COORDINATING EVOLUTION
Designing a Self-adapting Distributed Genetic Algorithm

Nikolaos Chatzinikolaou
School of Informatics, University of Edinburgh, Informatics Forum, Crichton Street, Edinburgh, Scotland, U.K.

Keywords: Genetic algorithms, Distributed computation, Multi-agent learning, Agent coordination.

Abstract: In large scale optimisation problems, the aim is to find near-optimal solutions in very large combinatorial spaces. This learning/optimisation process can be aided by parallelisation, but it normally is difficult for engineers to decide in advance how to split the task into appropriate segments attuned to the agents working on them. This paper chooses a particular style of algorithm (a form of genetic algorithm) and describes a framework in which the parallelisation and tuning of the multi-agent system is performed automatically using a combination of self-adaptation of the agents plus sharing of negotiation protocols between agents. These GA agents are optimised themselves through the use of an evolutionary process of selection and recombination. Agents are selected according to the fitness of their respective populations, and during the recombination phase they exchange individuals from their population as well as their optimisation parameters, which is what lends the system its self-adaptive properties. This allows the execution of optimal optimisations without the burden of tuning the evolutionary process by hand. The architecture we use has been shown to be capable of operating in peer to peer environments, raising confidence in its scalability through the autonomy of its components.

1 INTRODUCTION

1.1 Genetic Algorithms

Since their inception by John Holland in the early 70's (Holland, 1975) and their popularisation over the last few decades by works such as (Goldberg, 1989), Genetic Algorithms (GAs) have been used extensively to solve computationally hard problems, such as combinatorial optimisations involving multiple variables and complex search landscapes.

In its simplest form, a GA is a stochastic search method that operates on a population of potential solutions to a problem, applying the Darwinian principle of survival of the fittest in order to generate increasingly better solutions. Each generation of candidate solutions is succeeded by a better one, through the process of selecting individual solutions from the current generation according to their relative fitness, and applying the genetic operations of crossover and mutation on them. The result of this process is that later generations consist of solution approximations that are better than their predecessors, just as is natural evolution.

GAs have proved to be flexible and powerful tools, and have been successfully applied to solve problems in domains too numerous and diverse to list here (examples are provided in surveys such as (Ross and Corne, 1995)). Despite their widespread success, however, there's still a number of issues that make their deployment by the uninitiated a non-trivial task. Two themes that keep recurring in the literature are parameter control, which involves determining the optimal set of parameters for a GA; and parallelisation, which involves distributing the computational load of a GA between multiple computational units. It is on these two themes that this research concentrates.

1.2 Scope of this Paper

The principal objective of this research is the design and implementation of a scalable architecture that will enable large numbers of agents (in the form of computers participating in an open network) to cooperate in order to solve complex problems that would require a prohibitively long time to solve in a standard, non-parallel GA.

The architecture we propose aims to address both of the issues of parameter control and parallelisation at the same time, by using a novel approach: that of implementing an open, peer-to-peer network of inter-

Chatzinikolaou N. (2010).
COORDINATING EVOLUTION - Designing a Self-adapting Distributed Genetic Algorithm.
In *Proceedings of the 12th International Conference on Enterprise Information Systems - Artificial Intelligence and Decision Support Systems*, pages 13-20
DOI: 10.5220/0002871200130020
Copyright © SciTePress

connected GAs, in which no a priori assumptions will need to be made regarding their configuration. Instead, the capacity of each GA for solving the problem at hand will be evolved itself during runtime. In addition, the self-organising nature of the system will ensure that all and any available resources (participating computers) will be efficiently exploited for benefiting the system.

The general concept underlying this research involves exploiting self-organisation and coordination among agents through the use of mobile protocols described using a process calculus, and in particular the Lightweight Coordination Calculus (LCC) as specified in (Robertson, 2004a; Robertson, 2004b).

2 RELATED WORK

2.1 Self Adaptation in GAs

In every application of a GA, the designer is faced with a significant problem: tuning a GA involves configuring a variety of parameters, including things such as population sizes, the operators used for selection and mutation, type and size of elitism etc. As a general case, before a GA can be deployed successfully in any problem domain, a significant amount of time and/or expertise has to be devoted to tuning it.

