Overview Paper

A parallel algorithm for train rescheduling

Sai Prashanth Josyula*, Johanna Törnquist Krasemann, Lars Lundberg

Department of Computer Science, Blekinge Institute of Technology, Valhallavägen 1, 371 41 Karlskrona, Sweden

ABSTRACT

One of the crucial factors in achieving a high punctuality in railway traffic systems, is the ability to effectively reschedule the trains when disturbances occur. The railway traffic rescheduling problem is a complex task to solve both from a practical and a computational perspective. Problems of practically relevant sizes have typically a very large search space, making them time-consuming to solve even for state-of-the-art optimization solvers. Though competitive algorithmic approaches are a widespread topic of research, not much research has been done to explore the opportunities and challenges in parallelizing them. This paper presents a parallel algorithm to efficiently solve the real-time railway rescheduling problem on a multi-core parallel architecture. We devised (1) an effective way to represent the solution space as a binary tree and (2) a novel sequential heuristic algorithm based on a depth-first search (DFS) strategy that quickly traverses the tree. Based on that, we designed a parallel algorithm for a multi-core architecture, which proved to be 10.5 times faster than the sequential algorithm even when run on a single processing core. When executed on a parallel machine with 8 cores, the speed further increased by a factor of 4.68 and every disturbance scenario in the considered case study was solved within 6 s. We conclude that for the problem under consideration, though a sequential DFS approach is fast in several disturbance scenarios, it is notably slower in many other disturbance scenarios. The parallel DFS approach that combines a DFS with simultaneous breadth-wise tree exploration, while being much faster on an average, is also consistently fast across all scenarios.

1. Introduction

Decision-making is the process of identifying, assessing and making appropriate decisions to solve a problem. Scheduling is a decision-making process that involves making choices regarding allocation of available resources to tasks over a given time period with a goal to optimize one or more objectives (Pinedo, 2016). Scheduling is a frequently employed crucial operation in several organizations and sectors e.g., manufacturing industries and the railway transport sector.

In railway traffic network management, the ability to efficiently schedule the trains and the network maintenance, influences the punctuality of trains and Quality of Service (QoS) significantly. The importance is reflected in the goal set by the Swedish railway industry stating that by year 2020, 95 % of all trains should arrive at the latest within five minutes of the initially planned arrival time (Trafikverket, 2015). Similar goals have been set by the railway industries in Australia (NSW-Transport, 2016), Netherlands (Nederlandse-Spoorwegen, 2016) and several countries across the world, thus emphasizing the importance of train punctuality and QoS. The punctuality of trains is primarily affected by (1) the occurrence of disturbances, (2) the robustness of the train schedules (i.e., the timetables) and the associated ability to recover from delays, as well as (3) the ability to effectively reschedule trains when

* Corresponding author.

E-mail addresses: sai.prashanth.josyula@bth.se (S.P. Josyula), johanna.tornquist.krasemann@bth.se (J. Törnquist Krasemann), lars.lundberg@bth.se (L. Lundberg).

https://doi.org/10.1016/j.trc.2018.07.003
Received 28 November 2017; Received in revised form 27 June 2018; Accepted 4 July 2018

0968-090X/ © 2018 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).
disturbances occur, so that consequences, e.g., delays, are minimized.

In this paper we focus on the latter, and present an algorithm for efficient rescheduling of railway traffic during disturbances. The purpose of the algorithm is to compute, in a short time, a relevant set of alternative revised schedules to support the train traffic dispatchers in the real-time decision-making. In order to benefit from the advances in computer hardware, and with an aim of generating revised schedules of good quality faster, we design and implement corresponding parallel algorithms for the initially designed algorithm.

The paper is organized as follows: In the next section, we describe the rescheduling problem in more detail and the scope of this study. In Section 3, we present an overview of related research work and a brief discussion of the main research challenges addressed in this paper, along with the expected research contributions. In Section 4, we present the basic terminology used and the design of the algorithmic approach, along with the rationale for crucial design choices and the types of rescheduling decisions that are applied by the algorithm. We conclude the section with a description and discussion of the parallel algorithm. In Section 5, we describe the experimental platform, the chosen case study, and the key aspects that are considered for performance evaluation. In Section 6, we present, analyze and discuss the results. We present conclusions and suggested future work in Section 7.

2. Problem description

In the railway sector, day-to-day train services are based on preplanned timetables which ensure feasibility of the services by respecting the applicable constraints. Typically, such constraints enforce safety by requiring a minimum time separation between consecutive trains passing through the same railway track. A disturbance in a railway network is an unexpected event that renders the originally planned timetable infeasible by introducing ‘conflicts’. A conflict is considered to be a situation that arises when two trains require an infrastructure resource during overlapping time periods in a way such that one or more system constraints are violated.

Disturbances are triggered by incidents such as over-crowded platform(s) that possibly lead to unexpectedly long boarding times and minor delays, or larger incidents such as power shortages, train malfunctions, signalling system failures that cause more significant delays. Railway timetables are planned with appropriate time margins in order to recover from minor delays. Hence, in case of a minor disturbance, the affected train(s) may be able to recover from the effects of the disturbance provided there is sufficient buffer in the original timetable. In case of a disturbance that causes a significant delay to one or more trains, conflicts arise in the original timetable and it becomes operationally infeasible. The resolution of these conflicts to obtain a feasible timetable during operations, constitutes real-time railway traffic rescheduling.

In order to resolve a conflict, the following three tactics are frequently employed: (1) Retiming, i.e., allocating new arrival and departures times to one or more trains, (2) local rerouting, i.e., allocating alternative tracks to one or more trains, (3) reordering, i.e., prioritizing a train over another. Apart from these tactics, conflicts can also be handled by: (4) Globally rerouting the trains, or (5) partially/fully cancelling the affected train services. The algorithmic approach presented in this paper applies only the first three mentioned rescheduling actions.

During a disturbance, rescheduling the railway traffic is typically handled manually by train dispatchers who have very limited access to decision support systems (Törnquist Krasemann, 2012; Larsen et al., 2014). The time available for analysing alternative decisions is often very limited. Under these circumstances, a safe rescheduling strategy often employed by train dispatchers is to reduce the delay of important trains by prioritizing them over other trains (Törnquist Krasemann, 2012). This strategy does not always lead to the best rescheduling solution as several potentially desirable alternative schedules are never considered. Thus, it is a challenge for the decision-maker to analyze alternative desirable solutions and motivate his/her rescheduling choices within the available time.

3. Related work

In this section, we present an overview of the work that is most relevant to our research objectives. The various problem formulations, models and solution approaches employed for real-time railway (re) scheduling have been surveyed time and again by researchers (Törnquist, 2006; Cacchiani et al., 2014; Fang et al., 2015). In recent work, Fang et al. (2015) present a comprehensive survey of various types of modelling and solution approaches for the railway rescheduling problem. According to their survey, the most frequently used models for the rescheduling of railway traffic networks are mixed integer linear programming (MILP) models, alternative graph (Mascis and Pacciarelli, 2002) models and integer programming (IP) models, in the mentioned order. The survey by Fang et al. (2015) also reveals that heuristic approaches are most frequently employed by researchers to solve real-time railway rescheduling problems.

Real-time railway rescheduling can be considered as a combinatorial optimization problem. In combinatorial optimization, Graphics Processing Unit (GPU) computing has been successfully used by meta-heuristic algorithms (Bozejko et al., 2010, 2012; Luong et al., 2013) as well as exact algorithms (Melab et al., 2012; Dabah et al., 2016) to achieve significant speedups. Bozejko et al. (2010) report about achievements of significant speedups (2×–55×) for various benchmark instances of the flexible job shop problem, by parallelizing their algorithm on GPUs.

The branch and bound algorithmic approach to the flow shop problem has also been parallelized on the GPU in recent works (Melab et al., 2012; Dabah et al., 2016). Melab et al. (2012) use the computing power of the GPU for the calculation of lower bounds (rather than using it for the parallel exploration of the search tree). Their approach is well-motivated as the explored search tree is highly irregular, thus making the tree exploration not well-suitable for parallelization on the GPUs. The algorithm uses a large pool of threads on the GPU to compute the lower bounds while the CPU performs elimination, selection and branching operations. The
authors claim to achieve significant speedups (of the order of 100×) from their GPU implementation in comparison with the corresponding sequential implementation. Chakroun et al. (2013) extend the work of Melab et al. (2012) and present an improved parallel algorithm with reduced thread divergence.

In the domain of scheduling and rescheduling of railway traffic, there have been relatively few works (Abramson et al., 1994; Mu and Dessouky, 2011; Iqbal and Grahn, 2012; Petersson, 2015) that employ parallelization techniques. In one of the early works, Abramson et al. (1994) accelerate the execution of their genetic algorithm for the computation of efficient train schedules by parallelizing their algorithm. They parallelize their algorithm both on shared memory architectures (e.g. multi-core CPUs) and distributed architectures (cluster of computers). Mu and Dessouky (2011) propose a parallel heuristic algorithm for scheduling freight trains. Though their parallel implementation runs faster than the sequential counterpart, it produces a train schedule with higher delays compared to the schedule produced by the sequential implementation.

In relatively recent works, Petersson (2015) and Iqbal and Grahn (2012) present the parallelization of a greedy algorithm for real-time railway rescheduling on GPUs and multi-core CPUs respectively. The work of Iqbal and Grahn (2012) focuses on improving the quality of obtained solutions, while the work of Petersson (2015) focuses on achieving better speedup values. Their work shows promising potential to explore more solutions per unit time in comparison to the sequential algorithm.

In the greedy heuristic presented by Törnquist Krasemann (2012), the rescheduling problem is modelled as a search for the best order in which the train events are to be scheduled or prioritized. Each branch in the search tree corresponds to an order in which the train events are scheduled. This approach has the advantage that all the branches in the tree are of the same size resulting in a very regular search tree (with a fixed determinate depth) that is well-suited for parallelization. On the other hand, since changing the order of the events does not always result in different feasible schedules, redundant branches exist in the search tree. Thus, some of the computing power in the GPU parallelized algorithm (Petersson, 2015) is wasted in exploring redundant solutions. Petersson (2015) mentions that in one of the chosen disturbance scenarios, 1770 practically equal solutions are explored simultaneously.

