PARALLEL IMPLEMENTATION OF MULTIOBJECTIVE GENETIC ALGORITHMS

Carlos M. Fonseca
A.D.E.E.C. – U.C.E.H.
Universidade do Algarve
Campus de Gambelas
8000 FARO
Portugal

Peter J. Fleming
Dept. A.C.S.E.
University of Sheffield
Mappin Street
Sheffield S1 3JD
U.K.

ABSTRACT

In earlier work, Genetic Algorithms (GAs) have been extended so as to handle multiple objectives and constraints in a unified manner. The resulting Multiobjective Genetic Algorithms (MOGA) have been applied to problems such as controller design, systems identification and production scheduling.

This paper reports on the parallelization of an existing Matlab MOGA implementation via the PVM (Parallel Virtual Machine) message-passing software, and on the application of the resulting parallel MOGA strategy to a gas turbine engine controller parameter tuning problem studied previously. Results concerning both execution speed and the quality of the solutions produced by a number of runs of the algorithm are presented.

1 INTRODUCTION

In earlier work [1], a unified means of handling multiple objectives and constraints in Genetic Algorithms (GAs) was developed. The resulting Multiobjective Genetic Algorithm (MOGA) was then applied to the optimization of the low-pressure spool-speed governor of a (non-linear) Pegasus gas turbine engine model [2], and shown to be useful, not only as an optimizer, but also as an interactive decision-making tool. Similar multiobjective genetic algorithms have now been successfully applied to non-linear system identification [3], multiobjective production scheduling of chilled ready meals [4], multiobjective optimization of distributed magnetic bearing controllers [5], and integrated gas turbine engine controller design [6], among other problems.

Unfortunately, the objective functions associated with the problems above can be computationally demanding, leading to long optimization runs and making user interaction with the MOGA impractical. As is the case for single-objective GAs, distributing the MOGA over an available network of workstations should provide a convenient, efficient, and cost-effective means of improving its response time. In the present case, this has been implemented as an interface between the Matlab numerical package and the PVM message-passing software, which allows function evaluations to be distributed over the main host plus a number of remote workstations on a farmer/worker topology, while retaining the ability to run the MOGA and, if so desired, the function evaluations, under Matlab.

The Matlab-PVM interface is described after an introduction to Multiobjective Genetic Algorithms and a discussion of some of the issues raised by their parallelisation. Experimental results based on the Pegasus controller design problem mentioned above are then presented, concentrating both on the speed-up achieved and on the quality of the solutions produced by 9 runs of the algorithm. In particular, the set of tightest design goals attained simultaneously by all runs is computed and presented along with the overall best designs obtained, providing a quantitative characterization of the performance of the algorithm on this problem. The paper concludes with a discussion of the results and perspectives for further work.

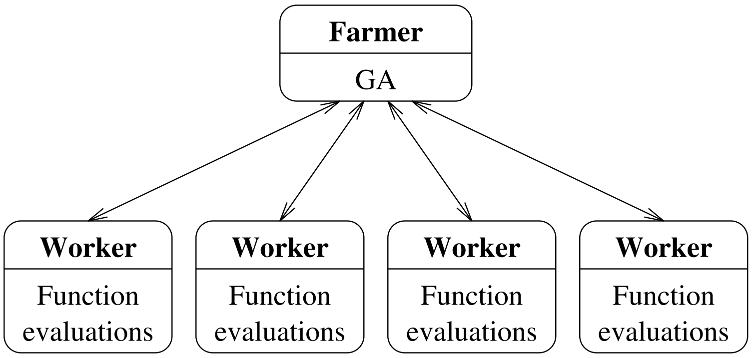

Figure 1: Farmer-worker implementation of the GA.

2 MULTIOBJECTIVE GENETIC ALGORITHMS

The development of genetic algorithms for multiobjective optimization has received growing attention in the literature (see [7] for a review). Current multiobjective genetic algorithm (MOGA) approaches rank the GA population according to the definition of Pareto-optimality, respecting the fact that objectives may not be commensurable. As a result, a *family* of non-dominated solutions (candidates) can be evolved concurrently in a single algorithm run.

