Parallelisation of a Genetic Algorithm for the Computation of Efficient Train Schedules

David Abramson
School of Computing and Information Technology
Griffith University
Kessels Rd, Nathan, Qld 4111
davida@cit.gu.edu.au

Graham Mills
Division of Mathematics and Statistics
C.S.I.R.O.
Private Bag #2
Glen Osmond, SA 5064
grahamm@adl.dms.csiro.au

Sonya Perkins
Scheduling and Control Group
University of South Australia
The Levels South Australia 5095
sep@scg.levels.unisa.edu.au

Abstract

This paper discusses the parallelisation of a genetic algorithm which computes train schedules. The program requires departure times for trains operating in two directions on a single-line track, and attempts to compute an optimal crossing plan which minimises lateness at the destination stations. Single-line track complicates the problem as trains are restricted to cross and overtake at fixed crossing locations.

Genetic algorithms require a problem to be coded as a string. The complexity of this train scheduling problem has resulted in a complex routine for decoding. This has resulted in longer calculation times which prompted the investigation of parallelisation techniques. The conversion of the code from single processor to multi processor calculations takes into account a number of different parallel computer architectures, including conventional sequential machines, shared memory machines, distributed memory multiprocessors and a network of workstations. The code was tested on a 16 processor Encore Multimax, a 128 processor Fujitsu AP1000 and a network of DEC Work Stations. Results are presented for the speedup attained. The best time was on the AP1000, which achieved a 50 times speedup over its own uni-processor time.

1. Introduction

This paper concerns the use of high performance parallel and distributed computers for accelerating the execution of a Genetic Algorithm (GA). GA's are heuristic search algorithms and they have been shown to be effective at providing high quality solutions, however, the process can be time consuming. Parallel computers provide an excellent mechanism for delivering increased performance because GA's are inherently parallel.

This document reports on the parallelisation of an existing program, originally written only for sequential execution on a conventional workstation. It begins by giving the basic flow and operation of GA's, particularly as applied to train schedules. It then gives a general overview of parallel processing architectures and the implications for algorithms such as GA's. The techniques used for parallelisation are discussed followed by some results and issues for further work.

2. The Train Scheduling Problem

The basic train scheduling problem involves the determination of appropriate train schedules for a fleet of trains operating on a single-line section of track with crossing loops. Crossing loops are short sections of double-line (sometimes multi-line) track at which trains can cross and overtake. Only one train can be on a single-line section of a track at a time.

A train schedule is a set of arrival and departure times at key stations for a fleet of trains. A key station is a station at which a train is required to be loaded or unloaded, or where a train is to cross or overtake another train. The aim is to determine a train schedule that minimises an objective function. For further information on the train scheduling problem see [Mills et al] and [Kraay et al].

Mathematically the problem can be formulated using the following parameters, objective function and constraints. The aim is to minimise equation (1) subject to equation (2) through (5).

Track Parameters:

$S = \{i \mid i = (1, 2, \ldots, n)\}$	set of stations
S_i	station i
$S_{i,j}$	track section between the consecutive stations i and j, i is at the start of the section and j is at the end
C_i	capacity of station i
N_i	number of trains currently at each station
$N_{i,j}$	number of trains on the track section $S_{i,j}$

Train Parameters:

$T = \{t \mid t = (1, 2, \ldots, m)\}$	set of trains
T_t	train t
$\tau_t^{i,j}$	minimum time for train t to traverse track section $S_{i,j}$ from station i to j. It is not necessary that $T_t^{i,j} = T_t^{j,i}$
O_t	origin station of train t
D_t	destination station of train t
a_t^*	earliest possible arrival time of train t at its destination station D_t
d_t^*	earliest possible departure time of train t from its origin station O_t
W_t^i	dwell time of train t at station i
a_t^i	arrival time of train t at station i
d_t^i	departure time of train t from station i

