A Parallel Branch-and-Bound for the Assembly Line Worker Assignment and Balancing Problem

Leonardo de Miranda Borba
Instituto de Informática – Universidade Federal do Rio Grande do Sul
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
lmborba@inf.ufrgs.br

Marcus Rolf Pieter Ritt
Instituto de Informática – Universidade Federal do Rio Grande do Sul
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
mrpritt@inf.ufrgs.br

ABSTRACT

The Assembly Line Worker Assignment and Balancing Problem (ALWABP) defines the optimization of assembly lines with heterogeneous workers. Heterogeneous workers take different times to execute a task or may even be incapable of executing some tasks. Furthermore, the tasks should satisfy precedence constraints. More specifically the paper address the type two of the ALWABP (ALWABP-2) which optimizes the cycle time of the assembly line for a fixed number of workers.

This paper proposes a task-oriented parallel branch-and-bound to solve the problem. Based on extensive computational tests, this article shows the efficacy of our method and it shows competitive results to the state-of-art.

KEYWORDS. ALWABP. Branch-and-Bound. Parallelism.

Main Area: Combinatorial Optimization.

RESUMO

O Problema de Atribuição de Trabalhadores e Balanceamento em Linhas de Montagem (ALWABP) define a otimização de linhas de montagem com trabalhadores heterogêneos. Trabalhadores heterogêneos demoram tempos diferentes para concluir uma tarefa ou podem até ser incapazes de executar algumas tarefas. Além disso, as tarefas devem satisfazer restrições de precedência. Mais especificamente, o artigo aborda o ALWABP do tipo 2 (ALWABP-2) que otimiza o tempo de ciclo da linha de montagem para um número fixo de trabalhadores.

Este artigo propõe um branch-and-bound paralelo orientado a tarefas. Baseado em extensivos testes computacionais o artigo apresenta a eficácia do método e mostra resultados competitivos frente aos métodos recentes da literatura.

PALAVRAS CHAVE. ALWABP, Branch-and-Bound, Paralelismo.

Área Principal: Otimização Combinatória

1. Introduction

Sheltered work centres for disabled (SWDs) are not-for-profit industries that employ mostly persons with disabilities (Miralles et al., 2007). The SWDs are supported by laws in some countries and provide job opportunities for disabled persons. In some cases the job opportunities are temporary and are responsible for the technical qualification of persons with disabilities. The creation of SWDs is a necessary measure, since there is much less participation of disabled persons in the job market than in the entire society (World Health Organization, 2011), due to the lack of public policies concerning disabled people.

The use of assembly lines has been shown to be more suitable for use in SWDs, since repetitive small tasks are an excellent therapeutic treatment for workers with severe disabilities and could decrease the differences among workers (Miralles et al., 2007). While the Simple Assembly Line Balancing Problem (SALBP) has been widely studied in the literature the Assembly Line Workers Assignment and Balancing Problem started receiving attention only recently. The SALBP is applied to traditional assembly lines. The problem considers the workers equally capable of executing tasks. Therefore, a task could be executed by any of the workers in the same amount of time. In SWDs, however, each worker executes a task in a different way and in a different amount of time or even is incapable of executing some tasks. Such situation is better described by the ALWABP (Miralles et al., 2008).

The ALWABP consists of a set of tasks T, a set of workers W and a set of workstations S, with $|S| = |W|$. The stations are allocated in sequence along a conveyor belt. We have to assign each task to each worker and each station will receive one and only one worker. The main difference from SALBP is that each worker executes a task using a different amount of time. Let us define p_{tw} corresponding to the amount of time needed by worker $w \in W$ in order to execute task $t \in T$. Also, since a worker w could be incapable of executing some task t, we define I_{tw}, which is 0 if worker w could not execute task t and 1 otherwise. As in SALBP, the partial order of tasks is given by a directed acyclic graph (DAG). The vertices of DAG $G(T, E)$ are the tasks of the problem and the edges E define dependencies among tasks. If E contains an arc (t, t') then the task t' depends on task t to be executed.

Since a SWD has a fixed number of workers for a work day, our objective is to minimize the makespan of the problem. Therefore, the production rate, and consequently the profits, of the company are maximized. Thus, the SWD could hire more workers and expand the social impact of the SWD. This type of problem in which the makespan is minimized and the number of worker is fixed is the type 2 of ALWABP (ALWABP-2).