The formulation of the MOGA proposed in [1] introduces the concept of *preferability given a prioritized goal vector* and enables the engineer to take part in the search process by interactively communicating preferences to the algorithm via a graphical user interface. Goals and priorities for the various objectives may usually be set based on problem specifications. Goals may also be initially set to either Utopian values (e.g., zero or $-\infty$) or "trivial" values (e.g., $+\infty$), and then refined as the search proceeds.

Unfortunately, MOGAs (and GAs in general) usually require many function evaluations to be performed, which, for computationally demanding problems, makes their interactive use less attractive. This speed concern can be addressed by resorting to parallel implemention.

3 PARALLEL GENETIC ALGORITHMS

It has been noted many times before that genetic algorithms are particularly easy to parallelize, since the computational load associated with them is almost always concentrated in the form of evaluations of the objective function(s) characterizing the problem, *and* many of these calls are largely independent from one another. Thus, rather than attempting to parallelise the objective function(s), one may simply distribute the various objective function evaluations by multiple processors. Additionally, a particular GA can often be modified to better reflect the parallel architecture it is to be implemented on, sometimes resulting in novel GA models with their own sets of properties. Incremental replacement [8] and structured populations [9] are good examples of models which are favoured in the context of parallel genetic algorithms.

3.1 The Farmer/Worker Approach

Incremental replacement, as opposed to generational replacement, consists of generating a small number of offspring at a time (typically one or two), and reintroducing them in the population immediately after evaluation, so that they may take part in the next selection step. If one initially generates as many offspring as the number of processors available, and then allows a new offspring to be generated whenever an evaluation completes, the GA can be run asynchronously in a farmer/worker configuration as depicted in Figure 1. The workers can thus be kept busy almost all the time, except for the time it takes to communicate the data involved and to generate a new offspring. This can be further improved by generating the new offspring *before*, as opposed to *after*, the next evaluation completes.

The above model, while convenient, does suffer from a bias towards solutions which can be evaluated quicker than their competitors (say, they can be ruled out as invalid early in the evaluation process). Such

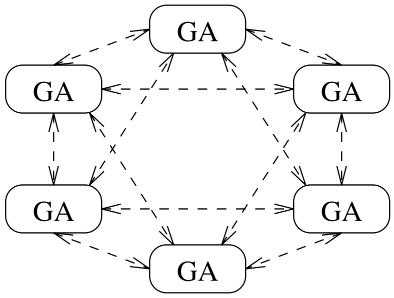

Figure 2: The island model.

solutions return to the population quicker than others, and so are given an unfair reproductive advantage. In some cases, a replacement strategy based on fitness may attenuate this effect, but not always.

Another potential disadvantage of incremental GAs over generational GAs is the increased possibility of occurrence of stochastic errors at the selection stage. In the generational case, these can be minimized via Baker's Stochastic Universal Sampling (SUS) algorithm [10], but the same cannot be said for the purely incremental case, where SUS reduces to simple roulette-wheel selection. Thus, incremental GAs often implement random selection and replacement-of-the-worst, which results in elitism and thus imposes a high selective pressure.

Despite more common practice, there is no reason why a quasi-generational GA cannot be implemented asynchronously. This can be done by placing an excess of p individuals in the initial population, where p is the number of worker processors available. As long as a new population can be generated while the last p individuals are evaluated, the workers will always be kept busy. This approach is proposed here as a means of simultaneously reducing bias towards individuals easy to evaluate and retaining the ability to use lower selective pressures and SUS, with the parallelisation of existing generational MOGAs in mind.

3.2 The Island Model

The farmer/worker approach requires processors to exchange two sets of data per individual evaluated, namely the chromosome and the result of the evaluation. In order to cut down inter-processor communication further, individuals would have to be generated on the processor on which they are to be evaluated.

