Applied Soft Computing 10 (2010) 284–295

Contents lists available at ScienceDirect

Applied Soft Computing

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

Discrete cooperative particle swarm optimization for FPGA placement

Mohammed El-Abd [a,*], Hassan Hassan [b], Mohab Anis [a], Mohamed S. Kamel [a], Mohamed Elmasry [a]

[a] University of Waterloo, ECE Department, 200 University Av. W., Waterloo, Ontario, N2L3G1, Canada
[b] Actel Corporation, 2061 Stierlin Court, Mountain View, CA 94043, USA

ARTICLE INFO

Article history:
Received 10 October 2007
Received in revised form 20 July 2009
Accepted 26 July 2009
Available online 5 August 2009

Keywords:
Particle swarm optimization
Cooperative search
Discrete applications
FPGA placement

ABSTRACT

Particle swarm optimization (PSO) is a stochastic optimization technique that has been inspired by the movement of birds. On the other hand, the placement problem in field programmable gate arrays (FPGAs) is crucial to achieve the best performance. Simulated annealing algorithms have been widely used to solve the FPGA placement problem. In this paper, a discrete PSO (DPSO) version is applied to the FPGA placement problem to find the optimum logic blocks and IO pins locations in order to minimize the total wire-length. Moreover, a co-operative version of the DPSO (DCPSO) is also proposed for the FPGA placement problem. The problem is entirely solved in the discrete search space and the proposed implementation is applied to several well-known FPGA benchmarks with different dimensionalities. The results are compared to those obtained by the academic versatile place and route (VPR) placement tool, which is based on simulated annealing. Results show that both the DPSO and DCPSO outperform the VPR tool for small and medium-sized problems, with DCPSO having a slight edge over the DPSO technique. For higher-dimensionality problems, the algorithms proposed provide very close results to those achieved by VPR.

1. Introduction

Particle swarm optimization (PSO) [1,2] is an optimization algorithm widely used to solve continuous nonlinear functions. It is a stochastic optimization technique that was originally developed to simulate the movement of a flock of birds or a group of fish looking for food.

Field programmable gate arrays (FPGAs) are digital circuits that provide a programmable alternative to ASIC designs for prototyping and small-volume production. The market shares of FPGAs had witnessed a huge increase recently since FPGA vendors started providing a variety of FPGA sizes for different applications. The design process of FPGAs involves synthesizing a user defined circuit and placing it on the programmable resources of FPGAs. The placement problem in FPGAs has always been the limiting factor for FPGA performance. The FPGA placement problem is a combinatorial problem where the logic and IO blocks are distributed among the available physical locations to either minimize the total wire-length or minimize the delay along the critical path. The most widely used optimization algorithm in the FPGA placement problem is simulated annealing (SA) [3].

In this work, a discrete PSO (DPSO) version is developed and applied to the FPGA placement problem to minimize the design total wire length. The algorithm is applied to several FPGA benchmarks with varying problem sizes to investigate the algorithm scalability. The results are compared to a well-known academic FPGA placement tool, versatile place and route (VPR) [3]. VPR uses SA as an optimization algorithm to find the placement that achieves the minimum cost. The work also proposes reducing the DPSO computational cost by adopting a cooperative version of the algorithm.

This paper is organized as follows; in Section 2, an overview of the conventional PSO algorithm is presented. A survey of the previous versions of the different discrete PSO algorithms proposed in the literature is introduced in Section 3. In Section 4, the FPGA placement problem is discussed. Section 5 presents the proposed DPSO algorithm and the proposed FPGA placement problem formulation. The cooperative version is discussed in details in Section 6. The experimental setup and results are discussed in Section 7. Finally, the paper is concluded in Section 8.

2. Particle swarm optimization

The PSO method is regarded as a population-based method, where the population is referred to as a swarm. The swarm consists of a number of individuals called particles. Each particle i in the swarm holds the following information: (i) the current position x_i, (ii) the current velocity v_i, (iii) the best position, the one associated with the best fitness value the particle has achieved so far $pbest_i$, and (iv) the global best position, the one associated with the best

* Corresponding author.
E-mail address: mhelabd@pami.uwaterloo.ca (M. El-Abd).

1568-4946/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.asoc.2009.07.011

fitness value found among all of the particles *gbest*. In every iteration, each particle adjusts its own trajectory in space, x, in order to move towards its best position and the global best according to the following relations:

$$v_{ij}(t+1) = w \otimes v_{ij}(t) + c_1 \times \text{rand}() \otimes (pbest_{ij} - x_{ij}) + c_2 \times \text{rand}() \otimes (gbest_{ij} - x_{ij}), (1)$$

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1), \quad (2)$$

for $j \in \{1, \ldots, d\}$ where d is the number of dimensions, $i \in \{1, \ldots, n\}$ where n is the number of particles, t is the iteration number, w is the inertia weight, rand() generates a random number uniformly distributed in the range $(0, 1)$, and c_1 and c_2 are the acceleration factors.

Afterwards, each particle updates its personal best according to (assuming a minimization problem)

$$pbest_i(t+1) = \begin{cases} pbest_i(t) & \text{if } f(pbest_i(t)) \le f(x_i) \\ x_i(t) & \text{if } f(pbest_i(t)) > f(x_i) \end{cases} \quad (3)$$

where $f(.)$ is a function that evaluates the fitness value for a given position. Finally, the global best of the swarm is updated using

$$gbest(t+1) = \arg \min_{pbest_i} \{ f(pbest_i(t+1)) \}, \quad (4)$$

This model is referred to as the *gbest* (global best) model.

3. Discrete PSO

Different discrete versions have been proposed for PSO in the literature [4–18]. Binary PSO algorithms are not covered in this survey, but rather we cover the PSO algorithms where the particles were defined as permutations.

3.1. Traveling salesman problem

In Ref. [4], a discrete PSO was proposed to solve the TSP problem. The particles were defined as permutations of cities. The velocities were defined as a sequence of swaps that could be applied on the position. The addition, subtraction and multiplication operators were all redefined to work using the new position and velocity definitions. The addition of a velocity and a position was done applying the swaps operations defined by the velocity on the position. The subtraction of two positions was done by finding the sequence of swaps that transform one position to the other. Multiplying the velocity by a constant c was defined as either truncating the velocity sequence ($c < 1$) or augmenting it with swaps taken from the start of velocity vector ($c > 1$). The algorithm was successfully applied to a TSP problem with a dimensionality of 17.

In Ref. [5], the authors solved the TSP problem using the same definitions as in Ref. [4] extending it with what is known as the *basic swap sequence*. The authors argued that two different swap sequences when applied to a certain position could yield the same resultant position, such sequences belong to the *equivalent set of swap sequences*. The basic swap sequence is the one that belong to the same equivalent set but having the minimum number of swaps. When two positions are subtracted, the authors always find the basic swap sequence and when the velocity of a particle is updated it is always transformed to its basic swap sequence. The authors successfully applied this algorithm to an instance of 14 cities finding its optimum solution.

The authors of Ref. [6] proposed a discrete PSO algorithm for solving the generalized TSP (GTSP) problem. The proposed algorithm had a modified subtraction operator and employed two local searching techniques for accelerating the speed of convergence and was applied to problems with up to 40 groups.

A modified PSO (MPSO) algorithm for solving the TSP problem was proposed in Ref. [7]. Each particle was defined as a permutation of cities while the velocities were defined as a set of *adjustment operators*. Each adjustment operators $T(i, j)$ was applied by removing the node at index i, inserting it at index j and shifting the rest of the position vector. The authors argued that this operator is better than the swap operator used in Refs. [4,5] as it prevents the position from returning back to the same permutation it was at before if the velocity is kept the same for two consecutive iterations. All the operations were defined as in Refs. [4,5] keeping in mind the new interpretation of the velocity. In addition, if the *gbest* value does not improve for a predefined number of iterations, a group of the particles are selected and their velocities are randomly initialized. To further improve the performance and prevent premature convergence, each particle, at the start of the search, moves towards its personal best and a randomly selected particle with a high probability. Towards the end of the search, this probability gets lower and each particle follows the global best instead. The method was only applied to two instances of a dimensionality of 14 and 29 performing a large number of function evaluations, 2×10^5 and 3×10^6, respectively.

3.2. Assignment problems

A discrete PSO was used to solve the task assignment problem in Ref. [8]. The particle was defined as an array whose element values are the tasks while the indices are the processors. On the other hand, the velocity was kept as a real-coded vector. The algorithm was applied to benchmarks of up to 169 tasks, PSO provided better results than a generational GA for most of the test cases studied while minimizing the computational cost.