Some exact solutions for the ALWABP-2 have been proposed (Vilà and Pereira, 2014; Borba and Ritt, 2014), yet both of them fail to fully solve larger instances in smaller time. A lot of the larger instances evaluated were not solved by both methods. Also, these two methods do not fully use the computational resources they have, since they only use one processor to solve these methods. Therefore, in this paper we propose a parallel branch-and-bound solution for the type 2 of the Assembly Line Worker Assignment and Balancing Problem.

1.1. Structure of the Paper

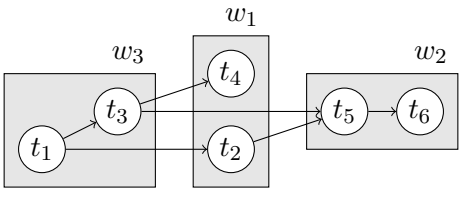
In the following sections we will present a parallel branch-and-bound for the ALWABP-2 using notation presented in Table 1. In Section 2 we will introduce a detailed definition of the problem and a Mixed Integer Programming (MIP) model for ALWABP-2. Literature on ALWABP-2 and related problems will be explored in Section 3, introducing concepts needed in the remaining sections. Section 4 will present our task-oriented branch-and-bound method and introduce its parallelization. Next, the method will be analyzed and compared with state-of-the-art methods in Section 5. At last, Section 6 will present our conclusions and a look to future work.

Table 1: Notation for ALWABP-2.

S	set of stations;
W	set of workers;
T	set of tasks;
$G(T, E)$	transitively reduced precedence graph of tasks;
$G^*(T, E)$	transitive closure of $G(T, E)$;
P_t	direct predecessors of task t in $G(T, E)$;
P_t^*	all predecessors of task t in $G^*(T, E)$;
F_t	direct followers of task t in $G(T, E)$;
F_t^*	all followers of task t in $G^*(T, E)$;
A_w	set of tasks feasible for worker w;
A_t	set of workers feasible for task t;
p_{tw}	execution time of task t by worker w;
I_{tw}	1 if worker w is capable of executing task t, 0 otherwise;
$\overline{S}_t$	the station in which task t is executed;
$\overline{W}_t$	the worker executing task t;
x_w	subset of tasks executed by worker w;
x_{tw}	1 if task t is executed by worker w, 0 otherwise;
$C \in \mathbb{R}$	cycle time of a solution.

2. Problem Definition

Let T be a set of tasks that should be executed to conclude a product. Let S be a set of workstations placed sequentially along a conveyor belt labeled $1, 2, \ldots, |S|$ according to their position in the assembly line. Finally let W be the set of workers, $|W| = |S|$. Each worker $w \in W$ is responsible to execute a subset of tasks $x_w \subseteq T$ and will be assigned to a workstation $s \in S$. The tasks are partially ordered by a directed acyclic graph $G(T, E)$. Since in a conveyor belt the tasks can only move forward and since the tasks are ordered partially, the placement of a worker affects directly the task assignment. Let us define S_t, the station executing task t. Thus, if task t is executed at station S_t, task t' at station $S_{t'}$ and task t' depends on task t, then $S_{t'} \geq S_t$. A valid solution is an assignment of workers to stations together with an assignment of tasks to workers that satisfies the precedence constraints.



p_{tw}	t_1	t_2	t_3	t_4	t_5	t_6
w_1	4	4	3	1	1	6
w_2	∞	5	6	5	2	4
w_3	3	4	2	∞	3	∞

Figure 1: Example of an ALWABP instance. The tasks precedences and an assignment of tasks to workers (in grey) are shown in the upper part. The task execution times are shown below.

We can define $D_w = \sum_{t \in x_w} p_{wt}$ as the total execution time of worker w. Also the cycle time of the line C is defined as the maximum total execution time $\max_{w \in W} D_w$ among all workers. In an assembly line we usually optimize two factors: the cycle time and the number of stations. As stated before, SWDs aim to optimize their production rate while securing job opportunities for as

many workers as possible. Therefore we will address the type 2 of the problem and minimize C while fixing the number of workers. Figure 1 shows an example of an ALWABP-2 instance. For the assignment given in the figure, we have $D_{w_1} = 5$, $D_{w_2} = 6$, $D_{w_3} = 5$, and a cycle time of $C = \max\{D_{w_1}, D_{w_2}, D_{w_3}\} = 6$.

