Scheduling Multiprocessor Tasks in Presence of Correlated Failures *

Ireneusz Czarnowski[1]), Walter J.Gutjahr[2]),
Piotr Jędrzejowicz[1]), Ewa Ratajczak[1]),
Aleksander Skakowski[3]), Izabela Wierzbowska[1])

[1]) Department of Information Systems, Gdynia Maritime Academy, Gdynia, Poland
{irek, pj, ewra, iza}@wsm.gdynia.pl
[2]) Department of Statistics and Decision Support Systems, University of Vienna, Austria
walter.gutjahr@univie.ac.at
[3]) Department of Computer Science,Gdynia Maritime Academy, Gdynia, Poland
askakow@wsm.gdynia.pl

Abstract

The paper deals with scheduling multiple processor (m-p) tasks on multiple processors to maximize overall schedule reliability under the requirement that all tasks meet their deadlines. An m-p task can be represented in a schedule by one or more independently developed variants. We allow correlation of variant failures within a task and model the correlation using a hyperparameter model with Beta prior. Because the considered problem belongs to NP-hard class, four different approximation algorithms based on soft-computing methods are proposed to solve the problem. The algorithms include island-based evolutionary, neural network, population learning, and hybrid 3opt-tabu search approaches. Experiment results show that the proposed algorithms find good quality solutions in a reasonable time.

1 Introduction

As it was pointed out by [3], [22], [6], and [7] there exist scheduling problems where tasks are processed on more than one processor at a time. During the execution of these multiprocessor (m-p) tasks communication among processors executing the same task is implicitly hidden in a "black

*Acknowledgment: This research has been supported by the KBN grant no. 8T11 02019.

box" denoting an assignment of this task to a subset of processors during some time interval. This paper deals with scheduling multiple variant software on multiple processors to maximize schedule reliability. Multiple variant software is modelled by m-p tasks. The proposed approach addresses one of the frequently encountered conflicts involving high dependability and safety standards required versus system performance and cost.

One of the potential application areas for multiprocessor task scheduling is the fault-tolerant computing under time constraints. Among several techniques used to achieve software fault tolerance, some best known include N-version programming (NVP) [4], recovery blocks (RB) [24], [20], and N-version self-checking programming (NSCP) [30], [21]. A hybrid solution integrating NVP and recovery blocks is known as the consensus recovery blocks [27]. In all the above listed techniques, the required fault tolerance is achieved by increasing the number of independently developed program variants, which in turn leads to higher reliability at a cost of the additional resources used. It should be noted that the m-p task concept could be used to model variety of the fault-tolerant structures, since all require processing of redundant variants, internal communication and execution of an adjudication algorithm. In such a model, the size of a task would correspond to the number of its redundant variants.

Scheduling m-p tasks is understood as assigning processing elements to tasks and, at the same time, deciding on task size and structure, in such a way that all constraints are satisfied and some overall performance goal is optimized. Scheduling m-p tasks differs from traditional, single-variant task scheduling problems, by the extended solution space. It includes not only an assignment of tasks to processors but also a decision as to which combination of redundant variants, from a set of the available ones, should be used to construct each task (at least one variant of each task has to be included within a schedule). Algorithms for scheduling multiprocessor tasks constructed from multiple program variants have been proposed in [19], [8]. The discussed algorithms are based on the assumption that failures of multiple program variants are statistically independent.

The present approach differs by allowing correlation of failures within a task. Hence, the earlier assumption is not needed any longer.

In this paper the global optimization criterion is schedule reliability understood as a probability that all scheduled tasks will be executed without failures. Schedule reliability is to be maximized under the requirement that all tasks meet their respective deadlines. Unfortunately the discussed scheduling problem belongs to the NP-hard class. To enable obtaining solutions within reasonable time the following approximation algorithms, based on soft-computing methods, have been designed and tested:

- island-based evolution algorithm (IBEA);
- neural network algorithm (NNA);
- hybrid 3opt-tabu search algorithm (O3TSA);

- population learning scheme (PLS).

The paper is organized as follows. Section 2 contains the problem formulation. Section 3 contains a description of the multiprocessor task concept and the corresponding reliability model. Section 4 gives a short description of the proposed approximation algorithms. Section 5 contains a numerical example and the results of computational experiments carried out. Finally, in Section 6 suggestions for future work and conclusions are made.

2 Problem formulation

A set N of n of multiprocessor tasks to be processed under time constraints is considered. It is assumed that the following information with respect to each task in N is available:

- ready time - a_j, $j = 1, \ldots, n$;
- deadline - d_j, $j = 1, \ldots, n$;
- number of available variants - NV_j, $j = 1, \ldots, n$;
- variant processing times - p_{ji}, $j = 1, \ldots, n$, $i = 1, \ldots, NV_j$;
- variant reliabilities - r_{ji}, $j = 1, \ldots, n$, $i = 1, \ldots, NV_j$;
- level of failure correlation between variants - θ_j, $j = 1, \ldots, n$.

