Article

Parallel improvements of the Jaya optimization algorithm

Héctor Migallón [1]*, Antonio Jimeno-Morenilla [2] and Jose-Luis Sanchez-Romero [2]

1 Department of Physics and Computer Architecture, Miguel Hernández University, E-03202, Elche, Alicante, Spain; hmigallon@umh.es
2 Department of Computer Technology, University of Alicante, E-03071, Alicante, Spain; {jimeno,sanchez}@dtic.ua.es
* Correspondence: hmigallon@umh.es; Tel.: +34966658390

Version March 21, 2018 submitted to Appl. Sci.

Abstract: A wide range of applications use optimization algorithms to find an optimal value, often a minimum one, for a given function. Depending on the application, both the optimization algorithms behavior, and its computational time, can prove to be critical issues. In this paper, we present our efficient parallel proposals of the Jaya algorithm, a recent optimization algorithm that enables to solve constrained and unconstrained optimization problems. We tested parallel Jaya algorithms for shared, distributed and heterogeneous memory platforms, obtaining good parallel performance while leaving Jaya algorithms behavior unchanged. Parallel performance was analyzed using 30 unconstrained functions reaching a speed-up of up to 57.6x using 60 processors. For all tested functions, the parallel distributed memory algorithm obtained parallel efficiencies that were nearly ideal, and combining it with the shared memory algorithm allowing us to obtain good parallel performance.

Keywords: Jaya; optimization problems; parallel; heuristic; OpenMP; MPI; hybrid MPI/OpenMP

1. Introduction

Optimization algorithms aim at finding an optimal value for a given function within a constrained domain. However, functions to be optimized can be highly complex and may present different numbers of parameters (or design variables). Indeed, many functions have local minima, so finding the absolute optimal value among the whole range of possibilities can be difficult.

Optimization methods fall into two main categories: deterministic and heuristic approaches. Deterministic approaches take advantage of the problem's analytical properties to generate a sequence of points that converge towards a global optimal solution. Deterministic approaches (e.g. linear programming, non-linear programming, and mixed integer non-linear programming) can provide general tools for solving optimization problems to obtain a global or an approximate global optimum (see [1]). Nonetheless, in the case of non-convex or large-scale optimization problems, the issues can be so complex that deterministic methods may not allow to easily derive a globally optimal solution within a reasonable time frame.

Heuristic optimization algorithms are usually classified into two main groups: Evolutionary Algorithms (EA) and Swarm Intelligence (SI) algorithms. Among EA algorithms, worthy of mention, among others, are: Genetic Algorithm (GA), Evolutionary Strategy (ES), Evolutionary Programming (EP), Genetic Programming (GP), Differential Evolution (DE), Bacteria Foraging Optimization (BFO), and Artificial Immune Algorithm (AIA). Among SI algorithms, worthy of mention are Particle Swarm Optimization (PSO), Shuffled Frog Leaping (SFL), Ant Colony Optimization (ACO), Artificial Bee Colony (ABC), and Fire Fly (FF) algorithm, among others. Moreover, other algorithms based on phenomena in nature have been developed, such as Harmony Search (HS), Lion Search (LS), Gravitational Search Algorithm (GSA), Biogeography-Based Optimization (BBO), and Grenade Explosion Method (GEM), among others.

The success of the vast majority of these algorithms is largely based on the parameters they use, which basically guide the search process and contribute chiefly to exploring the search space. The proper tuning of algorithm specific parameters represents a crucial success factor towards finding the global optimum. There are other algorithm-specific parameters such as population size, maximum number of generations, elite size, inertia weight, and acceleration rate (in PSO); onlooker bees, employed bees, and scout bees (in ABC); harmony memory, number of improvisations, and pitch adjusting rate (in HS) and so on.

Recently, two optimization algorithms, called TLBO [2] and Jaya [3], have been put forward allowing to dispense with specific parameter tuning. In fact, only general parameters such as number of iterations and population dimension are required. The TLBO (Teacher-Learner Based Optimization) and the Jaya algorithm are quite similar, the main difference being that TLBO uses two phases at every iteration (teacher and learner phases), while the Jaya algorithm performs only one. The Jaya algorithm in particular has sparked great interest that is growing within a range of diverse scientific areas, see [4–9] among others.

Some recent works show the advantages of using parallel architectures when executing optimization algorithms. Authors, in [10], implemented the TLBO algorithm on a multicore processor within an OpenMP environment. The OpenMP strategy emulated the sequential TLBO algorithm exactly, so calculation of fitness, calculation of mean, calculation of best, and comparison of fitness functions remained the same, while small changes were introduced to achieve better results. A set of 10 test functions were evaluated when running the algorithm on a single core architecture, and were then compared on architectures ranging from 2 to 32 cores. They obtain average speed-up values of $4.9x$ and $6.4x$ with 16 and 32 processors respectively.

Authors, in [11], implemented the Dual Population Genetic Algorithm on a parallel architecture. This algorithm is based on the original GA, but the Dual Algorithm adds a reserve population so as to avoid premature convergence proper to this kind of algorithm. A set of 8 test functions were optimized. Although they obtain average speed-up values of $1.64x$ using both 16 and 32 processors.

