NERV: A Parallel Processor for Standard Genetic Algorithms*

R. Hauser, R. Männer

Lehrstuhl für Informatik V
Universität Mannheim
D–68131 Mannheim

M. Makhaniok

Institute for Engineering Cybernetics
Belarus Academy of Sciences
220012 Minsk, Rep. Belarus

Abstract

This paper describes the implementation of a standard genetic algorithm (GA) on the MIMD multiprocessor system NERV. It discusses the special features of the NERV hardware which can be utilized for an efficient implementation of a GA without changing the structure of the algorithm.

1 Introduction

In recent years genetic algorithms (GAs) [1,2] have found considerable interest as a means of solving optimization problems. They do this by exploiting ideas which are drawn from natural evolution. The basic idea is to first choose a representation for a solution to a given optimization problem. In the following we will assume that the representation will be in the form of a bit string although other representations are possible. Then several operators are iteratively applied to a set of solutions. This improves their quality. The terminology of GAs is mostly drawn from biology, so the set of solutions is called the population, the quality of a given solution the fitness, one solution is called an individual etc. The basic operators of GA are modeled after their natural counterpart and consist of selection, crossover and mutation.

1.1 Selection

The fitness of each individual in the population is evaluated. The fitness of each individual relative to the mean value of all other individuals gives the probability with which this individual is reproduced in the next generation. Therefore the frequency h_i of an individual in the next generation is given by

$$h_i \propto \frac{f_i}{\bar{f}} \tag{1}$$

where f_i is the fitness of individual i and $\bar{f}$ the average over all fitness values. The effect of this procedure is that individuals with a higher–than–average fitness become more frequent in the population. Individuals with worse fitness will be reproduced with a smaller probability and therefore vanish from the population.

1.2 Crossover

The crossover operator takes two individuals from the population and combines them to a new one. The most general form is uniform crossover from which the so called one–point crossover and two–point crossover can be derived. First two individuals are selected. The strategy for this selection can again vary. A popular one is to select the first individual according to its fitness and the second one by random. Then a crossover mask M_j, $j = 1, \ldots, L$, where L is the length of the chromosome, is generated randomly. A new individual is generated which takes its value at position j from the first individual if $M_j = 1$ and from the second one if $M_j = 0$. One gets, e.g., the usual one–point crossover operator if $M_j = 1$ for $j = 1, \ldots, k$, and $M_j = 0$ for $j = k + 1, \ldots, L$. The crossover operator is applied with probability P_C. The old individual is simply copied to the new population if no crossover happens. The reasoning behind this operator is that it might happen that two individuals each have found an optimum in different subspaces and the combination of these solutions gives also a good solution in the combined subspaces.

1.3 Mutation

Each bit of an individual is changed (e.g. inverted) with probability P_M. This probability is a parameter of the algorithm. One important motivation for this

*This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grants Ma 1150/8–1 and 436 WER 113–1–3.

operator is the case where all individual of a population have bit k set to zero. Neither selection nor crossover is able to change to this bit to the other value. If the optimal solution happens to lie in the subspace of the configuration space where bit $k = 1$ then this optimum can never be reached.

All three steps above are iterated for a given number of generations (or until one can no longer expect a better solution).

2 Parallel Genetic Algorithms

It has long been noted that genetic algorithms are well suited for parallel execution. Several parallel implementations of GAs have been demonstrated on a variety of multiprocessor systems. This includes MIMD machines with global shared memory [11] as well as message passing systems like transputers [3,4] and hypercubes [8] as well as SIMD architectures [6,7] like the Connection Machine.

It is easy to see that the following steps in the algorithm can be trivially parallelized:

1. *Evaluation of fitness function* The fitness of each individual can be computed independently from all others. This could give a linear speedup with the number of processing elements. The maximum speedup can be achieved if the number of processing elements is equal to the number of individuals in the population. It might be possible, of course, that e.g. the fitness evaluation of each individual can itself be parallelized.

