Genetic Algorithms for Combinatorial Optimization: The Assembly Line Balancing Problem

Edward J. Anderson
University of Cambridge
Judge Institute of Management Studies
Mill Lane, Cambridge CB2 1RX, England
eja2@phx.cam.ac.uk

Michael C. Ferris
Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706
ferris@cs.wisc.edu

August 1991
Revised June 1992
Revised January 1993

Abstract

Genetic algorithms are one example of the use of a random element within an algorithm for combinatorial optimization. We consider the application of the genetic algorithm to a particular problem, the Assembly Line Balancing Problem. A general description of genetic algorithms is given, and their specialized use on our test-bed problems is discussed. We carry out extensive computational testing to find appropriate values for the various parameters associated with this genetic algorithm. These experiments underscore the importance of the correct choice of a scaling parameter and mutation rate to ensure the good performance of a genetic algorithm. We also describe a parallel implementation of the genetic algorithm and give some comparisons between the parallel and serial implementations. Both versions of the algorithm are shown to be effective in producing good solutions for problems of this type (with appropriately chosen parameters).

Subject classifications: Programming: combinatorial optimization, parallel processing. Production/Scheduling: assembly line balancing.

Keywords: Genetic algorithms; combinatorial optimization; parallel processing; assembly line balancing.

0 Introduction

Algorithms based on genetic ideas were first used to solve optimization problems more than twenty years ago[4] following the development of the fundamental ideas of genetic algorithms by John Holland at the University of Michigan. During the 1970's this work continued, but was largely unknown. In the last few years, however, there has been increasing interest in genetic algorithms (see for example the conference proceedings[25, 6] and the books[7, 8, 13]).

A number of researchers have looked at the application of genetic algorithms to optimization of nonlinear functions; our interest, however, is in the application of this technique to combinatorial optimization problems. There has been a variety of work done in this area, much of it considering the application of genetic algorithms to the solution of the travelling salesman problem[19, 29, 16]. This is not surprising given the importance of the travelling salesman problem and its frequent use as a vehicle for testing new methods in combinatorial optimization. However there are particular difficulties with the use of a genetic algorithm for the travelling salesman problem and it is our belief that the usefulness of the method for other combinatorial optimization problems cannot be fairly assessed on the basis of its performance on the travelling salesman problem alone.

It is appropriate to start with an outline description of the way in which a genetic algorithm works. The method operates with a set of potential solutions. This is referred to as a *population* and members of the population are sometimes called *individuals*. The population changes over time, but always has the same size, N say. Each individual is represented by a single string of characters. At every iteration of the algorithm a fitness value, $f(i)(i = 1, \ldots, N)$, is calculated for each of the current individuals. Based on this fitness function a number of individuals are selected as potential parents. These form what is sometimes called a mating pool. The mating pool will have N members but some will be duplicates of each other, so that it contains several copies of some individuals in the current population and no copies of others. Individuals which are not selected for the mating pool are lost.

Two new individuals can be obtained from two parents in the mating pool 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 called *crossover*. It is not necessary for every member of the mating pool to be involved in crossover; some will be left unchanged. Individuals in the mating pool may also change through random mutation, when characters within a string are changed directly. Normally each character is given such a small probability of being changed, that most of the time individuals are left unaltered. The processes of crossover and mutation are collectively referred to as recombination. 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 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 in Exhibit I.

Exhibit I . Model Algorithm

repeat
  for each individual i **do evaluate** fitness $f(i)$
  create **mating pool** of size N based on fitness values $f(i)$
  for $i = 1$ **to** $(N/2)$ **do**
    remove pairs of individuals $\{j, k\}$ from mating pool
    recombine using individuals j and k
until population variance is small

A more detailed description of some elements of the method is given in the next section. It is important to realize that we have chosen one simple method of implementing a genetic algorithm, as has been described by [Gol89]. There are many other versions which have been suggested, some of which do not operate on a generation by generation basis and others of which use different mechanisms for reproduction. At the heart of all these methods, however, is the idea that good new solutions can be obtained by using as building blocks parts of previously existing solutions.

There has been an increasing interest in the use of search techniques which contain a stochastic element as methods for the solution of hard combinatorial optimization problems. Simulated Annealing[23, 9, 14] has this property, as does Tabu Search[12, 17] in some implementations. There has also been some recent work which has considered techniques which combine local search and the genetic algorithm ideas outlined above. For example Aarts et al.[1] have shown that such a technique is competitive with Simulated Annealing for some job shop scheduling problems. Also, recent careful computational work by Johnson et al.[20] has demonstrated that simulated annealing, for example, can be competitive with the best available heuristic methods for certain very large combinatorial optimization problems. This is the context in which the genetic algorithm approach should be evaluated. An optimistic assessment of the potential of this method is that, for at least some difficult combinatorial problems, a suitably tuned genetic algorithm may be competitive with the best available heuristic techniques.

There are a number of similarities between a genetic algorithm and other stochastic search techniques. The running time of these methods 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. Also such 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. One of the major differences between genetic algorithms and other stochastic search methods is that genetic algorithms operate using a whole population of individuals, and in this sense they have a kind of natural parallelism not found in Simulated Annealing or Tabu Search techniques.

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. One advantage of simulated annealing, for example, is that for many problems a very small amount of programming effort and a single long computation will suffice to find as good a solution as would be obtainable with a much faster and more sophisticated method which might take weeks to understand and program. Might something similar be true for genetic algorithms?

The primary aim of this paper is to demonstrate that genetic algorithms can be effective in the solution of combinatorial optimization problems and to give some guidance on the implementation of the genetic algorithm for both serial and parallel architectures. Many previous implementations of this method have been guided by experiments initially carried out by [22] and later extended by a number of other authors (for example [25]). In the great majority of these experiments nonlinear function optimization problems have been solved rather than combinatorial optimization problems. We believe that there is much more to be learnt about the best implementation of genetic algorithms for combinatorial problems.

The paper consists of two parts. In the first part we describe a fairly standard implementation of the genetic algorithm technique for the Assembly Line Balancing Problem. A careful discussion is given of the appropriate setting for the various parameters involved and some experiments are reported to indicate the relative importance of the genetic operators of crossover and selection. Provided that an appropriate scaling of fitness values is carried out, coupled with a relatively high mutation rate, the genetic algorithm we have implemented is an effective solution procedure for this problem. In the second part of the paper we describe an alternative parallel version of the algorithm for use on a message passing system. Some computational comparisons are carried out.

