A Parallel Genetic Algorithm for Solving the School Timetabling Problem

D. Abramson □
J. Abela

High Performance Computation Project
Division of Information Technology
C.S.I.R.O.
723 Swanston St,
Carlton, 3053
Australia.

15 Australian Computer Science Conference, Hobart, Feb 1992.

ABSTRACT

Genetic algorithms (GA) have been applied to a number of optimisation problems with some success [1]. The algorithms mimic the process of natural selection, with the effect of creating a number of potentially optimal solutions to some complex search problem. One of the major disadvantages of genetic algorithms is that they are very slow. In this paper we discuss the application of a GA to the school timetabling problem, and show how the execution time can be reduced by using a commercial shared memory multiprocessor. The paper reports some results from the sequential execution of the algorithm as well as the speedup attained from the parallel solution.

1 GENETIC ALGORITHMS

Genetic algorithms (GA) mimic the process of natural selection and can be used as a technique for solving complex optimisation problems which have very large search spaces. Unlike many heuristic schemes which only have one sub-optimal solution at any time, GAs maintain many individual solutions in the form of a population. Individuals (parents) are chosen from the population and are then *mated* to form a new individual (a child). The child is further mutated to introduce diversity into the population.

In natural selection, individuals are born with various attributes of their parents, and therefore inherit properties of their parents. Normally these attributes are indicated by particular genes, which are combined into chromosomes. The attributes of a child may be further modified by mutation, which alters the chromosomes to give new features not found in either parent. If the combined attributes of a child make it more suited to the environment, then the probability that the child will survive, and breed, is increased. Consequently, the desirable attributes of the child will be passed on to future generations. However, if a child is born with undesirable attributes, which make it less able to handle the environment, then it is likely that the child will die before reproducing, and the less desirable attributes will be removed from future generations. The net effect of *survival of the fittest* is that the average fitness of the population increases with each generation. Because of mutation there is also diversity in the population, allowing new and better attributes to be created.

The simulation of evolution allows the principle of survival of the fittest to be applied to other optimisation problems, where the goal is to find an individual which is very fit. For example, if the optimisation problem requires a schedule to be created in which a teacher does not appear more than once at any given time in a timetable, then the solution will be represented by an individual timetable which has no teacher clashes. The population consists of a set of different timetables, many of which would not be optimal. Such an individual may have been born to parents which each had teacher clashes, but the particular combination of genes from the parent allowed a new timetable to be constructed with no clashes.

In this paper we discuss the application of a GA to the school timetabling problem, and show how the execution time can be reduced by using a commercial shared memory multiprocessor. In the next section, we will discuss some of the aspects of a simplified timetabling problem, and in section 3 we show how to apply GAs. Section 4 shows a parallel solution and section 5 gives some experimental results.

2 TIMETABLING PROBLEM

The problem of creating a valid timetable involves scheduling classes, teachers and rooms into a fixed number of periods, in such a way that no t*eacher, class or room* is used more than once per period. For example, if a class must meet twice a week, then it must be placed in two different periods to avoid a clash. A class consists of a number of students. We assume that the allocation of students into classes is fixed, and that classes are disjoint, that is, they have *no students in common*. In another paper we discuss an extension which allows classes to overlap [6]. In this scheme, a correct timetable is one in which a class can be scheduled concurrently with any other class. In each period a class is taught a *subject*. It is possible for a subject to appear more than once in a period. A particular combination of a teacher, a subject, a room and a class is called an *tuple*. A tuple may be required more than once per week. Thus, the timetabling problem can be phrased as scheduling a number of tuples such that a teacher, class or room does not appear more than once per period.

The task of combining a class, teacher and room combination into a tuple is handled as a separate operation, and is not the subject of this paper. Tuples are formed with knowledge of the requirements of the various classes and teachers, as well as information on room availability. It is convenient to partition the problem in this way as it reduces the complexity of the scheduling task. In many cases it is possible to form the tuples without the need to use an optimisation scheme because the relationship between classes, teachers and rooms is often fixed.