The considered problem of scheduling m-p tasks, denoted using an extended Graham's [15] notation as $P|m-p, a_j, d_j, \theta_j|R$, is characterized by a set of multiple, identical processors P, and a set of multiple-variant tasks N. Each task has the maximum size (i.e., number of variants) NV_j, and the minimum size equal to 1. Tasks are independent and non-preemptable with ready times and deadlines differing per task. Tasks have processing times, which may differ per combination of variants chosen for execution. Task j of the size $m = |G_j|$ requires m parallel processors for processing, where G_j denotes the set of the indices of those variants that are selected for task j. The optimization criterion is the reliability of the overall schedule, calculated as the product of the task reliabilities. Task reliabilities are calculated using the reliability model described in Section 3. In general failures of variants *within* a task can be statistically correlated. The decision variables include assignment of tasks to processors and combination of variants to be executed for each task. Tasks cannot be delayed.

3 Multiprocessor tasks and their reliability model

A set N of n of multiprocessor tasks to be processed under time constraint is considered. A multiprocessor (m-p) task is built from a finite set of possibly independently developed program variants. It is assumed that scheduling such a task requires two types of resources - processors and time. The decision problem consists of:

1. selecting the set $G_j \subseteq \{1, 2, \ldots, NV_j\}$ for each task j,
2. scheduling the tasks in such a way that each task j gets access to $m = |G_j|$ processors executing in parallel and p_j time.

The number of processors needed is equal to the number of program variants used to construct an m-p task. The time required to process an m-p task can be calculated from its variant processing times - p_{ji}, $j = 1, \ldots, n$, $i = 1, \ldots, NV_j$, where NV_j is a number of available variants. The processing time of the task j is calculated as:

$$p_j = \begin{cases} max_i\{p_{ji}\} + o_j; \quad i \in G_j \quad if \quad |G_j| \geq 2 \\ p_{ji} \quad otherwise \end{cases} \tag{1}$$

where: o_j is an overhead time and G_j is a set of variant indexes indicating variants used to construct task j.

From the problem assumptions it is clear that tasks are allocated to processors only after their size and the internal structure, i.e. combination of variants to be processed, have been chosen. It also follows from these assumptions that when constructing a task from, say, NV available variants there are $2^{NV} - 1$ possible structures to be considered.

To reduce the solution space a concept of an effective task structure is introduced. A structure of a given size is considered effective if there exists no other combination of program variants with the same (or shorter) processing time and higher reliability. For a task j of the size equal to x $(1 \leq x \leq NV_j)$ there are at most $NV_j + 1 - x$ effective structures. The effective structures of a task j are generated as follows:

```
Procedure ESG
Begin
  Order variants according to their non-decreasing processing
  times as the first criterion and reliability as the second.
  for x=2 to NV_j do
    for i=x to NV_j do
      among all variants v_k, k < i choose x-1 variants
      of the highest reliability;
      make a new combination consisting of the v_i and
      the x-1 variants chosen in the previous step.
    end;
  end;
  For each size x (1 <= x <= NV_j) choose the effective
  structures among the above obtained structures.
end.
```

Generating effective structures for a given task size requires polynomial time. Altogether, for a task j with the number of available variants equal to NV_j, there are, at most, $NV_j(NV_j + 1)/2 = \sum_{x=1}^{NV_j} (NV_j + 1 - x)$ effective structures.

A nontrivial aspect of our investigation is the computation of the task reliability R_j from the variant reliabilities r_{ji}, $i \in G_j$. As stated in Section 1, we want to overcome the idealized assumption that redundant variants fail independently from each other. Models for the reliability computation of fault-tolerant structures with variants the failures of which are correlated have been developed in [10], [26] and [28]. These approaches, however, assume that the estimated reliability values of the single variants of a task are all equal, which is a too severe restriction within our context. We need an extended model where these estimates may vary. Nevertheless, in the special case of equal single reliability estimates, the model should be consistent with the framework proposed in the articles cited above.

In the following, we describe an extension of the Nicola-Goyal approach serving this purpose. The formal background of this extension is presented in more detail in [16]. Here, we concentrate on the basic ideas. For simplicity, let us assume that $G_j = \{1, \ldots, k\}$ with $1 \le k \le NV_j$ (which can always be achieved by a suitable re-assignment of the variant indices).

The key for an appropriate modelling of failure dependencies between redundant program variants has been given in [10]. There is no reason why independently developed variants should not fail independently on some *fixed* program input y; nevertheless, on a random program input Y, the failure events for different variants get correlated simply by the fact that not all inputs y have the same probability leading to a program failure: some of them are very easy to process, such that they are most probably not failure-causing, while others are difficult to process, such that all variants may have a high probability of failing on them. This induces a positive correlation between the failures of the variants in an execution of the task.

