ch_{3,j} x_j^{rand} \right) + ch_{4,j} \left(x_{j,best} - ch_{5,j} x_j^{rand} \right)$

end if

if $a < ch_j < b$ **then**

$\quad x_j^{'} = ch_{1,j} x_j^{rand} + ch_{2,j} \left(x_j - ch_{3,j} x_j^{rand} \right) + ch_{4,j} \left(x_{j,worst} - ch_{5,j} x_j^{rand} \right)$

end if

if $ch_j > b$ **then**

$\quad x_j^{'} = ch_{1,j} x_{j,best} + ch_{2,j} \left(x_j^{rand} - S_F x_{j,best} \right)$

end if

It is also worth mentioning that the initial population is not computed as in the original Jaya algorithm (see lines 4–6 of Algorithm 1). It is generated through the 2D chaotic map, as shown in lines 3–9 of Algorithm 4.

3. Proposed parallel algorithms

As mentioned above, parallel coarse- and fine-grained algorithms are developed in this work; the former is based on sub-populations. However, due to the computational characteristics of the chaotic algorithm, it is not possible to obtain high efficiency using the same parallel strategies like those used in [54]. It can be predicted, as discussed in Section 2.2, that the computational cost per iteration will depend on the computational cost of the function to be optimised, the selection of the chaotic values to be used and their extraction from the chaotic map.

A general flowchart of the parallel coarse-grained algorithms is illustrated in Fig. 1. The most important steps and improvements introduced to accelerate the algorithm are detailed below.

First, we analyse the decisions made regarding the efficient use of memory. Our parallel coarse-grained algorithms are based on sub-populations, and all of them divide the individuals of the whole population among the available computing processes, thus creating sub-populations. It is worth noting that these algorithms are executed on a multicore computing platform, i.e., in a shared memory computer architecture. However, each process (or thread) in the initial step copies the sub-population that has been assigned to it to its local private memory. In contrast, the chaotic optimisation algorithm considered here is based on a 2D cross chaotic map, for which the calculation is shown in Algorithm 2. To ensure efficient behaviour of the parallel algorithms, the chaotic map must be stored in global memory, and there is no overhead due to contention in memory writing.

In Algorithm 3, up to three individuals (the best, worst and a random individual) from the current population may be needed to generate a new individual. Since we avoid using extra memory to store the new sub-population, a copy of these individuals must be stored during the generation of the new sub-population, allowing these new individuals to replace others in the same sub-population if they represent improvements. Algorithm 4 shows the computation of both the initial population and the sub-

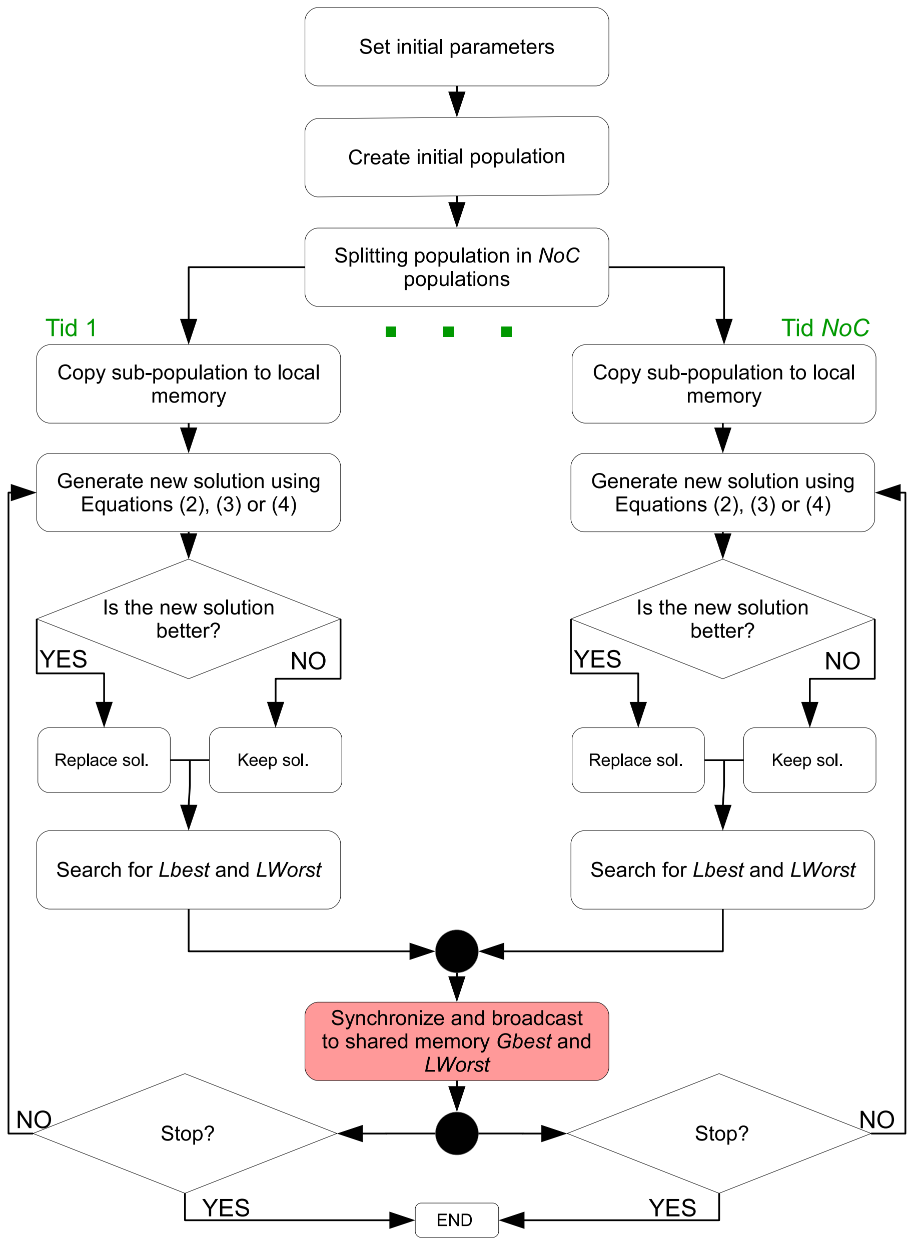

Figure 1: General flowchart of the proposed parallel coarse-grained algorithms.

210 population sizes. Both processes are performed sequentially before the parallel region is spawned.

Algorithm 4 Computing of the initial population and sub-population sizes

```
Set parameters (Iterations and PopulationSize) and define cost function
Set the number of computing processes (NoC)
for i = 1 to PopulationSize do
    for j = 1 to VARS do
        Obtain rnd: random integer value in range [1, maxDimMap]
        x_j^i = MinValue + (MaxValue − MinValue) ∗ ch(rnd)
    end for
    Compute and store F(x^i)
end for
SubPopSize = PopulationSize/NoC
for i = 1 to NoC do
    SubPopSizeArray[i] = SubPopSize
    if i <= (PopulationSize%NoC) then
        SubPopSizeArray[i] + +
    end if
end for
```

The first step in the parallel region is to copy the assigned sub-population to local memory, and then to search for the best and worst individuals in the sub-population. Algorithm 5 shows how these processes are performed within a given parallel region.

Algorithm 5 Copy of sub-population and search for the best and the worst.

```
Inside a parallel region:
Identify thread Tid in range [1, NoC]
MySubPopSize = SubPopSizeArray[Tid]
Allocate memory in private memory for population of size MySubPopSize
IniSubPop = 0
for i = 1 to Tid − 1 do
    IniSubPop+ = SubPopSizeArray[i]
end for
EndSubPop = IniSubPop + SubPopSize
Copy sub-population (IniSubPop − EndSubPop) into private memory
Search for local best (LBest) and local worst (LWorst)
```

3.1. *CP-CJaya parallel algorithm*

Two different parallel coarse-grained strategies for accelerating the optimisation algorithm were developed. In the first one, called the CP-CJaya (Communicated Parallel Chaotic Jaya) algorithm, the different processes share information, and coordination processes between them are therefore necessary. Hence, in this strategy, as can be seen in Algorithm 6, after having searched for the best and worst individuals in each sub-population, all the threads must be coordinated in order to select the global best and global worst individuals. Copies of both individuals are stored in global memory to allow access by all threads. Algorithm 6 includes two synchronisation points (lines 11 and 24), and between these two points there are two critical regions in which the code is executed sequentially by all threads, i.e. with mutual exclusion. As mentioned above, up to three individuals can be used to create a new generation, i.e. in addition to the best and worst, another individual is randomly chosen. In the latter case, in order to avoid increasing the number of communications and coordination processes, it is randomly selected by each thread from the individuals in its sub-population, and is stored in private memory.

The communications and coordination processes in Algorithm 6 cause a loss of parallel efficiency if they are performed at each iteration, and to solve this problem, we include flags to detect whether it is necessary to update the best or worst global individual. As shown in Algorithm 7, if the best global individual is to be updated, it is not necessary to include synchronisation, and only a critical region is needed. However, a synchronisation point is needed in order to check whether the worst global individual needs to be updated, and this point cannot be removed (line 20 of Algorithm 8). If the worst global individual is to be updated, the process includes a search for the worst local individual, two synchronisation points (lines 24 and 30) and a critical region; these procedures are only performed if the flag F_G_GWorst is set.

3.2. NCP-CJaya parallel algorithm

The second coarse-grained strategy, called the NCP-CJaya (Non-Communicated Parallel Chaotic Jaya) algorithm, was developed to remove all synchronisation points. Algorithm 9 shows the NCP-CJaya algorithm, in which the three individuals used to create a new generation are stored in private memory. Hence, no global memory space is used except that used to store the chaotic map, as mentioned above.

In both strategies, CP-CJaya and NCP-CJaya, domain decomposition is implemented in order to develop parallel algorithms. Load balancing is especially essential in the CP-CJaya algorithm, as the synchronisation barriers can lead to high idle times if the load is not correctly balanced. In contrast, NCP-CJaya, which does not include synchronisation barriers, is more versatile and allows load balancing techniques to be applied.

3.3. DGP-CJaya parallel algorithm

The third parallel algorithm presented here, called DGP-CJaya (Diffusion Grid Parallel Chaotic Jaya), was developed following a fine-grained strategy. In order to increase the parallel efficiency, the whole population and the best and worst individuals must be stored in global memory (or shared memory). Here, the random individual

Algorithm 6 CP-CJaya: initial search for the global best and global worst.

Allocate memory in shared memory for $GBest$ (Global Best)
Allocate memory in shared memory for $GWorst$ (Global Worst)
Inside a parallel region:
Identify thread Tid in range $[1, NoC]$
Allocate memory in private memory for x^{rand}
Obtain rnd: random integer value in range $[1, MySubPopSize]$
Copy rnd individual to x^{rand}
Master thread:
  Copy $LBest_{Tid}$ to $GBest$
  Copy $LWorst_{Tid}$ to $GWorst$
Sync Barrier
All threads in parallel CRITICAL region:
{
if $LBest_{Tid}$ is better than $GBest$ **then**
    Copy $LBest_{Tid}$ to $GBest$
end if
}
All threads in parallel CRITICAL region:
{
if $LWorst_{Tid}$ is worse than $GWorst$ **then**
    Copy $LWorst_{Tid}$ to $GWorst$
end if
}
Sync Barrier

Algorithm 7 CP-CJaya: search for the global best based on flags.

