Engineering Applications of Artificial Intelligence 23 (2010) 177–187

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

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

Parallel scalable hardware implementation of asynchronous discrete particle swarm optimization

Amin Farmahini-Farahani *, Shervin Vakili, Sied Mehdi Fakhraie, Saeed Safari, Caro Lucas

School of Electrical and Computer Engineering, University of Tehran, North Kargar Ave., Tehran 14395-515, Iran

ARTICLE INFO

Article history:
Received 13 June 2007
Received in revised form
17 August 2009
Accepted 1 December 2009
Available online 12 January 2010

Keywords:
Evolutionary algorithms
Particle swarm optimization
Parallel architecture
Multiprocessing
Field programmable gate array (FPGA)
System-on-a-programmable-chip
Real-time applications

ABSTRACT

This paper presents a novel hardware framework of particle swarm optimization (PSO) for various kinds of discrete optimization problems based on the system-on-a-programmable-chip (SOPC) concept. PSO is a new optimization algorithm with a growing field of applications. Nevertheless, similar to the other evolutionary algorithms, PSO is generally a computationally intensive method which suffers from long execution time. Hence, it is difficult to use PSO in real-time applications in which reaching a proper solution in a limited time is essential. SOPC offers a platform to effectively design flexible systems with a high degree of complexity. A hardware pipelined PSO (PPSO) Core is applied with which the required computational operations of the algorithm are performed. Embedded processors have also been employed to evaluate the fitness values by running programmed software codes. Applying the subparticle method brings the benefit of full scalability to the framework and makes it independent of the particle length. Therefore, more complex and larger problems can be addressed without modifying the architecture of the framework. To speed up the computations, the optimization architecture is implemented on a single chip master–slave multiprocessor structure. Moreover, the asynchronous model of PSO gains parallel efficacy and provides an approach to update particles continuously. Five benchmarks are exploited to evaluate the effectiveness and robustness of the system. The results indicate a speed-up of up to 98 times over the software implementation in the elapsed computation time. Besides, the PPSO Core has been employed for neural network training in an SOPC-based embedded system which approves the system applicability for real-world applications.

1. Introduction

Evolutionary algorithms (EAs) are general-purpose search algorithms used to solve difficult numerical optimization problems by simulating natural evolution over populations of candidate solutions (Schwefel, 1981; Holland, 1975; Fogel, 1994; Bäck et al., 1997). Numerical optimization has been widely used in engineering to solve a variety of NP-complete problems in areas such as structural optimization, neural network training, layout and scheduling problems, and control system analysis and design (Fogel, 1991; Bäck et al., 1997; Zitzler et al., 2000; Freitas, 2002; Bäck, 1996; Deb, 2001; Dasgupta and Michalewicz, 1997). Particle swarm optimization (PSO) is one of the emerging computation techniques that was developed in 1995 (Kennedy and Eberhart, 1995) as an evolutionary optimization methodology over a complex solution space. PSO deals with the concept of social interaction. It was inspired by the social behavior of bird flocking or fish schooling. The PSO algorithm exploits the gathered information of the particles in a swarm during their food-searching activities and affects the trajectory of particles. Each particle flies through the search space with a velocity which is dynamically adjusted according to its own experience as well as the experiences of its neighbors. Therefore, the particles have a tendency to fly towards the better and better search area over the progress of search process.

Like the other evolutionary computational techniques, PSO is a derivative-free, stochastic and population-based search algorithm which is initialized with a population (swarm) of random solutions, called particles. Unlike the other evolutionary computation techniques, each particle in PSO is also associated with a velocity.

The PSO-based approaches converge faster than genetic-algorithm-(GA)-based techniques, and require less computational complexity (Song and Gu, 2004). PSO has few parameters to adjust and general values for these parameters are not devastative (Carlisle and Dozier, 2001). Moreover, PSO is well suited to large-scale

* Corresponding author. Silicon Intelligence and VLSI Signal Processing Laboratory of School of Electrical and Computer Engineering, University of Tehran, North Kargar Ave., Tehran 14395-515, Iran. Tel.: +98 21 88013196; fax: +98 21 88778690.

E-mail addresses: a.farmahini@ece.ut.ac.ir (A. Farmahini-Farahani), s.vakili@ece.ut.ac.ir (S. Vakili), fakhraie@ut.ac.ir (S.M. Fakhraie), saeed@ut.ac.ir (S. Safari), lucas@ipm.ir (C. Lucas).

0952-1976/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.engappai.2009.12.001

and complex optimization problems and is mainly used in NP-complete problems (Hu et al., 2004). PSO is inherently parallel since each particle can be considered as an independent agent (Schutte et al., 2004). However, the iterative evolution process of PSO like other heuristic algorithms is time consuming. For many real-world applications, PSO can run for days, even when it is executed on a high performance workstation. Computational parallelism is an applicable approach to alleviate the problem of the prolonged execution time of PSO.

Particle swarm engines are one instance of the bio-inspired computing systems that is employed in the embedded systems. Considering the real-time requirements for embedded applications, most embedded processor cores lack the performance to run particle swarm computations or to emulate other bio-inspired subsystems. Therefore, employing particle swarm in embedded applications requires efficient custom hardware intellectual property (IP) cores implemented for them (Mathew et al., 2004). On the other hand, the competitive market of embedded systems requires solutions that take shorter time in design, are cost-efficient in development, have flexibility in utilization, expose simplicity in integration, and exhibit reusability (Zhang et al., 2001). A particle swarm IP core for embedded systems should offer these capabilities as an off-the-shelf component. Nevertheless, obtaining an optimal solution (if exists) in real time for large-scale problems is difficult. Since making a decision in a limited time is vital for many practical problems, obtaining a suitable solution in real time is much better than finding the optimal solution off-line. Thus, our objective is to find appropriate solution in a limited time.

The complexity of the modern chips is rising and fundamental changes in system design are being more essential. The system-on-a-programmable-chip (SOPC) concept is bringing a major revolution in the design of integrated circuits, due to the flexibility it provides and the complexity it caters to. The SOPC embedded systems refer to the packaging of all the necessary electronic functions, memory blocks, interfaces, microprocessors, and so forth of different functions onto a single chip to form a complete electronic system. SOPC brings the combination of programmable logic and embedded processors, mixing software and hardware. Thus, SOPC technology allows all of the various components to be integrated together on a chip rather than connecting those components on a circuit board to construct an electronic system.

In this paper, an on-chip multiprocessing SOPC-based architecture has been proposed for high performance realization of the particle swarm algorithm in embedded systems to speed up its calculations. The proposed architecture can obtain high computational power due to its parallel processing architecture. In addition, the software fitness evaluator along with the parameterized hardware PSO Core provide a scalable framework for a range of discrete PSO applications. Hence, the architecture is intended to facilitate the use of PSO-based techniques in embedded systems. The implemented system performs fitness evaluation in software and all other PSO operations in hardware. In addition, the employed master–slave parallel system uses an asynchronous and discrete model of PSO. The proposed system has been implemented on an Altera Stratix Development Kit, and its performance has been compared with that of the corresponding software implementation. Test beds are MaxOne problem and optimizing four classic arithmetic functions.

The remaining sections are organized as follows: Section 2 reviews the previous work on applications and hardware implementations of the PSO algorithm. Section 3 gives an overview of the basis of PSO, different ways to deal with particles, and parallel characteristics of PSO. The proposed framework is described in Section 4. Also, the implementation of the system and details of the employed architecture are described in this section. Section 5 explains the experimental results over five benchmarks and compares the results with the software-based implementation as well as pure-hardware implementation. Section 6 presents an adoption of our PSO implementation for neural network training as a case study of a real-world application. Section 7 concludes the paper and explains our future work.

2. Previous work

Particle swarm optimization is a new population-based stochastic optimization technique. More and more researchers are interested in this new algorithm and it has been investigated from various perspectives (Song and Gu, 2004; Hu et al., 2004). To reduce the execution time of heuristic algorithms, several methods have been offered, including parallel and/or distributed processing of these algorithms along with their hardware implementation (Konfrst, 2004; Chen et al., 2005; Calaor et al., 2002; Hidalgo et al., 2003).