1 The Assembly Line Balancing Problem

We will look at the application of genetic algorithms to a particular problem occurring in Operations Management, namely the Assembly Line Balancing Problem (ALBP). It is helpful to have a specific problem class in mind when discussing the details of a genetic algorithm implementation and we believe that the issues which arise in dealing with the ALBP are in many cases the same issues which arise in tackling other combinatorial optimization problems. As we shall see the ALBP has a natural coding scheme which makes it attractive as a vehicle for testing genetic algorithms. The problem can be described as follows:

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 each 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 common type of balancing problem – in a scheduling context this would be equivalent to minimizing makespan on identical 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). Figure 1 gives an example of an ALBP, with arrows used to show the direction of precedence constraints. A possible solution to the problem with three stations is shown. Station 1 is assigned operations 1,2 and 4 and has a total processing time of 38 minutes. Stations 2 and 3 have total processing times of 42 and 33 minutes respectively.

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 [27, 21]. 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

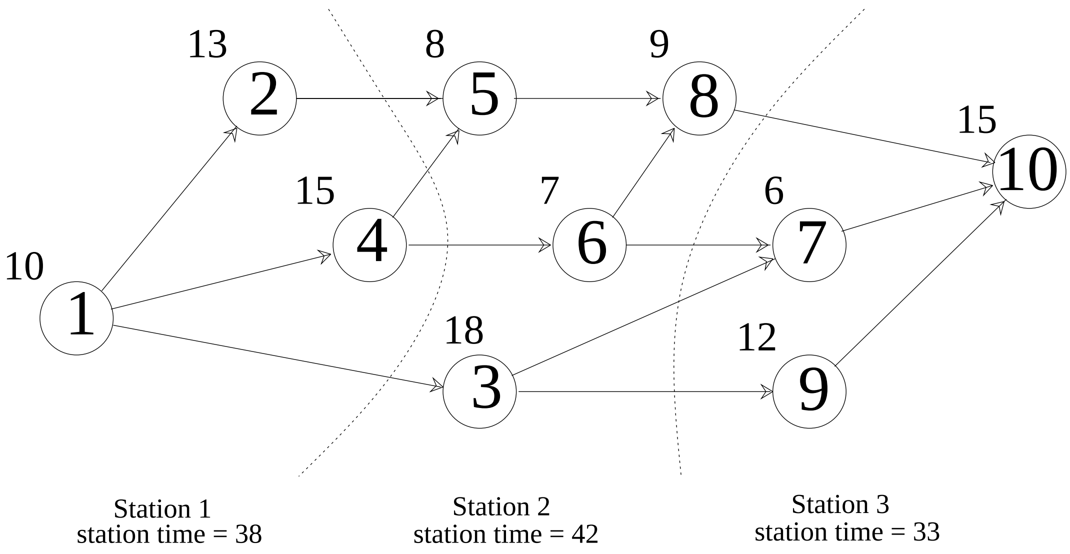

Figure 1: An example of a precedence graph

minimized.

Note that we do not expect a genetic algorithm to be as effective as some of the special purpose heuristic methods for the ALBP. As we mentioned above our aim is not to demonstrate the superiority of a genetic algorithm for this particular problem, but rather to give some indication of the potential of this technique for other combinatorial optimization problems.

There are a number of issues to be resolved in implementing a genetic algorithm for the ALBP. We will now deal with the major issues in turn. At this point we are concerned with the underlying mechanisms which should be used, rather than the most computationally efficient implementation.

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 this coding a solution is represented by a string each element of which is a number. The number in the i'th place in the string is the station to which the i'th operation is to be assigned. This still leaves one aspect of the coding undetermined: the numbering of the operations. This numbering 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. It seems sensible to put the operations into an order consistent with the precedence relations (so that if one operation is necessary before starting another its assignment will come in an earlier position in the string). For the example of Figure 1 we might choose to order the operations within the string with the operation numbers given. Then the assignment shown would be coded as 1121223233.

It is perhaps appropriate to note that the majority of genetic algorithm research has been carried out on strings using a binary alphabet (so that each element in a string is either a 0 or a 1). We do not believe that translating the coding scheme we have used into a binary alphabet would be sensible for this problem.

Achieving feasible solutions

Many of the possible assignments of operations to station are infeasible because they break one or more of the precedence constraints. It is a characteristic of the genetic algorithm that, for this and many other problems, infeasible solutions are often generated by both crossover and mutation operators. Some authors have advocated the use of non-standard forms of crossover and mutation in order to guarantee that only feasible solutions are produced. We have chosen to use the the standard genetic operators, and with this restriction there are three ways in which the problem can be dealt with. One method is to use some penalty function to drive the solutions towards feasibility. A second approach is to force each string generated to correspond to a feasible solution. Thus after crossover and mutation have been carried out a correction routine needs to be invoked which adjusts the string so that it becomes feasible. The third approach is similar and can be described as a decoding of the string. The individual string remains unaltered in the population, but it is decoded using rules which guarantee a feasible assignment. Thus the genetic information within an infeasible string remains present in the population, but this is not expressed in the current individual. For this problem, we have carried out some limited experiments using these different methods, and the penalty approach appears to be the most effective.

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 will include an element corresponding to the total time for the operations assigned to the slowest station. In addition, as mentioned above, we also wish to assign some penalty cost to any solutions which are infeasible because of precedence constraints. In fact we define the value of a solution as

$$\max_i(S_i) + kN_v$$

where S_i is the total time for operations assigned to station i, N_v is the number of precedence violations and k is a constant which we set equal to the largest single operation time.

Given this definition we now wish to minimize the objective function. Notice that the value of the objective is not a measure of fitness, since we consider the maximization of fitness not its minimization. There are obviously a number of ways in which we can proceed. One approach is to take as the fitness value some large constant minus the objective value. The constant is chosen in order that the fitness values are positive. A second idea is to take fitness as the reciprocal of the objective value, assuming that all objective values are positive. Yet another technique is to choose the fitness by

$$f_i = \exp(-hv_i)$$

where h is chosen to make the fitnesses lie in a particular range. The drawback of both the latter schemes is that they alter the relative fitnesses of different individuals in quite a complicated way, while the first scheme involves the choice of an arbitrary constant. These difficulties can be overcome by taking into account the scaling of fitness values more explicitly. Many researchers have recognized the usefulness of a scaling procedure in order to achieve some degree of control over the speed of convergence of a genetic algorithm.

