A Genetic Algorithm for Scheduling Tasks in a Real-Time Distributed System

Yannick MONNIER
LESTER - Université de Bretagne Sud
2 rue Le Coat Saint-Haouen
56100 Lorient - FRANCE
monnier@iuplo.univ-ubs.fr

Jean-Pierre BEAUVAIS and Anne-Marie DÉPLANCHE
IRCyN (UMR CNRS 6597) - ECN, Université de Nantes
BP 92101 - 44321 NANTES Cedex 03 - FRANCE
beauvais@lan.ec-nantes.fr
deplanche@lan.ec-nantes.fr

Abstract

Real-Time systems must often handle several independent periodic macro-tasks, each one represented by a general tasks graph, including communications and precedence constraints. The implementation of such applications on a distributed system communicating via a bus , requires tasks assignment and scheduling, as well as the taking into account of the communication delays. As periodicity implies macro-tasks deadlines, the problem of finding a feasible schedule is critical. This paper addresses this NP-hard problem resolution, by using a genetic algorithm, under off-line and non-preemptive scheduling assumptions. This algorithm performances are evaluated on a large simulation set, and compared to classical list-based algorithms, a simulated annealing algorithm and a specific clustering algorithm.

Keywords : *Genetic Algorithms, Scheduling algorithms, Real-time tasks, Distributed computer control systems.*

1. Introduction

Computing real-time systems are increasingly used to control processes in numerous application fields like aircraft, automotive, manufacturing process. These systems have to deal with a major constraint : the computation results must be provided inside a delay which allows the system to keep the process under its control, or else serious damages can occur. To ensure shorter delays, computational tasks can be concurrently executed on a distributed system. This implies to be able to resolve the tasks mapping and scheduling problem.

This paper focuses on a Genetic Algorithm (GA) implementation to solve a complex real-time tasks mapping and scheduling problem. The objective is the pre-run-time distribution of a software configuration onto a distributed hardware system, guarantying the respect of temporal constraints. The software configuration is composed of several independent *periodic macro-tasks*, considering ends of periods as deadlines. Each macro-task is defined by a directed weighted acyclic graph, representing non-preemptive computational tasks, communications, and precedence constraints. The hardware architecture is composed of several identical processor sites, communicating through a bus-like network which allows only one communication at a time. Numerous heuristics have been developed to solve different mapping or scheduling problems, often NP-hard. Our problem is here near the Ramamritham's one treated in [1]. A few researchers have already experimented GA to solve nearby problems [2, 3], but none of them have included periodicity topics and bus contention as ours.

GA is an *optimization method* developed by Holland [4] to mimic the adaptive mechanism of natural systems, and which has been adapted with great success to various NP-hard problems including scheduling problems as JSP (Job-shop Scheduling Problem). It seems to become an excellent competitor with the more classical methods, *Simulated Annealing* (SA) and *Hill-Climbing* (HC). GA differs from these methods by many ways, mainly:

i) it simultaneously explores several points (the "*population*") of the searching space, each point represented by an "*individual*".

ii) it manipulates a coding of the solution (a "*chromosome*") rather than the solution itself.

iii) new individuals are produced using *probabilistic* operations over the population.

iv) population evolution is based on the survival of the *fittest* individuals.

By this way, local extrema have a lower attraction on GA which can reach several of them to select the best, whereas other methods generally stop when reaching one of them. Using the fact that GA is intrinsically parallel [5], it is possible to drastically reduce its computation time. As disadvantages, the GA larger exploration generally implies a longer computation time, and as all stochastic techniques, each run of the algorithm produces its own results, depending on the random features. We'll show the encouraging effectiveness

of our "standard implementation" GA (described in detail in [6]), compared with several other algorithms, over a large simulation set.

2. The problem to resolve

2.1. Formulation

The problem is to find a pre-run-time schedule for the logical software components onto the required physical resources in hard real-time systems. It is concerned with three closely related problems :

- *task allocation* : the allocation of the software objects of the logical architecture to processors of the distributed physical architecture.