Different types of parallel implementations of PSO have been introduced in literature. Early parallel PSO implementations have employed synchronous evolution methods (Schutte et al., 2004; Chang et al., 2005; Cui and Weile, 2005; Jin and Rahmat-Samii, 2005), while asynchronous evolution has been emphasized recently (Koh et al., 2006; Venter and Sobieszczanski-Sobieski, 2005). Parallel implementations of PSO are mostly based on cluster computing and message passing interface protocol (Koh et al., 2006; Schutte et al., 2004, 2003; Venter and Sobieszczanski-Sobieski, 2005; Gies and Rahmat-Samii, 2003; Jin and Rahmat-Samii, 2005).

Although different hardware approaches have been proposed for GA so far (Zhu et al., 2006), a few hardware-based implementations of PSO have been reported. In Kokai et al. (2006), the authors have used hardware implementation of PSO in an FPGA for the employment with blind adaptation of the directional characteristic of array antennas. They have introduced multi-swarm architecture in which each single swarm optimizes only a single parameter of the application. However, they have not mentioned the hardware implementation cost, achieved frequency, and also performance.

In Reynolds et al. (2005), the authors have implemented a modified particle swarm optimizer and neural network in FPGA. In their architecture, many of the computations are preformed in parallel to reduce computation time as compared to software implementation. They have employed two Xilinx XC2V6000 FPGAs. One FPGA was used for fitness function calculations, and the other was used for the particle swarm operations. At 100 MHz, their implementation was about 60 times faster than software implementation. Also, they have not reported the hardware cost.

In Pena et al. (2006), a hybrid swarm optimization technique has been offered to accommodate embedded or hardware dedicated applications. Their approach does not require multiplication and consists of a population of neural networks in an FPGA that are evaluated in an embedded processor and are trained by the proposed algorithm. In Pena and Upegui (2007), they have presented an architecture for a hardware-friendly version of PSO. The architecture is composed of a number of particle computation blocks connected to each other to shape a ring topology. Each computation block consists of memory units, a 6-stage pipelined particle update unit, and a personal best update unit. Each computation block updates a particle, performs particle computation, and is connected to a fitness evaluation block. All the particles in the swarm are evaluated and updated in parallel, increasing hardware cost linearly with the population size. The architecture has been implemented on a Xilinx Virtex-4

LX FPGA running at 160 MHz. Updating particles in this architecture (excluding fitness computation and other calculations) is 8600 times faster than the software implementation in MATLAB.

In Duren et al. (2007), the authors have used the PSO algorithm implemented on an SRC-6e reconfigurable computer to invert the neural network to search for a set of inputs to the network. The network is implemented in one FPGA and a 3-stage pipelined PSO is implemented in a separate Virtex-2 FPGA running at 100 MHz. According to the master–slave relationship, PSO acts as the master and the network, used to evaluate fitness, acts as the slave. By using a hardware fitness evaluator, they have achieved a speed-up of a factor 65 over a conventional personal computer.

3. Particle swarm optimization

3.1. Particle swarm optimizer algorithm

The particle swarm optimizer is a swarm intelligence algorithm that emulates a flock searching over the solution landscape by sampling points and converging the swarm on the most promising regions. A particle moves through the solution space along with a trajectory defined by its velocity. The structure of a particle is significantly more complex than that of a member of a GA population. A particle consists of five components:

- x, a vector containing the current location in the solution space. The size of x is dictated by the number of variables used by the problem being solved.
- *fitness*, the quality of the solution represented by the vector x, as computed by a problem-specific evaluation function.
- v, a vector containing the velocity for each particle. The velocity of a particle indicates the changes of the corresponding x vector (particle location) in the next iteration. Altering the v vector values changes the movement direction of the particle through the search space.
- $fitness_p$ is the fitness value of the best solution yet encountered by a particular particle. Each particle keeps track of its coordinates in the problem space, which are associated with the best solution (fitness) it has achieved so far.
- *pbest* (*personal best*) is a copy of x for the location of the best solution yet encountered by a particular particle.

Each particle is also aware of *gbest* (*global best*), the particle reporting the current best fitness in the neighborhood for any given iteration. A neighborhood may consist of some small group of particles. Alternately, the entire swarm may be considered as a single neighborhood, and *gbest* is applied globally (global PSO). Also, $fitness_g$ is the fitness value of the *gbest*.

The PSO begins with a random population and searches for fitness optimum just like the GA. But in the PSO algorithm, particles will evolve by cooperation and competition among the individuals themselves through iterations instead of using genetic operators (Shi and Eberhart, 1998).

The heart of the PSO algorithm is the process by which v is modified, forcing the particles to search through the most promising areas of the solution space recurrently. At each time step, the PSO concept consists of velocity changes for each particle toward its *pbest* and *gbest* locations. Moving toward these locations is weighted by a random term, with separate random numbers being generated for acceleration of particle movements toward *pbest* and *gbest* locations. This randomness insures that the step size will be varying to avoid aliasing. It also insures that the particle does not get trapped into local optima, where the particle endlessly follows the exact same path. The modified velocity and location of each individual particle can be calculated using the current velocity and the distance to $pbest_{id}$ and to $gbest_d$ (i, d, and k represent particle, dimension, and iteration, respectively), as shown in the following equations.

$$v_{id}^{k+1} = w \cdot v_{id}^{k} + c_1 \cdot r_1 \cdot (pbest_{id} - x_{id}^{k}) + c_2 \cdot r_2 \cdot (gbest_d - x_{id}^{k}) \quad (1)$$

$$x_{id}^{k+1} = x_{id}^{k} + v_{id}^{k+1} \quad (2)$$

where x_{id}^{k} is the current location of the particle i at the iteration k, which has v_{id}^{k+1} as the velocity vector. This velocity satisfies $V_{min} \le v_{id}^{k+1} \le V_{max}$. Besides, there are five parameters that should be defined, w is the inertia weight factor, c_1 and c_2 are acceleration constants, and r_1 and r_2 are uniform random numbers between 0 and 1.

As Eq. (1) shows velocity changes of a particle consist of three parts. The first part is known as the "momentum" component and represents a proportional movement to the previous velocity. This part is the tendency of the particle to continue in the previous direction. On each iteration, the previous values of v constitute the particle's momentum. This momentum is essential, as it is the feature of PSO that allows particles to escape local extrema. The second part, known as the "cognitive" component, attracts the particles toward their own best location found so far, exploiting their own personal experience. The third part is known as the "social" component and pulls the particles toward the best location found so far by any other particle, exploiting group experience (Ratnaweera et al., 2004). These components together enable particles to utilize the local and global information on each iteration. The evolution process usually begins with an initial random distribution and evolves as defined by Eqs. (1) and (2).

In the procedures above, the parameter V_{max} determines the resolution in which regions between the present location and the target location are searched. If V_{max} is too high, particles may fly over the good solutions. If V_{max} is too small, particles may not sufficiently explore beyond local solutions.

The constants c_1 and c_2 represent the weighting of the stochastic acceleration terms that pull each particle toward *pbest* and *gbest* locations. Low values allow particles to wander far from target regions before being pulled. On the other side, high values result in abrupt movements toward the target regions or pass the target regions. Hence, the acceleration constants c_1 and c_2 are often set to be 2.0.

3.2. Discrete particle swarm algorithm

The original PSO algorithm can only optimize problems in which the elements of the solution are continuous real numbers. Discrete particle swarm optimization can be obtained by replacing Eqs. (2) and (3):

$$x_{id}^{k+1} = \begin{cases} 1 & \text{for } rand(\cdot) < S(v_{id}^{k+1}) \\ 0 & \text{for } rand(\cdot) \ge S(v_{id}^{k+1}) \end{cases} \quad (3)$$

where $S(v)$ is a sigmoid limiting transformation function, $S(v) = 1/(1+e^{v})$, and $rand(\cdot)$ is a random number selected from a uniform distribution in [0.0, 1.0].

V_{max} in the discrete particle swarm is employed to limit further exploration after convergence of the swarm. In fact, it acts exactly opposite the mutation rate. Although higher V_{max} in the continuous-valued PSO raises the mutation rate and the exploration range of a particle, the opposite happens in a discrete PSO algorithm. Smaller V_{max} causes more exploration (Kennedy and Eberhart, 1997).