Flag to find the worst in shared memory: $F\_G\_GWorst = false$
Inside a parallel region:
Identify thread Tid in range $[1, NoC]$
Flag to find the best in private memory: $F\_P\_Gbest = false$
for $i = 1$ to $SubPopSize$ **do**
    Generate $x^{'i}$
    if $F(x^{'i}) < F(x^i)$ **then**
        Replace solution in population
        if $F(x^{'i}) < F(LBest_{Tid})$ **then**
            Update $LBest_{Tid}$
            if $F(x^{'i}) < F(GBest)$ **then**
                $F\_P\_Gbest = true$
            end if
        end if
        if $i == LWorst_{Tid}$ **then**
            $F\_G\_GWorst = true$
        end if
    end if
end for
if $F\_P\_Gbest == true$ **then**
    CRITICAL region:
    if $F(LBest_{Tid}) < F(GBest)$ **then**
        Copy $LBest_{Tid}$ to $GBest$
    end if
    $F\_P\_GBest = false$
end if

Algorithm 8 CP-CJaya: search for the global worst based on flags.

Flag to find the worst in shared memory: $F\_G\_GWorst = false$
Inside a parallel region:
Identify thread Tid in range $[1, NoC]$
Flag to find the best in private memory: $F\_P\_Gbest = false$
for $i = 1$ to $SubPopSize$ **do**
  Generate $x^{'i}$
  if $F(x^{'i}) < F(x^i)$ **then**
    Replace solution in population
    if $F(x^{'i}) < F(LBest_{Tid})$ **then**
      Update $LBest_{Tid}$
      if $F(x^{'i}) < F(GBest)$ **then**
        $F\_P\_Gbest = true$
      end if
    end if
    if $i == LWorst_{Tid}$ **then**
      $F\_G\_GWorst = true$
    end if
  end if
end for
Sync Barrier
if $F\_G\_GWorst == true$ **then**
  Search for $LWorst_{Tid}$
  SINGLE thread: Copy $LWorst_{Tid}$ to $GWorst$
  Sync Barrier
  CRITICAL region:
  if $F(LWorst_{Tid}) > F(GWorst)$ **then**
    Copy $LWorst_{Tid}$ to $GWorst$
  end if
  SINGLE thread: $F\_G\_GWorst = false$
  Sync Barrier
end if

Algorithm 9 NCP-CJaya: parallel region without synchronizations.

Inside a parallel region:
Allocate memory in private memory for $LBest$ (Local Best)
Allocate memory in private memory for $LWorst$ (Local Worst)
Allocate memory in private memory for x^{rand}
Flag to find the worst in private memory: $F\_P\_LWorst = false$
Flag to find the best in private memory: $F\_P\_LBest = false$
Find and store $LBest$ and $LWorst$
Obtain rnd: random integer value in range $[1, MySubPopSize]$
Copy individual to x^{rand}
for $i = 1$ to $SubPopSize$ **do**
    Generate $x^{'i}$
    if $F(x^{'i}) < F(x^i)$ **then**
        Replace solution in population
        if $F(x^{'i}) < F(LBest_{Tid})$ **then**
            Update $LBest_{Tid}$
            $F\_P\_LBest = true$
        end if
        if $i == LWorst_{Tid}$ **then**
            $F\_P\_LWorst = true$
        end if
    end if
    if $F\_P\_LBest == true$ **then**
        Update $LBest$
        $F\_P\_LBest = false$
    end if
    if $F\_P\_LWorst == true$ **then**
        Find and store $LWorst$
        $F\_P\_LWorst = false$
    end if
end for

used to obtain a new individual (cf. Eqs. (2), (3) and (4)) will be obtained from the current population, and will not be the same for the whole population; in fact, it will be different when obtaining each new individual. In particular, DGP-CJaya was developed using a diffusion grid (see, for example, [62, 63]). In this approach, the whole population is stored in a two-dimensional array, in which each element is an individual of the population. Figure 2 shows the diffusion grid for a population of 120 individuals, organised into a two-dimensional array of 10 rows by 12 columns, where each square represents an individual. To generate a new individual based on the selected individual in Fig. 2, the global best and worst individuals and a random individual are used, where the random individual is chosen from the eight adjacent individuals, as shown in Fig. 2.

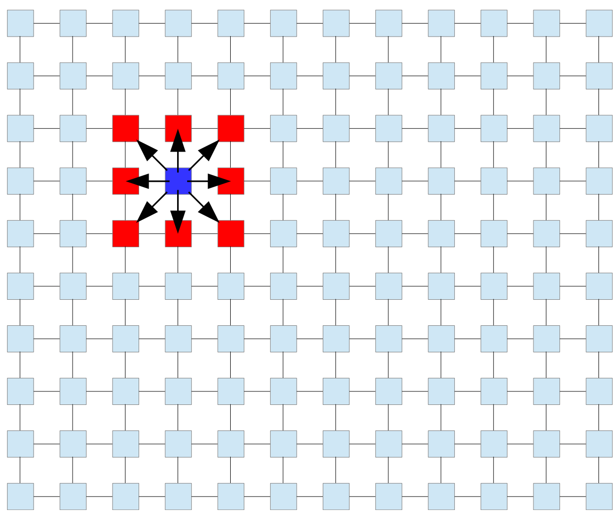

Figure 2: DGP-CJaya diffusion grid for population size equal to 120 (10x12).

Algorithm 10 shows the generation of the initial population using the diffusion grid of individuals (i.e. a cubic structure) for the DGP-CJaya algorithm. The numbers of rows ($PopulationSize\_i$) and columns ($PopulationSize\_j$) are computed depending on the population size to generate a matrix that is as square as possible.

Algorithm 11 shows the processing that is carried out at each iteration of the parallel DGP-CJaya algorithm to obtain a new generation. As mentioned above, the random individual used to obtain a new individual is adjacent to the current individual (see

Algorithm 10 DGP_CJaya: Generation of initial population.

```
for i = 1 to PopulationSize_i do
    for j = 1 to PopulationSize_j do
        for k = 1 to VARS do
            Obtain rnd: random integer value in range [1, maxDimMap]
            x_k^(i,j) = MinValue + (MaxValue − MinValue) ∗ ch(rnd)
        end for
    end for
    Compute and store F(x^(i,j))
end for
```

line 2). The search for the global best or worst individual is optimised via the use of flags stored in global memory (see lines 5 and 9). Since the population (x) has to be stored in global memory, the parallel construct "for" in OpenMP is used (see line 14). The search for the global worst individual in the DGP-CJaya algorithm (see line 29) is similar to that in the CP-CJaya algorithm (see Algorithm 8).

Finally, a symmetric extension of the population structure is created to allow individuals located at the edges of the diffusion grid to access eight adjacent individuals, i.e. the inner individuals. It is noteworthy that the relative position of the element used to obtain the new individual is the same for the whole population at each iteration. Fig. 3 shows such symmetric extension, which is created by copying memory pointers rather than by copying data.

3.4. Improved computing performance (ICP) technique

The use of the 2D chaotic map increases the computational cost of the Chaotic Jaya sequential algorithm compared to the original Jaya sequential algorithm. The selection of the population update option shown in Algorithm 3, leads to computational cost increase and may decrease the parallel performance of the parallel algorithms proposed.

It should be noted that the random numbers a and b used in Algorithm 3 are calculated before each new individual is generated, while the other five random values are obtained to compute of each variable for each new individual. In order to reduce the

Algorithm 11 DGP_CJaya algorithm.

{SM: x, $GBest$, $GWorst$, i_rand, j_rand, i_best, j_best, i_worst, j_worst}
Compute i_rand and j_rand in range $[-1, 1]$ (value $(0, 0)$ not allowed)
Inside a PARALLEL REGION:
SINGLE region {
if $F\_G\_Best == true$ **then**
    $GBest = x^{(i\_best, j\_best)}$
end if
$F\_G\_Best = false$ }
SINGLE region {
if $F\_G\_Worst == true$ **then**
    $GWorst = x^{(i\_worst, j\_worst)}$
end if
$F\_G\_Worst = false$ }
PARALLEL FOR
for $m = 1$ to $PopulationSize\_i * PopulationSize\_j$ **do**
    Obtain i, j from m {Random individual is $x^{(i+i\_rand, j+j\_rand)}$}
    Compute the new individual $x^{'(i,j)}$
    if $F(x^{'(i,j)}) < F(x^{(i,j)})$ **then**
        Replace solution in population
        if $F(x^{'(i,j)}) < F(x^{'(i\_best, j\_best)})$ **then**
            CRITICAL region { $i\_best = i$; $j\_best = j$; $F\_G\_Best = true$}
        end if
    end if
    if $(i == i\_worst) \&\& (j == j\_worst)$ **then**
        ATOMIC region { $F\_G\_Worst = true$ }
    end if
end for
if $F\_G\_Worst == true$ **then**
    IN PARALLEL: Find i_worst and j_worst
end if

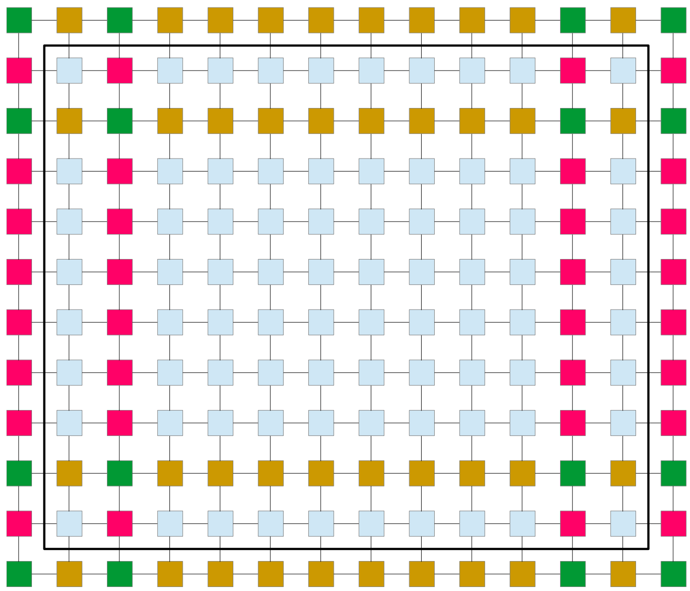

Figure 3: DGP-CJaya symmetric extension of the population structure.

computational cost and improve the parallel efficiency in computing each variable for each new individual, only one new chaotic number is extracted from the chaotic map, and the other four are reused. In this way, only one new random value needs to be obtained for the extraction of only one chaotic value. This modification in the use of the chaotic map is called ICP (Improve Computational Performance), and it is shown in Algorithm 12.

