Heuristics to Optimize the Speed-Up of Parallel Programs

AGUILAR Jose

Departamento de Computación. Facultad de Ingeniería
Universidad de los Andes. Av. Tulio Febres Cordero
Mérida - Estado Mérida. 5101. Venezuela
email: aguilar@ing.ula.ve jose@math-info.univ-paris5.fr
Telf : (58.74) 440002 Fax: (58.74) 402979

Abstract. In this paper we propose methods to optimize the speed-up which can be obtained for a parallel program in a distributed system by modelling the assignment of the tasks of a parallel program as a graph partitioning problem. The tasks (set of instructions that must be executed sequentially) which compose the program are represented by weighted nodes, and the arcs of the graph represent the precedence order between tasks. Because this problem is in general NP-hard we propose and investigate several heuristic algorithms, and compare their performance. The approaches we present are: a neural network based algorithm (based on the random neural model of Gelenbe), an algorithm based on simulated annealing and a genetic algorithm based heuristic.

1. Introduction

In the context of parallel systems, the task assignment problem is an important issue. It is closely related to which is the manner of implementing task assignment so that the program's parallel execution is speeded up. Both directed and non-directed graphs may be used to represent parallel programs. A non-directed graph will only represent information exchange and concurrent execution between tasks, while a directed graph will deal with precedence relations concerning execution sequences of tasks, which may also be accompanied by information exchange. Both nodes and arcs of the graph model will in general be weighted, the former representing task execution times while the latter represent communication times or amounts of data being exchanged.

In this work we will retain a parallel program execution model based on an acyclic directed graph whose nodes represent the tasks and whose arcs represent the relations between tasks. The optimization of the parallel execution speed-up can be formulated in terms of a partitioning problem of the task graph, where each partition represents the tasks which have been assigned to a particular processor. The cost function used for partitioning characterizes the speed-up of the parallel program's execution on a set of identical parallel processors having local memory. An appropriate measure of speed-up should consider both computing times and communication times. Thus the problem we address is that of establishing a partition of the task graph representation of a parallel program, with certain objective functions to be optimized.

Many graphs partitioning problems are NP-hard, and in the present research we will consider a variety of heuristics which yield effective and nearly optimal solutions in for the cases of interest to this study. Clearly, we study approximate solution methods because exact solutions have excessive execution times when the program size is large. The approximate methods we discuss give suboptimal solutions in a reasonable (polynomial) execution time. In this research we consider several such heuristics: one of the heuristic will be based on the random neural model of Gelenbe [3, 4], one will use genetic algorithms [1, 5] and finally one will be based on the simulated annealing optimization heuristic [6, 9].

2. Problem Definition

In order to fix the framework of the work to be undertaken let us recall some of the basic considerations. We consider a distributed system architecture which consisting of a collection of K homogeneous processors with distributed memory, i.e., with sufficient memory at each processor. The processors are fully interconnected connected with a reliable high-speed network.

A task graph is denoted by: $\Pi = (N, A, \Omega, c)$, where $N = \{1, \ldots, n\}$ is the set of n tasks that compose the program, and Ω, c denote the times related to task execution and to communication between tasks. Thus each task i that has a weight $\Omega(i)$ which defines its execution time, $i=1, \ldots, n$. c_{ij} will denote the communication time needed to inform task j that task i has terminated its execution. $A = \{a_{ij}\}$ is the adjacency matrix representing the precedence order between the tasks. Since the graph is acyclic, we may number the tasks in a manner such that $a_{ij}=0$ if $i > j$ [7, 8].

The problem is that of assigning the n tasks to K processors. This means that we have to create a partition $(\Pi_1, \ldots, \Pi_K)$ of the set of n tasks in a way which optimizes performance. The problem is then characterized by the following objective: the total effective execution time of the parallel program must be minimized.

