Parallel Integer Optimization for Crew Scheduling*

Panayiotis Alefragis, Peter Sanders, Tuomo Takkula, Dag Wedelin

Abstract

Performance aspects of a Lagrangian relaxation based heuristic for solving large 0-1 integer linear programs are discussed. In particular, we look at its application to airline and railway crew scheduling problems. We present a scalable parallelization of the original algorithm used in production at Carmen Systems AB, Göteborg, Sweden, based on distributing the variables. A lazy variant of this approach which decouples communication and computation is even useful on networks of workstations. Furthermore, we develop a new sequential *active set strategy* which requires less work and is better adapted to the memory hierarchy properties of modern RISC processors. This algorithm is also suited for parallelization on a moderate number of networked workstations.

Keywords: airline crew scheduling, combinatorial optimization, Lagrangian relaxation, memory hierarchy, parallel 0/1 integer linear programming.

1 Introduction

In this paper we describe our work on improving the performance of the 0-1 integer optimizer [48] developed at Chalmers, and used in the Carmen crew scheduling system [4, 16], which is in use at most major European airlines. The main contribution of this paper is to develop this optimizer further with respect to performance. For the sequential algorithm this includes improvements both on the algorithmic level and on the implementation level. We also present several approaches to parallelization useful for different architectures, emphasizing approaches that can be parallelized on networks of workstations. For the Carmen system, this allows an airline to use its existing hardware more efficiently, but is also challenging since the inter-node communication is much slower than on dedicated parallel machines. The work has been done within the ESPRIT project PAROS (Parallel large scale automatic scheduling) which deals with crew scheduling in the airline and railway industry.

The main source of our problems is *crew pairing optimization*, see [2, 4, 27, 29] for surveys. Crew pairing is a crucial step in the chain of scheduling problems solved by an airline or railway. For an airline, the problem input is a set of *legs*, which are non-stop flights (from a specified time period), and a set of rules implicitly defining how legs may be combined. The output is a set of *pairings*, which are sequences of legs over one or several days that begin and

*This work has been supported by the ESPRIT HPCN program, project PAROS

end at a crew base. Each pairing has a well defined cost, and the desired solution is a set of pairings that covers all legs at minimal cost. The rules to which each pairing has to comply to, can be very complicated and consist of international and national regulations, union agreements, company rules, quality of life and soft fairness rules etc. Due to the complicated rules and the large problem sizes (up to 50000 legs for some airlines), pairing problems as such are not modelled directly as mathematical programming problems. Often there is some top level heuristic that selects a part of the entire crew pairing problem to be solved by *pairing generation*, where a number of legal pairings are generated, and *pairing selection*, where a cost effective subset of these pairings is selected. Many approaches formulate the pairing selection problem as a set partitioning or set covering problem

$$\begin{array}{rc} & \min\ c^T x \\ \text{s.t.} & Ax \sim \mathbf{1} \\ & x \in \{0,1\}^n \end{array} \tag{1}$$

where $A \in \{0,1\}^{m \times n}, c \in \mathbb{Q}^n$ and $\mathbf{1}$ is a vector of all ones. The relation $\sim$ is either '=' in the set partitioning case or '$\geq$' in the set covering case (used when also *deadheads*, i.e. passive transfers of crew are allowed). The rows of A correspond to the flight legs and the columns of A correspond to the pairings, with $A_{ij} = 1$ if and only if leg i is operated by pairing j. In addition, there are usually a few additional *base capacity constraints* which have a more general form and model the availability of crews at different home bases. Throughout the paper we will use N to denote the number of nonzeros in A.

An important modelling feature of the Carmen system [4, 16] is that rules are entered in a rule language and handled by an independent rule system. This avoids hard coding of rules into the model, which is particularly useful in Europe where the rules can be very different for different airlines and change frequently. The rule system is used by the pairing generator which is based on a pruned tree search generating a subset of all possible pairings. For a comparison with other approaches to pairing generation see Section 1.1.

Our optimizer is used for the selection problem and can be viewed as a Lagrangian based heuristic, which efficiently finds very good integer solutions to most problem instances. These problems can become very large, and commonly exceed the capability of currently available LP-codes even when it comes to solving the LP-relaxation in a reasonable time. Problems can have up to 10^6 variables (pairings) and 100 to 10^4 constraints (legs). They are very sparse, usually having about 5 to 10 nonzeros per column.

The basic algorithms and most of the implementation work of this paper are applicable also to other kinds of large scale integer optimization problems. This gives an additional motivation of our work which is independent of the specific crew scheduling application, which we prefer to see as an important case study. In this wider perspective it is also relevant to note that whether a heuristic or exact optimization method is preferred can only be answered in a given application context.

We conclude this introduction with a review of related work. In Section 2 we briefly describe the original algorithm on which our optimizer is based. In Section 3 we consider several performance improvements of the sequential optimizer, including a new active set strategy which directs the work effort to the most important part of the problem. In Section 4 we develop a family of parallelizations of the original algorithm and Section 5 does the same for the active set strategy. The last section is devoted to conclusions and future work.

1.1 Related Work

To overcome the computational difficulties of pairing generation and selection, a variety of techniques have been developed, especially concerning the way in which pairing generation and pairing selection interact. The reason for separating the solution process in the two steps of generation and optimization is due to the difficulty of handling the rules directly in an optimizer. At the same time this creates problems, mainly that one usually cannot generate all legal pairings and that the integer optimization problems become intractably large.

One way to overcome these problems is the idea of *column generation*, see [19, 21, 23]. This is usually done by relaxing the selection problem into an LP problem, and then generating pairings with negative reduced cost, which therefore may improve the current LP solution. This significantly reduces the number of pairings generated, keeps the optimization problem small and potentially makes it possible to solve also large problems to proven optimality. On the other hand, it also creates other difficulties of how to generate the best legal pairings, and how to find a final integer solution, and in practice some heuristic shortcuts are necessary for large real problems. Many variations exist, and we can give only a few pointers to the literature. The generation problem is usually based on shortest path calculations in a network, where short paths correspond to pairings with low reduced cost, see Lavoie, Minoux and Odier [38] and the in-depth treatment in Desrosiers et al. [23]. Desaulniers et al. [22] describe an approach to handling rules within the network, and also describe how to obtain integer solutions using a branch and bound scheme. Barnhart et al. [8] describe how to obtain integer solutions within a column generation scheme, called *branch and price*.

Other aspects are explored by Vance [47] who describes an approach where *duty periods* (legal one-day sequences of legs) are pre-generated and considered as primitive elements in the generation, by Barnhart and Shenoi [9] who exploit the special structure of long haul problems and by Barnhart et al. [7] who describe an approach for deadhead selection. Chu et al. [18] present a successful implementation.

The Carmen system was not based on column generation since this approach is more difficult to combine with its main modelling feature, the rule language. In contrast to existing column-generation systems where the end user is restricted to a fixed set of predefined parameterized rules, the rule language enables users to formulate their rules themselves. While this approach requires some education on the user side, it greatly enhances the modelling possibilities and allows for very sophisticated scenario testing, features which have been critical for the commercial success of the Carmen system. With this design, the ability of our optimizer to solve very large integer optimization problems has been important, since it eliminates the optimization bottleneck and allows the system to generate the large number of pairings needed for obtaining good final solutions in the selection step. The design is heuristic, but extensive airline benchmarking has shown that for real problems the Carmen system provides solution quality comparable with the best column generation systems. This is possible since different systems make heuristic shortcuts in different ways, either in the modelling stage to be able to solve the resulting problem optimally, or in the solution procedure of a more carefully modelled problem. For a more extensive description and discussion of the Carmen modelling approach see [4].

Turning to the selection problem we first note that even the plain set covering problem is $\mathcal{NP}$-hard, and it is unlikely that there are polynomial algorithms which find approximations within a factor $\ln n$ from an optimal solution [25]. Lagrangian relaxation approaches for set covering [10] have been investigated for railway problems of similar size to the problems in our test set, see e.g. [14, 15, 17]. Hoffman and Padberg [34] apply branch and cut techniques to smaller set partitioning problems with base constraints. Marsten et al. [39] apply the interior point solver of the linear programming package CPLEX [35] for set partitioning and covering and experience that for the problems they address, the barrier code outperforms both the primal and the dual simplex, even regarding the time for reoptimization. Beasley and Cao [11] use a generic integer programming formulation to the crew scheduling problem.