3.3. Asynchronous particle swarm optimization

The main difference between synchronous and asynchronous PSO concepts is the method used to update particle locations and velocities. Synchronous PSO updates particle locations and velocities at the end of each iteration. Therefore, after calculating the fitness values of all particles, *gbest* is updated and new location and velocity of particles are generated. Thus, synchronous PSO entails a synchronization point before advancing the next iteration. By virtue of synchronization point, some blocks stand idly by in hardware implementations. In contrast, asynchronous PSO updates particle locations and velocities ceaselessly with respect to the most recently available information. So, the processing of particles can progress continuously in a pipelined manner.

3.4. Parallel particle swarm optimization

PSO like the other population-based algorithms is parallelizable. For each iteration, the process of each particle is independent of other particles, causing particles to be effortlessly analyzed in parallel (Schutte et al., 2004). On the other hand, the PSO algorithms suffer from high computational cost and high elapsed time. Since complex engineering optimization problems demand high computational cost, the development of parallel optimization algorithms has been motivated. Parallel PSO algorithm is also valuable due to its global search capabilities.

Synchronous parallel implementations of PSO do not make efficient use of computational power of hardware resources since they require a synchronization point. In software-based designs, parallel synchronous implementation is ideal when processors take a constant amount of time to evaluate any set of design variables throughout the optimization (Koh et al., 2006). However, in many optimization problems such as those of scientific and industrial problems, the time required for fitness evaluation varies for different particles. In such cases, an asynchronous implementation is a better choice to reduce execution times and maintain high parallel efficiency. The convergence rate of asynchronous parallel PSO is comparable to that of the synchronous one (Koh et al., 2005).

4. Realization of parallel PSO framework

4.1. Hardware implementation

A parallel hardware design requires a partitioning of the algorithm into independent blocks in order to be utilized efficiently. Parallel synchronous implementation of PSO makes blocks partly idle and requires a huge amount of local storage for temporary data. On the other hand, parallel asynchronous method is well suited when the computational powers of diverse blocks are different. Since hardware implementation of asynchronous PSO eliminates the demands of synchronization and multiple port memories, the algorithm utilizes hardware logics remarkably and also reduces area cost. In this case, the parallel nature of the algorithm is appropriately utilized. This method provides an extremely parallel architecture to gain better speed-up.

Hardware-based parallelism has been exploited at two levels: at the fitness evaluation level and at the particle evolution level. At the fitness evaluation level, an on-chip multiprocessing architecture has been used to compute the fitness values in parallel. Since fitness function evaluation is the most time consuming part of the algorithm, a number of embedded processors work simultaneously to prepare fitness values. Moreover, since fitness evaluation is completely problem-dependent, it is implemented in software through programming the memories of the embedded processors to readily adapt the system to different kinds of problems without modifying the hardware.

The master–slave paradigm (Cantú-Paz, 1998) is a well-known model for many parallel evolutionary algorithms. Our proposed architecture follows an improved master–slave model as shown in Fig. 1. Our model is able to reach the most parallel efficacy by virtue of asynchronously updating particles and employing two temporary data storages. In this model, there are several fitness evaluator units (Nios II Processor) acting as the slave units. Each unit independently evaluates the fitness values of particles using software codes that are loaded into the program memory of the processor. After completion of fitness computation of a particle, the processor is ready to receive and evaluate the fitness of another particle. Our framework is a composition of such programmable embedded processors which are not exclusively committed to a specific particle. The instruction and data memories of embedded processors are assumed to be on-chip for simplicity.

After fitness evaluation, they pass the corresponding fitness value to the *Fitness Value Memory* which is shared among the *pipelined particle swarm optimization* (*PPSO*) *Core* (described below) and the embedded processors. In this model, the *PPSO Core* acts as the master unit. The *Fitness Value Memory* has a read port and a write port. Fitness values are read by the *PPSO Core* and are written by an *Arbiter Block*. The *Arbiter Block* prevents Nios II processors from occurring write conflicts while accessing the *Fitness Value Memory*. The *Arbiter Block* is in charge of writing the recently computed fitness value of each particle, which is the output of a processor, to the *Fitness Value Memory*. To keep the processors more active, whenever the *Fitness Value Memory* is busy, the *Arbiter Block* stores the newly computed fitness values in a temporary fitness value storage. The *Fitness Value Status Register* is responsible for holding information about the fitness evaluation operations. This register is also tested to determine the status of the fitness value of each particle.

The master unit performs all the particle-related processes such as updating velocity/location. The *FIFO Block*, as another temporary storage, holds the queue of particles ready to be sent to slave processors. The *PPSO Core* writes an updated particle location into the end of the queue and the Multiprocessor Controller reads the particle at the head of the queue. Once a slave processor finishes its corresponding tasks, the head particle in the queue is sent to it by the Controller. The whole design uses a centralized controller to direct the flow of data communication.

To find a solution for an optimization problem using PSO, each particle consists of a number of variables that determine the location of the particle in the search space. One of the drawbacks of PSO in solving complex problems is the particle length. Complex problems usually need a huge number of variables (long bit strings as candidate solutions). Long particles of the complex problems increase the area cost and reduce the processing speed. To handle this issue, at another parallelism level introduced here, the particles are split into a small number of variables so-called *subparticle* as shown in Fig. 2. Each subparticle is processed in parallel regardless of other subparticles. In fact, the subparticle method brings a trade-off between speed and hardware area. Subparticle length defines the particle update time and hardware cost. Long subparticles require more area due to the higher level of parallelism, but reduce particle processing time. The user determines the subparticle length as a parameter at compile time with respect to the required speed and the available area. This method of parallelism reduces the hardware area cost and makes the architecture independent of particles length. Particles with different lengths can be processed without increasing the

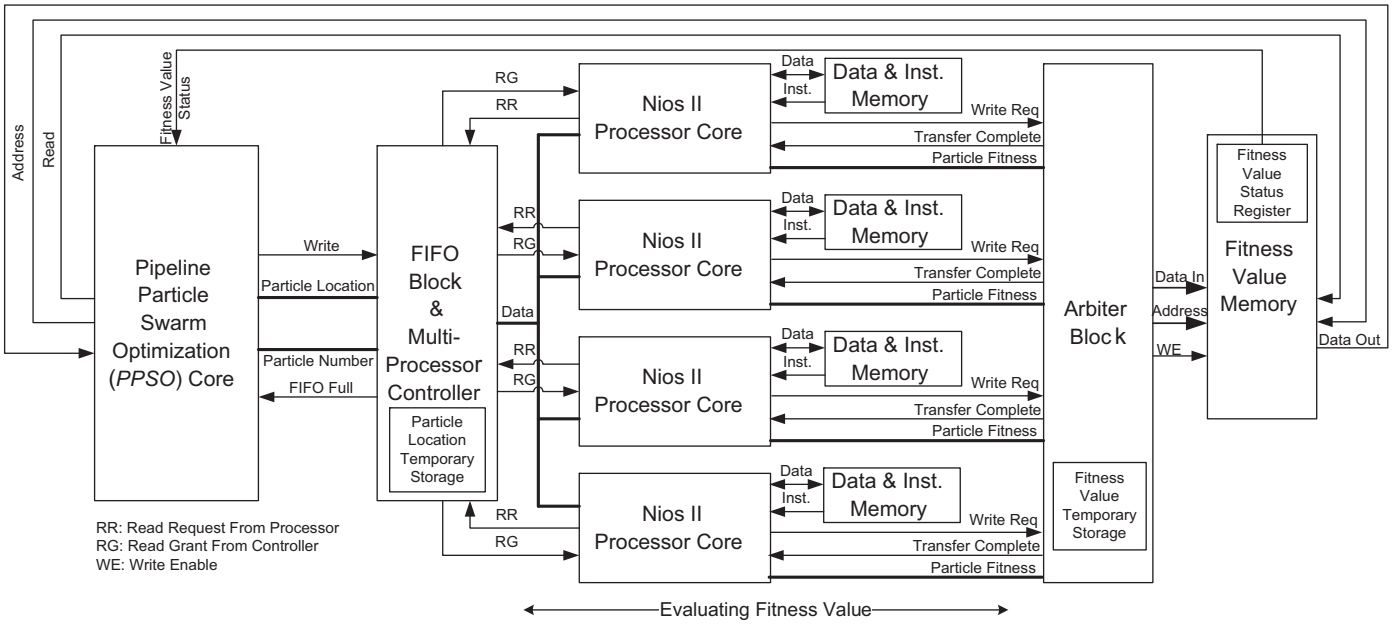

Fig. 1. Realization of the parallel PSO framework using four Nios II processors. The *PPSO Core* as the master unit updates particles, while Nios II processors as the slave units evaluate fitness values.