Van Thielen et al. (2017) present a heuristic conflict prevention technique in which they construct a solution tree based on conflicts at the nodes. Though our intended search tree has similar characteristics, we explore comparatively higher number of nodes and also prune branches based on the cost of the global best solution.

The railway rescheduling problem of practically relevant sizes has a very large search space. Owing to that, it is a time-consuming problem to solve, even for state-of-the-art solvers. Not much research has been done to explore the opportunities and challenges in parallelizing the algorithmic approaches for real-time railway rescheduling. It is evident that the way in which the problem is modelled plays a vital role in the ‘intelligent’ navigability of the search space, the avoidance of redundant solutions, and the parallelizability of the algorithm. Thus, an algorithmic approach needs to have (1) an effective way of defining the feasible search space and (2) a potentially parallelizable, cost-effective search strategy. In this paper we address both the aspects, but focus on the latter. We did not come across any research studies that jointly deal with the aforementioned issues within the domain of railway traffic distance management and rescheduling. In our work, we make an effort to contribute towards filling this research gap. The main contributions of the work presented in this paper are the following:

(i) an effective way to represent the search space in the form of a train-conflict tree.
(ii) a novel heuristic sequential algorithm for real-time railway rescheduling that quickly traverses the conflict tree.
(iii) a parallelized algorithmic approach incorporating the sequential algorithm, and implemented on a multi-core architecture in order to achieve speedup and consistently lower execution times, and
(iv) a systematic performance assessment of the presented algorithms based on real data.

4. Algorithmic framework

4.1. Definitions, assumptions and restrictions

In this sub-section, we introduce the main terminology, notation and constraints while the detailed corresponding mathematical formulation is outlined in Appendix A.

An event is a resource request by a certain train for a specific section (Törnquist Krasemann, 2012). A section can be of two types: (1) line section (comprising of one block or a sequence of several consecutive blocks), (2) station section. Every section of the infrastructure has a limited number of individual bi-directional tracks. We do not consider the restrictions that may arise due to the individual properties of tracks (e.g., track length).

The schedule of a train i is a series of consecutive train events. Each event (i, k) has an initially planned start time $b_{ik}^{\text{initial}}$, end time $e_{ik}^{\text{initial}}$, rescheduled start time $x_{ik}^{\text{begin}}$, rescheduled end time $x_{ik}^{\text{end}}$, minimum occupation time $d_{ik}$ and a binary parameter $h_{ik}$ to indicate if it is an event occurring at a station section with a scheduled passenger stop in which case the train cannot depart before the initially planned end time. A given timetable can be viewed as a set of train-event lists or alternatively as a set of section-event lists, which in turn can be viewed as a collection of track-event lists. The latter perspective is particularly useful when detecting the conflicts in a timetable. All the event lists are always in a chronological order. Each pair of consecutive events in a train-event list correspond to a pair of adjacent sections in the route of the train. Each pair of consecutive events in a section-event list correspond to a pair of trains that occupy the section successively.

We assume that we have an initial feasible timetable $\mathcal{F}_{\text{orig}}$ and a disturbance renders the timetable infeasible. We also assume that the disturbance as well as the disturbed timetable are known to the decision maker (e.g., infrastructure manager) at a wall clock time $\mathcal{W}_0$. At the time of disturbance, several events of the trains have already finished/started and therefore those events cannot be
assigned new times in the rescheduled timetable.

The algorithm strives to minimize the total final delay of all trains, where a delay is non-negative. That is, early trains do not reduce the total final delay.

The main restrictions are outlined below:

4.1. Train restrictions

- No-wait constraints: For each train $i$, for every pair of consecutive train events $(i, k)$, $(i, k+1)$, the second event begins as soon as the first event ends ($x_{ik}^{\text{begin}} = x_{ik+1}^{\text{end}}$).
- Run time and dwell time constraints: On each line section, a train can run faster than initially planned, but never faster than minimum occupation time $d_{ik}$. On station sections, $d_{ik}$ corresponds to the minimum required dwell time.
- Departure time constraints: In the case of events that occur on a station, if the corresponding train $i$ of an event $(i, k)$ has a commercial stop (i.e., if $h_{ik} = 1$), then it cannot depart from the station (i.e., the event cannot end) before its initially planned departure time.

4.1.2. Infrastructure restrictions

- Track occupancy constraints at sections: A train can occupy only one track of a section and not more. At most one train is permitted to occupy any specific track at a time.
- Track consistency constraints: In two consecutive line sections, a train has to occupy the same track, i.e., it cannot switch track.
- Clear time constraints: Trains allocated to the same track $t$ of a section $j$ should be separated by a minimum clear time $\Delta_j$ whenever:
  1. they are running in the opposite direction on the section (denoted by $\leftrightarrow$), or
  2. section $j$ is a single-block section (denoted by $|B_j| = 1$).
- Headway constraints: Trains allocated to the same track $t$ of a section $j$ should be separated by a minimum headway time $H_j$ whenever:
  1. they are in the same direction (denoted by $\rightarrow$, and
  2. section $j$ is a multi-block section (denoted by $|B_j| > 1$).

Clear time $\Delta_j$ between two trains $(i, k)$ and $(\hat{i}, \hat{k})$ on the same track of section $j$ is the minimum time separation between the ‘tail’ of the first train and the ‘head’ of the next train ($x_{i,k}^{\text{begin}} - x_{\hat{i},\hat{k}}^{\text{end}}$). In contrast, the headway $H_j$ is the minimum time separation between the heads ($x_{i,k}^{\text{end}} - x_{\hat{i},\hat{k}}^{\text{begin}}$), as well as the tails of both the trains ($x_{i,k}^{\text{end}} - x_{\hat{i},\hat{k}}^{\text{end}}$). In Fig. 1, headway constraints are applicable for the trains travelling in the same direction and assigned to the same track of Line section I. On all other sections, as well as when $\leftrightarrow$ on Line section I, clear time constraints are applicable for the trains assigned to the same track. These constraints are mathematically formulated in the corresponding MIP formulation, see Appendix A.3.1.

4.2. Design choices

With an aim to significantly reduce the size of the search tree and the number of redundant solutions, we represent the search tree with conflicts as the nodes and rescheduling decisions as the edges. An example of an alternative design choice is to represent the tree with train events as the nodes (Törnquist Krasemann, 2012). We refer to the node of the tree as a conflict node as it corresponds to a conflict in a partial timetable. We refer to the edge as a decision edge as it corresponds to rescheduling decision(s). Leaf nodes in the unpruned branches correspond to feasible solutions.

The next design choice is related to the conflict nodes. Typically, a conflict can be between two or more trains. But we chose a conflict node to represent a conflict between exactly two trains. The rationale for this choice is to ensure that the search tree is binary, so that it is more structured and easily parallelizable. We recall that one of the aims of the algorithm is to have a good workload for parallelization. Alternative design choices are (1) to make a conflict node to represent a conflict between two or more trains, (2) to

![Fig. 1. Illustration of the infrastructure’s granularity (Törnquist Krasemann, 2012).](image-url)
make a conflict node represent multiple conflicts. However, in both cases the construction of each child node (which involves the resolution of the conflict in the parent node) might take varying amount of time. This potentially leads to unequal thread workloads during parallelization.

The next design choice is related to the metric that determines the branches to be pruned. We chose the predicted total final delay (i.e., sum of all train delays at their final destinations) as the pruning (or guiding) criteria since it adequately captures the propagation of delay over time and space. Nevertheless, there are also other important metrics to consider (Josyula and Törnquist Krasemann, 2017), e.g., the number of delayed trains. However, such a metric (i.e., the number of delayed trains) is ideally applied when passenger flow data is available to indicate the weight (or importance) that is to be assigned to different trains and departures. Thus, during the construction of the search tree, though we observe the number of delayed trains, we use the predicted total final delay as the pruning criteria. In a situation where passenger flow data is available, the algorithm can easily be re-designed (i.e., by changing only a few lines in the corresponding source code) to enable the application of the aforementioned alternative metric.

The aim of the algorithm is to produce one or more feasible timetables from the disturbed timetable by retiming, local rerouting and reordering of affected trains. Retiming a train means adjusting the departure and/or arrival times of the train (by shifting the originally allocated time slots, delaying or speeding up the train). Throughout our algorithm, we retime a train by employing the following tactics:

1. by increasing the dwell time or run time on a chosen station section or line section respectively, and/or
2. by decreasing the dwell and run times on all other line and station sections respectively, while satisfying the respective constraints,
3. by appropriately shifting the timeslots of the events, in order to satisfy the No-Wait constraint,
4. by appropriately increasing the dwell times at commercial stations, in order to satisfy the Departure Time constraint.

In order to fulfill our aim, we implement the algorithm in the following consecutive stages:

(A) Utilizing available time supplements (UTS) (prior to the construction of the search tree).

The conflicts in the initial disturbed timetable are identified and ‘handled’ (not necessarily resolved) by retiming the trains involved in those conflicts through employing all of the retiming tactics 2–4.

(B) Iterative conflict detection and resolution (ICDR).

Each conflict is resolved by reordering trains (i.e., prioritizing one train over another) and by employing one of the following rescheduling tactics:

(i) Local rerouting (the unprioritized train).
   – Track reallocations at line sections.
   – Platform reassignments at station sections.
(ii) Retiming (the unprioritized train) by employing all of the retiming tactics 1–4.

While retiming in the context of utilizing the time supplements, we do not delay a train unless it is violating the Departure time constraint at a station section. The process of Iterative Conflict Detection and Resolution (ICDR) is logically equivalent to construction (and simultaneous navigation) of a Full Binary Tree starting with the root node, in a depth-first manner. We explicitly utilize the time supplements prior to the ICDR stage (i.e., prior to the search tree construction). The rationale for this choice is to maximize the use of time supplements at a prior step, so that the solution cost typically increases along every branch of the tree. This enables us to reduce the discarding of potentially desirable branches while pruning. In contrast, these time supplements can be utilized while making the rescheduling decisions (i.e., while adding decision edges to the search tree), in which case the cost of a node (i.e., the cost of a partial solution representing the node) could often be less than the cost of its ancestor node.

