A GENETIC APPROACH USING DIRECT REPRESENTATION OF SOLUTIONS FOR THE PARALLEL TASK SCHEDULING PROBLEM

ESQUIVEL S.C., GATICA C. R., GALLARD R.H. [1]

Abstract

In scheduling, a set of machines in parallel is a setting that is important, from both the theoretical and practical points of view. From the theoretical viewpoint, it is a generalisation of the single machine scheduling problem. From the practical point of view the occurrence of resources in parallel is common in real-world.
When machines are computers, a parallel program can be conceived as a set of parallel components (tasks) which can be executed according to some precedence relationship.
In this case efficient scheduling of tasks permits to take full advantage of the computational power provided by a multiprocessor or a multicomputer system. This kind of planning involves the assignment of partially ordered tasks onto the system architecture processing components.
This paper shows the problem of allocating a number of non-identical tasks in a multiprocessor or multicomputer system. The model assumes that the system consists of a number of identical processors and only one task may execute on a processor at a time. All schedules and tasks are non-preemptive. The well-known Graham's list scheduling algorithm (LSA) is contrasted with an evolutionary approach using a direct representation of solutions.

2. Key Words

Parallel task allocation, Genetic Algorithm, List Scheduling Algorithm, Schemes of representation, Indirect and Direct representation, Optimisation.

1. INTRODUCTION

A parallel program is a collection of tasks, some of which must be completed before than others begin. The precedence relationships between tasks are commonly outlined in a directed acyclic graph known as the *task graph*. Nodes in the graph identify tasks and their duration while arcs represent the precedence relationship. Factors, such as number of processors, number of tasks and task precedence constraints contribute to make difficult a good assignment.
The problem to find an schedule on $m > 2$ processors of equal capacity, that minimizes the whole processing time of independent tasks has been shown as belonging to the NP-complete class (Horowitz and Sanhi 1976 [9])
Task scheduling can be classified as static and dynamic. In the case of static scheduling some strong reasons make it applicable. First, static scheduling sometimes results in lower execution times than dynamic

[1]The Research Group is supported by the Universidad Nacional de San Luis and the ANPCYT (National Agency to Promote Science and Technology). Proyecto UNSL-338403. Departamento de Informática. Universidad Nacional de San Luis. Ejército de los Andes 950 – Local 106. 5.700 – San Luis, Argentina. E-mail {esquivel, crgatica, rgallard }@unsl.edu.ar

scheduling. Second static scheduling allows only one process per processor, reducing process creation, synchronisation and termination overhead. Third, static scheduling can be used to predict speedup that can be achieved by a particular parallel algorithm on a target machine, assuming that no preemptions of processes occur.

3. A DETERMINISTIC MODEL

In a deterministic model, the execution time for each task and the precedence relations between them are known in advance. This information is illustrated in a directed graph, usually known as the task graph. In Fig. 1 we have eight tasks with the corresponding duration and their precedence relations

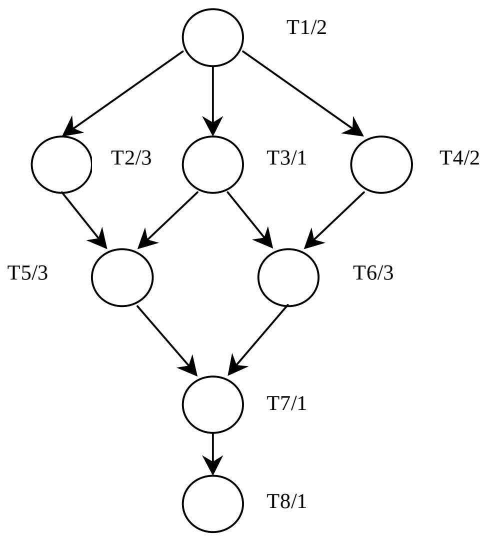

Fig. 1 . The model task graph

A task graph is a simplified representation of a parallel program execution, ignoring overheads due to interrupts for accessing resources etc. Nevertheless, it provides a basis for static allocation of processors.
A schedule is an allocation of tasks to processors which can be depicted by a Gantt chart.
In a Gantt chart, the initiation and ending times for each task in the available processors is indicated and the makespan (total execution time of the parallel program) of the schedule can be easily derived.

