EVOLUTIONARY APPROACHES FOR THE PARALLEL TASK SCHEDULING PROBLEM : THE REPRESENTATION ISSUE

ESQUIVEL S.C., GATICA C. R., GALLARD R.H.

Laboratorio de Investigación y Desarrollo en Inteligencia Computacional (LIDIC)[1]
E-mail: {esquivel, crgatica, rgallard}@unsl.edu.ar
Phone: + 54 2652 420823
Fax : +54 2652 430224

ABSTRACT

The problem of how to find a schedule on $m > 2$ processors of equal capacity that minimises the whole processing time of independent tasks has been shown as belonging to the NP-complete class (Horowitz and Sahni [12]). Evolutionary Algorithms (EAs) have been used in the past to implement the allocation of the components (tasks) of a parallel program to processors [12], [13], [14], [16], [17]. Those approaches showed their advantages when contrasted against conventional approaches and different chromosome representations were proposed.
This paper shows four algorithms to solve the problem of allocating a number of non-identical related 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. Three evolutionary algorithms, using an indirect-decode representation, are contrasted with the well-known Graham's [11] list scheduling algorithm (LSA). All of them use the conventional Single Crossover Per Couple (SCPC) approach and indirect-decode representation but they differ in what is represented by the decoders. In the first representation scheme, decoders represent processor dispatching priorities, in the second decoders represent tasks priority lists, and in the third decoders represent both processor dispatching priorities and tasks priority lists in a bipartite chromosome. Chromosome structure, genetic operators, experiments and results are discussed.

Key words: Parallel task allocation, Evolutionary algorithm, multirecombination, indirect-decode representation, List Scheduling Algorithm, Optimisation.

[1] The Research Group is supported by the Universidad Nacional de San Luis and the ANPCYT (National Agency to Promote Science and Technology). http://www-pr.unsl.edu.ar/proyecto338403//home_page.html.

1. INTRODUCTION

A parallel program is a collection of tasks, with precedence constraints to be satisfied among them. These precedence constraints between tasks are commonly delineated in a directed acyclic graph known as the *task graph*. Nodes in the graph identify tasks and their duration and arcs represent the precedence relationship. Factors, such as number of processors, number of tasks and task precedence make harder to determine a good assignment. Fig. 1 shows a task graph with six tasks and a simple single precedence constraint.

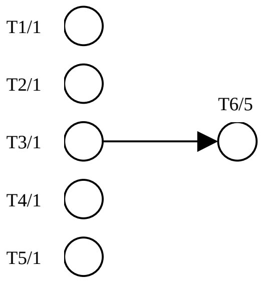

Fig. 1: A task graph

P_1	T_2	T_4	T_6			
P_2	T_1	T_3	T_5	▨			
Time slice	1	2	3	4	5	6	7

P_1	▨	T_6				
P_2	T_3	T_1	T_5	T_4	T_2	▨
Time slice	1	2	3	4	5	6

Fig. 2. Two feasible schedules for the task graph of figure 1. Related to the makespan the shortest schedule is optimal.

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 running on the available processors are indicated. By simple observation of the Gantt chart the completion time of the last task abandoning the system, called makespan, the processors utilisation, the speed up and other performance measures can be easily determined. Connected with the makespan, an optimal schedule is one such that the total execution time is minimised. Other performance variables, such as individual processor utilisation or evenness of load distribution can be considered (see fig.2). Some simple scheduling problems can be solved to optimality in polynomial time while others can be computationally intractable.

As we are interested in the scheduling of arbitrary task graphs onto a reasonable number of processors we would be content with polynomial time scheduling algorithms that provide good solutions even though optimal ones can not be guaranteed. In the following sections we discuss conventional and evolutionary algorithms to solve this problem on a set of selected instances.