In Ref. [9], a discrete PSO was applied to the weapon-target assignment (WTA) problem. Each particle position is represented as a vector, where the index is the target and the value is the weapon assigned to it. A greedy search algorithm was applied in every iteration to *gbest*. Updating a particle's position was done by means of permutation. With a fixed probability *FP*, a number of elements $W \times FP$ will have the same index in the current position as they have in the *gbest*. And with an unfixed probability *UFP*, a number of elements $W \times UFP$ in *pbest* will be allowed to change indices into the current position. The algorithm successfully found the optimal solution for a problem of size 10. It also outperformed both a generic GA and a GA with greedy eugenics [19] for problems with a dimensionality up to 60. However, the number of function evaluations used in the experiments was not reported.

3.3. Scheduling applications

A dual similar PSO algorithm (DSPSOA) was proposed in Ref. [10] for solving the job-shop scheduling (JSS) problem. The particle was defined as an array with a number of elements equal to the number of jobs. The velocity was not explicitly defined as the position was updated through a sequence of crossover and mutation operators. First, for every particle, a crossover operation is applied to *pbest* and *gbest*, then the resultant vector is mutated. The result of the mutation process is again crossovered with the current position with the result being mutated again. The algorithm consisted of two PSO algorithms, referred to as the outer and inner PSOs. Both algorithms used different operators for the crossover and mutation. The algorithm was only applied to a small sized problem consisting of 10 jobs and 5 machines with 600 function evaluations for the outer PSO and 3000 function evaluations for the inner one.

For a single machine job scheduling problem, a discrete PSO algorithm was proposed in Ref. [11]. The particle position was made of a vector of $n + b$ jobs, where n is the number of jobs and b is the number of dummy jobs. To update the position, a swap operator is applied to the position with a certain probability w. Then a one-cut crossover is applied between the swapped position and *pbest* with a probability c_1 to obtain a more updated position. Finally, a two-cut crossover operator is performed between the updated position and *gbest* with a probability c_2 to find the new current position. Local search was applied to the *gbest* at every iteration to improve the solution quality. Their method was applied successfully to problems having up to 1000 jobs. The number of function evaluations performed had a maximum of 1.6×10^4 for the smallest problem and 3×10^5 for the biggest one.

A discrete PSO algorithm was applied to the permutation flowshop scheduling problem in Ref. [12]. The particles were defined as arrays containing the numbers of the jobs to be scheduled and the velocities were defined as sequences of swaps. The authors applied the work to different benchmarks containing $n = 20$ jobs and varying number of machines $m = 5, 10$ and 20. The algorithm provided the best performance when the number of jobs is less than the number of available machines. However, the allocated number of allowed function evaluations was relatively high at $1000 \times n \times m$. This work was extended in Ref. [13], by adding a local search mechanism where every position in the sequence is swapped with all the other positions one by one, if a better fitness is reached then the exchange is made and the local search process is stopped. Otherwise, no exchange of positions is done.

The authors in Ref. [14], proposed a permutation-based PSO algorithm to solve the task scheduling problem. In the permutation form, every particle was represented as an array where the element value is the task number and the index is its priority. The authors used the same swap operators as defined in Refs. [4,5] as their velocity. To improve the global search ability of the algorithm, the authors adopted a simulated annealing scheme when accepting the new solutions generated by the PSO equations instead of just replacing the old positions in the normal PSO algorithm. The algorithm was successfully applied to task scheduling problems having a number of tasks up to 100 and up to 12 machines.

3.4. Miscellaneous applications

In Ref. [15], an improved PSO was applied to the steelmaking charge plan problem, where the authors used a discrete presentation. The charge plan scheduling problem was modeled as a TSP problem. The particle's position was modeled as an array, where the array index is the slab number and the array element is the charge number. The velocity was not defined. Instead, all the operations were applied directly on the position. Moving towards the *gbest* and the *pbest* positions was done through two crossover operations. The first crossover operation was between the current position and *gbest*, then the second crossover operation was done between the resultant of the first crossover operation and *pbest*. Finally, multiplying the velocity by the inertia weight was modeled by a mutation operation (swapping two random cities) that is applied to the result of the second crossover operation. The method was successfully applied to problems with a dimensionality up to 16. In Ref. [16], the same approach was applied to problems with a dimensionality up to 30.

In Ref. [17], the authors proposed a discrete PSO algorithm, referred to as SetPSO, to optimize the structure of RNA molecules. The addition of two particles was defined as the union of the two sets. On the other hand, the subtraction was defined as the set-theoretic difference of the two sets. SetPSO was applied to 4 benchmarks of dimensionality 118, 122, 784 and 945 performing 35,000 function evaluations. The percentage of the correct pairs detected were 89%, 76.3%, 36.7% and 15.9% respectively.

In Ref. [18], the authors proposed the use of geometric PSO (GPSO) to solve the Sudoko problem. GPSO adopts a new set of equations to update the particles' position. Each particle is evolved by implementing a convex combination between the current position, *pbest* and *gbest*. This was based on the original PSO equations after dropping the velocity term. Then, the combination process is followed by a mutation operator. The authors successfully solved a Sudoko problem in 36 out of 50 attempts employing 100 particles and the Von-Neumann model as the underlying topology.

PSO was also applied to the field programmable gate arrays (FPGA) placement problem in Ref. [20]. Although the FPGA placement problem is a discrete optimization one, the authors solved it in the continuous space. Moreover, the algorithm was only applied to two problems without comparing the results to any other optimization approach, and the best known solution for these problems were not reported.

4. FPGAs placement problem

FPGAs consist of programmable logic resources (logic clusters) embedded in a sea of programmable interconnects (routing channels), as shown in Fig. 1. Moreover, programmable IO pads are distributed along the edges of the fabric (Fig. 1).

The programmable logic resources are configured to implement any logic function, while the interconnects provide the flexibility to connect any signal in the design to any logic resource. FPGA logic blocks are made of programmable look-up tables (LUTs). Logic blocks and the programmable interconnects are controlled by built-in configuration SRAM cells, which control their operation. The process of programming the FPGA employs transforming the design to a series of zeros and ones, which are transferred to the configuration SRAM cells and are used to configure the programmable logic and interconnects.

A typical CAD flow for FPGAs is shown in Fig. 2. Designs are synthesized by mapping them to the LUT FPGA architecture. Afterwards, the synthesized LUTs are grouped into a group of cluster logic blocks that correspond to that of the physical FPGA structure. In the placement design phase of FPGAs, the logic blocks and the IO blocks of the given design are distributed among the physical logic blocks and IO pads, respectively, in the FPGA fabric. The connections between the placed logic clusters are established by programming the routing resources during the routing phase.

Placement algorithms try to minimize the longest delay along the paths in the circuit and/or the total wire length. In this work, the cost function is chosen to be the total wire length of the design. The wire length is calculated as the length of the bounding box of each wire. For example, if block i is placed in location (x_i, y_i) and block j is placed in location (x_j, y_j), then the length of the wire connecting them is calculated as

$$\text{wirelength} = |x_i - x_j| + |y_i - y_j|. \tag{5}$$

The FPGA placement problem is considered one of the most difficult CAD problems in the very large scale integration (VLSI) design process. It is very difficult to have a mathematical formulation of the problem since the problem does not depend only on finding an optimal placement for the logic blocks to minimize the total wire length. However, the problem has other dimensions, like the resulting design has to be routable. In other words, the routing requirements of the final placement has to be met by the available limited number of routing resources. Hence,

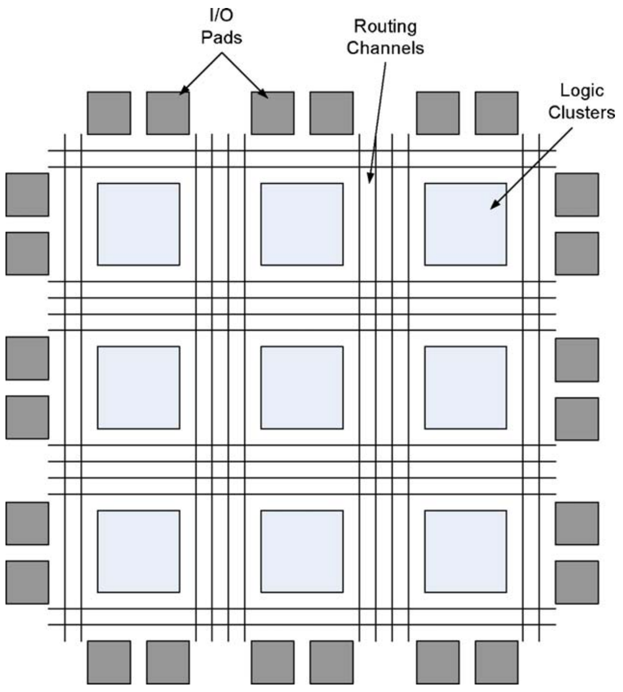

Fig. 1. FPGA fabric architecture.

the problem is not only an assignment problem as it looks in the first instant, but rather a more difficult one. There has been several trials to find an accurate and complete mathematical representation for the FPGA placement problem, however, none of the trials resulted in a complete formulation.