P_2	▨	▨	T_3	T_4		T_6			▨	▨
P_1	T_1		T_2			T_5			T_7	T_8
Time slot	1	2	3	4	5	6	7	8	9	10

Time 0 1 2 3 4 5 6 7 8 9 10

Fig. 2. Scheduling 8 tasks onto 2 processors by LSA

Figure 2, shows a Gantt chart corresponding to one possible schedule of the parallel tasks of the task graph of figure 1 onto two processors. By simple observation we notice a makespan[2] of 10 and an utilisation of a 100% for processor P_1 and an utilisation of 60% for processor P_2. Also an speed-up of 1.6 can be easily derived.

Connected with the makespan, an optimal schedule is such that the total execution time is minimized. Other performance variables, such as individual processor utilization or evenness of load distribution can be considered. As we can see some simple scheduling problems can be solved to optimality in polynomial time while others can be computationally intractable. As we are interested in scheduling of arbitrary tasks graphs onto a reasonable number of processors we would be content with polynomial time scheduling algorithms that provide good but no optimal solutions.

[2] For any arbitrary environment it is defined as the completion time of the last task leaving the system.

3. THE LIST SCHEDULING ALGORITHM (LSA)

For a given list of tasks ordered by priority, it is possible to assign tasks to processors by always assigning each available processor to the first unassigned task on the list whose predecessor tasks have already finished execution. Let be:

T={T_1,....,Tn} a set of tasks,

μ: T→ (0, ∞) a function which associates an execution time to each task,

≤ a partial order in T and

L a priority list of tasks in T.

Each time a processors is idle, it immediately removes from L the first ready task; that is, an unscheduled task whose ancestors under ≤ have all completed execution. In the case that two or more processors attempt to execute the same task, the one with lowest identifier succeed and the remaining processors look for another adequate task. The Gantt chart of Fig. 2, resulted of applying the list scheduling algorithm to the task graph of Fig. 1, with the priority list L = [T_1, T_2, T_3, T_4, T_5, T_6, T_7].

3.1 ANOMALIES OF THE LIST SCHEDULING ALGORITHM

Using this heuristic, contrary to the intuition, some anomalies can happen. For example, as shown in Fig. 3, increasing the number of processors, decreasing the execution times of one or more tasks, or eliminating some of the precedence constraints can actually increase the makespan. In his work Graham [8] presented the following examples using the same priority task list *L* = [T_1, T_2, T_3, T_4, T_5, T_6, T_7,T_8,T_9] for each schedule.

