# Parallel Cooperative Approaches for the Labor Constrained Scheduling Problem

Cristina C.B. Cavalcante * V.C. Cavalcante † Celso C. Ribeiro ‡ Cid C. de Souza §

February 2000

**Abstract**

In this paper we consider the labor constrained scheduling problem (LCSP), in which a set of jobs to be processed is subject to precedence and labor requirement constraints. Each job has a specified processing time and a labor requirements profile, which typically varies as the job is processed. Given the amount of labor available at each period, the problem consists in determining starting times so as to minimize the overall makespan, subject to the precedence and labor constraints. We propose two parallel cooperative algorithms for LCSP: an asynchronous team and a parallel tabu search strategy. Both algorithms make use of cooperative processes that asynchronously exchange information gathered along their execution. Computational experiments on benchmark instances show that these parallel algorithms produce significantly better solutions than all sequential algorithms previously proposed in the literature.

**Keywords:** project scheduling, labor constraints, heuristics, parallel tabu search, asynchronous teams.

## 1 Problem definition

In broad terms, the labor constrained scheduling problem (LCSP) involves sequencing a set of jobs on multiple machines, subject to precedence constraints given in terms of a direct (acyclic) graph. Each job has a specified processing time and a labor requirements profile, which typically varies as the job is processed. Given the amount of labor available at each time period, the problem consists in finding a job sequence within each machine, so as to minimize the completion time of the last job to be finished (i.e., the makespan), subject to the precedence and labor constraints.

The particular version of LCSP motivating this study is a simplification of an industrial problem from BASF A.G. This problem is NP-hard and was first discussed in [20]. Since then, some methods have been proposed for its solution: constraint programming [21], linear programming-based heuristics [29, 28], sequential tabu search [5], and integer programming [30]. The last three approaches are discussed in [6].

*Institute of Computing, Universidade Estadual de Campinas, Caixa Postal 6176, Campinas, SP 13083-970, Brazil. Research supported by FAPESP grant 96/10270-8

†Institute of Computing, Universidade Estadual de Campinas, Caixa Postal 6176, Campinas, SP 13083-970, Brazil.

‡Catholic University of Rio de Janeiro, Department of Computer Science, R. Marquês de São Vicente 225, Rio de Janeiro, RJ 22453-900, Brazil. Research supported by CNPq (grants 302281/85-1 and 202005/89-5) and FAPERJ (grant 150966/99). E-mail: celso@inf.puc-rio.br

§(Corresponding author) Institute of Computing, Universidade Estadual de Campinas, Caixa Postal 6176, Campinas, SP 13083-970, Brazil. Research supported by CNPq (grant 300883/94-3) and FINEP (ProNEx-107/97). E-mail: cid@dcc.unicamp.br

The main features of the LCSP instances from the industrial application cited above are the following. We are given a set $J$ of $n$ jobs. Each job $j$ is formed by a sequence of $p_j$ unit-duration tasks. All such tasks must be processed immediately one after the other and therefore, job preemption is not permitted. The labor requirements profile is specified by a vector $(\ell_{j,1}, \ell_{j,2}, \ldots, \ell_{j,p_j})$, where $\ell_{j,s}$ denotes the numbers of workers needed to perform the $s$-th task of job $j$. Typically labor requirements take values in the set $\{2, 3, 6, 9, 12, 18\}$ while the total amount of workers $L$ available at any time instant, is either 18 or 24. The jobs are partitioned into groups called *orders*. An order is composed by a set of jobs with identical labor profiles and linked by a long path in the precedence graph. A few precedence constraints exist between jobs on different orders. Besides, all the jobs in an order are executed in a single machine and the total number of orders is small enough so as that each machine can be assigned to a unique order. All machines are considered to have the same speed.

A solution to LCSP is a schedule given by the starting times of all jobs in $J$. A schedule is feasible if the starting times of the jobs respect the precedence relations and, for each instant $t$ in the planning horizon, the total amount of workers required by all jobs being executed at $t$ is not greater than $L$. The problem is then to find a feasible scheduling that minimizes the makespan. Figure 1 illustrates a small example of such an instance.

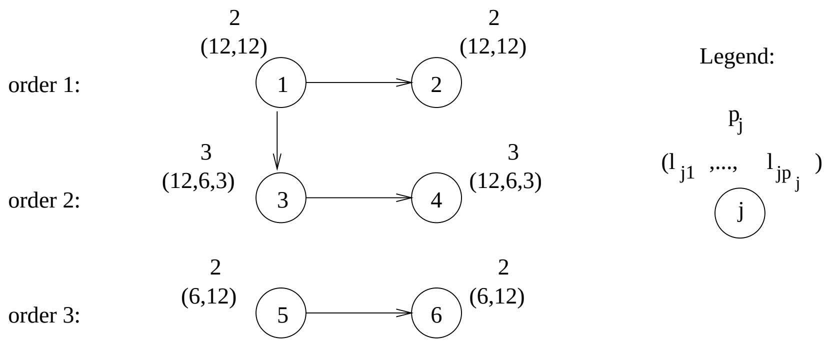



Figure 1: An instance of LCSP.

We notice that there exists a *reverse instance*, associated to each instance of LCSP. The reverse instance is obtained from the original one by reversing the labor requirements profiles of all the jobs and the direction of all precedence constraints (arcs of the digraph). It is not hard to see that feasible schedules of the original instance and the reverse one are in a one-to-one correspondence. Thus, the optimal makespan remains the same independently of whichever of these two instances is solved. This fact is explored in some algorithms for LCSP.

In this paper, we present two parallel cooperative algorithms for LCSP: an asynchronous team and a parallel implementation of tabu search. Both of them make use of cooperative processes that asynchronously exchange information gathered during their execution. The goal is to improve on the individual performance of each process by embedding them within a cooperative team of algorithms. The paper is organized as follows. Sections 2 and 3 describe, respectively, the asynchronous team and the parallel tabu search for LCSP. Section 4 presents some computational results obtained with the application of these two algorithms to a set of benchmark instances of LCSP. Finally, in Section 5 we draw some concluding remarks.

## 2 An Asynchronous Team for LCSP

An *asynchronous team* (a-team) is a software organization in which multiple algorithms, each acting as an autonomous agent, cooperate with each other exchanging information through shared memories

[31, 34] (see also [2, 11, 25, 32, 35] about references and applications of asynchronous teams, as well as [27] for a similar approach on another context). The main purpose of this cooperation is to increase the chances of finding better solutions than those produced by each agent acting alone. Usually, there are four types of agents in an a-team designed to solve a specific combinatorial problem:

- Construction agents: algorithms that generate complete or partial solutions of the problem;
- De-construction agents: algorithms that generate a partial solution from one or more complete solutions;
- Improvement agents: local search algorithms that modify complete solutions; and
- Destroyer agents: algorithms that remove solutions from shared memories.

In a generic a-team, these four types of agents are continuously active: new solutions are built, existing solutions are de-constructed or modified, and solutions of low quality are destroyed. Shared memories store complete or partial solutions and can be accessed at any moment by all agents.

Graphically, an a-team is represented by a set of arcs and boxes, which refer to agents and memories, respectively. Figure 2 shows an example of a-team. Agents $A$ and $B$ read from memory 1 and write in memories 1 and 3 respectively. Agent $C$ reads from memory 2 and writes in memory 1. Agent $D$ reads in memory 3 and can write either in memory 1 or in memory 2 (this situation is represented by the large box containing both box 1 and box 2). Agent $E$ reads from memory 2 and writes in memory 3. Agent $F$ initializes memories 1 and 2, while agent $G$ is a destroyer responsible for deleting solutions from those two memories.

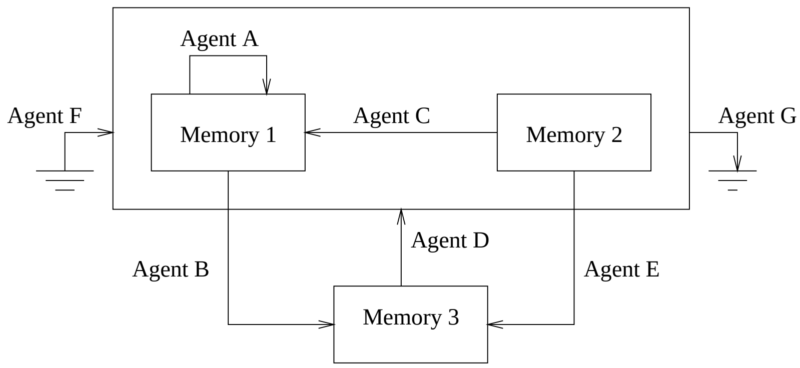



Figure 2: A-team example.

