A GENETIC ALGORITHM FOR THE ASSEMBLY LINE BALANCING PROBLEM

by

Edward J. Anderson
and
Michael C. Ferris

Computer Sciences Technical Report #926

March 1990

A GENETIC ALGORITHM FOR THE ASSEMBLY LINE BALANCING PROBLEM *

EDWARD J. ANDERSON† AND MICHAEL C. FERRIS‡

Abstract. Randomized algorithms are being used extensively in optimization. We discuss one such algorithm, called a genetic algorithm, which can be used for combinatorial optimization. We will consider the application of the genetic algorithm to a particular problem, the Assembly Line Balancing Problem, and its implementation on a parallel architecture. The general scheme of a genetic algorithm is given, and its specialized use on our test bed problems is outlined. Although extensive parallel processing is available in these methods, the problems of communication and synchronization have not been considered in detail. We describe a prototype local neighborhood genetic algorithm for which communication is greatly reduced and give results of experimentation on several different neighborhood structures. A possible asynchronous scheme is also mentioned.

1. Introduction. Algorithms based on genetic ideas were first used to solve optimization problems more than twenty years ago (e.g. [Bag67]). During the 1970's this work continued, but was largely unknown. In the last five years, however, there has been increasing interest in genetic algorithms. There have been three conferences devoted to this topic and two books have appeared [Dav87, Gol89].

Many researchers have concentrated their efforts on nonlinear function optimization or the nonlinear programming problem. Our interest is in the application of genetic algorithms to combinatorial optimization problems. It is appropriate to start with an outline description of this type of approach to combinatorial optimization. A genetic algorithm (GA) works with a whole population of potential solutions, $(i = 1, \ldots, popsize)$, which we will call individuals. The population changes over time, but always has the same number of members. Each individual is usually represented by a single string of characters. At every iteration of the algorithm a fitness value, $f(i)$, is calculated for each of the current individuals. Based on this fitness function a number of individuals are selected as potential parents. Two new individuals can be obtained from two parents by choosing a random point along the string, splitting both strings at that point and then joining the front part of one parent to the back part of the other parent and vice versa. Thus parents A-B-C-A-B-C-A-B-C and A-A-B-B-C-C-C-B-A might produce offspring A-B-C-B-C-C-C-B-A and A-A-B-A-B-C-A-B-C when mated. This process is usually called crossover. Individuals may also change through random mutation when elements within a string are changed directly (normally this happens with only a low probability). The processes of crossover and mutation are collectively referred to as reproduction. The end result is a new population (the next "generation") and the whole process repeats. Over time this leads to convergence within a population with fewer and fewer differences between individuals. When a genetic algorithm works well the

* This material is based on research supported by NSF Grant DCR–8521228 and Air Force Office of Scientific Research Grant AFOSR–89–0410

† University of Cambridge, Institute of Management Studies, Mill Lane, Cambridge CB2 1RX, England

‡ Computer Sciences Department, University of Wisconsin, Madison, Wisconsin 53706

population converges to a good solution of the underlying optimization problem and the best individual in the population after many generations is likely to be close to the global optimum. A model algorithm is given below:

Model Algorithm:
repeat
    for each individual i **do evaluate** f(i)
    for i=1 **to** (popsize/2) **do**
        select pairs of individuals j and k to mate based on their fitness
        reproduce using individuals j and k
until population variance is small

There are some similarities between a GA and the method of simulated annealing. Both approaches involve some random element in the way that the algorithm proceeds. In both cases the running time of the method will depend on certain parameter settings, with a greater likelihood that an optimal or near–optimal solution is found if the algorithm is allowed to run for a long time. Both methods have applicability across a wide range of problem domains – part of their attraction is that they hold out the promise of effectiveness without being dependent on a detailed knowledge of the problem domain. On the other hand there are substantial differences between the two approaches: for example since GAs operate using a whole population of individuals, they have a kind of natural parallelism not found in simulated annealing.

It is not easy to assess the effectiveness of this type of algorithm. For any particular problem there are likely to be special purpose techniques and heuristics which will outperform a more general purpose method. In a sense this may not be important. The value of simulated annealing, for example, is that for many problems half a day's programming and a day's computation will suffice to find as good a solution as would be obtainable after an half an hour's computation using a sophisticated method which might take three weeks or more to get working. Might something similar be true for genetic algorithms?

The paper consists of two parts. In the first part we have tried to establish that, for one particular type of problem, GAs have a clear advantage over the simplest of all generic approaches, which is the use of a neighborhood search technique with multiple starts. This is perhaps the minimal condition for genetic algorithms to be considered as a serious contender in solving hard combinatorial optimization problems. As far as we are aware this type of comparison has not been carried out before.