Suppose we are given the values of our current individuals and we wish to determine a fitness distribution. If we have a maximization problem, we define the fitness of individual i, f_i, as the value of the objective for that individual v_i. If we have a minimization problem, then we take as the fitness, $-v_i$. This has the advantage of making as little change as possible to the structure of the original objective function.

We now perform a linear scaling on the values, with the aim of producing a fitness distribution, F_i, $i = 1, \ldots, N$ with the following properties:

1. $\sum_{i=1}^{N} F_i = 1$
2. $\max_i F_i = \lambda \sum_{i=1}^{N} F_i/N$
3. $F_i \geq 0$ for all i

where λ is a scale factor chosen in order to achieve a particular speed of convergence. It is not always possible to achieve all three aims, in which case we choose to vary the choice of the scale factor λ.

Define the linear scaling by $F_i = mf_i + c$. We use the first two criterion above in order to choose m and c. We obtain

$$m = \frac{\lambda - 1}{N(\max_i f_i) - \sum_{i=1}^{N} f_i}$$

and

$$c = \frac{N(\max_i f_i) - \lambda \sum_{i=1}^{N} f_i}{N(N(\max_i f_i) - \sum_{i=1}^{N} f_i)}$$

However, this will not guarantee that each F_i is nonnegative. We choose to modify the scale parameter λ in such a way as to guarantee that all F_i are nonnegative. We therefore require

$$\frac{(\lambda - 1)f_k}{N(\max_i f_i) - \sum_{i=1}^{N} f_i} + \frac{N(\max_i f_i) - \lambda \sum_{i=1}^{N} f_i}{N(N(\max_i f_i) - \sum_{i=1}^{N} f_i)} \geq 0$$

for all k. A simple calculation shows that this gives the choice of λ as

$$\lambda = \min\{\lambda_g, \lambda_*\}$$

where λ_g is the given value of scaling required, and λ_*, the maximum value possible for λ, is

$$\lambda_* = \min_k \left\{ \frac{(\max_i f_i) - f_k}{\sum_{i=1}^{N} f_i/N - f_k} \;\middle|\; f_k \leq \sum_{i=1}^{N} f_i/N \right\}.$$

We have carried out some limited experiments comparing this approach to the problems of fitness and scaling with some other (more complicated) approaches. We found it to be more effective than the other methods we tried and we have used this method in all the experiments we report here.

We also tried a number of different methods to select individuals for the mating pool. The underlying approach is to select an individual with a probability proportional to its fitness. Thus each time a selection is made individual i is chosen with probability given by F_i, the scaled fitness value. It is found to be helpful to alter this simple approach in such a way as to limit the variability inherent in this scheme. It is better to ensure that any individual with above average fitness is automatically selected. The method we use to achieve this is called Stochastic Universal Sampling and is described in detail by Baker[5]. In brief, the procedure can be described as follows. Take the members of the population and reorder them randomly. Assign to each individual an interval proportional to its fitness, and scaled so that the total length of all the intervals is N. Consider the intervals laid end to end along the line from 0 to N. Choose a random number x uniformly on $[0, 1]$ and put the individuals corresponding to the intervals in which the points $x, x + 1, x + 2, ...x + N$ lie into the mating pool. Pairs of individuals are then removed from the mating pool and recombined using crossover and mutation. We made some comparisons with another commonly used method called "remainder stochastic sampling without replacement", and found this to be similar in performance to Stochastic Universal Sampling.

Crossover and mutation

We turn next to consider the operations of crossover and mutation, which together make up the process of recombination. 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[16]. For our problem, however, these approaches do not appear to lead to any substantial improvements over the more standard crossover mechanism described in the introduction. When a pair of individuals has been selected for mating, crossover occurs at a single random point with probability p_c, and with probability $1 - p_c$ the offspring are identical to the parents.

Following mating the two offspring produced then undergo mutation. Though we have carried out some experiments with different forms of mutation, there seems to be little, if any, disadvantage in adopting the standard approach as it appears in the literature of genetic algorithms. This involves the random change of allocations of operations to stations. For each operation in turn with some small probability, p_m, we change the station to which it is assigned either to the station immediately before it, or to the station immediately after it. Thus we change particular elements in the string by plus or minus one. Actual implementation of this mutation operation is most efficiently carried out by generating a random variable to determine the next position on the string to which mutation will be applied.

There are two ways in which we have adjusted the standard procedures for reproduction. Firstly we have inserted into a random position in the population an individual from the previous generation with the best fitness. This has the effect of guaranteeing that the final generation contains the best solution ever found, which is desirable from the point of view of assessing the relative performance of different versions of the genetic algorithm. This procedure, called elitism, has been used by a number of other researchers. The second change we have made has, as far as we are aware, not been investigated previously. When offspring are produced after crossover and mutation (or just one of these), if either of the offspring has a worse fitness than the worst member of the previous generation then that individual is not retained and instead one of the parents is allowed to continue unchanged into the next generation. The effect of this is that the worst member of the population is never worse than the worst in the previous generation. Our experiments show that this can substantially improve the convergence behavior of the algorithm. Clearly by forcing the worst individual in each generation to be no worse than that of the previous generation we will speed up the convergence of the algorithm. In some circumstances this would not be an advantage - if convergence is too fast we are likely to end up at a poor quality solution. In this case, however, we can adjust the rate of convergence using the scaling mechanism described above, so that the overall speed of convergence is unchanged. This turns out to give a significant net benefit in terms of the algorithm performance.

2 Experimental results from the serial implementation

We have carried out a number of experimental tests. First to find appropriate settings for the parameters involved in the genetic algorithm, and second to assess the effectiveness of the genetic algorithm in comparison with other approaches. There are a number of difficulties involved in carrying out such experiments and it may be helpful to make some general comments before describing our results in detail.

First note that we have not made any attempt to optimize the code used to implement the genetic algorithm. There are a number of areas where we could expect to improve on computation times by using relatively more sophisticated programming techniques. Our aim in these tests is to find good solutions with a relatively small number of function evaluations and we will look only at the total number of generations in the genetic algorithm rather than the computation time. In fact for this problem the work in function evaluation (which includes counting the number of violated constraints) was in any case the dominant part of the computation.

