A Parallel, Linear Programming-based Heuristic for Large-Scale Set Partitioni...
Jeff T Linderoth; Eva K Lee; Martin W P Savelsbergh
INFORMS Journal on Computing; Summer 2001; 13, 3; ABI/INFORM Global
pg. 191

A Parallel, Linear Programming-based Heuristic for Large-Scale Set Partitioning Problems

Jeff T. Linderoth • Eva K. Lee • Martin W. P. Savelsbergh
Axioma, Inc., 501-F Johnson Ferry Rd., Marietta, Georgia 30075, USA
School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332-0205, USA
School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332-0205, USA
jlinderoth@axiomainc.com • evakylee@isye.gatech.edu • mwps@isye.gatech.edu

We describe a parallel, linear programming and implication-based heuristic for solving set partitioning problems on distributed memory computer architectures. Our implementation is carefully designed to exploit parallelism to greatest advantage in advanced techniques like preprocessing and probing, primal heuristics, and cut generation. A primal-dual subproblem simplex method is used for solving the linear programming relaxation, which breaks the linear programming solution process into natural phases from which we can exploit information to find good solutions on the various processors. Implications from the probing operation are shared among the processors. Combining these techniques allows us to obtain solutions to large and difficult problems in a reasonable amount of computing time.
(*Set Partitioning; Parallel Computing; Linear Programming-based Heuristics*)

1. Introduction

Given $A \in \{0,1\}^{m \times n}$, $c \in \Re^n$, the set partitioning problem (SPP) is

$$\min\{c^T x : Ax = e, x \in \{0,1\}^n\},$$

where e is a vector of ones. A large number of real-life problems, including vehicle routing and airline crew scheduling, can be formulated as SPPs, so the problem has received a good deal of study. See Balas and Padberg (1976) for a survey of some applications and early solution methods.

Among recent solution procedures, Fisher and Kedia (1990) give a dual ascent heuristic for solving SPP, Harche and Thompson (1994) introduce a column subtraction algorithm, Wedelin (1995) presents a Lagrangian dual approach, Müller (1998) presents a constraint programming approach, and Chu and Beasley (1998) discuss a genetic algorithm. Marsten and Shepardson (1981) discuss a branch-and-bound approach, and Hoffman and Padberg (1993) give a branch-and-cut approach that is very successful in solving airline crew scheduling SPPs. Borndörfer (1997) gives a branch-and-cut approach to solving SPPs arising from scheduling buses for the disabled. A heuristic combining ideas of Hoffman and Padberg and Wedelin is given by Atamtürk et al. (1995).

Due to the wide practical applicability of the SPP, solving large, difficult SPPs in a short amount of time is important. One way to reduce the computation times associated with a given algorithm is to adapt the algorithm to a parallel computing architecture. For a number of difficult optimization problems,

0899-1499/01/1303/0191$5.00
1526-5528 electronic ISSN

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parallel algorithms have been implemented and been shown to be effective (Eckstein 1994; Pardalos and Li 1990). A number of authors have recently employed parallel processing in their solution approaches to the SPP. Levine (1994) parallelizes a genetic algorithm approach, Klabjan et al. (2000b) use parallel processing within a number of their algorithm's components to solve crew scheduling SPPs, and Esö (1999) uses the SYMPHONY (Ralphs and Ladányi 2000) framework (formerly called COMPSys) to develop a parallel branch-and-cut algorithm for the SPP.

The goal of this paper is to show how to exploit parallelism in a linear programming-based solution procedure for solving the SPP. The procedure is very similar to that presented by Atamtürk et al. (1995). In particular, both of the procedures are heuristics. Thus, our goal is not to find optimal solutions to the SPP, but rather to find provably good feasible solutions in a reasonable amount of time, and to exploit parallelism to help us reach this objective.

The paper is organized as follows. In Section 2, we discuss the techniques that make up our solution procedure. In Section 3, we give a scheme for parallelizing the overall solution procedure and give the framework of our implementation. In Section 4, we discuss the implementation of our parallel solution scheme. Section 5 gives computational results on a wide variety of problem instances.

2. The Sequential Algorithm

The linear programming-based SPP heuristic consists of a number of different basic components. The order in which the components are executed is indicated by the flowchart in Figure 1. In this section, we outline how each of the sequential components is performed. The manner in which parallelism is exploited in executing the various components and more detailed algorithmic control issues are discussed in Section 3.

2.1. Preprocessing and Probing

A number of authors (in particular Borndörfer (1997) and Esö (1999)) suggest simple methods for identifying variables that may be fixed and rows that

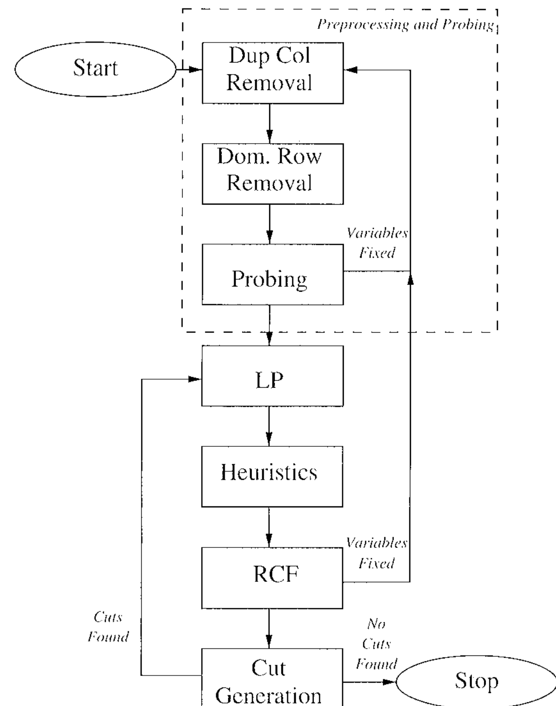

Figure 1 **Set Partitioning Heuristic**

may be removed from the problem. We perform three preprocessing methods:

Duplicate Column Removal. In many applications, the columns of A are not unique. If duplicate columns exist, obviously we need to keep only the one with minimum cost. Because the number of columns is typically quite large, performing a pairwise comparison of columns in order to find duplicates is inefficient. Instead, a random hash function h is used (Knuth 1998). If the hash values for two columns are the same, then a pairwise comparison is done to determine if they are indeed duplicates.

Dominant Row Removal. For each row $i = 1, \ldots, m$, let $T(i) = \{j : a_{ij} = 1\}$. If $T(i) \subseteq T(j)$ for two rows $i \neq j$, we say that row i *dominates* row j. If row i dominates row j, then row j is redundant to the formulation and can be removed. In addition, we may set $x_k = 0 \ \forall k \in T(j) \setminus T(i)$. In order to determine dominant rows, a pairwise comparison is performed. Variables appearing in a row are stored in increasing order of their indices so that non-dominance can be detected as early as possible when comparing two rows.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Probing. Probing techniques are based on the investigation of logical consequences. In the context of SPP, probing is performed by tentatively setting a variable to one and observing the logical implications. See Savelsbergh (1994) for an in depth discussion of probing. Many implications not immediately evident from the constraint matrix may be deduced in this way—in fact, probing may lead to a logical contradiction resulting in the fixing of variables.

Probing is one of the main techniques used in constraint programming (Tsang 1993), which is an effective method for solving tightly constrained scheduling problems. Because obtaining feasibility of an instance of the SPP can be difficult, we may conjecture that probing would be effective for the SPP as well.

2.2. Solving the Linear Program

In order to solve the linear programming relaxation of the SPP, Hu and Johnson (1999) describe a technique called the *primal-dual subproblem simplex method.* For problems with few rows and many columns (as is the case for our instances of the SPP), they show the primal-dual subproblem simplex method to be more effective than the standard simplex method and the primal subproblem simplex method (or SPRINT approach) developed by John Forrest and described by Anbil et al. (1992). We now briefly discuss the primal-dual subproblem simplex method and its application to the set partitioning problem.

Consider the linear programming relaxation (P) to SPP and its dual (D).

$$\text{(P): } \min\ c^T x \quad \text{(s.t.) } Ax = e,\ x \geq 0 \qquad\qquad \text{(D): } \max\ e^T \pi \quad \text{(s.t.) } \pi^T A \leq c$$

In the primal-dual subproblem simplex method, a sequence of subproblems is solved where only a subset of the columns is considered. Let $K \subseteq \{1, 2, \ldots n\}$ be the index set of columns considered in a subproblem, and let A_K, c_K, and x_K be the restrictions of A, c, and x to K. We also use the notation $\bar{c}^\pi \equiv c - \pi^T A$ to denote the reduced costs with respect to a dual solution π.

A primal optimal solution to a subproblem, extended by adding 0 to the columns not in the subproblem, is a primal feasible solution for (P). However, a dual optimal solution for the subproblem is usually not feasible for (D). If it is feasible for (D), then it is also optimal.