2. A CONVENTIONAL APPROACH: 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:
$T=\{T, ...,Tn\}$ a set of tasks,
$\mu: T\rightarrow (0, \infty)$ a function which associates an execution time to each task,
$\leq$ a partial order in T and
L a priority list of tasks in T.
Each time a processor is idle, it immediately removes from L the first ready task; that is, an unscheduled task whose ancestors under $\leq$ 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 remain-

ing processors look for another adequate task. Using this heuristic, contrary to the intuition, some anomalies can happen. For example, 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. We are seeking for heuristics, which are free from these anomalies.

3. HEURISTICS TO FACE TASK SCHEDULING PROBLEMS

The task allocation problem has been investigated by many researchers [3], [4], [8], [9], [10], [13], [14]. Various heuristic methods have 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. With respect to solution representation these methods can be roughly categorised as *direct* and *indirect* representations [1]. In direct representation [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. In the case of indirect representation of solutions the algorithm works on a population of encoded solutions. Because the representation does not directly provide a schedule, a schedule builder is necessary to transform a chromosome into a schedule, validate and evaluate it. The schedule builder guarantees the feasibility of a solution and its work depends on the amount of information included in the representation. In this paper the idea is to use a *decoder*. A decoder is a mapping from the representation space into a feasible part of the solution space, which includes mappings between representations that evolve and representations that constitute the input for the evaluation function. This simplifies implementation and produces feasible offspring under different conventional crossover methods, avoiding the use of penalties or repair actions.

The problem is what to decode? In the problem we are facing, allocation of parallel related tasks onto processors, we can address the search by looking at the processors or the task priorities, or even by looking both at the same time.

In this work we propose three different EAs using different decoders and consequently different chromosome representations. To guide our search towards the optimal makespan, in the first representation processor dispatching priorities are encoded. In the second one, tasks priority lists are encoded, and in the third scheme a bipartite chromosome represents both processor dispatching priorities and tasks priority lists.

4. THE REPRESENTATION ISSUE

The representation adopted in an EA is crucial to its performance. A particular representation defines the areas of the problem space to be searched making the search more or less effective and determines possible genetic operators to be applied making the process more or less efficient. In the following section we discuss the indirect-decode representations used in this work.

4.1 PROCESSOR DISPATCHING PRIORITIES INDIRECT-DECODE REPRESENTATION

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. 3:

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

Fig. 3. Chromosome structure for the task allocation problem

The idea is to use a decoder to reflect which processor has the priority to dispatch a given task. Here the chromosome gives instructions to a decoder on how to build a feasible schedule. In our case the 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. The task priority list is defined at the beginning and remains the same during the search (i.e. canonical order). The processor priorities for dispatching tasks change while searching for an optimal schedule (minimum makespan).

Under this representation simple recombination operators such as one point crossover can be safely used. Also, simplest mutation operators can be implemented by a simple swapping of values at randomly selected chromosome positions or by a random change in an allele. The new allele value identifies any of the involved processors.

4.2 TASK PRIORITY LISTS INDIRECT-DECODE REPRESENTATION

In this second proposal each individual in the population represents a list of task priorities. Each list is a permutation of task identifiers. Here a chromosome is an n-vector where the i^{th} component is an integer in the range 1..(n – i +1). The chromosome is interpreted as a strategy to extract items from an ordered list L and builds a permutation. We briefly explain how the decoder works. Given a set of tasks represented by a list L and a chromosome C, the gene values in the chromosome indicate the task positions in the list L. The decoder builds a priority list L´ as follows: traversing the chromosome from left to right it takes from L those elements whose position is indicated by the gene value, puts this element in L´ and deletes it from L (shifting elements to left and reducing L length). It continues in this way until no element remains in L. Fig. 4 shows the list L, the chromosome C and the resulting priority list L´.

L =	1	2	3	4	5	6
chromosome C =	3	2	2	1	1	1
L' =	3	2	4	1	5	6

Fig.4: L is the task list, L' is the priority task list.

Once decoded, LSA as defined in section 2 (giving priority to the processor with lowest identifier when conflict arise), is used to build the schedule. As in the previous case simple genetic operators can be safely applied.

