Massively Parallel Genetic Algorithms

M. Schwehm

Universität Erlangen-Nürnberg, IMMD VII, Martensstr. 3, D-91058 Erlangen, Germany

The genetic algorithm is an iterative random search technique for nonlinear or combinatorial problems. In this contribution, first the development from the classical genetic algorithm (GA) via the parallel genetic algorithm (PGA) to the massively parallel genetic algorithm (MPGA) is described. Then experimental results with an implementation of the MPGA on the array processor MasPar MP-1 are displayed, which exemplify robustness and adaptive behavior of the algorithm. The observed properties of the MPGA are finally combined for an improved method of mapping load onto a massively parallel hardware.

1. INTRODUCTION

The line that separates massively parallel processing from parallel processing is floating. Perhaps we can agree that at least 1000 processors should be involved. Thinking about the future, millions of processors are to be utilized. What kind of algorithms will then be used? The explicit mapping of a problem onto available processors is a hard problem even for a small number of processors. In the context of massively parallel processing, algorithms which could map their load adaptively onto the available processors are desirable.

In this contribution genetic algorithms are investigated as a massively parallel programming technique. Genetic algorithms are iterative improvement techniques for optimization and adaptation inspired by the natural process of evolution. Main characteristic of the genetic algorithm compared to other search techniques is that they are population based. The algorithm operates in parallel on a whole population of solutions. This population and its operators do not have special requirements for its operation, so a data parallel version of this algorithm can easily be implemented on any parallel processor. For such an implementation, the population size is tied to the number of available processors. But increasing the size of the population does not necessarily increase convergence speed; the additional computing power might merely increase convergence security. It is the goal of this paper to demonstrate how a very large number of processors could nevertheless contribute to an overall speedup of the algorithm.

The next section gives a short introduction to genetic algorithms. Section 3 describes three different parallelization techniques, with emphasis on the massively parallel or cellular genetic algorithm. In section 4, some experiments are carried out to help understanding the internal behavior of the genetic algorithm. The results of these experiments suggest a new parallelization technique which is discussed in section 5.

2. GENETIC ALGORITHMS

Genetic algorithms belong to evolutionary algorithms, which imitate natural evolution to adapt their output to some given artificial environment. While other evolutionary algorithms are specialized to produce a certain type of output, e.g. parameter vectors (evolution strategies, [1]) or programming code (genetic programming, [2]), the genetic algorithm is not specialized. Its output is a bit string, which has to be interpreted and evaluated by the environment. The genetic algorithm operates blindly on the bit strings, since it does not know how any modification of the bit string influences the quality of the output. It is the environment which gives the genetic algorithm a feedback about the quality of the produced output, so that the genetic algorithm is forced to produce better and better output. If the environment is static, the genetic algorithm will thus solve an optimization problem, otherwise if the environment is dynamic, the genetic algorithm will permanently adapt to the changing environment and can thereby be used as adaptive control mechanism. See Bäck et al [3] for a commented list of references to applications of evolutionary algorithms.

Let us have a closer look at the operation steps of a genetic algorithm. First, a population of individuals is created, each represented by a random bit string. A bit string is interpreted as a solution of a given optimization problem by the environment, so that the population is just a subset of the search space. To improve the quality of the population, for each individual ($\otimes$, see figure 2) a mating partner is selected and their bit strings are recombined and mutated (reproduction) to produce some offspring. After evaluation, the offspring (eventually) replaces the parents. This process is repeated until to a stopping condition (see figure 1). The algorithm shows good results for many optimization problems, if the population size and the number of iterations (also called generations) is large, so that the implementation on a parallel computer is recommended.