Let $\mathcal{F}(\text{P})$ be the set of feasible solutions of (P) and $\mathcal{F}(\text{D})$ be the set of feasible solutions for (D). Given $x \in \mathcal{F}(\text{P})$, $\pi \in \mathcal{F}(\text{D})$, Algorithm 1 is a basic description of the primal-dual subproblem simplex method. If $x \in \mathcal{F}(\text{P})$ is not known, then a two-phase approach using artificial variables is used. The objective in the first phase is to minimize the sum of the artificial variables. In our applications, $c \geq 0$, so $\pi = 0 \in \mathcal{F}(\text{D})$.

Algorithm 1 The Primal-Dual Subproblem Simplex Method

1. Let $x \in \mathcal{F}(\text{P})$, $\pi \in \mathcal{F}(\text{D})$. The initial set of columns K consists of both $F = \{j : x_j > 0\}$ and the $|K| - |F|$ columns (not in F) with smallest reduced costs $\bar{c}^\pi$.
2. Solve the subproblem $\min\{c_K^T x_K : A_K x_K = e,\ x \geq 0\}$. Call the corresponding dual solution ρ.
3. If $\rho \in \mathcal{F}(\text{D})$, or $\pi^T e = c_K^T x_K$ then stop. The optimal solution to (P) is x_K.
4. Let $\pi' = \theta\pi + (1 - \theta)\rho$, for $0 \leq \theta \leq 1$ such that $\pi'^T A \leq c$, and $\pi'^T e$ is maximized. In closed form,
$$\theta = \max_{j:\bar{c}_j^\rho < 0} \left\{ 0, \frac{-\bar{c}_j^\rho}{(\bar{c}_j^\pi - \bar{c}_j^\rho)} \right\}.$$
5. Construct a new set of columns K' consisting of the basic columns of x_K and the $|K| - m$ (nonbasic) columns with the smallest reduced costs $\bar{c}^{\pi'}$. Let $\pi = \pi'$, $K = K'$, and go to 2.

Hu and Johnson (1999) discuss a number of techniques to improve the performance of the algorithm. One algorithmic suggestion not made is for an appropriate size of $|K|$. We have found that a subproblem size of
$$|K| = \min(\lfloor 2m + 0.025n \rfloor, 2000)$$
works well on a large majority of set partitioning instances.

2.3. Primal Heuristics

Obtaining provably good feasible integer solutions quickly is the goal of our parallel set partitioning

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

heuristic. For this purpose, three different heuristics for the set partitioning problem are included as components.

Heuristic I. The first heuristic is a slight modification of a dual-based heuristic due to Fisher and Kedia (1990). The heuristic of Fisher and Kedia attempts to improve a given dual feasible solution to the LP relaxation of the SPP by adjustments involving exactly three dual variables. Our 3-opt procedure has been randomized so that 3-exchanges that do *not* improve the dual objective value are also performed with some probability.

Given a dual feasible solution obtained from the randomized 3-opt procedure, a primal feasible solution is computed by a simple greedy procedure. The choice of variables to be one is made in a greedy fashion as suggested by Chvátal (1979) for the set covering problem, with the modification that the variables are ordered by their reduced costs instead of by their cost coefficients. Note that (as we would expect), there is no guarantee of obtaining a feasible solution in this manner.

Heuristic II. The second heuristic is the Lagrangian dual cost perturbation heuristic of Wedelin (1995). A Lagrangian relaxation of the SPP is obtained by moving the equality constraints to the objective:

$$L(\pi) = e^T\pi + \min_{x\in\{0,1\}} (c - \pi^T A)x.$$

The Lagrangian dual is

$$\max_{\pi\in\Re^m} L(\pi).$$

For a reduced cost vector $\bar{c}^\pi = c - \pi^T A$, if the Lagrangian dual has an optimal solution $(\hat{x}, \hat{\pi})$ such that $\hat{x}$ is feasible to SPP and $\bar{c}^\pi_j < 0$ or $\bar{c}^\pi_j > 0$ for all j, then $\hat{x}$ is an optimal solution to SPP.

In Wedelin's heuristic, the Lagrangian dual is solved by a coordinate ascent method (see Nocedal and Wright (1999) for an explanation of this method). Let A_i denote the ith row of A, and let r^- and r^+ be the smallest and second-smallest reduced costs of variables with coefficient 1 in A_i. It can be shown that the Lagrangian is maximized in the ith coordinate direction by moving an amount $\theta^* = (r^+ + r^-)/2$. Hence, if $T(i)$ is the set of variables appearing in row i, we update $\bar{c}^\pi_j$ for all $j \in T(i)$ by $\bar{c}^\pi_j := \bar{c}^\pi_j - \theta^*$.

The solution of the Lagrangian dual by this method is unlikely to yield solutions that have all $\bar{c}^\pi_j \neq 0$. Wedelin proposes a scheme for perturbing the vector c in such a way that when the Lagrangian dual is solved, the resulting reduced costs satisfy either $\bar{c}^\pi_j < 0$ or $\bar{c}^\pi_j > 0$ for all j. Specifically, the reduced costs are updated as

$$\bar{c}^\pi_j := \begin{cases} \bar{c}^\pi_j - \theta^* - \frac{\kappa(r^+ - r^-)}{1-\kappa} - \delta, & \text{if } \bar{c}^\pi_j \leq r^-, \\ \bar{c}^\pi_j - \theta^* - \frac{\kappa(r^+ - r^-)}{1-\kappa} + \delta, & \text{if } \bar{c}^\pi_j \geq r^+, \end{cases} \tag{1}$$

where $\kappa \in [0, 1]$ and δ is a small positive constant.

The solution quality depends highly on the parameter κ, and it is difficult to know beforehand what a suitable value of κ might be. Wedelin (1995) discusses the algorithm in full detail and proposes an adaptive "sweep" strategy for choosing appropriate values for the parameters κ and δ. We will discuss our implementation of such a strategy in Section 4.2.2.

Heuristic III. The final heuristic is a primal heuristic and is based on the observation that the linear programming relaxations of small-sized set partitioning problems often have integer solutions or yield integer solutions with relatively little branching in a branch-and-bound procedure. Ryan and Falkner (1988) give some theoretical evidence to support this empirical observation. Our heuristic is to choose a "suitable" subset of the columns of A and to solve the integer program (with an imposed time limit) over these columns. The nodes are searched in a depth-first fashion during the heuristic. Of course, choosing the columns is the most difficult part of this heuristic. Our strategy for performing the column choice will be discussed in Section 4.2.3.

2.4. Reduced Cost Fixing

Reduced cost fixing is a technique to fix variables by considering optimality criteria. If the reduced cost $\bar{c}_j$ for variable x_j satisfies

$$z_{LB} + \bar{c}_j \geq z_{UB},$$

where z_{LB} is a lower bound on the solution of the SPP (say from the linear programming relaxation), and z_{UB}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is an upper bound on the solution value of SPP (say from a feasible solution), then variable x_j may be fixed at zero.

When solving the SPP using a branch-and-cut approach, reduced cost fixing is often an extremely powerful tool—allowing as many as 90% of the variables to be fixed. However, obtaining a good feasible solution is critical in allowing a large percentage of variables to be fixed by reduced cost.

2.5. Cutting Planes

The set packing problem (PACK) is closely related to the SPP, and its feasible region can be expressed as

$$\mathcal{F}(\text{PACK}) = \{x \mid Ax \leq e, x \in \{0,1\}^n\}.$$

Since the feasible region of the SPP is contained in the feasible region of the PACK, valid inequalities for the PACK can be used as valid inequalities for the SPP. A number of authors (Padberg 1973; Nemhauser and Trotter Jr. 1974; Cheng and Cunningham 1997) have studied the polyhedral structure of the PACK and derived classes of facet-defining inequalities for it.

An important concept for generating these inequalities is that of the conflict graph CG, where $V(CG) = \{1, 2, \ldots, n\}$ and

$$E(CG) = \{(i,j) \in V : \text{ both } x_i \text{ and } x_j \text{ cannot be one in a feasible solution to the SPP}\}.$$

A discussion of conflict graphs in a general context is given by Atamtürk et al. (2000). For the SPP, many edges $e \in E(CG)$ can be found by direct inspection of the matrix A. In particular, if columns A_j and A_k are not orthogonal, then $(j,k) \in E(CG)$. Edges $e \in E(CG)$ are also found as implications during probing.

