The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Multipopulation based multi-level parallel enhanced Jaya algorithms

H. Migallón · A. Jimeno-Morenilla · J.L. Sánchez-Romero · H. Rico · R. V. Rao

Received: date / Accepted: date

Abstract To solve optimization problems, in the field of engineering optimization, an optimal value of a specific function must be found, in a limited time, within a constrained or unconstrained domain. Metaheuristic methods are useful for a wide range of scientific and engineering applications, which accelerate being able to achieve optimal or near-optimal solutions. The metaheuristic method called Jaya has generated growing interest because of its simplicity and efficiency. We present Jaya-based parallel algorithms to efficiently exploit cluster computing platforms (heterogeneous memory platforms). We propose a multilevel parallel algorithm, in which, to exploit distributed memory architectures (or multiprocessors), the outermost layer of the Jaya algorithm is parallelized. Moreover, in internal layers, we exploit shared memory architectures (or multicores) by adding two more levels of parallelization. This

This research was supported by the Spanish Ministry of Economy and Competitiveness under Grant TIN2015-66972-C5-4-R and Grant TIN2017-89266-R, co-financed by FEDER funds.(MINECO/FEDER/UE)

H. Migallón
Department of Physics and Computer Architecture. Miguel Hernández University, E-03202 Elche, Spain.
Tel.: +34-966658390
Fax: +34-966658814
E-mail: hmigallon@umh.es

A. Jimeno-Morenilla
Department of Computer Technology, University of Alicante, E-03071, Alicante, Spain.

J.L. Sánchez-Romero
Department of Computer Technology, University of Alicante, E-03071, Alicante, Spain.

H. Rico
Department of Computer Technology, University of Alicante, E-03071, Alicante, Spain.

R. V. Rao
Sardar Vallabhbhai National Institute of Technology, Surat-395 007, Gujarat State, India.

two-level internal parallel algorithm is based on both a multipopulation structure and an improved heuristic search path relative to the search path of the sequential algorithm. The multilevel parallel algorithm obtains average efficiency values of 84% using up to 120 and 135 processes, and slightly accelerates the convergence with respect to the sequential Jaya algorithm.

Keywords Jaya, optimization, metaheuristic, multipopulation, parallelism, MPI/OpenMP

1 Introduction

Deterministic approaches to solving optimization problems take advantage of the problems' analytical properties to generate a sequence of points that moves, or attempts to move, towards a global optimal solution. These methods are commonly based on the computation of the gradient of the response variables. Deterministic approaches can provide general tools for solving optimization problems to obtain a global or an approximate global optimum (see [12]). Nonetheless, due to the complexity of the problems, deterministic methods may not be able to easily derive a globally optimal solution within a reasonable time frame, and, moreover, these approaches do not guarantee an optimal solution. This may be due to some of these functions having local minima, so finding an absolute value can prove to be very difficult.

Metaheuristic methods are able to both avoid local minima and accelerate convergence, and avoid restrictions in the functions being optimized. Such methods employ guided (random) search techniques to satisfactorily solve the problem, although its adequacy with respect to the target problem cannot be formally proven. In fact, metaheuristic methods are capable of achieving the global or almost global solution without having all the information about the problems that need solving. Metaheuristic algorithms do not use the gradient of the function, which means that the function does not need to be differentiable, as required by classic deterministic optimization, such as gradient descent or quasi-newton methods.

Some of the well-known metaheuristic optimization algorithms are Genetic Algorithms (GA) and its variants, Differential Evolution (DE) and its variants, Particle Swarm Optimization (PSO) and its variants, Simulated Annealing (SA) algorithm, Tabu Search (TS) algorithm, Evolutionary Strategy (ES), Evolutionary Programming (EP), Genetic Programming (GP), Artificial Bee Colony (ABC), Shuffled Frog Leaping (SFL), Ant Colony Optimization (ACO), and the Fire Fly (FF) algorithm. Some algorithms based on phenomena in nature worthy mentioning are Harmony Search (HS), Lion Search (LS), the Gravitational Search Algorithm (GSA), Biogeography-Based Optimization (BBO), and the Grenade Explosion Method (GEM).

Most of these algorithms' success is greatly conditioned by their specific parameters. For example, GA needs crossover probability, mutation probability, selection operator, etc. to be set correctly; SA algorithm needs the initial annealing temperature and cooling schedule to be tuned; PSO's specific parameters are inertia weight and social and cognitive parameters; HSA needs the harmony memory consideration rate, number of improvisations, etc. to be set correctly; and the immigration rate, emigration rate, etc., need to be tuned for BBO.

Two optimization methods were recently proposed, namely Teacher-Learner Based Optimization (TLBO) [24] and Jaya [19,21], to overcome the problem of tuning algorithm-specific parameters. Both optimization algorithms only need general parameters to be set, such as the number of iterations and population size. Interest in the Jaya algorithm is growing in many scientific and engineering fields because of its simplicity and efficiency (see [1,5,8,10,11,14,16,26,29,30], among others, for example). Elitist-Jaya [23], self-Jaya [22] and the quasi-oppositional-based Jaya [20] algorithms are modifications of the Jaya algorithm meaning that it can be applied to more scientific fields. In particular, the self-Jaya algorithm uses the multipopulation technique, which is employed in our parallel proposal. Multipopulation optimization methods are used to improve search diversity by splitting the entire population into groups (subpopulations) and allocating these throughout the search space so that problem changes can be detected effectively.

Some parallel proposals for metaheuristic optimization algorithms can be found in the literature. For example, the authors of [28] implemented the TLBO algorithm on a multicore processor; the OpenMP strategy emulated the sequential TLBO algorithm exactly. A set of ten test functions were evaluated when running the algorithm on a single core architecture, and were then compared in multiprocessors ranging from two to 32 cores. They obtained average speed-up values of $4.9x$ and $6.4x$ with 16 and 32 processors, respectively.

