Extremely fast heuristic event-driven job shop scheduler with a new class of extended petri nets

Accepted 7th June 2019

ABSTRACT

A heuristic job shop scheduler which is implemented with a new class of extended Petri nets is proposed. The used Petri nets outlined in the paper attributed tokens and represent the complete algorithmic system. The scheduler creates a schedule sequentially without any form of enumerative search. Its structure and Petri net scheme were also explained. The scheduler includes models of machines and runs in the environment of WINSIM system which implements the used class of Petri nets. With the scheduler, two groups of experiments were conducted using three different greedy strategies in each group. In the first group, the scheduler was tested on single published benchmark data instances of $10 \times 10$ and $15 \times 15$ scheduling tasks. In the second group, $10 \times 10$, $15 \times 15$ and $20 \times 20$ scheduling tasks were investigated with a series of 1000 generated random task instances each to obtain statistics on average makespan, its standard deviation, average loads of machines and relative unbalance of machine loads. It was observed that for each of the studied problems, the developed scheduler runs hundreds or thousands of times faster than any known job shop scheduling method due to a unique mechanism of monitoring events in the active Petri net.

Key words: Event-driven job shop scheduler, extended Petri nets, scheduling experiments, Petri-net tool, WINSIM.

INTRODUCTION

Job shop scheduling, being an important class of scheduling problem in general is a topic of numerous publications starting at 50’s of the previous century, with non-decreasing number of works from the beginning of this century. In its deterministic form, this problem can be stated as follows (Jain and Meeran, 1999). Consider two finite sets of jobs and machines given as:

$$J = \{J_1, J_2, ..., J_n\}, M = \{M_1, M_2, ..., M_m\},$$

(1)

Where $n$ and $m$ are the numbers of jobs and machines, respectively. Assume that each job $J_i$ consists of an ordered sequence of steps, or operations:

$$S_i = [S_{i_1}^{k_1}, S_{i_2}^{k_2}, ..., S_{i_m}^{k_m}], \quad i = 1, 2, ..., n,$$

(2)

Where $S_{i_r}^{k_r}$ is the $i_r$th step of job $i$ to be processed by machine $k$, $k, r \in \{1, 2, ..., m\}$, with $i \in \{1, 2, ..., n\}$. It is assumed also that the number of steps of each job is equal to the number of machines, and each step of each job should be processed on one and only one pre-assigned machine. It further assumed that for each job step, there is an associated time to process the step on each machine.

As shown, the sequence of step (2) is in general different for different jobs. It is assumed that a new step of each job can be started on a free machine only after finishing its previous step. In addition, any machine can be used in a mutually exclusive way by different job steps. If all steps of each job must be processed by corresponding machines, then there are $nm$ job steps in total, although in some statements of the problem some steps of some jobs may be
skipped without the use of machines. We assume that release date of all jobs is the same, that is, all jobs become ready for scheduling at the same starting time. The problem then is to create a feasible schedule for which the chosen objective function has the best value.

Quite often, job shop schedule is created with the use of integer programming technique, genetic algorithms, simulated annealing, tabu search, cutting plane schemes, timed Petri nets or combinations of these methods. As an objective function, makespan value is usually used.

The common characteristic of all these methods is that they apply some form of enumerative search in large space of possible schedules. With \( n \) jobs and \( m \) machines, the total size of this space is \( (n!)^m \). The complete search in such a space is known to be NP hard. Even if the size of search space is restricted in some ways (as is done, for example, in branch and bound method or in tabu search method), the search remains in principle enumerative and generally requires large computing time. This complicates the practical use of these methods.

In the scheduling paradigm proposed in this paper, any form of enumerative search in the space of schedules is avoided. Instead, a schedule is created dynamically and sequentially in a series of scheduling actions which are triggered by events in the Petri net that implements the scheduler. As a result, creation of the schedule requires extremely small computer time which grows polynomially with the problem size. The proposed scheduler is implemented with the use of a new class of extended Petri nets.