The search tree represents only a subset of the entire solution space, primarily due to the following choices made in the rescheduling strategy:

• If we can resolve a conflict by reallocating tracks of a train, we do not explore the retiming strategy for that train to resolve that particular conflict. The rationale for this is to balance the number of local-rerouting and retiming decisions that are made during rescheduling.
• If both the trains involved in a conflict are in the same direction on the conflict section, we use a First-come, First-served (FCFS) strategy. The second train to enter the conflict section is delayed on the adjacent section. Our experimental trials with few alternative strategies, while not producing better results, also resulted in larger search trees.
• In the retiming strategy for two trains in the opposite direction, while delaying the unprioritized train, it is delayed on the first multi-track section prior to the section of conflict. The rationale behind this design choice is to delay the train as late as possible (ALAP).

4.3. Utilization of available Time Supplements (UTS)

First and foremost, we detect the conflicts in the disturbed timetable and utilize available time supplements on the trains involved in the conflicts. The detected conflicts are the direct effect of the disturbance on the original timetable. Every conflict corresponds to the violation of a clear time constraint or a headway constraint on a section, whichever is applicable in the scenario. For example, in
The conflicts between Train A and Train B at Station A and Line Section I are due to the violation of clear time and headway constraints respectively. The conflict between Train A and Train C at Station B is due to the violation of a clear time constraint.

We detect the conflicts in an infeasible timetable by the procedure illustrated in Algorithm 1. At each track of every section in the infrastructure, for each pair of consecutive events in the corresponding track-event list, we check if the applicable constraint is violated. In case of such a violation, we record it as a conflict and add the conflict information to the list of detected conflicts. We handle the initially detected conflicts by ‘greedily’ decreasing the run times and dwell times of the trains involved in conflicts by making use of available time supplements, whenever possible. One alternative strategy that we considered is to decrease the run times, dwell times of the trains only up to their conflict event. However, our experimental trials suggested us to adopt the greedy strategy. In order to satisfy the No-wait constraint, we shift the time slots of the events whenever necessary. Additionally, in order to satisfy the Departure Time constraint, we also increase the dwell time of the train on a commercial station if it reaches the station before the originally planned time. This usually occurs as a result of decreasing run times and dwell times on prior sections.

It is important to note that new conflicts may arise due to the retiming that is carried out in this stage. All the existing unresolved conflicts and the newly arisen conflicts are resolved in the ICDR stage, where the time supplements of trains involved in the newly arisen conflicts are also utilized.

Algorithm 1. Detect conflicts

[Algorithm 1: Detect conflicts]

Fig. 2. A time-distance graph of the railway traffic network (illustrated in Fig. 1) to describe the type of conflicts. Note: Assume that clear time and headway time intervals = 3 min.
Algorithm 2. Utilization of time supplements prior to construction of the tree

<table>
<thead>
<tr>
<th>Input:</th>
<th>Timetable $\mathcal{T}$, Disturbed event $(i_d, k_d)$, Disturbance length $t$, Minimum run/dwell times, List of commercial stops.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>Partial timetable.</td>
</tr>
</tbody>
</table>

1. Generate the disturbed timetable as follows:

   foreach event $E$ do
   
   ```
   \begin{align*}
   x^\text{begin}_E &= b^\text{initial}_E \\
   x^\text{end}_E &= e^\text{initial}_E \\
   x^\text{end}_{i_d,k_d} &= e^\text{initial}_{i_d,k_d} + t \\
   \end{align*}
   ```

2. foreach event $E \in (i_d, k_d + 1), (i_d, k_d + 2), \ldots$ do

   ```
   \begin{align*}
   x^\text{begin}_E &= b^\text{initial}_E + t \\
   x^\text{end}_E &= e^\text{initial}_E + t \\
   \end{align*}
   ```

3. Detect conflicts in the disturbed timetable using Algorithm 1.

4. Modify the disturbed timetable as follows:

   foreach conflict $c = ((i, k), (i, \hat{k}), j)$ in the detected conflicts do

   ```
   \begin{align*}
   x^\text{begin}_E &= x^\text{end}_{E-1} \\
   x^\text{end}_E &= \text{maximum}(x^\text{begin}_E + d_E, e^\text{initial}_E * h_E) \\
   \end{align*}
   ```

4.3.1. Detailed UTS algorithm

In the following section, we discuss in detail the algorithm employed to utilize the time supplements (Algorithm 2) prior to construction of the search tree. The disturbed timetable is generated by: (i) copying the original timetable (lines 2–4), (ii) updating the disturbed event (line 5), and (iii) adding the disturbance time to the begin and end times of all the subsequent events in the event list of the disturbed train (lines 6–8). In line 9, we detect all the conflicts in the disturbed timetable. In lines 11–13, we select the events for which we intend to utilize the time supplements. Then, for each of the selected event (that is not the first event in its train event list), in line 15, we shift the begin time of the event so that the No-wait constraint is not violated. In line 16, we change the end time of the event $E$, while ensuring that the Departure Time constraint is satisfied. This is done by introducing $\text{max}(\cdot)$ and $e^\text{initial}_E * h_E$. The actual value of the time supplement of an event $E$ is $(e^\text{initial}_E - b^\text{initial}_E - d_E)$. In order to utilize this time supplement, we only need to know the minimum time of the event $E$, i.e., $d_E$. The timings of event $E$ are changed irrespective of the fact that it may cause new conflicts.

The aim of this algorithm is not to resolve the conflicts detected in line 6, but only to utilize the time supplements of the trains involved in those conflicts. As mentioned earlier, the rationale for this design choice is to maximize the use of time supplements in a stage prior to search tree construction (i.e., in the current UTS stage), so that the solution cost (i.e., the predicted total final delay) typically increases along every branch of the search tree. Such an increase in cost enables us to reduce the discarding of potentially desirable branches while pruning. Hence, after the disturbed timetable is modified (i.e., after execution of line 16), the number of conflicts in the resulting timetable may increase, decrease or remain the same in comparison to the number of conflicts detected in line 9. This algorithm results in a partial timetable which is the starting point for the construction of the decision tree in the ICDR stage.

4.4. Iterative Conflict Detection and Resolution (ICDR)

After the rescheduling performed in the previous stage (by employing retiming tactics 2–4), we now have a partial timetable with conflicts. We obtain one or more feasible timetables from this partial timetable by iteratively detecting conflicts and resolving them by (i) reordering and (ii) by either local rerouting or retiming (by employing tactics 1–4).

We can resolve a conflict in two alternate ways; by prioritizing either of the train over the other one that is involved in the
conflict. These two alternatives constitute the two decision edges of a conflict node in the search tree. While finding feasible solutions, we consider both the alternatives with respect to prioritizing trains. This can be seen from line 13 and line 19 of Algorithm 3. At each conflict node, we apply either of the following rescheduling tactics on the unprioritized train in order to build the child node:

(i) Local rerouting
– Reallocating the track of the unprioritized train at a station section or a line section.

(ii) Retiming
– Delaying the unprioritized train in favour of the prioritized train by increasing its run time (or dwell time) on the chosen section.
– Utilizing the time supplements of the unprioritized train if it has not already been done in the previous stage.

We locally reroute the unprioritized train whenever an empty track is available on the conflict section throughout the time that it occupies that section. Otherwise, we retime the train by employing the delaying strategy in Table 1 and also the retiming tactics 2–4. In case of a multi-track conflict section with no empty track, the unavailability of an empty track is because all the other tracks are occupied at least for some time throughout the required time duration. In this case, local rerouting is not a straightforward option and any attempt to locally reroute the train may generate several additional conflicts. This is the rationale for employing retiming in such a scenario. While rerouting, we allocate the unprioritized train to the first empty track of the conflict section that is available for the time that the train occupies that section.

In order to efficiently construct the search tree and to avoid potentially infeasible and undesirable solutions, we do not explore a conflict node further when certain conditions are satisfied. The conditions are as follows:

1. When the cost of the partial timetable (corresponding to the conflict node) is greater than the cost of the best solution found so far.  
   Explanation: Typically, the cost of a child node is greater than (e.g., when a train is delayed in favour of another train) or equal to (e.g., when a track is reallocated) the cost of the parent node.

2. When both the conflict trains are travelling in the same direction on the conflict section and the first train to enter the section is unprioritized.  
   Explanation: When two trains travelling in the same direction have a conflict, we adopt the FCFS strategy on the conflict section.

<table>
<thead>
<tr>
<th>Conflict Trains’ direction</th>
<th>Delay section</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opposite</td>
<td>The multi-track section in the route of the train that is prior to and closest to the conflict section.</td>
<td>The train is delayed on the delay section until the prioritized train enters that section and such that the clear time constraint on the section adjacent to the delay section is fulfilled.</td>
</tr>
<tr>
<td>Same</td>
<td>The section in the route of the train that is prior to and closest to the conflict section.</td>
<td>The train is delayed on the delay section for the least possible time such that the violated constraint on the conflict section is fulfilled.</td>
</tr>
</tbody>
</table>

Table 1
Delaying strategy for the unprioritized train: Increase of run times/dwell times.
Algorithm 3. DFS construction of a Full Binary Tree (ICDR)

```
Input: Partial Timetable resulting from the UTS stage, Infrastructure.
Output: Feasible solutions.

1 Cost of the global best solution = ∞.
2 Construct(Root node).

3 Function Construct(node):
   4 | Detect conflicts in the partial timetable using Algorithm 1.
   5 | if no conflict detected then /* Leaf node */
   6 | Save the feasible timetable and Compute cost () of the timetable.
   7 | Update the cost of the global best solution, and return.
   8 | else if node is infeasible then return. /* Leaf node */
   9 | else /* Internal node */
  10 | Select the 'earliest' conflict from the detected conflicts.
  11 | Compute cost () of the partial timetable.
  12 | if cost of the partial timetable ≤ cost of global best solution then
  13 | Prioritize train that first enters the section. /* Left child node */
  14 | Resolve the selected conflict by employing the appropriate rescheduling tactic,
  15 | using Table 1.
  16 | Construct(left child node).
  17 | Restore the state of the parent node.
  18 | if conflict trains are in the same direction then Set the right child node to
  19 | infeasible.
  20 | else
  21 | Prioritize train that next enters the section. /* Right child node */
  22 | Resolve the selected conflict by employing the appropriate rescheduling tactic,
  23 | using Table 1.
  24 | Construct(right child node).
  25 | Restore the state of the parent node.
  26 | return

27 Function Compute cost():
   28 | foreach train do
   29 | | foreach last event \((i, n_i)\) do
   30 | | \(z_i = z^\text{begin}_{i, n_i} - z^\text{initial}_{i, n_i}\) /* Calculate Delay */
   31 | | if \((z_i > 0)\) then \(C = C + z_i\) /* Sum up the experienced delays */
   32 | end
   33 | end
   34 | return
```

Each feasible solution obtained from the construction of the tree has a corresponding complete branch in the tree. We perform Depth-first Search (DFS) to construct (and simultaneously traverse) the tree, as illustrated in Fig. 3.

