Comparative Analysis of Parallel Gene Transfer Operators in the Bacterial Evolutionary Algorithm

Miklós F. Hatwágner

Department of Information Technology
Jedlik Ányos Faculty of Engineering
Széchenyi István University
Egyetem tér 1, H-9026 Győr, Hungary
e-mail: miklos.hatwagner@sze.hu

András Horváth

Department of Physics and Chemistry
Jedlik Ányos Faculty of Engineering
Széchenyi István University
Egyetem tér 1, H-9026 Győr, Hungary
e-mail: horvatha@sze.hu

Abstract: The Bacterial Evolutionary Algorithm (BEA) is an evolutionary method, originally meant to optimize the parameters of fuzzy systems. The authors have already proposed three modified versions of the original algorithm in a previous paper to make it usable in engineering applications with time-consuming object functions as well. Section 1 summarizes the earlier results. It presents the operators of the original BEA and the suggested parallel version. In Section 2, the optimal parameter settings and the analytical estimation of wall clock time in parallel computations are investigated. In Section 3, the paper deals with genetic diversity in different BEA versions. The effect of the modified gene transfer operators on genetic diversity is measured. The conclusion is that the proposed methods have quite good efficiency in all cases, and we can reach the ideal case if we have full control over the parameters.

Keywords: optimization; Bacterial Evolutionary Algorithm; genetic diversity; parallel computing; parallel efficiency

1 Introduction

The Bacterial Evolutionary Algorithm (BEA) [13] [12] is a relatively new member of the populous family of evolutionary algorithms [2]. It is a descendant of the Genetic Algorithm (GA) [6] and the Pseudo-Bacterial Genetic Algorithm (PBGA) [14]. The BEA was proposed by Norberto Eiji Nawa and Takeshi Furuhashi in the late '90s.

The BEA inherited many properties of the GA: it is also a global search algorithm, which is useful if a near optimal, approximate solution of a problem is acceptable. The algorithm is able to solve complex optimization problems even if they have non-linear, non-continuous, multimodal, high-dimensional properties. In contrast to gradient based methods, the BEA does not demand the use or the existence of the derivatives of the objective functions. Furthermore, the operators of the BEA achieve some functions, e.g. elitism, that can be implemented in the canonical GA only with additional code. This nature of the BEA helps to keep the program more compact and reliable as well.

The BEA and the GA are of course heuristic type optimization methods, thus there is no guarantee of finding the location of the global extreme value. Despite this, these algorithms perform well in real-life optimization problems, and theoretically the probability of finding the global optima can be made arbitrarily large (see [15]).

The BEA was originally developed to optimize fuzzy systems' parameters, but it could be a proper tool to solve complex design and engineering optimization problems related to computational fluid dynamics (CFD) or finite element models (FEM). However, such models need huge computational power, because every object function evaluation in the optimization process contains a full CFD or FEM calculation, which can take 0.1 to 5 hours of CPU-time. In a typical industrial application, the number of design variables is 10 to 30 and the whole optimization needs thousands of object function evaluations [10]. Thus the necessary CPU-time is in the order of 1 week to some months, and therefore parallelization is necessary. Sometimes the problem itself can be parallelized, but it is more adequate if the optimization process is executed in a parallel way. Unfortunately, the BEA is inherently sequential so this method in its original version is practically inapplicable in this area.

1.1 Shortcomings of the Bacterial Evolutionary Algorithm

[13] contains an exhaustive review of the BEA, and thus we will give only a short introduction here.

Similarly to the GA, the BEA also uses a record of possible solutions. These solutions are often called bacteria as well. The bacteria together form the population.

There are two main operators of the BEA: bacterial mutation and gene transfer. The repeated utilization of these operators results in a series of generations. When some kind of termination condition is fulfilled, the best bacterium of the last population is accepted as the result of the optimization.

Bacterial mutation (Fig. 1) optimizes the bacteria individually. That is why all the bacteria can be mutated at the same time. The mutation functions in the following way. Every bacterium has K clones. Initially the clones are copies of the original bacterium. In each step of the mutation, exactly one gene at a specified position is modified randomly in every clone. If a better gene value (allele) has been found, it is copied into the other clones. At the end of mutation, if the objective value of the best clone is better than the value of the original bacterium, the bacterium is replaced with this clone.

Consequently, the objective function has to be evaluated K times in one step, and such a step is repeated g (the number of genes) times during the operation. As was already shown in [9], the mutation operator evaluates the objective function $E_m = PKg$ times (P is the population size). Because several genes cannot be evaluated in parallel, the theoretical maximum speedup of the evaluation of the bacteria is $S_m = E_m/g = PK$, and it can be achieved with $C = PK$ processors. (It is assumed that the evaluation time of all the bacteria is the same and it is independent of the alleles.)

In a typical calculation, $P \approx 30\text{-}100$, $K \approx 20\text{-}50$, thus $C_{max} \approx 600\text{-}5000$. In most cases this number is much bigger than the number of processors in today's systems, and thus bacterial mutation is suitable to run on most of the multiprocessor systems without modifications.

The second operator of the BEA is the so-called gene transfer (Fig. 2). It operates with the ordered list of bacteria. The bacteria with better objective values get into the superior half, the others into the inferior half. The operator repeats T times the following: it chooses one bacterium from the superior half and one from the inferior half. After that it selects one portion of the genes of the superior bacterium and copies it into the inferior bacterium. This modification of the inferior bacterium involves the re-evaluation of its objective function, and the re-sorting of the bacteria. Depending on the objective value of the modified bacterium it may get into the superior half.