An important characteristic of an a-team is that data (i.e., solutions of the problem to be solved) cycle within its structure and are renewed along this process. Agents retrieve, modify, and store information in shared memories. Thus, solutions generated by an algorithm are potentially reused as input for other algorithms of the team. This continuous flow of information allows the interaction between the agents and, eventually, leads to a refinement of the solutions generated.

It is worth noticing that the a-team structure is particularly well suited for distributed computing. The autonomy of the agents, the asynchronous communication, and the absence of a centralized control allow the simultaneous execution of the different algorithms on the many processors available in a computer network. Successful applications of this technique to other hard combinatorial problems were reported in [7, 19, 23, 31] and motivated its investigation in the context of the LCSP.

The a-team we proposed for LCSP is depicted in Figure 3. There are two shared memories: "Complete Solutions" memory (CSM) and "Partial Solutions" memory (PSM). An initializer agent generates the initial set of complete solutions. De-construction agents read solutions from CSM and generate partial solutions that are stored in PSM. Construction agents read solutions from PSM, complete them, and write the new solutions obtained in CSM. Improvement agents read solutions from CSM, modify them, and write the improved solutions in CSM. A destroyer agent is responsible for removing bad solutions from CSM.

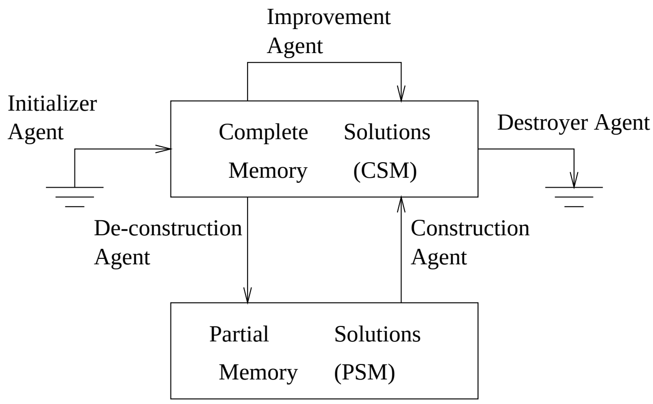



Figure 3: A-team for LCSP.

We associate a job sequence $s$ with each solution of an LCSP instance. However, this sequence alone is not enough to represent a solution, unless a precise rule is given that generates a feasible schedule from it. Given a job sequence $s$, once can build a solution by scanning the jobs in the order they appear in $s$ and scheduling them at the earliest possible time so that all precedence and labor constraints are satisfied. Another solution can be built by scanning the jobs in $s$ in reverse order, considering the reverse instance. Finally, we associate with sequence $s$ the schedule with minimum makespan among the two above described solutions. Thus, the cost of any given sequence $s$ can be easily computed in time $O(nPT)$, where $P$ and $T$ denote respectively the duration of the longest job and the time horizon. However, we observed in practice that this cost computation takes linear time in the number of jobs $n$. In the remaining of the text, we also refer to a job sequence $s$ as a solution for LCSP since, according with the above rule, it corresponds to a unique schedule of the jobs on the machines. Using this solution representation, the processes of the proposed a-team for LCSP are defined as follows.

## 2.1 Process ATC (complete solutions memory)

This process has five basic functions: to initialize and to maintain the CSM; to initialize all other processes of the a-team; to send complete solutions to de-construction and improvement agents; to receive complete solutions from construction and improvement agents; and to terminate the computation. These functions are detailed below.

**CSM initialization.** The CSM is maintained by ATC as a list with `CSM_SIZE` elements. The first step of ATC initializes the list with the following solutions: the best solution generated by a greedy heuristic based on a priority rule; the best solution obtained by a heuristic based on classes of schedules; and the remaining solutions generated by a heuristic with a randomized priority rule. Priority rule based greedy heuristics construct schedules from scratch. At each step, they select the next job to schedule as the one minimizing some pre-defined greedy function representing the priority rule. Typical functions (cf. [15, 22]) are:

- minimum slack: choose the job with minimum *slack time*, i.e., the job minimizing the difference between the latest and earliest starting times computed from the precedence relations and the planning horizon.

- longest critical path: choose the job which together with its successors in the precedence graph forms an induced subgraph with the largest critical path.
- longest order duration: choose the job belonging to the order with the largest total duration.
- most remaining order duration: choose the job belonging to the order with the largest remaining duration (i.e., the sum of the processing times of the currently unscheduled jobs in the order).
- shortest processing time: choose the job with the smallest processing time.
- least resource demand: choose the job requiring less workers.
- random: randomly choose the job.

The initialization strategy allows for both diversity and good quality solutions to start the search. Solutions are stored in increasing order of their costs and no repetitions are allowed.

**A-team initialization.** After the initialization of the CSM, process ATC initializes the remaining processes of the team: ATP (partial memories); IBC and DBC (de-construction agents); HND, HH, and SPLCA (construction agents); and IMPJ, IMPCP, IMPALL, and IMPSWP (improvement agents). Next, ATC starts its autonomous execution by (i) waiting for new complete solutions sent by construction or improvement agents, or (ii) sending already solutions to de-construction or improvement agents.

**Solution reception.** Whenever a construction or improvement agent finds a good quality complete solution, a message is sent to process ATC containing solely the cost of this solution. If this cost is not larger than $1 + \gamma\%$ times the cost of the best solution in the CSM, then ATC sends a message to the calling agent informing that it is ready to receive the actual solution, represented by a job sequence. The threshold parameter $\gamma > 0$ is used to avoid poor quality solutions in CSM, as well as to cut the amount of long messages sent to ATC containing solutions which could prevent it from quickly treating other messages requests. Solutions received by ATC which are already in CSM are discarded. Otherwise, a solution from CSM is chosen and replaced by the new one. The choice of the solution to be eliminated follows a linear distribution probability: the best solution has a null probability of being eliminated, while for the others this probability increases linearly up to the worst solution in CSM. Reports on computational experiments with other combinatorial problems show that this strategy maintains the diversity of the solutions in CSM, while keeping high quality solutions, see e.g. [7].

**Solution sending.** ATC sends messages with one complete solution to improvement agents and with two complete solutions to de-construction agents. The solutions to be sent are randomly chosen, so as that all solutions have equal chances of being treated by those agents.

**Termination.** The decision of when to terminate the computation is left to process ATC. Whenever `MAX_SOLUTIONS` have been accepted to enter into the CSM, ATC stops the execution of all remaining processes. Afterwards, it halts its own execution and returns the best solution in the CSM.

### 2.2 Processes IBC and DBC - De-construction Agents

A de-construction agent starts with one or more complete solutions and produces a partial solution. In the case of LCSP, a partial solution is represented by a set of subsequences of jobs. The proposed a-team has two de-construction agents, both based on consensus algorithms [31]. A consensus algorithm tries to capture similarities or dissimilarities between two or more good quality solutions.

The first de-construction agent is implemented via the IBC process and is based on an *intersection criterion*, while the second is implemented by process DBC and is based on a *difference criterion*. These processes are continuously asking process ATC for complete solutions from CSM. The partial solutions generated are then sent to process ATP (cf Subsection 2.3).

We now explain the two criteria used to implement the consensus algorithms. Let $s_a$ and $s_b$ be sequences representing two different solutions. The intersection criterion creates a set of subsequences in which $s_a$ and $s_b$ coincide. For the difference criterion, assume that $s_a$ has a better cost than $s_b$. Then, a partial solution is created which contains the subsequences in $s_a$ that do not appear in $s_b$. In both cases the comparison between the sequences is done in a position by position basis. Thus, it is straightforward to implement these algorithms with time complexity $O(n)$. These ideas are illustrated in Figure 4

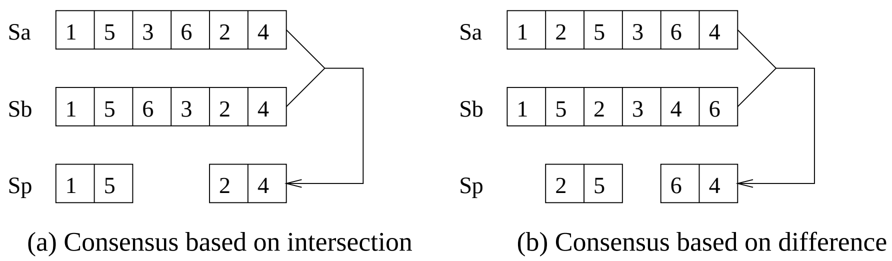



Figure 4: Generating a partial solution $s_p$ using the intersection and difference consensus criteria.