Objective Function

$$L = \sum_t L_t = \sum_t max(0, a_t^{D_j} - a_t^*) \quad (1)$$

Constraints

$(N_{i,j} = N_{j,i}) \in \{0,1\}$ maximum of one train on a simgle-line track at a time (2)

$N_i \leq C_i$ number of trains at a station cannot exceed the station capacity (3)

$a_t^j = d_t^i + \tau_t^{i,j}$ where i and j are consecutive stations (4)

$d_t^j \geq a_t^j + W_t^j$ (5)

3. Genetic Algorithms

Genetic Algorithms (GA's) are heuristics which can be used to optimise complex cost functions subject to various constraints. They are based on the theory behind evolution, a process which causes the *average fitness* of a population to increase from one generation to another. The *fitness* increases because good individuals are selected for *mating*, and thus good properties are transmitted from one generation to another, whilst less desirable properties are de selected from future populations.

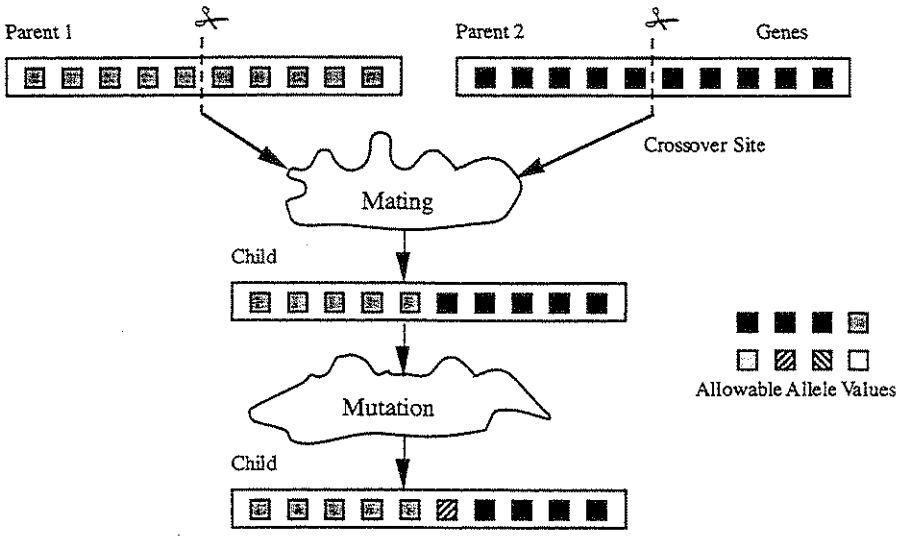


Figure 1 - Mating and Mutation process

To employ a GA, to solve an optimisation problem, it is necessary to choose a string representation for a solution to the problem. A fitness function is also required to determine the quality of the solution. An initial population, in which there are many solutions with varying quality, is generated randomly. In order to produce subsequent populations individuals from the previous population are selected, according to their fitness, and mated. This produces new individuals which possess properties from their

parents. Random mutation is also applied to the newly produced individuals to ensure that diversity in the population is maintained. The average fitness of the population increases during this process as parent strings are selected according to their fitness. Less fit individuals are less likely to have their attributes transmitted from one generation to the next. The algorithm terminates after a set number of generations have been produced. Other termination criteria, such as when the variance in the population fitness falls below a threshold value, or when a perfectly fit individual is created, can be used.

In order to apply genetic mating, it is necessary to represent the problem as a collection of *genes*, each of which encodes some attribute or feature of the problem. Genes are usually encoded as some numeric value, called an *allele*. Thus, an individual can be represented as a string of genes. The mating operation usually consists of computing a random crossover site within the string, and then combining one section of one parent and one section of the other into a new child. Mutation chooses a gene at random and alters its value. This process is summarised in Figure 1. In practical GA's the mating and mutation process is often more complex, involving issues such as multi-site cross-over and clever mutation driven by heuristics. The string representation of the problem also impacts upon the types of crossover and mutation techniques that can be applied.