ALWABP can be formally defined by only assigning tasks to workers. A worker graph is induced by the precedences between tasks and if this graph is acyclic it is possible to assume that the solution is valid and extract a valid assignment of workers to stations. Therefore, we define a MIP model M that uses variables x_{tw}, with $x_{tw} = 1$ if task t is assigned to worker w, and define d_{vw}, with $d_{vw} = 1$ if v precedes w in the workers induced graph (Borba and Ritt, 2014).

$$\textbf{minimize} \quad C, \tag{1}$$

$$\textbf{subject to} \quad \sum_{t \in A_w} p_{tw}\, x_{tw} \leq C, \qquad \forall w \in W, \tag{2}$$

$$\sum_{w \in A_t} x_{tw} = 1, \qquad \forall t \in T, \tag{3}$$

$$d_{vw} \geq x_{tv} + x_{t'w} - 1, \qquad \forall (t,t') \in E, v \in A_t, w \in A_{t'} \setminus \{v\}, \tag{4}$$

$$d_{uw} \geq d_{uv} + d_{vw} - 1, \qquad \forall \{u,v,w\} \subseteq W, |\{u,v,w\}| = 3, \tag{5}$$

$$d_{vw} + d_{wv} \leq 1, \qquad \forall v \in W, w \in W \setminus \{v\}, \tag{6}$$

$$x_{jw} \geq x_{tw} + x_{kw} - 1, \qquad \forall i \in T, j \in F_i^*, k \in F_j^*, w \in A_i \cap A_j \cap A_k. \tag{7}$$

$$x_{kw} + x_{iw} \leq 1, \qquad \forall i \in T, j \in F_i^*, k \in F_j^*, w \in A_i \cap (T \setminus A_j) \cap A_k. \tag{8}$$

$$x_{tw} \in \{0,1\}, \qquad \forall w \in W, t \in A_w, \tag{9}$$

$$d_{vw} \in \{0,1\}, \qquad \forall v \in W, w \in W \setminus \{v\}, \tag{10}$$

$$C \in \mathbb{R}. \tag{11}$$

The cycle time C is defined by constraint (2). Every task should be executed so that the product is concluded, as defined by constraint (3). Constraint (4) defines the induced dependencies of the workers graph. When we have two tasks t and t' and t' depends on t, then $\overline{W}_t$ should precede $\overline{W}_{t'}$ on the workers induced graph. The anti-symmetry and transitivity of the induced graph are ensured by constraints (5) and (6), respectively. Finally, two continuity constraints are specified. If two tasks are assigned to worker w, every task between them, in $G(T,E)$, should also be assigned to the same worker w, as ensured by constraint (7). Also, suppose a task t precedes a task u and task u precedes another task t', if a worker w is incapable of executing u, w can only execute one of the two remaining tasks, t and t', as assured by constraint (8).

3. Related Work

Some heuristics and meta-heuristics have been explored in ALWABP-2 literature along the past decade. Various methods have been presented to find approximate solutions to the problem. Early development included two clustering search methods (Chaves et al., 2007, 2009). Later, a simple yet efficient tabu search (Moreira and Costa, 2009), was shown to perform better than the clustering methods. Based on a preliminary station-oriented branch-and-bound (Miralles et al., 2008) an iterative beam search was also proposed (Blum and Miralles, 2011). This beam search was, later, improved with new priority rules and using a probabilistic search (Borba and Ritt, 2014). A simple procedure using a constructive heuristic was developed to test various combinations of priority rules to select tasks and workers. The results of this constructive heuristic were used as preliminary solution for a genetic algorithm (GA) (Moreira et al., 2012). A different genetic algorithm produced valid tasks ordering and applied an iterated local search to attribute tasks to workers in the selected order (Mutlu et al., 2013).

Two types of branch-and-bound algorithms were explored to solve the ALWABP-2: a station-oriented and a task-oriented method. In a station-oriented branch-and-bound, each station is filled before the next one is analyzed. First, all possibilities of filling the first station and selecting a worker for it are generated. Then, for each possibility a new branch is created and the next station is filled and so on until all tasks have been assigned (Miralles et al., 2008; Vilà and Pereira, 2014). For SALBP, station-oriented branching has been shown to be more effective (Scholl and Becker, 2006). However, the branching factor increases substantially in the ALWABP-2 because of the additional worker selection. In a task-oriented branch-and-bound, otherwise, each task is evaluated at each time and a branch is created for each worker available for the task. The task is then assigned to the worker selected for branching and the algorithm continues until a worker has been selected for each task (Borba and Ritt, 2014). Station-oriented methods are more suitable for memoization, while it is possible to make a task-oriented method that achieves good results using much less memory space. Since memory is an important bottleneck for parallel programs, in this work we implement a task-oriented branch-and-bound.