### 2.3 Process ATP - PSM Management

This process handles the management of the partial solutions memory. It acts like a buffer that receives partial solutions from the de-construction agents and, whenever requested, sends them to the construction agents. PSM is initially empty. Each element is composed by a set of subsequences of jobs. PSM grows as partial solutions are received from de-construction agents, and diminishes when solutions are sent to construction agents. Notice that it may be the case that a construction agent requests a partial solution for ATP when the PSM is empty, in which case the request cannot be served.

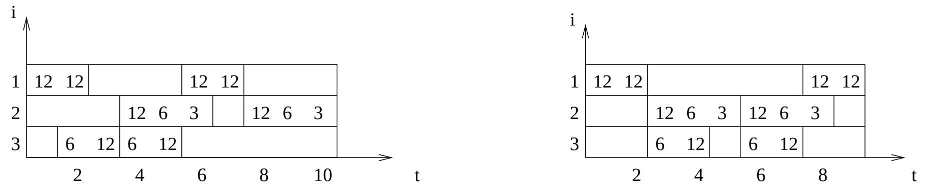
### 2.4 Processes HND, HH, and HA - Construction Agents

The goal of the construction agents in the a-team is to build complete solutions, starting from partial ones. Jobs not belonging to the partial solution enter it on a one by one basis in an order determined by a complete solution previously generated by one of the heuristic methods specified below. Each of these jobs is inserted in the current partial solution in such a way that (i) subsequences from the partial solution are not broken by the insertion of new jobs, and (ii) the new job must be inserted prior to its successors and after all its predecessors.

We now describe the heuristics that generate the job sequence that guides the construction agents. Schedules produced by these methods can be classified as follows:

- non-delay schedules: schedules where no resource (worker) remains unused (idle) if there exists a job that can start earlier and use it.
- active schedules: schedules where no job can start earlier without delaying the starting of another job or violating the labor constraints.

Figure 5 illustrates these two definitions for schedules generated for the example in Figure 1.




(a) Non-delay schedule (b) Active schedule

Figure 5: Classes of schedules.

The heuristics are based on the classical algorithms of Giffler and Thompson (see e.g. in [3]) which generate respectively all active and non-delay schedules for the job shop scheduling problem. We have adapted them to return a single active or non-delay schedule for the LCSP. To describe the heuristics we introduce the following notation:

$\sigma_j$: earliest time that job $j$ can start, considering precedence and labor constraints;

$\phi_j = \sigma_j + p_j$: earliest time that job $j$ can finish, considering precedence and labor constraints;

$P_k$: a partial schedule with $k$ jobs; and

$S_k$: the set of schedulable jobs at step $k$, i.e. jobs whose predecessors already belong to $P_k$.

The pseudo-codes of heuristics HA and HND which produce, respectively, active and non-delay schedules for LCSP, are given below. We notice that both algorithms make use in step 4 of one of the priority rules described in Subsection 2.1 to break ties, whenever necessary.

<u>Heuristic HA</u>:
1. Set $k \leftarrow 0$, $P_k \leftarrow \emptyset$, and $S_k \leftarrow$ {jobs with no predecessors}.
   Initialize the amount of resources available at any time instant $u = 1, \ldots, T$ with $labor[u] = L$.
2. For all $j \in S_k$ do:
   $\sigma_j \leftarrow \max\{e_j, \min\{t : \ell_{j,u-t+1} \leq labor[u]\ \forall u = t, \ldots, t + p_j - 1\}\}$
   $\phi_j \leftarrow \sigma_j + p_j$
3. Compute $\phi^* = \min\{\phi_j : j \in S_k\}$.
4. Choose $j' \in S_k$ such that $\sigma_{j'} < \phi^*$. Ties are broken by one of the priority rules of Subsection 2.1.
   If ties persist, they are broken by a random choice.
5. Build a partial schedule $P_{k+1}$ by adding $j'$ to $P_k$ with starting time $\sigma_{j'}$.
6. (a) Set $labor[u] \leftarrow labor[u] - \ell_{j',u-\sigma_{j'}}$ $\forall u = \sigma_{j'}, \ldots, \sigma_{j'} + p_{j'} - 1$.
   (b) Set $e_j \leftarrow \max\{e_j, \sigma_{j'} + p_{j'}\}$ for every successor $j$ of job $j'$.
   (c) $k \leftarrow k + 1$
   (d) Set $S_k \leftarrow$ {all jobs not yet scheduled whose predecessors are in $P_k$}.
7. If $k < n$ go back to step 2. Otherwise, stop.

<u>Heuristic HND</u>:
1. Set $k \leftarrow 0$, $P_k \leftarrow \emptyset$, and $S_k \leftarrow$ {jobs with no predecessors}.
   Initialize the amount of resources available at any time instant $u = 1, \ldots, T$ with $labor[u] = L$.
2. For all $j \in S_k$ do:
   $\sigma_j = \max\{e_j, \min\{t : \ell_{j,u-t+1} \leq labor[u]\ \forall u = t, \ldots, t + p_j - 1\}\}$
3. Compute $\sigma^* = \min\{\sigma_j : j \in S_k\}$.
4. Choose $j' \in S_k$ such that $\sigma_{j'} = \sigma^*$. Ties are broken by one of the priority rules of Subsection 2.1.
   If ties persist, they are broken by a random choice.
5. Build a partial schedule $P_{k+1}$ by adding $j'$ to $P_k$ with starting time $\sigma_{j'}$.
6. (a) Set $labor[u] \leftarrow labor[u] - \ell_{j',u-\sigma_{j'}}$ $\forall u = \sigma_{j'}, \ldots, \sigma_{j'} + p_{j'} - 1$.
   (b) Set $e_j \leftarrow \max\{e_j, \sigma_{j'} + p_{j'}\}$ for every successor $j$ of job $j'$.
   (c) $k \leftarrow k + 1$
   (d) Set $S_k \leftarrow$ {all jobs not yet scheduled whose predecessors are in $P_k$}.
7. If $k < n$ go back to step 2. Otherwise, stop.

Finally, heuristic HH can be viewed as the generation of hybrid schedules between non-delay and active ones. This idea was introduced by Storer, Wu, and Vaccari [33] and implemented via an adaptation of the algorithms of Giffler and Thompson. This is done through the introduction of a variable $\delta \in [0, 1]$, such that for $\delta = 1$ the algorithm generates an active schedule, otherwise it generates a hybrid schedule. Details are given in the pseudo-code below. The motivation for using this strategy is to increase the diversity of the solutions stored in the memory.

Heuristic HH:

1. Set $k \leftarrow 0$, $P_k \leftarrow \emptyset$, and $S_k \leftarrow$ {jobs with no predecessors}.
   Initialize the amount of resources available at any time instant $u = 1, \dots, T$ with $labor[u] = L$.
2. For all $j \in S_k$ do:
   $\sigma_j = \max\{e_j, \min\{t : \ell_{j,u-t+1} \leq labor[u]\ \forall u = t, \dots, t + p_j - 1\}\}$
3. Compute $\sigma^* = \min\{\sigma_j : j \in S_k\}$ and $\phi^* = \min\{\phi_j : j \in S_k\}$
4. Choose $j' \in S_k$ such that $\sigma_{j'} \leq \sigma^* + \delta(\phi^* - \sigma^*)$. Ties are broken by one of the priority rules of Subsection 2.1.
   If ties persist, they are broken by a random choice.
5. Build a partial schedule $P_{k+1}$ by adding $j'$ to $P_k$ with starting time $\sigma_{j'}$.
6. (a) Set $labor[u] \leftarrow labor[u] - \ell_{j',u-\sigma_{j'}}\ \forall u = \sigma_{j'}, \dots, \sigma_{j'} + p_{j'} - 1$.
   (b) Set $e_j = \max\{e_j, \sigma_{j'} + p_{j'}\}$ for every successor $j'$ of job $j$.
   (c) $k \leftarrow k + 1$
   (d) Set $S_k \leftarrow$ {all jobs not yet scheduled whose predecessors are in$P_k$}.
7. If $k < n$ go back to step 2. Otherwise, stop.