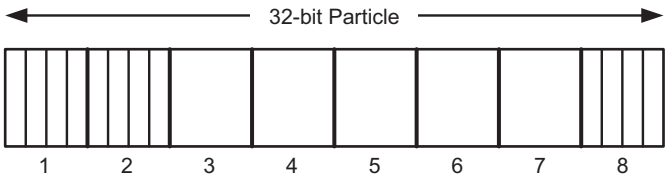

Fig. 2. A 32-bit particle is divided into eight 4-bit subparticles. The updating operation of each subparticle is performed in parallel, i.e., four bits are updated concurrently.

hardware cost. Long particles take more cycles, however, this method does not lead to modification in the hardware. Thus, the proposed architecture is flexible enough to compute any particle swarm optimization problem without requiring to redesign the hardware system.

A hardware unit called *PPSO Core* has been designed and implemented to treat the subparticles. This core accomplishes all the particle processes. The *PPSO Core* architecture is designed to attain the throughput of one subparticle per clock. To this end, it has a 5-stage pipelined structure. It consists of different memories, pipelined computational stages, and other diverse blocks. The internal organization of the proposed core and interconnections of different components are depicted in Fig. 3. Below is a description of the components of the *PPSO Core*.

Random Number Generation (RNG) Block: This component is very important in PSO hardware implementation. For example, for updating a 32-bit particle in 1000 iterations, 96,000 random numbers are needed (excluding random numbers required for fitness evaluation). To generate pseudorandom numbers, generally, linear feedback shift registers (LFSR) or cellular automata have been employed (Martin, 2002). The random numbers produced by the LFSRs supply enough performance in order to be used in evolutionary computations (Martin, 2002). Since an LFSR output histogram is similar to that of a uniform random variable (Reynolds et al., 2005), these units have been used to generate random numbers.

Memory Blocks: Three separate memories are used in the *PPSO Core* to keep the necessary vectors and values. These are accessible only by means of the *PPSO Core*. The *Particle Location Memory* (PLM) keeps the location vectors (x) for the particles in a swarm. Continually, location vectors of particles are updated and stored in this memory. Since we are aiming to implement the discrete model of PSO, each dimension is represented by one bit. The *Particle Velocity Vector Memory* (PVM) holds velocities of the particles. The dimensional velocity vectors of a particle are represented with 9 bits (4 bits for integer part, and 5 bits for fraction part), providing enough accuracy for our applications. Since the *PPSO Core* is fully parameterized, more accurate velocity vectors can also be used for the particles. The *Pbest Memory* keeps the best visited location of each particle (pbest vector) and its corresponding fitness value. Each fitness value is 32 bits. Therefore, for a swarm of eight 32-bit particles, *PLM*, *PVM*, and *Pbest Memory* hold 256, 2.3 K, and 512 bits, respectively.

Gbest Block: This block retains the best solution yet achieved by all particles (gbest vector) and corresponding fitness value.

Fetch Stage: This is the first stage of the pipeline. In this stage, the required values and vectors are fetched from the memories.

Update Gbest and Pbest Stage: This stage compares the calculated fitness of current location with the stored old ones to update *pbest* and *gbest* in each calculation step.

Update Particle Velocity Vector (1) and (2) Stages: These stages implement Eq. (1) in hardware. Updating velocity vector is broken into two stages to gain higher frequency. Random numbers generation and multiplication take considerable clock cycles in every processor. In fact, the run-time of PSO in a workstation depends dramatically on the power of a processor in performing these calculations.

Update Particle Location: This stage calculates a new particle location according to Eq. (3). The new particle location is indicated by a comparison between a randomly generated number and the sigmoid value of the velocity vector.

Sigmoid Function Approximation Block: This block implements an efficient approximation of the sigmoid function. This block is purely combinational and does not claim memory and multiplier, however, it yields the required accuracy (Tommiska, 2003). The input and output bit-widths of the sigmoid block are selected based on the required precision of the output.

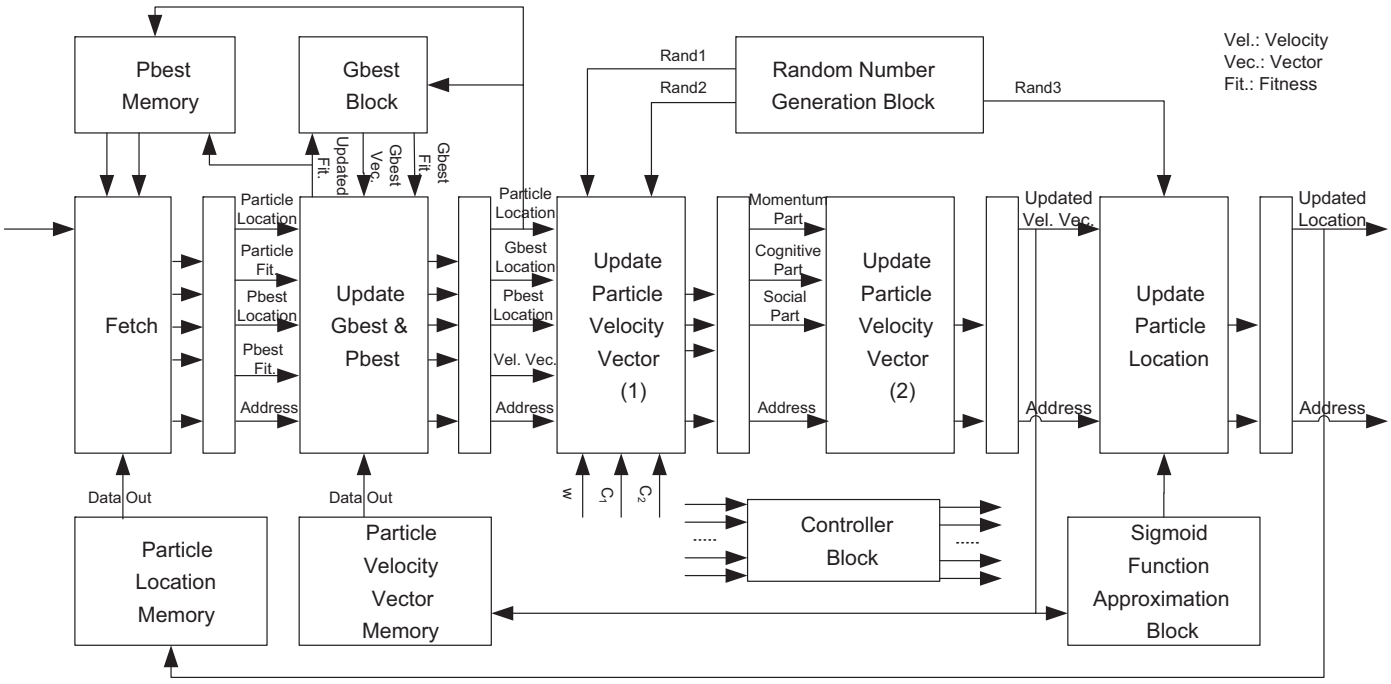

Fig. 3. The 5-stage *pipelined particle swarm optimization* (*PPSO*) *Core*.

Controller Block: This block is composed of state machines to track the operations of the algorithm. In the initialization step, this block writes a group of random values into the *PLM* and *PVM* by means of the *RNG Block*. It also initializes the finite state machines. This block generates the required control signals for all other blocks. To make Fig. 3 simpler, the connection signals of the *Controller Block* to other blocks have been omitted from the figure.