4. Numerical experiments

In this section, the parallel chaotic Jaya algorithms described in Section 3 are analysed in terms of their parallel performance and optimisation behaviour. The reference algorithm presented in [60] and the parallel algorithms proposed here were implemented in the C programming language, using the GCC v.4.8.5 compiler [64], and the flags `std=gnu++0x -fopenmp -O3` were applied in the compilation process. The parallel approaches were designed for shared memory parallel platforms using the OpenMP API v3 [65], no OpenMP thread binding or affinity approaches were used, i.e. the execution environment moved the OpenMP threads between OpenMP

Algorithm 12 Improved computing performance (ICP) applied to the selection of the population update option

Obtain two ordered integer random numbers and one chaotic number:
$a = min$(rnd1,rnd2)
$b = max$(rnd1,rnd2)
$\{$Initially ch_j are randomly selected chaotic values$\}$
for $i = 5$ to 2 **do**
    $ch_j = ch_{j-1}$
end for
ch_1 new randomly selected chaotic value
Select one of (2), (3) or (4):
if $ch_j < a$ **then**
    $x_j^{'} = ch_{1,j} x_j^{rand} + ch_{2,j} \left(x_j - ch_{3,j} x_j^{rand} \right) + ch_{4,j} \left(x_{j,best} - ch_{5,j} x_j^{rand} \right)$
end if
if $a < ch_j < b$ **then**
    $x_j^{'} = ch_{1,j} x_j^{rand} + ch_{2,j} \left(x_j - ch_{3,j} x_j^{rand} \right) + ch_{4,j} \left(x_{j,worst} - ch_{5,j} x_j^{rand} \right)$
end if
if $ch_j > b$ **then**
    $x_j^{'} = ch_{1,j} x_{j,best} + ch_{2,j} \left(x_j^{rand} - S_F x_{j,best} \right)$
end if

places (cores). The parallel computing platform used was equipped with two Intel Xeon X5660 processors, each of which contained six processing cores at 2.8 GHz, and hyperthreading was not activated. The operating system was CentOS Linux 7 (kernel 3.10.0-327.36.3.el7.x86_64). All experiments described here were performed on this platform except where otherwise specified. The performance was analysed using 18 unconstrained functions (cf. Tables 1 and 2).

Table 1: Benchmarks, dimensions and domains.

Id.	**Name**	**Dim. (V)**	**Domain (Min, Max)**
F1	Sphere	30	$-100, 100$
F2	SumSquares	30	$-10, 10$
F3	Beale	2	$-4.5, 4.5$
F4	Easom	2	$-100, 100$
F5	Zakharov	10	$-5, 10$
F6	Schwefel problem 1.2	10	$-100, 100$
F7	Rosenbrock	30	$-30, 30$
F8	Branin	2	$x_1 : -5, 10; x_2 : 0, 15$
F9	Bohachevsky_1	2	$-100, 100$
F10	Booth	2	$-10, 10$
F11	Michalewicz_2	2	$0, \pi$
F12	Bohachevsky_2	2	$-100, 100$
F13	Bohachevsky_3	2	$-100, 100$
F14	GoldStein-Price	2	$-2, 2$
F15	Hartman_3	3	$0, 1$
F16	Ackley	30	$-32, 32$
F17	Langermann_2	2	$0, 10$
F18	Langermann_10	10	$0, 10$

Table 2: Benchmarks objective functions.

Id.	Function
F1	$f = \sum_{i=1}^{V} x_i^2$
F2	$f = \sum_{i=1}^{V} i x_i^2$
F3	$f = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2$ $+(2.625 - x_1 + x_1 x_2^3)^2$
F4	$f = -\cos(x_1)\cos(x_2)\exp\left(-(x_1 - \pi)^2 - (x_2 - \pi)^2\right)$
F5	$f = \sum_{i=1}^{V} x_i^2 + \left(\sum_{i=1}^{V} 0.5 i x_i\right)^2 + \left(\sum_{i=1}^{V} 0.5 i x_i\right)^4$
F6	$f = \sum_{i=1}^{V} \left(\sum_{j=1}^{i} x_j\right)^2$
F7	$f = \sum_{i=1}^{V-1} \left(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right)$
F8	$f = \left(x_2 - \frac{5.1}{4\pi^2} x_1^2 + \frac{5}{\pi} x_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos x_1 + 10$
F9	$f = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7$
F10	$f = (x_1 - 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$
F11	$f = -\sum_{i=1}^{2} \sin x_i \left(\sin\left(\frac{i x_i^2}{\pi}\right)\right)^{20}$
F12	$f = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1)\cos(4\pi x_2) + 0.3$
F13	$f = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1 + 4\pi x_2) + 0.3$
F14	$f = \left[1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)\right]$ $\left[30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)\right]$
F15	$f = -\sum_{i=1}^{4} c_i \exp\left[-\sum_{j=1}^{3} a_{ij}(x_j - p_{ij})^2\right]$
F16	$f = -20\exp\left(-0.2\sqrt{\frac{1}{V}\sum_{i=1}^{V} x_i^2}\right) - \exp\left(\frac{1}{V}\sum_{i=1}^{V}\cos(2\pi x_i)\right) + 20 + e$
F17 F18	$f = -\sum_{i=1}^{5} c_i \left[\exp\left(-\frac{1}{\pi}\sum_{j=1}^{V}(x_j - a_{ij})^2\right)\cos\left(\pi\sum_{j=1}^{V}(x_j - a_{ij})^2\right)\right]$

An analysis of the computational costs of both the original Jaya algorithm and the chaotic algorithm was performed first. Table 3 shows the sequential computational times for a population size of 240, where the number of iterations was $50,000$ and the number of independent executions was 30. The computational cost of the chaotic algorithm is generally higher than that of the original algorithm. Also, the sequential implementation (cf. Algorithm 3) has some drawbacks in terms of efficient parallel development, such as the computing of up to seven random numbers (to calculate a and b and to select the chaotic values) and the extraction of up to five values from the chaotic map.

Table 3: Comparison of sequential computational times: Chaotic Jaya vs. Jaya.

	Time (s.)				
	Original Jaya	Chaotic Jaya	Inc. (%)	Chaotic Jaya (ICP)	Inc. (%)
F1	186.3	1227.7	559%	884.9	375%
F2	193.6	1241.1	541%	898.4	364%
F3	45.7	76.1	67%	49.0	7%
F4	43.6	79.6	83%	49.6	14%
F5	123.5	478.8	288%	255.7	107%
F6	353.9	1556.8	340%	1224.3	246%
F7	202.4	625.1	209%	240.7	19%
F8	27.7	59.0	113%	31.0	12%
F9	29.3	52.9	81%	25.7	-12%
F10	16.4	43.7	167%	16.5	1%
F11	92.8	141.3	52%	109.7	18%
F12	26.7	49.7	87%	22.0	-17%
F13	27.6	50.8	84%	22.7	-18%
F14	20.8	46.6	124%	19.5	-6%
F15	76.6	116.6	52%	77.9	2%
F16	162.1	357.1	120%	167.2	3%
F17	154.2	188.6	22%	169.1	10%
F18	239.7	400.2	67%	269.5	12%

The results shown in Table 3 can be used to analyse the sequential algorithm applied to develop the parallel CP-CJaya and NCP-CJaya algorithms. The sequential algorithm used to create the parallel DGP-CJaya algorithm is different from that of CP-CJaya and NCP-CJaya. Since, as seen in Algorithm 11, both the storage structure of the population (diffusion grid) and the selection of the random individual used to generate new individuals differs from those of the original chaotic Jaya algorithm. Also, the ICP technique has been applied to this algorithm. The comparison of sequential computational cost concerning the Jaya algorithm is shown in Table 4.

Table 4: Comparison of sequential computational times: Diffusion grid Chaotic Jaya vs. Jaya.

	Time (s.)				
	Original Jaya	Diffusion grid Chaotic Jaya	Inc. (%)	Diffusion grid CJaya (ICP)	Inc. (%)
F1	186.3	1324.1	611%	939.9	404%
F2	193.6	1339.1	592%	979.2	406%
F3	45.7	78.8	72%	51.8	13%
F4	43.6	82.5	89%	52.7	21%
F5	123.5	458.8	272%	161.7	31%
F6	353.9	1599.8	352%	1195.2	238%
F7	202.4	629.9	211%	243.9	20%
F8	27.7	61.3	122%	34.0	23%
F9	29.3	55.2	89%	28.1	-4%
F10	16.4	46.4	184%	20.4	25%
F11	92.8	146.4	58%	112.0	21%
F12	26.7	51.1	92%	27.4	3%
F13	27.6	50.9	85%	24.8	-10%
F14	20.8	48.7	134%	22.5	8%
F15	76.6	123.2	61%	79.9	4%
F16	162.1	357.3	120%	169.2	4%
F17	154.2	192.9	25%	166.4	8%
F18	239.7	404.4	69%	264.6	10%

The higher computational cost of the chaotic Jaya sequential algorithms compared to the original sequential Jaya algorithm (cf. Tables 3 and 4) is undoubtedly offset by faster convergence. However, the use of the chaotic map was computationally analysed and modified to accelerate the parallel algorithms proposed in Section 3, using the ICP technique.

The results from Tables 3 and 4 show that the version in which the improved computing performance (ICP) technique was included, significantly improved the computation times for the chaotic Jaya sequential algorithm. It can be observed from the tables that the computational cost for both of the chaotic Jaya sequential algorithms is reduced by including the ICP technique. In some cases (the negative values in the rightmost columns of Tables 3 and 4), the chaotic Jaya sequential algorithms with the ICP technique are computationally less expensive than the original sequential Jaya algorithm.

Tables 5 and 6 show the behaviour of the algorithms that include optimisation to improve computational performance (ICP), including the number of function evaluations required to obtain an error of less than $1e-1$. The optimisation algorithm was run ten times, and Tables 5 and 6 show the maximum, minimum and average values of the number of cost function evaluations for a population of size 240. The error for the Rosenbrock function (F7) was $1e+2$, and it can be seen that the behaviour does not differ markedly from one algorithm to another. Slight decreases in optimisation performance are shown in some cases; however, the large reduction in computational cost compensates for these slight decreases (see Tables 3 and 4).

Table 7 shows a significant improvement in the ratio of convergence between the chaotic algorithms and the original Jaya algorithm for the functions that required more iterations. For the functions that required fewer iterations, the improvement was significant in some cases, while in others, the behaviour was similar. A more exhaustive comparative analysis for other algorithms can be found in [59].

In order to analyse the parallel behaviour of the peoposed parallel algorithms, experiments were performed with 30 independent executions and population sizes of 240, 120 and 60. Tables 8 and 9 show the speed-up achieved by CP-CJaya when no improvements were applied and when performance computing improvement was included, re-

Table 5: Comparison of the number of cost function evaluations respect to the version with improved computing performance (ICP). Chaotic Jaya vs Jaya.