As a result, numerous methods on parameter optimisation have appeared over the years (Eiben et al., 2000). These generally fall in one of two categories:

- Parameter Tuning, in which the set of GA parameters are determined a priori, and then applied to the GA before it is executed.
- Parameter Control, in which the parameters change (adapt) while the GA is running.

It was discovered early on (Hesser and Männer, 1991; Tuson, 1995) that simple a priori parameter tuning is generally insufficient to produce adequate results, as different stages in the evolutionary process are likely to require different parameter values. Therefore, in our research we concentrate on dynamic parameter adaptation, along the lines of work presented in (Back, 1992; Eiben et al., 2000; Meyer-Nieberg and Beyer, 2006).

2.2 Parallel GAs

Even after a set of optimal parameters has been established, traditional (canonical) GAs suffer from further difficulties as problems increase in scale and complexity. (Nowostawski and Poli, 1999) has identified the following:

- Problems with big populations and/or many dimensions may require more memory than is available in a single, conventional machine.
- The computational (CPU) power required by the GA, particularly for the evaluation of complex fitness functions, may be too high.
- As the number of dimensions in a problem increases and its fitness landscape becomes more complex, the likelihood of the GA converging prematurely to a local optimum instead of a global one increases.

To some extend, these limitations can be alleviated by converting GAs from serial processes into parallel ones. This involves distributing the computational effort of the optimisation between multiple CPUs, such as those in a computer cluster.

(Lim et al., 2007) identify three broad categories of parallel genetic algorithm (PGA):

- **Master-slave PGA.**
  This scheme is similar to a standard, or canonical, genetic algorithm, in that there is a single population. The parallelisation of the process lies in the evaluation of the individuals, which is allocated by the master node to a number of slave processing elements. The main advantage of master-slave PGAs is ease of implementation.
- **Fine-grained or Cellular PGA.**
  Here we have again a single population, spatially distributed among a number of computational nodes. Each such node represents a single individual (or a small number of them), and the genetic operations of selection and crossover is restricted to small, usually adjacent groups. The main advantage of this scheme is that it is particularly suitable for execution on massively parallel processing systems, such as a computer system with multiple processing elements.
- **Multi-population or Multi-deme or Island PGA.**
  In an island PGA there are multiple populations, each residing on a separate processing node. These populations remain relatively isolated, with "migrations" taking place occasionally. The advantages of this model is that it allows for more sophisticated techniques to be developed.

The approach of parallelisation of genetic algorithms becomes even more appropriate in the light of recent developments in the field of multi-processor computer systems (Munawar et al., 2008), as well as the emergence of distributed computing, and particularly the new trend towards cloud computing (Foster et al., 2008).

The work presented in this paper was originally influenced by (Arenas et al., 2002), which follows the island paradigm (Tanese, 1989; Belding, 1995). This scheme, however, is just one of many. Among others, (Cant-Paz, 1998; Nowostawski and Poli, 1999; Alba and Troya, 1999) each provide an excellent coverage of the work done on this theme.

2.3 Multi-agent Coordination

This architecture is based on the concept of coordinating the interactions between individual GA agents using shared, mobile protocols specified in the Lightweight Coordination Calculus (LCC) (Robertson, 2004b; Robertson, 2004a). It is the coordination between the agents that is going to guide the GA agents as a system, by providing an open and robust medium for information exchange between them, as well as the necessary evolutionary pressure.

Apart from being used to specify declaratively the interactions between agents, LCC is also an executable language, and as such it will be used to dictate the interactions that will "glue" the GA agents together. LCC is designed to be a flexible, multi-agent language, and has proved successful in the implementation of open, peer-to-peer systems, as was demonstrated by the the *OpenKnowledge* framework (Robertson et al., 2006).

3 ARCHITECTURE

3.1 "Intra-agent" Genetic Algorithm

The system we have developed consists of a network of an arbitrary number of identical agents. Each agent contains an implementation of a canonical GA that acts on a local population of genomes, performing standard crossover and mutation operators on them. We call this GA the "intra-agent GA", and its steps are that of a typical GA. For a population of size n:

1. Evaluate each member of the population.
2. Select n pairs of parents using roulette wheel selection.
3. For each pair of parents, recombine them and mutate the resulting offspring.
4. Repeat from step 1 for the newly created population.