2. *Crossover* If we choose to generate each individual of the next generation by applying the crossover operator, we can do this operation in parallel for each new individual. The alternative would be to apply crossover and to put the resulting individual in the existing population where it replaces e.g. an individual with a bad fitness. This would obviously introduce race conditions when applied concurrently to several individuals. One processing element may start with an old individual which is replaced by a new one during processing. For these reasons we will use the first variant. Again a linear speedup with the number of processing elements can be achieved as long as the number of processing elements is less than or equal to the number of individuals.

3. *Mutation* The mutation operation can be applied to each bit of each individual independently. Besides from the bit value the only information needed is the global parameter P_M.

It should be noted that it is usually not possible to gain a larger speedup for steps 1) and 2) because of data dependencies between the different steps of the algorithm. This can be seen e.g. for step 2: If the crossover operation selects one of the parents it does this according to its relative fitness. However this can only be done if the fitness values of all other individuals are already computed so that the mean value is available. Therefore the three steps will in general be done one after each other.

Up to now we did not assume any concrete implementation for our parallel processing system. We did not mention how fine grained our parallel processing can be and how the memory system is organized. Especially the last point can have an enormous impact on the resulting performance of the system. The problem is that we always assumed that all data are available in a global shared memory. The crossover procedure e.g. takes two arbitrary individuals to create a new one. Therefore it must have access to the whole population. If we have a multiprocessor system with local memory only, we must first transfer the whole population to all processing elements before we can proceed. In the following we will point out what kind of data each processing element must access to perform the different steps of the algorithm:

1. *Fitness evaluation*: Each processing element must have access only to those individuals whose fitness it is going to compute. In the optimal case (number of processing elements = number of individuals) this means one individual. However the result of this computation is needed by all other processing elements since it is used in computing the mean value of all function evaluations which is needed in step 2.

2. *Crossover*: Each processing element which creates a new individual must have access to all other individuals since each one may be selected as a parent. Furthermore to make this selection the procedure needs all fitness values from step 1.

3. *Mutation*: As in step 1 each processing element need only the individual(s) it deals with. As mentioned above the parallelization could be even more fine grained as in steps 1 and 2, in which case each processing element would need only one bit of each individual. This could usually only be achieved by a SIMD style machine.

Unfortunately a multiprocessor system with a globally shared memory using e.g. a bus system is usually restricted to only a few processing elements. Otherwise the bus will become a serious bottleneck. Other interconnection networks like a crossbar switch do not scale up very well with the number of processing elements. Therefore many systems with a large number of processors use only local memory instead and provide communication e.g. via message passing. Algorithms where the processing elements are only loosely coupled perform well on these machines and the number of processing elements can often be scaled to several hundreds of processors.

The drawback of these systems when applied to genetic algorithms is that some of the data must be passed to all processors. This data transfer may take an enormous amount of time, especially if there is no direct connection between two arbitrary processors but the data must be passed on by several intermediate nodes. This is the case e.g. for a transputer system where a processor may be connected to only four neighboring processing elements.

This seems to be the reason that many implementors of parallel genetic algorithms have decided to change the standard algorithm in several ways.

One popular approach is the partitioning of the population into several subpopulations [5,9]. The evolution of each subpopulation is handled independently from each other. From time to time there is however some interchange of genetic material between different subpopulations. Sometimes a topology is introduced on the population, so that individuals can only interact with nearby chromosomes in their neighborhood [3,6,10,12,13]. All these methods obviously reduce the coupling between different processing elements. Therefore an efficient implementation on multiprocessor systems with local memory is possible.

Some authors argue that their changes to the original algorithm actually improve the performance. E.g. the splitting in different subpopulation allows each subpopulation to evolve to a different suboptimum without interference from other subpopulations. The danger that the whole population evolves into a suboptimal solution is greatly reduced. The combination of two different subpopulation with good suboptimal solutions may result in further improvement.

