# Parallel Mixed Integer Programming

*Robert Bixby*
*William Cook*
*Alan Cox*
*Eva Lee*

**CRPC-TR95554**
**June 1995**

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

# Parallel Mixed Integer Programming

Robert E. Bixby,* William Cook,† Alan Cox‡ and Eva K. Lee§

June 30, 1995

**Abstract**

Numerical experiments for a parallel implementation of a branch-and-bound mixed 0/1 integer programming code are presented. Among its features, the code includes cutting-plane generation at the root node, and employs a new branching-variable selection rule within the search tree. The code runs on a loosely-coupled cluster of workstations using TreadMarks as the parallel software platform. Numerical tests were performed on all mixed 0/1 MIPLIB instances as well as two previously unsolved MIP instances, one arising from telecommunication networks and the other a multicommodity flow problem.

**Keywords:** Parallelism; Mixed Integer Programming

**Abbreviated title:** Parallel MIP code

*Department of Computational and Applied Mathematics, Rice University, Houston, Texas. Supported in part by the Center for Research on Parallel Computation, Rice University, and by NSF Grants CCR-881504 and CCR-9407142

†Research Institute for Discrete Mathematics, University of Bonn, Bonn, Germany

‡Department of Computer Sciences, Rice University, Houston, Texas

§Department of Industrial Engineering and Operations Research, Columbia University, New York, New York. Supported in part by postdoctoral fellowship from the Center for Research on Parallel Computation, and by NSF/NATO grant GER-9452935, NSF CAREER grant CCR-9501584

# Parallel Mixed Integer Programming

## 1 Introduction

We report results for a rudimentary 0/1 mixed integer programming code, powerful enough to solve instances of real interest, including all the 0/1 problems in MIPLIB [5], and at least two difficult, previously unsolved models. Two interesting features are that the code runs in parallel on a variety of architectures, including networks of workstations, and that it employs an apparently new branching rule called *strong branching*, which was developed as part of work on the traveling-salesman problem [1].

Other research that exploits parallelism in integer programming includes work on pure 0/1 problems [9], the traveling salesman problem [1], and general mixed integer programming [12]. In Canon and Hoffman [9], a complex branch-and-cut algorithm was run on a network of 9 DECstations, joined to form a "Local Area VAXcluster." Data, such as the global queue of active nodes, were shared through disk files. The test set was a subset of those used in Crowder *et al.* [10]. In Applegate, Bixby, Chvátal and Cook [1], the computations were very coarse-grained, with individual "tasks" often running for a large fraction of a day on the hardest instances. The parallelism, which employed a rather complex list of tasks, was implemented using the master-slave paradigm. Data were shared through message passing over TCP/IP sockets. This code ran on heterogeneous networks of Unix workstations. Eckstein's code [12], in contrast, was written for a specific, dedicated parallel computer, the Thinking Machines CM-5. His code also used message passing to share data. There have been numerous implementations of parallel branch-and-bound as a simple search procedure on similar large-scale parallel computers, but Eckstein's work was, to our knowledge, the first based upon a robust algorithm, capable of solving practical MIP instances of real interest.

Portability and simplicity in our parallel implementation are achieved by using TreadMarks[1]. TreadMarks [17] is a parallel programming system that allows distributed memory computing machines to be programmed as if they were shared memory machines. Thus, our code, which is written in C, should run on any machine to which TreadMarks has been ported[2] and, with minor syntactic changes, shared memory multiprocessors such as the DEC 2100 or the SGI Challenge. The advantage of a system such as TreadMarks is that it is typically much easier to modify sequential programs to use shared memory than to directly use message passing.

A *0/1 mixed integer programming problem* (0/1 MIP) is an optimization problem of the form

[1] TreadMarks is a trademark of Parallel Tools, L.L.C.

[2] TreadMarks is available for DEC's OSF/1, HP's HPUX, IBM's AIX, SGI's IRIX, and Sun's SunOS and Solaris.

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b \\ & x \geq 0 \\ & x_j \in \{0,1\} \ (j = 1, \ldots, p) \end{array} \tag{M}$$

where $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$, $c \in \mathbf{R}^n$, and $p \leq n$.

The remainder of the paper is organized as follows. We begin with sections discussing the basic features of our algorithm: Preprocessing, cutting planes, branching variable and node selection, reduced-cost fixing, and a heuristic for finding integral feasible solutions. This discussion is followed by a section on TreadMarks and a section giving an overall description of our parallel implementation. Finally we present computational results.

# 2 Basic Features of the Algorithm

## 2.1 Preprocessing

Problem preprocessing has been shown to be a very effective way of improving integer programming formulations prior to and during branch-and-bound [7, 10, 15, 19]. Rather than writing our own preprocessor, we have simply employed the CPLEX 3.0 [3] preprocessor, invoking it not only once, but repeatedly until no further reductions result. In addition to applying standard linear programming (LP) reductions, also valid for integer programs, CPLEX applies "coefficient reduction" and "bound strengthening," see [19]. Statistics for the problems solved and the preprocessed versions are given in Table 1. We remark that, without preprocessing, our code could not solve the model *mod011* from MIPLIB, and its performance was seriously affected in a number of other cases.

## 2.2 Cutting planes

The basic algorithm is branch-and-bound. As an additional preprocessing step, before branching begins, we strengthen the formulation by generating cutting planes: Where $P \subseteq \mathbf{R}^n$ is the convex hull of integral feasible solutions of (M), and $x^* \in \mathbf{R}^n \backslash P$, a *cutting plane* for $x^*$ is an inequality $a^T x \leq \alpha$, satisifed by all $x \in P$ and violated by $x^*$ ($a^T x^* > \alpha$). Typically, $x^*$ is the solution of some linear program obtained by relaxing the integrality restrictions of (M).

We have included only three kinds of cutting planes: disjunctive cuts, knapsack cuts, and a very restricted kind of clique cuts. These are discussed in the subsections that follow.

[3] CPLEX is a registered trademark of CPLEX Optimization, Inc.

### 2.2.1 Disjunctive cuts

Disjunctive cuts were introduced by Balas [2], and their computational properties studied extensively in recent work by Balas, Ceria and Cornéjols [3]. Consider the polyhedron

$$P_I = \text{conv}\{x \in \mathbf{R}^n : \hat{A}x \leq \hat{b},\ x_j \in \{0,1\},\ j = 1,\ldots,p\},$$

where we assume that $\hat{A}x \leq \hat{b}$ includes $Ax \leq b$, the restrictions $0 \leq x_j \leq 1$ for $j = 1,\ldots p$, and $x_j \geq 0$ for $j = p+1,\ldots,n$. Let $x^*$ be a feasible solution of $\hat{A}x \leq \hat{b}$ such that $0 < x_i^* < 1$ for some $i \in \{1,\ldots,p\}$ and consider the pair of polyhedra

$$\begin{aligned} P_0 &= \{x \in \mathbf{R}^n : \hat{A}x \leq \hat{b},\ x_i = 0\}, \\ P_1 &= \{x \in \mathbf{R}^n : \hat{A}x \leq \hat{b},\ x_i = 1\}. \end{aligned}$$

Clearly $P_I \subseteq P_{x_i} \equiv \text{conv}(P_0 \cup P_1)$. Assume that both $P_0 \neq \emptyset$, and $P_1 \neq \emptyset$ (otherwise, $x_i$ can be eliminated). Now if $x^* \notin P_{x_i}$, the following procedure will find an inequality valid for $P_I$ and violated by $x^*$.