Since the modified bacterium can belong to any of the two halves, it is obvious that the consecutive gene transfers are not independent. Because of this behaviour parallel gene transfers cannot be realized, and therefore modification of the gene transfer operator is needed for parallelization.

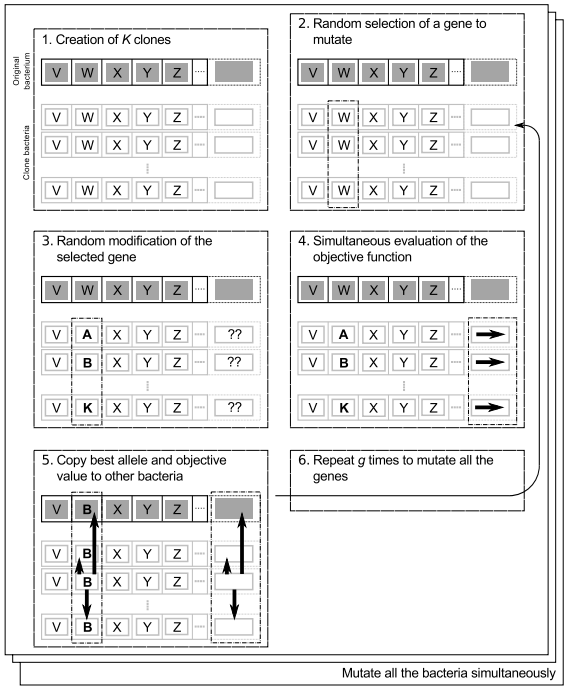

Figure 1
Schematic view of the bacterial mutation operator

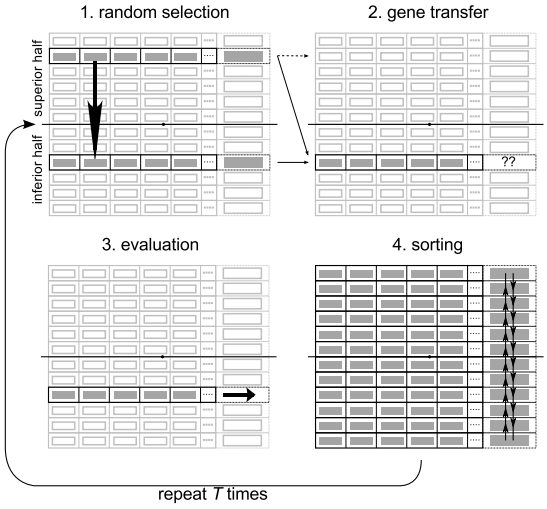

Figure 2
Schematic view of the gene transfer operator

1.2 Overview of the Suggested Modified Gene Transfer Operators

Three modified gene transfer operators were suggested in [9]. All of them are the derivatives of the original version with slight modifications.

"Original gene transfer with auxiliary population" (BEA Aux., Fig. 3) also keeps a record of superior and inferior bacteria based on the objective values. Even the selection of the superior and inferior bacterium is the same as before. This version of gene transfer keeps the inferior bacterium untouched; the modified bacterium goes into an "auxiliary population". The operator first fills the whole auxiliary population with A modified bacteria, and only then starts to evaluate the objective values of them simultaneously. After the evaluation the best P of $P+A$ bacteria form the population, and the other, worse bacteria are dropped. This procedure has to be repeated until the desired number of total gene transfers (T) is reached.

The second suggested gene transfer was called "gene transfer inspired by Microbial Genetic Algorithm" (pMGA, Fig. 4). The Microbial Genetic Algorithm (MGA) [8] is a simplistic GA. The MGA gave the idea of a new gene transfer because its selection and crossover can be regarded as a gene transfer. pMGA creates random and disjoint pairs of bacteria. The better bacterium (the so called "winner") of such a pair transfers some portion of its genetic material to the worse bacterium (loser). Because the pairs are disjoint, the gene transfer and the evaluation of the modified bacteria are independent from other pairs. This property allows parallel execution. Unfortunately, the size of the population limits the number of parallel gene transfers to $P/2$. If more gene transfers are required, the operation has to be repeated.

The MGA inspired other researchers as well to modify and use some of its simple genetic operators in the Bacterial Memetic Algorithm [7] [11] [14]. The MGA inspired gene transfer operation performs well on several important problems, e.g. the travelling salesman problem [4]. It was pointed out that it is easy to implement the parallel version of this gene transfer operator.

The third suggested version of gene transfer, "gene transfer inspired by MGA with auxiliary population," (see [9]) is a mixture of the previous two, in order to eliminate the $P/2$ barrier of the pMGA. The pMGA Aux. uses an auxiliary population (like the BEA Aux.) and places the modified bacteria into it, instead of the instant overwriting of the loser bacteria.

Note that the usage of the modified gene transfers suggested above influences the optimization process. For example, in the case of the original gene transfer, the inferior bacterium is always overwritten, no matter how good or bad it is; but with an auxiliary population it can survive if the auxiliary population contains mostly worse individuals. This is somehow similar to elitism, and thus it increases the average fitness of the next population but keeps the genetic diversity lower.