The Parallel Dual Population Genetic Algorithm, presented in [27], is based on the original GA, but the Dual Algorithm adds a reserve population to ensure that premature convergence proper to this kind of algorithm is avoided. The average speed-up values obtained are equal to $1.64x$ using both 16 and 32 processors.

In [2] and [3], a maximum speed-up of almost $2.5x$ was reached using Message Passing Interface (MPI), OpenMP, and hybrid MPI and OpenMP implementations of population-based metaheuristics. [6] presents a parallel implementation of the Ant Colony Optimisation (ABC) algorithm to solve an industrial scheduling problem in an aluminium-casting centre., Maximum speed-up was obtained using eight processors, however, speed-up decreased as the number of processors further increased. The maximum speed-up achieved was equal to $5.94x$ using eight processors, which dropped to $5.45x$ with 16 processors.

In Section 2, the recent Jaya optimization algorithm is presented; in Section 3, the multi-level parallel algorithms and the improvement included in the optimization procedure for the parallel algorithms are described; in Section 4, we analyse the latter, both in terms of parallel performance and optimization behaviour, and some conclusions are drawn in Section 5.

2 The Jaya algorithm

The Jaya algorithm is based on the fact that the optimal solution for a given problem can be obtained by moving towards the best solution while avoiding the worst individual of the set of individuals in the current population. As aforementioned Jaya is an algorithm-specific parameter-less algorithm, i.e. only population size (number of different individuals) and generations (number of iterations) should be configured. Compared with other optimization methods, such as GA, ABC, DE, PSO and TLBO, Jaya obtained better results in terms of best, mean and worst values of different constrained and unconstrained benchmark functions [25].

The Jaya algorithm can be described as follows: let $f(x)$ be the objective function to be minimised (or maximised), where x is a vector with a dimension n, which depends on the particular function being optimized. Each element of vector x is a design variable of function $f(x)$. At any k iteration, there are n design variables (i.e. $j = 1, 2, \dots n$) corresponding to the function in question, and m candidate solutions (i.e. population size, $i = 1, 2, \dots m$). Therefore, the whole population can be considered a matrix of dimension (m, n). The best candidate obtains the best value of $f(x)$ (i.e. $f(x)_{best}$) out of all candidate solutions, and the worst candidate obtains the worst value of $f(x)$ (i.e. $f(x)_{worst}$) out of all candidate solutions. If $X_{j,k,i}$ is the value of the jth variable for the kth candidate during the ith iteration, then this value is modified by the following equation:

$$X'_{j,k,i} = X_{j,k,i} + r_{1,j,i} \left(X_{j,best,i} - |X_{j,k,i}| \right) - r_{2,j,i} \left(X_{j,worst,i} - |X_{j,k,i}| \right), \quad (1)$$

where $X_{j,best,i}$ is the value of the j variable for the best candidate, and $X_{j,worst,i}$ is the value of the j variable for the worst candidate. In Equation (1), $X'_{j,k,i}$ is the updated value of $X_{j,k,i}$, and $r_{1,j,i}$ and $r_{2,j,i}$ are two random numbers, uniformly distributed in the range $[0, 1]$, for the jth variable computed in the ith iteration.

Algorithm 1 shows the skeleton of sequential Jaya algorithm implementation. The “Runs” parameter corresponds to the number of independent executions performed, therefore, in line 26 of Algorithm 1, the different “Runs” solutions should be evaluated. A more detailed description can be found, for example, in in [13], [19] or [21], which describe “Create New Population” and “Update Population” functions in detail.

3 Multi-level parallel Jaya algorithms

To efficiently exploit the maximum number of processors in a computing cluster we developed a multi-level algorithm, which will be described in this section. Two of these levels are developed to exploit shared memory architectures, being the most external level of these two levels applied to the concept of subpopulations. The strategy developed is similar to the structure developed in

Algorithm 1 Sequential Jaya algorithm

1: Define function to minimize
2: Set $Runs$, $Iterations$ and $PopulationSize$ parameters
3: **for** $l = 1$ to $Runs$ **do**
4: **Create New Population:**
5: {
6: **for** $i = 1$ to $PopulationSize$ **do**
7: **for** $j = 1$ to m **do**
8: Obtain 2 random numbers
9: Compute the design variable of the new member $Member_j^i$ {using Equation (1)}
10: **if** $Member_j^i < MinValue$ **then**
11: $Member_j^i = MinValue$
12: **end if**
13: **if** $Member_j^i > MaxValue$ **then**
14: $Member_j^i = MaxValue$
15: **end if**
16: **end for**
17: Compute and store $F(Member_j^i)$ {Function evaluation}
18: **end for**
19: }
20: **for** $l = 1$ to $Iterations$ **do**
21: Update Population
22: **end for**
23: Store Solution
24: Delete Population
25: **end for**
26: Obtain Best Solution and Statistical Data

[18], in which the whole, initially created population (lines 4 to 19 of Algorithm 1) is divided into sub-populations. However, in contrast to the sequential proposal presented in [18], the sub-populations are static, i.e. no population migrations are allowed. Note that, the sub-population structure is performed to parallelise the sequential algorithm.

The first parallel level developed (or outer level), which is suitable for shared-memory platforms, exploits a multipopulation structure, so that each sub-population is assigned to one shared-memory thread (OpenMP process). In this outer shared-memory parallel level, we will analyse two parallel options, the first one considers that the sub-populations share the best and the worst current solution, which we call "PMPS_Jaya" (Parallel MultiPopulation Single). In the second one, each process stores its own best and worst solutions, which we call "PMPM_Jaya" (Parallel MultiPopulation Multiple). Attending to the parallel performance, the first proposal involves memory contentions to access the global memory where the best and worst solutions are stored, while the latter proposal avoids synchronization processes, meaning that almost all processing tasks are performed in private memory, improving parallel behaviour.

