Yugoslav Journal of Operations Research
34 (2024), Number 1, 93–107
DOI: https://doi.org/10.2298/YJOR230415018B

PARALLEL EVOLUTIONARY ALGORITHMS FOR THE RECONFIGURABLE TRANSFER LINE BALANCING PROBLEM

Pavel BORISOVSKY
Sobolev Institute of Mathematics SB RAS, Novosibirsk, Russia
pborisovsky@ofim.oscsbras.ru

Received: April 2023 / Accepted: August 2023

Abstract: This paper deals with an industrial problem of machining line design, which consists in partitioning a given set of operations into several subsets corresponding to workstations and sequencing the operations to satisfy the technical requirements and achieve the best performance of the line. The problem has a complex set of constraints that include partial order on operations, part positioning, inclusion, exclusion, cycle time, and installation of parallel machines on a workstation. The problem is NP-hard and even finding a feasible solution can be a difficult task from the practical point of view. A parallel evolutionary algorithm (EA) is proposed and implemented for execution on a Graphics Processing Unit (GPU). The parallelization in the EA is done by working on several parents in one iteration and in multiple application of mutation operator to the same parent to produce the best offspring. The proposed approach is evaluated on large scale instances and demonstrated superior performance compared to the algorithms from the literature in terms of running time and ability to obtain feasible solutions. It is shown that in comparison to the traditional populational EA scheme the newly proposed algorithm is more suitable for advanced GPUs with a large number of cores.

Keywords: CNC machines, partial order, setup times, split decoder, parallel computing, scalability.

MSC: 90B30, 90C06, 90C59.

1. INTRODUCTION

In the general problem of machining line design, there is a set of mechanical *operations* that must be performed on a flow of machining parts. The line is represented as a sequence of *workstations* that contain tools to perform certain

operations. The line balancing problem basically consists in partitioning the operations into several subsets corresponding to workstations (like in the well-known Bin Packing problem). In the *Assembly Line Balancing Problem*, a partial order on operations is given and the allocation of operations in the line must respect this order. The problem of our interest known as the *Reconfigurable Transfer Line Balancing Problem* (RTLBP) has the same origin but contains many special technical requirements that reflect the particular properties of machining lines composed of computer numerical control (CNC) machines. In addition to the mentioned partial order, the line must satisfy such constraints as *inclusion* (some operations are required to be installed on the same workstation), *exclusion* (some operations cannot be installed on the same workstation), *accessibility* (all the operation installed on the same workstation must be able to operate without changing the position of the processing part). A special feature of RTLBP is that a workstation may consist of several identical *machines* that work in parallel, which increases the performance of the workstation proportionally to the number of machines. Besides, the need to change tools when switching from one operation to another introduces the sequence dependent *setup times*, which makes RTLBP related to the vehicle routing problems (VRP). Finding even a feasible solution of RTLBP is NP-hard since it requires to find a minimal length Hamiltonian path.

Note that the described constraints are quite known and appear separately in many line balancing problems. Broad reviews of different problem formulations that cover many possible serial-parallel lines structures can be found in [1, 2]. References concerning the sequencing and setup times optimization are given in [3, 4]. Reconfigurable manufacturing systems including the problem of our interest are surveyed in [5]. The considered RTLBP was formulated in [6] where its industrial background is discussed and a Mixed Integer Programming (MIP) model is proposed. This research was further extended in [7], where the extensive computational investigation of this model with IBM CPLEX solver is done, which showed that problems with up to 20 operations can be solved with good approximation quality. Unfortunately, using this approach for problems with more than 100 operations seems to be not realistic even with modern MIP solvers and computer equipment and development of metaheuristic algorithms is of great interest in this case.

Among the heuristics one may notice a greedy algorithm enhanced by a local search [8] and an ant colony algorithm [9], which were tested on instances with about 100 operations. In [10] a Genetic Algorithm (GA) with a MIP decoder is proposed and tested on randomly generated instances with 200 operations. Its results are improved in [11] using the Iterated Local Search (ILS) heuristic based on the *Split* procedure for optimal cutting a permutation into several parts minimizing some additive objective function. An advantage of this approach consists in reduction of the considered line balancing problem to the search of an optimal permutation, which allows to use a vast experience of (meta)heuristics design for permutational problems. It should be noted that the Split procedure is widely used in genetic algorithms for solving different types of VRP [12] or Bin Packing problems [13]. A simple but rather effective genetic algorithm (without Split

procedure) is developed in [14], in which a finely tuned penalty function and a greedy heuristic are proposed, and the solutions of the test instances are further improved.