3.2 "Inter-agent" Genetic Algorithm

At the same time, every agent has (and executes locally) a copy of a shared, common LCC protocol that dictates how this agent coordinates and shares information with its peers. The result of this coordination is a secondary evolutionary algorithm, which evolves not the genomes in each agent but the agents themselves, and in particular the population and parameters that each of them use for their respective "intra-agent" GA. We call this secondary GA the "inter-agent" GA:

1. Perform a number of "intra-agent" GA iterations.

   This step is equivalent to step 1 of the "intra-agent" GA, as it essentially establishes a measure of that agent's overall fitness. This fitness is based on the average fitness of all the individual genomes in the agent's population, as established by the "intra-agent" GA.

2. Announce agent's fitness to neighbouring peers, wait for them to announce their own fitness, and select a fit mate using roulette wheel selection.

   Again, this is similar to step 2 above. The only difference this time is that every agent gets to select a mate and reproduce, as opposed to the "intra-agent" GA where both (genome) parents are selected using roulette wheel selection.

3. Perform crossover between self and selected agent (population AND parameters).

   Here we have the recombination stage between the two peers, during which they exchange genomes from their respective populations (migration) as well as parameters. The new parameters are obtained by averaging those of the two peers, and adding a random mutation amount to them.

4. Repeat from step 1.

By combining the GA parameters of the agents in addition to the genomes during the migration/recombination stage (step 3), we ensure that these parameters evolve in tandem with the solution genomes, and thus remain more-or-less optimal throughout the evolutionary process. It is this characteristic that lends our system its self-adaptive properties. The idea behind this approach is that we use the principle that forms the basis of evolutionary computation to optimise the optimiser itself.

3.3 Agent Autonomy and Motivation

As is the norm in most multi-agent systems, each agent in our implementation is fully autonomous. This autonomy is evident in the fact that the agents are able to function even without any peers present, or when peer-to-peer communication is compromised.

This characteristic has the obvious advantage of improved robustness.

However, an agent operating in isolation will not be able to evolve its own GA parameters, and hence its performance will remain at a steady, arbitrary level dictated by the current set of GA parameters it uses.

This is where the motivation of the agents to interact with their peers stems from: by having agents collaborate/breed with their peers, the system as a whole evolves, adapts, and improves its performance.

3.4 Comparison with Existing Systems

Our architecture shares some characteristics with other approaches in the field. For example, in a typical "island" based parallel GA (Arenas et al., 2002) there is usually migration of genomes, but no evolution of GA parameters. On the other hand, in typical meta-GA implementations (Grefenstette, 1986; Clune et al., 2005), there is adaptation of the genetic operators but no parallelisation of the evolutionary process.

Key to our approach is the fact that the optimisation of the GA agents does not happen before they are used to solve the actual problem at hand, but instead their optimisation happens as a continuous, dynamic process during their operational lifetime. This is of particular relevance, especially in enterprise environments where requirements between different applications fluctuate significantly.

4 PRELIMINARY EXPERIMENTS

4.1 Objectives

Following the implementation of our architecture, we performed a number of experiments in order to evaluate its performance compared to (a) traditional (canonical) genetic algorithms, and (b) an island-based parallel genetic algorithm with simple population migration.

4.2 Test Case

As our optimisation test case, we used the Rastrigin equation, which is widely adopted as a test function in the field. Its general form is given in equation 1.

$$F(\bar{x}) = kA + \sum_{i=1}^{k}(x_i^2 - Acos(2\pi x_i)) \quad (1)$$

This is a minimisation problem, which implies that the aim of the GA is to make the fitness measure as small as possible, with the optimal value being zero.

In all our experiments, the steepness A was set to 10, and the number of Rastrigin variables k was set to 30. The range of x was -0.5 to +0.5, encoded in 16-bit Gray code. The choice of these parameters was influenced by similar experiments in the literature (e.g. (Yoshihiro et al., 2003)).

4.3 Intra-agent GA Configuration

Since we are mainly interested in observing the adaptation that occurs in the individual agents' GAs themselves during the evolutionary process, we tried to keep things simple and controllable by only allowing a single parameter to adapt: that of the mutation rate. All the rest of the parameters were kept constant: the population size was fixed at 100 individuals, and no elitism was used. Also, and despite being at odds with established GA practice, the intra-agent crossover rate was set to 0 - effectively disabling it. Again, this was done so as to better observe the impact of the mutation rate adaptation on the overall fitness progression. Finally, the roulette wheel selection scheme was used.

