Abstract—In this paper, we consider an approach to developing parallel versions of the algorithms based on the modified probability changing method for constrained pseudo-Boolean optimization. Optimization algorithms are adapted for the systems with shared memory (OpenMP) and cluster systems (MPI). The parallel efficiency is estimated for the large-scale non-linear pseudo-Boolean optimization problems with linear constraints and traveling salesman problem.

Keywords—discrete optimization; parallel computing; MPI

I. LARGE-SCALE PSEUDO-BOOLEAN OPTIMIZATION PROBLEM

Most exact solution approaches to the problem of discrete (combinatorial) optimization (knapsack problem, the traveling salesman problem etc.) are based on branch-and-bound method (tree search) [1,2,5]. Unfortunately, most of such problems are in the complexity class NP-hard and require searching a tree of the exponential size and even parallelized versions of such algorithms do not allow us to solve some large-scale pseudo-Boolean optimization problems in acceptable time without significant simplification of the initial problem.

The heuristic random search methods do not guarantee any exact solution but random search methods are statistically optimal. I.e. the percent of the initial problem.

Let's consider the problem:

\[ F(X) = \max \left( \sum_{i=1}^{N-V} a_i x_i \right) \left( \sum_{i=1}^{N-V} c_i x_i \right) \]

\[ \sum_{i=1}^{N-V} b_{ik} x_i \leq b_k \forall 1 \leq k \leq N_{\text{constr}} \]  

\[ \sum_{i=1}^{l-V+1} x_i \leq 1 \forall 1 \leq l \leq N; \]  

Here, \( a_i, c_i, b_{ik} |1 \leq i \leq N\cdot V| \) are some constants, \( x_i (1 \leq i \leq N) \) are Boolean variables. The objective function is non-linear.

The real large-scale problems have sometimes millions of variables. For example, the problem of assortment planning of the retail trade company [6] may include thousands goods names to be selected which can be shipped from hundreds suppliers and have 3-10 variants of retail price. In general, problems of such kind can be solved only with random search algorithms.

Being initially designed to solve the unconstrained optimization problems, the probability changing method (MIVER) is a random search method organized by the following common scheme [1,2].

1. \( k=0 \), the starting values of the probabilities \( P_i = \{p_{i1}, p_{i2}, \ldots, p_{iN}\} \) are assigned where \( p_{ii} = P\{x_i=1\} \). Correct setting of the the starting probabilities is a very significant question for the constrained optimization problems.

2. With probability distributions defined by the vector \( P_k \), we generate a set of the independent random points \( X_k \).

3. The function values in these points are calculated: \( F(X_k) \).

4. Some function values from the set \( F(X_k) \) and corresponding points \( X_k \) are picked out (for example, point with maximum and minimum values).

5. On the basis of results in item 4, vector \( P_k \) is modified.

6. \( k=k+1 \), if \( k<R \) then go to 2. This stop condition may differ.

7. Otherwise, stop.

To be implemented for the problems like (1-3), this method has to be modified. The modified version of the variant probability method, offered in [6,7] allows us to solve large-scale problems with dimensions up to millions of Boolean variables.

In case of the large-scale problems, even the calculation of the linear objective function takes significant computational resources. The number of constrains usually also grows with the increase of the problem dimension. So, the calculation of the objective function and the constrains is a very large computational problem if it is repeated lot of times.

That is why, the distribution of the computational tasks between the parallel processors or cluster nodes is very important.

In this paper, we do not consider greedy algorithms (which are deterministic or also randomized [8, 10]) though they are often used to improve the results of the random search methods as the final step of them.
Also, we do not consider the parallel genetic algorithms [11] though some approaches offered for genetic algorithms may be implemented for random search algorithms parallelization.

Here, we offer an approach of adaptation of the existing programs realizing the random search methods of constrained pseudo-Boolean optimization to be implemented in the parallel systems.

II. SERIAL VERSION OF THE ALGORITHM AND ITS PARALLELIZATION WITH OpenMP

The scheme of the algorithm for the serial systems is shown in Fig. 1, variant 1.