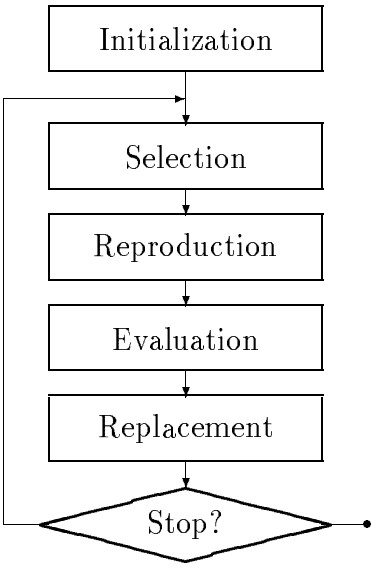

Figure 1. GA overview

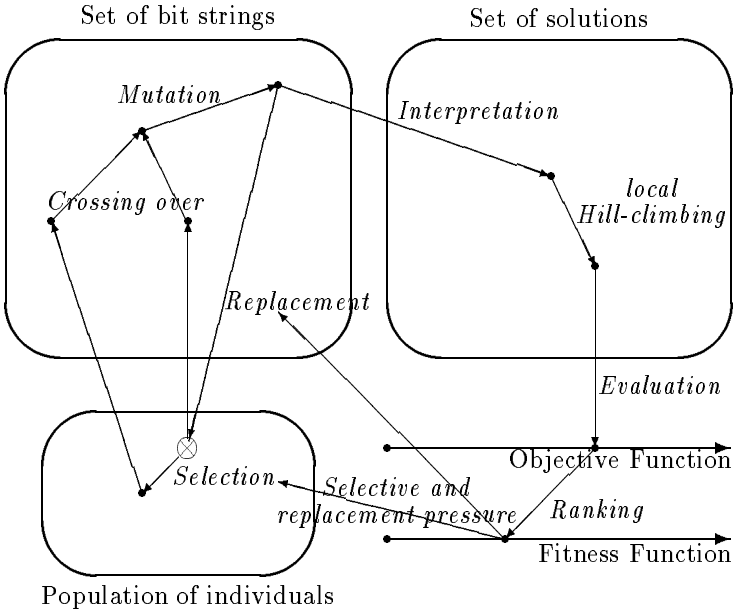

Figure 2. Genetic Operators

3. PARALLELIZATION OF GENETIC ALGORITHMS

Although the genetic algorithm imitates a process that is parallel in nature, the classical genetic algorithm is sequential, due to the lack of parallel hardware, when the algorithm was first defined by Holland [4] in 1975. Some of the internal structure of this sequential algorithm turns out to be hard to parallelize. For example, to imitate the concept of 'selective pressure', the individuals are allowed to reproduce with a probability proportional to their relative quality in the whole population (panmictic selection). This does not imitate the process in nature in detail, but it holds from a global point of view and can be implemented efficiently on a scalar processor, because in its plain memory each individual of the population can be accessed with the same effort. On a parallel hardware, the access to a single individual can be very expensive, because costs for communication and/or collisions have to be considered. An efficient parallel implementation has to avoid expensive storage accesses and should thus base the selective pressure on easy accessible (local) neighbors of each individual.

Meanwhile many parallel implementations of the genetic algorithm have been proposed (see e.g. [5]–[10]). Depending on how the population is mapped onto the available memory, three different implementations can be classified and different internal behavior can be observed:

GA The classical sequential genetic algorithm operates on a (single) population of individuals. Selection and replacement of individuals is due to global criteria, so the selective pressure for each individual is proportional to its fitness relative to the whole population. This method has the risk of premature convergence, if a relatively good solution spreads —due to the global selective pressure— too fast over the whole population. The population can be trapped in a local optimum while there is not enough time for other individuals to search the remaining regions of the search space. Parallelization of this algorithm without modification could at best be implemented on a shared memory multiprocessor.

PGA The parallel GA subdivides the population into several subpopulations and maps one or several of them onto each of the available processors. Each subpopulation is optimized independently, while in regular intervals the best individuals of each subpopulation are broadcast to neighboring subpopulations (migration). Selective pressure is now mainly inside a subpopulation and only to a lower rate between them. This variant of the GA reduces the risk of premature convergence, since a suboptimal solution will only spread over one subpopulation, while the other subpopulations are not disturbed to search other areas of the search space. This algorithm is intended for distributed memory multiprocessors where each processor runs its own GA, while migration is realized via message passing [5][6].

MPGA For the massively parallel genetic algorithm (MPGA), each individual is mapped onto its own processor. Selection and replacement (in contrast to the scalar GA) use only local criteria. Thus the selective pressure for each individual is proportional to its quality relative to its local neighbors. This local neighborhood is determined by the underlying interconnection topology, in most cases a grid. A suboptimal solution here will spread over its local neighborhood, and a region with similar

individuals evolves. The suboptimal individual needs several generations to spread its information over the whole population, while individuals in other regions of the processor topology have a chance to prevail. This algorithm was designed for array processors, where the large amount of communication can be realized efficiently.