The restriction to use only individuals taken from a given neighborhood can be justified by biological reasons. Here an individual is obviously not able to choose an arbitrary individual from the whole population. Furthermore the separation of subpopulations is often considered as an essential point for the evolution of new species. As long as there is a constant flow of genetic material the two populations will not evolve in two different directions.

Despite these arguments we consider it as a drawback that not the original standard GA could be efficiently implemented. The reason is that a genetic algorithm is often a computational intensive task. It often depends critically on the given parameters used for the simulation (e.g. P_M and P_C). There are some theoretical results about how to choose these parameters or the representation of a given problem, but most of them deal with the standard GA only. Even then one often has to try several possibilities to adjust the parameters optimally.

Therefore it is desirable that the standard GA can be parallelized and simulated efficiently. If one changes the algorithm itself in the process of parallelization the theoretical assumptions will usually no longer apply. Such a simulation e.g. cannot be directly used to support some theoretical results. Of course this does not speak against the changed algorithms, it simply argues that for a fast simulation system it should be possible to parallelize the standard version of the algorithms so that the results can be directly compared to a single processor version.

In the following we present some results of such a parallelization on the multiprocessor system NERV. We will show that only a small number of properties are required to get an efficient parallel program which implements the standard GA.

3 The NERV multiprocessor system

The NERV multiprocessor [14] is a system which has been originally designed for the efficient simulation of neural networks. The general layout can be seen in Fig. 1. It is based on a standard VMEbus system [15] which has been extended to support several special functions. Each processing elements consists of a MC68020 processor with static local memory (currently 512 kB). Each VME board contains several processor boards. The NERV system can therefore be considered as a MIMD machine, since each processor may run a different program in its local memory. However usually the system is run in a SIMD style mode, which means that the same program is downloaded to each processing element, while the data to be processed are distributed among the boards.

The whole multiprocessor is connected to a workstation via a parallel interface. Programs can be transparently downloaded and run from each workstation

which has a network connection to the special workstation which is physically connected to the NERV system. The user usually does not recognize the underlying network protocol which handles all data transfers. Program development is done on the user's local workstation. The GNU C and C++ compilers [16] are used for cross development.

At first sight it might seem that this setup includes two disadvantages we mentioned above: There is no global memory while at the same time the interconnection network used for communication is based on a bus system which does not scale up very well with the number of processing elements. However we will show that this is not the case if we make use of some special features of the NERV hardware.

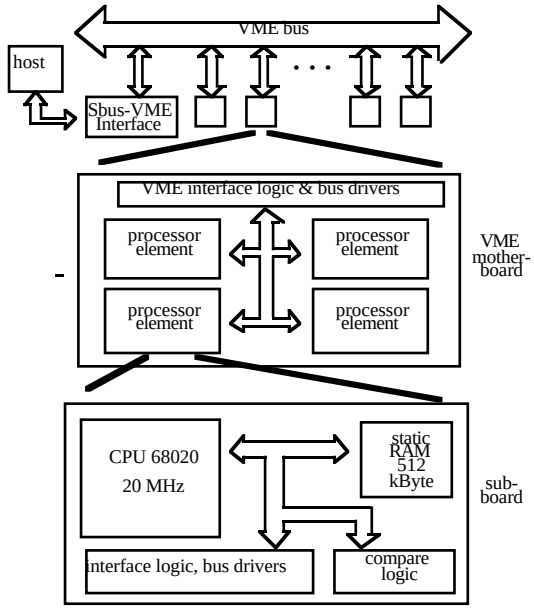

Figure 1: General layout of the NERV multiprocessor system

This following extensions to the VMEbus have been implemented in the NERV system:

The first one is a broadcast facility which is not part of the standard VME protocol. It allows each processor to access the memory of all other processor boards with a single write cycle. Therefore it is easy to transfer information to all processors simultaneously. From the programmers point of view the address space of each processor is divided into several regions where one is dedicated to local memory, one to broadcast address space and the others to local registers on board. A broadcast transfer can be simply initiated by writing to the broadcast address region. This will result in a data transfer to the memory of all other processors (including the local one). On the other hand a read from the broadcast region will simply return the data in the local memory of the processor. The software is usually written in C or C++ where a programmer might take advantage of this property in the following way:

