Dynamic Demes Parallel Genetic Algorithm

Mariusz Nowostawski
Information Science Department
The University of Otago
PO BOX 56, Dunedin, New Zealand
MNowostawski@infoscience.otago.ac.nz

Riccardo Poli
School of Computer Science
The University of Birmingham
Edgbaston, Birmingham B15 2TT, UK
R.Poli@cs.bham.ac.uk

***Abstract*—Dynamic Demes is a new method for the parallelisation of evolutionary algorithms. It was derived as a combination of two other parallelisation algorithms: the master-slave distributed fitness evaluation model and the static subpopulation model. In this paper we present the algorithm, perform a theoretical analysis of its performance and present experimental results where we compared Dynamic Demes with other algorithms.**

I. Parallel Genetic Algorithms

Sequential GAs have been shown to be very successful in many applications and in very different domains. However, there exist some problems in their utilisation which can all be addressed with some form of Parallel GA (PGA):

- For some kind of problems, the population needs to be very large and the memory required to store each individual may be considerable (for example in genetic programming [1]). In some cases this makes it impossible to run an application efficiently using a single machine, so some parallel form of GA is necessary.
- Fitness evaluation is usually very time-consuming. In the literature computation times of more than 1 CPU year have been reported for a single run in complex domains (e.g. see [2]). It stands to reason that the only practical way of provide this CPU power is the use of parallel processing.
- Sequential GAs may get trapped in a sub-optimal region of the search space thus becoming unable to find better quality solutions. PGAs can search different subspaces of the search space in parallel, thus making it less likely to become trapped by low-quality subspaces.

For the first two reasons PGAs are studied and used for applications on massively parallel machines [3], transputers [4], and also on distributed systems [5]. However, the most important advantage of PGAs is that in many cases they provide better solutions than single population-based algorithms, even when the parallelism is simulated on conventional machines. The reason is that multiple populations allow speciation, a process by which different populations evolve in different directions (i.e. toward different optima) [6]. For this reason Parallel GAs are not only an extension of the traditional GA sequential model, but they represent a new class of algorithms in that they search the space of solutions differently.

The way in which GAs can be parallelised depends on the following elements:

- How fitness is evaluated and mutation is applied
- If single or multiple subpopulations (demes) are used
- If multiple populations are used, how individuals are exchanged
- How selection is applied (globally or locally)

Depending on how each of these elements is implemented, several different methods of parallelising GAs can be obtained. These can be classified into eight classes:

1. Master-Slave parallelisation (also known as distributed fitness evaluation)
2. Static subpopulations with migration
3. Static overlapping subpopulations (without migration)
4. Massively parallel genetic algorithms
5. Dynamic demes (dynamic overlapping subpopulations)
6. Parallel steady-state genetic algorithms
7. Parallel messy genetic algorithms
8. Hybrid methods (e.g. static subpopulations with migration, with distributed fitness evaluation within each subpopulation)

In the following subsections we provide a short description of two parallelisation methods on which dynamic demes is based and to which it will be compared later on in the paper.

A. Master-Slave parallelisation

In this parallelisation method, also known as distributed fitness evaluation, the algorithm uses a single population and the evaluation of the individuals and/or the application of genetic operators are performed in parallel. Selection and mating are done globally, hence each individual may compete and mate with all the others. The operation that is most commonly parallelised is the evaluation of the fitness function. This is usually implemented by master-slave programs, where the master stores the population and the slaves evaluate the fitness, apply mutation, and sometimes exchange bits of the genome (as part of crossover).

The algorithm is said to be *synchronous*, if the master stops and waits to receive the fitness values for all the population before proceeding with the next generation. A synchronous master-slave GA has exactly the same properties as a simple GA, except its speed, i.e. this form of parallel GA carries out exactly the same search as a simple GA. An *asynchronous* version of the master-slave GA is also possible. In this case the algorithm does not stop to wait for any slow processors. For this reason the asynchronous master-slave PGA does not work exactly like a simple GA, but is more similar to parallel steady-state GAs. The difference lies only in the selection operator.

In an asynchronous master-slave algorithm selection waits until a fraction of the population has been processed, while in a steady-state GA selection does not wait, but operates on the existing population.

B. Static Subpopulations With Migration

The important characteristics of the class of static subpopulations with migration parallel algorithms are the use of multiple demes and the presence of a migration operator. Multiple-deme GAs are one of the most popular parallelisation methods, and many papers have been written describing details of their implementation [7]. This parallelisation method requires the division of a population into some number of demes (subpopulations). Demes are separated from one another (*geographic isolation*), and individuals compete only within a deme. An additional operator called *migration* is introduced: from time to time, some individuals are moved (copied) from one deme to another.