Second we should note that it is important to carry out comparisons between different methods only when the number of generations used is roughly the same. By changing the scaling factor it is possible to obtain convergence in a greater or lesser number of generations. Runs in which a large number of generations were necessary to achieve convergence will typically have considered many more individual solutions and thus are, for this reason alone, likely to do better than runs which converge more quickly, if all runs are allowed to continue till convergence. There is a danger that some change to the parameter settings appears to be making an improvement when in fact it is a change in the speed of convergence. Thus we choose to make comparisons by fixing the number of generations in a run and looking at the best individual in the final population. We can use the scaling parameter to adjust the speed of convergence. Our experimental results, reported below, show that it is best to adjust the scaling factor to ensure convergence at about the same time as the run will come to an end.

To understand the reason for this it is helpful to see the behavior of the algorithm shown graphically. The graphs in Figure 2 show how the value of the worst and the best in the population plotted against generation number for three different scaling factors. These are taken for single runs of the algorithm on one problem, but are reasonably representative. One can see that with unlimited computational time a low scaling factor would be best, but if runs were to be stopped after 200 generations, for example, it would be best to choose a scaling factor leading to convergence at about this point or a little later.

Thirdly, it is not clear how the genetic algorithm population should be initialized. We experimented with two kinds of starting population. 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 the second scheme we generated a set of initial solutions using a method due to Arcus[3]. 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 the GA and in the other cases the best of the Arcus solutions is never far from the best solution found. Interestingly, however, the performance of the GA scheme was not as good from this preselected starting population as it was from a random start. 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. Consequently the results we report here all start with a random initial population.

Experiments were performed on randomly generated problems having 50 operations; these are to be assigned to 5 stations. Problems were generated so that operation times are uniformly distributed

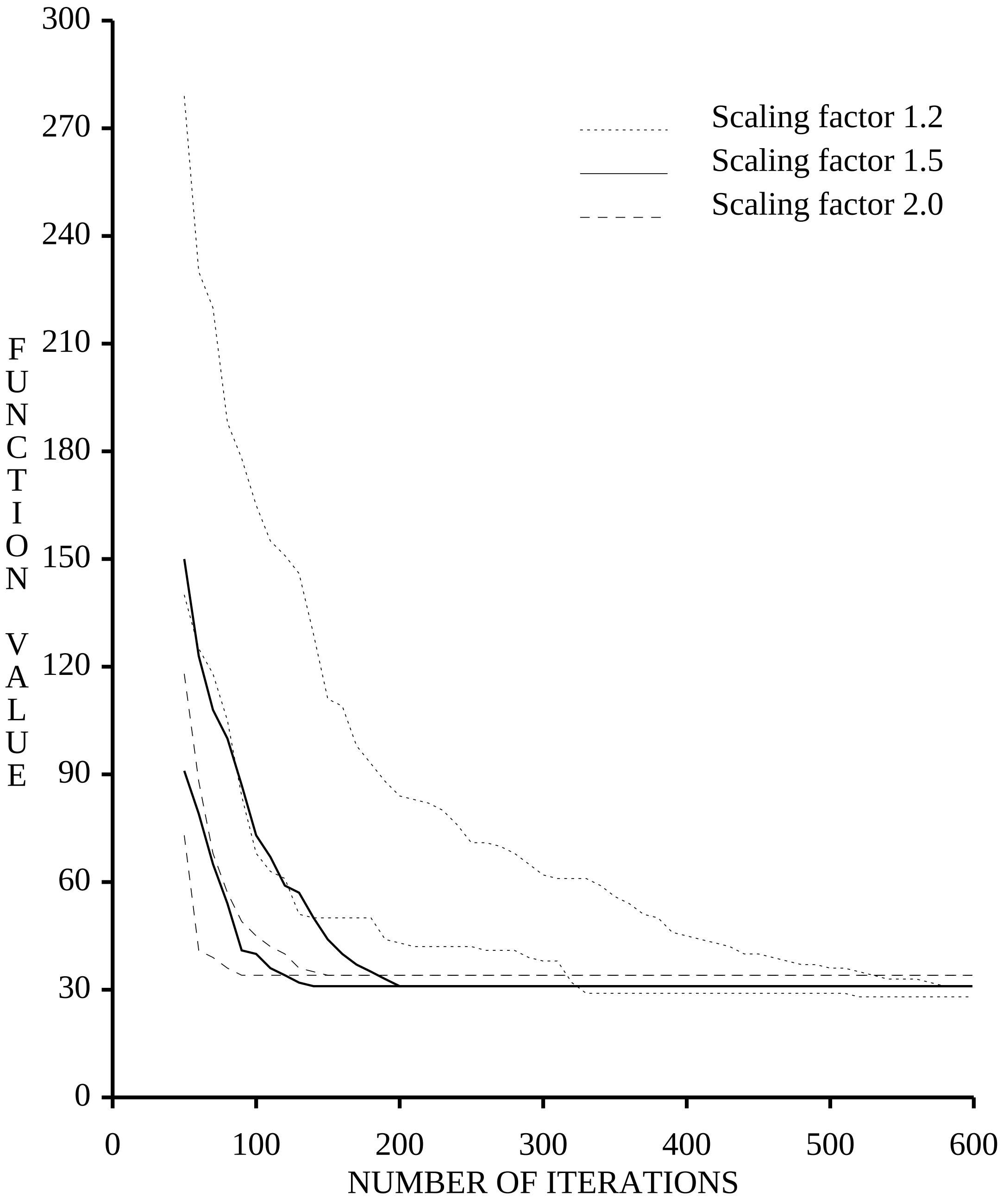

Figure 2: Effects of scaling

integers between 50 and 500. We construct the precedence relations using the upper triangular part of a matrix W. Task i precedes j if and only if $W(i,j) \neq 0$, where W is a matrix determined as follows. W has $\frac{n(n-1)}{2} * \rho$ nonzero entries, where ρ is the density of precedence, normally set to 0.2. Two distinct indices from a uniform distribution are generated and the corresponding entry of the upper triangular part of W is made nonzero. This process is repeated until W has the correct number of entries.

We begin by describing the results of experiments we carried out to find appropriate values for the three parameters: probability of crossover p_c; probability of mutation p_m; and desired scale factor λ_g. We arbitrarily chose 350 generations as the limit for each run (the parameter values that we choose would be markedly different if the length of a run was altered). We have not carried out extensive tests on populations with different sizes; all our experiments have used a population size of 64. What little experimentation we have carried out leads us to believe that this is a reasonable value for the size of problem we have tackled. It is also appropriate for the parallel implementation for which we use a hypercube architecture with 32 processors.