The second level developed to exploit shared memory architectures, is located in processes more internal than the previous level. The main goal of this second level is to be able to increase the optimal number of processes in shared memory architectures. This second shared-memory parallel level (or inner level), is based on the parallel proposal presented in [13], i.e. parallelisation focuses on the "Update Population" function (line 21 of Algorithm 1), which is usually executed thousands, tens of thousands, or hundreds of thousands of times, depending on the value of the "Iterations" parameter (lines 20 to 22 of Algorithm 1). At this level of computation, there are already several threads (OpenMP processes) in execution, and as such, nested parallelism [4] needs to be used. Therefore, each outer parallel level thread have to spawn a group of threads to distribute the computational load associated to each population, i.e. each outer thread creates an inner parallel region. The maximum number of inner parallel region threads depends on the number of outer threads (i.e. the number of populations in the multipopulation algorithm) and the number of available cores in the shared-memory platforms, or the maximum number of threads if hyper-threading is enabled.

Figure 1 shows the two levels of parallelism implemented for a shared memory platforms. The number of processes in the outer level P is equal to the number of populations in the multipopulation structure. As described previously, the maximum number of processes spawned in each inner level is equal to the number of available cores divided by P.

In the parallel algorithm presented in [13], the size of the population should be increased to obtain good parallel efficiencies. Thanks to the double level of parallelism implemented in the current study, the maximum number of processes working in a single population is limited to a small number, and, therefore, it is possible to work efficiently with smaller populations.

Finally, the third level of parallelism is the outest level (or highest level), designed to exploit distributed-memory platforms. This third level, developed using MPI, is shown in Algorithm 2. Said algorithm exploits the fact that all iterations from line 3 in Algorithm 1 are actually independent executions. Therefore, the total number of executions ("Runs") to be performed is divided among the available distributed memory processes. The dispatcher process allows not to distribute the workload statically; a load-balancing procedure is intrinsically included in Algorithm 1. Moreover, the dispatcher process is executed in combination with to a worker process in a single core, as no significant overhead is introduced in the overall parallel algorithm performance. Since a multi-level algorithm has been developed, in practice all or only some of these levels can be exploited.

3.1 Optimization procedure improvement for the parallel algorithms

As described above, the Jaya algorithm is based on the concept that the solution obtained for a given problem should move towards the best solution and avoid the worst solution. Following Equation (1), where $X_{j,best,i}$ is the value of the j variable for the best candidate, and $X_{j,worst,i}$ is the value of the j variable for the worst candidate,$X^{'}_{j,k,i}$ is the updated value of $X_{j,k,i}$, and $r_{1,j,i}$ and $r_{2,j,i}$ are two random numbers, uniformly distributed in the range $[0, 1]$, for the jth

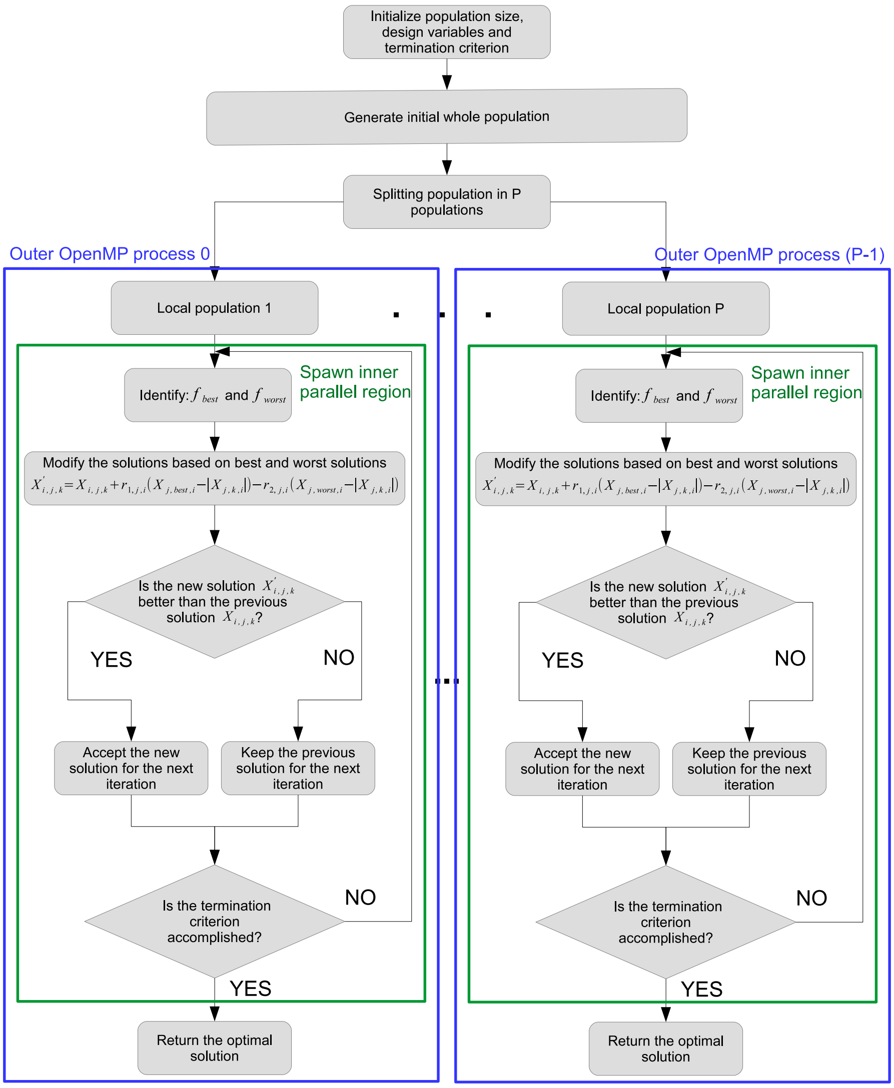

Figure 1 Multipopulation two-level parallel Jaya algorithm.