II. Dynamic Demes Model

The idea of parallelising GAs using dynamic demes (DDs) was first proposed in preliminary form in [8], and then developed and briefly evaluated in [9].

The main advantages of dynamic demes are:

- High scalability and flexibility (DDs can be used to implement a broad range of algorithms from coarse grained to highly fine grained models)
- Fault tolerance (some of the processors can crash, but the algorithm will correctly continue)
- Dynamic load balancing
- Easy monitoring

DDs are implemented in an object oriented library called MPGA developed in C++ with PVM. The library also contains other parallel GA models. The library is publicly available from `http://studentweb.cs.bham.ac.uk/~mxn/cirrus`. The DD algorithm is relatively simple. The population is divided into subpopulations (demes). Selection and mating are applied to the demes similarly to other parallelisation methods for GAs. However, in DDs the subpopulations are created dynamically after each processing cycle, and so the demes are not fixed. A more detailed description of the algorithm will be given in the following sections.

A. Features

Both the master-slave parallel GA and the static subpopulation GA suffer from some problems. DDs overcome these problems by combining the best features of these methods. DDs is a combination of global parallelism (the algorithm can work as a simple master-slave distributed GA) with a coarse-grained GA (overlapping subpopulations model). In DDs the population is treated as a collection of separated demes. There is no migration operator as such, but individuals are exchanged via a dynamic reorganisation of the demes at each processing cycle. The main reason for reorganising the demes is to cut down the waiting time for the last (slowest) individuals in the master-slave model. This happens because new demes are created and executed as soon as enough individuals have been evaluated.

From the parallel processing point of view the dynamic demes approach fits perfectly into the MIMD category (Flyn classification) as an asynchronous multiple master-slave algorithm.

The algorithm is fully scalable. Starting from global parallelism with fitness-processing distribution, one can scale up the algorithm up to a fine grained version, with a few individuals within each deme and a large number of demes i.e. thanks to its scalability it can be run efficiently in systems with several Processing Elements (PEs) as well as in massively parallel systems with large number of PEs. The algorithm can be run on shared- and distributed-memory parallel machines.

B. Algorithm Description

Each individual is represented by a separate process (which we call a *slave*), which is capable of performing the following:

1. Fitness evaluation
2. Applying mutation to itself (with a predefined mutation rate)
3. Doing crossover with another individual (this is done by passing to each individual the process ID of another individual with which it should perform crossover)

All the individuals run concurrently. The ideal case is when a single processing element processes a single individual. When this is not the case parallelism is simulated by the operating system or, like in our case, by a parallel virtual machine (thanks to the PVM library).

There are additional processes, called *masters*, which are responsible for selection and mating. Masters handle a fixed fraction of the population and apply selection and mating on it. Therefore, each master represents a separate deme. However, unlike other PGAs, as explained below, in DDs the individuals belonging to each deme change dynamically. The number of masters is a parameter of the algorithm. If there is only one master DDs is actually a classic distributed fitness evaluation algorithm.

Each master process performs selection and mating concurrently with the other master. Mating requires sending the appropriate slave ID to the individuals chosen for crossing over. When the slaves receive a partner ID they perform crossover, and then proceed with fitness evaluation and mutation.

In addition to masters and slaves there is also a process (possibly more) responsible for load balancing, called *counter*. After crossover, fitness evaluation and mutation each individual is dynamically assigned to a deme (possibly different from the one it belonged to previously). This happens when the individual notifies the counter process. The counter process knows which master processes are currently idle waiting for their subpopulation to be filled and it sends to the individual the process ID of one such master.

The last process within the system is called *sorter*. This process is informed by all of the individuals finishing their evaluation, takes their genotype and fitness, and saves them in

appropriate log files. The sorter process is also responsible for stopping the search when a termination criterion is met (e.g. when a solution of sufficient quality is found or when a fixed number of individuals has been processed).

Because the algorithm was developed in a heterogeneous multiuser environment,[1] normally there was no need for introducing a strategy to pass information between demes (like for example the idea of having partially overlapping demes). The differences in speed of the processing elements was enough for disturbing the regularities in the processing cycles, thus allowing the mixing of the individuals in different demes. Usually in our experiments with heterogeneous environments the algorithm was configured so that each master was in charge of a fraction of the population including

$$n_{ind} = \frac{N_{ind}}{n} \tag{1}$$

individuals, where N_{ind} is the total number of slaves (individuals) and n is the number of dynamic demes and also the number of master processes. Equation 1 ensures, that we do not have too many slaves waiting for available masters. We also do not need in general more master processes than n, where n is also equal to the number of demes.