4.2. Processing time

The proposed architecture is intended to realize different particle swarm applications. Some of the parameters of the system such as subparticle length must be determined statically. Our object in this subsection is to aid designers to make engineering decisions about the structure of the required parallel PSO system. The following formulated guideline describes the relationship between the computation speed and different parameters of the problem and hardware architecture. Then, to represent the relationship, an equation is drawn that properly approximates the required time to compute one iteration. This representation helps us with more convenient studying about the effects of different parameters involved in determining processing time. Consider the following definitions:

p_{no}	number of particles in the swarm
p_{len}	particle length, i.e., number of bits that specifies the location of a particle in the search space
sp_{len}	subparticle length
μP_{no}	number of employed embedded processors for calculating fitness values
h_{time}	time that a handshaking operation takes between a processor and other blocks to transfer 32-bit data
h_{no}	number of handshaking operations for each particle between a processor and the *FIFO Block*
fit_{time}	required time for evaluating fitness of one particle (fitness computation time)
$pUpdate_{time}$	required time to update one particle using the *PPSO Core*
$busActivity_{time}$	time that the common data bus between the processors and the *FIFO Block* is occupied (communication time between the processors and the *FIFO Block*)
T	processing time for one iteration

Since most of the processors have 32-bit I/O ports, in each data transferring step, a maximum of 32 bits can be sent into or taken from a processor. The length of the fitness value is set to 32 bits. Thus, transferring the fitness value for each particle always needs one communication operation and is performed through the *Arbiter Block*. Regarding the optimization problem, the length of a particle may be more than 32 bits. In this case, $\lceil p_{len}/32 \rceil$ defines the necessary number of handshaking operations which is required to transfer a particle location to a processor, where $\lceil \cdot \rceil$ denotes the ceiling operation. The following equation presents the number of handshaking operations required for transferring a particle location between a processor and the *FIFO Block*. This operation is performed through a common data bus:

$$h_{no} = \lceil p_{len}/32 \rceil \tag{4}$$

Also, note that these operations must be replicated for each particle in the swarm. Therefore, the number of handshaking operations for a swarm in one iteration is $p_{no}.h_{no}$.

The communication time between a processor and the *FIFO Block* is given as $h_{no}.h_{time}$. Communications between the *FIFO Block* and processors rely on a common data bus. As the number of processors increases, more data are communicated between the processors and the *FIFO Block* and, as a result, bus transactions grow. Thus, the required time for the communications which are performed through the bus can be stated by the following equation:

$$busActivity_{time} = h_{no}.h_{time}.\mu P_{no} \tag{5}$$

Increasing the number of processors increases the ratio of $busActivity_{time}$ to fit_{time}. Nevertheless, $h_{no}.h_{time}$ is much smaller than fit_{time} for different benchmarks, and hence, a large number of

processors can be employed without any risk of bus congestion. In the worst case of the benchmarks, the ratio of $h_{no}.h_{time}$ to fit_{time} is 0.087. So, 11 processors can be efficiently used in parallel without degrading the performance of each single processor. Essentially, real-world applications require longer time to evaluate fitness as compared to the benchmarks and communication time can be considered negligible.

The proposed *PPSO Core* has a 5-stage pipelined architecture with throughput of one subparticle per clock cycle. The required time for updating one particle is a direct function of the particle and subparticle lengths:

$$pUpdate_{time} = \lceil p_{len}/sp_{len} \rceil \tag{6}$$

fit_{time} is directly determined according to the complexity of the problem and capability of the employed processors. All of the processors perform fitness evaluation in parallel. Therefore, swarm size is divided into $p_{no}/\mu P_{no}$ subswarms. To reach the maximum parallel efficacy, the following condition must be satisfied (the required time for transferring a fitness value is assumed the same as h_{time}) :

$$\frac{fit_{time}+(\lceil p_{len}/32 \rceil+1)h_{time}}{\mu P_{no}} = pUpdate_{time} \tag{7}$$

Eq. (8) shows the required time for computing one iteration:

$$T = \max((p_{no}.pUpdate_{time}), ((p_{no}/\mu P_{no}).((((\lceil p_{len}/32 \rceil+1)h_{time})+fit_{time}))) \tag{8}$$

In Eq. (8), the processor initialization time is assumed to be negligible. Since fitness evaluation is performed using software programmed codes, left expression of Eq. (7) takes more time than the right expression. Thus, fit_{time} is the dominant part and Eq. (8) can be simplified to become irrespective of $pUpdate_{time}$. Investigating the fitness computation time for different benchmarks confirm this simplification. Eq. (8) is not restricted to the SOPC-based architecture and is also applicable to the pure-hardware implementation in which the processors are replaced with special purpose hardware blocks.

Increasing the complexity of fitness evaluation intensifies the importance of the number of processors (μP_{no}) in performance. Hence, employing proper number of processors can effectively decrease the total processing time. Also, larger particles increase h_{no} and $pUpdate_{time}$ which are not influenced by μP_{no}.

5. Experimental results

To measure the SOPC-based system performance, the system has been simulated and implemented on an Altera Stratix 1S10ES Development Kit (chip EP1S10F780C6ES) (Altera Corp, 2006a). All hardware blocks are developed using synthesizable Verilog hardware description language. The area usage of the *PPSO Core* depends on the subparticle length. For instance, for 2-bit subparticles, it uses less than 1000 logic elements. The required amount of memory for saving PSO related vectors is less than 3400 memory bits for a swarm of eight 32-bit particles.

For fitness evaluation, Nios II/s general purpose RISC processor (Altera Corp, 2006b) is employed with hardware multiplication and divide units. Altera has introduced this soft-core embedded processor for its SOPC systems. The Nios II instruction set is designed to support programs compiled using C and C++. The Nios II/s processors are employed to perform fitness evaluation in the proposed system. Each Nios II/s processor requires almost 1500 logic elements and the required memory bits for each data and instruction memory depend on the programmed fitness function.

The effectiveness of our SOPC implementation of PSO has been demonstrated on five test functions. The utilized test functions are the MaxOne problem and four arithmetic functions of the very standard DeJong Test Suite (De Jong, 1975) with binary encoding. In the MaxOne problem, the sum of all the bits of a particle must be maximized. In the other functions, the fitness value must be minimized. The algebraic forms of these functions are given in Table 1. First and second columns represent the function name and equation. The third column shows the value range of particles for each function. The fourth column determines the number of bits for each particle (function dimension), and the last column represents the maximum number of iterations required for each function which in fact indicates the termination condition. Although the length of particles for each function is different, the functionality of the proposed system is independent of the particle length and the system is capable of handling particles with various bit-string lengths.

It is worth noting that the purpose of the multiprocessing framework is performing fitness evaluation in parallel. Consequently, multiprocessing reduces the computation time of the algorithm. However, as the number of processors on a chip increases, the maximum acquired frequency decreases due to higher congestion and longest path delays on an FPGA. In this design, the maximum operating frequency for one, two, three and four processors on a single chip is 61.6, 59.4, 58.9, and 58.6 MHz, respectively. This degradation of maximum frequency is tolerable and does not conceal the effectiveness of the parallel design. Using a more powerful FPGA is a way to achieve higher frequency and better performance. The hardware cost of the proposed SOPC-based PSO system using four Nios II/s processors as the fitness evaluators is given in Table 2. The maximum operating frequency of each circuit is also shown in Table 2. The last column of the table reports the number of used DSP blocks. The Altera Stratix FPGAs contain a flexible embedded block, called a DSP block. These blocks can perform accumulate functions as well as multiply operations. In this table, 2-bit subparticles are employed in the *PPSO Core* and the swarm size is eight.

The convergence rate of the software implemented asynchronous discrete PSO has been compared with the SOPC-based implemented asynchronous PSO to make an investigation of the performance of our SOPC-based implementation. In all the experiments, the PSO algorithms use parameter values $w=1$, $c_1=c_2=2$, and the swarm size is eight. During the run of the algorithm, a maximum velocity $V_{max}=8$ was applied for every component of the velocity vector. The results are depicted in

Table 1
Test suites.

Function	Equation	Search range	Dimension	Max iteration
MaxOne	$f_0(\overline{x})=\sum_{i=1}^{n} x_i,\ n=32$	{0, 1}	32	50
Sphere	$f_1(\overline{x})=\sum_{i=1}^{n} x_i^2,\ n=3$	[−5.12, 5.12)	30	100
Rosenbrock	$f_2(\overline{x})=\sum_{i=1}^{n-1}(100(x_i^2-x_{i+1})^2+(1-x_i)^2),\ n=2$	[−2.048, 2.048)	24	100
Step	$f_3(\overline{x})=6n+\sum_{i=1}^{n}\lfloor x_i \rfloor,\ n=5$	[−5.12, 5.12)	50	100
Rastrigin	$f_4(\overline{x})=\sum_{i=1}^{n}(10+x_i^2-10\cos(2\pi x_i)),\ n=3$	[−5.12, 5.12)	30	100

Fig. 4. In this figure, graphs represent the average of best evaluations in 50 repeated runs over the number of iterations. Fig. 4 shows that the performance of the SOPC-based implementation is comparable to the software one.