Authors, in [12] and [13], analyzed the performance of population-based meta-heuristics using MPI, OpenMP, and hybrid MPI/OpenMP implementations in a workstation with a multicore processor to solve a vehicle routing problem. A speed-up near $2.5x$ was reached in some cases, although in other cases a speed-up of only $1.0x$ to $1.5x$ was obtained.

Authors, in [14], present a parallel implementation of the ant colony optimization metaheuristic to solve an industrial scheduling problem in an aluminium casting centre. The number of processors was set from 1 to 16. Results indicated that maximum speed-up was achieved when using 8 processors, but speed-up decreased as the number of processors further increased. Obtaining a maximum speed-up of 5.94 using 8 processors, which goes down to 5.45 using 16 processors.

The above review of the state-of-the-art shows that is generally feasible to implement optimization techniques on a parallel architecture. However, there can be drawbacks in cases where implementations constrain the speed-up increment of parallel solutions when compared to sequential execution. Therefore, parallel implementation of these kinds of algorithms must be performed carefully to benefit from the advantages of parallel architectures.

We will now present in Section 2 the recent Jaya optimization algorithm and its advantages. In Section 3, we will describe the parallel algorithms that have been developed, and in Section 4, we analyze the latter both in terms of parallel performance and Jaya algorithm behavior. Conclusions are drawn in Section 5.

2. The Jaya algorithm

We review here some different studies of the Jaya algorithm and summarize their conclusions. Author, in [3], tested the performance of the Jaya algorithm, by means of a series of experiments on 24 constrained benchmark problems. The goal of the algorithm was to get closer to the best solution, but in so doing it also moves away from the worst solution. Results obtained by using the Jaya algorithm were compared with results obtained by other optimization algorithms such as GA, ABC, TLBO, and a few others. The superiority of Jaya was shown by means of two statistical tests: the Friedman rank test and the Holm-Sidak test. The Jaya algorithm came first in the case of best and mean solutions for all considered benchmark functions, while TLBO came second. With regard to the results of the Friedman rank test for the Success Rate solutions obtained, Jaya again came first followed by TLBO. The Holm-Sidak test provided a difference index related to the results obtained by Jaya and the other algorithms. This test showed a maximal difference between Jaya on the one hand and GA and BBO on the other, and a minimal difference with TLBO.

In the same study, Jaya performance was tested further on 30 unconstrained benchmark functions that are well known in the literature on optimization. Results obtained using Jaya were compared with results obtained using other optimization algorithms such as GA, PSO, DE, ABC and TLBO. Mean results obtained were compared with other algorithms. Jaya obtained better results in terms of best, mean and worst values of each objective function and standard deviation.

Authors, in [15], applied Jaya to 21 benchmark problems related to constrained design optimization. In addition to these problems, the algorithm's performance was studied over four constrained mechanical design problems. An analysis of the results revealed that Jaya was superior to, or could compete with, when applied to problems in question. Authors, in [16], showed that Jaya is applicable to data clustering problems. Results demonstrated that the algorithm exhibited better performance in most of the considered real-time datasets and was able to cluster appropriate partitions. We want to emphasize that our parallel algorithms do not modify the behavior of the Jaya algorithm.

3. Parallel approaches

First we will describe how the Jaya algorithm has been implemented in order to identify exploitable inherent sources of parallelism. Algorithm 1 shows the skeleton of the sequential implementation of the Jaya algorithm. The "Runs" parameter corresponds to the number of independent executions performed, therefore, in line 26 of Algorithm 1, "Runs" different solutions should be evaluated. First, for each independent execution, an initial population is computed (lines 7 – 19), for each population member VAR design variables can be obtained, note that the population size is an input parameter of the optimization algorithm while the number of design variables is an intrinsic characteristic of the function to be optimized. The second input parameter is the number of "Iterations", i.e. the number of new populations created based on the current population, once a new population is created, each member is compared with its corresponding one of the new population, being replaced if it improves the evaluation of the function. Detailed description of this procedure is shown in Algorithm 2.

As said, Algorithm 2 shows the main steps of the "Update Population" function (line 21 of Algorithm 1), which is usually executed thousands, or tens of thousands, or hundreds of thousands of times, that is, almost all the computing time is consumed by said function.

In lines 18–22 of Algorithm 2 a new member is computed using the Jaya algorithm, i.e. using Equation (1). Note that this computing uses both the current best and worst solution. In Equation (1) iterators j, k and i refer respectively to the design variable of the function, the member of the population and the current iteration, while $r_{1,k,i}$ and $r_{2,k,i}$ are random numbers uniformly distributed .