Finding an optimal solution for the FPGA placement problem is very difficult. As a matter of fact, the problem in NP-hard [3]. As a result, all researchers as well as the industry solve the problem using heuristics. One of the most well-known techniques that are used to solve the FPGA placement problem and achieve great results is the versatile place and route (VPR) tool [3]. As a result, the VPR tool has been widely used in the literature as a measure of how good a placement is. As a matter of fact, the VPR tool is the core of the placement tool developed by Altera.

4.1. FPGA placement algorithms in the literature

FPGA placement algorithms can be divided into three different types: partitioning-based algorithm, force-directed algorithms, and simulated-annealing (SA) algorithms.

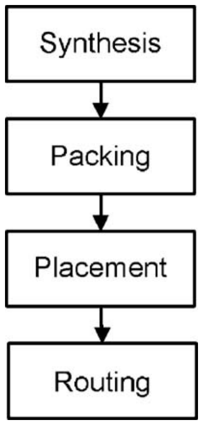

Fig. 2. FPGA design CAD flow.

An example of partitioning-based FPGA placement algorithm is the Fiduccia–Mattheyses (FM) algorithm [21]. The FPGA is divided into two parts, and the FM algorithm is used to distribute the logic blocks among the two halves. Each of the 2 parts is further partitioned into two halves each and so on. This continues until the resulting partitions are small enough. Afterwards, a greedy algorithm is used to place the logic blocks in each partition. The partitioning is performed in order to maximize the attraction, in terms of connections, between the logic blocks and the partitions. Blocks placed in a certain partition are locked and not allowed to move to another partition.

Force-directed placement algorithms are based on the force algorithm presented in Ref. [22]. The logic blocks are initially randomly placed, then each logic block is moved to its best location. The best location is the closest available location to the centroid of all the logic blocks connected to it. The logic block is then locked in its place. After all the logic blocks are locked, they are unlocked and the whole process is repeated until no further improvement in the cost function is observed or a maximum number of iterations is performed.

Simulated-annealing placement algorithms are based on the VPR algorithm [3]. The logic blocks are randomly placed in the FPGA fabric. Afterwards, random blocks are selected as candidates for swapping, a random location is selected and the swap is performed. Swaps that reduce the quality of the cost function are allowed initially, but the probability of accepting such swaps decreases with the number of iterations.

In Ref. [23], a comparison between the quality of the three different placement algorithms is performed. It was found that the VPR-based algorithm results in the best placement quality in terms of the placement cost function. As a result, in this work, the results achieved from the proposed DPSO placement algorithm, which will be presented in Section 5, will be compared to those achieved by the VPR tool.

5. Proposed discrete PSO placement algorithm

In the discrete PSO (DPSO) version used in this work, which is based on [4], each particle position corresponds to the available physical locations in the whole FPGA array. For example, if the FPGA consists of a 10×10 array, the particle position is an array of 100 elements, where each element represents one location in the FPGA array. Hence, the element index is the location number on the FPGA array and its value is the block number occupying that location. Elements denoting empty locations have the value of -1.

As an example, Fig. 3 provides an insight of the DPSO problem formulation for a 3×3 FPGA array. Assume that a certain design has three IOs (*A*, *B*, and *C*) and four logic blocks (*D*, *E*, *F*, and *G*) and that design is to be mapped into the 3×3 FPGA fabric in Fig. 3. The fabric has 9 different logic locations and 24 IO locations. Hence, each particle consists of 33 locations, where each location represent a physical FPGA location. The first 24 locations in the particle can only be occupied by IO blocks and the rest of the locations can only be occupied by logic blocks. If a location in the particle has a "-1" entry, then the corresponding physical location in the FPGA fabric is empty for that particle, as shown in Fig. 3. However, if the location contains a block name, this means that this block is located in that physical location.

In the proposed DPSO formulation, the particle velocity is represented as a sequence of swaps. For example, a velocity $v = \{(1, 4)\}$ represents a swap between the first and fourth elements of the particle position. Since v contains only one swap operation, then the size of v is equal to 1. The two blocks swapped should be of the same type, either logic blocks or IO pins, swapping an IO pin with a logic block is not allowed. During particle initialization, the

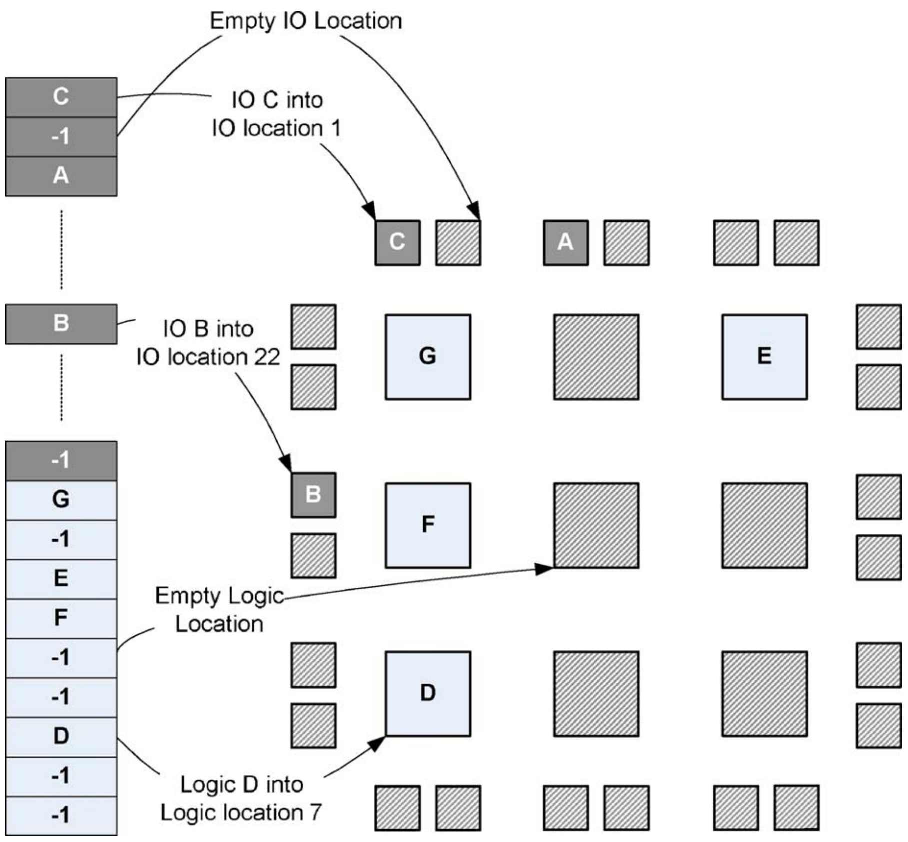

Fig. 3. DPSO problem formulation.

velocities for each particle are initialized with random velocities of random size ranging from 0 to $V_{\max}$. The value of $V_{\max}$ is selected by conducting several experiments with different values for $V_{\max}$ and examining its impact on the value of the cost function.

5.1. DPSO operations

From the conventional PSO relations given in (1) and (2), the position update procedure consists of three different operations; addition of a velocity to a position, subtracting two positions and multiplying a velocity by a constant.

The addition of a velocity v to a position x is carried out by applying the sequence of swaps defined by v to x. This results in another position, which has the same size as the original position x.

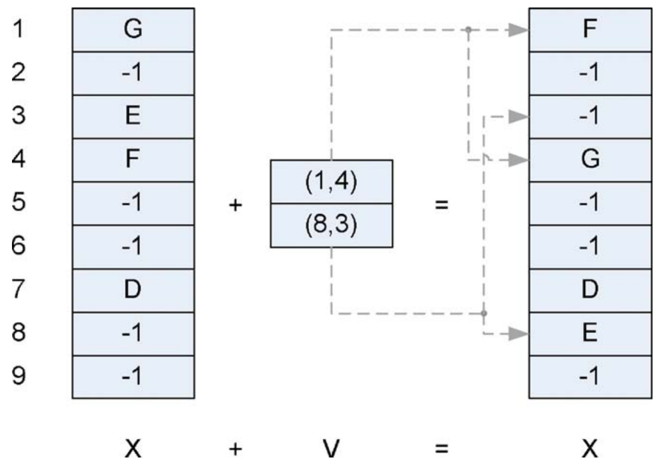

Fig. 4. A position plus velocity example.

The swaps are applied in the same order as they are listed in v. An example for the addition operator is depicted in Fig. 4.