Consider the system

$$\begin{aligned} \hat{A}y - y_0\hat{b} &\leq 0 \\ \hat{A}z - z_0\hat{b} &\leq 0 \\ y_i &= 0 \\ z_i - z_0 &= 0 \\ y_0 + z_0 &= 1 \\ y + z &= x \end{aligned}$$

where auxiliary variables $y, z \in \mathbf{R}^n$ and $y_0, z_0 \in \mathbf{R}$ have been introduced. It is easy to see that this system is feasible for every $x \in P_{x_i}$, and that it is infeasible for $x = x^*$ if $x^* \notin P_{x_i}$. In this latter case, consider the following feasible LP:

$$\begin{array}{lll} \text{minimize} & \alpha & \\ \text{subject to} & \hat{A}y - y_0\hat{b} & \leq 0 \\ & \hat{A}z - z_0\hat{b} & \leq 0 \\ & y_i & = 0 \\ & z_i - z_0 & = 0 \\ & y_0 + z_0 & = 1 \\ & y + z + \alpha e & \geq x^* \\ & -y - z + \alpha e & \geq -x^* \end{array} \qquad \text{(DISJ)}$$

where $e = (1,\ldots,1)^T \in \mathbf{R}^n$. Let $\alpha^* > 0$ be the optimal value of (DISJ), and let $u^*$, $v^*$, $\beta^*$, $\gamma^*$, $\delta^*$, $s^*$, $t^*$ be an optimal dual solution, where $u^*, v^* \in \mathbf{R}^m$, $\beta^*, \gamma^*, \delta^* \in \mathbf{R}$, and $s^*, t^* \in \mathbf{R}^n$. Then $(t^* - s^*)^T x \geq \delta^*$ is a valid inequality for $P_{x_i}$, but $(t^* - s^*)^T x^* - \delta^* = -\alpha^*$. As an alternative, one might also consider the LP that results by replacing the last two constraints of (DISJ) with $y + z + w = x^*$ and minimizing $\sum_i |w_i|$;

however, the cuts produced by this second alternative seem to be considerably denser, and we did not use it in our tests.

Our implementation of the above idea is straightforward, and, hence, computationally too expensive to be applied as a default: Consider a given $x^*$. For each 0/1 variable $x_i$ such that $0.0001 < x_i^* < 0.9999$, we solve the corresponding instance of (DISJ). If $\alpha^* > 0.001$, we add the resulting valid inequality to our formulation. After all such valid inequalities have been added, a new $x^*$ is computed and the procedure repeated. We stop as soon as the gain in the objective value in the LP relaxation is less than 0.1% over a span of three consecutive $x^*$ computations. Cuts are dropped if the corresponding dual variables remain zero in the solution of 8 consecutive LP relaxations.

The same violation, stopping and cut-deletion criteria are used for knapsack and clique cuts, described below.

The MIPLIB models for which we found it necessary to apply disjunctive cuts are *set1** and *modglob*.

### 2.2.2 Knapsack cuts

A commonly employed technique is to generate cutting planes by analyzing individual constraints. This approach was applied in [10], for pure 0/1 problems, using the well-developed theory of knapsack polyhedra. One way of applying knapsack cuts to mixed 0/1 problems proceeds as follows. Given a constraint

$$\sum_{j=1}^{p} a_j x_j + \sum_{j=p+1}^{n} a_j x_j \leq b,$$

taken from (M), the knapsack inequality

$$\sum_{j=1}^{p} a_j x_j \leq \bar{b}$$

is valid for (M), where

$$\bar{b} = b_i - \sum_{\substack{j=p+1 \\ a_j>0}}^{n} a_j l_j - \sum_{\substack{j=p+1 \\ a_j<0}}^{n} a_j u_j,$$

and $l_j, u_j$, $j = p+1, \ldots, n$, are valid lower and upper bounds for the corresponding continuous variables. Let $x^*$ be a fractional solution, and let $\bar{a}$ and $\bar{x}$ be the vectors obtained after complementing binary variables where necessary to obtain $\bar{a}_j \geq 0$ for $j = 1, \ldots, p$. In our procedure, violated covers are identified using a greedy approach on the nonzero fractional variables, in a fashion similar to that described in [10]. We approximate the optimal objective for the knapsack problem

$$\min\{\sum_{j=1}^{p}(1-\bar{x}_j)s_j : \sum_{j=1}^{p}\bar{a}_j s_j > \bar{b},\ s_j \in \{0,1\},\ j = 1, \ldots, p\},$$

by setting $s_j$ to 1 in nondecreasing order with respect to the ratios $(1-\bar{x}_j^*)/\bar{a}_j$, $j = 1, \ldots, p$. During this solution process, the minimum constraint coefficient among the chosen variables, $\bar{a}_{j_{min}}$, together with its objective coefficient are recorded. There are two cases to consider:

Case 1: If the corresponding objective value is less than 1, a violated cover is obtained. We then check if the sum of the coefficients, excluding $\bar{a}_{j_{min}}$, is less than $\bar{b}$, in which case the cover is minimal; otherwise, we modify it by first discarding $\bar{a}_{j_{min}}$, then resetting as many $s_j$'s to 0 as possible without leaving the feasible region.

Case 2: If the objective value is greater than or equal to 1, and if the objective coefficient corresponding to $\bar{a}_{j_{min}}$ is positive, we correct this cover by discarding $\bar{a}_{j_{min}}$. If that results in a feasible solution with an objective value less than 1, we again obtain a violated cover.

After identifying a violated cover, it is lifted (in both forward and reverse passes) [4, 20, 21]. We approximate the lifting coefficients by solving the linear programming relaxations of the corresponding lifting problems.

### 2.2.3 Clique Cuts

We employ the following exact procedure to find lifted 2-covers. Consider again a constraint of the form:

$$\sum_{j \in B} \bar{a}_j \bar{x}_j \leq \bar{b}.$$

where $\bar{a}$, $\bar{b}$, and $\bar{x}$ are as described in the previous section. Let $B = \{1, \ldots, p\}$,

$$B' = \{j \in B : \bar{a}_j > \bar{b}/2\},$$

and for each $k \in B \setminus B'$, let

$$B'_k = \{k\} \cup \{j \in B' : \bar{a}_k + \bar{a}_j > \bar{b}\}.$$

Then the following *clique inequalities* are valid:

$$\sum_{j \in B'} \bar{x}_j \leq 1$$

$$\sum_{j \in B'_k} \bar{x}_j \leq 1 \quad k \in B \setminus B'.$$

Note that if the elements of $\bar{a}$ are sorted in nonincreasing order, then the entire collection of sets $B'_k$ can be computed in time linear in $|B \cup C|$. For a given $x^*$, determining the clique inequalities violated by $x^*$ is also a linear-time computation.

### 2.2.4 Node and variable selection

To obtain upper bounds at the nodes of the branching tree, we solve the corresponding linear programming relaxations. If the solution is integral, its objective value is exceeded by the value of the best known integral solution, or the LP relaxation is infeasible, the processing of the node is complete; otherwise, a branching variable is selected and two new nodes are created. The rule we use to select the branching variable is *strong*

*branching*, described below. To select the next node for processing we use the *best-bound rule*, taking the active node with the largest objective value.

Strong branching works as follows. Let $N$ and $K$ be positive integers. Given the solution of some linear programming relaxation, make a list of $N$ binary variables that are fractional and closest to 0.5 (if there are fewer than $N$ fractional variables, take all fractional variables). Suppose that $I$ is the index set of that list. Then, for each $x_i$, $i \in I$, starting with the optimal basis for the LP relaxation, fix $x_i$ first to 0.0 and then to 1.0 and perform $K$ iterations of the dual simplex method with steepest-edge pricing using as "normalizing" factors the $L_2$ norms of the rows of the basis inverse (option "dgradient 2" in CPLEX, see [13]). Let $L_i$, $U_i$, $i \in I$, be the objective values that result from these simplex runs, where $L_i$ corresponds to fixing $x_i$ to 0.0 and $U_i$ to fixing it to 1.0. In our implementation we use $N = 10$ and $K = 50$, and select as the branching variable one that minimizes $10.0 \max\{L_i, U_i\} + \min\{L_i, U_i\}$. We note that CPLEX 3.0 implements a special routine for efficiently computing the $L_i$ and $U_i$. A much more detailed study of strong branching is presented in [1].

### 2.2.5 Reduced-cost Fixing and Heuristics

Reduced-cost fixing refers to the fixing of variables to their upper or lower bounds by comparing their reduced-costs to the gap between a linear programming optimum and the current problem lower bound, the best known integral-feasible solution. We perform reduced-cost fixing both globally– at the root node–and locally. Global fixing is applied whenever the gap between the root linear program and the current lower bound changes; local fixing is carried out at each node before and after each heuristic call.