In the second part of the paper, we will describe more fully the parallelism which is inherent to the genetic algorithm and explain the various extensions which enable efficient implementation of the algorithm on message passing systems. Related work can be found in [GS89, Mö89, SG87, Tan87]

2. The Assembly Line Balancing Problem. We will look at the application of genetic algorithms to the Assembly Line Balancing Problem (ALBP). Suppose we wish to design a manufacturing line using a given number of stations, n. At each station

someone will perform a number of operations on an item being made, before passing it on to the next station in the line. The problem is to assign the operations to the n stations in such a way as to produce a balanced line, given the time that each operation will take. The total output of the line will be determined by the slowest station, which is the station with the most work assigned to it. Our aim is to minimize the amount of work assigned to this station and thus to maximize the total throughput of the line. Thus far we have described a type of standard balancing problem – in a scheduling context this would be equivalent to minimizing makespan on parallel machines. The crucial feature of the ALBP, however, is that certain operations must be performed before others can be started. If we have this type of precedence relation between operations A and B, for example, then we cannot assign operation B to a station earlier than operation A (though both operations may be assigned to the same station).

The ALBP has attracted the attention of many researchers. Both heuristic and exact methods have been proposed for its solution. For a review of some of these methods see the papers [TPG86, Joh88]. Note that the ALBP is sometimes posed with the total operation time for each station constrained by some upper bound (the desired "cycle time") and the number of stations as the variable to be minimized. The ALBP is attractive as a test bed for GAs since there is a natural coding of a solution given by the station assignment for each operation. Also, though we do not expect a GA to be as effective as some of the special purpose heuristic methods, it will nevertheless be interesting to make comparisons between the two approaches.

There are a number of issues to be resolved in implementing a genetic algorithm for the ALBP. We will now deal with three of these issues; namely the coding scheme used, the method of calculating fitness and selecting individuals for mating, and the recombination mechanism (of mutation and crossover).

Coding. There are two aspects to the coding scheme for a genetic algorithm. One is the way that a solution is related to the elements of the string which codes for it. As we mentioned above a natural coding is available for the ALBP in which each operation is associated with a fixed position on the string and the code is simply the number of the station to which that operation is assigned. The other aspect of coding is the location of the operations on the string. This will be important since the crossover operation will be less likely to separate two pieces of information ("genes" in the genetic description) if they are close together on the string. We have chosen to put the operations into an order given by increasing numbers of predecessors, with ties broken by looking at the number of immediate predecessors.

Fitness and selection for the mating pool. At the heart of a GA is some calculation of fitness for each member of the population. This will determine how likely it is that an individual survives into the next generation, or is selected for mating. For the ALBP the fitness must include an element corresponding to the total time for the operations assigned to the slowest station. However we also wish to avoid solutions which are infeasible because of precedence constraints. Rather than rule these out directly, we will assign a large penalty cost to any of these infeasible solutions.

The fitness function we use is defined to be

$$exp(-k(Tmax + d * V + e * T2 + f * (Tmax - Tmin)))$$

Here $Tmax$ is the slowest station time, $T2$ is the second slowest station time, $Tmin$ is the fastest station time and V is the number of precedence violations. The constants d, e, f and k are chosen as follows. $d \gg 1$ in order that the precedence violations are removed as quickly as possible. e is chosen so that the second slowest time is taken into account (to decrease the solution value it would be valuable to have this time smaller than $Tmax$) but so that $Tmax$ dominates. The final term is added to try and force a balanced line (but with f chosen suitably small). The constant k is chosen so that the values of the fitness function lie within reasonable bounds.

We have also implemented a linear scaling of the fitness values. The scaling is performed in such a way that the average fitness remains constant but the maximum fitness is a multiple (usually 1.5) of this average value.

We have used two different methods to select individuals for mating. The first (stochastic sampling with replacement) selects each individual with a probability proportional to its fitness. Thus each time a selection is made individual i is chosen with probability given by

$$pselect_i = f_i / \sum_j f_j$$

The second method, which is usually called "remainder stochastic sampling without replacement" is similar but involves some selections being made in a deterministic way. First each individual is allocated samples according to the integer parts of $e_i = pselect_i * n$. The remaining samples are taken one by one, with at each stage the probability of individual i being selected proportional to the fractional part of e_i until the new population has the desired size. This scheme has the advantage that all above average individuals will survive to mate in the next generation. The second of these methods is generally acknowledged to be superior and this has been confirmed by numerical testing on the ALBP.