Conceptually, we may consider a population Pop of programs potentially written to the given specification, and assume that the program variants $1, \ldots, k$ are (randomly) selected from subsets $Pop_1, \ldots, Pop_k$ of Pop, respectively. By $p(y)$ and $p_i(y), (1 \le i \le k)$, we denote the fraction of programs in Pop resp. in Pop_i that fail at input y. Different program variants are assumed to fail independently at each *fixed* input y. Now let us suppose that also the input is selected randomly according to some usage distribution, i.e., the input is a random variable Y. Then, the fractions of failing programs in Pop resp. in Pop_i, $p(Y)$ resp. $p_i(Y)$, are random variables as well. The random variable $p(Y)$ can be interpreted as the probability that during operational usage, a program variant selected arbitrarily from Pop will fail, and $p_i(Y)$ has analogous interpretation. Nicola's and Goyal's approach is based on the assumption that $p(Y)$ has a Beta distribution $Beta(\alpha, \beta)$ with parameters α and β , which are chosen as follows:

$$\begin{aligned} \alpha &= \pi/\theta \\ \beta &= (1 - \pi)/\theta. \end{aligned} \tag{2}$$

The density $f(p)$ of $Beta(\alpha, \beta)$ is given by

$$f(p) = p^{\alpha-1}(1-p)^{\beta-1}/B(\alpha, \beta),$$

where $B(\alpha, \beta)$ is the Beta function. In (2), π is the estimated unreliability of a variant drawn from Pop (i.e., the expected value of $p(Y)$), and $\theta \geq 0$ is a correlation level. The boundary case $\theta = 0$ represents failure independence, whereas the other boundary case $\theta = \infty$ represents identical failure behavior of all versions (one of them fails exactly if all others fail). This can be seen as follows: On the assumptions above, the number K of variants, out of N given variants, failing at a random input Y, is beta-binomially distributed with parameters α, β and N. In particular, the probability that both of two given variants fail at Y is

$$\frac{B(\alpha+2, \beta)}{B(\alpha, \beta)} = \frac{(\alpha+1)\alpha}{(\alpha+\beta+1)(\alpha+\beta)},$$

and this expression tends to $\alpha(\alpha+\beta) = \pi$ as $\theta \to 0$, which is the failure probability of two variants with identical failure behavior, and to π^2 as $\theta \to 0$, which is the failure probability of two independently failing variants.

Let us say a few words of explanation to the choice of Beta distributions. This type of model distributions is widely used, especially in Bayesian statistics, but also in other areas (such as, e.g., project planning by PERT). They combine the advantage of easy computational tractability with flexibility in application: mean value and variance can be tuned similarly as by using the normal distribution, but contrary to the last, also skewed distributions can be modelled, and the range of possible values can be restricted to an interval (in our case, the interval $[0, 1]$ is required). In principle, we could adapt our technique also to other families of distributions, but the family of Beta distributions seems to be the most appropriate from the viewpoint of ease of application.

In our extended model, we assume that also the random variables $p_i(Y)$ $(1 \leq i \leq k)$ have Beta distributions, with parameters α_i and β_i chosen as follows:

$$\begin{aligned} \alpha_i &= \pi_i/\theta \\ \beta_i &= (1-\pi_i)/\theta \end{aligned} \tag{3}$$

where π_i is the estimated unreliability (probability of failing) of variant i drawn from Pop_i, and θ is the correlation level as explained above.

Furthermore, a certain functional relation between $p(y)$ and $p_i(y)$ is assumed. We suppose that the fraction $p(y)$ of programs in Pop failing on a certain input y already determines the fractions $p_i(y)$ of programs in Pop_i failing on this input y. Of course, this functional dependency (we denote it by φ_i) cannot be chosen arbitrarily. Instead, it must be established in such a way that the distribution $Beta(\alpha, \beta)$ of $p(Y)$ is transformed just to the distribution $Beta(\alpha_i, \beta_i)$ of $p_i(Y)$.

In [16] it is shown that the following function φ_i satisfies this requirement: Let F be the distribution function of $Beta(\alpha, \beta)$, and let F_i be the distribution function of $Beta(\alpha_i, \beta_i)$. We set

$$\varphi_i(p) = F_i^{-1}(F(p)). \tag{4}$$

The relation between $p(y)$ and $p_i(y)$ is then given by

$$p_i(y) = \varphi_i(p(y)). \tag{5}$$

On the assumptions above, the unreliability $unrelj = 1 - R_j$ of task j, i.e., the probability that all its variants i $(i = 1, \ldots, k)$ fail on a random input Y, can be computed by elementary probabilistic calculations. One obtains:

$$unrel_j = \int_0^1 \varphi_1(p) \ldots \varphi_k(p) f(p) dp \tag{6}$$

where $f(p) = F'(p)$ is the first derivative of the distribution function F, i.e., the corresponding density function.

As it has been demonstrated in [16], the value $unrel_j$ computed from the formula above is independent of the chosen basic distribution F: another F yields another density $f(p)$, but also other transformation functions $\varphi_i(p)$, leading finally to the same computation result. The distribution function F of a $Beta(\alpha, \beta)$ distribution is given by