We use the term *(primal) heuristic* to refer broadly to heuristic procedures for constructing "good, approximately optimal" integral feasible solutions from available solutions that are in some sense "good," but fail to satisfy integrality. We incorporate an "adaptive" heuristic based on the heuristic used in [6].

At some node in the branch-and-bound tree, assume that an LP relaxation has been solved and that the optimal solution is fractional. The heuristic works as follows. If some problem lower bound is currently available, reduced-cost fixing is applied (as indicated above). Second, all variables that are identically equal to 1.0 in the current LP solution are fixed to 1.0. Finally, where $\epsilon$ is the current integrality tolerance ($10^{-4}$ by default), the following procedure is applied iteratively until either the LP relaxation yields an integral solution, is infeasible, or has an optimal value that is exceeded by the current lower bound: Let $x^*$ be an optimal solution of the current LP relaxation. Let

$$x^*_{min} = \min\{x^*_j : \epsilon \leq x^*_j \leq 1.0 - \epsilon\}$$

$$x^*_{max} = \max\{x^*_j : \epsilon \leq x^*_j \leq 1.0 - \epsilon\}.$$

The heuristic fixes variables in the following manner.

Case 1: $|x^*_{max} - x^*_{min}| \geq \epsilon$: If $\epsilon \leq x^*_j \leq x^*_{min}$, set $x_j = 0.0$; otherwise, if $x^*_{max} \leq x^*_j$, set $x_j = 1.0$.

Case 2: $|x^*_{max} - x^*_{min}| < \epsilon$: Set $x_j = 1.0$ where $j$ is the smallest index satisfying $x^*_{min} \leq x^*_j \leq x_{max}$.

Instead of applying the heuristic to every branch-and-bound node, our (simple) default selects every node whose depth from the root is a multiple of 4.

# 3 TreadMarks

TreadMarks is a *distributed shared memory* (DSM) system for networks of Unix workstations and distributed-memory multiprocessors, such as the IBM SP2. DSM enables processes running on different workstations to share data through a network-wide virtual memory, even though the hardware provided by the network lacks the capability for one workstation to access another workstation's physical memory [18]. For example, Figure 1 illustrates a DSM system consisting of $N$ workstations, each with its own physical memory, connected by a network. The DSM software implements the abstraction of a network-wide virtual memory, denoted by the dashed line in the figure, in which each processor can access any data item, without the programmer having to worry about where the data is, or how to obtain its value. In contrast, in the "native" programming model directly provided by the hardware, *message passing*, the programmer must decide *when* a processor needs to communicate, with *whom* to communicate, and *what* data to communicate. For programs requiring complex data structures and parallelization strategies, this implementation can become a difficult task. On a DSM system, the programmer can focus on the development of a good parallel algorithm rather than on partitioning data among the workstations and communicating values. In addition to ease of programming, DSM provides the same programming environment as that on (hardware) shared-memory multiprocessors, allowing for portability between the two environments.

TreadMarks is provided to the user as an ordinary software library that is linked with the user's parallel program. Standard Unix compilers and linkers are used to build TreadMarks programs. Furthermore, no kernel modifications or special (super-user) privileges are required to execute parallel programs.

The challenge in developing an efficient DSM system is to minimize the amount of communication (message-passing) required to implement the shared memory abstraction, in particular, to insure *data consistency*. Data consistency is the guarantee that changes to shared memory variables get propagated to each processor before that processor tries to use the variable. Various techniques are used by TreadMarks to meet this challenge, including lazy release consistency [16] and a multiple-writer protocol [8].

Lazy release consistency is a novel algorithm that implements the release consistency memory model developed by Gharachorloo *et al.* [14]. From the programmer's standpoint, release consistency is identical to the traditional (hardware) multiprocessor shared-memory model, sequential consistency, if the data accesses by different processors are correctly synchronized. However, unlike sequential consistency, release consistency does not require data consistency at each write to shared memory. Instead, lazy release consistency enforces data consistency when a synchronization object, such as a lock, is acquired. In contrast, earlier im-

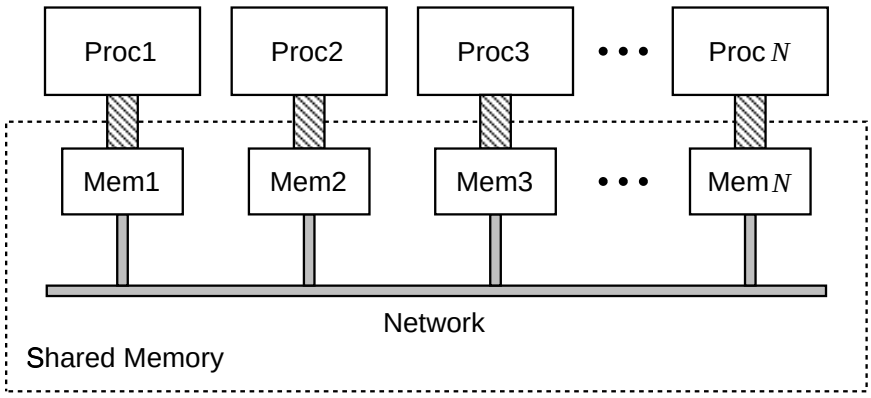


Figure 1: Distributed Shared Memory

plementations of release consistency enforced data consistency when a synchronization object was released. This difference has the effect that lazy release consistency only requires data consistency messages to travel between the last releaser and the new acquirer, instead of a global broadcast at each release. As a result, lazy release consistency requires fewer messages to be sent.

All shared-memory multiprocessors, such as the DEC 2100 and SGI Challenge, and most DSM systems use single-writer protocols. These protocols allow multiple readers to access a given page simultaneously, but a writer is required to have sole access to a page before performing any modifications. Single-writer protocols are easy to implement because all copies of a given page are always identical. So a processor that needs a copy of the page can retrieve one from any processor that has a current copy. Unfortunately, this simplicity often comes at the expense of message traffic. Before a page can be written, all other copies must be invalidated. These invalidations can then cause subsequent requests for the page if the processors whose pages have been invalidated are still accessing data.

As the name implies, multiple-writer protocols allow multiple processes to simultaneously modify the same page, with data consistency messages deferred until a later time, when synchronization occurs. TreadMarks uses the virtual memory hardware to detect accesses and modifications to shared memory pages. Shared pages are initially write-protected. When a write occurs, the protocol creates a copy of the virtual memory page, a *twin*, and saves the twin in system space. When modifications must be sent to another processor, the current copy of the page is compared with the twin on a word-by-word basis and the bytes that vary are saved into a "diff" data structure. Once the diff has been created, the twin is discarded. With the exception of the first time a processor accesses a page, its copy of the page is updated exclusively by applying diffs; a new complete copy of the page is never needed. Thus, the diffs provide two benefits that outweigh the computation overhead. First, they can be used to implement a multiple-writer protocol, reducing the number of messages sent between processors; second, they usually reduce the amount of data sent because a diff only contains the parts of a page that changed.

# 4 Implementation

At the startup of the parallel code, one processor is responsible for reading the problem. That processor also solves the initial linear programming relaxation. If the optimal solution is integral feasible, the algorithm is done; otherwise, the heuristic is called. If it succeeds, reduced-cost fixing is performed. Cut generation is then called in an attempt to improve the upper bound (in the case of maximization). Once "enough" cuts are generated, reduced-cost fixing is performed again. After that, sequential branch-and-bound is performed until the number of active nodes accumulated exceeds a predetermined threshold. This threshold is an increasing function of the number of processors. We delay the beginning of the parallel branch-and-bound to avoid having processors contend for access to the global active list until nodes are available for processing. Such contention generates useless communication, which slows the accumulation of active nodes.

