Parallel Genetic Algorithm on the CUDA Architecture

Petr Pospichal, Jiri Jaros, and Josef Schwarz

Brno University of Technology, Faculty of Information Technology, Department of Computer Systems, Bozetechova 2, 612 66 Brno, Czech Republic
Tel.: +420-54114 1364; Fax: +420-541141270
{ipospichal,jarosjir,schwarz}@fit.vutbr.cz

Abstract. This paper deals with the mapping of the parallel island-based genetic algorithm with unidirectional ring migrations to nVidia CUDA software model. The proposed mapping is tested using Rosenbrock's, Griewank's and Michalewicz's benchmark functions. The obtained results indicate that our approach leads to speedups up to seven thousand times higher compared to one CPU thread while maintaining a reasonable results quality. This clearly shows that GPUs have a potential for acceleration of GAs and allow to solve much complex tasks.

1 Introduction

Genetic Algorithms (GA) [3] are powerful, domain-independent search techniques inspired by Darwinian theory. In general, GAs employ selection, mutation, and crossover to generate new search points in a state space. A genetic algorithm starts with a set of individuals that forms a population of the algorithm. Usually, the initial population is generated randomly using a uniform distribution. On every iteration of the algorithm, each individual is evaluated using the fitness function and the termination function is invoked to determine whether the termination criteria have been satisfied. The algorithm ends if acceptable solutions have been found or the computational resources have been spent. Otherwise, the individuals in the population are manipulated by applying different evolutionary operators such as mutation and crossover. Individuals from the previous population are called parents while those created by applying evolutionary operators to the parents are called offsprings. The consecutive process of replacement forms a new population for the next generation.

Although GAs are very effective in solving many practical problems, their execution time can become a limiting factor for some huge problems, because a lot of candidate solutions must be evaluated. Fortunately, the most time-consuming fitness evaluations can be performed independently for each individual in the population using various types of parallelization.

There are different ways of exploiting parallelism in GAs: master-slave models, fine-grained models, island models, and hybrid models [6].

One of the most promising variant is an island model. Island models can fully explore the computing power of course grain parallel computers. The population

C. Di Chio et al. (Eds.): EvoApplications 2010, Part I, LNCS 6024, pp. 442–451, 2010.

is divided into a few subpopulations, and each of them evolves separately on different processor. Island populations are free to converge toward different suboptima. The migration operator is supposed to mix good features that emerge locally in the different subpopulations.

Nowadays modern Graphic Processing Units (GPU), although originally designed for real-time 3D rendering can be seen as very fast highly parallel general-purpose systems [4,5] and hence, employed with advantage to accelerate GAs. The second section introduces the main features of nVidia GPU platform. Section 3 describes the mapping of parallel GA onto nVidia CUDA architecture taking into account restrictions of data-parallel processing. Experimental results are presented and discussed in section 4. Section 5 concludes the paper.

2 General Purpose Computation on GPU

Driven by ever increasing requirements from the video game industry, GPUs have evolved into very powerful and flexible processors, while their price remained in the range of consumer market. They now offer floating-point calculation much faster than today's CPU and, beyond graphics applications; they are very well suited to address general problems that can be expressed as data-parallel computations (i.e. the same code is executed on many different data elements).

Moreover, several general purpose high-level languages for GPUs have become available such as CUDA [7] and OpenCL [8] and thus developers do not need any more to master the extra complexity of graphics programming APIs when they design non graphics applications [9].

Modern graphics cards are in fact very powerful massively parallel computers that have (among others) one main drawback: all the elementary processors on the card are organised into larger multi-processors. They have to execute the same instruction at the same time but on different data (SIMD model, for Single Instruction Multiple Data).

GAs need to run an identical evaluation function on different individuals (that can be considered as different data), meaning that this is exactly what GPUs have been designed to deal with. The most basic idea that comes to mind when one wants to parallelize an evolutionary algorithm is to run the evolution engine in a sequential way on some kind of master CPU (potentially the host computer CPU), and when a new generation of offsprings have been created, get them all to evaluate rapidly on a massively parallel computer. This approach has been examined in [12]. The proposed evolutionary algorithm reaches the speedup about 100. But, the bottleneck can be seen in slow data transfers from host memory to GPU and back, especially for small transactions [7].

