Evolutionary Algorithms for the Optimization of Simulation Models Using PVM

Thomas Bäck, Thomas Beielstein, Boris Naujoks
Informatik Centrum Dortmund
Joseph-von-Fraunhofer-Straße 20
D–44227 Dortmund

Jochen Heistermann
ZFE T SN 44
Siemens AG
D–81730 München

June 14, 1995

Abstract

Simulation models usually share some specific characteristics that make the automatic optimization of their input parameters an extremely difficult task. Evolutionary algorithms — search and optimization methods gleaned from the model of organic evolution — are applicable to this problem and known to be able to yield good solutions for many difficult practical optimization problems. The paper presents a parallel, steady-state evolutionary algorithm which exploits the available parallel machine configuration in an optimal manner. The algorithm is implemented under PVM and runs in a LAN of SUN SPARC workstations. The basic algorithm is applicable to arbitrary simulation models, and only the individual structure and the genetic operators must be specified for a particular application. As an application example, the problem of optimizing pressurized water reactor core reload designs is briefly discussed and first experimental results are presented.

1 Simulation Models

Presently, many computer models of technical and natural systems of high complexity exist and allow for a simulation of the dynamics of these systems. Generally speaking, such simulation models are conceivable as a kind of "black box" that receives a vector $\vec{x} = (x_1, \ldots, x_n)$ of input parameters and delivers some output $\vec{y}$. The output can only be determined by performing such a simulation experiment, i.e., no analytical knowledge about the input-output mapping is available.

Usually, simulation models are used in the manner of "what happens, if?" questions by varying the input parameters by hand and assessing the resulting output behaviour. In principle, however, the more important goal is to *optimize* the input parameters of the simulation model such that a certain criterion for the model output is maximized or minimized, i.e., the simulation model is interpreted as an objective function $f : M \rightarrow \boldsymbol{R}$, mapping the input $\vec{x} \in M \subseteq M_1 \times \ldots \times M_n$ into a real-valued quantity (notice that, in order to obtain a scalar quantity, the output vector $\vec{y}$ of the simulation model has to undergo some appropriate mapping — both the simulation model *and* this mapping are combined here into the function f).

The resulting problem to optimize the function f:

$$f(\vec{x}) \rightarrow \min \tag{1}$$

(the maximization problem is equivalent), i.e., to find a vector $\vec{x}^* \in M$ such that $f^* = f(\vec{x}^*) \leq f(\vec{x})$ for all $\vec{x} \in M$, is called the *global optimization problem* and is in most cases practically unsolvable. Nevertheless, even if practical processes might be improved only slightly by performing a computer aided optimization of the corresponding simulation model, the improvement can be valuable and easily overcompensate the resources invested.

So far, however, an optimization of simulation models is rarely attempted, because the corresponding objective functions are characterized by a number of properties that increase the complexity of the problem even further:

1. No analytical information about derivatives of f is available.
2. The objective function is *multimodal*.
3. An optimum f^* might be sensitive with respect to small changes of the input parameter values,

such that a technical realization of an optimum might be impossible.

4. The optimum point $\vec{x}^*$ might vary over time.

5. Usually, the set of *feasible solutions* (defined by additional constraints) is only a subset of M.

6. The objective function is usually subject to stochastic perturbations.

7. A single evaluation of the objective function (a run of the simulation model) is costly regarding CPU-time and possibly other hardware resources.

Especially property (7) normally restricts the number of possible experiments to a few dozens.

For these reasons, traditional optimization methods are not well suited for the optimization of simulation models or fail completely. This was demonstrated by Schwefel (1979, 1995), who performed extensive comparisons of the most popular optimization methods and *evolution strategies*, a representative of the class of *evolutionary algorithms*. These algorithms, which are briefly described in the next section, have many properties that recommend their application to simulation models.

2 Evolutionary Algorithms

Gleaned from the model of organic evolution, *evolutionary algorithms* are search and optimization methods which utilize the basic concepts of Darwinian evolution, i.e., a *population* of *individuals* which evolves towards better and better regions of the search space M by means of (randomized) processes of *selection*, *mutation*, and *recombination*. Each individual represents a search point in the space of potential solutions to a given problem, and the objective function value is interpreted in the sense of a fitness of the corresponding individual. The selection operator favors individuals of higher fitness to reproduce more often than those of lower fitness, recombination mixes the information of (usually two) individuals, and mutation introduces innovation into the population.