In the present paper, the Split procedure is used as a decoder in a parallel evolutionary algorithm (EA) adapted for running on a GPU. The main feature of a GPU is a possibility to run a large number of simultaneous threads, which can drastically speed up the code comparing to the traditional CPU computing, provided the algorithm can be highly parallelized. The use of GPUs in the applied evolutionary computation has become widely recognized and there are a number of implementations of exact and heuristic approaches for different problems including VRP [15, 16], although there seem to be no published papers on GPU computing for solving line balancing problems. In this study, a special scheme of an EA is proposed for better adaptation to the parallel execution. The computer experiments on hard large-scale instances showed the advantage of the new scheme comparing to the traditional one in terms of solution quality. The improvement of the running time due to parallelization in comparison to the algorithms from the literature is also demonstrated.

The rest of the paper is structured as follows. Section 2 gives a formal definition of RTLBP. In Section 3, the usage of the split procedure in the EA is described and the parallel EA scheme is presented. In Section 4, the proposed EA is compared to the traditional parallel implementation of the EA and to the other algorithms from the literature. Section 5 gives the conclusions and the directions of future research.

2. PROBLEM DESCRIPTION

Let us introduce the following notation. $N = \{1, ..., n\}$ is a set of all operations that must be performed on each machining part. All the machining parts are similar and form an infinite input stream so that at any moment there is always a new available part to be processed. A workstation is a device that contains machines to perform a certain set of operations. In what follows, a workstation is associated with a sequence of operations assigned to it. A part can be fixed on a workstation in different positions, the set of all possible positions is given by $A = \{1, ..., a\}$. When the part is fixed, its position can not be changed until all the operations of the workstation are complete. For each operation $i \in N$ define:

t_i is an execution time of operation i;

$A_i \subseteq A$ is a subset of part positions for which the execution of operation i is possible (accessibility relations);

$P_i \subset N$ is a subset of operations that must be performed before operation i (precedence relations).

For a pair of operations (i, j) a setup time required to switch tools when operation j is executed immediately after i on the same workstation is given by s_{ij}. When operations i and j are assigned to different workstations, the setup time is not applied. These times are sequence dependent and no any assumptions about symmetry or triangle inequality properties are made for them.

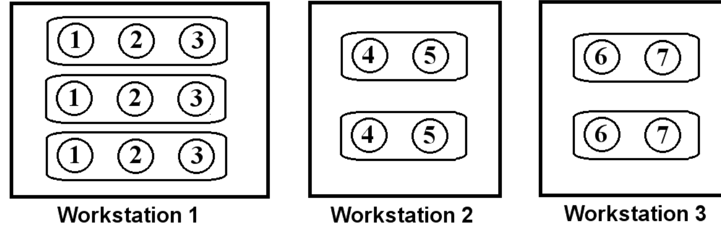

Figure 1: Line with parallel machines on workstations

$\mathcal{E}$ is a set of exclusion subsets: each $E \in \mathcal{E}$ is a subset of operations that cannot be placed together on one workstation, but any proper subset $E' \subset E$ is allowed to be put on one workstation unless $E' \in \mathcal{E}$.

$\mathcal{I}$ is a set of inclusion subsets: each $I \in \mathcal{I}$ is a subset of operations that must be placed together on one workstation.

For the whole line, the following limits are defined.

$T^{\max}$ is a limit on the takt time of the line.

s_0 is the maximal number of operations that can be assigned to one workstation.

n_0 is the maximal number of parallel machines that can be installed on one workstation.

m is the maximal number of workstations in the line.

The problem asks to find a sequence of operations and their partitioning into workstations respecting the described constraints. For any workstation $w = (i_1, ..., i_k)$, its worktime is defined as the sum of processing and setup times: $T(w) = t_{i_1} + s_{i_1,i_2} + t_{i_2} + ... + s_{i_{k-1},i_k} + t_{i_k}$. If this workstation consists of n_w parallel machines, then its takt time is defined as $T(w)/n_w$ and it must not exceed $T^{\max}$. The total number of machines is to be minimized. An illustration of a line with parallel machines is shown in Figure 1.

3. SOLUTION APPROACH

Evolutionary algorithms and genetic algorithms in particular are known to be among the most successful metaheuristic approaches to solve NP-hard optimization problems of large size. This study is concentrated on development and comparison of different ways of parallel EA implementation with an extensive use of GPU computing.

Basically, in the EA, some set of solutions of the problem is represented as a population of individuals that evolves in time by means of random mutations and selection of good-quality solutions. Note that the usual crossover operator is not considered in this paper; although it can be easily incorporated, the experiments did not show significant advantages when using some widely known operators for permutational problems [17], so it is omitted for the simplicity (by this reason the considered algorithms are referred to as EAs rather than GAs).

3.1. Populational evolutionary algorithm

In the GA and EA practice, several replacement schemes are known. In the *populational* scheme, which is regarded as a canonical GA form, the entire population is updated at each iteration. To ensure that good solutions are not lost, a subset of best individuals (*elite*) are copied to the new population. A particular scheme with one elite individual that is used as a baseline in this paper is given below.