PGA and MPGA both introduce some spatial structure to the population, to improve the efficiency of the parallel implementation. Moreover it has been reported several times, that the parallel versions of the algorithm found better solutions or found the optimum faster than the sequential one [5,8,10].

4. ADAPTIVE BEHAVIOR OF GENETIC ALGORITHMS

The experiments in this section were carried out using an implementation of the MPGA on the MasPar MP-1 as described in [7]. The experiments shed some light onto the the way, genetic algorithms work internally. The problem solved in these examples is the multilevel graph partitioning problem as described in [8]. This is a difficult test problem with many local optima and two optimal solutions. Each of the following experiments was repeated 50 times and the figures 3–6 display the average number of generations needed to find the optimal solution.

4.1. Adaptation to interconnection topologies

The spatial structure of the population introduced by the massively parallel implementation reflects in many cases the interconnection topology of the hardware used. Examples are [9] with an implementation on the CM-2 (hypercube) and [10] for the DAP (2-dim torus). How does a particular interconnection topology influence the performance of the genetic algorithm? Figure 3 shows the average number of generations needed to solve the above test problem while emulating different interconnection topologies. Torus and X-Net (a torus with additionally the diagonal neighbors) seem to outperform the remaining topologies. This is only because other parameters of the genetic algorithm, which also influence the selective pressure, like the selection strategy and the number of neighbors visited, were set optimal for the X-net topology, which is the standard topology on the MasPar MP-1.

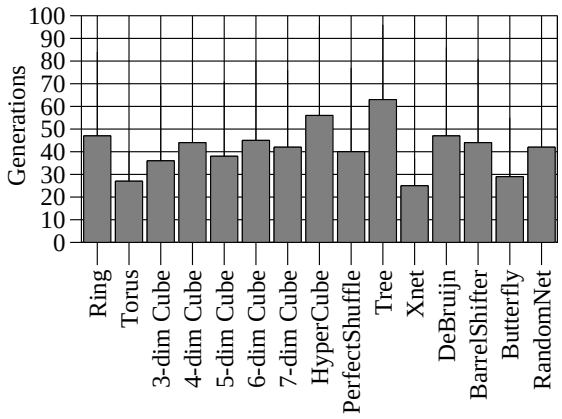

Figure 3. Performance of some topologies

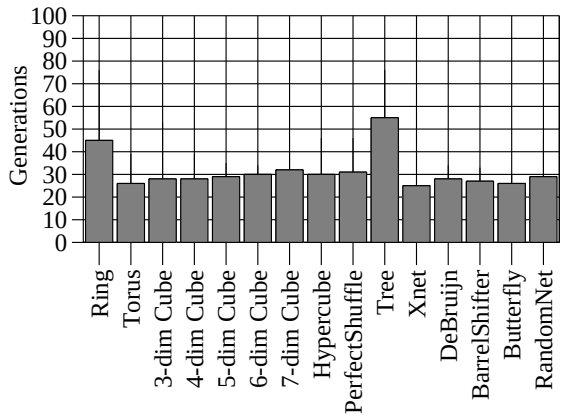

Figure 4. Ditto with optimized parameters

In figure 4 the experiment is repeated with individually optimized selection strategy. The differences in the performance of the topologies decreases, only topologies with a very long diameter (ring) or with a bottleneck (tree) perform bad again. Topologies with a very short diameter (hypercube) perform slightly worse than those with mediocre diameter. Even a random interconnection scheme with four neighbors per processing element showed acceptable performance. For further details refer to [11].

This observation can be explained as follows: The only step that uses communication between processors is the selection process. For each individual a mating partner is searched in the local neighborhood. An efficient selection strategy is the random walk strategy. In each generation, a new random path from neighbor to neighbor is generated. The mating partner will now be chosen as one of the individuals found along this path. The probability to become selected as mating partner for any given individual decreases with the distance. The 'random walk length' can compensate the influence of a different interconnection topology.