With respect to parallelization we point out that compared to conventional integer linear programming codes based on branch and bound, parallelization of our optimizer is more difficult since our algorithm is an iterative numerical method. We contrast this with the quite straightforward parallelization of the branch and bound search (or search in general), as described in [24], where the tree shape of the computations implies some very coarse grained parallelism. Our situation is much closer to parallel approaches to linear programming [3, 12, 13, 32].

2 The original algorithm and its implementation

Since our approach has been described in detail in [48] we give only a brief summary for the purposes of this paper.

One way to understand the algorithm is to view it as a heuristic based on Lagrangian relaxation [26]. Consider problem (1) with a general non-negative integer right hand side b and equality relation (inequalities are handled by implicitly adding binary slack variables, see [48]). The unconstrained optimization problem

$$\min_{x \in \{0,1\}^n} \left\{ c^T x + y^T (b - Ax) \right\} \tag{2}$$

is a Lagrangian relaxation of (1) for any value of y. It can be written as

$$y^T b + \min_{x \in \{0,1\}^n} \bar{c}^T x \tag{3}$$

where $\bar{c}^T = c^T - y^T A$ is the *reduced cost* with respect to y. This relaxation is useful to us since

- For any y, (3) is trivial to solve. For every variable, it is sufficient to look at the sign of $\bar{c}_j$, setting $x_j := 1$ if $\bar{c}_j < 0$, and $x_j := 0$ if $\bar{c}_j > 0$ (for $\bar{c}_j = 0$ any value of x_j is optimal) [1]. We will be interested in relaxations where $\bar{c}$ does not contain any zeros and for which the solution is therefore unique.

[1] Note that we do not assume that y is chosen so that $\bar{c} \geq 0$, which is the case when it corresponds to an optimal basic feasible solution in the linear programming sense.

- If an optimal solution to (2) happens to be feasible also for (1), then it is also an optimal solution to (1).

As a first step towards understanding the algorithm, we consider a simple iterative algorithm attempting to find a vector y for which these properties hold. We observe that with a single component y_i of y, we can control all the values (and signs) of the components $\bar{c}_j$ of $\bar{c}$, for which $a_{ij} = 1$. This affects the solution x to (2) and thereby also the value of the left hand side of constraint i. The idea of the algorithm is to iteratively consider *one constraint at at time* and update y_i so that the solution x to (2) is feasible for constraint i. We refer to this as a *constraint update.* Usually there is an interval in which y_i can be chosen, and we then choose y_i in the middle of this interval. This can be shown to correspond to a coordinate search in y_i for the piecewise linear *dual problem*

$$\max_{y} \min_{x\in\{0,1\}^n} \left\{c^T x + y^T(b - Ax)\right\}. \tag{4}$$

The simple algorithm introduced in the previous paragraph works only for some easier problems. The most fundamental reason is that a vector y with the desired properties can be shown to exist only if there is a unique integer solution to the LP-relaxation of (1). For nontrivial problems, many components of $\bar{c}$ instead converge towards 0 and a solution feasible for (1) cannot easily be found.

To remedy this, we introduce a heuristic element in the constraint update, which disturbs the costs in order to make the algorithm converge to a feasible solution of (1). The basic structure of updating $\bar{c}$ considering one constraint at a time remains, but the constraint update is modified to force $\bar{c}$ away from 0. The strength of the heuristic element of the algorithm is controlled by the parameter κ, where 0 means no heuristic and 1 gives maximum effect. To obtain convergence with a minimal disturbance of the costs, the heuristic element

$\bar{c} := c$, $\kappa := 0$, $x := 0$
while there are infeasible constraints **do**
    increase κ to some value in [0,1]
    for each constraint in some random order **do**
        update that constraint
for $1 \le j \le n$ **do**
    if $\bar{c}_j < 0$ **then** $x_j := 1$ **else** $x_j := 0$

Figure 1: Top-level description of the algorithm

is slowly increased during iteration (the *κ schedule*) until a sign pattern corresponding to a feasible integer solution appears in $\bar{c}$, see Figure 1. For best results, the entire algorithm is repeated with refined schedules such that the increase is very slow where previous trials have indicated that convergence is possible. Usually, four to five trials are enough to yield good solutions for our problems.

2.1 The heuristic constraint update

We now explain in detail the constraint update of constraint i. For the update we must remember the additive contribution of the constraint to $\bar{c}$, which we can do in the sparse vector $s^i \in \mathbb{Q}^n$. This allows us to generalize the definition of $\bar{c}$ to

$$\bar{c} = c + \sum_i s^i \tag{5}$$

where s^i can now be of a more general form than $-y_i A_{i\cdot}$. Although s^i is of the same length as $\bar{c}$, only indices corresponding to variables of constraint i are of interest (all other components are 0), so it can be implemented as a short dense vector together with a separate short vector Z^i which is an ordered set of indices to the variables of the constraint. Prior to the first update all components of s^i are initialized to 0.

The first step of the constraint update is to cancel out the effect of the last update of constraint i from $\bar{c}$. The result is the sparse vector r^i, computed as

$$r^i := \bar{c} - s^i. \tag{6}$$

In the actual implementation this operation requires the use of the index vector Z^i to select the right components of $\bar{c}$. We then determine the *critical values* r^- and r^+ as the b-smallest and the $(b+1)$-smallest components of r^i, considering only indices in Z^i (we consider r^- and r^+ as local variables and drop the index i for notational convenience). The variables associated with these values are called *critical.* We compute the *Lagrangian dual* which makes constraint i feasible as

$$y_i := \frac{r^- + r^+}{2}. \tag{7}$$

Note that the interval $r^- \leq y_i \leq r^+$ is where (4) is maximized along the y_i coordinate, so simply using this y_i gives us our first simple algorithm. Now, the heuristic instead uses two different values y_i^- and y_i^+, defined as

$$y_i^- := y_i + \frac{\kappa}{1-\kappa}(r^+ - r^-) \text{ and } \quad y_i^+ := y_i - \frac{\kappa}{1-\kappa}(r^+ - r^-). \tag{8}$$

For $\kappa = 0$, $y_i^- = y_i^+ = y_i$, and for $0 < \kappa \leq 1$, $y_i^+ < y_i^-$. The intention is to replace y_i with y_i^- for variables that would get a negative reduced cost after an update using y_i only, and y_i^+ for variables that would get a positive reduced cost. So the new contribution s^i to $\bar{c}$ is computed to

$$s_j^i := \begin{cases} -y_i^- & \text{if } r_j^i \leq r^-, \\ -y_i^+ & \text{if } r_j^i \geq r^+, \end{cases} \qquad j \in Z^i. \tag{9}$$

Finally, $\bar{c}$ is updated to its new value

$$\bar{c} := r^i + s^i. \tag{10}$$

The net effect is that the components of $\bar{c}$ (considering only indices in Z^i) which are negative after an update with the simple algorithm are lowered, and all components that are positive are raised (values of 0 are avoided by adding noise, see Section 2.2).

A full motivation of the heuristic constraint update would lead too far for the purposes of this paper, but we mention a few central points. Usually, a feasible solution is found for a value of κ larger than 0, but much less than 1. For $\kappa = 0$ the heuristic is not active, and the algorithm can be analyzed within the framework of linear programming. For $\kappa = 1/3$ it can be shown that the algorithm becomes equivalent to a distributed dynamic programming algorithm, which solves the problem optimally if the node/constraint hypergraph (one node per variable, hyperedges defined by the variable sets of the constraints) is acyclic. For $\kappa = 1$ the algorithm can be shown to be a simple greedy algorithm. The parameter therefore effectively creates an interpolation between these different solution principles. Also, from an LP perspective the heuristic improves the speed of convergence of the otherwise slow coordinate search for solving the dual problem.

Some of the test problems used in Sections 3.2 and 5.2 have additional base constraints. These constraints have been handled in a simple way by a straightforward generalization of (7) which maximizes (4) along this coordinate. No heuristic element was used for these constraints. More advanced generalizations of the algorithm involving the solution of knapsack problems are possible, but fall outside the scope of this paper.

2.2 Numerical details

In order to enable reproducible results we give some additional technical details which are useful to obtain the best quality from the algorithm.

There are several different ways to test for convergence. One of the safer ways is to monitor all sign changes in $\bar{c}$ that occur during one pass over all constraints. If there are no sign changes, a feasible solution has been found. This is not a definite convergence criterion however, since it is possible to continue iteration and after additional sign changes a possibly better solution may be found. A mathematically exact termination criterion appears to be difficult to establish.