- *network scheduling* : managing the shared resource of the network so as to evaluate and to take into account message delays.

- *processor scheduling* : determining the schedule which will ensure that all macro-tasks, according to their allocation, will meet their deadlines.

The distributed system consists of a set P of m sites, $P = \{p_j; j = 1, \ldots, m\}$, each with one processor. The sites are identical, *i.e.* the speeds of all processors are equal. They are fully connected by a multiple access network. It is assumed that the contention-free communication delay does not depend on the distance between sites, but on the amount of exchanged data. However, when two communicating tasks are allocated on the same site, the communication delay between them is considered to be negligible and is set to zero.

The software system is modeled as a set G=$\{G_i; i = 1, \ldots, g\}$ of g periodic hard real-time macro-tasks whose deadline is the end of the period. Due to communication and precedence constraints, a macro-task G_i is a directed acyclic weighted graph of period τ_i and is defined by a tuple $G_i = (S_i, E_i, \tau_i)$. $S_i = \{s_{ij}; j = 1, \ldots, n_i\}$ which denotes the set of task nodes or tasks s_{ij} of the graph G_i. Each task is valued by its worst case processing time with no preemption on a processor : q_{ij}. The set E_i represents the oriented edges symbolizing the precedence and communication relations which may exist between two tasks of a macro-task. $E_i = \{e_{ijl}; j = 1, \ldots, n_i; l = 1, \ldots, n_i\}$ means that the task s_{ij} precedes s_{il}. Each edge is weighted by c_{ijl}, the communication time incurred along the edge (measured as if the network were dedicated exclusively for the communication between the two tasks). The only resource constraints we consider are thus restricted to the processor and the bus. The model considered is deterministic, *i.e.* all the hardware and software characteristics are assumed to be known *a priori*.

Because of the periodic behavior of macro-tasks, the study can be restricted to the interval $[0, L]$ where L is the least common multiple of the periods $\tau_i (i = 1, \ldots, g)$. Each macro-task G_i will give rise to L/τ_i instances in the time interval $[0, L]$. Its instance number k ($k \in [1, \ldots, L/\tau_i]$) is denoted $G_i^k = (S_i^k, E_i^k)$ with $S_i^k = \{s_{ij}^k\}$ and $E_i^k = \{e_{ijl}^k\}$.

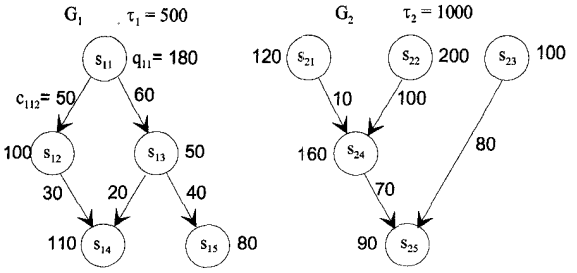

Figure 1. A software configuration of two macro-tasks

The problem is then to find three applications (A, S, N) :

- **mapping application** A which associates a processor to each task

$$\bigcup_{i=1}^{g} S_i \xrightarrow{A} P$$
$$s_{ij} \longmapsto A(s_{ij}) = p_u, u \in [1..m]$$

- **scheduling application** S which associates, to each task instance s_{ij}^k over L, a start time $st(s_{ij}^k)$.

$$So \xrightarrow{S} I\!R$$
$$s_{ij}^k \longmapsto S(s_{ij}^k) = st(s_{ij}^k)$$

if G_i^k is the k^{th} instance of G_i over L, $G_i^k = (S_i^k, E_i^k) = \left(\{s_{ij}^k\}, \{e_{ijl}^k\}\right)$, then $So = \bigcup_{i=1}^{g} \left(\bigcup_{k=1}^{L/\tau_i} S_i^k\right)$