In our algorithm, we use two classes of valid inequalities that have been shown to be effective in improving the linear programming relaxation of the SPP. The first class consists of the set of *clique inequalities*. Recall that a clique C in a graph is a set of nodes such that each pair of nodes is connected by an edge. If $C \subseteq V(CG)$ is a clique in the conflict graph, then the clique inequality

$$\sum_{j \in C} x_j \leq 1$$

is a valid inequality for the PACK (and hence for the SPP). The separation problem for clique inequalities is known to be NP-Complete. Therefore, we use a simple greedy heuristic in order to identify violated clique inequalities. The heuristic is initialized with a clique $\widehat{C}$ of size 1 (a vertex $v \in V(CG)$) and incrementally adds vertices $v \in V(CG) \setminus \widehat{C}$ with the largest values of x_v such that the resulting subgraph is also a clique. The procedure is repeated until the clique cannot be made any larger.

The second class of inequalities used in our algorithm consists of the set of *odd-cycle* inequalities. If $H \subseteq V(CG)$ is a cycle in the conflict graph and $|H|$ is odd, then the odd-cycle inequality

$$\sum_{j \in H} x_j \leq \frac{|H| - 1}{2}$$

is valid for the PACK and the SPP.

The separation problem for odd-cycle inequalities is known to be polynomially solvable (Grötschel et al. 1988). However, as pointed out by Hoffman and Padberg (1993), the exact separation procedure is often computationally unsatisfactory. Thus, in order to identify violated odd-cycle inequalities, we use a slight modification of the enumerative procedure described by Bixby and Lee (1994).

For the SPP, the size of the conflict graph can be very large, which may cause the procedures for finding violated inequalities to be very time consuming. We improve the efficiency of the separation procedures by considering only the subgraph CG_F of CG induced by the fractional variables in a solution to the linear programming relaxation. Specifically, given a fractional linear programming solution $\hat{x}$, we construct CG_F with vertex set $V(CG_F) = \{j : 0 < \hat{x}_j < 1\}$ and edge set

$$E(CG_F) = \{(j,k) \in E(CG) : j \in V(CG_F) \cap k \in V(CG_F)\}$$

and look for violated inequalities in CG_F.

After a violated clique inequality is found in CG_F, it is lifted to a stronger inequality by a simple exact procedure. For a clique inequality with corresponding clique C, the lifting coefficient a_j for $j \notin F$ is given by

$$a_j = \begin{cases} 1, & \text{if } (j,k) \in E(CG)\ \forall k : a_k = 1. \\ 0, & \text{otherwise.} \end{cases}$$

Note that a_k could be one because $k \in C$ or because the variable with index $k \neq F$ has already been lifted.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In practice, even this simple procedure can be very time-consuming, so we decided to improve coefficients (or lift) only a fixed fraction of the variables. The variables to lift are chosen in order of increasing reduced cost.

The odd-cycle inequalities can also be lifted in order to obtain stronger inequalities; however, in this algorithm, lifting of odd-cycle inequalities is not performed. The interested reader is referred Nemhauser and Sigismondi (1992), which describes a fast algorithm for generating all distinct lifted odd-cycle inequalities given a violated (unlifted) odd-cycle inequality.

3. Parallelizing the Sequential Algorithm

In this section, we discuss the approach we have taken to parallelize the solution procedure described in the previous section. The parallel architecture on which we have chosen to focus is the *message passing* or *distributed memory* architecture. As the name suggests, on a distributed memory parallel machine, each processor comprising the machine has a distinct memory address space. In order to share information among the processors, a message must be passed, for which some overhead is incurred.

Creating a parallel solution approach by breaking a problem into slices that can be performed simultaneously is not always a trivial matter. We have used both *domain decomposition*, in which the data are divided among the processors, and *control decomposition* or *functional decomposition*, in which the algorithmic tasks themselves are divided among the processors. See Foster (1995) for more explanation of these parallel programming methods.

In the domain decomposition approach to problem partitioning, we partition the computation that is to be performed by associating each operation with the data on which it operates. This partitioning yields a number of tasks, each composed of some data and a set of operations on these data. An operation may require data from several tasks. In this case, communication is required to move data between tasks and to synchronize the operations.

In functional decomposition, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The goal is to divide the computation into disjoint tasks that can be executed in parallel. If the data requirements of the divided tasks are also disjoint, then we have successfully parallelized the program. Usually there is at least some overlap in the data requirements of the tasks, in which case communication will be needed.

The algorithm presented in Section 2 has seven main computing tasks:

- duplicate column removal,
- dominant row identification,
- solving linear programs,
- performing heuristics,
- reduced cost fixing,
- probing, and
- cut generation.

Some of the individual tasks are parallelized using domain decomposition, and some of the tasks are parallelized by using functional decomposition—either by performing multiple tasks simultaneously or by performing multiple copies of the same task. Conceptually, the algorithm contains a number of worker processes that perform the algorithmic components and a controlling process that oversees the synchronization and overall flow of the algorithm. We mention this here in order to aid the discussion of the parallelization of the algorithm. A more detailed discussion of the parallel implementation is described in Section 4. Below, we first provide the details of how each of the computing tasks is parallelized.

3.1. Duplicate Column Removal

The task of duplicate column removal amounts to computing hash values for all columns, finding duplicate hash values, and verifying whether or not duplicate hash values indeed correspond to duplicate columns.

To compute the hash values for the columns and verify duplicate columns, a domain decomposition approach to parallelization is taken. A random hash function over the rows (as described in Section 2.1) is computed and passed to the worker processors at the time of their initialization. When duplicate columns are to be identified, the controller determines a begin-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ning and ending index b_i and e_i for each worker process W_i. The indices b_i and e_i are chosen so as to divide the number of non-zeroes in A evenly among the different processes W_i.

The workers receive the indices b_i and e_i and determine hash values $h(A_j), \forall\, b_i \le j \le e_i$. The hash values are passed back to the controller process, which collects the values and determines the duplicates. The work of verifying that duplicate hash values are really duplicate columns is also divided among the worker processors in a manner similar to that of computing the hash values. Once the controller receives the information about which columns are actually duplicates, the information is broadcast to the worker processors, so that the representation of the problem remains current on all of the processors.

Ignoring overhead, if t is the number of non-zeroes of A, and we have p worker processes, we would expect to take time $O(t/p + n\log n)$ in order to identify potential duplicate columns. This complexity could be reduced by performing the sorting operation needed to find duplicate hash values in parallel. Duplicate column identification takes up a very small percentage of the overall time of the algorithm, so we assume that the complexity of implementing a parallel sorting algorithm outweighs the potential benefits.

3.2. Dominant Row Identification

Dominant row identification is another natural task to be parallelized using domain decomposition. The rows must be checked pairwise, and we would like to divide the number of comparisons to be made evenly among the p worker processors. We say a row index $k < m$ is *validated* if row A_k is checked for dominance against each row A_j, $m \ge j > k$. For each worker process W_i, we need to determine a set of rows R_i for W_i to validate such that the number of row comparisons is roughly equal for each processor. We choose the sets R_i as $R_1 = \{1, 2, \ldots, e_1\}$, $R_2 = \{e_1+1, e_1+2, \ldots, e_2\}, \ldots R_p = \{e_{p-1}+1, e_{p-2}+2, \ldots, m\}$. In order to validate the set of rows in R_i, worker W_i must perform $\sum_{k=e_{i-1}+1}^{e_i}(m-k)$ row comparisons. The total number of row comparisons is $m(m-1)/2$, so each worker should perform $m(m-1)/2p$ comparisons to distribute the work equally. Thus, e_1 should satisfy

$$\sum_{k=1}^{e_1}(m-k) = \frac{m(m-1)}{2p}.$$

Solving this equation yields

$$e_1 = \left\lfloor m - \frac{1}{2} - \sqrt{\left(1-\frac{1}{p}\right)m^2 - \left(1-\frac{1}{p}\right)m + \frac{1}{4}} \right\rfloor.$$

Recursively, and in a similar fashion, we can show that for $1 \le j \le p-2$,

$$e_{j+1} = \left\lfloor m - \frac{1}{2} - \sqrt{\left(1-\frac{1}{p}\right)m^2 - \left(1+2e_j-\frac{1}{p}\right)m + \frac{1}{4} + \frac{e_j(e_j+1)}{2}} \right\rfloor.$$

This simple scheme should prove sufficient to divide the work evenly among the processors. The work of determining the variables to be fixed is left to the controller process, which broadcasts this information to the worker processes in order to keep the problem description consistent.

The dominance relation between rows is transitive. Hence, sharing information about dominant rows among the processors could lead to a reduction in the number of pairs of rows checked for dominance. Dominant row identification is performed only on the original rows of A and is not a very computationally intensive task, so the overhead required to share dominance information likely outweighs the benefit. We do not share dominant row information among the processors.

Note that removing dominant rows can lead to more duplicate columns. Therefore, the simple and fast operations of duplicate column removal and row dominance testing are iterated upon until no more problem reductions can be done.

3.3. Solving the Linear Program and Finding Feasible Solutions

Once the initial linear programming relaxation of the SPP is to be solved, there is an opportunity to decompose the computing task using functional decomposition and to reduce the amount of synchronization required by the algorithm. The primal-dual subproblem simplex method is performed on one processor, while the remaining processors search for a feasi-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ble solution using one of the heuristics described in Section 2.3.