**A NEW CLASS OF EXTENDED PETRI NETS**

The proposed scheduler is implemented in terms of a new class of extended Petri nets with attributed tokens. This section outlines these nets. The minimal functionally complete structural element of these nets is an elementary net, which is a graph consisting of a transaction and an incident input and output places with associated procedures. Formally, an elementary net \( E(t) \) pertaining to transition \( t \) is defined as:

\[
E(t) = \langle C, P_1, P_2, r_1, r_2, d, f \rangle,
\]

Where \( C \) is a necessary (but generally not sufficient) condition to fire the transition \( t \); \( P_1 \) and \( P_2 \) are finite sets of input and output places of \( t \), with \( P_1 \cap P_2 = \emptyset \) and \( P_1 \cup P_2 \neq \emptyset \); \( r_1 \) and \( r_2 \) are functions of input place selection and output place selection, respectively; \( d \) is a delay function and \( f \) is a data transformation function. There are two types of places in these nets - simple places (S-places) and queue places (Q-places). S-place can hold one token at a time and the number of tokens in Q-place is not limited.

One can see from definition (3) that an elementary net reflects four fundamental properties of system processes: transfer of information (in the form of tokens), control, time delay and information processing. Together, elementary nets represent the functionally complete algorithmic system.

For modeling and simulation of arbitrary information systems, it is sufficient to have quite a limited number of types of elementary nets. These are types \( T, Y, X \) and \( I \). They constitute the alphabet of extended Petri nets. Connecting elementary nets with each other, it is possible to build nets of any size and complexity. The points for connection of elementary nets are places. For the purpose of this paper, only types \( T, X \) and \( Y \) are necessary.

Elementary net of type \( T \) is intended for merging of tokens at the input places and putting of tokens at the output places, along with the data transformation of token attributes and, may be, of other data items. The necessary condition \( C \) to fire the transition in this type of elementary net is the presence of a token in all input places and the absence of tokens in all output places. The emptiness is not required for output Q-places. This is also the sufficient condition for firing the transition in this net. Formally, the condition \( C \) for firing the net of this type of elementary is as follows:

\[
C = B(x_1) \land ... \land B(x_m) \land \neg(B(y_1) \lor ... \lor B(y_j)),
\]

\[
r_1 = \{\text{TRUE} \rightarrow \{x_1, ..., x_m\}\},
\]

\[
r_2 = \{\text{TRUE} \rightarrow \{y_1, ..., y_j\}\},
\]

\[
M'(x) = M(x) - 1, M'(y) = M(y) + 1, i = 1, 2, ..., m, j = 1, 2, ..., n,
\]

\[
v_i(y) = v_i(x_i)
\]

(4)

Where \( i = 1, 2, ..., k = \min(s(x_i), s(y_j)), v_i(y_j) = 0, i = k + 1, ..., s(y_j), x_i \) and \( y_j \) are identifiers of input and output places, \( v_i(x_i) \) and \( v_i(y_j) \) are attribute values of token in places \( x_i \) and \( y_j \); \( B(x) \) is the predicate that has the true value if place \( x \) contains a token; \( r_1, r_2 \) are default control functions, and \( M \) and \( M' \) represent initial and target markings. If some or all output places \( y_j \) are Q-places, then, the corresponding predicate \( B(y_j) \) is excluded from the logical expression for condition \( C \). That is, the necessary condition does not require that the output Q-place is empty. This is true for all types of elementary nets.

Logically, an elementary net of type \( T \) represents a simple timed data processing block with the additional possibility to synchronize tokens that arrive at its input places and to produce copies of a token at its output places from one of its input place. Figure 1 illustrates the change of marking of this type of elementary net as a result of firing of its transition.

Elementary net of type \( X \) provides a conditional routing of tokens from input places to one of its output places. The transition bar of this elementary net has a short horizontal line segment that is directed to output places of the net. The necessary (but generally not sufficient) condition \( C \) for the transition to fire requires that each input place of the net has a token and at least one output place allows a token to be put into it (this may be an empty S-place or any Q-place). If the condition \( C \) is true, then, the function of output
The new function $\text{TRUE}$ fires. At the end of its activity interval, a token is removed from each input place and one token is put into the place $y_b$. The default data transformation function $f$ assigns the attribute values of the token from $x_1$ to the corresponding attributes of the new token in the output place $y_b$.