Parallel populational evolutionary algorithm

1 Build an initial population $\Pi^{(0)}$ of N random individuals.
2 For t := 1,2,... do:
  2.1 Repeat N times:
    2.1.1 Select parent p from $\Pi^{(t-1)}$.
    2.1.2 $c := Mut(p)$.
    2.1.3 Add c to new population $\Pi^{(t)}$.
  2.2 Find the worst individual in $\Pi^{(t)}$ and replace it by the best individual from $\Pi^{(t-1)}$.
  2.3 If the termination condition is satisfied, stop and return the best found solution

The populational EA is naturally parallel and can be rather easily implemented on the GPU. On the other hand, many studies report a superior performance of the *steady-state* replacement scheme, in which the population is not regenerated completely each time, but at most two individuals are changed. Namely, at each iteration, only one or two parents are selected and generated offspring are included in the population instead of some existing individuals of inferior quality. Unfortunately, this scheme is essentially sequential and cannot be implemented in parallel. A modification of an EA, in which offspring solution are improved by a local search, is known as a *memetic* algorithm [18]. It shows outstanding performance in practice and usually admits parallelization [19].

3.2. Combined evolutionary algorithm

In this paper, another reproduction scheme is proposed. It can be viewed as an attempt to combine the features of the populational, steady-state, and memetic algorithms for better adaptation to the parallel computing. At each iteration, a set of parents is selected and for each parent, several offspring are generated by means of mutation. Then among the offspring of the same parent the best one is chosen and included in the population.

The formal description is given below and an illustration is shown in Figure 2. The most time consuming part consists in applying mutations and evaluating the offspring solutions, which can be easily parallelized on the GPU. The number of parents and offspring define the degree of parallelism and can be easily adjusted depending on the capability of the GPU device.

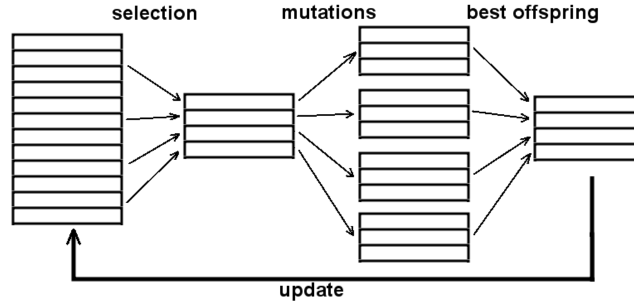

Figure 2: Scheme of the parallel combined EA

Parallel combined evolutionary algorithm

1 Π is an initial randomly built population.
2 Until a termination condition is satisfied do
  2.1 Select $p^1, ..., p^l$ individuals (parents) from Π.
  2.2 For each parent p^k generate h offspring $c^{k1} = Mut(p^k), ..., c^{kh} = Mut(p^k)$ and let c^{k*} be the best one among them.
  2.3 Choose $q^1, ..., q^l$ from Π for replacement.
  2.4 For each $i = 1, ..., l$ if c^{i*} is better than q^i, then remove q^i from Π and insert c^{i*} .
3 Return the best found solution.

Clearly, this general scheme may admit variations in details. In this implementation, each parent is selected with the *tournament* selection operator [17], in which a certain number of individuals are chosen randomly and the best one among them is returned as a parent. The individuals for replacement are selected similarly, but the worst individual among the randomly chosen subset is returned instead of the best one. The mutation operators are developed depending on the particular solution representation, which will be described below. As a termination condition the running time limit is assumed.

3.3. Solution Representation and Split Decoder

In the proposed EA, an individual is stored as a permutation of all operations (*giant sequence*) and the Split algorithm is used to construct an actual solution. This approach has become a common practice in development of heuristics for a wide range of vehicle routing problems (see, e.g., [12]). It was adapted for RTLBP in [11], where an Iterated Local Search heuristic was developed on its base. To cope with the RTLBP constraints, a special randomized algorithm was proposed to construct a *weakly compatible* sequence that can be decoded to an "almost feasible" solution, in which only the number of workstations may exceed m. Unfortunately, this algorithm relies on the dynamic programming procedure and formally has an exponential execution time, which may restrict its usage on large scale problems.

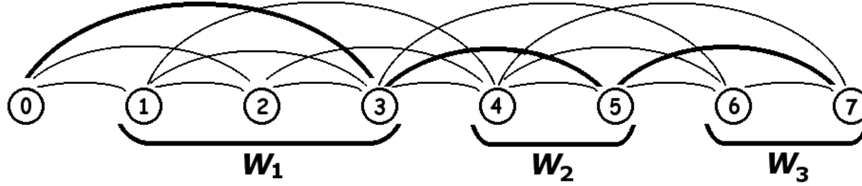

Figure 3: Shortest path approach for cutting a sequence

In this paper, a simpler approach is used that consists in decoding any permutation to some solution. The precedence constraints are ensured by a preliminary correction procedure, and for the other constraints, the penalties are introduced. Initial population is built by generating random permutations. Offspring individuals are obtained by the well-known *swap* and *insertion* mutation operators [17, Sect. 32.3.3], the choice between them is made randomly with equal probability.