The elapsed time of our SOPC-based pipelined PSO was benchmarked against a pure-software-based solution of PSO written in MATLAB (MathWorks Inc, 2007) and implemented on a single 3-GHz Pentium IV processor, 1-GB RAM, and Windows XP Pro OS. Table 3 presents a comparison in speed between software and SOPC-based versions for different numbers of processors. The results are reported in terms of 1000 clock cycles and milliseconds considering a period of 20 ns (50 MHz) as well as regarding 2-bit subparticles. Swarm size (number of particles) for all the problems is assumed to be eight. Termination condition is the maximum number of iterations as described in Table 1.

The major bottleneck of the system is the fitness evaluation with the Nios II processors. It is possible to design specific hardware for fitness evaluation (instead of software implementation) in case of simple problems to speed up the system. Table 4 shows the results of the pure-hardware implementation using one fitness evaluator block. PSO Parameters are the same as the previous experiment. In this case, in addition to updating particles, a custom fitness evaluator block is implemented in hardware. As expected, the elapsed time is considerably less than SOPC-based architecture. Nevertheless, an expert hardware engineer must design a specific fitness evaluator block for every different fitness function. This is a time consuming and error prone task that sometimes is practically impossible for complex functions. Thus, in this way, modularity and flexibility are lost and the design becomes more cumbersome.

Table 2
Hardware usage of the SOPC-based asynchronous discrete PSO system using Altera Stratix Kit.

Block	Required memory bits					LEs[a]	Max freq. (MHz)	No. of DSPs
	MaxOne	Sphere	Rosenbrock	Step	Rastrigin			
Pipelined PSO Core	3.3k	3.2k	2.6k	4.9k	3.2k	805	76.3	4
Four Nios II/s Cores & Memories[b]	242k	402k	400k	330k	654k	6894	61.7	32
Arbiter and FIFO	1k	1k	1k	1k	1k	255	131.3	–
Other	–	–	–	–	–	298	–	–

[a] Logic elements.
[b] Nios II processor internal memories in addition to on-chip data and instruction memories which hold the software codes of fitness function.

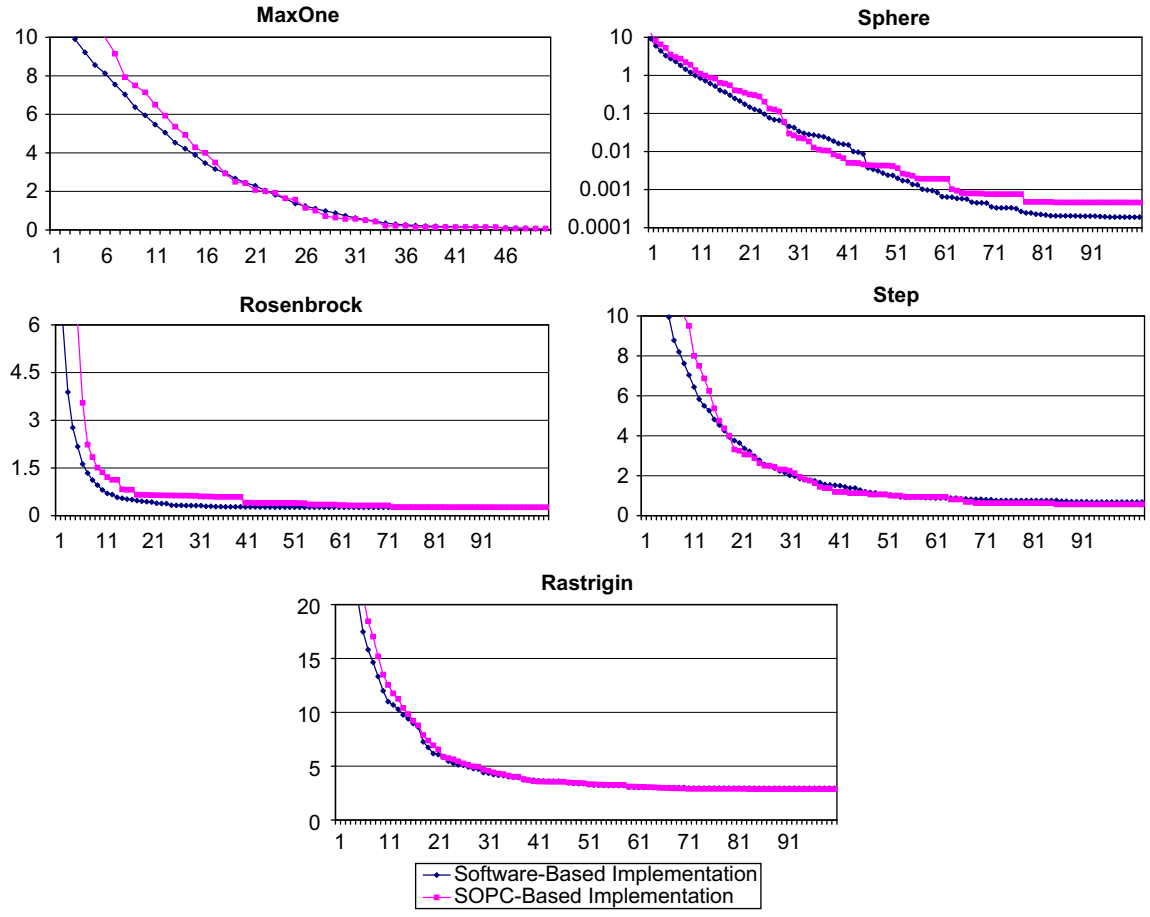

Fig. 4. Average performance of the software-based floating point implementation of asynchronous discrete PSO and the SOPC-based implementation of it for different test functions. For each graph, the *x*-axis shows the iteration number and the *y*-axis represents the average fitness values computed over 50 trials.

Table 3
Elapsed time of the SOPC-based and software implementations of asynchronous discrete PSO.

Function	SOPC-based implementation elapsed time[a]				Pure-software impl.
	One processor	Two processors	Three processors	Four processors	Elapsed time (ms)
MaxOne	124 (2.5 ms)	62 (1.2 ms)	42 (0.8 ms)	32 (0.6 ms)	28
Sphere	8881 (177.6 ms)	4595 (91.9 ms)	2983 (59.7 ms)	2229 (44.6 ms)	1868
Rosenbrock	7617 (152.3 ms)	3768 (75.4 ms)	2512 (50.2 ms)	1903 (38 ms)	1372
Step	6206 (124.1 ms)	3101 (62 ms)	2032 (40.6 ms)	1544 (30.9 ms)	3059
Rastrigin	40,774 (815.5 ms)	21,042 (420 ms)	14,820 (296 ms)	11,108 (222 ms)	1875

[a] The elapsed time is reported in 1000 clock cycles. Termination condition for each function is the maximum iteration number reported in Table 1.

Table 4
Pure-hardware implementation using one fitness evaluator block.

Function	Using DSP blocks			Without using DSP blocks		Fitness time[a]	Elapsed time[b]
	No. of DSPs	LEs	Frequency (MHz)	LEs	Frequency (MHz)		
MaxOne	4	993	72.0	1215	71.2	5	6573
Sphere	10	1063	71.0	1821	48.5	4	12,165
Rosenbrock	24	1040	53.6	2380	37.2	5	9741
Step	24	1021	70.1	1730	47.5	5	20,245
Rastrigin	18	1374	60.3	2909	47.7	22	21,587

[a] Required clock cycles for calculating fitness value of a particle using the hardware fitness evaluator block.
[b] The elapsed time is reported in terms of clock cycles. Termination condition for each function is the maximum iteration number reported in Table 1.

By implementing fitness functions in software using embedded processors, the complexity of the fitness evaluation does not alter the structure of the architecture. Regarding Table 3, a roughly leaner speed-up has been measured with increasing the number of processors. Also, a speed-up of up to 98 times has been obtained in moving to SOPC-based architecture, while the SOPC-based clock speed is almost 60 times slower than the Pentium processor used here for the pure-software implementation. This issue shows the capability of the system in embedded environments as a real-time optimizer.