There are two motivations for breaking the problem up in this way. The first is that the primal-dual subproblem algorithm is not especially suited for parallelism, so it makes sense to use the remaining processors to do the useful work of searching for feasible solutions. (Klabjan et al. (2000a) discuss a parallelization of the primal-dual subproblem simplex method that achieves small speedups.) Second, at each iteration of the primal-dual subproblem simplex method, we obtain information that can be used to help guide the heuristics. We now describe our functional decomposition approach in more detail.

The linear program is solved by the controlling process, and the worker processes perform the heuristics. To begin, each of the worker processes is passed a dual feasible solution and performs the randomized dual based Heuristic I of Section 2.3. The solution of the linear program itself is naturally broken into two phases. In the first phase, the primal solution x from the subproblem is not feasible to the linear programming relaxation, and in the second phase x is LP-feasible. After each iteration of the first phase, the feasible dual solution ρ is passed to the workers, and the workers use this new feasible solution as a starting point for the dual-based heuristic. After each iteration of the second phase, the set of columns K from the iteration is passed to one of the workers who is performing the dual-based heuristic. This worker stops the dual-based heuristic and performs the primal-based Heuristic III of Section 2.3 by solving the integer program over the set of columns K.

All feasible solutions found by the heuristics are passed to the controlling process, which broadcasts new solution values to the worker processes. The primal-based Heuristic III is speeded up dramatically by only searching for a solution better than a given value.

If a worker finishes Heuristic III before the solution of the initial LP is complete, it returns to performing the dual-based Heuristic I. The dual feasible solution ρ found at each iteration of the primal-dual subproblem simplex method is stored for use as a starting point to Heuristic II. The use of Heuristic II will be discussed in more detail in Section 4.2.1. Only the initial linear programming relaxation is solved by the primal-dual subproblem simplex method. All other linear programs are solved using the dual simplex method.

3.4. Reduced Cost Fixing

Determining the variables that can be fixed based on reduced costs is very similar to computing the hash values used in determining duplicate columns. However, in order to perform reduced cost fixing in parallel, the vector of dual variables from the linear programming solution would need to be broadcast to the worker processes each time reduced cost fixing was to be performed. We therefore presume that the communication costs required from performing the reduced cost fixing in parallel outweighs the benefit. The controller process determines which variables can be fixed based on the their reduced costs and then broadcasts this information to the worker processes.

3.5. Probing

Probing is another natural task to parallelize using a domain decomposition approach. The amount of work required to probe on a variable is related to the number of implications found, and determining this information before actually performing the probing is difficult. Therefore, we assume that probing on every variable requires the same amount of work. Thus to divide the work of probing evenly among the processors, each processor is given an equal number of variables on which to probe.

An important difference between probing and the other tasks we have parallelized by domain decomposition is that the implications found at one processor during probing can be quite useful to the probing operation at other processors. By sharing implication information among the processors, we may be able to reduce the number of rounds required to produce all implication information. Alternatively, if we are only performing one round of probing (we probe on each variable exactly once), then the positive effects of sharing implications would manifest themselves as finding more implications and fixing more variables

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in parallel than if the round of probing were done sequentially.

During probing, implications are found very often and constitute a small amount of information. An efficient means to share this type of information is to use a *buffered-prefetch* approach. Implications found by one processor are stored in a buffer. When the implication buffer is full, then the implications are broadcast to the other worker processes who incorporate them in their local conflict graphs.

In addition to the synergetic effects of sharing implications, we obtain the normal speedup effects from decomposing the work of the problem. Since probing is generally the most time-consuming task in the sequential algorithm, we expect that the time saved by doing probing in parallel will generally be quite significant.

3.6. Cut Generation

Of all the solution techniques mentioned so far, cut generation seems the least natural to decompose. Fortunately, our job of decomposing cut generation is made easier by the separation heuristics used to find violated inequalities. Both the heuristic used for clique inequality separation and the heuristic used for odd-cycle inequality separation start with a given node $v \in V(CG_F)$ and perform a limited search for violated inequalities starting from this vertex. Hence, we may use a domain decomposition approach to finding violated inequalities by dividing the nodes from which the heuristics start their search for violated inequalities. Each processor gets an equal number of vertices of $V(CG_F)$ from which to begin a search for violated inequalities.

Note that the same violated inequality may be found multiple times. A multiplicative hash function (Knuth 1998) is used to identify duplicate cuts quickly.

4. PSPS—A Parallel Set Partitioning Solver

In this section, we describe PSPS—a parallel set partitioning solver based on the ideas presented in the preceding sections. First, the software infrastructure on which PSPS is built is described, and then we will discuss details of algorithm implementation within PSPS.

4.1. Software Infrastructure

PSPS uses a software system called NAYLAK for message passing that is based on an "entity-FSM" paradigm (Perumalla and Schwan 1996). In NAYLAK, each process consists of a set of entities, and message passing is performed at the entity level rather than at the process level. Also supported in NAYLAK is the concept of a finite state machine (FSM). An FSM is an arbitrary graph of *states*, where each state is a set of statements that are executed indivisibly (atomically). Every FSM is associated with a single entity, called its owner, and an entity can have zero or more FSMs running for it. An FSM is in effect a thread of computation, and each FSM state is a unit of computation. NAYLAK provides another layer above the message passing library, and the overheads incurred for this extra layer are mostly in extra memory copying instructions for messages. In this work, NAYLAK has been configured to use the PVM (3.3) message passing library (Geist et al. 1994).

There are two types of entities in PSPS—a *controller* entity and *worker* entities. The controller entity is run on one processor, and worker entities are run on the remaining processors. From the discussion in Section 3 of how the heuristic is parallelized, the duties of each entity should be clear. The controller entity has one FSM that is responsible for managing the overall flow of the heuristic and solving the lower bounding linear programs. The worker entity has FSMs to perform the majority of the work of the heuristic: checking duplicate columns, checking dominant rows, performing primal heuristics, probing, and generating cuts.

For our heuristic procedure to be effective, it is quite important that the operations at the worker entities be interruptible. For example, when performing heuristics, once one processor finds a good feasible solution, there is little need to wait for the other heuristics to complete. The actions of the worker entities are performed by FSMs whose states consist of sufficiently fine-grained operations in NAYLAK, so it is a simple matter to interrupt them if a good feasible solution is found.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

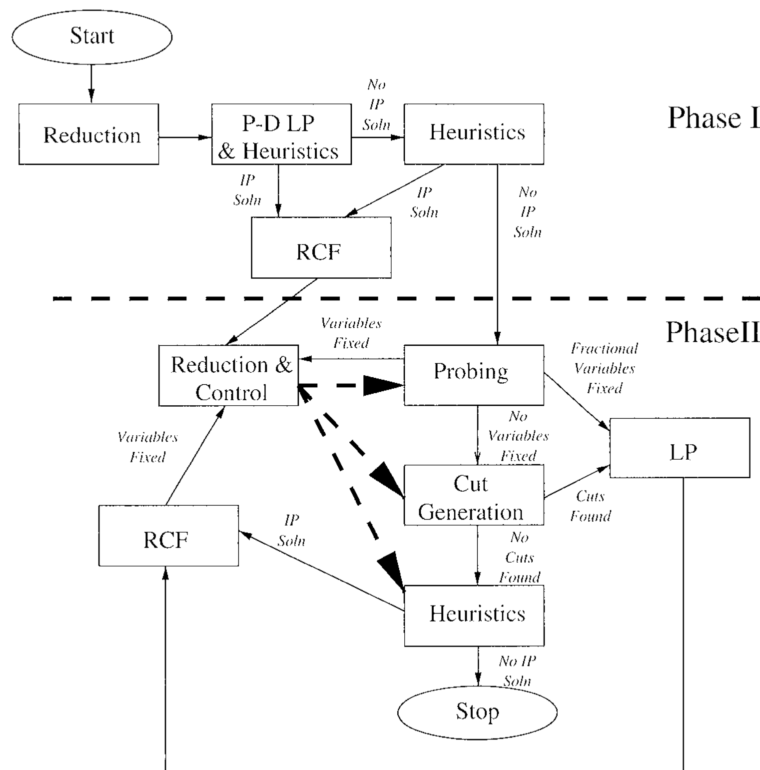

Figure 2 Parallel Set Partitioning Heuristic

4.2. Computational and Control Issues

4.2.1. Algorithm Flow. Now that all the basic components have been described, and strategies for parallelizing the components have been explained, we can state the order in which the components are performed. Figure 2 is a flowchart of our parallel heuristic for solving the SPP.