4.4.1. Example of search tree construction

An example of a search tree is given in Fig. 4. In this example, a train experiences a delay of 5 min (300 s) at a section. After utilizing the time supplements, we construct the tree by following the steps in Algorithm 3, which correspond to a depth-first construction of the tree as illustrated in Fig. 3. Since Construct() is a recursive function, constructing the root node by calling Construct(Root node) builds the entire tree through recursive function calls. Line 2 of Algorithm 3 corresponds to initiating the construction of the root node (and consequently the entire search tree). Lines 4–24 are executed at each node of the tree. We present and explain the crucial steps in Algorithm 3 that happen during the construction of the root node:

1. Conflict detection (Line 4): In this step, conflicts in the partial timetable of a node are detected (by means of Algorithm 1). Each
detected conflict is in the form of a tuple: \( ((i, k), (\tilde{i}, \tilde{k})), j \), in which the event \((i, k)\) precedes event \((\tilde{i}, \tilde{k})\) on section \(j\) in the corresponding partial timetable.

**Unsorted conflicts:** We say that a conflict \(c_1\) precedes another conflict \(c_2\) if the first event in its tuple precedes its counterpart in the tuple of conflict \(c_2\). Based on this definition, it is important to note that the detected conflicts are not in a chronological order, i.e., they are unsorted.

2. **Selecting a conflict to resolve (Line 10):** In this step, we select the ‘earliest’ occurring conflict among the detected conflicts, in order to resolve at the node. In a C++ implementation, this is achieved by simply making use of `std::sort()` and a custom comparator.

In Fig. 4, this step corresponds to labelling the node, e.g., in case of the root node, selecting the conflict between *Train 1267* and *Train 94979* at the single-tracked section ÖND1-VÖV.

3. **Prioritization of a train in the selected conflict (Line 13):** The step prior to resolving the selected conflict is to prioritize a train among the two trains involved in that conflict. In Fig. 4, this prioritization corresponds to the left edge of the root node, labelled as “94979 waits for 1267...”. The first train in the selected conflict of the root node (Fig. 4) is *Train 1267*. Hence, while constructing the left edge, *Train 1267* is prioritized over the other train, and thus the *Train 94979* is made to wait.
4. Resolving the selected conflict (Line 14): After prioritizing the trains involved in the conflict, we resolve the conflict at the node by delaying the unprioritized train according to the strategy in Table 1. In Fig. 4, this corresponds to the selection of the section where the unprioritized train is to be delayed: “94979 waits for 1267 at ÖND1”.

Surely, as part of this step, we update the partial timetable as per the above decision. After this step, the above steps are repeated for the left child of the root node, owing to the recursive function call on Line 15 of Algorithm 3.

4.5. Parallel algorithm

We design the parallel algorithm by decomposing the sequential DFS construction of the search tree into several disjoint tasks which can be computed in parallel on the processing cores of the underlying hardware. At every node starting with the root node, the parent thread responsible for performing computations on the node determines whether it is going to be a leaf node or an internal node\(^1\) by detecting the number of conflicts in the corresponding node-related partial timetable. If it is going to be an internal node, the parent thread (1) spawns a child thread, (2) assigns to it the task of constructing the right sub-tree, (3) initializes its Thread-local Storage (TLS) by copying the state of the parent node, and (4) continues with the construction of the left sub-tree. Thus, each thread constructs and traverses a branch of the search tree. Throughout the execution of the parallel program, the operating system dynamically assigns threads to the available processors, and all the threads share and update the value of global best solution. By means of this design, the parallel algorithm is functionally equivalent to the sequential algorithm, i.e., it obtains the same best solutions when run to completion.

However, we noticed that this algorithmic design is not scalable when applied on larger problem instances, due to the creation of several child threads. The thread creation, synchronization and initialization of TLS is a non-trivial computational task and has a significant effect on the performance of the parallel program. In order to overcome this drawback, we introduce a parameter that limits the number of spawned child threads.

Prior to the execution of the parallel program, we set a maximum limit on the number of spawned child threads. During the execution of the program, once the specified number of child threads are created, each thread runs in parallel an instance of the aforementioned sequential DFS algorithm with the appropriate node as its root node (illustrated in Fig. 5). After this point in time, no new child threads are spawned throughout the execution of the program. A minor drawback of this approach is that as time progresses, the existing threads may finish building their sub-tree and hence terminate. Therefore, the number of threads could decrease with a progress of time.

When the sequential program is used to solve a disturbance scenario, the same search tree is constructed across several runs of the program. In the case of the parallel program, the search tree that is constructed before terminating at the best solution varies slightly across several runs. This is because the scheduling of parallel threads of execution by the operating system of the hardware platform cannot be determined at the program runtime.

4.5.1. Methods for performance evaluation

In order to analyze the performance of a parallel implementation and to evaluate the benefit of parallelism, a comparison with the execution time of a sequential implementation is crucial (Rauber and Rünger, 2013). Speedup (expressed as a relative saving in execution times) is a frequently employed metric for the practical evaluation of parallel implementations (Rauber and Rünger, 2013). The baseline implementation used to calculate speedup values is often chosen from the following alternatives (Crowl, 1994); (1) the equivalent sequential implementation, (2) the best known (i.e., fastest) sequential implementation, or (3) the parallel implementation running on one processing core.

---

\(^1\) In a binary tree, an internal node is any node that has either one or two child nodes. All other nodes with no child nodes are called leaf nodes.
5.2. Case study

Based on the cause, disturbances can be classified into three categories:

(C1) A train enters the traffic management district with a certain delay or it suffers from a temporary delay at one section within the district.

(C2) A train has a permanent malfunction resulting in increased running times on all line sections it is planned to occupy.

(C3) An infrastructure failure causing, e.g., a speed reduction on a certain section, which results in increased running times for all trains running through that section.

According to this classification, every disturbance belongs to either of the three categories (C1–C3), unless it results in full blockage of specific track(s). Alternatively, based on their severity, disturbances can be exhaustively classified into two types: minor
disturbances, and major disturbances (Lamorgese et al., 2018). Theoretically, the algorithm can deal with all the aforementioned disturbances; no aspect of the algorithm assumes or requires the disturbance to be of a specific category. However, in our case study, we evaluate the performance of the algorithm only for disturbances where the delay situation is initiated by a train that suffers from a temporary delay (i.e., Category 1 disturbances). This delay may then propagate within the studied network depending on the level of congestion and the way in which the trains are re-scheduled.

In representative sampling, a researcher purposely selects cases for a study such that they match the larger population on specific characteristics (Urdan, 2016). In order to reduce the threat to generalizability of the conclusions, we use representative sampling with an aim to ensure that the sample (i.e., the set of chosen disturbance scenarios) is a sufficiently close representative of the population (i.e., set of all possible Category 1 disturbance scenarios).

We conduct experiments with the railway network from Karlskrona-Tjörnarp (illustrated in Fig. 6). The infrastructure consists of a single-track line with 59 sections (including stations), and all tracks are bi-directional. The original timetable is from 15:50 to 21:10 (5 h 20 min). The disturbance scenarios 1–8, 9–16, 17–24, 25–32, and 33–40 correspond to induced delays of 5 min, 13 min, 17 min, 21 min and 25 min respectively. The time windows for the scenarios vary between 2 h and 4.9 h. The 40 disturbance scenarios comprising the case study are enumerated in Table 4.

![Fig. 7. A screen-shot of the interface of the train timetable visualization tool.](image)

Table 2

<table>
<thead>
<tr>
<th>Subject topic</th>
<th>Recorded Metric</th>
<th>Seq</th>
<th>Par₈</th>
<th>Par₉</th>
<th>MIP Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valuation of the first solution</td>
<td>Cost (i.e., Total final delay), Delayed trains</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Execution time</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Properties of the search tree</td>
<td>Depth (Max, Avg)</td>
<td>✓</td>
<td>*</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Total internal nodes</td>
<td>✓</td>
<td>*</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Feasible solutions</td>
<td>✓</td>
<td>*</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Pruned branches</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Valuation of the best solution</td>
<td>Cost (i.e., Total final delay), Delayed trains</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Average Execution time</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Occurrence of the best solution</td>
<td>Branch number</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branch depth</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Valuation of the optimal solution</td>
<td>Cost (i.e., Total final delay), Delayed trains</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Execution time</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* We record these values but do not present them for the sake of brevity. We observe that the properties of the search tree are highly similar irrespective of the parallel algorithm being executed on 1 core or 8 cores.

Note: Execution time refers to the time taken by the program to find the solution.
In every considered disturbance scenario of the case study, we have a single source of disturbance. Nonetheless, in real-life scenarios where disturbances occur throughout the day, the algorithm could potentially also be used for iteratively re-scheduling the existing timetable based on the potential propagation of delays and the newly occurred disturbance.

5.3. Evaluation and validation of the algorithms

In order to evaluate the algorithms, we implement them in C++ and solve the problem scenarios in the case study using four methods: the sequential program, the parallel program (on 1 core and 8 cores) with a threshold of 64 child threads, and the commercial MIP solver (Gurobi). The latter provides the optimal solution to the MILP formulation of the problem (Appendix A), whereas the former three programs provide the obtained ‘best’ solution. In order to execute the parallel program on a single processing core of the parallel machine, we employed the command: `taskset –cpu-list 0`.