If the function $r_2$ is undefined (this happens, in particular, when the selected output place $y_b$ is a non-empty $S$-place), then the transition does not fire. When this elementary net is a component of a larger Petri net then the function $r_2$ is recomputed every time after any other transition fires in this Petri net until the result is a legal output place. Function of input selection $r_1$ for this net always has constant value equal to the set of input places of the net. Formally, the condition $C$ for firing the transition of this type of elementary is as follows:

$$C = B(x_1) \land \ldots \land B(x_m) \land \neg(B(y_1) \land \ldots \land B(y_n)),$$

$$r_1 = [\text{TRUE} \rightarrow \{x_1, \ldots, x_m\}], \quad r_2 = [\neg B(y_1) \rightarrow \{y_1, \ldots, \neg B(y_n) \rightarrow \{y_n\}].$$

$$M'(x_i) = M(x_i) - 1, \quad i = 1, 2, \ldots, m, \quad M'(y_i) = M(y_i) + 1, \quad M'(y) = M(y), \quad j = 1, 2, \ldots, n, \quad n \neq b, \quad b \in \{1, 2, \ldots, n\}, \quad v(y_b) = v(x_1), \quad i = 1, 2, \ldots, k, \quad k = \min\{s(x_1), s(y_b)\}, \quad v(y_b) = 0, \quad i = k + 1, \ldots, s(y_b), \quad \text{if } k < s(y_b).$$

(5)

Figure 2 illustrates a possible change of marking of an elementary net of type $X$ after firing of its transition. It is assumed that default functions $r_2$ and $f$ are used in this example. The dashed arc in the figure means that attribute values of the token in the first input place $x_1$ are assigned, by the default function $f$ at the end of transition firing, to attributes of a new token in the first empty output place $y_2$ computed by the default function $r_2$. Marking of other output places does not change after firing of the transition of this net. The undefined value of function $r_2$ is used to implement, with this elementary net, a conditional waiting, without explicit specification of the duration of waiting time. In this case, the transition of the net becomes inactive until some condition specified by function $r_2$ becomes true in the entire Petri net. An example of such a condition can be the existence of a token in some places of the Petri net. Logically, the elementary net of this type provides a conditional routing of a token to some output place and works as a switching block.

Elementary net of type $Y$ implements the conditional selection of one of its input place to remove a token from it at the end of transition firing and then puts a token into each output place. Graphically, the transition bar of this elementary net has a short horizontal line segment that is directed to input places of the net. To fire a transition in this net, it is necessary (but generally not sufficient) that there is a token in at least one of its input place and that all output $S$-places are empty (that is, not marked). The condition $C$ for firing the transition in this type of elementary net is as follows:

$$C = (B(x_1) \lor \ldots \lor B(x_m)) \land \neg(B(y_1) \lor \ldots \lor B(y_n)),$$

$$r_1 = [B(x_1) \rightarrow \{x_1\}, \ldots, B(x_m) \rightarrow \{x_m\}], \quad r_2 = [\text{TRUE} \rightarrow \{y_1, \ldots, y_n\}], \quad M'(x_b) = M(x_b) - 1, \quad M'(x) = M(x), \quad i = 1, 2, \ldots, m, \quad i \neq b, \quad b \in \{1, 2, \ldots, m\}, \quad M'(y) = M(y) + 1, \quad j = 1, 2, \ldots, n, \quad v(y) = v(x), \quad i = 1, 2, \ldots, k, \quad k = \min\{s(x_b), s(y)\}, \quad v(y) = 0, \quad i = k + 1, \ldots, s(y), \quad \text{if } k < s(y).$$

(6)

On default, function $r_1$ will select only one input place of the net to remove a token from this place. In this case, the input places are tested in an order starting with place $x_1$. If condition $C$ is true, then at least one input place $x_b$ of the net contains a token, so that the corresponding predicate $B(x_b)$ is true, function $r_1$ is defined and selects place $x_b$ as its result.
Figure 2: Elementary net of type X before and after transition firing.

Figure 3: Elementary net of type Y before and after transition firing.

Figure 3 exemplifies a possible change of marking of this net as a result of firing of its transition. Dashed arrows show that with default functions \( r_1 \) and \( f \), attribute values of the token from the first marked input place \( x_3 \) will be assigned at the end of transition firing to attributes of a new token in each output place of the net. Note that, after this assignment, a token will be removed only from place \( x_3 \). Marking of other non-empty input places, like place \( x_{m+1} \), will not change after transition firing. With the use of a non-default function \( r_1 \), any input place can be selected for removing a token. Correspondingly, with a non-default function \( f \), any desired values can be computed and assigned to attributes of tokens in each output place of the net.

As with the net of type X, the undefined value of function \( r_1 \) can be used to implement, with this elementary net, a conditional waiting without explicit specification of the duration of waiting time. In this case, the transition of the net becomes inactive until some condition specified by function \( r_1 \) becomes true in the entire running Petri net. Logically, an elementary net of type Y implements a specific form of multiplexing for its inputs, with the controlled selection of an input and the possibility to realize a conditional waiting.