Subtracting a position vector x_1 from another position x_2 is performed by identifying the sequence of swaps v that would transform x_1 to x_2. The maximum complexity of this operation is proportional to the position size, which can grow up significantly depending on the problem size. Hence, the velocity size is clamped to $V_{\max}$, to reduce the computational complexity of the algorithm. As a result, applying the swaps given in v might not result in transforming x_1 to x_2 exactly, if the distance between them is larger than $V_{\max}$. It should be noted that adding v to x_2 will not result in x_1, rather applying the swaps in v in a reverse order will transform x_2 to x_1.

Multiplying a velocity v by a constant c affects the velocity size. If c is equal to zero, the size of v is set to zero. If c is less than 1, v is truncated by removing swaps from the end of the vector such that the size of the resulting velocity is equal to the size of the original v multiplied by c. If c is larger than one, new swaps are added to the

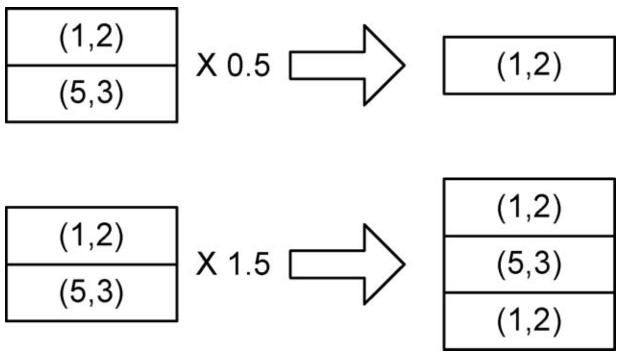

Fig. 5. A constant times velocity example.

end of v to increase its size. The newly added swaps are extracted from the top of v. Fig. 5 shows an example for the multiplication of a velocity by a constant.

5.2. DPSO problem formulation

In the proposed DPSO problem formulation, each position element corresponds to a unique physical location on the FPGA, including both logic block IO pins, as explained above. To make sure that no swaps occur between logic blocks and IO pins, the position vector x of each particle is divided into two vectors x_{Logic} and x_{IO}. Moreover, the velocity vector v of the particle is also composed of two parts v_{Logic} and v_{IO}. To update the position and the velocity of each particle, equations (1) and (2) are applied twice, using the newly defined operators in Section 5.1, for both the logic and IO parts. In order to evaluate the fitness of the new particle position, both parts are used to construct the complete position x. This implementation ensures that the dependency between the logic blocks and IO pins (due to connections between them) is kept intact.

Algorithm 1 shows the pseudocode for the algorithm used for DPSO in this work, where the *stopping criteria* refers to performing a predefined number of function evaluations, which will be explained later.

Algorithm 1 Proposed DPSO implementation.

```
1: Initialize the swarm
2: while stopping criteria not met do
3:     for every particle $i$ do
4:         Update $V^i_{IO}$, $V^i_{Logic}$
5:         Update $X^i_{IO}$, $X^i_{Logic}$
6:         Construct $X^i = [X^i_{IO}, X^i_{Logic}]$
7:         Update $Pbest^i$
8:     end for
9:     Update Gbest
10:    if Gbest is not improved then
11:        Local Minima Avoidance
12:    end if
13: end while
14: return $gbest$
```

5.3. Local minima avoidance

In Ref. [4], the author proposed several approaches to enhance local search ability and avoid premature convergence for discrete PSO algorithms. If the solution does not improve for a specified number of iterations, the different particles move back to their *Pbest* positions and perform a *lazy descent* type of search in order to find a better solution for a limited number of iterations. After that, if the swarm is too small (due to having particles sharing the same position) the swarm is completed with a newly initialized set of particles. More local search strategies were incorporated as well such as *deep descent* and *adaptive descent*.

In our experiments, only *local descent* is used as shown in Algorithm 5.3, as other methods increased the time complexity of the algorithm without resulting in an improvement in the solution quality. The lazy descent method employed in this work is invoked if the global solution, *gbest*, does not improve for a specific number of iterations. It was found that a threshold of 3 non-improving iterations works best for a wide range of the benchmarks tested. The lazy descent generates a random velocity v_{lazy} of size one, which is then added to the particle *pbest*, i.e. only one swap is applied to *pbest*. If the particles best solution is improved, the lazy descent terminates and *pbest* is updated, as well as *gbest*, if needed. Otherwise, the above steps are repeated for a specific number of iterations. If no improved position is found, the lazy descent terminates without changing the original *pbest* position.

In addition to lazy descent, this work employs a *scattering* process to jump over local minima. Similar to the lazy descent, if the *gbest* does not change for a specific number of iterations, the particles positions, not their *pbest* are re-initialized. Instead of scattering (re-initializing) a big number of particles, only the particles that are too close to the *gbest* are scattered. This strategy has a low computational cost while not compromising the provided solution quality. Moreover, the scattering algorithm is only invoked after the number of iterations performed exceeds half the total number of iterations. It was noticed that as the DPSO algorithm progresses, the particles positions tend to group around the *gbest*, thus they are only scattered towards the end of the algorithm.

Algorithm 2 The lazy descent algorithm.

```
1: for a specified number of iterations do
2:     for every particle $i$ do
3:         $X^i = Pbest^i$
4:         Generate $V_{lazy}$
5:         $X^i = X^i + V_{lazy}$
6:         if $f(X^i) < f(Pbest^i)$ then
7:             $Pbest^i = X^i$
8:             Break
9:         end if
10:    end for
11: end for
```

6. Discrete cooperative PSO

A cooperative approach was incorporated into PSO and proposed for continuous non-linear function optimization in Ref. [24] referred to as cooperative PSO (CPSO). The approach relied on optimizing every element of the solution space independently.

Assuming we are trying to optimize a function with a dimensionality of d, instead of using one single swarm for optimizing the solution vector as a whole, CPSO proposes the use of d different swarms with each one optimizing a different element.

In order to evaluate the fitness of a certain particle i in a swarm j, a solution vector is composed by placing the value of that particle in element j in that solution vector while filling the other elements with the *gbest* values of all the other swarms. Hence, the communication between the different swarms is done through the particles' evaluation phase as it needs the *gbest* information of all the swarms.

In the original proposed model, the number of swarms used was equal to the number of the problem dimensions, which means that each swarm was optimizing a single dimension. Hence, the effectiveness of such a model would greatly increase if there are no dependencies between the problem variables, this model was referred to as *CPSO_S*. In addition, the authors also proposed a different model *CPSO_S$_k$*, where each swarm is optimizing a number of k dimensions. Hence, the effectiveness of such a model would greatly increase if there are dependencies between those k problem variables.

The cooperative version studied in this work is based on this approach as we split the elements of the problem to be optimized by different swarms. The optimum splitting choice is to use a separate swarm to optimize a number of blocks that are highly dependent on each other (having a higher number of interconnections). However, we decided to decompose the search space into two sub-spaces, the Logic sub-space and the I/O sub-space. In other words, two discrete PSO swarms are employed, one is optimizing the placement of the Logic blocks while the other optimizes the placement of the I/O blocks. This choice will simplify the cooperative approach as well as naturally preventing the swap between different types of blocks.

To evaluate the fitness of any particle i in the I/O swarm, a solution vector is composed by filling the I/O part with the placement forced by this particle and filling the Logic part with the placement forced by the *gbest* of the Logic swarm. This overall solution vector is evaluated and the fitness value is associated with this particle i. On the other hand, to evaluate the fitness of any particle i in the Logic swarm, a solution vector is composed by filling the Logic part with the placement forced by this particle and filling the I/O part with the placement forced by the *gbest* of the I/O swarm. This overall solution vector is evaluated and the fitness value is associated with this particle i. This approach is highlighted in Fig. 6.

Of course, it is not guaranteed that splitting the problem search space into the Logic and I/O sub-spaces would improve the solution since this is closely related to the degree of dependency between the two sub-spaces. The main advantage though, in addition of keeping the model simple, is that adopting this model will reduce the computational cost required by the original algorithm since all the mathematical operations would be performed using small vectors (sub-spaces) rather than a big vector (the overall solution). Algorithm 6 shows the pseudocode for the algorithm used for DCPSO in this work. The complete solution is always being updated with the *gbest* value of the swarm that has just been evolved insuring this solution will always have the minimum cost.

Algorithm 3 Proposed Cooperative DPSO implementation.

```
1: Initialize the I/O swarm
2: Initialize the Logic swarm
3: while stopping criteria not met do
4:    Update I/O Swarm
5:    Update the complete solution
6:    Update Logic Swarm
7:    Update the complete solution
8: end while
```

7. Results and discussion

7.1. Experimental setup and parameters selection

In order to test the two proposed placement techniques for FPGAs, DPSO and DCPSO, both algorithms are implemented and applied to several standard FPGA benchmark circuits of increased dimensionality. Table 1 lists the benchmark circuits tested and their associated problem sizes, where $P(L, I)$ represents a problem with a total size of P having L Logic locations and I IO locations.