The shared data in our implementation consists of the best lower bound (for maximization problems), its corresponding solution, and the global list of active nodes. For an individual processor, the initial setup consists of reading in a transparent copy of the linear programming relaxation, as well as all the cuts appended to this linear program after performing cut generation at the root. The processors then perform the following procedure repeatedly until the entire list of active nodes is exhausted and every processor is idle, signaling the completion of the parallel processing. Each idle processor fetches an active node from the list, using best-node selection, and reads the current best lower bound. The linear program is then solved. If an integral solution is obtained, this node is fathomed without further branching; otherwise, local reduced-cost fixing is performed, and the heuristic is called according to the heuristic interval setting. If the heuristic is performed and a better lower bound is obtained, the best lower bound and solution are "updated." If there is no gap between the linear programming objective value and this lower bound, for the current node, the node is fathomed; otherwise, a branching variable is selected and two new nodes corresponding to the selected variable are added to the list of active nodes.

Our approach is centralized in the sense that a global list of active nodes is stored in a single, shared data structure. When a processor becomes idle, it selects the current best-node from the global list. To insure that only one processor accesses this list at a time, a "lock" is acquired before the list is accessed and subsequently released after processing is complete; locks are a standard synchronization facility provided by TreadMarks. After fetching an active node from the list, a processor is only certain of the current lower bound until the lock is released. As the computation on the node proceeds, the other processors may update the lower bound. The processor that is working on the node will not discover the update until it attempts to read or update the shared lower bound itself. While it is possible that some unnecessary computation is performed, it is equally important that the amount of communication not become excessive.

We have experimented with various ways of handling critical sections in the code, and our current implementation tries to strike a balance between computation and communication overhead. If the amount of work between consecutive accesses to the active list is too small, then the overhead associated with the

lock mechanism can be significant. This fact suggests the alternative strategy of fetching several nodes, rather than just one, with each access to the active list. While this idea certainly deserves further testing, in our limited examination, it did not improve the results, and thus was not included in our final testing. For example, on *stein45*, when our code was modified to fetch two nodes instead of one per access, the running time on two processors increased from 3227.9 seconds to 3429.2 and the node count increased by 399.

# 5 Numerical Results

Numerical tests were performed on all of the mixed 0/1 instances from MIPLIB and on two additional, previously unsolved mixed 0/1 integer programs. One was a multicommodity flow instance, supplied to us by Dan Bienstock, and the other was a telecommunication network problem. The former model included a significant number of non-trivial cutting planes (424 in total) added as a result of the research by Bienstock and Günlük [11].

Let $T_n$ denote the time elapsed when $n$ processors are used. In our tests, $T_n$ was always measured using "wall-clock" time, and was recorded starting from reading in the problem instance to the final shutdown of all processors after printing the solution. We define the *speedup* for $n$ processors to be the ratio $T_1/T_n$. (See the later discussion of the alternative measure $\frac{T_1 - T_{\text{startup}}}{T_n - T_{\text{startup}}}$.)

Table 1 shows problem statistics. Here, *Name, Original Rows, Cols*, and *0/1 var* denote, respectively, the name of the test instance, the initial number of rows, the number of columns, and the number of 0/1 variables in the constraint matrix. *Preprocessed Rows, Cols* and *0/1 var* denote the size after running CPLEX's presolve procedure. *Initial LP Objective, Preprocessed LP Objective* and *Optimal MIP Objective* record, respectively, the optimal objective value for the initial LP relaxation, the optimal objective value for the initial LP relaxation after preprocessing, and the optimal objective value for the original integer program. Of 51 problem instances, the presolve procedure closed the gap for 16; on some of the more difficult models (e.g., *fixnet**, *set1al*) the gap was reduced by over 70%.

The code is set to use the dual simplex method for resolving the branch-and-bound nodes, strong branching with 50 dual steepest pivots, and a simple "adaptive" primal heuristic with heuristic interval 4. In addition, branch-and-bound nodes are stored internally in a global list, a best-node selection strategy is employed, and clique and knapsack cuts are applied at the root node. Cuts are removed at the root after being inactive for 8 consecutive LP solves. Cuts are not removed at non-root nodes. (More details of these algorithmic components are given in sections 2 and 4.) Nodes are fathomed (for maximization problems) when a provable bound is known that is within 0.01 of best-known feasible solution; in the special case that only integral variables appear in the objective, and all objective coefficients are also integral, this cutoff tolerance is increased to 0.99.

Tables 2a and 2b record statistics of problems after initial cut generation at the root node. Only those

problems for which cuts were actually generated are included. *Clique passes*, *Cliques Found*, *Clique Time*, *Knap. passes*, *Knap. Found* and *Knap. Time* record, respectively, the number of clique passes, the number of cliques generated, total time for clique generation, the number of knapsack passes, the number of knapsack cuts found and total time for knapsack generation. *Total Cuts added* denotes the final number of cuts appended to the LP relaxation after weak cuts are discarded. *Initial Obj.*, *Cut Obj.*, and *Optimal IP Obj.* denote, respectively, the optimal objective value of the initial LP relaxation, the optimal LP objective value after presolve and cut generation, and the optimal objective value for the integer program itself. The last column gives the percentage of the gap closed due to presolve and cut generation. Table 2b shows the statistics for the instances where disjunctive cuts were necessary. There were four such cases, and these were the only four models for which disjunctive cuts were used.

Tables 3 and 4 show, respectively, the solution time (in seconds) and the total number of branch-and-bound nodes searched for each model, running on $n$ SPARC20s' where $n$ ranged from 1 to 8. For each $n$, we performed four independent runs on each model. The best sequential time was then recorded for $n = 1$. The parallel running time is an average over the four runs. $T_{startup}$ records the time elapsed before parallel execution began. Observe that our implementation of disjunctive cuts, while effective in closing the gap, can be computationally expensive (see *modglob* and *set1** in Tables 2b and 3). In contrast, clique and knapsack cuts — which are also very effective in closing the gap — are computationally inexpensive to generate. As a measure of speedup, we used the ratio $\frac{T_1 - T_{\text{startup}}}{T_8 - T_{\text{startup}}}$ rather than the more-standard $\frac{T_1}{T_8}$. The former ratio reflects the speedup in the parts of the algorithm that we actually attempted to parallelize.

Few of the problems with sequential solution times under 100 seconds achieved significant speedup. Indeed, in some instances, the parallel runnings times were actually slower than the sequential time. Such behavior was not unexpected, and can be largely attributed to communication overhead. In addition, several of these models simply do not generate enough nodes to justify (or necessitate) the use of parallelism. We observe, in addition, that parallelism does occasionally lead to extra work, and in some instances that extra work is excessive. Consider, for example, the model *rentacar*. When running sequentially, only 18 nodes were solved and the number of nodes in the queue was never greater than 2. However, when multiple processors were used, as many as 50 total nodes were processed, with considerable effort expended on processing nodes that were fathomed in the sequential run.

In Table 5, we present a summary of the speedup for problems with sequential running times greater than 1000 seconds. We observe that significant speedup is realized in most of these problems. Note that in problems *vpm1*, *misc07* and *stein45*, the node counts remained essentially constant regardless of the number of processors used. The constant node counts were due to the fact that for each of these models, an optimal solution was found very early in the branch-and-bound process (after 400 nodes in both *misc07*, and *stein45*, and 2400 nodes in *vpm1*.) Note also that for *vpm1*, performance deteriorated significantly when the number of processors exceeded 4: The LP relaxations for this model are extremely easy. The number of simplex

iterations per node solved is around 2 to 15. As a result, the processors spent the majority of their time waiting to acquire the lock on the list of active nodes. For example, on 8 processors, each processor spent 62% of its time waiting. In contrast, for *misc07*, each processor spent only 4% of its time waiting.

The problems *air04* and *air05* exhibit some of the same behavior as, for example, *rentacar*: Nodes are generated too slowly to effectively use a larger number of processors. Thus, when the number of processors is small, performance (speedup) remains good, but eventually, as the number of processors grows, the number of nodes also grows. Since the linear programs for these problems are far from trivial (taking about 50 seconds, and 30 seconds, respectively, per node), any increase in node counts directly influences the solution time.

For problem *p6000*, consistent superlinear speedup is realized. The explanation is three fold. First, when running in parallel, the optimal solution is found much more quickly. Second, the value of the objective is integral. Thirdly, the gap between the optimal value and the objective of the LP relaxation is relatively small. Thus, where k denotes the optimal value of the objective, failure to find k early means, in this problem, that a large number of nodes with LP value in the gap $(k-1, k]$ are unnecessarily processed.