2.1. The Formal Problem

The sequential execution time of the program is $T_S = \Sigma^n{}_{i=1}\,\Omega(i)$, assuming that all tasks reside on the same processor and exchange information through common memory. The parallel execution time T_P, assuming an unlimited number of processors, is the largest sum of task execution times and communication times on a path in the graph. This can be written using the instant at which task i will terminate, which is

$$t_i = \Omega(i) \text{ if task i has no predecessors in the graph,}$$

$$t_i = \Omega(i) + \max_{j<i} \{ t_j + c_{ji} \} \text{ if } i \text{ has predecessors in the graph.}$$

Then it can be seen that $T_P = \max_{1 \leq i \leq n} \{ t_i \}$. Indeed, this is the time it will take if each task is executed on a different processor. Now, we assume that the tasks have been assigned to processors by the partition $\Pi = (\Pi_1, \ldots, \Pi_K)$. Clearly, once the program is partitioned and placed on K processors, the actual execution time will depend in a complex manner on the schedule that is used, since all tasks which have been assigned to the same processor will execute sequentially, and some tasks on different processors will also have to execute sequentially due to the precedence relations. The exact computation of this execution time is not trivial. However an optimistic estimate can be obtained. It is useful to notice that if the execution schedule is not organized correctly, the task assignment can lead to deadlocks. However, a deadlock-free schedule can be obtained for an acyclic task graph if the processor is always allocated to a task which has no non-executed predecessors. Thus in a distributed system, such decisions have to be taken with global information and not locally, since a task's predecessors will not necessarily reside on the same processor.

Assuming that tasks residing on the same processor communicate in zero time, the new communication times can be readily computed. It suffices to remove from communication times which deal with two tasks which are in the same block Π_u of the partition. More specifically, the new intertask communication times will become

$$C(\Pi)_{ij} = c_{ij} \quad \text{if } i \in \Pi_u , j \in \Pi_v \text{ and } u \neq v,$$
$$C(\Pi)_{ij} = 0 \quad \text{if } i \in \Pi_u , j \in \Pi_v \text{ and } u = v.$$

To estimate the best possible execution time of the program with the partition Π we first notice that the time $t(\Pi)i$ when task i terminates now satisfies the following inequalities:

$$\Omega(i) \leq t(\Pi)i \leq \Sigma_{j\in \Pi_u} \Omega(j), \quad \text{if } i \in \Pi_u \text{ and has no predecessors outside } \Pi_u,$$

$$\Omega(i) + \max_{j<i} \{ t(\Pi)j + C(\Pi)_{ji} , j \text{ not in } \Pi_u \}$$
$$\leq t(\Pi)i \leq \Sigma_{j\in \Pi_u}\Omega(j) + \max_{j<i} \{ t(\Pi)j + C(\Pi)_{ji} , j \text{ not in } \Pi_u\},$$
$$\text{if } i \in \Pi_u \text{ and has predecessors outside } \Pi_u .$$

Clearly, when it is assigned to the *K* processors according to the partition Π, the program as a whole will terminate at some instant

$$T_p(\Pi) = \max_{1\leq i\leq n} \{ t(\Pi)i \}$$

Ideally, we would like choose the assignment Π which will minimize $T_p(\Pi)$. However this appears to be a very hard problem except under very simple assumptions. Notice also that though the assignment issue is very important, performance is also influenced by the order in which tasks are executed, since it is important to always choose to execute those tasks which will then enable the execution tasks at other processors [2].

3. The Solution Methods

From the above discussion, it appears reasonable to formulate some related optimization problems which are easier to address and whose solution would contribute to the optimum task assignment problem.

3.1 The Random Neural Model

The neural networks have been used over the last several years to obtain heuristic solutions to hard optimization problems. We will propose an approach using the random neural network model which has the property of being mathematically tractable and computationally efficient.

The random neural network model has been developed by Gelenbe [3, 4] to represent a dynamic behavior inspired by natural neural systems. This model has a remarkable property called "product form" which allows the computation of joint probability distributions of the neurons of the network. The basic descriptor of a neuron random network [3, 4] is the i-th neuron's probability of being excited $q(i)$, $i=1, \dots, n$, which satisfy a set of non-linear equations:

$$q(i) = \{ \Sigma^{n}_{j=1} q(j)r(j)p^{+}(j,i) + L(i) \} / \{ \Sigma^{n}_{j=1} q(j)r(j)p^{-}(j,i) + l(i) \} \qquad (1)$$

Where:

- $L(i)$ is the rate at which *external excitation signals* arrive to the i-th neuron,
- $l(i)$ is the rate at which *external inhibition signals* arrive to the i-th neuron,
- $r(i)$ is the rate at which neuron i fires when it is excited,
- $p^{+}(i,j)$ and $p^{-}(i,j)$, respectively, are the probabilities that neuron i (when is excited) will send an *excitation* or an *inhibition signal* to neuron j.