Algorithm 1 Skeleton of the Jaya algorithm

```
Define function to minimize
Set Runs parameter
Set Iterations parameter
Set PopulationSize parameter
for l = 1 to Runs do
    Create New Population:
    for i = 1 to PopulationSize do
        for j = 1 to VARS do
            Obtain 2 random numbers
            Compute the design variable of the new member Member_j^i {using Equation (1)}
            if Member_j^i < MinValue then
                Member_j^i = MinValue
            end if
            if Member_j^i > MaxValue then
                Member_j^i = MaxValue
            end if
        end for
        Compute and store F(Member_j^i) {Function evaluation}
    end for
    for l = 1 to Iterations do
        Update Population
    end for
    Store Solution
    Delete Population
end for
Obtain Best Solution and Statistical Data
```

Algorithm 2 Update Population function of the Jaya algorithm

1: **Update Population:**
2: {
3: {Obtain the current best and worst solution}
4: $F^{worst} = F(Mem^1)$
5: $Index^{worst} = 1$
6: $F^{best} = F(Mem^1)$
7: $Index^{best} = 1$
8: **for** $k = 2$ to $PopulationSize$ **do**
9:     **if** $F^{best} > F(M^k)$ **then**
10:         $Index^{best} = k$
11:     **end if**
12:     **if** $F^{worst} < F(M^k)$ **then**
13:         $Index^{worst} = k$
14:     **end if**
15: **end for**
16: **for** $k = 1$ to $PopulationSize$ **do**
17:     $old = k$
18:     **for** $j = 1$ to $VARS$ **do**
19:         Obtain 2 random numbers
20:         Compute the design variable of the new member $NewM_j$ {using Equation (1)}
21:         Check the bounds of $NewM_j$
22:     **end for**
23:     Compute $F(NewM)$ {Function evaluation}
24:     **if** $F(NewM) < F(Mem^{old})$ **then**
25:         $F(Mem^{old}) = NewM$
26:     **end if**
27: **end for**
28: {Search for current best and worst solution as in lines 4–15 }
29: }

$$x'_{j,k,i} = x_{j,k,i} + r_{1,k,i}\left(x_{j,best,i} - \left|x_{j,k,i}\right|\right) - r_{2,k,i}\left(x_{j,worst,i} - \left|x_{j,k,i}\right|\right). \tag{1}$$

As said, the number of design variables for each member of each population (represented by "VAR" in Equation (2)) depends on the function to optimize. In most of this study, we used the Rosenbrock function, as test function, shown in Equation (2), where the number of design variables (VAR) is equal to 30. Regarding Algorithm 2, much of the computational cost corresponds to lines 16 to 27. Note that in line 20 the best and worst solutions are used, so this procedure depends on the i iteration. In line 23 of Algorithm 2, the new member is evaluated using, for example, Equation (2), which corresponds to the Rosenbrock function. Naturally the computational cost of Algorithm 2 can vary significantly depending on the function to be optimized. Note that the total number of function evalutions depends on both the number of populations updates ("Iterations" parameter) and the population size.

$$F_{min} = \sum_{i=1}^{VAR}\left[100\left(x_i^2 - x_{i+1}\right)^2 + \left(1 - x_i\right)^2\right] \tag{2}$$

Algorithm 3 shows the skeleton of the shared memory parallel approach, of Algorithm 1. Algorithm 3 focuses on the number of performed population updates (i.e. "Iterations"), distributing these population updates among the **c** available processes ($r = 1, 2, \ldots, c$). In Algorithm 3, $\sum_{r=1}^{c} Iterations_r = Iterations$ must be satisfied, where $Iterations_r$ is the number of population updates performed by process r, since a dynamic scheduling strategy has been used the number of populations updates per thread is not a fixed number. Since this algorithm has been designed for shared memory platforms, using OpenMP, all solutions are stored in memory. Consequently, following parallel computation, the "sequential thread (or process)" obtains the best global solution and computes statistical values of all solutions obtained. As aforementioned, the number of iterations performed by each thread is not fixed, this number depends on the computational load assigned to each core in each particular execution, the automatic load balancing is implemented using the dynamic scheduling strategy of the OpenMP parallel loops. Note that the total number of functions evaluations remains unchanged.

Algorithm 3 Skeleton of shared memory parallel Jaya algorithm.

for $l = 1$ to $Runs$ **do**
    Parallel region:
    {
    Create New Population {Lines 7–19 of Algorithm 1}
    parallel for $i = 1$ to $Iterations$ **do**
        Update Population
    end for
    Store Solution
    Delete Population
    }
end for
Sequential thread:
    Obtain Best Solution and Statistical Data

Regarding algorithms 2 and 3, data dependencies exist in the "Update Population" function, solved in Algorithm 4, which shows the parallel "Update Population" function used in Algorithm 1. Note that to solve these data dependencies, Algorithm 4 includes up to 2 flush memory operations (lines 3 and 30) and up to 3 critical sections (lines 18, 21 and 39). Note that only the "flush" procedure of line 3 is performed in all iterations, the rest of flush and critical sections depends on the particular and non deterministic computation. An analysis of data dependencies of the "Update Population" function reveals that its corresponding parallel function must be designed for shared memory platforms. Note that the "flush" operations are performed to ensure that all threads have the same view of the memory variables in which current best and worst solutions are stored. Furthermore the critical sections are used to avoid hazards in memory accesses. Some optimizations have been implemented in Algorithm 4, improving both the computational performance and the Jaya algorithm behavior. On one hand in line 20 of the Algorithm 4, when a new global minimum is obtained it is quasi immediately used by all processes, on the other hand, in line 29, the search of the current worst member is performed only by the thread that has removed the previous worst element.