- **scheduling application** N which associates to each communication to be scheduled during L, a start time $st(e_{ijl}^k)$:

$$Eo \xrightarrow{N} I\!R$$
$$e_{ijl}^k \longmapsto N(e_{ijl}^k) = st(e_{ijl}^k)$$

$$E_0 = \bigcup_{i=1}^{g} \left(\bigcup_{k=1}^{L/\tau_i} \left\{e_{ijl}^k \in E_i^k | A(s_{ij}) \neq A(s_{il})\right\}\right)$$

while **respecting the constraints :**

- (1) **ready time = start of period**

$$\forall s_{ij}^k \in So | Pred(s_{ij}) = \emptyset,\ st(s_{ij}^k) \geq (k-1) \times \tau_i$$

with $Pred(s_{ij})$ as the predecessors set of s_{ij}.

- (2) **deadline = end of period**

$$\forall s_{ij}^k \in So,\ ft(s_{ij}^k) \leq dt(s_{ij}^k) = k \times \tau_i$$

with ft as the finish time ($ft(s_{ij}^k) = st(s_{ij}^k) + q_{ij}$ and $ft(e_{ijl}^k) = st(e_{ijl}^k) + c_{ijl}, \forall e_{ijl}^k \in E_0$) and dt as the deadline time.

- (3) **precedence and communication**

$$\forall s_{ij}^k \in So | Pred(s_{ij}) \neq \emptyset,$$

$$st(s_{ij}^k) \geq \max \begin{pmatrix} \max\limits_{s_{il} \in Pred(s_{ij}) \wedge A(s_{ij}) = A(s_{il})} ft(s_{il}^k) \\ \max\limits_{e_{ilj}^k \in Eo} ft(e_{ilj}^k) \end{pmatrix}$$

$$\forall e_{ijl}^k \in Eo, st(e_{ijl}^k) \geq ft(s_{ij}^k)$$

2.2. The global graph

The algorithms have to deal with a global graph obtained by concatenating the macro-task instances over L. This graph owns dummy initial and final tasks. Dummy edges have been introduced to connect these dummy tasks and also the macro-task instances between two consecutive periods. Fig.1 shows an example of a task configuration we have to deal with, whereas Fig.2 depicts the global graph that results from it.

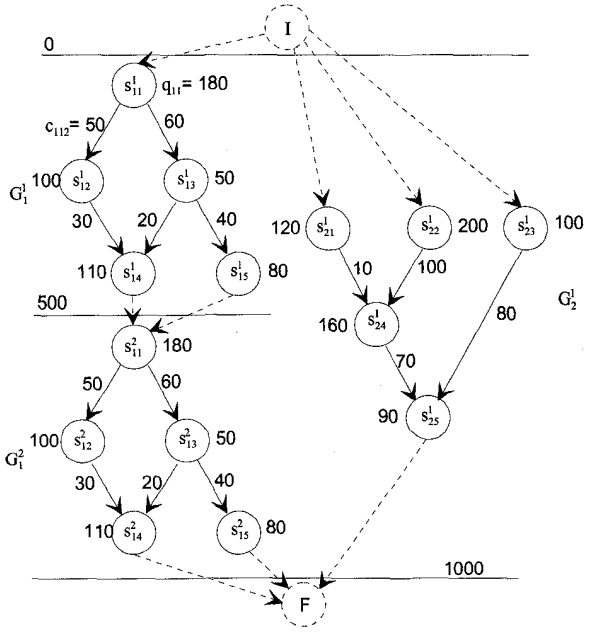

Figure 2. A global graph

3. The Genetic Algorithm

3.1. Cost function choice

As GA is an optimization method, we must define a cost function. The searching space can be seen as a set