For parallel systems with uniform PEs in problems where fitness evaluation and other operations are performed in constant time it may be necessary to allocate more slaves to each master than the number given by Equation 1 in order to avoid the case when the processing is completely synchronous and the algorithm works as a subpopulation model without migration. A simple solution is to allow demes to overlap. Therefore, we need add an *overlapping factor*, simply by adding a particular number of individuals $n_{overlap}$ to the value n_{ind}, and allow $n_{overlap}$ number of individuals in each processing cycle to be accessed by two master processes. This will create a ring architecture within our multiple-dynamic-deme model, where some of the individuals are part of two dynamically created demes.

III. Comparison to Other Parallel GA Models

Dynamic demes are quite similar to the asynchronous master-slave method of parallelising a genetic search. The most important difference is that in the master-slave case there is only one master and there are multiple slaves to perform computationally intensive tasks (like fitness evaluation). This model is best suited when fitness evaluation is the heaviest task. In the dynamic demes model, the presence of multiple demes allows the search to be easily scaled up from fine- to coarse grained, which means that the algorithm is as useful for long and time consuming fitness evaluations as it is for short and quick ones. By offering scalability and fault tolerance (in distributed fitness evaluation it is sufficient to stop one PE for the whole search process to hang, while in dynamic demes only one deme stops), the algorithm is much more flexible and powerful than simple distributed fitness evaluation.

[1] The very first version was developed and tested on a cluster of Linux, MS-Windows and Hewlett-Packard Unix workstations. Further research was conducted on DEC Alpha clusters.

It is possible to combine dynamic demes with a parallel steady-state algorithm, since both are managed by the same main principles. Then it would be possible to run steady-state version of dynamic demes efficiently on shared-memory multiprocessors.

IV. Theoretical Performance Prediction

The main obstacles in predicting the speedup of parallel genetic algorithms are the hardware and configuration differences. When using distributed systems it is also a problem to predict the load due to message-passing and communication costs (*parallel overhead*). Some attempts to define "ideal" cases, and on this basis to predict the speedup of parallel genetic algorithms, have been reported in the literature [10].

Because of the parallel overhead one cannot scale up parallel algorithms to infinity. For a given problem there is always a point, when the algorithm's speedup as a function of the number of processing elements (PEs) stops increasing and starts decreasing because the communication cost is bigger than the advantage of having more PEs available. In this section we want to estimate the optimal number of individuals for dynamic demes.

For simplicity in the following we will assume that the number of individuals equals the number of processing elements available for slave processes. Let us assume that:
$T_{counter}$ is the time required for processing the whole population (all individuals) by the counter process (counter cycle)
T_{master} is the processing time for one master cycle (i.e. building one deme and applying selection and mating to it)
t_{ind} is the time required for one full cycle of a slave process (i.e. receiving a genotype, applying crossover, mutation and fitness evaluation, and waiting for another genotype). Notice that $t_{ind} = t_{sind} + t_{communication}$, where t_{sind} is the processing time required by crossover, mutation and fitness evaluation of one individual in a sequential GA.
t_{send} is the time needed for sending a single message.

For overlapping demes Equation 1 needs to be modified, becoming:

$$n_{ind} > \frac{N_{ind}}{n} \tag{2}$$

where n is the number of demes, N_{ind} is the total number of individuals, and n_{ind} is the number of individuals per deme. In the following we will assume an ideal case in which some of the processing times are negligible (e.g. for loops, jumps or if-like statements). So, for the counter the total time for processing the whole population is:

$$T_{counter} = N_{ind} \cdot t_{send} \tag{3}$$

and for the master the total time for processing one deme is:

$$T_{master} = n_{ind} \cdot t_{send} + t_{selection} \tag{4}$$

where $t_{selection}$ is the time required to apply selection to the entire deme.