	Number of functions evaluations ($error < 10e-1$)					
	Chaotic			Chaotic (ICP)		
	Average	Maximum	Minimum	Average	Maximum	Minimum
F1	5232	6240	3840	5328	6240	4560
F2	4752	5520	4080	4320	6240	3120
F3	552	960	480	552	720	480
F4	2808	4800	720	3264	9120	1920
F5	3216	4320	1680	3096	5520	960
F6	10416	12720	7440	9360	12000	7200
F7 (*)	3912	4560	2880	3936	5280	2880
F8	960	2640	480	1176	3840	480
F9	2376	3600	1680	2880	2160	1200
F10	1656	3120	720	2613	5760	480
F11	1032	2160	480	1224	2640	480
F12	2016	2640	1200	1752	2400	960
F13	1800	2640	960	1512	2160	960
F14	1848	3120	960	2256	3600	1200
F15	672	1200	480	936	2160	480
F16	4920	6240	4080	4488	6000	3360
F17	504	720	480	480	480	480
F18	480	480	480	480	480	480

Table 6: Comparison of the number of cost function evaluations respect to the version with improved computing performance (ICP). Diffusion grid Chaotic Jaya vs Jaya.

	Number of functions evaluations ($error < 10e-1$)					
	Diffusion grid Chaotic			Diffusion grid Chaotic (ICP)		
	Average	Maximum	Minimum	Average	Maximum	Minimum
F1	4392	5040	3840	4533	5520	3840
F2	4187	4800	3120	3990	4560	3120
F3	480	480	480	480	480	480
F4	2064	3120	960	2070	3600	1440
F5	2184	3120	1440	2472	4080	1440
F6	6024	7680	4080	7032	9120	3840
F7 (*)	2976	4080	1680	3336	4080	2400
F8	768	1440	480	1080	4080	480
F9	1173	1680	480	1056	1680	480
F10	1248	2400	480	3624	7680	720
F11	480	480	480	480	480	480
F12	864	1680	480	840	1440	480
F13	960	1200	480	912	1440	480
F14	1560	2400	480	1573	2400	1200
F15	672	1440	480	864	3120	480
F16	3744	4800	2400	3360	4800	1920
F17	528	720	480	507	720	480
F18	480	480	480	480	480	480

Table 7: Comparison of the number of cost function evaluations respect to the original Jaya algorithm.

Number of functions evaluations ($error < 10e-1$)

	Original Jaya	Chaotic Jaya	Chaotic Jaya (ICP)	Diffusion grid CJaya	Diffusion grid CJaya (ICP)
F1	532560	5232	5328	4392	4533
F2	441750	4752	4320	4187	3990
F3	780	552	552	480	480
F4	44580	2808	3264	2064	2070
F5	240960	3216	3096	2184	2472
F6		10416	9360	6024	7032
F7 (*)	644760	3912	3936	2976	3336
F8	690	960	1176	768	1080
F9	8880	2376	2880	1173	1056
F10	1710	1656	2613	1248	3624
F11	810	1032	1224	480	480
F12	7980	2016	1752	864	840
F13	8190	1800	1512	960	912
F14	8220	1848	2256	1560	1573
F15	600	672	936	672	864
F16	293550	4920	4488	3744	3360
F17	510	504	480	528	507
F18	480	480	480	480	480

spectively.

From both tables, it can be seen that excellent acceleration is obtained for the functions of higher computational cost, even with 10 processes, for a population size of 240. As the population size decreases, the speed-up decreases. Note, for example, that for a population of 60 individuals and 10 processes, the sub-population size is only six individuals, which makes the communication and coordination processes more expensive than the computing time.

Since the sequential reference algorithm is the same in all figures, it can be seen that the use of ICP in the CP-CJaya algorithm (Table 9) improves the behaviour of the original parallel CP-CJaya algorithm without the ICP technique (Table 8). Note that less marked improvements are obtained for small sub-populations, for example, for a sub-population size of six (i.e., a population size of 60 using 10 processes).

Tables 10 and 11 show the speed-up achieved by the parallel NCP-CJaya algorithm. The data in both tables were calculated using the same sequential reference algorithm as that used in Tables 8 and 9, i.e. the chaotic Jaya sequential algorithm (without the use of the diffusion grid). From comparisons of Tables 10 and 8, and Tables 11 and 9, it can be observed that the parallel behaviour of the NCP-CJaya algorithm significantly improves the parallel behaviour of CP-CJaya, both without (Tables 8 and 10) and with the ICP technique (Tables 9 and 11). From Tables 10 and 11, it can be seen that the parallel behaviour of the NCP-CJaya algorithm using the ICP technique (Table 11) improves significantly for the case where ICP is not used (Table 10), even for a small sub-population of six (population 60 with 10 threads). As described above, the speed-up results show good behaviour, even for 10 processes and small population sizes. In some cases, super-speed-up is shown, i.e. a value greater than the number of processes. This is due both to the efficiency of the cache memory and, more importantly, to the fact that the costs of the reference algorithm and the parallel algorithm are not the same.

Tables 12 and 13 show the speed-up for the parallel DGP-CJaya algorithm, without and with the use of ICP, respectively. An analysis of the results in these tables shows that the ICP method improves speed-up, in an analogous way to the behaviour demonstrated by the CP-CJaya and NCP-CJaya algorithms. The results in Tables 8, 10 and 12 show similar parallel behaviour of DGP-CJaya concerning CP-CJaya, while

Table 8: Speed-up of the CP-CJaya method without IPC.

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	1.83	5.12	8.18	1.73	4.70	7.22	1.54	3.95	5.82
F2	1.86	5.09	8.21	1.70	4.52	6.92	1.47	3.84	5.59
F3	1.82	3.87	4.52	1.75	3.04	2.92	1.71	2.20	1.84
F4	1.76	4.24	5.19	1.95	3.63	3.47	1.59	2.30	1.99
F5	1.70	4.54	7.26	1.24	3.31	4.69	0.84	2.04	2.64
F6	1.92	5.37	8.61	1.92	5.35	8.29	1.81	4.80	7.21
F7	1.97	5.43	8.27	1.95	5.19	7.30	1.87	4.81	6.13
F8	1.88	4.08	4.62	1.75	2.94	2.68	1.53	1.99	1.59
F9	1.85	3.85	4.35	1.74	2.82	2.49	1.79	1.85	1.40
F10	1.82	3.65	3.81	1.70	2.54	2.09	1.43	1.54	1.20
F11	1.77	3.89	5.00	1.73	3.39	3.79	1.66	2.73	2.66
F12	1.81	3.80	4.16	1.72	2.69	2.33	1.48	1.79	1.37
F13	1.86	3.85	4.24	1.73	2.80	2.38	1.54	1.75	1.36
F14	1.85	3.78	4.02	1.71	2.60	2.19	1.54	1.79	1.33
F15	1.88	4.59	6.08	1.84	3.75	4.13	1.70	3.13	2.94
F16	1.96	5.28	7.87	1.92	4.90	6.66	1.87	4.48	5.44
F17	1.95	5.07	7.13	1.91	4.35	5.10	1.80	3.54	3.62
F18	1.94	5.26	7.94	1.94	4.90	6.73	1.89	4.35	5.30

Table 9: Speed-up of the CP-CJaya method with IPC.

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	2.52	6.62	10.59	2.33	6.10	9.13	2.03	5.07	7.11
F2	2.55	6.76	10.55	2.25	5.89	8.69	1.96	4.83	6.74
F3	2.71	5.19	5.16	2.70	3.73	3.31	2.20	2.48	1.85
F4	2.90	5.68	6.49	2.95	4.70	4.07	2.32	2.77	2.15
F5	2.23	6.06	9.14	1.70	4.19	5.83	1.09	2.44	3.02
F6	2.44	6.84	10.81	2.52	6.66	10.35	2.30	5.96	8.42
F7	4.92	12.13	17.84	4.88	10.91	12.70	4.53	8.89	9.65
F8	3.39	5.99	5.92	3.36	3.77	3.02	2.41	2.31	1.69
F9	3.51	5.96	5.33	3.09	3.51	2.81	2.42	2.14	1.48
F10	3.95	5.70	4.92	3.37	3.30	2.39	2.44	1.88	1.30
F11	2.28	4.82	5.88	2.37	4.12	4.32	2.08	3.17	2.88
F12	3.67	5.84	5.42	3.15	3.40	2.67	2.50	2.02	1.46
F13	3.55	6.01	5.54	3.26	3.67	2.74	2.40	1.95	1.43
F14	3.94	6.01	5.32	3.27	3.46	2.52	2.64	2.07	1.46
F15	2.88	6.53	7.83	2.85	4.95	4.77	2.49	3.78	3.23
F16	4.08	10.09	13.98	3.97	8.66	10.09	3.66	6.92	7.36
F17	2.20	5.82	7.96	2.26	4.81	5.69	2.11	3.84	3.79
F18	2.97	7.75	11.30	2.94	6.96	8.84	2.81	5.84	6.29

Table 10: Speed-up of the NCP-CJaya method without IPC.

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	2.05	6.64	12.66	2.08	8.70	17.65	2.11	8.65	14.29
F2	2.00	6.62	12.71	2.14	8.71	16.85	2.29	8.71	15.61
F3	1.99	5.19	8.63	1.74	5.17	8.40	1.75	5.01	8.30
F4	1.91	5.27	9.28	1.90	5.70	9.28	1.96	5.31	8.68
F5	2.11	7.37	18.07	2.17	8.72	14.77	2.10	7.33	11.97
F6	1.99	5.78	9.92	1.99	7.47	16.76	1.98	10.59	17.60
F7	2.00	5.47	9.29	1.98	5.30	9.55	1.82	5.68	9.39
F8	1.97	5.14	8.55	1.75	5.12	8.49	1.77	5.16	8.43
F9	1.91	5.09	8.37	1.89	5.15	8.53	1.85	5.44	8.93
F10	1.75	5.21	9.09	1.78	5.44	8.80	1.79	5.28	8.53
F11	1.93	5.21	8.59	1.75	4.80	8.53	1.74	5.10	8.48
F12	1.75	4.94	8.07	1.84	5.18	8.55	1.75	5.23	8.05
F13	1.78	5.20	8.53	1.77	5.22	8.48	1.74	5.16	8.59
F14	1.76	5.20	8.82	1.77	5.23	8.56	1.83	5.21	8.37
F15	1.92	5.18	8.67	1.76	5.21	8.63	1.78	5.20	8.92
F16	1.97	5.23	8.71	1.82	5.25	9.03	1.75	5.17	8.63
F17	1.90	5.22	8.69	2.00	5.23	8.94	1.75	5.22	9.13
F18	1.99	5.31	8.71	1.93	5.20	8.65	1.77	5.27	8.76

Table 11: Speed-up of the NCP-CJaya method with IPC.