With respect to Algorithm 4, the computational load of one execution of the "Update Population" function can be not significant, depending on the computational cost of the function evaluation (line 16), which obviously depends on the particular function to be optimized. For example, in Equation (2), the number of floating point operations is 7 for each iteration of the sum, so only 239 floating point operations have to be performed in each evaluation. Therefore it is important to reduce both "flush" processes and "critical" sections, note that we have developed the parallel algorithm avoiding synchronization points. Reducing both the "flush" procedures and the critical sections besides the automatic load balancing allows to obtain good results both in efficiency and scalability. Worthy to note that, due to the large number of iterations performed, any poorly designed or implemented detail in the parallel proposal can significantly worse both the parallel performance and scalability.

As will be confirmed in Section 4, the good parallel behavior of the shared memory proposal of the Jaya optimization algorithm, encourage to develop a parallel algorithm to be executed in clusters, in order to be able to efficiently increase the number of processes, reducing, drastically, the computing time. In order to use heterogeneous memory platforms (clusters) on the one hand we must to identify a high level inherent parallelism and on the other hand we must to develop an hybrid memory model algorithm.

As explained in Section 2, and as can be seen in Algorithm 1, the Jaya algorithm performs several fully independent executions ("Runs"). Therefore, the Jaya algorithm offers great inherent parallelism at a higher level, but a key aspect must be the load balance. As aforementioned, we have developed a shared memory algorithm in which we have not used synchronization points and we have implemented techniques to ensure computational load balancing. The high level parallel algorithm must accomplish these objectives, and must be able to include the previously described algorithm.

The high level parallel Jaya algorithm exploits the fact that all iterations of line 5 in Algorithm 1 are actually independent executions. Therefore, the total number of executions ("Runs") to be performed is divided among **p** available processes, but taking into account that it can not be distributed statically. The high level parallel algorithm must be designed for distributed memory platforms using MPI, on the one hand we must to develop a load balance procedure, and on the other hand a final data gathering process (data collection from all processes) must be performed .

The hybrid MPI/OpenMP algorithm developed is shown in Algorithm 5 and will be analyzed on a distributed shared memory platform. First, note that if the number of worker processes desired is equal to p, the total number of distributed memory processes will be $p+1$. This is due there is a critical process (distributed memory process) that will be in charge of distributing the independent executions among the p available working processes, we call it work dispatcher. Although the work dispatcher process is a critical one, will be running in one of the nodes with worker processes as no significant overhead is introduced in the overall parallel algorithm performance. The work dispatcher will be waiting to receive a signal of work request from a idle worker process. When a particular worker process request a new work (independent execution), the dispatcher will assign a new independent execution or send an end of work signal. In lines 6 to 13 of Algorithm 5 it

Algorithm 4 Update Population function of the shared memory parallel Jaya algorithm

Update Population:

FLUSH operation over population and best and worst indices
for $k = 1$ to $PopulationSize$ **do**
    $old = k$
    for $j = 1$ to $VARS$ **do**
        Obtain 2 random numbers
        Compute the design variable of the new member $NewM_j$
        if $NewM_j < MinValue$ **then**
            $NewM_j = MinValue$
        end if
        if $NewM_j > MaxValue$ **then**
            $NewM_j = MaxValue$
        end if
    end for
    Compute $F(NewM)$ {Function evaluation}
    if $F(NewM) < F(Mem^{old})$ **then**
        CRITICAL SECTION to:
            $F(Mem^{old}) = NewM$
        if $F(NewM) < F(Mem^{best})$ **then**
            CRITICAL SECTION to:
                $Index^{Best} = i$ $\{Mem^{best} = NewM\}$
        end if
        if $Index_{worst} == k$ **then**
            $FlagUpdateWorst = 1$
        end if
    end if
end for
if $FlagUpdateWorst == 1$ **then**
    FLUSH operation over population
    $FlagUpdateWorst = 0$
    $FTemp^{worst} = F(Mem^1)$
    $IndexTemp^{worst} = 1$
    for $k = 2$ to $PopulationSize$ **do**
        if $FTemp^{worst} < F(Mem^k)$ **then**
            $IndexTemp^{worst} = k$
        end if
    end for
    CRITICAL SECTION to:
        $Index^{worst} = IndexTemp^{worst}$. $(Mem^{worst} = Mem^{IndexTemp^{worst}})$
end if

can be verified that the computational load of dispatcher process is negligible. In lines 15 to 24 of Algorithm 5 the shared memory parallel Jaya algorithm is used, i.e. Algorithm 3 setting "Runs" parameter equal to 1. The total number of processes is equal to $tp = p * c$, where p is the number of distributed memory worker processes (MPI processes) and c is the number of shared memory processes (OpenMP processes or threads). When distributed shared memory platforms (clusters) are used, they are probably heterogeneous multiprocessors, taking into account that the proposed algorithms include load balancing procedures at two levels the load balance is assured. Note that if the number of shared memory processes is equal to 1 ($c = 1$), Algorithm 5 is a distributed memory algorithm, and, to work with the hybrid algorithm if as shared memory algorithm the number of distributed memory processes must be equal to 2, that is one dispatcher process and one worker process. In all cases only worker distributed memory processes spawn shared memory threads.