It is possible to define an *objective* or *cost function* for evaluating a given timetable. This function is an arbitrary measure of the quality of the solution. A convenient cost function calculates the number of clashes in any given timetable. An acceptable timetable has a cost of 0. The optimisation problem becomes one of minimising the value of this cost function. The cost of any period can be expressed as the sum of three components corresponding to a class cost, a teacher cost and a room cost. It is not strictly necessary to sum the components, providing they can be combined to reflect the quality of the solution. However, by using a sum, it is easy to weight the various costs so that one may be made more important than the others. In this way the optimisation process can be guided towards a solution in which some types of clash are more important than others.

The class cost is the number of times each of the classes in the period appears in that period, less one if it is greater than zero. Thus, if a class appears no times or once in a period then the cost of that class is zero. If it appears many times the the class cost for that class is the number of times less one. The class cost for a period is the sum of all class costs. The same computation applies for teachers and rooms. The cost of the total timetable is the sum of the period costs. Therefore, any optimisation technique should aim to find a configuration with the lowest possible cost.

3 APPLYING GENETIC ALGORITHMS TO THE TIMETABLING PROBLEM

A timetable can be represented by a fixed set of tuples, each of which contains a class number, a teacher number and a room number. The scheduling algorithm must assign a period number to each of these tuples such that a given class, teacher or room does not appear more than once in a period.

In order to use a genetic algorithm to solve this problem, it is necessary to devise a representation for a timetable which maps onto chromosomes. Each chromosome is composed of genes, each of which represents some property of the individual timetable. «IF EA = "NO"»Figure 1a shows a sample timetable as a collection of tuples. «ENDIF»Each period contains a number of tuples, identified by their *label*. The values of the fields of the tuple would contain valid class, teacher and room numbers.

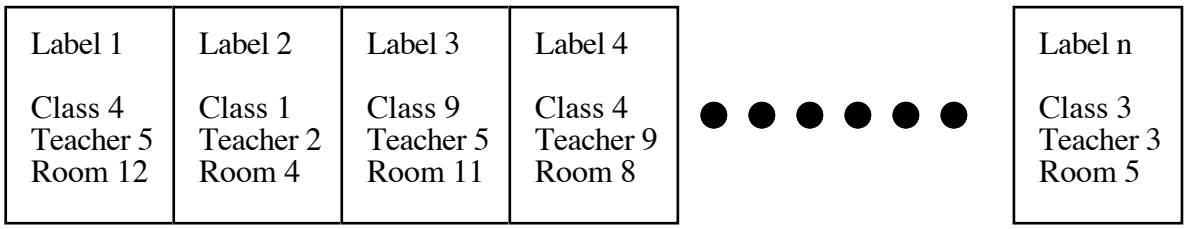
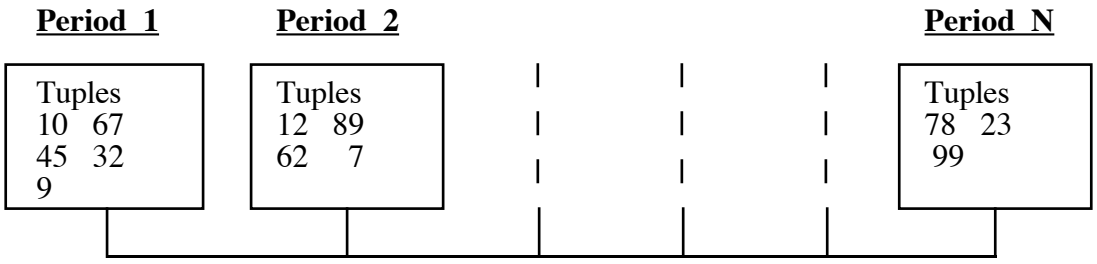