As mentioned in Section 3.2, the duplicate column removal and row dominance techniques can potentially be iterated upon many times. The *Reduction* box in Figure 2 denotes an entire sequence of column removal and row dominance operations until both of these operations fail to reduce the problem size. The problem reduction techniques are executed in parallel as discussed in Sections 3.1 and 3.2. The *P-D LP & Heuristics* box denotes the simultaneous execution of the primal-dual subproblem simplex method and heuristics as explained in Section 3.3. *RCF* represents reduced cost fixing, and *LP* denotes the dual simplex method.

The solution approach is conceptually broken into two phases. In the first phase, we are interested in obtaining rough upper and lower bounds on the optimal solution by solving the initial linear programming relaxation and by finding feasible solutions. In the second phase, we are interested in refining the upper and lower bounds and in reducing the problem size through the use of probing, cutting planes, and more sophisticated heuristics.

Probing is a time-consuming operation, and we would like to probe on as small a problem as possible. Therefore, before we enter the second phase of our procedure, if we have found no feasible solution (and hence can fix no variables based on their reduced cost), we perform the operations denoted by the *Heuristics* box in Figure 2. In this step, we use parallelism in a functional decomposition manner. Some processors perform the dual-based Heuristic II, and some processors perform the primal-based Heuristic III. Which processors perform which

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

heuristic and the information used to start the heuristics will be discussed in Sections 4.2.2 and 4.2.3.

The main control decision to be made is represented by the three large dashed arrows in Figure 2. In order to help decide which component to perform at this point, two statistics are kept: the percentage of variables fixed since the last time the algorithm performed a round of probing (ζ) and the number of consecutive rounds of cut generation (θ). Figure 3 is a flowchart of the decision-making process of the *Reduction & Control* box in Figure 2. The rationale of the control decision is as follows. During preliminary testing, it was observed that if the number of variables fixed by probing or reduced cost fixing was small, then the further implications gathered by doing additional rounds of probing was also small. Since probing is costly, a round is performed only if significant benefit will be attained (i.e. ζ is large). If probing is not to be performed, then the decision of whether to add cuts or to perform heuristics must be made. The question to answer here is whether the upper or lower bound on the solution value is likely to be improved. Initially, if variables are fixed based on their reduced cost, then it is assumed that the upper bound from the feasible solution is good and we decide to try to improve the lower bound through the addition of cutting planes. If after a number of consecutive rounds of cut generation, we are unable to fix a significant percentage of the variables, we decide to try and improve the upper bound by performing more heuristics.

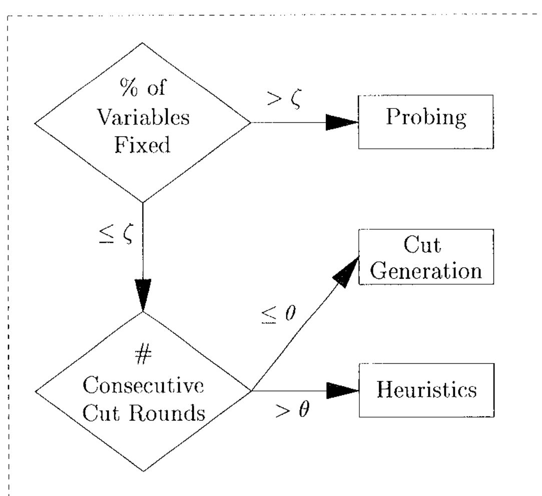

Figure 3 **Main Control Decision**

As mentioned in Section 3.5, parallelism can help speed up the probing operation significantly. Thus, we may wish to probe more often in a parallel algorithm than in a sequential one. The amount of probing can be easily increased by reducing the percentage of variables ζ that must be fixed before another round of probing is performed.

4.2.2. Heuristics. When we come to the point of the procedure denoted by the *Heuristics* box in Figure 2, there is another choice to be made. We must decide the number of each type of heuristic to run.

Computational testing quickly revealed that Heuristic I was a very fast procedure, but rarely yielded good solutions to difficult problems. Therefore, Heuristic I is not performed in the *Heuristics* component of the procedure, only while the initial linear programming relaxation is being solved.

We adopted the following simple adaptive scheme for determining the number of instances of Heuristic II and Heuristic III to run. If the best solution so far was found by Heuristic II, then 2/3 of the worker processes perform Heuristic II and 1/3 perform the primal-based Heuristic III. Conversely, if the best solution so far was found by Heuristic III, then 2/3 of the worker processes perform this heuristic and the remaining 1/3 perform Heuristic II. If no feasible solution exists or if the best solution was found by Heuristic I, then 1/2 of the worker processes perform Heuristic II and 1/2 perform Heuristic III.

The appropriate choice of the parameters κ and δ in Equation (1) to guide Heuristic II is an important and difficult one. In general, the smaller the value of κ, the less likely the algorithm is to find a feasible solution, but the solutions found will be of high quality. We take the suggestion of Wedelin (1995) and perform a quick "sweep" for an appropriate value of κ, where a feasible solution is found, and then a more thorough search for smaller and smaller values of κ. We use values of κ ranging from 0.1 to 0.5. The initial increment of κ is 0.01, but once a solution is found, this increment is reduced. An appropriate value of δ in Equation (1) was found to be far less crucial to the heuristic's effectiveness. A value of $\delta = 0.001$ is used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.3. Choosing Columns. If the procedure reaches the point denoted by the *Heuristics* box in Figure 2, then the sets of columns K from iterations of the primal-dual subproblem simplex method were insufficient for the primal-based heuristic to find a good solution. To increase the chances that the set contains a good integer solution, we increase its size. In this section, we describe our strategy for performing this task.

We have two goals in choosing sets of columns for processors. First, we would like to choose sets of columns that are likely to contain good feasible integer solutions. Second, we would like to ensure that the different processors that are to perform the primal heuristic get sufficiently different sets of columns. We have adopted the following column-choice strategy.

In order to improve the chances that the set of columns we choose will contain a feasible solution, we would like to have each row covered enough. Let $T(i) = \{j : a_{ij} = 1\}$ be the set of columns for which there is a "1" entry in the ith row of A. Define $\Phi(s, T(i))$ to be a set of s randomly chosen columns of $T(i)$, and for a set of columns $\widehat{K}$, let $\tau_{\widehat{K}}(i)$ be the number of "1" entries in the ith row of the matrix created from the columns of $\widehat{K}$. Given a starting row index i_0, a target covering number r, a set of columns $\widehat{K}$, and a target size R, Algorithm 2 is a procedure for increasing the number of columns in $\widehat{K}$ to R.

Algorithm 2 `ColumnChoice` $(i_0, r, \widehat{K}, R)$
0. **Initialize.** $i = i_0$
1. **Increase Set.** If $\tau_{\widehat{K}}(i) < r$, $\widehat{K} = \widehat{K} \cup \Phi(\min(r - \tau_{\widehat{K}}(i), |T(i)|), T(i))$.
2. **Complete?** If $|\widehat{K}| \geq R$, stop. Else $i = i+1$. If $i > m$, $i = 1$. Go to 1.

For each processor that is to perform the primal heuristic, a different value of i_0 is chosen and `ColumnChoice` $(i_0, \alpha n, K, \Lambda|K|)$ is called, where α is the density of the matrix A, K is the set of columns from the last iteration of the primal-dual subproblem simplex method, and Λ is a constant greater than one.

The time spent performing the heuristic phase is limited to a maximum time of T_H. If a primal heuristic procedure initialized with the set of columns K_0 finishes in a time less than T_H and is unsuccessful in finding an improved integer feasible solution, a new primal heuristic procedure is started with a set of columns generated from `ColumnChoice` $(i_0, \alpha n, K_0, \Lambda|K_0|)$.

If the set of columns passed to the primal heuristic is the entire set of columns remaining in the problem, then upon successful completion of the primal heuristic, we can conclude that the solution returned is optimal. If in time T_H none of the heuristic procedures is able to find an improved solution, then the procedure stops.

5. Computational Experiments

5.1. A Test Suite of Problems

The problems on which we have tested PSPS are taken from a variety of real-world applications. The first set of instances is a subset of the famous test suite of crew scheduling problems of Hoffman and Padberg (1993). From their full test suite, we chose all instances that were not solved at the root node by their branch-and-cut procedure. The second set of instances arose in the context of scheduling buses for the disabled in Berlin (Borndörfer 1997). A third set of instances consists of set partitioning formulations of some capacitated vehicle routing instances (Reinelt 1991). A final instance arose in a two-phase approach to solving an inventory routing problem (Campbell et al. 1998).