For the κ schedule we have the following remarks. For the best quality we begin with a number (typically 100) of initial iterations with $\kappa = 0$ or close to zero. The algorithm then moves close to the optimum of the dual before the heuristic mode is entered. Similarly, it is important that the increase of κ is slow when the solution approaches feasibility, especially immediately before a feasible solution is found. A pragmatic way of doing this is to have several trials, with a linear increase of κ in the first trial, and then another schedule for the following trials, where the increase is slower when κ approaches the value for which convergence was previously obtained. If there are ties, such that there are several solutions of equal cost, or if the choice of critical variables is not unique, it may be necessary to break the symmetry by adding some noise to the costs. Adding noise at the beginning is not sufficient since ties due to values in $\bar{c}$ near zero tend to turn up after a number of iterations. A way to solve this is to check $\bar{c}$ regularly, and if some cost is sufficiently close to zero, noise is added. Note that this replaces the use of the δ parameter in [48] which was used for the same purpose.

For some problems, a significantly better quality can be obtained with a slower κ schedule. For others this has little effect, and then it can help to increase the noise or simply to increase the number of trials. We also note that the quality is usually better if the problems are preprocessed so that duplicate rows and columns (where appropriate) are avoided.

2.3 Complexity and solution quality

In this paper we are primarily concerned with improving the performance of the algorithm for the very large problems that we have encountered in the Carmen context. The main benefit of our algorithm, compared to many other methods, is that under the assumption that the number of iterations can be kept constant, the complexity of the algorithm becomes linear in the size of the problem. It therefore scales very well for large problems with up to several million variables, where other methods typically fail. For the quality of the solutions, it has been shown in [48] that the algorithm with default parameter settings gives the same quality as CPLEX [36] for typical crew scheduling problems of interest to us (for sizes solvable by CPLEX). For larger problems this is also supported by the extensive experience with the algorithm within the Carmen system. In the same paper we also find or even improve on the best known solutions for some difficult artificial problems from [5]. Some comparisons made in [14] also support this.

3 Sequential performance improvements

We now turn to a number of sequential improvements that significantly increase the performance of the original algorithm. This forms the basis for the parallel versions we develop in the following sections. In Section 3.1 we consider improvements in the implementation of the original algorithm. In Section 3.2 we present an active set strategy that most of the time works only with the most important part of the problem.

3.1 Basic performance issues

The original implementation is quite straightforward, with the inner loop of the constraint calculation proceeding according to the steps (6) to (10), and with an explicit (sparse) representation of the s-vectors. This performed well on machines where floating point operations dominated the execution time. On modern hardware, memory accesses have almost completely taken over this role. On a typical RISC machine, the main memory is very slow compared to the processor, and smaller and faster first and second level cache memory is common. Therefore, unnecessary random accesses to the main memory should be avoided and preferably be overlapped with useful computations. Another issue is to write the inner loops in such a way that the instruction scheduler can generate code which exploits the superscalarity of the processor and avoids pipeline stalls.

The most obvious difficulty with cache efficiency for our algorithm is the random access pattern to the $\bar{c}$ values which is due to the basic property that the variable costs are updated constraintwise, and for every constraint all the costs for the variables in the constraint must be accessed and updated. This aspect of the algorithm appears to be fundamental, but we

method SetPartitioningConstraint::update(**var** $\bar{c} : \mathbb{Q}^n$, $\kappa : [0, 1)$)
$\bar{c}_{j^-} := \bar{c}_{j^-} - y^- + y^+$ -- patch
$(r^-, r^+, \hat{j}^-, \hat{j}^+) :=$ minIndex2Indirect($\bar{c}$, Z)
$r^- := r^- - y^+$ -- undo offset
$r^+ := r^+ - y^+$ -- undo offset
$y := \frac{1}{2}(r^+ + r^-)$
$\hat{y}^- := y + \frac{\kappa}{1-\kappa}(r^+ - r^-)$
$\hat{y}^+ := y - \frac{\kappa}{1-\kappa}(r^+ - r^-)$
for $1 \le l \le |Z|$ **do**
    $\bar{c}_{Z[l]} := \bar{c}_{Z[l]} - (\hat{y}^+ - y^+)$
$\bar{c}_{\hat{j}^-} := \bar{c}_{j^-} - (y^+ - \hat{y}^-)$ -- patch
$(y^-, y^+, j^-, j^+) := (\hat{y}^-, \hat{y}^+, \hat{j}^-, \hat{j}^+)$

Figure 2: Pseudo-code for set partitioning constraint update.

have found improvements to the constraint update, that roughly increase the speed of the constraint update by a factor of three.

In the new implementation, frequently used constraint types have specialized versions for memory (and memory bandwidth) efficiency in order to exploit arithmetic simplifications. As an example, a specialized pseudo-code for a set partitioning constraint (set covering is very similar) is given in Figure 2. This code is written as a method of a set partitioning constraint class. All variables in the code except for the input parameters are local to the corresponding constraint. So compared to the mathematical description of Section 2.1, the constraint index i is implicitly present for all variables, implying that e.g. y is a scalar in this code.

Only the nonzero indices Z and the previously shifted duals y^+, y^- together with their indices of occurrence, j^+, j^- are stored for each constraint, and the s-vector is not needed. By patching $\bar{c}_{j^-}$ we can for indices in Z access $\bar{c} - s + (y^+, \ldots, y^+)$ directly in the $\bar{c}$ vector and we use this shifted vector instead of an explicit representation of $r = \bar{c} - s$. The new critical elements $(\hat{y}^-, \hat{y}^+)$ can be found by locating the two minimal elements of $\bar{c}$ restricted to indices in Z plus a constant number of scalar operations. Finding the two minimal elements and their indices (in the function "minIndex2Indirect") is almost as easy as finding one minimum alone and can be done using $|Z| + \mathrm{O}(\log |Z|)$ comparisons and an equal number of double indirect memory accesses on the average (refer to [20, Problem 6.2] for some discussion). All other operations have negligible cost. Computing the new (shifted) duals needs only some scalar operations now. Finally, $\bar{c}$ can be updated by subtracting a constant offset $(\hat{y}^+ - y^+)$ for all indices in Z and by patching $\bar{c}_{\hat{j}^-}$. Effectively, we have fused the two vector additions from equations (6) and (10) into a single offset computation. The cost for this is dominated by $|Z|$ double indirect memory read and write operations.

3.2 The active set strategy

The general motivation behind the active set strategy comes from the observation that in our optimization algorithm only a small number of variables become critical during a complete run of the algorithm. Since these are the only variables that affect the numerical progress of the algorithm, it would in principle be possible to ignore all other variables and receive the same result much faster. Unfortunately, we see no efficient way to predict which variables will be needed. New variables become critical during the entire run of the algorithm. Especially towards the end, just before the iteration converges, a relatively large number of new variables become critical. Often, the pairings represented by these new variables would be considered quite inefficient if looked upon in isolation.

While a-priori knowledge of critical variables does not appear to be possible, these observations give intuitive support for the idea of letting the algorithm work only with a smaller number of variables within an *active set*, in order to exclude those variables that are unlikely to become critical. After a number of active set iterations, we regularly do a global iteration over all variables to see if more variables should become active. To avoid an uncontrolled growth of the active set for long running ill-behaved problems, we also deactivate variables that have not been critical for a long time. We also note the similarity with the revised simplex method, and with column generation.

The most obvious performance benefit of the active set is that we do less work per iteration. Additionally, if we let the active set be in a separate data structure, its vector $\bar{c}$ will often fit into the (second level) cache, thus eliminating the slow random access in main memory to $\bar{c}$ discussed in Section 3.1.

Exactly how the global iteration should be implemented depends on a number of quality and efficiency considerations. One way would be to simply run some iterations with all variables. Another approach is to let the algorithm of Section 2 work only on the active set, and use a separate code for doing a *global scan* inspecting all variables in order to update the active set.

The selection heuristic can then work as follows: Hypothetically consider all variables as active. Then look at every constraint and decide which variables would be critical if this constraint were to be iterated first in the next iteration. These variables are then moved into the active set. To enable this test, $\bar{c}$ and all s^i must be implicitly defined also for the variables outside the active set. Since these variables are assumed to have value 0 and clearly were not critical in the previous active set iteration, the new components of s^i are considered to have the value y_i^+ for all $i \in Z^i$, which also defines $\bar{c}$ for all variables through (5). The test for every constraint can then be performed simply by computing r^i from (6) and checking which variables would be critical.