Assume that you have a data structure, e.g. an array, which is most of the time only read. However the values of the array must be the same and consistent in all processing elements even if an update of an element occurs. Then a programmer might take the address of the array and pass it to a special function *mk_global()* which modifies the address in such a way that it now is part of the broadcast address region. The pointer returned by this function can now be used to access the array. Whenever we read a value from the array we simply get the local value. No other processors or the bus are involved. However if we write into any element of this array a broadcast will automatically be initiated since the address is part of the broadcast region. Therefore this element will be updated on all other processors. Note however that there is no explicit synchronization between the processors. If two processors update the same element, the last one will win. This will not happen if e.g. each processor is only allowed to update a certain range of array elements.

Here is an example C fragment:

```
int  vector[100];
int *p;
int  a;

p = mk_global(vector); /* p is now pointer into
                          the broadcast address
                          region */
a = p[10];            /* this is a read from
                          local memory */
p[50] = 5;            /* this is an implicit
                          broadcast transfer
                          vector[50] on all
                          processors will now
                          contain the value 5 */
```

If we can restrict our communication to broadcast transfers only, we have a very efficient way for updating global information, although the information itself will be duplicated on each processor's local memory. The last point solves the bottleneck problem usually associated with a single global shared memory. The first one reduces the communication time between the processing elements. Note that a broadcast transfer facility cannot be implemented with such efficiency in a system without a global bus.

A second extension on the VMEbus includes a hardware synchronization of all processing elements. This is necessary at several points in the algorithm. E.g. we must be sure that the fitness of all individuals has been computed before we proceed with crossover. If this would be done in software it would require at least one bus transfer for each processing element. The NERV system uses a special open–collector line on the bus which can be set via a local register by each processor. After all processors have set this register the bus line will become high and signal a successful synchronization. Processors which reached the synchronization point first will simply poll this line until the last one is finished. Therefore additional processors will have no influence on the overall performance of the synchronization process. The programmer uses a special procedure *synchronize()* which will only return after all processors have set their line. So each synchronization point in the program consists of a call of this procedure.

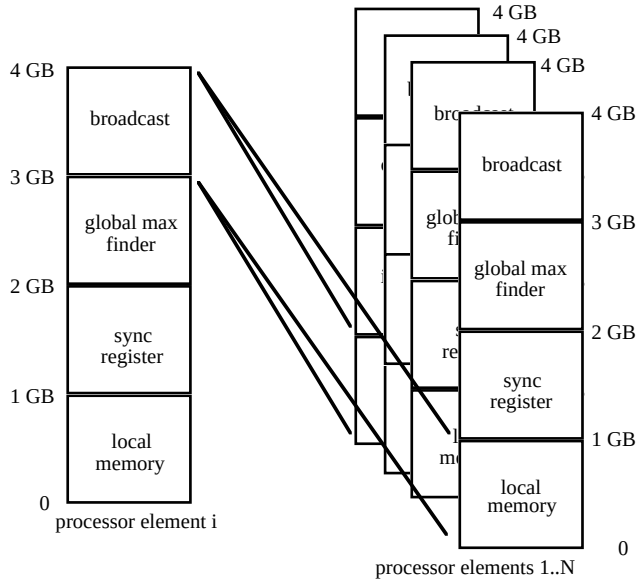

Figure 2: Address space of a NERV processor module

A third hardware feature of the NERV system cannot be directly exploited by the genetic algorithm. It implements a distributed global maximum finder for small numbers. The realization is similar to the arbitration schemes used in several multiprocessor bus systems like Futurebus+. Each processor puts a number on some special open–collector bus lines and the hardware selects the largest one. Since it is not used in the GA we will not discuss it further.

4 Implementation of the Parallel Genetic Algorithm

The previous sections suggest the following setup for the algorithm.