Due to space limitation, more complete description of these elementary net is omitted. Detailed information about them, with numerous examples of their use in models of information systems is given in the book (Kostin and Ilushechkina, 2010). This book contains, on the attached CD, also the Petri net tool WINSIM for running the models written in terms of these Petri nets.

**STRUCTURE OF THE SCHEDULER**

Figure 4 shows the structure of the scheduler. The main component of this structure is the scheduling manager with associated scheduling procedure written in the Model
Figure 4: Structure of the scheduler.

Description Language (MDL) of the system (WINSIM, 2001). This system implements the used extended Petri nets. As input data, the scheduling manager uses an array of step durations, array of step numbers on machines and some working state arrays and variables. Usually, the manager is in a state of waiting for an event in the Petri net. It runs the control procedure as a response to each event (transition firing) in the net.

The unit of work of the manager is an attempt to schedule one step of some job on the related machine. If, at the event moment, a step of a job can be scheduled for processing by the machine corresponding to this step, then the manager prepares related scheduling information for the step (such as machine identifier, job number, its step number and step duration). This information is passed in the form of attributes of a token to the corresponding machine for step processing. Thereafter, the manager returns to its waiting state. After the machine finishes the step, information on the scheduled step is stored in a file. After creation of the schedule for a given job shop task, the file will have complete scheduling information in the form of text lines with machine number, job number, step number, step duration, start and finish time of step. Basically, this information represents a text form of Gantt chart. Since the release time of all jobs is assumed to be the same and equal to zero for all jobs, the makespan of the schedule is the finish time of the last processed job step.

Figure 5 shows Petri net scheme of the scheduler for 10 machines, which are modeled by transitions $T_1, T_2, ..., T_{10}$, with dynamically assigned time delays equal to job step durations. End of firing of each of these transitions signifies that a machine becomes idle and produces a driving event for the scheduler.

The scheduling manager is implemented by elementary net $X_{100}$ of type $X$ with one input place and two output places. When this transition is enabled and fires, then a token from place $S_{100}$ goes into place $S_{101}$ or $S_{103}$ dependent on the value of reserved word $OUT$. In particular, if $OUT = 1$ then, the token goes to place $S_{101}$. This corresponds to the case where a job step is scheduled for some machines. The corresponding scheduling data prepared by the scheduler as token attributes is passed to the machine for processing the job step. The second output with $OUT = 2$ signifies the end of the scheduling process for a given scheduling data instance.

Transition $T_{101}$ duplicates a token from place $S_{101}$ into two tokens: one goes into place $S_{100}$ to reset the manager, while the other token moves through switching transition.
Figure 5: Petri net scheme of the scheduler (with models of machines).

X101 to one of the places S1, S2, ..., S10 to load the selected machine. Reappearance of a token in place S100 after firing of transition T101 enables transition X100 that initializes in its control procedure and value OUT to zero to put transition X100 into the waiting state. Data transformation procedure associated with transition T101 assigns attribute values to the token in place S102. These values are the final values prepared by the scheduling manager for the machine to be loaded.

If for the current event, no job step can be scheduled, then transition X100 does not fire and remains in the enabled waiting state until the next event occurs in the Petri net.

The size of the Petri net in Figure 5 depends only on the number of machines $m$. The net consists of $(6 + m)$ transitions and $(5 + 2m)$ places independent of the number of jobs. This is in a clear contrast with the use of known timed Petri nets and the size and structure which depends not only on the number of machines, but also on the number of job steps. For example, the size of timed Petri net for representation of the notorious $10 \times 10$ job shop problem is 200 transitions and 400 places (Aalst, 1995). Thus, for the problem of $n$ jobs and $m$ machines, the net size is $2nm$ transitions and $4nm$ places. To find a schedule with such timed Petri net, it is necessary, as proposed in the mentioned work to initially create the net and then determine its reachability graph, which becomes quite a
challenging task for a sufficiently large job shop problem.

THE SCHEDULING ALGORITHM

As earlier stated, a unit of work of the scheduling manager is an attempt to schedule one job step as a response to an event in the active Petri net. Usually this means that some machines become idle and can be loaded with a ready job step. In general, a few steps of different jobs can be ready for processing at this moment. In this case the manager will decide which of the claiming job steps should be selected for processing by the machine. Recall that each job step has two parameters – step number and step duration for the corresponding machine.