Consider an arbitrary permutation of operations $\pi = (\pi_1, ..., \pi_n)$. If it is not feasible w.r.t. the precedence constraints, it can be corrected as follows.

Precedence correction

1 Create new empty permutation $\pi' = ()$.

2 For $i = 1, ..., n$ do

  2.1 Find the leftmost operation in π that does not have precedent operations in π.

  2.2 Put this operation at the i-th place in π' and remove from π.

For the sake of simplicity, in what follows assume that $\pi = (1, 2, ..., n)$ and it satisfies the precedence constraints. The Split algorithm is used for optimal cutting the permutation into several parts. It consist of two main steps: build an auxiliary graph and find the shortest path in it. The graph has $n + 1$ vertices $V = \{0, ..., n\}$ and the set of arcs $A = \{(i, j) : i < j \leq i + s_0\}$. Any arc (i, j) represents a workstation with a sequence of operations $w = (i + 1, ..., j)$ and has an associated length c_{ij} equal to the "cost" of this workstation (the number of machines). Namely, if the workstation is feasible, then its takt time is computed as the sum of processing and setup times: $t_{ij} := t_{i+1} + s_{i+1,i+2} + t_{i+2} + ... + s_{j-1,j} + t_j$. Then the required number of machines on the workstation is computed as $c_{ij} := \lceil t_{ij}/T^{max} \rceil$. If the workstation w is not feasible, c_{ij} is increased by a sufficiently large value M for each violation of inclusion, exclusion, or accessibility constraints, or excess of the limit on the number of machines. The next step is to find the shortest path with at most m arcs from vertex 0 to n. Each arc of the shortest path defines a workstation, namely the endpoint vertex of the arc corresponds to the last operation of the current workstation, and the next vertex marks the start of the new workstation. This is illustrated in Figure 3 where the shortest path is depicted by thick arcs.

The shortest path can be found using the algorithm from [11]. This algorithm for each vertex i builds a set of labels $L_i = \{(a, b)\}$, where a is a current path length from 0 to i and b is the number of arcs in this path. A *dominance* rule is used to reduce the number of labels: label (a, b) dominates (a', b') if $a \leq a'$ and $b \leq b'$. It was shown in [11] that if set L_i does not contain dominated labels, then their number does not exceed m.

Below, the formal description of the algorithm from [11] is reproduced with a minor modification, which consists in introducing an upper bound B so that the labels (a, b) such that $a \geq B$ are not considered (step 2.1.1). In this form, the algorithm returns either a solution of cost less or equal to B or $+\infty$ if such a solution does not exist. Setting the value B close to the expected fitness of the offspring solution in the GA helps to reduce the computation time of the shortest path evaluation.

Constrained shortest path algorithm [11]

1 Initialize labels: $L_0 := \{(0, 0)\}$; $L_i := \emptyset, i > 0$.
2 For each vertex $t = 0, ..., n-1$ do
  2.1 For each arc (t, i) of vertex t do (possibly in parallel)
    For each label (a_t, b_t) of vertex t do
    2.1.1 If $(b_t + 1 < m$ or $i = n)$ and $a_t + c_{ti} < B$ then
      2.1.1.1 Create new label $l := (a_t + c_{ti}, b_t + 1)$.
      2.1.1.2 If l dominates some label $l' \in L_i$ then remove l' from L_i.
      2.1.1.2 If l is not dominated by any label of L_i then add l to L_i.
3 Return $\min_{(a,b) \in L_n} a$.

Actually, the described algorithm computes only the length of the shortest path, which is then used as a fitness of a given individual. The path itself is constructed only once for the best found permutation and is returned as a final solution. To do this, modified labels (a, b, c) can be considered, where c tracks the previous vertex in the path.

4. COMPUTER EXPERIMENTS

For the experimental evaluation, the algorithms were coded in C++ and compiled with g++ 7.5. The GPU part was implemented using CUDA 11.2. The programs were run on the server with Xeon 4210 CPU and Tesla V100 GPU under Ubuntu 18.04.

4.1. GPU implementation

The classic populational and the newly proposed combined EAs described above were implemented with parallelization using CUDA for running on a GPU. The main part of CUDA programming consists in development of one or several *kernel* functions. Each such function defines a piece of code that must be executed large number of times in parallel. Recall that N is the population size, l and h are

the number of parents and the number of offspring of each parent in the combined EA. For the considered algorithms, the following kernels are realized.

Kernel_select_mutate. Selection of parent individual, applying mutation and the precedence correction procedure. For the populational EA, this kernel is executed N times in parallel. For the combined EA, the selection is implemented as a separate kernel and is executed l times; the mutation and precedence correction are executed lh times in parallel.

Kernel_build_arcs. Building the graph for the Split algorithm. Here we have two levels of parallelizm: many offspring individuals and many vertices of the graph of each individual. So, the kernel function must build all the outgoing arcs from vertex i of the graph of offspring c. Collecting all the offspring and vertices together we obtain in total $N(n-1)$ parallel executions for the populational EA and $lh(n-1)$ executions for the combined one.