4.2. Robustness against noise

A second observation is that the genetic algorithm is very robust against failures. For figure 5, a randomly selected subset of the processors was forced to return constantly a very bad fitness to the neighboring processors. Besides some fluctuations, which is a side effect of the random nature of the algorithm, the performance decreases only slightly with the number of faulty processors. This is not surprising —considering the results of the previous section— since a faulty processor has a similar effect as randomly changing the topology plus reducing the population size.

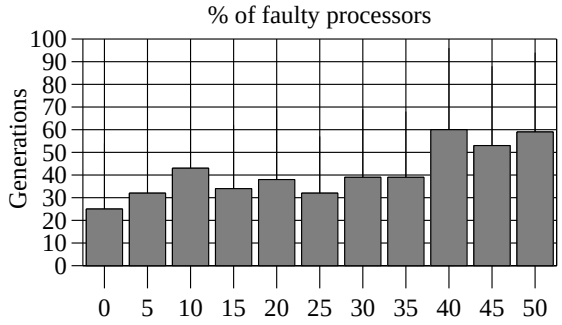

Figure 5. Performance with faulty procs.

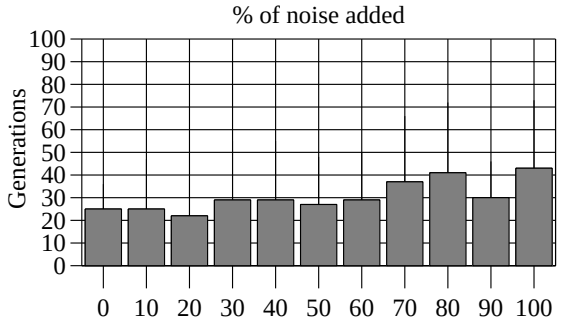

Figure 6. Performance with noise added

The genetic algorithm also works, if noise is added to the output of the objective function, as displayed in figure 6. This observation can be explained by the fact that the objective function only determines the *probability* of an individual to reproduce. So not always the best individual is selected to reproduce: to avoid premature convergence (see section 3) other individuals also get a chance. This slows down convergence, but also avoids convergence to local optima. Adding noise increases this effect and could easily be compensated by increasing internal selective pressure of the genetic algorithm.

4.3. Implicit load balancing

The advantage of the PGA —to subdivide the population into subpopulations which now can search different areas of the search space independently— seems to be lost for the MPGA. But the local interconnection topology isolates the individuals by distance and thus also makes the search on distant processors independent from each other.

Figure 7 shows a sample run of the genetic algorithm. For every third generation a hamming map is displayed. The hamming maps display for each individual (represented as string of 64 bits) of a 32×32 grid-connected population the (hamming-) distance to its local neighbors. White denotes minimal, black maximal distance. The multilevel graph partitioning problem has two optimal solutions, which are coded by complementary bit strings. This allows to visualize processors searching in the same or the complementary area of the search space. While for the PGA the number of subpopulations has to be set explicitly (and will in most cases be set to the number of available processors), the MPGA starts with uniformly distributed individuals and then rapidly accumulates more and more processors to search in interesting regions. The regions grow and vanish according to the needs of the to-be-solved problem.

