Line search method for solving a non-preemptive strictly periodic scheduling problem

Clément Pira, Christian Artigues

MISTA 2013

Line search method for solving a non-preemptive strictly periodic scheduling problem

Clément Pira · Christian Artigues

Abstract We study a non-preemptive strictly periodic scheduling problem. This problem arises for example in the avionic field where a set of N periodic tasks (measure of a sensor, data presentation, etc.) has to be scheduled on P processors distributed on the plane. In this article, we consider an existing heuristic which is based on the notion of equilibrium. Following a game theory analogy, each task tries successively to optimize its own schedule and therefore to produce the best response, given the other schedules. We present a new method to compute the best response, and a propagation mechanism for non-overlapping constraints, which significantly improves this heuristic. These improvements allow the heuristic to compare favorably with MILP solutions.

1 Periodic scheduling problem and its MILP formulation

1.1 Context of the work

The problem of scheduling a set of N periodic tasks on a set of P processors has a long history. The preemptive case has received more attention, starting with the work of Liu and Layland [11]. Under this hypothesis, good schedulability conditions can be derived, and the problem is efficiently solved by the EDF algorithm (Earliest Deadline First), at least in the uniprocessor context. In the non-preemptive case, schedulability conditions are generally much weaker and the problem becomes NP-complete in the strong sense, even with one processor, as shown by Jeffay *et al* [8]. The latter still consider a loose notion of periodicity in which the tasks have to execute during each time period, but not always at the same relative position. In order to enforce regularity of the schedule, some works have considered the problem of minimizing the jitter [4]. In the extreme case where the jitter is imposed to be zero, the problem is qualified

Clément Pira
CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
Univ de Toulouse, LAAS, F-31400 Toulouse, France
E-mail: clement.pira@laas.fr

Christian Artigues
CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
Univ de Toulouse, LAAS, F-31400 Toulouse, France
E-mail: christian.artigues@laas.fr

as a strictly periodic problem. In the avionic field, a preemptive or loosely periodic schedule is problematic because of the need to certify solutions. In this article, we consider a non-preemptive strictly periodic scheduling problem [9,6,7,1]. As shown by Baruah *et al* [3] and Korst *et al* [9,10], this is a strongly NP-complete problem. In this problem, each task i has a fixed period T_i and a processing time p_i. They are subject to non-overlapping constraints : no two tasks assigned to the same processor can overlap during any time period (see Figure 1). A solution is given by an assignment of the tasks to the processors and, for each task by the start time t_i (offset) of one of its occurrences, the other starting at $t_i + kT_i$ by strict periodicity.

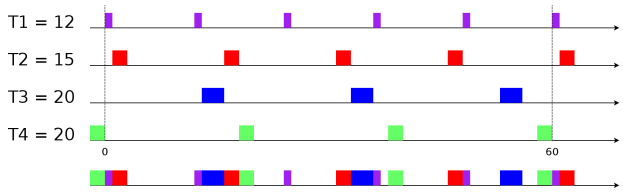

Fig. 1 $N = 4$ non-overlapping periodic tasks on $P = 1$ processor

In the following, unlike the papers mentioned above, we adopt a more general model in which processing times p_i are generalized by positive latency delays $l_{i,j} \geq 0$ (or time lags). The former case is the particular case where $l_{i,j} = p_i$ for all other tasks j. This variant has been studied mostly in the monoperiodic case [5,14]. The proposed heuristic adapts easily to this generalization.

In this paper, we only consider the case where the offsets t_i are integers. This hypothesis is important to prove convergence of the method described in section 2.1.2. Since the problem is periodic, all the periods T_i also need to be integers[1]. However the latency delays need not be integer *a priori*, especially in the case of the optimization problem presented in section 1.2.2.

1.2 Definition of the uniprocessor problem

1.2.1 Non-overlapping constraints

In the formulation of a uniprocessor periodic scheduling problem, a latency delay $l_{i,j} \geq 0$ has to be respected whenever an occurrence of a task j starts after an occurrence of a task i. Said differently, the smallest positive difference between two such occurrences has to be greater than $l_{i,j}$. Using Bézout identity, the set $(t_j+T_j\mathbb{Z})-(t_i+T_i\mathbb{Z})$ of all the possible differences is equal to $(t_j - t_i) + g_{i,j}\mathbb{Z}$ where $g_{i,j} = \gcd(T_i, T_j)$. The smallest positive representative of this set is $(t_j - t_i) \mod g_{i,j}$ (in particular, we consider a classic positive modulo, and not a signed modulo). Therefore, we simply consider the following constraint :

$$(t_j - t_i) \bmod g_{i,j} \geq l_{i,j}, \qquad \forall (i,j),\ i \neq j \tag{1}$$

[1] Indeed, if t_i is an integer solution then $t_i + T_i$ should also be an integer solution, hence their difference too, *i.e.* T_i.

Note that except for the modulo, these constraints have a clear analogy with classical precedence constraints of the form $t_j - t_i \geq l_{i,j}$. However, they are much harder to handle since they induce an exclusion between tasks. In particular, a disjunctive constraint $t_j - t_i \geq l_{i,j} \vee t_i - t_j \geq l_{i,j}$ can always be represented by the two constraints $(t_j - t_i) \bmod T \geq l_{i,j}$ and $(t_i - t_j) \bmod T \geq l_{j,i}$ for a large enough period T.

Classically, processing times are strictly positive, however our model allows zero delays. Since equation (1) is trivially satisfied when $l_{i,j} = 0$, we introduce an additional graph $\mathcal{G}$ containing the arcs (i, j) for which $l_{i,j} > 0$, and we consider equation (1) only for those couples. As we will see, the constraints associated with (i, j) and (j, i) work naturally together and some complications are introduced when only one of the couples has a strictly positive delay. To avoid special cases, we will suppose in the following that the graph $\mathcal{G}$ is symetric[2] : two delays $l_{i,j}$ and $l_{j,i}$ are either both zero, or both strictly positive. With this hypothesis, a consequence of proposition 1 (see Appendix) is that equation (1) can be replaced by the following interval constraint for each couple $(i, j) \in \mathcal{G}$ with $i < j$ (see [9]) :

$$l_{i,j} \leq (t_j - t_i) \bmod g_{i,j} \leq g_{i,j} - l_{j,i}, \qquad \forall (i,j) \in \mathcal{G},\ i < j \tag{2}$$

Since $\mathcal{G}$ is symetric, it can be seen as an undirected graph. We will write $\mathcal{G}(i)$ for the set of neighbors of i, *i.e.* the set of tasks with which i is linked through non-overlapping constraints.