This heuristic works almost as if one would temporarily enter variables into the active set. It results in very simple computations and works well in practice. Also, it turns out that the exact way in which variables are activated is not so critical as long as it is not too restrictive and enters too few variables. For example, the selection heuristic above uses y^+ values in the global scan, but it also works reasonably well to base the selection on low reduced cost, using y-values from a recent iteration of the active set (for base constraints we simply use y). Note however, that the selection criterion is more complicated than just selecting the

```
for i := 1 to m do m0[i] := m1[i] := ∞
for j := 1 to n do
    v := c̄_j
    foreach i such that A_ij = 1 do x := v + y_i^+          -- unroll, remember nonzeros
    foreach i such that A_ij = 1 do                         -- unroll, reuse nonzeros
        if v < m1[i] then                                   -- happens rarely if N ≫ m
            if v < m0[i] then                               -- v < m0[i] ≤ m1[i]
                (m0[i], j0[i], m1[i], j1[i]) := (v, j, m0[i], j0[i])
            else                                            -- m0[i] ≤ v < m1[i]
                (m0[i], j0[i]) := (v, j)
(* The new critical variables are {j0[i] | 1 ≤ i ≤ m} *)
```

Figure 3: Finding critical variables by columnwise traversal.

columns with negative cost in $\bar{c}$, since r^i is computed separately for every constraint.

The global scan can be implemented in several different ways. However, going through all the constraints to update $\bar{c}$ and then again checking all the constraints in order to find critical inactive variables would be as expensive as an iteration of the old algorithm. But we can do better now. We store the nonzero entries of the variables in a columnwise fashion and also traverse the data in this order. For the more general base constraints not only indices but also the A coefficients have to be stored, but the idea is otherwise similar. Figure 3 gives some pseudocode for the pure set partitioning/covering case: We now work one variable at a time, update its $\bar{c}$ entry and then immediately check whether it is a new minimum $m0[i]$ or second minimum $m1[i]$ for some constraint i. Before, accessing $\bar{c}$ implied a cache miss – now the current entry is held in a register. We pay for this by having to hold the y^+ vector, and the minima for all constraints in a frequently accessed array now. These arrays are so small however, that they are likely to fit into the (first level) cache. The nonzeros of the current column will always fit into first level cache and often our code can even hold them in registers. Compared to a global iteration of the basic algorithm we are down from about two cache faults per nonzero $(2N + \mathrm{O}(n))$ to $N/C + \mathrm{O}(n/C)$ where C is number of integers fitting into a cache line.[2] The remaining cache faults stem from sequentially reading indices of nonzero elements and $\bar{c}$. These faults can therefore be hidden by prefetching (e.g., [44]).

This difference can also be observed in practice. Table 1 compares the speeds of rowwise and columnwise codes on a Sun Ultra-1/140. Both versions of the algorithm are highly tuned. The speeds are given for three instances of increasing size. For a small problem, the columnwise traversal is just a little bit faster. In contrast, for a large problem which does not fit into cache, it yields a three times higher raw speed. An analytic expression catching most of these effects can be found in Theorem 3.

Table 2 gives some information about the test data we are using in the central experiments. Throughout the paper, m, n and N denote the number of variables, constraints and nonzeros in the problem, respectively. Column m_{bc} gives the number of base constraints in

[2]Exploiting that a column index fits into 2 bytes we could at least cut that in half.

Table 1: Speed of traversing the problem rowwise and columnwise.

Name	m	n	N	nz/s rowwise	nz/s columnwise
sj_daily_04sc	429	38 148	272 254	6 656 088	6 821 938
sj_daily_34sc	419	156 197	1 243 387	4 666 720	7 692 274
r1kx1M	1 000	1 000 000	4 915 901	1 435 391	4 684 531

Table 2: Test data set.

Name	m	m_{bc}	n	N	N_{sc}	N_{bc}
lh_dl26_02	682	6	642613	6384539	6270518	114021
lh_dl26_04	154	6	121714	1081745	1007435	74310
lh_dt1_11	5287	9	266966	1239454	1216661	22793
lh_dt58_02	5339	9	409350	1892437	1815675	76762
rail2536	2536	0	1081841	10993311	10993311	0
rail2586	2586	0	920683	8008776	8008776	0
rail4284	4284	0	1092610	11279748	11279748	0
rail4872	4872	0	968672	9244093	9244093	0
rail507	507	0	63009	409349	409349	0
rail516	516	0	47311	314896	314896	0
rail582	582	0	55515	401708	401708	0
set3000	500	0	3000	8123	8123	0
sj_daily_00sc	438	0	61084	498359	498359	0
sj_daily_01sc	434	0	45397	337000	337000	0
sj_daily_02sc	429	0	38391	274460	274460	0
sj_daily_03sc	434	0	44837	332764	332764	0
sj_daily_04sc	429	0	38148	272254	272254	0
sj_daily_34	425	6	156197	1354785	1243387	111398
sj_daily_34sc	419	0	156197	1243387	1243387	0

Table 3: Computational results with optimized active set code.

	prob1		no active set		active set		
Name	CPU [s]	obj	CPU [s]	obj	CPU [s]	obj	best
lh_dl26_02			24595	733110	421	733110	733110
lh_dl26_04			4741	339220	60	339220	339215
lh_dt1_11			9075	16758592	470	16758625	16758506
lh_dt58_02			12514	16538051	722	16537995	16537967
sj_daily_00sc	857	267094	1111	266935	93	267014	266924
sj_daily_01sc	463	266236	522	266226	34	266204	266199
sj_daily_02sc	371	261303	381	261272	32	261272	261272
sj_daily_03sc	505	266047	570	266037	33	266037	266016
sj_daily_04sc	201	261385	115	261358	30	261358	261358
sj_daily_34			7871	260530	460	261907	260386
sj_daily_34sc	1895	259988	2408	259896	102	259896	259896
rail507	737	174	773	175	54	175	174
rail516	336	182	301	182	93	182	182
rail582	1138	211	800	211	73	211	211
rail2536	38541	693	31747	697	1654	700	691
rail2586	27118	948	24082	956	1230	955	947
rail4284	43202	1068	35133	1077	3036	1081	1065
rail4872	32199	1536	32515	1542	2269	1543	1534
set3000	17	313	10	314	7	313	312

the problem and the columns N_{sc} and N_{bc} break down N in the number of nonzeros in the set covering constraints and in the base constraints, respectively.

The problems lh$\star$ and sj$\star$ are real world airline crew scheduling and railway crew scheduling problems, respectively, and these have base constraints. Note, that although the number of base constraints is small compared to the total number of constraints, these are quite dense. The rail$\star$ problems have been used in the FASTER competition organized by Ferrovie dello Stato and the Italian Operational Research Society, *AIRO*, [15, 17]. These problems have only three different cost coefficients. The problem set3000 is a small but difficult problem which CPLEX [36] does not solve well. All problems are available from the authors. Several problems are so large that a code like CPLEX is not able to find a solution even to the LP-relaxation.

In Table 3 we give a comparison of typical results between the old *prob1* code which is in use at Carmen Systems, and the new *paqs* code without and with the active set strategy, on one processor of a Sun Ultra Enterprise 10000/249. The runtimes are given as *user time*, that is, the CPU time in seconds dedicated to the computing process. We also give the best known results, some from our algorithm with other parameter settings, others from other sources [6, 31]. Note that base constraints are implemented only in the new code.

We observe a significant performance improvement of about a factor of 10, without any obvious change in the solution quality. Per iteration the new code is faster already without the active set, but this is not seen in the total runtimes since the parameters of the new code have been optimized for runs with the active set. All problems have been run with the same parameter settings, which are tuned primarily for our problems. It is possible to obtain slightly better results on the rail⋆ problems with some tuning (and more time). In order to further determine the solution quality of the new code we have also run the active set code for more problems in the literature, such as all set covering problems from Beasleys OR-library [42] and the NW⋆ problems of Hoffman and Padberg [34]. Most of these are quite small, so the scalability benefit of our algorithm is not important, but we obtain optimal or best known solutions for most problems with the same standard parameter settings as used above.