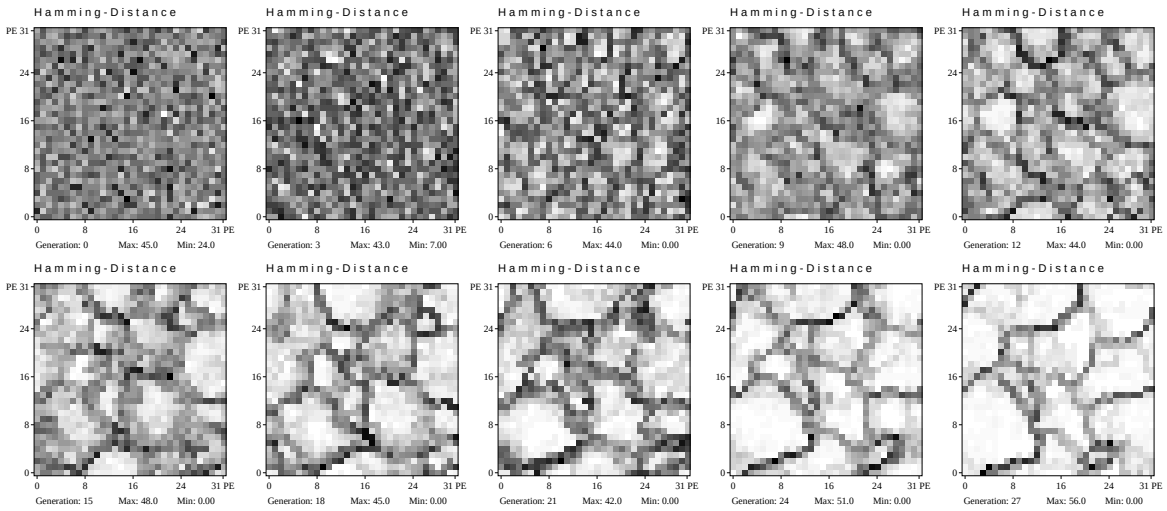

Figure 7. Hamming maps of a sample run of the genetic algorithm

The regions observed in this example are not always desirable, since inside the regions the genetic algorithm looses its driving force: the crossover does not have any effect if applied to almost identical bit strings. The example was run with the tournament selection strategy, which searches for a mating partner only in the local neighborhood. Better optimization results were obtained with the above mentioned random walk strategy, which might look at the same number of individuals, but they can have a larger distance. With this strategy, no regions are observed, but the point of this section remains true: The genetic algorithm dynamically accumulates more and more processing power to 'interesting' areas of the search space.

5. RESULTS AND DISCUSSION

The two observations from section 4.2 and 4.3 contain a hint that the genetic algorithm as implemented is not very efficient. If the algorithm converges even if noise has been added this means that the time needed for the *exact* evaluation of the objective function is wasted. In the above experiment, additional computing time was needed to add the noise. But if the exact evaluation of the objective function is replaced by a quick approximative algorithm, the total computing time could be reduced, if the gain achieved by reducing computing time for each generation exceeds the loss by slower convergence.