4. A Parallel Task-Oriented Branch-and-Bound Solution

Our method begins with an incumbent solution produced by a probabilistic beam search (PBS) (Borba and Ritt, 2014). If this incumbent is not proven optimal, the algorithm proceeds with a depth-first search. Algorithm 1 shows this depth-first search used in the proposed task-oriented method. It selects a task t not yet assigned (line 4) and creates a branch for each of the feasible workers (A_t) of task t in the partial solution (loop in lines 5–11). If a complete solution has already been found, the method updates the incumbent (line 1) and returns. At each node four lower bounds are applied (Borba and Ritt, 2014):

- $\mathbf{LC_1}$ assumes that all tasks are executed by the best worker available and are equally divided among workers.
- $\mathbf{LC_2}$ uses the Pigeonhole Principle to select a minimum set of tasks that should be held together.
- $\mathbf{LC_3}$ defines the earliest and latest station a task could be assigned, according to problem properties and a fixed cycle time, and if it turns to be impossible to have a solution with this cycle time, it is incremented and the algorithm continues until the condition holds.
- $\overline{\mathbf{L}}_{\mathbf{1}}^{\mathbf{a}}$ - uses a subgradient relaxation with improvements based on knapsack problem as proposed by Martello et al. (1997).

Priority rules define which task will be selected first and which worker will be branched before the others. For the tasks selection (line 4), the method chooses the one with more infeasible workers. If an assignment of a task to a worker induces an immediate cycle in the workers dependencies graph or the lower bound LC_1 is greater or equal than the value of the incumbent, the worker is called infeasible for the task. In case of tasks tied, the largest lower bound is used to select one. The lower bound of a task is given by the smallest lower bound LC_1 among its feasible workers. This rule gives preference to tasks whose lower bound is nearby the value of the incumbent and, so, the method does not waste time with bad solutions. Other ties are broken by task index. For the branches order, the one with smaller lower bound is selected first. Ties are also broken by the worker index.

To verify the precedence constraints we maintain a workers graph with the precedences induced by the tasks precedences graph. If there is a cycle in this workers graph it is possible to assume that the solution is invalid and so the method can continue. Since this operation is costly it is not applied in the tasks selection. Instead, only an evaluation of a direct cycle is applied (if a

(t, t') edge exists, an edge (t', t) can not be created). For best performance we use the data structure proposed by (Italiano, 1986) to maintain this information.

After an assignment of a task t to a worker w, some reduction rules are applied to strengthen our lower bounds (line 6). Since they are costly methods, they are executed only if no other cuts have been applied. Three reduction rules are repeatedly applied until no more tasks can be assigned or excluded:

1. Since only one worker executes a task, we can set $p_{tw'} = \infty$, for $w' \neq w$.

2. A reduction by continuity is also applied. For example, if another task t' was already assigned to w, all tasks between t and t' should also be assigned to w, as in constraint (7). If there is a task t' succeeding t assigned to w', all tasks t'' succeeding t' cannot be assigned to w and $p_{t''w}$ is set to infinity.

3. Tasks that cannot be assigned to worker w because are too distant of t in the precedence graph are also excluded and set to infinity.

4.1. Parallelization

The approach proposed in this paper modifies the depth first search and for each new branch we have two options: If all processors are occupied, the new branches are executed in the current thread. Otherwise, the new branch is sent to a new thread and the current branch continues to be executed.

Assume we have P processors. When a processors is freed, the remaining $P - 1$ processors will provide each a new branch to the new processor. The simpler solution selects the first branch and executes it on the free processor. However, a better approach is possible. Since branches with better lower bounds are more likely to produce good solutions, we chose to explore the branch with smaller lower bound in a new processor.

To implement parallelism we start P threads that wait for new branches to compute. Every time one of this branches ends, a condition variable begins to wait for the remaining threads to produce branches.

The branch-and-bound tree of our method tends to be larger near to the root. Therefore if we do not choose the better branches at the beginning, some processor could be stuck in bad branches. Also, the selection of branches is made early in the branching tree and after the processors are stabilized and executing a branch, new branches will be sent to other processors only at a deep level.

Algorithm 1: $\mathbf{branchTasks}(llb, A)$

Input : An upper bound gub, a set $A \subseteq T$ of assigned tasks, and a local lower bound llb.

```
1  if A = T then
2    if llb < gub then gub ← llb;
3    return
4  select a task t ∈ T \ A ;
5  foreach w ∈ W | assignmentIsValid(t, w) do
6    apply reduction rules;
7    newllb ← lower bound with new assignment (t, w);
8    if newllb < gub then
9      setAssignment(t, w);
10     branchTasks(newllb, A ∪ {t});
11     unsetAssignment(t, w);
```

Table 2: Instance characteristics. The 320 instances are grouped by five two-level experimental factors into 32 groups of 10 instances.