In this work, the performance of the proposed DPSO and DCPSO algorithms in minimizing the cost function is compared to that of the VPR tool. VPR employs an adaptive SA algorithm, where the number of iterations depends on the problem size. In this work, VPR is left to run until it reports its best found solution. In order to have a fair comparison, both DPSO and DCPSO algorithms are allowed to perform the same number of function evaluations performed by VPR. The number of function evaluations performed by the VPR algorithm for every benchmark is listed in Table 1. It can be noticed in the table that the number of function evaluations performed by

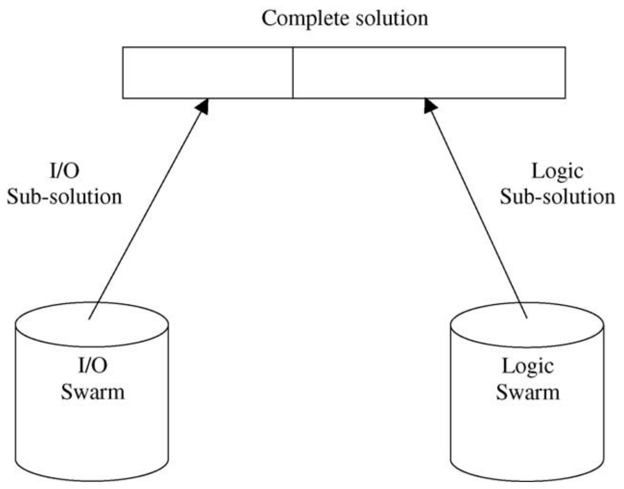

Fig. 6. The discrete cooperative PSO.

VPR is not directly proportional to the problem size. However, the number of function evaluations performed by VPR depends on the difficulty faced by VPR in solving the problem at hand.

In both the DPSO and DCPSO algorithms, c_1 and c_2 of Eq. (1) are set to 2 and w is set to 0.5, similar to [4]. The lazy descent technique is applied if the *gbest* does not improve for 3 iterations and a maximum of 5 steps are performed every time. All the results reported in this work are the averages for the cost function taken over 10 runs.

The value of $V_{\max}$ actually controls the search space limit given to each particle in the swarm. Limiting $V_{\max}$ to a small value speeds up the execution time of each iteration, however, it limits the search space, hence, the quality of the final result. Moreover, increasing the value of $V_{\max}$ beyond the optimum value for each benchmark will strip the particles from their local search abilities. Fig. 7 plots the cost for both the b9 and b01 benchmarks versus $V_{\max}$. As it can be noticed from Fig. 7, there is an optimum value for $V_{\max}$ that yields the lowest cost for each design. Moreover, that optimum value for $V_{\max}$ depends on the problem size of the benchmark.

As a result, in this work, a variable $V_{\max}$ is used to achieve a compromise between the runtime and the quality of the result according to the problem size. Circuits with large problem size will have a bigger $V_{\max}$ than those with smaller problem size. The formula used to update the value of $V_{\max}$ in this work is given below

$$V_{\max} = 0.1067 \times P + 20.92 \quad (6)$$

where P is the problem size. The values of the constants given in Eq. (6) are evaluated empirically by running the algorithms with several values of $V_{\max}$ and the values that achieved the best cost

Table 1
Problem sizes for the different benchmarks used.

Benchmark	Problem size	Max evaluations
cm42a	36(4, 32)	16,441
lion	57(9, 48)	5,585
b02	57(9, 48)	5,270
daio	57(9, 48)	5,312
dk27	80(16, 64)	6,843
b01	80(16, 64)	13,932
my_adder	80(16, 64)	98,982
count	80(16, 64)	98,231
s208.1	132(36, 96)	31,023
b9	132(36, 96)	107,070
s832	225(81, 144)	135,880
s967	260(100, 160)	204,920
ex5p	561(289, 271)	800,382
apex4	665(361, 304)	850,377

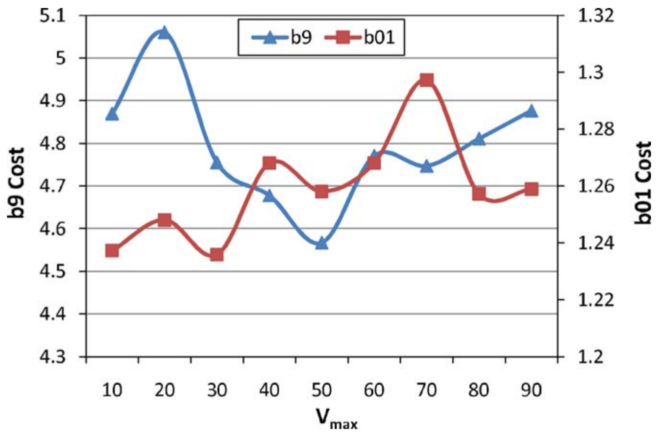

Fig. 7. The impact of V_{max} on the cost function for the b9 and b01 benchmarks using the DPSO algorithm.

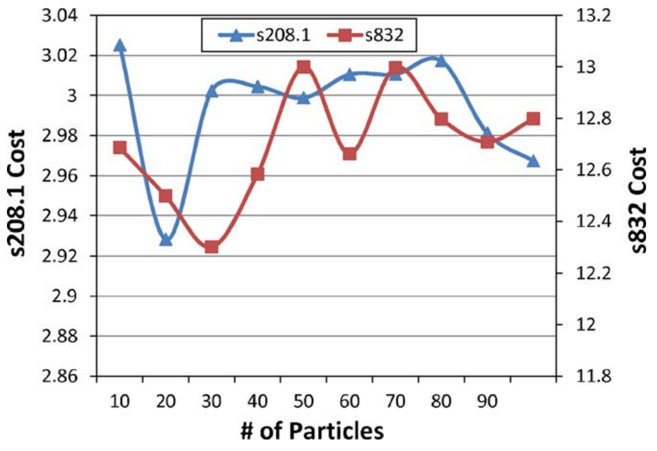

Fig. 9. The impact of the number of particles on the cost function for the s208.1 and s832 benchmarks using the DPSO algorithm.

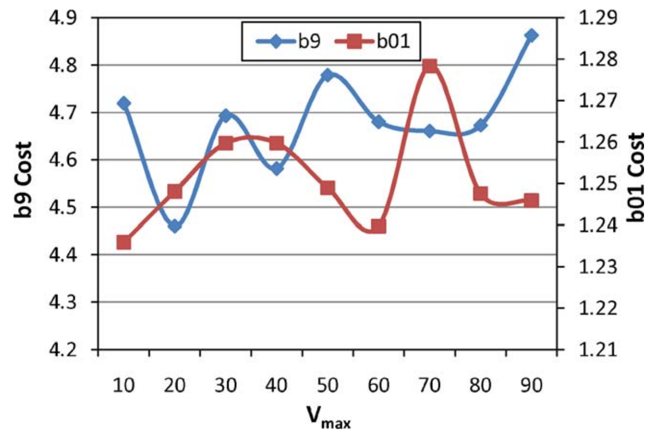

Fig. 8. The impact of V_{max} on the cost function for the b9 and b01 benchmarks using the DCPSO algorithm.

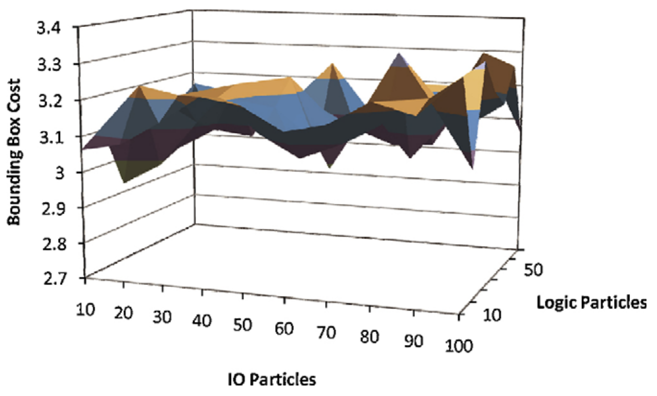

Fig. 10. The impact of the number of particles on the cost function for the s208.1 benchmark using the DCPSO algorithm.

function are recorded in each case. Afterwards, the values of the optimum V_{max} are fitted into a line equation with respect to the problem size P.

A similar analysis was performed for the DCPSO algorithm and similar relationships were derived. Fig. 8 plots the change in the bounding box cost for the b9 an b01 benchmark. Similarly, it can be noticed that increasing the problem size increases the value of V_{max} that yields the optimum cost. The linear relation between the value of V_{max} and the problem size used in this work is given by

$$V_{max} = 0.026 \times P + 38.94 \tag{7}$$

Again, Eq. (7) was derived empirically by performing several runs using several values for V_{max} followed by performing linear regression over the resulting cost.