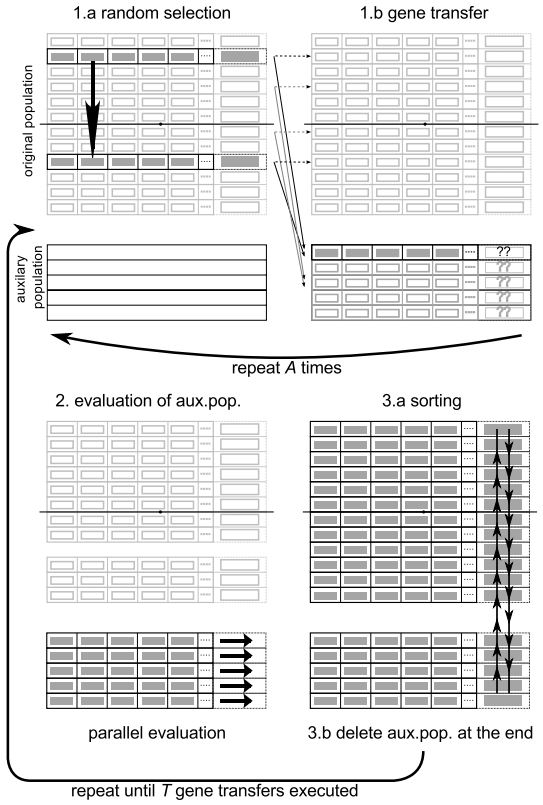

Figure 3
Schematic view of the original gene transfer with auxiliary population (BEA Aux.)

Another side effect of the modified operators can also be realized. The original gene transfer evaluates the objective function T times. This means that the modified allele has at most $T-1$ chances to infiltrate into other bacteria during the same gene transfer. But in the case of using an auxiliary population, the number of chances to infiltrate drops to at most $(T/A)-1$. The situation is similar in the case of the pMGA: a better allele can be inherited at most $(T/(P/2))-1$ times.

At this point, some important questions arise, e.g.: Do modified gene transfers sacrifice genetic diversity on the altar of parallel execution? What are the scaling properties of the different gene transfer versions? These questions can be answered easily after exhaustive empirical tests.

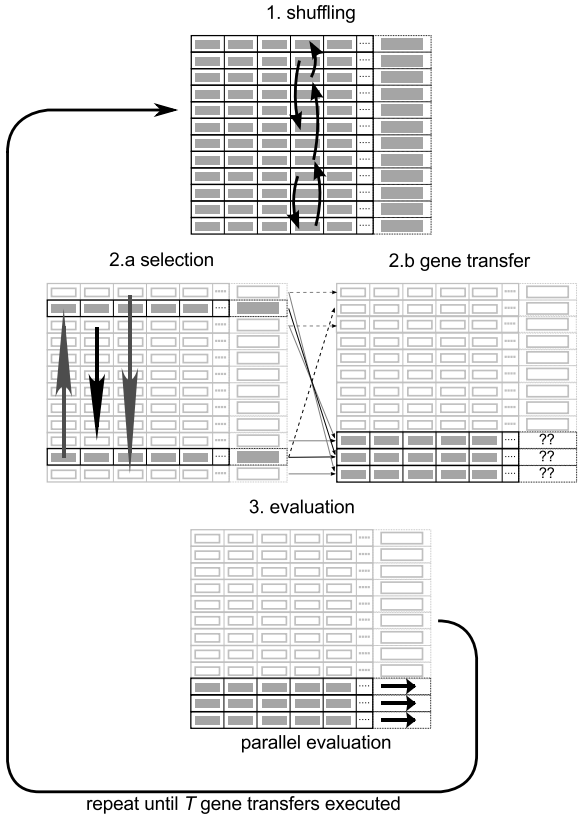

Figure 4
Schematic view of the gene transfer inspired by MGA (pMGA)

1.3 Preceding Results

The first investigation of parallel versions of the BEA and the MGA can be found in [9]. Here a custom optimization program with a master-slave model was used. The slave computers calculated the value of the objective function, while all other tasks fell upon the master computer, including the execution of the genetic operators.

Five test functions (De Jong's 1st and 3rd, Step, Rastrigin, Keane [3] [11] [16]) were used. These functions are well known in literature, and their qualitative properties and global extremes are also known. They are of different types, e.g. Step is non-continuous, Rastrigin is smooth but has many local minima, etc., and therefore they represent different kind of problem types from real life. The big difference between these functions and practical problems is the execution time: in an engineering application, where parallel execution is important, an objective

function evaluation takes many seconds, and therefore communication time is negligible; but in a modern hardware test function evaluation takes only a small fraction of a second. In order to simulate real life problems, a small artificial delay (approx. 0.005 sec.) was built into the test functions. The overhead of communication became negligible with this trick.

Based on the test calculations it can be concluded that the three modified gene transfer operators are applicable in real life problems. On the contrary, the acceleration of the optimization using the original gene transfer is the consequence of bacterial mutation only. It is not recommended to use the original version if more than one CPU is available, but in some cases the modified operators proved to be faster even with one CPU.

All the modified versions have good scaling properties. In [9] the authors used at most 16 processors. In this range the pMGA was slightly the fastest one, but the difference was very small. In this paper the range of investigation is extended to a higher number of CPUs. The ideal setting of number of gene transfers and the genetic diversity in different methods is also examined.