It must be noted at this point that these parameters were deliberately selected with simplicity in mind rather than performance. In fact, it can be argued that the parameter selection described above is rather inefficient, yet in being so, it allows us to better observe the impact of the design of our architecture in the overall performance of the GA.

4.4 Inter-agent GA Configuration

Regarding the set-up of the multi-agent system, we conducted experiments using 3, 6, 12, 24, 48 and 96 agents at a time. We also had to specify how many iterations each agent would perform before crossover with the other agents occurred. This parameter is dependent on the available bandwidth available by the computational platform on which the system is deployed. In our case, this value was set to 10, which was empirically found to perform best for our platform.

We performed runs using three different schemes for the inter-agent crossover:

1. No crossover: essentially, each agent's GA run in isolation from the others. This experiment was implemented using MATLAB's Genetic Algorithm toolbox, which provided us with an independent and solid performance benchmark.
2. Population crossover: during the inter-agent crossover phase, only individuals between the dif-

ferent sub-populations were exchanged, while GA parameters were not recombined.

3. Full crossover: this is the scheme that we propose in our architecture. In this case, the parameters of the agents' GA were also recombined in addition to the population exchange.

The first two schemes were implemented not as an integral part of our architecture, but instead as a benchmark for evaluating its performance. Essentially, scheme 1 emulates a set of traditional, serial GAs using static parameters covering the full available spectrum, while scheme 2 emulates a typical "island-based" GA, with migration taking place among individual GAs that - again - use static parameters.

For the first and second configuration, each agent was given a mutation rate equal to half that of the previous agent, starting at 1.0. This means that, as more agents were introduced in the system, their mutation rate was reduced exponentially. The reason for this decision is the fact that, in almost all GAs, later generations benefit from increasingly smaller mutation rates (Eiben et al., 2000).

For the second and third case, agents were selected by their peers for crossover using roulette wheel selection, where each agent's fitness was dictated by the average fitness of its current population.

Finally, in order to compensate for the stochastic nature of the experiments and produce more meaningful results, all runs were executed 10 times and their output was averaged.

4.5 Evaluation of Performance

In general, the performance of a GA is measured by the time it takes to achieve the required result (usually convergence to a stable or pre-set fitness). In our experiments, however, we deemed it more appropriate to use the number of generations as a measure of performance. The reason for this is that the actual execution time depends on variables which are unrelated to the algorithm itself, such as the capabilities and load of the processing elements or the bandwidth of the network on which these reside.

In all of our experiments, the population size was the same, and thus the execution time required for the evaluation of every population (typically the most computationally intensive task in a GA) was also the same. Therefore, we can assume that the number of generations taken by the algorithm to converge is proportional to the actual time it would require on a benchmark computational system.

This, of course, does not take into account the overhead incurred by network communication; however, as this overhead is again more or less equivalent in all experiments, we can safely factor it out.

5 RESULTS

5.1 Speed of Convergence

For our first experiment, we executed runs using different numbers of agents and all three inter-agent crossover schemes. Each run was stopped as soon as a fitness of 50.0 was reached by any of the agents in that run. Figure 1 shows the relative performance of the three schemes (note that the x-axis is shown in logarithmic scale).

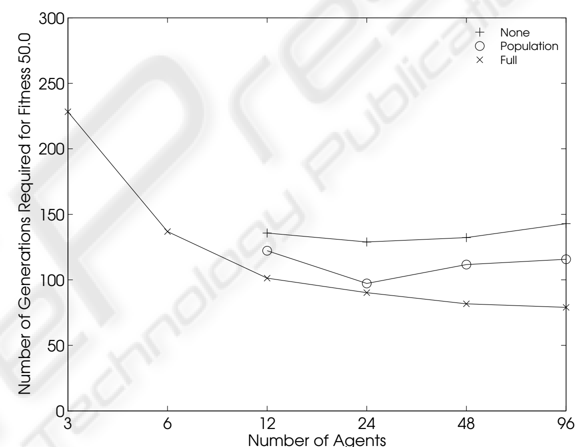

Figure 1: Relative speed performance of the three inter-agent crossover schemes.

As can be seen, the worst performer was the first scheme, which emulates a number of isolated sequential GAs. In addition to being the slowest, it also failed to reach the target fitness when using too few agents (n = 3 and n = 6) - the reason for this being that the agents' mutation rates were too high to allow them to converge to the target fitness.