Kernel_find_path(i). A series of kernel functions that evaluate the length of the shortest path. Again, there are two levels of parallelism: many offspring and many arcs. One function corresponds to an offspring individual c, a vertex i, and an outgoing arc (i,j), and its action consists in updating the labels in vertex j. The kernels for vertices $i = 0, ..., n-1$ are executed sequentially one after another, and for each fixed i, the computations for all c and j are done in parallel.

The population is stored as a matrix, in which the columns represent individual. Such a way allows to increase the chances of the *coalesced access* to the GPU memory (*https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/*, see also [15]), which may significantly speed up the computations. With the indicated equipment, the developed GPU code shows about 120 times faster computing speed comparing to its CPU equivalent.

4.2. Comparison to the algorithms from the literature

The comparison to the heuristics from the literature was done on the 15 test instances from [10] with the following characteristics: $n = 200, m = 25, n_0 = 5$, $s_0 = 10$, processing and setup times are uniformly distributed numbers $t_i \in [1, 10]$, $s_{ij} \in [0, 2]$, and $T^{\max} = 50$. The precedence relations comprise from 50 to 70 arcs, the numbers of inclusion and exclusion sets are from 7 to 15, each set contains up to 4 elements. Unfortunately, the test instances from the other previous studies are not available.

The following settings are used in the tests: in the combined EA, the population size is $N = 1024$, the number of parents is $l = 64$, the number of offspring for each parent is $h = 256$. In the populational EA, $N = 5120$. The tournament size in the selection operator equals 20 for both EAs. In the Split algorithm, the bound B is defined as the average fitness of the selected parent individuals. Note that introducing this bound drastically improves the time spent on the fitness calculations, but has a side effect that this time depends on the currently achieved quality of individuals in the population and so the evaluation of the algorithms in terms of the number of iterations becomes rather meaningless.

The results are given in Table 1. On each problem instance, the algorithms made 20 independent runs. In each run the time was limited by one minute (this

is indicated in the table with mark "20 x 1 m."). Each cell in the table shows the best found fitness value (in what follows it is denoted by F), the numbers in the parenthesis show how many times the best and the next best results were obtained. For example, "27(16,4)" means that the solutions with $F = 27$ were found in 16 runs out of 20, and the solutions with $F = 28$ were found in four runs. The last two rows show the number of runs (out of 300) where feasible solutions were found and the average fitness computed over only the feasible solutions. Also, the EAs are compared to the algorithms from the literature: the GA-MIP [10] that uses CPLEX in the decoder, the Split-based ILS heuristic [11] and the steady-state GA with a different solution encoding and a greedy improvement procedure, GA-GRD [14].

Table 1: Experimental comparison of heuristics

Name	Parallel EA, 20 x 1 m.		GA-MIP	ILS	GA-GRD	GA-GRD
	Populational	Combined	20 x 10 m.	83 m.	20 x 1 m.	20 x 10 m.
a1	27(20,0)	27(16,4)	32(2,4)	28	27(13,2)	**25**(5,15)
a2	**25**(2,16)	**25**(2,16)	32(1,1)	26	26(7,9)	**25**(4,16)
a3	26(19,0)	**25**(2,17)	29(1,2)	26	–	26(19,1)
a4	25(2,18)	25(4,16)	30(8,7)	26	26(6,7)	**24**(12,8)
a5	26(11,7)	26(10,10)	31(1,4)	27	26(14,5)	**25**(2,18)
a6	26(19,1)	26(17,3)	31(1,6)	26	26(6,12)	**25**(8,12)
a7	27(17,1)	27(16,2)	31(2,0)	28	26(6,14)	**25**(4,11)
a8	**26**(3,17)	27(19,1)	31(1,5)	28	27(11,8)	**26**(13,7)
a9	26(19,1)	26(19,1)	30(2,8)	27	26(20,0)	**25**(2,18)
a10	**28**(18,2)	**28**(6,14)	33(2,5)	29	**28**(9,11)	**28**(13,7)
a11	**25**(1,19)	26(20,0)	30(4,8)	26	26(7,11)	**25**(2,18)
a12	26(1,19)	27(20,0)	31(1,8)	28	27(13,4)	**25**(1,9)
a13	26(6,5)	27(1,4)	32(1,3)	28	27(3,6)	**25**(20,0)
a14	26(14,6)	26(13,7)	30(2,5)	27	26(10,7)	**25**(15,5)
a15	**26**(15,4)	**26**(11,8)	31(2,3)	27	**26**(9,10)	**26**(20,0)
N feas	283	280	274		266	300
Avg F	26.6	26.7	32.9	27.1	27	25.9

The results for the ILS are copied from [11], where it was executed once on each instance with 5000 seconds (83.3 minutes) time limit on i7-4790 CPU. For each instance, a feasible solution was found. To avoid misunderstanding, the number of feasible solutions is not shown in the table for this algorithm.