The three heuristics run quite fast in time complexity $O(n^2 + T)$, assuming that the largest job duration is bounded by a constant. Nine different values for $\delta = 0.1, 0.2, \dots, 0.9$ are used for heuristic HH and the best solution among all runs is retained. Preliminary tests have indicated that heuristic HH leads to better results when the ties are broken in step 5 using a priority rule based on the longest critical path.

Construction agents continuously ask process ATP for partial solutions. Whenever one of such requests is satisfied, the corresponding agent generates a complete solution according to the associated heuristic. Next, it sends a message containing the cost of this complete solution to process ATC. If ATC accepts the cost of this solution, the corresponding sequence of jobs is sent by the construction agent to it.

### 2.5 Processes IMPJ, IMPCP, IMPALL, and IMPSWP - Improvement Agents

The improvement agents designed for the LCSP continuously send messages to process ATC, requesting complete solutions. Whenever they receive a solution, they use it to start a local search procedure. If the local search encounters a better solution, its cost is sent to process ATC process. As before, the latter checks whether the acceptance criterion is fulfilled, in which case the new complete solution is sent to it.

Four improvement agents have been implemented. They differ by the neighborhoods they use for local search. Neighborhoods are characterized by two basic move operations. Given a job sequence $s$ and two jobs $i$ and $j$, movement *Insert*$(i, j)$ inserts job $j$ immediately in front of job $i$ in the sequence. Movement *Swap*$(i, j)$ interchanges the positions of jobs $i$ and $j$ in the sequence. Only movements leading to feasible schedules are permitted, i.e. a movement is performed solely if the precedence constraints are obeyed.

To detect moves violating precedence constraints, we proceed as follows. First, for each job $j$ we pre-compute the set $PPL_j$ of jobs which are neither successors nor predecessors of $j$. Given a feasible sequence $s$, the forbidden movements are: (i) *Insert*$(i, j)$ for all pairs $i$-$j$ such that $j$ is a successor of

$i$ not belonging to $PPL_i$, and (ii) *Swap*$(i,j)$ for all pairs $i$-$j$ such that $j$ does not belong to $PPL_i$. However, avoiding the two types of movements in (i) and (ii) is not enough to guarantee that the new sequence corresponds indeed to a feasible solution. This is because we still depend on the positions of the predecessors and successors of jobs $i$ and $j$, which can only be computed in running time, when the actual sequence $s$ is known. Straightforward routines running in time $O(n)$ were implemented to check the feasibility of *Insert* and *Swap* movements.

Thus, given a job sequence $s$, the following neighborhood variants are used in each of the four improvement agents of the a-team:

- *IMPJ*: solutions obtained by applying all possible insertion movements *Insert*$(i,j)$ to a randomly chosen job $j$ ($O(n)$ movements);
- *IMPCP*: solutions obtained by applying the insertion movement *Insert*$(i,j)$ to every pair $i$-$j$ of jobs such that one of them is in the critical path of the precedence graph ($O(n^2)$ movements);
- *IMPALL*: solutions obtained by applying the insertion movement *Insert*$(i,j)$ to every pair $i$-$j$ of jobs ($O(n^2)$ movements); and
- *IMPSWP*: solutions obtained by applying all possible swap movements *Swap*$(i,j)$ to a randomly chosen job $j$ ($O(n)$ movements).

The improvement agents are based on the computation of best-improving moves for each local search iteration. As explained before, the makespan computation for each solution can be done in time $O(n)$ and, therefore, the cost of exploring the neighborhood is $O(n^2)$ (resp. $O(n^3)$) for *IMPJ* and *IMPSWP* (resp. *IMPCP* and *IMPALL*).

A detailed schema of the a-team designed for the LCSP is given in Figure 6. Computational results will be reported in Section 4.

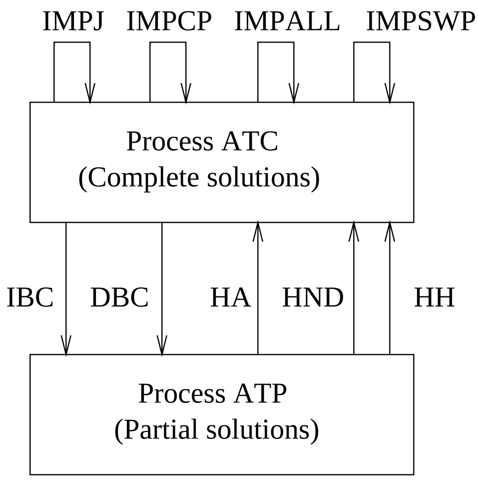



Figure 6: A-team for LCSP (detailed).

## 3 Parallel Tabu Search for LCSP

Tabu search is an adaptive metaheuristic for solving combinatorial optimization problems, which guides local search methods to continue the search beyond local optimality [16, 17, 18]. It makes use of flexible adaptive memory to modify the neighborhood of the current solution as the search progresses. Moves toward solutions deteriorating the current one are accepted under certain circumstances. Parallel tabu search implementations described in the literature are a promising alternative to further improve solution quality and algorithm robustness. Successful applications and implementations of tabu search are described e.g. in [1, 9, 10, 12, 13, 14, 24, 26, 36, 37].

Cung et al describe and compare strategies for the parallel implementation of metaheuristics in [14], providing a classification scheme based on that originally proposed in [38] for the parallelization of local search heuristics. Strategies for the parallelization of tabu search range from the speedup of some phases of the algorithm to its complete redesign, in which several searches are performed in parallel and cooperate along their execution. In the case of *single-walk* strategies, a unique search path is explored. Either the neighborhoods or the problem domain are distributed among the processors. In the case of strategies based on neighborhood partition, each processor explores part of the neighborhood and the best move overall is retained. Within the class of *multiple-walk* strategies, each processor performs a potentially different sequential algorithm, using different parameters or starting from different initial solutions. We make the distinction between algorithms that perform independent searches and those that cooperate. In the case of *independent* strategies, processes do not communicate and the best solution is collected among those found by each processor. Process communicate within *cooperative* strategies by exchanging high quality solutions and other relevant information gathered along their trajectories, so as to drive the search towards more promising regions of the solution space. Single- and multiple-walk strategies can be implemented as *synchronous* or *asynchronous* algorithms.

The parallel tabu search procedure described in Subsection 3.2 follows an asynchronous cooperative multiple-walk strategy. Processors performing different sequential tabu search algorithms exchange information in an asynchronous manner, through a central memory which is handled by a central processor. Therefore, this algorithm acts very much like as an a-team. Indeed, it can be viewed as an a-team whose agents are sequential tabu search algorithms. The TSLCSP sequential tabu search algorithm for LCSP on which this team is based was introduced and described in details in [6]. Its main characteristics are summarized below.

### 3.1 Sequential Tabu Search

Solution representation and neighborhood definitions are the same as those described in Section 2. In broad terms, our sequential short-term-memory-based tabu search algorithm TSLCSP can be described as follows. Starting from an initial schedule, it proceeds by choosing at each iteration the best admissible movement (i.e., either an insertion or a swap movement which is not prohibited or satisfies the aspiration criterion) from a set of candidate ones. Only movements leading to feasible schedules are considered. At the end of each iteration, the current solution is replaced by the new schedule obtained by local search. Algorithm TSLCSP returns the best schedule found over all iterations. We now briefly discuss some components of this algorithm.

**Initial solution:** Initial solutions are generated by three different approaches:

- *Best_PRH*: the best solution generated by the greedy heuristics with the priority rules defined in Section 2.1;
- *Best_SSH*: the best solution generated by the heuristics based on classes of schedules discussed in Section 2.4; and
- *Sol_PLH*: a solution obtained from the optimal solution of the linear program relaxation of LCSP (see [6] for details).