The results we obtained are shown in Tables 1, 2 and 3 for probabilities of crossover of 0.6, 0.7 and 0.8 respectively. The probability of mutation (of each element of the string) is chosen from the set of values 0.005, 0.01, 0.015, 0.02, 0.025, 0.030, 0.035 and 0.04. The scaling factor is chosen from the set of values 1.1, 1.2, 1.3, 1.4, 1.5, 1,6, 1,7, 1.8 and a value which tries to force a scaling of 2.5 (labeled big, since in many cases this scaling is not achieved due to the non-negativity requirement). We chose to carry out experiments using 5 randomly generated problems, for each of which we carried out 8 runs of the genetic algorithm using different random number seeds. For each set of parameter values we thus have the results of 40 runs. These are summarized in the tables by giving: (a) the average of the percentage deviation of the best solutions in the final generation from the best solution ever found for each problem; (b) the average percentage of individuals in the final population which had the same value as the best in that generation; and (c) the first generation at which an average of 90% or more of the individuals in the population have the same value as the best in the population (or left blank if this still has not occurred by generation 350).

Hence the first number given is a measure of the quality of the solutions obtained, while the second and third numbers reflect the speed of convergence. To make the tables easier to read we have omitted entries for parameter values which lead to convergence either much earlier than generation 350 or much later. To be precise we have left out entries either where convergence (measured by 90% of the individuals having the same value) occurs before generation 150, or where, on average, the number of individuals in the final population having the same (best) value is less than 3.

Interpretation of these results is not entirely clearcut. Notice that the lowest value that occurs in any table is that of 1.34 with $p_m = 0.04$, $p_c = 0.7$ and scaling factor "big". The use of a big scaling factor, however, can lead to erratic results and for this reason we would recommend the values $p_m = 0.03$, $p_c = 0.6$, scale factor = 1.8. if runs are stopped after 350 generations. These choices are not particularly critical. Notice that this represents a very much higher level of mutation than has been recommended in previous studies. Since each individual is represented by a string of length 50, the probability that an individual does not receive any mutation is 0.282 with this value of p_m. The reason that we can use such a large probability of mutation is that towards the end of the run many of the mutations generate individuals worse than the worst current individual, and are thus ignored. In effect our method benefits from a higher mutation rate early in the run, without this interfering with the convergence of the algorithm at the later stages. This approach thus lends support to the suggestion (see for example [10]) that a reduction in mutation during the course of a run may be beneficial, in a similar way to the reduction in "temperature" which is used in simulated annealing methods.

Table 1: $p_c = 0.6$

—	-	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	big
	a	17.37	24.25	33.10	37.23					
0.005	b	15.39	99.02	100	100	*	*	*	*	*
	c	—	254	191	157					
	a	17.49	5.03	8.83	11.53	11.09				
0.010	b	3.48	27.27	100	100	100	*	*	*	*
	c	—	—	264	207	176				
	a		9.19	5.20	4.83	7.72	9.63	10.57		
0.015	b	*	3.67	28.05	99.80	100	100	100	*	*
	c		—	—	276	232	182	165		
	a			6.34	2.72	2.40	3.64	4.04	4.61	4.76
0.020	b	*	*	4.02	19.57	81.52	99.45	100	100	100
	c			—	—	—	269	235	226	214
	a			10.01	5.93	3.83	2.50	3.02	2.54	3.35
0.025	b	*	*	3.24	4.61	14.14	77.73	94.41	99.88	98.36
	c			—	—	—	—	320	279	279
	a				9.64	5.80	2.84	1.90	1.68	1.68
0.030	b	*	*	*	3.01	4.96	9.34	35.70	64.38	90.70
	c				—	—	—	—	—	349
	a					10.03	5.46	3.75	2.31	2.34
0.035	b	*	*	*	*	3.36	4.49	9.18	22.19	49.96
	c					—	—	—	—	—
	a						6.74	6.46	4.16	1.75
0.040	b	*	*	*	*	*	3.75	4.18	7.15	26.18
	c						—	—	—	—

Table 2: $p_c = 0.7$

—	-	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	big
	a	13.00	18.27	28.07						
0.005	b	26.68	100	100	*	*	*	*	*	*
	c	—	241	186						
	a	18.37	6.37	8.56	12.19	17.88				
0.010	b	3.67	32.77	100	100	100	*	*	*	*
	c	—	—	256	189	169				
	a		8.03	3.51	5.17	7.40	8.98	10.16		
0.015	b	*	3.52	36.52	97.70	100	100	100	*	*
	c		—	—	286	217	178	152		
	a			6.70	4.91	2.20	3.47	4.50	5.54	6.40
0.020	b	*	*	4.41	31.84	92.81	99.80	100	100	100
	c			—	—	340	267	233	226	218
	a				5.10	3.05	3.21	3.34	2.60	2.52
0.025	b	*	*	*	4.80	18.52	73.01	96.13	99.57	100
	c				—	—	—	333	285	255
	a				6.43	4.45	3.31	2.48	2.17	2.09
0.030	b	*	*	*	3.24	5.23	11.72	42.77	68.98	91.84
	c				—	—	—	—	—	359
	a					6.14	3.71	3.00	2.16	1.93
0.035	b	*	*	*	*	3.32	5.00	11.80	21.95	63.32
	c					—	—	—	—	—
	a						5.74	4.68	2.60	1.34
0.040	b	*	*	*	*	*	3.67	4.57	8.88	28.36
	c						—	—	—	—

Table 3: $p_c = 0.8$

—	-	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	big
	a	14.10	15.55	20.11	31.48					
0.005	b	35.98	100	100	100	*	*	*	*	*
	c	—	239	183	150					
	a	11.90	5.59	5.77	9.03	16.15				
0.010	b	4.06	61.52	100	100	100	*	*	*	*
	c	—	—	245	191	169				
	a		7.56	3.12	4.96	4.81	6.87	9.12		
0.015	b	*	3.83	50.98	100	100	100	100	*	*
	c		—	—	271	215	187	161		
	a			4.63	1.72	3.68	3.64	4.32	3.21	3.76
0.020	b	*	*	5.23	37.85	99.92	97.58	100	100	100
	c			—	—	290	269	229	224	204
	a			7.33	3.44	2.09	2.91	2.48	3.11	2.41
0.025	b	*	*	3.05	5.12	24.69	81.76	97.66	95.82	97.70
	c			—	—	—	—	305	281	262
	a				6.73	4.92	2.92	2.50	2.25	2.42
0.030	b	*	*	*	3.32	5.74	16.68	45.63	61.88	92.54
	c				—	—	—	—	—	354
	a					4.94	3.27	3.76	2.24	2.15
0.035	b	*	*	*	*	3.32	5.20	8.83	19.69	63.63
	c					—	—	—	—	—
	a						3.78	3.21	3.23	1.86
0.040	b	*	*	*	*	*	3.55	5.47	7.70	28.09
	c						—	—	—	—