The other parameter that needs tuning is the number of particles used inside each swarm. Increasing the number of particles inside the swarm allows the swarm to explore more space, especially if the particles are initialized to be distributed

Table 2
VPR and DPSO results for several FPGA benchmarks.

Benchmark	VPR		DPSO		Error
	Mean	Std.	Mean	Std.	Margin (%)
cm42a	0.4286	0.0067	**0.4256**	0	−0.7
lion	0.6088	0.0082	**0.6009**	0.0086	−1.30
b02	0.5649	0.0060	**0.5603**	0.0007	−0.81
daio	0.6120	0.0175	**0.5900**	0.008	−3.59
dk27	0.9659	0.0161	**0.9428**	0.008	−2.39
b01	1.2461	0.0076	**1.2359**	0.019	−0.82
my_adder	2.091	0.0373	**2.090**	0.0605	−0.05
count	2.1592	0.04197	**2.1342**	0.0833	−1.16
s208.1	2.973	0.0242	**2.928**	0.0544	−1.51
b9	4.4795	0.0325	4.5672	0.109	1.95
s832	**11.7551**	0.0783	12.3020	0.260	4.65
s967	**18.6722**	0.0869	19.4568	0.326	4.20
ex5p	**103.1914**	0.3329	108.9920	0.882	5.62
apex4	**117.049**	0.4911	123.333	3.104	5.37

Table 3
DCPSO results for several FPGA benchmarks.

Benchmark	DCPSO	
	Mean	Std.
cm42a	0.4256	0.0048
lion	0.6009	0.0083
b02	0.5603	0.0045
daio	0.590	0.0139
dk27	0.9427	0.0115
b01	1.2359	0.0121
my_adder	2.050	0.059
count	2.0942	0.0435
s208.1	2.925	0.0593
b9	4.4602	0.0976
s832	12.0874	0.229
s967	19.1277	0.291
ex5p	108.206	2.343
apex4	122.75	1.201

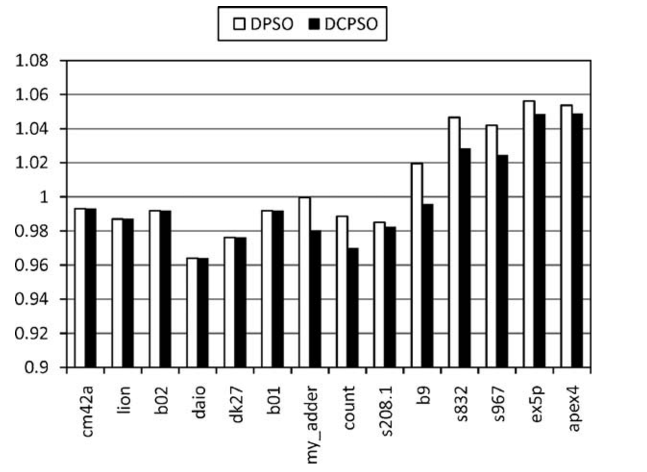

Fig. 11. Relative bound box cost function of both the DPSO and DCPSO algorithms normalized to the VPR cost.

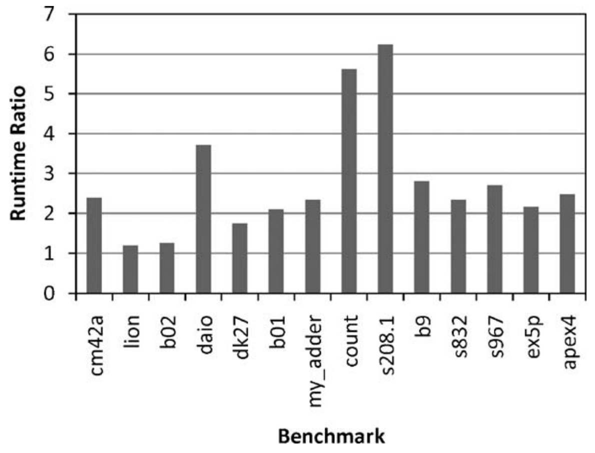

Fig. 12. Runtime ratio between DPSO and DPCSO.

uniformly along the whole search space. However, since the total number of function evaluations for each benchmark is known in advance, then increasing the number of particles will reduce the number of iterations performed by each particle, hence, limiting the exploration performed by each particle. On the other hand, decreasing the number of particles inside each swarm might result in under exploration for the search space, especially if $V_{\max}$ is not large enough. Fig. 9 plots the change in the cost function for the s208.1 and s832 benchmarks for a range of number of particles. From Fig. 9, it can be deduced that there is an optimum value of the number of particles that should be used for each benchmark that is proportional to the problem size at hand.

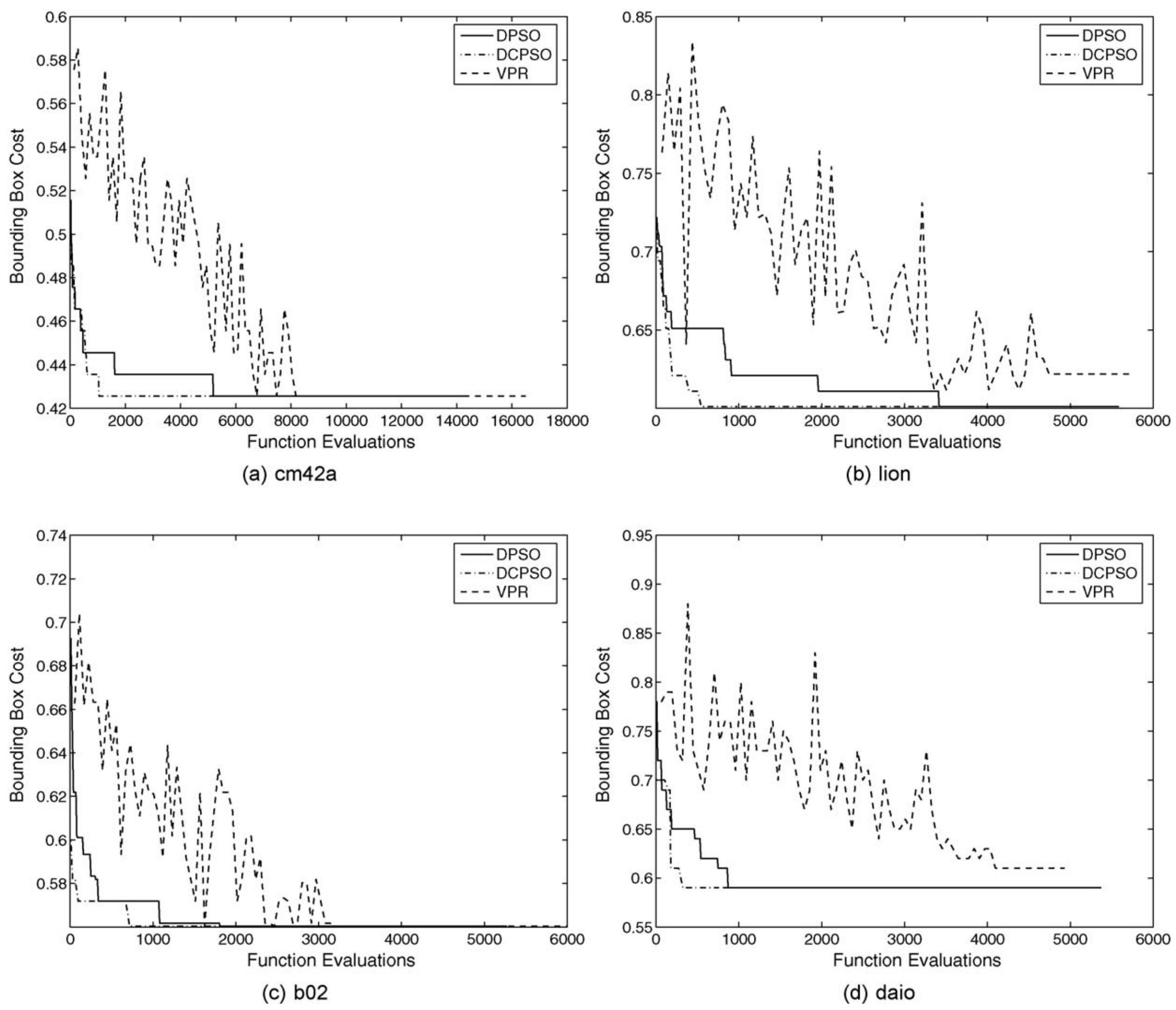

Fig. 13. Convergence behavior for problems with a dimensionality within 60.

Similar to $V_{\max}$, the number of particles is evaluated dynamically according to the problem size using the following formulation:

$$\text{num\_particles} = 0.033 \times P + 13.37 \quad (8)$$

The constants in Eq. (8) are evaluated empirically by curve fitting the optimum number of particles to a straight line equation against the problem size.