Another way, how to parallelize GA is to move the whole algorithm on GPU. However, very few researchers so far have gone this way. They usually used Cg language [10,11] which does not allow access to some GPU features (i.e. manual thread and block control). A parallel genetic algorithm targeted to numerical optimization has been published in [13]. Unfortunately, this implementation reached only small speedups between 1.16 and 5.30 depending on population size. Several interesting publication can be also found in [9].

We would like to show, that the movement of entire genetic algorithm can be accomplished in a straightforward way. Moreover, excluding the system bus from the execution, much higher speedups could be achieved.

3 GPU-Based Genetic Algorithm

We have chosen CUDA (Compute Unified Device Architecture) [7] framework to implement our GA on GPU. This toolkit promises best achieved speedups on GPU so far and vast community of developers. CUDA can be performed on any nVidia graphics card from GeForce 8 generation on both Linux and Windows platform. Natural parallelism of computation on GPU is expressed by a few compiler directives added to the well known C programming language.

As mentioned earlier, nVidia GPUs consist of multiprocessors capable to perform tasks in parallel. Threads running in these units are very lightweight and can be synchronized using barriers so that data consistency is maintained. This can be done with very low impact on the performance in a multiprocessor, but not between multiprocessors. This limitation forces us to evolve islands either completely independent or perform migrations between them asynchronously.

The memory attached to graphics cards is divided into two levels — main memory and on-chip memory. Main memory has a big capacity (hundreds of MB) and holds a complete set of data as well as user programs. It also acts as an entry/output point during communication with CPU. Unfortunately, big capacity is outweighed with high latency. On the other hand, on-chip memory is very fast, but has very limited size. Apart from per-thread local registers, on-chip memory contains particularly useful per-multiprocessor shared segments. This 16KB array acts as a user managed L1 cache. The size of on-chip memory is a strongly limiting factor for designing efficient GA, but existing CUDA applications greatly benefit there.

In order to summarize earlier paragraphs, our primary concern during designing GA accelerated by GPU is to create its efficient mapping to CUDA software model with a special focus on massive parallelism, usage of shared memory within multiprocessors and avoiding the system bus bottleneck.

Fig. 1 shows the GA mapping to CUDA software model. We assume an island based GA with the migration along an unidirectional ring. Every individual is controlled by a single CUDA thread. The local populations are stored in shared on-chip memory on particular GPU multiprocessors (CUDA blocks). This ensures both computationally intensive execution and massive parallelism needed for GPU to reach its full potential. Our implementation also utilizes a uniform and Gaussian fast random number generators described in [2].

The proposed algorithm begins with the input population initialization on the CPU side. Then, chromosomes and GA parameters are transferred to the GPU main memory using the system bus. Next, the CUDA kernel performing genetic algorithm on GPU is launched. Depending on kernel parameters, the input population is distributed to several blocks (islands) of threads (individuals). All threads on each island read their chromosomes from the main memory to the fast shared (on-chip) memory within a multiprocessor. From this point, shared

