ICARIS 2005, Banff, Canada, August 2005.
LNCS 3627:43-55. Springer, 2005

A generic framework for population-based algorithms, implemented on multiple FPGAs

John Newborough and Susan Stepney

Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK

Abstract. Many bio-inspired algorithms (evolutionary algorithms, artificial immune systems, particle swarm optimisation, ant colony optimisation, …) are based on populations of agents. Stepney *et al* [2005] argue for the use of conceptual frameworks and meta-frameworks to capture the principles and commonalities underlying these, and other bio-inspired algorithms. Here we outline a generic framework that captures a collection of population-based algorithms, allowing commonalities to be factored out, and properties previously thought particular to one class of algorithms to be applied uniformly across all the algorithms. We then describe a prototype proof-of-concept implementation of this framework on a small grid of FPGA (field programmable gate array) chips, thus demonstrating a generic architecture for both parallelism (on a single chip) and distribution (across the grid of chips) of the algorithms.

1 Introduction

Many bio-inspired algorithms are based on populations of agents trained to solve some problem such as optimising functions or recognising categories. For example, Evolutionary Algorithms (EA) are based on analogy to populations of organisms mutating, breeding and selecting to become "fitter" [Mitchell 1996]. The negative and clonal selection algorithms of Artificial Immune Systems (AIS) use populations of agents trained to recognise certain aspects of interest (see de Castro & Timmis [2002] for an overview): negative selection involves essentially random generation of candidate recognisers, whilst clonal selection uses reinforcement based on selection and mutation of the best recognisers. Particle swarm optimisation (PSO) [Kennedy & Eberhart 2001] and social insect algorithms [Bonabeau 1999] use populations of agents whose co-operations (direct, or stigmergic) result in problem solving.

Stepney *et al* [2005] argue for the use of conceptual frameworks and meta-frameworks to capture the principles and commonalities underlying various bio-inspired algorithms. We take up this challenge, and, in section 2, outline a generic framework abstracted from the individual population-based models of the following classes: genetic algorithms (GA), AIS negative selection, AIS clonal selection, PSO, and ant colony optimisation (ACO). The framework provides a basis for factoring out the commonalities, and applying various properties uniformly across all the classes of algorithms, even where they were previously thought particular to one class (section 3).

In section 4 we describe our proof-of-concept prototype implementation of the generic framework on a platform of multiple field programmable gate array (FPGA) chips. Thus the generic architecture naturally permits both parallelism (multiple individuals executing on a single chip) and distribution (multiple individuals executing across the array of chips) of the algorithms. In section 5 we outline what needs to be done next to take these concepts into a fully rigorous framework architecture and implementation.

2 The generic framework for population algorithms

There are many specific algorithms and implementation variants of the different classes. To take one case, AIS clonal selection, see, for example [Cutello *et al* 2004] [Garrett 2004] [Kim & Bentley 2002]. It is not our intention to capture every detail of all the variants in the literature. Rather, we take a step back from the specifics, and abstract the *basic underlying concepts*, particularly the more bio-inspired ones, of each class of algorithm. So when we refer to "GA" or "AIS clonal selection", for example, we are not referring to any one specific algorithm or implementation, but rather of the general properties of this class. We unify the similarities between these basics in order to develop a generic framework. The intention is that such a framework provides a useful starting point for the subsequent development of more sophisticated variants of the algorithms.

Basic underlying concepts

The generic algorithm is concerned with a population of *individuals*, each of which captures a possible solution, or part of a solution. Each individual contains a set of *characteristics*, which represent the solution. The characteristics define the (phase or state) space that the population of individuals inhabit. The goal of the algorithm is to find "good" regions of this space, based on some *affinity* (a measure that relates position in the space to goodness of solution, so defining a *landscape*). The individuals and characteristics of the specific classes of algorithm are as follows:

GA : the individuals are *chromosomes*; each characteristic is a *gene*.

AIS negative selection : the individuals are *antibodies*; each characteristic is a *shape receptor*.