2 Analysis of the Suggested Operators

2.1 Right Settings of the Optimization Program

2.1.1 Maximizing CPU Utilization in Gene Transfer

In [9] the authors drew their conclusions using the results of their custom optimization program. The settings of this program were carefully chosen before the start of the executions. These settings were optimal in the sense that the program produced the same result with the least evaluation of the objective function. However, in a system containing a lot of CPUs the number of object function evaluations may be not proportional to wall clock time if we have a lot of idle CPUs due to bad parameters. In this section a simple model for CPU utilization is presented.

Let's assume that the evaluation time of an object function:

- is constant and one unit long,
- takes much more CPU time than bacterial (genetic) operators and master-slave communication.

Using these assumptions we can divide the calculation process into "computational rounds": if we have C CPUs, the master can send at most C object

function evaluations to them simultaneously, then it collects the results and sends another object functions again. CPU utilization is ideal if in every computational round exactly C object function evaluation is to be sent to the slaves.

Let N denote the maximum number of new bacteria that can be evaluated simultaneously during gene transfer. Without an auxiliary population, it is the half of the population size: $N_{woa} = \lfloor P/2 \rfloor$; with an auxiliary population, it is the size of the auxiliary population: $N_{wa} = A$. ($\lfloor . \rfloor$ is the rounding down function often referred to as "floor".)

If N is not a multiple of C, there will be idle CPUs during the evaluation of these individuals. This means that $\lceil N/C \rceil$ computational rounds are needed to evaluate N individuals. ($\lceil . \rceil$ is the rounding up function often referred to as "ceil".)

Another case is when idle CPUs appear if the total number of transfers (T) is not the multiple of N. In this case the $\lceil N/C \rceil$ computational rounds mentioned above must be executed $\lfloor T/N \rfloor$ times, but $T - \lfloor T/N \rfloor N$ evaluations remain, which implies $\lceil (T - \lfloor T/N \rfloor N)/C \rceil$ extra computational rounds.

Thus the utilisation of the slave CPUs during gene transfer can be specified with the following formula:

$$U_{C,T} = \frac{T}{\left\lfloor \frac{T}{N} \right\rfloor \left\lceil \frac{N}{C} \right\rceil C + \left\lceil \frac{T - \lfloor T/N \rfloor N}{C} \right\rceil C} \tag{1}$$

For example, in [9] in the case of the optimization of the Rastrigin function the authors used $P = 40$, $T = 400$, $A = 20$, $C = 16$. In this case the CPU utilisation is only $U_{C,T} = 62.5\%$, whether an auxiliary population was used or not.

It is easy to achieve 100% utilisation, if we know the number of processors in advance: let N be a multiple of C and T a multiple of N, thus we get $U_{C,T} = 1.0$.

2.1.2 The Optimal Number of Gene Transfers

Changing the gene transfer algorithm may change the optimal number of using this operator. A series of test calculations was performed to study this effect. For the sake of compactness, only the results of the optimization of the Rastrigin function are reviewed in this section. The main parameters of the test were: $K = 1$, $C = 64$, $P = 128$, $A = 64$. ($U_{C,T} = 100\%$) Wall clock time of reaching 0.01 object function value was measured and averaged over 20 independent calculations. (1 to 4% relative standard deviation was observed.)

For the sake of compactness, only the results of the optimization of the Rastrigin function are reviewed in Table 1. The phenomenon is the same in the case of other test functions.

Table 1
The effect of various *T/P* ratios on optimization time

T/P	BEA	BEA Aux.	pMGA	pMGA Aux.
4	*130.982*	13.588	15.699	13.255
6	146.793	11.736	13.619	11.786
8	169.898	*10.723*	12.634	11.371
12	190.434	10.688	12.117	*10.829*
16	242.417	11.177	*11.837*	11.156
24	322.155	11.970	12.899	12.956
32	429.840	13.300	14.649	13.696

Table 1 shows that for original BEA a small *T/P* ratio is optimal. The reason is simple: gene transfers in the BEA scale poorly for 64 CPUs, and for small number of transfers, the mutation operator dominates. (However, the wall clock time value is much higher than the one in parallel versions.)

For parallel versions the *T/P* ratio has an optimal range. One can conclude that the best *T/P* ratio for the parallel gene transfer operators is between 8 and 16 and there is only a small difference within this range. The tests showed similar results for other test problems and CPU numbers.

2.2 Test Methodology

Considering the observations mentioned above, one can choose good parameters for *T*, *A*, based on the *C* number of CPUs. There is however another problem: to measure the efficiency of different methods a lot of independent calculations must be performed for all the test problems. It takes a lot of time if we use real life object functions with many seconds of CPU-time consumption.

One possible solution is to use easily formulated test functions with very small calculation time, but apply a small amount of artificial delay, which makes the object function evaluation longer than the communication time. This method was used successfully in [9] for at most 16 CPUs, but it is not a good method for a much higher number of processors. Test calculations with 64 CPUs and 0.05s artificial delay showed more than 100% extra time originating from the communication bottlenecks. (Note that the communication between master and slaves consists of smaller than 1kB data blocks, but at the end of a computational round, when all the slaves want to send the data back to the master, a significant bottleneck arises.)