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	2.96	10.23	18.12	3.22	16.35	42.36	4.52	22.80	32.15
F2	3.01	11.00	20.40	3.30	16.87	43.03	4.39	20.75	31.70
F3	3.08	8.90	9.37	2.73	7.83	12.34	2.70	7.40	11.25
F4	3.61	10.39	9.80	3.48	9.40	14.29	2.81	7.92	12.41
F5	4.01	20.57	21.96	6.17	19.47	32.57	4.39	12.50	18.45
F6	2.53	7.46	11.42	2.70	10.56	36.06	3.25	17.62	27.63
F7	5.23	14.53	15.47	4.94	13.93	20.35	4.91	11.98	17.58
F8	3.84	10.38	10.05	3.30	9.21	14.63	3.29	8.80	13.40
F9	4.04	10.91	11.02	3.54	9.60	15.52	3.30	9.44	14.60
F10	5.11	14.10	11.99	4.59	11.54	17.65	4.07	10.04	14.52
F11	2.63	7.49	8.98	2.64	6.74	10.94	2.49	6.46	10.27
F12	4.33	11.37	13.67	3.77	10.17	16.58	3.64	9.45	14.16
F13	4.11	11.30	11.14	3.72	10.17	16.54	3.61	9.81	14.60
F14	4.62	13.25	11.63	4.44	11.26	17.31	3.98	10.38	15.25
F15	3.07	8.54	8.24	3.08	8.27	13.43	2.63	7.35	11.81
F16	4.34	12.05	11.79	3.79	10.33	16.45	4.05	9.95	15.31
F17	2.38	6.68	7.15	2.33	6.19	9.97	2.06	5.98	9.85
F18	3.03	8.65	9.88	2.97	7.88	12.76	2.89	7.56	12.12

NCP-CJaya shows the best parallel behaviour. On the other hand, the results in Tables 9, 11 and 13 show that NCP-CJaya significantly outperforms CP-CJaya and especially DGP-CJaya, which was designed following a fine-grained strategy. This performance improvement is enhanced by decreasing the sizes of the populations and increasing the number of threads, meaning that DGP-CJaya does not show good parallel scalability.

Table 12: Speed-up of the DGP-CJaya method without IPC.

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	1.84	5.14	8.26	1.70	4.61	7.11	1.53	3.49	3.76
F2	1.84	5.02	8.19	1.76	4.74	7.29	1.63	3.77	4.06
F3	1.80	3.68	4.56	1.63	2.71	2.69	1.51	2.03	1.86
F4	1.79	4.00	5.08	1.60	2.85	3.17	1.55	2.26	1.93
F5	1.99	6.04	9.27	1.71	4.56	6.54	1.84	3.43	3.51
F6	1.94	5.47	8.94	1.95	5.29	8.29	1.99	4.68	4.86
F7	1.95	5.35	8.46	1.94	5.15	7.45	2.05	4.44	4.59
F8	1.74	3.79	4.36	1.62	2.59	2.46	1.72	1.83	1.67
F9	1.82	3.64	4.20	1.61	2.47	2.25	1.46	1.84	1.67
F10	1.68	3.31	3.67	1.53	2.15	1.96	1.34	1.47	1.28
F11	1.75	3.81	4.95	1.65	3.15	3.60	1.64	2.69	2.60
F12	1.81	3.52	4.05	1.57	2.28	2.15	1.34	1.68	1.44
F13	1.86	3.71	4.32	1.60	2.38	2.15	1.47	1.77	1.42
F14	1.75	3.41	3.83	1.52	2.18	2.02	1.35	1.52	1.29
F15	1.82	4.39	5.89	1.75	3.49	3.86	1.72	2.75	2.53
F16	1.97	5.24	8.00	1.90	4.71	6.46	2.01	4.13	4.12
F17	1.90	4.83	7.01	1.83	4.07	5.11	1.92	3.35	3.20
F18	1.95	5.23	8.00	1.88	4.74	6.62	1.99	4.08	4.18

It is shown from Tables 8-13 that both CP-CJaya and NCP-CJaya algorithms behave computationally better than DGP-CJaya, especially for parallel scalability. Note

Table 13: Speed-up of the DGP-CJaya method with IPC.

	Pop. Size: 240			Pop. Size: 120			Pop. Size: 60		
	N. of threads			N. of threads			N. of threads		
	2	6	10	2	6	10	2	6	10
F1	2.52	6.91	11.02	2.39	6.24	9.33	2.02	4.72	5.03
F2	2.44	6.74	10.68	2.33	6.21	9.21	2.24	5.39	5.45
F3	2.58	4.74	5.36	2.53	3.24	3.03	2.05	2.49	2.18
F4	2.59	5.26	6.21	2.49	3.83	3.49	2.11	2.63	2.22
F5	5.41	14.70	19.54	4.56	11.08	12.34	4.32	7.17	6.83
F6	2.54	6.93	11.35	2.60	6.98	10.14	2.61	6.01	6.46
F7	4.79	12.29	18.14	4.68	11.00	14.03	4.75	9.26	8.98
F8	2.96	5.38	5.68	2.64	3.34	2.79	2.10	2.24	1.92
F9	3.08	5.30	5.69	2.68	3.09	2.63	2.01	2.21	1.82
F10	3.29	5.02	5.09	2.77	2.80	2.20	1.98	1.85	1.46
F11	2.22	4.67	5.98	2.17	3.84	4.10	2.04	3.21	2.87
F12	3.30	5.35	5.21	3.25	2.95	2.39	1.92	2.00	1.65
F13	3.39	5.64	5.44	3.30	3.03	2.55	2.13	2.05	1.65
F14	3.24	5.34	4.93	2.87	2.85	2.30	2.06	1.82	1.49
F15	2.69	6.01	7.53	2.69	4.51	4.52	2.32	3.35	2.99
F16	3.96	9.90	14.23	3.88	8.21	9.89	3.78	6.88	6.48
F17	2.19	5.51	7.70	2.13	4.54	5.40	2.09	3.69	3.49
F18	2.93	7.72	11.45	2.91	6.65	8.73	2.90	5.70	5.72

that the reference algorithm is always the same, i.e. the Chaotic Jaya sequential algorithm without a diffusion grid.

The parallel coarse-grained algorithms were then analysed using a more recent platform to investigate parallel scalability behaviour. The second parallel computing platform used was equipped with two Intel Xeon E5-2620 v2 processors, each of which contained six 2.1 GHz processing cores. The operating system was CentOS Linux 6.6 (kernel 2.6.32-504.16.2.el6.x86_64). The GCC v.4.4.7 compiler [64] was used, and the flags `std=gnu++0x -fopenmp -O3` were set in compilation process. No OpenMP thread binding or affinity approaches were applied. Tables 14 and 15 show the speed-up for the CP-CJaya algorithm without and with the use of ICP, respectively, while Tables 16 and 17 show the speed-up for the NCP-CJaya algorithm without and with the use of ICP, respectively. In all cases, it is confirmed that the speed-up is improved and hence the parallel scalability increases when using a newer processor. It is pointed out that in all these tables, the sequential reference algorithm is always the same, i.e. the chaotic Jaya sequential algorithm without a diffusion grid.

We also analysed the parallel scalability using hyper-threading of the processors of an Intel Xeon E5-2620 v2, i.e., employing more than 12 threads and up to 24 threads for the functions with a higher computational cost. Tables 18 and 19 show the speed-up when the number of threads exceeds the number of physical cores for the CP-CJaya and NCP-CJaya algorithms, respectively. As expected, the scalability of CP-CJaya is limited, and it has an average efficiency of 39% using hyperthreading and without ICP, which rises to 53% when ICP is used (Table 18). In contrast, the NCP-CJaya algorithm has outstanding scalability, with an average efficiency of 80% without ICP; this rises to 163% when ICP is used (Table 19). This means that the algorithm obtains super-speed-up values, and as noted above, this is because the sequential reference algorithm does not include the ICP technique.

We can conclude through Tables 14-19 that the parallel performance is similar for both computing platforms, and the parallel scalability increases for the Intel Xeon E5-2620 platform.

Table 20 shows the sequential computational times of chaotic Jaya in terms of the population size. It is found that the computational cost ratio is not proportional to

Table 14: Speed-up of the CP-CJaya method without IPC (Intel Xeon E5-2620).

	Pop. Size: 240				Pop. Size: 120			
	N. of threads				N. of threads			
	2	6	10	12	2	6	10	12
F1	1.80	4.77	7.50	8.59	1.70	4.35	6.52	7.38
F2	1.75	4.66	7.36	8.48	1.70	4.32	6.48	5.44
F3	1.64	3.28	3.42	3.31	1.54	2.49	2.44	2.07
F4	1.81	3.64	4.03	3.79	1.70	2.70	2.67	2.20
F5	1.76	4.46	6.55	6.01	1.33	3.22	4.27	4.47
F6	1.98	4.94	6.70	7.51	1.95	5.21	6.87	9.04
F7	1.78	4.75	5.17	8.26	1.76	3.62	6.34	7.03
F8	1.72	3.28	3.11	2.80	1.58	2.21	1.98	1.64
F9	1.68	3.14	3.06	2.76	1.53	2.17	1.87	1.52
F10	1.71	2.94	2.59	2.26	1.51	1.87	1.60	1.25
F11	1.80	3.82	4.58	4.65	1.72	3.22	3.44	3.22
F12	1.76	3.18	3.03	2.70	1.59	2.11	1.89	1.52
F13	1.70	3.11	2.93	2.56	1.48	2.05	1.81	1.44
F14	1.76	2.95	2.84	2.52	1.50	2.01	1.78	1.50
F15	1.84	4.03	4.73	4.86	1.71	3.29	3.31	2.82
F16	1.84	4.76	6.84	5.04	1.79	4.34	5.47	5.90
F17	1.75	4.27	5.47	5.67	1.71	3.65	4.27	3.97
F18	1.94	5.02	7.15	7.07	1.87	4.56	5.89	6.08

Table 15: Speed-up of the CP-CJaya method with IPC (Intel Xeon E5-2620).

	Pop. Size: 240				Pop. Size: 120			
	N. of threads				N. of threads			
	2	6	10	12	2	6	10	12
F1	2.80	6.49	10.49	12.12	2.64	6.51	7.00	10.01
F2	2.74	6.60	10.45	11.34	2.59	6.40	8.92	9.86
F3	2.51	4.25	4.02	3.64	2.27	3.01	2.62	2.17
F4	2.61	4.74	4.69	4.31	2.35	3.20	2.90	2.47
F5	2.84	6.43	8.86	8.85	2.18	4.39	5.32	4.34
F6	2.75	6.71	11.26	12.45	2.63	6.45	9.96	11.29
F7	4.62	10.89	14.65	16.10	4.44	9.57	11.13	11.72
F8	2.93	4.42	3.82	3.40	2.43	2.74	2.30	1.80
F9	2.99	4.36	3.62	3.13	2.29	2.61	2.06	1.60
F10	3.56	4.21	3.18	2.57	2.83	2.33	1.74	1.38
F11	2.23	4.40	5.04	4.94	2.11	3.59	3.74	3.31
F12	3.36	4.48	3.53	3.01	2.75	2.64	2.02	1.58
F13	3.26	4.38	3.45	3.03	2.69	2.61	2.00	1.74
F14	3.45	4.44	3.35	2.78	2.59	2.56	1.97	1.48
F15	2.77	5.43	5.83	5.62	2.55	4.28	3.77	3.16
F16	3.92	8.97	7.63	11.66	3.75	7.39	8.02	5.16
F17	2.20	5.14	6.39	6.46	2.10	4.27	4.80	4.28
F18	2.80	6.96	9.56	8.12	2.73	6.08	7.40	7.31

Table 16: Speed-up of the NCP-CJaya method without IPC (Intel Xeon E5-2620).