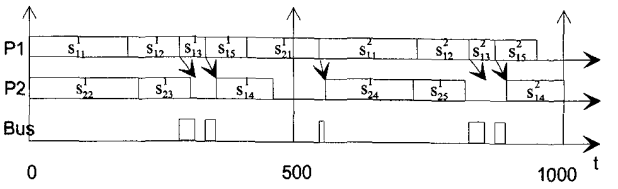

Figure 3. A solution to the problem

$H = \{h_u, u = 1, \ldots\}$ of *schedules* defining the allocation and the start time of all the tasks instances s_{ij}^k and of communications e_{ijl}^k (Fig.3) corresponding to the three applications defined earlier (task allocation, network scheduling, task scheduling), verifying all listed constraints except deadlines meeting ((1),(3)). This one is "included" in the cost $tard(h_u)$ of a schedule, called *tardiness*, using the calculation of the *tasks instances delay*, $tid(s_{ij}^k)$: $tid(s_{ij}^k) = 0$ if $ft(s_{ij}^k) \leq dt(s_{ij}^k)$; meaning s_{ij}^k meets its deadline ; $tid(s_{ij}^k) = ft(s_{ij}^k) - dt(s_{ij}^k)$ else. $tard(h) = \sum_k \sum_i \sum_j tid(s_{ij}^k)$ where s_{ij} has no successor (s_{14} , s_{15} , and s_{25} for our example Fig.1). In this way, the objective is to **find a zero tardiness schedule**, by minimizing the *tardiness* function over the searching space H.

3.2. Coding

According to the main principle of GA, our implementation does not directly manipulate the *schedules* of the searching space H, but a coded representation of them, the *chromosomes*. The choice of this coding is a major feature of a GA, because it defines the links between "genetic area" and solutions [7], and constrains the genetic operators choice. We adopted a string representation specially adapted to our particular problem (Fig.4), and which allows using "standard" crossover operators. Each task s_{ij} is represented by a complex gene which includes both its allocation $A(s_{ij})$ and a *priority number* $Pr(s_{ij}^k)$ for each of its instances. This *priority number* determines the task instances execution order on each processor.
$(\forall (i,j,k), \forall (i',j',k')/A(s_{ij}) = A(s_{i'j'}),$
$Pr(s_{ij}^k) < Pr(s_{i'j'}^{k'}) \Rightarrow st(s_{ij}^k) < st(s_{i'j'}^{k'}))$

Thus, the individual illustrated Fig.4 represents a schedule respecting the sequences built by ordering, on each processor, the task instances in ascending order of their *priority number* :

$$\text{site 1} : s_{11}^1 - s_{12}^1 - s_{14}^1 - s_{11}^2 - s_{12}^2 - s_{14}^2 - s_{23}^1$$
$$\text{site 2} : s_{13}^1 - s_{21}^1 - s_{22}^1 - s_{15}^1 - s_{24}^1 - s_{25}^1 - s_{15}^2 - s_{13}^2$$

Note that to be able to represent a *valid* schedule, a chromosome must take into account the precedence constraints.

	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	s_{21}	s_{22}	s_{23}	s_{24}	s_{25}
$A(s_{ij})$	1	1	2	1	2	2	2	1	2	2
$Pr(s^1_{ij})$	2	3	1	5	7	4	6	14	9	10
$Pr(s^2_{ij})$	8	11	15	12	13					

Figure 4. A chromosome for our software configuration on a two sites system

This imposes two conditions on the *priority numbers* distribution :

i) the precedence relations must be respected on each processor (inter-processors precedence relations will be treated as network communications). Our example Fig.4 does not fit because on site 2, s^2_{15} and s^2_{13} *priority numbers* are not in accordance with s_{13} to s_{15} relation.

ii) it must not create a *cycle*, illustrated by the sequences :

$$\text{site 1 : } s^1_{12} - [...] s^1_{21} - [...]$$
$$\text{site 2 : } s^1_{24} - [...] s^1_{11} - [...]$$

where s^1_{12} must wait for the communication from s^1_{11} which can start only after s^1_{24} , which must wait for the communication from s^1_{21} , and this one can start only after s^1_{12} ! No task can be scheduled.

We'll systematically *repair* the chromosomes which do not respect condition i), by recursively swapping the *priority numbers* of each pair of tasks which violates this condition. Unlike the first one, we will not take care of condition ii) when creating chromosomes, because it will be often "naturally" respected, and we wanted to avoid time consuming and influence of a repair algorithm. So, we decided to detect and break possible cycles only during the schedule building (see 'Chromosome evaluation' below), without any change in the chromosome. In this way, we can maintain execution order near the chromosome's *priority numbers*.