4.3 TASK PRIORITY LISTS AND PROCESSOR DISPATCHING PRIORITIES INDIRECT-DECODE REPRESENTATION

Under this representation scheme, a task priority list and a processor dispatching priority are encoded in the first and second half of a chromosome (see fig.5).

3	2	2	1	1	1	1	2	1	2	2	1
1	2	3	4	5	6	1	2	3	4	5	6

task priority list decoder

processor dispatching priority decoder

Fig. 5: Chromosome structure to combine decoders

Here, the left half of the chromosome is decoded first to obtain the corresponding task priority list L’. Then LSA is applied to build the schedule by using L´ and assigning each task to a processor according to the dispatching priority described in the second half of the chromosome. The simple genetic operators are applied now separately to each chromosome half.

5. EXPERIMENTS AND RESULTS

The experiments implemented evolutionary algorithms with *indirect* representation of solutions, randomised initial population of size fixed at 50 individuals for the smaller instances (1 to 8) and 100 individuals for the larger instances (9 to 12). Series of ten runs each were performed on the 12 testing cases, using elitism, one point crossover under SCPC, and big creep mutation. The maximum number of generations was fixed at 2000, but a 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 at 0.65 and 0.01, respectively. The testing cases corresponded to:

Instance 1: Task graph G1 (6 tasks and 2 processors)
Instance 2: Task graph G2 (7 tasks and 3 processors)
Instance 3: Task graph G3 (9 tasks and 3 processors)
Instance 4: Task graph G3 (9 tasks and 4 processors)
Instance 5: Task graph G4 (9 tasks and 3 processors, decreasing task’s duration)
Instance 6: Task graph G5 (9 tasks and 3 processors, eliminating precedence constraints)
Instance 7: Task graph G7 (10 tasks and 2 processors)
Instance 8: Randomly generated task graph G8 (13 tasks and 3 processors)
Instance 9: Randomly generated task graph G9 (25 tasks and 5 processors)
Instance 10: Randomly generated task graph G10 (50 tasks and 5 processors)
Instance 11: Randomly generated task graph G11 (25 tasks and 5 processors)
Instance 12: Randomly generated task graph G12 (70 tasks y 10 processors)

In contrast to other scheduling problems as Flow Shop or Job Shop, which have different machine environments and more oriented to production scheduling, after an intensive search in the literature we could find few benchmarks. The first 7 instances were extracted from the literature [11], [15] and they have known optima. Instances 8 to 12 were generated by random assignment of relations and tasks duration. Their optimum values are unknown. Nevertheless for these instances several trials were run (under this and previous approaches [5] to determine the best quasi-optimal solution under the whole set of contrasted algorithms. This value will be referred in what follows as an estimated optimal value (the best known value assumed as optimal). Diverse performance variables were considered to contrast the algorithms. To measure the *quality of solutions* provided by EAs we used:

Ebest: (Abs(opt_val – best value)/opt_val)/100. It is the percentile error of the best found individual in one run when compared with the known (or assumed) optimum value opt_val. It gives us a measure of how far are we from that opt_val.

Best: is the best (minimum) makespan value found throughout all the runs. This value has been reached by different solutions (schedules).

AvEbest: is the mean value of the error, over the total number of runs.

AvBest: is the mean value, over the total number of runs, of the best makespan found in each run (average best individual).

MinEbest: is the minimum value of the error, over the total number of runs.

We defined the *versatility* of an evolutionary algorithm as its ability to find diverse schedules of similar minimum makespan. To measure the versatility of the algorithms we used:

TotB defined as the total number of best solutions. It is the mean total number of distinct best solutions found by the algorithm throughout all runs.

In the following tables quality of solutions and versatility of the algorithms are contrasted. EA1 corresponds to the EA using processor dispatching priorities decoders, EA2 corresponds to the EA using tasks priority lists decoders, and EA3 corresponds to the EA using processor dispatching priorities and tasks priority lists decoders.