variable computed in the ith iteration. The term $r_{1,j,i}\,(x_{j,best,i} - |X_{j,k,i}|)$ designates the tendency (or intensity) of the algorithm to move closer to the best solution, whereas the term $-r_{2,j,i}\,(x_{j,worst,i} - |X_{j,k,i}|)$ designates the tendency (or intensity) of the algorithm to avoid the worst solution. The new candidate $(X'_{j,k,i})$ is accepted only if it gives a better function evaluation.

In our enhanced algorithm, we propose that both intensities are not random, instead only one of the intensities will be random and the other one depends on the previous process performed in each variable of each individual. To this end, in our proposal, when we compute a particular design variable

Algorithm 2 High level parallel algorithm

```
Define function to minimize
Set Runs, Iterations and PopulationSize parameters
Worker processes:
{
   while true do
      Request job to the dispatcher process
      if No remaining work then
         Send Solutions
         Break while
      else
         Obtain the number of populations (or number of outer processes) P
         Compute the size of nested parallel regions
         Compute 1 run of two level parallel Jaya algorithm
         Store Solution
      end if
   end while
}
Dispatcher process:
{
   for l = 1 to Runs do
      Receive work request
      Send processing order message
   end for
   for l = 1 to P do
      Receive work request
      Send No remaining work message
      Receive Solutions
   end for
   Obtain Best Solution and Statistical Data
}
```

of the individual, we obtain one random number to set the intensity used to move closer to the best solution, but the intensity used to avoid the worst solution is equal to the intensity that was used in the previous step to move closer to the best solution. The new computation of one design variable (line 9 in Algorithm 1) is shown in Algorithm 3. This change in the search path aims, on the one hand, to improve the speed of convergence and, on the other hand, to accelerate the parallel algorithms, both effects will be analyzed in Section 4.

4 Numerical experiments

In this section, we analyse the enhanced multi-level parallel Jaya algorithms based on the multipopulations presented in Section 3. We examine the parallel behaviour and the optimization performance of the parallel proposals. To perform the tests, we developed the reference algorithm (presented in [19]) in C language to implement the parallel algorithms and we used the GCC v.4.8.5 compiler [9]. We choose MPI v2.2 [15] for the high-level parallel approach and OpenMP API v3.1 [17] for the shared-memory parallel algorithms. The parallel platform used was a cluster composed of HP ProLiant SL390 G7 computing

Algorithm 3 Enhanced computing of the design variables

for $i = 1$ to $PopulationSize$ **do**
  Matrix allocation and initialization with $i * j$ random numbers (MtR)
  for $j = 1$ to m **do**
    Obtain a random number r_1
    Obtain $r_2 = MtR[i, j]$
    Compute the design variable of the new member $Member_j^i$ as:
    $NewMember_j^i = Member_j^i + r_1 \left(MemberBest_j^i - \left| Member_j^i \right| \right)$
    $-r_2 \left(MemberWorst_j^i - \left| Member_j^i \right| \right)$
    Store $MtR[i, j] = r_1$
  end for
end for

nodes, where each node was equipped with two Intel Xeon X5660 processors. Each X5660 includes six processing cores at 2.8 GHz, i.e. 12 cores per node with no hyper-threading enabled. Quadruple Data Rate Infiniband was used as the communication network.

Performance was analysed using 30 unconstrained functions (employed as benchmark in [19]), which are listed and described in tables 4 and 4.

Table 1: Benchmark 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	$f = 0.26(x_1^2 + x_2^2) - 0.48 x_1 x_2$
F6	$f = 100(x_1^2 - x_2)^2 + (x_1 - 1)^2 + (x_3 - 1)^2 + 90(x_3^2 - x_4)^2$ $+10.1\left((x_2 - 1)^2 + (x_4 - 1)^2\right) + 19.8(x_2 - 1)(x_4 - 1)$
F7 F8	$f = \sum_{i=1}^{V} (x_i - 1)^2 - \sum_{i=2}^{V} x_i x_{i-1}$
F9	$f = \sum_{i=1}^{V} x_i^2 + \left(\sum_{i=1}^{V} 0.5 i x_i \right)^2 + \left(\sum_{i=1}^{V} 0.5 i x_i \right)^4$
F10	$f = \sum_{i=1}^{V} \left(\sum_{j=1}^{i} x_j \right)^2$
F11	$f = \sum_{i=1}^{V-1} \left(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right)$

