Núcleo de Computação Eletrônica

Scheduling Multiprocessor Tasks with Genetic Algorithms

Ricardo C. Corrêa
Afonso Ferreira
Pascal Rebreyend

NCE - 02/96
outubro

Relatório Técnico

Universidade Federal do Rio de Janeiro

Scheduling Multiprocessor Tasks with Genetic Algorithms *

Ricardo C. Corrêa†
NCE-UFRJ
Caixa Postal 2324
CEP 20001-970 RJ
Brazil
correa@nce.ufrj.br

Afonso Ferreira‡
LIP - ENS-Lyon
CNRS URA 1398
69364 Lyon Cedex 07
France
ferreira@lip.ens-lyon.fr

Pascal Rebreyend
LIP - ENS-Lyon
CNRS URA 1398
69364 Lyon Cedex 07
France
prebreye@lip.ens-lyon.fr

Abstract

In the *multiprocessor scheduling problem* a given program is to be scheduled in a given multiprocessor system such that the program's execution time is minimized. This problem being very hard to solve exactly, many heuristic methods for finding a suboptimal schedule exist. We propose a new combined approach, where a genetic algorithm is improved with the introduction of some knowledge about the scheduling problem represented by the use of a list heuristic in the *crossover* and *mutation* genetic operations. This knowledge-augmented genetic approach is empirically compared with a "pure" genetic algorithm and with a "pure" list heuristic, both from the literature. Results of the experiments carried out with synthetic instances of the scheduling problem show that our knowledge-augmented algorithm produces much better results in terms of quality of solutions, although being slower in terms of execution time.

Keywords: Multiprocessors, scheduling problems, list heuristics for scheduling problems, genetic algorithms, NP-hard, optimization.

*This work was partially supported by the HCM project *SCOOP - Solving Combinatorial Optimization Problems in Parallel* - of the European Union. A preliminary short version of this paper appeared in the IEEE Symposium on Parallel and Distributed Processing, New Orleans, October 1996.

†Partially supported by a CNPq fellowship and by the PROTEM-CC-II project ProMet of the CNPq (Brazil). He was with LMC - IMAG, BP 53, 38041 Grenoble Cedex 9, France.

‡Corresponding author. Part of this work was done while with the School of Computer Science, Carleton University, Ottawa K1S 5B6, Canada. Partially supported by a Region Rhône-Alpes fellowship and NSERC.

1 Introduction

Let a *(homogeneous) multiprocessor system* be a set of m identical processors, $m > 1$. Each processor has its own memory, and each pair of processors communicate exclusively by message passing through an interconnection network. Additionaly, let a *parallel program* be a set of communicating tasks to be executed under a number of precedence constraints. To each task is associated a cost, representing its execution time. A weighted acyclic *task digraph* can be used to represent the tasks (vertices of the task digraph) and the precedence constraints (arcs of the task digraph). In order to be executed, each task of a *given* parallel program must be *scheduled* to some processor of a *given* multiprocessor system. Consequently, tasks that communicate in the parallel program may be scheduled to different processors, which leads these processors to communicate during the execution of the parallel program. In general, these communications slow down the execution of the parallel program. Considering these communications and the precedence constraints between tasks, it follows that different schedules of each task satisfying the precendence constraints lead to different execution times of the parallel program. This is the motivation to the optimization problem informally defined below.

Given a parallel program to be executed on a multiprocessor system, the *(multiprocessor) scheduling problem* consists of finding a task schedule that *minimizes* the execution time of the parallel program and the number of required processors. In this paper, we deal with a slightly easier (although also NP-hard[1]) version of the scheduling problem where the number of processors is fixed. Due to the importance of this optimization problem, it has been extensively studied by a large number of researchers (see for instance [2, 3, 4, 5, 6] and references therein). Since an exhaustive search is often unrrealistic, most of the work has been done on fast heuristic methods to find *suboptimal solutions*, i.e., solutions whose optimality cannot be guaranteed. In other words, the purpose of such heuristic methods is to be able to determine a good solution, even when the instance size leads the exhaustive search to be too long. The most studied heuristic methods for multiprocessor scheduling problems are the so called *list heuristic* [4].

Another heuristic method used in the scheduling problem context is the meta-heuristic known as *genetic algorithms* [7, 8]. A genetic algorithm is a guided random search method where elements (called *individuals*) in a given set of solutions (called *population*) are randomly combined and modified (we call these combinations *crossover* and *mutation*, respectively) until some termination

[1] Since it can be expressed as a quadratic assignment problem (see [1]).

condition is achieved. The population evolves iteratively (in the genetic algorithm terminology, through *generations*) in order to improve the *fitness* of its individuals. The fitness of an individual s_1 is said to be *better* than the fitness of another individual s_2 if the solution corresponding to s_1 is closer to an optimal solution than s_2. In each iteration, the crossovers generate a new population in which the individuals are supposed to keep the good characteristics of the individuals of the previous generation.

In the context of scheduling problems, Hou, Ansari and Ren [9], and Wang and Korfhage [10] proposed pure genetic algorithms whose main difference lays in the way the individuals are coded. Wang and Korhage use a bi-dimensional matrix to code a schedule, while Hou, Ansari and Ren proposed a coding based on strings. In both algorithms, no knowledge about the problem is taken into account, and the search is accomplished entirely at random considering only a subset of the search space.

In this paper, we study the impact of (a) adopting genetic operators such that all feasible solutions are considered in the search, and (b) integrating knowledge – in the form of list heuristics – into a genetic algorithm for multiprocessor scheduling. The idea of integrating knowledge into a genetic algorithm for the MSP has been recently and independently addressed by Ahmad and Dhodhi in [11]. In their algorithm, a chromosome represents, for each task, a priority. Priorities are defined, before the execution, as the longest path from a node to a node which sends no messages. At any given point in the algorithm, a scheduling can be deduced from the priorities by means of a list algorithm. Their genetic algorithm works on the priorities with standard crossover and mutation. The list algorithm is then used only to build a solution from the chromosome-coded priorities.

In our approach we use the coding from [9] and propose two new genetic algorithms which differ from the ones discussed in [9, 11] in that the knowledge is integrated inside the crossover and mutation operators. Notice that this method is more flexible because different operators (i.e., different knowledges) can be used in the different operators.

The first original genetic algorithm we propose, when compared to the one in [9], extends the set of feasible solutions that is considered in the search. Analitically, we were able to demonstrate that a very important drawback of the algorithm from [9] is the fact that it does not take *all* the search space into consideration, while ours does not suffer of this drawback. Our second genetic algorithm keeps this positive feature, and integrate some knowledge about multiprocessor scheduling in the

form of a list heuristics. In this second algorithm, knowledge-augmented crossovers and mutations are proposed.