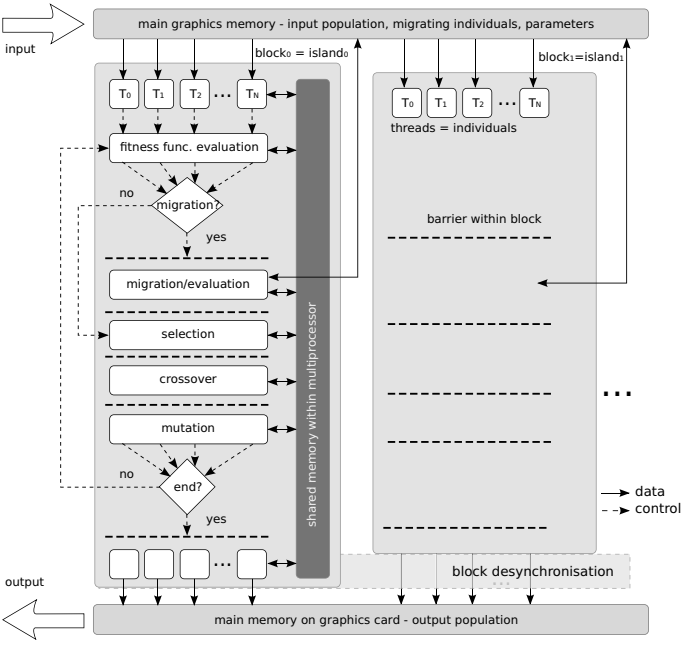


Fig. 1. Mapping of the genetic algorithm to CUDA software model

memories maintain local island populations. The process of evolution then proceeds for a certain number of generations in isolation, whereas, the islands as well as individuals are evolved on the graphics card in parallel. Each generation consists of fitness function evaluation and application of the selection, crossover and mutation (see section 3.1). The operators are separated by CUDA block barriers with zero overhead [7] so that data consistency is ensured.

In order to mix up suitable genetic material from isolated islands, the migration (see section 3.2) is employed. Because migration requires an inter-island communication, slower main memory has to be used for this process. Moreover, since CUDA blocks (islands) cannot be synchronized easily without a significant performance loss, the migration is done asynchronously (it does not wait for the neighbours to complete the predefined number of generations). This is unacceptable for common applications, where data consistency is required, but it works well for stochastic method like GA.

The algorithm iterates until a terminating condition is met (currently the maximum number of generations is set). Finally, every thread writes its evolved chromosome back to the main memory from where it will be read by CPU through the bus.

3.1 Implementation of Genetic Operators

Tournament selection and arithmetic crossover are tightly connected, as it is evident from Fig. 2. Limited shared memory is thereby used efficiently.

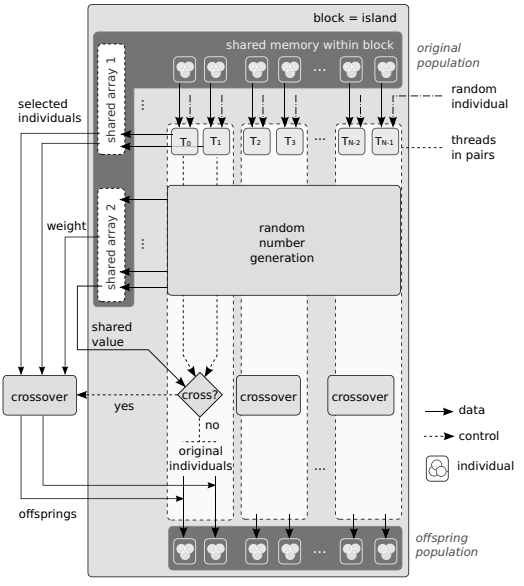


Fig. 2. Scheme of tournament selection with crossover

Threads (individuals) are grouped into pairs using shared variables and barriers so that crossover can be performed in parallel for the whole island population. First, each thread from a pair randomly selects one parent to crossover and it compares the fitness of its own individual using. Then, the index of the better one is written to the shared memory (shared array 1) to notify the other thread in the pair of a more suitable partner to crossover. Next, the parallel uniform random numbers generation is performed in the whole island and the results are written to the shared array 2. Pairs of threads then look up their common random number in this array and compare it with the crossover probability to decide whether perform the crossover or not. This task consumes the first half of the shared array 2. The second half is exploited during the arithmetic crossover as aggregation weights:

$$O_1 = a \cdot P_1 + (1 - a) \cdot P_2 \tag{1}$$

$$O_2 = (1 - a) \cdot P_1 + a \cdot P_2 \tag{2}$$

where O_1 and O_2 represent offsprings, P_1 and P_2 represent parents and a denotes the aggregation weight. This approach wastes the selection of individuals in the case that the crossover is not finally made, but it is from 0.1 to 2% faster (depending on island population size) due to SIMD GPU optimization.

The Gaussian mutation and fitness evaluation are performed in parallel for each thread (see Fig. 1). Finally, the newly generated offsprings replace the parents and the evolutionary cycle is repeated. The elitism is not ensured as crossover may destroy the best chromosome in the island population.