AIS clonal selection : there are two populations. In the *main population* the individuals are *antibodies*; each characteristic is a *shape receptor*. There is also a population of *memory cells* drawn from this main population.

Swarms : the individuals are *boids*; the characteristics are *position*, *velocity* and *neighbourhood group* (the other visible individuals).

Ants: the individuals are the *complete paths* (not the ants, which are merely mechanisms to construct the complete paths from *path steps*); the characteristics are the *sequence of path steps*, where each step has an associated characteristic of *length* and *pheromone level*.

Algorithm stages

The different specific algorithms each exhibit six clearly distinct stages, comprising a *generation*. These are generalised as:

1. **Create** : make novel members of the population
2. **Evaluate** : evaluate each individual for its affinity to the solution
3. **Test** : test if some termination condition has been met
4. **Select** : select certain individuals from the current generation, based on their affinity, to be used in the creation of the next generation
5. **Spawn** : create new individuals for the next generation
6. **Mutate** : change selected individuals

We describe each of these stages, covering the generic properties, and how they are instantiated for each specific class of algorithm. Using this framework results in descriptions that sometimes differ from, but are equivalent to, the traditional descriptions of the algorithms. For example, rather than saying that some individuals survive from generation to generation, for uniformity we consistently consider each generation to be a completely fresh set of individuals, with some possibly being *copies* of previous generation individuals. As another example, the pheromone changes in the Ant algorithm is mapped to the generic mutate step.

Create

Creation makes novel members of the populations. In the first generation, the whole population is set up, and the members have their characteristics initialised. On subsequent generations, creation "tops up" the population with fresh individuals, as necessary.

GA: an individual chromosome is created usually with *random* characteristics, giving a broad coverage of the search space

AIS negative selection : an individual antibody is created usually with *random* shape receptors

AIS clonal selection : an individual antibody in the main population is created usually with *random* shape receptors; memory cells are not created, rather they are *spawned* from the main population

Swarms : an individual boid is created usually with *random* position and velocity characteristics, giving a broad coverage of the search space; the neighbourhood characteristic is usually set to implement a ring, grid or star connection topology

Ants : each path step is initially set up usually with a *fixed* pheromone level, and with the relevant (fixed) path length; the population of paths is *created* by the ants from these steps each generation

Evaluate

The *affinity* measures how well each individual solves (part of) the problem. It is a user-defined function of (some of) an individual's characteristics. This function should ideally (but does not always) have the structure of a *metric* over the space defined by the characteristics.

GA : the affinity is the *fitness* function, a function of the values of the genes

AIS : the affinity is a measure of how closely the shape receptors *complement* the target of recognition, inspired by the "lock and key" metaphor

Swarms : the affinity, or *fitness* function, is a function of the current *position*

Ants : the affinity is the (inverse of the) *path length*

Test

The test for termination is either (a) a sufficiently good solution is found, or (b) enough generations have been run without finding a sufficiently good solution. On termination, the solution is:

GA, Swarms, Ants : the highest affinity (fittest) individual

AIS negative selection : the set of individuals with above-threshold affinities

AIS clonal selection : the population of memory cells

Select

High affinity individuals are *selected* to contribute somehow to the next generation's population. There are several selection algorithms commonly used. *n best* selects the *n* highest affinity individuals from the current population. *Threshold* selects all the individuals with an affinity greater than some given threshold value. *Roulette wheel* selection randomly chooses a given number of individuals, with probability of selection proportional to their affinity, or to their ranking. *Tournament* randomly selects teams of individuals, and then *selects* a subset of individuals from each team.

GA : different variants use any of the above methods of selection, to find the *parents* that will produce the next generation

AIS negative selection : *threshold* selection is used to find the next generation

AIS clonal selection : a combination of *n best* and *threshold* selection is used to find the next generation of the main population; *all* individuals of the memory cell population are selected to become the basis of its next generation

Swarms : *all* individuals are selected to become the basis of the next generation

Ants : *no* individuals are specifically selected to become the next generation: each generation is *created* afresh from the path steps (whose characteristics are changed by the *mutate* step)

Spawn

Production of new individuals for the next generation usually involves *combining* the characteristics of *parent* individuals from the *selected* population (ants are a special case).