As a prior step to the evaluation of algorithms, we numerically validate our programs (using Algorithm 1) by ensuring that the output timetable is feasible (i.e., conflict-free). We then visualize a subset of the timetables associated with the best solutions and validate them ‘manually’ by means of a train timetable visualization tool (Fig. 7). This ‘manual validation’ is performed in order: (i) to understand the interaction between trains, (ii) to comprehend the kind of prioritization that was done by the algorithm, and (iii) to analyze whether the obtained ‘best’ feasible solutions are indeed reasonable. Since the interactions between trains cannot be fully grasped via numbers, and since it is easier to have a larger view of such interactions via graphical timetables, we make use of a visualization tool.

We ensure that the `taskset` command is indeed running the parallel program on 1 core (as intended) through monitoring with the `top` utility and verifying that the %CPU usage\(^2\) of the program is not greater than 100%. We validate the correctness of our algorithmic design by verifying equality in the results (i.e., costs of the best solutions) of the sequential and the parallel programs. Apart from the above validations, we also compare the obtained solution with the optimal solution. The value of the optimal solution is a lower bound for the value of the best solution obtained by the algorithms.

In order to compute the average execution time for obtaining the best solution, we solve each problem scenario 5 times (i.e., in total, 200 runs each of sequential program, parallel program (on 1 core and 8 cores)). All metrics that are used to evaluate the algorithms are enumerated in Table 2. While recording the execution times, we disable all kinds of Disk I/O operations performed by the program (e.g., logging the rescheduling procedure, saving the feasible solutions to disk, etc.) as such operations can heavily distort the measurements. While validating the recorded properties of the full binary tree (Table B.1), we employed the following characteristic of a Full binary tree: Number of branches = Number of internal nodes + 1.

6. Results and discussion

6.1. Goodness of heuristic

The best solutions obtained by the algorithm are reasonably close to optimal solutions in majority of the disturbance scenarios (the average relative optimality gap\(^3\) = 0.1888), thus indicating the goodness of the designed heuristic (refer Tables 3, 4, Fig. 8).

---

**Table 3**

<table>
<thead>
<tr>
<th>Topic of discussion</th>
<th>Metric</th>
<th>Obtained value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goodness of heuristic</strong></td>
<td>Average Optimality gap × 100</td>
<td>3.52 min</td>
</tr>
<tr>
<td></td>
<td>Optimality gap</td>
<td>18.88%</td>
</tr>
<tr>
<td><strong>Drawbacks of sequential algorithm</strong></td>
<td>Range of execution times</td>
<td>[0.08 s, 6.4 min]</td>
</tr>
<tr>
<td></td>
<td>Average execution time</td>
<td>36.86 s</td>
</tr>
<tr>
<td></td>
<td>Standard deviation</td>
<td>79.20 s</td>
</tr>
<tr>
<td><strong>Benefits of parallel algorithm design</strong></td>
<td>Range of execution times</td>
<td>[0.08 s, 31.87 s]</td>
</tr>
<tr>
<td></td>
<td>Average execution time</td>
<td>3.51 s</td>
</tr>
<tr>
<td></td>
<td>Standard Deviation</td>
<td>7.68 s</td>
</tr>
<tr>
<td></td>
<td>Average Speedup (on 1 core)</td>
<td>(\frac{36.86}{3.51} = 10.5)</td>
</tr>
<tr>
<td><strong>Further benefits due to parallel architecture (8 cores)</strong></td>
<td>Range of execution times</td>
<td>[0.08 s, 5.41 s]</td>
</tr>
<tr>
<td></td>
<td>Average execution time</td>
<td>0.75 s</td>
</tr>
<tr>
<td></td>
<td>Standard deviation</td>
<td>1.44 s</td>
</tr>
<tr>
<td></td>
<td>Average Speedup (on 8 cores)</td>
<td>(\frac{3.51}{0.75} = 4.68)</td>
</tr>
</tbody>
</table>

---

\(^2\) By default, the `top` utility displays the %CPU usage as a percentage of a single CPU. For example, on a multi-core processor with 8 CPUs, the CPU usage of a program can vary between 0% and 800%.

\(^3\) Average \(\frac{\text{Optimality gap}}{\text{Optimal solution cost}}\) is computed over 37 disturbance scenarios. In case of the remaining 3 disturbance scenarios (scenario numbers 1, 2, S.P. Josyula et al. Transportation Research Part C 95 (2018) 545–569
<table>
<thead>
<tr>
<th>Nr#</th>
<th>Scenarios</th>
<th>Detailed description of the disturbance scenario</th>
<th>Time Window</th>
<th>Obtained</th>
<th>Optimal</th>
<th>Difference</th>
<th>Opt sol</th>
<th>Obt sol</th>
<th>Average Runtime of 5 runs (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Train 1076 delayed 5 min at VÖV ÖND1</td>
<td>2.67 h</td>
<td>0.00 min</td>
<td>0.00 min</td>
<td>0.00 min</td>
<td>0</td>
<td>0</td>
<td>0.13</td>
<td>0.22</td>
</tr>
<tr>
<td>2</td>
<td>Train 1097 delayed 5 min at GUA NÄT</td>
<td>4.20 h</td>
<td>0.00 min</td>
<td>0.00 min</td>
<td>0.00 min</td>
<td>0</td>
<td>0</td>
<td>3.23</td>
<td>0.92</td>
</tr>
<tr>
<td>3</td>
<td>Train 1250 delayed 5 min at VÖV ÖND1</td>
<td>3.37 h</td>
<td>12.65 min</td>
<td>8.03 min</td>
<td>4.62 min</td>
<td>2</td>
<td>3</td>
<td>0.82</td>
<td>0.42</td>
</tr>
<tr>
<td>4</td>
<td>Train 1267 delayed 5 min at CR</td>
<td>2.03 h</td>
<td>1.50 min</td>
<td>1.23 min</td>
<td>0.27 min</td>
<td>1</td>
<td>1</td>
<td>0.08</td>
<td>0.14</td>
</tr>
<tr>
<td>5</td>
<td>Train 1846 delayed 5 min at VÖV ÖND1</td>
<td>4.97 h</td>
<td>2.13 min</td>
<td>1.95 min</td>
<td>0.18 min</td>
<td>2</td>
<td>2</td>
<td>0.17</td>
<td>0.18</td>
</tr>
<tr>
<td>6</td>
<td>Train 1977 delayed 5 min at SAK,L3 SÖG</td>
<td>3.71 h</td>
<td>2.80 min</td>
<td>2.80 min</td>
<td>0.00 min</td>
<td>2</td>
<td>2</td>
<td>0.09</td>
<td>0.12</td>
</tr>
<tr>
<td>7</td>
<td>Train 1978 delayed 5 min at BML SÖG</td>
<td>2.73 h</td>
<td>3.48 min</td>
<td>2.52 min</td>
<td>0.96 min</td>
<td>1</td>
<td>1</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>8</td>
<td>Train 6175 delayed 5 min at CR1 KAP</td>
<td>3.11 h</td>
<td>10.32 min</td>
<td>0.00 min</td>
<td>10.32 min</td>
<td>0</td>
<td>1</td>
<td>0.09</td>
<td>0.26</td>
</tr>
<tr>
<td>9</td>
<td>Train 1076 delayed 13 min at VÖV ÖND1</td>
<td>2.67 h</td>
<td>3.57 min</td>
<td>2.65 min</td>
<td>0.92 min</td>
<td>2</td>
<td>2</td>
<td>0.09</td>
<td>0.37</td>
</tr>
<tr>
<td>10</td>
<td>Train 1097 delayed 13 min at GUA NÄT</td>
<td>4.20 h</td>
<td>16.10 min</td>
<td>15.73 min</td>
<td>0.37 min</td>
<td>3</td>
<td>3</td>
<td>83.06</td>
<td>1.68</td>
</tr>
<tr>
<td>11</td>
<td>Train 1250 delayed 13 min at VÖV ÖND1</td>
<td>3.37 h</td>
<td>26.77 min</td>
<td>24.53 min</td>
<td>2.24 min</td>
<td>4</td>
<td>2</td>
<td>6.28</td>
<td>1.68</td>
</tr>
<tr>
<td>12</td>
<td>Train 1267 delayed 13 min at CR</td>
<td>2.03 h</td>
<td>9.50 min</td>
<td>9.23 min</td>
<td>0.27 min</td>
<td>1</td>
<td>1</td>
<td>0.09</td>
<td>0.19</td>
</tr>
<tr>
<td>13</td>
<td>Train 1846 delayed 13 min at CR</td>
<td>4.97 h</td>
<td>18.05 min</td>
<td>17.35 min</td>
<td>0.70 min</td>
<td>4</td>
<td>4</td>
<td>0.64</td>
<td>0.50</td>
</tr>
<tr>
<td>14</td>
<td>Train 1977 delayed 13 min at SAK,L3 SÖG</td>
<td>3.71 h</td>
<td>21.23 min</td>
<td>18.80 min</td>
<td>2.43 min</td>
<td>2</td>
<td>5</td>
<td>0.49</td>
<td>0.52</td>
</tr>
<tr>
<td>15</td>
<td>Train 1978 delayed 13 min at BML SÖG</td>
<td>2.73 h</td>
<td>18.80 min</td>
<td>16.87 min</td>
<td>1.93 min</td>
<td>2</td>
<td>2</td>
<td>0.08</td>
<td>0.13</td>
</tr>
<tr>
<td>16</td>
<td>Train 1076 delayed 17 min at VÖV ÖND1</td>
<td>3.11 h</td>
<td>26.48 min</td>
<td>9.20 min</td>
<td>17.28 min</td>
<td>3</td>
<td>4</td>
<td>1.81</td>
<td>0.92</td>
</tr>
<tr>
<td>17</td>
<td>Train 1267 delayed 17 min at CR</td>
<td>2.03 h</td>
<td>15.52 min</td>
<td>13.63 min</td>
<td>1.89 min</td>
<td>2</td>
<td>2</td>
<td>0.08</td>
<td>0.14</td>
</tr>
<tr>
<td>18</td>
<td>Train 1846 delayed 17 min at VÖV ÖND1</td>
<td>4.97 h</td>
<td>30.60 min</td>
<td>28.68 min</td>
<td>1.92 min</td>
<td>4</td>
<td>5</td>
<td>178.39</td>
<td>0.72</td>
</tr>
<tr>
<td>19</td>
<td>Train 1977 delayed 17 min at SAK,L3 SÖG</td>
<td>3.71 h</td>
<td>26.80 min</td>
<td>26.80 min</td>
<td>0.00 min</td>
<td>2</td>
<td>2</td>
<td>0.09</td>
<td>0.15</td>
</tr>
<tr>
<td>20</td>
<td>Train 1978 delayed 17 min at BML SÖG</td>
<td>2.73 h</td>
<td>27.52 min</td>
<td>24.87 min</td>
<td>2.65 min</td>
<td>2</td>
<td>3</td>
<td>0.10</td>
<td>0.17</td>
</tr>
<tr>
<td>21</td>
<td>Train 1250 delayed 21 min at VÖV ÖND1</td>
<td>3.11 h</td>
<td>44.92 min</td>
<td>36.52 min</td>
<td>8.40 min</td>
<td>6</td>
<td>7</td>
<td>7.81</td>
<td>10.04</td>
</tr>
<tr>
<td>22</td>
<td>Train 1267 delayed 21 min at CR</td>
<td>3.11 h</td>
<td>55.60 min</td>
<td>52.48 min</td>
<td>3.12 min</td>
<td>5</td>
<td>5</td>
<td>0.11</td>
<td>0.62</td>
</tr>
<tr>
<td>23</td>
<td>Train 1977 delayed 21 min at SAK,L3 SÖG</td>
<td>4.20 h</td>
<td>46.52 min</td>
<td>37.00 min</td>
<td>9.52 min</td>
<td>4</td>
<td>5</td>
<td>137.11</td>
<td>7.94</td>
</tr>
<tr>
<td>24</td>
<td>Train 1978 delayed 21 min at BML SÖG</td>
<td>2.73 h</td>
<td>44.53 min</td>
<td>43.57 min</td>
<td>0.96 min</td>
<td>2</td>
<td>2</td>
<td>0.08</td>
<td>0.21</td>
</tr>
<tr>
<td>25</td>
<td>Train 6175 delayed 25 min at CR1 KAP</td>
<td>3.11 h</td>
<td>68.28 min</td>
<td>53.18 min</td>
<td>15.10 min</td>
<td>7</td>
<td>4</td>
<td>384.15</td>
<td>28.25</td>
</tr>
</tbody>
</table>