Table 6 reports the solution time for two previously unsolved problems: a multicommodity flow problem *quasiunif2*, and a telecommunication network problem *teleicm*. Here, *Rows*, *Cols*, *0/1 var* denote, respectively, the initial numbers of rows, columns and 0/1 variables in the problem. *Cuts* denotes the total number of cuts added. *LP Obj.*, *Cut Obj.*, and *Optimal Obj.* are, respectively, the objective value of the initial LP relaxation, the objective of the LP with cuts appended, and the optimal MIP objective value. It is interesting to note that while the gap for *quasiunif2* is relatively small after the addition of cuts, this problem is very difficult to solve. (This difficulty is the reason for not reporting running times with fewer than eight processors.)

# 6 Conclusion

We have presented a simple parallel branch-and-bound implementation for mixed integer programs. The implementation is built on TreadMarks, a distributed shared memory software environment that provides the abstraction of a network-wide virtual memory. Such an environment provides for ease of programming on networks of Unix workstations, as well as portability across platform and network types.

The MIP code incorporates strategies such as heuristics, problem reformulation, and cutting plane generation that have repeatedly been shown to be effective — particularly when combined — in solving difficult, real-world MIP instances. In addition, we use an apparently new branching approach, called strong branching, whereby a branching variable is selected based upon a rule that involves performing a fixed number of dual simplex pivots on each LP in a sequence of LP's derived from sequentially fixing each variable in a collection of fractional 0/1 variables to its upper and lower bound.

Our numerical results demonstrate that this code is powerful enough to solve all the mixed 0/1 MIPLIB problem instances, as well as two other difficult, real instances. Moreover, the speedup achieved on the harder instances is in most cases close to linear, and in some cases superlinear. Thus, this work provides some justification for the time-consuming task of developing even more sophisticated mixed integer programming codes in a similar environment.

Table 1. Problem Statistics of 51 0/1 MIPLIB Problems

| | Original | | | Preprocessed | | | Initial | Preprocessed | Optimal | Percentage (%) |
|---|---|---|---|---|---|---|---|---|---|---|
| Name | Rows | Cols | 0/1 var. | Rows | Cols | 0/1 var. | LP Objective | LP Objective | MIP Objective | Gap closed |
| air01 | 23 | 771 | 771 | 23 | 771 | 771 | 6743.0 | 6743.0 | 6796 | 0 |
| air02 | 20 | 6774 | 6774 | 20 | 6774 | 6774 | 7640.0 | 7640.0 | 7810 | 0 |
| air03 | 124 | 10757 | 10757 | 122 | 10755 | 10755 | 338864.25 | 338864.25 | 340160 | 0 |
| air04 | 823 | 8904 | 8904 | 777 | 8873 | 8873 | 55535.436 | 55355.436 | 56137 | 0 |
| air05 | 426 | 7195 | 7195 | 408 | 7195 | 7195 | 25877.609 | 25877.609 | 26374 | 0 |
| air06 | 825 | 8627 | 8627 | 757 | 8560 | 8560 | 49616.364 | 49616.364 | 49649 | 0 |
| bm23 | 20 | 27 | 27 | 20 | 27 | 27 | 20.57 | 20.57 | 34 | 0 |
| cracpb1 | 143 | 572 | 572 | 125 | 573 | 572 | 22199.0 | 22199.0 | 22199 | 0 |
| diamond | 4 | 2 | 2 | 1 | 1 | 1 | -1.0 | 0.5 | Infeasible | - |
| egout | 98 | 141 | 55 | 40 | 52 | 28 | 149.589 | 367.085 | 568.1007 | 52.0 |
| enigma | 21 | 100 | 100 | 21 | 100 | 100 | 0.0 | 0.0 | 0.0 | - |
| fixnet3 | 478 | 878 | 378 | 477 | 877 | 378 | 40717.018 | 50285.766 | 51845 | 86.0 |
| fixnet4 | 479 | 878 | 378 | 477 | 877 | 378 | 4257.966 | 7689.478 | 8922 | 73.6 |
| fixnet6 | 479 | 878 | 378 | 477 | 877 | 378 | 1200.884 | 3190.042 | 3981 | 71.6 |
| khb05250 | 101 | 1350 | 24 | 100 | 1299 | 24 | 95919464.0 | 95919464.0 | 106940226 | 0 |
| l152lav | 97 | 1989 | 1989 | 97 | 1989 | 1989 | 4656.363 | 4656.363 | 4722 | 0 |
| lp4l | 85 | 1086 | 1086 | 85 | 1086 | 1086 | 2942.5 | 2942.5 | 2967 | 0 |
| lseu | 28 | 89 | 89 | 28 | 88 | 88 | 834.68 | 944.754 | 1120 | 38.6 |
| misc01 | 54 | 83 | 82 | 53 | 78 | 78 | 57.0 | 57.0 | 563.5 | 0 |
| misc02 | 39 | 59 | 58 | 38 | 54 | 54 | 1010.0 | 1010.0 | 1690 | 0 |
| misc03 | 96 | 160 | 159 | 95 | 153 | 153 | 1910.0 | 1910.0 | 3360.0 | 0 |
| misc04 | 1725 | 4897 | 30 | 1079 | 4155 | 30 | 2656.424 | 2656.424 | 2666.699 | 0 |
| misc05 | 300 | 136 | 74 | 257 | 122 | 70 | 2930.9 | 2930.9 | 2984.5 | 0 |
| misc06 | 820 | 1808 | 112 | 664 | 1519 | 112 | 12841.689 | 12841.689 | 12850.861 | 0 |
| misc07 | 212 | 260 | 259 | 211 | 253 | 253 | 1415.0 | 1415.0 | 2810 | 0 |
| mod008 | 6 | 319 | 319 | 6 | 319 | 319 | 290.931 | 290.931 | 307 | 0 |
| mod010 | 146 | 2655 | 2655 | 146 | 2655 | 2655 | 6532.083 | 6532.083 | 6548 | 0 |
| mod011 | 4480 | 10958 | 96 | 2387 | 8050 | 96 | -6212982.552 | -6212982.552 | -54558535.0 | 0 |
| mod013 | 62 | 96 | 48 | 62 | 96 | 48 | 256.016 | 256.016 | 280.95 | 0 |
| modglob | 291 | 422 | 98 | 289 | 389 | 98 | 20430947.619 | 20430947.619 | 20740508.1 | 0 |
| p0033 | 16 | 33 | 33 | 15 | 32 | 32 | 2520.57 | 2819.357 | 3089 | 52.6 |
| p0040 | 23 | 40 | 40 | 13 | 40 | 40 | 61796.55 | 61829.081 | 62027 | 14.1 |
| p0201 | 133 | 201 | 201 | 113 | 195 | 195 | 6875.0 | 7125.0 | 7615 | 33.8 |
| p0282 | 241 | 282 | 282 | 161 | 202 | 202 | 176867.5 | 180000.300 | 258411 | 3.8 |
| p0291 | 252 | 291 | 291 | 66 | 103 | 103 | 1705.13 | 2921.375 | 5223.749 | 34.6 |
| p0548 | 176 | 548 | 548 | 151 | 477 | 477 | 315.29 | 3126.383 | 8691 | 33.6 |
| p2756 | 755 | 2756 | 2756 | 729 | 2684 | 2684 | 2688.75 | 2701.144 | 3124 | 2.8 |
| p6000 | 2176 | 6000 | 6000 | 2171 | 5995 | 5995 | -2451537.325 | -2451537.325 | -2451377 | 0 |
| pipex | 25 | 48 | 48 | 25 | 48 | 48 | 773.751 | 773.751 | 788.263 | 0 |
| rentacar | 6803 | 9557 | 55 | 1392 | 3208 | 27 | 28806137.644 | 28928379.620 | 30356760.984 | 7.9 |
| rgn | 24 | 180 | 100 | 24 | 180 | 100 | 48.799 | 48.799 | 82.199 | 0 |
| sample2 | 45 | 67 | 21 | 45 | 64 | 21 | 247.0 | 247.0 | 375 | 0 |
| sentoy | 30 | 60 | 60 | 30 | 60 | 60 | -7839.278 | -7839.278 | -7772 | 0 |
| set1al | 493 | 712 | 240 | 432 | 652 | 220 | 11145.628 | 14508.272 | 15869.75 | 71.2 |
| set1ch | 493 | 712 | 240 | 446 | 666 | 235 | 32007.729 | 33537.021 | 54537.75 | 6.8 |
| set1cl | 493 | 712 | 240 | 431 | 651 | 220 | 1671.958 | 1827.674 | 6484.25 | 3.2 |
| stein9 | 13 | 9 | 9 | 13 | 9 | 9 | 4.0 | 4.0 | 5 | 0 |
| stein15 | 36 | 15 | 15 | 36 | 15 | 15 | 7.0 | 7.0 | 9 | 0 |
| stein27 | 118 | 27 | 27 | 118 | 27 | 27 | 13.0 | 13.0 | 18 | 0 |
| stein45 | 331 | 45 | 45 | 331 | 45 | 45 | 22.0 | 22.0 | 30 | 0 |
| vmp1 | 234 | 378 | 168 | 174 | 270 | 120 | 15.416 | 15.416 | 20 | 0 |