Table 1 shows some statistics for the instances in our test suite. For each instance, we present the number of rows m, the number of columns n, and the density of the matrix α. Our computational experiments suggest that the number of rows m is a good indicator of an instance's difficulty, at least for our LP based solution approach. The difficulty of an instance seems to have only a weak dependence on the number of columns n. We present two more statistics for each instance. For a row i of $A \in \{0,1\}^{m \times n}$, let τ_i denote the number of "1" entries in row i, and let v_j denote the number of "1" entries in column j. Now let $\bar{\tau}$ be the average of the values of τ_i, and let $\bar{v}$ be the average of the values of v_j. Finally, let the symbol $\sigma(\tau)$ denote the standard deviation of the values $\tau_1, \tau_2, \ldots, \tau_m$ and $\sigma(v)$ denote the standard deviation of $v_1, v_2, \ldots, v_n$. The "coefficients of variation" for the τ_i and v_j, defined respectively as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 1 Characteristics of Set Partitioning Problem Test Instances

Name	m	n	$\alpha(\%)$	$\sigma(\tau)/\bar{\tau}$	$\sigma(v)/\bar{v}$	CPLEX Time	CPLEX Gap (%)
hp1	61	118607	14.0	0.801	0.169	3:01	0
hp2	55	7479	13.7	0.721	0.283	0:41	0
hp3	801	8308	0.99	0.801	0.280	10:44	0
air04	823	8904	1.00	0.649	0.276	19:51	0
air05	426	7195	1.70	0.609	0.263	37:57	0
nw04	36	87842	20.22	0.680	0.174	4:00:00	∞*
t0415	1518	7524	0.44	1.41	0.766	4:00:00	∞*
t0416	1771	9345	0.38	1.35	0.877	4:00:00	∞*
t1716	467	56865	0.94	0.00870	0.445	4:00:00	∞*
v0416	1771	19020	0.17	2.74	0.333	4:00:00	0.20
v1616	1439	67441	0.25	1.69	0.300	4:00:00	0.28
v1617	1619	113655	0.23	1.79	0.407	4:00:00	0.41
eil33	32	4516	30.62	0.473	0.272	4:00:00	18.12
eilA76	75	25175	12.15	0.616	0.227	4:00:00	∞*
eilA101	100	5073	11.04	0.481	0.352	4:00:00	17.64
eilB76	75	19349	9.35	0.760	0.248	4:00:00	0.48
eilC76	75	28599	14.68	0.561	0.213	4:00:00	∞*
eilD76	75	2535	13.33	0.359	0.305	58:44	0
irp1	39	20315	12.40	0.658	0.132	59:48	0

*No feasible solution found.

$\sigma(\tau)/\bar{\tau}$ and $\sigma(v)/\bar{v}$, may give an indication of the difficulty of solving the set partitioning problem instance using linear programming based techniques, because the values $\sigma(\tau)/\bar{\tau}$ and $\sigma(v)/\bar{v}$ have an impact on the number of fractional extreme points of the underlying polyhedron. The smaller the variation, the more likely "highly fractional" extreme points exist, and the more likely the instance is difficult to solve. This intuition is based on a characterization of perfect matrices in terms of forbidden submatrices by Padberg (1974), and a similar observation was made by Ryan and Falkner (1988). We had hoped that these statistics would provide additional insight into what makes an instance difficult, but from our computational experiments it is not clear that they do. As a final indication of the difficulty of an instance, we also show how the mixed integer programming package CPLEX (version 4.0) (CPLEX Optimization 1995) fares in solving it on an IBM RS/6000 Model 390. The default CPLEX settings were used, except for the branching strategy. We choose to use strong branching, because this strategy is known to work well on set partitioning problems. We want to stress that it is not our intention to compare our computational results with CPLEX—we provide this information only to give the reader a feeling for the "difficulty" of each instance in our test suite. In Table 1, the instances above the first line are the crew scheduling instances, above the second line are the bus scheduling instances, above the third line are the vehicle routing instances, and the final instance comes from an inventory routing problem.

5.2. Speedup and Performance

In this section, we demonstrate the effectiveness of PSPS on the instances in our test suite. Implementations of parallel algorithms in which control decisions depend on the order that tasks are completed can exhibit stochastic behavior due to the nondeterministic running time of various tasks. Therefore, experiments on a given instance should be repeated a number of times in order to reduce the effects of this randomness. There is the possibility that PSPS will exhibit this sort of random behavior. For example, implications that are passed between processors during probing can affect the overall outcome of a probing pass. However, computational testing revealed that the variation between runs on the same instance was very small or non-existent. This is hardly surprising, since there is a large synchronous component of our algorithm, and our experiments were run on dedicated processors. Thus, we choose to report results based on only one trial per instance.

The computational environment was a cluster of 16 Quad Pentium Pro 200 MHz servers, each with 256MB RAM linked via fast ethernet to a Cisco 5505 network switch. Only one processor per machine was used, so that all message passing was done over the ethernet. Figure 4 shows the settings of PSPS used in our computational experiments.

Our strategy of running multiple heuristics simultaneously, and also the inherent synchronous components of our parallel algorithm lead us to believe that near-linear speedups will not be attained by our algorithm. This is not the goal of this research. Instead, we hope to exploit parallelism to improve significantly the running time on solvable instances and to find significantly better solutions on unsolvable instances in a given amount of running time. Achieving these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

θ	Maximum consecutive rounds of cut generation	5
ζ	Percentage of variables that must be fixed before beginning another round of probing	5%
Λ	Factor by which to increase the number of columns before beginning heuristic III	1.5
T_H	Maximum time for heuristic III	20 minutes

Figure 4 Settings for PSPS

goals over a wide variety of set partitioning problem instances is difficult.

The PSPS code was able to find the optimal solution and prove it optimal in ten of the nineteen instances. Table 2 shows the time required to solve these eight instances on 2, 4, 8, and 16 processors. In PSPS there must be at least one worker process and one controller process. To compute a rough estimate of the time for the "sequential" (one-processor) version of the PSPS code, the times for the case $p = 2$ can be multiplied by two.

From Table 2, we see that using our parallel approach can be beneficial in solving problems more quickly, but only on instances requiring over 1000 seconds to solve. We also note that our heuristic was able to solve (find the optimal solution and prove its optimality) all the instances CPLEX was able to solve, plus some others.

For instances where parallelism is useful, it is interesting to see in which section of the algorithm taking advantage of parallelism has the greatest effect. To that end, consider Table 3, where the times in various components of the algorithm are reported. We restrict ourselves to instances taking more than 1000 seconds and also report, in parentheses, the number of times the probing and cutting plane phases of the algorithm were performed for each instance.

Table 3 clearly demonstrates the benefit of performing probing in parallel. It also shows that in order to see benefits from performing simple preprocessing operations in parallel, larger instances must be considered. In some cases, the amount of time spent performing heuristics is also reduced because a good solution is found more quickly.

Table 3 also displays several instances where the separation phase does not parallelize well. There are two reasons for this. First, our simple strategy of parallelizing the separation heuristics by giving different starting points to different processors often leads to finding duplicate cuts, implying that duplicate work has been done. Second, our separation strategy was to separate first for clique inequalities at a processor, and only if no clique inequalities were found to separate for odd-cycle inequalities. With each processor getting fewer starting points to look for clique inequalities, it is often the case that many times the (often time-consuming) odd-cycle separation phase is entered. This, coupled with the fact that we wait for all processors to perform their separation heuristics from all starting points before resolving the linear program can lead to an increase in the amount of time to perform a separation phase. We tested alternative parallel-separation strategies and found that despite not parallelizing as well, the default strategy used was the most computationally effective.

Table 4 shows the performance of PSPS on instances for which it cannot prove the optimality of the best solution found in one hour of computing time. The time at which PSPS finds its best solution is reported in the column *Time to Best*. The column heading *Best Known Gap* refers to the percentage gap between the solution found by our procedure z_{HEUR} and the best known solution to the problem z_{BEST}, i.e., $100(z_{BEST} - z_{HEUR})/z_{HEUR}$. A value less than zero implies that our solution procedure improved on the best known solution to the problem. The column heading *Provable Gap* gives the value $100(z_{HEUR} - z_{LP})/z_{HEUR}$, where z_{LP} is the solution to the final linear programming relaxation in the our SPP heuristic. The column *Best Heuristic* shows which heuristic found the best solution for each instance, and the column

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 2 PSP Performance on Solved Instances

Name	p	Time (sec.)
hp1	2	81
hp1	4	59
hp1	8	59
hp1	16	66
hp2	2	10
hp2	4	28
hp2	8	9
hp2	16	14
hp3	2	125
hp3	4	128
hp3	8	123
hp3	16	136
air04	2	1767
air04	4	1369
air04	8	1452
air04	16	1179
air05	2	2550
air05	4	888
air05	8	792
air05	16	659
nw04	2	1118
nw04	4	466
nw04	8	433
nw04	16	343
eil33	2	976
eil33	4	723
eil33	8	638
eil33	16	635
eilB76	2	1783
eilB76	4	1352
eilB76	8	1179
eilB76	16	770
eilD76	2	2304
eilD76	4	993
eilD76	8	987
eilD76	16	984
irp1	2	1416
irp1	4	599
irp1	8	447
irp1	16	357