$$F(p) = \int_0^p u^{\alpha-1}(1-u)^{\beta-1}/B(\alpha, \beta) du \tag{7}$$

with $B(\alpha, \beta)$ denoting the Beta function as before. Using this fact and a Taylor expansion of the Beta function, we obtain the following numerical procedures for the computation of $unrel_j$:

```
Function B(α, β, x)
Begin
  value = 0;
  for j := 0, 1, ..., β − 1 do
    value := value +(−1)^j (β − 1)! x^(α+j) / (j!(β − 1 − j)!(α + j));
  return value;
end.

Function phi(α, β, α0, β0, p)
Begin
  c := B(α0, β0, p) ∗ B(α, β, 1)/B(α0, β0, 1);
  x := 0.1;
  while (B(α, β, x) > c) do
    x := x/10;
  x_old := x;
  repeat until (|x − x_old| smaller than some pre-defined real
  precision constant)
    x_old := x;
    x := x_old − (B(α, β, x) − c)/(x^(a−1) ∗ (1 − x)^(β−1));
  return x;
end.
```

```
Function integrand(k, α[ ], β[ ], α_0, β_0, p)
Begin
  q := 1;
  for i := 1 to k do
    q := q * phi(α[i], β[i], α_0, β_0, p);
  return q * p^(α_0 − 1) * (1 − p)^(β_0 − 1) / B(α_0, β_0, 1);
end.
```

```
Function unrel(k, α[ ], β[ ], α_0, β_0)
Begin
  return result of numerical integration of integrand
  (k, α[ ], β[ ], α_0, β_0, p) between p = 0 and p = 1,
  e.g., by Simpson's rule;
end.
```

Therein, α_0 and β_0 are the parameters chosen for the basic Beta distribution F, and $\alpha[\,]$ and $\beta[\,]$ are the arrays of the numbers α_i and β_i. Since $\beta_i = (1-\pi_i)/\theta$, and since π_i and θ are small in practical applications, the values β_i can be rounded to integers. The procedures above are based on the assumption that this has been done.

4 Scheduling algorithms

4.1 Island based evolution algorithm - IBEA

An island based evolution algorithm (IBEA) belongs to the distributed algorithm class. To improve efficiency of genetic algorithms (GA) several distributed GA's were proposed in [14], [5], [2]. Their ideas included an island-based approach where a set of independent populations of individuals evolves on "islands" cooperating with each other. The island-based approach brings two benefits: a model that maps easily onto the parallel hardware and extended search area (due to multiplicity of islands) preventing from sticking in local optima. Promising results of the island-based approach motivated the authors to design the IBEA for scheduling multiple-variant tasks.

The proposed island-based evolutionary algorithm (IBEA) works on two levels with two corresponding types of individuals. To evolve individuals of the lower level a population-based evolutionary algorithm (PBEA) is proposed. On the higher level the following assumptions are made:

- an individual is an island $I_k, k = 1, 2, \ldots, K$, where K - is the number of islands;

- an island is represented by a set of the lower level individuals;

- all islands are located on the directed ring;

- populations of lower level individuals evolve on each island independently;

- each island I_k regularly sends its best solution to the successor $I_{(kmodK)+1}$ in the ring;

- the algorithm stops when an optimality criterion is satisfied or the preset number of generations on each island have been generated;

- when IBEA stops the best overall solution is the final one.

On the lower level the following assumptions are made:

- an individual is represented by a n-element vector $S_u = \{c_i^j | i = 1, 2, \ldots, n;$
  $1 \leq j \leq n\}$, used to construct a list of tasks, where i - is an index describing a place of the task j on the list, and c_i^j - is a code representing the combination of variants used to construct task j;

- all S_u representing feasible solutions are potential individuals;

- an initial population is composed in part from the potential individuals for whom combinations of variants and order of tasks on the list are random (random size and order), and in part from the potential individuals for whom combinations of variants are random with tasks order determined by a non-decreasing ready time as the first criterion and a non-decreasing deadlines as the second (random size and fixed order);

- each individual can be transformed into a solution by applying LSG-$P|m-p, a_j, d_j, \theta_j|R$, which is a specially designed algorithm for list-scheduling m-p tasks;

- each solution produced by the LSG-$P|m-p, a_j, d_j, \theta_j|R$ can be directly evaluated in terms of its fitness;

- new population is formed by applying several evolution operators: selection and transfer of some more "fit" individuals, random generation of individuals, crossover, and mutation.

The following pseudo-code shows main stages of the IBEA- $P|m-p, a_j, d_j, \theta_j|R$ algorithm:

```
Procedure IBEA-P|m-p,a_j,d_j,θ_j|R
Begin
  Set number of islands K, number of generations PN for
  each island, size of the population for each island PS.
  For each island I_k, generate an initial population PP_0.
    While no stopping criteria is met do
      For each island I_k do
```

```
        Evolve $PN$ generations using PBEA.
        Send the best solution to $I_{(k mod K)+1}$.
        Incorporate the best solution from $I_{((K+k-2) mod K)+1}$
        instead of the worst one.
     EndWhile
  Find the best solution across all islands and save it as
  the final one.
end.
```

PBEA-$P|m-p, a_j, d_j, \theta_j|R$ algorithm is shown in the following pseudocode:

```
Procedure PBEA-$P|m-p, a_j, d_j, \theta_j|R$
Begin
  Set population size $PS$, generate a set of $PS$ individuals
  to form an initial population $PP_0$;
  Set $ic$:= 0; ($ic$ - iteration counter);
  While no stopping criteria is met do
    Set $ic$:= $ic$ + 1;
    Calculate fitness factor for each individual in $PP_{ic-1}$
    using LSG-$P|m-p, a_j, d_j, \theta_j|R$;
    Form new population $PP_{ic}$:
      Select randomly a quarter of $PS$ individuals from $PP_{ic-1}$
      (probability of selection depends on fitness of an indi-
      vidual);
      Produce a quarter of $PS$ individuals by applying cross-
      over operator to previously selected individuals from
      $PP_{ic-1}$;
      Produce a quarter of $PS$ individuals by applying muta-
      tion operators to previously selected individuals from
      $PP_{ic-1}$;
      Generate half of a quarter of $PS$ individuals from set
      of potential individuals (random size and order);
      Generate half of a quarter of $PS$ individuals from set
      of potential individuals (random size and fixed order);
  EndWhile
end.
```

