Energy Efficient Scheduling Algorithms for Problems with Relative Deadlines

Filip Krejci
Czech Technical University in Prague
DCE FEE, Karlovo namesti 13
121 35 Prague 2, Czech Republic
krejci.filip@gmail.com

Zdenek Hanzalek
Czech Technical University in Prague
DCE FEE, Karlovo namesti 13
121 35 Prague 2, Czech Republic
hanzalek@fel.cvut.cz

Abstract

Energy efficiency and invulnerability of the timing properties are two important issues spanning over distributed embedded systems. This paper deals with non-preemptive time-triggered scheduling problem with temporal constraints and the Energy Efficiency criterion. The temporal constraints allow to model end-to-end deadlines, release dates, absolute deadlines and synchronization of tasks that are executed on dedicated resources (processors and networks).

The scheduling problem, classified as RCPS/TC, is solved by three different algorithms: Particle Swarm Optimization (PSO) with original repair function, Genetic Algorithm (GA) and GPSO as a combination of PSO and Genetic Algorithm. Performance of the three algorithms PSO, GPSO and GA with various configurations are compared and evaluated on computational experiments.

1. Introduction

Energy Efficiency (EE) is an important issue, since embedded systems also operate in environments which only provide access to power sources whose capacity is limited. At the same time embedded systems are deployed for monitoring and controlling processes which are part of safety-critical applications where guaranteeing the invulnerability of the timing properties of the systems is of major importance.

Therefore, we are faced with an optimization problem, since the guarantee of timing properties is often in contrast with energy efficiency. Unfortunately, such optimization problem is difficult to decompose up to physical resources (i.e. processors and eventually network segments) due to the aspects spanning several resources. These (non-disjoint) aspects are the following:

- Timing properties: Guarantee of the worst-case end-to-end response time for safety critical applications usually leads to sparse schedules, in order to cover the worst case situations.
- Energy efficiency: In contrast to the previous one, energy efficiency leads to dense schedules, in order to save energy while entering sleep mode.

Power-aware processors have built-in support for active, idle and sleep modes. In sleep mode, substantially more energy savings can be obtained but it requires significant amount of time to switch into and out of that mode [13]. Given (1) the set of resources with this energy model, (2) the set of non-preemptive tasks to be executed on these resources and (3) timing constraints relating start-times of the tasks, we aim at finding feasible schedule which minimizes energy consumption.

GA [1, 2, 10, 14, 15] and PSO algorithms [3, 4, 6, 7] are often used in the project scheduling. In the case of temporal constraints, they are not used as often [14, 8, 9], as in the case of simple precedence relations [1, 2, 6] or with earliness-tardiness criterion [4, 7]. Schedule makespan C_{max} [1, 2, 3, 6] is the most often used criterion.

The paper is organized as follows: In the next section, the problem under consideration is formulated as Resource Constrained Project Scheduling with Temporal Constraints (RCPS/TC) including the Energy Efficiency criterion. In the following section the Particle Swarm Optimization (PSO) with our repair function is shown for RCPS/TC. Further a GPSO as a combination of PSO and Genetic Algorithm (GA) is introduced. The three algorithms PSO, GPSO and GA with various configurations are compared and evaluated on the test instances while assuming EE criterion and Earliness/Tardiness criterion.

2. Problem statement

Traditional off-line scheduling problems typically assume that deadlines are absolute, i.e. the deadlines are related to the beginning of the schedule. On the other hand, when assuming, for example, the required end-to-end deadline in the real-time system, a given timing requirement may be represented conveniently by a deadline of task Tj related to the start time of task Ti. In such a case, the absolute deadlines cannot be calculated a priori, and therefore we use “relative” deadlines.

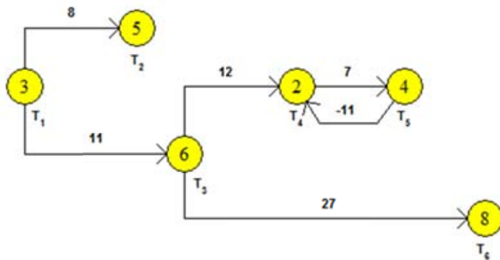
The concept covering such requirements, called (generalized) temporal constraints is covered by RCPS/TC where the set of n tasks $T = \{T1, T2, ..., Tn\}$ with temporal constraints is given by a task-on-vertex graph. Task Ti is represented by vertex Ti in the graph and has a non-negative duration corresponding to p_i, the processing time of the task. Often two dummy tasks (task zero and task n+1) are added. Task zero precedes all other tasks and task n+1 follows all other taks.