Mean of the runtimes of sequential implementation, parallel implementations (on 1 core and 8 cores): 36.86 s 3.51 s 0.75 s

* The sequential implementation did not return results in these 2 scenarios, since the recursive implementation of DFS led to the over ow of the call stack.
an average, the cost of the obtained best solution is \( \approx 19\% \) more than the cost of the optimal solution. (see Table A.1).

### 6.2. Drawbacks of sequential algorithm

Though the employed heuristic returns reasonable solutions across all disturbance scenarios and does so quickly (\( \leq 15 \) sec) for several scenarios, it executes notably slower (1.4–6.4 min) on some disturbance scenarios, particularly those with larger disturbance times. The average speed of the sequential algorithm across the entire range of problem scenarios suggests room for improvement.

The execution times span across a wide range of values; consequently, the standard deviation is very high. This shows the lack of reliability in execution speed of the sequential algorithm.

The aforementioned facts indicate that a DFS strategy is perhaps the reason for the unreliability in speed of execution. It can be conjectured that the distributions of best solutions in the search space are unfavourable for the application of DFS across all disturbance scenarios. The results in Table B.1 indeed reveal that the slowness of the sequential algorithm in respective scenarios is due to the position of the obtained best solution in the search space. Though a Breadth-first search (BFS) strategy appears to be a decent alternative, it is accompanied by practical limitations concerning memory requirements; for the problem under consideration, a breadth-first approach will typically exhaust the memory available on the experimental platform. This hints at the requirement of an alternative hybrid approach that combines DFS and breadthwise tree exploration. Interestingly, our parallel DFS algorithm when run on a single-core architecture fulfils the aforementioned requirement.

### 6.3. Benefits of parallel algorithm design

The obtained results (Table 4) show an improvement in the algorithmic performance when moved from the depth-first approach to a hybrid approach. The parallel algorithm running on 1 processing core makes significant performance gains; a speedup of 10.5 in finding the best solution, and a substantial reduction in the range of execution times. One of the main factors affecting the time taken to find a solution is the portion of search space traversed in order to find that solution. Assuming within reason that the execution time is proportional to the number of explored internal nodes (i.e., number of resolved conflicts), we expect the parallel algorithm on 1 core to gain a speedup of \( \approx 3 \), since the average number of internal nodes in the explored search tree is 3.21 times lesser than the sequential counterpart (see Table B.1).

But, our calculated average speedup equals 10.5 due to the following reason: The properties of the search tree presented in Table B.1 correspond to the size of the search tree when the algorithm determines that it has the ‘best’ solution in its search space (i.e., just before it terminates). The execution times presented in Table 4 correspond to the time when the algorithm finds the best solution, at
which point it does not know that the found solution is the best candidate in the solution space. So, even after finding the best solution, the algorithm explores all other unexplored potential nodes and appropriately prunes them based on the chosen criteria. Therefore, though the parallel algorithmic design one-third the workload in determining the best solution, it actually finds the best solution significantly (i.e., 10.5 times) faster. We expect to achieve further speedup when the hybrid approach employed by the parallel algorithm is realized on a multi-core architecture.

6.4. Discussion on superlinear speedups

When \( p \) processors (or cores) simultaneously perform DFS on disjoint parts of a state-space tree to find the ‘best’ solution, the average speedup in comparison to the sequential DFS algorithm (\( sp_{seq} \)) can often be superlinear (\( > sp \)) (Rao and Kumar, 1988). This massive speedup can be attributed to one or more of the following factors:

1. The time when the first solution is discovered by one of the processors (Rao and Kumar, 1988; Kumar and Rao, 1987).
2. The discovery of a ‘good’ solution by one of the processors (Barr and Hickman, 1993).
3. The order of discovery of good solutions (Barr and Hickman, 1993).

---

Fig. 9. An overview of the cost of solutions obtained during rescheduling as time progresses (a single run of the sequential program and multiple runs of the parallel program on 8 cores).

5. The increased cache memory in a parallel hardware configuration (Ristov et al., 2016).

The discovery of a good solution can eliminate a large portion of the tree from consideration. The order of discovery of good solutions has a significant effect on the proportion of the explored search tree. If the best solutions are randomly distributed in a relatively small region of the search space, then the average speedup \( S_{avg} \) can be superlinear (Rao and Kumar, 1988). Typically, searching algorithms that terminate when one of the processors finds the best solution, experience superlinear speedups because the work performed by the parallel algorithm can be significantly less than that performed by the sequential algorithm. Rao and Kumar (1988) present and discuss a comprehensive analytical model to discuss such speedups.

6.5. Performance of parallel algorithm on multi-core architecture

Owing to the aforementioned factors, a superlinear speedup can be seen in our parallel program when compared to the sequential program (notice the values of Seq and Par in Table 4). The average speedup obtained by running on 1 core reveals that these speedups are primarily caused due to the way in which the best solutions are distributed across the search tree for the respective disturbance scenario. Thus, evaluating the performance of the parallel algorithm based on these figures will likely overestimate the achieved speedup. The speedup values presented in Table 4 are computed with respect to the parallel program on 1 core, thus ensuring that we have a good estimate of the performance gain due to parallelism.

In conformance with our expectations, the computed speedup values \( S_{avg} \) show further improvement in the execution times of the parallel algorithm when migrated from 1 core to 8 cores. We attribute this improvement to the parallel algorithm design that provides benefit from the execution on multi-core architecture. The obtained speedup values are sub-linear, i.e., they are less than the number of cores in the experiment platform. This was expected as the number of parallel threads could decrease with a progress in the execution of the parallel program.

Finding the best solution: Through our experiments, we observe that the parallel program (on 8 cores) consistently runs faster than the sequential counterpart for multiple runs on various disturbance scenarios (Fig. 9). The scheduling of threads by the underlying Operating system does not have a considerable impact on the results. In many scenarios, the sequential algorithm starts with a ‘bad’ solution and slowly converges to the best solution. In contrast, the parallel algorithm quickly converges to the best solution from its first found solution, irrespective of the cost of the first solution. This is because the parallel program simultaneously explores 64 branches of the search tree while avoiding unnecessary explorations by sharing the updated cost of the global best solution among all parallel threads of execution.

Nevertheless, the parallel program typically finds a better first solution because it starts exploring (quite early in time) other breadthwise regions of the tree that are otherwise unexplored (at that point in time) by the sequential program. In all the disturbance scenarios where the sequential program outperforms the parallel program, both of them finish execution in less than 0.25 s. For complicated disturbance scenarios resulting in several initial conflicts, the parallel algorithm has a significantly better performance.

7. Conclusions

In this paper, conducive to efficiently solving the real-time railway rescheduling problem on a multi-core parallel architecture, we represented the search space in the form of a train-conflict binary tree. In order to search the tree for the ‘best’ solution, we designed a novel heuristic algorithm based on DFS strategy. The heuristic quickly finds a decent solution for many disturbance scenarios in the considered case study, thus indicating the effectiveness of the devised search space representation. However, we observe that employing the DFS strategy renders the heuristic algorithm unstable, i.e., it runs significantly slower across few disturbance scenarios. Based on our results, we conclude that in the context of real-time railway rescheduling and solution space navigation, a sequential DFS is not a preferable approach as it could often end up searching quite many unnecessary parts of the tree.