3.1 Program Structure

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
        choose a cross over site within the genes, creating two regions per parent
        copy region 1 from parent 1 and region 2 from parent 2 to new child
        apply mutation to child
        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
```

3.2 Train Scheduling representation

This section briefly describes the encoding used in the solution of train timetables, and does so only to illustrate the parallelisation strategy. The exact encoding is not discussed, and interested readers are referred to detailed program documentation. The problem consists of computing arrival and departure times for a number of trains at a number of distinct stations along a single-line track, as shown in Figure 2. The scheduling problem arises because trains travel in both directions, and can only over-take or cross each other by using special crossing loops which are placed strategically along the track. Thus, a schedule must guarantee that only one train is travelling on any one section of track at a time in order to be feasible.

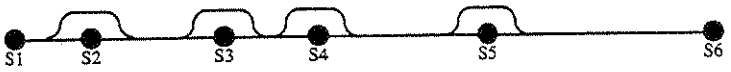


Figure 2 - Example Single-line Track, Sidings and Stations

The output of the scheduling operation can be displayed as a train graph which shows when various trains arrive and depart from stations in relation to each other. A sample is shown in Figure 3. A good schedule is one which minimises the lateness at the end of the journey for all trains. This is visible by inspecting the time of arrival for the last train and comparing it to the earliest time the train could have arrived.

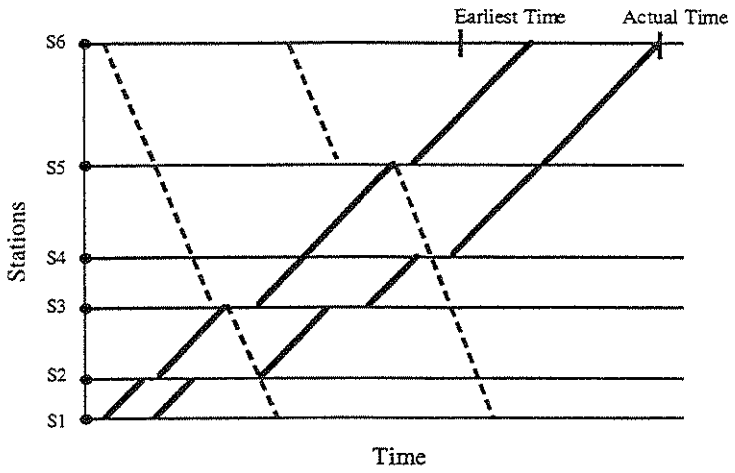


Figure 3 - Train Graph

The schedule is represented by a chromosome composed of individual genes. The number of genes in a chromosome is equal to number-of-trains*(number-of-stations -1), and these are combined into one string. Each gene is used to denote the next train to be moved from its current station to the next station on that trains journey. The string does not contain arrival and departure times, but these must be *decoded* from the inherent ordering of the trains and the train's initial departure times from the origin stations. This decoding operation is sequential. It starts with the first train and requires the times for one train to be computed before it can move to the next. It is therefore not possible to parallelise the decoding operation. From the algorithm described in the previous section, it can be seen that the decode operation must be performed for every individual in every generation, and thus most of the time is spent decoding the chromosomes.

4. Parallel Processing

There are a number of parallel architectures currently available. This paper is concerned with Multiple Instruction Multiple Data machine (MIMD) in which a number of separate Von-Neumann machines are able to work on sections of one larger problem and communicate only when necessary. The two main classes of MIMD system are shared memory and distributed memory. We were able to map the genetic algorithm to both classes, and will briefly describe each.

4.1 Shared Memory Architectures