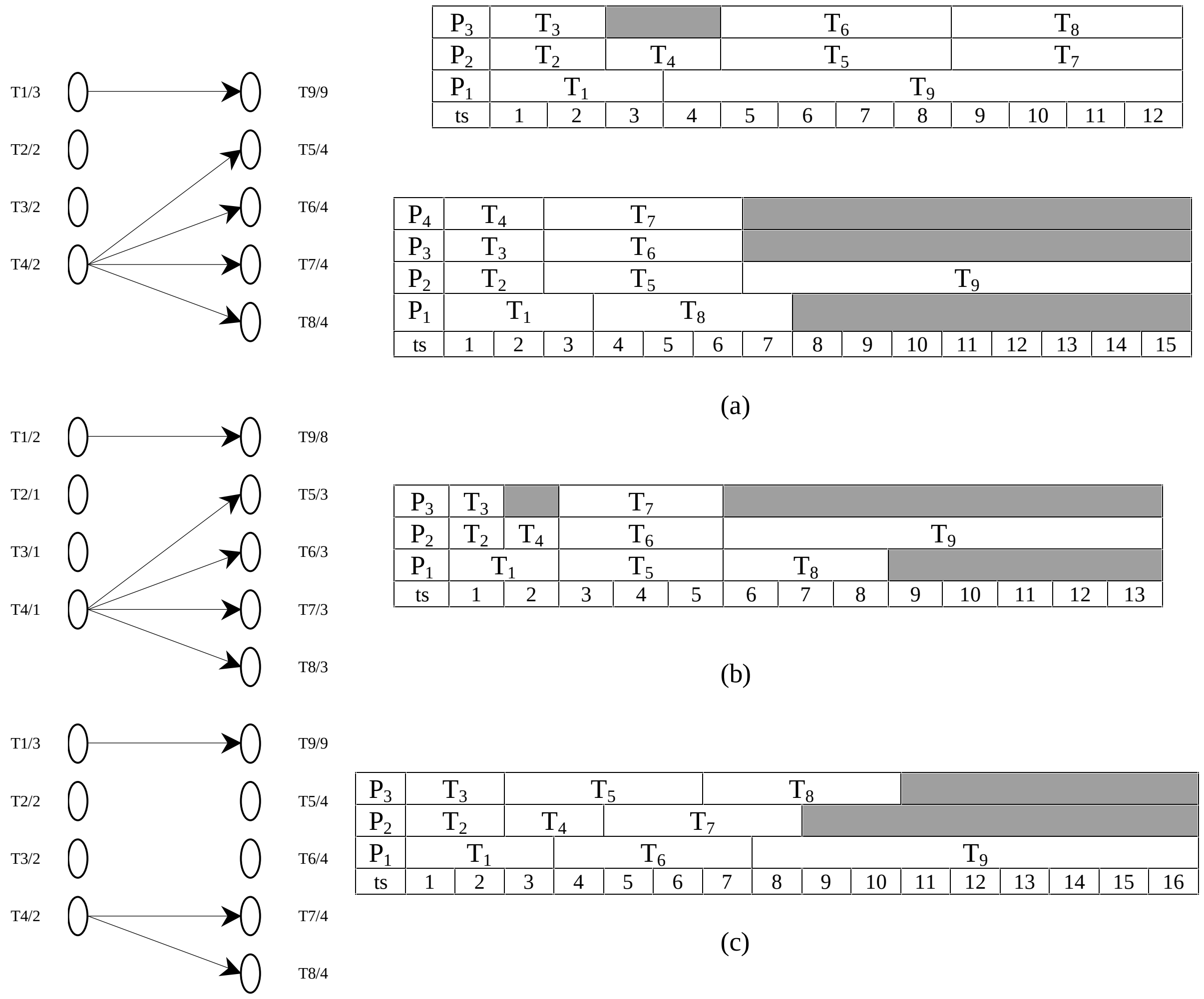

Fig. 3. Increasing number of processors, decreasing tasks duration and eliminating precedence constraints can increase the makespan using the Graham's heuristic.

4. USING EVOLUTIONARY ALGORITHMS TO PROVIDE NEAR-OPTIMAL SOLUTIONS

The task allocation problem has been investigated by many researchers [3] ,[4], [5], [6], [7], [10], [11]. Several heuristics methods has been proposed, such as mincut-based heuristics, orthogonal recursive bisection, simulated annealing, genetic algorithms and neural networks.
From the representation perspective many evolutionary computation approaches to the general scheduling problem exists. According to solution representation these methods can be roughly categorised as *indirect* and *direct representation* (Bagchi et al, 1991 [1]).
In the case of indirect representation of solutions the algorithm works on a population of encoded solutions. Because the representation do not directly provides a schedule a scheduler builder is necessary to transform a chromosome into a schedule, validate and evaluate it. The scheduler builder guarantees the feasibility of a solution and its work depends on the amount of information included in the representation.
In direct representation (Bruns 93 [2]) a complete and feasible schedule is an individual of the evolving population. The only method that performs the search is the evolutionary algorithm because the represented information comprises the whole search space.

We devised different evolutionary computation approaches to task scheduling, regarding two different representation schemes; direct and indirect. In this paper we present a direct approach with the *as-soon-as-possible* crossover method and an *indirect-decode* representation. We will concentrate only on results of direct representation.