4 Parallelizing the original algorithm

In Section 4.1 we first investigate how coarse grained parallelism between different constraints can be identified. While this does not yield a useful parallelization by itself, it is useful for refining the fine grained parallel approach based on distributing variables introduced in Section 4.2. The combination of both approaches is described in Section 4.3. The communication costs are further reduced in Section 4.4.

In order to make our results applicable for different kinds of parallel hardware we will model the processors and the communication costs as abstractly as possible. For our machine model, we assume P processing elements (PEs) interconnected by some network. Each PE is a high performance RISC processor with at least two levels of cache and its own local memory. There may or may not be hardware support for remote memory access. For the communication cost, let $T_{\text{start}} + lT_{\text{byte}}$ denote the time needed to communicate a message of size l and let $T_{\text{coll}}(l)$ be a common bound for the cost of a global broadcast or reduction for operands of length l. Note that on many high performance networks these collective operations can be implemented such that $T_{\text{coll}}(l) = \mathrm{O}(T_{\text{start}} \log P + T_{\text{byte}} l)$. Communication cost is compared with the cost for internal computations. For this comparison, we use T_{nz}, the time needed by the sequential algorithm per nonzero element of the constraint matrix, i.e., the time spent per iteration divided by the number of nonzeros, N, for this comparison. In reality, the time per iteration is a function of n, m and N which is nonlinear due to cache effects. However, for large problems of roughly similar size and density, the sequential time per iteration will grow almost linearly in N.

4.1 Parallelism between constraints

Parallelizing the basic algorithm is not trivial because its iterative nature only allows parallelization within an iteration. Conventional wisdom suggests to look for the most coarse grained source of parallelism available. In our case this means the loop updating constraints. Indeed, constraints which do not share nonzeros can be updated in parallel. If we consider the constraint dependence graph where the nodes are constraint numbers and edges connect constraints which share variables, we can identify subsets of independent constraints using a graph coloring algorithm. All the constraints colored with the same color are independent and can therefore be iterated in parallel.

The $\mathcal{NP}$-completeness of the graph coloring problem forces us to use heuristics to find the coloring. We have implemented a simple first fit coloring heuristic and also an algorithm by Mehrotra and Trick [40] and the latter turns out to need a few percent less colors, sometimes up to 25 % less.

Coloring might be further improved using the following observation: Our optimization algorithm does not really require all constraints to be updated every global iteration. Rather, we can achieve the same effect if all constraints are considered equally often on the average. Therefore in order to perform k updates of all constraints we are free to make k iterations using the same coloring or to use one "multi-coloring" where each node is colored by k different colors which must not coincide with any of the colors of adjacent nodes.

For crew scheduling, constraint parallelism alone is not useful. Since $n \gg m$ there are many constraint dependencies and, even worse, we have to update a huge $\bar{c}$ vector after every iteration of a color. Since the update of the reduced cost vector is replicated over all processors and takes about 50% of the iteration update time, the best possible speedup cannot be greater than 2 in the general case. For special structured problems, for which a fast suboptimal graph partitioning algorithm can yield a reasonable partition of the variables, we expect better results as each processor would only have to update a part of the reduced cost vector. The communication costs can also be mitigated by updating $\bar{c}$ less frequently but our experiments showed that this harms convergence before the communication costs get low enough to be useful. However, constraint parallelism can be a good choice for instances with $m \gg n$ on shared memory hardware. In Section 4.3 we explain how independent constraint groups can be used to improve the variable parallel approach discussed in the next section.

4.2 Parallelizing by distributing variables

Although the parallelism in the innermost loops is very fine grained, it is much easier to use than constraint parallelism since it does not depend on the problem structure.

Our general approach is to make PE k responsible for some subset V_k of variables, i.e., PE k stores the entries of $\bar{c}$ and the nonzeros referring to the variables in the subset V_k. There are two loops we need to parallelize for set covering constraints. Finding critical elements (the function 'minIndex2Indirect' in Figure 2) and adding a constant offset (the for-loop in Figure 2). The latter is easy – just broadcast the offset and perform the remaining operations locally. Finding the critical elements is only slightly more difficult. First determine the two locally minimal elements r^-, r^+ and their positions j^-, j^+ and then compute the global critical elements using a global reduction with the associative operator

$$(r_1^-, r_1^+, j_1^-, j_1^+) \otimes (r_2^-, r_2^+, j_2^-, j_2^+) :=
\begin{cases}
(r_1^-, r_1^+, j_1^-, j_1^+) & \text{if } r_1^+ \leq r_2^- \\
(r_1^-, r_2^-, j_1^-, j_2^-) & \text{if } r_1^- \leq r_2^- < r_1^+ \\
(r_2^-, r_1^-, j_2^-, j_1^-) & \text{if } r_2^- < r_1^- \leq r_2^+ \\
(r_2^-, r_2^+, j_2^-, j_2^+) & \text{if } r_2^+ < r_1^-
\end{cases} . \tag{11}$$

Note that such operations are easy to parallelize using a logarithmic number of communication steps. MPI even offers library routines for parallelizing arbitrary associative operations [45].

Load balancing and analysis

We now outline how the following bound on the expected execution time for the simple variable parallel algorithm can be obtained:

Theorem 1 *Let I denote the number of required iterations, N the number of nonzeros, T_{nz} the time per nonzero, m the number of constraints and T_{coll} the time needed to evaluate the associative operator defined in (11) on P PEs. Then the expected parallel execution time is bounded by*

$$\mathbf{E}[T_{\mathrm{par}}] \leq I \left(T_{\mathrm{nz}} \frac{N}{P} + \mathrm{O}\left(\sqrt{\frac{N}{P} T_{\mathrm{nz}} m \log P} + m \log P + m T_{\mathrm{coll}} \right) \right).$$

Note that we can get almost perfect speedup if $N \gg mP \log P$ and $NT_{\mathrm{nz}} \gg mT_{\mathrm{coll}}$. Unfortunately, the latter will only be the case for very large N/m and small P.

It is clear that we need Im collective computations of the associative iterator. *If* all PE would have the same amount of work to do in every constraint update, we would only have to account internal work $T_{\mathrm{nz}}N/P$ per iteration. How should the variables be distributed to the PEs to come close to that ideal? Let l_{ik} denote the number of nonzeros on PE k for constraint i. $W := \sum_{i=1}^{m} \max_{k=1}^{P} l_{ik}$ should be close to N/P in order to achieve low *load imbalance*[3]. This looks like a nontrivial problem. Since the PEs have to synchronize after every constraint update, a single partitioning has to keep load imbalance small for all (or at least almost all) individual constraint updates. Fortunately, randomization saves the situation. We simply distribute the variables randomly. Let N_i denote the number of nonzeros in constraint i, and $W_i = \max_{k=1}^{P} l_{ik}$ the maximum load for constraint i. W_i can be bounded using the following quite versatile criterion:

Lemma 1 *Given n subproblems of size $l_1, \ldots, l_n$ which are distributed uniformly and independently at random over P PEs. Let $L := \sum_{i=1}^{n} l_i$ and $l_{\max} := \max_{i=1}^{n} l_i$. The maximum expected load received by any PE is bounded by*

$$\frac{L}{P} + \mathrm{O}\left(\sqrt{\frac{L}{P} l_{\max} \log \frac{PL}{l_{\max}}} + l_{\max} \log PLl_{\max} \right).$$

Proof: Let L_k denote the load of PE k, $\hat{L} := \max_{k=1}^{P} L_k$ and $L_+ := L/P + c(\sqrt{L/P \log P} + l_{\max} \log P)$. We show that $\mathbf{E}[\hat{L}] \leq L_+ + l_{\max}$ for an appropriate choice of the constant c. First, we argue that it suffices to show that $\mathbf{Pr}\left[L_k > L_+\right] \leq l_{\max}/(LP)$. This implies that $\mathbf{Pr}\left[\hat{L} > L_+\right] = \mathbf{Pr}\left[\exists k' : L_{k'} > L_+\right] \leq l_{\max}/L$. Furthermore, since $\hat{L} \leq L$, we get

$$\mathbf{E}[\hat{L}] \;\leq\; L_+ + L \cdot \frac{l_{\max}}{L} \;\leq\; L_+ + l_{\max}.$$

[3]Similarly, we can define load imbalance in other situation as the ratio between the maximum work performed on a processor and the average work. These times are measured between synchronization points. If there are several synchronized phases the overall load imbalance is the average imbalance over all phases