**Objective function:** Two alternative objective functions are considered. The first one certainly is the makespan, denoted here by $F_1$. The second function is denoted by $F_2$ and is defined as follows. For each job $j$ in a schedule $s$, let $\sigma_j$ be its starting time, $p_j$ its duration, and $\alpha_j$ the size of the critical path induced in the precedence graph by job $j$ and its successors. Thus,

$$F_2(s) = \sum_{j=1}^{n} \alpha_j \sigma_j + (F_1(s)+1) \sum_{j \in n} T\alpha_j,$$

where $T$ is the horizon under consideration and $F_1(s) = \max\{\sigma_j + p_j : j = 1, \dots, n\}$ is the makespan.

The objective function $F_2$ was first introduced in [6]. A simple proof by contradiction shows that an optimal schedule with respect to the above objective function is also optimal with respect to the makespan (see [8] for details). We remind that, in any case, the corresponding schedule associated with a feasible sequence is computed by taking the least cost schedule between those obtained from the direct and the inverse instances.

**Aspiration criterion:** A tabu restriction can be dropped whenever the corresponding move leads to a schedule better than the currently best.

**Stopping criteria:** The tabu search algorithm stops whenever one of these two conditions is detected: either a maximum number `MAX_TOTAL_ITER` of iterations is attained or a maximum number `MAX_BAD_MOVES` of iterations are performed without improvement in the best solution found.

**Post-optimization:** We try to improve on the best solution by applying a deterministic local search based on insertion movements. For each job $j$, all insertion movements *Insert*$(i, j)$ are examined, where $i$ is a direct successor of $j$ or $i \in PPL_j$ (so as to avoid the generation of neighbors which violate the precedence constraints).

**Neighborhood:** The different movement definitions considered here lead to different neighborhood definitions. To speedup the neighborhood search, our strategy only considers a candidate set formed by movements leading to a restricted subset of the full neighborhood. The search does not consider all possible pairs of jobs $i$-$j$ for insertions and swaps, but instead just a restricted subset of pairs. Notice that insertion and swap movements are the same as defined in Section 2 for the improvement agents of the a-team but, for the reasons given above, they are used to define smaller neighborhoods.

Different neighborhood definitions, together with different tabu restrictions and tabu tenures, were considered and tested in [6]. Among all configurations compared in that study, eight configuration have been retained, as those which empirically have been shown to lead to the best solutions. Those configurations are briefly summarized below. As before, $s$ is defined as the current sequence of jobs representing a feasible solution.

- *TSLCSP1:* We use a restricted neighborhood in which we explore the ten first feasible solutions obtained from $s$ by applying insertion movements *Insert*$(i, j)$ to pairs of randomly chosen jobs $i$-$j$. The tabu restriction imposes that job $j$ cannot be moved for a certain number (tabu tenure) of iterations, corresponding to an integer value randomly chosen in the interval $[\lfloor 0.5\sqrt{n} \rfloor, \lceil 0.8\sqrt{n} \rceil]$.
- *TSLCSP2:* The neighborhood definition is the same as for *TSLCSP1*. However, in this case the tabu restriction prohibits job $j$ to be inserted immediately before job $i$ in the sequence and a dynamic tabu tenure randomly chosen in the interval $[\lfloor 1.2\sqrt{n} \rfloor, \lceil 1.5\sqrt{n} \rceil]$ is used.
- *TSLCSP3:* In this case, we explore a restricted neighborhood in which the ten first feasible solutions obtained from $s$ by applying swap movements to pairs of randomly chosen jobs $i$-$j$ are investigated. Both the tabu tenure and the tabu restriction are the same adopted for *TSLCSP1*,

but the latter is extended also to encompass the prohibition of movements involving job $i$. Movements involving anyone of jobs $i$ and $j$ are forbidden.

- *TSLCSP4:* The neighborhood definition is the same as for *TSLCSP3*. The tabu restriction prevents jobs $i$-$j$ from being swapped for a certain number of iterations randomly chosen in the interval $[\lfloor 0.9\sqrt{n}\rfloor, \lceil 1.1\sqrt{n}\rceil]$.
- *TSLCSP5:* The restricted neighborhood encompasses all feasible solutions which can be obtained from the current solution $s$ by applying insertion movements *Insert*$(i,j)$ to all pairs of jobs involving some specific randomly chosen job $j$. The tabu restrictions and the tabu tenures are the same as for *TSLCSP1*.
- *TSLCSP6:* The restricted neighborhood is the same as for *TSLCSP5*, while the tabu restrictions and the tabu tenures are the same as for *TSLCSP2*.
- *TSLCSP7:* In this case, we consider a restricted neighborhood in which we explore all feasible solutions obtained from $s$ by applying insertion moves *Insert*$(i,j)$ to all pairs of jobs involving some specific randomly chosen job $i$. The tabu restrictions and the tabu tenures are the same adopted for *TSLCSP1*.
- *TSLCSP8:* Finally, we use the same neighborhood as for *TSLCSP7*, while the tabu restrictions and the tabu tenures are the same as for *TSLCSP2*.

Each of these eight configurations was tested in [8] for the six possible combinations of initial solution (*Best_PRH*, *Best_SSH*, and *Best_PLH*) and objective function ($F_1$ and $F_2$). For each pair of initial solution and objective function, five runs of the eight configurations were made for all instances of the benchmark data set. None of them have dominated the others in terms of solution quality. The best solution found among those runs was kept and used as one of the possible starting points in the parallel version of tabu search algoritm described in the next section.

### 3.2 The tabu team

The structure of the search processes follows the same idea used in [1] to build a parallel tabu search for the circuit partitioning problem. The eight configurations for the TSLCSP algorithm described in Section 3.1 were selected to act as agents of the parallel tabu search scheme since, as reported in [8], they have produced the best computational results for the sequential tabu search algorithm.

The framework of the parallel tabu search algorithm proposed for problem LCSP is depicted in Figure 7. Eight search processes corresponding to each of the eight configurations of the basic sequential tabu search cooperate through a central process, which maintains a pool containing the best solutions found by the search processes. The search processes start from different initial solutions, as described in Section 4, and use different search strategies to explore the solution space. Whenever a search process improves the best solution it has already visited, it sends the new solution to the central process. After a certain number of iterations without improving its best solution, a search process requests a new solution from the pool of elite solutions kept by the central process. Thus, cooperation among the search processes is done through the central process and consists in the reuse of solutions obtained by other search processes. Globally, this mechanism can be viewed as an intensification scheme, since it forces the exploration of the neighborhood of a solution which has been visited earlier. However, from the point of view of the algorithm running on the processor which receives a solution, it resembles to a diversification step, since the search algorithm jumps from its current solution to a new one which, in general, does not belong to the current neighborhood. Each search process decides to send solutions to or to request solutions from the central process based exclusively on its own search path. Thus, communication between search and central processes is clearly asynchronous. The operation of the search and central processes is further detailed below.

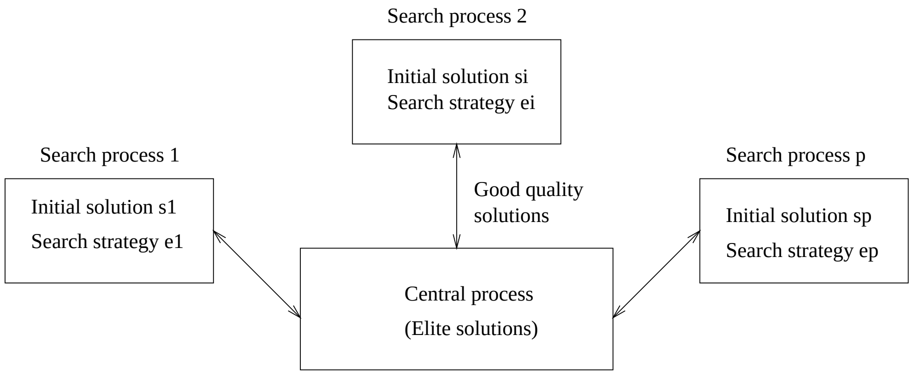



Figure 7: Parallel tabu search for problem LCSP.

**The central process:** Similarly to the a-team case, the central process has the following basic functions: (i) to start up the search processes; (ii) to manage the pool of elite solutions, and (iii) to determine the end of the search. Most of these tasks are executed precisely in the same way as those performed by the a-team described in Section 2. The major difference consists in the form by which solutions are stored in memory and exchanged between processes.