Table 2a. Problems Statistics After Default Cut Generation

| Name | Clique passes | Clique Found | Clique Time | Knap. passes | Knap. Found | Knap. Time | Total Cuts added | Initial Obj. | Cut Obj. | Optimal IP Obj. | Percentage (%) Gap closed |
|---|---|---|---|---|---|---|---|---|---|---|---|
| bm23 | 1 | 0 | 0.0 | 3 | 5 | 0.1 | 4 | 20.57 | 21.344 | 34 | 5.8 |
| enigma | 1 | 0 | 0.0 | 2 | 1 | 0.0 | 1 | 0.0 | 0.0 | 0.0 | – |
| lp4l | 1 | 1 | 0.1 | 0 | 2 | 0.2 | 1 | 2942.5 | 2943 | 2967 | 2.0 |
| lseu | 1 | 0 | 0.0 | 3 | 10 | 0.1 | 7 | 834.68 | 948.446 | 1120 | 40.0 |
| mod010 | 1 | 0 | 0.6 | 3 | 2 | 0.8 | 2 | 6532.083 | 6532.6 | 6548 | 3.2 |
| p0033 | 2 | 1 | 0.0 | 4 | 14 | 0.0 | 6 | 2520.57 | 2881.834 | 3089 | 63.6 |
| p0040 | 1 | 0 | 0.0 | 3 | 2 | 0.0 | 2 | 61796.55 | 61973.57 | 62027 | 76.8 |
| p0201 | 1 | 0 | 0.1 | 2 | 4 | 0.2 | 4 | 6875.0 | 7125 | 7615 | 33.8 |
| p0282 | 3 | 7 | 0.3 | 8 | 81 | 0.7 | 48 | 176867.5 | 252367.75 | 258411 | 92.6 |
| p0291 | 2 | 2 | 0.0 | 3 | 3 | 0.1 | 5 | 1705.13 | 4896.64 | 5223.749 | 90.7 |
| p0548 | 6 | 13 | 0.9 | 15 | 110 | 2.5 | 96 | 315.29 | 7063.01 | 8691 | 80.6 |
| p2756 | 3 | 27 | 9.2 | 7 | 257 | 18.1 | 212 | 2688.75 | 3062.30 | 3124 | 85.8 |
| p6000 | 1 | 0 | 13.9 | 53 | 68 | 36.1 | 24 | -2451537.325 | -2451524.79 | -2451377 | 7.8 |
| pipex | 1 | 1 | 0.0 | 4 | 11 | 0.1 | 7 | 773.751 | 776.27 | 788.263 | 17.4 |
| sentoy | 1 | 1 | 0.0 | 5 | 16 | 0.3 | 7 | -7839.278 | -7832.49 | -7772 | 10.1 |

Table 2b. Problems Statistics After Disjunctive Cut Generation

| Name | Disj. passes | Disj. Found | Disj. Time | Disj. Cuts added | Initial Obj. | Cut Obj. | Optimal IP Obj. | Percentage (%) Gap closed |
|---|---|---|---|---|---|---|---|---|
| modglob | 7 | 84 | 1762.9 | 74 | 20430947.619 | 20612619.17 | 20740508.1 | 58.7 |
| set1al | 5 | 189 | 321.3 | 185 | 11145.628 | 15602.13 | 15869.75 | 94.4 |
| set1ch | 12 | 878 | 2314.2 | 339 | 32007.729 | 54516.98 | 54537.75 | 99.9 |
| set1cl | 1 | 180 | 158.4 | 180 | 1671.958 | 6218.76 | 6484.25 | 94.4 |