Increasing the artificial delay may help and it could bring the test calculations closer to real life, but results in very long test calculation time. To overcome these difficulties, another approach is used in this paper:

- No artificial delay is used.

- Load balancing is realised in calculations.
- All object function evaluations are logged with their sequential number, and value.

Load balancing is a key part of the measurements. Instead of sending the next job to the first idle slave, the master sends the jobs to the slaves in a predefined order to ensure as similar CPU loads as possible. Otherwise if there are several slaves (e.g. 64 or more) in the system and the evaluation time of the objective function is short compared to the communication time between master and slave, the first slaves would be fully loaded while the rest of the slaves remain idle. In this way the artificial delay included in the objective functions is not needed anymore.

This kind of test calculation gives enough information to reconstruct how many computational rounds would be needed if the optimization was executed in a load-balanced way on a *C*-processor machine. Assuming nearly identical time for object function evaluations, the number of computational rounds is proportional to wall-clock time. In the next subsection deduction of the number of computational rounds is presented.

2.3 The Number of Computation Rounds

The flow of a bacterial-type optimization begins with a random population generation and evaluation of the individuals. This means that P object function calls happen in the 0^{th} generation. After this initialization new generations are produced by $E_m = PKg$ mutations and $E_T = T$ gene transfers. (See Sec 1.1 for notations.) Thus the E_G number of objective function evaluation needed to create the next generation can be expressed as:

$$E_G = E_m + E_T = PKg + T \tag{2}$$

In a test calculation we measure how many objective function evaluations are required to reach a specific target objective value. (Naturally, an average number of independent calculations is used.) Let us denote this number of evaluations with M. If this number is known, one can calculate how many generations and how many computational rounds are needed for the optimization, and thus get a good approximation of wall-clock time.

The number of fully completed generations (except the 0^{th} generation, which needs P evaluations) can be expressed as:

$$G_f = \left\lfloor \frac{M - P}{E_G} \right\rfloor \tag{3}$$

For the last (possibly non-finished) generation E_l objective function evaluations remain.

$$E_l = (M - P) - G_f E_G \tag{4}$$

In the last generation the number of mutations and gene transfers may be less than E_m and E_T. If we perform the mutation first, the number of objective function evaluations used by mutation in the last generation will be:

$$E_{l,m} = \min(E_m, E_l) \tag{5}$$

and the number of objective function evaluations executed by gene transfers in the last generation is:

$$E_{l,T} = \max\left(0, E_l - E_{l,m}\right) \tag{6}$$

Now it is easy to express the needed number of computational rounds.

All the methods need R_0 computational rounds to evaluate the 0th generation:

$$R_0 = \lceil P / C \rceil \tag{7}$$

All the methods examined here use PK independent mutations, therefore

$$E_{m,p} = \min(C, PK) \tag{8}$$

evaluations can be made in parallel (in one computational round).

The original BEA needs $R_{f,BEA}$ computational rounds for every fully evaluated generations (remember that gene transfers are sequential operations in the BEA.):

$$R_{f,BEA} = \lceil PK / E_{m,p} \rceil g + E_T \tag{9}$$

Thus the number of computational rounds required by optimization using the BEA can be expressed as:

$$R_{BEA} = R_0 + G_f R_{f,BEA} + \lceil E_{l,m} / E_{m,p} \rceil + E_{l,T} \tag{10}$$

Using similar considerations the number of computational rounds of parallel versions can be expressed also.

$$\begin{aligned} R_{parallel} &= R_0 + \\ &G_f \left(\left\lceil \frac{PK}{E_{m,p}} \right\rceil g + \left\lfloor \frac{T}{N} \right\rfloor \left\lceil \frac{N}{C} \right\rceil + \left\lceil \frac{T - \lfloor T/N \rfloor N}{C} \right\rceil \right) + \\ &\left(\left\lceil \frac{E_{l,m}}{E_{m,p}} \right\rceil + \left\lfloor \frac{E_{l,T}}{N} \right\rfloor \left\lceil \frac{N}{C} \right\rceil + \left\lceil \frac{E_{l,T} - \lfloor E_{l,T}/N \rfloor N}{C} \right\rceil \right) \end{aligned} \tag{11}$$

$R_{pMGA\ Aux}$ is the number of computational rounds needed by optimization using the pMGA Aux. The value of it is the same as the number of computational rounds used up by the BEA Aux, $R_{BEA\ Aux}$.

$$R_{pMGAAux} = R_{BEAAux} = R_0 + G_f\left(\left\lceil \frac{PK}{C} \right\rceil g + \left\lfloor \frac{T}{A} \right\rfloor \left\lceil \frac{A}{C} \right\rceil + \left\lceil \frac{T - \left\lfloor \frac{T}{A} \right\rfloor A}{C} \right\rceil\right) + \left(\left\lceil \frac{E_{l,m}}{R_m} \right\rceil + \left\lfloor \frac{E_{l,T}}{A} \right\rfloor \left\lceil \frac{A}{C} \right\rceil + \left\lceil \frac{E_{l,T} - \left\lfloor \frac{E_{l,T}}{A} \right\rfloor A}{C} \right\rceil\right) \quad (12)$$

Here we used the same notation as in Sec 2.1.1, namely N is the maximum number of new bacteria that can be evaluated simultaneously during gene transfer. For the BEA Aux. and the pMGA Aux. methods $N = N_{wa} = A$, for pMGA $N = N_{woa} = \lfloor P/2 \rfloor$.

