High Performance Integer Optimization for Crew Scheduling*

Peter Sanders[1], Tuomo Takkula[2], Dag Wedelin[2]

[1] Max-Planck-Institute for Computer Science, Saarbrücken, Germany
[2] Chalmers University of Technology, Computing Science, Göteborg, Sweden

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 and a new sequential *active set strategy* which requires less work and is better adapted to the memory hierachy properties of modern RISC processors. The active set strategy can even be parallelized on networks of workstations.

1 Introduction

In this paper we describe our work on improving the performance of the integer optimizer of the Carmen system [2, 5, 18], which is in use at most major airlines in Europe. The optimizer can solve a wide class of integer linear programming problems (ILP), but here we focus on *pairing optimization*, which is a crucial step in the scheduling process. The optimization problems are then of the set partitioning or set covering type [2]

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

where $A \in \{0,1\}^{m \times n}, c \in \mathbb{Q}_+^n$ and 1 is a vector of all ones. The relation $\sim$ is either '=' in the set partitioning case or '$\geq$' in the set covering case. The rows of A correspond to non-stop flights (*legs*) which are operated by the carrier and which need to be staffed with crews. The columns of A correspond to so-called *pairings*. A pairing corresponds to a crew schedule in terms of a sequence of legs, starting at a home base, and returning (sometimes after a few days) to the home base. Thus $A_{ij} = 1$ if and only if leg i is operated by pairing j. The pairings and their associated cost coefficients are usually computed in a complicated process taking crew utilization, regulations, union agreements, overnight cost, work plan robustness, credit time and many other factors into account. The

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

process of generating the matrix A is called *pairing generation*. It is very time-consuming and discussed in [11]. In addition, there are usually a few additional *base constraints* which have a more general form and model the availability of personnel at different home bases of the company.

Let N denote the number of nonzero entries in the $m \times n$-matrix A. The large problems we are most interested in have up to several hundred thousand variables and typically between a few hundred to a few thousand constraints. They are very sparse, usually having only 5 to 10 nonzeros per column. These problems are among the largest 0-1 problems solved in commercial applications, and generally available commercial solvers such as CPLEX cannot handle problems of this size. Our algorithm does not use branch & bound, but can be viewed as a Lagrangian based heuristic, which can efficiently find solutions to most of these problems in a few hours to within a thousandth of the optimal solution. For other Lagrangian relaxation methods applied to pairing problems in railway industry see for instance [4, 6], which also address problems of similar size using specialized heuristics. Recent work in airline crew scheduling include [3, 8, 12–14, 17]. Usually, the ILPs considered there are smaller than those considered in this paper.

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. It will turn out that the communication cost of our algorithms can be modeled quite abstractly by the cost of a global broadcast or reduction for operands of length m. Let $T_{\mathrm{coll}}(m)$ be a common bound for this time. This communication cost is compared with the cost for internal computations. We use T_{nz}, the time spent per iteration divided by N, for this comparison.

In Sect. 2 we briefly describe the existing optimizer and then discuss performance improvements and a new variant of the algorithm which needs less work and is additionally better adapted to the memory hierarchy. Sect. 3 introduces two promising approaches to parallelization. The implementation and first measurements are presented in Sect. 4. The last two sections are dedicated to conclusions and future work.

2 Sequential Algorithms

2.1 The basic algorithm

Our approach has been described in detail in [18], to which we refer for full detail and for quality comparisons with other algorithms. A simple way to understand the algorithm is to view it as a heuristic based on Lagrangian relaxation (see [9]). Consider problem (1) with a general non-negative integer right hand side b. Then the corresponding Lagrangian relaxation subproblem with equality [1] is

$$\min_{x \in \{0,1\}^n} c^T x + y^T (b - Ax), \quad y \in \mathbb{R}^m. \tag{2}$$

[1] The "$\geq$" variant differs from (2) only by requiring y to be nonnegative.