3.3. Chromosome evaluation

The purpose of the evaluation is to calculate the *tardiness* of the schedule corresponding to a chromosome. For this, we build, according to the data of the chromosome, a schedule - *i.e.* an "execution sequence" - for the *global graph* described in Section 2.2. Let us call, from now, a *task*, each of the n non-dummy node of this *global graph* ($n = \sum_i (L/\tau_i) \times n_i$), previously named *task instance*. The way the schedule is built is as follows :

Step 1 : initialize.
For each site fill up a site list with the tasks to be scheduled on it, in ascending order of their priority number. Initialize an empty network list, to be filled up later with the inter-sites communication requests.

Step 2 : repeat steps 21 to 24 until all lists are empty.
Step 21: schedule tasks which don't need network communication.
While the first task in any site list is *ready* - *i.e.* all its incoming communications are scheduled - do step 211 to 213.
Step 211: schedule this first task s^k_{ij} on its site :

$$st(s^k_{ij}) = \max \begin{pmatrix} \max_l(ft(e^k_{ilj})), \\ (k-1) \times \tau_i, \\ \max_{A(s_{i'j'})=A(s_{ij}) \wedge Pr(s^{k'}_{i'j'}) < Pr(s^k_{ij})} ft(s^{k'}_{i'j'}) \end{pmatrix}$$

$ft(s^k_{ij}) = st(s^k_{ij}) + q_{ij}$
Step 212: remove it from the list.
Step 213: treat all its outgoing communications ($\{e^k_{ijl}\}$) : add inter-sites ones ($\{e^k_{iju} | A(s_{iu}) \neq A(s_{ij})\}$) in the network list, with a *request time* $rt(e^k_{iju}) = ft(s^k_{ij})$, and immediately schedule the others with no communication delay : $ft(e^k_{ijv}) = st(e^k_{ijv}) = ft(s^k_{ij})$.
Step 22: detect a *cycle*
If network list is empty do step 23 else do step 24
Step 23: break the *cycle*.
Find, among all site lists, the nearest (from the top of its list) ready task.
Shift it to the top of its list.
Step 24: schedule the earliest communication - *i.e.* having the minimum *request time* - in the network list.
$st(e^k_{ijl}) = \max(rt(e^k_{ijl}), ft(e^{k'}_{i'j'l'}))$ where $e^{k'}_{i'j'l'}$ is the latest communication scheduled on network.
$ft(e^k_{ijl}) = st(e^k_{ijl}) + c_{ijl}$
Remove it from the network list.

3.4. Initial population

In order to set a great diversity in the initial population of size *Pop_Size*, each chromosome is randomly generated as follows : first, affectation is determined by randomly choosing each $A(s_{ij})$ in {1,...,m}, then, the numbers {1, ..., n} are randomly distributed over the n priority numbers $Pr(s^k_{ij})$. Finally, the chromosome is repaired in order to fit condition i).

3.5. Selection : fitness calculation

Selection is the main way GA mimics evolution in natural systems : fitter an individual is, the highest is its probability to be selected. The commonly strategie called *"Roulette Wheel Selection"* has been used, with the *"Linear Fitness Normalization technique"* [7]. The fitness $F(h)$ of each individual is calculated as follows (see Fig.5) :
1 - individuals are ordered in increasing order of their *tardiness* (so, from "best" to "worst").