We have $\Sigma^{n}_{j=1}\ p^{+}(i,j) + p^{-}(i,j) \leq 1$, for $1 \leq i \leq n$. Notice that the model is based on rates, much as natural neural systems operate. Thus, this is a "frequency modulated" model, which translates rates of signal emission into excitation probabilities via equation (1). For instance $q(j)r(j)p^{+}(j,i)$ denotes the rate at which neuron j excites neuron i. Equation (1) can also be translated into a special form of sigmoid which treats excitation (in the numerator) asymetrically with respect to inhibition (in the denominator).

The Random Neural Model for our Problem.

In this approach we will construct a random neural network of the type discussed above composed of nK + K neurons [9], where n is the number of tasks and K is the number of processors.

For each (task, processor) pair (i,u) we will have a neuron m(i,u) whose role is to "decide" whether task i should be assigned to processor u. We will denote by q(m(i,u)) the probability that m(i,u) is excited: thus if it is close to 1 we will be encouraged to assign i to u. In order to reduce communication times in the selected partition, will tend to *excite* any neuron m(j,u) if j is a successor or predecessor of i, and will tend to *inhibit* m(j,v) if j is successor or predecessor of i and $u \neq v$. Similarly, m(i,u) will *inhibit* m(j,u), $\forall_{v=1, \dots, K}$, if j is not a predecessor or successor of i. On the other hand, neurons m(i,u) and m(i,v), $u \neq v$, will *inhibit* each other so as to indicate that the same task should not be assigned to different processors.

For each processor u we will have a neuron $\pi(u)$ whose role is to let us know whether u is heavily loaded with work or not. If u is very heavily loaded, it will attempt to reduce the load on processor u by *inhibiting* neurons m(i,u), and it will attempt to increase the load on processors $v \neq u$ by *exciting* neurons $\pi(v)$. In the same way, $m(i,u)$ will *excite* neuron $\pi(u)$ to increase the load on processor u. The parameters of the random network model expressing these intuitive criteria are chosen as follows:

- $L(m(i,u)) = 0$
- $L(\pi(u)) = n/K$, to express the desirable equal load sharing property,
- $l(m(i,u)) = 0$,
- $l(\pi(u)) = 0$,
- $r(m(i,u)) = nK$
- $r(\pi(u)) = n+K-1$
- $r(m(i,u))p^+(m(i,u),m(j,v))$ = 1 if $(a_{ij} = 1$ or $a_{ji} = 1)$ and $u=v$, 0 otherwise.
- $r(m(i,u))\, p^-(m(i,u),m(j,v))$ = 1 if $u \neq v$ and $(a_{ij}= 1$ or $a_{ji} = 1$ or $i = j)$, or if $a_{ij}=0$ and $a_{ji}=0$, 0 otherwise.
- $r(m(i,u))p^+(m(i,u), \pi(v))$ = 1 if $u=v$, 0 otherwise.
- $r(\pi(u))p^-(\pi(u),m(i,u))$ = 1 if $q(\pi(u)) \sim 1$, 0 otherwise
- $r(\pi(u))p^+(\pi(u),\pi(v))$ = 1 if $q(\pi(u)) \sim 1$, 0 otherwise

The equation (1) for this case is:

$$q(m(i,u)) = \{ \Sigma_{(a_{ij} = 1 \text{ or } a_{ji} =)1}\, q(m(j,u))r(m(j,u))p^+(m(j,u),m(i,u)) \} / \{r(m(i,u)) + \Sigma_{v \neq u}\Sigma_{(a_{ij}=1 \text{ or } a_{ji}=1 \text{ or } i=j)}\, q(m(j,v))r(m(j,v))p^-(m(j,v),m(i,u)) + \Sigma_v \Sigma_{a_{ij}=0\ \&\ a_{ji}=0}\, q(m(j,v)r(m(j,v))p^-(m(j,v),m(i,u)) + q(\pi(u))r(\pi(u))p^-(\pi(u),m(i,u))\}$$

$$q(\pi(u)) = \{ L(\pi(u)) + \Sigma^n{}_{j=1}\, q(m(j,u))r(m(j,u))p^+(m(j,u),\pi(u)) + \Sigma^K{}_{v=1}\, q(\pi(v))r(\pi(v))p^+(\pi(v),\pi(u)) \} / r\,(\pi(u))$$