For selection of one job step for processing on a specific machine, three greedy heuristics, or strategies are proposed and implemented. In strategy 1, the manager selects from all claiming steps, a step with minimal duration. Clearly, with this strategy processing of long job steps will be delayed. In strategy 2, the manager selects a step, which has the minimal duration and minimal step number. Here, steps of all jobs tend to be processed with approximately the same rate.

Finally, in strategy 3 the manager selects a step, which has the minimal number and maximal duration to give a preference to long job steps. Here is the pseudocode of the control procedure associated with transition X100 in Figure 5 and implements scheduling algorithm of a job step with strategy 1:

1. Initialization of working arrays.
2. If steps of all jobs are scheduled then set OUT ← 2 /* Fire X100 to stop scheduler */ else proceed with algorithm steps [3] - [13].
3. Loop 1: initialize data for jobs 1 .. n.
4. Run external loop 2 for jobs 1 .. n.
5. Run internal loop 3 for steps s = 1 .. m of job j.
6. Find and save the smallest number smin of not yet scheduled step of job j.
7. Save job number j, step number smin and corresponding machine number ma.
8. End of loop 3.
9. Check the selected step number smin for a possible scheduling: if machine ma is busy or the previous step of job j is not finished then continue loop 2.
10. Save results of loop 3 in working arrays A (job number, selected step number, machine number) and B (processing time of step on this machine).
11. End of loop 2.
12. Run loop 4: in array B, find a step with minimal duration minstep and its job number jb.
13. If minstep is not valid ([has the value initialized at the beginning of the loop), then set OUT ← 0 /* No job step can be scheduled at the moment, the manager remains in the waiting state */ else BEGIN save job number jb, its step number, machine number ma, step duration, step finish time; label the step as being processed, label machine ma busy, increment the number of scheduled job steps, set OUT ← 1 /* Fire transition X100 with output 1 to put a scheduling token into place S101 */ END.

Complexity of scheduling of one job step is O(nm) due to double loop (with loops 2 and 3). Since the number of steps for all jobs is nm, the complexity of the scheduler is O(m^2n^2). Thus, the scheduler runs in polynomial time.

TESTING THE SCHEDULER WITH SINGLE INSTANCES OF DATA

Experiments with the proposed scheduler were conducted in WINSIM system, running as an application in Windows 7 on a laptop computer with 32-bit CPU Intel Celeron 1.5 GHz. For the experiments, Petri net of Figure 5 was described in Model Description Language (MDL) of WINSIM. The source text includes not only the structure of Petri net, but also procedures, associated with elementary nets and declarations of token attributes, arrays and other variables. Kostin and Ilushechkina (2010) gave a detailed description of MDL. The prepared source text is compiled and linked in WINSIM. The result is the executable file of the scheduler. Input data for it are presented in Modeling Control Language (MCL) of WINSIM. The data are two arrays of processing times of job steps and job step numbers on machines. These data are submitted to WINSIM at the start of the scheduler run.

The conducted experiments were divided into two groups. In the first group, known 10 x 10 and 15 x 15 job shop problems with a single instance of scheduling data each were solved and compared with results obtained in published works. In each experiment, elapsed computer time was also registered by WINSIM. For each tested problem, three scheduling strategies previously described were applied. Results of experiments of the first group are achieved makespan value, a text form of Gantt chart and statistics on transitions and places of Petri net. These statistics include, among other items, average utilization of transitions and average time of activities of transitions. The former statistics is used for estimation of machine loads.

Figure 6 shows obtained statistics on transitions of the scheduler for 10 x 10 job shop task in strategy 1, with input data taken from the report of Fisher and Thompson (1963). Note that each of the transitions T1, T2, ..., T10 representing 10 machines fires 10 times, since each machine processes exactly one step each of the 10 jobs. Utilizations of transitions T1, T2, ..., T10 are the loads of machines. In particular, the load of machine 1 represented by transition T1 is 0.459.

Figure 7 shows fragment of the text form of Gantt chart output by the scheduler for the same 10 x 10 job shop task. Each row in these fragments corresponds to one step of
some job performed by a specified machine, with start and finish time of the step on the machine. Clearly, the number of rows in the complete Gantt chart is $10 \times 10 = 100$. The last row contains obtained makespan of 1074.