6. A real-world application as a case study: neural network training

This paper proposes a framework for hardware implementation of PSO algorithm bringing flexibility and speed. This work is intended for real-world applications. In this section, a practical application of the proposed framework has been introduced. The implementation results of the application validate the aforementioned method.

The proposed architecture for PSO algorithm can be exploited to train multi-layer perceptron (MLP) neural networks. Artificial neural networks (ANNs) have been successfully used in a wide range of scientific and engineering applications (Haykin, 1999; Widrow et al., 1994; Fiesler and Beale, 1997). By learning or training from examples, a neural network (NN) is capable of exhibiting intelligent behavior and modeling complex non-linear functions which makes it proper to variable conditions. Training is the process of gradually adjusting the weights of connections. In this paper, the PSO algorithm has been employed for training feedforward neural networks. In Zhao et al. (2005), Liu et al. (2004), Carvalho and Ludermir (2006), Gudise and Venayagamoorthy (2003), and Mendes et al. (2002), the authors have proven the speed and accuracy of using PSO to determine the neural network weights.

In NN training, the main goal is to obtain a set of weights that minimizes mean squared error. In order to address the problem of neural network training to PSO, we represent each set of weights and biases of a network by a single particle. Thus, each particle is a string that encodes a candidate solution for the weights and biases of all neurons in the network. The length of a particle depends on the network intended to train. A pool of particles is considered as a swarm for PSO. By repetitively updating particles of the swarm, the most suited network weights are gradually determined.

In order to train a neural network using the proposed framework, the software codes that are used to train the network are loaded into the program memory of the Nios processors (Fig. 1). Weights and biases are evolved by the *PPSO Core*. In addition, a set of training inputs are applied to the network by the processors. Each processor first computes output value(s) of the network using weight, bias, and input values and then computes a fitness value. Fitness value is considered as the cumulative error of network output(s) for the entire training set. In other words, differences between the target (desired) outputs and actual outputs of the neural network are calculated. The differences for the entire training set are summed up to achieve a fitness value. In this manner, particles are updated by the *PPSO Core* and fitness values are computed by the processors. Consequently, different NNs can be automatically trained by using the proposed framework. A designer should only write the software codes for processors without the need to interact with hardware which reduces design time and enhances flexibility.

A pure-hardware version of this system has been also presented by the authors in Farmahini-Farahani et al. (2008). In the pure-hardware architecture, the embedded processors are replaced with four specific neural network processing elements to reduce computation time, while flexibility is removed and design time increases dramatically. Table 5 presents speed comparison between the SOPC-based implementation and the pure-hardware implementation for three different networks. The results are reported in terms of seconds considering a period of 20 ns (50 MHz) and shows the processing time for one iteration. Swarm size for all the networks is assumed to be eight. As expected, the pure-hardware implementation has far less

Table 5
Elapsed time of the SOPC-based and pure-hardware implementations of neural network training using the proposed framework.

Network Structure	SOPC-based implementation elapsed time				Pure-hardware implementation (μs)	No. of training inputs
	One processor (ms)	Two processors (ms)	Three processors (ms)	Four processors (ms)		
2-2-1	53.7	28.9	19.3	14.1	14.08	4
3-6-2	325.8	162.7	110.1	83.6	46.88	8
4-5-5-3	1153.1	567.2	390.1	288.5	152	16

execution time than the SOPC-based implementation. However, design and verification time is always the major problem of a pure-hardware design. Ultimately, The flow presented here explains a real-world application of the proposed framework as a parallel architecture for implementing PSO algorithm using embedded processors.

7. Conclusions and future work

In this paper, a scalable embedded implementation has been proposed for an asynchronous discrete PSO realization using the system-on-a-programmable-chip (SOPC) approach. In doing so, on-chip parallelism and also a pipelined PSO Core have been employed as a real-time function optimizer. The proposed framework contains a set of parallel embedded processors that are connected by an on-chip bus architecture. Hence, increasing the number of processors is feasible to bear computationally intensive fitness functions. Also, different discrete problems with different particle lengths can be applied to the framework. Therefore, it is scalable in terms of both the number of integrated processors and particle length. The architecture targets an assortment of pipelined and parallel structures that enables it to realize different kinds of particle swarm optimization problems in a single framework.

All of the structural parameters formulated here aid the designer in choosing the proper number of processors and best architecture for the PSO Core. The reusability of the core was strengthened with supporting various discrete particles and the formulated design guidelines. One scheme of realizing the parallel PSO framework was presented along with a block diagram of the complete pipeline. The proposed implementation has shown a considerable speed-up over a software-based implementation, especially for long particles. The experimental results prove the benefits of our mixed hardware–software solution in comparison to having an exclusively software or hardware implemented system.

As a result of hardware–software co-implementation, speed improvement is the advantage of this system in comparison to a pure-software implementation. The benefits of this system as compared to a pure-hardware implementation are remarkable. Various applications with different fitness functions are easily managed by programming the soft fitness evaluation core (modularity), and, furthermore, it is easily reconfigurable in comparison to an ASIC design. Utilizing the soft cores for fitness evaluation along with scalability of the system makes it appropriate for various applications, and, accordingly, more convenient in comparison with the pure-hardware implementation.

Our solution provides a trade-off between the speed of a pure-hardware solution and the flexibility of a pure-software implementation. Therefore, it has balanced speed against the ease of programmability. Also, the proposed architecture makes the design independent of particle length, allowing solving diverse PSO problems readily without altering the design.

Finally, the design flow of utilizing the *PPSO Core* was presented in the form of embedded implementation of a PSO-based neural network training algorithm. The whole design is realized on an SOPC bed in favor of Nios II microprocessors. The architecture of the design is suited for real-world applications which delivers fast, flexible and low cost development of products.

Our future work is expanding this architecture to cover continuous-valued PSO, in which the system is more dependent on arithmetic functions and less on RNG units. Also, adding some other modules such as inertia weight and implementation of other topologies of PSO are under consideration. Another interesting work is devising an approach to overcome the drawbacks to common data bus regarding the features of the PSO algorithm.

References

Altera Corp., 2006a. Nios II Development Kit, Stratix Edition. 〈http://www.altera.com/products/devkits/altera/kit-nios_1S10.html〉.

Altera Corp., 2006b. Nios II Processor Reference Handbook.

Bäck, T., 1996. Evolutionary Algorithms in Theory and Practice. Oxford University Press, New York, NY.

Bäck, T., Harnmel, U., Schwefel, H.P., 1997. Evolutionary computation: comments on the history and current state. IEEE Transactions on Evolutionary Computation 1 (1), 3–17.

Calaor, A.E., Hermosilla, A.Y., Corpus Jr., B.O., 2002. Parallel hybrid adventures with simulated annealing and genetic algorithms. In: Proceedings of the International Symposium on Parallel Architectures Algorithms and Networks, Makati City, Metro Manila, Philippines, May, pp. 33–38.

Cantú-Paz, E., 1998. A survey of parallel genetic algorithms. Calculateurs Paralléles. Réseaux et Systémes Répartis 10 (2), 141–171.

Carlisle, A., Dozier, G., 2001. An off-the-shelf pso. In: Proceedings of the Workshop Particle Swarm Optimization, Indianapolis, IN, April, pp. 1–6.

Carvalho, M., Ludermir, T.B., 2006. An analysis of PSO hybrid algorithms for feed-forward neural networks training. In: Proceedings of the Brazilian Symposium on Neural Networks, Ribeirao Preto, Brazil, October, pp. 2–7.

Chang, J.-F., Chu, S.-C., Roddick, J.F., Pan, J.-S., 2005. A parallel particle swarm optimization algorithm with communication strategies. Journal of Information Science and Engineering 21 (4), 809–818.

Chen, D., Lee, C.-Y., Park, C.H., 2005. Hybrid genetic algorithm and simulated annealing (HGASA) in global function optimization. In: Proceedings of the IEEE International Conference on Tools with Artificial Intelligence, Hong Kong, China, November, pp. 126–133.

Cui, S., Weile, D.S., 2005. Application of a parallel particle swarm optimization scheme to the design of electromagnetic absorbers. IEEE Transactions on Antennas and Propagation 53 (11), 3616–3624.

Dasgupta, D., Michalewicz, Z. (Eds.), 1997. Evolutionary Algorithms in Engineering Applications. Springer, Berlin.