1.2.2 Objective to maximize

Robustness is concerned with the problem of computing solutions which can withstand uncertainties on the parameters. In the context of hard real time system, like the one encountered in the avionic field, the uncertainties on the delays (processing times, latencies) are particularly critical. A task may last longer than expected, and therefore these parameters are generally overestimated *a priori*. Instead of this, we define here a measure of the robustness of a schedule which capture its capacity to handle increases of the durations. For this, we suppose that the possible variations of the delays are proportional to their original values : tasks with large processing times are more likely to be subject to large overrun. Hence, all the delays $l_{i,j}$ are made proportional to a common factor $\alpha \geq 0$ that we try to optimize (see [1,2]).

$$\begin{aligned}
\max \quad & \alpha && && (3)\\
s.t. \quad & l_{i,j}\alpha \leq (t_j - t_i) \bmod g_{i,j} \leq g_{i,j} - l_{j,i}\alpha && \forall (i,j) \in \mathcal{G},\ i < j && (4)\\
& t_i \in \mathbb{Z} && \forall i && (5)\\
& \alpha \geq 0 && && (6)
\end{aligned}$$

Whenever a solution with $\alpha \geq 1$ is found, we have a solution for the feasibility problem. If a larger factor is found, the schedule can handle critical situations where all the delays have been multiplied by this factor.

The addition of this objective is especially usefull for heuristic approaches (e.g. local search), for which we need a criterium to select better solutions than the current one. The α-criterium given above amounts to maximize space between tasks, and this idea is at the base of the heuristic presented in this paper (see section 2).

[2] Note that this hypothesis is quite natural (and can be enforced anyway) : if $l_{i,j} > 0$, then the tasks i and j are constrained not to start at the same time. Hence the same is true for the tasks j and i. Therefore, $l_{j,i}$ can also be supposed strictly positive.

1.3 MILP formulation of the multiprocessor problem

In this section, we present a MILP formulation of the multiprocessor problem which will be used to compare the performance of our heuristic.

1.3.1 MILP formulation

In the multiprocessor case, the scheduling problem is coupled with an assignment problem. Thus, we introduce binary variables $a_{i,k}$ which indicate if a task i is assigned to a processor k, and variables $x_{i,j}$ which indicate if i and j are on different processors. These variables must satisfy (8) and (9). For a given couple of tasks $(i,j) \in \mathcal{G}$, with $i < j$, the non-overlapping constraint (4) can be rewritten as $l_{i,j}\alpha \leq t_j - t_i + g_{i,j}q_{i,j} \leq g_{i,j} - l_{j,i}\alpha$. Indeed, the modulo $(t_j - t_i) \bmod g_{i,j}$ is the only element $t_j - t_i + g_{i,j}q_{i,j}$ in the interval $[0, g_{i,j})$, where $q_{i,j}$ is an additional 'quotient variable'. More precisely, the non-overlapping constraints have to be satisfied whenever two tasks are on the same processor. Therefore, we replace the term $l_{i,j}\alpha$ by $l_{i,j}\alpha - l_{i,j}\alpha_{\max}x_{i,j}$, where $\alpha_{\max}$ is an upper bound on the value of α (see section 1.3.3). When i and j are on the same processor ($x_{i,j} = 0$), we find back the original term $l_{i,j}\alpha$, otherwise we obtain a negative term $l_{i,j}\alpha - l_{i,j}\alpha_{\max}$ which makes the constraint trivially satisfiable. This yields constraints (10) and (11).

$$\max \quad \alpha \tag{7}$$

$$\sum_k a_{i,k} = 1 \qquad \forall i \tag{8}$$

$$x_{i,j} \leq 2 - a_{i,k} - a_{j,k} \qquad \forall k, \quad \forall (i,j) \in \mathcal{G}, i < j \tag{9}$$

$$t_j - t_i + g_{i,j}q_{i,j} \geq l_{i,j}\alpha - \alpha_{\max}l_{i,j}x_{i,j} \qquad \forall (i,j) \in \mathcal{G}, i < j \tag{10}$$

$$t_j - t_i + g_{i,j}q_{i,j} \leq g_{i,j} - l_{j,i}\alpha + \alpha_{\max}l_{j,i}x_{i,j} \qquad \forall (i,j) \in \mathcal{G}, i < j \tag{11}$$

$$t_i \in \mathbb{Z} \qquad \forall i \tag{12}$$

$$q_{i,j} \in \mathbb{Z} \qquad \forall (i,j) \in \mathcal{G}, i < j \tag{13}$$

$$x_{i,j} \in [0,1] \qquad \forall (i,j) \in \mathcal{G}, i < j \tag{14}$$

$$a_{i,k} \in \{0,1\} \qquad \forall k, \forall i \tag{15}$$

1.3.2 Bounds on the offsets and quotients

The previous system is invariant under translation $t_i \mapsto t_i + n_i T_i$. More precisely, if (t_i) is feasible and if (n_i) is a vector of integers, then the solution (t'_i) defined by $t'_i = t_i + n_i T_i$ is also feasible, provided we define new quotients $q'_{i,j} = q_{i,j} + n_i \frac{T_i}{g_{i,j}} - n_j \frac{T_j}{g_{i,j}}$. In the previous reasoning, the only important fact is that $q'_{i,j}$ is again integer since $g_{i,j}$ divides T_i and T_j. Therefore, it remains true if we replace T_i with an updated period :

$$T_i^* = \operatorname{lcm}((g_{i,j})_{j \in \mathcal{G}(i)}) = \gcd(T_i, \operatorname{lcm}((T_j)_{j \in \mathcal{G}(i)})) \tag{16}$$

The updated period T_i^* always divides T_i and we can have a strictly smaller value for example if a prime factor p only occurs in one period T_i (or more generally if a factor p occurs with multiplicity m in T_i but with strictly smaller multiplicities in any other periods of a task connected to i). Indeed, even if the (T_i) are the initial parameters,

only their GCD appear in the constraints. Therefore, if a prime factor occurs in only one period, then it completely disappear in the $g_{i,j}$. By replacing the initial periods (T_i) by the updated ones (T_i^*), we simply remove some irrelevant factors. In the following, we will suppose that the periods have been updated to have no proper factors, *i.e.* $T_i = T_i^*$.

Since $t_i \bmod T_i = t_i + n_i T_i$ for some $n_i \in \mathbb{Z}$, we deduce that $(t_i \bmod T_i)$ is also a solution. We can therefore impose additionnal bounds $t_i \in [0, T_i - 1]$. And since, we have $0 \leq t_j - t_i + g_{i,j} q_{i,j} \leq g_{i,j}$, these bounds on the offsets immediately induce associated bounds on the quotients :

$$1 - \frac{T_j}{g_{i,j}} \leq q_{i,j} \leq \frac{T_i}{g_{i,j}} \tag{17}$$

In particular, if T_j divides T_i, then $q_{i,j} \geq 0$ (in fact $q_{i,j} \in [0, T_i/T_j]$), which shows that in the harmonic case, we can impose positive variables. Conversely if T_i divides T_j then $q_{i,j} \leq 1$. Finally, if $T_i = T_j$, then $q_{i,j} \in \{0, 1\}$ which shows that the monoperiodic case correspond to the case of binary variables.