At the step of initialization, all the variables \( p \) (components of the probability vector \( P \)) are set to their initial values \((0 < p < 1 \quad \forall \quad p \in P \)). Since our algorithm realizes the method of constrained optimization, the initial value of the probability variables is sometimes very important. Then, we generate the vectors \( X \) of optimized boolean variables. In our case of constrained problem of knapsack type, the large values of vector \( P \) components generate the values of \( X \) which are out of the feasible solutions area due to the constraints (9,10). Due to the constraints (3), the optimal initial values of the vector \( P \) components do not exceed \( 1/(L+1) \) [7]. We set the initial values of the vector \( P \) to \( 1/(L+1) \) but we have to reduce this value if several starts of our algorithm give us no results in feasible solutions area. This process is not illustrated in Fig. 1 for the simplicity.

Instead of maximum number of steps (\( M \) in Fig. 1), we can use the maximum run time as the stop condition. In some cases, it is reasonable to use the maximum number of steps which do not give us the result better than previous ones as the stop condition.

![Figure 1. Parallel algorithm for systems with shared memory](image-url)
In the cycle \((i=1,N)\), we generate the set of \(N\) vectors \(X_i\) in accordance with the probability vector \(P\). Then, the objective function is calculated for each \(X_i\). Also, we calculate the values of the left parts of our constraints, and introduce the second objective function \(f^p\) (penalty function):

\[
f_{ki} = f^p_k[X] = C_{\text{penalty}} \sum_{k=1}^{N \cdot V} F_{pk}^k[X];
\]

\[
f^p_k[X] = \begin{cases} 
0, & \sum_{i=1}^{N \cdot V} b_{ik} \leq B_k; \\
\sum_{i=1}^{N \cdot V} b_{ik} - \sum_{i=1}^{N \cdot V} b_{ik} > B_k, & \end{cases}
\]

Here, \(C_{\text{penalty}}\) is some coefficient. If some estimation of the maximum value of the objective function is available, we can use it as the value of the coefficient \(C_{\text{penalty}}\). For linear objective functions:

\[
C_{\text{penalty}} = \sum_{i=1}^{N \cdot V} |a_{ik}|.
\]

The modified objective function \(f^M\) is the sum of the objective function and penalty.

When all the values of \(f_k\) and \(f^M_k\) are calculated, we choose the best (maximum) value for \(f^M_k\), and, if there were the variants of the vector \(X_i\) which satisfy all the constraints \(f^M_k=0\) then we choose the maximum value of the objective function \(f_k\) for those variants of the vector \(X_i\).

In case of the constrained optimization, we have to use a special method of adaptation of the vector \(P\). The solution of different practical tasks shows the best result if we use the multiplicative adaptation with rollback procedure [2,6]. In this case, the components of the vector \(P\) are never set to the value of 0 or 1 which may cause that all the further generations of the \(X\) vector have the same value of the corresponding component which does not give the feasible solution due to constraints (3).

\[
p_{k-1,j} = \begin{cases} 
0.5, & x_j^{\max} = 1 \wedge x_j^{\min} = 0 \wedge p_{k-1,j} < 0.5,
1 - 0.5, & x_j^{\max} = 1 \wedge x_j^{\min} = 0 \wedge p_{k-1,j} \geq 0.5,
1, & x_j^{\max} = 0 \wedge x_j^{\min} = 1 \wedge p_{k-1,j} < 0.5,
1 - 1, & x_j^{\max} = 0 \wedge x_j^{\min} = 1 \wedge p_{k-1,j} \geq 0.5,
1 - 0.5, & p_{k-1,j} < 0.5,
0.5, & p_{k-1,j} \geq 0.5.
\end{cases}
\]

After several steps, the values of \(P\) vector elements are close to 0 or 1 and the decrease of the adaptation step \((d)\) results in generation of the similar vector \(X\) exemplars which correspond to some local maximum. The rollback procedure is helpful to avoid that situation. It sets the values of \(P\) vector and its adaptation step \(d\), to initial (or other) values. In simplest case, rollback is performed after several steps which do not improve the best objective function value.