Recombination. There are two aspects to recombination: crossover and mutation. We have made some limited experiments with different crossover mechanisms with the aim of incorporating some problem specific knowledge. This has been shown to be effective in some previous studies [Gre87]. One method that we tested was to concentrate the changes that crossover introduces within the slowest station. For each parent, this was achieved by changing some of the station assignments for operations currently assigned to one of the slowest stations to the station assignment that that operation has in the other parent. However this approach did not lead to any very substantial improvements over the more standard crossover mechanism described in the introduction. For our problem, a more natural scheme for crossover is to randomly generate an operation number and cross over that operation and operations which are its successors in the precedence graph. This appears to work slightly better than the

scheme outlined in the introduction and is the scheme of choice for the remaining results reported here.

We have implemented two forms of mutation. The first is standard in the literature of genetic algorithms and involves the random change of a particular allocation of operation to station. We move a random operation from its station to the station immediately before or after it. The second scheme is motivated by our particular problem. Mutation is achieved by choosing two adjacent stations and moving each task from one of these stations to the other with small probability. We also tested concentrating the mutations within the stations achieving the slowest time. Over a number of runs this usually gave a slightly better solution, but the variance of the solutions was much greater and frequently infeasible solutions resulted.

3. Experimental Results (Comparison). Experiments were performed on randomly generated problems having 40 operations; these are to be assigned to 6 stations. One factor which we varied was the number of precedence relations included. This can be conveniently measured as a "density" giving the number of precedence relations actually present or implied divided by the total number possible.

As stated earlier our aim is to compare an implementation of a genetic algorithm with a simple neighborhood search scheme. We have implemented the neighborhood search by taking a population of initial solutions and at each generation replacing each individual with a randomly generated neighbor. The randomly generated neighbor is given by applying the mutation operation described above to the individual and only replacing the current individual by its neighbor if the neighbor is as good as the individual. For the genetic algorithm we used a probability of crossover of 0.6 and a probability of mutation of 0.03 (for each element in a string). For both the neighborhood search method and the genetic algorithm we used a population size of 40.

We experimented with two kinds of starting solution. The first scheme generated the initial population entirely at random. In order that one can effectively program a genetic code quickly, this would be the method of preference for generating initial solutions. In this case, the genetic algorithm performed far better than the neighborhood scheme. Indeed, in most cases, the neighborhood scheme was unable to find a feasible solution. However, the genetic algorithm was able to find a feasible individual whose maximum time was within 10% of the optimal time for the problem in every case we tried.

However the situation is different when a preselected initial population is used. We generated a set of initial solutions using a method due to Arcus (see [Arc66]). This is extremely effective. In a significant proportion of cases the initial set of Arcus solutions contains at least one which is never improved upon by either the GA or neighborhood search method. In the other cases the best of the Arcus solutions is never far from the best solution found. Using the same parameters as before for the GA and with a limit of 40 iterations the performance of the GA and the neighborhood search scheme was comparable. It seems that there is premature convergence of the method around the few individuals which are generated by the Arcus scheme. With insufficient variability in the population the GA is unable to work well.

The total time assigned to the slowest station for these randomly generated problems was around 50. In cases where the GA or the neighborhood scheme did better it was usually by a margin of 1 unit, implying that the throughput of the line would be improved by about 2%.

There are a number of conclusions that can be drawn from these experiments. First we note how difficult it is to improve on the "standard" version of the genetic algorithm; the only area where changes were able to show some improvement for the problem we looked at was in the use of different mutation schemes. This characteristic of genetic algorithms is clearly beneficial if it is desired to use the technique on problem domains about which one has little knowledge. We have shown that for the assembly line balancing problem a genetic algorithm performs significantly better than simple neighborhood search from a random initial population. The effectiveness of the method does depend somewhat on the density of the problem, the best results being obtained for problems with relatively low densities. If an initial population of good solutions is used then a genetic algorithm may offer little benefit in comparison with a more straightforward neighborhood search scheme. However, even in this case, both methods give an improvement of about 2% over the initial solutions after 40 generations.

4. Parallel Implementation Using Local Neighborhoods. Up to this point we have only been concerned with a comparison of the genetic algorithm with other schemes used to solve our problem. In the remainder of this paper we will look at the parallelism associated with the genetic algorithm initialized with a random population and try to improve on synchronization and communication costs. A GA would appear to be ideal for parallel architectures since evaluation and reproduction can be performed concurrently. In fact, for a large population of individuals, the use of many processors would seem enticing. However, this ignores the problem of communication and synchronization which are inherent in the selection mechanism described above. In this section we discuss how selection can be performed in a way which reduces the message passing required in a parallel architecture. Not only is this beneficial from the point of view of communication penalties but computational experience demonstrates that it will also improve the quality of solutions obtained.

Note that for both of the techniques of selection described previously, the processor effecting the selection needs to acquire the fitness values of every individual. This involves a large communication penalty for any message passing system.