Instead of testing our genetic algorithm with randomly generated instances, as in [9], [10] and [11], we preferred to use as testbench some relatively large instances provided by ANDES-Synth [12, 13]. ANDES-Synth is a tool that generates synthetic task digraphs whose shapes represent known parallel programs, as divide and conquer, prolog solving, Gauss elimination, etc. In order to be close to reality, the costs of these task digraphs are estimated in function of evaluations carried out over an IBM SP-1 parallel computer. Using this testbench, we compared three algorithms, namely the pure genetic algorithm from [9], the list (greedy) heuristic from [14] and our combined algorithm.

The three genetic algorithms were implemented, and the results of extensive experiments compared to a list heuristic. The tests results show that the solutions found with our genetic algorithms are much better than those of [9]. They yield better quality initial populations and crossover offsprings than pure random choices. From the experimental results, we can conclude that integrating knowledge into genetic algorithms could help improving the quality of solution, and the search space containning necessarily an optimal scheduling is also an improvement on the results in [9]. In fact, the only bad characteristic of our approaches is their longer execution time. The first genetic algorithm we propose gives better solutions than [9], but needs a longer execution time to achieve the final solution. The same is true when comparing our second genetic algorithm with our first one. Finally, the algorithm in [11] was tested on instances of size 30, while we were able to run instances with up to 1482 tasks.

Our paper is organized as follows. The scheduling problem and the principles of genetic algorithms are precisely defined in Section 2. In Section 3, the algorithm from [9] is reviewed and analyzed. An example where it cannot find the optimal solution is presented. Our first improved genetic algorithm is represented in Section 4. The knowledge-augmented genetic algorithm is then introduced in Section 5. Results, comparisons and analyses are shown in Section 6. We close the paper with concluding remarks and ways for further research.

2 Preliminaries

In this section, we define more formally the multiprocessor scheduling problem and the principles of genetic algorithms.

2.1 Multiprocessor scheduling

In order to formalize the multiprocessor scheduling problem, we first define a (homogeneous) multiprocessor system and a parallel program. A *(homogeneous) multiprocessor system* is composed of a set $\mathcal{P} = \{p_1, \cdots, p_m\}$ of m identical processors. They are connected by a complete communication network, where all links are identical. Each processor can execute at most one task at a time and task preemption is not allowed. While computing, a processor can communicate through one or several of its links.

The parallel program is described by an acyclic digraph $\mathcal{D} = (\mathcal{T}, A)$. The vertices represent the set $\mathcal{T} = \{t_1, \ldots, t_n\}$ of tasks and each arc represents the precedence relation between two tasks. An arc $(t_{i_1}, t_{i_2}) \in A$ represents the fact that at the end of its execution, t_{i_1} sends a message whose contents are required by t_{i_2} to start execution. In this case, t_{i_1} is said to be an *immediate predecessor* of t_{i_2}, and t_{i_2} itself is said to be an *immediate successor* of t_{i_1}. We suppose that t_1 is the only task without any immediate predecessor. A *path* is a sequence of nodes $< t_{i_1}, \cdots, t_{i_k} >$, $1 < k \leq n$ such that t_{i_l} is an immediate predecessor of $t_{i_{l+1}}$, $1 \leq l < k$. A task t_{i_1} is a *predecessor* of another task t_{i_k} if there is a path $< t_{i_1}, \cdots, t_{i_k} >$ in $\mathcal{D}$. To every task t_i, there is an associated weight representing its duration, known before the execution of the program. In addition, all the communications are also known at compile-time. Thus, to every arc $(t_{i_1}, t_{i_2}) \in A$ there is an associated weight representing the transfer time of the message sent by t_{i_1} to t_{i_2}. To compute the transfer time we take as cost model $\tau L + \beta$, where β is the cost to initialize the communication, τ is the transfer time for a byte, and L is the length of the message in bytes. If both message source and destination are scheduled to the same processor, then the cost associated to this arc becomes null.

Hence, a *schedule* is a vector $s = \{s_1, \cdots, s_n\}$, where $s_j = \{t_{i_1}, \cdots, t_{i_{n_j}}\}$, i.e., s_j is the set of the n_j tasks scheduled to p_j. For each task $t_{i_l} \in s_j$, l represents its execution rank in p_j under the schedule s. Further, for each task t_i, we denote $p(t_i, s)$ and $r(t_i, s)$, respectively, the processor and the rank in this processor of t_i under the schedule s. The execution time yielded by a schedule is called *makespan*. We consider uniquely the schedules whose computation of the introduction dates for the tasks is done in a special way. They follow a *list heuristic* whose principle is to schedule each task t_i to $p(t_i, s)$ according to its rank $r(t_i, s)$. In addition, the task is scheduled as soon as possible depending on the schedule of its immediate predecessors.

A list heuristic builds a schedule step by step. At each step, the tasks that can be scheduled (called *free tasks*) are those whose all predecessors have already been scheduled. Then, we choose

one of such tasks, say t_i, according to a certain rule R_1. Additionaly, we choose a processor, say p_j, to which t_i will be scheduled according to another rule R_2. We then schedule t_i to p_j as soon as possible. This algorithm finishes when all tasks have been scheduled. At an iteration k of this algorithm, let $O(k)$ be the set of tasks remaining to be scheduled, and $F(k)$ the set of free tasks from $O(k)$. Initially, $O(0) = \mathcal{T}$ and $F(0) = \{t_1\}$. Thus, at an iteration $k > 0$, we choose a task from $F(k)$, we take it out from both $O(k)$ and $F(k)$, and we schedule it to $p(t_i, s)$, as soon as possible. This algorithms finishes when $F(k) = \emptyset$.

We define that a schedule s is *feasible* if and only if the above algorithm that constructs s finishes at iteration $k = n$. This means that all tasks could be scheduled since exactly one task is scheduled at each iteration. It is clear that the schedule obtained is minimal with respect to the makespan. Figure 1 illustrates a schedule and the introduction dates computed by the list heuristic above.