A generic shared memory machine is shown below in Figure 4. It consists of a number of distinct processing elements, each of which communicates with a shared memory via some Interconnection Network (IN). In many cases the IN is a shared bus, but some machines use IN's made of cross bar switches, hyper-cube interconnects and various multi-stage networks. For the purposes of this document, the exact structure of the IN is irrelevant. Each processor module is composed of a conventional processor, a cache memory and a network interface. Almost all shared memory machines have cache

control logic which allows them to maintain a coherent view of shared memory regardless of particular sharing patterns.

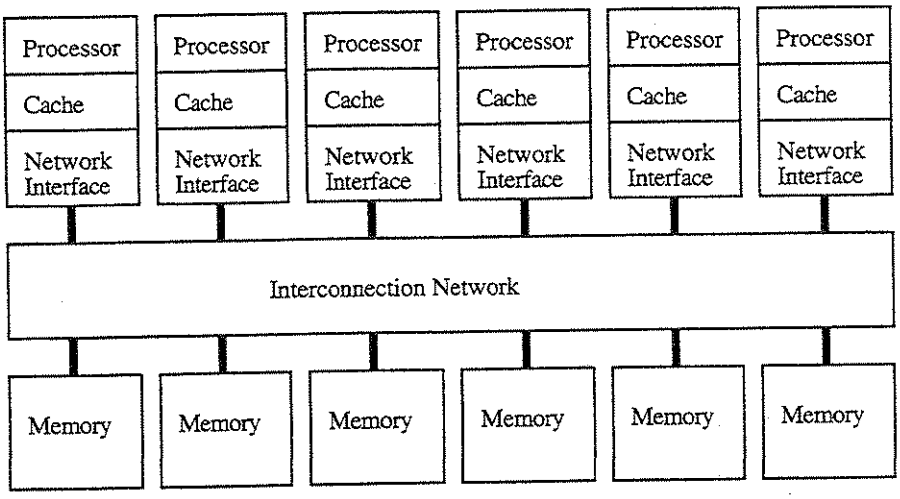


Figure 4 - A Generic Shared Memory Machine

Because the architecture of the shared memory machine is not substantially different from a conventional uni-processor, it is often fairly easy to parallelise existing sequential programs. The program must be decomposed into a number of co-operating processes which can communicate information via blocks of shared memory. In a UNIX™ environment, this usually means *forking* a number of processes, and *creating* shared memory segments. Program synchronise using shared semaphores and barriers. Shared memory machines usually provide reasonable speedup because the IN provides a very rapid path to memory regardless of whether the variable is shared or not. Thus, speedup is usually limited by algorithmic properties and excessive synchronisation rather than architectural limitations.

4.2 Distributed Memory Architectures

Distributed memory machines are similar to shared memory ones in that they utilise conventional Von Neumann processors. However, rather than providing a transparent path to shared variables, they only allow for explicit communication through a network. This has a fairly dramatic effect on the program structure because shared information must be explicitly transmitted and received. Figure 5 shows a generic distributed memory machine.

There are many examples of interconnection networks which have been used in distributed memory multiprocessors. In our experiments we used a point-to-point network (in the case of the transputers), a mesh (in the case of the AP1000) and an ethernet (in the case of the DEC Stations). Each of these has quite different characteristics in terms of latency and throughput, and this affected the results.

Distributed memory machines require a different programming paradigm from sequential machines. Programs must be decomposed into co-operating sequential processes which communicate via message passing. Because of this difference, it is usually necessary to substantially modify sequential programs for execution on distributed memory machines. The code presented in this study was converted for execution on both shared memory and distributed memory machines, however,

conditional code compilation makes it possible to maintain one source program. This is discussed later in the paper.

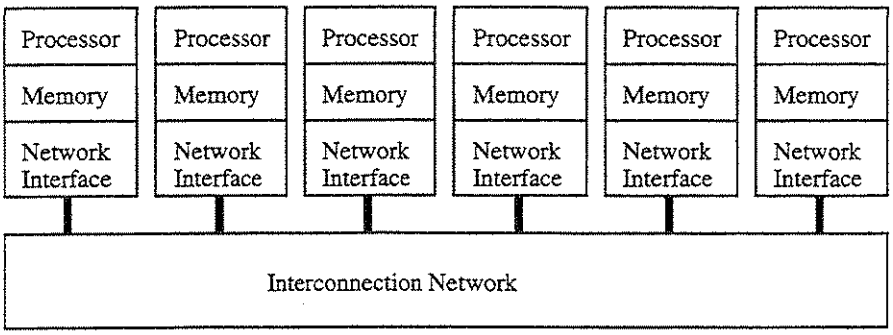


Figure 5 - A Generic Distributed Memory Machine

4.3 Parallel Processing of Genetic Algorithms

Genetic algorithms are inherently parallel. The new population is built by a number of independent mating operations. The mating operation selects individuals from an old population and builds a new individual without regard to other members of the new population. The schematic in Figure 6 shows a number of *agents* responsible for mating. Each agent selects two parents from the old population, which can be considered as read-only, and produces a number of new individuals in the new population. The new population is write-once, that is, each individual is only written once, and only by one agent. This means that there is very little synchronisation required between agents. One synchronisation barrier is required to determine when the new population has been built completely, and when it can become the old population for the next generation.