Recently, several researchers have experimented with neighborhood schemes in which the fitness information need only be transmitted within the local neighborhood. The pioneers in this area are H. Mühlenbein and M. Gorges–Schleuter who have developed the ASPARAGOS system to implement an asynchronous parallel genetic algorithm [GS89, M89]. In this paper we will concentrate on two issues: first the determination of the best neighborhood structure to use, and second the difference between synchronous and asynchronous versions of the algorithm. We aim to clarify these issues by discussing the performance of a parallel GA on a particular problem.

A model algorithm for a scheme in which fitness information is only compared locally is as follows.

Local Neighborhood Algorithm:

repeat
  for each individual i **do**
    evaluate f(i)
    broadcast f(i) in the neighborhood of i
    receive f(j) for all individuals j in the neighborhood
    select individuals j and k to mate from neighborhood based on fitness
    request individuals j and k
    synchronize
    reproduce using individuals j and k
until population variance is small

For a parallel implementation, we assume that each individual in the population resides on a processor and communication is carried out by message passing. Our experimental results are concerned with several neighborhood schemes which we describe briefly now:

global: Here every individual is in the neighborhood of every other individual. The neighborhood size is the size of the population.

hypercube:

$$i \in \text{nhd(j)} \iff \text{d}_\text{H}(\text{i},\text{j}) \leq 1$$

where $d_H(i,j)$ is the number of different bits in the binary expansions of i and j. This can be viewed as each individual residing on the vertex of a hypercube with adjacent vertices giving its neighbors. The neighborhood size is one more than the dimension of the hypercube.

ring4:

$$i \in \text{nhd(j)} \iff |\text{i} - \text{j}| \leq 2$$

This can be viewed as each individual residing on a ring and its neighbors are those no further than two links away. The neighborhood size is four.

ring8:

$$i \in \text{nhd(j)} \iff |\text{i} - \text{j}| \leq 4$$

This can be viewed as each individual residing on a ring and its neighbors are those no further than four links away. The neighborhood size is eight.

grid4: Suppose $popsize = r^2$ and individuals are labeled as (u,v) with $u,v \in \{1,\ldots,r\}$. Let $d_r(a,b) := |(a \bmod r) - (b \bmod r)|$. Then

$$(u,v) \in \text{nhd((x,y))} \iff \text{d}_\text{r}(\text{u},\text{x}) + \text{d}_\text{r}(\text{v},\text{y}) \leq 1$$

This can be viewed as each individual lying on a grid and only communicating with the four grid points which differ by one in at most one component (with wrap around at the edges).

grid8: Using the same setup as grid4

$$(u,v) \in \mathrm{nhd}((\mathrm{x},\mathrm{y})) \iff \max\{\mathrm{d_r}(\mathrm{u},\mathrm{x}), \mathrm{d_r}(\mathrm{v},\mathrm{y})\} \leq 1$$

This can be viewed as each individual lying on a grid and only communicating with the eight grid points which differ by less than one in every component.

island: Suppose $popsize = r * s$ and individuals are labeled as (u,v) with $u \in \{1,\ldots,r\}$ and $v \in \{1,\ldots,s\}$. Then

$$(u,v) \in \mathrm{nhd}((\mathrm{x},\mathrm{y})) \iff \begin{cases} u = x \text{ and } |v-y| \leq 2 \text{ or} \\ v = y = 1 \text{ and } |u-x| \leq 1 \end{cases}$$

This can be viewed as r island populations which can only communicate with the rest of the world through a particular individual. The neighborhood size is between 4 and 6.

We now describe some further details of an implementation of this algorithm. Because we can no longer think of a single mating pool it is unclear how to implement remainder stochastic sampling without replacement. Instead we carry out selection using stochastic sampling with replacement. In the form given above we select two individuals j and k from the neighborhood based on fitness. In order to reduce communication it is possible to set $k = i$. This technique was used on our test problems with each of the neighborhood schemes above and performed at least as well and frequently better than the original scheme. For the rest of our experimentation, this technique was used. We postulate that (although the new technique allows potentially poor solutions to be involved in mating which could degrade performance) the greater variability in solutions considered is helpful.

Reproduction produces two offspring. Our strategy was to replace the current individual with its best offspring provided this offspring is better than the worst individual in the neighborhood. The question arises whether it is possible to use the other offspring? In order to answer this, the following techniques were tested:

noret: The less fit offspring is discarded.

retpar: The less fit offspring is sent to its other parent which it replaces if it is fitter than this parent.

retran: The less fit offspring is sent to a random neighbor which it replaces if it is fitter than this neighbor.

In the sequel, we shall refer to these techniques as "return policies".