F12	$f=(x_1-1)^2+\sum_{i=2}^{V} i\left(2x_i^2-x_{i-1}\right)^2$
F13	$f=\left[\frac{1}{500}+\sum_{j=1}^{25}\frac{1}{j+\sum_{i=1}^{2}(x_i-a_{ij})^6}\right]^{-1}$
F14	$f=\left(x_2-\frac{5.1}{4\pi^2}x_1^2+\frac{5}{\pi}x_1-6\right)^2+10\left(1-\frac{1}{8\pi}\right)\cos x_1+10$
F15	$f=x_1^2+2x_2^2-0.3\cos(3\pi x_1)-0.4\cos(4\pi x_2)+0.7$
F16	$f=(x_1-2x_2-7)^2+(2x_1+x_2-5)^2$
F17 F18	$f=-\sum_{i=1}^{V}\sin x_i\left(\sin\left(\frac{ix_i^2}{\pi}\right)\right)^{20}$
F19	$f=x_1^2+2x_2^2-0.3\cos(3\pi x_1)\cos(4\pi x_2)+0.3$
F20	$f=x_1^2+2x_2^2-0.3\cos(3\pi x_1+4\pi x_2)+0.3$
F21	$f=\left[1+(x_1+x_2+1)^2(19-14x_1+3x_1^2-14x_2+6x_1x_2+3x_2^2)\right]$ $\left[30+(2x_1-3x_2)^2(18-32x_1+12x_1^2+48x_2-36x_1x_2+27x_2^2)\right]$
F22	$f=\sum_{j=1}^{V}\left[\sum_{i=1}^{i}(i^j+\beta)\left(\left(\frac{x_i}{i}\right)^j-1\right)\right]^2$
F23	$f=-\sum_{i=1}^{4}c_i\exp\left[-\sum_{j=1}^{3}a_{ij}(x_j-p_{ij})^2\right]$
F24	$f=-20\exp\left(-0.2\sqrt{\frac{1}{V}\sum_{i=1}^{V}x_i^2}\right)-\exp\left(\frac{1}{V}\sum_{i=1}^{V}\cos(2\pi x_i)\right)+20+e$
F25	$f=0.1\{\sin^2(3\pi x_1)+\sum_{i=1}^{V-1}(x_i-1)^2\left[1+\sin^2(3\pi x_{i+1})\right]$ $+(x_V-1)^2\left[1+\sin^2(2\pi x_V)\right]\}+\sum_{i=1}^{V}u(x_i,5,100,4),$ $u(x_i,a,k,m)=$ $k(x_i-a)^m, x_i>a; 0, -a\le x_i\le a; k(-x_i-a)^m, x_i<-a.$
F26 F27 F28	$f=-\sum_{i=1}^{5}c_i\left[\exp\left(-\frac{1}{\pi}\sum_{j=1}^{V}(x_j-a_{ij})^2\right)\cos\left(\pi\sum_{j=1}^{V}(x_j-a_{ij})^2\right)\right]$
F29 F30	$f=-\sum_{i=1}^{V}c_i\left(A_i-B_i\right)^2;$ $A_i=-\sum_{j=1}^{V}\left(a_{ij}\sin\alpha_j+bij\cos\alpha_j\right),$ $B_i=-\sum_{j=1}^{V}\left(a_{ij}\sin x_j+bij\cos x_j\right)$

Table 2: Benchmark functions.

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	Matyas	2	$-10, 10$
F6	Colville	4	$-10, 10$
F7	Trid 6	6	$-V^2, V^2$
F8	Trid 10	10	$-V^2, V^2$
F9	Zakharov	10	$-5, 10$
F10	Schwefel_1.2	30	$-100, 100$
F11	Rosenbrock	30	$-30, 30$
F12	Dixon-Price	30	$-10, 10$
F13	Foxholes	2	$-2^{16}, 2^{16}$
F14	Branin	2	$x_1 : -5, 10; x_2 : 0, 15$
F15	Bohachevsky_1	2	$-100, 100$
F16	Booth	2	$-10, 10$
F17	Michalewicz_2	2	$0, \pi$
F18	Michalewicz_5	5	$0, \pi$
F19	Bohachevsky_2	2	$-100, 100$
F20	Bohachevsky_3	2	$-100, 100$
F21	GoldStein-Price	2	$-2, 2$
F22	Perm	4	$-V, V$
F23	Hartman_3	3	$0, 1$
F24	Ackley	30	$-32, 32$
F25	Penalized_2	30	$-50, 50$
F26	Langermann_2	2	$0, 10$
F27	Langermann_5	5	$0, 10$
F28	Langermann_10	10	$0, 10$
F29	Fletcher-Powell_5	5	$x_i, \alpha_i : -\pi, \pi; a_{ij}, b_{ij} : -100, 100$
F30	Fletcher-Powell_10	10	$x_i, \alpha_i : -\pi, \pi; a_{ij}, b_{ij} : -100, 100$

The parallel algorithm presented in [13] needs work with medium and large populations in order to obtain good parallel efficiencies. In some cases, the use of large populations can increase the number of function evaluations performed to approach the optimum. Table 3 shows the number of function evaluations needed to reach an error less than 10^{-3}. As can be observed, working with small population sizes is preferable in most cases. Furthermore, with regard to the results presented in Table 3 and the rest of experiments performed, we can conclude that populations should not be extremely small.

As explained in Section 3, the outer level of the parallel algorithm developed for shared memory platforms is applied to the concept of subpopulations. We have developed two different versions of this level, both versions divide the

Table 3 Function Evaluations $\epsilon < 10^{-3}$.

Pop. Size	F1	F5	F10	F15	F20	F25	F30
10	13083	198	12804	520	557	15774	29151
12	10366	405	10791	596	682	25000	35552
14	10724	361	9817	705	788	32402	47568
16	11134	535	10373	879	981	35985	101240
32	24934	708	24007	1908	1696	25877	164324
48	45789	1250	43296	2858	2922	45378	183197
64	69376	1297	66583	3964	3255	69617	331307
80	97152	2021	91512	4600	5155	98899	296339
96	126246	1616	119293	6102	4963	134602	462411
112	158125	2315	147866	7183	6321	168052	436411
128	189871	2615	178278	7966	7727	208563	432267

whole population into sub-populations, but one of them uses the best and worst global individuals ("PMPS_Jaya"), while the other one uses local best and worst individuals for each population of the multipopulation structure ("PMPM_Jaya"). Therefore, the first one emulates the sequential Jaya algorithm almost exactly. If we were to fix the size of the sub-populations instead of the size of the whole population for the latter parallel algorithm, the whole population size would be equal to *"Sub-population Size x Number of outer processes"*. Table 4 shows the number of iterations performed to reach an error less than 10^{-3}, only using the outer parallel algorithm and setting the sub-population size to 14. Considering that the number of iterations is not reduced as the number of processes increases and the expected parallel performance improvement, the number of outer OpenMP processes should be set to a small value.