1.3.3 Upper bound on the α-value

Let i and j be two tasks on the same processor. Then, constraint (4) implies the following upper bound : $\alpha \leq g_{i,j}/(l_{i,j} + l_{j,i})$. Taking the integrality assumption into account, we get an even tighter upper bound :

$$\alpha \leq \alpha_{\max}^{i,j} = \max\left(\frac{1}{l_{i,j}} \left\lfloor \frac{g_{i,j}}{l_{i,j} + l_{j,i}} l_{i,j} \right\rfloor \ , \ \frac{1}{l_{j,i}} \left\lfloor \frac{g_{i,j}}{l_{i,j} + l_{j,i}} l_{j,i} \right\rfloor \right) \tag{18}$$

These bounds are illustrated on Figure 2. In the monoprocessor case, we deduce that $\alpha_{\max} = \min_{i,j} \alpha_{\max}^{i,j}$ is an upper bound on the value of α. In the multiprocessor case, we have at least the trivial upper bound $\alpha_{\max} = \max_{i,j} \alpha_{\max}^{i,j}$. However, since $\alpha_{\max}$ is used as a 'big-M' in the MILP formulation, we would like the lowest possible value in order to improve the efficiency of the model. For this, we solve a preliminary model, dealing only with the assignment (hence without variables (t_i) and $(q_{i,j})$), in which non-overlapping constraints (10-11) are replaced by the following weaker constraint :

$$\alpha \leq \alpha_{\max}^{i,j} + (\alpha_{\max} - \alpha_{\max}^{i,j}) x_{i,j}, \qquad \forall (i,j) \in \mathcal{G},\ i < j \tag{19}$$

Intuitively, this constraint indicates that α should be less than $\alpha_{\max}^{i,j}$ if i and j are assigned to the same processor ($x_{i,j} = 0$), otherwise it is bounded by $\alpha_{\max}$ which is the trivial upper bound. Solving this model is much faster than for the original one (less that $1s$ for instances with 4 processors and 20 tasks). It gives us a new value $\alpha_{\max}$ which can be used in the original model.

2 An equilibrium-based heuristic

Given a set of precedence constraints, the Bellman-Ford algorithm allows to find compatible offsets. A way to implement this algorithm is to cyclically choose a task and to push it forward, up to the first feasible offset. In the case of non-overlapping constraints, we can find the next feasible offset for a task, if it exists, by a method analogous to the propagation mechanism presented in section 2.2.5. However there isn't necessarily

a solution and a similar mechanism which would cyclically move one task at a time to the next feasible location would have only a small chance to converge towards a feasible solution. The addition of the α-value solves this problem : instead of choosing the next feasible location (possibly nonexistent), we search for the best location with respect to this additional objective α, what we call the best-response method. By moving the tasks round after round, we increase the space between them. This is essentially the principle of the heuristic presented in this section.

2.1 Heuristic and the mutiprocessor best response method

2.1.1 The multiprocessor best response method

The main component of the algorithm is called the best response procedure. It takes its name from a game theory analogy. Each task is seen as an agent which tries to optimize its own assignment a_i and offset t_i, while the other assignments (a_j^*) and offsets (t_j^*) are fixed. Instead of using a binary vector $(a_{i,k})$ as in the MILP formulation, we represent an assignment more compactly by a variable $a_i \in [1, P]$. For each agent i, and each processor p, we first define a method $\textsc{BestOffset}_i^p$ which returns the best possible offset for task i on processor p. More formally, the $\textsc{BestOffset}_i^p$ procedure consists in solving the following program :

$$
\begin{aligned}
(BO_i^p)\ \max \quad & \alpha && && (20)\\
s.t. \quad & l_{j,i}\alpha \leq (t_i - t_j^*) \bmod g_{i,j} \leq g_{i,j} - l_{i,j}\alpha && \substack{\forall j \in \mathcal{G}(i)\\ a_j^* = p} && (21)\\
& t_i \in \mathbb{Z} && && (22)\\
& \alpha \geq 0 && && (23)
\end{aligned}
$$

Contraints (21) are the non-overlapping contraints of task i with other tasks j currently on this processor ($a_j^* = p$). We define N_p to be the number of such contraints (we have $\sum_p N_p = |\mathcal{G}(i)| \leq N - 1$). Note that there are only two variables, t_i and α, since the other offsets and assignments $(t_j^*, a_j^*)_{j \in \mathcal{G}(i)}$ are parameters. Following the discussion in section 1.3.2, the previous system is invariant under tranlation by T_i, and even by the possibly smaller value $T_i^p = \text{lcm}((g_{i,j})_{j \in \mathcal{G}(i) | a_j^* = p})$. Hence, we can impose t_i to belong to $[0, T_i^p - 1]$. Since t_i is integer, we can trivially solve this program by computing the α-value for each of the offsets $\{0, \cdots, T_i^p - 1\}$, using the following expression, and selecting the best one.

$$
\alpha = \min_{\substack{j \in \mathcal{G}(i)\\ a_j^* = p}} \min \left(\frac{(t_i - t_j^*) \bmod g_{i,j}}{l_{j,i}} \ , \ \frac{(t_j^* - t_i) \bmod g_{i,j}}{l_{i,j}} \right) \tag{24}
$$

A much better approach to solve this program will be presented in section 2.2. Given this procedure, we define a method $\textsc{MultiProcBestResponse}_i$ which returns the best assignment and offset for the task i, given the current assignments and offsets (t_j^*, a_j^*) of all the tasks. In order to choose the assignment, an agent simply compute the best-offset on each processor and select the best one. It starts with the current processor a_i, which has priority in case of equality. Some of the next processors can sometimes be skipped. Indeed, for a given processor p, we can compute an upper bound on the objective : $\alpha_{\max}^p = \min_{j \in \mathcal{G}(i) | a_j^* = p} \alpha_{\max}^{i,j}$ (similar to section 1.3.3). Therefore, if the current best solution found on previous processors is already better than this upper

bound, there is no way to improve the current solution with processor p, which can be skipped. The multiprocessor best-response procedure is summarized in Algorithm 1.

Algorithm 1 The multiprocessor best-response

procedure MULTIPROCBESTRESPONSE$_i$($(t_j^*)_{j\in I}, (a_j^*)_{j\in I}$)
  $a_i \leftarrow a_i^*$ ▷ The initial best assignment is the current one
  $(t_i, \alpha) \leftarrow \text{BESTOFFSET}_i^{a_i^*}((t_j^*)_{j\in I}, (a_j^*)_{j\in I})$ ▷ The current best offset and value
  for all $p \neq a_i^*$ **do** ▷ We test the other processors
    $\alpha_{\max}^p \leftarrow \min_{\substack{j\in\mathcal{G}(i)\\ a_j^*=p}} \alpha_{\max}^{i,j}$ ▷ We compute an upper bound on α when i is on p
    if $\alpha_{\max}^p > \alpha$ **then** ▷ An improvement can possibly be found on p
      $(\tau, \beta) \leftarrow \text{BESTOFFSET}_i^p((t_j^*)_{j\in I}, (a_j^*)_{j\in I})$ ▷ We search the best offset on p
      if $\beta > \alpha$ **then** ▷ An improvement has been found on p
        $a_i \leftarrow p$; $t_i \leftarrow \tau$; $\alpha \leftarrow \beta$ ▷ We update the best solution
      end if
    end if
  end for
  return (t_i, a_i, α)
end procedure

Since the BESTOFFSET_i^p method runs in $O(T_i N_p)$, the MULTIPROCBESTRESPONSE$_i$ method runs in $O(T_i N)$.