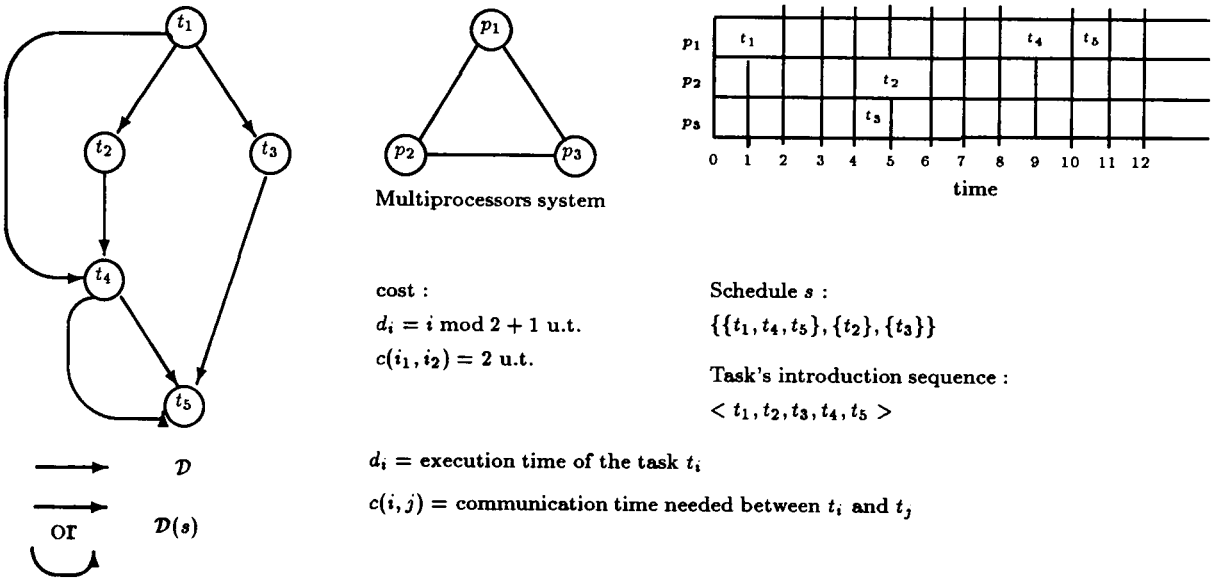

Figure 1: Example of a schedule.

2.2 Genetic algorithms

As we have seen in the Introduction, a genetic algorithm starts with an initial population that evolves through generations. This evolution starts with an initial population randomly generated, and the ability of an individual to span through different generations and to reproduce depends on

its *fitness*. In our case, the fitness of an individual is defined as the difference between its makespan and the one of the individuals having the largest makespan in the population. Notice that the best individual correspond to the one having the smallest makespan and the largest fitness. In what follows, we review the operators that compose a genetic algorithm.

The *selection* operator allows the algorithm to take biased decisions favoring good individuals when changing generations. For this, some of the good individuals are replicated, while some of the bad individuals are removed. As a consequence, after the selection, the population is likely to be "dominated" by good individuals. Starting from a population P_1, this transformation is implemented iteratively by generating a new population P_2 of the same size as P_1, as follows. Initially, the best individual of P_1 is replicated, with only copy kept in P_1 and the other inserted in P_2. Then, at each iteration, we randomly select an individual $s_1 \in P_1$ according to its fitness. Then, s_1 is duplicated into a new individual s'_1, and s_1 is kept in P_1 while s'_1 is inserted into P_2. This process is repeated until P_2 reaches the size of P_1. Notice that, using this scheme, each individual can be selected more than once or not at all. Thus, some individuals are eliminated from generation to generation.

Genetic algorithms are based on the principles that crossing two individuals can result on offsprings that are better than both parents, and that a slight mutation of an individual can also generate a better individual. The *crossover* takes two individuals of a population as input and generates two new individuals, by crossing the parents characteristics. Hence, the offsprings keep some of the characteristics of the parents. The *mutation* randomly transforms an individual that was also randomly chosen. It is important to notice that the size of the different populations are all the same. Therefore, it is desirable that "bad" individuals generated by crossover and mutation operators tend to be eliminated, while "good" individuals tend to survive and to reproduce. Thus, the selection operator eliminates some individuals with poor fitness from generation to generation.

The structure of the algorithm is a loop composed of a selection followed by a sequence of crossovers and a sequence of mutations. Let the population be randomly divided in pairs of individuals. The sequence of crossovers corresponds to the crossover of each of such pairs. After the crossovers, each individual of the new population is mutated with some (low) probability. This probability is fixed at the begining of the execution and is constant. Moreover, the termination condition may be the number of iterations, execution time, results stability, etc.

3 The starting genetic algorithm

As mentioned before, our example of a "pure" genetic algorithm for multiprocessor scheduling is the algorithm from [9], henceforth denoted HAR, for short. In the following we recall its basic ideas.

3.1 Coding of solutions

The coding of an individual s is composed of m strings $\{s_1, s_2, \cdots, s_m\}$. There is a one to one correspondance between processors and strings, where each string represents the tasks scheduled to some specific processor. Each string s_j represents the tasks scheduled to processor p_j in s, and these tasks appear in s_j in the order of their execution in the schedule s. Figure 1 shows an example of a coding for three processors (hence, with three strings). It is easy to see that this encoding scheme using strings may represent schedules not satisfying the precedence constraints. For this reason, a method that guarantees that all strings in the initial population or produced by crossovers or mutations will correspond to feasible schedules was proposed in HAR. This method is based on the concept of *height* of tasks. Let t_i be a task, $hp(t_i)$ be the maximum length of a path between t_1 and an immediate predecessor of t_i, and $hs(t_i)$ be the maximum length of a path between t_1 and an immediate successor of t_i. Each task t_i is then assigned a random *height* whose value is such that $hp(t_i) < height(t_i) < hs(t_i)$. For instance, in Figure 1 we have

$$height(t_1) = 0,$$
$$height(t_2) = 1,$$
$$height(t_3) = 1 \text{ or } 2,$$
$$height(t_4) = 2,$$
$$height(t_5) = 3.$$

The tasks heights induce a partial order on the tasks that helps representing the task dependencies in terms of precedence relations. If a task t_{i_1} is a predecessor of a task t_{i_2}, then $height(t_{i_1}) < height(t_{i_2})$. Finally, in order to guarantee the feasibility of a given schedule coded as above, the tasks are ordered according to their heights in each string.

3.2 Initial population

The initial population is randomly generated, the tasks being scheduled to the processors according to their height as follows. Let $T(h)$ be the set of tasks with height h in $\mathcal{D}$. For each height h,

the following steps are performed. Choose at random r tasks, $0 \leq r \leq |T(h)|$, from $T(h)$ to be assigned to p_1. Then, remove these r tasks from $T(h)$ and assign them to p_1. Repeat this step for all processors $p_2, \cdots, p_{m-1}$. Finally, schedule all remaining tasks from $T(h)$ to p_m.