Algorithm 5 Hybrid parallel Jaya algorithm for distributed shared memory platforms

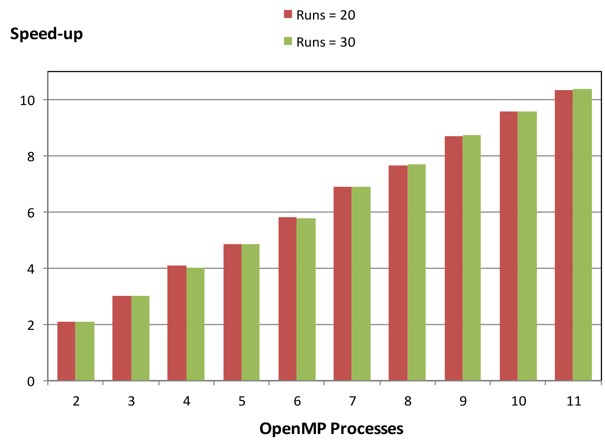
```
1:  Define function to minimize
2:  Set Iterations parameter (input parameter)
3:  Set population size (input parameter)
4:  Obtain the number of distributed memory worker processes p (input parameter)
5:  if is work dispatcher process then
6:      for l = 1 to Runs do
7:          Receive idle worker process signal
8:          Send independent execution signal
9:      end for
10:     for l = 1 to p do
11:         Receive idle worker process signal
12:         Send end of work signal
13:     end for
14: else
15:     while true do
16:         Send idle worker process signal to dispatcher process
17:         if Signal is equal to end of work signal then
18:             Break while
19:         else
20:             Obtain the number of shared memory processes c
21:             Compute 1 run of shared memory parallel Jaya algorithm
22:             Store Solution
23:         end if
24:     end while
25: end if
26: Perform a gather operation to collect all the solutions
27: Sequential thread of the root process:
28:   Obtain Best Solution and Statistical Data
```

4. Results and Discussion

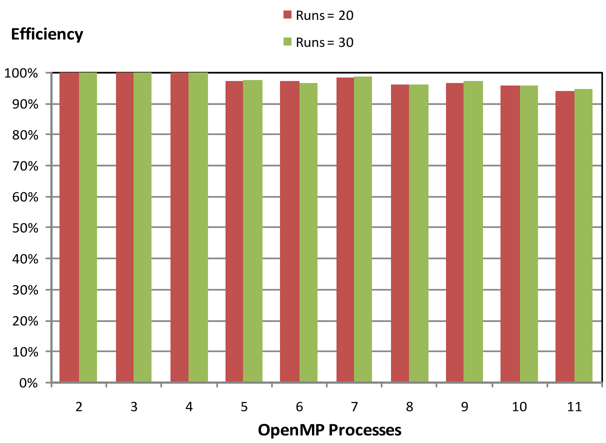
In this section we analyze the parallel Jaya algorithms, presented in Section 3. We analyze the parallel behavior and verify that the optimization performance of the Jaya algorithm slightly improves or remains unchanged respect to the sequential algorithm. In order to perform the tests, we developed the reference algorithm, presented in [3], in C language to implement the parallel algorithms, and used the GCC v.4.8.5 compiler [17]. We choose MPI v2.2 [18] for the high level parallel approach and OpenMP API v3.1 [19] for the shared memory parallel algorithm. The parallel platform used was composed of 10 HP Proliant SL390 G7 nodes, where each node was equipped with two Intel

Xeon X5660 processors. Each X5660 included six processing cores at 2.8 GHz, and QDR Infiniband was used as the communication network.

We will now analyze parallel behavior of the parallel algorithm described in Algorithm 4, i.e. the shared memory parallel algorithm. Figure 1 shows results setting the "Iterations" parameter equal to 30000 and using populations of 512 members. We observe that parallel efficiency being equal to respectively 100% and 94% when 2 and 11 processes are used, regardless of the value of "Runs". We can conclude that, based on results presented in Figure 1, and applying the Rosenbrock function, for population size equal to 512 the scalability is almost ideal.



(a) Speed-up



(b) Efficiency

Figure 1. Shared memory parallel Jaya algorithm. Iterations=30000, Population=512. (**a**) Speed-up respect to the sequential execution. (**b**) Efficiency of the parallel algorithm.

Figure 2 shows behavior related to population size and number of iterations. Regarding both this figure and the rest of experiments performed, the number of iterations does not affect parallel performance, while population size has been observed to be a critical parameter to obtain good parallel performance. Results presented in Figure 2 indicate that in order to obtain good parallel performance, population size must be greater than 64 members. In particular, in Figure 2, for population sizes greater than 128 members the efficiency is always higher than 90%.

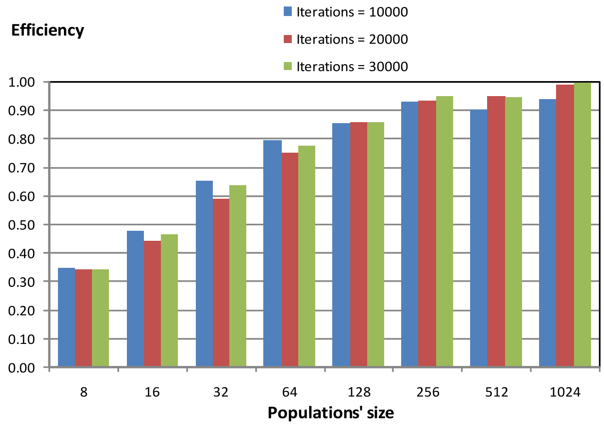


Figure 2. Efficiency of shared memory parallel Jaya algorithm. OpenMP processes=6. Runs=30.