2.1.2 The concept of equilibrium and principle of the method to find one

Definition 1 A solution (t_i, a_i) is an equilibrium *iff* no task i can improve its assignment or offset using procedure MULTIPROCBESTRESPONSE$_i$.

In order to find an equilibrium, the heuristic uses a counter N_{stab} to count the number of tasks known to be stable, *i.e.* which cannot be improved. It starts with an initial solution (for example randomly generated) and tries to improves this solution by a succession of unilateral optimizations. On each round, we choose cyclically a task i and try to optimize its schedule, *i.e.* we apply MULTIPROCBESTREPONSE$_i$. If no improvement was found, then one more task is stable, otherwise we update the assignment and offset of task i and reinitialize the counter of stable tasks. We continue until N tasks are stable. This is summarized in Algorithm 2.

Remark 1 The main reason for the use of integers is that it allows to guarantee the convergence of the heuristic. The termination proof relies on the fact that the underlying game is an ordinal potential game. This potential strictly increases on each round, preventing the algorithm from cycling. And since it can take only a finite number of values, the algorithm converges after a finite number of steps (we refer to [1] for the proof of termination and correction). This is not the case with fractional offsets, for which the heuristic is unlikely to converge in a finite number of steps.

An equilibrium is only an approximate notion of optimum. Hence, in order to find a real optimum, the idea is now to run the previous heuristic several times with different randomly generated initial solutions, and to keep the best result, following a standard multistart scheme.

Algorithm 2 The heuristic

procedure ImproveSolution($(t_j)_{j\in I}, (a_j)_{j\in I}$)
  $N_{\text{stab}} \leftarrow 0$ ▷ The number of stabilized tasks
  $i \leftarrow 0$ ▷ The task currently optimized
  while $N_{\text{stab}} < N$ **do** ▷ We run until all the tasks are stable
    $(new\_t_i, new\_a_i, \alpha_i) \leftarrow$ MultiProcBestResponse$_i((t_j)_{j\in I})$
    if $new\_t_i \neq t_i$ **or** $new\_a_i \neq a_i$ **then**
      $t_i \leftarrow new\_t_i$; $a_i \leftarrow new\_a_i$ ▷ We have a strict improvement for task i
      $N_{\text{stab}} \leftarrow 1$ ▷ We restart counting the stabilized tasks
      $\alpha \leftarrow \alpha_i$
    else ▷ We do not have a strict improvement
      $N_{\text{stab}} \leftarrow N_{\text{stab}} + 1$ ▷ One more task is stable
      $\alpha \leftarrow \min(\alpha_i, \alpha)$
    end if
    $i \leftarrow (i+1) \bmod N$ ▷ We consider the next task
  end while
  return $(\alpha, (t_j)_{j\in I}, (a_i)_{j\in I})$
end procedure

2.2 The best-offset procedure on a given processor

We now need to implement the BestOffset$_i^p$ method, *i.e.* to solve (BO_i^p), more efficiently than by the trivial linear procedure presented in section 2.1.1. Since the latter runs in $O(T_i N_p)$, any method should at least be faster. In [2], the authors propose a method consisting in computing the α-value only for a set of precomputed intersection points (in the next section we will see that a fractional optimum is at the intersection of an increasing and a decreasing line). This already improves the trivial procedure. In the following, we present an even more efficient method which relies on the fact that (BO_i^p) is locally a two-dimensional linear program.

2.2.1 Structure of the solution set

If we try to draw the function $\alpha = (t_i - t_j^*) \bmod g_{i,j}/l_{j,i}$ representing the optimal value of α respecting one constraint $(t_i - t_j^*) \bmod g_{i,j} \geq l_{j,i}\alpha$ when t_i takes rational values, we obtain a curve which is piecewise increasing and discontinuous since it becomes zero on $t_j^* + g_{i,j}\mathbb{Z}$. In the same way, if we draw $\alpha = (t_j^* - t_i) \bmod g_{i,j}/l_{i,j}$, we obtain a curve which is piecewise decreasing and becomes zero at the same points. It is therefore more natural to consider the two constraints jointly, *i.e.* (4), which gives a continuous curve (see Figure 2) [3].

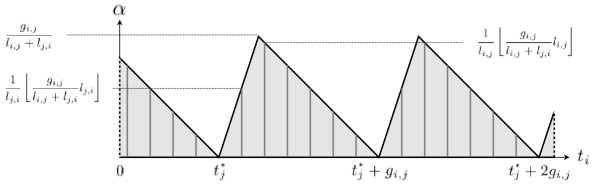

Fig. 2 Possible values for (t_i, α) when constrained by a single task j

[3] Note that if $\mathcal{G}$ was not symetric, we would need to consider some degenerate cases were one of the slope (increasing or decreasing) would be infinite (since $l_{i,j} = 0$ or $l_{j,i} = 0$).

Given some fixed offsets $(t_j^*)_{j\in\mathcal{G}(i)}$, the set of solutions (t_i, α) of (BO_i^p) is the intersection of the sets decribed above for each $j \in \mathcal{G}(i),\ a_j^* = p$ (see Figure 3). It is composed of several adjacent polyhedra. We can give an upper bound on the number n_{poly} of such polyhedra. A polyhedron starts and ends at zero points. For a given constraint j, there is $T_i/g_{i,j}$ zero points in $[0, T_i - 1]$, hence n_{poly} is bounded by $T_i \sum_{\substack{j\in\mathcal{G}(i)\\ a_j^*=p}} \frac{1}{g_{i,j}}$. This upper bound can be reached when the offsets are fractional, since in this case, we can always choose the offsets t_j^* such that the sets of zero points $(t_j^* + g_{i,j}\mathbb{Z})_{j\in\mathcal{G}(i)}$ are all disjoint. In the case of integer offsets, there is obvioulsy at most T_i zero points in the interval $[0, T_i - 1]$ and therefore, at most T_i polyhedra.

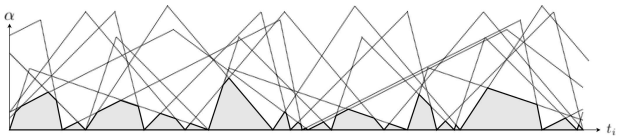

Fig. 3 Possible values for (t_i, α) constrained by all tasks $j \in \mathcal{G}(i),\ a_j^* = p$

When solving the best-response method, we try to find the new location t_i which maximizes the associated α-value.