The number of OpenMP processes must be able to be increased to exploit parallel platforms with a greater number of cores. To this end, we developed the multi-level parallel algorithm in which each outer process spawn an inner parallel region. In our experiments, the outer parallel algorithm is executed with two, three or four OpenMP threads, and the inner parallel regions will spawn two or three inner threads.

Table 4 Number of Iterations $\epsilon < 10^{-3}$.

Subpop. Size	Num. proc.	F1	F5	F10	F15	F20	F25	F30
14	1	760	29	716	54	55	755	3934
14	2	746	28	700	57	61	396	3636
14	3	879	26	841	60	60	441	4298
14	4	1018	26	965	60	53	291	4287
14	5	1143	22	1073	64	56	419	3786
14	6	1242	18	1180	63	55	328	4383
14	7	1325	21	1250	63	54	303	4522
14	8	1403	24	1329	60	52	368	3593

Tables 5 and 6 show the speed-up obtained for both “PMPM_Jaya” and “PMPS_Jaya” algorithms, respectively. As can be observed, “PMPM_Jaya’s” parallel behaviour outperforms “PMPS Jaya’s” parallel performance in most cases, as expected.

Table 5 Speed-up for algorithm PMPS_Jaya.

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
2 pr.	1.69	1.69	1.83	1.73	1.70	1.63	1.76	1.94	1.84	1.77
3 pr.	2.43	2.37	2.50	2.24	2.01	2.09	2.51	2.60	2.48	2.46
4 pr.	3.02	3.06	3.27	2.81	2.39	2.98	3.24	3.30	3.36	3.22
	F11	F12	F13	F14	F15	F16	F17	F18	F19	F20
2 pr.	1.98	1.89	1.98	1.67	1.64	1.13	1.84	1.84	1.48	1.59
3 pr.	2.57	2.57	2.74	2.49	1.92	1.53	1.65	1.63	1.57	2.05
4 pr.	3.18	3.18	3.51	3.14	2.50	2.34	1.21	1.50	2.00	2.70
	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30
2 pr.	1.50	1.83	1.78	1.45	1.83	1.78	1.79	1.76	1.89	1.89
3 pr.	2.20	2.67	2.69	1.95	3.17	2.64	2.71	2.72	2.28	2.27
4 pr.	2.88	3.49	3.36	2.22	3.69	3.56	3.63	3.25	2.76	3.20

Table 6 Speed-up for algorithm PMPM_Jaya.

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
2 pr.	1.87	1.90	1.84	1.61	1.60	1.65	1.76	1.94	1.83	1.87
3 pr.	2.65	2.79	2.76	2.47	2.06	2.46	2.65	2.90	2.66	2.65
4 pr.	3.43	3.43	3.61	3.25	2.43	3.20	3.39	3.62	3.36	3.45
	F11	F12	F13	F14	F15	F16	F17	F18	F19	F20
2 pr.	1.85	1.83	1.73	1.75	1.64	1.25	1.85	1.74	1.42	1.57
3 pr.	2.66	2.65	2.61	2.47	2.39	1.80	2.76	2.74	2.04	2.32
4 pr.	3.41	3.48	3.42	3.13	3.17	2.39	3.42	3.50	2.58	3.07
	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30
2 pr.	1.49	1.77	1.78	1.14	2.27	1.81	1.80	1.76	1.79	1.90
3 pr.	2.08	2.63	2.70	1.68	3.13	2.63	2.66	2.64	2.68	2.72
4 pr.	2.94	3.51	3.29	2.05	3.70	3.51	3.54	3.48	3.47	3.53

When applying the enhanced search path explained in Section 3.1 to the multipopulation-based “PMPM_Jaya” algorithm, improvements were expected in both parallel behaviour and optimization performance for the new algorithm named “E_PMPM_Jaya” (Enhanced search path PMPM_Jaya). Table 7 shows the improvement in the “E_PMPM_Jaya” algorithm’s parallel behaviour with respect to the sequential Jaya, obtaining, in some cases, super speed-ups (i.e. speed-ups greater than the number of processes). Note that the sequential algorithm Jaya does not include the enhanced search path strategy. For the rest of the functions, the results are similar to those obtained with "PMPM_Jaya" algorithm.

Table 7 Speed-up for algorithm E_PMPM_Jaya.

	F1	F2	F8	F9	F10	F11	F12	F28	F29	F30
2 pr.	3.03	3.02	2.92	2.35	2.23	2.96	2.92	1.76	1.85	1.91
3 pr.	4.19	4.17	3.69	3.20	3.16	4.17	4.11	2.64	2.64	2.86
4 pr.	4.67	4.64	3.94	3.54	3.96	5.06	4.79	3.65	3.44	3.62

Table 8 shows the efficiency for the higher computational cost functions, when the two-level algorithm for shared-memory platforms is used, when using "E_PMPM_Jaya" as the inner algorithm. The efficiencies obtained are, on average, greater than 85%, which is a good value, especially considering that both the population size, equal to 14, and the number of iterations, equal to 10000, are small values.

Table 8 Efficiency for two level algorithm, (E_PMPM_Jaya included).