The population exchange scheme performed significantly better in terms of speed, although it also failed to converge when few agents were used (again, the target fitness was not reached for n = 3 and n = 6).

The full crossover scheme performed even better in terms of speed, but its most significant advantage is the fact that it managed to reach the target fitness even when using few agents - although at the cost of more generations.

Finally, the downward slope of this scheme's curve as the number of agents increases, provides a first suggestion towards its superior scaling properties.

5.2 Quality of Solution

The next experiment involved executing runs for 1000 generations each, again using all three inter-agent crossover schemes for different numbers of agents. This allowed us to see how close to the optimal fitness of 0.0 each configuration converged.

Figures 2, 3 and 4 show the resulting graphs from these runs, with the actual fitness results provided in table 1. The y-axis of the graphs has been made logarithmic in order to improve the legibility of the plots.

From these results, it becomes obvious that using the full crossover scheme achieves the best solution in terms of quality, in addition to being the fastest of the three.

Furthermore, it is becoming more obvious at this stage that the full crossover scheme scales significantly better as the number of agents increases. The first two schemes seem to be "hitting a wall" after the number of agents is increased beyond 24. For the case of the full crossover, however, adding more agents seem to be contributing to the performance of the system all the way up to, and including, $n = 96$.

Finally, the ability of this scheme to perform well even when using a small number of agents can also be seen in figure 4.

Table 1: Best (minimum) fitness after 1000 generations.

Scheme	Best Fitness
1 (No crossover)	6.06 (at n=96)
2 (Population crossover)	1.57 (at n=48)
3 (Full crossover)	0.17 (at n=96)

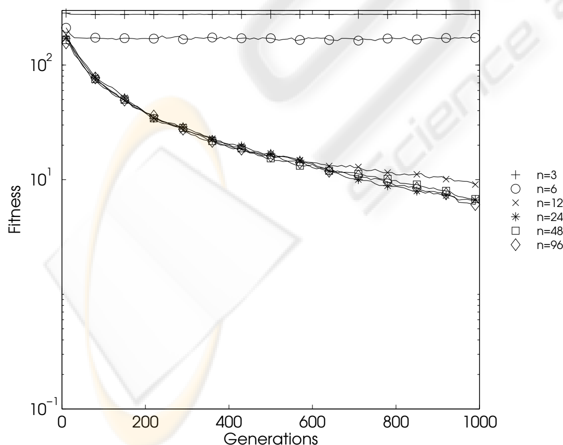

Figure 2: Run of 1st scheme (no inter-agent crossover) for different numbers of agents.

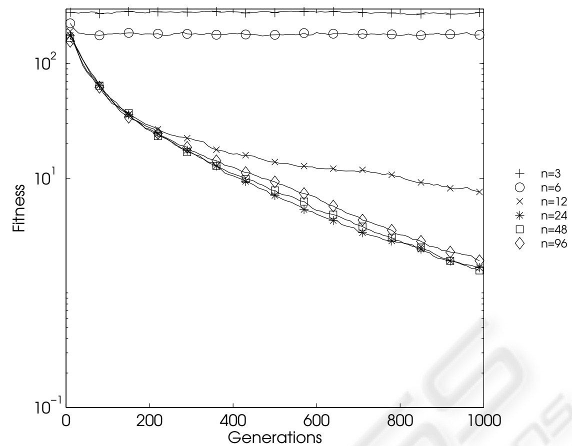

Figure 3: Run of 2nd scheme (population inter-agent crossover) for different numbers of agents.

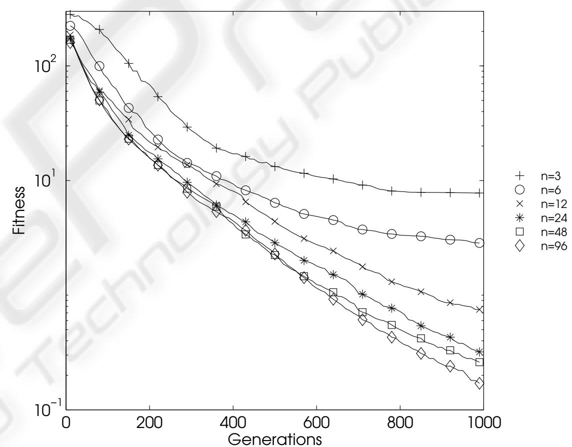

Figure 4: Run of 3d scheme (full inter-agent crossover) for different numbers of agents.