Figure 3 shows behavior related to number of independent executions, i.e. the number of different solutions obtained. As expected, the number of independent executions does not affect the parallel behaviorof the shared memory parallel algorithm. We can conclude that the shared memory parallel algorithm obtains good parallel results of a minimum population size, and the rest of the parameters does not affect, or does it very slightly, the parallel performance.

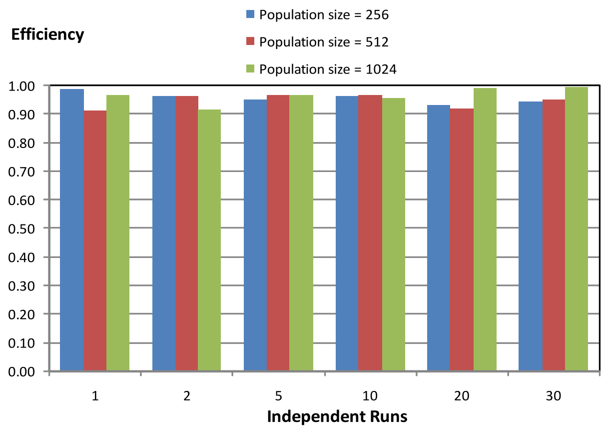
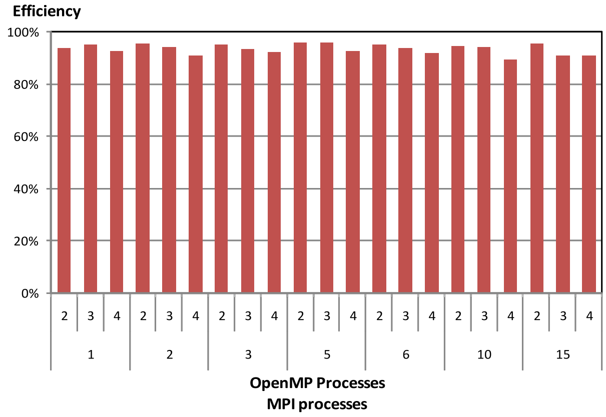
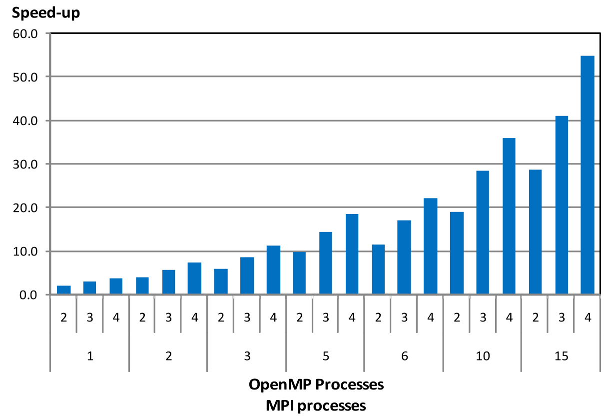


Figure 3. Efficiency of shared memory parallel Jaya algorithm. OpenMP processes=6. Iterations=30000.

The hybrid parallel algorithm developed combines the shared memory parallel algorithm analyzed and a high level parallel algorithm based on the distribution, among nodes (multiprocessors), of the computing load associated with the independent executions that will be carried out. The latter, described in Algorithm 5, having been developed with MPI while the former with OpenMP. However, in order to efficiently use all processing units of the parallel platform we mapped, where possible, more than one MPI process into one computing node. Figure 4 shows efficiency and speed-up for the hybrid parallel Jaya algorithm, executed in the heterogeneous memory platform previously described. It can be seen that the proposed hybrid parallel algorithm offers good scalability, note that we obtained a speed-up up to 54.6*x* using 60 processors of the heterogeneous memory platform. Worthy to note that the hybrid parallel algorithm exploits two levels of parallelism, and at both levels it includes load balancing mechanisms.



(a) Efficiency



(b) Speed-up

Figure 4. Hybrid parallel Jaya algorithm. Iterations=30000. Population=256. Runs = 30. (**a**) Efficiency of the parallel algorithm. (**b**) Speed-up respect to the sequential execution.

We must bear in mind that the execution of the Jaya algorithm is not a deterministic one, since Equation (1) depends on a random function. Each experiment in our study was performed by computing both the sequential and the parallel algorithm systematically and verifying that the results obtained were almost identical, i.e. each parallel experiment has been preceded by its corresponding sequential experiment, checking if the difference between both optimal solutions exceeds 10^{-3}. No errors have been produced except for experiments with very low number of function evaluations. On the other hand, in [3], author performs an exhaustive analysis of the optimization performance of the Jaya algorithm.

To finish, we will analyze parallel behavior depending on the functions in question. We used the same benchmark functions as in [3] listed in Table 1.

Table 1. Benchmark functions.