Temporal constraints among the vertices are represented by the set of directed edges. An edge from vertex Ti to vertex Tj is labeled by an integer weight w_{ij}, which constrains start times s_i and s_j by the inequality

$s_j - s_i \geq w_{ij}$.

There are two kinds of edges: the edges with positive weights and the edges with negative weights. The edge with positive weight w_{ij}, giving the minimum time lag, indicates that s_j, the start time of Tj, must be at least w_{ij} time units after s_i, the start time of Ti. The edge from vertex Tj to vertex Ti with a negative weight w_{ji} (giving the maximum time lag) indicates that s_j must be at most w_{ji} time units after s_i. Therefore, each negative weight w_{ji} represents a relative deadline of Tj depending on s_i.

For example an edge from task zero (scheduled at time 0) to task Tj with positive weight w_{0j} may be used to encode the release date (also called offset) of task Ti. An edge with negative w_{j0} from task Tj to task zero may be used to encode absolute deadline of task Tj with value $w_{j0}+p_j$.



$$w = \begin{pmatrix} -\infty & 8 & 11 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 12 & -\infty & 27 \\ -\infty & -\infty & -\infty & -\infty & 7 & -\infty \\ -\infty & -\infty & -\infty & -11 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \end{pmatrix}$$

$$p' = (3 \quad 5 \quad 6 \quad 2 \quad 4 \quad 8)$$

$$c' = (1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1)$$

$$r = (1)$$

Figure 1.The example problem consists of n = 6 tasks to be processed on a single resource with a unary capacity. The problem is drawn in the graph above and represented by the input data at the bottom.

Tasks are to be scheduled on set of dedicated resources $R = \{1, ..., m\}$. Each resource has capacity r_i units represented by a positive integer. Each task has resource demand c_{ik} given for each resource $k \in R$, therefore a task can occupy several units of several resources. An example problem on one resource with one unit is shown on Figure 1.

Start times vector s which satisfies all temporal and resource constraints represents a feasible solution of the problem.

2.1. Scheduling criteria

This article focuses on the Energy Efficiency while assuming the following model of resources: All devices start in the sleep mode. Before processing a task, the device must wake up. After processing the task, the device can process another task or it can become idle or after some time it can switch back into the sleep mode. In the idle mode, as well as during wake-up or suspension, the device consumes energy (in this paper we assume 1 energy unit per one time unit) and does not perform any processing. In the sleep mode, the device consumes far less energy (0.001 per time unit) but suspension takes 3 units and wake-up takes 7 units. The energy consumed during the processing of a task is constant independently on the schedule, so it is not a part of the optimization criterion. The state machine representing the energy consumption of a device in different modes is shown on Figure 2 (specific energy consumption values are based on [13]). Energy Efficiency (EE) criterion is the sum of energy consumed by all units of all resources during idle mode, sleep mode, wake-up time and suspension time. Being energy-efficient means to group the tasks into blocks with minimum idle time and to have sufficient time to suspend the device into the sleep mode between the blocks. Significant amount of energy is lost due to the idle gaps among the tasks that are shorter than the time required to enter the sleep mode.

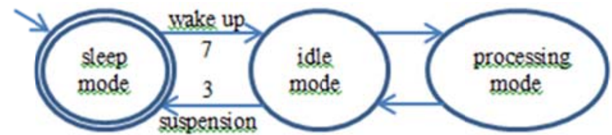


Figure 2. A state machine representing energy consumption per time unit in different modes and during the transitions.

Let's take a look to our example problem. One feasible solution, represented by vector s = (0, 8, 12, 24, 31, 39), is shown on Figure 3A with the EE criterion value of 30.

A better solution can be found: Instead of scheduling task T2, we lengthen the idle time after T1 and schedule task T3 before T2. In our case the local “suboptimality”

is traded for a global improvement (see Figure 3B). The EE criterion has a value of 29.