The time required by a slave cycle (from one fitness evaluation to another fitness evaluation) is the result of applying sequentially the GA operators, plus performing the communication with the counter, the master and another individual during crossover:

$$t_{ind} = t_{mutation} + t_{crossover} + t_{fitness} + 3 * t_{send} \quad (5)$$

where $t_{mutation}$ is the time taken by mutation on a single individual, $t_{crossover}$ is the time taken by crossover on a single individual, and $t_{fitness}$ is the time required by the fitness evaluation for a single individual. If master processing is relatively quick, then:

$$t_{mutation} + t_{crossover} + t_{fitness} \gg t_{selection} \quad (6)$$

If there is a single counter[2] in order for the algorithm to work properly we have to keep:

$$T_{counter} \leq t_{ind} \quad (7)$$

i.e. using Equation 3 and assuming that individuals do not have to wait for the master process:

$$\mathbf{N_{ind}} \leq \frac{\mathbf{t_{ind}}}{\mathbf{t_{send}}} \quad (8)$$

which, knowing the computational cost of the genetic operators and of the fitness evaluation, allows us to estimate the optimal number of individuals (i.e. of PEs) to be used for a given problem. From this equation we can see that if the communication overhead is high, a smaller number of individuals is better, as there is less communication between them. It should be noted that the population size achieving best performance in terms of processing time is not necessarily a good population size from the GA point of view, i.e. the quality of the solutions reached in "optimal" time might be poor because the population is too small. The ideal case is when the communication time is very small, and processing one individual lasts long, and thus increasing the number of PEs increases the speedup.

If dynamic demes are simulated on a cluster of workstations, the theoretical estimation of efficiency is difficult, because t_{send} which is used in Equation 8, depends on hardware, network speed, machine load and network load. Performance simulations are needed to determine the best configuration for the given problem class.

In general we can say that for GAs with very simple GA operators and fitness evaluations, the dynamic demes approach does not behave as well as for computationally intensive GAs.

A. Predicting the Speedup

It is always difficult to predict the speedup of a given parallel GA, because of many different factors which can influence the behaviour of the algorithm. In the dynamic demes model, let us assume that we are always keeping the optimal parallelisation model (i.e. upper bound in Equation 8). If we define:

[2] When more than one counter is used, the algorithm becomes a hybrid method.

$t_{sind} = t_{mutation} + t_{crossover} + t_{fitness}$
T_{serial} = clock time necessary to perform one generation of a sequential GA
$T_{parallel}$ = clock time necessary to perform one generation of dynamic demes GA

then:

$$T_{serial} = N_{ind} * t_{sind} + t_{selection} \quad (9)$$

and for the parallel implementation (assuming that all of the individuals are processed in parallel, concurrently, and that selection is done only for a fraction of population also concurrently) we have:

$$n = \frac{N_{ind}}{n_{ind}} \quad (10)$$

and

$$T_{parallel} = t_{sind} + \frac{1}{n} * t_{selection} + t_{communication} \quad (11)$$

where, from Equations 3 – 5, we define:

$$t_{communication} = \underbrace{3 * t_{send}}_{slave} + \underbrace{N_{ind} * t_{send}}_{counter} + \underbrace{n_{ind} * t_{send}}_{master} \quad (12)$$

The speedup of paint with Dynamic Demes is:

$$speedup = \frac{T_{serial}}{T_{parallel}} \quad (13)$$

hence from Equations 9, 11, 12 we have after simple transformation:

$$speedup = (N_{ind} * t_{sind} + t_{selection}) / (t_{sind} + \frac{1}{n} * t_{selection} + 3 * t_{send} + N_{ind} * t_{send} + \frac{N_{ind}}{n} * t_{send}) \quad (14)$$

Because of Equation 8 we can simplify the above equation, assuming that $N_{ind} * t_{send}$ cannot be bigger than t_{sind}, obtaining:

$$speedup \geq (N_{ind} * t_{sind} + t_{selection}) / (t_{sind} + \frac{1}{n} * t_{selection} + 3 * t_{send} + t_{sind} + \frac{1}{n} * t_{sind}) \quad (15)$$

and finally:

$$speedup \geq \frac{n * N_{ind} * t_{sind} + n * t_{selection}}{(2n+1) * t_{sind} + 3 * n * t_{send} + t_{selection}} \quad (16)$$

In some cases we can omit the constant value t_{send}, e.g. if $t_{sind} \gg t_{send}$, (which can be interpreted as a slightly longer initialisation of the parallel algorithm) obtaining:

$$speedup \gtrsim \frac{n * N_{ind} * t_{sind} + n * t_{selection}}{(2n+1) * t_{sind} + t_{selection}} \quad (17)$$

The value of the parameter n is fixed and constant, and it is between 1 and N_{ind}. When n is 1, we have a synchronous master-slave version of the algorithm. A linear speedup for this should be expected. For $n > 1$ we can gain even more, because we run selection in parallel.