Id.	Function	VAR
F1	Sphere	30
F2	SumSquares	30
F3	Beale	5
F4	Easom	2
F5	Matyas	2
F6	Colville	4
F7	Trid 6	6
F8	Trid 10	10
F9	Zakharov	10
F10	Schwefel 1.2	30
F11	Rosenbrock	30
F12	Dixon-Proce	30
F13	Foxholes	2
F14	Branin	2
F15	Bohachevsky 1	2
F16	Booth	2
F17	Michalewicz 2	2
F18	Michalewicz 5	5
F19	Bohachevsky 2	2
F20	Bohachevsky 3	2
F21	GoldStein-Price	2
F22	Perm	4
F23	Hartman 3	3
F24	Ackley	30
F25	Penalized 2	30
F26	Langerman 2	2
F27	Langerman 5	5
F28	Langerman 10	10
F29	FletcherPowell 5	5
F30	FletcherPowell 10	10

Table 2 shows the results for all functions listed in Table 1, over 30 independent executions, 30000 iterations and populations of 256 members. Results shown in Table 2 are sequential computational time, parallel computational time, speed-up and parallel efficiency, where 10 MPI processes and 6 OpenMP processes were used, i.e. using 60 processors of the parallel platform. Note that in general, all functions obtained good parallel behavior. In most cases it is over 90%, and on average the efficiency is equal to 87% . Considering only functions with efficiency above previous average, the average efficiency is 92%. We can increase the efficiency decreasing the total number of processes used, note that the sequential processing time is up to only 12.0s..

Table 2. Sequential and parallel Jaya results. Population=256. Iterations=30000. 10 MPI processes. 6 OpenMP processes. Runs=30.

Function	Seq. time (s.)	Par. time (s.)	Speed-up	Effic.
F1	126.0	2.58	48.9	81%
F2	129.9	2.67	48.6	81%
F3	108.5	2.02	53.7	90%
F4	26.7	0.50	53.7	90%
F5	70.2	1.42	49.3	82%
F6	18.4	0.41	44.6	74%
F7	26.6	0.53	50.0	83%
F8	44.7	0.87	51.6	86%
F9	67.7	1.55	43.7	73%
F10	254.9	4.85	52.6	88%
F11	131.9	2.46	53.7	90%
F12	132.4	2.44	54.2	90%
F13	999.9	17.59	56.8	95%
F14	16.3	0.33	49.9	83%
F15	17.3	0.34	51.0	85%
F16	9.4	0.23	41.4	69%
F17	54.9	1.05	52.5	87%
F18	171.8	3.15	54.5	91%
F19	12.4	0.26	48.2	80%
F20	16.2	0.32	51.1	85%
F21	12.0	0.27	44.3	74%
F22	330.3	5.74	57.6	96%
F23	45.5	0.81	56.1	94%
F24	465.6	8.21	56.7	95%
F25	583.8	10.25	56.9	95%
F26	82.9	1.54	53.7	90%
F27	474.4	8.57	55.3	92%
F28	1999.5	34.80	57.5	96%
F29	362.1	6.44	56.2	94%
F30	1471.8	25.81	57.0	95%

Reducing the number of processes and leaving other parameters unchanged, parallel behavior improved as expected. Table 3 shows results corresponding to Table 2 where only 2 OpenMP processes, i.e. using 20 processors, for those functions with lower computational cost. As can be seen in Table 3 efficiency values improve significantly, as anticipated, taking into account that high level parallel algorithm offers better scalability.

Table 3. Sequential and parallel Jaya results. Population=256. Iterations=30000. 10 MPI processes. 2 OpenMP processes. Runs=30.

Function	Seq. time (s.)	Par. time (s.)	Speed-up	Effic.
F6	18.5	0.97	19.2	96%
F7	24.4	1.46	16.7	84%
F14	16.7	0.89	18.8	94%
F15	17.2	0.88	19.6	98%
F16	9.0	0.55	16.3	82%
F19	12.0	0.64	18.8	94%
F20	15.9	0.87	18.2	91%
F21	11.9	0.61	19.5	98%

Finally, worthy to note that the parallel code has been fully optimized, thus improving parallel proposals of similar algorithms, moreover our hybrid proposal includes load balancing mechanisms at two levels. For example in [10] and [11] parallelizing using OpenMP and using 8 processes the maximum efficiency achieved is 55% and 56% respectively, while in the case of [12] it is only 23%. Under these conditions our shared memory algorithm and our hybrid algorithm obtains an average efficiency higher than 90%. There is a recent algorithm, called HHCPJaya presented in [20], which obtains good results on both parallel and optimization performance. However this algorithm has some drawbacks: it does not seem to be a general optimization algorithm, that is, the function to be optimized must be coded according to the partition at the level of the design variable performed, for example, a function like "Dixon-Price" function can not be encoded using its general formulation; the method seems deterministic and not heuristic, because the seed used is always the same and does not use a random function; the method uses a deterministic function to generate the sequence of random numbers; and when the random function is used instead the deterministic function the method shows poor scalability. We have presented results using worker processes of up to 60, in our implementation each of these processes uses a different seed for the generation of the sequence of random numbers, on the other hand our algorithms do not need neither hyperpopulations nor functions with a large number of design variables.