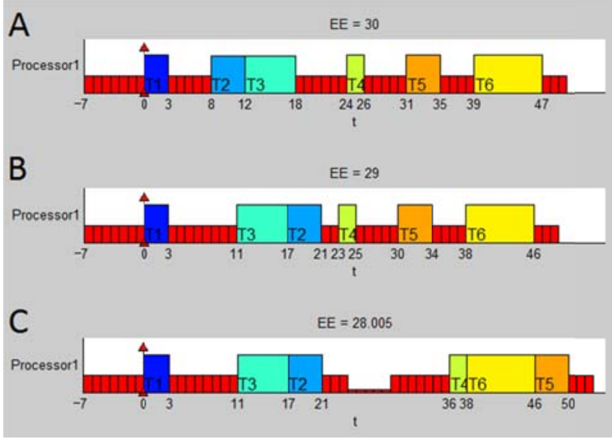


Figure 3.Three solutions of the example problem shown in Figure 1. Red color rectangles are used to display the time units assumed in the EE criterion.

The best solution to this problem is to delay task T4 to be processed just before task T6. Task T5 is thus processed after task T6. This extends the makespan by 4 but between tasks T2 and T4 there is sufficient time to switch into the sleep mode for 5 units and save more energy (see Figure 3C). In this case, the EE criterion has a value of 28.005.

In addition to the EE criterion, which is the main objective of this paper, we test our scheduling algorithms on another non-trivial criterion: the earliness-tardiness, where each task Ti has a specified due date d_i. The tardiness penalty ω_i corresponds to the late completion of task Ti. The earliness penalty (may correspond to the storage cost of the data produced by task Ti) is $\lambda_i \cdot \omega_i$, where λ_i determines the earliness/tardiness ratio of task Ti. In our case $\lambda_i = 0.1$ for all tasks.

The cost of each task grows linearly with distance between the completion time and the due date. Earliness-tardiness value of given schedule s is the following:

$$ET(s) = \sum_{i \in 1..n} \left[\omega_i \cdot \max(s_i + p_i - d_i, \lambda_i \cdot (d_i - (s_i + p_i))) \right]$$

3. Description of the algorithms

The Particle Swarm Optimization algorithm (PSO) belongs to the population based evolutionary algorithms, which simulate behavior of animals in nature. These algorithms perform parallel local optimization from number of positions in multidimensional search space. Parallel optimization allows the exchange of information about the search space in the population.

3.1. Basic PSO

PSO maintains a population of particles (individuals). Particles in the swarm have relationship, which determines the set of neighbors for each particle. Cardinality and structure of these sets have significant influence on the behavior of the algorithm. It controls the interchange rate of information across the population and allows balance of exploration and exploitation abilities. Neighborhood relationship is created in the initialization phase and remains unchanged during the algorithm execution.

Each particle i has its position in the search space $x_i = (x_{i1}, x_{i2}, ..., x_{in})$ and its velocity $v_i = (v_{i1}, v_{i2}, ..., v_{in})$. There is a criterion function f, which assigns a value to each vector x_i. Particles have also a memory while storing particle-best and neighborhood-best positions. Particle-best position b_i is the best vector x_i found by particle i so far. Neighborhood-best position g_i is the best vector x_j found up to now by any particle j from the neighborhood of i. Particle position x_i is changing over time and its new position is affected by the current position and vectors particle-best and neighborhood-best.

While assuming position x_i to represent the start times of n tasks, the PSO algorithm adapted to RCPS/TC problem is as follows:

1 **for** each particle i **do**
2 initialize (x_i)
3 initialize (v_i)
4 $b_i = x_i$
5 **end**
6 **for** each particle i **do**
7 g_i = argmin (f ($p_{neighborhood(i)}$))
8 **end**
9 **for** 1 **to** number of movements **do**
10 **for** each particle i **do**
11 $vNew_i = \varphi_0 \cdot v_i + \varphi_1 \cdot rand_1 \cdot (b_i - x_i) + \varphi_2 \cdot rand_2 \cdot (g_i - x_i)$
12 $vNew_i$ = limit_velocity($vNew_i$,longest)
13 $xNew_i = x_i + vNew_i$
14 s_i = fix_start_times (round ($xNew_i$))
15 **if** (s_i != **null**) **then**
16 $x_i = s_i$
17 $v_i = vNew_i$
18 **if** f (s_i) < f (b_i) **then**
19 $b_i = s_i$
20 **end**
21 **end**
22 **end**
23 **for** each particle i **do**
24 g_i = argmin (f ($p_{neighborhood(i)}$))
25 **end**
26 **end**

Lines 1-8 initialize the population. The initial particle position is found as feasible schedule by IRS heuristic [5]. It is possible to obtain up to 4 different vectors due to time inversion and due to the division of the timing instance parameters by 2. These vectors represent diverse and high quality solutions with respect to the C_{max} criterion. The matrix of particle velocities v is set randomly. The main loop is performed up to the desired number of particle's movements (calculated as floor(4004/population_size)).