	Pop. Size: 240				Pop. Size: 120			
	N. of threads				N. of threads			
	2	6	10	12	2	6	10	12
F1	2.00	5.88	9.95	11.89	2.00	5.93	9.92	11.92
F2	2.00	5.85	9.94	11.84	1.99	5.90	9.86	11.93
F3	1.96	5.47	9.03	10.80	1.95	5.44	8.98	10.70
F4	1.95	5.46	9.05	10.84	1.93	5.43	9.00	10.75
F5	2.00	5.75	9.94	11.88	1.84	5.96	9.91	11.94
F6	1.96	5.56	9.42	11.77	1.91	5.93	9.98	11.94
F7	1.95	5.52	9.19	11.01	1.97	5.52	9.17	10.99
F8	1.96	5.39	8.86	10.59	1.95	5.36	8.86	10.56
F9	1.92	5.30	8.84	10.56	1.93	5.40	8.83	10.83
F10	1.95	5.48	9.07	10.83	1.95	5.43	8.92	10.63
F11	1.96	5.40	8.91	10.66	1.98	5.40	8.92	10.62
F12	1.96	5.42	8.90	10.64	1.66	5.46	9.02	10.71
F13	1.93	5.41	9.00	10.78	1.94	5.41	8.99	10.81
F14	1.95	5.46	9.04	10.79	1.95	5.43	8.89	10.60
F15	1.95	5.46	9.06	10.86	1.91	5.45	9.06	10.82
F16	1.95	5.49	9.12	10.94	1.95	5.49	9.12	10.92
F17	1.97	5.49	9.11	10.91	1.96	5.48	9.07	10.87
F18	1.96	5.48	9.09	10.85	1.96	5.49	9.13	10.94

Table 17: Speed-up of the NCP-CJaya method with IPC (Intel Xeon E5-2620).

	Pop. Size: 240				Pop. Size: 120			
	N. of threads				N. of threads			
	2	6	10	12	2	6	10	12
F1	3.28	11.00	22.24	28.28	3.58	15.51	32.40	37.46
F2	3.18	10.46	21.00	28.87	3.58	15.21	30.12	35.86
F3	2.97	8.11	12.84	15.61	2.96	7.92	12.65	14.69
F4	2.98	8.33	13.59	16.16	2.92	8.12	13.12	15.54
F5	4.62	20.78	35.86	42.38	5.58	16.25	25.66	30.02
F6	2.75	7.86	13.31	17.97	2.71	10.81	32.28	38.00
F7	5.01	13.67	22.07	26.06	4.90	13.06	20.62	24.11
F8	3.52	9.48	15.39	18.26	3.51	9.20	14.80	17.31
F9	3.78	10.19	16.32	19.28	3.86	9.97	16.12	18.41
F10	5.23	13.82	21.98	25.77	5.11	12.97	19.89	22.92
F11	2.54	6.97	11.44	13.51	2.50	6.90	11.31	12.34
F12	4.29	11.48	18.05	21.56	4.17	11.09	17.79	20.77
F13	4.23	10.92	18.17	21.04	3.98	10.89	17.78	20.15
F14	4.70	12.77	20.52	24.16	4.48	12.16	18.80	21.81
F15	3.14	8.64	14.11	16.77	3.13	8.44	13.60	16.06
F16	4.21	11.60	18.78	22.26	4.18	11.17	17.76	20.83
F17	2.23	6.18	10.28	12.31	2.17	6.17	10.18	12.10
F18	2.92	8.15	13.43	16.00	2.94	8.06	13.13	15.50

Table 18: Speed-up of the CP-CJaya method when using hyper-threading (Intel Xeon E5-2620).

Whitout ICP

	Pop. Size: 240			Pop. Size: 120		
	N. of threads			N. of threads		
	15	20	24	15	20	24
F1	8.33	9.69	11.26	6.99	7.43	8.88
F2	7.03	10.07	9.90	6.00	8.18	8.71
F5	6.76	7.74	6.97	4.09	4.44	4.30
F6	9.47	12.22	13.52	7.79	10.26	11.19
F7	6.84	8.05	8.96	5.93	6.50	6.40
F16	6.38	6.58	7.48	4.99	5.20	4.99
F18	6.86	7.41	7.82	5.22	5.54	5.16

Using ICP

	Pop. Size: 240			Pop. Size: 120		
	N. of threads			N. of threads		
	15	20	24	15	20	24
F1	11.77	13.72	15.23	7.89	10.28	9.42
F2	11.41	13.63	12.99	8.98	10.50	10.46
F5	7.42	9.24	8.73	4.86	5.08	5.04
F6	13.00	15.72	13.38	10.89	11.40	11.69
F7	12.09	14.43	14.72	9.91	7.87	6.85
F16	9.98	8.79	10.06	6.74	6.37	5.98
F18	8.85	9.05	9.42	6.30	6.46	6.20

Table 19: Speed-up of the NCP-CJaya method when using hyper-threading (Intel Xeon E5-2620).

Whitout ICP

	Pop. Size: 240			Pop. Size: 120		
	N. of threads			N. of threads		
	15	20	24	15	20	24
F1	14.85	19.39	22.26	14.30	18.32	21.75
F2	14.83	19.60	23.39	14.48	19.01	22.25
F5	14.82	19.34	23.72	12.32	16.17	15.65
F6	14.88	19.51	23.53	14.77	19.52	23.37
F7	9.65	12.22	14.45	9.53	12.29	14.50
F16	9.17	11.88	14.01	9.01	11.79	13.96
F18	9.91	12.94	15.31	9.82	12.95	15.26

Using ICP

	Pop. Size: 240			Pop. Size: 120		
	N. of threads			N. of threads		
	15	20	24	15	20	24
F1	34.61	45.99	54.26	31.17	39.30	45.09
F2	33.88	43.46	48.22	30.28	37.50	43.15
F5	33.27	42.93	50.00	23.52	29.53	33.96
F6	28.19	47.62	55.35	33.99	42.81	49.41
F7	21.85	27.21	31.67	19.72	24.39	28.00
F16	17.19	21.91	25.76	15.93	20.06	23.14
F18	14.44	18.74	22.23	13.89	18.01	21.22

the size of the population. The computational cost associated with each thread will, therefore, depends on the size of its sub-population.

Table 20: Sequential computational times (s.).

	Population size		
	240	120	60
Cjaya	895.82	394.71	156.81
Increment (respect pop. size = 60)	5.71	2.52	
Increment (respect pop. size = 120)	2.27		
Cjaya (ICP)	1227.48	573.55	234.50
Increment (respect pop. size = 60)	5.23	2.45	
Increment (respect pop. size = 120)	2.14		

It should be noted that in most experiments for the NCP-CJaya algorithm, the speed of convergence increases with the number of processes. This behaviour is related to the fact that in general, the CJaya algorithm behaves better for small populations. It can obtain an optimal value of the function with populations of only six individuals. Table 21 demonstrates this behaviour, and shows the maximum number of iterations needed to obtain an optimal value with a tolerance of $1e-1$.

Finally, we will analyse the behaviour of the proposed parallel algorithms when solving two real-world cases involving constrained engineering design problems. The first is the welded beam problem, whereas the second is the pressure vessel problem. A detailed description of both problems can be found in [22, 66]. The pressure vessel problem is given by Eq. (5), while the welded beam problem is formulated as Eq. (6). Some auxiliary functions and constant values of welded beam problem are detailed in Eq. (7). Herein, F is the cost function and g_i are the constraints.

Table 21: Maximum number of cost function evaluations with improved computing performance (ICP). Tolerance $1e-1$

	Maximum number of functions evaluations		
	Pop. Size = 240	Pop. Size = 120	Pop. Size = 60
F1	6240	3360	1020
F2	6240	2040	900
F3	720	240	120
F4	9120	1800	480
F5	5520	1680	1140
F6	12000	3720	1560
F7 (*)	5280	2520	1200
F8	3840	600	420
F9	2160	720	540
F10	5760	4080	900
F11	2640	360	1140
F12	2400	1200	540
F13	2160	720	300
F14	3600	2760	10680
F15	2160	1320	660
F16	6000	2280	960
F17	480	480	120
F18	480	240	120

Pressure vessel problem:

$$F = 0.6224x_1x_3x_4 + 1.7781x_2x_3^2 + 3.1661x_1^2x_4 + 19.84x_1^2x_3$$

Constraints:

$$\begin{aligned}
&g_1 = -x_1 + 0.0193x_3 \le 0 \\
&g_2 = -x_2 + 0.00954x_3 \le 0 \\
&g_3 = -\pi x_3^2 x_4 - (4/3)\pi x_3^3 + 1296000 \le 0 \\
&g_4 = x_4 - 240 \le 0 \\
&\quad 0.0625 \le x_1, x_2 \le 99 * 0.0625 \\
&\quad 10 \le x_3, x_4 \le 240
\end{aligned} \tag{5}$$

Welded beam problem:

$$F = 1.10471x_1^2x_2 + 0.04811x_3x_4(14.0 + x_2)$$

Constraints:

$$\begin{aligned}
&g_1 = \tau(x) - \tau_{max} \le 0 \\
&g_2 = \sigma(x) - \sigma_{max} \le 0 \\
&g_3 = x_1 - x_4 \le 0 \\
&g_4 = 0.10471x_1^2 + 0.04811x_3x_4(14.0 + x_2)5.0 \le 0 \\
&g_5 = 0.125 - x_1 \le 0 \\
&g_6 = \delta(x) - \delta_{max} \le 0 \\
&g_7 = P(x) - P_c(x) \le 0 \\
&\quad 0.1 \le x_1, x_4 \le 2.0 \\
&\quad 0.1 \le x_2, x_3 \le 10.0
\end{aligned} \tag{6}$$

Auxiliary functions and constant values of welded beam problem:

$$
\begin{aligned}
\tau(x) &= \sqrt{(\tau')^2 + 2\tau'\tau''\frac{x_2}{2R} + (\tau'')^2}; \tau' = \frac{P}{\sqrt{2}x_1x_2}; \tau'' = \frac{MR}{J} \\
M &= P\left(L + \frac{x_2}{2}\right); R = \sqrt{\frac{x_2^2}{4} + \left(\frac{x_1 + x_3}{2}\right)^2} \\
J &= 2\left\{\sqrt{2}x_1x_2\left[\frac{x_2^2}{12} + \left(\frac{x_1 + x_3}{2}\right)^2\right]\right\} \\
\sigma(x) &= \frac{6PL}{x_4x_3^2} \\
\delta(x) &= \frac{4PL^3}{Ex_x^3x_4} \\
P_c(x) &= \frac{4.013E\sqrt{\frac{x_3^2x_4^6}{36}}}{L^2}\left(1 - \frac{x_3}{2L}\sqrt{\frac{E}{4G}}\right) \\
P &= 6000lb; L = 14in; \delta_{max} = 0.25in; E = 30e + 6psi; G = 10e + 6psi \\
\tau_{max} &= 13600psi; \sigma_{max} = 30000psi
\end{aligned}
\tag{7}
$$

Tables 22 and 23 show the results for parallel speed-up and the solutions obtained for the pressure vessel problem and the welded beam problem, respectively, both with and without the ICP technique. These results show the same behaviour described in the previous analysis for the parallel behaviour. For the solution obtained, these results confirm that good solutions are obtained in all cases [22, 66]. Furthermore, neither the use of parallel methods nor the use of the ICP technique worsens the obtained solutions.