Now, define the 0-1 random variable Z_i to be one if and only if subproblem i is assigned to PE k. Let $Y := \sum_{i=1}^{n} l_i Z_i / l_{\max}$ denote the "normalized load" received by PE k. Let $\mu := \mathbf{E}[Y] = L/(Pl_{\max})$. Since the Z_i are independent and the weights $l_i/l_{\max}$ are in $(0, 1]$ we can apply the weighted Chernoff bound from [43, Theorem 1] to see that

$$P_{\text{fail}} := \mathbf{Pr}\left[Y > (1+\delta)\mathbf{E}[Y]\right] \;\leq\; \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

for any $\delta \geq 0$. This is the same bound as for the unbounded case discussed by Motwani and Raghavan [41, Theorem 4.1] so that the claimed results follows immediately from the relations (4.11) and (4.12) in the same book (setting $\epsilon = l_{\max}/(LP)$). ■

Lemma 1 has a more general form than we need here since we use this later. Setting $l_j := A_{ij}$ and $l_{\max} = 1$ and using that $P \leq N, N_i \leq N$ implies $\log(N_i P) = \mathrm{O}(\log N)$, we get

$$\mathbf{E}[W_i] = N_i/P + \mathrm{O}(\sqrt{N_i/P \log(N_i P)} + \log(N_i P)).$$

$$\text{Hence}\quad \mathbf{E}[W] = \sum_{i=1}^{m} \mathbf{E}[W_i] \;\leq\; \sum_{i=1}^{m} \left(\frac{N_i}{P} + \mathrm{O}\left(\sqrt{\frac{N_i}{P}\log N} + \log N\right)\right)$$
$$= \frac{N}{P} + \mathrm{O}\left(m \log N + \sqrt{\frac{\log N}{P}} \sum_{i=1}^{m} \sqrt{N_i}\right)$$
$$\leq \frac{N}{P} + \mathrm{O}\left(m \log N + \sqrt{\frac{N}{P} m \log N}\right).$$

■

4.3 Parallelizing over both constraints and variables

Constraint parallelism gives us coarse granularity but little parallelism while variable parallelism gives us plenty of parallelism (for large problems) yet fine granularity. These two advantages can be combined by using a variable parallel approach yet iterating a set of independent constraints together. That is, we use only a single vector valued broadcast respectively reduction for all the constraints in a color class.

Using a similar argument as before we get the following bounds on the parallel execution time:

Theorem 2 *Let I denote the number of required iterations, N the number of nonzeros, T_{nz} the inner loop time per nonzero, m' the number of colors needed for the constraint dependence graph and $T_{\text{coll}}(l)$ the time needed to evaluate the associative operator defined in (11) on P PEs for a batch of l inputs. Then the expected parallel execution time of the variable parallel approach with constraint coloring is bounded by*

$$\mathbf{E}[T_{\text{par}}] = I\left(T_{\text{nz}}\frac{N}{P} + \mathrm{O}\left(\sqrt{\frac{N}{P}T_{\text{nz}} m' \log N} + m' \log N + m' T_{\text{coll}}\left(\frac{m}{m'}\right)\right)\right)$$

plus the time needed for coloring. □

Table 4: Parallel speedup on the SGI Origin2000.

# of PEs	4	8	16
speedup (per iteration)	4.17	7.06	7.20

Since $m' \leq m$, we get more coarse grained communication and slightly improved load balancing compared to Theorem 1.

We have made an experimental implementation of this algorithm for the SGI Origin 2000 using its native compiler `#pragmas` for parallelization. For coloring we used a simple $\mathrm{O}(N + m^2)$ time implementation of the first fit heuristic. For the set covering constraints of the (large but not huge) problem instance `lh_dl26_09` with $m = 176$, $n = 464222$ and $N = 4048428$ the first fit graph coloring heuristic colored the constraints using 83 colors. The running time per global iteration is shown in Table 4. We achieve a speedup of 7 on 8 PEs but 16 PEs are no faster. This could be further improved by tuning the reduction operation which is currently based on two barrier synchronizations and a centralized computation of the critical elements. The superlinear speedup for 4 PEs can be explained by cache effects.

4.4 Lazy update of Lagrangian costs

On machines which can effectively overlap communication and computation the communication costs can be hidden to some extent. After receiving the offsets for the constraints in color group i, only the urgent variables, which are the variables occurring in the next color group $i + 1$, are updated. This suffices to iterate group $i + 1$. The remaining updates stemming from group i are performed while waiting for the reception of the offsets for group $i + 1$. This measure incurs a slight overhead for a data structure which distinguishes urgent and normal variables, which has to be rebuilt whenever the order of the constraint groups is changed. Also, libraries such as MPI (and MPI-2) do not yet offer asynchronous reduction and broadcast operations so that careful manual implementations are necessary which can compete with the performance of the synchronous library functions.

However, for small numbers of PEs and large problem instances this approach opens the way to attain some speedup even on workstations connected by Ethernet. Table 5 shows some results for HP 715/100 workstations and switched standard Ethernet. During the scatter of the problem the master PE is assigned a substantially smaller portion of the problem, in order to balance the load introduced by the reduction and broadcast communication overhead of the offsets to the worker PEs.

5 Parallel active set

We start with a simple data parallel approach in Section 5.1 which suffices to explain the main ideas. Section 5.2 reports the implementation approach and first measurements. In Section 5.3 we sketch a refined variant which is able to work on the active set and the global scan concurrently.

Table 5: Speed-up per iteration for the "lazy" variable parallel approach using the original sequential algorithm and two to four PEs with a lazy algorithm and the coloring heuristic from [40].

			number of PEs		
Problem	rows	columns	2	3	4
lh_dl145_00	682	288552	1.73	2.08	2.41
lh_dl145_01	682	289084	1.73	2.08	2.48
lh_dl26_01	676	384632	1.77	2.21	2.59
Average			1.74	2.13	2.49

5.1 A simple approach

The parallel active set strategy opens the way to a more coarse grained parallelization – at least for the global scan. We perform the same operations as the basic algorithm, that is, adding offsets and finding minima. But we do it in a batched way for all the constraints at once. Therefore we can broadcast all m duals together and we only need to perform a reduction operation for a vector valued input of length m.

This implies message lengths of several kilobytes so that startup overhead for communication is no longer the limiting factor, even on networks of workstations. The bandwidth of the network now becomes an issue but is unproblematic for our case where the work per PE, $T_{\mathrm{nz}}N/P$, is large compared to the communication volume $\mathrm{O}(m)$. Using a pipelined implementation of the collective communication operations we will have $T_{\mathrm{coll}}(m) = \mathrm{O}(m)$ on many network topologies.

Even on a slow shared medium like a 1MByte/s Ethernet the situation is not too bad. For example, on 4 PEs and for the instance `lh_dl26_09` used above we get about 30ms for communication and about 160ms for computations on a Sun Ultra-1/140.

We note that the global scan works also for base constraints, which do not present any significant difficulties compared to the difficulty of parallelizing such constraints within the active set. This is an advantage over the variable based approaches to parallelization which become more complicated in this case.

Load balancing and analysis

In the simplest realization of the parallel active set algorithm, the variables are (randomly) distributed to the PEs and PE 0 is additionally responsible for the active set. With this approach, using a similar analysis as before, we get the following runtime bound:

Theorem 3 *Let I' denote the number of required active set iterations, I'' the number of global scans, N the overall number of nonzeros, N' the average number of nonzeros in the active set, T'_{nz} the work per nonzero in the active set, T''_{nz} the work per nonzero for a global scan, m the number of constraints, m'' the maximum number on nonzeros in a column (i.e., the largest pairing), and $T_{\mathrm{coll}}(l)$ the time needed to evaluate the associative operator defined in (11) on P PEs for a batch of l inputs. Then the expected parallel execution time of the* *parallel active set algorithm is bounded by*

$$\mathbf{E}[T_{\text{par}}] \leq I'N'T'_{\text{nz}} + I''\left(\frac{N}{P}T''_{\text{nz}} + \sqrt{\frac{N}{P}T_{\text{nz}}m''\log N} + m''\log N + T_{\text{coll}}(m)\right) .$$

□

Let us compare this to the bound for the variable parallel approach from Theorem 1 and 2. Although $I' > I$, for our application we usually have $I'N' \ll IN$. Furthermore, since $I'' \ll I$ the active set algorithm performs much less work. This effect is magnified by the fact that $T'_{\text{nz}} < T_{\text{nz}}$ and $T''_{\text{nz}} < T_{\text{nz}}$ for large n due the cache friendlyness of both active set iterations and global scans.