3.3 Genetic operators

The genetic operators *selection*, *crossover* and *mutation* used in HAR are described in the following.

3.3.1 Selection

Recall the principle of a selection operation discussed in Subsection 2.2. In what follows, we present the "roulette wheel" principle used to randomly select an individual from P_1 in HAR. In its implementation, each individual is assigned an interval, whose length is proportional to its fitness. For instance, task t_i is assigned to the interval $[1, fitness(t_1)]$, task t_2 is assigned to the interval $[fitness(t_1 + 1), fitness(t_1) + fitness(t_2)]$ and so on. A number between 1 and $\sum_{i=1}^{n} fitness(t_i)$ is drawn at random. An individual is then selected if the randomly drawn number belongs to its interval. Thus, the better the fitness of an individual, the better the odds of it being selected.

3.3.2 Crossover

The crossover in HAR consists of cutting each string of each of the two parents in two parts – left and right. This is obtained by simply randomly choosing a height h and separating the tasks whose height is larger than h – right part – from the ones whose height is smaller than h – left part. The left part of each string remains the same, while the right parts of the strings are exchanged. To ensure consistency, a partition V_1, V_2 of the tasks is defined such that the left parts contain only tasks in V_1 and the right parts contain only tasks in V_2. Consistency is ensured since there is no dependency from a task in V_2 to a task in V_1.

3.3.3 Mutation

Mutation of a schedule s is implemented through a very simple protocol. First, a task t_{i_1} is randomly chosen. Then, among all the tasks with the same height as t_{i_1}, another task t_{i_2} is randomly chosen. Finally, the positions of tasks t_{i_1} and t_{i_2} are exchanged in the schedule s, generating a new, mutated schedule.

3.4 Shortfalls

The improvements we suggest on HAR are based on the following observations.

Observation 1 (Initial population) *In the initial population of HAR, a processor p_i has in average more tasks than p_{i+1}, $1 \leq i < m$. This happens because the task distribution over the processors is not* uniform *due to the initial population generation scheme.*

Observation 2 (Crossover) *The method proposed to implement the crossover operation is simple and fast, but suffers of a severe drawback, namely that* some feasible solutions cannot be generated. *As a matter of fact, the search space of HAR may not contain any optimal solution.*

As an example of Observation 2, let us use the program described in Figure 2 below, where tasks 1 to 6 and 8 to 10 have execution time 1, and task 7 has execution time 10. Suppose that we have two processors p_1 and p_2 and that communication times are null. Applying HAR, we have:

$$height(1) = height(2) = height(3) = 0$$
$$height(4) = height(5) = height(6) = 1$$
$$height(7) = height(8) = height(9) = 2$$
$$height(10) = 3.$$

An optimal schedule assigns tasks 1, 4, 7 and 10, in this order, to processor p_1, and the tasks 2, 5, 8, 3, 6 and 9, in this order, are assigned to processor p_2. The makespan of this schedule is 13. Notice, however, that by respecting the constraint over the heights as in HAR, task 3 (of height 0) is necessarily scheduled before the tasks of height 1. In particular, task 3 is scheduled before tasks 4 and 5. In this case, task 7 has to be delayed since either task 4 or 5 will also be delayed. Hence, the makespan can never equal 13 time units, showing therefore that the search space in HAR may not contain, in general, the optimal solution for the scheduling problem under consideration.

Observation 3 (Absence of knowledge) *Finally, we note that the only knowledge about the problem that is taken into account in the algorithm is of a* structural *nature, through the verification of feasibility of the solutions. While working towards the correction of the main observations above, we also tried to integrate into our algorithm the notion of* quality *of individuals.*

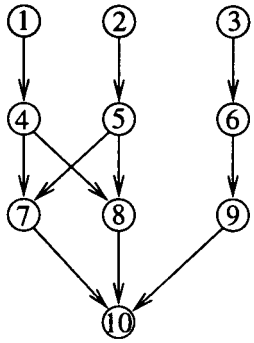

Figure 2: Example where the best schedule does not satisfy the constraints of height.

4 The full search genetic algorithm

The aim of the full search genetic algorithm presented in this section is to overcome the drawback discussed in Observations 1 and 2. Instead of verifying the feasibility of a solution through the tasks height values, we decided to use the task digraph $\mathcal{D} = (\mathcal{T}, A)$ in order to determine whether a transformation is possible. Thus, we need another precedence relation. It stems from the precedences implyed by the tasks scheduled to the same processor in a given schedule, say s, and is defined as follows.

$$\begin{aligned} A(s) = A \cup \{(t_{i_1}, t_{i_2}) \mid\ & (t_{i_1}, t_{i_2}) \notin A, \\ & p(t_{i_1}, s) = p(t_{i_2}, s) \text{ and} \\ & r(t_{i_1}, s) = r(t_{i_2}, s) - 1\}. \end{aligned} \tag{1}$$

We denote $\mathcal{D}(s)$ the digraph $(\mathcal{T}, A(s))$. We also define the relation A^+ as the transitive closure of A, and analogously, $A^+(s)$ as the transitive closure of $A(s)$. From these definitions, we have the following property of feasible schedules.

Proposition 1 *A schedule s is feasible if and only if $\mathcal{D}(s)$ is acyclic.*

Proof. $\Rightarrow$ Let us be given a feasible schedule s. Let (t_{i_1}, t_{i_2}) be any arc in $A(s) - A$. By definition, $p(t_{i_1}, s) = p(t_{i_2}, s)$ and $r(t_{i_1}, s) = r(t_{i_2}, s) - 1$. By contradiction, suppose that there is a cycle in $\mathcal{D}(s)$ including (t_{i_1}, t_{i_2}). Then, $(t_{i_2}, t_{i_1}) \in A^+(s)$. Therefore, since s is feasible, we observe that $r(t_{i_2}, s) < r(t_{i_1}, s)$, a contradiction.

$\Leftarrow$ Consider the list algorithm that constructs s. Suppose, by contradiction, that it finishes in an iteration $k_0 < n$, implying that s is not feasible. Then, there is an iteration $1 \leq k \leq k_0$ of the list algorithm such that $O(k) \neq \emptyset$, but $F(k) = \emptyset$. This implies that every task in $O(k)$ has an immediate predecessor in $O(k)$. Since $\mathcal{D}$ is finite, there is a cycle in $\mathcal{D}(s)$, a contradiction. $\square$

In the following, we describe our full search genetic algorithm based on (1) and Proposition 1. In the rest of the paper, we call this algorithm FSG, for short.