This is achieved in the parallel *island model* by running several GAs independently on different processors. These GAs periodically exchange small fractions of their populations, implementing *migration* between the geographically isolated populations (see Figure 2).

When compared to single-population approaches, not only does the island model cut down communication time, but it is also reported to exhibit better global search behaviour. In the multiobjective case, this parallelisation model does not raise any special issues, with the possible exception of the choice of which individuals to actually migrate.

The farmer/worker approach also assumes that the farmer can communicate efficiently with all the workers. This may be true for a networked workstation cluster, but other architectures may impose restrictions on which processors can communicate with each other. In that case, migration may be implemented between directly-connected nodes only.

3.3 The Diffusion Model

In the *diffusion model*, the population is distributed over a grid (or a ring, or some other topology), and each individual interacts only with its direct neighbours (see Figure 3). This fine-grained approach is especially targeted at massively-parallel architectures, although some of its properties justify their sequential implementation as well. Rowe [11] has investigated multiobjective diffusion GAs for the natural niching effects they exhibit and noted that the resulting algorithm, when applied to a sensitivity analysis problem, produced better and more diverse trade-off sets than a panmictic MOGA with sharing. Apparently, no mating restriction was used in combination with sharing, which may explain the poorer performance of the panmictic MOGA (see [12] for an example of this).

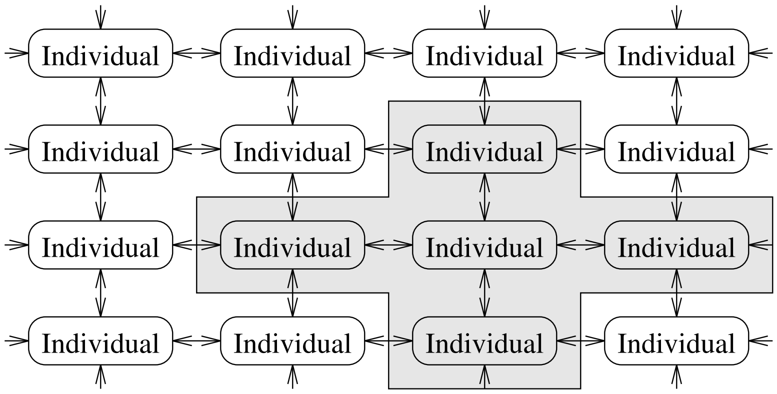

Figure 3: The diffusion model.

3.4 Trade-offs for Multiobjective GAs

Genetic algorithms are often run in batch mode, i.e., until a given stopping criterion is satisfied. In the case of a distributed GA, this allows different processors to maintain progress statistics locally and only report them at the end of the run. On the other hand, if continuous, centralized monitoring of the progress of the GA is required, e.g., for visualisation and interaction purposes, the need for efficient communication between each processor and the one in charge of the monitoring arises.

In the single-objective case, reporting a single real value describing the performance of the best individual seen by each processor may provide an acceptable level of monitoring for some purposes. In the multiobjective case, however, it may be necessary to communicate much larger amounts of data, since each solution will be characterized by a number of performance measures, and there will not generally be a single best solution. Therefore, while it is possible to decentralize the GA and cut on communication by structuring and distributing the population, requiring also centralized progress monitoring in a MOGA significantly undermines this effort.

A farmer/worker approach already implements all the necessary communication for effective progress monitoring, but it does require the farmer to take on board the additional load associated with the monitoring itself. In the context of a workstation cluster, it is legitimate to expect this to be acceptable.

4 PARALLEL MOGA IMPLEMENTATION

In this section, the parallelisation of an existing MOGA implementation is described. The target platform is a network of heterogeneous, general-purpose SUN workstations, with varied performance characteristics and running under different operating systems (SunOS 4.1.x and Solaris 2.x). Parallel operation is achieved by interfacing the PVM message-passing libraries [13] with the Matlab numerical package [14], under which the MOGA is run. An overview of the system is provided in Figure 4.