The common working scheme of evolutionary algorithms is summarized in algorithm 1, where a high-level pseudocode notation is used for reasons of clarity and shortness:

Algorithm 1 (General EA)

$t := 0$;
initialize $P_t = (\vec{a}_1, \ldots, \vec{a}_\mu) \in I^\mu$;
evaluate P_t;
while not *terminate* **do**
  $P'_t :=$ *recombine* P_t;
  $P''_t :=$ *mutate* P'_t;
  evaluate P''_t;
  $P_{t+1} :=$ *select* $(P''_t \cup Q)$;
  $t := t + 1$;
od

Initialization of the μ individuals of the population P_0 is often performed at random. Notice that the individual space I might contain more information than just the position within the search space $M \subseteq I$ of the underlying problem. During evaluation, the objective function value of each member of the population is calculated. The main loop of the algorithm iterates the operations recombination, mutation, evaluation, and selection until a termination criterion (in most cases, a maximum number of function evaluations or CPU-time) is fulfilled. $Q \in \{\emptyset, P_t\}$ denotes a set of individuals that are additionally taken into account by selection. As a final result, the algorithm usually yields the best individual which was encountered during the evolution process.

The main representatives of this general evolutionary algorithm, *genetic algorithms*, *evolution strategies*, and *evolutionary programming* (see (Bäck and Schwefel 1993) for a detailed overview and comparison of these algorithms), have clearly demonstrated their capability to yield good approximate solutions in case of complicated multimodal, discontinuous, non-differentiable, and even noisy or moving response surfaces of optimization problems. They are well suited to deal with most of the major difficulties of optimizing simulation models, because they are *direct* (derivative-free) methods (1), they are robust even for noisy (6) or time-varying (4) functions, have global search capabilities (2), and a reasonable set of constraint handling techniques have been developed and successfully applied (5).

A major difficulty, however, is caused by the need to perform a large number of function evaluations to evaluate a population of individuals over several generations, because this might require a huge amount of CPU-time. In the next section, we present a solution to this problem which is based on the parallel evaluation of objective function values using PVM in a workstation-LAN.

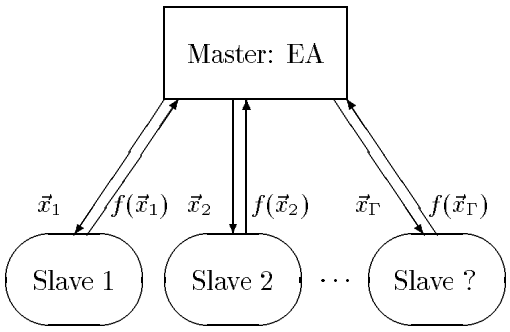

Figure 1: Master-Slave concept for the parallel evaluation of objective function values.

3 The Parallel Evolutionary Algorithm

Clearly, the most time-consuming part of the evolutionary algorithm as outlined in algorithm 1 consists in the evaluation of objective function values, if simulation models are considered for optimization. Consequently, a parallelization on the level of objective function evaluations is necessary, and we decided to evaluate one objective function value per available workstation at the same time. The appropriate parallelization model uses a simple *master-slave* approach as shown in figure 1.

Each of the slaves receives a parameter vector $\vec{x}_i$ (representing an individual of the evolutionary algorithm) from the master, performs a run of the simulation model with these input parameters, and returns the result $f(\vec{x}_i)$ of the run to the master.

It is important to notice that the time required for the evaluation of objective function values is different, because of heterogenous hardware resources and workloads in a local area network and input-depending running times of the simulation models. Therefore, in order to prevent idle times of the available machines, a generational synchronization is not appropriate. Instead, a so-called *steady-state* selection scheme (Whitley 1989) is utilized, which allows an asynchronous update of the population when an objective function value is received: If a newly created (by means of recombination and mutation) individual performs better than the worst individual of the population, it substitutes the worst one. This kind of replacement of the worst is called a $(\mu+1)$-selection in evolution strategy terminology (Bäck, Hoffmeister, and Schwefel 1991).