LSG-$P|m-p, a_j, d_j, \theta_j|R$ algorithm used within PBEA-$P|m-p, a_j, d_j, \theta_j|R$ is carried out in the three steps:

```
Procedure LSG-$P|m-p, a_j, d_j, \theta_j|R$
Begin
  Step 1.  Construct a list of tasks from the code representing
          individuals.  Set a loop over tasks on the list.
  Step 2.  Within the loop, allocate current task to multiple
          processors minimizing the beginning time of its
          processing.  Continue with tasks until all have been
```

```
        allocated.
  Step 3.  If the resulting schedule has task delays, a
        fitness of the individual S_u is calculated as
        R_u = -(1 - ∏ R_l) where l belongs to a set of the
        delayed tasks.  Otherwise, R_u = ∏ R_j, j = 1,...,n.
end.
```

4.2 Neural network algorithm - NNA

Neural networks have been used to obtain solutions of problems belonging to NP-hard class (see for example [1], [17], [18], and [31]). The NNA proposed in this paper is based on a dedicated neural network with a genetic algorithm used during the training phase. The architecture of the NNA - $P|m-p, a_j, d_j, \theta_j|R$ depends on the number and properties of m-p tasks, which are to be scheduled. The set of tasks is partitioned to form subsets of tasks with identical ready times. The number of such subsets corresponds to the number of layers in the neural network. Each layer consists of neurons representing tasks to be scheduled. Neurons at each layer send their signals to adders, the number of which corresponds to the number of available processors. Connection weights between neurons and adders can have binary value {0, 1} interpreted as absence or presence of the connection. The signals between the neurons and the adders represent variant processing times. The adders sum input signals and produce output signals representing completion times of the tasks executed. If such a schedule has delays then the network weights are changed in a learning process using classical genetic algorithm [25] and [29]. The learning process is initialized by an independent decision block.

A layer chromosome is a string of bits, the length of which depends on the number of tasks and processors in this layer. While determining weights of connections the genetic algorithm uses standard genetic operators: one-point crossover, mutation and elitist selection. The proposed Neural Network Algorithm involves executing steps shown in the following pseudo-code:

```
Procedure NNA-P|m-p, a_j, d_j, θ_j|R
Begin
  Set the number of layers k;
  While no stopping criteria is met do
    For i :=1 to k do
      Train network using GA;
    Evaluate the objective function for the obtained network
    structure;
    If the objective function is improved then
    best network structure:= obtained structure;
  endwhile
  Treat the best network structure as the final one and decode
  the problem solution;
```

```
end.
```

4.3 Hybrid 3opt-tabu search algorithm - O3TSA

The proposed approach is an extension of the tabu-search meta-heuristic [12], [13], [11]. The algorithm - $O3TSA - P|m-p, a_j, d_j, \theta_j|R$ is based on the following assumptions and notations:

- A solution Q is a list of objects, each representing a task and a selected combination of variants. An object has attributes including a task number j, $j = 1, \ldots, n$ and a combination number k, $k = 1, \ldots, NK_j$, where NK_j is the number of possible effective combinations of variants for task j. A set of effective combinations is generated by removing all dominated combinations from the set of all possible combinations.
- Tasks represented by objects from the list are allocated to the processors using the following rule: take objects from Q one by one, each time allocating variants with indexes in G_i to the first $|G_j|$ free processors.
- 3opt algorithm used to sort Q by repeating a search for the best schedule for the three chosen objects. First the least reliable task, its predecessor and successor are considered. Next after the best solution for the three is found the least reliable task becomes a forbidden one and the next least reliable task replaces it. Algorithm stops when n-2 tasks are forbidden.
- A move involves moving an object from its position in the schedule to the other position and changing the respective combination. Attributes of the move are denoted as $m(j, p_s, k_s, p_d, k_d, m_v)$, where j is the task number, p_s - position in the schedule, k_s - combination number, p_d - new position, k_d - new combination number, m_v - move value.
- $N(Q)$ denotes a set of all possible moves from Q, called a neighbourhood of Q.
- $N^*(Q)$ denotes a modified neighbourhood of Q, that is a neighbourhood without all tabu-active moves and moves of which it is easy to say that they will not improve the current solution.
- TAM denotes list of tabu-active moves.