2.2.2 Local (two dimensional) polyhedron

We want to compute the local polyhedron which contains a reference offset t_i^{ref}. Locally, the constraint $(t_i - t_j^*) \bmod g_{i,j} \geq l_{j,i}\alpha$ is linear, of the form $t_i - o_j \geq l_{j,i}\alpha$. Here, o_j is the largest $\tau \leq t_i^{\text{ref}}$ such that $(\tau - t_j^*) \bmod g_{i,j} = 0$. In the same way, we can compute the decreasing constraint $o'_j - t_i \geq l_{i,j}\alpha$. In this case o'_j is the smallest $\tau \geq t_i^{\text{ref}}$ such that $(o'_j - \tau) \bmod g_{i,j} = 0$. By proposition 2 (see Appendix), we have :

$$o_j = t_i^{\text{ref}} - (t_i^{\text{ref}} - t_j^*) \bmod g_{i,j} \qquad \text{and} \qquad o'_j = t_i^{\text{ref}} + (t_j^* - t_i^{\text{ref}}) \bmod g_{i,j} \tag{25}$$

Therefore, we obtain a local polyhedron (see figure 4). Note that when $(t_i^{\text{ref}} - t_j^*) \bmod g_{i,j} > 0$, we simply have $o'_j = o_j + g_{i,j}$ by proposition 1 (see Appendix). However, when $(t_i^{\text{ref}} - t_j^*) \bmod g_{i,j} = 0$, we have $o_j = o'_j = t_i^{\text{ref}}$. In this case, the polyhedron is degenerated since it contains only $\{t_i^{\text{ref}}\}$, and the α-value at t_i^{ref} is zero. Instead of computing this polyhedron, we prefer to choose either the polyhedron on the right, or on the left (see figure 5). If we choose the one on the right, this amounts to defining o'_j to be the smallest $\tau > t_i^{\text{ref}}$ which sets the constraint to zero. By proposition 2, we have $o'_j = t_i^{\text{ref}} + g_{i,j} - (t_i^{\text{ref}} - t_j^*) \bmod g_{i,j}$. Therefore, this simply amounts to enforcing $o'_j = o_j + g_{i,j}$. Choosing the polyhedron on the left amounts to defining o'_j using (25) and to enforcing $o_j = o'_j - g_{i,j}$.

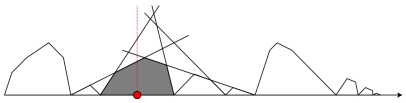

Fig. 4 Selection of the polyhedron containing a reference offset t_i^{ref}

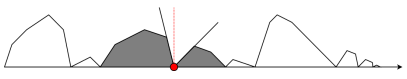

Fig. 5 Two possibilities in the degenerate case $\alpha = 0$

2.2.3 Solving the local best offset problem

Once the local polyhedron has been defined, the problem is now to solve the following MILP :

$$
\begin{aligned}
(Loc{-}BO_i^p) \qquad \max \quad & \alpha && && (26)\\
s.t. \quad & t_i - l_{j,i}\alpha \geq o_j && \forall j \in \mathcal{G}(i),\ a_j^* = p && (27)\\
& t_i + l_{i,j}\alpha \leq o'_j && \forall j \in \mathcal{G}(i),\ a_j^* = p && (28)\\
& t_i \in \mathbb{Z} && && (29)
\end{aligned}
$$

We can first search for a fractional solution, and for this we can use any available method of linear programming. However, since the problem is a particular two dimensional program, we can give special implementations of these methods. In the following, we present a simple primal simplex approach, which runs in $O(N_p^2)$ in the worst case but has a good behaviour in practice. A local polyhedron is delimited by increasing and decreasing lines, and the fractional optimum is at the intersection of two such lines. Hence a natural way to find the optimum is to try all the possible intersections between an increasing line, of the form $t_i - l_{j,i}\alpha = o_j$, and a decreasing line, of the form $t_i + l_{i,k}\alpha = o'_k$, and to select the one with the smallest α-value. The coordinates of these intersection points are given by[4] :

$$t_i = \frac{l_{j,i}o'_k + l_{i,k}o_j}{l_{j,i} + l_{i,k}} \qquad \text{and} \qquad \alpha = \frac{o'_k - o_j}{l_{j,i} + l_{i,k}} \tag{30}$$

Since there is N_p lines of each kinds, the algorithm runs in $O(N_p^2)$. In practice, a better approach (with the same worst case complexity) is to start with a couple of increasing and decreasing lines, and alternatively to try to improve the decreasing line (see Figure 6), then the increasing one (see Figure 7), and so on, until no improvement is made. The overall solving process is illustrated on Figure 8.

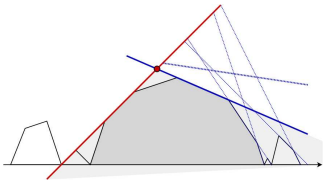

Fig. 6 The lowest intersection point of a fixed increasing line with decreasing lines

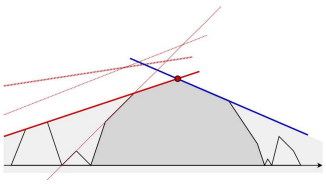

Fig. 7 The lowest intersection point of a fixed decreasing line with increasing lines

Remark 2 Suppose we just improved the decreasing line (Figure 6) and we get a new intersection point. We know that the whole local polyhedron lies inside the cone oriented below and defined by the increasing and the decreasing line. Then all the decreasing lines with a smaller slope than the new decreasing line, *i.e.* with a larger delay $l_{i,j}$, lie completely outside this cone, and therefore cannot be active at the optimum. We can therefore drop these lines in the subsequent phases. In other words, the delay $l_{i,j}$ of the decreasing line strictly decreases with time (except for the first improvement, since the initial decreasing line can be any line). The same is true for the increasing line.

[4] Note that in the computation only the α-coordinate is needed since the t_i-coordinate of the selected intersection point can be computed afterward by $t_i = o_j + l_{j,i}\alpha$

Let's define a phase to be the computation of a new increasing or decreasing line. Then the previous remark implies that the number of "decreasing" phases (*resp.* "increasing" phases) is bounded by the number of distinct delays $l_{i,j}$ (*resp.* $l_{j,i}$). Since a phase runs in $O(N_p)$, the method runs in $O(N_p^2)$ in the worst case of distinct delays. This gives us a method with the same complexity to compute an integral solution since once a fractional solution has been found, we can deduce an integral solution with an additional computation in $O(N_p)$. Indeed, if t_i is integer, then (t_i, α) is the desired solution. Otherwise we can compute the α-values α_- and α_+ associated with $\lfloor t_i \rfloor$ and $\lceil t_i \rceil$ and take the largest one. Note that since $\lfloor t_i \rfloor$ (*resp.* $\lceil t_i \rceil$) is on the increasing phase (*resp.* decreasing phase), only the corresponding constraints are needed to compute α_- (*resp.* α_+) :

$$\alpha_- = \min_{\substack{j \in \mathcal{G}(i) \\ a_j^* = p}} (\lfloor t_i \rfloor - o_j)/l_{j,i} \qquad \text{and} \qquad \alpha_+ = \min_{\substack{k \in \mathcal{G}(i) \\ a_k^* = p}} (o'_k - \lceil t_i \rceil)/l_{i,k} \tag{31}$$

We call this a primal approach since this is essentially the application of the primal simplex algorithm. However the primal simplex is not applied on $(Loc{-}BO_i^p)$ but on its dual. A dual approach is illustrated on Figure 9 and presented in a technical report [13]. From a theoretical perspective, this approach is outperformed by Megiddo algorithm [12] which allows to find a solution in $O(N_p)$. However Megiddo algorithm is more complex and generally slower in practice.