5. Experimental Results (Parallelism). The aim of these tests was to determine, if possible, which neighborhood scheme and which return policy was optimal. Although these results may depend on our particular problem, it is hoped that the conclusions of this research will be applicable in the general context of genetic algorithms applied to combinatorial optimization problems. Our experiments were confined to a randomly generated test set with the number of stations varying between 2 and 6, the number of operations varying between 40 and 50 and the density of the precedence graph ranging between 0.2 and 0.8. The operation times were generated either from a

	optimality tolerance	noret	retpar	retran	twosel
global	0.1%	245	256	52	64
	1.0%	805	829	632	640
	2.0%	827	896	640	704
hypercube	0.1%	64	192	128	0
	1.0%	704	768	704	448
	2.0%	704	832	768	576
ring4	0.1%	197	216	85	128
	1.0%	826	790	768	448
	2.0%	831	804	832	640
ring8	0.1%	192	192	0	128
	1.0%	832	960	512	640
	2.0%	944	960	731	768
grid4	0.1%	128	64	256	0
	1.0%	832	576	704	448
	2.0%	832	640	768	576
grid8	0.1%	128	0	128	64
	1.0%	704	576	704	384
	2.0%	832	704	704	832
island	0.1%	349	309	355	64
	1.0%	829	854	768	640
	2.0%	829	876	916	704

TABLE 1
Number of individuals within optimality tolerance

uniform distribution on [50,500] or a binomial distribution with parameters $n = 30$ and $p = 0.25$, The generator used to produce these problems is described in more detail in [TPG86]. Each problem was allowed to run for 400 generations on a population size of 64 starting from 5 random initial populations. Table 1 is a summary of the experimental results we obtained. For each of the neighborhood schemes we calculate how many individuals from the population at generation 400 are within 0.1%, 1.0% and 2.0% of the optimal solution for each of the problems and take their sum. We feel this is the significant figure for each of the procedures - it is an attempt to ascertain which neighborhood structure and which return policy works best, independent of other factors (such as density of precedence graph, number of operations, etc) on the problem. The maximum figure that could appear in the table is 1280.

In all cases, the minimum value found in the final generation by the genetic algorithm was within one–tenth of a percent of optimality. We do not consider the best solution found over all generations, since the communication involved in determining this solution would be enormous. However, in a serial implementation, the best solution found in the final generation was close to and in general equal to this value. Furthermore, the local neighborhood algorithm always outperformed the standard schemes

	global	hypercube	ring4	ring8	grid4	grid8	island
global	–	3:1	2:2	3:1	3:1	3:1	0:3*
hypercube	1:3	–	1:3	1:3	1:2*	2:2	0:4
ring4	2:2	3:1	–	2:2	3:1	3:1	1:3
ring8	1:3	3:1	2:2	–	3:1	3:1	1:3
grid4	1:3	2:1*	1:3	1:3	–	3:1	0:4
grid8	1:3	2:2	1:3	1:3	1:3	–	0:4
island	3:0*	4:0	3:1	3:1	4:0	4:0	–

TABLE 2
Neighborhood scheme rating

where selection was performed (with or without replacement) on the entire population.

Note that the values reported in Table 1 are frequently multiples of 64. This is due to the fact that after 400 iterations, the GA has frequently converged and all the individuals have the same value. It is clear that different schemes will have different convergence rates. Some of the schemes above converged after around 200 generations and produced inferior quality solutions, mainly due to the fact that they did not look at enough different individuals. In order to obtain a fair comparison, we set the scale factor in the linear scaling of the fitness values to 1.2 instead of 1.5 (in the case of the hypercube or grid8 neighborhoods). Figures 1 and 2 show the effect of changing the scale factor. The graphs show the maximum, minimum and average values of the objective function for a particular problem instance averaged over 5 runs of the genetic algorithm as a function of generation. In Figure 1, the linear scaling factor makes the maximum fitness 1.5 times the average fitness, whereas in Figure 2 the scale factor is 1.2. Notice that the convergence is slower in the second example, but the histograms (which are plots of the numbers of individuals against the maximum station time) show the quality of solutions is much better.