GA : the characteristics of *pairs* of selected parents are combined by using a *crossover* mask (predefined or randomly generated) to generate two new individuals. If the crossover mask is set to the identity, then the two new individuals are clones of the two parents.

AIS negative selection : the selected parents become the basis of the new generation (which is topped up to the population size by *creating* sufficient new individuals). If the threshold is a constant value throughout the run, this has the effect that an individual, once selected, continues from generation to generation, and only the newly created individuals need be evaluated.

AIS clonal selection : in the main population new individuals are spawned as *clones* of each parent, with the number of clones being produced proportional to the parent's affinity; in the memory cell population, the selected parents become the basis of the new generation, and a new individual is spawned, as (a copy of) the best individual of the main population.

Swarms : a new individual is spawned from the sole parent and the highest affinity individual in that parent's neighbourhood group, with the intention of making the new individual "move towards" the best neighbour. The new position is derived from the parent's position and velocity, the velocity is modified to point towards the best neighbour, and the neighbourhood group is copied from the parent.

Ants : *no* individuals are specifically spawned for the next generation: each generation is *created* afresh from the path steps (whose characteristics are changed by the *mutate* step)

Mutate

Mutation involves altering the characteristics of single individuals in the population. It would be possible to unify spawning and mutation into a single *generate* stage, but since most algorithms consider these to be separate processes, we have followed that view, rather than strive for total generality at this stage. The mutation rate might be globally random, or based on the value of a characteristic or the affinity of each individual. How a characteristic is mutated depends on its type: a boolean might be flipped, a numerical value might be increased or decreased by an additive or multiplicative factor, etc.

GA, Swarms : individuals are mutated, usually randomly, in order to reintroduce lost values of characteristics; e*volutionary strategy* algorithms encode mutation rates as characteristics

AIS negative selection : no mutation occurs. (That is, the next generation consists of copies of the selected above threshold individuals, topped up with newly created individuals. An alternative, but equivalent, formulation in terms of this framework would

be to consider that *all* the individuals are *selected*, and that only the below-threshold individuals are *mutated*, into completely random individuals. However, this is at odds with the traditional description of the algorithm, and also with the view that mutation makes relatively small alterations.)

AIS clonal selection : the new *clone* individuals are mutated, by an amount inversely proportional to their affinity.

Ants : new pheromone is *laid* on each path step by an amount proportional to the affinity of the complete paths in which it occurs, and decreased by a constant *decay* factor.

3 Generalising across algorithms and implementations

Once we have all the algorithms in a common framework, we can see ways of generalising each in a natural manner (that is bio-inspired, but by a different aspect of biology). We discuss two such cases here – niching and elitism – and outline other possibilities.

Niching

Some population-based algorithms include "niching": developing sub-populations separately, with occasional migration of individuals [Brits *et al* 2002] [Mahfoud 1995] [Watkins & Timmis 2004]. Niching is motivated by the biological evolution of populations on separate islands. It is useful for solving multi-objective problems, with sub-populations focussing on separate objectives, but we do not here consider that aspect. Here we are interested in its use for efficiently distributing the algorithm across multiple processors, by minimising the amount of communication (of details of the high affinity individuals) needed between processors. We use these ideas to add a *migrate* step to the generic framework.

In niching, we have *N islands* of separately developing sub-populations, with each island following the simple algorithm, and each having a neighbourhood that is of a subset of the other islands (capturing locality of islands). Every *g* generations, modify the current population by replacing *n* individuals (suitably *selected* to be of low affinity, or chosen randomly) with *n* migrants from the neighbourhood (suitably *selected* to be of high affinity, or chosen randomly). This results in the following specifics:

GA: migration is simple population replacement; the solution is the fittest individual across all islands.