5. Conclusions

This paper proposed three parallel algorithms for accelerating the heuristic optimisation algorithm using a chaotic 2D map. For each of these algorithms, a strategy for reducing the computational cost by varying the use of the chaotic map is analysed. These algorithms are analysed in detail at the level of parallel performance and the level of optimisation behaviour. The employing of the chaotic map, besides the parallel

Table 22: Pressure vessel problem. Population size 120.

			Solution				
N. T.	ICP	Speed-up	F_{cost}	x_1	x_2	x_3	x_4
			CP-Cjaya				
2	N	1.81	6109.14294	0.75000	0.37500	38.85843	234.36939
6	N	3.79	6109.01921	0.75000	0.37500	38.85632	234.38143
10	N	4.19	6108.58298	0.75000	0.37500	38.85952	234.33183
2	Y	3.11	6109.74827	0.75000	0.37500	38.85726	234.40986
6	Y	5.46	6110.72962	0.75000	0.37500	38.85540	234.47527
10	Y	5.22	6111.81471	0.75000	0.37500	38.83579	234.69953
			NCP-Cjaya				
2	N	1.86	6109.06710	0.75000	0.37500	38.85667	234.38077
6	N	5.31	6109.00484	0.75000	0.37500	38.85579	234.38530
10	N	9.28	6109.08271	0.75000	0.37500	38.85993	234.35336
2	Y	3.29	6109.39554	0.75000	0.37500	38.85312	234.42803
6	Y	9.73	6109.03636	0.75000	0.37500	38.85752	234.37186
10	Y	15.51	6108.88119	0.75000	0.37500	38.85944	234.34753

Table 23: Welded beam problem. Population size 120.

			Solution				
N. T.	ICP	Speed-up	F_{cost}	x_1	x_2	x_3	x_4
			CP-Cjaya				
2	N	1.91	1.58737	0.16801	4.06735	10.00000	0.16803
6	N	4.44	1.58742	0.16800	4.06759	10.00000	0.16803
10	N	5.39	1.58738	0.16801	4.06837	10.00000	0.16802
2	Y	2.62	1.58773	0.16793	4.07219	10.00000	0.16802
6	Y	5.60	1.58739	0.16800	4.06713	10.00000	0.16804
10	Y	6.27	1.58786	0.16792	4.07282	10.00000	0.16803
			NCP-Cjaya				
2	N	1.93	1.58733	0.16801	4.06832	10.00000	0.16801
6	N	5.36	1.58743	0.16801	4.06872	10.00000	0.16802
10	N	8.67	1.58769	0.16806	4.06769	10.00000	0.16805
2	Y	2.48	1.58773	0.16793	4.07219	10.00000	0.16802
6	Y	7.74	1.58739	0.16800	4.06713	10.00000	0.16804
10	Y	11.66	1.58786	0.16792	4.07282	10.00000	0.16803

computing, enhance the convergence speed. Although the CP-CJaya and DGP-CJaya algorithms have less scalability, the NCP-CJaya algorithm offers optimal parallel scalability. Furthermore, two real problems were studied, and it was found that the proposed parallel algorithms and the Chaotic Jaya algorithm were suitable for solving real-world problems. In future work, we will design hybrid versions at the level of chaotic maps and the number and size of the sub-populations to accelerate convergence without losing parallel efficiency. A parallel version for multiple cores will be designed by using Nvidia GPUs to assign independent executions to different multiprocessors of the GPU. To achieve excellent occupation of the GPU, each CUDA thread will only be assigned one variable of one individual.

Acknowledgments

This research was supported by the Spanish Ministry of Science, Innovation and Universities and the Research State Agency under Grant RTI2018-098156-B-C54 co-financed by FEDER funds, and by the Spanish Ministry of Economy and Competitiveness under Grant TIN2017-89266-R, co-financed by FEDER funds.

References

[1] M.-H. Lin, J.-F. Tsai, C.-S. Yu, A review of deterministic optimization methods in engineering and management, Mathematical Problems in Engineering 2012 (Article ID 756023) (2012) 15. doi:10.1155/2012/756023.

[2] R. Poli, J. Kennedy, T. Blackwell, Particle swarm optimization, Swarm Intelligence 1 (1) (2007) 33–57. doi:10.1007/s11721-007-0002-0.

[3] D. Karaboga, B. Basturk, On the performance of artificial bee colony (abc) algorithm, Appl. Soft Comput. 8 (1) (2008) 687–697. doi:10.1016/j.asoc.2007.05.007.

[4] M. Eusuff, K. Lansey, F. Pasha, Shuffled frog-leaping algorithm: a memetic metaheuristic for discrete optimization, Engineering Optimization 38 (2) (2006) 129–154. doi:10.1080/03052150500384759.

[5] M. Dorigo, G. Di Caro, New ideas in optimization, McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999, Ch. The Ant Colony Optimization Meta-heuristic, pp. 11–32.
URL http://dl.acm.org/citation.cfm?id=329055.329062

[6] H.-P. Schwefel, Evolutionsstrategie und numerische Optimierung, Dr.-Ing. Thesis, Technical University of Berlin, Department of Process Engineering (1975).

[7] J. R. Koza, Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems, Tech. rep., Stanford, CA, USA (1990).

[8] T. Bäck, G. Rudolph, H. P. Schwefel, Evolutionary programming and evolution strategies: Similarities and differences, in: In Proceedings of the Second Annual Conference on Evolutionary Programming, 1997, pp. 11–22.

[9] Y. Xin-She, Firefly algorithm, lvy flights and global optimization, Research and Development in Intelligent Systems XXVI (2009) 209–218doi:10.1007/978-1-84882-983-1_15.

[10] E. Rashedi, H. Nezamabadi-pour, S. Saryazdi, Gsa: A gravitational search algorithm, Information Sciences 179 (13) (2009) 2232 – 2248, special Section on High Order Fuzzy Sets. doi:10.1016/j.ins.2009.03.004.

[11] H. Ma, D. Simon, P. Siarry, Z. Yang, M. Fei, Biogeography-based optimization: A 10-year review, IEEE Transactions on Emerging Topics in Computational Intelligence 1 (5) (2017) 391–407. doi:10.1109/TETCI.2017.2739124.

[12] A. Ahrari, A. A. Atai, Grenade explosion methoda novel tool for optimization of multimodal functions, Applied Soft Computing 10 (4) (2010) 1132 – 1140. doi:10.1016/j.asoc.2009.11.032.

[13] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, MIT Press, 1992.

[14] J. D. Farmer, N. H. Packard, A. S. Perelson, The immune system, adaptation, and machine learning, Phys. D 2 (1-3) (1986) 187–204. doi:10.1016/0167-2789(81)90072-5.

[15] K. V. Price, New ideas in optimization, McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999, Ch. An Introduction to Differential Evolution, pp. 79–108.
URL http://dl.acm.org/citation.cfm?id=329055.329069

[16] L. Ingber, Simulated annealing: Practice versus theory, Mathematical and Computer Modelling 18 (11) (1993) 29 – 57. doi:10.1016/0895-7177(93)90204-C.

[17] F. Glover, Interfaces in computer science and operations research, Springer, Boston, MA, 1997, Ch. Tabu Search and Adaptive Memory Programming Advances, Applications and Challenge, pp. 1–75.
URL http://dl.acm.org/citation.cfm?id=329055.329069

[18] R. V. Rao, V. Savsani, D. Vakharia, Teaching-learning-based optimization: A novel method for constrained mechanical design optimization problems, Computer-Aided Design 43 (3) (2011) 303–315. doi:10.1016/j.cad.2010.12.015.

[19] J. H. Kim, Harmony search algorithm: A unique music-inspired algorithm, Procedia Engineering 154 (2016) 1401 – 1405, 12th International Conference on Hydroinformatics (HIC 2016) - Smart Water for the Future. doi:10.1016/j.proeng.2016.07.510.

[20] S. Mishra, P. K. Ray, Power quality improvement using photovoltaic fed dstatcom based on jaya optimization, IEEE Transactions on Sustainable Energy 7 (4) (2016) 1672–1680. doi:10.1109/TSTE.2016.2570256.

[21] C. Huang, L. Wang, R. S. Yeung, Z. Zhang, H. S. Chung, A. Bensoussan, A prediction model-guided Jaya algorithm for the PV system maximum power point tracking, IEEE Transactions on Sustainable Energy 9 (1) (2018) 45–55. doi:10.1109/TSTE.2017.2714705.

[22] B. Akay, D. Karaboga, Artificial bee colony algorithm for large-scale problems and engineering design optimization, Journal of Intelligent Manufacturing 3 (4) (2010) 1001–1014. doi:10.1007/s10845-010-0393-4.

[23] K. Abhishek, V. R. Kumar, S. Datta, S. S. Mahapatra, Application of jaya algorithm for the optimization of machining performance characteristics during the turning of cfrp (epoxy) composites: comparison with tlbo, ga, and ica, Engineering with Computers (2016) 1–19doi:10.1007/s00366-016-0484-8.

[24] A. Choudhary, M. Kumar, D. R. Unune, Investigating effects of resistance wire heating on aisi 1023 weldment characteristics during asaw, Materials and Manufacturing Processes 33 (7) (2018) 759–769. doi:10.1080/10426914.2017.1415441.

[25] D. Dinh-Cong, H. Dang-Trung, T. Nguyen-Thoi, An efficient approach for optimal sensor placement and damage identification in laminated composite structures, Advances in Engineering Software 119 (2018) 48 – 59. doi:10.1016/j.advengsoft.2018.02.005.

[26] S. P. Singh, T. Prakash, V. Singh, M. G. Babu, Analytic hierarchy process based automatic generation control of multi-area interconnected power system using Jaya algorithm, Engineering Applications of Artificial Intelligence 60 (2017) 35–44. doi:10.1016/j.engappai.2017.01.008.