Factor	Low Level	High Level
Number of tasks $\|T\|$	$25 - 28$	$70 - 75$
Order strength (OS)	22% - 23%	59% - 72%
Number of workers $\|W\|$	$\|T\|/7$	$\|T\|/4$
Task time variability (Var)	$[1, t_i]$	$[1, 2t_i]$
Number of infeasibilities (Inf)	10%	20%

5. Results

The Parallel Task-Oriented Branch-and-Bound for the ALWABP (PTOBB) was implemented using C++ and compiled in a GNU C compiler 4.63 using the flag -O3 for maximum optimization. The threads library from Boost was used to provide parallelization. The experiments were run on a PC with a 3.60GHz AMD FX-8150 8-Core Zambezi processor and 32 GB of main memory, running a 64-bit Ubuntu Linux.

For the tests, the known instances in the literature were used (Chaves et al., 2007). Five experimental factors describe the instances: number of tasks ($|T|$), order strength (OS)[1], number of workers ($|W|$), task time variability (Var), and infeasible task-worker pairs over total pairs ratio (Inf). For each factor, there are two levels, as shown in Table 2. With these factors, 32 groups of instances are specified with 10 instances each. These instances are based on the instances defined for SALBP: *Heskia* (low $|T|$, low OS) , *Roszieg* (low $|T|$, high OS), *Wee-mag* (high $|T|$, low OS), and *Tonge* (high $|T|$, high OS). The execution time of each task p_t indicates the execution times of the first worker, and the remaining workers execute the task in a time selected in the range $[1, p_t]$, for low variability, or $[1, 2p_t]$, for high variability.

5.1. Impact of the number of processors

First of all, we will analyze the effect of multiple processors in PTOBB, the parallel branch-and-bound defined in Section 4. For the small instances Roszieg and Heskia all instances are solved and are proven to be optimal in a few seconds. Table 3 shows the number of nodes needed to prove a solution optimal (column "Nodes"), the time needed ("t(c)"), and the time needed per node ("ms/node").

For Roszieg instances, the number of nodes visited in each of the tests is equal for all cases. The time varies only in the two groups with high number of workers and high variability. The time needed to execute some instances of these two groups is slightly greater than one second for one processor. For two or more processors all Roszieg instances are solved in less than a second. For the Heskia instances, however, there is a variation of time between the one, two and four processors cases. In these cases the number of nodes, as expected, varies. Since the branching tree may be different according to the current incumbent solution, the number of processors affects the number of nodes. The time needed to prove optimality decreases from one to two processors (37.05%) and from two to three processors (22.81%) and so does the time needed per node. The eight processors case has very similar values to the four processors case. It indicates that speed up is limited to four processors, with more than that there is no speed up. After that new processors only cause an overhead of trading branches between threads. This tests show the efficacy of using multiple processors until a limit of four for small instances.

For the bigger instances, not all cases are solved in the interval of an hour. However an evolution is seen when the number of processors increases. Table 4 shows the number of solutions

[1]Order strength is the ratio between number of edges and number of feasible edges of the instance, calculated as $\binom{|T|}{2}$.

Table 3: Multiple processors results for the small instances: Roszieg and Heskia.

				Number of Processors											
				1			2			4			8		
Instance	$\|W\|$	Var	Inf	Nodes	t(s)	ms/node	Nodes	t(s)	ms/node	Nodes	t(s)	ms/node	Nodes	t(s)	ms/node
Roszieg	4	L	10%	32.0	0.0	0.00	32.0	0.0	0.00	32.0	0.0	0.00	32.0	0.0	0.00
			20%	16.5	0.0	0.00	16.5	0.0	0.00	16.5	0.0	0.00	16.5	0.0	0.00
		H	10%	41.8	0.0	0.00	41.8	0.0	0.00	41.8	0.0	0.00	41.8	0.0	0.00
			20%	37.7	0.0	0.00	37.7	0.0	0.00	37.7	0.0	0.00	37.7	0.0	0.00
	6	L	10%	119.9	0.0	0.00	119.9	0.0	0.00	119.9	0.0	0.00	119.9	0.0	0.00
			20%	79.9	0.0	0.00	79.9	0.0	0.00	79.9	0.0	0.00	79.9	0.0	0.00
		H	10%	195.8	0.3	1.53	195.8	0.0	0.00	195.8	0.0	0.00	195.8	0.0	0.00
			20%	130.2	0.1	0.77	130.2	0.0	0.00	130.2	0.0	0.00	130.2	0.0	0.00
Heskia	4	L	10%	49.6	0.0	0.00	51.1	0.0	0.00	51.1	0.0	0.00	51.1	0.0	0.00
			20%	40.1	0.0	0.00	40.4	0.0	0.00	40.3	0.0	0.00	40.4	0.0	0.00
		H	10%	48.8	0.1	2.05	48.8	0.0	0.00	48.8	0.0	0.00	48.8	0.0	0.00
			20%	57.4	0.0	0.00	57.6	0.0	0.00	59.1	0.0	0.00	59.0	0.0	0.00
	7	L	10%	15.8	1.6	101.27	15.8	1.1	69.62	15.8	0.9	56.96	15.8	0.9	56.96
			20%	15.6	2.1	134.62	15.6	1.4	89.74	15.6	1.0	64.10	15.6	1.0	64.10
		H	10%	14.7	2.2	149.66	14.7	1.3	88.44	14.7	1.0	68.03	14.7	1.0	68.03
			20%	19.3	2.5	129.53	19.3	1.5	77.72	19.3	1.2	62.18	19.3	1.2	62.18
Averages				57.2	0.6	32.46	57.3	0.3	20.34	57.4	0.3	14.68	57.4	0.3	15.70