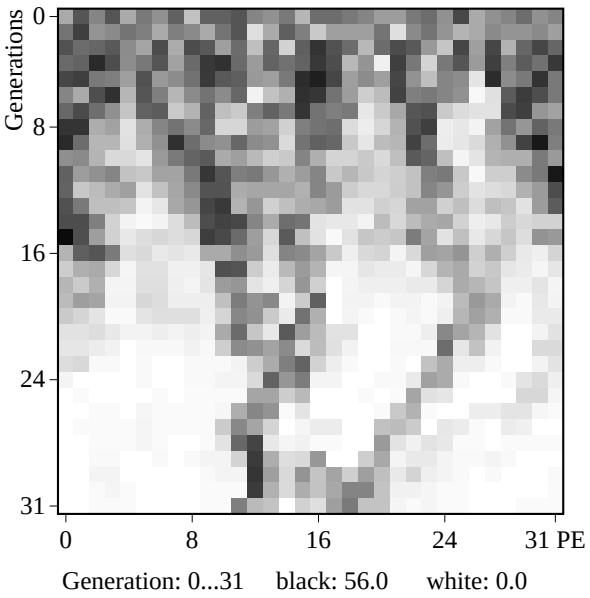

Figure 8. Time-space hamming map

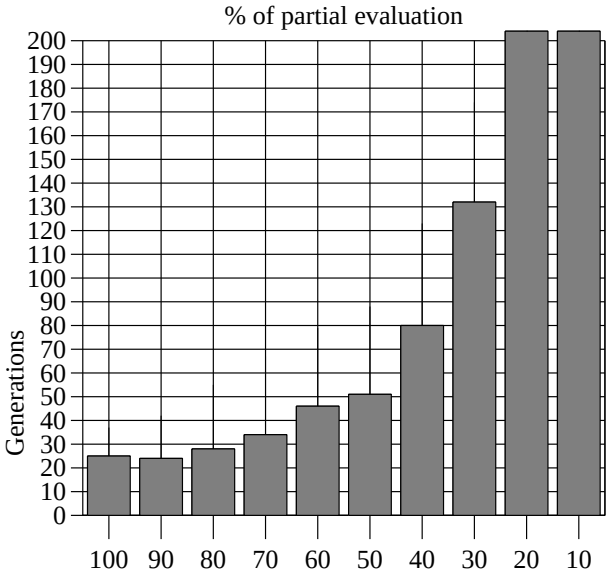

Figure 9. Performance with lazy evaluation

Replacing the objective function by a quick approximation also makes sense if we look at the hamming maps of section 4.3. Figure 8 shows a cut trough the first 32 generations of the sample run of figure 7. The top row displays one row of the hamming map after initialization. The differences between the individuals are large, and so are the differences of their fitness. Instead of wasting time by exact evaluation of the objective function, a quick approximation could do the same job. Later in the run, regions have evolved, and the corresponding region of processors evaluates —generation for generation— almost identical solutions. A whole conic section in the time-space domain of the available computing power is used for almost the same job. If this computing power could somehow be coordinated! If the noise added to the objective function by the quick approximation is randomly distributed, it would make sense to reevaluate the objective function over and over again. With growing regions and each additional generation, the individual evaluations of the objective function sum up to a more and more exact global value.

In figure 9 this approach has been applied to the multilevel graph partitioning problem. Only a random subset of the edges of the multilevel graph is tested in each generation,

resulting in a corresponding reduction of computing time and corresponding noise. Up to a partial evaluation of 50%, this gain is not exhausted by the additional generations needed to converge. So the computation is merely distributed over a larger number of faster generations. Better results can be achieved, if the noise introduced by the partial evaluation grows less than linear, as it is the case in many practical applications e.g. if the objective function is itself an approximation algorithm with exponential convergence rate.

6. CONCLUSIONS

Future generation computers will provide a huge number of processors. The massively parallel (= data parallel) implementation of the genetic algorithm can utilize this additional computing power only to compute larger and larger populations. This increases convergence security, but not —as desired— convergence speed. The partial evaluation of the objective function as described in this paper offers a method to distribute its computation adaptively over several generations and processors.

Moreover the proposed method brings the genetic algorithm closer to the situation in nature, where species have evolved more properties, than can be evaluated during the lifespan of a single individual.

REFERENCES

1. T. Bäck et al, A Survey of Evolution Strategies, in: [12] (1991) pp. 2–9
2. J.R. Koza, Genetic Programming: On the Programming of Computers by means of Natural Selection, MIT Press Cambridge/MA (1992)
3. T. Bäck and F. Hoffmeister, and H.-P. Schwefel, Applications of Evolutionary Algorithms, Tech. Report No. SYS-2/92, University of Dortmund, Germany
4. J.H. Holland, Adaptation in Natural and Artificial Systems, (1975), 2nd edn. (1992), MIT Press Cambridge/MA
5. T. Starkweather and D. Whitley and K. Mathias, Optimization using Distributed Genetic Algorithms, in: H.-P. Schwefel and R. Männer, Parallel Problem Solving from Nature, LNCS 496, Springer-Verlag Berlin (1990) pp. 176–185
6. H. Mühlenbein and M. Schomisch and J. Born, The Parallel Genetic Algorithm as Function Optimizer, Parallel Computing, Vol. 17 (1991) pp 498–516
7. M. Schwehm, A Massively Parallel Genetic Algorithm on the MasPar MP-1, in: R.F. Albrecht et al, Proc. Int. Conf. ANNGA '93, Springer-Verlag Wien (1993) pp. 502–507
8. R.J.Collins and D.R Jefferson, Selection in Massively Parallel Genetic Algorithms, in: [12] (1991) pp. 249–256
9. R. Tanese, Parallel Genetic Algorithm for a Hypercube, in: J. Grefenstette, Proc. 2nd Int. Conf. Genetic Algorithms, Lawrence Erlbaum, (1987) pp. 177–183
10. P. Spiessens and B. Manderick, A Massively Parallel Genetic Algorithm: Implementation and First Analysis, in [12] (1991) pp. 279–285
11. M. Schwehm, Implementation of genetic Algorithms on Various Interconnection Networks, in: M. Valero et al, Proc. Int. Conf PaCTA'92: IOS Press (1992) pp. 195–203
12. R.K. Belew and L.B. Booker, Proc. Int. Conf. Genetic Algorithms, San Diego, Morgan Kaufmann Publishers (1991)