4.1 Parallel Virtual Machine

Parallel Virtual Machine (PVM) is a software system which enables a set of heterogeneous computers to be used as a coherent parallel computing resource. The software is available on the Internet under a free license. The computers may be interconnected by a variety of networks, including ethernet. PVM applications may be written in C, C++, and Fortran, and access the system through library calls.

4.2 Interfacing with Matlab

User programs written in C or Fortran may communicate with Matlab in one of two ways. By means of dynamic linking, custom C or Fortran code may be called from within Matlab just like any built-in or interpreted function, allowing users to implement compiled-language versions of their interpreted scripts for

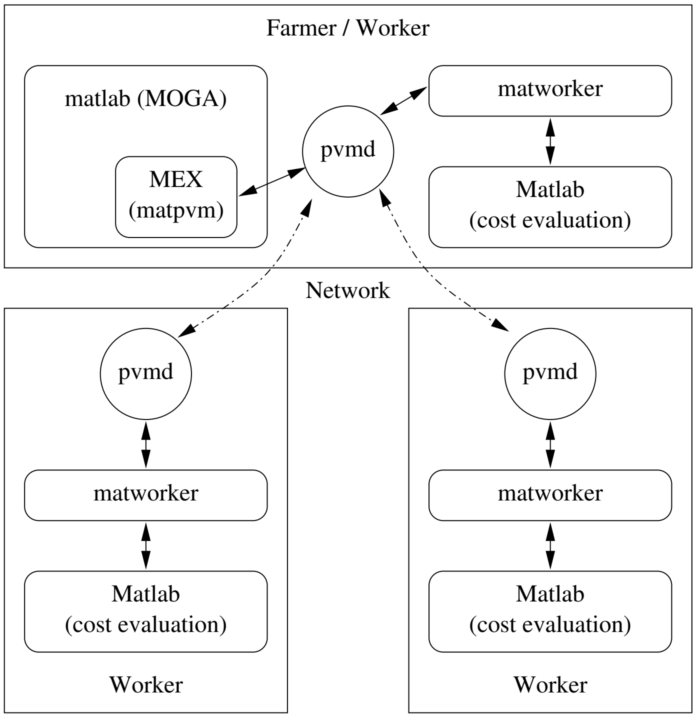

Figure 4: Overview of the system.

efficiency and/or speed, access data acquisition hardware, etc. Such code, once compiled, is known as a Matlab EXecutable file, or MEX-file, for short. This is how the MOGA farmer, which consists mostly of Matlab code, accesses PVM in order to communicate with the workers.

Alternatively, C or Fortran code may start a Matlab computation engine as a separate process and access it through library calls. For the purpose of this study, a dummy worker was implemented in C, which starts up a Matlab engine once, and then simply interfaces between the engine and the farmer. The two processes would normally communicate via pipes, although a memory leak in the Matlab Engine libraries had to be worked around by using files to implement part of the communications. This setup was chosen in order to allow the same objective functions (already implemented in Matlab and SIMULINK) to be used directly, and for the purpose of comparison with a Matlab only implementation. In a production environment, the objective functions should be implemented purely as compiled code, both for speed and for economy, as each engine instance consumes one Matlab license!

4.3 The `matpvm` Gateway

The interface to the PVM libraries was encapsulated in a single MEX function, `matpvm`. This was required due to PVM making use of global C variables, which are not shared between different MEX-functions. `matpvm` allows one to spawn workers, both local and remote, from the Matlab command line or Matlab scripts or functions, as well as handling all communication with them.

When used to spawn new workers, the following syntax is used:

```
nw = matpvm('spawn','<command>');
```

where `<command>` represents the PVM-aware executable to run on the worker processors. Provided no errors occur, the number of workers spawned, `nw`, will be the same as the number of nodes which compose the virtual machine. Once the workers have started, they send a message to the farmer stating that they are `READY`.

Communication between the farmer and the workers is handled by the command:

```
Result = matpvm('recsend', Data);
```

where each row of the Matlab matrix `Data` contains a work request to be sent to the first worker that is ready. `Result` is a matrix containing the workers' responses.