Table 3. Running Time (in seconds) on $n$ SPARC20's

| Name | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ | $\frac{T_1 - T_{\text{startup}}}{T_8 - T_{\text{startup}}}$ | $T_{\text{startup}}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| air01 | 0.8 | 3.0 | 3.3 | 3.8 | 4.2 | 3.7 | 4.7 | 5.0 | – | 0.8 |
| air02 | 46.1 | 50.5 | 66.5 | 61.2 | 58.6 | 53.0 | 60.5 | 69.4 | 0.64 | 5.5 |
| air03 | 60.1 | 74.5 | 85.0 | 83.7 | 98.8 | 78.5 | 95.1 | 80.9 | 0.62 | 26.3 |
| air04 | 8750.7 | 4556.8 | 3324.7 | 2590.0 | 2481.9 | 2947.0 | 2577.3 | 3210.4 | 2.9 | 296.7 |
| air05 | 27655.9 | 14071.8 | 9474.8 | 7143.1 | 5786.3 | 5392.0 | 4630.2 | 4237.0 | 6.7 | 96.8 |
| air06 | 215.0 | 203.3 | 287.7 | 243.9 | 279.8 | 220.3 | 247.2 | 267.6 | 0.68 | 104.7 |
| bm23 | 3.2 | 2.4 | 3.2 | 2.2 | 3.2 | 3.8 | 7.7 | 8.9 | 0.35 | 0.1 |
| cracpb1 | 1.2 | 1.1 | 1.1 | 1.1 | 1.2 | 1.1 | 1.1 | 1.3 | 0.5 | 1.1 |
| diamond | 0.1 | 0.2 | 0.3 | 0.3 | 0.4 | 0.4 | 0.5 | 0.7 | – | 0.1 |
| egout | 79.8 | 58.6 | 52.8 | 52.3 | 41.7 | 53.3 | 47.4 | 52.2 | 1.53 | 0.1 |
| enigma | 16.7 | 9.6 | 9.2 | 8.5 | 8.7 | 9.0 | 10.0 | 11.7 | 1.43 | 0.1 |
| fixnet3 | 11.4 | 7.8 | 7.6 | 7.1 | 8.1 | 7.3 | 9.3 | 8.6 | 1.35 | 0.6 |
| fixnet4 | 45.4 | 26.0 | 18.6 | 15.3 | 18.6 | 13.9 | 14.5 | 12.4 | 3.85 | 0.8 |
| fixnet6 | 73.9 | 37.8 | 27.1 | 22.2 | 26.9 | 19.7 | 18.8 | 17.3 | 4.45 | 0.9 |
| khb05250 | 272.0 | 164.6 | 111.0 | 90.1 | 81.7 | 67.4 | 60.1 | 49.2 | 5.58 | 0.5 |
| l152lav | 1273.4 | 730.4 | 467.4 | 344.3 | 289.4 | 238.5 | 200.1 | 186.0 | 6.94 | 2.9 |
| lp4l | 44.1 | 11.4 | 15.2 | 12.9 | 14.8 | 14.5 | 13.9 | 15.1 | 3.09 | 1.2 |
| lseu | 63.9 | 40.2 | 31.1 | 26.1 | 24.3 | 24.8 | 28.2 | 31.1 | 2.06 | 0.2 |
| misc01 | 26.0 | 13.8 | 10.9 | 8.6 | 9.0 | 8.8 | 9.7 | 9.8 | 2.67 | 0.1 |
| misc02 | 2.0 | 1.9 | 2.5 | 2.5 | 2.9 | 3.3 | 3.4 | 4.0 | 0.49 | 0.1 |
| misc03 | 82.7 | 49.7 | 55.5 | 60.8 | 63.4 | 63.7 | 68.4 | 67.5 | 1.23 | 0.2 |
| misc04 | 19.1 | 19.7 | 21.4 | 21.7 | 21.8 | 23.7 | 22.0 | 24.1 | 0.68 | 8.5 |
| misc05 | 76.4 | 42.8 | 36.8 | 23.7 | 18.0 | 16.3 | 14.4 | 14.1 | 5.52 | 0.3 |
| misc06 | 27.6 | 18.4 | 18.6 | 15.6 | 16.4 | 16.2 | 17.9 | 17.7 | 1.64 | 2.1 |
| misc07 | 20410.3 | 10532.1 | 6953.2 | 5255.1 | 4269.1 | 3541.1 | 3017.9 | 2626.1 | 7.78 | 0.5 |
| mod008 | 94.5 | 58.4 | 44.6 | 35.7 | 32.9 | 37.3 | 35.6 | 35.1 | 2.70 | 0.1 |
| mod010 | 78.1 | 39.9 | 42.3 | 43.2 | 44.7 | 40.9 | 42.2 | 40.4 | 2.10 | 5.9 |
| mod011* | 187982.9 | 103321.3 | 67298.9 | 50675.5 | 40816.5 | 38573.4 | 27644.4 | 21240.9 | 8.86 | 13.2 |
| mod013 | 10.1 | 6.8 | 6.6 | 4.6 | 4.8 | 6.3 | 7.9 | 8.0 | 1.27 | 0.1 |
| modglob+ | 7340.7 | 4527.8 | 3722.1 | 3254.6 | 3110.9 | 2806.6 | 2771.5 | 2598.3 | 7.44 | 1861.9 |
| p0033 | 4.0 | 3.5 | 3.7 | 3.7 | 5.0 | 4.8 | 6.1 | 7.2 | 0.55 | 0.1 |
| p0040 | 0.2 | 0.2 | 0.2 | 0.6 | 0.6 | 0.6 | 0.6 | 0.8 | 0.14 | 0.1 |
| p0201 | 62.7 | 34.6 | 24.7 | 18.0 | 21.0 | 20.0 | 19.6 | 20.7 | 3.08 | 0.5 |
| p0282 | 31.0 | 19.5 | 15.4 | 13.4 | 13.0 | 12.7 | 13.4 | 13.5 | 2.42 | 1.2 |
| p0291 | 2.3 | 1.4 | 1.5 | 1.7 | 2.0 | 2.2 | 2.6 | 2.8 | 0.81 | 0.2 |
| p0548 | 13806.3 | 7667.1 | 5206.4 | 3926.0 | 3161.1 | 2666.8 | 2306.3 | 2021.1 | 6.85 | 4.8 |
| p2756 | 8209.4 | 4090.9 | 2725.5 | 2060.2 | 1677.3 | 1417.2 | 1237.8 | 1089.2 | 7.71 | 28.6 |
| p6000 | 5889.1 | 2823.0 | 1933.8 | 1331.5 | 1109.7 | 949.7 | 850.1 | 755.1 | 9.32 | 137.8 |
| pipex | 16.8 | 11.6 | 10.1 | 9.5 | 9.3 | 10.8 | 11.1 | 12.3 | 1.41 | 1.2 |
| rentacar | 120.9 | 117.6 | 120.3 | 127.2 | 129.7 | 154.4 | 149.0 | 150.1 | 0.79 | 8.9 |
| rgn | 64.9 | 40.6 | 33.1 | 26.4 | 24.8 | 29.8 | 28.4 | 27.5 | 2.37 | 0.2 |
| sample2 | 4.4 | 3.3 | 3.9 | 4.7 | 3.4 | 5.1 | 5.6 | 5.5 | 0.80 | 0.1 |
| sentoy | 15.8 | 8.0 | 6.3 | 5.1 | 5.1 | 7.0 | 4.9 | 7.5 | 2.63 | 2.4 |
| set1al+ | 616.4 | 471.9 | 433.6 | 407.1 | 389.9 | 382.8 | 381.2 | 373.1 | 6.41 | 328.1 |
| set1ch+ | 2440.9 | 2372.9 | 2368.2 | 2361.6 | 2340.2 | 2333.1 | 2337.2 | 2329.1 | 11.26 | 2318.2 |
| set1cl+ | 208.5 | 292.9 | 253.1 | 232.3 | 215.5 | 211.9 | 213.4 | 198.3 | 1.28 | 161.5 |
| stein9 | 0.3 | 0.7 | 0.6 | 0.8 | 0.8 | 3.9 | 2.6 | 0.6 | 0.4 | 0.1 |
| stein15 | 0.6 | 0.8 | 2.8 | 4.8 | 5.4 | 2.5 | 3.6 | 3.2 | 0.16 | 0.1 |
| stein27 | 152.6 | 86.3 | 63.8 | 47.5 | 41.8 | 37.2 | 38.0 | 37.4 | 4.09 | 0.1 |
| stein45 | 5927.9 | 3227.0 | 2138.5 | 1614.1 | 1310.3 | 1121.7 | 966.0 | 802.1 | 7.39 | 0.3 |
| vmp1 | 27624.5 | 16079.0 | 11671.7 | 9691.9 | 9339.5 | 9922.2 | 9773.5 | 10403.6 | 2.66 | 1.4 |

* Run on an SP1 with 8 nodes.

+ Disjunctive cuts activated.

Table 4. Nodes Searched on $n$ Processors

| Name | $n = 1$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| air01 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| air02 | 10 | 11 | 11 | 11 | 11 | 11 | 10 | 11 |
| air03 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| air04 | 176 | 182 | 188 | 173 | 183 | 195 | 184 | 220 |
| air05 | 1061 | 1031 | 1032 | 1039 | 1031 | 1194 | 1214 | 1225 |
| air06 | 6 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| bm23 | 82 | 78 | 82 | 79 | 131 | 109 | 170 | 82 |
| cracpb1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| diamond | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| egout | 2706 | 2706 | 2706 | 2708 | 2708 | 2707 | 2708 | 2715 |
| enigma | 472 | 467 | 549 | 573 | 612 | 684 | 649 | 761 |
| fixnet3 | 41 | 44 | 53 | 55 | 61 | 61 | 66 | 69 |
| fixnet4 | 114 | 114 | 114 | 116 | 121 | 108 | 156 | 154 |
| fixnet6 | 241 | 242 | 242 | 239 | 244 | 266 | 276 | 268 |
| khb05250 | 1648 | 1896 | 1650 | 1898 | 1897 | 1898 | 1901 | 1903 |
| l152lav | 545 | 599 | 590 | 564 | 584 | 557 | 584 | 559 |
| lp4l | 36 | 16 | 24 | 27 | 31 | 40 | 35 | 43 |
| lseu | 1540 | 1542 | 1538 | 1537 | 1539 | 1545 | 1497 | 1566 |
| misc01 | 192 | 192 | 192 | 192 | 192 | 193 | 196 | 193 |
| misc02 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 |
| misc03 | 217 | 287 | 320 | 323 | 326 | 330 | 326 | 332 |
| misc04 | 9 | 13 | 13 | 14 | 15 | 13 | 14 | 14 |
| misc05 | 199 | 204 | 202 | 211 | 176 | 175 | 156 | 150 |
| misc06 | 39 | 46 | 51 | 60 | 66 | 67 | 71 | 74 |
| misc07 | 13401 | 13401 | 13401 | 13401 | 13401 | 13401 | 13401 | 13402 |
| mod008 | 1627 | 1630 | 1635 | 1647 | 1635 | 1654 | 1627 | 1687 |
| mod010 | 28 | 23 | 31 | 34 | 40 | 45 | 45 | 48 |
| mod011* | 12102 | 10725 | 10725 | 10726 | 10726 | 10726 | 10727 | 10726 |
| mod013 | 280 | 280 | 281 | 280 | 285 | 268 | 281 | 300 |
| modglob$^+$ | 12406 | 12405 | 12407 | 12408 | 12408 | 12409 | 12410 | 12409 |
| p0033 | 198 | 202 | 199 | 200 | 286 | 207 | 207 | 206 |
| p0040 | 2 | 2 | 4 | 4 | 4 | 4 | 4 | 4 |
| p0201 | 251 | 259 | 262 | 155 | 169 | 179 | 176 | 198 |
| p0282 | 232 | 242 | 249 | 273 | 302 | 367 | 400 | 451 |
| p0291 | 42 | 44 | 42 | 64 | 70 | 67 | 80 | 72 |
| p0548 | 25963 | 25990 | 25969 | 25963 | 25967 | 25967 | 25967 | 25990 |
| p2756 | 6105 | 6153 | 6143 | 6157 | 6172 | 6180 | 6227 | 6235 |
| p6000 | 3521 | 3513 | 3391 | 2878 | 2874 | 2848 | 2836 | 2832 |
| pipex | 703 | 700 | 709 | 710 | 710 | 716 | 703 | 719 |
| rentacar | 18 | 30 | 39 | 44 | 47 | 50 | 43 | 50 |
| rgn | 1410 | 1410 | 1412 | 1414 | 1416 | 1381 | 1161 | 1151 |
| sample2 | 148 | 150 | 153 | 155 | 159 | 160 | 169 | 165 |
| sentoy | 200 | 201 | 209 | 198 | 218 | 308 | 231 | 377 |
| set1al$^+$ | 1156 | 1156 | 1158 | 1162 | 1167 | 1169 | 1174 | 1170 |
| set1ch$^+$ | 48 | 30 | 54 | 56 | 54 | 55 | 62 | 68 |
| set1cl$^+$ | 939 | 941 | 941 | 946 | 951 | 948 | 953 | 955 |
| stein9 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| stein15 | 37 | 37 | 37 | 37 | 37 | 37 | 37 | 37 |
| stein27 | 1175 | 1175 | 1175 | 1175 | 1175 | 1175 | 1182 | 1176 |
| stein45 | 15678 | 15678 | 15679 | 15679 | 15679 | 15679 | 15679 | 15679 |
| vmp1 | 155636 | 155636 | 155636 | 155636 | 155636 | 155636 | 155636 | 155636 |