3.2. Simulated Annealing

Simulated Annealing (SA) is a well known method [6] which uses the physical concepts of "temperature and energy" to represent and solve optimization problems using a Montecarlo simulation. The objective function of the optimization problem is treated as the "energy" of a dynamical system, while temperature is introduced to randomize the search for a solution. The state of the dynamical system being simulated is related to the state of the system being optimized. The idea of this method is to start with an initial solution, and then try to improve the solution through local changes. It is based on static mechanics, inspired in the analogy of a physical system behavior in the presence of a hot bath. The procedure is the following [6]: the system is submitted to high temperature and it is slowly cooled through a temperature level series. For each temperature level, we search for the system equilibrium state through an elementary transformation series, which will be accepted if they reduce the system energy $E_{new}<E_{old}$. As the temperature decreases, smaller energy increments are accepted, and the system eventually settles on a low energy configuration that is very close, if not identical, to the global minimum.

For this method, we have studied several parameters in [6, 9]: the initial temperature, the cooling rate and the threshold (Fac_accep) which define the probability that an uphill move of size Δ will be accepted. For the initial temperature, if it is very hot we have CPU time useless; or it is very cold we obtain bad results. In [6, 9] we have made a complete study and we have arrived to follow conclusions: The initial temperature value influences the execution time, which is larger for small initial temperature value. For graph of large size ($\geq$ 35 nodes), it is necessary to take a temperature $\approx$ 90° C to obtain good results.

The cooling rate defines the procedure to reduce the temperature: a reduction very rapid implies a bad local optimum. A reduction very slow is spendthrift in CPU time. Normally, we use a lineal reduction. Good results have been obtained when the reduction factor of the temperature is 0.93 between two steps. For low temperature values we consider that the system has reached a state near the minimum energy (ground state) which corresponds to an optimal solution, consequently we decrease slowly the temperature (0.965).

There are several functions to determine the probability of acceptance, normally named "heat bath" functions. We use $\exp(-\Delta /T)$, because there are mathematical motivations for using the exponential [9]. Other appealing possibility is the function $1-\Delta T$, which involves just one division and at least approximates the exponential. We study the value of probability (Fac_accep) to accept a movement. For this parameter, if the graph is large we obtain the best results for 0.9. For small graph is not important this parameter. This factor has a relation with the execution time. For small values the execution times are generally better because the system arrives more quickly to the equilibrium on each temperature level, but the results are bad.

3.3 Genetic Algorithm

This is an optimization algorithm based on the principles of evolution in biology. A genetic algorithm (GA) follows an "intelligent evolution" process for individuals based on the utilization of evolution operators such as mutation, inversion, selection and crossover [1, 5]. The idea is to find the best local optimum, starting from a set of initial solutions, by applying the evolution operators to successive solutions so as to generate a new and better local minimum. The procedure evolves until it remains trapped in a local minimum.

The GA applied in our problem follows the following next procedure [6, 9]: we define a space of research of n vectors where everyone represents an individual, and every individual represents a possible solution. Each vector has n elements and every element has a value among *1...K*, according to the group to which it belongs. Furthermore, we use the cost function defined on the first part to determine the cost of every individual. We begin with an initial population of individuals randomly defined and we choose the individuals with minimal cost for generating new individuals using the genetic operators. Since the population is constant, we substitute the worst individuals of initial solution by the best individuals generated. The procedure stops if we exceed a given number of generations without finding a better solution.

In this method two parameters are studied: the maximum number of generations (NUMGEN) and the probability (PM) of use the mutation operator after the crossover operator. The first parameter allows to optimize the speed-up of the algorithm to reach an optimal solution. We remark than the quality of the solutions improves more rapidly in the first generations that in the following. Thus, a satisfactory quality can be obtained rapidly without to wait that the algorithm converges. We define the maximum number of generations (NUMGEN) necessary to arrive to good results. If the graph is large, there is a relation between the generation number and the problem size. In this case, a large generation number will be necessary to have good results, which implies more execution time.

In this work, we used the crossover operator and then the mutation operator according to the PM probability. For the PM parameter, if the probability is large we obtain good results, specially for large graph. In this case, the crossover operator is inefficient because it is going to reproduce almost all time the same individual. A large PM implies an execution time large. In [11], Talbi and others define the probability to use each operator, which allow a dynamic variation of use of them in the population. Next works will study this approach.