Figure 1a - Representing the timetable problem using tuples



N = Periods Per Week

Figure 1b - Mapping tuples onto periods

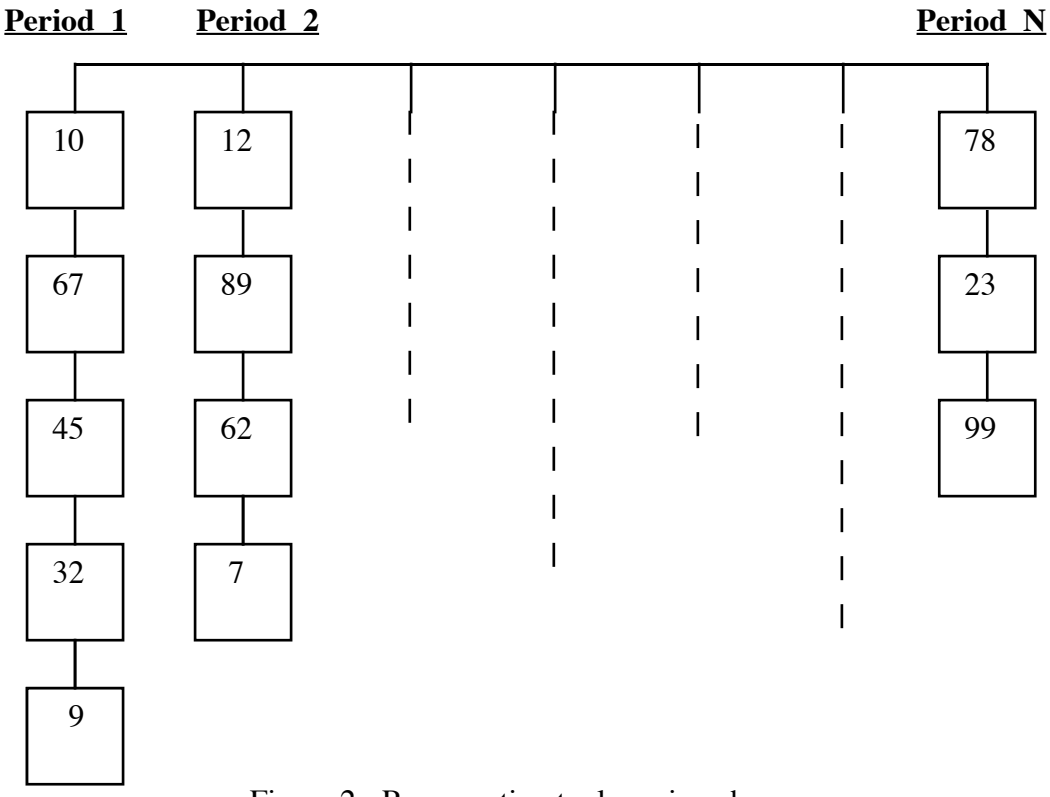


Figure 2 - Representing tuples using chromosomes

In Figure 1b the data has been mapped onto periods, and shows many tuples packed into each period. The cost of a timetable can then be computed by adding up the number of clashes in each period. The good and bad attributes of a timetable are related to the quality of the packing of tuples into periods, rather than the absolute period location of any one tuple. Thus, the genetic representation must preserve good packings, remove bad ones and allow new packings to form as the population evolves.

Figure 2 shows one possible mapping of tuples onto chromosomes. In this scheme each period *represents* a chromosome. Using this mapping allows good packings to be preserved between generations, because corresponding periods are mated, as described shortly. We discarded a number of representations because they did not allow packings to propagate between generations.

During mating a child is created which has some of the genes from one parent, and the remainder from the other. This process is called cross-over, and is implemented as a random selection of genes from each parent. During mating, the child may be mutated. It is mutation which introduces diversity into the population, and allows new class, teacher and room mappings to evolve. Mutation is performed on a randomly selected gene, which is then mutated in a random way. In the timetable problem, this corresponds to altering the period in which a tuple is located.

Figure 3 shows the process of mating two timetables to produce a new timetable.The diagram only shows the cross over of one period. Each period is crossed over independently. In this example, a random crossover site is chosen, and the new period in the child is constructed with the first 3 genes of parent 1, and the last 2 of parent 2. Each period of the child is created by the same process. If the cross-over site is at either end of the chromosome, then all of that period is taken from only one parent. The fitness of the new child can be computed incrementally as the change in cost of the period.

Figure 4 shows how the child can receive further mutation of its mapping. In this example, a randomly chosen tuple from a randomly chosen period is moved to another randomly chosen period.

Genetic algorithms require a measure of the fitness of an individual in order to determine which individuals in the population survive. The higher the fitness, the more likely the individual will survive. The cost measure developed for the timetabling problem represents a high quality solution with a minimal cost value (a value of 0 is optimal). Thus, a fitness value is computed from the cost of a timetable by subtracting the cost from a fitness ceiling. In this way, an increased cost generates a lower fitness value. Fitness values are further scaled so that unfit individuals have a negative fitness value, and are then removed from the population.