The first time `recsend` is passed to `matpvm`, a work request from the `Data` matrix will be issued to the corresponding worker for each `READY` message received. After that, the farmer waits for any worker to send a response and issues the next request to it. The function returns as soon as all requests have been *issued*. The master then proceeds with the main MOGA loop, while the workers process the last `nw` requests. The corresponding results are collected the next time `matpvm` is called with `'recsend'` as its first argument, in place of the `READY` messages mentioned above.

The command

```
clear matpvm
```

terminates the workers and checks out of the PVM system before unloading `matpvm` from the Matlab workspace.

4.4 The Parallel MOGA

In the context of the MOGA, the work requests passed to `matpvm` consist simply of the individual chromosome representations, which the workers will decode and subsequently evaluate. The results returned by the workers consist of the concatenation of the chromosomes, the decision variables, and the objective function values which characterize each individual. This generic implementation allows chromosomes to be "repaired" during the evaluation process, if needed, and makes it very simple to update the population at each generation. Arguably, if the chromosomes are never repaired, only genuinely new information needs to be communicated back by the workers. However, this would require individuals to be cached on the farmer side, and was not implemented for simplicity.

5 EXPERIMENTAL RESULTS

An existing MOGA implementation in Matlab was parallelised using the `matpvm` gateway described earlier, and applied to the gas turbine engine controller tuning problem described in [2]. The problem involves 8 controller parameters, and 7 performance objectives:

1. Closed-loop stability
2. Gain margin
3. Phase margin
4. Rise time
5. Settling time
6. Overshoot
7. Output error

All objectives were implemented as functions to be minimized, and goals were set based on domain-specific knowledge.

The MOGA was run on 4 Sun workstations connected to the ACSE departmental network (10Base2 ethernet): one Sparc2 running Solaris 2, and 3 identical Sparc 5 running SunOS 4, as detailed in Table 1. The relative performance of the workers with respect to the farmer was estimated by evaluating the final population of a 100-generation run on the farmer and on one of the slaves, *locally* (i.e., not using PVM).

5.1 Execution Speed

Before the speed-up achieved by running the MOGA over PVM on the given network can be measured, it must be noted that, in this case, the time to evaluate one solution depends on the solution itself. For example, an evaluation will complete as soon as the controller is found not to stabilize the engine, so non-stabilizing controllers will evaluate faster than stabilizing ones. Therefore, it was chosen to measure the time taken to

Table 1: Network node description.

Node name	Machine class	Operating system	Relative performance	Rôle
serin	Sparc 2	Solaris 2.4	1.0	farmer/worker
stork	Sparc 5	SunOS 4.1	0.625	worker
skylark	Sparc 5	SunOS 4.1	0.625	worker
jay	Sparc 5	SunOS 4.1	0.625	worker

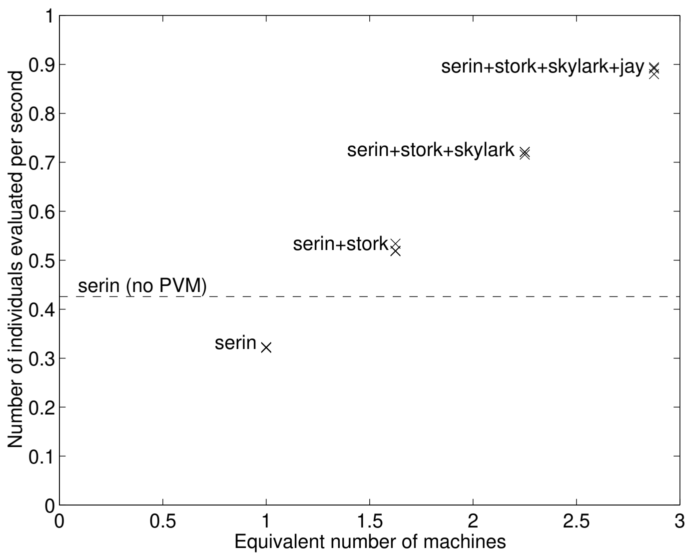

Figure 5: Population evaluation speeds on various configurations.