4.1. DIRECT REPRESENTATION OF SOLUTIONS

Here we propose to use a schedule as a chromosome. Suppose we have two different schedules, (a) and (b) (Fig. 4), for the model task graph of Fig. 1, represented by the following Gannt charts.

P_2	▨	▨	T_3	T_4		T_6			▨	▨
P_1	T_1		T_2			T_5			T_7	T_8
ts	1	2	3	4	5	6	7	8	9	10

Schedule (a)

P_2	▨	▨	T_3	▨	▨	T_5			▨	▨	▨	T_8
P_1	T_1		T_2			T_4		T_6			T_7	▨
ts	1	2	3	4	5	6	7	8	9	10	11	12

Schedule (b)

Fig. 4. Feasible schedules for the model task graph.

The precedence relation described in the task graph can be properly represented in the corresponding precedence matrix A, where element a_{ij} is set to 1 if task i precedes task j and it is set to 0 otherwise. A gene in the chromosome can be the following four-tuple:

<*task_id, proc_id, init_time, end_time* >

where,
task_id, identifies the task to be allocated
proc_id, identifies the processor where the task will be allocated
init_time, it is the commencing time of the *task_id* in *proc_id.*
end_time, it is the termination time of the *task_id* in *proc_id.*

With this structure the list of the corresponding predecessors tasks is easily retrieved by entering the column of A indexed by the *task_id* value.
The corresponding chromosomes C_a and C_b for schedules (a) and (b) are:

C_a :

1,1,0,2	2,1,2,5	3,2,2,3	4,2,3,5	5,1,5,8	6,2,5,8	7,1,8,9	8,1,9,10

C_b:

1,1,0,2	2,1,2,5	3,2,2,3	4,1,5,7	5,2,5,8	6,1,7,10	7,1,10,11	8,2,11,12

This representation has a problem. If we use conventional crossover such as one-point crossover invalid offspring (unfeasible schedules) can be created. For example, if we decide to apply this operator after the fifth position we would obtain two invalid chromosomes.

$C_{a'}$:

1,1,0,2	2,1,2,5	3,2,2,3	4,2,3,5	5,1,5,8	6,1,7,10	7,1,10,11	8,2,11,12

$C_{b'}$:

1,1,0,2	2,1,2,5	3,2,2,3	4,1,5,7	5,2,5,8	6,2,5,8	7,1,8,9	8,1,9,10

Both of them violate the restriction that a processor processes a task at a time. Genes 5 and 6 in $C_{a'}$ and $C_{b'}$ describe invalid schedules where the same processor (P_1 for the case of $C_{a'}$ and P_2 for the case of $C_{b'}$) processes two tasks at the same time interval.

Penalty functions or repair algorithms can be used to remedy this situation [12]. Penalty functions of varied severity can be applied to invalid offspring in order to lower their fitness values but allowing them to remain in the population aiming to retain valuable genetic material.
Repair algorithms attempt to build up a valid solution from an invalid one. This approach is embedded in the *knowledge-augmented crossover* operator proposed by Bruns. Here a *collision* occurs if an operation (task processing) inherited from one of the parents cannot be scheduled in the specified time interval on the assigned processor. In this case the processor assignment is unchanged and it is delayed into the future until the processor is available.
In our example, this advanced crossover would generate the following chromosomes and corresponding feasible schedules (Fig. 5):

$C_{a''}$:

1,1,0,2	2,1,2,5	3,2,2,3	4,2,3,5	5,1,5,8	6,1,8,11	7,1,11,12	8,2,12,13

$C_{b''}$:

1,1,0,2	2,1,2,5	3,2,2,3	4,1,5,7	5,2,5,8	6,2,8,11	7,1,11,12	8,1,12,13

P_2	▨	▨	T_3	T_4		▨	▨	▨	▨	▨	▨	▨	T_8
P_1	T_1		T_2			T_5			T_6			T_7	▨
ts	1	2	3	4	5	6	7	8	9	10	11	12	13

Schedule (a”)

P_2	▨	▨	T_3	▨	▨	T_5			T_6			▨	▨
P_1	T_1		T_2			T_4		▨	▨	▨	▨	T_7	T_8
ts	1	2	3	4	5	6	7	8	9	10	11	12	13

Schedule (b”)

Fig 5. Feasible offspring schedules for the model task graph (Bruns).