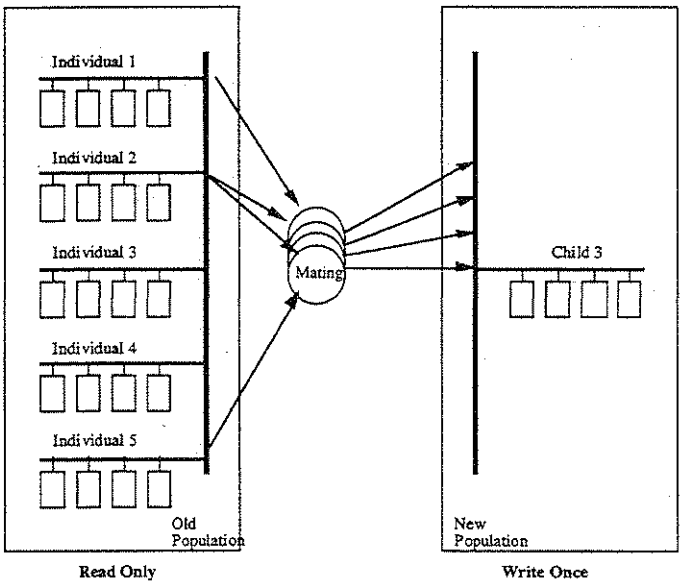


Figure 6 - A schema for a parallel GA

When the parallel algorithm is implemented on a shared memory machine, the old and new populations are placed into shared memory. The population is divided up across the available worker processes. A barrier semaphore is placed in shared memory and marks when the new generation has been built.

When implemented on a distributed memory machine, the new population must be transmitted to all processors. Each process builds its own section and then broadcasts it to the others. Each process then copies the sections it receives from others into a local version of the new population. No further synchronisation is required as a process does not proceed to the next generation until it receives all of the sections for the new population.

5. Parallelisation of the Train Scheduling Problem

5.1 Pascal -> C

The original code supplied was written in Pascal for a HP workstation. Unfortunately many of the tools required for parallelisation are geared for C programs. The Pascal was converted to C using a freeware program P2C. The conversion tools produces fairy readable code. The following table shows the properties of each of the codes.

Program	#Lines	#Bytes
Pascal	1229	4579
C	1551	5377
Parallel C	1842	5895

The program was converted automatically initially and then modified for parallel execution manually. The parallel version uses conditional compilation to incorporate code which is specific for the shared memory and distributed memory machines.

5.2 Repeatability

In the sequential version of the code, one central random number generator is used by all sections of the algorithm. When the code is parallelised, each process requires its own stream of random numbers. It is possible to implement a centrally shared number generator, however, this causes the code to behave non-deterministically. Consequently, the results are not repeatable. It also has the disadvantage that the random number generator can become a bottleneck. Another alternative is to provide a separate random number generator for each process. This has the advantage that there is no bottleneck, and the code is deterministic. However, when the program is executed with a different number of processes the random number streams vary, thus it is difficult to validate that the program is working correctly. For example, a single processor run generates a different solution to a 4 processor run.

The solution which was adopted is to maintain a different random number stream for each member of the population. In this way, the same stream is generated regardless of the number of processors used. Because a number of generators are used, there is no bottleneck.

5.3 Amdahl's Law

Amdahl's law governs the maximum speedup possible in a parallel program regardless of the number of processors that are available. The speedup can be expressed as:

$$S = 1 + \frac{Ts}{Tp}$$