Table 3 Amount of Time in Solution Phases

Name	p	Time (sec.)	Init Time	Prep. Time	Probe Time	Cut Time	LP Time	Heur Time
air04	2	1767	6	6	33 (4)	81 (11)	256	1385
air04	4	1369	10	3	13 (4)	76 (11)	251	1016
air04	8	1452	13	2	12 (5)	28 (9)	281	1116
air04	16	1179	9	2	9 (4)	97 (9)	265	797
air05	2	2550	2	1	20 (3)	35 (6)	18	2474
air05	4	888	4	1	8 (3)	29 (6)	18	828
air05	8	792	3	1	6 (3)	28 (6)	18	736
air05	16	659	4	1	6 (3)	39 (6)	22	587
nw04	2	1118	29	2	894 (7)	94 (17)	8	91
nw04	4	466	30	2	315 (5)	24 (5)	6	89
nw04	8	433	35	3	261 (6)	33 (14)	8	93
nw04	16	343	32	3	130 (7)	67 (17)	10	101
eil33	2	976	1	0	258 (6)	44 (15)	8	665
eil33	4	723	2	0	128 (6)	44 (15)	10	539
eil33	8	638	5	0	74 (6)	36 (13)	10	513
eil33	16	635	7	1	38 (6)	40 (15)	10	539
eilB76	2	1783	4	2	696 (14)	79 (21)	12	990
eilB76	4	1352	6	2	219 (17)	107 (34)	18	1000
eilB76	8	1179	6	2	107 (15)	38 (24)	10	1016
eilB76	16	770	10	2	71 (13)	41 (17)	9	637
eilD76	2	2304	1	0	6 (1)	27 (5)	6	2264
eilD76	4	993	1	0	0 (0)	0 (0)	1	991
eilD76	8	987	1	0	0 (0)	0 (0)	1	985
eilD76	16	984	3	0	0 (0)	0 (0)	1	980
irp1	2	1416	5	2	1203(12)	151 (25)	16	39
irp1	4	599	6	1	459 (14)	66 (24)	15	52
irp1	8	447	8	1	345 (10)	55 (18)	13	25
irp1	16	357	7	2	218 (11)	78 (18)	18	34

Heuristic Solutions shows the number of different solutions found by each heuristic: (I, II, III).

Many events are of note in Table 4. In some instances, the best solution is found very close to the one-hour mark, even with 16 processors. This implies that a "sequential version" of our algorithm that switched between different heuristics would take close to 16 hours to find a solution of equal quality. For the instances t0415 and t0416, PSPS was able to improve on the best known solutions, but all of the heuristics fail to find a solution. An integer feasible solution was obtained as a primal feasible iterate x in the solution of the initial LP. Finally, it is apparent that the different set partitioning instances are amenable to different heuristics, as evidenced by the fact that each heuristic type in some case found the best solution.

From the results of our performance experiments, we conclude that parallelism is often helpful in solving set partitioning problems in a reasonable amount of time. The main benefits from parallelism come from the ability to probe the constraint matrix more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4 PSP Performance on Unsolved Instances

Name	p	Time (sec.)	Time to Best (sec.)	Best Known Gap (%)	Provable Gap (%)	Best Heur.	Heur. Solutions
eilA101	2	3600	3	58.1	62.8	I	(2,0,0)
eilA101	4	3600	855	2.08	11.4	III	(2,2,1)
eilA101	8	3600	1074	2.08	11.9	III	(4,1,1)
eilA101	16	3600	775	2.08	10.4	III	(4,1,1)
eilA76	2	3600	12	17	18.5	I	(2,0,0)
eilA76	4	3600	1033	16.3	17.7	III	(2,2,2)
eilA76	8	3600	3438	13	14.4	III	(4,2,4)
eilA76	16	3600	3474	6.38	7.48	III	(3,2,10)
eilC76	2	3600	12	30.5	33.9	I	(3,0,0)
eilC76	4	3600	59	16.4	18.9	II	(6,3,2)
eilC76	8	3600	45	16.3	18.7	II	(5,3,2)
eilC76	16	3600	38	16.3	18.7	II	(7,1,2)
t0415	2	3600	263	−0.000107	8.27	LP	(0,0,0)
t0415	4	3600	263	−0.000107	8.27	LP	(0,0,0)
t0415	8	3600	264	−0.000107	8.27	LP	(0,0,0)
t0415	16	3600	263	−0.000107	8.27	LP	(0,0,0)
t0416	2	3600	261	−0.2	4.69	LP	(0,0,0)
t0416	4	3600	262	−0.2	4.69	LP	(0,0,0)
t0416	8	3600	264	−0.2	4.69	LP	(0,0,0)
t0416	16	3600	265	−0.2	4.69	LP	(0,0,0)
t1716	2	3600	–	∞	∞	–	–
t1716	4	3600	2118	81.4	86	II	(0,3,0)
t1716	8	3600	1783	81.4	86	II	(0,3,0)
t1716	16	3600	1580	81.4	86	II	(0,2,0)
v0416	2	3600	9	7.00	6.56	I	(3,0,0)
v0416	4	3600	10	3.05	2.98	I	(11,0,0)
v0416	8	3600	3579	0.154	0.183	III	(25,0,9)
v0416	16	3600	3577	0.109	0.137	III	(37,0,25)
v1616	2	3600	653	0.988	0.995	III	(2,0,1)
v1616	4	3600	3321	0.594	0.607	III	(38,0,3)
v1616	8	3600	749	0.274	0.289	III	(43,0,3)
v1616	16	3600	2046	0.232	0.242	III	(41,0,4)
v1617	2	3600	36	9.50	10.9	I	(1,0,0)
v1617	4	3600	49	6.97	7.43	I	(8,0,0)
v1617	8	3600	61	5.65	5.71	I	(10,0,0)
v1617	16	3600	639	1.49	1.55	III	(40,0,1)

quickly and the ability to run a number of different heuristics concurrently. Running multiple heuristics has two positive effects. First, the code is able to find feasible solutions in many more instances, increasing its robustness. Second, the ability to run multiple heuristics can lead to finding good feasible solutions quickly. By finding good solutions quickly, more variables can be removed by reduced cost fixing, which in turn makes the lower bounding procedures more effective.

5.3. Case Study: Crew Scheduling

Section 5.2 showed that using parallelism to solve the SPP can have a positive impact. In this section, we describe our experiences in using PSPS to solve crew scheduling problems coming from a major domestic airline carrier, showing how PSPS could be used as a component in a procedure to solve extremely large, practical SPP instances.

The variables in crew scheduling problems are often called *pairings* and represent a number of flight segments that a crew may serve without violating FAA or union contract restrictions. A more in-depth description of the problem is given by Anbil et al. (1992).

With a variable for every feasible set of flights, crew scheduling problems give rise to enormous set partitioning problem instances, the number of variables being estimated at close to a trillion for some problems (Johnson 1998). One way to obtain good feasible solutions to these instances is to use a heuristic branch-and-price procedure (Barnhart et al. 1998; Vance et al. 1997), in which the variables are considered only as needed. An alternative approach to branch-and-price is to enumerate a large number of pairings and solve the set partitioning problem instance over these pairings. This is the approach we take here. Set partitioning instances created in this manner might contain millions of columns, so applying our code in a straightforward way to such instances is not possible due to core memory limitations. Instead, PSPS was used in conjunction with other tools.

We will report results for two different fleets using two different techniques to obtain good feasible solutions. The first instance (for an Airbus fleet) consists of 190 rows and 8,122,371 columns, and the second instance (for a Boeing fleet) consists of 342 rows and 12,618,766 columns. The procedure used to create the instances is described by Klabjan and Schwan (1999).

To find solutions for the Airbus instance, the linear programming relaxation was solved using the parallel primal-dual subproblem simplex algorithm of Klabjan

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

et al. (2000a). Next, a subset of columns of "reasonable" size was chosen. A reasonable number of columns in this context meant that the problem could fit into the core memory (256MB) of our machines. In this case, we chose the 250,000 variables with smallest reduced cost (with respect to the dual-optimal solution of the linear programming relaxation). PSPS was run on this instance on eight processors, producing an optimal solution and proving its optimality over the given set of columns in less than seven hours. Table 5 shows the progression of PSPS over time.

Next, reduced cost fixing was performed over the set of columns of the original problem left out of the 250,000 variable instance. In so doing, *all* remaining variables could be fixed. Thus, PSPS was able to solve *to optimality* an eight-million-variable SPP instance! The solution obtained was 3.71% better than the best known solution to this problem.