As expected both children have a larger makespan but still are feasible. In the proposed Bruns's *knowledge-augmented* crossover only a child is generated where the part taken from the first parent build a consistent schedule. Then the assignment of the missing tasks are chosen from the second parent maintaining the assignment order and the processor allocations to tasks. Timing adjustments are included if neccesary. The latter decision can imply, as we showed, larger makespans for the children.

In our work we adopted an *as-soon-as-possible* (ASAP) approach similar to the Brun's proposal but modified because delays are avoided. This was implemented by moving the assignment to the earliest possible time, by random selection of one available processor at the ready time of the unassigned task. In this way no processor will remain idle if a task is available to be executed and the precedence constraints are satisfied.
The available processor is selected in way such to minimize the assignment changes in the part of the offspring corresponding to the second parent.
In our example this decision provides only one alternative and would give us the following chromosomes and their corresponding schedules, which differs from their parents in the assignments of tasks T_7 and T_8 only. See figure 6.

$C_{a'''}$:	1,1,0,2	2,1,2,5	3,2,2,3	4,2,3,5	5,1,5,8	6,2,5,8	7,1,8,9	8,2,9,10

$C_{b'''}$:	1,1,0,2	2,1,2,5	3,2,2,3	4,1,5,7	5,2,5,8	6,1,7,10	7,1,10,11	8,1,11,12

P_2	▨	▨	T_3	T_4		T_6			▨	T_8
P_1	T_1		T_2			T_5			T_7	▨
ts	1	2	3	4	5	6	7	8	9	10

Schedule (a''')

P_2	▨	▨	T_3	▨	▨	T_5			▨	▨	▨	▨
P_1	T_1		T_2			T_4		T_6			T_7	T_8
Ts	1	2	3	4	5	6	7	8	9	10	11	12

Schedule (b''')

Fig 6. Feasible offspring schedules for the model task graph (ASAP).

For mutation also a similar operator was conceived. If the chromosome undergoes mutation then a search is done, from left to right, until one gene is modified in the following way: choosing an alternative free processor or moving the assignment to the earliest possible time. This would imply modifying subsequent genes of the chromosome to create a valid offspring.

4.2. INDIRECT REPRESENTATION OF SOLUTIONS

Under this approach a schedule is encoded in the chromosome in a way such that the task indicated by the gene position is assigned to the processor indicated by the corresponding allele, as shown in Fig. 7:

processor	→	1	2	3	2	1	3	1	2
task	→	1	2	3	4	5	6	7	8

Fig. 7. Chromosome structure for the task allocation problem

The idea is to use a decoder. A *decoder* is a mapping from the space of representations that are evolved to the space of feasible solutions that are evaluated. Here the chromosome gives instructions to a decoder on how to build a feasible schedule. Regarding the task allocation problem, to build a feasible schedule, a decoder is instructed in the following way: By following the priority list, traverse the chromosome and assign the corresponding task to the indicated processor as soon as the precedence relation is fulfilled. Under this approach the restriction on avoiding processor idleness while a task is ready is relaxed. We believe that this less restrictive approach will contribute to population diversity. One advantage of decoders resides in their ability to create valid offspring even by means of simple conventional operators. One disadvantage is an slower evaluation of solutions. For the model task graph of Fig. 1, with three processors, parent chromosomes C_1 and C_2, and one point crossover after the fourth position we obtain the valid offspring $C_{1'}$ and $C_{2'}$.

C_1:	1	2	3	1	1	1	3	2
C_2:	3	1	2	2	3	1	2	3

$C_{1'}$:	1	2	3	1	3	1	2	3
$C_{2'}$:	3	1	2	2	1	1	3	2

The interpretation of these four decoders with priority task list L = [T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8] are the schedules of Fig. 8.