Movement of particle is carried out in 4 steps. In the first one new candidate velocity vector $vNew_i$ is calculated (line 11). It is given a weighted sum of following three entries:

- inertia $\varphi_0 \cdot v_i$ of the movement, where $0 \leq \varphi_0 \leq 1$,
- randomly weighted distance from the particle-best position $\varphi_1 \cdot rand_1 \cdot (b_i - x_i)$,
- randomly weighted distance from the neighborhood-best position $\varphi_2 \cdot rand_2 \cdot (g_i - x_i)$,

$rand_1$ and $rand_2$ are vectors of random numbers chosen from the interval (0, 1), and for φ_0, φ_1, φ_2 we started with recommended [6] values of $\varphi_0 = 0.729844$, $\varphi_1 = \varphi_2 = 1.49618$. The value of some entry of the new velocity $vNew_i$ is reduced, whenever it exceeds 10 % of the longest schedule (line 12).

In the second step new candidate position $xNew_i$ is computed (line 13). Next step is to run the repair procedure, whose goal is to create feasible schedule based upon $xNew_i$. If the procedure succeeds, the update of particle's position x_i and its velocity v_i is performed as a final step of the movement (lines 16-17). If the criterion value of new position x_i is better than the one of the particle-best position b_i, the update of particle-best position is made (line 19). After evaluation of all particles, an update of neighborhood-best positions is carried out.

3.2. Schedule repair function

It is often the case that a new position of the particle $xNew_i$ violates the temporal constraints or resource constraints of the scheduling problem. That is the reason for a repair function fix_start_times with unscheduling step. Input parameter of the function is vector $xNew_i$, which may violate some constraints. First of all, the function first corrects evident violations of the positive temporal constraints and then for each task it finds a new start time, which is as small as possible but not less than the one in the input vector. The result is the vector satisfying all the constraints, or null if the reparation failed. The function was inspired by [14] and after modifications it looks as follows:

1 **function fix_start_times** ($xNew_i$)
2 initialize_resources ()
3 *unscheduleAvailable* = *n*
4 *negativeError* [1 **to** *n*] = 0
5 *idxs* = order_tasks_up_to_start_times ($xNew_i$)
6 *idxs* = repair_indexes (*idxs*)
7 *counter* = 1
8 **while** *counter* ≤ *n* **do**
9 *task* = *idxs* [*counter*]
10 *earliestStart* = max (earliest start time determined by temporal constraints from already scheduled tasks, $xNew_i$ [*task*], *negativeError* [*task*], earliest possible place for task *task* on resources)
11 **if** scheduling of *task* at *earliestStart* violates a negative temporal constraint **then**
12 **if** *unscheduleAvailable* ≤ 0 **then**
13 **return null**
14 **else**
15 *unscheduleAvailable* – –
16 **end**
17 *negTask* = select the task with the lowest start time participating on violated negative constraint
18 *delay* = *earliestStart* + w[*task*][*negTask*]
19 *negativeErr* (*negTask*) = *delay*
20 unschedule all tasks up to *negTask* inclusive
21 *counter* = *counter* – number of unscheduled tasks
22 **else**
23 s_i [*task*] = *earliestStart*
24 place *task* to resources at *earliestStart*
25 *counter* + +
26 **end**
27 **end**
28 **return** s_i
29**end**

The function first initializes the resources and sets the maximum number of unscheduling steps, which is equal to the number of tasks (recommended in [14]). Then the tasks are ordered according to the increasing start times. In the case of equality of start times the task order is determined randomly. Function *repair_indexes* (line 6) adjusts this order to satisfy all positive temporal constraints.

The main loop determines start times for each task so that negative temporal constraints and resource constraints are fulfilled. The task processing order is set by function *repair_indexes* and doesn't change during computation. Start time of the processed task is set as early as possible and satisfies the following constraints (line 10):

- all positive temporal constraints from the currently scheduled tasks are met,
- capacity of resources allows for the task processing,

- it must be greater than or equal to the desired value in the input vector (the task is not always scheduled immediately when allowed by temporal and resource constraints, since the optimal start time of this task can be different),
- it must be greater than or equal to the minimum time that requires some negative temporal constraint from another task which could not be scheduled (caused by unscheduling step).