Using similar procedure, a relationship for the number of particles that yields the minimum cost value is derived for the DCPSO algorithm. However, in that case, the regression is performed separately over the number of particles used in the IO swarm and Logic swarm. Fig. 10 plots the bounding box cost function for the s208.1 benchmark using several values for the number of particles for the IO and Logic swarms. The relationships used in this work to calculate the number of particles used in the DCPSO algorithm are given by

$$\text{num\_particles}_{\text{IO}} = 0.016 \times P + 26.45 \quad (9)$$

$$\text{num\_particles}_{\text{Logic}} = 0.026 \times P + 38.94 \quad (10)$$

7.2. Experimental results

7.2.1. DPSO and DCPSO results

The results of applying VPR and DPSO on several FPGA benchmarks are shown in Table 2, where the better results are highlighted in bold. It can be seen that the DPSO approach produces better results than the VPR placement tool in problems with a dimensionality up to 132. When the dimensionality increases up to 665, the DPSO produces results that are within a 5% margin from the results supplied by VPR.

Table 3 lists the average bounding box cost function and the standard deviation achieved over 10 runs by applying the DCPSO algorithm on the FPGA placement problem. By inspecting the results in Table 3, it can be noticed that the DCPSO outperforms the DPSO algorithm in all the benchmarks tested.

Fig. 11 plots the average bounding box cost achieved by both the DPSO and DCPSO algorithms for all the benchmarks tested normalized to the cost achieved by the VPR tool. From Fig. 11, it can be deduced that the DCPSO algorithm is still unable to outperform the VPR tool for large problem sizes, however, it is much closer to the results achieved by VPR than the DPSO algorithm.

Fig. 12 plots the ratio between the runtime of the DPSO algorithm and the DCPSO algorithm ($T_{\text{DPSO}}/T_{\text{DCPSO}}$). From Fig. 12, it can be noticed that on average, the DCPSO algorithm is faster than the DPSO algorithm by 2.7 $\times$. Moreover, in some benchmarks, the DCPSO can be as fast as 6 $\times$ when compared to the DPSO algorithm.

Figs. 13–15 plot the convergence behavior of VPR, DPSO, and DCPSO for all the benchmarks tested in this work. It can be seen that both DPSO and DCPSO have very similar behaviors in most of the benchmarks and they tend to reach their minimum cost at the same iteration number. The DPSO and DCPSO algorithms always

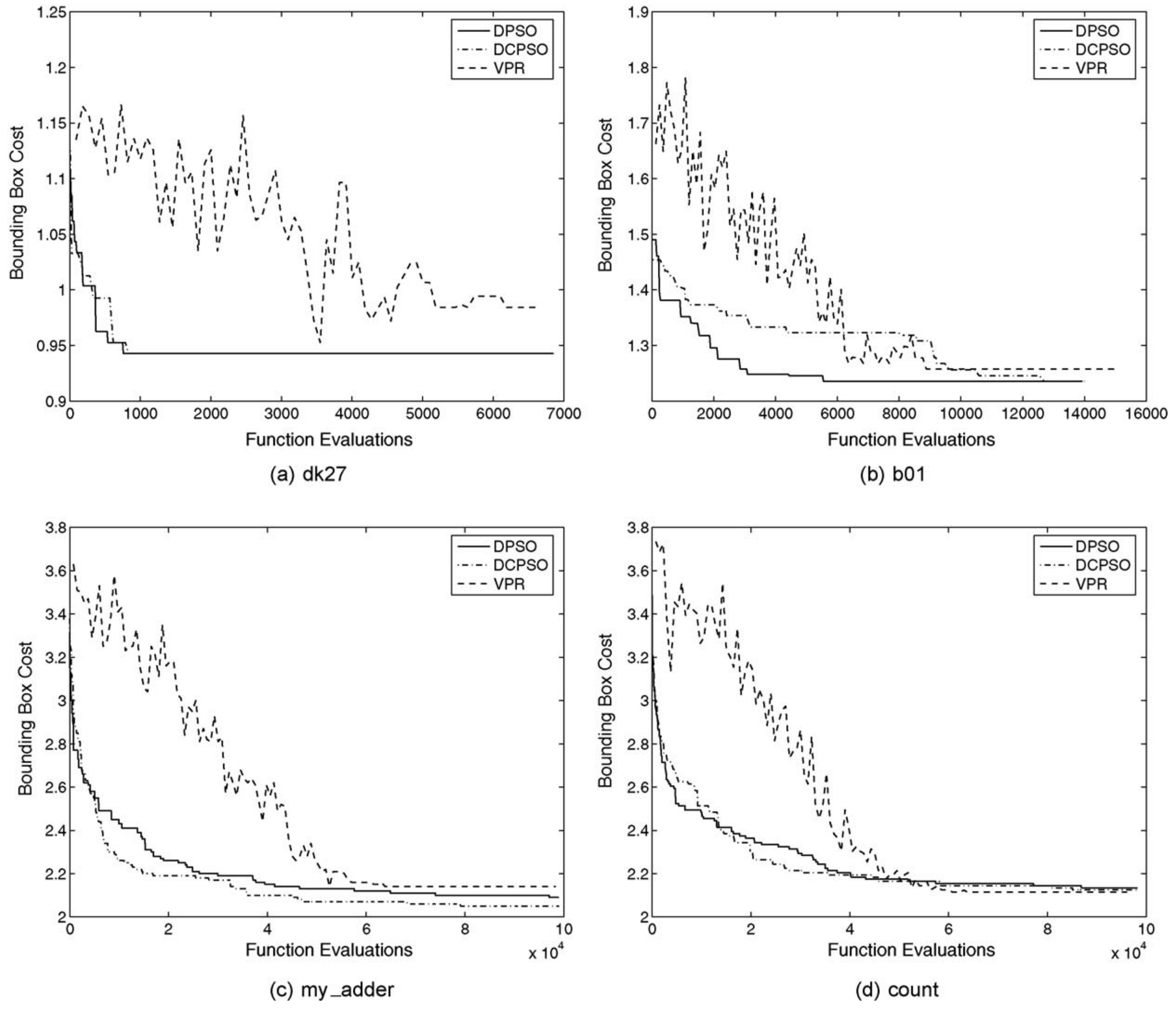

Fig. 14. Convergence behavior for problems within a dimensionality of 80.

start from a better quality solution due to the initialization of a population of candidate solutions. Both of them quickly identify good regions in the search space and move to these regions much faster than VPR. However, these algorithms fail to fine tune the final solution using local search. Although the local descent was used to improve the local search capabilities of PSO, it was not enough to improve the DPSO and DCPSO behavior in large benchmarks. However, one of the main strengths of proposed DPSO and DCPSO algorithm, is their abilities to converge to a good solution using a fewer number of iterations compared to VPR. On the other hand, VPR manages to fine tune the solution better than DPSO and DCPSO because the SA algorithm is superior in local