2 - the first one receives the maximum *fitness* value $F(1) = Max\_Fit$.
3 - the others receive linear decreasing fitness values $F(h) = F(h-1)$ - Dec_Fit.
We chose $Max\_Fit = Pop\_Size$, and $Dec\_Fit = 1$, so the "worst" individual has a chance to be selected.
The roulette is then built by calculating $CT(h)$, the *"cumulative total"* of the fitness values. The selection consists in generating a random number $r \in [1, \sum_i F(i)]$ and in choosing the individual h for which $CT(h-1) < r \leq CT(h)$. Fig.5 illustrates the selection of individual B in a 5 individuals population, as the result of a random number $r = 8$.

individual	A	B	C	D	E
tardiness	51	82	243	245	250
fitness	5	4	3	2	1
cumulative total	5	9	12	14	15

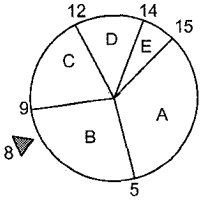

Figure 5. Roulette Wheel Selection with Linear Normalization fitness

3.6. Crossover operator

The classical *one point crossover operator* [8] has been used. This operator creates two new individuals (*the children*) by mating two chromosomes (*the parents*), which are combined as shown in Fig.6 : the parent chromosomes are cut in two parts at a unique crossover point randomly chosen, and each child chromosome is built with the first part of one parent and the second part of the other. As *invalid* individuals can result from this "mixing", we repair all the chromosomes produced by this operator.

3.7. Mutation operators

The purpose of mutation is a little change in the chromosome. As our coding includes two aspects, we use two mutation operators, one working on allocations, the other one on priorities.
The first one randomly chooses a gene in the chromosome, and changes its *allocation* for a different one, randomly chosen too. The second swaps two *priority numbers* randomly selected in the chromosome. In the two cases, we also need to repair the mutated chromosome.

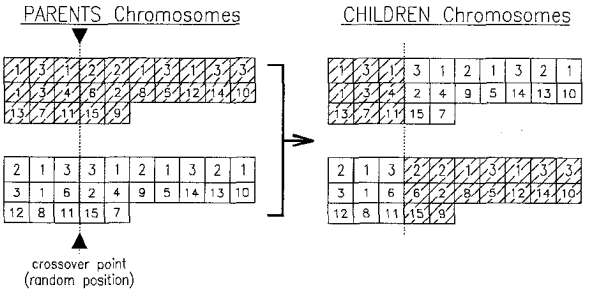

Figure 6. Crossover operator

3.8. Reproduction and population replacement

During reproduction step, *Chd_Nb* children are created by mating, with probability Pc, pairs of parents selected in the current population. These children are then mutated with probability Pm, randomly using one of the mutation operator. Finally, their tardiness are calculated, and they are added to the current population.
Replacement step consists in discarding the *Chd_Nb* individuals with the higher *tardiness* values, to build the constant-size new population with the Pop_Size fittest individuals.
This iterative evolution process is stopped as soon as one solution is found. However, we limit the number of children produced to *Max_Indiv*, in order to avoid prohibitive calculation time, and to ensure that the GA will stop when treating an unfeasible problem. Consequently, our Genetic Algorithm obeys to the informal pseudo-code given Fig.7.

- Generate the Pop_Size chromosomes of the initial population and evaluate them.
- Repeat
  - Produce *Chd_Nb* "children" chromosomes by mating, with probability Pc, pairs of "parents" chromosomes selected in the population.
  - Mutate each child, with prob. Pm.
  - Evaluate the children.
  - Add the children to the population.
  - Discard the *Chd_Nb* individuals having the highest *tardiness*.
- Until either a zero *tardiness* individual is found either more than Max_Indiv "children" have been produced.
- Return success if a zero *tardiness* schedule have been found.

Figure 7. Our GA informal pseudo-code

4. Simulation results

4.1. The benchmark

The evaluation of our AG performances has been done over 226 randomly generated software configurations. Our *"tasks set generator"* takes a lot of parameters as inputs : the *number of macro-tasks* g, varies from 2 to 6 ; the *number* n_i *of tasks* in a macro-task G_i is randomly set to 6 or 10 ; the *periods* are either all equal to 1000, either different and alternatively equal to 400, 1000, and 500, so that L can rise 2000 ; the *laxity*, which is the ratio between the size of the period τ_i and the amount of processing time $\sum_j q_{ij}$, varies from 0.9 to 1.6 ; the *graph types* of the macro-tasks are chain (each node has one predecessor and one successor), tree (one successor), rtree (one predecessor) or general.
A generated configuration is rejected if the minimum necessary computing time (according to the precedence constraints) of a macro-task is larger than its period. Each non rejected configuration is tested for a *number of sites* from 2 to 6, if the system is not *a priori* overloaded, thus realizing a set of 725 problems.