For such designated start time we verify validity of all negative constraints from this task to all other scheduled tasks (line 11). If no corruption is detected, the task is scheduled and the next task is processed (lines 23-25). When some negative constraint is violated and some unscheduling cycles are available, the earliest scheduled task which violates constraint is selected (line 17). For this task we calculate its earliest (delayed) start time which meets the temporal constraints (lines 18-19). All tasks up to the delayed one are unscheduled (line 20) and algorithm continues from the delayed task. The complexity of the function is $O(n^3)$.

3.3. Improvements of PSO

After performing our research with the PSO and GA, we created a new algorithm Genetic PSO (GPSO) combining both of them. Algorithm uses the basic scheme of PSO (particle-best and neighborhood-best positions, the same number of updates for each particle), but instead of classical velocity update equation it performs genetic operations, mutation and crossover, with three vectors x_i, particle-best b_i and neighborhood-best g_i.

The performance of the three algorithms is very sensitive to the appropriate parameter settings (the same usually holds for other evolutionary algorithms). To find the optimal settings we used two test sets each of 50 instances and EE criterion. The first set contained various instances with 30 tasks and the second was crucial and contained instances with 100 tasks.

We examined the tradeoff between the number of movements and the number of individuals with the product of these number less than or equal to 4004. We focused on high quality but little diversified initial population and compared this setting with more diversified population of lower average quality. To reduce exploration ability of the algorithm in favor of the exploitation ability, we tried a particle update modification. The new particle position $xNew_i$ was accepted only when its criterion value was greater than the criterion of its current position x_i.

After our extensive tests the following parameter settings for each of the algorithms were selected:

- PSO* – 63 particles with 63 movements, high-quality initial population of particles with randomly initialized velocity, $\varphi_0 = 0.25$, $\varphi_1 = \varphi_2 = 1.50$.
- GPSO* – 91 particles evolving for 44 movements, high-quality initial population, global-best topology, two-point unbalanced crossover, unbalanced block mutation in the range of $[-2/3 \cdot p_ø; 1/3 \cdot p_ø]$ and the probability of 0.1.
- GA* – Steady-state model with 63 particles, high-quality initial population, 2-tournament selection, 1-point crossover with probability 0.75, unbalanced block mutation in range of $[-4/3 \cdot p_ø; 2/3 \cdot p_ø]$ and the probability of 0.1.

4. Experimental results

In this section we compare the performance of PSO*, GPSO* and GA* algorithms described in previous section. Up to our knowledge, there are no standard benchmarks for RCPS/TC with EE or ET criterion, hence the test sets with randomly generated instances have been used. Experiments were performed on a PC with AMD Athlon II X4 at 2.80 Ghz with 4 GB of RAM. Algorithms were implemented in Matlab R2009a.

4.1. Test sets description

RCPS/TC includes very different types of problems. There are two essential characteristics that can be applied to divide the problems: presence of parallel identical processors (resources with capacity greater than 1) and presence of multiprocessor tasks. To identify the type of problem, we used two-letter notation. If the problem contains parallel identical processors, the first letter is P otherwise X. If the problem contains multiprocessor tasks, the second letter is M otherwise X. So we have 4 groups of problems denoted as XX, PX, XM and PM.

Test sets of instances can be described by a string in the format A_B_nC, where A specifies the number of instances in the set; B determines the type of problems (XX, PX, XM, PM, MIX) and C the number of tasks. Type MIX represents the case, when the problems of all 4 types are equally frequent in the set. For example the test set 50_PM_n100 consists of 50 instances with parallel identical processors and 100 multiprocessor tasks.

4.2. The main test results

All three algorithms were tested on 1800 instances divided into 8 groups (2 different criteria and 4 types of problems). Each group contained 50 instances of n = 30, n = 100, n = 200, n = 300 and 25 instances of n = 500.