O3TSA-$P|m-p, a_j, d_j, \theta_j|R$ involves executing steps shown in the following pseudo-code:

Procedure O3TSA-$P|m-p, a_j, d_j, \theta_j|R$
Begin
  $TAM = \emptyset$;

```
  Generate two initial solutions:
    Q1 - includes the shortest combinations of each task
    ordered by their non-decreasing ready times as the first
    criterion and non-decreasing deadlines as the second;
    Q2 - includes the shortest combinations of each task
    ordered by their non-decreasing deadlines as the first
    criterion and non-decreasing ready times as the second;
  Calculate total tardiness T;
  If T > 0 then problem has no solution, exit;
  Try to improve both solutions using 3-opt algorithm,
  providing the total tardiness remains equal to zero;
  Set number of iterations it;
  max = the better reliability of the Q1 nad Q2;
  For both initial solutions Q1 and Q2 do
    For i = 1 to it do
      While best move is not chosen do
        Consider all moves in N*(Q), select and remember the
        best move (i.e.  with the best move value)
        m(j, p_s, k_s, p_d, k_d, m_v);
        If the best move cannot be found delete from TAM all
        moves with the least tabu tenure;
      endwhile
      Execute the best move chosen;
      Update TAM, for example:
        Add m1(j, 0, 0, p_s, k_s, m_v), it/4;
        Add m2(0, p_s, 0, 0, 0, m_v), it/2;
        Add m3(j, p_s, k_s, p_d, k_d, 0), it/2;
        Delete moves with zero tabu tenure;
      If m_v > max then
        max = m_v;
        Remember the new solution;
      endif
    endfor
  endfor
end.
```

4.4 Population learning scheme - PLS

To obtain a solution of the $P|m-p, a_j, d_j, \theta_j|R$ problem we have adopted the Lin-Kerninghan algorithm [23] originally applied to solving travelling salesman problems, and used it as a population learning tool. The approach is based on the improvement procedure, which can be applied to the existing schedule, possibly not even a feasible one. The procedure aims at improving a schedule by reducing the number of the delayed tasks and, at the same time, increasing schedule reliability. Within the proposed PLS the improvement procedure is applied to a population of solutions. Constructing a list of tasks, sorting it according to non-decreasing deadlines and allocating tasks from this list to the least loaded processors generates an initial population. Tasks are allocated to processors in order of their appearance on the list with some probability q. For each task a number of program variants to be executed is randomly drawn. The improvement procedure aims at maximizing schedule reliability minus the number of delayed tasks. Its idea is to consider one by one the delayed tasks. For each task z a set of other tasks is constructed. The tasks are chosen so that replacing z with $k \in K$ best improves the value of the goal function. As the result a schedule with a double task k appears. Improvement is again applied to the second appearance of the task k. If set K cannot be constructed a schedule is upgraded in such a way that each task is scheduled exactly once and the result compared with the best schedule achieved so far. At the first two levels of recursion set K contains 5 elements, later on, only 1 element.

5 Numerical example and the results of computational experiments

As an example the $P|m-p, a_j, d_j, \theta_j|R$ problem with 11 tasks and 5 processors has been solved using all of the proposed approaches. The respective data is shown in Table 1. All algorithms have generated feasible schedules - that is schedules without delayed tasks, with at least one variant of each task scheduled, and all tasks having the required resources (processor and time) allocated. The respective probabilities of the successful completion of all tasks are:

- island based evolution algorithm - 0.917;
- neural network algorithm - 0.894;
- hybrid 3opt-tabu search algorithm - 0.915;
- population learning scheme - 0.917.

The schedules generated by the respective algorithms are shown in Figure 1. To evaluate the proposed algorithms computational experiment has been carried. It has involved 20 randomly generated problems. Numbers of tasks per problem have been generated as random integers from the uniformly distributed interval U[10, 28], numbers of available processors from U[2, 12], numbers of variants per task from U[2, 5], variant processing times from U[1, 15]. Variant reliabilities have been generated as random real numbers from U[0.8999, 0.9999], and variants failure correlations from U[0.1, 0.2]. The following measures have been used to evaluate the algorithms:

- mean relative error related to the best known solution - MRE;
- maximum relative error encountered - Max RE;

Table 1: Example $P|m-p, a_j, d_j, \theta_j|R$ problem

task number	variants per task	variant number	arrival time a_j	variant processing time p_{ij}	deadline time d_j	variant reliability r_{ij}	correlation between variant failures	time overhead o_j
1	5	1	0	2	9	0,925	0,1705	2
		2	0	2	9	0,9301		
		3	0	4	9	0,9365		
		4	0	4	9	0,9873		
		5	0	5	9	0,9998		
2	5	1	0	3	16	0,945	0,1533	2
		2	0	3	16	0,9569		
		3	0	3	16	0,96		
		4	0	4	16	0,9723		
		5	0	4	16	0,9825		
3	4	1	1	5	21	0,9029	0,1579	1
		2	1	5	21	0,9563		
		3	1	6	21	0,9762		
		4	1	7	21	0,9835		
4	4	1	3	7	25	0,9441	0,1289	2
		2	3	8	25	0,9619		
		3	3	9	25	0,9558		
		4	3	9	25	0,9608		
5	5	1	4	3	19	0,9143	0,1301	2
		2	4	3	19	0,923		
		3	4	4	19	0,9312		
		4	4	4	19	0,9337		
		5	4	4	19	0,9801		
6	5	1	9	8	25	0,906	0,1774	2
		2	9	8	25	0,9092		
		3	9	10	25	0,9237		
		4	9	11	25	0,9254		
		5	9	11	25	0,9367		
7	3	1	13	5	26	0,9074	0,1014	1
		2	13	5	26	0,9497		
		3	13	6	26	0,9633		
8	4	1	14	6	30	0,9029	0,176	2
		2	14	7	30	0,9448		
		3	14	7	30	0,984		
		4	14	8	30	0,9896		
9	2	1	18	5	27	0,9274	0,1814	1
		2	18	6	27	0,9794		
10	2	1	23	5	32	0,9187	0,1709	1
		2	23	6	32	0,9621		
11	4	1	26	2	32	0,9047	0,1045	2
		2	26	2	32	0,9749		
		3	26	2	32	0,9831		
		4	26	2	32	0,9915		