5.3 Adaptation of the Mutation Rate

As a final investigation on how the mutation rate adapts in the full inter-agent crossover scheme, we plotted the progress of the best agent's mutation rate against the generations, in a typical run using three agents. Figure 5 illustrates the results (again using a logarithmic y-axis). From this plot, we can see that the mutation rate drops exponentially in order to keep minimising the fitness, which agrees with our expectations.

6 CONCLUSIONS

The results presented above are encouraging, as they prove that this preliminary version of the architecture

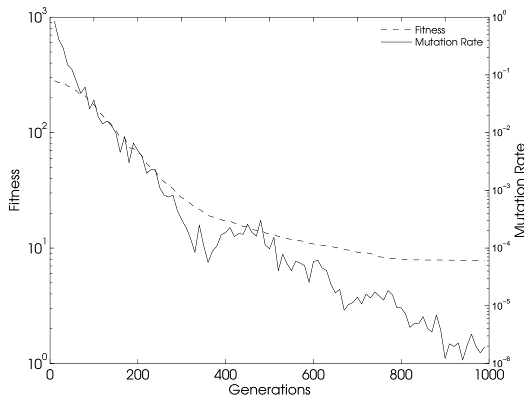

Figure 5: Adaptation of the mutation rate.

we propose is effective. By distributing the load among multiple agents, the system manages to converge to near-optimal solutions in relatively few generations.

The most important contribution, however, is the fact that, by applying the principle of natural selection to optimise the GA agents themselves, the evolutionary algorithm becomes self-adaptive and thus no tuning is required. By eliminating the need for tuning and thus taking the guesswork out of GA deployment, we make evolutionary optimisation appeal to a wider audience. In addition, the peer-to-peer architecture of the system provides benefits such as improved robustness and scalability.

7 FUTURE WORK

7.1 Complete GA Adaptation

As stated earlier, and in order to aid experimentation, only the mutation rate is currently adapted in our system. This is of course not very effective for a real-life application, where the full range of genetic algorithm parameters (population size, elite size, crossover rate, selection strategy etc.) needs to be adapted as the evolutionary process progresses. This extension is relatively straightforward to implement, as the basic characteristics of the architecture's implementation remain unaffected.

7.2 Asynchronous Agent Operation

Currently, all agents in our platform work synchronously. This means that they all perform the same number of iterations before every inter-agent crossover stage, with faster agents waiting for the slower ones to catch up. When the platform is deployed in a network consisting of computational elements of similar capabilities, this strategy works fine. However, in networks with diversified computational elements, this scheme is obviously inefficient. We aim to modify the current coordination protocol in order to resolve this, by using time- or fitness-based cycle lengths rather than generation-based ones.

7.3 Extended Benchmarking

Although we have established that our platform performs significantly better than standard, canonical GAs (such as the basic MATLAB implementation we compared against), we can obtain a better picture of how our architecture compares with other systems in the field by performing more test runs and comparing results. For instance, the systems proposed by (Yoshihiro et al., 2003) and (Kisiel-Dorohinicki et al., 2001; Socha and Kisiel-Dorohinicki, 2002) seem to share some characteristics with our own system, even though the two architectures differ in their particulars. Even though our focus is more on the openness of the system, it would be helpful to have an idea of the relative performance of our system with the status quo. However, at the time of writing, implementations of these systems were not readily accessible.

We also intend to perform more benchmarks using alternative test functions, such as the ones presented in (Schwefel, 1981; Ackley, 1987; Michalewicz, 1996). This way we will be able to better assess the capability of our system to adapt to different classes of problem, and will give us significant insight into which evolutionary behaviour (expressed as the evolutionary trajectory of each parameter) better suits each class of problem.

7.4 Additional Solvers

Finally, it will be interesting to take full advantage of the openness inherent to our architecture and LCC, by allowing additional kinds of solvers to be introduced in the system (e.g. gradient search, simulated annealing etc.). This will require the re-design of our protocol regarding the inter-agent crossover, or possibly the co-existence of more than one protocol in the system. We believe that the effort required will be justified, since, by extending our architecture in this way, we will effectively be creating an open, peer-to-peer, self-adaptive hybrid optimisation platform.

ACKNOWLEDGEMENTS

This research is funded in part by the EPSRC.