In heterogeneous environments where not all PEs are working exactly with the same speed or when $t_{fitness}$ is not constant, we may have a bottle-neck effect, since we may have to wait for the slowest of PEs to finish. Indeed for $n = 1$ we have to wait for the slowest PE every single generation. However, for $n > 1$ there is no waiting for slow PEs, as they can join the algorithm at any point in time.

Equation 16 means that in the ideal case, when the communication costs are negligible, the algorithm provides a linear speedup.

The above theoretical analysis assures that the DD algorithm works at least as efficiently as an synchronous master-slave algorithm. This was confirmed by our experimental results.

V. Experimental Results

A. Sequential Optimisation

A series of experiments were conducted on simple sequential optimisation problems using of MPGA and the dynamic demes model [9].

One of the problems was to order a sequence of characters in a string to form a fixed pattern. This was a very simple problem for GAs, and thus we used only 8 individuals for solving the problem. The length of the string was fixed to 8, and in each position 35 different ASCII characters could be present. So the total search space size was 35^8. The traditional single machine based GA could only achieve a processing speed of around 64 individuals per second. With five machines and a utilisation of about 5% on each, the dynamic demes algorithm could process 130 individuals per second. The optimal configuration with 14 machines processed 165 individuals per second. In contrast, a single population based algorithm with distributed fitness evaluation on the same 14 machines was capable of achieving only 130 individuals per second[3]. The speedup achieved in this experiments was relatively poor ($\frac{165}{64} \approx 2.57$) due to the high communication costs and the quick fitness evaluation and operators (single send/receive message routine lasts about 5 times longer than fitness evaluation).

Additional tests were run with more computationally intensive fitness evaluations. In the previous case, fitness evaluation itself was about 0.03sec. In these experiments, fitness evaluation took 100 times longer (3sec). In a sequencing algorithm it would be possible to achieve 0.33 individuals per second at most. With distributed fitness evaluation on 14 machines a speed of 2.63 individuals per second was reached, giving a speedup factor of 7.9. With the DD approach also on 14 machines the results were: with 2 demes, 2.83 individuals per second (8.5 speedup); with 4 demes, 3.53 individuals per second, which means a **10.6 speedup** .

[3] The tests were done on the network of 233MHz DEC Alpha workstations in the School of Computer Science (6 with 160MB RAM, 8 with 64 MB RAM).

B. Cluster Geometry Optimisation

We tested DD also on a class of real-world problems where the speedup provided by parallelisation is really important. One of these problems (cluster geometry optimisation) is to find the optimal coordinates of the atoms in a cluster, to form a cluster with minimal energy. The energy potential is calculated by using an energy potential function, in our case the Morse Potential. The Morse potential is a model for the interaction only for a pair of atoms. While it is based on a simple harmonic model, the Morse potential improves on this model by allowing for dissociation of the atomic pair [11]. For atoms i and j, the potential function is:

$$V_{ij} = D_e \cdot \left(e^{\alpha(1-\frac{r_{ij}}{r_e})}\left\{e^{\alpha(1-\frac{r_{ij}}{r_e})} - 2\right\}\right) \qquad (18)$$

where D_e and r_e are appropriate constants, and r_{ij} is the distance between the atom i and the atom j. The parameter α is used for simulating different slopes of the energy function. With big α the cost of local optimisation of the cluster can be very high.

The fitness function was a linear combination of V_{ij} for each pair of atoms. The cluster geometries represented by each individual in GA were further optimised using a local relaxation algorithm (the Broyden-Fletcher-Goldfarb-Shanno algorithm called BFGS) [12]. This method requires $O(N^2)$ storage, and is based on the Conjugate Gradient Method and one-dimensional line minimisation[13]. Because of the local relaxation, processing of the single individual lasts a significant time for big clusters.

For consistency the speed was measured on the same hardware platform[4], for the same objective function and GA operators. Everything apart from the way parallelisation was achieved was common for different methods. Because the problem itself was harder than the sequential optimisation ones tested before, we used larger populations, including 50-200 individuals, but we still had at most 40 PEs.

In normal conditions for the geometry optimisation of a cluster of 30 atoms, performing 100 iterations with he50 individuals in the population and BFGS minimisation, the master-slave model could achieve 7.9 individuals per second with 23 DEC Alphas connected into single Virtual Parallel Machine via PVM. In the same environment dynamic demes achieved 14.3 individuals per second using 2 dynamic demes (using more demes even more than 14.6 individuals per second could be processed). With fewer PEs DDs performed even better than this, achieving 5 individuals per second on 8 machines, while master-slave methods achieved only about 1 individual per second in this configuration. Dynamic demes performs much better than the master-slave method because it can benefit from asynchronous processing. In optimal parallel and hardware configurations dynamic demes performs still better than synchronous master-slave, but perhaps less markedly.