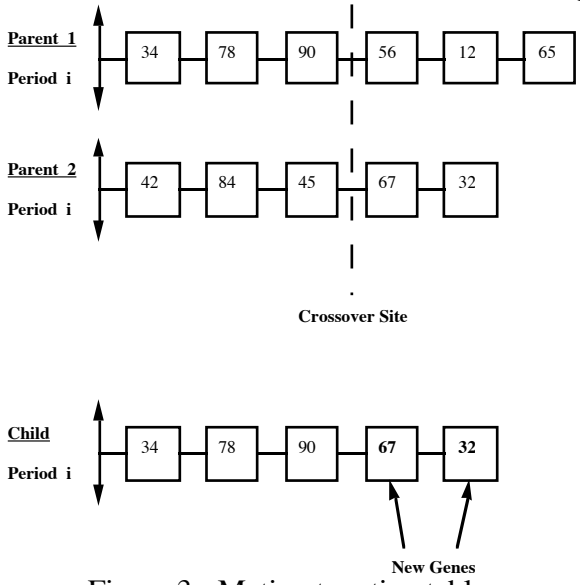


Figure 3 - Mating two timetables

The representation scheme was chosen because it allows the good properties of a timetable to propagate from one generation to the next. However, a problem with the reproduction scheme described above is that after a mating, the contents of the timetable have been altered, and no longer reflect the original data set. This problem is known as the label replacement problem [5], and is caused because the cross over takes tuples period by period, rather than tuple by tuple. Thus, the disadvantage is that the child no longer contains the correct set of tuples. This problem is illustrated in Figure 5, where label 26 has been duplicated in the child.

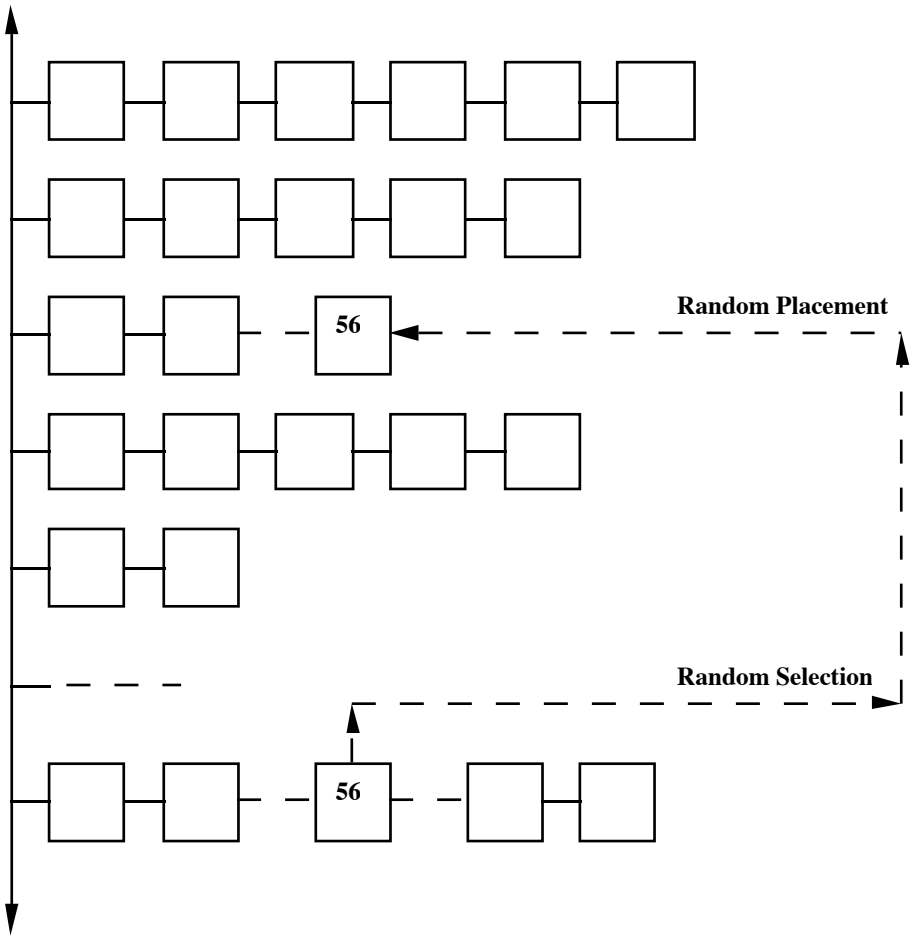


Figure 4 - Mutating a child

The solution to the problem of loss of tuples is to use a label replacement algorithm. After a child has been created, it is modified to restore the original tuple set. Thus, a tuple which has been lost through mating, is restored in a random location, effectively mutating the child chromosome. Similarly, copies of duplicated tuples are removed. Label replacement can be seen as a form of mutation because it introduces extra diversity into the population.