The representation of the search tree could possibly be improved: (1) by making a node denote a conflict between more than 2 trains, whenever the need arises, and (2) by effectively resolving multiple conflicts at every node. The pruning strategy in the heuristic algorithm can be improved: (1) by pruning the branches based on multiple factors (e.g., solution cost, delayed trains, re-timed trains, etc.), and (2) by finding an initial bound for search tree construction, typically by employing a dispatching rule. The algorithm could possibly be improved further by considering different strategies while selecting the conflict to be resolved at each node (in contrast to merely selecting the earliest conflict from the list of conflicts in the timetable associated with the node). An example of such a strategy is to select a conflict which, when resolved, leads to no new conflicts (or few new conflicts).

We designed a parallel algorithm that employs a DFS strategy to simultaneously explore breadthwise disjoint parts of the search tree. This parallel approach which incorporates the sequential heuristic, significantly increases its speed (by a factor of 10.5) and runs consistently fast across all scenarios, even when executed on 1 processing core. When this parallel algorithm is executed on multi-core architecture (with 8 cores), we observe further speedup (of 4.68) and a higher consistency in execution speeds (i.e., lower standard deviation in recorded execution times). Through a performance assessment of the devised algorithms, we demonstrated that though the sequential algorithm is notably slower on many disturbance scenarios, the parallel algorithm gives us an opportunity to find the best solutions fast (i.e., within 6 s) across all the scenarios.

The average superlinear speedup of the parallel algorithm (compared to the sequential algorithm) suggests a comprehensive investigation into the distribution of solutions in the search space. Owing to the diverse distribution of solutions across several disturbance scenarios, intelligent traversal of the search tree is an important topic for further investigation. The parallel algorithm
can be further improved by allowing thread creation whenever an existing thread terminates, even after the threshold limit (for the number of spawned threads) is once reached. Finally, in order to gain further benefits due to parallelism, the existing parallel algorithm can be developed to enable the use of computations on Graphics processing units.

Acknowledgements

The research presented in this paper has been conducted within the research project TRANS-FORM which is funded by grants from the municipality of Karlshamn as well as the Swedish Research Council (FORMAS) via ERA-NET. The project has also received support by Trafikverket (The Swedish Transport Administration) and Blekingetrafiken.

Appendix A. Mathematical formulation of the problem

In the following section, we present an adaptation of the Mixed Integer Linear Programming (MILP) model that was originally outlined in Törnquist and Persson (2007).

Problem statement: Given the parameters, find the values of the decision variables while minimizing the objective function, and subject to the constraints.

A.1. Basic notation

- \( T \) Ordered set representing trains. \( T = \{1, 2, 3, \ldots\} \).
- \( K_i \) Ordered set containing event indices of a train \( i \). \( K_i = \{1, 2, 3, \ldots, n_i\} \).
- \( n_i \) The index of the last event of a train \( i \). Note that \( n_i = |K_i| \).
- \( B \) Ordered set representing sections. \( B = \{1, 2, 3, \ldots\} \).
- \(|B_j|\) Number of block sections in section \( j \).
- \( L_j \) Ordered set representing the events on a section \( j \). \( L_j = \{(i_1, k_1), (i_2, k_2), (i_3, k_3), \ldots\} \).
- \( P_j \) Ordered set representing parallel tracks of section \( j \). \( P_j = \{1, 2, 3, \ldots\} \).

All the ordered sets are ordered according to the original timetable and infrastructure.

A.2. Parameters and decision variables

- \( W_0 \) Wall clock time when the decision maker is aware of the disturbance.
- \( T_{\text{orig}} \) Original timetable.
- \( M \) A sufficiently large constant.
- \( b_{i,k}^{\text{initial}} \) Planned begin time of event \( k \) of train \( i \).
- \( e_{i,k}^{\text{initial}} \) Planned end time of event \( k \) of train \( i \).
- \( S_{i,k} \) Binary parameter indicating the type of event \( k \) of train \( i \).
  \[
  S_{i,k} = \begin{cases}
  1, & \text{if event } k \text{ of train } i \text{ occurs at a line section.} \\
  0, & \text{if event } k \text{ of train } i \text{ occurs at a station section.}
  \end{cases}
  \]
- \( d_{i,k} \) Minimum time taken by the event \( k \) of train \( i \).
  \[
  d_{i,k} = \begin{cases}
  \text{Minimum running time on a line section,} & \text{if } S_{i,k} = 1. \\
  \text{Minimum waiting time at a station section,} & \text{if } S_{i,k} = 0.
  \end{cases}
  \]
- \( h_{i,k} \) Binary parameter indicating a commercial stop of event \( k \) of train \( i \).
  \[
  h_{i,k} = \begin{cases}
  1, & \text{if it is a commercial stop.} \\
  0, & \text{otherwise.}
  \end{cases}
  \]
- \( d_{i,k} \) Direction of the train \( i \) during event \( k \).
- \( \Delta_j \) Minimum clear time on section \( j \).
- \( H_j \) Minimum headway time on section \( j \).

Note: The clear time parameters \( \Delta_j \) must be larger than 0 for the MILP model to work as intended.

The following variables constitute the ‘rescheduled timetable’ and typically obtain different values compared to their counterparts in the original timetable: (1) The begin and end times of the events, (2) the track \( t \in P_j \) used by an event \( (i, k) \) allocated to a section \( j \),

563
(3) the order of events on a section \( j \). Note that we do not perform global rerouting, i.e., we do not change the set of events on a section \( j \), but only their order. The optimization model contains six types of decision variables.

For each train event, we have the following two continuous, non-negative decision variables:

\[
x_{i,k}^{\text{begin}} \quad \text{Variable representing the rescheduled begin time of the event } k \text{ of train } i.
\]

\[
x_{i,k}^{\text{end}} \quad \text{Variable representing the rescheduled end time of the event } k \text{ of train } i.
\]

For each train, we have the following continuous, non-negative decision variable:

\[
z_i \quad \text{Variable representing the delay experienced at the final destination by train } i.
\]

For each event \((i, k) \in L_j\), we have the decision variables \( q_{i,k,t} \), \( \hat{r}_{i,k,k}^l \), and \( \hat{\lambda}_{i,k,k}^l \), where \( q_{i,k,t} \) is a binary variable that specifies whether track \( t \) (of the associated section \( j \)) is used by event \( k \) of train \( i \).

For every pair of events in \( L_j \), we have the binary decision variables \( \hat{\gamma}_{i,k,k}^l \) and \( \hat{\lambda}_{i,k,k}^l \). That is, for any two events \((i, k) \in L_j\), we introduce \( \hat{\gamma}_{i,k,k}^l \) and \( \hat{\lambda}_{i,k,k}^l \) if \( K_k < K' \) (see Table A.1).

<table>
<thead>
<tr>
<th>Notation</th>
<th>Quantifier</th>
<th>Verbal interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{\gamma}_{i,k,k}^l )</td>
<td>( (\text{dir}<em>{i,k} \neq \text{dir}</em>{i',k'}) ) ( (\text{dir}<em>{i,k} = \text{dir}</em>{i',k'}) )</td>
<td>For trains running in the opposite direction For trains running in the same direction</td>
</tr>
</tbody>
</table>

\[
q_{i,k,t} = \begin{cases} 1, & \text{if event } k \text{ of train } i \text{ uses track } t \\ 0, & \text{otherwise} \end{cases}
\]

\[
\hat{r}_{i,k,k}^l = \begin{cases} 1, & \text{if event } (i, k) \text{ occurs before event } (\hat{i}, \hat{k}) \text{ (as in the original timetable)}, \text{ where } \mathcal{X} < \hat{\mathcal{X}} \\ 0, & \text{otherwise} \end{cases}
\]

\[
\hat{\lambda}_{i,k,k}^l = \begin{cases} 1, & \text{if event } (i, k) \text{ is rescheduled to occur after event } (\hat{i}, \hat{k}), \text{ where } \mathcal{X} < \hat{\mathcal{X}} \\ 0, & \text{otherwise} \end{cases}
\]

### A.3. Objective function and constraints

Our objective is to minimize the total final delay of all the trains.

\[
\min \sum_{i \in T} z_i
\]

\( \mathcal{X} < \hat{\mathcal{X}} \)

\( j \in B, (i, k), (\hat{i}, \hat{k}) \in L_j; h_{i,k}^{\text{initial}} \leq h_{i,k}^{\text{initial}} \)

Event \((i, k)\) precedes event \((\hat{i}, \hat{k})\) with respect to the order in the set \( L_j \).

\( \epsilon_{\mathcal{X}} \)

\( (\text{dir}_{i,k} \neq \text{dir}_{i',k'}) \)

For trains running in the opposite direction

\( \mathcal{X} \)

\( (\text{dir}_{i,k} = \text{dir}_{i',k'}) \)

For trains running in the same direction

\[
q_{i,k,t} = \begin{cases} 1, & \text{if event } k \text{ of train } i \text{ uses track } t \\ 0, & \text{otherwise} \end{cases}
\]

\[
\hat{r}_{i,k,k}^l = \begin{cases} 1, & \text{if event } (i, k) \text{ occurs before event } (\hat{i}, \hat{k}) \text{ (as in the original timetable)}, \text{ where } \mathcal{X} < \hat{\mathcal{X}} \\ 0, & \text{otherwise} \end{cases}
\]

\[
\hat{\lambda}_{i,k,k}^l = \begin{cases} 1, & \text{if event } (i, k) \text{ is rescheduled to occur after event } (\hat{i}, \hat{k}), \text{ where } \mathcal{X} < \hat{\mathcal{X}} \\ 0, & \text{otherwise} \end{cases}
\]

At the time of disturbance, several events of the trains have already started and therefore those events cannot be assigned new start times in the rescheduled timetable.

\[
x_{i,k}^{\text{begin}} = b_{i,k}^{\text{initial}} \quad i \in T, \quad k \in K_i; \quad b_{i,k}^{\text{initial}} < W_0
\]

No-wait constraints: For each train \( i \), the next event begins as soon as the prior event ends.

\[
x_{i,k}^{\text{end}} = x_{i,k}^{\text{begin}} \quad i \in T, \quad k \in K_i-\{n_i\}
\]

Run time and dwell time constraints: An event \((i, k)\) requires a minimum time of \( d_{i,k} \) units to complete.

\[
x_{i,k}^{\text{end}} \geq x_{i,k}^{\text{begin}} + d_{i,k} \quad i \in T, \quad k \in K_i
\]