From the point of view of parallelization, we have reduced the number of communication calls per global scan to 1 and since $m'' \ll m$ (usually, m'' is a small constant) load balancing is also much improved. The only downside is that the active set iterations are a sequential bottleneck and cannot even be done in parallel to the global scan. We address these problems in Sections 5.4 and 5.3 respectively.

The analysis is again simple except for load balancing. In principle, we could give a good deterministic load balancer now. Nevertheless, randomly permuting the variables is still advisable because otherwise we need an accurate model for the relative computational cost per nonzero and per variable. Due to cache effects, these costs are a complicated function of n, m, N and P. The requirements for load balancing can be deduced from Lemma 1. This time, l_j is the number of nonzeros, $L = N$ and $l_{\max} = m''$. The factor $\log(NP/m'')$ can be simplified to $O(\log N)$ for $N \geq P$.

5.2 Implementation and experiments

For the implementation (in C++), all parts of the parallel active set code depending on a particular parallel environment were isolated in a small module. By avoiding any global or `static` variables, care was taken to be compatible with thread libraries like POSIX threads. However, MPI [45] was chosen as the first parallelization environment. The functions `MPI_Barrier` and `MPI_Reduce` proved to be a perfect match for the operations required by the parallel active set algorithm. These operations are not only simpler to use than shared memory primitives but a good MPI implementation can also come close to the peak performance of the hardware for the long inputs we use. Only setting up the problem is more cumbersome than in our previous experiments using a shared memory machine. It proved to be unproblematic to port the code to LAM [37], `mpich` [30] and the native implementations from Hewlett Packard [33] and Sun Microsystems [46]. The code works on machines from Sun Microsystems, Silicon Graphics and Hewlett Packard and on PCs running NetBSD and Linux.

In practice, the parallel global scan can only be exploited if this task dominates the computation time, which depends on the character of the problem and a number of parameter settings. For some problems global scans are very rare and the sequential active set algorithm will do much less work than the original algorithm. Otherwise, we have a significant parallelization potential for the global scan.

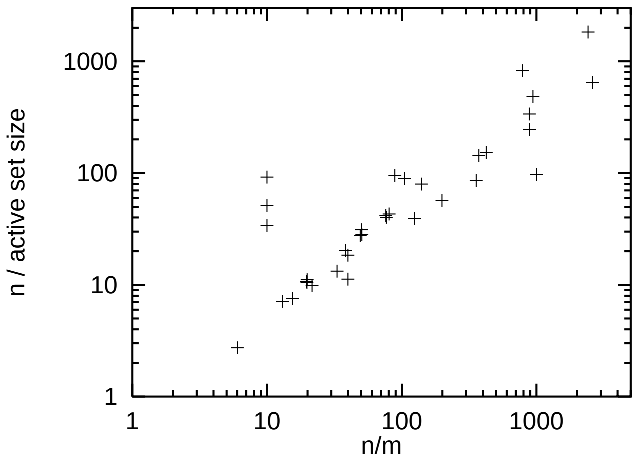

Figure 4: Double-logarithmic plot of ratio between total number of variables and active set size (averaged over a run). The problem suite used mainly consists of problems from Lufthansa and Swedish Railways. The outliers for $n/m = 10$ are artifical problems which are much denser than typical crew scheduling problems [31]

For our crew scheduling problems, and for our preferred parameter settings, the truth lies somewhere in the middle. Sixteen active set iterations per global scan work well. This value is not very sensitive. The size of the active set depends on the individual problem instances and on the desired tradeoff between speed and quality. Figure 4 shows how much larger n is compared to the active set. We see that for constraint matrices with large aspect ratio n/m the active set is very small. Considering the total optimization time of the sequential active set algorithm, the percentage of time spent for the active set iterations can vary between 5-50%, with a corresponding 95-50% for the global scan. All in all, this allows us to combine a significant sequential improvement plus a moderate parallelism.

Since the right balance between active set iterations and global scans depends on so many factors, we have decided to decouple the scalability evaluation of our parallel implementation from this issue and only measure the time per global scan (including communication overhead) in the following tables. Thus we are independent of the overall number of iterations (which fluctuates slightly around 1000 per trial due to the random aspects of our algorithm) and we are independent of the particular balance chosen for the ratio between active set iterations and global scans. Table 6 shows some typical running time results for the global scan on a Sun Enterprise 10000/249, and in Table 7 a Sun Ultra-1/140 network connected by a shared Fast Ethernet. The problems selected are all the larger problems in Table 3. We see that the global scan can be parallelized with a speedup of up to three on four processors even on a network of workstations. For the entire parallel active set algorithm, it follows that we can obtain a performance improvement by a factor of 3–10 due to the improved sequential code and an additional speedup of up to 2.5 for large problems using four networked workstations.

Table 6: Time per 100 global scans on a Sun E10000 (seconds wall clock time)

	number of CPUs							
Name	1	2	3	4	5	6	7	8
lh_dl26_02	78.7	39.8	28.8	20.1	18.5	13.1	11.8	9.2
lh_dl26_04	14.9	5.0	3.1	2.7	2.2	1.5	1.6	1.7
lh_dt1_11	21.0	14.1	11.2	9.2	8.8	8.2	9.3	9.8
lh_dt58_02	31.7	20.3	13.2	11.2	9.1	10.4	12.0	10.8
rail2536	137.1	74.3	47.9	37.6	32.0	28.6	21.6	22.3
rail2586	109.4	54.8	35.7	27.6	25.0	21.7	20.2	18.4
rail4284	149.9	68.7	52.6	40.6	34.6	32.7	31.6	24.1
rail4872	121.8	61.8	47.8	36.2	29.8	24.6	24.4	21.5
sj_daily_34	24.5	14.6	7.1	5.8	5.2	3.2	4.0	3.8
sj_daily_34sc	13.0	5.6	6.7	2.6	2.0	1.7	1.5	1.3

Table 7: Time per 100 global scans on a network of workstations (seconds wall clock time)

	number of workstations					
Name	1	2	3	4	5	6
lh_dl26_02	97.6	49.7	36.7	29.8	27.2	23.0
lh_dl26_04	20.5	9.7	6.7	7.3	5.7	4.6
lh_dt1_11	29.5	23.3	22.2	41.8	39.9	38.2
lh_dt58_02	42.6	32.0	25.2	45.3	46.3	43.5
rail2536	249.6	123.0	98.4	79.4	66.3	62.7
rail2586	148.7	75.6	56.2	49.4	50.0	41.0
rail4284	233.0	118.5	100.6	90.4	78.9	73.4
rail4872	160.4	91.1	76.5	73.4	69.8	69.3
sj_daily_34	31.9	21.2	11.2	11.5	12.2	8.3
sj_daily_34sc	17.5	9.0	7.0	6.2	5.5	5.1

5.3 Concurrent work during active set iterations

In our simple data parallel active set algorithm all PEs but number 0 are idle during active set iterations. An algorithm variant which does not have this problem dedicates PE 0 to the active set and frees it from the work on global scans. Then a global scan can be performed concurrently to the ongoing active set iterations. Although this approach requires the global scan to work with slightly outdated duals, we expect this degradation to be negligible compared to the win in terms of scalability.

A particularly elegant variant of this approach is based on the assumption that the more active set iterations we do the better the results. In this case it is best to simply send new duals whenever a global scan is completed. We do no longer need to tune the ratio between active set iterations and global scans.

5.4 Parallelism inside the active set

Our measurements show that the active set heuristic often works so well that relatively few global scans are required. Although this is good news, it limits the benefits of the parallel algorithms. So the fastest available machine should work on the active set. A next step is to also parallelize the active set iterations using the algorithms from Section 4. Even a small improvement can have a large effect here. The active set parallelization need not be efficient as long as it is fast. For example, if we manage to double the speed of active set iterations, say using four PEs of a symmetric multiprocessor, then the achievable overall speedup predicted by Amdahl's law may double from eight to sixteen. Note that the color classes from Section 4.3 may be much larger for the active set since most constraint dependencies are only due to inactive variables.

6 Conclusions

Based on the "industrial strength" Lagrangian heuristic for solving large sparse 0/1 integer programs in the Carmen system, we have achieved a number of significant performance improvements. On the sequential side, we have not only reformulated the necessary mathematics to better fit modern CPUs with multilevel caches, but with the active set strategy we also have a new algorithm which can handle problems with many variables much more efficiently.