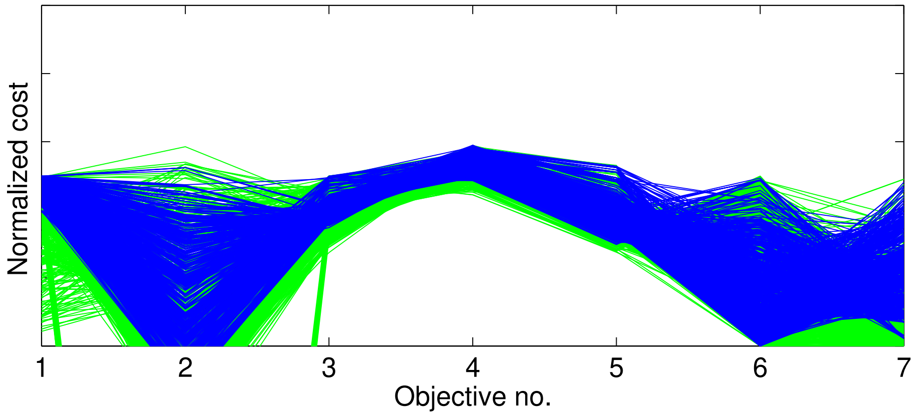

Figure 6: Best and 'worst' of 9 parallel MOGA runs.

evaluate a the final population of a given GA run on an increasing number of machines. Figure 5 shows observed evaluation speeds on one to four-machine configurations (three runs on each).

The most striking result is perhaps the significant overhead introduced by the communication path between the two Matlab sessions on the single machine configuration. This overhead was certainly aggravated by the use of files to communicate between the `matworker` and the worker Matlab processes. On the other hand, if this overhead is charged to the individual evaluation itself, approximately linear speed up can be observed.

As for the overall MOGA speed, it became clear that the monitoring effort imposed on the farmer cannot be ignored as the number of non-dominated individuals found increases. In fact, a point was reached where all workers were forced to wait until the farmer could deliver the new population for evaluation. This raises the need for improved algorithms to keep track of all non-dominated individuals produced during the course of a run. Alternatively, monitoring may either be postponed until expressly required, or be farmed out to a dedicated worker, or set of workers.

5.2 Solution Quality

Nine runs of the algorithm were performed, for 100 generations each, and the preferable solutions found in each run were recorded. These data were then post-processed in order to determine:

1. the set of preferable solutions found in all runs. This is the closest approximation to the real trade-off surface gained as a result of the 9 runs.

2. the set of all 'tightest' goal vectors which could be satisfied by every run of the algorithm. As explained in [15], this set describes the worst-case behaviour of the algorithm over the 9 runs.

In other words, each and every solution found in any of the 9 runs is dominated by or equal to at least one point in the first set, and dominates or is equal to at least one in the second set. As such, plotting one set over the other serves to illustrate how much the quality of the solutions found varied across the 9 runs. This is depicted in Figure 6, using the parallel-coordinate approach.

The importance of Figure 6 should not be underestimated. This plot *quantifies* the amount of variation in solution quality across the 9 parallel MOGA runs, in the same way the best and worst solution of a number of runs quantify variation in the single-objective case. In addition, it should be noted that the computation of the second set is not trivial, having required approximately 20 hours of CPU time on an Intel Pentium Pro processor running at 200 MHz. The development of more efficient algorithms for this and other related tasks is currently being persued by the authors.

6 CONCLUDING REMARKS

In this work, an existing Matlab implementation of a Multiobjective Genetic Algorithm was parallelized and run on a heterogeneous network of four workstations. Although approximately linear speed-up was observed when evaluating a single population, the communication path between the farmer and worker processes was shown to introduce considerable overhead when compared to evaluating the same population directly in a single Matlab process. In addition, the monitoring effort implemented by the MOGA at the farmer level gradually became an obstacle to the efficient utilization of the workers as the number of non-dominated solutions found increased. Better algorithms for handling non-dominated points in d-dimensional space are therefore needed, so that the monitoring overhead can be reduced.