The data stored for each solution in the pool are: its cost, the job sequence, and a list of forbidden processes to which this solution cannot be sent, since they have already visited it. The size of the pool is determined by a parameter `POOL_SIZE`. The pool is initially empty and solutions are stored in increasing order of cost, without repetitions. Whenever a new solution is accepted for insertion in the pool and the latter is full, the currently worst solution is discarded and replaced by the new one. The list of forbidden processes associated with some solution in the pool is initialized with the process that sent it when it entered into the pool. The central process updates this list in two situations: (i) whenever a solution is sent to another process by request to start an intensification phase, and (ii) whenever another process finds the same solution and tries to insert it into the pool.

After all search processes halt, the central process applies a simple local search procedure based on insertion moves to all solutions currently in the pool. This post-optimization step essentially checks for the possibility of improving the solutions in the pool by testing all possible insertion movements, as previously described for the sequential tabu search algorithm.

**Search processes:** There are three situations in which a search process has to communicate with the central one: (i) when it sends a good quality solution, (ii) when it requests a new solution to start an intensification phase, or (iii) when it halts its execution. A search process sends a new solution to the central process whenever this new solution improves the best solution it previously visited. As for the a-team in the previous section, the search process first sends only the cost of the solution to the central process. The latter, according to the acceptance rule, decides whether or not this solution should be stored in the pool. If so, the search process is requested to send the job sequence corresponding to the new solution.

To start an intensification phase, a search process periodically requests a new solution to the central process to restart its search. There are two situations leading to such requests: (i) a certain upper bound `PAR_MAX_BAD_MOVES` on the number of iterations without improvement in the best solution visited by this process is reached, and (ii) a certain number `MAX_NOT_INTENSIFIED_ITER` of iterations were performed since the last time a request for a new solution coming from this process failed to be served. Requests for new solutions are made and served according to the following procedure. First, the search process sends a message to the central process asking for the solution. The central process checks in the pool for the minimum cost solution whose list of forbidden processes do not contain the one which originated the request. If such a solution is found, it is sent to the search process and its list of forbidden processes is updated. The search process sets this solution as its current one, updates the current best solution if necessary, reinitializes its search memory, and the search continues. On the other hand, if the central process pool was not able to serve the request with a solution from the pool, the search process continues its search from point where it was interrupted and runs until it requests a new solution to the pool or one of the termination criteria is reached.

Finally, there are two conditions which determine the end of the computation of a given search process: (i) a certain upper bound `PAR_MAX_TOTAL_ITER` on the total number of iterations executed by this process is attained, or (ii) two consecutive requests for new solutions have not been served by the central process and the search process was unable to improve its best solution in between.

## 4 Computational Experiments

We have implemented both the a-team and the parallel tabu search for LCSP using the `C++` language. Communication between the processes is performed using the *Parallel Virtual Machine* (`PVM`) library, version 3.4. The computational results presented here were obtained on a workstation `Sun SPARC` 1000, with 308 MB of memory and 8 processors.

A-team and parallel tabu search codes have been tested on all the 25 instances from the benchmark data set (descriptions and data available at http://www.dcc.unicamp.br/ cid/SPLC/welcome.html) described in [4]. Two of these instances were provided by BASF (*Ins_4o_24j_A* and *Ins_10o_88j_A*), while the remaining ones were randomly generated. Instance names are denoted by strings of the form *Ins_Xo_Yj_Z*, where $Y$ is the number of jobs, $X$ is the number of orders, and $Z$ is an optional character to distinguish instances with the same number of orders or jobs. The instances solved correspond to scenarios with 18 workers available at each period.

Several parameter settings have been tested in preliminary experiments to tune the codes. Those leading to the best results were kept for the final experiments. These tests were run on the five smaller instances whose optima are known and on the largest 88-job instance provided by BASF. We give the final values assigned to the parameters at the end of the experiments. The a-team code uses $\gamma = 10\%$ (acceptance criterion in the complete memory solution), `CSM_SIZE` = 100 (size of CSM), and `MAX_SOLUTIONS` = 15000 (termination criterion defined by the number of solutions which entered CSM). The parallel tabu search implementation also uses $\gamma = 10\%$ (acceptance criterion for a solution to enter into the pool), `MAX_NOT_INTENSIFIED_ITER` = 100 (maximum number of iterations between two requests of new solutions to the pool), `PAR_MAX_BAD_MOVES` = 100 (maximum number of non-improving moves before the request of a new solution to the pool), and `PAR_MAX_TOTAL_ITER` = 1000 (maximum number of iterations performed by each search process).

Each search process uses one of the neighborhood configurations *TSLCSP1* to *TSLCSP8* described in Section 3.1. To improve solution quality and to reduce the influence of the initial solutions, each run of the parallel tabu search algorithm is performed in two passes, using two different initialization strategies. In the first pass, a *random* strategy initializes each process with a pair initial solution - objective function randomly chosen from the set $\{$*Best_PRH, Best_SSH, Sol_PLH*$\} \times \{F_1, F_2\}$. Next, in the second pass, a *best-choice* strategy initializes each process with the pair initial solution - objective function set as the one which produced the best solution among the six combinations for the corresponding neighborhood configuration. Whenever they are processed independently, none of these strategies seems to clearly dominate the other. Table 1 summarizes comparative results obtained for the set of benchmark instances. For each instance, we indicate the number of jobs and the cost of the best solutions independently found by the random and the best-choice strategies. Numbers in bold face represent best values. The two strategies found equivalent solutions for 12 out of the 25 instances. The best-choice strategy produced the best solution for nine of the remaining instances, while the random strategy was better for only four instances.

| Instance | jobs | *Random* | *Best-choice* |
|---|---|---|---|
| *Ins_4o_21j_A* | 21 | **82** | **82** |
| *Ins_4o_23j_A* | 23 | **58** | **58** |
| *Ins_4o_24j_A* | 24 | **68** | **68** |
| *Ins_4o_24j_B* | 24 | **72** | **72** |
| *Ins_4o_27j_A* | 27 | **67** | **67** |
| *Ins_6o_41j_A* | 41 | 141 | **140** |
| *Ins_6o_41j_B* | 41 | **110** | **110** |
| *Ins_6o_41j_C* | 41 | **126** | 128 |
| *Ins_6o_44j_A* | 44 | **117** | **117** |
| *Ins_6o_44j_B* | 44 | **137** | **137** |
| *Ins_8o_63j_A* | 63 | 260 | **259** |
| *Ins_8o_63j_B* | 63 | 316 | **314** |
| *Ins_8o_63j_C* | 63 | 297 | **294** |
| *Ins_8o_65j_A* | 65 | **406** | **406** |
| *Ins_8o_65j_B* | 65 | 384 | **383** |
| *Ins_10o_84j_A* | 84 | 635 | **634** |
| *Ins_10o_84j_B* | 84 | 554 | **550** |
| *Ins_10o_85j_A* | 85 | **783** | 791 |
| *Ins_10o_87j_A* | 87 | 582 | **581** |
| *Ins_10o_88j_A* | 88 | **450** | **450** |
| *Ins_10o_100j_A* | 100 | **1468** | **1468** |
| *Ins_10o_102j_A* | 102 | 1158 | **1155** |
| *Ins_10o_106j_A* | 106 | **1087** | **1087** |
| *Ins_12o_108j_A* | 108 | **1271** | 1275 |
| *Ins_12o_109j_A* | 109 | **1324** | 1328 |

Table 1: Best makespans obtained with random and best-choice initialization strategies

Table 2 shows, for each test instance, the number of jobs, and the makespan of the best schedules found by the a-team and the two-pass parallel tabu search algorithm, together with those obtained by the other three cited sequential approaches used to solve this problem: constraint programming (CP) [4], linear programming-based heuristics (LPH), and sequential tabu search (STS) [6]. To allow a better evaluation of the efficiency gain resulting from the parallel approaches, we have also included the best results (BCA) obtained among those generated individually by the construction agents of the a-team. These heuristics have been tested in [8], from where the results in Table 2 are taken, and show that they are outperformed by all other methods. For each instance, the result displayed for the sequential tabu search algorithm corresponds to the best value found after three runs (each using a different initial seed for the random number generator) with the eight tabu processes running independently and performing $8 \times \texttt{MAX\_TOTAL\_ITER} = 8000$ iterations each. For each instance, the value of the best solution found is displayed in bold face in Table 2. All heuristics but the construction agents of the a-team found solutions with the same makespan for the first five instances. These values have been proved to be optimal in [6]. The construction agents of the a-team found optimal solutions only for the first two instances.