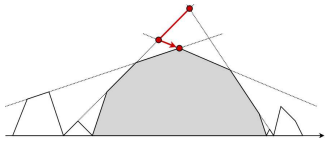

Fig. 8 Finding the fractional optimum with a primal simplex approach

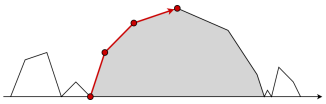

Fig. 9 Finding the fractional optimum with a dual simplex approach

2.2.4 The case of processing times

In the case of processing times, no such refinement as Megiddo algorithm is needed in order to obtain a complexity in $O(N_p)$. In fact, in this case the primal simplex method already runs in $O(N_p)$ since it stops after three phases. Indeed, in the case of processing times, we have $l_{i,j} = p_i$, which means than all the decreasing line have the same slope $-1/p_i$ (see Figure 10). Thus, even if initially there are N_p decreasing lines with equations $o'_j - t_i = \alpha p_i$, the one with the smallest o'_j will be selected after the first phase. In a second phase, we search for a better increasing line. The third phase can be dropped since we will not find a better decreasing line : we are at the optimum.

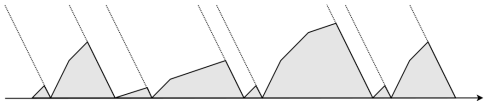

Fig. 10 All the decreasing lines are parallel in the case of processing times

2.2.5 Finding the next improving offset

After a first local optimization, we obtain a solution t_i^{lb} with value α_{lb}, which becomes a lower bound. In the rest of the procedure, we are only interested by polyhedra which could improve this value. As α_{lb} is improved, more and more polyhedra will lie completely below this level and will be skipped (see darker polyhedra in Figure 11).

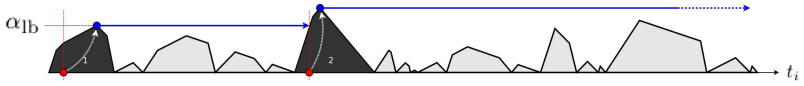

Fig. 11 Propagating up to the next strictly improving offset

Starting from a current solution t_i^{lb} with value α_{lb}, we want to find a new integer solution t_i greater than the current solution which strictly improves its value. Hence, this solution should satisfy :

$$\alpha_{\text{lb}} l_{j,i} < (t_i - t_j^*) \bmod g_{i,j} < g_{i,j} - \alpha_{\text{lb}} l_{i,j} \qquad \substack{\forall j \in \mathcal{G}(i) \\ a_j^* = p} \tag{32}$$

However, since the middle expression gives an integer value, we can round the bounds and obtain the equivalent interval constraint (34). Moreover, among all the possible solutions, we would like to find the smallest one :

$$\min \quad t_i \tag{33}$$

$$s.t. \quad \lfloor \alpha_{\text{lb}} l_{j,i} \rfloor + 1 \leq (t_i - t_j^*) \bmod g_{i,j} \leq g_{i,j} - \lfloor \alpha_{\text{lb}} l_{i,j} \rfloor - 1 \qquad \substack{\forall j \in \mathcal{G}(i) \\ a_j^* = p} \tag{34}$$

$$t_i \in \{t_i^{\text{lb}}, \cdots, t_i^{\text{end}}\} \tag{35}$$

Here, t_i^{end} (which will be defined in the next section) indicates when to stop the search (since there is possibly no strictly improving solution). In order to solve this program, we will start with $t_i = t_i^{\text{lb}}$ and propagate on the right until we find a feasible solution. On a given iteration, we choose a constraint j and check if it is satisfied. We could check the two inequalities of the interval constraint (34), however we can also remark that this is equivalent to :

$$(t_i - t_j^* - \lfloor \alpha_{\text{lb}} l_{j,i} \rfloor - 1) \bmod g_{i,j} \leq g_{i,j} - \lfloor \alpha_{\text{lb}} l_{i,j} \rfloor - \lfloor \alpha_{\text{lb}} l_{j,i} \rfloor - 2 \tag{36}$$

Therefore we can compute $r = g_{i,j} - (t_i - t_j^* - \lfloor \alpha_{\text{lb}} l_{j,i} \rfloor - 1) \bmod g_{i,j}$. If $r < \lfloor \alpha_{\text{lb}} l_{i,j} \rfloor + \lfloor \alpha_{\text{lb}} l_{j,i} \rfloor + 2$, the constraint is violated. In this case, we compute the smallest offset τ strictly greater than the current one, and which satisfies the current constraint, *i.e.* we compute $\tau > t_i$ such that $(\tau - t_j^* - \lfloor \alpha_{\text{lb}} l_{j,i} \rfloor - 1) \bmod g_{i,j} = 0$. By proposition 2, we have $\tau = t_i + r$. For this offset τ, the current constraint is now verified. We set $t_i = \tau$ and continue the process with the next constraint. We cycle along the constraints, until no update is made during N_p successive rounds or the offset t_i becomes greater than t_i^{end}. In the former case, all the constraints are satisfied by the current offset, otherwise there is no solution and we return the special value $\varnothing$. This procedure is summarized in algorithm 3. For convenience, we suppose here that the constraints $j \in \mathcal{G}(i),\ a_j^* = p$ are numbered from 0 to $N_p - 1$.