4. Performance Evaluation

We have used the parameters that give the better performance in every method, according to the results of the work [6, 9]. We have used a SUN SPARCstation IPC with 16M of memory and a matrix as data structure. The random graphs used are defined for the average number of nodes (*n*) and the average degree of the successor nodes of a node (*d*). For each graph, the successors of a node are chosen randomly from a uniform distribution in the interval [1, d]. The execution time is in seconds. The parameters of the simulations are the following: the total number of subgraphs (K), the mean number of nodes per graph (n), the mean number of successors per node (d) and the balance factor (b). We generate 50 random graphs for the set of parameters where $n = \{10, 20, 50\}$, $K=\{2, 5\}$ and $d=2$.

We study the following performance criteria: the execution time of the heuristics and the mean value of the solutions. Due to space limitations, the results presented in this section were chosen because they are representative of the phenomena studied.

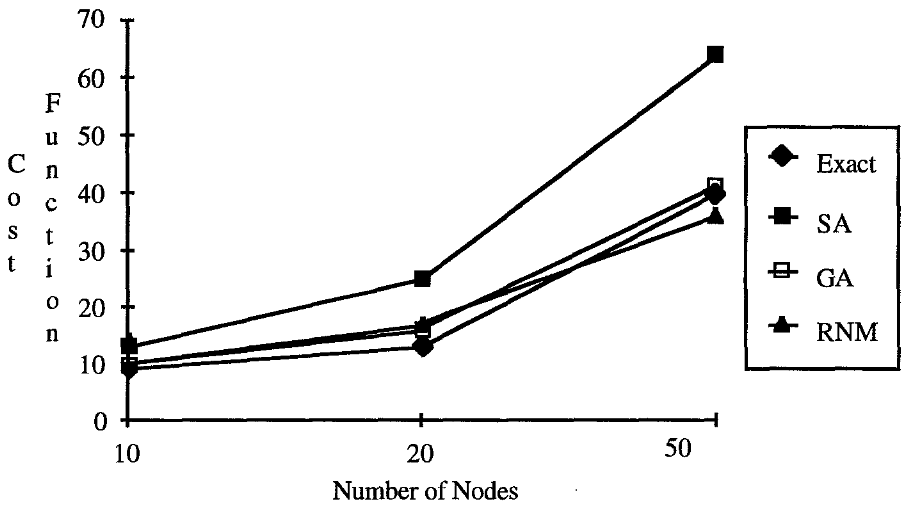

Fig. 1. Results of the simulation for b = 1, K = 2 and d = 2

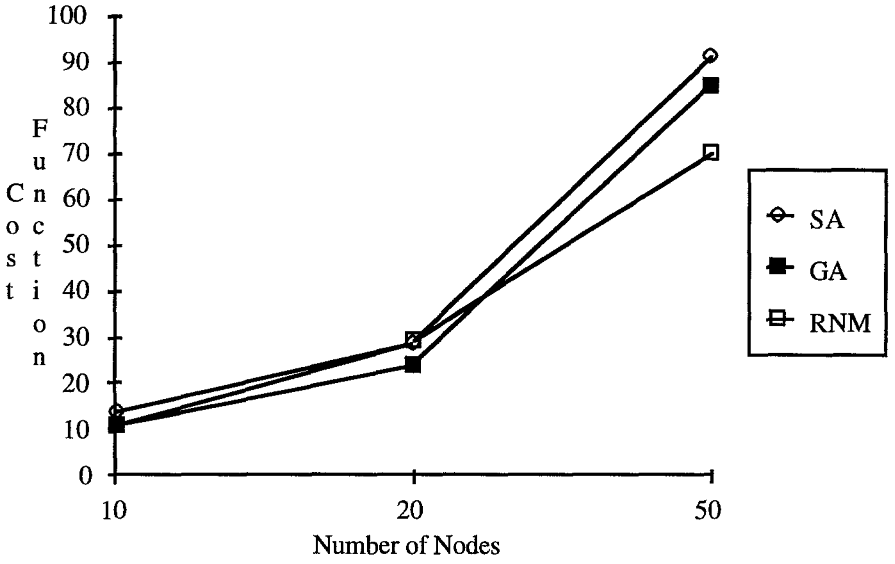

Fig. 2. Results of the simulation for b = 1, K = 5 and d = 2

The execution times are very large (Figures 1, 2, 3). The genetic algorithm and the simulated annealing, for graphs of *50* or more nodes, need a very large time to reach the suboptimal solutions. For graph of little size (of *20* or less nodes), the difference between the exact solution and the results of the other methods is not important, but the execution times are similar, what makes more interesting the exact solutions.