The experiments with GA-MIP presented in [10] were obtained on the outdated CPU and with the old CPLEX version, so they were reproduced on Xeon 4210 CPU using CPLEX 12.10 and the new results are presented. On each problem, GA-MIP made 20 runs and since the MIP solution is rather time consuming, it was given ten minutes in each run. The obtained results are slightly better than in [10], although the improvement is not essential. Besides, as soon as GA-GRD is not parallel and does not rely on high performance computing, it was also interesting to evaluate it with larger execution time, so an additional experiment with GA-GRD giving it ten minutes in each run was performed.

Table 2: Experiments on large scale instances

Name	Parallel EA, 20 x 15 min.		GA-GRD
	Populational	Combined	20 x 150 min.
c1	**81**(1,2)	**81**(1,1)	87(1,0)
c2	80(2,0)	**79**(2,0)	–
c3	81(1,0)	81(2,1)	–
c4	80(2,3)	80(1,5)	**76**(2,5)
c5	83(1,0)	**81**(1,0)	–
c6	80(4,3)	**79**(1,4)	–
c7	80(3,6)	80(7,5)	–
c8	82(7,0)	80(2,3)	**78**(1,0)
c9	83(1,0)	**80**(2,0)	–
c10	79(7,0)	78(2,1)	**76**(2,0)
c11	–	**80**(2,0)	–
c12	–	–	–
c13	80(1,2)	–	**79**(1,0)
c14	79(1,10)	79(1,10)	–
c15	77(1,8)	**76**(1,7)	–
N feas	82	79	26
Avg F	80.76	80.39	78.8

As one can see, the two EAs demonstrate very similar performance and they are superior to the ILS and GA-GRD with one minute time limit. GA-MIP looks rather outdated and it cannot compete with the other algorithms even on the modern hardware. Since the ILS is based on the similar Split approach, its comparison to the EAs gives the most clear illustration of the benefits of parallelization and GPU usage. Note that the results for GA-GRD are better than the ones reported in [14] due to faster CPU (except for a3 where it could not find any feasible solution). Giving it ten times longer running time provides surprisingly good solutions, which means that the greedy heuristics is yet rather promising and it could be worthwhile to try using it in a high-performance parallel algorithm.

4.3. Evaluation on large-scale instances

In order to test the algorithms on large-scale problems, another benchmark set was generated (it is available at *https://github.com/pborisovsky/RTLBP*). The new instances contain 600 operations and 50 stations, from 2 to 7 inclusion sets, and from 26 to 36 exclusion sets. To generate precedence relations, the number of arcs was taken from 300 to 600 and the arcs were added to the precedence graph at random. If the resulting instance was infeasible due to cycles, it was neglected and a new one was generated until the required 15 instances were ready.

For this experiment, only the main competitors, i.e., the populational and the combined EA and GA-GRD are chosen, because they have shown a clear advantage over GA-MIP and ILS in the previous experiments. GA-MIP provides rather poor solutions and ILS takes quite a long time and still does not outperform the other considered heuristics. Solving larger and harder problems with GA-MIP and ILS will be very time consuming and the superior results are hardly expected.

On each of the 15 instances, the EAs were run 20 times by 15 minutes. GA-GRD could not find any feasible solutions within this limit, so it was given 150 minutes in each run. The results of comparison are given in Table 2. In these experiments, both EAs show a similar ability to find feasible solutions, but the combined EA has better approximation quality: for nine instances out of 15 it found better solutions and in five cases the results were the same. In most cases, GA-GRD could not find feasible solutions, but for four instances the outputs are better than the ones of the EAs. For problem c12 neither of the algorithm could find a solution within given time, but in an additional extensive computation, a solution with $F = 80$ was found by the combined EA after three hours.

These results illustrate the difference between the Split and the greedy approaches. While the Split based EA show much better ability to find feasible solutions, the greedy heuristic provides better results in case the feasibility is obtained. Although GA-GRD is steady-state and so it cannot be straightforwardly run on a GPU, presumably a hybrid approach combining the Split decoder and the greedy heuristic could be developed, which seems to be a promising direction for the future research.

4.4. Scalability of the two EAs

By the term *scalability* we mean an adaptiveness of an algorithm to newer generations of computing hardware so that devices with better characteristics provide faster and more accurate results and allow to treat larger problems. During 1990s – 2000s, when the main trend in the development of hardware consisted in growth of CPU speed, this property was trivially fulfilled and so it was not of great interest. It became an important issue in the area of distributed computing systems comprised of a large number of computers. Ideally, the performance of a software should grow linearly with the size of the system, which is usually not an easy task to implement in practice. For GPU computing, scalability also plays an important role. In the last decade, the performance characteristics of GPUs are rapidly grown. Comparing to the devices of late 2010s with several dozens of basic parallel processing elements (CUDA cores in Nvidia terms) modern GPUs have thousands of them and this number is expected to keep growing in the future. On the other hand, a GPU clock speed still remains in the range from 1500 to 2500 MHz. For example, the current top Nvidia device RTX 4090 has 16384 CUDA cores and a clock speed of 2230 MHz. Clearly, this means that the performance of a GPU-oriented algorithm should rely on the number of cores rather than on the clock speed.