2.4 Test Results

Some test optimizations with different settings were performed to check the correctness of the above formulas, but there were no differences between the calculated and the measured number of computational rounds.

Six standard problems were used for testing the behaviour of modified gene transfers: Rastrigin, Keane, Step, Ackley, DeJong's 1st and DeJong's 3rd functions. (See [3], [11], [16].) Some of these are unimodal (eg. De Jong's 1st), others are multimodal (eg. De Jong's 3rd). Some of them are continuous (eg. Rastrigin) while others are not (eg. Step). This means the results are valid for a wide range of problems.

Even though the test functions have very different properties, and thus they represents a wide range of problems, the authors plan to execute more sophisticated tests with a much wider and more easily parameterizable set of test functions in the future. These test problem sets could be generated with the appropriate functions, see e.g. [1] [5].

Table 2 shows the measured values of M for the Rastrigin function with the target objective value of 0.01, and the calculated number of computational rounds as well. The main settings of the optimization program were the following: g = 20, P = 128, A = 64. According to the considerations, the numerical experiment must be performed only for one specific C value, and the others can be calculated from this. (Because the authors have been able to use a 64-core machine, C = 64 was used in the calculations.)

Table 2
Computational rounds needed with different number of CPUs

C	1 (=M)	2	8	64	256
R_{BEA}	128048	74520	34374	22665	21829
R_{BEAAux}	122732	61366	15342	1918	1117
R_{pMGA}	141582	70791	17698	2213	1291
$R_{pMGAAux}$	124608	62304	15576	1947	1134

The parallel efficiency of the calculations is also presented in Table 3. ($E_p=(R_1/R_X)/C$, where R_x is the number of computational rounds in x CPU-case.)

Table 3
Parallel efficiency of optimization with different number of CPUs (Rastrigin fn.)

C	1	2	8	64	256
$E_{p,BEA}$	1.000	0.859	0.466	0.088	0.023
$E_{p,BEAAux}$	1.000	1.000	1.000	1.000	0.429
$E_{p,pMGA}$	1.000	1.000	1.000	1.000	0.428
$E_{p,pMGAAux}$	1.000	1.000	1.000	1.000	0.429

Tables 2 and 3 show that the parallel versions scale ideally until full utilization is achieved. In the last column $N > A$, and therefore this is not true, and all the methods will slow down, but the original BEA shows bad performance for a much smaller number of CPUs. These results are in good correspondence with the ones in [9].

The R values for the $C = 1$ case shows the difference of the methods in 1 CPU case. It is not obvious that parallel versions are comparable with the original BEA in this case. Table 2 shows that the pMGA is worse than the BEA, but all the methods with auxiliary populations are better than the original bacterial algorithm even on 1 core. Due to the good scaling properties, even the pMGA beats the BEA in all $C > 1$ cases.

Testing with other functions show completely similar structure, and therefore only a small, significant part of the results are presented here.

Table 4 shows the ratio of the computational rounds needed by the modified gene transfers and the original version. Using only one slave CPU, the pMGA usually needs slightly more computational rounds (ie. wall-clock time) than the original gene transfer. In every other case, except for the Keane-function, all of the modified gene transfers perform better even in the $C = 1$ case, but the difference is only 1-2% in this case. The two versions using auxiliary populations are the best. These gene transfer methods provide practically the same performance.

Table 4
Ratios of computational rounds in case of 1 CPU (C=1)

	Rastrigin	Keane	Step	Ackley	DeJong's 1st	DeJong's 3rd
$\frac{R_{BEAAux.}}{R_{BEA}}$	0.958	1.018	0.832	0.935	0.947	0.949
$\frac{R_{pMGA}}{R_{BEA}}$	1.106	1.169	1.045	1.126	0.997	1.090
$\frac{R_{pMGAAux.}}{R_{BEA}}$	0.973	1.023	0.791	0.966	0.928	0.876

Table 5 shows the same ratios as Table 4 for the $C = 64$ case. Because of the good scaling properties of BEA Aux., pMGA and pMGA Aux. methods, all the values are lower than 1, which means that they are significantly better than the original BEA. The two methods with auxiliary populations are approximately efficient in the same degree, and the pMGA is slightly worse than these two methods.

It is clear that for a known C value, one can choose the other parameters for ideal scaling. But in practice sometimes the number of CPUs is not a fixed, predefined number. For example, some of the CPUs in the cluster are allocated for other jobs. It is important to examine the scaling properties of these methods for a "random" number of CPUs also. The (12) and (13) formulas can be used for such calculations.

Table 6 shows the parallel efficiency values in case of non-optimal values of C. (Other parameters are the same as in the above optimization of Rastrigin function.)