P_3	▨	▨	T_3	▨	▨	▨	▨	▨	▨	▨	T_7	▨
P_2	▨	▨	T_2			▨	▨	▨	▨	▨	▨	T_8
P_1	T_1		T_4		T_5			T_6			▨	▨
ts	1	2	3	4	5	6	7	8	9	10	11	12

a) Parent C_1

P_3	T_1		▨	▨	▨	T_5			▨	T_8
P_2	▨	▨	T_3	T_4		▨	▨	▨	T_7	▨
P_1	▨	▨	T_2			T_6			▨	▨
ts	1	2	3	4	5	6	7	8	9	10

b) Parent C_2

P_3	▨	▨	T_3	▨	▨	T_5			▨	T_8
P_2	▨	▨	T_2			▨	▨	▨	T_7	▨
P_1	T_1		T_4		▨	T_6			▨	▨
ts	1	2	3	4	5	6	7	8	9	10

c) Offspring C’$_1$

P_3	T_1		▨	▨	▨	▨	▨	▨	▨	▨	▨	T_7	▨
P_2	▨	▨	T_3	T_4		▨	▨	▨	▨	▨	▨	▨	T_8
P_1	▨	▨	T_2			T_5			T_6			▨	▨
ts	1	2	3	4	5	6	7	8	9	10	11	12	13

d) Offspring C’$_2$

Fig 8. Parents and feasible offspring schedules using decoders

Also, simplest mutation operators can be implemented by a simple swapping of values at randomly selected chromosome positions or by a random change in the allele. The new allele value identifies any of the involved processors.

5. EXPERIMENTS AND RESULTS

The preliminary experiments implemented a generational GA *with direct representation* of chromosomes and randomised initial population of size fixed to 50 individuals. Many runs were performed on five testing cases, using elitism, one point crossover and **big creep** mutation. The maximum number of generations was fixed to 100, but an stop criterion was used to accept convergence when after 20 consecutive generations, mean population fitness values differing in $\varepsilon \leq 0.001$ were obtained. Probabilities for crossover and mutation were fixed to 0.65 and 0.001.

The testing cases corresponded to:

Case 1:	Task graph of Fig. 1, excluding task 8 (7 tasks and 3 processors)
Case 2:	Task graph of Fig. 3.a (9 tasks and 3 processors)
Case 3:	Task graph of Fig. 3.a (9 tasks and 4 processors)
Case 4:	Task graph of Fig. 3.b (9 tasks and 3 processors, decreasing task's duration)
Case 5:	Task graph of Fig. 3.c (9 tasks and 3 processors, eliminating precedence constraints)

The following performance variables were considered to contrast the genetic approach (GA) versus the LSA:

Alt: Number of alternative solutions. It is the mean number of distinct alternative solutions found by the algorithm including optimum and non-optimum solutions.
Opt: : Number of optimal solutions. It is the mean number of distinct optimum solutions found by the algorithm per run.
Topt : Total number of optima. It is the total number of distinct optimal solutions found by the algorithm throughout all the runs.

Case	Alt		Opt		Topt	
	GA	LSA	GA	LSA	GA	LSA
1	10.6	1	9.2	1	57	1
2	4.2	1	4.2	1	15	1
3	9.1	1	9.1	-	56	-
4	3.6	1	3.6	-	10	-
5	4.0	1	4.0	-	10	-

Table 1. GA versus LSA, comparative performance

The stop criterion allowed to run the GA a number of generations between 40 to 80. In some of the alternative solutions, 1 or more processors remained idle (no tasks allocated to them). As the permutation of processors provides new alternative solutions, all the allocation list of an occupied processor can be switched to an idle one. Consequently a fault tolerance scheme can be implemented when the GA provides schedules with idle processors.

By observing table 1 the following comparisons can be done:

- The genetic approach found many and no a single optimal solution for any case as LSA does.
- All the anomalies observed with LSA do not hold when GA is applied, because:
    - When the number of processors is increased the minimum (optimum) makespan is also found.
    - When the duration of tasks is reduced this reduction is reflected in a reduced optimum makespan.
    - When the number of precedence restrictions is reduced the optimum makespan is preserved.