From Table 1 we produce two further tables which we claim show which neighborhood scheme is best and which return policies work well. Essentially, we order the triples found in Table 1. Two triples, (a, b, c) and (d, e, f) are ordered as follows:

$$(a,b,c) \text{ "is better than" } (d,e,f)$$
$$\iff \begin{cases} d/a < 0.95 \\ \text{or } 0.95 \leq d/a \leq 1.05 \text{ and } e/b < 0.95 \\ \text{or } 0.95 \leq d/a \leq 1.05 \text{ and } 0.95 \leq e/b \leq 1.05 \text{ and } f/c < 0.95 \end{cases}$$

Essentially, one triple is better than another if is is lexicographically greater than the other (with two elements of the triple being treated as equal if they differ by less than 5%). To obtain the ijth entry of Table 2 we compare rows of Table 1. Thus entry (hypercube, ring8) in the table is 1:3 which means that for 1 return policy the hypercube neighborhood structure was better than the ring8 structure, whereas for the other 3 return policies, the ring8 structure gave better solutions. A * represents the fact that a tie occurred. From the results given in Table 2 we conclude that island is the best

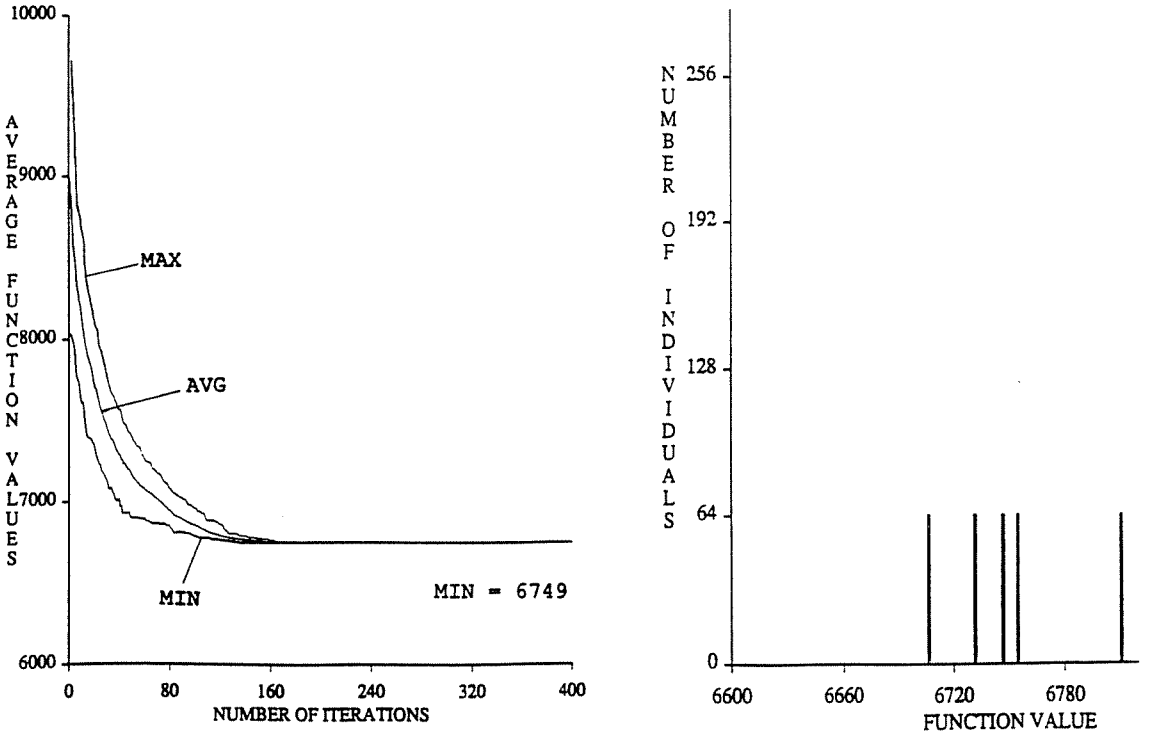

Figure 1: Scale factor 1.5, grid8 neighborhood

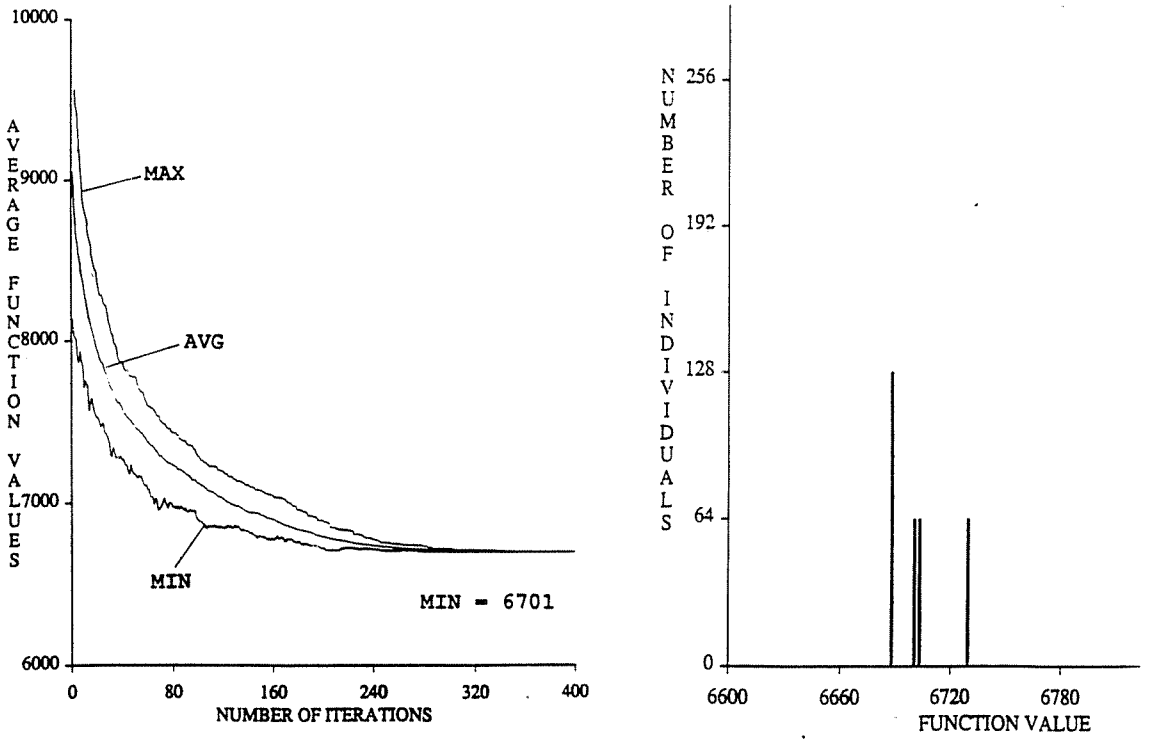

Figure 2: Scale factor 1.2, grid8 neighborhood

	noret	retpar	retran	twosel
noret	–	3:4	4:3	7:0
retpar	4:3	–	4:3	6:1
retran	3:4	3:4	–	4:3
twosel	0:7	1:6	3:4	–

TABLE 3
Return policy rating

neighborhood scheme, followed by the global scheme and then the schemes ring4 and ring8. The grid4, grid8 and hypercube schemes are comparable to each other but not very good at all. We also note that the global scheme requires much more computation, since the neighborhood size is much bigger.

Table 3 gives the comparison of return policies by comparing columns of Table 1 in a similar manner as above. The results given in Table 3 are not as conclusive. In fact, noret, retpar and retran perform somewhat similarly. For completeness, we have included a column labeled twosel. This corresponds to selecting two individuals from the neighborhood and using policy noret for the worst offspring. As mentioned above, the results in the table show this does not perform as well as the other schemes and is clearly not a good policy to consider.