A general property of the relaxation is that if we can find a vector y so that the solution to (2) is feasible for (1), then we have an optimal solution to (1) (This is related to the dual problem of maximizing (2) with respect to y). The relaxed unconstrained problem (2) can easily be solved optimally by considering $\bar{c}^T := c^T - y^T A$ and setting $x_j := 1$ if and only if $\bar{c}_j$ is negative. In most cases however it is not possible to find a relaxation (defined by y) that has this desirable property. The algorithm therefore introduces a heuristic scheme that makes this possible.

The algorithm proceeds by modifying $\bar{c}$ by considering one constraint. We call this process *iterating* a constraint. For constraint i we define a row index set $z^i := \{j \mid A_{ij} = 1\}$ and a sparse vector s^i representing its contribution to $\bar{c}$, i.e., $\bar{c} = c + \sum_{i=1}^{m} s^i$. The algorithm differs from pure Lagrangian relaxation in the way in which s^i is computed, and the fact that these vectors are kept between iterations.

First, the change in $\bar{c}$ from the last update is cancelled out by computing $r := \bar{c} - s^i$. We then determine *critical values* r^- and r^+ as the b-smallest and the $(b+1)$-smallest elements of r (considering only indices in z^i). The variables associated with these values are called *critical variables*. We compute the *Lagrangian dual* corresponding to constraint i as $y_i := \frac{r^- + r^+}{2}$ and shifted values

$$y^- := y_i - \frac{\kappa}{1-\kappa}(r^+ - r^-), \quad y^+ := y_i + \frac{\kappa}{1-\kappa}(r^+ - r^-)$$

where κ is a control parameter in $[0, 1)$. The new contribution s^i to $\bar{c}$ is updated to

$$s^i_j := \begin{cases} -y^- & \text{if } \bar{r}_j \leq r^-, \\ -y^+ & \text{if } \bar{r}_j \geq r^+, \end{cases} \qquad \text{for } j \in z^i.$$

and $\bar{c}$ is set to its new value $\bar{c} := r + s^i$.

On the top level, this constraint iteration is done for every constraint over and over again. During these iterations κ is slowly increased from 0, until a feasible solution is obtained from $\bar{c}$. Here is a high-level description of the algorithm:

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

The algorithm above is repeated with refined schedules for the increase of κ. Usually, four to five trials are enough to yield excellent solutions for large scale problems. An extension to constraint matrices in $\{-1, 0, +1\}^{m \times n}$ is easy [18].

2.2 Basic Performance Issues

The original implementation used a structured programming approach with a uniform representation of all constraint types which included an explicit (sparse) representation of the s-vectors. This performed well on machines where floating point operations dominated the execution time. But on today's machines memory accesses have almost completely taken over this role. Therefore our new code is object oriented with specialized representations for important constraint types.

A specialized pseudo-code for a set partitioning constraint (set covering is very similar) is given below. Only the set of nonzero indices z and the previous shifted duals y^+, y^- together with their indices of occurrence, j^+, j^- are stored for each constraint – but no s-vector. By patching $\bar{c}_{j^-}$ we can compute $\bar{c} - s$ in constant time up to a collective shift by the old y^+, and we can then avoid the explicit use of the intermediate r vector. The new critical elements 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 [7, 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 for all indices in z and patching $\bar{c}_{j^-_{\mathrm{new}}}$. Effectively, we have fused the two vector additions from the abstract iteration scheme into a single offset computation. The cost for this is dominated by $|z|$ double indirect memory read and write operations.

Specialized code for a set partitioning constraint:

method SetPartitioningConstraint::iterate(**var** $\bar{c} : \mathbb{R}^n$, $\kappa : [0,1)$, z : array of $\mathbb{Z}$)