AIS negative selection: migration is simple population replacement; the solution is the union of all above-threshold individuals across all islands.

AIS clonal selection : migration can occur for both the main population and the memory cell population; the solution is the union of all memory cell populations across all islands.

Swarms: migration is simple population replacement, where the replaced individuals copy in the position and velocity characteristics of the migrants, but retain their original neighbourhood characteristic; the solution is the fittest individual across all islands. (Swarms admit another distribution strategy that can be efficiently implemented if the nearest individual neighbourhood relation is that of a simple ring topology. A single swarm can be distributed across a ring of processors, with the only communication between processors being the very local neighbourhood properties.)

Ants: migration is simple population replacement; the solution is the fittest individual (shortest path) across all islands. The (affinity of the) migrated paths will affect only the pheromone update stage; the migrated paths need not be recreated in the next generation.

Elitism

Some GAs include *ad hoc* elitism: copying the best individual(s) into the next generation, in order to preserve the currently best solution (and hence make the best solution monotonic with generations). This is not a particularly bio-inspired process. AIS algorithms with their (constant) threshold selection, on the other hand, are naturally elitist: if an antibody exceeds the threshold, (a copy of) it survives in future generations. We use these ideas to modify the *spawn* step, and add generic elitism to the framework.

When spawning the next generation, copy the n best of the previous population, as well as spawning any new individuals from the selected parents. Also, these particular n individuals should be exempt from mutation in this stage. This results in the following specific modifications:

GA : n fewer individuals need to be spawned by crossover

AIS : no change, provided n individuals are above the threshold

Swarms : the solution is a property of the position characteristic only, and the position is modified by the velocity. So the previous best solution is copied, with its velocity set to zero, so that it "hovers" over the current best solution

Ants : an "elite ant" recreates the current shortest path, ensuring that the solution remains in the population, and that its steps get their pheromone levels updated

Other generalisations

The generic framework allows further features of one specific algorithm to be generalised to the others.

- Evolutionary Strategies encode the mutation rates as characteristics: a similar approach can be used in the other algorithms. For example, the ant algorithm could allow the pheromone decay rate to be a characteristic.
- Genetic Algorithms use crossover to combine characteristics of parents: a similar spawn operator can be used in the other algorithms. For example, AIS clonal selection could spawn new antibodies by crossover.

- Traditionally unchanging characteristics could be mutated, for example, swarm neighbourhood.
- AIS use affinity-based mutation, to preferentially shake up poorer solutions: all mutation schemes could take this approach.
- AIS clonal selection increases the number of good solutions by affinity based cloning (thereby increasing their probability of being selected, so allowing more exploration of the nearby space). The other algorithms could be adapted to variable population size with cloning.
- The range of selection strategies can be employed across all the algorithms that have a non-trivial selection stage. In particular, AIS clonal selection has two populations: selection strategies could be used on the memory cell population too.

4 The prototype implementation

There is much opportunity for *parallelism* in these algorithms: individuals can (to some degree) be evaluated, selected, and created in parallel. This suggests efficiency gains by executing these algorithms on parallel hardware.

FPGAs and Handel-C

We chose as our prototype implementation platform a small grid of FPGAs, executing the framework implemented in Handel-C.

An FPGA is *programmable hardware*: its array of logic gates can be configured and connected for each specific program. This removes the need to fetch and decode instructions, fetch data, and store results; it is programmed to be a direct hardware representation of the code. It can be configured to provide genuine parallel execution (see for example [Brown & Rose 1996]). So each individual FPGA can host multiple individuals executing in parallel, and multiple FPGAs allow distributed implementations.

The framework described above has been prototyped as a proof of concept. It demonstrates that a suitably flexible generic framework can indeed be developed to support multiple classes of population-based algorithms, and that it can be distributed on a grid of FPGAs. It has been tested on an array of four FPGA algorithm engines, each connected to a fifth monitoring FPGA, in turn connected to a PC.