Since the communication costs associated with noret are smaller than those associated with retpar and retran and the quality of their solutions is very comparable, we conclude that discarding the less fit offspring is the best return policy and that the island neighborhood scheme (or a closely related variant of this) should be chosen as the neighborhood when these scale factors are used.

We also carried out further experimentation as to the effect of scaling on our results. Tables 4, 5 and 6 are the corresponding results when we require the maximum fitness to be 1.15 times the average fitness.

The results given in Table 4 are better than those given in Table 1 as can be seen by a similar comparison as the one described above. Some of the conclusions we can draw from Tables 5 and 6 are slightly different from our previous conclusions. The striking difference is that the global scheme now performs badly. The ring4 and ring8 schemes are now the best, with the grid4, grid8 and island schemes being comparable to each other, but not as good. The global and hypercube schemes do not perform well. Furthermore, it appears that the best return policy is retran. However, this effect should be balanced against the extra communication cost.

A plausible explanation for the above behavior is that information is dissipated through the population much slower in the ring4, ring8 and island schemes, which allows many more local optima to be explored. The island scheme works better than the other schemes when the fitness values are not scaled down as much because it still allows several local optima to be explored, even if there is one dominant local optima.

We conclude from this that an appropriate scaling of the fitness values will improve the quality of the solutions found. Clearly, experimentation has to be performed in order to calculate the correct scale factor for a given problem. Generally, increasing the

	optimality tolerance	noret	retpar	retran	twosel
global	0.1%	94	144	4	192
	1.0%	583	758	702	768
	2.0%	886	873	704	832
hypercube	0.1%	320	256	320	0
	1.0%	832	832	896	384
	2.0%	832	896	896	576
ring4	0.1%	425	370	168	384
	1.0%	866	965	873	960
	2.0%	941	1012	896	960
ring8	0.1%	299	376	448	128
	1.0%	768	960	960	576
	2.0%	896	960	960	832
grid4	0.1%	255	128	320	128
	1.0%	896	896	896	512
	2.0%	896	896	896	768
grid8	0.1%	256	256	192	64
	1.0%	896	896	832	768
	2.0%	1024	896	896	896
island	0.1%	108	145	378	256
	1.0%	865	793	810	576
	2.0%	879	896	960	768