proven optimal in an hour ("Prov.") and the gap of the solutions to the best known value in literature ("Gap").

Table 4: Multiple processors solutions for the bigger instances: Tonge and Wee-mag.

			-	Number of Processors							
				1		2		4		8	
Instance	$\|W\|$	Var	Inf	Prov.	Gap	Prov.	Gap	Prov.	Gap	Prov.	Gap
Tonge	10	L	10%	10	0.0	10	0.0	10	0.0	10	0.0
			20%	10	0.0	10	0.0	10	0.0	10	0.0
		H	10%	10	0.0	10	0.0	10	0.0	10	0.0
			20%	10	0.0	10	0.0	10	0.0	10	0.0
	17	L	10%	9	0.0	10	0.0	10	0.0	10	0.0
			20%	9	0.6	9	0.0	10	0.0	10	0.0
		H	10%	7	0.6	6	0.5	9	0.0	10	0.0
			20%	9	0.0	10	0.0	10	0.0	10	0.0
Wee-Mag	11	L	10%	3	1.8	2	1.4	4	1.1	6	0.3
			20%	1	2.0	3	2.0	5	2.0	6	0.0
		H	10%	2	2.1	2	1.9	4	1.7	6	1.3
			20%	5	1.5	5	0.9	5	0.9	8	0.0
	19	L	10%	1	3.2	3	3.2	7	2.1	9	1.1
			20%	5	0.9	6	0.9	7	0.9	7	0.9
		H	10%	5	1.9	7	0.6	9	0.6	10	0.0
			20%	5	1.7	5	1.7	7	1.1	8	0.6
Averages				101	1.0	108	0.8	127	0.6	140	0.3

-

It is possible to note in the Table 4 a clear increase of the number of solutions proven optimal and also a decreasing of the gap to the best known values according to the number of processors used. Table 5 explains this behavior. The columns are the number of nodes visited by the program ("Nodes"), the time needed to prove optimality of a solution ("t(c)"), one hour if the program reaches the time limit before finding an optimal solution) and the time executing a node ($ms/node$).

It is possible, first, to identify that the time needed to prove a solution to be optimal always decreases when the number of processors increases. Yet, the number of nodes tends to be greater according to the number of processors. The format of the branching tree depends on the incumbent solution found until the nodes are branched. Specially, in cases with wide and shallow branching trees, the incumbent solution has a huge impact. This takes place because in this type of tree bad decisions taken early affect a larger part of the tree. With multiple processors, we are searching bad branches early, without good incumbent solutions. So, especially in the case of Wee-Mag instances,

Table 5: Multiple processors results for the large instances: Tonge and Wee-Mag.