The same program is loaded in each processor. Every processor has a copy of all individuals in his local memory. The population is initialized at program start by a single processor (usually processor number 1) and transferred to all others by broadcast. The current population and the population of the next generation are accessed by two pointers which have been prepared by *mk_global()* so that they both point into the broadcast region. The same holds for an array which contains the fitness values of all individuals. After each generation the two population pointers are simply exchanged. Let N be the number of processing elements in the system. The general strategy will be to distribute the computational load equally among all processing elements by assigning $\frac{P}{N}$ individuals to each processor.

```
Chromosome pop1[POP_SIZE],pop2[POP_SIZE];
Chromosome *population,*newPopulation;

population = mk_global(pop1);
newPopulation  = mk_global(pop2);
```

The parallelization of each GA operator is now straightforward.

4.1 Fitness evaluation

Each processor evaluates the fitness of the individuals it has been assigned. No interaction is required between different processors. The fitness values are simply written into the mentioned array which will automatically initiate a broadcast. Since each processor is responsible for another set of individuals no overlap will occur. After this step is finished P broadcast transfers have occurred and the fitness array on each processor contains the up–to–date values.

```
int fitness_values[POP_SIZE];
int *fitness;

fitness = mk_global(fitness_values);

for ( i = "first individual"; i <= "last
                          individual", i++)
  fitness[i] = eval(i);

synchronize();
```

Note that the evaluation function uses only the local copy of the population. The access to *fitness[i]*

is the (implicit) broadcast. After the *synchronize()* call the processors can continue, e.g. by computing the mean value of all function evaluations. The computation of the first and last individual for each processor is simple if $P \bmod N = 0$. Otherwise it may happen that some processors have been assigned more chromosomes than the rest. Since these details are not important for the algorithm itself they have been omitted.

4.2 Crossover

As already mentioned we decided to make the next generation by looping over all individuals of the new population and either copying an individual from the old one or create a new one by crossover from two parents. Again each processor will be responsible for a part of the population. In the case of one–point crossover, the general algorithm looks like this:

```
for ( i =  "first individual"; i <= "last
                              individual"; i++) {

  offspring = &newPopulation[i];

  parent1 = select();
  parent2 = random_select();

  if (random(CROSSOVER_PROB) < CROSSOVER_PROB) {
    k = random(CHROM_LENGTH);

    for(j = 0; j < k; j++)
      offspring[j] = parent1[j];

    for(j = k; j < CHROM_LENGTH; j++)
       offspring[j] = parent2[j];

  } else /* copy individual */

    for(j = 0; j < CHROM_LENGTH; j++)
      offspring[j] = parent1[j];
}
synchronize();
```

The function *select()* selects an individual according to its relative fitness (e.g. using a roulette wheel algorithm), *random_select()* selects an individual by random. Each of these functions uses only local information. *offspring* is a pointer to the new individual. Since it gets its value from the *newPopulation* pointer it will also point into the broadcast region. This means that each access to *offspring* in the inner for–loops will be a broadcast. Again each element in the *newPopulation* array will only be written by exactly one processor, so no conflicts will arise.

After this step $P \cdot L$ elements will have been broadcasted (assuming that we encode e.g. each bit in a separate character) and each processing element will have a complete copy of the new population.

4.3 Mutation

The mutation operator is parallelized in the same fashion as the other operators. Again each processor handles $\frac{P}{N}$ chromosomes and broadcasts the results.

```
for(i = "first individual"; i <= "last
                              individual"; i++) {

  individual = &newPopulation[i];

  for(j = 0; j < CHROM_LENGTH; j++)
    if (random(MUTATE_PROB) < MUTATE_PROB)
      individual[j] = !individual[j];
}
synchronize();
```

Each bit changed by mutation must again be broadcasted to all other processors. This is done by the assignment to *individual[j]*. Note that the right hand side of this assignment will only access local memory since it is a read access. After the synchronization the pointer to the old and the new population can be exchanged and the next generation can be computed.