Table 4: selection only, replace if improved

—	1.05	1.07	1.1	1.12	1.15	1.18	1.2	1.3
0.005	490.65	447.01	378.95	332.85	308.20	292.73	253.61	212.30
0.010	268.88	218.14	172.74	179.97	141.83	139.43	117.20	115.97
0.015	149.97	112.30	89.25	79.53	70.24	72.81	58.28	69.00
0.020	92.77	72.72	50.94	44.59	42.59	36.01	42.45	46.83
0.025	64.41	52.89	40.42	35.01	24.08	26.80	35.16	42.13
0.030	46.18	38.45	30.84	30.56	23.92	27.77	22.69	39.21
0.035	39.58	32.30	26.57	22.52	20.61	26.34	27.65	27.61
0.040	38.02	28.21	23.62	25.06	17.92	23.79	21.01	23.98
0.045	32.37	24.89	19.09	17.49	18.01	21.50	26.24	22.78
0.050	35.74	26.61	21.72	18.20	15.12	19.03	18.29	25.34
0.055	28.40	24.51	19.39	18.86	22.07	16.38	21.42	25.81
0.060	33.66	23.77	18.88	16.85	15.15	22.72	19.44	21.45
0.065	31.72	20.25	18.00	17.03	18.11	20.45	18.10	18.44
0.070	31.71	25.36	19.92	16.44	18.45	19.30	19.46	20.25
0.075	34.54	26.41	16.82	21.42	18.60	16.52	17.96	23.56

The genetic algorithm involves a selection operator which can be introduced without the use of crossover. We carried out some experiments to estimate the proportion of the improved performance of the genetic algorithm, compared with straightforward local search, which was accounted for by the selection operator. There are two implementations of this approach which are of interest. First we can carry out selection on a population in which mutation only takes place if the previous solution is improved. This is similar to a simple local search technique, except that we throw away some solutions that are not progressing well and replace them with copies of solutions which are progressing better. There are now two parameters to choose, p_m and the scale factor λ_g. Table 4 gives the percentage deviation from optimality for various choices of these parameters using this scheme. The best choices of these were found to be $p_m = 0.05$ and $\lambda_g = 1.15$ at which values the best solution after 350 generations was on average 15% away from the best ever found. Note that the percentage deviation from optimality is much greater than in Tables 1, 2 or 3.

An alternative implementation of the same basic idea is to allow the mutation operator to generate a solution which is worse than the current solution, and use the selection operator to obtain improvements in the overall quality of the solutions represented in the population. This is closer to the spirit of the genetic algorithm and allows the possibility of jumping out of a local optimum. In implementing this approach we incorporated the elitist approach of inserting the previous best solution back into the population, and also the elimination of any mutations which produced a solution worse than the worst member of the previous population. With this implementation the values of p_m and λ_g which worked best were 0.01 and 2.0. With these parameter settings we then achieved best solutions after 350 generations which were on average 32% away from the best ever found (see Table 5).

We note that the results given above support the conclusions given in [18] that the combination of selection and crossover is far more effective for optimization than an exclusive use of either technique. The choice of a suitable weighting of these operators may be problem specific, but their combined usage is of great importance in producing an effective method for the given problem.

Table 5: selection only, always replace

—	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0	2.1	big
0.005	345.3	235.5	183.7	127.0	119.1	112.4	106.2	97.2	100.5	109.6	105.0
0.010	295.5	167.2	97.6	72.0	49.9	46.8	41.9	45.0	32.2	39.0	42.5
0.015	299.1	166.1	116.0	70.0	51.0	39.5	34.1	44.8	37.1	35.2	37.4
0.020	335.4	218.4	141.7	85.9	65.4	54.4	46.2	43.8	43.9	45.3	39.4
0.025	326.8	229.8	165.1	116.5	94.1	68.6	61.3	62.1	57.4	52.3	53.4
0.030	372.5	259.3	191.7	142.6	114.0	87.2	75.3	74.6	67.7	64.0	65.3
0.035	393.4	268.1	211.8	163.9	129.6	107.1	90.9	89.7	88.7	90.2	86.1
0.040	423.1	323.3	234.0	186.3	151.0	127.7	120.7	108.5	104.5	105.7	97.3

3 A Parallel Implementation Using 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. A genetic algorithm would appear to be ideal for parallel architectures since evaluation and reproduction can be performed concurrently. 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. For a parallel implementation, we assume that each individual in the population resides on a processor (with one or more individuals assigned to each processor) and communication is carried out by message passing. For descriptions of other implementations of parallel genetic algorithms see the references[11, 19, 24, 26, 28].

We consider first the mechanisms that would be involved in an effective parallelization of the genetic algorithm we have described above. First of all, a global sum operation should be used to compute the sum of all the scaled fitness values, after which each individual could compute its scaled fitness value. Using a parallel prefix operation, the partial sums of the scaled fitness values could be calculated on each processor. These values could be used by the processors in order to sample the individuals residing on that processor. The mating pool is thus formed but is distributed over all the processors. The pairs of individuals used for mating are then selected from this pool by some process of random requests. In the serial case, this is achieved by randomly permuting the mating pool and then taking pairs of individuals from the top of the pool. In the parallel case, the permutation could be generated in parallel so that the individuals required (for mating) at a given processor would be known. Each processor would also need to know the processor holding the ith entry of the mating pool so that it could request this individual from that processor. This information could be determined by an all-to-all personalized communication (in fact just the number of individuals in the mating pool residing on each processor is needed).

Though the parallelization outlined above is clearly feasible, there are obvious communication costs and some difficulties in specifying the precise way in which some parts of the process are carried out. We have chosen instead to make comparisons with a different form of genetic algorithm using a parallel scheme in which each individual resides at a "location". In the sequel, we shall associate the location and the (unique) individual that is residing there. We need to introduce the notion of a location because, in practice, the computations required by each individual will be carried out on a single processor. It is convenient, however, to distinguish between processors and locations because a single processor may handle more than one individual. Each location is connected to a small subset of the other locations which we will call its neighborhood. The assignment of locations to processors may be optimized to reduce communication costs. Though this approach will mean some substantial changes in the basic algorithm, it does lead to a method which is very well suited to parallel processing. Recently, this type of approach has been successfully used by a number of researchers, see for example the work of H. Mühlenbein and M. Gorges–Schleuter who have developed the ASPARAGOS system to implement an asynchronous parallel genetic algorithm[11, 24].