				Number of Processors											
				1			2			4			8		
Instance	$\|W\|$	Var	Inf	Nodes	t(s)	ms/node	Nodes	t(s)	ms/node	Nodes	t(s)	ms/node	Nodes	t(s)	ms/node
Tonge	10	L	10%	238756.1	242.1	1.01	207048.3	146.4	0.71	181562.9	103.7	0.57	186121.2	86.8	0.47
			20%	137843.0	161.2	1.17	148674.6	119.3	0.80	155009.9	97.0	0.63	144443.8	84.1	0.58
		H	10%	399743.5	392.0	0.98	381224.3	251.6	0.66	380483.7	151.3	0.40	394189.2	117.4	0.30
			20%	248597.5	269.4	1.08	233062.6	157.0	0.67	235285.2	121.3	0.52	233930.3	99.3	0.42
	17	L	10%	660634.5	1034.8	1.57	557170.2	521.4	0.94	608507.7	306.3	0.50	614217.8	203.1	0.33
			20%	632140.9	997.4	1.58	995805.0	773.0	0.78	1115603.9	491.0	0.44	1170254.5	301.7	0.26
		H	10%	996078.7	1595.4	1.60	1768098.1	1651.8	0.93	2181709.8	976.7	0.45	2327228.5	592.9	0.25
			20%	1017553.0	1458.0	1.43	1133200.3	925.6	0.82	1067227.8	514.2	0.48	1087780.7	328.7	0.30
Wee-Mag	11	L	10%	4254359.8	3005.6	0.71	8002007.9	3002.0	0.38	13489828.1	2499.3	0.19	22287226.0	2151.8	0.10
			20%	5138178.7	3540.2	0.69	8107863.9	3088.0	0.38	13998188.6	2658.1	0.19	20194857.4	1924.4	0.10
		H	10%	4357923.3	3024.6	0.69	8360791.3	2969.9	0.36	14361423.6	2687.0	0.19	22720737.7	2182.1	0.10
			20%	3645888.1	2681.3	0.74	6479562.4	2339.8	0.36	10811350.2	2082.8	0.19	18443534.2	1819.3	0.10
	19	L	10%	4019388.5	3258.6	0.81	6352264.0	3057.2	0.48	10071860.9	2466.8	0.24	14622372.6	1824.2	0.12
			20%	3172054.8	2572.2	0.81	5571889.8	2331.5	0.42	7823388.6	1739.9	0.22	13077777.0	1521.6	0.12
		H	10%	3087639.0	2402.9	0.78	3767251.9	1765.6	0.47	5664634.1	1342.2	0.24	7559162.1	936.4	0.12
			20%	3360713.1	2505.3	0.75	5813440.0	2373.5	0.41	7901963.9	1833.6	0.23	10974821.1	1364.3	0.12
Averages				2210468.0	1821.3	1.02	3617460.0	1592.1	0.60	5628002.0	1254.5	0.35	8502416.0	971.1	0.24

there is an increase of nodes visited. For the Tonge instances, the behavior of the number of nodes is inconclusive, since there are both increases and decreases of number of nodes when the number of processors is greater. It is also possible to perceive that the time per node is divided by two when the number of processors is doubled, showing that the nodes are well divided among processors.

5.2. Comparison with related work

Our method was compared with the two sequential state-of-art exact methods: the station-oriented branch, bound-and-remember (BBR) (Vilà and Pereira, 2014) and the task-oriented branch-and-bound (TOBB) (Borba and Ritt, 2014). Table 6 shows the results for our method (PTOBB) and the two methods in literature. The columns are similar to those of Section 5.1. Column "t(c)" specifies the time needed to execute a task (bounded to an hour if the instance lasted more than an hour). The number of solutions proven to be optimal in an hour are shown in column "Prov.". The gap between the solution found after an hour and the best known value in literature are shown in column "Gap". Since no time results were presented for the BBR method, only the times of TOBB and PTOBB are shown.

Table 6: Comparison with state-of-art methods

				Methods								
				BBR		TOBB			PTOBB		
Instance	$\|W\|$	Var	Inf	Prov.	Gap	Prov.	Gap	t(s)	Prov.	Gap	t(s)
Tonge	10	L	10%	10	0.0	10	0.0	165.4	10	0.0	86.8
			20%	10	0.0	10	0.0	134.2	10	0.0	84.1
		H	10%	10	0.0	10	0.0	362.1	10	0.0	117.4
			20%	10	0.0	10	0.0	233.9	10	0.0	99.3
	17	L	10%	0	0.3	10	0.0	789.9	10	0.0	203.1
			20%	1	0.0	9	0.0	822.5	10	0.0	301.7
		H	10%	0	0.0	7	0.3	1438.2	10	0.0	592.9
			20%	0	0.0	10	0.0	1294.8	10	0.0	328.7
Wee-Mag	11	L	10%	10	0.0	2	2.6	3316.6	6	0.3	2151.8
			20%	10	0.0	1	3.2	3534.2	6	0.0	1924.4
		H	10%	10	0.0	1	3.6	3295.5	6	1.3	2182.1
			20%	10	0.0	3	1.9	2929.8	8	0.0	1819.3
	19	L	10%	9	0.0	3	3.2	3504.6	9	1.1	1824.2
			20%	7	0.0	4	1.9	2727.8	7	0.9	1521.6
		H	10%	8	0.6	6	3.0	2251.9	10	0.0	936.4
			20%	5	0.0	4	2.3	2677.1	8	0.6	1364.3
Averages				110	0.1	100	1.4	1842.4	140	0.3	971.1