Table 5
Ratios of computational rounds in case of 64 CPUs

	Rastrigin	Keane	Step	Ackley	DeJong's 1st	DeJong's 3rd
$\frac{R_{BEAAux.}}{R_{BEA}}$	0.085	0.083	0.141	0.056	0.062	0.146
$\frac{R_{pMGA}}{R_{BEA}}$	0.097	0.095	0.177	0.067	0.065	0.168
$\frac{R_{pMGAAux.}}{R_{BEA}}$	0.086	0.083	0.134	0.058	0.061	0.135

Table 6
Parallel efficiency of optimization with non-optimal number of CPUs

C	15	30	45	60	75
$E_{p,BEA}$	0.299	0.170	0.121	0.091	0.075
$E_{p,BEAAux}$	0.932	0.828	0.900	0.678	0.855
$E_{p,pMGA}$	0.931	0.826	0.898	0.674	0.853
$E_{p,pMGAAux}$	0.932	0.828	0.899	0.676	0.855

The original BEA scales poorly, the other ones scale in a very similar way, and the efficiency of parallel versions is never lower than 0.67 in these examples. One can construct a parameter set when the parallel versions scale significantly worse, but calculations showed that for plausible cases the efficiency is always above 0.5.

Note that the formula used to calculate the number of computational rounds (11) is the same in the last three cases in our parameter settings, but the corresponding values of M are different in each line. That is why the number of computational rounds and efficiency values are different.

Figure 5 also shows the same effect for a wider range of CPU numbers. The parallel methods give the same curves according to the paper, and therefore only one of them is plotted. It is clear that if we can control the parameters, ideal efficiency can be achieved, but even with an unexpected number of CPUs the efficiency is mostly above 0.8.

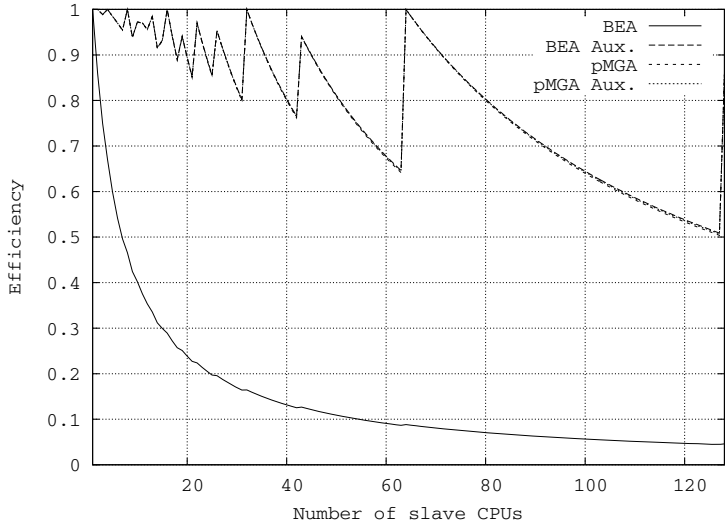

Figure 5
Efficiency of the algorithm as the function of the number of CPUs

3 The Effect of the Modified Gene Transfers on Genetic Diversity

3.1 Measuring Genetic Diversity

The modified gene transfer operators (BEA Aux., pMGA, pMGA Aux.) significantly differ from the original variant in one aspect: the new bacteria created by gene transfer are able to share their genetic information with other members of the population to a smaller degree, i.e., not more than $\lceil T/N \rceil$ times. This can decrease the genetic diversity, which can result in increasing the runtime of the program, and therefore it is important to measure the genetic diversity and prevent it from being too small.

The difference between two bacteria were defined with the following formula:

$$d_{ij} = \sqrt{\frac{\sum_{k=1}^{m}\left(\frac{x_{ik} - x_{jk}}{X_{k,\max} - X_{k,\min}}\right)^2}{g}} \tag{13}$$

This "distance" was originally proposed by Goldberg to realise niching with [6]. Here x_{ik} is the k^{th} chromosome of the i^{th} bacterium, $X_{k,max}$ and $X_{k,min}$ are the possible maximum and minimum values of a chromosome. Using this formula, the genetic diversity of a population can be determined in the following way:

$$D = \frac{1}{P-1}\sum_{i=1}^{P} d_{i,best} \tag{14}$$

The expressive meaning of this "genetic diversity" is straightforward: the value is between 0 and 1 and shows the average relative difference between chromosome values.

Figure 6 shows the change of genetic diversity during the optimization of the Rastrigin function. The chart shows the average of 15 repeated measurements. The main settings were the following: $C = 64$, $P = 64$, $K = 1$, $g = 20$, $T = 512$, $A = 64$.

It can be seen that all the methods show very similar genetic diversity functions. It is good news in the sense that the parallel methods are not worse than the original BEA. But all the methods show extremely low diversity at the end of the calculations, when the population consists of almost identical copies of an individual. When this happens, only the random search, due to the mutation operator, produces new values, which has very slow convergence.

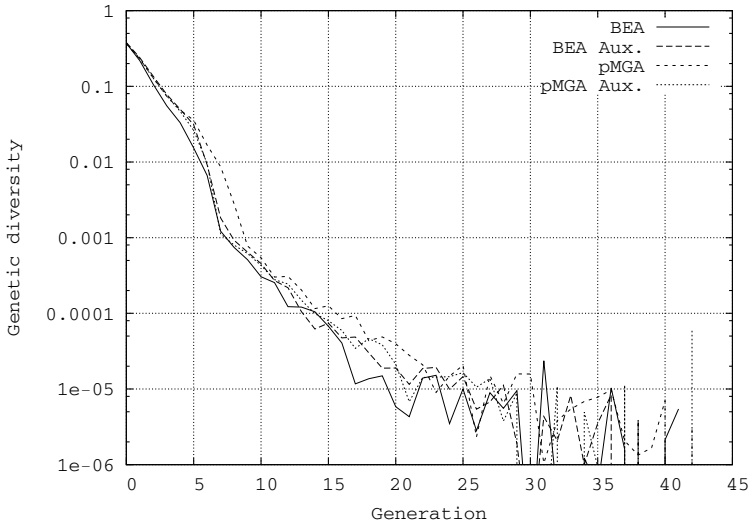

Figure 6
Average genetic diversity during optimization

The same phenomenon was discovered in the case of other test functions as well. This means that all the bacterial type optimizations produce slow convergence near the optimum because of the low genetic diversity. This conclusion is not surprising: the methods examined above have no mechanism to protect them against the reproduction of identical or very similar individuals.