Otherwise, the approximate methods are more interesting, because they have a reasonable execution time.

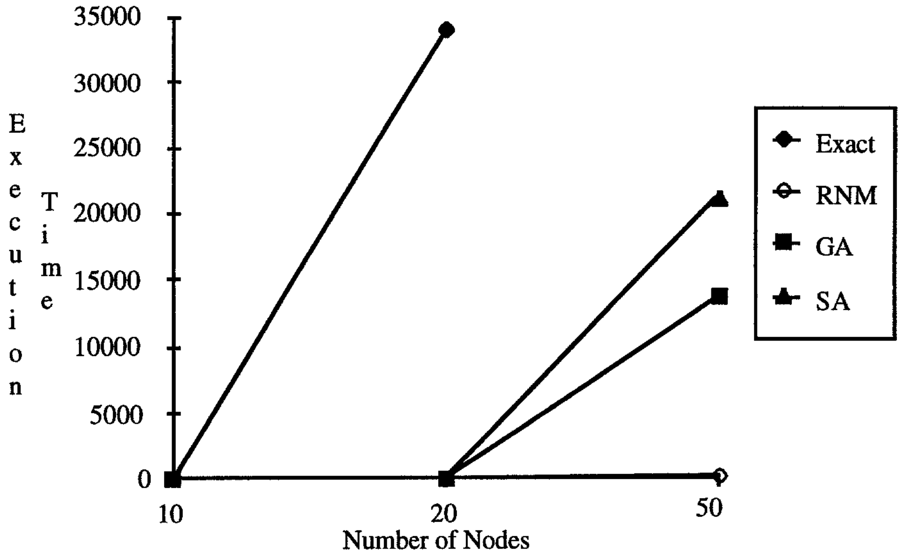

Fig. 3. Execution time of the simulation for b = 1, K = 2 and d = 2

5. Conclusions

The experiments we have run show that the results obtained by each approximate method vary widely depending on the size of the graphs considered. In our study, the Genetic Algorithm appears to give the best results, but with a substantially larger execution time. The Random Neural Model gives good results with short execution time.

The execution time for the Genetic Algorithm and Simulated Annealing are very large. For Genetic Algorithm, the reason is that generation calculations take relatively much more time. It is necessary to determine the better combination of genetics operators, to decrease the number of necessary generations to reach the suboptimal solution. For Simulated Annealing, since it is not possible determine coherent movements of nodes in every temperature level that decrease the energy, the solution is evaluated in a relatively longer time. The Genetic Algorithm and the Random Neural Model are easy to implement on a parallel machine, and this can considerably improve the speed obtained with these methods.

Future work will examine other combinatorial optimization methods for the solution of design problems in distributed systems (tasks migration, files allocation, ...), and will consider a combination of the Random Neural Model and Genetic Algorithms.

References

1. GOLDBERG, D. "Genetic algorithms in search, optimization and machine learning", Addison-Wesley, 1989.
2. GELENBE, E. "Multiprocessor Performance". J. Wiley & Sons. 1989.
3. GELENBE, E, "Random neural networks with positive and negative signals and product form solution", Neural Computation Vol. 1, No. 4, pp 502-511, 1989.
4. GELENBE, E, "Stable random neural networks", Neural Computation , Vol. 2, No. 2 pp 239-247, 1990.
5. TALBI, E. and BESSIERE, P. "Un algorithme génétique massivement parallèle pour le problème de partitionement de graphes". Technical Report. Laboratoire de Génie Informatique. Grenoble-France. 1991.
6. AGUILAR, J. "Combinatorial Optimization Methods. A study of graph partitioning problem". Proceedings of the Panamerican Workshop on Applied and Computational Mathematics, PWACM, Caracas, Venezuela, 1993.
7. AGUILAR, J. "Modelling the explicit and implicit parallelism of a parallel program", Proc. XIV Intl Conf. of the Chilean Computer Science Society, 1994.
8. AGUILAR, J. "Heuristic algorithms for task assignment of parallel programs", Proc. Intl. Conf. Massively Parallel Processing Applications and Development, Delft, Holland, 1994.
9. AGUILAR, J. "L'allocation de tâches, l'équilibrage de la charge et l'optimisation combinatoire", PhD thesis. Rene Descartes University, Paris, France, 1995.