The prototype framework is implemented in Handel-C [Celoxica 2004], a (relatively) high level language designed specifically for writing applications for FPGAs. Handel-C code is compiled down into the relevant FPGA net-list, which specifies how the FPGA is to be configured.

Handel-C is based on the process calculus of CSP (Communicating Sequential Processes) [Hoare 1985]. Handel-C is essentially an executable subset of CSP [Stepney 2003], with some extensions to support FPGA hardware. In particular, it has explicit support for parallel execution of processes:

```
for (i=0; i < imax; i++) {
  // the imax iterations execute in sequence
  // and occupy space independent of imax
}

par (i=0; i < imax; i++) {
  // the imax iterations execute in parallel
  // and occupy space proportional to imax
}
```

Being based on CSP, Handel-C uses *synchronous* communication between its parallel processes. There is currently no Handel-C language support for programs distributed across multiple FPGAs, and such configurations do not support synchronous communication between chips as a primitive. It would have been possible to design a protocol to implement this, allowing the distributed program to be (very close to) a pure Handel-C program. However, the communication between chips is deliberately restricted, to just the occasional migration data. So for this prototype, a simple handshaking protocol has been used, and the inter-chip communication hidden in a wrapper.

The implemented framework

The prototype implementation of the framework provides much of the functionality described above. The genericity means that there are many parameters and options: the prototype includes support for specific options, and hooks for a range of user-definable functions. The framework is structured so that the basic algorithm needs no alteration: the user merely selects certain features (such as population sizes, characteristics, mutation rates, style of creation and selection, crossover masks, number of FPGAs, and number of islands per chip), and provides the code for certain functions (the evaluation function and stopping condition, and, as required, creation, selection, spawning, and mutation functions).

The framework code and user-defined functions, both written in Handel-C, are compiled on the PC, then downloaded on to the FPGA array to run. The Handel-C compiler optimises away dead code, so options that are not selected by the user (such as various choices of creation or selection functions) do not appear in the compiled code.

When the algorithm terminates, the relevant results are communicated back to the PC. It is also possible to return intermediate results every generation, to allow investigation of the performance, or for debugging, but this introduces a communication bottleneck.

Each individual is represented as a bit string, with each bit or combination of bits in the string representing a characteristic. A user-specified flag selects whether these bits are initialised to 0s, to 1s, to random bit values, or to a user-defined alternative. Each FPGA chip holds a certain number of islands, each of which holds its individuals. Migration information is passed between the islands, and hence between the chips, as required.

Ideally, all individuals should be able to evaluate their own affinity in parallel, by a call to the user-defined evaluation function. For completely parallel implementation on an

FPGA, this would require one copy of the function per individual. This can result in much of the FPGA's resource being used by evaluation functions, limiting the number of individuals per chip. The framework instead allows a trade-off between number of individuals and amount of parallelism. During certain of the algorithm steps, the individuals in an island are considered to be grouped into f *families*, each with m *members*, giving at total of $f \times m$ individuals. Families are processed in parallel, but the members of a family are processed in sequence. Thus each family requires only one copy of each function, and space is proportional to f and execution time is proportional to m:

```
par (family=0; family < f; family++) {
  for (member=0; member < m; member++) {
    // ... code for  individual[family*m + member]
  }
}
```

The prototype implementation provides a timer function, to help the user chose a suitable trade-off for each step, and for each application.

The probabilistic nature of (parts of) the algorithm require the use of random numbers. The implementation provides one 32-bit linear feedback shift register per family for this purpose.

Tournament selection is used to divide the population into *teams*. If no tournament is required, the entire population forms one large team. Then the appropriate selection method is used on each team in parallel.

Restrictions due to the platform choice

Some of the design decisions for the framework prototype are due to specific features and limitations of FPGAs and Handel-C, and different platform choices could result in different decisions. For example, the use of *families* is to cope with the limited size of the FPGAs.