The two parallel approaches presented in this paper never failed in finding a better solution than the best one found by the sequential algorithms. Solutions generated by the a-team are better than or match the best one found by the sequential methods for 17 out of the 25 instances. Moreover, for nine instances the results produced by the a-team are strictly better. This comparison is still more striking in the case of parallel tabu search, which produced results at least as good as those found for the sequential methods for all 25 instances. For 15 of such instances, parallel tabu search produced solutions strictly better than all sequential techniques applied to this problem.

For 13 out of the 25 instances, parallel tabu search found strictly better solutions than the a-team, while the a-team was better in the case of only four instances. Moreover, parallel tabu search seems to be much more effective exactly for the larger, more difficult instances. Indeed, the parallel tabu search algorithm found strictly better solutions than all other algorithms (including the a-team) for nine among the ten largest instances. These results justify the use of the parallel approaches to tackle the LCSP. Moreover, they show that the cooperation between the different algorithms is crucial to obtain high quality solutions. As it can be seen by comparing the results obtained by the sequential and parallel tabu search algorithms, in all but one case both parallel tabu search strategies have been able to produce solutions of better quality than those obtained by each tabu search agent running independently. The only exception refers to instance *Ins_8o_63j_C* when the random strategy is used to generate initial solutions.

| Instance | jobs | AT | PTS | STS | BCA | CP | LPH |
|---|---|---|---|---|---|---|---|
| *Ins_4o_21j_A* | 21 | **82** | **82** | **82** | **82** | **82** | **82** |
| *Ins_4o_23j_A* | 23 | **58** | **58** | **58** | **58** | **58** | **58** |
| *Ins_4o_24j_A* | 24 | **68** | **68** | **68** | 69 | **68** | **68** |
| *Ins_4o_24j_B* | 24 | **72** | **72** | **72** | 73 | **72** | **72** |
| *Ins_4o_27j_A* | 27 | **67** | **67** | **67** | 69 | **67** | **67** |
| *Ins_6o_41j_A* | 41 | 143 | **140** | 141 | 150 | 152 | 142 |
| *Ins_6o_41j_B* | 41 | 111 | **110** | **110** | 115 | **110** | 112 |
| *Ins_6o_41j_C* | 41 | **126** | **126** | 128 | 136 | 134 | 130 |
| *Ins_6o_44j_A* | 44 | **116** | 117 | 117 | 123 | 122 | 118 |
| *Ins_6o_44j_B* | 44 | **137** | **137** | **137** | 146 | 149 | **137** |
| *Ins_8o_63j_A* | 63 | **259** | **259** | 261 | 283 | 281 | 273 |
| *Ins_8o_63j_B* | 63 | 316 | **314** | 316 | 347 | 344 | 323 |
| *Ins_8o_63j_C* | 63 | 301 | **294** | 296 | 325 | 344 | 308 |
| *Ins_8o_65j_A* | 65 | **403** | 406 | 406 | 421 | 445 | 411 |
| *Ins_8o_65j_B* | 65 | **382** | 383 | 384 | 408 | 411 | 402 |
| *Ins_10o_84j_A* | 84 | 641 | **634** | 636 | 708 | 730 | – |
| *Ins_10o_84j_B* | 84 | 567 | **550** | 556 | 606 | 616 | – |
| *Ins_10o_85j_A* | 85 | 793 | **783** | 791 | 879 | 912 | – |
| *Ins_10o_87j_A* | 87 | 585 | **581** | 582 | 615 | 610 | – |
| *Ins_10o_88j_A* | 88 | 456 | **450** | 460 | 693 | 473 | – |
| *Ins_10o_100j_A* | 100 | **1467** | 1468 | 1468 | 1519 | 1587 | – |
| *Ins_10o_102j_A* | 102 | 1158 | **1155** | 1166 | 1266 | 1239 | – |
| *Ins_10o_106j_A* | 106 | 1098 | **1087** | 1094 | 1163 | 1166 | – |
| *Ins_12o_108j_A* | 108 | 1277 | **1271** | 1277 | 1341 | 1412 | – |
| *Ins_12o_109j_A* | 109 | 1336 | **1324** | 1343 | 1461 | 1476 | – |

Table 2: Best values found for benchmark instances of LCSP

The results reported in Tables 1 and 2 for the parallel algorithms have been obtained without too much effort in tuning the best parameter values. Since the parallel algorithms have some randomized steps, we have investigated their robustness running both the a-team and the parallel tabu search three times for each instance, using different initial seeds. For both the a-team and the parallel tabu search algorithm, the relative error with respect to the best value observed for the worst and for the median values found never exceeded 1.5%, showing that randomness in the parallel algorithms seems to have a small influence in their performances. This behavior has been already observed in [6] for the sequential tabu search algorithm.

For all but the two smaller instances, we illustrate in Figure 8 the observed relative elapsed times to the best solution for the a-team and the parallel tabu search algorithm (the latter running with only one initialization pass and using the best-choice strategy). Elapsed times have been measured in exclusive mode, with each parallel algorithm running as a single user. We notice that the elapsed times to the best solution seem to be considerably smaller for the parallel tabu search algorithm (less than one third of the time taken by the a-team for most cases). Parallel tabu search finds better solution within the same elapsed times and are much likely to benefit from improved stopping criteria leading to earlier termination of either algorithm.

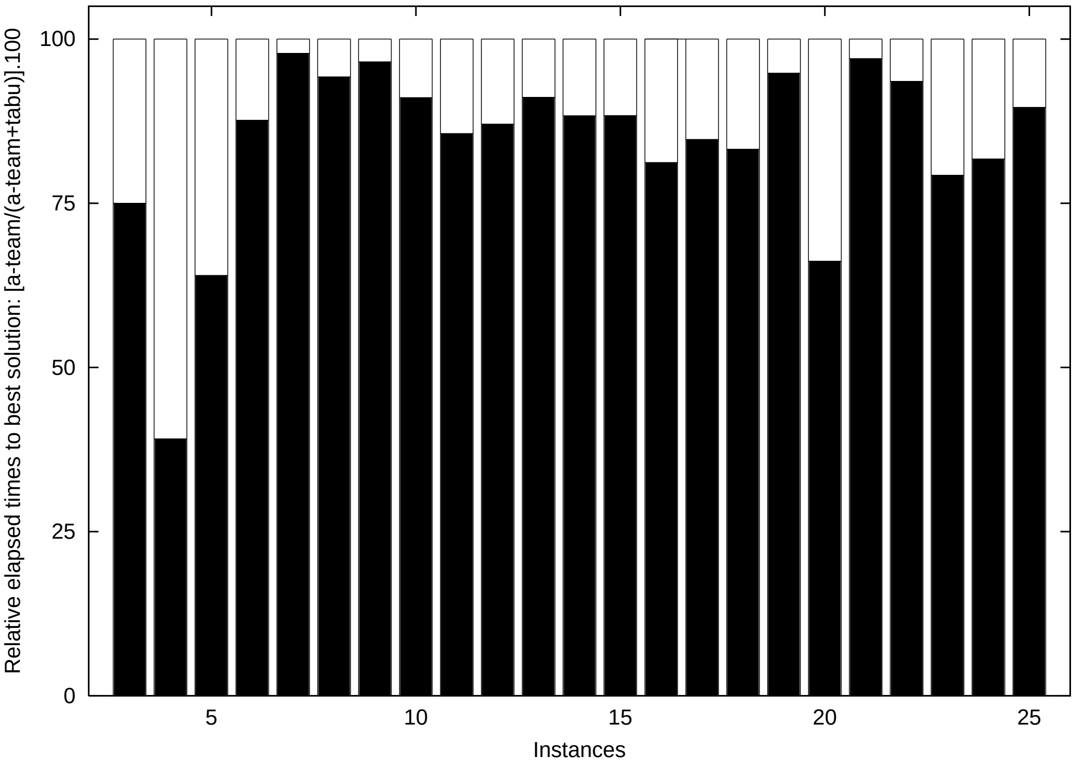



Figure 8: Relative elapsed times to the best solution for the parallel approaches

## 5 Conclusions

We proposed two parallel and cooperative algorithms for the labor constrained scheduling problem. The first is an a-team scheme, while the second is an asynchronous cooperative multiple-walk parallel tabu search strategy. The computational results obtained for a set of LCSP benchmark instances have shown that both parallel algorithms proposed are able to find high quality approximate solutions. In particular, they have matched or improved the best known solutions for all benchmark instances. Moreover, the computational results have also shown that both strategies benefit directly from the synergy induced by running several algorithms in parallel, since they have found better solutions than any individual sequential algorithm running in the same overall computation time.