A model algorithm for a scheme in which fitness information is only compared locally is displayed in Exhibit II.

Exhibit II . Synchronous 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 the neighborhood based on fitness
        request individuals j and k
        synchronize
        reproduce using individuals j and k
        remove individual i from its location and
        replace it with one of the offspring of j and k
until population variance is small
```

The first question that arises is how to define the neighborhood of each location. This may be dictated to some extent by the architecture of the parallel processor – we will wish to avoid passing messages which require transmission through large numbers of processors. We have carried out experiments with a number of neighborhood structures[2] without coming to a firm conclusion. In the experiments that we report below we have used a neighborhood structure we call *ring8*. This is defined by stating that locations i and j are considered neighboring if

$$i \in nbd(j) \iff |i-j| \leq 4$$

This can be viewed as every location being on a ring with neighboring locations no further than four links away. Strictly then we need to replace $|i-j|$ by

$$\min(|i-j|, |i+N-j|, |i-N-j|).$$

The neighborhood size is eight.

In the form given above, for each location the algorithm selects two individuals j and k from the neighborhood of that location based on fitness. In order to reduce communication costs it is possible to make one of these selections be the individual already residing at the location. In this case, we only need to select one individual from the neighborhood with which to mate. We have carried out some experiments which show that this technique performs at least as well and frequently better than the original scheme, and we have adopted it for all the results reported below. Since we only need to select one individual at each location the methods we discussed in the context of a serial implementation are no longer relevant and selection is carried out in the simplest way so that individual i is chosen from the neighborhood with probability given by

$$f_i / \sum_{j \in nbd(i)} f_j.$$

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? We have carried out some experiments in order to answer this[2] and it seems that the performance of the algorithm is not degraded by throwing the less fit offspring away. This is the policy which is adopted in the experiments reported below.

This method effectively deals 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 asynchronous scheme (see Exhibit III) has proven very effective in practice.

Exhibit III . Asynchronous 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.

4 Experimental Results for the Parallel Implementation

Our aim in this section of the paper is to compare the performance of the parallel implementation of the algorithm with a serial version. First we try to understand the behavior of the synchronous parallel algorithm by implementing this version on the Intel Hypercube iPSC/2 with 32 processors. This will enable us to assess the effect of moving to a parallel implementation, which as we have already seen is quite different in the way that selection is carried out. We tested the procedure on the same set of 5 test problems that were used in the serial implementation. For the ALBP, using

Table 6: *synchronous*, $p_c = 0.7$

—	-	1.10	1.20	1.30	1.40	1.50	1.60	1.70	1.80	big
	a	26.78	32.40	29.38	32.80	30.49	32.30	33.44	35.15	40.57
0.005	b	95.32	95.64	98.20	99.28	99.64	99.52	100	100	100
	c	122	94	94	84	86	72	73	69	36
	a	22.70	22.27	22.44	24.57	24.08	26.17	28.89	27.14	32.72
0.010	b	95.20	95.56	95.80	96.24	96.84	98.04	98.44	98.44	100
	c	157	151	142	124	116	120	100	101	53
	a	13.80	16.81	17.70	17.76	17.72	22.34	20.08	20.57	27.81
0.015	b	75.00	82.56	79.00	89.68	93.48	95.04	97.28	96.76	100
	c	—	—	—	—	207	162	143	139	73
	a	10.86	12.10	12.55	14.36	13.54	14.72	15.27	15.98	23.50
0.020	b	67.44	67.72	70.80	75.80	80.88	75.16	84.36	85.36	95.48
	c	—	—	—	—	—	—	—	—	98
	a	8.40	10.32	10.22	9.03	12.10	11.98	12.83	12.78	19.24
0.025	b	45.24	47.92	50.16	59.72	64.32	64.64	68.48	70.64	91.64
	c	—	—	—	—	—	—	—	—	115
	a	7.98	7.83	7.39	8.66	8.64	7.82	9.88	10.04	17.29
0.030	b	16.64	26.56	35.04	35.56	45.28	44.48	54.68	57.60	81.24
	c	—	—	—	—	—	—	—	—	—
	a	7.13	8.65	8.26	7.71	7.75	7.84	7.04	7.60	13.23
0.035	b	3.84	5.72	6.00	9.80	12.08	14.96	19.76	30.36	73.20
	c	—	—	—	—	—	—	—	—	—
	a	8.17	8.85	7.73	7.24	7.47	7.15	7.67	6.96	12.10
0.040	b	3.84	4.08	3.84	5.00	6.28	6.44	7.88	11.72	62.96
	c	—	—	—	—	—	—	—	—	—

the notion of neighborhoods, the computational timings are entirely dominated by the evaluation of fitness values which includes a check on the feasibility of newly generated solutions.

As for the serial case it is necessary to set three parameters. Similar tests were carried out to those reported in tables 1, 2 and 3. The best choice of the probability of crossover turned out to be 0.7 and Table 6 gives the results obtained with this value. The only difference to the experimental arrangement and that for the serial experiments relates to the population size in a run which is here set to 32. This is only half the number used in the serial implementation as at each generation a total of $2N$ individuals are looked at, two at each processor. Thus using half the population size will imply the same number of function evaluations (leaving aside any consideration of the saving in function evaluations when an individual is passed through to the next generation without undergoing either crossover or mutation).

It seems that good choices for the parameters are $p_m = 0.035$ and a scale factor of 1.7. The results, however, are not as good as for the sequential algorithm. Notice that the convergence is substantially slowed with many fewer cases of complete convergence. As we have pointed out earlier, premature convergence can be detrimental to the quality of the final solution. It is also clear from these results that the performance of the parallel algorithm is less sensitive to the correct choice of scaling factor than the serial algorithm.

Table 7: *asynchronous*, $p_c = 0.7$

—	-	1.10	1.20	1.30	1.40	1.50	1.60	1.70	1.80	big
	a	24.72	28.05	25.21	22.65	26.31	28.29	30.62	32.61	36.23
0.005	b	93.38	96.62	98.00	98.24	99.32	99.28	100	100	100