Immediately after selection, the parallel evolutionary algorithm creates a new offspring individual and sends it to the free slave machine for evaluation, such that the slave's idle time is minimal. In the following formulation of the parallel evolutionary algorithm, a high-level pseudocode description is used for reasons of clarity and shortness — the implementation of this basic algorithm using PVM primitives follows the master-slave approach outlined in (Geist, Beguelin, Dongarra, Jiang, Manchek, and Sunderam 1994):

Algorithm 2 (Parallel EA)

initialize $P := (\vec{x}_1, \ldots, \vec{x}_\mu)$;
{ evaluate initial population }
for $i = 1$ **to** ? **do**
  send $\vec{x}_i \rightarrow S_i$;
for $i = ? + 1$ **to** μ **do**
  receive $(\vec{x}, f(\vec{x})) \leftarrow S_\gamma$;
  send $\vec{x}_i \rightarrow S_\gamma$;
od
{ main recombine-mutate-select loop }
while $(i < i_{\max})$ **do**
  receive $(\vec{x}, f(\vec{x})) \leftarrow S_\gamma$;
  select: **if** $f(\vec{x}) \leq f(\vec{x}_{worst})$
  then $\vec{x}_{worst} := \vec{x}$;
  where $f(\vec{x}_{worst}) = \max\{f(\vec{x}') \mid \vec{x}' \in P\}$;
  recombine: $\vec{x}' := r(P)$;
  mutate: $\vec{x}'' := m(\vec{x}')$;
  send $\vec{x}'' \rightarrow S_\gamma$;
  $i := i + 1$;
od
{ collect missing slave results }
for $i = 1$ **to** ? **do**
  receive $(\vec{x}, f(\vec{x})) \leftarrow S_\gamma$;
  select; { as above }
od
return $\vec{x}_{best}$;

Here, ? denotes the number of available slave machines, and $\gamma \in \{1, \ldots, ?\}$ indicates the index of an arbitrary slave that has finished its computation and returns an individuals' function value. The first two for-loops of the algorithm perform the evaluation of the randomly initialized population P. The main loop continuously receives a result from an arbitrary slave S_γ, performs selection and eventually substitutes the worst population member by the individual evaluated by slave S_γ. Finally, a new individual $\vec{x}''$ is produced by recombination and mutation and sent to slave S_γ for evaluation. After termination of the main loop, the ? final objective

function values are collected, and the algorithm returns the best individual ever encountered during the search (notice that, using (μ+1)-selection, this is identical with the best individual that occurs in the actual population).

Like in evolution strategies, recombination is always performed for the creation of a new individual, and the parent individuals are chosen at random from the actual population. Recombination creates one new individual (a second one, often also obtained as a by-product, is discarded), which is then mutated and evaluated.

Notice that the algorithm described so far represents a general evolutionary heuristic. For specific application problems, the structure of the individual space I and the specific instances of the genetic operators recombination and mutation (as well as other topics such as constraint handling and the consideration of heuristic knowledge) have to be elaborated. In the next section, we present a brief overview of an application to a practical problem.

4 An Application Example

Our current application of algorithm 2 is the optimization of *pressurized water reactor* (PWR) core reload designs, where the task is to identify the arrangement of fresh and partially burnt fuel within the core such that the reactor's performance over the next burning cycle is optimized. The work on this application problem is performed in cooperation with Siemens AG Munich, and KWU Erlangen.

A core contains 193 fuel assemblies, which are arranged with quarter core symmetry. Consequently, only the arrangement of 48 (the center element is not exchanged) fuel assemblies (where assemblies are identified by integer numbers) and the corresponding assembly orientations (an integer $o \in \{0, -1, -2, -3\}$ for each assembly) have to be considered. The 48 fuel assemblies are chosen out of a total of 69 available assemblies (21 assemblies are kept in a storage), and an individual of the evolutionary algorithm consists of 48 integer values designating the fuel assemblies and their 48 corresponding orientation values.

The evaluation of the objective function value is performed by a simulation software that was provided by our industrial project partner. It has some very strong requirements for the available hardware configuration:

- A single run needs 90 sec. CPU-time on a SUN SPARC 10 machine.
- The simulator requires 28 MB RAM.
- 50 MB of temporary data are written to the harddisk for each simulation run.

For these reasons, up to now only 4 machines in our LAN are suitable for evaluating individuals. Nevertheless, this configuration allows to obtain about 3840 function values per day, which is a reasonable number to run the evolutionary algorithm.

For the preliminary experiments performed so far, we decided to use Goldberg's *order crossover* (see (Goldberg 1989), p. 174) as recombination operator. Mutation simply exchanges two randomly chosen fuel assemblies and mutates orientation values at random with small probability. A population size of 15 individuals is chosen as a compromise between genotypic diversity and selective pressure.

A typical minimization run where the quality of the best individual of the population is plotted over the number of function evaluations is shown in figure 2.