Regarding *quality of solutions*, from tables 1 and 2 it can be observed that EA2 is the best performer finding best minimum values (Best, Ebest) and best mean values (MBest, MEbest) in the hardest instances (9 and 10). Also, all EAs are free from LSA anomalies (instances 3, 4, 5 and 6).

As a general remark we can say that EA2, task priority list decoders, was the best representation and even more important it found, for instance 9, a new upper bound (270) which could not be reached by more sophisticated and costly multirecombinated evolutionary algorithms [5], [6], [7].
These results confirm the fact that if better representation is used less computational effort will be needed in order to provide high quality solutions.

Regarding *versatility*, table 3, except for hard instances 9 and 10, all EAs find more than a single best solution (distinct schedules with the same optimal or quasi optimal makespan). For instance 9 only EA2 finds a single optimum while for instance 10 non of the EAs here studied reach the opt_val, which was determined by means of a multirecombined EA [5], [6], [7]. In this table it can be observed that all EAs are more versatile than LSA, providing in most cases more than a single solution. EA3 is the algorithm that shows highest versatility.

Inst.	Opt-value	LSA		EA1		EA2		EA3	
		Best	Ebest	Best	Ebest	Best	Ebest	Best	Ebest
1	6	7	16.6	6	0.0	6	0.0	6	0.0
2	9	9	0.0	9	0.0	9	0.0	9	0.0
3	12	12	0.0	12	0.0	12	0.0	12	0.0
4	12	15	25	12	0.0	12	0.0	12	0.0
5	10	16	60	10	0.0	10	0.0	10	0.0
6	12	13	9.3	12	0.0	12	0.0	12	0.0
7	31	38	22.5	31	0.0	31	0.0	31	0.0
8	30	33	10	30	0.0	30	0.0	30	0.0
9	270	375	38.8	288	6.6	270	0.0	284	5.18
10	552	591	7.0	583	5.6	553	0.18	596	7.9
11	410	412	0.48	410	0.0	410	0.0	410	0.0
12	963	963	0.0	963	0.0	963	0.0	963	0.0
Avg.	193.08	207	15.8	197.1	1.01	193.16	0.01	197.91	1.09

Table1: Quality of solutions. Minimum makespans and minimum errors.

Inst.	Opt-value	EA1		EA2		EA3	
		MBest	MEbest	MBest	MEbest	MBest	MEbest
1	6	6	0.0	6	0.0	6	0.0
2	9	9	0.0	9	0.0	9	0.0
3	12	12	0.0	12	0.0	12	0.0
4	12	12	0.0	12	0.0	12	0.0
5	10	10	0.0	10	0.0	10	0.0
6	12	12	0.0	12	0.0	12	0.0
7	31	31	0.0	31	0.0	31	0.0
8	30	30.8	2.6	30	0.0	30	0.0
9	270	290.2	7.4	276.4	2.3	290.4	7.5
10	552	595.5	7.8	559.2	1.3	616.9	11.7
11	410	410	0.0	410	0.0	410	0.0
12	963	963	0.0	963	0.0	963	0.0
Avg.	193.08	198.45	1.4	194.21	0.3	200.19	1.6

Table2: Quality of solutions. Mean makespans and Mean errors.

Concerning convergence towards the best individual, the halting point of algorithms range from 60 to 200 generations in the smaller instances. For larger instances (9 to12) this halting point was ranging from 116 to 942 under EA2, but for EA1 and EA3 the algorithm halted at 2000 generations reaching the maximum number of generations allowed. We see that he best performer EA2 converges before reaching the maximum number of generations while this does not happens for EA1 and EA3. Consequently, we can expect better performance if we allow these versions to run for a longer time.

Inst.	LSA	EA1	EA2	EA3
	Totb	Totb	Totb	Totb
1	0	2	69	438
2	1	39	14	444
3	1	11	231	251
4	0	42	312	324
5	0	26	284	349
6	0	12	218	170
7	0	4	222	120
8	0	1	91	30
9	0	0	1	0
10	0	0	0	0
11	0	24	150	67
12	1	2	15	4
Avg.	0.25	13.58	133.91	183.08

Table 3: Versatility of the algorithms.