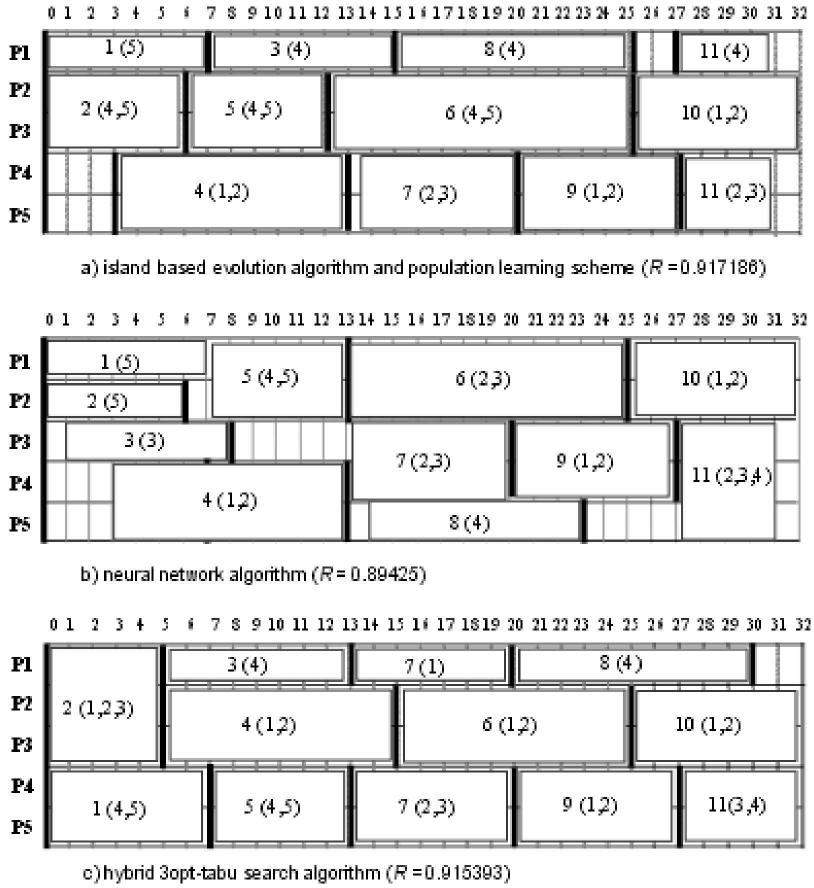

Figure 1: Schedules generated by the proposed algorithms

Table 2: Computational experiment results

Measure	IBEA	NNA	O3TSA	PLS
MRE	0.371%	6.501%	0.243%	1.255%
Max RE	2.21%	18.01%	2.21%	3.34%
Min RE	0%	0%	0%	0%
BRR	31.6%	10.5%	47.4%	15.8%

- minimum relative error encountered - Min RE;

- best result ratio - BRR.

Results of the experiment are shown in Table 2.

First experiment results show that the proposed algorithms generate good quality solutions. The hybrid 3opt-tabu search algorithm outperforms the remaining ones in terms of both - mean relative error and best result ratio. It also has a very low maximum relative error level. Island based evolution algorithm also offers good performance and a low maximum relative error level. However both approaches require more computational resources than the remaining two. Population learning scheme has been faster than IBEA by the factor of 2 to 3. It has taken IBEA from 9:00 to 15:00 minutes to find a solution on Pentium II-MMX CPU, 300MHz, NNA from 0:47 to 3:20 on Celerone 333MHz, and O3TSA from 3:00 to 30:00 on Pentium III CPU, 800MHz to perform the same job.

6 Conclusion

The main contribution of the paper is the introduction of the multiprocessor task scheduling problem with correlated task failures and a successful application of several soft computing approaches to obtain good quality solutions. Computational experiments have shown that even quite basic and simple computational schemes based on generic metaheuristics can produce competitive results, however the experiments need to be confirmed by further, more extensive experimental comparisons. It has been also shown that novel modelling approach overcoming the idealized assumption that redundant variants fail independently from each other lends itself to optimization and could be useful in generating realistic schedules for systems involving fault-tolerant program structures run under hard time constraints. The presented approach has been limited to a static case where information on set of tasks to be run is known beforehand. Further research should lead into investigating construction of dynamic schedulers where the suggested approach (static one) could be used as a base for an online scheduler. Another direction of research is further improvement of the soft computing algorithms and techniques resulting in improving either quality of result or reducing the demand for computational resources.