Outer proc.	Inner proc.	F3	F13	F22	F25	F26	F27	F28	F29	F30
2 pr.	4 pr.	83%	85%	85%	100%	82%	85%	87%	85%	84%
3 pr.	3 pr.	81%	85%	85%	93%	80%	85%	86%	84%	82%
4 pr.	2 pr.	85%	85%	85%	85%	82%	87%	86%	91%	83%

The results shown in Table 9 analyse the multilevel algorithm. Said results show the good parallel behaviour of the algorithm that exploits the parallelism of cluster computing platforms (distributed and shared memory platforms) including the third level of parallelism added, note that this third level is developed to exploit distributed memory platforms. Table 9 shows results corresponding to Table 8, where "Runs" equal to 30 and the number of distributed processes (MPI processes) equal to 15, being, on average, the efficiency equal to 85%, using between 120 and 135 processes.

Table 9 Efficiency for multilevel algorithm. 15 MPI processes.

Outer proc.	Inner proc.	F3	F13	F22	F25	F26	F27	F28	F29	F30
2 pr.	4 pr.	85%	84%	87%	99%	85%	85%	86%	82%	84%
3 pr.	3 pr.	85%	83%	82%	92%	81%	84%	87%	79%	80%
4 pr.	2 pr.	80%	84%	85%	88%	80%	84%	86%	90%	85%

Finally we will analyze the effect of the change in the optimization procedure explained in Section 3.1. In order to analyse the improvement in the random search path, we analysed the "E_PMPM_Jaya" algorithm and compared it to the original Jaya by conducting the Friedman's rank test [7]. This test's output includes the "p-value", a scalar value in the range $[0, 1]$, which is less than 0.05 when the results are statistically relevant, and χ^2 which is like a variance over the mean ranks. Mean and best values obtained for the benchmark test shown in tables 4 and 4 are considered for the test. Table 10 shows the results of the Friedman's rank test using a population size equal to 14, 1000 iterations and 30 "Runs", for the sequential algorithm. Furthermore, for the parallel algorithm, both the sub-population size and the number of function evaluations remain unchanged, i.e. the computational effort continues to be the same. We would like to comment that the "E_PMPM_Jaya" method obtains the best, statistically relevant results, when compared to the original Jaya results, and they improve as the number of processes increases, even improving the statistical relevance of said results. Worthy to note that the Jaya algorithm has been thoroughly analyzed and compared with the main reference algorithms, for example, in [1] Jaya is compared to GA and ICA (Imperialist Competitive Algorithm; in [10] it is compared to Cuckoo, PSO and TLBO; in [14] respect to GEM and TLBO; in [16] respect to BF (Brute Force); in [26] respect to PSO, DE and NMS (Nelder-Mead Simplex); in [29] respect to GA, PSO, GPS (Genetic Pattern Search), BBO, ABC and FA; and in [30] respect to TLBO, GOTLBO (Generalized Oppositional TLBO), LBSA (Learning Backtracking Search Algorithm), PSO, CLPSO (Comprehensive Learning PSO) and hybrid DE/BBO.

Table 10 Friedman rank's test comparing E_PMPM_Jaya and Jaya .

Number of parallel processes	Best value Rank	p-value	χ^2	Mean value Rank	p-value	χ^2
1 process	1.367	3.25E-02	4.57	1.267	1.70E-03	9.80
2 processes	1.250	3.00E-04	13.24	1.250	6.00E-04	11.84
3 processes	1.217	2.00E-04	13.76	1.167	2.01E-05	18.18
4 processes	1.150	2.66E-05	17.64	1.100	2.52E-05	22.15

5 Conclusions

In this study, we presented multi-level parallel Jaya algorithms, a recent optimization algorithm which is free of tuning parameters. We described the three levels of the parallel algorithms developed; two of which were for shared-memory platforms and the other one for distributed-memory platforms. We also proposed a modification in the random search path, which is proven effective, as demonstrated by Friedman's rank test, and moreover, means that parallel efficiency can be improved, especially for small populations. Both optimization and parallel performance were analysed using a benchmark of 30 unconstrained functions. Taking into account that engineering problems are usually complicated and have a large number of design variables, i.e. they are problems of high computational cost, the algorithms proposed could efficiently speed up resolving engineering optimization problems using supercomputing platforms or low-power computing platforms, also improving optimization performance.