Departure time constraints: An event \((i, k)\) cannot end (i.e., the train cannot depart from the corresponding station) before its initially planned time if the event corresponds to a commercial stop.

\[
x_{i,k}^{\text{end}} \geq e_{i,k}^{\text{initial}} \quad i \in T, \quad k \in K_i; \quad e_{i,k}^{\text{initial}} = 1
\]

The final delay \( z_i \) experienced by each train \( i \) is the delay of its last event \((i, n_i)\).

\[
z_i \geq x_{i,n_i}^{\text{begin}} - b_{i,n_i}^{\text{initial}} \quad i \in T
\]

Since the rescheduled begin and end times cannot be negative and since they can be continuous,
\[ x_{i,k}^{\text{begin}} \geq 0, \quad x_{i,k}^{\text{end}} \geq 0 \quad i \in T, \quad k \in K_i \]  

(A.7)

Since the final delay cannot be negative,

\[ z_i \geq 0 \quad i \in T \]  

(A.8)

Track occupancy constraints at sections: For each section, for each event belonging to the section event list, that event must occupy exactly one track of the section.

\[ \sum_{j \in J_i} q_{i,k,j} = 1 \quad j \in B, \quad (i, k) \in L_j \]  

(A.9)

Track consistency constraints: Constraints (A.10) are used when a train has two consecutive events where both are scheduled on a track of the same section. This ensures that if the trains are not using the same track, the values of \( \gamma_{i,k} \) and \( \lambda_{i,k} \) cannot have the value 1 simultaneously. Constraints (A.11), (A.12) imply that once one of the variables \( \gamma_{i,k} \) or \( \lambda_{i,k} \) needs to take the value 1. Constraints (A.13) and (A.14) specify that if two events are using the same track within a section, then either the constraints (A.13)–(A.15) or (A.16)–(A.18) become active, if one of the variables \( \gamma_{i,k} \) or \( \lambda_{i,k} \) needs to take the value 1. Constraints (A.13) and (A.16) specify that one event at a section \( j \) must end and a required separation time \( \Delta_j \) must elapse until next event may start at the same section, if the events are using the same track. Similarly, constraints (A.14), (A.15), (A.17) and (A.18) specify the requirement of a separation time \( H_j \) between the begin times and the end times of the events that are using the same track of the section, whenever the corresponding conditions are satisfied (i.e., whenever the trains are in the same direction on a multi-block section).

The binary variables \( \gamma_{i,k} \) and \( \lambda_{i,k} \) are used to ensure that if the trains \( i \) and \( \hat{i} \) are assigned the same track of section \( j \), they must be separated in time. This means that when the trains use the same track, either \( \gamma_{i,k} \) or \( \lambda_{i,k} \) must take the value 1. Constraints (A.11) and (A.12) imply that if two events are using the same track within a section, then either the constraints (A.13)–(A.15) or (A.16)–(A.18) become active, if one of the variables \( \gamma_{i,k} \) or \( \lambda_{i,k} \) needs to take the value 1. Constraints (A.13) and (A.16) specify that one event at a section \( j \) must end and a required separation time \( \Delta_j \) must elapse until next event may start at the same section, if the events are using the same track. Similarly, constraints (A.14), (A.15), (A.17) and (A.18) specify the requirement of a separation time \( H_j \) between the begin times and the end times of the events that are using the same track of the section, whenever the corresponding conditions are satisfied (i.e., whenever the trains are in the same direction on a multi-block section).

If the trains are not using the same track, the values of \( \gamma_{i,k} \) and \( \lambda_{i,k} \) are not required for single-tracked sections, provided that few additional constraints are formulated (as outlined in Törnquist and Persson (2007)).

Implementation details: While implementing the aforementioned mathematical formulation, the decision variables of the type \( q_{i,k,j} \) and \( \lambda_{i,k} \) are not required for single-tracked sections, provided that few additional constraints are formulated (as outlined in Törnquist and Persson (2007)).

Modelling the disturbances: The disturbance is modelled by selecting the disturbed train \( i \) and increasing its minimum run time \( d_{i,k} \) for the disturbed event \( (i, k) \). This ensures that the disturbed train indeed arrives at its next destination delayed. Further propagation of this delay depends on the way in which the trains are re-scheduled, i.e., the run times will increase whenever there is congestion and dwell times will increase when the trains are held-up at stations.
Appendix B. Results

See Table B.1.

### Table B.1
Comparison of properties of the explored search tree for sequential and parallel programs. The properties of search tree explored by the parallel program (on 1 core and 8 cores) are highly similar. Hence we only present the properties of the latter.

<table>
<thead>
<tr>
<th>Nr</th>
<th>First solution (Seq)</th>
<th>Best solution (Seq)</th>
<th>Comparison of tree properties (Seq vs Par)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time taken</td>
<td>Solution cost</td>
<td>Branch number</td>
</tr>
<tr>
<td>1</td>
<td>0.08 s</td>
<td>1.97 min</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>0.1 s</td>
<td>21.52 min</td>
<td>2172</td>
</tr>
<tr>
<td>3</td>
<td>0.09 s</td>
<td>25.10 min</td>
<td>635</td>
</tr>
<tr>
<td>4</td>
<td>0.08 s</td>
<td>2.25 min</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0.09 s</td>
<td>9.37 min</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>0.08 s</td>
<td>2.80 min</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0.08 s</td>
<td>3.48 min</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0.09 s</td>
<td>10.32 min</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0.08 s</td>
<td>12.52 min</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>0.11 s</td>
<td>46.87 min</td>
<td>57,520</td>
</tr>
<tr>
<td>11</td>
<td>0.09 s</td>
<td>41.00 min</td>
<td>5326</td>
</tr>
<tr>
<td>12</td>
<td>0.08 s</td>
<td>9.88 min</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>0.12 s</td>
<td>19.63 min</td>
<td>315</td>
</tr>
<tr>
<td>14</td>
<td>0.08 s</td>
<td>23.65 min</td>
<td>315</td>
</tr>
<tr>
<td>15</td>
<td>0.08 s</td>
<td>18.80 min</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>0.1 s</td>
<td>38.63 min</td>
<td>1603</td>
</tr>
<tr>
<td>17</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>18</td>
<td>0.11 s</td>
<td>48.77 min</td>
<td>90,494</td>
</tr>
<tr>
<td>19</td>
<td>0.08 s</td>
<td>32.03 min</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>0.08 s</td>
<td>15.52 min</td>
<td>1</td>
</tr>
<tr>
<td>Scen</td>
<td>First solution (Seq)</td>
<td>Best solution (Seq)</td>
<td>Comparison of tree properties (Seq vs Parₜₜ)</td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>---------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>Nr#</td>
<td>Time taken</td>
<td>Solution cost</td>
<td>Branch number</td>
</tr>
<tr>
<td>21</td>
<td>0.15 s 127.20 min</td>
<td>103,335</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>0.08 s 26.80 min</td>
<td>12 8</td>
<td>12</td>
</tr>
<tr>
<td>23</td>
<td>0.08 s 37.42 min</td>
<td>24 14</td>
<td>14</td>
</tr>
<tr>
<td>24</td>
<td>0.1 s 53.75 min</td>
<td>15,433</td>
<td>33</td>
</tr>
<tr>
<td>25</td>
<td>0.09 s 40.52 min</td>
<td>2 26</td>
<td>35</td>
</tr>
<tr>
<td>26</td>
<td>0.14 s 107.02 min</td>
<td>94,101</td>
<td>28</td>
</tr>
<tr>
<td>27</td>
<td>0.09 s 60.25 min</td>
<td>985 22</td>
<td>38</td>
</tr>
<tr>
<td>28</td>
<td>0.08 s 19.53 min</td>
<td>13 15</td>
<td>15</td>
</tr>
<tr>
<td>29</td>
<td>0.15 s 136.13 min</td>
<td>82,727</td>
<td>36</td>
</tr>
<tr>
<td>30</td>
<td>0.09 s 37.97 min</td>
<td>53 14</td>
<td>16</td>
</tr>
<tr>
<td>31</td>
<td>0.09 s 47.37 min</td>
<td>230 5</td>
<td>21</td>
</tr>
<tr>
<td>32</td>
<td>0.1 s 56.88 min</td>
<td>7150 47</td>
<td>49</td>
</tr>
<tr>
<td>33</td>
<td>0.09 s 59.48 min</td>
<td>21 30</td>
<td>32</td>
</tr>
<tr>
<td>34</td>
<td>0.11 s 57.97 min</td>
<td>112,580</td>
<td>23</td>
</tr>
<tr>
<td>35</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>36</td>
<td>0.08 s 26.80 min</td>
<td>1 11</td>
<td>16</td>
</tr>
<tr>
<td>37</td>
<td>0.16 s 157.72 min</td>
<td>79,724</td>
<td>35</td>
</tr>
<tr>
<td>38</td>
<td>0.09 s 55.53 min</td>
<td>1078 19</td>
<td>29</td>
</tr>
<tr>
<td>39</td>
<td>0.08 s 44.53 min</td>
<td>1 9</td>
<td>14</td>
</tr>
<tr>
<td>40</td>
<td>0.12 s 139.03 min</td>
<td>352,857</td>
<td>43</td>
</tr>
</tbody>
</table>

Max Branch Depth | Avg Branch Depth | Total Internal Nodes | Feasible solution branches | Pruned branches (Cost) | Pruned branches (FCFS) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>21</td>
<td>34,877</td>
<td>800</td>
<td>25,730</td>
<td>8349</td>
</tr>
</tbody>
</table>

Sequential program (rounded average):

(continued on next page)
Table B.1 (continued)

<table>
<thead>
<tr>
<th>Seq</th>
<th>First solution (Seq)</th>
<th>Best solution (Seq)</th>
<th>Comparison of tree properties (Seq vs PAR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scen</td>
<td>Time taken</td>
<td>Solution cost</td>
<td>Branch number</td>
</tr>
<tr>
<td>----</td>
<td>--------------</td>
<td>--------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Parallel program on 8 cores (rounded average):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ratio of sequential with respect to parallel:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

References