Another example of this choice is the selection implementation. Although each team performs selection in parallel, the selection within each team is sequential. One might think that roulette wheel, or even a random, selection from a collection of individuals could be performed in parallel. However, this would result in the need for many parallel accesses to random number generators, and the FPGA's silicon would quickly become dedicated to these. Certain parts of the selection can be performed in parallel, for example, to find the n best, where each individual can read the affinity of all its team-mates in parallel. Even so, care needs to be taken, because the highly interconnected read accesses can result in quite complex (and therefore slow to compile) routing.

Handel-C supports variable bit-width values, requiring explicit casting between values with different widths. This can lead to arcane code, particularly when trying to write generic routines. For example, consider the case where some particular size n (such as population size) is a power of 2. Then the variable that stores the size is one bit larger

than the variable used for indexing into an array of this size, running from 0 to $n-1$, so comparing the index and the size requires some fancy casting.

5 Preliminary results

Sizes: The number of (families of) individuals possible per chip varies depending on the settings. For example, if no survival, generation, or niching is done, it is possible to have 30–40 individuals per chip, each with 8 bit-characteristics. With all the capabilities turned on, this number drops to about 18 individuals run sequentially, or four if run in parallel, the reduction being due to the increased routing and copies of code. The limiting constraint is routing tables rather than logic gates: every individual is accessed by each of the six algorithm stage functions, and the implementation uses all the routing tables, but only about 10-15% of the logic gates. Similar size results apply when hosting two islands on a single chip: this halves the *total* population possible, because of code replication and routing constraints.

The FPGAs being used (300K gate Xilinx SpartanIIE chips) are relatively small: it was thought more important for this proof of concept work to get the maximum *number* of FPGAs for the budget, rather than the maximum size of each one. Clearly, more individuals would be possible with larger FPGAs. However, the architectural design needs to be done carefully to optimise use of the resources: design experience of these style of algorithms for FPGAs is still in its infancy.

Parallelism: How much speedup does *parallelism* give?

The experiment compares running four individuals in parallel (the most supportable in this experiment) against running four in sequence, solving a simple optimisation problem. Linear speedup would result in a 400% improvement, but the parallel form has a speed-up of only about 30% over the sequential form. This low value indicates that there is still a great deal of sequential execution in this prototype implementation. This sequential bottleneck occurs mainly in the *select* stage of the framework: the speedup in the *evaluate* stage (calculation of the affinity function) is essentially linear. More complicated evaluation functions would therefore result in an increase in the parallel efficiency, but would also increase the demand on silicon resource if the complicated functions physically took up more space.

Parallelism versus population size: more parallelism takes up more silicon, resulting in few individuals being supportable. Parallelism lets the algorithm run faster, in that each generation takes less time to execute, but larger populations allow greater diversity and exploration of the search space per generation, so the algorithm could require fewer generations to find a solution. What is better: more parallelism or a larger population?

The experiment compares running four individuals in parallel (the most supportable in this experiment) against running 12 in sequence (the most supportable sequential individuals in this experiment). The parallel form gives no speed-up overall compared to

the sequential form. However, looking at only the *evaluate* stage shows the sequential form taking about twice as long as the parallel form. Given the linear speedup noted above, the sequential case takes 12 times as long per generation, and so is executing only about one sixth the number of generations before finding a solution. Even so, the parallel form is taking less time overall (in the *evaluate* state), suggesting that parallelism outweighs population size in this case.

However, these experiments need to be tried on larger populations, and a wider range of affinity function complexity (which affects the parallel to sequential population size ratio), before any more definitive statements can be made.

Niching: What is the effect of using multiple chips? How much speedup does *distribution* give?

The experiment compares running four individuals in parallel on one chip versus four individuals in parallel on each of the five chips (20 individuals in total), migrating the two best individuals every 100 generations. There is a speedup of about a factor of two (over the whole algorithm run, not just the *evaluate* stage). These experiments need to be repeated for different migration rates to see if there is an optimum rate.