De Jong, K.A., 1975. An analysis of the behavior of a class of genetic adaptive systems. Ph.D. Thesis, University of Michigan, Ann Arbor.

Deb, K., 2001. Multi-objective Optimization using Evolutionary Algorithms. Wiley, New York, NY.

Duren, R.W., Marks, R.J., Reynolds, P.D., Trumbo, M.L., 2007. Real-time neural network inversion on the SRC-6e reconfigurable computer. IEEE Transactions on Neural Networks 18 (3), 889–901.

Farmahini-Farahani, A., Fakhraie, S.M., Safari, S., 2008. Scalable architecture for on-chip neural network training using swarm intelligence. In: Proceedings of the Design Automation and Test in Europe Conference, Munich, Germany, March, pp. 1340–1345.

Fiesler, E., Beale, R., 1997. Handbook of Neural Computation. IOP Publishing Ltd, Oxford University Press.

Fogel, D.B., 1991. System Identification Through Simulated Evolution: A Machine Learning Approach to Modeling. Ginn Press, Needham Heights, MA.

Fogel, D.B., 1994. An introduction to simulated evolutionary optimization. Neural Networks 5, 3–14.

Freitas, A.A., 2002. Data Mining and Knowledge Discovery with Evolutionary Algorithms. Springer, Berlin.
Gies, D., Rahmat-Samii, Y., 2003. Particle swarm optimization for reconfigurable phase-differentiated array design. Microwave and Optical Technology Letters 38 (3), 168–175.
Gudise, V.G., Venayagamoorthy, G.K., 2003. Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks. In: Proceedings of the IEEE Swarm Intelligence Symposium, Indianapolis, IN, USA, April, pp. 110–117.
Haykin, S., 1999. Neural Networks: A Comprehensive Foundation, second ed. Prentice-Hall, NJ, USA.
Hidalgo, J.I., Prieto, M., Lanchares, J., Baraglia, R., Tirado, F., Garnica, O., 2003. Hybrid parallelization of a compact genetic algorithm. In: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing, Genova, Italy, February, pp. 449–455.
Holland, J.M., 1975. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI.
Hu, X., Shi, Y., Eberhart, R., 2004. Recent advances in particle swarm. In: Proceedings of the IEEE Congress on Evolutionary Computation, vol. 1, Portland, OR, USA, June 19–23, pp. 90–97.
Jin, N., Rahmat-Samii, Y., 2005. Parallel particle swarm optimization and finite-difference time-domain (PSO/FDTD) algorithm for multiband and wide-band patch antenna designs. IEEE Transactions on Antennas and Propagation 53 (11), 3459–3468.
Kennedy, J., Eberhart, R., 1995. Particle swarm optimization. In: Proceedings of the IEEE International Conference on Neural Networks, vol. 1, pp. 1942–1948.
Kennedy, J., Eberhart, R., 1997. A discrete binary version of the particle swarm algorithm. In: Proceedings of the IEEE International Conference on Systems, Cybernetics and Informatics, vol. 5, Orlando, FA, USA, October 12–15, pp. 4104–4108.
Koh, B.-I., Fregly, B.J., George, A.D., Haftka, R.T., 2005. Parallel asynchronous particle swarm for global biomechanical optimization. In: Proceedings of the International Symposium on Computer Simulation in Biomechanics, Cleveland, OH, USA, July 28–30, pp. 86–87.
Koh, B.I., George, A.D., Haftka, R.T., Fregly, B.J., 2006. Parallel asynchronous particle swarm optimization. International Journal for Numerical Methods in Engineering 67 (4), 578–595.
Kokai, G., Christ, T., Frhauf, H.H., 2006. Using hardware-based particle swarm method for dynamic optimization of adaptive array antennas. In: Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems, Istanbul, Turkey, June 15–18, pp. 51–58.
Konfrst, Z., 2004. Parallel genetic algorithms: advances, computing trends, applications and perspectives. In: Proceedings of the IEEE Parallel and Distributed Processing Symposium, Santa Fe, NM, USA, pp. 162–169.
Liu, H.-B., Tang, Y.-Y., Meng, J., Ji, Y., 2004. Neural networks learning using vbest model particle swarm optimisation. In: Proceedings of the IEEE International Conference on Machine Learning and Cybernetics, vol. 5, August, pp. 3157–3159.
Martin, P., 2002. An analysis of random number generators for a hardware implementation of genetic programming using FPGAs and Handel-C. In: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, 9–13 July, pp. 837–844.
Mathew, B., Davis, A., Parker, M., 2004. A low power architecture for embedded perception. In: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, Washington, DC, USA, pp. 46–56.
MathWorks Inc., 2007. Matlab. 〈http://www.mathworks.com/〉.
Mendes, R., Cortez, P., Rocha, M., Neves, J., 2002. Particle swarms for feedforward neural network training. In: Proceedings of the International Joint Conference on Neural Network, vol. 2, Honolulu, HI, USA, May, pp. 1895–1899.
Pena, J., Upegui, A., 2007. A population-oriented architecture for particle swarms. In: Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems, Edinburgh, UK, August, pp. 563–570.
Pena, J., Upegui, A., Sanchez, E., 2006. Particle swarm optimization with discrete recombination: an online optimizer for evolvable hardware. In: Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems, Istanbul, Turkey, June 15–18, pp. 163–170.
Ratnaweera, A., Halgamuge, S.K., Watson, H.C., 2004. Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients. IEEE Transactions on Evolutionary Computation 8 (3), 240–255.
Reynolds, P.D., Duren, R.W., Trumbo, M.L., Marks, R.J., 2005. FPGA implementation of particle swarm optimization for inversion of large neural networks. In: Proceedings of the IEEE Swarm Intelligence Symposium, Pasadena, TX, USA, June 8–10, pp. 389–392.
Schutte, J.F., Fregly, B.J., Haftka, R.T., George, A.D., 2003. A parallel particle swarm optimizer. In: Proceedings of the Congress of Structural and Multidisciplinary Optimization, Lido di Jesolo, Italy, May, pp. 19–23.
Schutte, J.F., Reinbolt, J.A., Fregly, B.J., Haftka, R.T., George, A.D., 2004. Parallel global optimization with particle swarm algorithm. International Journal for Numerical Methods in Engineering 61, 2296–2315.
Schwefel, H.P., 1981. Numerical Optimization of Computer Models. Wiley, New York.
Shi, Y., Eberhart, R., 1998. A modified particle swarm optimizer. In: Proceedings of the IEEE International Conference on Computational Intelligence, Piscataway, NJ, USA, pp. 69–73.
Song, M.P., Gu, G.C., 2004. Research on particle swarm optimization: a review. In: Proceedings of the International Conference on Machine Learning and Cybernetics, vol. 4, August, pp. 2236–2241.
Tommiska, M.T., 2003. Efficient digital implementation of the sigmoid function for reprogrammable logic. In: Proceedings of the IEE Computers and Digital Techniques, vol. 150 (6), November, pp. 403–411.
Venter, G., Sobieszczanski-Sobieski, J., 2005. A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. In: Proceedings of the Congress of Structural and Multidisciplinary Optimization, Brazil, May/June 31–03, pp. 3351–3360.
Widrow, B., Rumelhart, D.E., Lehr, M.A., 1994. Neural networks: applications in industry, business and science. Communications of the ACM 37 (3), 93–105.
Zhang, T., Benini, L., Micheli, G.D., 2001. Component selection and matching for IP-based design. In: Proceedings of the IEEE International Conference on Design, Automation and Test in Europe, Munich, Germany, March, pp. 40–46.
Zhao, F., Ren, Z., Yu, D., Yang, Y., 2005. Application of an improved particle swarm optimization algorithm for neural network training. In: Proceedings of the IEEE International Conference on Neural Networks and Brain, vol. 3, Beijing, China, October, pp. 1693–1698.
Zhu, Z., Mulvaney, D.J., Chouliaras, V., 2006. A novel genetic algorithm designed for hardware implementation. International Journal of Computational Intelligence 3 (4), 281–288.
Zitzler, E., Teich, J., Bhattacharyya, S.S., 2000. Evolutionary algorithms for the synthesis of embedded software. IEEE Transactions on Very Large Scale Integration Systems 8 (4), 452–455.