The best results are demonstrated with methods of partial rollback procedure which change some part of \(P\) vector components or change all the components so that their new values depend on previous results. We can use the following rollback formula:

\[
p_0 = (p_{k-1} + q_0 p_0)/(1+q_0), \text{if } p_{k-1} < p_0 \quad (8)
\]

Here, \(p_0\) is the initial value of the probability. The coefficient \(q_0\) may be constant or vary depending on the results of previous steps. For example, it may depend on the quantity of the steps which do not improve the maximum result \((s_m)\).

\[
q_0 = w / s_m. \quad (9)
\]

The weight coefficient \(w\) has to be chosen experimentally.

In case of constrained optimization, the choice of the initial value \(p_0\) can be very important. In some cases, the incorrect initial value causes the generation of \(X\) vector sets that lay out of the feasible solutions area. After several steps, the penalty function minimization process results in adaptation of \(P\) vector which allows to generate \(X\) vector exemplars that satisfy our constraint conditions. But the rollback procedure returns the probability vector to its initial (usually incorrect) value. Let’s consider a simplest example. Let our problem have only one constraint like that:

\[
b_1 x_1 + b_2 x_2 + b_3 x_3 + ... + b_N x_N < B. \quad (10)
\]

Here, \(D\) is the dimension of our problem \((D=N \cdot V)\). The elements of the vector \(X\) at the first steps are generated so that it is set to the value of 1 with probability \(p_0\). The expectation of the left part of (10) is

\[
M = p_0 b_1 + p_0 b_2 + p_0 b_3 + ... + p_0 b_D = \sum_{i=1}^{D} p_0 b_i.
\]

The maximum values of the objective function are usually achieved at the points where the condition is barely satisfied \((b_1 x_1 + b_2 x_2 + b_3 x_3 + ... + b_N x_N = B\) in our case). Therefore, the optimal value of the initial probability is

\[
p_0 = B/(b_1 + b_2 + b_3 + ... + b_N). \quad (12)
\]

The negative effect of the rollback procedure can be reduced by adaptation of the initial value of \(p_0\):

\[
p_{0k} = (p_{0k-1} + p_{0k-1} + ... + p_{0k-1}) / D. \quad (13)
\]

This version of rollback procedure is most actual in case of partial rollback which performed at each step of our algorithm. The implementation of this kind of adaptation&rollback procedure is described below for
the parallel version of the optimization algorithm for systems with no-remote memory access (clusters).

The adaptation of our algorithm for multiprocessor systems with shared memory can be performed by the parallel generation of the exemplars of the $X$ vector and their estimation. The scheme of that version of our algorithm is shown in Fig. 1, variant 2. If our system has $N_p$ processors, the cycle of generation of $N$ exemplars of the vector $X$ can be divided between the processors. Each processor has to generate $N/N_p$ exemplars of the vector $X$ and calculate the value of the objective function, left parts of the constraint conditions and calculate the modified objective function values.

Organizing of the parallel thread takes significant computational expenses. In [14], they estimate that expenses as 1000 operations of real number division. The experiments at 4-processor system with linear 100-dimension problem (105 constraints) show that the parallel version runs 2.8 times faster than the serial one.

III. VERSION FOR CLUSTERS

Our experiments [6,7] with adaptation of random search discrete optimization algorithms [1,2] for parallel execution gave us rather effective results with use of multiprocessor systems with shared memory and OpenMP [14]. But the intensive data interchange in running mechanisms of MPI reduce the efficiency of the realization of the same algorithms for PVM and MPI clusters when the number of the nodes increases. That issue makes the usage of the computing cluster ineffective even in comparison with low-cost multi-core systems.