References

[1] Al Wu, Tam PKS (1999) A Neural Methodology and Strategy of Quadratic Optimisation. Neural Computing and Applications. Springer-Verlag London Limited, pp 283-289

[2] Alba E, Troya J (1999) Analysis of Synchronous and Asynchronous Parallel Distributed Genetic Algorithms with Structured and Panmictic Islands. In: Jose Rolim et al. (Eds.) Proceedings of the 10th Symposium on Parallel and Distributed Processing, 16-gru April. San Juan, Puerto Rico, USA, pp 248-256

[3] Avizienis A, Gilley GC, Mathur FP, Rennels DA, Rohr JA,Rubin DK (1971) The star (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. IEEE Transactions on Computers 20 (11): 1312-1321

[4] Avizienis A, Chen L (1977) On the implementation of the N-version programming for software fault tolerance during execution. Proc. IEEE COMPSAC 77, pp 149-155

[5] Belding TC (1995) The Distributed Genetic Algorithm Revisited. In: Eshelman L.J. (ed.) Proceedings of the Sixth International Conference on Genetic Algorithms. Morgan Kaufmann, San Francisco, CA, pp 114-121

[6] Błażewicz J, Ecker KH, Pesch E, Schmidt G, Węglarz J (1996) Scheduling Computer and Manufacturing Processes. Springer Berlin

[7] Błażewicz J, Drabowski M, Węglarz J (1986) Scheduling Multiprocessor tasks to minimize schedule length. IEEE Transactions on Computers 35: 389-393

[8] Czarnowski I, Jdrzejowicz P, Ratajczak E (1999) Scheduling Fault-Tolerant Programs on Multiple Processors to Maximize Schedule Reliability. In: M.Felici, K.Kanoun, A.Pasquini (Eds.) Computer Safety, Reliability and Security, Lecture Notes in Computer Science. Springer, pp 385-395

[9] Duch W, Korczak J (1998) Optimization and Global Minimization Method Suitable for Neural Networks. Neural Computing Surveys 2, http://www.icsi.berceley.edu/ jagopta/NCS

[10] Eckhardt DE, Lee LD (1985) A theoretical basis for the analysis of multiversion software subject to coincident errors. IEEE Transactions on Software Engineering SE-11, 1511-1517

[11] Glover F, Laguna M (1987) Tabu Search. Kluver Boston

[12] Glover F (1989) Tabu Search - Part I. ORSA Journal of Computing

[13] Glover F (1990) Tabu Search - Part II. ORSA Journal of Computing

[14] Gordon VS, Whitley D (1993) Serial and Parallel Genetic Algorithms as Function Optimizers. In: Forrest S. (ed.) Proceedings of the Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA, pp 177-183

[15] Graham RL, Lawler EL, Lenstra JK, Rinnooy Kan AHG (1979) Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals Discrete Math. 5: 287-326

[16] Gutjahr WJ, (2000) A Reliability Model for Inhomogeneous Redundant Software Versions with Correlated Failures. To appear in: Computer Systems – Science and Engineering.

[17] Hopfield JJ, Tank DW (1985) Neural. Computations of Decision in optimization Problems. Biological Cybernetics 52, 141-152

[18] Janiak A (1999) Wybrane Problemy i Algorytmy Szeregowania Zadań i Rozdziału Zasobów. Akademicka Oficyna Wydawnicza PLJ, Warszawa

[19] Jedrzejowicz P, Czarnowski I, Szreder H, Skakowski A (1999) Evolution-Based Scheduling of Fault-Tolerant Programs on Multiple Processors. In: Jose Rolim et al. (Eds.) Parallel and Distributed Processing, Lecture Notes in Computer Science. Springer, pp 210-219

[20] Kim KH (1984) Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults. Proc. 4th International Conference on Distributed Computing Systems. IEEE Computer Society Press, pp 526-532

[21] Laprie JC, Arlat J, Beounes C, Kanoun K (1990) Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures. IEEE Computer 23, 7: 39-51

[22] Lloyd EL (1981) Concurrent task systems, Operations Research 29(1): 189-201

[23] Lin S, Kerningham BW (1973) An Effective Heuristic Algorithm for the Travelling-Salesman Problem. Operations Research 21: 498-516

[24] Melliar-Smith PM, Randell, B (1977) Software reliability: the role of programmed exception handling. Sigplan Notices 12, 3: 95-100

[25] Michalewicz Z (1992) Genetic Algorithm + Data Structures = Evolution Programs. Springer-Verlag

[26] Nicola VF, Goyal A (1990) Modeling of correlated failures and community error recovery in multiversion software. IEEE Transactions on Software Engineering SE-16, 35-359

[27] Scott RK, Gault JW, Mc Allister DF (1987) Fault tolerant software reliability modeling. IEEE Transactions on Software Engineering 13, 5: 582-592

[28] Tomek LA, Muppala JK, Trivedi KS (1993) Modeling correlation in software recovery blocks. IEEE Transactions on Software Engineering SE-19, 1071-1086

[29] Whitley D (1995) Genetic Algorithm and Neural Networks, Genetic Algorithm and Computer Science. In: G.Winter, J.Periaux, M.Galan and P.Cuesta (Eds.) John Wiley, pp 203-216

[30] Yau SS, Cheung, RC (1975) Design of Self-Checking Software. Proc. Int. Conf. on Reliable Software. IEEE Computer Society Press, pp 450-457

[31] Zohu DN, Cherkassky V, Balwin TR, Olson DE (1991) A Neural Approach to job shop scheduling. IEEE Transactions on Neural Networks 2, 175-179