It is worth considering why label replacement does not occur in biological mating. The reason is that when cross-over occurs, the child inherits one set of genes from one parent, and the exact complement of genes from the other parent. The scheme works because a gene represents some quality of the individual, such as blue eyes. However, the representation we have chosen for the timetables means that the genes represent some packing of tuples in the individual, but not the exact identity of the tuple. Thus, after cross-over, the child has some genes duplicated, whilst losing others.

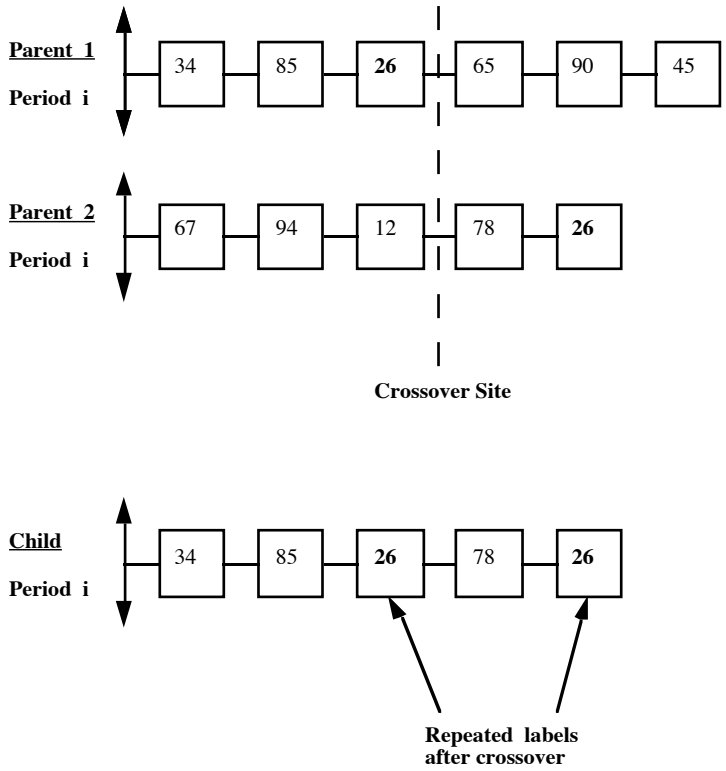


Figure 5 - Problem caused by loss of labels

4 A PARALLEL ALGORITHM

Biological mating is highly concurrent! In practice, individuals mate with little regard to the rest of the population. However, many computer simulations of evolution are sequential: the simulation proceeds by sequentially creating a new population as a sequence of matings. The algorithm discussed in this section has been designed for a shared memory multiprocessor.The most obvious target for concurrency is in the creation of the new population, because each mating operation is largely independent of any other. Another source of concurrency is the actual mating operation, which requires independent manipulation on an individual. However, the population size is typically much larger than the number of processors available in shared memory machines, and thus we need only consider the first source of concurrency.

The genetic algorithm can be summarised as follows:

```
while number of generations < limit & no perfect individual do
for each child in the new population do
        choose two living parents at random from old population
        create an empty child
        for each period of the parents do
                mate corresponding periods
                copy new child period to corresponding position in child od
        repair lost & duplicated labels
        apply mutation to randomly selected period & tuple
        measure fitness of individual.
        if fitness < minimum allowed fitness (based on fitness scaling) then
                set child status to born dead
          else
                set child status to living fi
od
old population = new population
od
```

The algorithm is viewed from the perspective of the child because this allows random selection of parents. The mating is performed with a randomly chosen cross-over site within the period, and is done for each period of the individual. Mutation is performed on randomly selected periods and tuples, and occurs with some specified probability.