For testing purposes, we used the following form of the objective function and constraints:

$$F[X] = \sum_{i=1}^{N} \sum_{j=1}^{V} a_{ij} x_{ij} \sum_{i=1}^{N} \left[ 1 - c_{ij} x_{ij} \right] \rightarrow \max; \quad (14)$$

with constraints:

$$\sum_{i=1}^{N} \sum_{j=1}^{V} b_{ij} x_{ij} \leq B_{C1};$$

$$\sum_{i=1}^{N} \sum_{j=1}^{V} b_{ij} x_{ij} \leq B_{C2};$$

$$\sum_{i=1}^{N} \sum_{j=1}^{V} b_{ij} x_{ij} \leq B_{Cn};$$

$$\sum_{j=1}^{V} x_{ij} \leq 1 \forall 1 \leq i \leq N. \quad (16)$$

The coefficients in the constraints in the test set are generated so that the feasible solutions set is 1000-1000000 smaller than $2^N$ (depends on $N$ and randomly selected values). The algorithm replaces the whole set of constraints with penalty function [2, 6, 7]. Problem of a traveling salesman [1, 2, 11]:

$$F[X] = -\sum_{i=1}^{N} a_{ij} x_{ij} \rightarrow \max; \quad (17)$$

The only constraint in this case is that the matrix of Boolean variables $X$ must describe the adjacency matrix of Hamiltonian graph. This cycle must be the only path in that graph. Here, $a_{ij}$ is the distance (cost, time, etc.) between $i$-th and $j$-th places (cities), $a_{ij} = 0 \forall i, a_{ij} = a_{ji} \forall i,j$. If $x_{ij} = 1$ then it means that the salesman must go from $i$-th place to the $j$-th one.

We tried to solve the optimization problems in Argo cluster (SISSA, Trieste, Italy) [15]. The program code was adapted for MPI running. Algorithm designed for OpenMP parallelizing (see Fig.1, variant 2) was implemented for parallelizing via MPI. MPI synchronizing procedure was used at the step “Synchronization”. Following this flowchart gave negative results in comparison even with a single node (8 nodes, 2 processes per node, 50 variants of $X$ sets in each generation on each node, 10000 Boolean variables, 11000 constraints).

Analysis of the problem showed that at the step of synchronization, each node spent approximately 12% of time waiting for the other nodes to finish the calculation, data interchange between nodes in this case is rather intensive (10000 Boolean variables had to be sent from each node, then, each one had to receive 1000 real numbers of the probability vector after each step).

Reconstruction of the algorithm so that each node performs $N$ steps (100-500 for that scale of the problem) separately, and comparing the results after the $N$-th step, choosing the best ones by master nodes and broadcasting new probability vector gave much better results. But, in this case, each node had to spend up to 43% of time waiting for the others to complete their calculations.

Another way to organize the parallel execution of the search processes with the maximum independence from each other is to organize multiple starts of the algorithm at all the nodes. The algorithm starts at the nodes with the same or different initial parameters (for example, different initial values of the probability vector elements). These multiple simultaneous starts are performed instead of the rollback procedure in the serial version of the optimization algorithm. Each node starts the cycle of the random $X$ vector generation with probability vector adaptation. If the algorithm does not improve the best objective function value during several steps, the rollback procedure is performed. When all the nodes reach the stop condition, the results of all the nodes are compared to figure out the best one as the final result of the optimization problem.
In its simplest form, the above approach does not need any modification of the implemented software. The serial version of the algorithm runs at all the nodes simultaneously and the results of the nodes are then compared by the operator or by the special node.

The simultaneous execution of the serial version of the algorithm improves the results of the calculations insignificantly due to the similar behavior of the algorithm at all the nodes. The comparison of the \( P \) vector values from different nodes after several steps shows that the difference between them reduces with each following step and the nodes generate the new exemplars of the \( X \) vectors in the very close area. In case of the algorithm with no average probability adaptation, the simultaneous execution of the serial version of the algorithm with different initial probability value gives the results much better than the serial algorithm executed at the single node. But in most cases, the version of the optimization algorithm with the average probability adaptation executed at the single node shows even better results. So, our approach must support the average probability adaptation for the optimization algorithms executed at different nodes (13).

The algorithm was re-written (Fig. 2) so that all the data transfer was performed at the step of checking of the restart conditions (text in bold used for the steps performing message passing). If the algorithm does not improve the local maximum values of the objective functions after \( c_{\text{max}} \) generations, the process tries to communicate to the other processes to check if any other node has improved the previous global maximum. If so, the process receives that value and the corresponding \( X \) vector values. If the local maximum value is better than the received one then we report our values as the new global maximums.