Table 1 shows results of solutions of known $10 \times 10$ and $15 \times 15$ job shop problems with different methods for a single instance of data. For comparison, only those published works in which scheduling experiments were conducted with well known benchmark data from the reports of Fisher and Thompson (1963) and Taillard (1993), with accurately documented results were used. In the experiment with $15 \times 15$ problem, the first instance

---

<table>
<thead>
<tr>
<th>Transition</th>
<th>State</th>
<th>Utilization</th>
<th>Firings</th>
<th>Mean firing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T200</td>
<td>0</td>
<td>0.000000E+00</td>
<td>100!</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>Y200</td>
<td>0</td>
<td>0.000000E+00</td>
<td>100!</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>T10</td>
<td>0</td>
<td>3.817805E-001</td>
<td>10!</td>
<td>4.100000E+00</td>
</tr>
<tr>
<td>T9</td>
<td>0</td>
<td>4.944134E-001</td>
<td>10!</td>
<td>5.310000E+00</td>
</tr>
<tr>
<td>T8</td>
<td>0</td>
<td>4.641624E-001</td>
<td>10!</td>
<td>4.990000E+00</td>
</tr>
<tr>
<td>T7</td>
<td>0</td>
<td>4.571695E-001</td>
<td>10!</td>
<td>4.910000E+00</td>
</tr>
<tr>
<td>T6</td>
<td>0</td>
<td>3.873371E-001</td>
<td>10!</td>
<td>4.160000E+00</td>
</tr>
<tr>
<td>T5</td>
<td>0</td>
<td>4.972067E-001</td>
<td>10!</td>
<td>5.340000E+00</td>
</tr>
<tr>
<td>T4</td>
<td>0</td>
<td>5.875233E-001</td>
<td>10!</td>
<td>6.310000E+00</td>
</tr>
<tr>
<td>T3</td>
<td>0</td>
<td>5.176909E-001</td>
<td>10!</td>
<td>5.650000E+00</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>5.102421E-001</td>
<td>10!</td>
<td>5.480000E+00</td>
</tr>
<tr>
<td>T1</td>
<td>0</td>
<td>4.503171E-001</td>
<td>10!</td>
<td>4.930000E+00</td>
</tr>
<tr>
<td>X101</td>
<td>0</td>
<td>0.000000E+00</td>
<td>100!</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>T102</td>
<td>0</td>
<td>0.000000E+00</td>
<td>100!</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>T103</td>
<td>0</td>
<td>0.000000E+00</td>
<td>1!</td>
<td>0.000000E+00</td>
</tr>
<tr>
<td>X100</td>
<td>0</td>
<td>0.000000E+00</td>
<td>101!</td>
<td>0.000000E+00</td>
</tr>
</tbody>
</table>

---

**Figure 6:** Statistics on transitions of Petri net of the scheduler in Figure 5 for $10 \times 10$ scheduling task, strategy 1.

**Figure 7:** Initial and final parts of Gantt chart for $10 \times 10$ job shop scheduling task, strategy 1.
Table 1: Solutions of job shop scheduling problems with a single instance of benchmark data.

<table>
<thead>
<tr>
<th>Problem size and source</th>
<th>Solution method</th>
<th>Obtained makespan</th>
<th>CPU or computer</th>
<th>Computer time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 × 10, data from (Fisher and Thompson, 1963)</td>
<td>Branch and bound (Carlier and Pinson, 1989)</td>
<td>930</td>
<td>PRIME 2655</td>
<td>3305 s</td>
</tr>
<tr>
<td></td>
<td>Simulated annealing (Laarhoven et al., 1992)</td>
<td>902 (?)</td>
<td>VAX 785</td>
<td>687 s</td>
</tr>
<tr>
<td></td>
<td>Cutting plane (Applegate and Cook, 1991)</td>
<td>n.a.</td>
<td>Sun Spark</td>
<td>420 s</td>
</tr>
<tr>
<td></td>
<td>Timed Petri Nets (Aalst, 1995)</td>
<td>1023</td>
<td>n.a.</td>
<td>22 s</td>
</tr>
<tr>
<td></td>
<td>Fast tabu search (Nowicki and Smutnicki, 1996)</td>
<td>n.a.</td>
<td>PC</td>
<td>30 s</td>
</tr>
<tr>
<td></td>
<td>Tabu search (Pezzella and Merelli, 2000)</td>
<td>930</td>
<td>Pentium 133 MHz</td>
<td>80 s</td>
</tr>
<tr>
<td></td>
<td>Very fast tabu search (Grabovsky and Wodecki, 2006)</td>
<td>n.a.</td>
<td>Sun Spark</td>
<td>420 s</td>
</tr>
<tr>
<td></td>
<td>Genetic algorithm (Crose et al., 1995)</td>
<td>946</td>
<td>Pentium 333 MHz</td>
<td>1.2 s</td>
</tr>
<tr>
<td></td>
<td>Proposed scheduler</td>
<td>1173/1217</td>
<td>Laptop, Intel</td>
<td>2.1/</td>
</tr>
<tr>
<td>15 × 15, data from (Taillard, 1993)</td>
<td>Local search (Vaessens et al., 1996)</td>
<td>1206 - 1218</td>
<td>Sun Spark 330</td>
<td>1385 s</td>
</tr>
<tr>
<td></td>
<td>Genetic algorithm (Crose et al., 1995)</td>
<td>1305</td>
<td>PC 486/25</td>
<td>1880 s</td>
</tr>
<tr>
<td></td>
<td>Tabu search (Pezzella and Merelli, 2000)</td>
<td>1241</td>
<td>Pentium 133 MHz</td>
<td>2175 s</td>
</tr>
<tr>
<td></td>
<td>Very fast tabu search (Grabovsky and Wodecki, 2006)</td>
<td>n.a.</td>
<td>Pentium 333 MHz</td>
<td>7.6 s</td>
</tr>
<tr>
<td></td>
<td>Proposed scheduler</td>
<td>1441/1376/1449</td>
<td>Laptop, Intel</td>
<td>6.1/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1441</td>
<td>Laptop, Intel</td>
<td>6.3/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1449</td>
<td>Intel Celeron</td>
<td>6.4 ms</td>
</tr>
</tbody>
</table>