[27] H. Li, K. Ge Li, J. An, K. Ge Li, An online and scalable model for generalized sparse non-negative matrix factorization in industrial applications on multi-GPU, IEEE Transactions on Industrial Informatics (2019) 1–1doi:10.1109/TII.2019.2896634.

[28] H. Li, K. Li, J. An, K. Li, MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs, IEEE Transactions on Parallel and Distributed Systems 29 (7) (2018) 1530–1544. doi:10.1109/TPDS.2017.2718515.

[29] H. Li, K. Li, J. An, W. Zheng, K. Li, An efficient manifold regularized sparse non-negative matrix factorization model for large-scale recommender systems on GPUs, Information Sciences 496 (2019) 464 – 484. doi:10.1016/j.ins.2018.07.060.

[30] N. Medina-Rodriguez, O. Montiel-Ross, R. Sepulveda, O. Castillo, Tool path optimization for computer numerical control machines based on parallel aco, Engineering Letters 20 (2012) 101–108.

[31] C. C. Columbus, S. P. Simon, A parallel abc for security constrained economic dispatch using shared memory model, in: 2012 International Conference on Power, Signals, Controls and Computation, 2012, pp. 1–6. doi:10.1109/EPSCICON.2012.6175239.

[32] N. C. Cruz, J. L. Redondo, J. D. Álvarez, M. Berenguel, P. M. Ortigosa, A parallel teaching–learning-based optimization procedure for automatic heliostat aiming, The Journal of Supercomputing 73 (1) (2017) 591–606. doi:10.1007/s11227-016-1914-5.
URL https://doi.org/10.1007/s11227-016-1914-5

[33] M. Z. Ali, N. H. Awad, P. N. Suganthan, R. M. Duwairi, R. G. Reynolds, A novel hybrid cultural algorithms framework with trajectory-based search for global numerical optimization, Information Sciences 334-335 (2016) 219 – 249. doi:10.1016/j.ins.2015.11.032.

[34] N. H. Awad, M. Z. Ali, P. N. Suganthan, R. G. Reynolds, Cade: A hybridization of cultural algorithm and differential evolution for numerical optimization, Information Sciences 378 (2017) 215 – 241. doi:10.1016/j.ins.2016.10.039.

[35] Y. Bai, S. Xiao, C. Liu, B. Wang, A hybrid iwo/pso algorithm for pattern synthesis of conformal phased arrays, IEEE Transactions on Antennas and Propagation 61 (4) (2013) 2328–2332. doi:10.1109/TAP.2012.2231936.

[36] M. Ghasemi, S. Ghavidel, S. Rahmani, A. Roosta, H. Falah, A novel hybrid algorithm of imperialist competitive algorithm and teaching learning algorithm for

optimal power flow problem with non-smooth cost functions, Eng. Appl. Artif. Intell. 29 (2014) 54–69. doi:10.1016/j.engappai.2013.11.003.

[37] N. Zhou, A. Zhang, F. Zheng, L. Gong, Novel image compressionencryption hybrid algorithm based on key-controlled measurement matrix in compressive sensing, Optics & Laser Technology 62 (2014) 152 – 160. doi:10.1016/j.optlastec.2014.02.015.

[38] M. Majumdar, T. Mitra, K. Nishimura, Optimization and Chaos, Springer, New York, 2000.

[39] E. Ott, Frontmatter, 2nd Edition, Cambridge University Press, 2002, pp. i–iv.

[40] A. Rezaee Jordehi, A chaotic-based big bangbig crunch algorithm for solving global optimisation problems, Neural Computing and Applications 25 (2014) 1329–1335. doi:10.1007/s00521-014-1613-1.

[41] A. Gandomi, X.-S. Yang, S. Talatahari, A. Alavi, Firefly algorithm with chaos, Communications in Nonlinear Science and Numerical Simulation 18 (1) (2013) 89 – 98. doi:10.1016/j.cnsns.2012.06.009.

[42] S. Gokhale, V. Kale, An application of a tent map initiated chaotic firefly algorithm for optimal overcurrent relay coordination, International Journal of Electrical Power & Energy Systems 78 (2016) 336 – 342. doi:10.1016/j.ijepes.2015.11.087.

[43] Z. S. Ma, Chaotic populations in genetic algorithms, Applied Soft Computing 12 (8) (2012) 2409 – 2424. doi:10.1016/j.asoc.2012.03.001.

[44] X. F. Yan, D. Z. Chen, S. X. Hu, Chaos-genetic algorithms for optimizing the operating conditions based on RBF-PLS model, Computers & Chemical Engineering 27 (10) (2003) 1393 – 1404. doi:10.1016/S0098-1354(03)00074-7.

[45] W.-C. Hong, Traffic flow forecasting by seasonal SVR with chaotic simulated annealing algorithm, Neurocomputing 74 (12) (2011) 2096 – 2107. doi:10.1016/j.neucom.2010.12.032.

[46] J. Mingjun, T. Huanwen, Application of chaos in simulated annealing, Chaos, Solitons & Fractals 21 (4) (2004) 933 – 941. doi:10.1016/j.chaos.2003.12.032.

[47] J. Saremi, S. Mirjalili, A. Lewisn, Biogeography-based optimisation with chaos, Neural Computing and Applications 25 (5) (2014) 1077 – 1097. doi:10.1007/s00521-014-1597-x.

[48] X. Wang, H. Duan, A hybrid biogeography-based optimization algorithm for job shop scheduling problem, Computers & Industrial Engineering 73 (2014) 96 – 114. doi:10.1016/j.cie.2014.04.006.

[49] D. Jia, G. Zheng, M. K. Khan, An effective memetic differential evolution algorithm based on chaotic local search, Information Sciences 181 (15) (2011) 3175 – 3187. doi:10.1016/j.ins.2011.03.018.

[50] C. Peng, H. Sun, J. Guo, G. Liu, Dynamic economic dispatch for wind-thermal power system using a novel bi-population chaotic differential evolution algorithm, International Journal of Electrical Power & Energy Systems 42 (1) (2012) 119 – 126. doi:10.1016/j.ijepes.2012.03.012.

[51] B. Alatas, Chaotic bee colony algorithms for global numerical optimization, Expert Systems with Applications 37 (8) (2010) 5682 – 5687. doi:10.1016/j.eswa.2010.02.042.

[52] S. Gao, C. Vairappan, Y. Wang, Q. Cao, Z. Tang, Gravitational search algorithm combined with chaos for unconstrained numerical optimization, Applied Mathematics and Computation 231 (2014) 48 – 62. doi:10.1016/j.amc.2013.12.175.

[53] R. V. Rao, A. Saroj, A self-adaptive multi-population based Jaya algorithm for engineering optimization, Swarm and Evolutionary Computation 37 (2017) 1 – 26. doi:10.1016/j.swevo.2017.04.008.

[54] H. Migallón, A. Jimeno-Morenilla, J.-L. Sánchez-Romero, H. Rico, R. V. Rao, Multipopulation-based multi-level parallel enhanced Jaya algorithms,

The Journal of Supercomputing 75 (2019) 1697 – 1716. doi:10.1007/s11227-019-02759-z.

[55] P. D. Michailidis, An efficient multi-core implementation of the Jaya optimisation algorithm, International Journal of Parallel, Emergent and Distributed Systems 0 (0) (2017) 1–33. doi:10.1080/17445760.2017.1416387.

[56] A. García-Monzó, H. Migallón, A. Jimeno-Morenilla, J.-L. Sánchez-Romero, H. Rico, R. V. Rao, Efficient subpopulation based parallel TLBO optimization algorithms, Electronics 8 (1). doi:10.3390/electronics8010019.

[57] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Transactions on Evolutionary Computation 6 (5) (2002) 443–462. doi:10.1109/TEVC.2002.800880.

[58] E. Alba, J. M. Troya, A survey of parallel distributed genetic algorithms, Complexity 4 (4) (1999) 31–52. doi:10.1002/(SICI)1099-0526(199903/04)4:4<31::AID-CPLX5>3.0.CO;2-4.

[59] A. Farah, A. Belazi, A novel chaotic Jaya algorithm for unconstrained numerical optimization, Nonlinear Dynamics 93 (2018) 1451 – 1480. doi:10.1007/s11071-018-4271-5.

[60] R. V. Rao, Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems, International Journal of Industrial Engineering Computations 7 (2016) 19–34. doi:10.5267/j.ijiec.2015.8.004.

[61] R. V. Rao, G. Waghmare, A new optimization algorithm for solving complex constrained design optimization problems, Engineering Optimization 49 (1) (2017) 60–83. doi:10.1080/0305215X.2016.1164855.

[62] B. Manderick, P. Spiessens, Fine-grained parallel genetic algorithms, in: Third international conference on Genetic algorithms, 1989, pp. 428–433.

[63] P. Spiessens, B. Manderick, A massively parallel genetic algorithm: implementation and first analysis, in: Fourth international conference on genetic algorithms, 1991, pp. 279–286.

[64] Free Software Foundation, Inc., GCC, the gnu compiler collection, https://www.gnu.org/software/gcc/index.html.

[65] OpenMP Architecture Review Board, OpenMP Application Program Interface, version 3.1, http://www.openmp.org.

[66] H. Garg, A hybrid pso-ga algorithm for constrained optimization problems, Applied Mathematics and Computation 274 (2016) 292 – 305. doi:10.1016/j.amc.2015.11.001.

AUTHOR DECLARATION

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

We confirm that the manuscript has been read and approved by all named authors and that there are no other persons who satisfied the criteria for authorship but are not listed. We further confirm that the order of authors listed in the manuscript has been approved by all of us.

We confirm that we have given due consideration to the protection of intellectual property associated with this work and that there are no impediments to publication, including the timing of publication, with respect to intellectual property. In so doing we confirm that we have followed the regulations of our institutions concerning intellectual property.

We understand that the Corresponding Author is the sole contact for the Editorial process (including Editorial Manager and direct communications with the office). He/she is responsible for communicating with the other authors about progress, submissions of revisions and final approval of proofs. We confirm that we have provided a current, correct email address which is accessible by the Corresponding Author and which has been configured to accept email from hmigallon@umh.es

Signed by the corresponding author on behalf of all authors:

Héctor Migallón (corresponding author)
Antonio Jimeno Morenilla
José Luis Sánchez Romero
Akram Belazi

Signed: Héctor Migallón

CRediT author statement

Héctor Migallón: Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing, Supervision, Project administration, Funding acquisition

Antonio Jimeno-Morenilla: Validation, Investigation, Writing - Original Draft, Writing - Review & Editing, Supervision, Project administration, Funding acquisition

José-Luis: Sánchez-Romero: Validation, Investigation, Writing - Original Draft, Writing - Review & Editing, Supervision

Akram Belazi: Conceptualization, Methodology, Software, Validation, Investigation, Writing - Original Draft, Writing - Review & Editing