where *Ts* is time spent in the sequential portion of the code and *Tp* is the time spent in the parallelisable section of the code. The equation clearly shows that the speedup is limited by the fraction of code which cannot be parallelised, or which is consumed by communication. In the code which was supplied, this effect was evident in two areas. First, the initialisation code which remained un-parallelised. Second, there were special cases which were applied to individuals in the population. These two areas had a substantial effect on the speedup, and required special attention.

5.3.1 Initialisation

The initialisation code reads in the station and track information, and builds the initial population. Whilst it is possible to parallelise sections of this code, given a sufficiently long run the initialisation code can be ignored.

5.3.2 Special code per generation

Far more serious was special code inserted to perform local optimisation on individuals in the new population. Because the parallelisation is performed across the population, an individual which receives special attention is processed sequentially. In the original code, the best members of the old population were copied into the new population, and then operated upon by a greedy local optimisation procedure. This procedure was very expensive and affected the speedup. It transpired that the optimisation had little overall effect on the quality of the solution, and thus could be removed.

All other sections of the code were parallelised.

5.4 Software Engineering Issues

As mentioned earlier in the paper, the one piece of source code is maintained across many different platforms. This was performed using conditional compilation in the C code, together with some routines for mapping PVM calls to other machine libraries. Where ever possible, code is shared amongst the three versions.

5.4.1 Shared Memory Code

The shared memory code is written using UNIX standard procedures for process invocation and control. The shared memory data structures are declared using a special statement in Encore C. This option is enabled by using -DSHARED when the program is compiled.

5.4.2 PVM

The distributed memory version is written using the PVM standard interface. Process invocation and control is performed through PVM (which is mapped onto UNIX in most implementations) and message passing is also through the PVM send and receive primitives. This option is enabled through the -DPVM flag when the program is compiled. Since PVM is available on many parallel platforms, this option provides a great deal of portability.

5.4.3 AP1000 Cell Library

The AP1000 provides its own message library. To avoid excessive duplication in the code, the PVM calls are mapped onto the AP1000 library using a small set of routines. In this way, the bulk of the code is the same for both the PVM and AP1000 message passing versions. This version is produced when the -DAP1000 flag is used during compilation.

6. Results

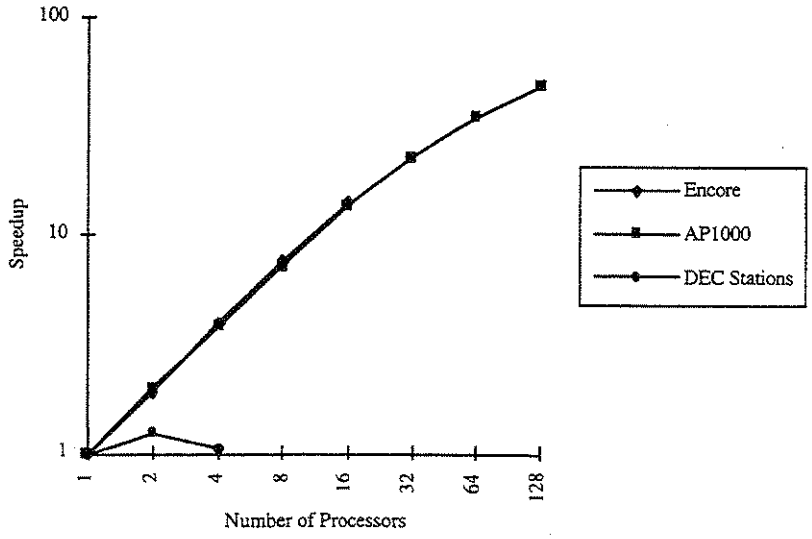
In this section we present the results of the execution of the program on an Encore Multimax shared memory multiprocessor, a Fujitsu AP1000 distributed memory

multiprocessor and a small network of DEC workstations, connected by ethernet. Table 1 shows the execution times for varying numbers of processors, together with the time computed assuming ideal speedup, and the actual speedup attained. In summary, the shared memory version, and the distributed memory version on the AP1000 performed very well up to 16 processors. The AP1000 version continued to show speedup up to 128 processors, but was well below ideal at 128 processors. All runs were performed with a population size of 256 individuals, which meant that the 128 processor run only allocated 2 individuals for every CPU.