Group	Algorithm	n = 30						n = 100						n = 200						n = 300						n = 500					
		impr	no im	best	valid	cmax	time	impr	no im	best	valid	cmax	time	impr	no im	best	valid	cmax	time	impr	no im	best	valid	cmax	time	impr	no im	best	valid	cmax	time
EE, XX	PSO*	11.28	0	26	94.9	10.0	21	11.06	0	2	51.0	26.8	131	5.43	6	0	8.2	17.7	571	3.60	42	0	3.1	8.8	1235	0.82	80	0	0.3	2.1	4052
	GPSO*	11.14	0	34	98.2	1.4	24	20.24	0	36	88.5	10.7	80	25.42	0	40	73.2	7.4	286	26.36	0	40	60.2	6.7	663	26.09	0	68	41.0	5.5	2486
	GA*	10.95	0	48	97.3	3.6	22	22.37	0	62	87.4	11.5	78	25.42	0	60	68.9	11.9	277	26.84	0	60	56.4	13.3	662	23.31	0	32	34.0	11.8	2311
EE, MX	PSO*	11.16	0	30	76.3	13.1	35	2.67	16	0	5.2	6.8	232	0.07	94	0	0.0	0.6	663	0	100	0	0.0	0	1298	0	100	4	0.0	0	3014
	GPSO*	12.07	0	38	95.4	2.9	34	14.63	0	52	70.6	2.8	145	9.44	0	76	41.6	2.5	518	7.02	0	88	25.1	1.5	1114	3.81	4	100	12.7	0.5	2911
	GA*	11.49	0	32	93.9	4.5	32	14.46	0	48	67.1	2.4	146	7.83	0	24	36.8	4.1	528	4.28	2	12	15.8	2.1	1120	1.73	20	4	5.4	0.7	2917
EE, XP	PSO*	3.24	0	46	99.4	3.1	20	5.32	0	20	90.7	8.4	48	2.82	0	0	67.5	4.9	336	2.55	10	0	49.8	3.3	836	0.29	56	0	12.6	0.9	3730
	GPSO*	2.82	0	58	99.9	0.9	24	6.77	0	26	97.9	1.8	66	9.58	0	60	95.1	1.3	167	11.42	0	76	90.8	1.0	332	14.50	0	68	77.6	1.5	1321
	GA*	2.79	0	58	99.5	0.7	22	7.74	0	54	96.7	3.4	66	8.83	0	40	90.6	3.0	196	9.39	0	24	84.9	2.6	411	12.00	4	32	64.0	3.5	2007
EE, MP	PSO*	5.03	0	28	97.7	6.4	28	3.17	8	2	39.2	6.4	216	0.53	64	0	7.8	1.2	791	0	100	0	1.1	0	1617	0	100	8	0	0	4326
	GPSO*	5.45	0	34	99.0	0.7	32	9.33	0	50	89.0	0.8	116	9.30	0	62	73.3	1.7	395	8.41	0	82	66.9	1.8	749	5.07	8	96	33.2	1.7	2810
	GA*	5.51	0	38	98.0	1.8	32	8.68	0	48	83.7	3.0	125	7.81	4	38	64.5	2.2	454	6.15	16	18	51.8	2.7	944	2.60	48	12	16.9	1.5	3290
ET, XX	PSO*	5.68	0	76	96.0	2.0	18	0.88	0	28	48.9	0.1	142	0.14	48	0	6.9	0.1	662	0.00	98	0	0.5	0	1390	0	100	0	0.0	0	4400
	GPSO*	4.13	0	18	98.9	0.2	22	1.71	0	46	90.4	-0.3	74	1.55	0	80	74.2	-0.6	275	1.10	0	80	58.5	-0.9	740	0.60	0	96	42.1	-0.6	2696
	GA*	3.76	0	6	98.0	0.3	20	1.47	0	26	85.5	0.1	81	1.14	0	20	62.0	-0.4	330	0.81	14	20	40.8	-0.4	981	0.29	40	4	19.5	-0.3	4123
ET, MX	PSO*	5.36	0	46	80.9	0.3	28	0.40	46	0	5.1	-0.2	235	0	100	0	0.1	0	702	0	100	2	0.0	0	1434	0	100	4	0.0	0	3049
	GPSO*	7.01	0	44	96.0	-0.9	27	6.21	0	56	72.6	-3.2	131	3.81	0	80	34.0	-2.1	508	2.79	2	96	25.0	-1.7	1199	2.27	4	96	15.3	-1.6	3036
	GA*	4.62	0	10	94.4	-0.5	26	5.36	0	44	69.7	-3.2	129	2.93	0	20	36.4	-1.6	524	1.92	12	6	15.5	-1.3	1289	1.29	44	8	6.7	-0.9	3224
ET, XP	PSO*	3.84	0	76	99.2	1.3	17	1.69	0	78	92.9	0.0	58	0.73	0	26	68.7	0.0	322	0.34	6	24	52.0	0.0	825	0.04	56	12	10.6	0	3957
	GPSO*	2.65	0	16	99.9	0.2	21	1.31	0	16	98.2	0.0	63	0.98	0	54	94.2	-0.1	176	1.00	0	58	90.9	-0.5	359	0.30	4	76	76.2	-0.1	1550
	GA*	2.17	0	8	99.6	0.5	19	0.98	0	6	96.7	0.0	72	0.78	0	20	89.5	-0.1	198	0.56	4	18	81.9	-0.3	497	0.16	32	20	55.5	-0.0	2768
ET, MP	PSO*	4.14	0	72	97.4	0.5	22	0.99	4	6	44.7	-0.1	192	0.06	72	0	9.1	-0.0	770	0.00	98	6	0.8	0	1684	0	100	32	0.0	0	4321
	GPSO*	3.36	0	20	99.6	-0.6	25	3.75	0	62	90.8	-1.1	99	1.49	0	78	74.3	-0.3	416	1.49	6	96	64.9	-0.7	806	0.45	32	96	33.0	-0.0	3037
	GA*	2.60	0	8	98.7	-0.2	24	2.73	0	32	86.0	-0.9	106	1.05	4	22	64.3	-0.4	474	0.51	28	10	48.8	-0.4	1022	0.17	80	36	17.7	0.0	3497