The program will transfer P fitness values (from step 1) and $P \cdot L$ bits for the new population (from step 2) over the common bus. In addition it must transfer the bits which are changed during mutation which may vary in each generation. This is all communication which will occur. All other values are usually fetched from local memory. A broadcast facility is the most efficient way to implement this since it does not depend on the number of processors. If we increase the number of processing elements we will decrease the time needed for each step while the communication overhead will stay constant.

From the consideration above we should expect a linear speedup with the number of processing elements. However this is not entirely true since if several processors want to broadcast at the same time only one request can be satisfied. This is due to the one–at–a–time property of a single bus. In practice this will lead to a serialization of the program. However the time for a single transfer is usually very small compared to the rest of the computations required, e.g. the fitness evaluation or the selection of a parent chromosome.

One appealing property of this implementation is that it behaves exactly the same if it is run on a single processor or a multiprocessor system (if we assume that our random number generators are initialized appropriately). The synchronization points take care that no data will be used by any processor before it is generated by another one. In fact for most parts the program looks exactly like its serial counterpart and without explanation one would not expect that the program may run in several processors while implicitly updating other processor's memory with broadcasts. Is it indeed possible to write some dummy routines for the special hardware procedures (*mk_global(), synchronize()*) and then run the same program on a workstation (although the actual implementation history was the other way round: additions were made to a serial implementation to take care of the special NERV features).

Therefore one can immediately compare the timing of the multiprocessor version with the single processor version. One implementation of the above algorithm tried to find a solution for the Quadratic Assignment Problem (QAP) which is known to be NP–hard. This special problem required several changes to the algorithm which have been omitted for the sake of clarity. E.g. the chromosomes were not a string of bits but a given permutation of the natural numbers 1 to L. Each solution was required to be such a permutation which puts constraints on the crossover and mutation operators. However all of these modification were only local and did not change e.g. the communication behavior of the program. The resulting program was run on a Macintosh IIci with A/UX as operating system. This machine uses an MC68030 processor so that the comparison of the execution times should be reasonable. The NERV system was running with one, two, or six processors respectively. Since there was no profiling tool on the NERV side the output of the UNIX time command is given. A NERV system with one processor is used as a reference point.

Only the real times are shown, since there is no meaningful interpretation of the user and sys times for the NERV system. The first part is for a program version which outputs several information after each generation (best value, mean value etc.). This is often desirable if one wants to look at the behavior of the algorithm during the run. The NERV system is however badly prepared for small outputs of data. Since it has no local storage it uses the mass storage of the workstation. Each *printf()* for example requires that the NERV system is stopped and waits for the host to handle the output transaction. One can see that the speedup is only a factor 2.2 in this case.

If the output is either disabled or handled in a different way, e.g. by collecting all data and outputting them at the end of the program with a single *fwrite()* command the NERV system performs much better. It achieves a speedup of a factor 5.2. This is still less than the maximal speedup of 6, partly due to the reasons explained above. Anyway one should keep in mind that two such different system are not directly comparable. The point is that the single processor version of the program could be mostly taken unchanged and put on the multiprocessor system with a significant speedup in time.

The measurements for the host above were taken on a workstation with a processor similar to the one in the NERV system. Today's workstations however are usually equipped with much faster processors. A typical RISC workstation (e.g. a Sparcstation 2) can easily outperform the NERV system with 6 processors. At the time of this writing a redesign of the NERV system is nearly finished. It uses a MC68040 processor with 25 MHz and 16 MByte of dynamic RAM for each module. The total system may contain up to 40 processing elements. Therefore one can again expect a significant decrease in computing time when a GA is implemented on such a system.

5 Conclusions

We have shown how to implement a standard genetic algorithm on a multiprocessor system. The speedup which has been achieved is proportional to the number of processors in the system. Putting in more processing elements reduces the computation time while the communication time remains constant.