4.1 Coding of solutions

We found out that the coding used in HAR was well suited for the extension of the searched space. Therefore, we also code the individuals of a population as m strings $\{s_1, s_2, \cdots, s_m\}$.

4.2 Initial population

As in HAR, each individual of the initial population is randomly genererated. For this purpose, we use the digraph $\mathcal{D}$ to propose an iterative method to generate each individual of the initial population. This method guarantees that the task distribution among the processors is uniform. At each iteration of this method, the following steps are performed.

I-1. Determine the set F of tasks whose all predecessors according to $\mathcal{D}$ have been already scheduled.

I-2. Randomly choose a task t_j in F.

I-3. Randomly choose a processor p_i.

I-4. Schedule t_j on p_i.

Notice that HAR cannot use this method because the tasks are not necessarily scheduled in the increasing order of their heights.

4.3 Genetic operators

The genetic operators must be revisited in order to take into account the new coding of solutions.

4.3.1 Selection

See Subsection 3.3.1.

4.3.2 Crossover

Let s_1 and s_2 be two individuals which should generate two offsprings. As in HAR, the first step consists of separating the two individuals into two parts. In order to ensure consistency, we need

again to determine a partition V_1, V_2 of the tasks such that there is no dependency from a task in V_2 to a task in V_1. However, since we do not schedule the tasks according to their heights, we use (1) to build a partition such that Proposition 1 is verified, as follows. We first define the digraph $(\mathcal{T}, A(s_1) \cup A(s_2))$ representing the dependencies stemming from the task digraph as well as from the two schedules s_1 and s_2. Then, let $T = \mathcal{T}$. We execute the following steps while $T \neq \emptyset$.

C-1. Choose randomly a task $t_i \in T$ and $V = V_j$, $j = 1$ or 2.

C-2. If $V = V_1$ then

$$\begin{aligned} V_1 \leftarrow \; & V_1 \cup \{t_i\} \cup \\ & \{t_{i'} : \; t_{i'} \in T \text{ and} \\ & \quad (t_{i'}, t_i) \in (A^+(s_1) \cup A^+(s_2))\} \end{aligned} \tag{2}$$

else

$$\begin{aligned} V_2 \leftarrow \; & V_2 \cup \{t_i\} \cup \\ & \{t_{i'} : \; t_{i'} \in T \text{ and} \\ & \quad (t_i, t_{i'}) \in (A^+(s_1) \cup A^+(s_2))\} \end{aligned} \tag{3}$$

C-3. Delete all tasks inserted into V_1 or V_2 from T.

In (2), t_i and all of its predecessors that remain in T are inserted in V_1. Equivalently, t_i and all of its successors that remain in T are inserted in V_2 in (3). It is not difficult to see that V_1 and V_2 correspond to the required partition when $T = \emptyset$.

Finally, the two offsprings are generated from s_1 and s_2 as in HAR. If the tasks in V_1 represent the left part of s_1 and s_2, and the tasks in V_2 their right part, the offsprings are generated by exchanging the right parts.

4.3.3 Mutation

Let s be an individual to which the mutation operator is to be applied. We start by constructing the digraph $\mathcal{D}(s) = (\mathcal{T}, A(s))$. Then, the new individual is generated using the iterative method for the generation of each individual of the initial population (Subsection 4.2), where step I-1 is replaced by the following:

I-1'. Determine the set F of tasks whose all predecessors according to $\mathcal{D}(s)$ have been already scheduled.

5 A combined genetic-list algorithm

The algorithm presented in previous section improves the original HAR in such a way that Observations 1 and 2 does not hold. However, Observation 3 remains valid. In this section, a combined genetic-list algorithm is described. In this combined algorithm, knowledge about the scheduling problem is integrated into the FSG algorithm in the form of new crossover and mutation operators based on a list heuristic.

5.1 Coding of solutions

See Subsection 3.1.

5.2 Initial population

See Subsection 4.2.

5.3 Genetic operators

The knowledge about the scheduling problem is integrated into the genetic operators as described in the following.

5.3.1 Selection

See Subsection 3.3.1.

5.3.2 Knowledge-augmented crossover

Let s_1 and s_2 be two individuals which should generate two offsprings. The first step consists of separating the two individuals into two parts. In other words, we need again to determine a partition V_1, V_2 of the tasks such that there is no dependency from a task in V_2 to a task in V_1 as in Subsection 4.3.2.

Integrating knowledge The problem is not completely solved, since we still need to generate the two offsprings s_1' and s_2' from s_1 and s_2. As in HAR and FSG, we let the scheduling s_1' be the same as s_1 for all tasks in V_1. On the other hand, instead of just exchange the right parts of the strings, we introduce some knowledge in the generation of the offsprings. For this, the remaining

tasks (those in V_2) are scheduled according to a greedy algorithm run over the graph $\mathcal{D}(s_2)$. We use a list heuristic with the following rules.

R_1: compute the minimal introduction date of each free task. This is computed in function of the precedence constraints and in function of the schedule of tasks previously scheduled. Choose the task with smallest introduction date, say t_i. In case of several possibilities, choose the one with more successors. In case of several possibilities, choose at random.

R_2: choose a processor at random among the processors where the task t_i can be scheduled as soon as possible.

The generation of s_2' is analogous, with the tasks in V_2 being scheduled under the constraints in $\mathcal{D}(s_1)$. Feasibility of both s_1 and s_2 is guaranteed by Proposition 1, since both $\mathcal{D}(s_1')$ and $\mathcal{D}(s_2')$ are acyclic. Call this list heuristic *earliest date/most immediate successors first* (ED/MISF).

5.3.3 Knowledge-augmented mutation

The knowledge represented by list heuristics is also integrated into the mutation operator as follows. Let s be an individual to which the operator mutation is to be applied. We start by constructing the digraph $\mathcal{D}(s) = (\mathcal{T}, A(s))$. Then, the new individual is formed by using a list heuristic. The rules used are the same as the crossover, where R_1 is modified such that the minimal introduction dates of the tasks are computed exclusively in function of the precedence constraints. This can be performed just once at the beginning of the operation.