The results of 9 parallel MOGA runs were analysed, and best and worst-case results were computed so as to quantitatively describe the quality of the solutions produced. These results show how the parallel MOGA behaved consistently across different runs, despite its stochastic nature.

Since the parallel MOGA implementation described here is not strictly equivalent to its conventional counterpart, methods of comparing MOGA performance are needed. Only then can the impact of algorithmic modifications to suit parallel implementation be assessed. This is currently an active area of research.

REFERENCES

[1] C. M. Fonseca and P. J. Fleming, "Multiobjective optimization and multiple constraint handling with evolutionary algorithms I: A unified formulation," *IEEE Transactions on Systems, Man and Cybernetics–Part A: Systems and Humans*, vol. 28, no. 1, pp. 26–37, 1998.

[2] C. M. Fonseca and P. J. Fleming, "Multiobjective optimization and multiple constraint handling with evolutionary algorithms II: Application example," *IEEE Transactions on Systems, Man and Cybernetics–Part A: Systems and Humans*, vol. 28, no. 1, pp. 38–47, 1998.

[3] C. M. Fonseca and P. J. Fleming, "Non-linear system identification with multiobjective genetic algorithms," in *Proceedings of the 13th IFAC World Congress*, vol. C, (San Francisco, California), pp. 187–192, 1996.

[4] J. Shaw and P. J. Fleming, "Including real-life problem preferences in genetic algorithms to enhance optimization of production schedules," in GALESIA [16], pp. 239–244.

[5] P. Schroder, A. J. Chipperfield, and P. J. Fleming, "Multi-objective optimisation of distributed active magnetic bearing controllers," in GALESIA [16], pp. 13–18.

[6] A. J. Chipperfield and P. J. Fleming, "More integrated gas turbine engine controller design," in GALESIA [16], pp. 239–244.

[7] C. M. Fonseca and P. J. Fleming, "An overview of evolutionary algorithms in multiobjective optimization," *Evolutionary Computation*, vol. 3, pp. 1–16, Spring 1995.

[8] G. Syswerda, "A study of reproduction in generational and steady state genetic algorithms," in *Foundations of Genetic Algorithms* (G. J. E. Rawlins, ed.), part 2, pp. 94–101, San Mateo, CA: Morgan Kaufmann, 1991.

[9] Y. Davidor, "A naturally occurring niche and species phenomenon: The model and first results," in *Genetic Algorithms: Proceedings of the Fourth International Conference* (R. K. Belew and L. B. Booker, eds.), pp. 257–263, San Mateo, California: Morgan Kaufmann, 1991.

[10] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," in *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (J. J. Grefenstette, ed.), pp. 14–21, Lawrence Erlbaum, 1987.

[11] J. Rowe, K. Vinsen, and N. Marvin, "Parallel genetic algorithms for multiobjective functions," in *Proceedings of the 2nd Nordic Workshop on Genetic Algorithms*, (Vaasa, Finland), pp. 61–70, 1996.

[12] C. M. Fonseca, *Multiobjective Genetic Algorithms with Application to Control Engineering Problems*. PhD thesis, University of Sheffield, 1995.

[13] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek, "The PVM concurrent computing system: Evolution, experiences, and trends," *Journal of Parallel Computing*, 1974.

[14] The MathWorks, Inc., MATLAB *Reference Guide*, Aug. 1992.

[15] C. M. Fonseca and P. J. Fleming, "On the performance assessment and comparison of stochastic multiobjective optimizers," in *Parallel Problem Solving from Nature – PPSN IV* (H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, eds.), vol. 1141 of *Lecture Notes in Computer Science*, pp. 584–593, Berlin, Germany: Springer, 1996.

[16] *Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, (Glasgow, UK), The Institution of Electrical Engineers, 1997. Conference Publication No. 446.