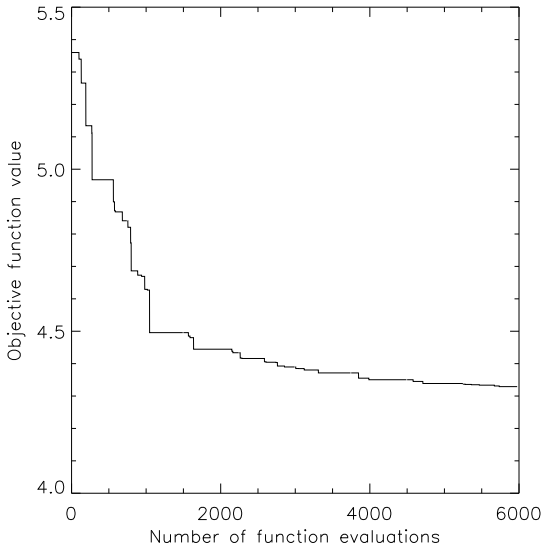

Figure 2: Plot of the objective function value of the best population member over the number of function evaluations performed.

The results of such runs are close to the best hand-optimized solution of quality 4.14, but did not improve it so far. However, several promising exten-

sions of the evolutionary algorithm are still under development:

- Operators are developed which take into account existing problem knowledge, especially regarding symmetry assumptions.
- Heuristic knowledge of experts is incorporated into the genetic operators and constraint handling mechanisms. The results obtained by (Poon and Parks 1992) with a so-called heuristic copy and match crossover operator seem to demonstrate that this is a promising approach.
- Better constraint handling techniques are incorporated (currently, the algorithm uses a penalty function method as outlined e.g. by (Smith and Tate 1993)).
- A number of additional machines are provided to increase the number of possible function evaluations by an order of magnitude.

In contrast to previous work on using genetic algorithms (Poon and Parks 1992) or simulated annealing (Kropaczek and Turinsky 1991) for this problem, the algorithm presented here exploits the implicit parallelism of evolutionary algorithms in the most natural way. For simulated annealing, which represents the state-of-the-art optimization method for this problem, a parallelization is not possible at all.

5 Conclusions

The parallel steady-state evolutionary algorithm presented here is a general approach for the optimization of simulation models, where the evaluation of a single objective function value usually requires a large amount of CPU-time, i.e., *communication time is small compared with computing time* of the parallel nodes. Using PVM, the algorithm is scalable according to both number and type of the available machines and can easily exploit growing networks of available machines.

A first practical example, the optimization of pressurized water reactor core reload designs, demonstrated the usefulness of the approach by yielding good results even when existing problem knowledge and expert heuristics were not considered for the design of the genetic operators and constraint handling techniques. The core reload design problem and other difficult optimization problems based on the goal to optimize the parameters of simulation models are important projects for further investigation and application of the algorithm presented in this paper.

Acknowledgements

The authors gratefully acknowledge support by the BMBF–project EVOALG (a cooperation of Informatik Centrum Dortmund, Siemens AG München, and Humboldt-Universität zu Berlin), grant 01 IB 403 A. This work was performed in cooperation with Siemens AG München and KWU Erlangen.

References

Bäck, T., F. Hoffmeister, and H.-P. Schwefel (1991). A survey of evolution strategies. In R. K. Belew and L. B. Booker (Eds.), *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 2–9. Morgan Kaufmann Publishers, San Mateo, CA.

Bäck, T. and H.-P. Schwefel (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation 1*(1), 1–23.

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam (1994). *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, MA.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, Reading, MA.

Kropaczek, D. J. and P. J. Turinsky (1991). In-core nuclear fuel management optimization for pressurized water reactors utilizing simulated annealing. *Nuclear Technology 95*, 9–32.

Poon, P. W. and G. T. Parks (1992). Optimising PWR reload core designs. In R. Männer and B. Manderick (Eds.), *Parallel Problem Solving from Nature 2*, pp. 371–380. Elsevier, Amsterdam.

Schwefel, H.-P. (1979, March). Direct search for optimal parameters within simulation models. In *Proc. 12th Annual Simulation Symposium*, Tampa, Florida/USA, pp. 91–102.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*. Sixth-Generation Computer Technology Series. Wiley, New York.

Smith, A. E. and D. M. Tate (1993). Genetic optimization using a penalty function. In S. Forrest (Ed.), *Proceedings of the 5th International Conference on Genetic Algorithms*, pp. 499–505. Morgan Kaufmann Publishers, San Mateo, CA.

Whitley, D. (1989). The GENITOR algorithm and selection pressure: Why rank–based allocation of reproductive trials is best. In J. D. Schaffer (Ed.), *Proceedings of the 3rd International Conference on Genetic Algorithms*, pp. 116–121. Morgan Kaufmann Publishers, San Mateo, CA.