5.4 Discussion

Notice that the list heuristic used in the mutation operator is simpler than the one used in the crossover operator. The difference is the computation of the minimal introduction dates. In the mutation operator, these values are computed just once at the beginning of the mutation operation since they depend uniquely on the extended task graph. However, in the crossover operator, these introduction dates take into account the schedule of the tasks previously scheduled during the execution of the list heuristic. Hence, the computation of these introduction dates must be performed at each iteration of the list heuristic. The reasons for adopting these two rules are the following.

Crossover The crossover operator is supposed to generate "good" individuals. Then, we adopt the rules that, in general, bring the list heuristic to give better schedules. The inconvenient is the time spent in each crossover.

Mutation The main objective of mutation operations is to produce a slight perturbation in the search in order to, eventually, quit local minima. Then, the accuracy of the new individual generated from a mutation is not crucial. For this reason, we adopt simple and fast rules for the mutation operator.

6 Results, comparisons and analyses

In this section, we present the comparison results and analyses of the different approaches. We implemented the four algorithms described before, namely ED/MISF, HAR, FSG, and our combined genetic-list algorithm, denoted CGL, for short. All our results correspond to tests run with our testbench, for scheduling tasks on a 16 processors multiprocessor system. Running times and makespans are given in seconds.

6.1 The testbench

We use as testbench instances provided by a tool called ANDES- Synth [12, 13]. As mentioned earlier, it generates synthetic task digraphs that capture the main features of well known parallel programs. The results reported in this work thus try to mimic reality. The task digraphs were the following.

1. *Bellford:* this digraph represents the algorithm known as Bellman-Ford, which solves the shortest path problem from all nodes to a single destination in a weighted directed graph [15].

2. *Diamond1:* the task digraph in this case is known as a space-time digraph representing a systolic computation [16].

3. *Diamond2:* a systolic matrix multiplication as in [17] is represented by this digraph.

4. *Diamond3:* this is the digraph of fine grained systolic matrix multiplication [18].

5. *Diamond4:* this digraph corresponds to the systolic computation of the transitive closure of a relation on a set of elements [19].

6. *Divconq:* this digraph has the shape of a tree. It represents a divide and conquer algorithm.

7. *FFT:* unidimensional fast Fourier transform.

8. *Gauss:* this digraph describes the execution of a Gaussian elimination used in the resolution of linear systems.

9. *Iterative:* it corresponds to a generic iterative algorithm, each iteration being represented in a same level of the digraph. The immediate successors of a task t_i at level k are tasks at level $k+1$, corresponding to the next iteration.

10. *MS-Gauss:* consider a parallel computation of the kind series-parallel, where the parallel component of the computation is composed of several iterations. The digraph MS-Gauss represents such a computation containing successive resolutions of linear systems by Gaussian elimination.

11. *Prolog:* this graph corresponds to the resolution of a logic program. Its structure is random.

12. *QCD:* represents a gradient method for linear systems [15].

Table 1 shows the size of these graphs, given by the number of tasks, as well as their normalized execution and communication times. The tasks execution times correspond to the number of integer operations executed, and the communication times correspond to the number of integers transferred. In order to obtain the actual execution times and to model an IBM SP-1, we multiply the values in Table 1 by $287\mu s$. The communication of L integers costs $(129 + 2430450L)\mu s$. The actual communication times can be obtained by replacing L by the edges' weights in Table 1. The digraphs were used with two sizes. For this reason, the smaller instance is called -m for middle, while the largest is called -l for large (size).

6.2 Initial population

In Table 2, we present the characteristics of the initial population in HAR, FSG and in CGL. We note that our method often produces more regular and better initial populations.

6.3 Termination condition

All of the implemented genetic algorithms stop in a generation if the improvement on the best solution of the initial population (in percentage points) becomes smaller than the number of gen-

erations. This termination condition is based on the idea that we should let the algorithm run as long as it is improving the solution in a reasonable way. In other words, this termination condition supposes that, when it is violated during the execution of the corresponding genetic algorithm, then this algorithm will not be able to improve the solution significantly. In practical applications, other termination condition can be used.

6.4 Final results

The aim of our experiments is to compare our genetic algorithms with ED/MISF and HAR in terms of the quality of the solution provided and the execution time to find this solution. Table 3 shows these comparative results[2]. In general terms, the results obtained with HAR was able to improve the ones obtained with ED/MISF with relatively small execution times (only for row 3 ED/MISF obtained a better solution). Considering our genetic algorithms, the results confirm that the search in the entire search space and the integration of knowledge allow us to dramatically improve the final solutions, but the elapsed time can be much higher.

Comparing the results obtained with FSG with those of HAR, we notice that the crossover operator in HAR, which reduces the space of solutions actually considered, is indeed a drawback in terms of the quality of the final solution. This is observed in the initial population (Table 2) and in the final solution. Considering the entire search space, FSG performs better than HAR due to two reasons. First, FSG randomly generates better solutions in the initial population. Second, FSG considers, during the search, some solutions not considered by HAR. Consequently, FSG is able to reach better solutions than HAR in its random search.

Table 3 also shows the comparison between the results obtained with our algorithm, where knowledge about the problem was implemented through knowledge-augmented crossovers and mutations, and those obtained with ED/MISF and HAR. The results of our combined genetic algorithm are better than the ones of ED/MISF. Indeed, our combined algorithm performs well even for instances where ED/MISF performs poorly. This may be explained by two facts. First, the genetic approach allows us to find good beginnings of solutions, based on its random nature, while the list approach is able to complete those partial solutions with efficient assignments. Using the Schema Theorem terminology [7, 8], the number schemata whose solution has the makespan below the average increases exponentially. In the specific case of CGL, the crossover and mutation operators

[2]The greedy ED/MISF algorithm was run 1000 times with each instance, and only the best result was kept.

lead two kinds of schemata to dominate, namely:

1. beginnings of below-average solutions; and

2. consider two tasks t_{i_1} and t_{i_2} such that $(t_{i_1}, t_{i_2}) \notin A^+$ and $(t_{i_2}, t_{i_1}) \notin A^+$. The schemata in this case are those defining a precedence relation between t_{i_1} and t_{i_2} corresponding to below-average solutions in which t_{i_1} and t_{i_2} are scheduled to the same processor.

Notice that the integration of knowledge also allows us to dramatically improve the solutions in HAR. Unfortunately, however, the elapsed time of our algorithm can be much higher in both cases.

An interesting point to observe is how HAR performs when run for as long as CGL is run. In order to give an insight into this issue, we also run HAR with a more permissive termination condition. This 'persevering' HAR algorithm stops when the number of generations is equal to 2 or 4 times the number of generations of HAR. We note that, in spite of execution times significantly larger than those of HAR, the improvements on the final solution are modest (see Table 4). In fact, persevering HAR spent an additional long time getting only slight improvements. This suggests that the inability of HAR to generate all feasible solutions is a severe drawback which prevents HAR to significantly improve the results in Table 3.