(Taillard, 1993) was chosen. Three values in the third and the fifth columns of Table 1, separated by slash are makespan and computer time values with strategies 1, 2 and 3 for the proposed scheduler.

As Table 1 demonstrates, the proposed scheduler requires thousands of times less computer time than other methods. Even for the very fast known tabu search method, for 10 × 10 problem, the computer time of the proposed scheduler is 600 times lesser when compared.

**TESTING THE SCHEDULER WITH SERIES OF DATA INSTANCES**

In the second group of experiments, 10 × 10, 15 × 15 and 20 × 20 job shop problems were solved with a series of 1000 random scheduling data instances each. The instances were generated according to the technique described in the study of Taillard (1993), with uniform probability distribution of integer-valued job step durations in range [1, 100] and job step numbers in range [1, m] for m machines. To generate the data instances, the Petri net was expanded to include the generator that sequentially submits data instances to the scheduling manager.

Figure 8 shows the expanded Petri net for this group of experiments. The instance generator is implemented by transition T1000. The token in place S1000 represents the initial marking of the net. The remaining added transitions and places are necessary to control the end of scheduling of each generated data instance and the end of the scheduling process after generation of the last data instance. Each unit of the generated data consists of two integer two-dimensional arrays – array of step durations of all jobs on all machines and array of step numbers. An example of these two arrays is given in Figure 9 for 15 × 15 scheduling task.

Clearly, such experiments represent a sort of a “mass production” of schedulers and to the best of our knowledge, were never conducted in previous works since they could require a prohibitively long computer time. The results here are the histogram of makespan values and statistics on transitions and places. These statistics were used for estimation of the average makespan, standard deviation of makespan, average loads of machines and relative unbalance of machine loads.

For a relative load unbalance of machines $u$, the following expression was used:

$$u = (L_{\text{max}} - L_{\text{min}})/L^*,$$  \hspace{1cm} (7)

Where $L_{\text{max}}$ and $L_{\text{min}}$ are maximal and minimal loads in the set of machines and $L^*$ is the average load over all machines. This metric has been proposed for estimation of load unbalance in a distributed multiserver queuing system.
Figure 8: Petri net scheme of the scheduler with instance generator for 15 × 15 scheduling problem.

TIMES ON MACHINES

40 61 43 38 51 71 19 83 64 38 35 76 45 91 47
47 85 80 72 70 28 15 75 20 42 3 58 4 27 16
6 70 31 35 20 49 42 42 2 65 99 85 89 86 50
85 65 86 31 7 96 89 56 42 50 13 96 33 59 15
17 91 70 78 53 57 41 1 9 18 91 22 70 16 60
22 25 32 6 94 14 42 23 69 66 7 84 10 80 9
14 58 54 87 22 70 22 84 6 99 43 21 33 26 20
17 96 96 63 50 93 60 71 39 62 95 54 37 78 33
85 48 61 36 71 91 31 62 51 80 63 12 61 16 21
26 56 7 97 28 58 74 63 53 8 53 4 47 90 69
36 30 55 11 62 69 67 87 59 11 85 61 75 6 82
93 69 54 68 84 4 69 82 27 40 59 47 68 7 81
10 59 61 32 1 68 54 32 14 6 87 7 4 94 60
42 57 86 10 16 33 58 83 80 10 69 32 33 47 88
9 34 60 77 25 10 47 34 65 92 76 99 96 62 34