In this section, the two implemented EAs are compared from the scalability point of view. Besides the already mentioned Nvidia Tesla V100 GPU, another device, RTX 2070 Super, was used in the experiment. It has 1770 MHz of maximal clock speed and 2560 CUDA cores. Tesla V100 has about 13.5% lower speed (1530 MHz) but twice as much CUDA cores, i.e. 5120. To see the dynamics of the search process, both algorithms were run 200 times by 1000 seconds, and in each run every second the current best value of the fitness function was stored. Then, for each second an average fitness was computed. The results are shown in

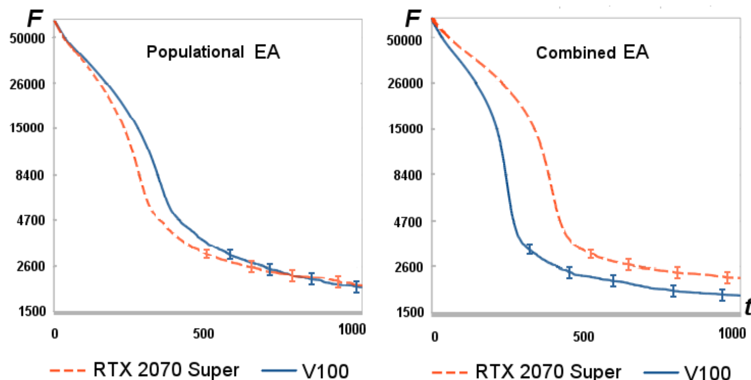

Figure 4: Comparison of two EAs on different GPUs

Figure 4. The computations were done on the instance c1. The axes represent the running time t and the average fitness value F. The 95%-confidence interval bars are shown for several time points (since the number of runs is 200, the classic large sample formula is used). Recall, that feasible solutions are very hard to obtain, so the individuals in the EAs are highly penalized, which explains the large values of F. To cope with this, a logarithmic scale is used for the y-axis. The first and the second diagram correspond to the populational and the combined EAs accordingly. On the first one we can see that the performance is very similar on the two devices. At the beginning, RTX 2070 Super provides slightly faster descent, but then two lines converge to the points very close to each other (the T-test does not show a significant difference). This means that the populational EA relies mostly on the processor speed rather than the number of cores. In contrast, the combined EA shows much better performance on V100 especially at the beginning. At the end the difference is still notable and statistically significant according to the T-test. From these results we may expect that the second EA will be more suitable for advanced GPUs with larger number of cores.

5. CONCLUSION

In this paper, a parallel GPU-accelerated evolutionary approach for one industrial machining line balancing problem is developed. Two reproduction schemes are considered: the classic populational scheme and a newly proposed one which combines ideas of populational, steady state, and memetic algorithms. Both schemes showed good improvements in terms of solutions quality and execution time comparing to the previously published iterative local search algorithm based on the same Split decoder. New hard large scale test instances are generated. On these instances, the combined EA showed a clear advantage over the populational one and the early proposed GA with a problem specific greedy heuristic.

On the other hand, this GA could achieve better approximation for moderate-size instances and for several large instances provided the running time was long enough. This observation leads to a natural idea of hybridization of the Split algorithm with the greedy heuristic in a GPU accelerated EA, which can be a promising direction for the future research.

Another issue addressed in this paper is a comparison of two parallel EAs in terms of their suitability to the GPU computing. Although the classic EA can be easily implemented on a GPU, obviously, there could be other evolutionary schemes that can better exploit the particular qualities of a GPU. In this study, it is shown that the proposed combined EA fits better for GPUs with a large number of cores and so could be more attractive for using in GPUs of next generations. It would be worthwhile to implement and test the new EA on other industrial optimization problems like routing and scheduling. In general, due to the rapid progress in development of high-performance devices, new ideas in metaheuristics design especially focusing on paralellization and scalability can be of great interest.

Acknowledgements. The computing cluster Tesla of Sobolev Institute of Mathematics, Omsk Department was used in the experiments.

Funding. This research was funded in accordance with the state task of the IM SB RAS (project FWNF-2022-0020).