* Run on an SP1 with 8 nodes.

+ Disjunctive cuts are activated.

Table 5. Speedup on $n$ SPARC20's

| Name | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ |
|---|---|---|---|---|---|---|---|
| air04 | 2.0 | 2.8 | 3.7 | 3.9 | 3.2 | 3.7 | 2.9 |
| air05 | 2.0 | 2.9 | 3.9 | 4.8 | 5.2 | 6.1 | 6.7 |
| l152lav | 1.7 | 2.7 | 3.7 | 4.4 | 5.4 | 6.2 | 6.9 |
| misc07 | 1.9 | 2.9 | 3.9 | 4.8 | 5.8 | 6.8 | 7.8 |
| modglob | 2.1 | 2.9 | 3.9 | 4.4 | 5.8 | 6.0 | 7.4 |
| mod011* | 1.8 | 2.8 | 3.7 | 4.6 | 4.9 | 6.8 | 8.9 |
| p0548 | 1.8 | 2.7 | 3.5 | 4.4 | 5.2 | 6.0 | 6.8 |
| p2756 | 2.0 | 3.0 | 4.0 | 5.0 | 5.9 | 6.8 | 7.7 |
| p6000 | 2.1 | 3.2 | 4.8 | 5.9 | 7.1 | 8.1 | 9.3 |
| set1ch | 2.2 | 2.5 | 2.8 | 5.6 | 8.2 | 6.5 | 11.3 |
| stein45 | 1.8 | 2.8 | 3.7 | 4.5 | 5.3 | 6.1 | 7.4 |
| vmp1 | 1.7 | 2.4 | 2.9 | 3.0 | 2.8 | 2.8 | 2.7 |

* Run on an SP1 with 8 nodes.

Table 6. Solution Status for *quasiunif2* and *teleicm*

| Name | Rows | Cols | 0/1 var. | LP Obj. | Cut Obj. | Optimal MIP Obj. |
|---|---|---|---|---|---|---|
| quasiunif2 | 240 | 521 | 56 | 11.72 | 62.64 | 65.67 |
| teleicm | 2672* | 7069* | 58 | 34818.42 | - | 39345 |

| Name | Runtime | Node Cnt. | machine type |
|---|---|---|---|
| quasiunif2 | 132277.5 | 349965 | SP2 thin node (16 nodes) |
| quasiunif2 | 374075.7 | 366420 | 8 SPARC20/61's. |
| teleicm | 306817.0 | 237802 | 8 SPARC20/61's. |

* Size after one presolve on CPLEX. Original size is 3276 rows and 9611 columns (63 0/1 variables).

# References


[1] D. Applegate, R. E. Bixby, V. Chvátal and W. Cook, "The Traveling Salesman Problem," in preparation (1995).

[2] E. Balas, "Disjunctive Programming: Cutting Planes from logical conditions," in *Nonlinear Programming 2* O. L. Mangasarian et al., eds., Academic Press, (1975b) 279–312.

[3] E. Balas, S. Ceria and G. Cornuéjols, "A Lift-and-Project Cutting Plane Algorithm for Mixed 0/1 Programs," MSRR 576 (1992), Carnegie Mellon University.

[4] E. Balas and E. Zemel, "facets of the knapsack polytope from minimal covers," *SIAM Journal of Applied Mathematics* 34 (1978) 119–148.

[5] R. E. Bixby, E.A. Boyd, S. S. Dadmehr and R. R. Indovina, "The MIPLIB Mixed Integer Programming Library," *COAL Bulletin* 22 (1993).

[6] R. E. Bixby and E. K. Lee, "Solving a Truck Dispatching Scheduling Problem Using Branch-and-Cut," TR93-37 (1993), Department of Computational and Applied Mathematics, Rice University.

[7] A. L. Brearley, G. Mitre and H. P. Williams, "Analysis of Mathematical programming problems prior to applying the simplex method," *Mathematical Programming* 5 (1975) 54–83.

[8] J. B. Carter and J. K. Bennett and W. Zwaenepoel, "Implementation and Performance of Munin," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, (1991) 152–164.

[9] T. L. Cannon and K. L. Hoffman, "Large-scaled 0/1 linear programming on distributed workstations," *Annals of Operations Research* 22 (1990) 181–217.

[10] H. Crowder, E. L. Johnson and M. Padberg, "Solving large-scale zero-one linear programming problems," *Operations Research*, 31 (1983) 803–834.

[11] D. Bienstock and O. Günlük, "Computational Experience with a Difficult Mixed-Integer Multicommodity Flow Problem," *Mathematical Programming* 68 (1995) 213– 237.

[12] J. Eckstein, "Parallel branch-and-bound algorithm for general integer programming on the CM-5," TMC-257 (1993), Thinking Machines Corporation.

[13] J. J. Forrest and D. Goldfarb, "Steepest-Edge Simplex Algorithms for Linear Programming," *Mathematical Programming* 57 (1992) 341–374.

[14] K. Gharachorloo, D. Lenoski and J. Laudon, P. Gibbons, A. Gupta and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, SIGARCH90 (1990) 15–26.

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

[16] P. Keleher, A. Cox and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (1992) 13–21.

[17] P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel, "TreadMarks: Distributed Memory on Standard Workstations and Operating Systems," *Proceedings of the 1994 Winter Usenix Conference*, (1994) 115–131

[18] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems* 4 (1989) Vol 7, 229–239.

[19] L. A. Oley and R. J. Sjoquist, "Automatic reformulation of mixed and pure integers models to reduce solution time in APEX IV," *SIGMAP Bulletin* 32 (1983).

[20] L. A. Wolsey, "Faces for a linear inequality in 0–1 variables," *Mathematical Programming* 8 (1975) 165–178.

[21] E. Zemel, "Easily computable facets of the knapsack polytope," *Mathematics of Operations Research* 14 (1989) 760–764.