	c	131	124	111	114	105	88	81	70	36
	a	20.23	20.01	19.49	17.52	20.14	20.37	24.91	25.41	33.27
0.010	b	91.26	93.58	94.88	96.28	96.64	98.68	98.88	100	100
	c	—	162	140	137	108	122	109	111	56
	a	12.51	15.92	18.73	16.51	16.71	22.68	21.48	19.51	28.18
0.015	b	72.04	80.62	80.04	87.62	90.40	94.82	97.44	98.72	100
	c	—	—	—	—	—	173	163	125	87
	a	9.88	10.01	11.58	12.49	12.51	13.02	16.22	16.90	24.08
0.020	b	67.40	68.78	70.02	72.08	76.82	76.84	80.32	83.62	97.58
	c	—	—	—	—	—	—	—	—	110
	a	9.45	9.36	11.02	8.41	10.19	12.67	13.78	13.73	20.25
0.025	b	40.42	42.98	48.22	53.28	60.34	64.88	69.08	70.04	89.46
	c	—	—	—	—	—	—	—	—	127
	a	6.78	7.89	7.99	8.91	8.31	7.43	6.34	9.03	15.21
0.030	b	17.44	28.52	31.40	38.50	40.24	44.56	52.86	55.06	79.40
	c	—	—	—	—	—	—	—	—	—
	a	6.42	7.25	6.29	5.80	7.35	7.62	6.56	5.98	13.33
0.035	b	2.88	4.76	8.08	10.82	11.56	14.04	18.66	32.66	72.02
	c	—	—	—	—	—	—	—	—	—
	a	8.62	8.98	7.93	6.94	6.42	7.15	7.82	7.06	10.15
0.040	b	2.84	3.08	4.82	5.02	6.22	6.48	7.98	12.76	60.60
	c	—	—	—	—	—	—	—	—	—

The asynchronous scheme has also been implemented on the Intel Hypercube iPSC/2. The behavior of this version of the algorithm is consistently better than the synchronous scheme. The corresponding results are given in Table 7. Nevertheless, the quality of the solutions remains worse than the best results obtained using the serial algorithm.

5 Conclusions

There are a number of conclusions that can be drawn from these experiments. We begin by noting that we have been quite successful with a version of the "standard" form of the genetic algorithm. For example we have not tried to combine ideas from genetic algorithms with local search procedures. The significant changes that we have made to the usual genetic algorithm are as follows. First we reject individuals who would decrease the fitness of the worst member of a population. Second we carefully control scaling to ensure the best behavior for the algorithm and finally we use much higher rates of mutation than has been usual in previous work. We recommend that others wishing to implement a genetic algorithm for combinatorial optimization problems adopt all three of these measures.

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.

The experimental parallel implementation that we use shows that it is still important within this framework to allow the GA sufficient time to converge. Certainly, it is better to consider a smaller population over a longer time span than a larger population over a shorter span. The local neighborhood scheme does slow convergence which may be a desirable feature if very good solutions are required. Furthermore, the convergence of the asynchronous scheme is slightly slower and in general leads to better solutions.

Acknowledgement

The authors would like to thank Mr. Menglin Cao for his help in testing the algorithm. This material is based on research supported by NSF Grant CCR–9157632 and Air Force Office of Scientific Research Grant AFOSR–89–0410.

References

[1] E.H.L. Aarts, P.J.M. van Laarhoven, and N.L.J. Ulder, 1991. Local-search-based algorithms for job shop scheduling. CQM-Not 106, Centre for Quantitative Methods, Nederlandse Philips Bedrijven B.V., P.O. Box 218, NL-5600 MD Eindhoven.

[2] E.J. Anderson and M.C. Ferris, 1990. A genetic algorithm for the assembly line balancing problem. In *Proceedings of the Integer Programming / Combinatorial Optimization Conference, Waterloo, Ontario, Canada, May 28–30.* University of Waterloo Press.

[3] A.L. Arcus, 1966. 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.

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

[5] J.E. Baker. Reducing bias and inefficiency in the selection algorithm. In Grefenstette [15], pp. 14–21.

[6] R.K. Belew, editor, 1991. *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Inc., San Mateo, California.

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

[8] L.D. Davis, 1990. *The Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

[9] R.W. Eglese, 1990. Simulated annealing: A tool for operational research. *European Journal of Operations Research* 46, 271–281.

[10] T.C. Fogarty, 1989. Varying the probability of mutation in the genetic algorithm. In Schaeffer [25], pp. 422–427.

[11] M. Georges-Schleuter, 1989. Asparagos: An asynchronous parallel genetic algorithm. In Schaeffer [25], pp. 416–421.

[12] F. Glover, 1989. Tabu search – part 1. *ORSA Journal on Computing* 1, 190–206.

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

[14] B.L. Golden and C.C. Skiscim, 1989. Using simulated annealing to solve routing and location problems. *Naval Research Logistics Quarterly* 33, 261–279.

[15] J.J. Grefenstette, editor, 1987. *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

[16] J.J. Grefenstette, 1987. Incorporating problem specific knowledge into genetic algorithms. In Davis [7].

[17] A. Hertz and D. de Werra, 1987. Using Tabu search techniques for graph coloring. *Computing* 29, 345–351.

[18] P. Jog, J.Y. Suh, and D. Van Gucht. The effects of population size, heuristic crossover and local improvement on a genetic algorithm for the travelling salesman problem. In Schaeffer [25], pp. 110–115.

[19] P. Jog, J.Y. Suh, and D. Van Gucht, 1991. Parallel genetic algorithms applied to the travelling salesman problem. *SIAM Journal on Optimization* 1, 515–5291.

[20] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon, 1989. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research* 37, 865–892.

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

[22] K.A. De Jong, 1975. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan. Dissertation Abstracts International 36(10), 5140B.

[23] M. Lundy and A. Mees, 1986. Convergence of an annealing algorithm. *Mathematical Programming* 34, 111–124.

[24] H. Mühlenbein, 1989. Parallel genetic algorithms, population genetics and combinatorial optimization. In Schaeffer [25], pp. 416–421.

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

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

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

[28] R. Tanese, 1987. Parallel genetic algorithms for a hypercube. In Grefenstette [15], pp. 177–183.

[29] D. Whitley, T. Starkweather, and D. Fuquay, 1989. Scheduling problems and travelling salesmen: The genetic edge recombination operator. In Schaeffer [25], pp. 133–140.