4.2. The results

We have submitted the same benchmark to many different algorithms, and we have selected the best of them to be compared with our GA. Each is coded on the following figures, corresponding to :

Ga, the genetic algorithm, uses these parameters : $Pc = 0.7$; $Pm = 0.3$; $Pop\_Size = 100$; $Chd\_Nb = 50$; its results include the successes obtained by three consecutive runs of the GA, limited to 400 and 800 generations ($Max\_Indiv = 20,000$ and $Max\_Indiv = 40,000$).

List_Est, List_Lst are classical list algorithms which allocate and schedule the tasks of the global graph one at a time, in a stepwise way. List_Est gives the priority to the ready task with the smallest earliest-start time while List_Lst gives it to the ready task with the smallest latest-start time. The particularity of these algorithms is that a decision of allocation can be postponed if a task with a better priority may appear in a near future.

Clus, a clustering algorithm, in a first phase, aims to reduce the communications by clustering "heavy communicating" tasks. After this clustering phase, all the tasks belonging to a cluster must be allocated to the same processor. The second phase produces a mapping and a scheduling, using a list algorithm which takes as priority, the smallest latest-start time of task.

Sa, is a classical *Simulated Annealing* method which allows to partially explore the search space of complete solutions by using the "Metropolis rule" [9].

All these algorithms are described in detail in [10]. Before computing the *success ratio*, we discard the possible unfeasible problems remaining in our benchmark, in spite of the verifications applied (deciding of the feasibility is a NP-complete problem !), by restricting the benchmark to the 638 problems solved by at least one algorithm. So, the *success ratio* of an algorithm is equal to the number of problems solved by this algorithm divided by the restricted benchmark number of problems.

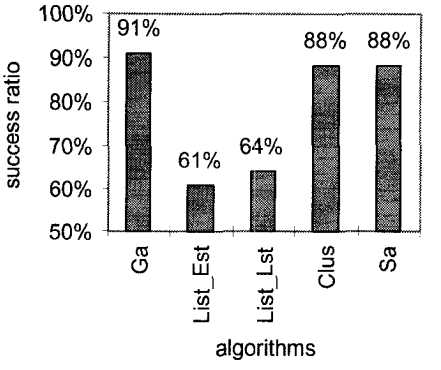

Figure 8. Overall simulation results

The overall results in Fig.8 show the great advantage of the GA over the classical heuristics, and we can see that its result is the highest over all the algorithms we tested, outpacing simulated annealing and clustering algorithms, which obtained the highest results in [10]. Fig.9 presents detailed results for two important characteristics of the problem : the number of sites, which has an effect on the size of the research space, and the communication ratio (the amount of communication times divided by the amount of processing times, calculated over the global graph). The first one shows that the GA obtains the best success ratio for almost all numbers of sites, even reaching 100% and 99% for 2 and 3 sites. It is only outpaced by the simulated annealing and the clustering algorithms for 6 sites. The second one shows that the GA results are on the top for the lowest com.ratio ranges, being really outpaced - always by simulated annealing and clustering algorithm - only for the 0.8-0.99 range (the highest range is not representative, with only 18 problems).