- $\bar{c}_{j^-} := \bar{c}_{j^-} - y^- + y^+$ -- make $\bar{c}_{j^-}$ comparable with other elements
- $(r^-, r^+, j^-_{\mathrm{new}}, j^+_{\mathrm{new}}) :=$ minIndex2Indirect($\bar{c}$, z, k)
- $(r^-, r^+) := (r^- - y^+, r^+ - y^+)$ -- undo offset
- $y_i = \frac{1}{2}(r^+ + r^-)$
- $(y^-_{\mathrm{new}}, y^+_{\mathrm{new}}) := (y_i, y_i) + \frac{\kappa}{1-\kappa}(r^+ - r^-)(1, -1)$
- **for** $1 \le i \le |z|$ **do** $\bar{c}_{z[i]} := \bar{c}_{z[i]} - (y^+_{\mathrm{new}} - y^+)$
- $\bar{c}_{j^-_{\mathrm{new}}} := \bar{c}_{j^-} - (y^+ - y^-_{\mathrm{new}})$ -- patch
- $(y^-, y^+, j^-, j^+) := (y^-_{\mathrm{new}}, y^+_{\mathrm{new}}, j^-_{\mathrm{new}}, j^+_{\mathrm{new}})$

2.3 An Active Set Strategy

Since $n \gg m$, it seems to be wasteful to go through all the variables all the time. Observations show that the set of variables which has recently been used for critical variables – we will call this the *active set* – remains quite stable most of the time. Since only critical variables affect the numerical progress of the algorithm, one would be tempted to forget about the rest of the variables and only work in the active set. However, closer inspection shows that from time to time variables never considered before make their way into the active set. In particular, towards the end, just before the iteration converges, a number of new variables is pulled in to create a feasible solution.

Out of several logically useful ways to exploit this idea consider the following one: Most iterations of the algorithm from Sect. 2.1 work on a copy of the problem containing only the information relevant for the active set. Note that often this reduced problem or at least its $\bar{c}$-vector will fit into the (second level) cache. From time to time we make a *global scan* to inspect all variables: First, we update the global $\bar{c}$-vector. Then we allow a number of previously inactive variables to become active. More precisely, we consider all those variables which would be critical for some constraint if a true global iteration were performed – in a sense we make a dry run of a global iteration. In order to avoid an uncontrolled growth of the active set for long running ill-behaved problems, we periodically deactivate all variables which have not been critical for a long time.

So far we have already harvested two advantages: We perform less work per iteration and most iterations are more cache friendly. 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. For the global scan we store the nonzero entries of all variables in a column wise fashion and also traverse the data in this order. We now work one variable at a time, update its $\bar{c}$ entry and then immediately check whether it is a new (second) minimum for some constraint. 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 Lagrangian duals 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 $2N + \mathrm{O}(n)$ cache faults to $N/C + \mathrm{O}(n/C)$ where C is number of `ints` 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. Experiments show that column wise traversal is about three times faster than row wise traversal for large problems on a 140MHz Sun Ultra-1. On newer machines with faster clock even higher differences should be expected.

3 Parallelization

Parallelizing the basic algorithm is not trivial because its iterative nature only allows parallelization within an iteration. As discussed above, a single iteration is so fast that we have only rather fine grained parallelism available. Even worse, parallelizing the outer loop is very difficult since many constraints are coupled by common entries of $\bar{c}$. So, we mostly have to rely on the very fine grained parallelism in the innermost loops. In Sect. 3.1 we show how we can achieve useful speed-ups nevertheless and in Sect. 3.2 we extend this approach for the global scan.

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

3.1 Parallelizing by Distributing Variables

We make PE k responsible for some subset V_k of variables, i.e., PE k stores those entries of $\bar{c}$ and those nonzeros which refer to the variables in V_k. We parallelize the innermost loops, i.e., finding minima and adding a constant offset: 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

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

Analysis: We also have to specify *how* the variables should be partitioned. Let l_{ik} denote the number of nonzeros on PE k for constraint i. $W := \sum_i \max_k l_{ik}$ should be close to N/P in order to achieve good load balancing. This looks like a nontrivial problem since a single partitioning should exhibit good load balancing for all constraints. Fortunately, randomization saves the situation. We simply distribute the variables randomly. Using Chernoff bounds (e.g., [15]) it can be shown that $W = N/P + \mathrm{O}(\sqrt{N/P}\ \log N)$ with high probability. As long as $N/m \gg P \log P$ this implies good load balance. This condition is fulfilled for all problem instances under consideration.