To find solutions for the Boeing instance a different approach had to be taken. After solving the linear programming relaxation, we found that there were about 10 million columns with reduced cost near or equal to zero. Therefore, it was impossible to select a good subset of columns of "workable" size simply by taking variables with the smallest reduced cost. Instead, we used a specialized code of Klabjan et al. (1999) to create the top nodes of the branch-and-bound tree with a specialized strong branching rule. Once a node was reached where a reasonable number of columns remained in the formulation, the PSPS code was used to solve this node. In this case, that meant instances with about 170,000 variables.

Table 5 **Solution Progress for the Airbus Instance**

Elapsed Time	Solution Value
406	1296
435	1177
496	1133
536	1127
614	1084
6813	1081
6866	1080
9105	1078

Table 6 **Solution Progress for the Boeing Instance**

Elapsed Time	Solution Value
3166	1261
10965	1243
17806	1178
29437	1148
50432	1128
56130	1124
68537	1109

On the first instance created this way, using eight processors, PSPS was able to find a solution with a value that was 23.5% lower than the previous best known solution to this problem. PSPS was terminated after 24 hours without proving optimality of the solution. Table 6 shows the progression of PSPS over time.

We have tried several other instances created this way and for one of them PSPS was able to find a feasible solution that was 24% lower than the previous best known solution to this problem. Both improved solutions translate into millions of dollars of savings in crew costs.

Our success in finding much-improved solutions to very large crew scheduling problems using PSPS as a component of a larger solution procedure indicated that parallelization can be a powerful tool in the solution of huge set partitioning problems.

Acknowledgments

The authors would like to thank Diego Klabjan for many useful discussions, and for creating and helping to solve the large crew scheduling instances described in Section 5.3. The authors also acknowledge Alper Atamtürk, who shared code from which Heuristic II was derived. The comments of anonymous Referees and the Associate Editor greatly improved the presentation. This work has been aided by a software grant from CPLEX, a division of ILOG and by a hardware grant from INTEL. The work of authors Linderoth and Lee was partially supported by NSF Grant #9721402. The work of author Savelsbergh was in part supported by NSF Grant DMI-9700285. This work was carried out in part while author Linderoth was at Argonne National Laboratory and supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. This support is gratefully acknowledged.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

Anbil, R., E. L. Johnson, R. Tanga. 1992. A global approach to crew-pairing optimization. *IBM Systems Journal* **31** 73–78.

Atamtürk, A., G. L. Nemhauser, M. W. P. Savelsbergh. 1995. A combined Lagrangian, linear programming, and implication heuristic for large-scale set partitioning problems. *J. Heuristics* **1** 247–259.

Atamtürk, A., G. Nemhauser, M. W. P. Savelsbergh. 2000. Conflict graphs in solving integer programming problems. *European J. Operational Research* **121** 40–55.

Balas, E., M. Padberg. 1976. Set partitioning: A survey. *SIAM Review* **18** 710–760.

Barnhart, C., E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, P. H. Vance. 1998. Branch and price: Column generation for solving huge integer programs. *Operations Research* **46** 316–329.

Bixby, R. E., E. Lee. 1994. Solving a truck dispatching scheduling problem using branch-and-cut. *Operations Research* **46** 355–367.

Borndörfer, R. 1997. Aspects of set packing, partitioning, and covering. Ph.D. thesis, Mathematics Department, Technischen Universtät Berlin, Germany.

Campbell, A. M., L. W. Clarke, A. J. Kleywegt, M. W. P. Savelsbergh. 1998. The inventory routing problem. T. G. Crainic, G. Laporte (eds). *Fleet Management and Logistics*. Kluwer, New York.

Cheng, E., W. H. Cunningham. 1997. Wheel inequalities for stable set polytopes. *Mathematical Programming* **77** 389–421.

Chu, P. C., J. E. Beasley. 1998. Constraint handling in genetic algorithms: The set partitioning problem. *J. Heuristics* **4** 323–357.

Chvátal, V. 1979. A greedy heuristics for the set covering problem. *Mathematics of Operations Research* **4** 233–235.

CPLEX Optimization. 1995. *Using the CPLEX Callable Library*. CPLEX Optimization, Inc., Incline Village, NV.

Eckstein, J. 1994. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM J. Optimization* **4** 794–814.

Esö, M. 1999. Parallel branch and cut for set partitioning. Ph.D. thesis, Department of Operations Research and Industrial Engineering, Cornell University.

Fisher, M., P. Kedia. 1990. Optimal solution of set covering/partitioning problems using dual heuristics. *Management Science* **36** 674–688.

Foster, I. 1995. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison Wesley, Reading, MA.

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. 1994. *PVM: Parallel Virtual Machine*. The MIT Press, Cambridge, MA.

Grötschel, M., L. Lovász, A. Schrijver. 1988. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, New York.

Harche, F., G. L. Thompson. 1994. Column subtraction algorithm: An exact method for solving weighted set covering, packing and partitioning problems. *Computers & Operations Research* **21** 689–705.

Hoffman, K., M. Padberg. 1993. Solving airline crew-scheduling problems by branch-and-cut. *Management Science* **39** 667–682.

Hu, J., E. L. Johnson. 1999. Computational results with a primal-dual subproblem simplex method – A new formulation and decomposition algorithm. *Operations Research Letters* **25** 149–157.

Johnson, E. 1998. Personal communication.

Klabjan, D., K. Schwan. 1999. Airline crew pairing generation in parallel. Technical Report TLI/LEC-99-02. The Logistics Institute, Georgia Institute of Technology, Atlanta, GA.

Klabjan, D., E. Johnson, G. Nemhauser. 1999. Solving large airline crew scheduling problems; Random pairing generation and strong branching. Technical Report TLI/LEC-99-11. The Logistics Institute, Georgia Institute of Technology, Atlanta, GA.

Klabjan, D., E. L. Johnson, G. L. Nemhauser. 2000a. A parallel primal-dual simplex algorithm. *Operations Research Letters* **27** 47–55.

Klabjan, D., E. Johnson, G. Nemhauser, E. Gelman, S. Ramaswamy. 2000b. Solving large airline crew scheduling problems; Random pairing generation and strong branching. *Computational Optimization and Applications*. Forthcoming.

Knuth, D. E. 1998. *The Art of Computer Programming. Volume 3: Sorting and Searching*. 2nd ed. Addison-Wesley, Reading, MA.

Levine, D. 1994. A parallel genetic algorithm for the set partitioning problem. Ph.D. thesis, Illinois Institute of Technology, Chicago, IL.

Marsten, R. E., F. Shepardson. 1981. Exact solution of crew problems using the set partitioning mode: Recent successful applications. *Networks* **11** 165–177.

Müller, T. 1998. Solving set partitioning problems with constraint programming. *Proceedings of the Sixth International Conference on Practical Applications of Prolog and the Fourth International Conference on the Practical Application of Constraint Technology—PAPPACT98*. The Practical Application Company Ltd., London, U.K.

Nemhauser, G. L., G. Sigismondi. 1992. A strong cutting plane/branch-and-bound algorithm for node packing. *Journal of the Operational Research Society* **43** 443–457.

Nemhauser, G. L., L. E. Trotter Jr. 1974. Properties of vertex packing and independence system polyhedra. *Mathematical Programming* **6** 48–61.

Nocedal, J., S. J. Wright. 1999. *Numerical Optimization*. Springer-Verlag, New York.

Padberg, M. 1973. On the facial structure of set packing polyhedra. *Mathematical Programming* **5** 199–215.

Padberg, M. 1974. Perfect zero-one matrices. *Mathematical Programming* **6** 180–196.

Pardalos, P. M., X. Li. 1990. Parallel branch and bound algorithms for combinatorial optimization. *Supercomputer* **39** 23–30.

Perumalla, K., K. Schwan. 1996 (December). *CoGEnt—A Distributed Active Object Framework*. Unpublished manuscript, College of Computing, Georgia Institute of Technology, Atlanta, GA.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ralphs, T., L. Ladányi. 2000. *SYMPHONY: A Parallel Framework for Branch, Cut, and Price.* Available from ftp://ftp.branchandcut.org/pub/reference/symphony.ps.

Reinelt, G. 1991. TSPLIB—a traveling salesman problem library. *ORSA J. Computing* 3 376–384.

Ryan, D. M., J. C. Falkner. 1988. On the integer properties of scheduling set partitioning models. *European J. Operational Research* **35** 442–456.

Savelsbergh, M. W. P. 1994. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing* **6** 445–454.

Tsang, E. 1993. *Foundations of Constraint Satisfaction.* Academic Press, San Diego, CA.

Vance, P., A. Atamtürk, C. Barnhart, E. Gelman, E. L. Johnson, A. Krishna, D. Mahidhara, G. L. Nemhauser, R. Rebello. 1997. A heuristic branch-and-price approach for the airline crew pairing problem. Technical Report TLI-97-06. School of ISyE, Georgia Institute of Technology.

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

Accepted by Jan Karel Lenstra; received March 1999; revised August 2000; accepted January 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.