3.2 Migration between Islands

Migration is illustrated in Fig. 3. The islands are interconnected by an unidirectional ring thus an island can only accept individuals from one neighbour. The exchange is done asynchronously using the GPU main memory. The number of migrated individuals is determined by the parameter M. First, the local island population is sorted according to its fitness using Bitonic-Merge sort [1]. Next, M best individuals are written to a part of the main memory belonging to the left neighbour while M worst individuals are overwritten by migrants from a part of the main memory belonging to the right neighbour. Both sorting and migrations are done in parallel for all individuals.

The experimental results show that the migration can significantly improve the convergence to the best solutions in the search space, see table 2.

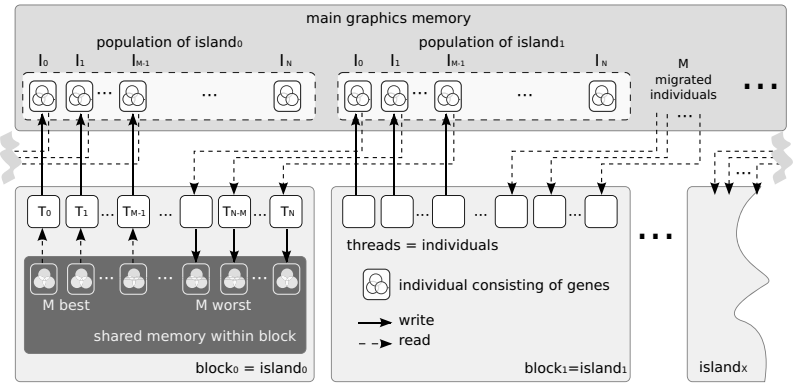


Fig. 3. Scheme of migrations between islands

4 Results

Achievable speedups and solution quality of the proposed GA were examined using Griewank's, Michalewicz's and Rosenbrock's artificial benchmark functions that are often used for GA analysis [15]. CPU version of GA is a single-thread program implemented using well known GAlib library [14].

4.1 Achieved Performance

The speedup of our implementation was investigated using intel Core i7 920 processor and several nVidia consumer-level graphics cards: 8800 GTX (16 multiprocessors / 128 cores), GTX 285 (30 multiprocessors / 240 cores) and GTX 260-SP216 (27 multiprocessors / 216 cores).

The speedup of GPU against CPU were investigated on two dimensional instances of the benchmark functions with mutation probability 5%, crossover rate 70%, no migration and terminating condition of 100 generations. Built-in CUDA timer functions were used to measure GPU kernel execution time [7].

We measured the performance using island population sizes from 2 to 256 individuals, and islands quantity from 1 to 1024. The performance unit was chosen to be population-size independent as IIGG=$\prod$(Island population size, number of Islands, Genotype length, number of Generations) per second. As we expected, CPU performance is almost constant while GPU performance highly varies according to the degree of parallelism (global population size). The performance of graphics cards is also greatly affected by compiler parameter `-use_fast_math` which causes usage of faster, but less precise mathematical functions. This parameter turned out to be profitable for GA because quality of results remains unaffected.

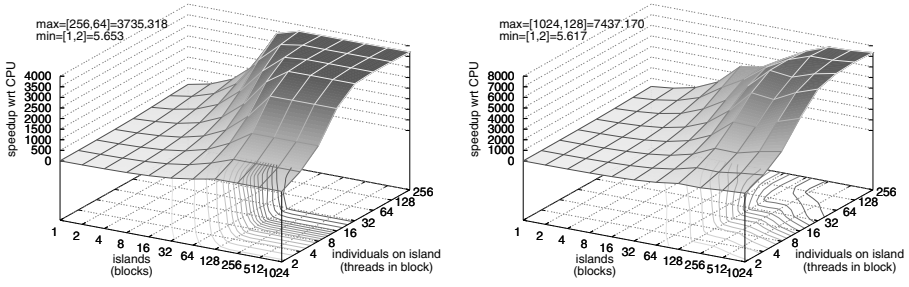
Table 1. Comparsion of CPU and GPU execution performance depending on island population size varying from 2 to 256 individuals. FastMath marks usage of -use_fast_math CUDA compiler parameter.