TABLE 4
Number of individuals within optimality tolerance

	global	hypercube	ring4	ring8	grid4	grid8	island
global	–	2:2	1:3	1:3	1:3	1:3	0:4
hypercube	2:2	–	1:3	1:3	2:1*	2:2	2:2
ring4	3:1	3:1	–	3:1	3:1	3:1	3:1
ring8	3:1	3:1	1:3	–	4:0	4:0	3:1
grid4	3:1	1:2*	1:3	0:4	–	2:2	1:3
grid8	3:1	2:2	1:3	0:4	2:2	–	2:2
island	4:0	2:2	1:3	1:3	1:3	2:2	–

TABLE 5
Neighborhood scheme rating

	noret	retpar	retran	twosel
noret	–	4:2	2:4	5:1
retpar	2:4	–	2:4	4:2
retran	4:2	4:2	–	4:2
twosel	1:5	2:4	2:4	–

TABLE 6
Return policy rating

scale factor speeds up convergence, and decreasing the scale factor gives better quality solutions, provided the genetic algorithm is allowed to run until it has converged.

6. Synchronization. The neighborhood schemes of the previous section effectively deal with communication penalties provided that care is taken to use a neighborhood structure which is appropriate for the machine architecture. The synchronization issue remains largely unsolved, but the following asynchronous scheme has proven very effective in practice.

Asynchronous Local Neighborhood Algorithm:
repeat
  for each individual i **do**
    evaluate f(i)
    broadcast f(i) in the neighborhood of i
    receive f(j) for all individuals j in the neighborhood
    select an individual j to mate from neighborhood based on fitness
    request individual j
    reproduce using individuals i and j
until population variance is small

Note that since we have removed the synchronization step, it may happen that we request an individual based on its fitness, but in fact receive an individual which has replaced the one requested. In practice, this does not seem to matter and in experiments carried out on our test examples, the above algorithm has produced results of comparable quality in somewhat smaller computational times.

7. Conclusions. The conclusions of this work are twofold. Firstly, the comparison of the standard genetic algorithm with a neighborhood search scheme with multiple restarts shows that the genetic algorithm outperforms this method and invariably produces better solutions.

We conclude from our experimental parallel work on the genetic algorithm that using a local neighborhood scheme is very important for achieving close to optimal solutions and in fact produces better solutions than the standard genetic algorithm. The choice of neighborhood seems to be machine dependent: our computational results indicate that a variant of the island or the ring schemes we define in this paper seems to give the best performance. If two offspring are produced from the reproduction

phase, it is better to discard the less fit offspring. Mating on each processor should be between the individual that is situated at the processor and an individual selected from the neighborhood. Appropriate choice of scaling for the fitness values can improve the quality of solutions or speed up the convergence rate.

Further work on these techniques is in progress. New techniques for evaluation of fitness under constraints and applications to database query optimization are being developed.

Acknowledgement. The authors would like to thank Mr. Menglin Cao for his help in testing the algorithm.

REFERENCES

[Arc66] A.L. Arcus. COMSOAL: A computer method of sequencing operations for assembly lines. In E.S. Buffa, editor, *Readings in Production and Operations Management.* John Wiley & Sons, New York, 1966.

[Bag67] J.D. Bagley. *The Behaviour of Adaptive Systems which employ Genetic and Correlation Algorithms.* PhD thesis, University of Michigan, 1967.

[Dav87] L. Davis, editor. *Genetic Algorithms and Simulated Annealing.* Pitman, London, 1987.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison–Wesley, Reading MA, 1989.

[Gre87] J.J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In Davis [Dav87].

[GS89] M. Georges-Schleuter. Asparagos. In Schaeffer [Sch89], pages 416–421.

[Joh88] R. Johnson. Optimally balancing large assembly lines with FABLE. *Management Science,* 34:240–253, 1988.

[MÖ89] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In Schaeffer [Sch89], pages 416–421.

[Sch89] J.D. Schaeffer, editor. *Proceedings of the Third International Conference on Genetic Algorithms,* San Mateo, California, 1989. Morgan Kaufmann Publishers, Inc.

[SG87] J.Y. Suh and D. Van Gucht. Distributed genetic algorithms. Technical Report 225, Computer Science Department, Indiana University, Bloomington, July 1987.

[Tan87] R. Tanese. Parallel genetic algorithms for a hypercube. In J.J. Grefenstette, editor, *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms,* pages 177–183, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates.

[TPG86] F.B. Talbot, J.H. Patterson, and W.V. Gehrlein. A comparative evaluation of heuristic line balancing techniques. *Management Science,* 32:430–454, 1986.