The DEC workstation speedups were very low, showing almost no speedup at 2 and 4 processors. This is because of the enormous overheads incurred when ethernet and UNIX TCP/IP are used as the communication protocol. Further, since ethernet is a contention based bus protocol, the large number of broadcasts causes excessive network traffic. Since each processor attempts to broadcast their sub-populations at the same time, there are a large number of collisions on the network. The performance was also poor because the PVM software which was used as the message library, implements broadcast as a number of multiple transmissions, rather than using any broadcast capabilities of the ethernet. Graph 1 summarises the speedups from the three machines.

	Encore Multimax NS32032			Fujitsu AP1000			DEC Stations		
#CPUs	Time	Ideal	Speedup	Time	Ideal	Speedup	Time	Ideal	Speedup
1	3725	3725	1.0	769	769	1.0	207	207	1.0
2	1929	1863	1.9	390	385	2.0	166	104	1.2
4	945	931	3.9	200	192	3.8	196	52	1.1
8	492	466	7.6	106	96	7.3			
16	266	233	14.0	57	48	13.5			
32				34	24	22.6			
64				22	12	35.0			
128				16	6	48.1			

Table 1 - Execution times and speedups



Graph 1 - Speedup for different processors

7. Further Work and Conclusions

In this paper we described the results of parallelisation of an existing sequential genetic algorithm. Because of the nature of genetic algorithms, the procedure was relatively simple. The code runs on a number of platforms ranging from shared memory to distributed memory machines and workstation farms. The AP1000 provided good speedup for a large number of processors. The work station farms provided very poor performance due to high communication costs.

One interesting avenue which would lower the communication requirements of the code is to not broadcast the sub populations on every time step, but perform a number of steps using only local members of the population. This would have the effect of producing a number of co-operating populations, and would allow exchanges between them. Similar systems are described in [Goldberg].

Acknowledgements

This project was carried out as part of a GIRD project, which is sited at the University of South Australia. Thanks go to the CSIRO Divisions of Mathematics and Statistics and Information Technology, to ANU Computer Science Department for access to their AP1000 and to the RMIT Department of Computer Systems Engineering for access to their Encore Multimax.

The train scheduling problem outlined in this paper is an actual problem facing railways with single-line track. The Scheduling and Control Group at the University of South Australia has interacted with Australian National throughout its research in order to obtain a clear perspective of the problem being solved. We wish to thank AN for their involvement.

References

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

Mills R. G., Perkins S. E., Pudney P. J., "Dynamic Rescheduling of Long-haul trains for improved timekeeping and energy conservation, APJORS, vol 8, 1991, pp 149-165

Kraay D., Harker P.T. and Chen B., "Optimal pacing of trains in freight railroads: Model Formulation and Solution", Decision Science Working Paper, The Wharton School, University of Pennsylvannia, 1985