6. CONCLUSIONS

In this paper we approached the general problem of the allocation of related parallel tasks, described by a task graph, to a number of processors attempting to minimise makespan. This is a difficult and important issue in computer systems. 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 solutions even though optimal ones can not be guaranteed. The list scheduling algorithm (LSA) satisfies this requirement providing a fast, acceptable single solution, but it is prone to show anomalies.

In previous woks multirecombined EAs [5], [6], [7] being free of LSA anomalies, showed their effectiveness providing not a single but a set of near optimal solutions to diverse graphs belonging to a selected test suite.

In this paper the test suite was augmented and the conventional version of recombination (SCPC) was run under three different indirect decode representations. After analysing the results, task priority list decoders (EA2) was detected as the best representation. The minimum makespan value (270) found for instance 9, is a new value which could not be reach by more sophisticated (and costly) multirecombined evolutionary algorithms. As a general remark we can say that by using a better representation less computational effort will be needed to provide high quality solutions.

Further work will be oriented to enhance these algorithms and exploit the combined decoders approach expecting to obtain better results, with different parameter settings. Multirecombined versions of the indirect-decode approaches will be studied.

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 4th International Conference on Genetic Algorithms, pp 10 – 17 (1991)

[2] Bruns R.: Direct Chromosome Representation and Advanced Genetic Operators for Production Scheduling. Proceedings of the 5th 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, (1995).

[4] Ercal F.: Heuristic Approaches to Task Allocation for Parallel Computing. Doctoral Dissertation, Ohio State University, (1988).

[5] Esquivel S., Gatica C., Gallard R: Alternative Recombination Methods in Evolutionary Algorithms for the Task Scheduling Problem, invited session on Current Trends in Evolutionary Computation to Face Scheduling Problems, 4th. International ICSC Symposium on Soft Computing and Intelligent Systems for Industry, presentado y publicado en los proceedings, Paisley, Scotland, United Kingdom, June 2001, pp136.

[6] Esquivel S., Gatica C., Gallard R.: Conventional and Multirecombinative Evolutionary Algorithms for the Parallel Task Scheduling Problem, Lecture Notes in Computer Sciences 2037: “Applications of Evolutionary Computing”, pp. 223 – 232, Springer , Abril 2001.

[7] Esquivel S., Gatica C., Gallard R.: Evolutionary Approaches with Multirecombination for the Paralell Machine Scheduling Problem, XX Conferencia Internacional de la Sociedad Chilena de Ciencias de la Computación, Noviembre 2000, Santiago, Chile. **IEEE Publishing Co**, págs. 1 – 6.

[8] 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).

[9] 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).

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

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

[12] Horowitz E. and Sahni S.: Exact and Approximate Algorithms for Scheduling non Identical Processors. Journal of the ACM, vol. 23, No. 2, pp 317-327, (1976).

[13] Kidwell M. :Using Genetic Algorithms to Schedule Tasks on a Bus-based System. Proceedings of the 5th International Conference on Genetic Algorithms, pp 368-374, (1993).

[14] Krause M., Nissen,V.: On Using Penalty Functions and Multicriteria Optimization Techniques in Facility Layout. Evolutionary Algorithms for Management Applications, ed J. Biethahn and V. Nissen (Berling: Springer), pp 153-166, (1995).

[15] Pinedo M.,: Scheduling: Theory, Algorithms and Systems. Prentice Hall International Series in Industrial and Systems Engineering, (1995).

[16] Syswerda G.: Scheduling Optimisation using Genetic Algorithms. Davis, L., Editor, Handbook of Genetic Algorithms, chapter 21, pp 332 – 349, Van Nostrand Reinhold, New York, (1991).

[17] Withley D., Starkweather T., Fuquay D'A: Scheduling Problems and Travelling Salesman: The Genetic Edge Recombination Operator. Proceedings of the 3th International Conference on Genetic Algorithms, pp 133-140. Morgan Kaufmann Publishers, Los Altos CA, (1989)