REFERENCES

[1] O. Bataïa and A. Dolgui, “Hybridizations in line balancing problems: A comprehensive review on new trends and formulations,” *International Journal of Production Economics*, vol. 250, no. 5, p. 108673, 2022. doi: https://doi.org/10.1016/j.ijpe.2022.108673

[2] Ö. Hazir, X. Delorme, and A. Dolgui, “A review of cost and profit oriented line design and balancing problems and solution approaches,” *Annual Reviews in Control*, vol. 40, pp. 14–24, 2015. doi: https://doi.org/10.1016/j.arcontrol.2015.09.001

[3] P. Chutima, “A comprehensive review of robotic assembly line balancing problem,” *Journal of Intelligent Manufacturing*, vol. 33, pp. 1–34, 2022. doi: https://doi.org/10.1007/s10845-020-01641-7

[4] A. Scholl, N. Boysen, and M. Fliedner, “The assembly line balancing and scheduling problem with sequence-dependent setup times: problem extension, model formulation and efficient heuristics,” *OR Spectrum*, vol. 35, pp. 291–320, 2013. doi: https://doi.org/10.1007/s00291-011-0265-0

[5] A. Yelles-Chaouche, E. Gurevsky, N. Brahimi, and A. Dolgui, “Reconfigurable manufacturing systems from an optimisation perspective: a focused review of literature,” *International Journal of Production Research*, vol. 59, no. 21, pp. 6400–6418, 2021. doi: https://doi.org/10.1080/00207543.2020.1813913

[6] X. Delorme, A. Dolgui, M. Essafi, L. Linxe, and D. Poyard, “Machining lines automation,” in *S.Y. Nof (ed.) Springer handbook of automation.* Springer, New York, 2009, pp. 599–617. [Online]. Available: https://doi.org/10.1007/978-3-540-78831-7_35

[7] M. Essafi, X. Delorme, A. Dolgui, and O. Guschinskaya, “A MIP approach for balancing transfer lines with complex industrial constraints,” *Computers and Industrial Engineering*, vol. 58, pp. 393–400, 2010. doi: https://doi.org/10.1016/j.cie.2009.04.009

[8] M. Essafi, X. Delorme, and A. Dolgui, “Balancing machining lines: a two-phase heuristic,” *Studies in Informatics and Control*, vol. 19, no. 3, pp. 243–252, 2010. doi: https://doi.org/10.24846/v19i3y201004

[9] ——, “Balancing lines with CNC machines: A multi-start ant based heuristic,” *CIRP Journal of Manufacturing Science and Technology*, vol. 2, no. 3, pp. 176–182, 2010. doi: https://doi.org/10.1016/j.cirpj.2010.05.002

[10] P. Borisovsky, X. Delorme, and A. Dolgui, “Genetic algorithm for balancing reconfigurable machining lines,” *Computers and Industrial Engineering*, vol. 66, no. 3, pp. 541–547, 2013. doi: https://doi.org/10.1016/j.cie.2012.12.009

[11] Y. Lahrichi, N. Grangeon, L. Deroussi, and S. Norre, “A new split-based hybrid metaheuristic for the reconfigurable transfer line balancing problem,” *International Journal of Production Research*, vol. 59, no. 4, pp. 1127–1144, 2021. doi: https://doi.org/10.1080/00207543.2020.1720929

[12] T. Vidal, T. Crainic, M. Gendreau, and C. Prins, “A unified solution framework for multi-attribute vehicle routing problems,” *European Journal of Operational Research*, vol. 234, no. 3, pp. 658–673, 2014. doi: https://doi.org/10.1016/j.ejor.2013.09.045

[13] M. Haouari and M. Serairi, “Heuristics for the variable sized bin-packing problem,” *Computers and Operations Research*, vol. 36, no. 10, pp. 2877–2884, 2009. doi: https://doi.org/10.1016/j.cor.2008.12.016

[14] P. Borisovsky, “Genetic algorithm for one machining line balancing problem with setup times,” in *2020 Dynamics of Systems, Mechanisms and Machines (Dynamics), Omsk, Russia*. IEEE, 2020. doi: https://doi.org/10.1109/Dynamics50954.2020.9306146 pp. 1–5.

[15] J. Cheng and M. Gen, “Accelerating genetic algorithms with GPU computing: A selective overview,” *Computers and Industrial Engineering*, vol. 128, pp. 514–525, 2019. doi: https://doi.org/10.1016/j.cie.2018.12.067

[16] C. Schulz, G. Hasle, A. Brodtkorb, and T. Hagen, “GPU computing in discrete optimization. Part II: Survey focused on routing problems,” *EURO Journal on Transportation and Logistics*, vol. 2, no. 1-2, pp. 159–186, 2013. doi: https://doi.org/10.1007/s13676-013-0026-0

[17] T. Bäck, D. Fogel, and Z. Michalewicz, *Evolutionary Computation 1: Basic Algorithms and Operators*. CRC Press, 2000. [Online]. Available: https://doi.org/10.1201/9781482268713

[18] F. Neri, C. Cotta, and P. Moscato, *Handbook of memetic algorithms*. Springer, 2012. [Online]. Available: https://doi.org/10.1007/978-3-319-07153-4_29-1

[19] V. Hrbek and T. Brandejský, “Memetic algorithm with GPU optimization,” in *Silhavy, R., Silhavy, P., Prokopova, Z. (eds) Data Science and Algorithms in Systems. CoMeSySo 2022. Lecture Notes in Networks and Systems*. Springer, Cham, 2023. doi: https://doi.org/10.1007/978-3-031-21438-7_15 pp. 174–185.