In our algorithm, we do not send \( P \) vector values which are rather long. The vector \( P \) for the problem with 10000 variables needs 40000 bytes to be sent. The node having achieved global optimum sends this value and the values of the vector \( X^*_G \) (1250 bytes in case of 10000 Boolean variables). The values of the new probability vector are calculated as

\[
p_i = \frac{(C_{\text{cor}} + (1 - 2C_{\text{cor}})x^*_G) p^\text{AVG}_G}{C_{\text{cor}} + (1 - 2C_{\text{cor}}) p^\text{AVG}_G}.
\]  

(18)

Here, \( x^*_G \) is the \( i \)-th element of the received vector \( X^*_G \), \( D \) is the dimension of the problem (number of the elements of \( X^*_G \)), \( C_{\text{cor}} \) is some small real value, \( p^\text{AVG}_G \) is the average value of the probability vector generated the vector \( X^*_G \). If the received message does not contain it then the node which has received the message evaluates it:

\[
p^\text{AVG}_G \approx \sum_{i=1}^{D} x^*_G / D.
\]  

(19)

Taking into consideration the constraints (3) if they exist, the value of the constant \( C_{\text{cor}} \) can be calculated as \( 0.5/V \).

A flowchart of our algorithm is shown in Fig.3.

The last version of the algorithm, though it is rather efficient, has very low data interchange between nodes (log shows that there was only 31-152 data transfer sessions for each node during 1 hour). So, with this intensity of data traffic, there is no need to implement any expensive network to build an efficient MPI cluster for random search problems solution.

IV. NUMERICAL RESULTS

For the parallel random search algorithms, the analytical estimation of the comparative efficiency of the parallel versions in comparison with the corresponding serial algorithms is difficult, described with the recurrent formulas [16] which need the very distinct information of the objective function (linearity, unimodality etc.) In case of constrained optimization, the analytical estimation of the algorithm efficiency is even more complex problem. The analytical estimation of the parallel version of the algorithm is based on the comparison of the time spent to solve the parallel part of the algorithm (Fig. 1, variant 2) and serial part of the algorithm which includes probability vector adaptation and possible roll-back procedure. The probability vector adaptation complexity depends on the problem dimension linearly. The complexity of the parallel part depends on the dimension of the problem and quantity of the constraints which also depends on the problem dimension linearly for most practical problem types. So, the complexity of the parallel part is proportional to the square of the problem dimension in case of linear objective function. That is why, the comparative efficiency of the parallel algorithm increases with the problem dimension growth. The exact analytical estimation of the comparative efficiency is difficult due to the reasons listed above. In case of the cluster systems, the analytical estimation is even more complex problem.

The experimental estimation of the comparative efficiency of the random search algorithms offered in [17] can be implemented for the constrained optimization case. In our case, we create a set of the test problems. The coefficients of the objective functions and constrains are set by a random generator [5,18]. The coefficients in the constraints are also generated. The right parts of the constraints are selected so that the problem has admissible solutions. The problems are solved with serial version of the algorithm. In this case, the stop condition is the time spent by the algorithm. For example, algorithm runs 10 minutes. Then, the serial algorithm is implemented \( K \) times to solve the same problem. The stop condition in this case is the achievement of the result which is equal or more than the result of the first run. Then, the parallel version of the algorithm runs at the same kind of hardware (in this case, more than one processor or cluster node is used)
K times. The stop condition in parallel case is also the achievement of the result exceeding the first achieved objective function value. The total amount of time spent to make K starts of the serial algorithm and K starts of the parallel version are then compared.

In case of the problem (14)-(16), in comparison with the best results achieved by the program working as 8 processes running separately separately (4 nodes, 2 processes per node, 10000 Boolean variables, 11000 constraints, non-linear objective function), our version of the algorithm achieves the same results in half time (181 minutes hours for the separately working nodes and 63 minutes for the parallel algorithm).