arch.	**fitness function**	(min – max) **IIGG**·10^6 **per second**		
	Rosenbrock	2.6 – 2.8		
CPU	Michalewicz	1.8 – 2.5		
	Griewank	2.5 – 2.8		
		8800GTX	**GTX260**	**GTX285**
	Rosenbrock	14.2 – 8877	12.0 – 13094	14.3 – 18669
GPU	Rosenbrock-FastMath	18.5 – 11914	15.5 – 17318	18.5 – 24288
	Michalewicz	6.9 – 5893	5.8 – 8850	7.0 – 12937
	Michalewicz-FastMath	11.7 – 9894	9.8 – 13692	11.6 – 19400
	Griewank	9.6 – 7108	8.0 – 10515	9.9 – 14496
	Griewank-FastMath	15.9 – 10507	13.3 – 15360	15.8 – 20920

Table 1 shows measured mean values of IIGG from 5 independent runs to reduce an influence of underlying operating system. GPUs always outperform CPU and truly excel at maximum speeds where measured performance is several thousand times better. Such a huge speedup promises solving problems that took hours to be completed in second-order time.

The results also show that two different generations of GPU, 8800 GTX and GTX285 offer the same performance for the low level of parallelism and differ significantly only at maximum speed. This clearly points out that the new technology is heading for massively-parallel computing rather than improving performance for single threaded applications (see Fig. 4).

The charts shown in Fig. 4 illustrate achieved speedups on two different GPUs against CPU, based on population sizes and the number of simulated island. The 8800 GTX graphic card saturates its computational resources from 256 islands and 32 individuals individuals per an island. The maximal speedup of 3735 against CPU is reached with 256 islands and 64 individuals per island. The GTX285 provides about twice better peak speedup, but it is necessary to provide much more computational work to it. The computational resources of



(a) 8800 GTX (b) GTX 285

Fig. 4. Speedup on Griewank's function depending on GA parameters and GPU

this GPU are not saturated even for 1024 islands and 128 individuals per island, where this GPU has attacked the speedup of 7437[1].

From the both charts in fig. 4 also flow that the usage of the GPU for a small number of island and/or a few individuals per island is not advantageous. The maximum performance is achieved for high number of simulated islands and increasing population size.

4.2 Solutions Quality

The proposed implementation of the tournament selection slightly differs from the original GAlib's one (see section 3.1). In order to ensure the same testing condition for the both CPU and GPU versions, the GAlib's selection were reimplemented. Arithmetic crossover and mutation were kept untouched as they have been defined in the same way.

Tests CPU and GPU1 were performed on artificial benchmark functions mentioned earlier on a single island (obviously with no migrations) with 32 individuals, 70% crossover probability, 5% mutation probability and elitism turned off. Each run was terminated after 100 generations of evolution and the best (lowest) fitness value was taken into consideration.

Test GPU2 was performed with the same GA parameters and benchmarks but with maximum GPU exploitation resulting from simulating 1024 populations (islands) in parallel. Additionally, migrations were performed every 10 generations with 3 individuals (approx. 10% of population).

To compare algorithms adaptability to rising problem complexity, varying number of genes (variables) was tested as well.

Table 2 shows mean value over 100 measured runs. Rosenbrock's and Griewank's functions have the value of the global optimum equal to 0. The value

[1] As it was mentioned earlier, a single threaded CPU implementation was tested. Benchmarked CPU Core i7 allows parallelisation to 4 physical cores + 4 virtual Hyper-Threading ones. Hence, ideally paralleled CPU version with 50% speed benefit from HT technology would change the maximum speedup from 7437 to approx. 1239 times. GAlib also computes variety of additional statistics.

Table 2. Comparison of the solutions quality

genes	mean best fitness								
	Rosenbrock			Michalewicz			Griewank		
	CPU	GPU1	GPU2	CPU	GPU1	GPU2	CPU	GPU1	GPU2
2	0.086	3.468	$7.57 \cdot 10^{-7}$	-1.022	-1.768	-1.801	0.0005	0.0020	$3.99 \cdot 10^{-12}$
3	1.897	4.996	0.447	-1.220	-2.336	-2.760	0.0051	0.0048	$1.06 \cdot 10^{-8}$
4	8.900	4.997	0.494	-1.459	-2.748	-3.696	0.0156	0.0188	$1.22 \cdot 10^{-7}$
5	22.112	17.332	2.042	-1.684	-3.184	-4.628	0.0246	0.0414	0.0001
6	48.450	56.045	4.313	-1.817	-3.654	-5.440	0.0408	0.0570	0.0005
7	83.455	42.509	6.903	-2.035	-3.646	-6.163	0.0479	0.0620	0.0012
8	128.710	155.233	9.257	-2.120	-3.805	-6.659	0.0650	0.1360	0.0027
9	167.329	131.737	12.045	-2.176	-4.830	-7.136	0.0749	0.1444	0.0042
10	233.364	184.370	15.379	-2.391	-5.009	-7.649	0.0805	0.1758	0.0058