The system circumvents the problems of a global shared memory by using a copy of all relevant data on every processor. The update of data is implemented by a broadcast facility. This ensures that all processor will immediately get a copy of any changed data. By using the broadcast facility and a global bus this can be accomplished much faster than with any message passing system. Since each changeable datum is assigned to a certain processor which is responsible for the update, no other hardware mechanisms are necessary to control exclusive access. Synchronization is only necessary after each application of an operator and is also efficiently supported by hardware.

Table 1: Absolute times for GA simulation on different processors.
Problem: Quadratic Assignment Problem, Problem Size = 30
Population Size = 120, Number of Generations = 250

	1 processor	2 processors	6 processors	Mac IIci
with output	300.1 s	192.3 s	132.03 s	309.35 s
without output	184.9 s	99.3 s	35.01 s	229.03 s

Table 2: Relative performance compared to an one–processor NERV system.

	1 processor	2 processors	6 processors	Mac IIci
with output	1.0	1.56	2.2	0.97
without output	1.0	1.86	5.2	0.81

References

[1] J.H. Holland, Adaption in Natural and Artificial Systems (The University of Michigan Press, Ann Arbor, 1975)

[2] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, (Addison–Wesley, Reading, 1988)

[3] M. Gorges–Schleuter, ASPARAGOS: An Asynchronous Parallel Genetic Optimization Strategy, *Proc. 3rd Intl. Conf. on Genetic Algorithms* (1989) 422–427

[4] T. Fogarty, Implementing the Genetic Algorithm on Transputer Based Parallel Processing Systems, *Parallel Problem Solving from Nature 1* (1991) 145–149

[5] J.P. Cohoon, W.N. Martin, D.S. Richards, A Multi–population Genetic Algorithm for Solving the K–Partition Problem on Hyper–cubes, *Proc. 4th Intl. Conf. on Genetic Algorithms* (1991) 244–248

[6] R.J. Collins, D.R. Jefferson, Selection in Massively Parallel Genetic Algorithms, *Proc. 4th Intl. Conf. on Genetic Algorithms* (1991) 249–256

[7] P. Spiessens, B. Manderick, A Massively Parallel Genetic Algorithm, *Proc. 4th Intl. Conf. on Genetic Algorithms* (1991) 279–285

[8] C.C. Pettey, M.R. Leuze, A Theoretical Investigation of a Parallel Genetic Algorithm, *Proc. 3rd Intl. Conf. on Genetic Algorithms* (1989) 398–405

[9] R. Tanese, Distributed Genetic Algorithms, *Proc. 3rd Intl. Conf. on Genetic Algorithms* (1989) 434–439

[10] M.G.A. Verhoeven, E.H.L. Aarts, E. van de Sluis, Parallel Local Search and the Travelling Salesman Problem, *Parallel Problem Solving from Nature 2* (1992) 543–552

[11] Tsutomu Maruyma, Akihiko Konagaya, Koichi Konishi: An Asynchronous Fine–Grained Parallel Genetic Algorithm, *Parallel Problem Solving from Nature 2* (1992) 563–572

[12] Hisashi Tamaki, Yoshikazu Nishikawa: A Parallel Genetic Algorithm based on a Neighborhood Model and Its Application to the Jobshop Scheduling, *Parallel Problem Solving from Nature 2* (1992) 573–582

[13] H. Mühlenbein, Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization; in J.D. Becker, I. Eisele, F.W. Mündemann (Eds.): *Parallelism, Learning, Evolution, Lect. Notes in Comp. Sci.* **565**, (Springer, Berlin, 1991) 398–406

[14] R. Hauser, H. Horner, R. Männer, M. Makhaniok, Architectural Considerations for NERV—A General Purpose Neural Network Simulation System; in J.D. Becker, I. Eisele, F.W. Mündemann (Eds.): *Parallelism, Learning, Evolution, Lect. Notes in Comp. Sci.* **565**, (Springer, Berlin, 1991) 183–195

[15] The VMEbus Specification, Rev. C, VMEbus International Trade Association (1987)

[16] R.M. Stallman, Using and Porting GNU CC, Free Software Foundation (1992)