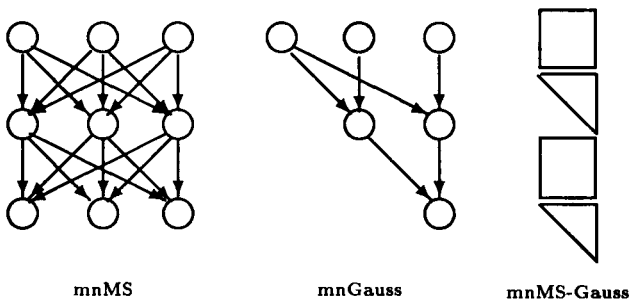

Figure 3: The shape of MS, Gauss and MS-Gauss.

Another interesting aspect of our results is the comparison between the quality of the final solution obtained with the algorithms we tested and the sequential solution, i.e., the execution time on one processor of the program corresponding to each of the DAG's considered. These comparisons are based on the *speedup*, which is defined as the ratio between the makespan of the sequential and parallel solution. The speedups are shown in Table 5. Recall that the parallel solutions correspond to 16 processors. With the medium size DAG's, the speedup is limited because the parallelism is

restricted by the precedence constraints. On the other hand, speedups close to 16 were obtained with the larger DAG's. Considering the average values in Table 5, we observe that some of the results obtained with CGL are very close to optimal.

It is clear from our experiences that FSG is a compromise between a pure and fast genetic approach which yields poor solutions, or the combined approach that we proposed here, which yields very good solutions, although paying the price of a larger computation time.

7 Conclusion

As mentioned in [9], genetic algorithms are well adapted to multiprocessor scheduling problems. In this paper, we tested three genetic algorithms with synthetic task digraphs, and we obtained solutions whose speedup are very close to linear. Moreover, one of the qualities of genetic algorithms we tested is that it can be easily extended to other instances of the scheduling problem, e.g., where a topology different from the complete graph is given for the interconnection network.

The experimental results presented in this paper demonstrate that the integration of knowledge about the multiprocessor scheduling problem, through the use of a list heuristic in knowledge-augmented crossovers and mutations, helps to dramatically improve the quality of the solutions that can be obtained with both a pure genetic and a pure list approaches. Unfortunately, the price to pay is the running time of the combined algorithm, which can be much larger than when running the pure genetic algorithm.

One should notice, however, that the running times (in the order of some hours for difficult graphs) are still reasonable, given that the problem is NP-hard and the instances are big. On the other hand, the running time is larger then pure genetic and list approaches, but smaller then the combined approach. As a perspective, we shall try to design a parallel version of our combined algorithm in order to lower its elapsed time.

Acknowledgement

We are very greatful to João Paulo Kitajima for his substantial help with ANDES-Synth.

References

[1] P. Pardalos and H. Wolkowicz, Eds., *Quadratic assignments and related problems*, vol. 16

of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, 1994.

[2] T. Casavant and J. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems", *IEEE Transactions on Software Engineering*, vol. 14, no. 2, Feb. 1988.

[3] E. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

[4] M. Cosnard and D. Trystram, *Parallel Algorithms and Architectures*, International Thomson Computer Press, 1995.

[5] B. Malloy, E. Lloyd, and M. Soffa, "Scheduling DAG's for asynchronous multiprocessor execution", *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 5, May 1994.

[6] M. Norman and P. Thanisch, "Models of machines and computations for mapping in multicomputers", *ACM Computer Surveys*, vol. 25, no. 9, pp. 263–302, Sep 1993.

[7] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Mass., 1989.

[8] J. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, Mass., 1992.

[9] E. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling", *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 2, pp. 113–120, Feb. 1994.

[10] P.-C. Wang and W. Korfhage, "Process scheduling using genetic algorithms", in *IEEE Symposium on Parallel and Distributed Processing*, San Antonio, Texas, USA, Oct. 1995, pp. 638–641.

[11] I. Ahmad and M.K. Dhodhi, "Multiprocessor Scheduling in a genetic paradigm", *Parallel Computing*, vol. 22, pp. 395–406, 1996.

[12] J. Kitajima and B. Plateau, "Modelling parallel program behaviour in ALPES", *Information and Software Technology*, vol. 36, no. 7, pp. 457–464, July 1994.

[13] J. Kitajima, C. Tron, and B. Plateau, "ALPES: A tool for the performance evaluation of parallel programs", in *Environments and Tools for Parallel Scientific Computing*, J. J. Dongarra and B. Tourancheau, Eds., Amsterdam, The Netherlands, 1993, pp. 213–228, North-Holland, Series: Advances in Parallel Computing vol. 6.

[14] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing", *IEEE Transactions on Computers*, vol. C-33, no. 11, pp. 1023–1029, Nov. 1984.

[15] D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall International Editions, 1989.

[16] O. Ibarra and S. Sohn, "On mapping systolic algorithms onto the hypercube", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 48–63, Jan. 1990.

[17] P.-Z. Lee and Z. Kedem, "Mapping nested loop algorithms into multidimensional systolic arrays", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 64–76, Jan. 1990.

[18] C.-T. King, W.-H. Chou, and L. Ni, "Pipelined data-parallel algorithms: part ii – design", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 4, pp. 486–499, Oct. 1990.

[19] C. Scheiman and P. Cappello, "A processor-time-minimal systolic array for transitive closure", *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 3, pp. 257–269, May 1992.

Digraph		number of tasks	length of a task t_i	weight of the arc (t_{i_1}, t_{i_2})
1	-m	365	10000	400
	-l	992	10000	400
2	-m	258	10000	400
	-l	1026	10000	400
3	-m	486	10000	400
	-l	1227	10000	400
4	-m	731	10000	400
	-l	1002	10000	400
5	-m	731	10000	400
	-l	1002	10000	400
6	-m	382	10000	5120, if $i_1 = 1$
	-l	766	10000	$\frac{c(t_{i_0}, t_{i_1})}{2}$, if $i_1 > 1$ and $t_{i_0} = pred(t_{i_1})$
7	-m	194	5000	800
	-l	1026	5000	800
8	-m	782	10000	4000
	-l	1227	10000	4000
9	-m	262	2500	25600, if $i_1 = 0$
	-l	938	2500	100, if $i_1 \neq 0$
10	-m	768	10000, if Gauss 76800, if MS	4000, if Gauss 76800, if MS
	-l	1482	10000, if Gauss 148200, if MS	4000, if Gauss 148200, if MS
11	-m	214	10000	4000
	-l	1313	10000	4000
12	-m	326	10000	4000
	-l	1026	10000	4000

Table 1: Characteristics of test graphs.