of the global optimum of Michalewicz's function varies based on the number of genes – it is approximately linear interpolation from -1.8 (2 genes) to -9.66 (10 genes). Lower value means better solution for all tested functions.

Michalewicz's and Rosenbrock functions are optimised much better on GPU in most cases. On the contrary, Griwank's function for a single island (GPU1) reaches better solutions on CPU. This can be an effect of simplified random number generator which uses very limited amount of shared GPU memory. However, any negative effects are greatly outperformed by massive parallelism. Test GPU2 shows that fully utilized GPU can achieve far better results in the same number of iterations.

Overall, GPU1 results are better than CPU by approx. 20%. This shows that proposed GPU implementation of GA is able to optimise numerical functions.

5 Conclusions

Speedups up to seven thousand times higher clearly show that GPUs have proven their abilities for acceleration of genetic algorithms during optimization of simple numerical functions. The results also show that the proposed GPU implementation of GA can provide better results in the shorter time or produce better results in equal time.

The future work will be oriented to introducing more complex numerical optimization inspired by real-world problems. Moreover, we would like to compare parallel island-based GA running on CPU with the proposed GPU version.

Acknowledgement

This research has been carried out under the financial support of the research grants "Natural Computing on Unconventional Platforms", GP103/10/1517 (2010-2013) of Grant Agency of Czech Republic, and "Security-Oriented Research in Information Technology", MSM 0021630528 (2007-13).

References

1. Pharr, M., Fernando, R.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, Reading (2005)
2. Nguyen, H.: GPU gems 3. Addison-Wesley Professional, Reading (2007)
3. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press (1975)
4. Jiang, C., Snir, M.: Automatic Tuning Matrix Multiplication Performance on Graphics Hardware. In: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, pp. 185–196 (2005)
5. Galoppo, N., Govindaraju, N.K., Henson, M., Manocha, D.: LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In: Proceedings of the ACM/IEEE SC 2005 Conference, vol. 3 (2005)
6. Cant-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Dordrecht (2000)
7. NVIDIA, C.: Compute Unified Device Architecture Programming Guide. NVIDIA: Santa Clara, CA (2007)
8. Munshi, A.: The OpenCL specification version 1.0. Khronos OpenCL Working Group (2009)
9. Harris, M., Luebke, D.: GPGPU: General-purpose computation on graphics hardware. In: Proceedings of the International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2005 Courses, Los Angeles, California (2005)
10. Yu, Q., Chen, C., Pan, Z.: Parallel genetic algorithms on programmable graphics hardware. In: Wang, L., Chen, K., S. Ong, Y. (eds.) ICNC 2005. LNCS, vol. 3612, pp. 1051–1059. Springer, Heidelberg (2005)
11. Li, J.-M., Wang, X.-J., He, R.-S., Chi, Z.-X.: An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In: IFIP International Conference on Network and Parallel Computing Workshops, NPC Workshops, pp. 855–862 (2007)
12. Maitre, Q., Baumes, L.A., Lachiche, N., Corma, A., Collet, P.: Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation table of contents, Montreal, Qubec, Canada, pp. 1403–1410 (2009) ISBN 978-1-60558-325-9
13. Wong, M.L., Wong, T.T.: Implementation of Parallel Genetic Algorithms on Graphics Processing Units. In: Intelligent and Evolutionary Systems, vol. 187, pp. 197–216. Springer, Heidelberg (2009)
14. Matthew, W.: GAlib: A C++ Library of Genetic Algorithm Components. Massachusetts Institute of Technology (1996)
15. Pelikan, M., Sastry, K., Cantú-Paz, E.: Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications. Studies in Computational Intelligence. Springer, Heidelberg (2006)