[4] We used a cluster of 20 DEC Alpha workstations in the IT lab of Department of Chemistry and 20 DEC Alphas in School of Computer Science.

VI. Conclusions

In this paper we have presented a new method to parallelise genetic algorithms, called dynamic demes. The method is based on the idea of constantly reorganising a set of subpopulations (demes) dynamically so as to avoid bottlenecks due to slow processors or fitness functions requiring a variable computation effort.

In the experiments reported the method has shown very promising speedups, which compare very favourably with those achievable using other parallelisation methods. Unlike other parallelisation methods the dynamic deme algorithm can be very efficient both when used as a fine grained parallel algorithm and when used as a coarse grained algorithm. These results are confirmed by a theoretical analysis which is also presented in this paper.

Like other parallelisation methods, the search performed by the dynamic demes algorithm is different from that of a sequential GA. As a consequence, it is difficult to know for which class of problems this new parallelisation methods is best suited. This should be the topic of future investigations.

Acknowledgements

The authors wish to thank the members of the Evolutionary and Emergent Behaviour Intelligence and Computation (EEBIC) group at Birmingham.

References

[1] John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, USA, 1992.

[2] Sean Luke, "Genetic programming produced competitive soccer softbot teams for robocup97," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, Eds., University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998, pp. 214–222, Morgan Kaufmann.

[3] Reiko Tanese, "Parallel genetic algorithms for a hypercube," in *Proceedings of the Second International Conference on Genetic Algorithms*, John J. Grefenstette, Ed. 1987, Lawrence Erlbaum Associates, Publishers.

[4] T. C. Fogarty and R. Huang, "Implementing the genetic algorithm on transputer based parallel processing systems," in *Parallel Problem Solving from Nature*, Berlin, Germany, 1991, pp. 145–149, Springer Verlag.

[5] R. Tanese, *Distributed Genetic Algorithms for Function Optimization*, Ph.D. thesis, University of Michigan, 1989, Computer Science and Engineering.

[6] David E. Goldberg, "Sizing populations for serial and parallel genetic algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms*, J. D. Schaffer, Ed., San Mateo, CA, 1989, Morgan Kaufman.

[7] Erick Cantú-Paz, "A survey of parallel genetic algorithms," IllGAL Report 97003, The University of Illinois, 1997, Available on-line at: `ftp://ftp-illigal.ge.uiuc.edu/pub/papers/IlliGALs/97003.ps.Z`.

[8] Halina Kwaśnicka and Mariusz Nowostawski, "Search engine for information systems based on parallel genetic algorithm," in *Proceedings of the 2nd International Conference on Parallel Processing & Applied Mathematics*, R. Wyrzykowski, H. Piech, and J. Szopa, Eds., Częstochowa, Poland, 1997, vol. 2, pp. 442–451, Institute of Mathematics and Computer Science, Technical University of Częstochowa.

[9] Mariusz Nowostawski, "Parallel genetic algorithm for sequencing optimisation," Second miniproject report during MSc in Advanced Computer Science 1997/98 course, The University of Birmingham, School of Computer Science. Available online at: `http://studentweb.cs.bham.ac.uk/~mxn/gzipped/rep2r.ps.gz`, May 1998.

[10] Erick Cantú-Paz and David E. Goldberg, "Predicting speedups of ideal bounding cases of parallel genetic algorithms," in *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, Thomas Bäck, Ed., San Francisco, CA, 1997, Morgan Kaufmann.

[11] Jonathan P.K. Doye and David J. Wales, "The effect of the range of the potential on the structure and stability of simple liquids: from clusters to bulk, from sodium to c_60," *J. Physics B*, , no. 49, pp. 4859–4894, 1996.

[12] Mariusz Nowostawski, "Parallel genetic algorithms in geometry atomic cluster optimisation and other applications," M.S. thesis, School of Computer Science, The University of Birmingham, UK, September 1998, `http://studentweb.cs.bham.ac.uk/~mxn/gzipped/mpga-v0.1.ps.gz`.

[13] D.A.H. Jacobs, Ed., *The State of the Art in Numerical Analysis*, chapter III.1, §§3-6, London: Academic Press, 1977, Written by K.W. Brodlie.