5. Conclusion

The recent Jaya algorithm has been shown to be an effective optimization algorithm. In this study, we developed parallel algorithms and presented a detailed analysis of them. We developed a hybrid MPI/OpenMP algorithm, the algorithm developed exploits inherent parallelism at two different levels. The lower level is exploited by parallel shared memory platforms, while the upper level is exploited by distributed shared memory platforms. Both algorithms obtain good results especially in scalability, so the hybrid algorithm is able to use a large number of processes with almost ideal efficiencies. In the experiments shown, up to 60 processes are used obtaining almost ideal efficiencies. We analyzed it using 30 unconstrained functions, obtaining good parallel efficiencies for all the test functions. In addition, both levels of parallelization include load balancing mechanisms that allow the execution of this algorithm in non-dedicated environments, either supercomputing platforms or low power computing platforms, without degrading computational performance.

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

Author Contributions: Héctor Migallón, Antonio Jimeno-Morenilla and José-Luis Sánchez Romero conceived the parallel algorithms; Héctor Migallón desined and codified the parallel algorithms; Héctor Migallón and Antonio Jimeno-Morenilla analyzed the data; Héctor Migallón and José-Luis Sánchez Romero wrote the paper.

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

References

1. Lin, M.H.; Tsai, J.F.; Yu, C.S. A Review of Deterministic Optimization Methods in Engineering and Management. *Mathematical Problems in Engineering* **2012**, *2012*, 15. doi:10.1155/2012/756023.
2. Rao, R.V.; Savsani, V.; Vakharia, D. Teaching-learning-based optimization: A novel method for constrained mechanical design optimization problems. *Computer-Aided Design* **2011**, *43*, 303–315. doi:10.1016/j.cad.2010.12.015.
3. Rao, R.V. Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems. *International Journal of Industrial Engineering Computations* **2016**, *7*, 19–34. doi:10.5267/j.ijiec.2015.8.004.
4. 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* **2017**, *60*, 35–44. doi:10.1016/j.engappai.2017.01.008.

5. Mishra, S.; Ray, P.K. Power quality improvement using photovoltaic fed DSTATCOM based on JAYA optimization. *IEEE Transactions on Sustainable Energy* **2016**, *7*, 1672–1680. doi:10.1109/TSTE.2016.2570256.
6. Rao, R.; More, K. Design optimization and analysis of selected thermal devices using self-adaptive Jaya algorithm. *Energy Conversion and Management* **2017**, *140*, 24–35. doi:10.1016/j.enconman.2017.02.068.
7. Rao, R.V.; Saroj, A. Constrained economic optimization of shell-and-tube heat exchangers using elitist-Jaya algorithm. *Energy* **2017**, *128*, 785–800. doi:10.1016/j.energy.2017.04.059.
8. Rao, R.V.; Rai, D.P.; Balic, J. A multi-objective algorithm for optimization of modern machining processes. *Engineering Applications of Artificial Intelligence* **2017**, *61*, 103–125. doi:10.1016/j.engappai.2017.03.001.
9. 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* **2016**, pp. 1–19. doi:10.1007/s00366-016-0484-8.
10. 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* **2015**, *7*, 19–34. doi:10.5815/ijisa.2015.07.08.
11. 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* **2015**, *1*, 59–65. doi:10.5815/ijieeb.2015.01.08.
12. Baños, R.; Ortega, J.; Gil, C. Comparing multicore implementations of evolutionary meta-heuristics for transportation problems. *Annals of Multicore and GPU Programming* **2014**, *1*, 9–17.
13. Baños, R.; Ortega, J.; Gil, C. Hybrid MPI/OpenMP Parallel Evolutionary Algorithms for Vehicle Routing Problems. Applications of Evolutionary Computation: 17th European Conference, EvoApplications 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers; Esparcia-Alcázar, A.I.; Mora, A.M., Eds.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2014; pp. 653–664.
14. Delisle, P.; Krajecki, M.; Gravel, M.; Gagné, C. Parallel implementation of an ant colony optimization metaheuristic with OpenMP. Proceedings of the 3rd European Workshop on OpenMP. Springer Berlin Heidelberg, 2001.
15. Rao, R.V.; Waghmare, G. A new optimization algorithm for solving complex constrained design optimization problems. *Engineering Optimization* **2017**, *49*, 60–83. doi:10.1080/0305215X.2016.1164855.
16. Kurada, R.R.; Kanadam, K.P. Automatic Unsupervised Data Classification Using Jaya Evolutionary Algorithm. *Advanced Computational Intelligence: An International Journal* **2016**, *3*, 35–42. doi:10.5121/acii.2016.3204.
17. Free Software Foundation, Inc.. GCC, the GNU Compiler Collection. *https://www.gnu.org/software/gcc/index.html*.
18. MPI Forum. : A Message-Passing Interface Standard. Version 2.2, 2009. available at: http://www.mpi-forum.org.
19. OpenMP Architecture Review Board. OpenMP Application Program Interface, version 3.1. *http://www.openmp.org* **2011**.
20. Michailidis, P.D. An efficient multi-core implementation of the Jaya optimisation algorithm. *International Journal of Parallel, Emergent and Distributed Systems* **2017**, *0*, 1–33. doi:10.1080/17445760.2017.1416387.