References

1. Abhishek, K., Kumar, V.R., Datta, S., Mahapatra, S.S.: 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 pp. 1–19 (2016). DOI 10.1007/s00366-016-0484-8
2. Baños, R., Ortega, J., Gil, C.: Comparing multicore implementations of evolutionary meta-heuristics for transportation problems. Annals of Multicore and GPU Programming **1**(1), 9–17 (2014)
3. Baños, R., Ortega, J., Gil, C.: Hybrid mpi/openmp parallel evolutionary algorithms for vehicle routing problems. In: A.I. Esparcia-Alcázar, A.M. Mora (eds.) Applications of Evolutionary Computation: 17th European Conference, EvoApplications 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers, pp. 653–664. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
4. Blikberg, R., Srevik, T.: Load balancing and openmp implementation of nested parallelism. Parallel Computing **31**(10), 984 – 998 (2005). DOI 10.1016/j.parco.2005.03.018. OpenMP
5. Choudhary, A., Kumar, M., Unune, D.R.: Investigating effects of resistance wire heating on aisi 1023 weldment characteristics during asaw. Materials and Manufacturing Processes **33**(7), 759–769 (2018). DOI 10.1080/10426914.2017.1415441
6. Delisle, P., Krajecki, M., Gravel, M., Gagné, C.: Parallel implementation of an ant colony optimization metaheuristic with openmp. In: Proceedings of the 3rd European Workshop on OpenMP. Springer Berlin Heidelberg (2001)
7. Derrac, J., Garca, S., Molina, D., Herrera, F.: A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. Swarm and Evolutionary Computation **1**(1), 3 – 18 (2011). DOI 10.1016/j.swevo.2011.02.002
8. Dinh-Cong, D., Dang-Trung, H., Nguyen-Thoi, T.: An efficient approach for optimal sensor placement and damage identification in laminated composite structures. Advances in Engineering Software **119**, 48 – 59 (2018). DOI 10.1016/j.advengsoft.2018.02.005
9. Free Software Foundation, Inc.: GCC, the gnu compiler collection. https://www.gnu.org/software/gcc/index.html
10. Gambhir, M., Gupta, S.: Advanced optimization algorithms for grating based sensors: A comparative analysis. Optik **164**, 567 – 574 (2018). DOI 10.1016/j.ijleo.2018.03.062
11. Ghavidel, S., Azizivahed, A., Li, L.: A hybrid Jaya algorithm for reliability-redundancy allocation problems. Engineering Optimization **50**(4), 698–715 (2018). DOI 10.1080/0305215X.2017.1337755
12. Lin, M.H., Tsai, J.F., Yu, C.S.: A review of deterministic optimization methods in engineering and management. Mathematical Problems in Engineering **2012**(Article ID 756023), 15 (2012). DOI 10.1155/2012/756023
13. Migallón, H., Jimeno-Morenilla, A., Sánchez-Romero, J.L.: Parallel improvements of the jaya optimization algorithm. Applied Sciences **8**(5) (2018). DOI 10.3390/app8050819
14. Mishra, S., Ray, P.K.: Power quality improvement using photovoltaic fed dstatcom based on jaya optimization. IEEE Transactions on Sustainable Energy **7**(4), 1672–1680 (2016). DOI 10.1109/TSTE.2016.2570256

15. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (2009). Available at: http://www.mpi-forum.org
16. Ocloń, P., Cisek, P., Rerak, M., Taler, D., Rao, R.V., Vallati, A., Pilarczyk, M.: Thermal performance optimization of the underground power cable system by using a modified jaya algorithm. International Journal of Thermal Sciences **123**, 162 – 180 (2018). DOI j.ijthermalsci.2017.09.015
17. OpenMP Architecture Review Board: OpenMP Application Program Interface, version 3.1. http://www.openmp.org (2011)
18. Rao, R., More, K.: Design optimization and analysis of selected thermal devices using self-adaptive Jaya algorithm. Energy Conversion and Management **140**, 24–35 (2017). DOI 10.1016/j.enconman.2017.02.068
19. Rao, R.V.: Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems. International Journal of Industrial Engineering Computations **7**, 19–34 (2016). DOI 10.5267/j.ijiec.2015.8.004
20. Rao, R.V., Rai, D.P.: Optimisation of welding processes using quasi-oppositional-based Jaya algorithm. Journal of Experimental & Theoretical Artificial Intelligence **29**(5), 1099–1117 (2017). DOI 10.1080/0952813X.2017.1309692
21. Rao, R.V., Rai, D.P., Balic, J.: A multi-objective algorithm for optimization of modern machining processes. Engineering Applications of Artificial Intelligence **61**, 103–125 (2017). DOI 10.1016/j.engappai.2017.03.001
22. Rao, R.V., Saroj, A.: A self-adaptive multi-population based Jaya algorithm for engineering optimization. Swarm and Evolutionary Computation **37**, 1 – 26 (2017). DOI 10.1016/j.swevo.2017.04.008
23. Rao, R.V., Saroj, A.: Constrained economic optimization of shell-and-tube heat exchangers using elitist-Jaya algorithm. Energy **128**, 785–800 (2017). DOI 10.1016/j.energy.2017.04.059
24. Rao, R.V., Savsani, V., Vakharia, D.: Teaching-learning-based optimization: A novel method for constrained mechanical design optimization problems. Computer-Aided Design **43**(3), 303–315 (2011). DOI 10.1016/j.cad.2010.12.015
25. Rao, R.V., Waghmare, G.: A new optimization algorithm for solving complex constrained design optimization problems. Engineering Optimization **49**(1), 60–83 (2017). DOI 10.1080/0305215X.2016.1164855
26. Singh, S.P., Prakash, T., Singh, V., Babu, M.G.: Analytic hierarchy process based automatic generation control of multi-area interconnected power system using Jaya algorithm. Engineering Applications of Artificial Intelligence **60**, 35–44 (2017). DOI 10.1016/j.engappai.2017.01.008
27. Umbarkar, A.J., Joshi, M.S., Sheth, P.D.: Openmp dual population genetic algorithm for solving constrained optimization problems. International Journal of Information Engineering and Electronic Business **1**, 59–65 (2015). DOI 10.5815/ijieeb.2015.01.08
28. Umbarkar, A.J., Rothe, N.M., Sathe, A.: OpenMP Teaching-Learning Based Optimization Algorithm over Multi-Core System. International Journal of Intelligent Systems and Applications **7**, 19–34 (2015). DOI 10.5815/ijisa.2015.07.08
29. Wang, S.H., Phillips, P., Dong, Z.C., Zhang, Y.D.: Intelligent facial emotion recognition based on stationary wavelet entropy and jaya algorithm. Neurocomputing **272**, 668 – 676 (2018). DOI 10.1016/j.neucom.2017.08.015
30. Yu, K., Liang, J., Qu, B., Chen, X., Wang, H.: Parameters identification of photovoltaic models using an improved jaya optimization algorithm. Energy Conversion and Management **150**, 742 – 753 (2017). DOI 10.1016/j.enconman.2017.08.063