A direct comparison of the two parallel approaches favors the tabu search algorithm. For the majority of the instances tested in our experiments, the parallel tabu search code generates solutions with smaller makespans and faster than the a-team code. A possible explanation for this performance comes from the fact that the agents in the tabu search code are more complex than those forming the a-team. In spite of this, it is somehow intriguing that the a-team was able to produce strictly better solutions than the tabu code in four out of the 25 instances of the benchmark data set (see Table 2).

## References


[1] R.M. Aiex, S.L. Martins, C.C. Ribeiro, and N.R. Rodriguez, "Cooperative multi-thread paralell tabu search with an application to circuit partitioning", *Lecture Notes in Computer Science* 1457 (1998), 310–331.

[2] L. Baerentzen, P. Avila, and S. Talukdar, "Learning network design for asynchronous teams", *Lecture Notes in Artificial Intelligence* 1237 (1997), 177–196.

[3] K.R. Baker, *Introduction to sequencing and scheduling*, Wiley, 1974.

[4] C.B. Cavalcante, Y. Colombani, S. Heipcke, and C.C. Souza, "Scheduling under labour resource constraints", *Constraints* 5 (2000), 415–422.

[5] C.B. Cavalcante and C.C. Souza, "A tabu search approach for scheduling problem under labour constraints", State University of Campinas, Institute of Computing, Technical report IC-97-13, 1997.

[6] C.B. Cavalcante, C.C. Souza, M.W. Savelsbergh, Y.Wang, and L.A. Wolsey, "Scheduling projects with labor constraints", *Discrete Applied Mathematics*, to appear.

[7] V.F. Cavalcante, *Asynchronous teams for the job shop scheduling problem: Construction Heuristic* (in Portuguese), M.Sc. Dissertation, State University of Campinas, Institute of Computing, 1995.

[8] C.C.B. Cavalcante, *Scheduling under labour constrints: heuristics and lower bounds* (in Portuguese), M.Sc. Dissertation, State University of Campinas, Institute of Computing, 1998.

[9] J. Chakaprani and J. Skorin-Kapov, "Massively parallel tabu search for the quadratic assignment problem", *Annals of Operations Research* 41 (1993), 327–341.

[10] J. Chakaprani and J. Skorin-Kapov, "Connection Machine implementation of a tabu search for the traveling salesman problem", *Journal of Computing and Information Technology* 1 (1993), 29–36.

[11] P. Chang, J. Dolan, J. Hemmerle, S. Talukdar, and M. Terk, "Asynchronous teams: An agent-based problem-solving architecture", *Artificial Neural Networks in Engineering* 7 (1997), 73–78.

[12] T.G. Crainic, M. Toulouse, and M. Gendreau, "Parallel asynchronous tabu search for multicommodity location-allocation with balancing requirements", *Annals of Operations Research* 63 (1993), 277–299.

[13] T.G. Crainic, M. Toulouse, and M. Gendreau, "Synchronous tabu search parallelization strategies for multicommodity location-allocation with balancing requirements", *OR Spektrum* 17 (1995), 113–123.

[14] V.-D. Cung, S.L. Martins, C.C. Ribeiro, and C. Roucairol, "Strategies for the parallel implementation of metaheuristics", in *Essays and surveys in metaheuristics* (C.C. Ribeiro and P. Hansen, eds.), Kluwer, 2001 (this volume).

[15] E.W. Davis and J.H. Patterson, "A comparison of heuristics and optimum solutions in resource constrained project scheduling", *Management Science*, 21(1975), 944–955.

[16] F. Glover, "Tabu Search - Part I", *ORSA Journal on Computing* 1 (1989), 190–206.

[17] F. Glover, "Tabu Search - Part II", *ORSA Journal on Computing* 2 (1990), 4–32.

[18] F. Glover and M. Laguna, *Tabu search*, Kluwer, 1997.

[19] E.G. Haddad, *Asynchronous teams for the job shop scheduling problem: Improvement heuristics* (in Portuguese), M.Sc. Dissertation, State University of Campinas, Institute of Computing, 1996.

[20] S. Heipcke, *A new constraint programming approach to large scale resource constrained scheduling*, Diploma-thesis, Mathematisch Geographische Fakultät, Katholische Universität Eichstätt, 1995.

[21] S. Heipcke and Y. Colombani, "A new constraint programming approach to large scale resource constrained scheduling", Workshop on Models and Algorithms for Planning and Scheduling Problems, Cambridge, 1997.

[22] J.H. Patterson, "Project scheduling: The effects of problem structure on heuristic performance", *Naval Research Logistics Quaterly* 20 (1976), 95–123.

[23] H.P. Peixoto, *A methodology for the specification of asynchronous teams for combinatorial optimization problems* (in Portuguese), M.Sc. Dissertation, State University of Campinas, Institute of Computing, 1995.

[24] S.C. Porto and C.C. Ribeiro, "Parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints", *Journal of Heuristics* 1 (1996), 107–223.

[25] J. Rachlin, R. Goodwin, S. Murthy, R. Akkiraju, F. Wu, S. Kumaran, and R. Das, "A-teams: An agent architecture for optimization and decision-support", *Lecture Notes in Artificial Intelligence* 1555 (1999), 261–276.

[26] C. Rego and C. Roucairol, "A parallel tabu search algorithm using ejection chains for the VRP", in *Metaheuristics: Theory and Applications* (I.H. Osman and J.P. Kelly, eds.), 253-295, Kluwer, 1996.

[27] Y. Rochat and É.D. Taillard, PROBABILISTIC DIVERSIFICATION AND INTENSIFICATION IN LOCAL SEARCH FOR VEHICLE ROUTING, *Journal of Heuristics* 1 (1995), 147–167.

[28] M.W. Savelsbergh, R.N. Uma, and J. Wein, "An experimental study of LP-based approximation algorithms for scheduling problems", *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 453–462, 1998.

[29] M.W. Savelsbergh, Y. Wang, and L.A. Wolsey, "Computational experiments with a large-scale resource constrained project scheduling problem", Note, Georgia Institute of Technology, 1996.

[30] C.C. Souza and L.A. Wolsey, "Scheduling projects with labour constraints", State University of Campinas, Institute of Computing, Technical report IC-97-13, 1997.

[31] P.S. Souza, *Asynchronous organizations for multi-algorithm problems*, Ph.d. Dissertation, Electrical and Computer Engineering Department, Carnegie Mellon University, 1993.

[32] P.S. Souza and S.N. Talukdar, "Genetic algorithms in asynchronous teams", *Proceedings of the Fourth International Conference on Genetic Algorithms*, 392–397, Morgan Kaufmann, 1991

[33] R.H. Storer and S.D. Wu, and R. Vaccari, "New Search Spaces for Sequence Problems with Applications to Job Shop Scheduling", *Management Science*, 38 (1992), 1495–1509.

[34] S.N. Talukdar, L. Baerentzen, and A. Gove, "Asynchronous teams: Cooperation schemes for autonomous agents", *Journal of Heuristics* 4 (1998), 295–321.

[35] S.N. Talukdar, S.S. Pyo, and T.C. Giras, "Asynchronous procedures for parallel processing", *IEEE Transactions on Power Apparatus and Systems* PAS-102 (1983), 3652–3659.

[36] E.D. Taillard, "Robust taboo search techniques for the quadratic assignment problem", *Parallel Computing* 17 (1991), 443-455.

[37] E.D. Taillard, "Parallel taboo search techniques for the job shop scheduling problem", *ORSA Journal on Computing* 6 (1994), 108-117.

[38] M.G.A. Verhoeven and E.H.L. Aarts, "Parallel local search", *Journal of Heuristics* 1 (1995), 43-65.