A more severe condition on the problem size is that the local computations should dominate the communication time, i.e., we need $T_{\mathrm{nz}} \frac{N}{P} \gg m T_{\mathrm{coll}}$ if we iterate one constraint at a time. Unfortunately, this will only be the case for very large problems even on tightly coupled parallel machines.

Parallelizing over both constraints and variables: We do not really need a separate collective operation for each constraint. Constraints which do not share nonzeros can be iterated independently. 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 using only one vector valued reduction and broadcast operation.

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 graph coloring heuristic colored the constraints using 83 colors. We obtained a speedup of 7 on 8 PEs but 16 PEs were no faster. This could be further improved by tuning the reduction operation.

Several lessons can be learned from this simple experiment: Good performance can be achieved for large problem instances on machines which allow low latency interaction between PEs. Bandwidth is of secondary importance in this case. Graph coloring reduces the synchronization overhead by a dearly needed constant factor but more should not be expected since large problem instances have a quite dense dependence graph. In a future version we plan to investigate the possibility to achieve coarser granularity by *multi-coloring*: Color each constraint with k colors. This suffices to iterate each constraint k times in unspecified order. This should yield more constraint parallelism since we relax the requirement to iterate each constraint in each iteration.

3.2 Parallel Active Set

The active set heuristic opens the way to a more coarse grained parallelization – at least for the global scan. We perform the same operations – adding offsets and finding minima – as the basic algorithm. But we do it in a batched way for all 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 the 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 $T_{\mathrm{coll}}(m) \in \mathrm{O}(m)$ on most networks.

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

In its simplest form, all variables are (randomly) distributed to the PEs and PE 0 is additionally responsible for the active set. So during the active set iterations the other PEs remain idle. This works well if the active set is so small that the global scan dominates the execution time.

Currently, we are working on variants with a dedicated fast PE for the active set (possibly even a multiprocessor). Concurrently, the other PEs scan their variables using slightly outdated duals or generate new pairings [10].

4 Implementation and Experiments

For the implementation, 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 [16] was chosen as the first parallelization platform. The functions `MPI_Barrier` and `MPI_Reduce` proved to be a perfect match for the operations required by the global scan of 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, `mpich` and the native implementations from SGI and SUN. The code works on machines from Sun, SGI and HP.

As expected, the global scan scales well now even on slow networks but this can only be exploited if this task dominates the computation time. Fig. 1 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.

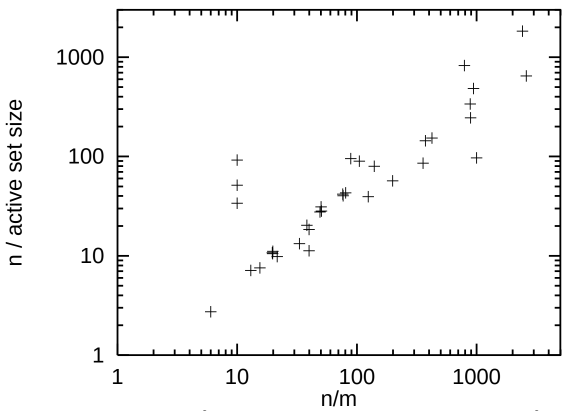

Fig. 1. Double-logarithmic plot of ratio between total number of variables and active set size (averaged over a run). The problem suite used mainly consists of problems from Lufthansa and Swedish Railways. The outliers for $n/m = 10$ are artificial problems which are much denser than typical crew scheduling problems [19].

How exactly the parallel active set can be used in practice depends on the character of the problem. For some problems all variables have to be inspected very rarely 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. 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. This gives us a significant sequential improvement plus a moderate parallelism.