Conclusions

The authors examined the effect of the recommendations and some other phenomenon as well in [9].

It was concluded that the ratio of gene transfers and population size heavily affects the optimization time. The ratio should be between 8 and 16 in case of using the parallel gene transfer operators. It is a rule of thumb if exhaustive tuning of the settings of the optimization cannot be realized.

Formulas have been given to estimate the change of wall clock time needed by optimization programs if the same optimization is executed with a different number of CPUs. It has been shown that the proposed parallel methods scale quite well even when the number of CPUs is not know in advance, while the original BEA's parallel efficiency is extremely low.

Lastly, it was pointed out that the parallel gene transfer operators do not worsen the genetic diversity in a considerable measure.

Based on this study the authors can recommend the parallel bacterial type methods with the optimal parameter setting described in this paper.

Acknowledgement

The authors' research is supported by the National Development Agency and the European Union within the frame of the project TAMOP 4.2.2-08/1-2008-0021 at the Széchenyi István University entitled "Simulation and Optimization - basic research in numerical mathematics".

References

[1] Addis, B., Locatelli, M.: A New Class of Test Functions for Global Optimization, Journal of Global Optimization, Vol. 38(3), 2007, pp. 479-501

[2] Bäck, T., Fogel, D. B., Michalewicz, Z.: Handbook of Evolutionary Computation, IOP Publishing and Oxford University Press, Abingdon, 1997

[3] De Jong, K. A.: An Analysis of the Behavior of a Class of Genetic Adaptive Systems, dissertation, University of Michigan, 1975

[4] Farkas, M., Földesi, P., Botzheim, J., Kóczy, L. T.: A Comparative Analysis of Different Infection Strategies of Bacterial Memetic Algorithms, in Proc. of 14th International Conference on Intelligent Engineering Systems (INES 2010), Las Palmas of Gran Canaria, Spain, May 5-7, 2010

[5] Gaviano, M., Kvasov, D. E., Lera, D., Sergeyev, Y. D.: Algorithm 829: Software for Generation of Classes of Test Functions with Known Local and Global Minima for Global Optimization, ACM Transactions on Mathematical Software, Vol. 29, No. 4, December 2003, pp. 469-480

[6] Goldberg, D. E.: Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley Publishing Company, Inc., USA, 1989

[7] Grosan, C., Abraham, A.: Hybrid Evolutionary Algorithms: Methodologies, Architectures, and Reviews, Studies in Computational Intelligence, Vol. 75/2007, 2007, pp. 1-17

[8] Harvey, I.: The Microbial Genetic Algorithm, in Proceedings of the Tenth European Conference on Artificial Life, editor G. Kampis et al, Springer LNCS., Heidelberg, 2009

[9] Hatwágner, M., Horváth, A.: Parallel Gene Transfer Operations for the Bacterial Evolutionary Algorithm, Acta Technica Jaurinensis, Vol. 4, 2011, pp. 89-113

[10] Horváth, A., Horváth Z.: Optimal Shape Design of Diesel Intake Ports with Evolutionary Algorithm, Proceedings of 5th European Conference on Numerical Mathematics and Advanced Applications (ENUMATH 2003), edited by Feistauer, M. et al., Springer Verlag, 2004, pp. 459-470

[11] Mühlenbein, H., Schomisch, D., Born, J.: The Parallel Genetic Algorithm as Function Optimizer, Parallel Computing, Vol. 17, 1991, pp. 619-632

[12] Nawa, N. E., Furuhashi, T.: A Study on the Effect of Transfer of Genes for the Bacterial Evolutionary Algorithm, Second Intenational Conference on Knowledge-based Intelligent Electronic System, editors Jain, L. C., Jain, R. K., Adelaide, Australia, 21-23 April 1998, pp. 585-590

[13] Nawa, N. E., Furuhashi, T.: Fuzzy System Parameters Discovery by Bacterial Evolutionary Algorithm, IEEE Transactions on Fuzzy Systems, Vol. 7, No. 5, 1999, pp. 608-616

[14] Nawa, N. E., Hashiyama, T., Furuhashi, T., Uchikawa, Y.: A Study on Fuzzy Rules Discovery Using Pseudo-Bacterial Genetic Algorithm with Adaptive Operator, Proceedings of IEEE Int. Conf. on Evolutionary Computation, ICEC'97, 1997

[15] Pintér, J. D.: Global Optimization in Action, Kluwer Academic Publishers, Dordrecht, Netherlands, 1996

[16] Törn, A., Zilinskas, A.: Global Optimization, Lecture Notes in Computer Science, No. 350, Springer-Verlag, Berlin, 1989