Figure 6 shows a template for a parallel mating process. Creation of each child is by random selection of parents, and then random mutation. Instead of one sequential piece of code creating all the children in the new population, a number of workers are spawned and each is then responsible for a fixed number of the children. It is important to minimise interprocess synchronisation to maximise the speedup. Since the parents are used in a read-only mode no synchrnonisation is required when the parents are accessed. Further, no synchronisation is required on creation of children because each child is created by only one worker, and the new population is only written once within a generation. In a shared memory machine it is possible for the workers to independently write into pre-allocated slots of the new population. Barrier level synchronisation is required at the end of each generation.

In order to provide a deterministic execution history, each worker uses a separately seeded pseudo-random number generator. Thus, two executions of the program with the same numbers of workers and the same initial seeds, generate the same result.

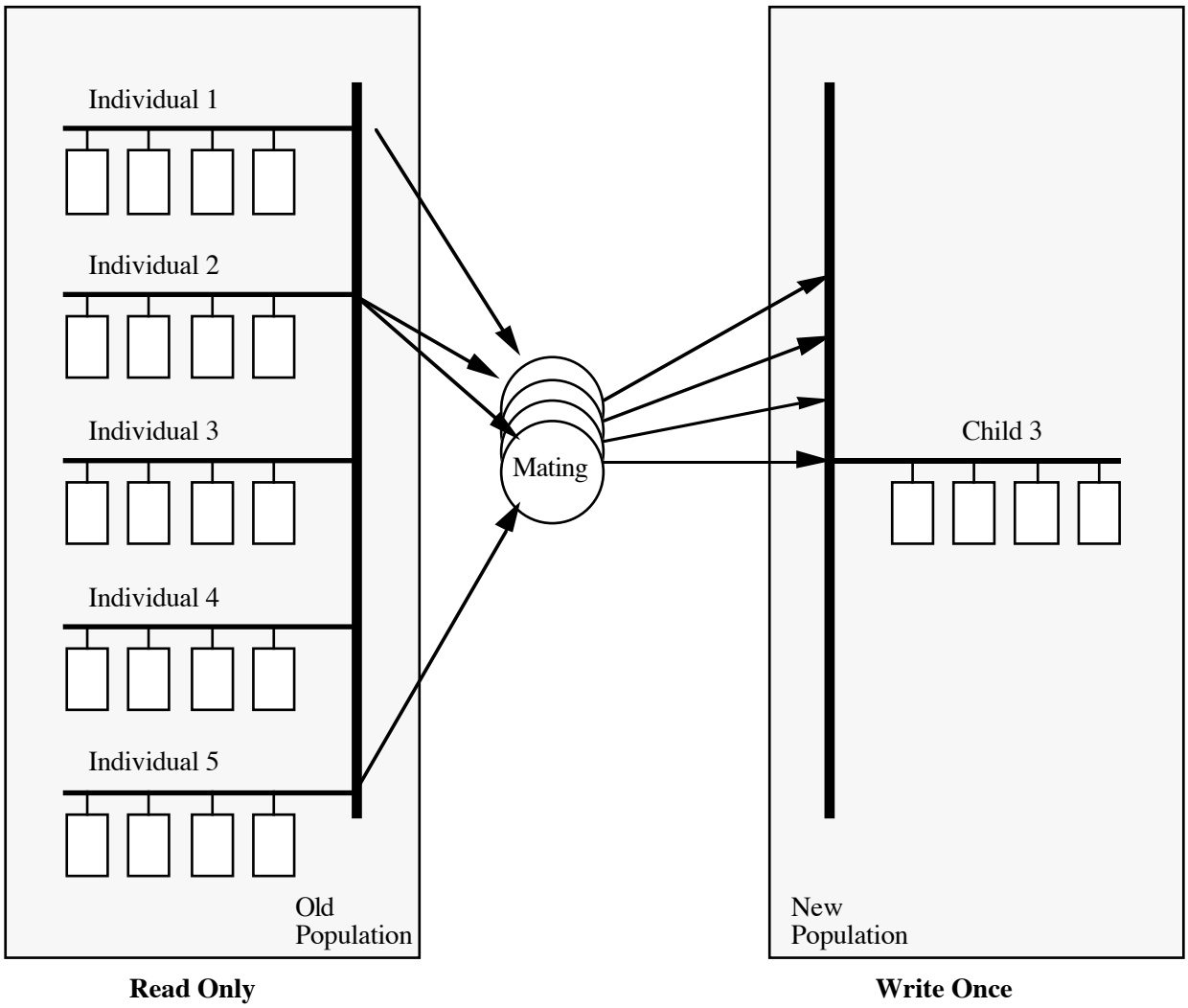


Figure 6 - a parallel mating scheme

5 SOME RESULTS

In this section we present some results of 9 data sets. The table shows the number of combinations of classes, teachers and rooms that were scheduled, as well as the number of classes, teachers and rooms in the data set. In all cases, the week is composed of 30 periods. The final cost is the number of clashes that remained in the best individual of the final population. The number of generations required to solve the problems is shown.