Table 4 shows some typical running time results on a Sun Ultra-1 network connected by a shared Fast Ethernet for a cross-section of our problem suite. We note that several problems are so large that a code like CPLEX is not able to find a solution even to the LP-relaxation. The code “prob1” is currently used in production at Carmen. Times in seconds are given for the optimized sequential code without active set, and then for the active set strategy running on 1, 2 and 4 workstations. Objective function values are also given for the new code with and without the active set. Since the active set code works in a different way it is here not possible just to compare time per iteration as before. This also means that the number of required iterations in individual runs can vary considerably, so that very large and very small overall speedups are possible. Even if all variables are forced to be active, our sequential tuning efforts roughly double the speed for instances where $\bar{c}$ fits in the cache. We also see that we achieve a performance improvement by a factor of 3–10 due to the improved sequential code and an additional speedup of three for large problems using four networked workstations. We finally note that there is no significant decrease in the quality due to the active set strategy.

Table 1. Solution quality and optimization time comparison for the code without active set and the parallel active set code.

problem			*prob1*	no active set		active set, no. of PEs				best
name	m	n	time	time	obj	1	2	4	obj	known obj
sj_daily04sc	429	38148	296	94	261358	37	31	24	261358	261358
sj_daily34sc	419	156197	1510	885	259896	139	106	50	259896	259896
lh_dl26_02	682	642613	9962	4071	733110	843	412	294	733110	733110
lh_dl26_04	154	121714	1256	373	339220	216	105	60	339220	339220
lh_dt1_11	5287	266966	1560	765	16758592	298	78	66	16758625	16758592
lh_dt58_02	5339	409350	2655	2305	16538051	924	406	207	16537995	16537995

5 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 multi-level 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 [11, 1] which also runs on a network of workstations. Other open questions have to do with parallelization strategies for the more general non set covering constraints, which is not a problem with the active set strategy, but more difficult for the variable based parallelizations.

References

1. P. Alefragis, C. Goumopoulos, E. Housos, P. Sanders, T. Takkula, and D. Wedelin. Parallel crew scheduling in PAROS. In *EUROPAR'98*, Lecture Notes in Computer Science, 1998. to appear.
2. E. Andersson, E. Housos, N. Kohl, and D. Wedelin. *OR in the Airline Industry*, chapter Crew Pairing Optimization. Kluwer Academic Publishers, Boston, London, Dordrecht, 1997.
3. C. Barnhart and R. G. Shenoi. An alternate model and solution approach for the long-haul crew pairing problem. Jul 1996.
4. A. Caprara, M. Fischetti, and P. Toth. A heuristic algorithm for the set covering problem. In *Lecture Notes in Computer Science*, pages 72–84, 1996.
5. The Carmen System, version 5.1. Carmen Systems AB, Göteborg, Sweden.
6. S. Ceria, P. Nobili, and A. Sassano. A Lagrangian-based heuristic for large-scale set covering problems. Technical report, Dipartimento di Informatica e Sistemistica, Università di Roma, La Sapienza, Italy, 1995.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.
8. 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.
9. M. L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, 1981.
10. C. Goumopoulos, P. Alefragis, and E. Housos. Parallel algorithms for airline crew planning on networks of workstations. In *International Conference on Parallel Processing*, Minneapolis, 1998.
11. C. Goumopoulos, E. Housos, and O. Liljenzin. Parallel crew scheduling on workstation networks using PVM. In *EuroPVM-MPI*, number 1332 in LNCS, Cracow, Poland, 1997.
12. K. L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.
13. 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 Operations Research*, 35:45–58, 1988.
14. R. Marsten. RALPH: Crew Planning at Delta Air Lines. *Technical Report. Cutting Edge Optimization*, 1997.
15. J. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
16. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI – the Complete Reference*. MIT Press, 1996.
17. P. H. Vance. *Crew Scheduling, Cutting Stock, and Column Generation: Solving Huge Integer Programs*. PhD thesis, Georgia Institute of Technology, August 1993.
18. 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.
19. A. Wool and T. Grossman. Computational experience with approxima- tion algorithms for the set covering problem. Technical Report CS94-25, Weizmann Institute of Science, Faculty of Mathematical Sciences, Jan. 1, 1994.