Table 1 Test results for all 8 groups of problems.

We started all algorithms with the same initial population for the selected instance and we evaluated the following parameters:

- improvement (*impr*) as one minus the ratio of the best solution found by the algorithm to the best solution in the initial population,
- instances with no improvement realized (*noim*),
- which of the algorithms found the best solution to the problem (*best*),
- proportion of feasible individuals after execution of repair function *fix_start_times* to all generated individuals (*valid*),
- the C_{max} change of the best schedule found (*cmax*),
- elapsed time (*time*).

Time was measured in seconds; the other values are in percents. Test results can be found in Table 3.

The results show that the most difficult are groups including multiprocessor tasks (MX, MP). The improvement is highly dependent on the algorithm ability to generate feasible individuals (this ability decreases most quickly for MX).

For the instances of n = 30 is PSO* the best algorithm. While looking at the column *cmax* we find that the schedules generated by PSO* are significantly longer than those generated by other algorithms. It can be concluded that the algorithm performs larger update operation (performed more explorations) which for larger instances greatly reduces the ability to find a feasible individual (already for MX_n100 only 5.2% of generated individual are feasible).

GPSO* proves to be the best and in most cases outperforms GA* and PSO*. On the largest instances it had no competitors. On the MX_n500 instances we can observe difficulties to generate feasible individuals.

While looking more closely at EE criterion, we see that the improvement increases with the growing size of the instance, since the larger instance provides more space for optimization of Energy Efficiency. On the other hand the complexity of the problem grows with greater instances. So the ability to generate feasible individuals decreases as well as the improvement. The best results achieves GPSO* for XX problems on the set with n = 300 (26.36%), for MX on n = 100 (14.63%), for XP on the n = 500 (14.50%) and for MP on n = 100 (9.33%).

GPSO* (and also GA*) does not extend C_{max} too much. Intuitive explanation is that the energy-efficient schedule is not much longer than the schedule created for the C_{max} criterion.

The computation of an n = 500 instance by our GPSO* algorithm typically lasts 20-50 minutes. This time can be reduced (in order of magnitude) by selecting more suitable implementation environment. The most demanding operations measured on MIX_n200 are the repair function *fix_start_times* (69%), the creation of initial population (14%), the EE criterion function computation (13%) and other operations such as crossover and mutation (4%).

Improvements made on the criterion of ET were significantly lower than on EE. With the increasing size of the problems the relative improvement decreases, but this is mainly due to the randomness of due date generation which does not take properly into account the length of the schedule.

GPSO* proves the best even in ET. GPSO* also generated significantly more of the best schedules than the rest of algorithms.

Surprisingly, GPSO* algorithm achieves the greatest improvement on MX-type problems (the most difficult group). The improvements are (in order from n = 30 to 500): 7.01%, 6.21%, 3.81%, 2.79% and 2.27%. The smallest improvement was achieved on XP problems.