The results in Table 1 show that the genetic algorithm is capable of finding the global minima in all cases. Data sets 6 through 9, whilst small, are highly constrained, and thus are quite difficult to solve. The population size was 100.

In Table 2 we present the speedup attained from a parallel implementation of the program. The code was written in Pascal and run on an Encore Multimax shared memory multiprocessor. The times were taken for a fixed number of generations (100) so that the effects of relative solution quality could be ignored. The table shows the execution time in seconds when 1, 2, 5, 10 and 15 processors were assigned to the mating algorithm. The speedup is defined as the sequential time divided by the best parallel time, thus ideal speedup should be equal to the number of processors used to achieve the best parallel time. Two different population sizes were chosen. The results show that the speedup is less than optimal. This is because there are a few sections of code in the critical path of the program which have not yet been parallelised.

Data	#Tuples	#Classes	#Teachers	#Rooms	Final Cost	# Gen
1	100	15	15	15	0	7
2	150	15	15	15	0	31
3	200	15	15	15	0	102
4	250	15	15	15	0	198
5	300	15	15	15	0	445
6	90	3	3	3	0	307
7	120	4	4	4	0	4098
8	150	5	5	5	0	3723
9	180	6	6	6	0	42973

Table 1 - Performance of sequential genetic algorithm

Data	#Tuples	Population	Time (secs) / Processors					Peak Speedup
			1	2	5	10	15	
7	120	120	752	376	197	108	81	9.3 @15 procs
7	120	200	1886	921	431	205	-	9.2 @10 procs

Table 2 - Speedup of tests on data set 7

6 CONCLUSIONS

We have shown that genetic algorithms can be applied to this complex scheduling problem. The results shown in the previous section indicate that the technique is quite powerful in finding the global minimum from an enormous search space. In a real school, there are many more constraints than have been addressed in this paper. These are discussed further in [6].

Because the genetic algorithms are quite slow to execute, a solution was developed for execution on a parallel processor. Because the process of breeding is inherently parallel, we observed quite good speedups over sequential execution of the algorithm.

Acknowledgements

The High Performance Computation Project is a joint project between the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) Division of Information Technology and the Royal Melbourne Institute of Technology (RMIT). Thanks go to Rhys Francis for editing a draft of this paper.

REFERENCES

[1] Goldberg, D. E. "Genetic Algorithms: In Search, Optimization and Machine Learning". 1989, Addison-Wesly Publishing Co.

[2] Wilson, S. W. "Hierarchical Credit Allocation in a Classifier System", Research Notes on Artificial Intelligence, Genetic Algorithms and Simulated Annealing, Pittman, 1987, London.

[3] Holland, J. H, Holyonk, K. J., Nisbett, R. E., Thayard, P. R. "Classifier Systems, Q-Morphisms and Induction".Research Notes on Artificial Intelligence, Genetic Algorithms and Simulated Annealing, Pittman, 1987, London.

[4] Robertson, G. "Parallel Implementation of Genetic Algorithms in a Classifier Design".Research Notes on Artificial Intelligence, Genetic Algorithms and Simulated Annealing, Pittman, 1987, London.

[5] Glover, D. E. "Solving a Complex Keyboard Configuration Problem Through Generalized Adaptive Search".Research Notes on Artificial Intelligence, Genetic Algorithms and Simulated Annealing, Pittman, 1987, London.

[6] Abramson, D. A. "Constructing School Timetables Using Simulated Annealing: Parallel and Sequential Solutions", Management Science, pp 98-113. Jan 1991.

[7] Abramson, D. A. "A Language and Compiler System for Specifying Timetable Requirements", Division of Information Technology. Commonwealth Scientific and Industrial Research Organisation. C/- Dept. of Communications and Electrical Engineering, RMIT.