5. Conclusion

In this paper we have described a "standard" Genetic Algorithm implementation to solve a periodic real-time tasks mapping/scheduling problem. The objective is to find a

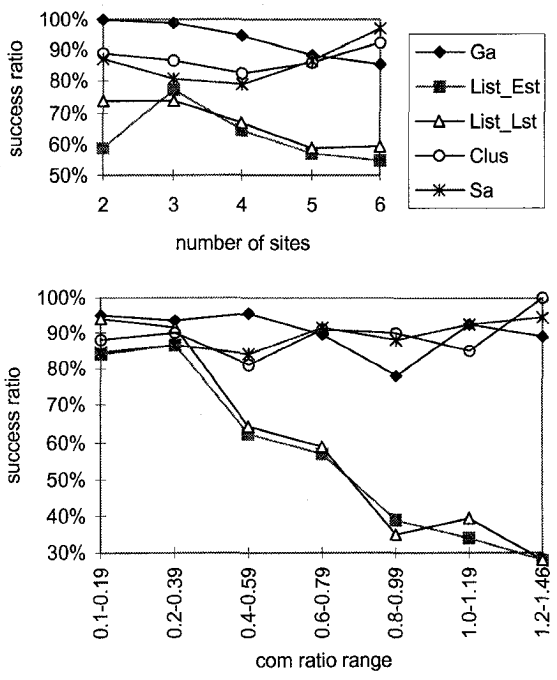

Figure 9. Simulation results - details

valid schedule - *i.e.* respecting all the constraints : precedence, communication, periodic activation, network contention, deadline - among all the possible schedules. As this problem is well known to be NP-hard, search heuristics are commonly used. Expressing this problem as an optimization problem allows the use of a GA, with the aim of exploring a larger research space.

This algorithm behavior has been compared with several other kinds of algorithms, over a large simulation set, randomly generated. We have noticed its very encouraging abilities, in spite of its standard character. However, we think our GA is actually nearing the maximum ability of a standard GA implementation, because doubling the generations limit (up to 1600) for 6 sites problems, did not bring any improvement. Indeed, numerous papers about GA insist on the necessity to include problem specific genetic operators in order to improve GA performances (*hybridized Genetic Algorithms* in [7]).

This determines the next step of our GA study. We plan to design specific operators in order to facilitate the convergence by restricting its research space. The preceding results show that clustering is a feature that seems to bring strong abilities for the highest numbers of sites and communication ratios. So, we'll try to include a "cluster oriented" operator in our GA, with the aim of giving it enough ability to reach its highest results all over our simulation set.

References

[1] K. Ramamritham "Allocation and scheduling of complex periodic tasks", *10th Int. Conf. on Distributed Computing Systems*, Paris, France, jun. 1990.

[2] H. Mitra, P. Ramanathan "A Genetic Approach For Scheduling Non-preemptive Tasks With Precedence and Deadline Constraints", *26th Hawaii International Conference on systems sciences*, vol II, pp 556-564, jan. 1993.

[3] E.S.H. Hou, N. Ansari, H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling", *IEEE Transactions on parallel and distributed systems*, vol 5, no 2, pp 113-120, feb. 1994.

[4] J.H. Holland, "Adaptation in natural and artificial systems", *Ann Arbor* - The University of Michigan Press, 1975.

[5] E-G. Talbi, T. Muntean, "Hill Climbing, Simulated Annealing and Genetic Algorithms : A Comparative Study and Application to The Mapping Problem", *26th Hawaii Int. Conf. on systems sciences*, vol II, pp 565-573, jan. 1993.

[6] Y. Monnier, "Un Algorithme Génétique pour le problème du Placement-Ordonnancement de Tâches Temps Réel", *Rapport de DEA*, Université de Nantes-ECN, septembre 1997.

[7] L. Davis, "Handbook of genetic algorithms", *Van Nostrand Reinhold*, 1991.

[8] D.E. Goldberg "Genetic Algorithms in Search, Optimization & Machine Learning", *Addison-Wesley*, 1989.

[9] S Kirkpatrick, C.D. Gelatt and M.P. Vecchi, "Optimization by Simulated Annealing", *Science*, 220(4598):671-680, May 1983.

[10] J-P. Beauvais and A-M Déplanche, "Affectation de tâches dans un système temps réel réparti", *Technique et science informatiques*, Vol 17, no 1, Hermes, janvier 1998.