REFERENCES

Ackley, D. H. (1987). *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, Norwell, MA, USA.

Alba, E. and Troya, J. M. (1999). A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52.

Arenas, M. G., Collet, P., Eiben, A. E., Jelasity, M., Guervós, J. J. M., Paechter, B., Preuß, M., and Schoenauer, M. (2002). A framework for distributed evolutionary algorithms. In *PPSN VII: Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, pages 665–675, London, UK. Springer-Verlag.

Back, T. (1992). Self-adaptation in genetic algorithms. In *Proceedings of the First European Conference on Artificial Life*, pages 263–271. MIT Press.

Belding, T. C. (1995). The distributed genetic algorithm revisited. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 114–121, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Cant-Paz, E. (1998). A survey of parallel genetic algorithms. *Calculateurs Paralleles*, 102.

Clune, J., Goings, S., Punch, B., and Goodman, E. (2005). Investigations in meta-gas: panaceas or pipe dreams? In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 235–241, New York, NY, USA. ACM.

Eiben, A. E., Hinterding, R., Hinterding, A. E. E. R., and Michalewicz, Z. (2000). Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3:124–141.

Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2008). Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA.

Grefenstette, J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 16(1):122–128.

Hesser, J. and Männer, R. (1991). Towards an optimal mutation probability for genetic algorithms. In *PPSN I: Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, pages 23–32, London, UK. Springer-Verlag.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.

Kisiel-Dorohinicki, M., Socha, K., and Communication, S. T. E. (2001). Crowding factor in evolutionary multiagent system for multiobjective optimization. In *Proceedings of IC-AI01 International Conference on Artificial Inteligence*. CSREA Press.

Lim, D., Ong, Y.-S., Jin, Y., Sendhoff, B., and Lee, B.-S. (2007). Efficient hierarchical parallel genetic algorithms using grid computing. *Future Gener. Comput. Syst.*, 23(4):658–670.

Meyer-Nieberg, S. and Beyer, H.-G. (2006). Self-adaptation in evolutionary algorithms. In *Parameter Setting in Evolutionary Algorithm*, pages 47–76. Springer.

Michalewicz, Z. (1996). *Genetic algorithms + data structures = evolution programs (3rd ed.)*. Springer-Verlag, London, UK.

Munawar, A., Wahib, M., Munetomo, M., and Akama, K. (2008). A survey: Genetic algorithms and the fast evolving world of parallel computing. *High Performance Computing and Communications, 10th IEEE International Conference*, pages 897–902.

Nowostawski, M. and Poli, R. (1999). Parallel genetic algorithm taxonomy. In *Proceedings of the Third International*, pages 88–92. IEEE.

Robertson, D. (2004a). A lightweight coordination calculus for agent systems. In *In Declarative Agent Languages and Technologies*, pages 183–197.

Robertson, D. (2004b). Multi-agent coordination as distributed logic programming. In *International Conference on Logic Programming*, Sant-Malo, France.

Robertson, D., Giunchiglia, F., van Harmelen, F., Marchese, M., Sabou, M., Schorlemmer, M., Shadbolt, N., Siebes, R., Sierra, C., Walton, C., Dasmahapatra, S., Dupplaw, D., Lewis, P., Yatskevich, M., Kotoulas, S., de Pinninck, A. P., and Loizou, A. (2006). Open knowledge semantic webs through peer-to-peer interaction. Technical Report DIT-06-034, University of Trento.

Ross, P. and Corne, D. (1995). Applications of genetic algorithms. In *On Transcomputer Based Parallel Processing Systems, Lecture*.

Schwefel, H.-P. (1981). *Numerical Optimization of Computer Models*. John Wiley & Sons, Inc., New York, NY, USA.

Socha, K. and Kisiel-Dorohinicki, M. (2002). Agent-based evolutionary multiobjective optimisation. In *in Proceedings of the Fourth Congress on Evolutionary Computation*, pages 109–114. press.

Tanese, R. (1989). Distributed genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 434–439, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Tuson, A. L. (1995). Adapting operator probabilities in genetic algorithms. Technical report, Master's thesis, Evolutionary Computation Group, Dept. of Artificial Intelligence, Edinburgh University.

Yoshihiro, E. T., Murata, Y., Shibata, N., and Ito, M. (2003). Self adaptive island ga. In *2003 Congress on Evolutionary Computation*, pages 1072–1079.