6 Discussion and future work

The unified framework allows the generalisation of concepts from across a range of algorithms types, for example, elitism and niching. Thus we have a chimerical computational framework that is inspired by biology, but not restricted to any one particular biological domain. Implementation of the framework on parallel and distributed architectures is (relatively) straightforward, and provides performance benefits (although currently significantly less than linear improvement). So this proof of concept has shown that the approach to generalising and parallelising population-based algorithms is feasible and useful. Thus it is worth pursuing the approach with more rigour and detail.

The relatively small speedups indicate the need for removing the remaining sequential bottlenecks, and the constraints on parallel individuals caused by routing limitations indicate the need for a more sophisticated parallelisation architecture targeted at the opportunities and limitations of FPGAs. Allowing six islands per chip to use the hardware for the six algorithm steps in a pipeline might provided further speedup.

An alternative distributed architecture would be to dedicate certain chips to algorithm steps, and move the individuals around. Different numbers of chips could be dedicated to each step, depending on the complexity of that step, to balance the processing load.

The prototype framework needs to be extended with more built-in options, and made more usable, by providing a configuration language for selecting parameter values, etc. It also needs to be made more flexible to accommodate other arrangements of FPGA arrays.

Future work also includes formalising the generic framework, to make it clearer which features of each algorithm are being represented and captured, and to generalise and include more details and capabilities from the range of variant algorithms in the literature.

A more rigorous framework will allow *analysis* at the generic and specific levels, *comparison* of instantiations, and further generalisations of various properties.

7 Acknowledgments

We would like to thank Wilson Ifill and AWE, who provided funding for the FPGAs used in this work. Also thanks to Neil Audsley and Michael Ward for turning a large box of components into a usable FPGA grid, and to Fiona Polack and Jon Timmis for detailed comments on earlier versions.

8 References

[1] E.W. Bonabeau, M. Dorigo, G. Theraulaz. *Swarm Intelligence: from natural to artificial systems*. Addison Wesley, 1999

[2] R. Brits, A.P. Engelbrecht, F. van den Bergh. A niching Particle Swarm Optimizer. *4th Asia-Pacific Conference on Simulated Evolution and Learning*, 2002.

[3] S. Brown, J. Rose. Architecture of FPGAs and CPLDs: a tutorial. *IEEE Design and Test of Computers*, **13**(2):42-57, 1996.

[4] Celoxica. *Handel-C Reference Manual, development kit v3.0*. 2004 http://www.celoxica.com/techlib/files/CEL-W0410251JJ4-60.pdf

[5] V. Cutello, G. Nicosia, M. Pavone. Exploring the capability of immune algorithms: a characterization of hypermutation operators. *ICARIS 2004*, LNCS **3239**:263-276

[6] L.N. de Castro, J. Timmis. *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer, 2002

[7] S.M. Garrett. Parameter-free, adaptive clonal selection. *CEC 2004*, pp 1052-1058. IEEE Press, 2004

[8] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985

[9] J. Kennedy, R.C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann, 2001

[10] J. Kim, P.J. Bentley. Immune memory in the dynamic clonal selection algorithm. *ICARIS 2002*, pp 59-67, Kent, 2002

[11] S.W. Mahfoud. A comparison of parallel and sequential niching methods. In L.J. Eshelman, ed, *Proc. 6th International Conference on Genetic Algorithms*, pp 136-143. Morgan Kaufmann, 1995.

[12] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996

[13] S. Stepney. *CSP/FDR2 to Handel-C translation*. Technical Report YCS-2003-357, University of York. June 2003.

[14] S. Stepney, R.E. Smith, J. Timmis, A.M. Tyrrell, M.J. Neal, A.N.W. Hone. Conceptual Frameworks for Artificial Immune Systems. *Int. J. Unconventional Computing*. **1**(3) 2005

[15] A. Watkins, J. Timmis. Exploiting Parallelism Inherent in AIRS, an Artificial Immune Classifier. *ICARIS 2004*, LNCS **3239**:427-438. Springer, 2004.