A more detailed analysis on each run detected that in most of the cases alternative solutions do not include, or include a low percentage, of non-optimal alternative solutions. That means that the final population is composed of many replicas of the optimal solutions due to a loss of diversity. This fact stagnate the search and further improvements are difficult to obtain.
To avoid this behaviour it would be necessary to continue experimentation with different parameter settings and recombination approaches.

6. Conclusions

The allocation of a number of parallel tasks in parallel supporting environments, multiprocessors or multicomputers, is a difficult and important issue in computer systems.
In this paper we approached allocation attempting to minimize makespan. Other performance variables such as individual processor utilization or evenness of load distribution can be considered.
Also results from this research can be applied to more general problems of parallel machine scheduling.
As we are interested in scheduling of arbitrary tasks graphs onto a reasonable number of processors, in many cases we would be content with polynomial time scheduling algorithms that provide good but no optimal solutions. The list scheduling algorithm (LSA) satisfy this requirement.
Here a genetic approach was undertaken to contrast its behaviour against the LSA. Preliminary results on the selected test suite showed two important facts. Firstly, GA provides not a single but a set of optimal solutions, providing fault tolerance when system dynamics must be considered. Secondly, GA is free of the LSA anomalies. This facts do not guarantee finding optimal solutions for any arbitrary task graph but show a better approach to the problem.
Consequently further research is necessary to investigate potentials and limitations of the GA approach under more complex test suites, different representations, and convenient genetic operators.

7. Acknowledgements

We acknowledge the cooperation of the project group for providing new ideas and constructive criticisms. Also to the Universidad Nacional de San Luis, and the ANPCYT from which we receive continuous support.

8. References

[1] Bagchi S., Uckum S., Miyabe Y., Kawamura K. – *Exploring problem-specific recombination operators for job shop scheduling*- Proceedings of the Fourth International Conference on Genetic Algorithms, pp 10-17, 1991.

[2] Bruns R. – *Direct chomosome representation and advanced genetic operators for production scheduling*. Proceedings of the Fifth International Conference on Genetic Algorithms, pp 352-359, 1993.

[3] Cena M.,Crespo M., Gallard R., - *Transparent Remote Execution in LAHNOS by Means of a Neural Network Device*- ACM Press, Operating Systems Review , Vol. 29, Nr. 1, pp 17-28, January 1995

[4] Ercal F.- *Heuristic approaches to Task Allocation for parallel Computing*- Doctoral Dissertation, Ohio State University, 1988.

[5] Flower J., Otto S., Salama M. – *Optimal mapping of irregular finite element domains to parallel processors*- Caltech C3P#292b, 1987.

[6] Fox G. C. –*A review of automatic load balancing and decomposition methods for the hipercube*- In M Shultz, ed., Numerical algorithms for modern parallel computer architectures, Springer Verlag, pp 63-76, 1988.

[7] Fox G.C., Kolawa A., Williams R. – *The implementation of a dynamic load balancer* - Proc. of the 2nd Conf. on Hipercube multiprocessors, pp 114-121, 1987.

[8] Graham R. L. – *Bounds on multiprocessing anomalies and packing algorithms*.- Proceedings of the AFIPS 1972 Spring Joint Computer Conference, pp 205-217, 1972.

[9] Horowitz E. and Sahni S – *Exact and approximate algorithms for scheduling nonidentical processors* – Journal of the ACM, vol. 23, No. 2, pp 317-327, 1976.

[10] Kidwell M. – *Using genetic algorithms to schedule tasks on a bus-based system*. - Proceedings of the Fifth International Conference on Genetic Algorithms, pp 368-374, 1993.

[11] Mansour N., Fox G.C. – *A hybrid genetic algorithm for task allocation in multicomputers*. Proceedings of the Fourth International Conference on Genetic Algorithms, pp 466-473, 1991.

[12] Michalewicz Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer Verlag , Third, Extended Edition, 1996.