The BBR was executed on a 3.2 GHz Core i5 Processor and the TOBB on a 2.8 GHz Core i7 processor, while the PTOBB was executed on the above mentioned AMD FX-8150. The performance of the three machines differs from one another by a factor of at most 1.5. The table shows that our PTOBB produces solutions in less time than the original TOBB, even considering this factor. Also, the PTOBB solves all Tonge instances in an hour. This is not the case for the other methods. BBR, on the other hand, solves only 41 Tonge instances and TOBB solves 76. For the Wee-Mag instances, the BBR solves more instances to optimality but for the instances with high number of workers the PTOBB produces better results. Overall PTOBB is the method that proves more instances to be optimal (241). The gap for the Tonge instances is zero for the PTOBB, but for the Wee-Mag case, some instances have a small gap. The Gap of PTOBB is much smaller than the original TOBB but is still bigger than the BBR one.

6. Conclusion

This paper presented a parallel task-oriented branch-and-bound method for the ALWABP-2. The new method has been shown to prove optimality faster than the current sequential methods. The PTOBB solved 140 of the 160 instances with high number of tasks, setting a new record for the exact methods for ALWABP. The number of nodes in the branch-and-bound tree, however, increases when the number of processors increases. Therefore, an interesting future line of research consists of creating techniques to maintain the sequential branch-and-bound tree structure while using multiple processors to visit the nodes. One way would be to parallelize processes inside the nodes, while using a sequential search to navigate the branch-and-bound tree. In another scenario every node is placed in a queue sorted by the deepest nodes and each processor selects a node from this queue that simulates the tree navigation.

References

Blum, C. and Miralles, C. (2011). On solving the assembly line worker assignment and balancing problem via beam search. *Computers & Operations Research*, 38(1):328–339.

Borba, L. and Ritt, M. (2014). A heuristic and a branch-and-bound algorithm for the assembly line worker assignment and balancing problem. *Computers & Operations Research*, 45:87 – 96.

Chaves, A., Lorena, L., and Miralles, C. (2009). Hybrid Metaheuristic for the Assembly Line Worker Assignment and Balancing Problem. In Blesa, M., Blum, C., Di Gaspero, L., Roli, A., Sampels, M., and Schaerf, A., editors, *Hybrid Metaheuristics*, volume 5818 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin / Heidelberg.

Chaves, A. A., Miralles, C., and Lorena, L. A. N. (2007). Clustering Search Approach for the Assembly Line Worker Assignment and Balancing Problem. *Proc. ICC&IE*, pages 1469–1478.

Italiano, G. (1986). Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273 – 281.

Martello, S., Soumis, F., and Toth, P. (1997). Exact and approximation algorithms for makespan minimization on unrelated parallel machines. *Discrete Applied Mathematics*, 75(2):169–188.

World Health Organization (2011). *World report on disability*. Geneva: WHO.

Miralles, C., García-Sabater, J. P., Andrés, C., and Cardos, M. (2007). Advantages of assembly lines in Sheltered Work Centres for Disabled. A case study. *International Journal of Production Economics*, 110(1-2):187–197.

Miralles, C., García-Sabater, J. P., Andrés, C., and Cardós, M. (2008). Branch and Bound Procedures for Solving the Assembly Line Worker Assignment and Balancing Problem: Application to Sheltered Work Centres for Disabled. *Discrete Applied Mathematics*, 156(3):352–367.

Moreira, M. C. O. and Costa, A. M. (2009). A Minimalist Yet Efficient Tabu Search for Balancing Assembly Lines with Disabled Workers. *Anais Do XLI Simpósio Brasileiro de Pesquisa Operacional, Porto Seguro*.

Moreira, M. C. O., Ritt, M., Costa, A. M., and Chaves, A. A. (2012). Simple heuristics for the assembly line worker assignment and balancing problem. *Journal of Heuristics*, 18:505–524.

Mutlu, O., Polat, O., and Supciller, A. A. (2013). An iterative genetic algorithm for the assembly line worker assignment and balancing problem of type-II. *Computers & Operations Research*, 40:418–426.

Scholl, A. and Becker, C. (2006). State-of-the-art exact and heuristic solution procedures for simple assembly line balancing. 168(3):666–693.

Vilà, M. and Pereira, J. (2014). A branch-and-bound algorithm for assembly line worker assignment and balancing problems. *Computers & Operations Research*, 44:105 – 114.