Algorithm 3 A propagation procedure to find an improving integral solution

procedure FindImprovingIntegralSolution($\alpha_{\text{lb}}, t_i^{\text{lb}}, t_i^{\text{end}}$)
  $t_i \leftarrow t_i^{\text{lb}}$ ▷ The current offset
  $N_{\text{sat}} \leftarrow 0$ ▷ The number of satisfied constraints
  $j \leftarrow 0$ ▷ The current constraint evaluated
  while $N_{\text{sat}} < N_p$ **do**
    $r \leftarrow g_{i,j} - (t_i - t_j^* - \lfloor \alpha_{\text{lb}} l_{j,i} \rfloor - 1) \bmod g_{i,j}$
    if $r \geq \lfloor l_{i,j}\alpha_{\text{lb}} \rfloor + \lfloor l_{j,i}\alpha_{\text{lb}} \rfloor + 2$ **then**
      $N_{\text{sat}} \leftarrow N_{\text{sat}} + 1$ ▷ One more constraint is satisfied
    else
      $t_i \leftarrow t_i + r$ ▷ Otherwise, we go to the first feasible offset
      if $t_i > t_i^{\text{end}}$ **then return** $\varnothing$ ▷ We reach the end without solution
      $N_{\text{sat}} \leftarrow 1$ ▷ We restart counting the satisfied constraint
    end if
    $j \leftarrow (j+1) \bmod N_p$ ▷ We consider the next constraint
  end while
  return t_i ▷ We return an improving offset
end procedure

2.2.6 Solving the best response problem

We are now able to describe a procedure which solves (BO_i^p). We saw that t_i can be supposed to belong to $[0, T_i^p - 1]$. More generally we can start at any initial offset t_i^{start}, for example the current value of t_i, and run on the right until we reach the offset $t_i^{\text{end}} = t_i^{\text{start}} + T_i^p - 1$. We can compute the local polyhedron (on the right) which contains the current offset. Using the primal simplex method, we solve the associated problem $(Loc{-}BO_i^p)$. We obtain a new local optimum $(t_i^{\text{lb}}, \alpha_{\text{lb}})$ which becomes our lower bound. We then use the propagation procedure to reach the next improving solution. We are in a new polyhedron and we restart the local optimization at this point, which gives us a better value. We continue until the propagation mechanism reaches t_i^{end} (see Figure 11). In the end, we obtain a best offset $t_i \in [t_i^{\text{start}}, t_i^{\text{end}}]$. If needed, we can consider $t_i \bmod T_i^p$ which is an equivalent solution in $[0, T_i^p - 1]$.

If we use Megiddo algorithm to solve the local problems, or if we use the primal simplex approach in the case of processing times, this procedure runs in $O(N_p n_{\text{poly}})$, which is bounded by $O(N_p T_i^p)$. Indeed, one iteration of the propagation mechanism runs in $O(1)$. Since the propagation progresses of one polyhedron every N_p iterations[5], the end of the period is reached after at most $N_p n_{\text{poly}}$ iterations. During this process, there is at most n_{poly} additionnal local optimizations each one running in $O(N_p)$. Note that if we use a primal simplex approach, the local optimizations run in $O(N_p^2)$ in the worst case, but most of the polyhedra are skipped by the propagation mechanism which still runs in $O(N_p n_{\text{poly}})$.

[5] Let us define a cycle to be N_p consecutive iterations of the propagation mechanism and let us show that after 2 cycles, the current offset t_i doesn't lie in the same local polyhedron. Note that after a cycle, the algorithm either stops or an update has been made. Consider a constraint j updated during the second cycle. If j was not updated during the first cycle, then this contraint was satisfied during the first cycle (hence the current offset was in a given polyhedron of Figure 2), but violated during the second. Therefore, at the end of second cycle, the offset has been pushed to the next polyhedron of Figure 2. If j was already updated during the first cycle, then the current offset has been pushed forward to the next feasible solution two times for this constraint. Therefore, the local polyhedron has also changed.

3 Results

We have tested the method on non-harmonic instances generated using the procedure described in [7] : the periods were choosen in the set $\{2^x 3^y 50 \mid x \in [0,4], y \in [0,3]\}$ and the processing times were generated following an inverse exponential distribution and averaging at about 20% of the period of the task. The experiments were performed on an Intel Core i5-480M 2.66GHz with 4GB of memory.

Table 1 presents the results on 15 instances with $N = 20$ tasks and $P = 4$ processors. Columns 4-8 contain the results of our new version of the heuristic. The value $\mathbf{start}_{10s}$ represents the number of time the heuristic was launched with different initial solutions during 10s. From this value, we compute $\mathbf{time}_{single} = 10/\mathbf{start}_{10s}$ which represents the average time needed for one execution of the heuristic. The value $\mathbf{start}_{sol}$ represents the number of starts needed to obtain the best result. Hence, the quantity $\mathbf{time}_{sol} = \mathbf{start}_{sol} \cdot \mathbf{time}_{single}$ gives approximately the time needed to obtain this best result (C++ timers were not precise enough so we preferred this approximation). Column $\mathbf{time}_{sol}$ of the MILP formulation (columns 2-3) represents the time needed by the Gurobi solver to obtain the best solution during a period of 200s[6]. In addition to being much faster, the heuristic sometimes obtains better results (see instances 1 and 3). This table also includes the results of the original heuristic presented in [1,2] (columns 9-10). The field $\mathbf{time}_{single}$ represents the time needed for a single run of their version of the heuristic. Averaging at 1.95s, these values are compatible with the results presented in [2]. The field $\mathbf{gain}_{speed}$, which is the ratio of the two values $\mathbf{time}_{single}$, shows that our version of the heuristic is incomparably faster (about 3200 times on these instances).

	MILP (200s)		New heuristic (10s)					Original heuristic [2]	
id	α_{MILP}	$\mathbf{time}_{sol}$	$\alpha_{heuristic}$	$\mathbf{starts}_{10s}$	$\mathbf{time}_{single}$	$\mathbf{starts}_{sol}$	$\mathbf{time}_{sol}$	$\mathbf{time}_{single}$	$\mathbf{gain}_{speed}$
0	2.5	159	2.5	15062	$6.64e^{-4}$	30	0.01992	1.43	2154
1	2	18	2.01091	15262	$6.55e^{-4}$	15	0.00983	3.27	4991
2	1.6	6	1.6	11018	$9.08e^{-4}$	2	0.00182	1.52	1675
3	1.6	4	1.64324	11970	$8.35e^{-4}$	45	0.03759	4.34	5195
4	2	5	2	10748	$9.30e^{-4}$	1	0.00093	3.48	3740
5*	3	7	3	20428	$4.90e^{-4}$	1	0.00049	1.63	3330
6	2.5	54	2.5	20664	$4.84e^{-4}$	30	0.01452	1.44	2976
7	2	19	2	14040	$7.12e^{-4}$	1	0.00071	0.23	323
8	2.12222	8	2.12222	17365	$5.76e^{-4}$	3	0.00173	1.03	1789
9*	2	11	2	26304	$3.80e^{-4}$	3	0.00114	2.42	6366
10	1.12	6	1.12	28778	$3.47e^{-4}$	69	0.02398	0.72	2072
11	2.81098	20	2.81098	13355	$7.49e^{-4}$	7	0.00524	3.78	5048
12	1.5	7	1.5	11444	$8.74e^{-4}$	4	0.00350	0.27	309
13	1.56833	49	1.56833	25997	$3.85e^{-4}$	1	0.00038	1.77	4601
14	2	8	2	21606	$4.63e^{-4}$	2	0.00093	1.85	3997

Table 1 Results of the MILP, the heuristic of [2], and the new version of the heuristic on instances with $P = 4$ processors and $N = 20$ tasks