STEPS ON MACHINES

8 4 3 15 10 1 9 2 6 13 5 12 11 14 7
7 5 1 6 11 8 2 10 4 13 15 9 12 14 3
8 13 10 11 5 3 6 4 15 14 9 2 12 1 7
7 11 4 6 2 14 10 12 13 8 15 9 3 1 5
15 3 10 9 13 2 14 4 8 1 11 12 6 7 5
10 2 7 5 4 13 6 14 15 12 8 3 11 1 9
11 1 10 14 5 15 8 7 12 3 4 2 13 6 9

Figure 9: Randomly generated instance of input data (arrays) for 15 × 15 job shop scheduling task (rows are jobs and columns are machines).
Table 2: Statistics on scheduling of a series of 1000 randomly generated instances for the proposed scheduler.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Achieved average makespan</th>
<th>Standard deviation of makespan</th>
<th>Computer time per instance, ms</th>
<th>Average machine load</th>
<th>Relative unbalance of machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 × 10</td>
<td>990/ 99/ 964</td>
<td>1.9/ 2.1/ 2.1</td>
<td>0.503/ 0.530/ 0.516</td>
<td>0.024/ 0.024/ 0.019</td>
<td></td>
</tr>
<tr>
<td>15 × 15</td>
<td>1516/ 124/ 1449</td>
<td>6.1/ 6.3/ 6.4</td>
<td>0.493/ 0.527/ 0.516</td>
<td>0.015/ 0.013/ 0.015</td>
<td></td>
</tr>
<tr>
<td>20 × 20</td>
<td>2024/ 134/ 1932</td>
<td>14.6/ 15.6/ 15.1</td>
<td>0.434/ 0.529/ 0.520</td>
<td>0.014/ 0.013/ 0.013</td>
<td></td>
</tr>
</tbody>
</table>

(Kostin and Ilushechkina, 2010).

Table 2 shows the results of running the proposed scheduler with 1000 random instances of data. Columns 2, 3, ..., 6 represent average values of the used metrics for three strategies. Three values in each column of this table, separated by slash are for three investigated strategies. As one can see from the table, strategy 2 provides the best (minimal) average makespan for all problem sizes and the highest average machine load. The relative unbalance of machines is small and approximately the same in three strategies. One can see that strategy 2 has proved to be the best in terms of makespan and machine utilization. Relative load unbalance of machines is quite small and is approximately the same for all strategies.

It is interesting that the best average makespan of 10 × 10 scheduling problem over 1000 instances, equal to 940 in strategy 2, is only 1% larger than the optimal makespan of 930 as obtained by Carlier and Pinson (1989) for this problem in a long computing process (Table 1). No comparative data are given in Table 2 since to the best of our knowledge, no similar experiments with a long series of scheduling tasks allowed led to the best of our knowledge, no similar experiments with a long series of scheduling tasks allowed led to obtaining of more reliable estimation of scheduler characteristics than with the use of a single instance. Like all heuristic methods, the scheduler does not guarantee obtaining of the optimal schedule. However, it provides a reasonable approximation to the optimal makespan which can be acceptable for practical purposes. Requiring extremely small computer time, the scheduler can be used for real time applications. In practice, this can be often more desirable than attempting to achieve the time-expensive optimal solution.

CONCLUSION

Table 1 show that the proposed scheduler requires hundreds or thousands of times less computer time than any other known scheduling method, exact or heuristic. Extremely small computer time for solving a scheduling task allowed, for the first time, to conduct experiments with long series of scheduling tasks. Testing the scheduler with a long sequence of instances of scheduling tasks allowed led to obtaining of more reliable estimation of scheduler characteristics than with the use of a single instance. Like all heuristic methods, the scheduler does not guarantee obtaining of the optimal schedule. However, it provides a reasonable approximation to the optimal makespan which can be acceptable for practical purposes. Requiring extremely small computer time, the scheduler can be used for real time applications. In practice, this can be often more desirable than attempting to achieve the time-expensive optimal solution.

REFERENCES