5. Conclusion

This paper presents a new criterion of energy efficiency for Resource Constrained Project Scheduling with Time Constraints. The basic idea is to minimize the energy consumed to run the devices. Being energy-efficient means to group tasks into blocks with minimum idle time and having sufficient time to suspend the device into sleep mode between the blocks.

We created algorithm GPSO combining the good features of PSO with the unbalanced two-point crossover and random mutation operators used in GA. We tested a number of modifications and various settings of our PSO, GPSO and GA algorithms. We tested the best configuration of each algorithm on 1800 randomly generated instances of 4 different types and 2 criteria (Energy Efficiency and Earliness-Tardiness). The largest instances consisted of 500 tasks.

The best results were achieved by GPSO algorithm. On some test sets it improved energy efficiency by more than 25%. For earliness-tardiness criterion, GPSO achieved an average improvement up to 6.2% simultaneously with shortening the average schedule length up to 3.2%.

References

[1] Franco, E. G. , Zurita, F. L. T., Delgadillo, G. M. A, “Genetic Algorithm for the Resource Constrained Project Scheduling Problem (RCPSP)”, 19th International Conference on Production Research, ICPR. 2007.

[2] Hartmann, S. A , “Competitive Genetic Algorithm for Resource-Constrained Project Scheduling”, Naval Research Logistics Vol 45, pp. 733-750, 1998.

[3] J. Czogalla and A. Fink, "Particle Swarm Topologies for Resource Constrained Project Scheduling", in Proc. NICSO, 2008, pp.61-73.

[4] Parsopoulos, K.E., Vrahatis, M.N., Studying the Performance of Unified Particle Swarm Optimization on the Single Machine Total Weighted Tardiness Problem, Lecture Notes in Artificial Intelligence (LNAI), Vol. 4304, pp. 760-769, 2006, Springer.

[5] Hanzálek, Z. and Šůcha, P., Time symmetry of project scheduling with time windows and take-give resources. 4th Multidisciplinary International Scheduling Conference: Theory and Applications, MISTA. Dublin. August, 2009.

[6] Wang, Q. and Qi, J. Improved Particle Swarm Optimization for RCP Scheduling Problem. Advances in Intelligent and Soft Computing, 2009, Vol. 56, 2009, pp. 49-57.

[7] Anghinolfi, D. and Paolucci M. A new discrete particle swarm optimization approach for the single-machine total weighted tardiness scheduling problem with sequence-dependent setup times. European Journal of Operational Research, Vol. 193, Issue 1, 2009, pp. 73–85.

[8] Policella, N. – Cesta, A. – Oddi, A. – Smith, S. F. From Precedence Constraint Posting to Partial Order Schedules: A CSP approach to Robust Scheduling. AI Communications - Constraint Programming for Planning and Scheduling, Vol. 20 Issue 3, August 2007.

[9] Lombardi, M. – Milano, M. A Precedence Constraint Posting Approach for the RCPSP with Time Lags and Variable Durations. Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, 2009.

[10] Van Peteghem, V. and Vanhoucke, M., ”A Genetic Algorithm for the Preemptive and Non-preemptive Multi-mode Resource-Constrained Project Scheduling Problem”, European Journal of Operational Research, 201, 409-418, 2010.

[11] Anghinolfi D., A. Boccalatte, Paolucci M., Vecchiola C., “Performance evaluation of an adaptive ant colony optimization applied to single machine scheduling”, in X. Li et al. (Eds.): Simulation Evolution and Learning, LNCS, Springer Verlag, 411-420, Vol. 5361, 2008.

[12] Clerc, M. and Kennedy, J., “The Particle Swarm–Explosion, Stability, and Convergence in Multidimensional Complex Space”. IEEE Trans. Evol. Comput., vol. 6, pp. 58–73, Feb. 2002.

[13] Rowe, A., Lakshmanan, K., Zhu, H., Rajkumar, R. “Rate-Harmonized Scheduling for Saving Energy”, RTSS '08 Proceedings of the 2008 Real-Time Systems Symposium.

[14] Franck, B. – Neumann, K. – Schwindt, Ch., “Truncated branch-and-bound, schedule-construction, and schedule-improvement procedures for resource-constrained project scheduling”, OR Spectrum, Vol. 23, 297-324. 2001.

[15] Van Peteghem, V. and Vanhoucke, M., ”Using resource scarceness characteristics to solve the multi-mode resource-constrained project scheduling problem”, Journal of Heuristics, Vol. 17, pp. 705-728, 2011.