These good results have encouraged us to perform additional tests on large instances (50 processors, 1000 tasks). Table 2 presents the results for 10 instances, where $\mathbf{starts}_{1000s}$ is the number of runs performed during 1000s, $\mathbf{time}_{single}$ is the average

[6] We fix a timeout of 200s because the solver almost never stop even after 1h of CPU time (except for instances 5 and 9 for which optimality has been proved in $7s$ and $20s$ respectively).

time for one run, $\mathbf{starts}_{\text{sol}}$ is the round during which the best solution was found, and $\mathbf{time}_{\text{sol}}$ is the corresponding time. This shows that our heuristic can give feasible solutions ($\alpha \geq 1$) in about 1min, while these instances cannot even be loaded by the MILP solver (since they involve millions of variables and constraints). In order to evaluate the contribution of the propagation mechanism to the solving process, we also present results where the propagation has been replaced by a simpler mechanism : once we are at a local optimum, we follow the decreasing line active at this point, until we reach the x-axis; this gives us a next reference offset and therefore a next polyhedron. While the impact of the propagation is quite small in the case of small instances, we see on these large instances that the propagation mechanism accelerates the process by a factor of 37 (average ratio of the two $\mathbf{time}_{\text{single}}$ values).

	With propagation					Without propagation				
id	$\alpha_{\text{heuristic}}$	$\mathbf{starts}_{1000s}$	$\mathbf{time}_{\text{single}}$	$\mathbf{starts}_{\text{sol}}$	$\mathbf{time}_{\text{sol}}$	$\alpha_{\text{heuristic}}$	$\mathbf{starts}_{1000s}$	$\mathbf{time}_{\text{single}}$	$\mathbf{starts}_{\text{sol}}$	$\mathbf{time}_{\text{sol}}$
0	1	386	2.59	6	13.2	1	16	66.07	6	405.45
1	1.16452	200	5.01	35	197.6	1.05	6	177.9	5	885.06
2	1.1	111	9.04	78	694.17	1.04	4	261.19	1	405.32
3	1.21795	86	11.69	8	62.57	1.176	4	300.03	1	157.18
4	1	105	9.57	1	7.24	1	4	309.70	1	306.34
5	1.66555	121	8.30	114	952.62	1.58795	2	798.01	1	800.95
6	1	240	4.17	6	26.49	1	15	72.29	6	403.05
7	1.2	88	11.54	37	484.91	1	4	235.44	1	278.17
8	1.39733	83	12.09	47	468.93	1.21127	2	618.75	2	1237.5
9	1.2	77	13.29	1	20.58	1.2	2	487.03	1	420.58

Table 2 Results of the new version of the heuristic on instances with $P = 50$ processors and $N = 1000$ tasks, with $timeout = 1000s$

4 Conclusion

In this paper, we have proposed an enhanced version of a heuristic, first presented in [1, 2], and allowing to solve a NP-hard strictly periodic scheduling problem. More specifically, we present an efficient way to solve the best-response problem. This procedure alternates between local optimizations and an efficient propagation mechanism which allows to skip most of the polyhedra. The results show that the new heuristic greatly improves the original one and compares favorably with MILP solutions. In particular, it can handle instances out of reach of the MILP formulation.

A Appendix

Proposition 1 *If* $x \bmod a = 0$*, then* $(-x) \bmod a = 0$*, otherwise* $(-x) \bmod a = a - x \bmod a$*. In particular* $x \bmod a > 0 \Leftrightarrow (-x) \bmod a > 0$*.*

Proposition 2 *(1) The smallest* $y \geq a$ *such that* $(y - b) \bmod c = 0$ *is given by* $y = a + (b - a) \bmod c$*. (2) The smallest* $y > a$ *such that* $(y - b) \bmod c = 0$ *is given by* $y = a + c - (a - b) \bmod c$*. (3) The largest* $y \leq a$ *such that* $(y - b) \bmod c = 0$ *is given by* $y = a - (a - b) \bmod c$*. (4) The largest* $y < a$ *such that* $(y - b) \bmod c = 0$ *is given by* $y = a - c + (b - a) \bmod c$*.*

Acknowledgements This work was funded by the French Midi-Pyrenee region (allocation de recherche post-doctorant n°11050523) and the LAAS-CNRS OSEC project (Scheduling in Critical Embedded Systems).

References

1. A. Al Sheikh. Resource allocation in hard real-time avionic systems - Scheduling and routing problems. PhD thesis, LAAS, Toulouse, France, 2011.
2. A. Al Sheikh, O. Brun, P.E. Hladik, B. Prabhu. Strictly periodic scheduling in IMA-based architectures. Real Time Systems, Vol 48, N°4, pp. 359-386, 2012.
3. S. Baruah, L. Rousier, I. Tulchinsky, D. Varvel. The complexity of periodic maintenance. In: Proceedings of the International Computer Symposium, 1990.
4. S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In International Conference on Real-Time Computing Systems and Applications, pp. 62-69, Hong Kong, 1999.
5. W. Dauscha, H.D. Modrow, A. Neumann. On Cyclic Sequence Type for Constructing Cyclic Schedules. Zeitschrift für Operations Research, Vol. 29, N°1, pp. 1-30, 1985.
6. F. Eisenbrand, N. Hähnle, M. Niemeier, M. Skutella, J. Verschae, A. Wiese. Scheduling periodic tasks in a hard real-time environment. In: Automata, languages and programming, pp. 299-311, 2010.
7. F. Eisenbrand, K. Kesavan, R.S. Mattikalli, M. Niemeier, A.W. Nordsieck, M. Skutella, J. Verschae, A. Wiese. Solving an Avionics Real-Time Scheduling Problem by Advanced IP-Methods. ESA 2010, pp. 11-22, 2010.
8. K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of period and sporadic tasks. In: Proceedings of the 12^{th} IEEE Symposium on Real-Time Systems, pp. 129-139, 1991.
9. J. Korst, E. Aarts, J.K. Lenstra, J. Wessels. Periodic multiprocessor scheduling. In: E.H.L. Aarts, M. Rem, J. van Leeuwen (eds.) PARLE 1991. LNCS, Vol. 505, pp. 166-178. Springer, Heidelberg, 1991.
10. J. Korst. Periodic multiprocessors scheduling. PhD thesis, Eindhoven university of technology, Eindhoven, the Netherlands, 1992.
11. C.L. Liu, J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM, Vol. 20, N°1, pp. 46-61, 1973.
12. N. Megiddo. Linear-Time Algorithms for Linear Programming in R^3 and Related Problems. SIAM J. Comput., Vol. 12, N°4, pp.759-776, 1983.
13. C. Pira and C. Artigues. An efficient best response heuristic for a non-preemptive strictly periodic scheduling problem. Technical Report 12668 LAAS-CNRS, Toulouse, 2012.
14. P. Serafini, W. Ukovich. A Mathematical Model for Periodic Scheduling Problems. SIAM J. Disc. MATH, Vol. 2, N°4, pp. 550-581, 1989.