Digraph		HAR				FSG and CGL			
		best schedule	worst schedule	average	standard deviation	best schedule	worst schedule	average	standard deviation
1	-m	438	799	609	88	170	234	201	15
	-l	1517	2114	1797	158	410	501	449	20
2	-m	439	534	480	22	272	373	319	28
	-l	1604	1936	1806	75	842	981	913	35
3	-m	593	672	633	20	591	852	711	50
	-l	1228	1429	1351	44	1210	1588	1397	110
4	-m	1028	1546	1301	114	458	554	510	23
	-l	1461	2109	1729	150	571	702	627	37
5	-m	588	770	670	37	391	487	433	21
	-l	783	1036	913	59	471	586	546	28
6	-m	497	765	651	78	224	286	251	16
	-l	943	1682	1300	187	365	448	396	22
7	-m	147	204	176	14	71	103	89	8
	-l	653	1083	913	98	221	278	250	12
8	-m	1182	1514	1348	74	504	593	550	20
	-l	1822	2146	2067	96	701	825	765	34
9	-m	89	140	120	12	81	109	95	7
	-l	369	466	419	25	237	285	261	12
10	-m	15946	17960	17149	537	8098	9051	8583	218
	-l	15549	17216	16350	458	6488	7127	6846	166
11	-m	241	526	374	70	124	182	145	15
	-l	1636	3212	2187	382	415	522	480	23
12	-m	5409	6870	6222	365	2299	2595	2441	76
	-l	17519	20078	18679	654	5709	6355	6015	163

Table 2: Characteristics of initial population for 26 individuals in HAR, FSG and CGL.

Digraph		Sequential solution	ED/MISF parallel solution	HAR		FSG		CGL	
				Parallel solution	Execution time	Parallel solution	Execution time	Parallel solution	Execution time
1	-m	1008	988	332	24	102	852	69	3766
	-l	2848	2810	1168	111	243	8879	181	21935
2	-m	736	360	326	11	191	228	116	849
	-l	2944	1681	1511	73	561	5571	274	18505
3	-m	704	333	525	20	408	925	83	12427
	-l	1760	970	1148	123	865	6794	152	87848
4	-m	2096	1452	933	31	280	3712	166	9230
	-l	2880	2038	1251	64	338	8315	223	15068
5	-m	1056	871	548	41	244	3344	110	13338
	-l	1440	1158	722	85	294	6780	110	33191
6	-m	1104	551	337	20	147	792	95	1871
	-l	2192	1102	864	19	248	3110	178	4802
7	-m	272	209	115	7	40	265	25	941
	-l	1472	1290	488	64	150	7784	98	12644
8	-m	2240	1754	967	43	345	3427	181	11182
	-l	3520	2882	1618	96	507	7953	261	26482
9	-m	192	181	76	6	46	523	21	1644
	-l	672	654	277	37	165	4687	66	13126
10	-m	26864	23776	14380	60	4761	2952	8064	476
	-l	26592	22852	13700	320	3615	17994	1999	42638
11	-m	608	422	215	5	78	286	61	632
	-l	3776	2566	1443	212	326	8737	256	11117
12	-m	9360	8323	4591	22	1257	618	1099	1821
	-l	29440	27638	15854	168	3151	8021	1903	36180

Table 3: Absolute values of sequential execution time and of final solution obtained with ED/MISF, HAR, FSG and CGL algorithms. The final solution of ED/MISF corresponds to the best one out of 1000 runs.

Digraph		HAR		Persevering HAR			
				2 times		4 times	
		Parallel solution	Execution time	Parallel solution	Execution time	Parallel solution	Execution time
1	-m	332	24	323	46	318	90
	-l	1168	111	1154	193	1129	355
2	-m	326	11	308	22	303	43
	-l	1511	73	1470	90	1443	123
3	-m	525	20	510	34	486	60
	-l	1148	123	1127	150	1089	203
4	-m	933	31	907	48	895	81
	-l	1251	64	1197	98	1166	157
5	-m	548	41	521	57	501	85
	-l	722	85	699	108	676	152
6	-m	337	20	326	43	309	86
	-l	864	19	844	39	807	71
7	-m	115	7	112	15	109	29
	-l	488	64	484	118	477	220
8	-m	967	43	917	70	882	121
	-l	1618	96	1572	125	1494	181
9	-m	76	6	74	12	71	24
	-l	277	37	269	75	261	143
10	-m	14380	52	14170	80	13864	117
	-l	13700	304	13493	364	13232	445
11	-m	215	5	215	11	198	21
	-l	1522	161	1433	420	1432	795
12	-m	4591	22	4591	39	4591	73
	-l	15854	168	15794	238	15794	370

Table 4: Absolute values of final solutions and execution times obtained with HAR and persevering HAR algorithms.

Digraph		ED/MISF	HAR	FSG	CGL
1	-m	1.02	3.04	9.88	14.61
	-l	1.01	2.44	11.72	15.73
2	-m	2.04	2.26	3.85	6.34
	-l	1.75	1.95	5.25	10.74
3	-m	2.11	1.34	1.72	8.48
	-l	1.81	1.53	2.03	11.58
4	-m	1.44	2.25	7.49	12.63
	-l	1.41	2.30	8.52	12.91
5	-m	1.21	1.93	4.33	9.60
	-l	1.24	1.99	4.90	13.09
6	-m	2.00	3.28	7.51	11.62
	-l	1.99	2.54	8.84	12.31
7	-m	1.30	2.37	6.80	10.88
	-l	1.14	3.02	9.81	15.02
8	-m	1.28	2.32	6.49	12.38
	-l	1.22	2.18	6.94	13.49
9	-m	1.06	2.53	4.17	9.14
	-l	1.03	2.43	4.07	10.18
10	-m	1.13	1.87	5.64	3.33
	-l	1.16	1.94	7.36	13.3
11	-m	1.44	2.83	7.79	9.97
	-l	1.47	2.62	11.58	14.75
12	-m	1.12	2.04	7.45	8.52
	-l	1.07	1.86	9.34	15.47
average	-m	1.43	2.34	6.09	9.79
	-l	1.36	2.23	7.53	13.21

Table 5: Speedup.