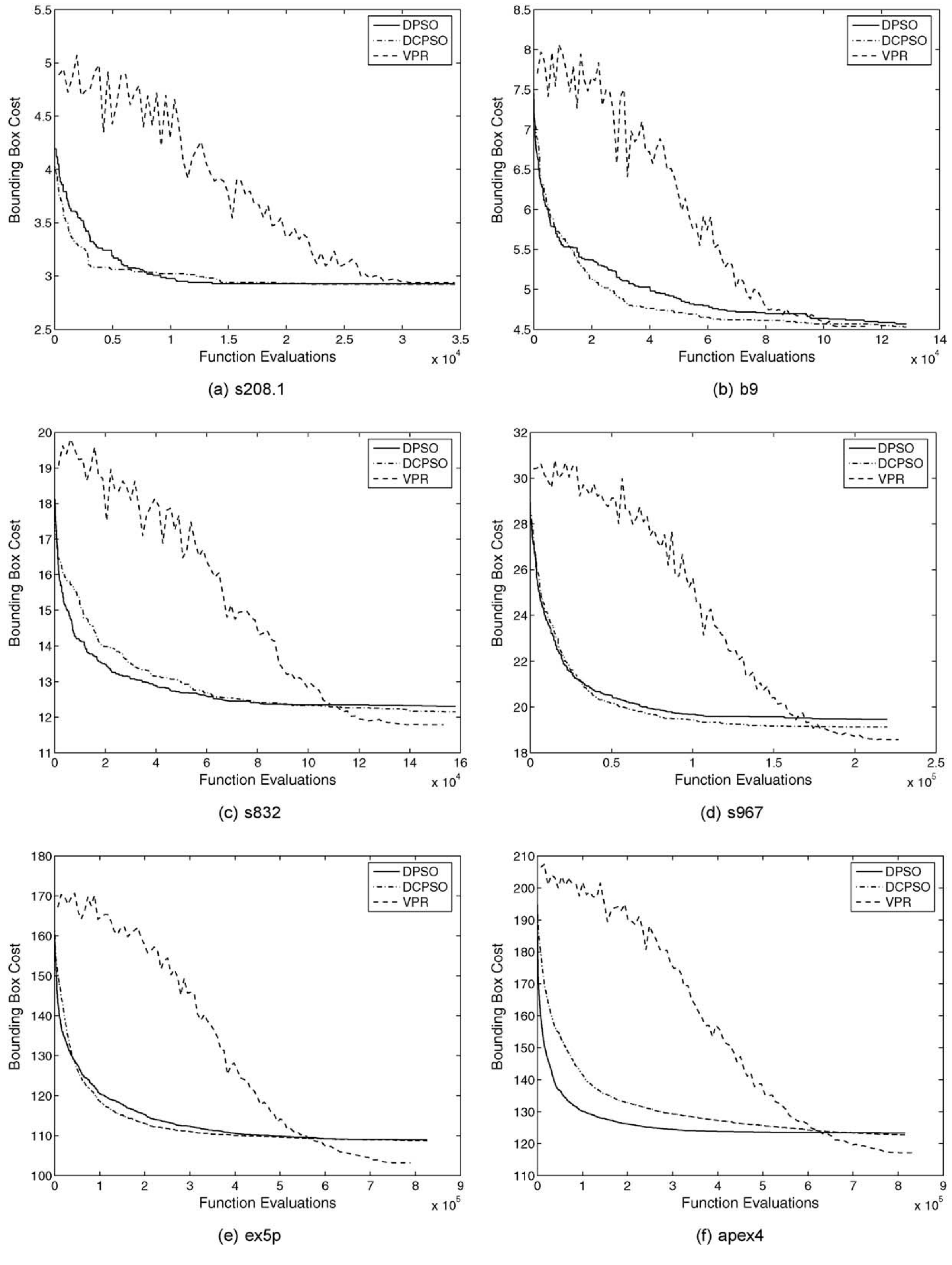

Fig. 15. Convergence behavior for problems with a dimensionality above 100.

search. Thus, it can be deduced that if convergence speed can be traded for the quality of solution, DPSO and DCPSO would be the best placement algorithm to be used. Moreover, the DPSO and DCPSO algorithms can be used as a starting point for SA-based placement techniques to converge to a good solution quickly, then SA is invoked to perform local search around the solution reached by either DPSO or DCPSO.

8. Conclusions

This paper introduced a discrete PSO (DPSO) algorithm applied to the FPGA placement problem. The proposed DPSO placement algorithm is applied to several FPGA benchmarks with increased dimensionality and compared to the academic VPR placement tool, which is based on simulated annealing.

The work provided extensive studies to appropriately set the $V_{\max}$ and the number of particles parameters used by DPSO. The results show that DPSO outperforms VPR for benchmarks with sizes within 132. In larger benchmarks, the error margin is within 5% for problems with sizes up to 665.

The work also proposed the use of a cooperative DPSO version where the placement of the I/O and logic block is being optimized by different swarms. The experimental results show that the cooperative version produces better results than DPSO while substantially reducing the computational cost.

In future work, it is intended to enhance the performance of the DPSO by incorporating several approaches to escape local minima. This is expected to improve the results for large benchmarks by providing the DPSO and DCPSO placement algorithms with fine tuning ability. Another direction is the use of a hybrid algorithm where DPSO is used to fastly identify and converge to good regions of the search space then, SA is used to fine tune the solutions found in these areas.

References

[1] J. Kennedy, R.C. Eberhart, Particle swarm optimization, in: Proceedings of the IEEE International Conference on Neural Networks, vol. 4, 1995, pp. 1942–1948.
[2] R.C. Eberhart, J. Kennedy, A new optimizer using particle swarm thoery, in: Proceedings of the 6th International Symposium on Micro Machine and Human Science, 1995, pp. 39–43.
[3] V. Betz, J. Rose, A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, Norwell, MA, 1999.
[4] M. Clerc, Discrete particle swarm optimization, in: New Optimization Techniques in Engineering, Springer–Verlag, 2004.
[5] K.-P. Wang, L. Huang, C.-G. Zhou, W. Pang, Particle swarm optimization for traveling salesman problem, in: Proceedings of the Second International Conference on Machine Learning and Cybernetics, 2003, pp. 1583–1585.
[6] X.H. Zhi, X.L. Xing, Q.X. Wang, L.H. Zhang, X.W. Yang, C.G. Zhou, Y.C. Laing, A disrete PSO method for generalized TSP problem, in: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, vol. 4, 2004, pp. 2378–2383.
[7] C. Wang, J. Zhang, J. Yang, C. Hu, J. Liu, A modified particle swarm optimization algorithm and its applications for solving travelling salesman problem, in: Proceedings of the International Conference on Neural Networks and Brain, ICNN, vol. 2, 2005, pp. 689–694.
[8] A. Salman, A. Imtiaz, S. Al-Madani, Particle swarm optimization for task assignment problem, Microprocessors and Microsystems 26 (8) (2002) 363–371.
[9] X. Zeng, Y. Zhu, L. Nan, K. Hu, B. Niu, X. He, Solving weapon-target assignment problem using discrete particle swarm optimization, in: Proceedings of the IEEE World Congress on Intelligent Control and Automation, vol. 1, 2006, pp. 3562–3565.
[10] Z. Lian, X. Gu, B. Jiao, A dual similar particle swarm optimization algorithm for job-shop scheduling with penalty, in: Proceedings of the IEEE World Congress on Intelligent Control and Automation, vol. 2, 2006, pp. 7312–7316.
[11] Q. Pan, F. Tasgetiren, Y. Liang, A discrete particle swarm optimization algorithm for single machine total earliness and tardiness problem with a common due date, in: Proceedings of the IEEE Congress on Evolutionary Computation, 2006, pp. 3281–3288.
[12] S. Chandrasekaran, S.G. Ponnambalam, R.K. Suresh, N. Vijayakumar, An application of particle swarm optimization algorithm to permutation flowshop scheduling problems to minimize makespan, total flowtime and completion time variance, in: Proceedings of the IEEE International Conference on Automation Science and Engineering, 2006, pp. 513–518.
[13] S. Chandrasekaran, S.G. Ponnambalam, R.K. Suresh, N. Vijayakumar, A hybrid discrete particle swarm optimization algorithm to solve flow shop scheduling problems, in: Proceedings of the IEEE Conference on Cybernetics and Intelligent Systems, 2006, pp. 1–6.
[14] X. Kong, J. Sun, W. Xu, Permutation-based particle swarm algorithm for tasks scheduling in heterogeneous systems with communication delays, International Journal of Computational Intelligence Research 4 (1) (2008) 61–70.
[15] Y. Xue, Q. Yang, J. Feng, Improved particle swarm optimization algorithm for optimum steelmaking charge plan based on the pseudo TSP solution, in: Proceedings of the Fourth International Conference on Machine Learning and Cybernetics, vol. 9, 2005, pp. 5452–5457.
[16] W. Jian, Y. Xue, Modified particle swarm optimization algorithm for steelmaking charge plan based on the pseudo TSP model, in: Proceedings of the 31st Annual Conference of IEEE Industrial Electronics Society, IECON, 2005, pp. 2273–2277.
[17] M. Neethling, A. Engelbrecht, Determining RNA secondary structure using set-based particle swarm optimization, in: Proceedings of the IEEE Congress on Evolutionary Computation, 2006, pp. 1670–1677.
[18] A. Moraglio, C.D. Chio, J. Togelius, R. Poli, Geometric particle swarm optimization, Journal of Artificial Evolution and Applications 2008 (2008) 1–14.
[19] Z. Lee, S. Su, C. Lee, Efficiently solving general weapon-target assignment problem by genetic algorithms with greedy eugenics, IEEE Transactions on System, Man, and Cybernetics, Part B 33 (1) (2003) 113–121.
[20] V.G. Gudise, G.K. Venayagamoorthy, Swarm intelligence for digital circuits implementation on field programmable gate arrays platforms, in: NASA/DoD Conference on Evolvable Hardware, 2004, 83–91.
[21] C.M. Fiduccia, R.M. Mattheyses, A linear time heuristic for improving network partitions, in: Proceedings of the IEEE/ACM Design Automation Conference, 1984, pp. 175–181.
[22] K. Shahookar, P. Mazumder, VLSI cell placement techniques, ACM Computing Surveys 23 (2) (1991) 143–220.
[23] C. Mulpuri, S. Hauck, Runtime and quality tradeoffs in FPGA placement and routing, in: Proceedings of the ACM International Symposium on FPGAs, 2001, pp. 29–36.
[24] F. van den Bergh, A.P. Engelbrecht, A cooperative approach to particle swarm optimization, in: Proceedings of the IEEE Transactions on Evolutionary Computation, vol. 8(3), 2004, pp. 225–239.