The parallel efficiency for the cluster system of 6 nodes is 0.64. An example (Fig. 4) shows the results of running of the algorithm on cluster (6 nodes) and on a single computer which is a part of that cluster. To build this diagram, we included in our algorithm special block which stores the maximum objective function value reached by the algorithm and the current time after each 10 steps to a special array.

The horizontal line indicates the time when the algorithm achieves the control value (65015.17). To reach this control value, the serial version of the algorithm has spent 229 minutes (13748 seconds), the parallel version has spent 59 minutes (3556 seconds).

The average value of the parallel efficiency (0.81 for the system of 2 nodes) is calculated as the average speed-up coefficient after 10 runs for 5 different objective functions. It is interesting, that at the very first steps, the parallel efficiency coefficient is less than at the following steps. The possible reason of this fact is the process of parameters tuning which is performed by each node at the first steps and the intensive message passing. The dependence of the time spent for solving randomly generated problem (14)-(16) with 10000 variables and 11000 constraints and parallel efficiency on the quantity of codes is shown in Fig.5.

Traveling salesman problem with 1000 nodes (1000000 Boolean variables) after 4 hours of calculation on 4 nodes had not given the exact results, the results were still improving at the last steps, problem with 500 points (250000 variables) after 4 hours of calculation gave rather good result, very close to the optimal solution.

```
Figure 2. Parallel algorithm description
```

a) Initialization (at each node), MPI_INIT
b) Generating N random sets of variables X, estimating F(X) and constraints (at each node)
c) Choosing the best and the worst X sets, adaptation of the probability vector P (at each node separately)
d) if this is the master node then:
   d1) check if any data transfer started at the previous steps has been completed
      d1.1) if so, then if it was the maximum result reached by the k-th node, check if this result is better than the global result
      d1.1.1) if so, send message “OK” to the k-th node
      d1.1.2) if not so, send “NOT OK” to the k-th node and send the actual global maximum and the corresponding X vector to the k-th vector
   d1.2) if the completed data transfer contains the best vector X
      d1.2.1) then recalculate F(X) and if it is better than the global maximum, assume that this is the new global maximum
   d2) if there are any messages from the other nodes waiting to be received and no receiving processes are in progress, start receiving (non-blocking, continue processing)
   d3) check if the stop conditions are reached. If so, send “STOP” message to all the processes, wait for all of them to receive it and then MPI_FINALIZE, stop
e) if this is not the master node and I have already started any sending process then
   e1) check if this process has been completed
   e2) if so, start receiving of the answer from the master node
f) if this is not the master node and I am waiting for a response from the master then
   f1) check if the master node has started sending anything
   f2) if so, start receiving (non-blocking)
g) if this is not the master node and I have already started receiving smth. from the master node then
   g1) if the receiving has been completed then
      g1.1) if it was “OK” message, start sending my best X vector (non-blocking)
      g1.2) if it was “NOT OK” message, start receiving global best X vector (non blocking)
      g1.3) if it was the global best X vector, recalculate F(X) and, if it is better than my own maximum, partially restart local process to perform further search around the global maximum
   g1.4) if it was the “STOP” message, then stop, MPI_FINALIZE
h) Checking for the restart conditions (at each node separately), algorithm has to be restarted if it hasn't improved the best results during several last steps
i) if the restart conditions are reached then
   e1) if this is not the master node, start sending the best F(X) value
   e2) reset probability vector
j) go to step b
V. CONCLUSION

The approach to parallelization of the probability changing algorithm proposed in this paper is efficient, its implementation gives significant speedup (4x in case of running on a cluster having 6 nodes for a problem with 1000000 Boolean variables).

An efficient realization of the algorithm for cluster systems needs specific mechanisms to be implemented and total reconstruction of the existing algorithm.

For building an MPI cluster for solving large-scale random search problems with the probability changing method, an expensive high-performance network is not needed because of very low intensity of data traffic.
Figure 4. Comparison of the results of serial and parallel versions

Figure 5. Dependence of the parallel efficiency on the quantity of nodes

REFERENCES