Both the original and the active set approach have been parallelized in different ways. The former scales well on tightly coupled machines and using the lazy update strategy it also achieves some speedup even on networks of workstations. The parallel active set code is even better suited for loosely coupled machines.

The new and much faster implementation is an important step towards significantly reducing one of the main time critical parts of the crew scheduling process, where shorter and more flexible planning cycles can be directly translated into economic benefits for the airlines. The fast and reliable solution of very large problems also opens up for new modelling possibilities, both in scheduling, as well as in other applications where large integer optimization problems have to be solved.

Within the PAROS project, the optimizer is not the bottleneck in the system any more, and the immediate task for Carmen and its partners will be an efficient integration of the parallel optimizer with the parallel pairing generator [28, 1] which also runs on a network of workstations.

References

[1] P. Alefragis, C. Goumopoulos, E. Housos, P. Sanders, T. Takkula, and D. Wedelin. Parallel crew scheduling in PAROS. In *Europar 98*, volume 1470 of *LNCS*, pages 1104–1113, Southampton,UK, September 1998.

[2] R. Anbil, J. J. Forrest, and W. R. Pulleyblank. Column generation and the airline crew pairing problem. In *Documenta Mathematica – Journal der Deutschen Mathematiker-Vereinigung*, number III in extra volume: proceedings of the ICM 1998, pages 677–686, Berlin, 1998. Deutsche Mathematiker-Vereinigung. Electronically accessible through `http://www.mathematik.uni-bielefeld.de/documenta/xvol-icm/17/Pulleyblank.*`.

[3] E. D. Andersen and K. D. Andersen. A parallel interior-point algorithm for linear programming. Technical Report 9808, CORE, Louvain-la-Neuve, Belgium, 1998.

[4] E. Andersson, E. Housos, N. Kohl, and D. Wedelin. *OR in the Airline Industry*, chapter Crew Pairing Optimization. Kluwer Academic Publishers, Boston, London, Dordrecht, 1998.

[5] D. Avis. A note on some computationally difficult set covering problems. *Mathematical Programming*, 18:138–145, 1980.

[6] E. Balas and M. C. Carrera. A dynamic subgradient-based branch-and-bound procedure for set covering. *Operations Research*, 44(6):875–890, 1996.

[7] C. Barnhart, L. Hatay, and E. L. Johnson. Deadhead selection for the long haul crew pairing problem. *Operations Research*, 43(3):491–499, 1995.

[8] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[9] C. Barnhart and R. G. Shenoi. An approximate model and solution approach for the long-haul crew pairing problem. *Transportation Science*, 32(3):221–231, 1998.

[10] J. Beasley. A lagrangian heuristic for set-covering problems. *Naval Res. Logist.*, 37(1):151–164, 1990.

[11] J. Beasley and B. Cao. A tree search algorithm for the crew scheduling problem. *European Journal of Operational Research*, 94:517–526, 1996.

[12] R. Bisseling, T. Doup, and L. Loyens. A parallel interior point method algorithm for linear programming on a network of transputers. *Annals of Operations Research*, 43:51–86, 1994.

[13] R. E. Bixby and A. Martin. Parallelizing the dual simplex. Technical Report SC-95-45, Konrad Zuse Zentrum für Informationstechnik Berlin (ZIB), Berlin, 1995.

[14] A. Caprara, M. Fischetti, and P. Toth. A heuristic algorithm for the set covering problem. In *Proceedings of the 5th IPCO Conference*, volume 1084 of *LNCS*, pages 72–84, 1996.

[15] A. Caprara, M. Fischetti, P. Toth, D. Vigo, and P. L. Guida. Algorithms for railway crew management. *Mathematical Programming*, 79:125–141, 1997.

[16] The Carmen System, version 5.1. Carmen Systems AB, Göteborg, Sweden.

[17] S. Ceria, P. Nobili, and A. Sassano. A Lagrangian-based heuristic for large-scale set covering problems. *Mathematical Programming*. To appear.

[18] H. D. Chu, E. Gelman, and E. J. Johnson. Solving large scale crew scheduling problems. *European Journal of Operational Research*, 97:260–268, 1997.

[19] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, New York and San Francisco, 1983.

[20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, Cambridge, MA, 1990.

[21] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, New Jersey, 1963.

[22] G. Desaulniers, J. Desrosiers, Y. Dumas, S. Marc, B. Rioux, M. Solomon, and F. Soumis. Crew pairing at Air France. *European Journal of Operational Research*, 97:245–259, 1997.

[23] J. Desrosiers, Y. Dumas, M. Solomon, and F. Soumis. *Time Constrained Routing and Scheduling*, volume 8 of *Handbooks in Operations Research and Management Science*, chapter 2, pages 35–139. North-Holland, 1995.

[24] J. Eckstein. Control strategies for parallel mixed integer branch and bound. In *Supercomputing '94*, pages 41–48, Silver Spring, MD, 1994. IEEE Computer Society.

[25] U. Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4):634–652, July 1998.

[26] M. L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, 1981.

[27] I. Gershkoff. Optimizing flight crew schedules. *Interfaces*, 19(4):29–43, 1989.

[28] C. Goumopoulos, E. Housos, and O. Liljenzin. Parallel crew scheduling on workstation networks using PVM. In *EuroPVM-MPI*, volume 1332 of *LNCS*, Cracow, Poland, 1997.

[29] G. Graves, R. McBride, and I. Gershkoff. Flight crew scheduling. *Management Science*, 6:736–745, 1993.

[30] W. Gropp and E. Lusk. *Users Guide for* `mpich 1.1.1`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, University of Chicago, Chicago, USA, 1999. Electronically accessible through `http://www.mcs.anl.gov/mpi/mpich`.

[31] T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem. Working paper, dated April 18, 1996. A previous version was published as Technical Report CS94-25, The Weizmann Institute of Science, Rehovot, Israel, 1994.

[32] J. A. J. Hall and K. I. M. McKinnon. ASYNPLEX, an asynchronous parallel revised simplex algorithm. *Annals of Operations Research*, 81:27–49, 1998.

[33] Hewlett Packard Company. *HP MPI User's Guide, 3rd Edition.* Palo Alto, CA, USA, June 1998. Released with HP MPI V1.4. Electronically accessible through `http://www.hp.com/go/mpi`.

[34] K. L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.

[35] ILOG, Inc., CPLEX Division. *CPLEX Base System, Barrier and Mixed Integer Solver.* Incline Village, NV, USA. Version unspecified.

[36] ILOG, Inc., CPLEX Division. *Using the CPLEX callable library, version 5.0, CPLEX Base System, Barrier and Mixed Integer Solver.* Incline Village, NV, USA, 1997.

[37] Laboratory for Scientific Computing. *LAM 6.1 Release Notes.* University of Notre Dame, USA, 1997. Electronically accessible through `http://www.mpi.nd.edu/lam`.

[38] S. Lavoie, M. Minoux, and E. Odier. A new approach for crew pairing problems by column generation with an application to air transportation. *European Journal of Operational Research*, 35:45–58, 1988.

[39] R. Marsten. RALPH: Crew Planning at Delta Air Lines. *Technical Report. Cutting Edge Optimization.* To appear in *Interfaces.*

[40] A. Mehrotra and M. A. Trick. A clique generation approach to graph coloring. *INFORMS Journal of Computing*, 8:344–354, 1996.

[41] J. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[42] *OR library – a collection of test data sets for a variety of Operations Research problems.* Imperial College Management School, London. Electronically accessible through `http://mscmga.ms.ic.ac.uk/info.html`. Maintained by J.E. Beasley.

[43] P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

[44] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25-2 of *Computer Architecture News*, pages 264–273, New York, 1997. ACM Press.

[45] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI – the Complete Reference.* MIT Press, Cambridge, MA, 1996.

[46] Sun Microsystems Computer Company. *Sun MPI3.0 Guide.* Palo Alto, CA, USA, 1997. Electronically accessible through `http://docs.sun.com`.

[47] P. H. Vance. *Crew Scheduling, Cutting Stock, and Column Generation: Solving Huge Integer Programs.* PhD thesis, Georgia Institute of Technology, August 1993.

[48] D. Wedelin. An algorithm for large scale 0-1 integer programming with application to airline crew scheduling. *Annals of Operations Research*, 57:283–301, 1995.