Implementation and Computational Results for the Hierarchical Algorithm for Making Sparse Matrices Sparser

S. FRANK CHANG
GTE Laboratories, Inc.
and
S. THOMAS MCCORMICK
University of British Columbia

If A is the (sparse) coefficient matrix of linear-equality constraints, for what nonsingular T is $\hat{A} \equiv TA$ as sparse as possible, and how can it be efficiently computed? An efficient algorithm for this *Sparsity Problem* (SP) would be a valuable preprocessor for linearly constrained optimization problems. In a companion paper we developed a two-pass approach to solve SP called the *Hierarchical Algorithm*. In this paper we report on how we implemented the Hierarchical Algorithm into a code called HASP, and our computational experience in testing HASP on the NETLIB linear-programming problems. We found that HASP substantially outperformed a previous code for SP and that it produced a net savings in optimization time on the NETLIB problems. The results allow us to give guidelines for its use in practice.

Categories and Subject Descriptors: F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*computations on matrices*; G.1.6 [**Numerical Analysis**]: Optimization—*linear programming*; G.4 [**Mathematics of Computing**]: Mathematical Software—*algorithm analysis; efficiency*

General Terms: Algorithms, Measurement, Performance, Verification

INTRODUCTION

Optimization problems involving large, sparse, linearly constrained coefficient matrices arise in many application areas, such as electricity supply, circuit design, traffic flow, cash flow, and mechanical and civil engineering. To be efficient, algorithms designed for solving these problems must take advantage of their sparsity. As an example of the economics available with

Both authors were partially supported by NSF grants CDR-84-21402 and ECS-84-04350 and by ONR contract N0014-87-K0214, and most of this work was done while both authors were in the Department of Industrial Engineering and Operations Research at Columbia University. The second author was partially supported by an NSERC Operating Grant.
Authors' addresses: S. F. Chang, GTE Laboratories, 40 Sylvan Rd., Waltham, MA 02254; S. T. McCormick, Faculty of Commerce and Business Administration, University of British Columbia, Vancouver, BC V6T 1Y8 Canada.

sparsity, solving

$$Bx = b \tag{0.1}$$

for $B \in \mathbf{R}^{m \times m}$ is $O(m^3)$ if B is dense, but is empirically only $O(m^2)$ if B is sparse (see Duff [5, Tab. 3]). In fact, solving (0.1) seems to depend more on the number of nonzeros in B than on m.

This raises the question of whether it would be profitable to increase the sparsity of A as a preprocessing step in order to speed up optimizations involving A. To this end we define the

Sparsity Problem (SP). Given $A \in \mathbf{R}^{m \times m}$, $b \in \mathbf{R}^m$, which define constraints $Ax = b$, find a nonsingular $T \in \mathbf{R}^{m \times m}$ such that $\hat{A} \equiv TA$ is as sparse as possible.

In a companion paper (Chang and McCormick [3, 4]) we developed a new algorithm to solve SP called the *Hierarchical Algorithm (HA)*, and we proved that HA optimally solves SP, assuming the following "nondegeneracy" property (the submatrix of A indexed by rows in I, columns in J is denoted by A_{IJ}; the *term rank* of A_{IJ} is the size of the largest matching in the nonzeros in A_{IJ}):

Matching Property ((MP)). For any $I \subseteq \{1, \dots, m\}$, $J \subseteq \{1, \dots, n\}$, term rank A_{IJ} = rank A_{IJ}.

Very few real-life matrices satisfy (MP), but SP is NP-Hard without (MP) (see McCormick and Chang [3, 4]). We are thus using an (MP)-optimal algorithm as a heuristic for problems that do not satisfy (MP).

This paper reports on an implementation of HA called HASP (HA for SP). We cover the formal algorithm in Section 1. In Section 2, we introduce various implementation details of HASP. Section 3 reports on computational testing of HASP on the NETLIB linear-programming problems (see Gay [7]). Section 3.1 reports tests of HASP against a previous code for SP called SPARSER (see McCormick [12]). Section 3.2 compares the results of running the original versus the reduced LPs through MINOS 5.0 (see Murtagh and Saunders [15]). Finally, Section 4 concludes with recommendations for using HA in practice. More extensive analysis of the computational testing can be found in Chang [2].

1. THE HIERARCHICAL ALGORITHM

We recall here that the formal version of HA as given in Chang and McCormick [3, 4], but without proofs. HA is a two-pass algorithm. The first pass combinatorially computes the sparsity pattern of an optimal transformation matrix T using a bipartite matching subroutine. This subroutine yields the sparsity pattern of one row of T at a time, expressed as U_i = [the set of column indices which are nonzero in $T_{i\bullet}$]. Thus, it is called the

One-Row Algorithm (ORA) for row i:

(1) The input is a submatrix A_{RC} of A where C is contained in the set of columns which are zero in row i, and $i \notin R$.

(2) Perform a maximum matching by labelling starting with row nodes in the bipartite graph corresponding to A_{RC}; then the optimal solution U_i for row i of T is the set of labelled rows at optimality.

Define $R_i = U_i \cup \{i\}$. It turns out that $j \in R_i$ if and only if $R_j \subseteq R_i$ (Theorem 4.1 in Chang and McCormick [4]), and that this implies that the R_i induces a canonical grouping of the rows of A into *blocks* (i and j are in the same block if and only if $R_i = R_j$), as well as a (transitively closed) partial order on the blocks. If we order the rows in a linear order consistent with the partial order of the blocks, then the blocks induce the block-triangular structure of an optimal T. Each diagonal block of T is completely dense, and each subdiagonal block is either completely dense or zero. For example, if A is

$$\begin{array}{c}1\\2\\3\\4\\5\\6\\7\\8\\9\\10\end{array}\begin{pmatrix}
\times & 0 & \times & \times & 0 & \times & 0 & 0 & 0 & 0 & 0 & \times & 0 & 0 & 0 & 0\\
\times & \times & \times & 0 & 0 & 0 & \times & 0 & \times & \times & \times & 0 & 0 & 0 & 0 & 0\\
\times & \times & \times & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
\times & 0 & \times & \times & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
\times & \times & \times & \times & 0 & 0 & \times & 0 & \times & \times & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & \times & \times & \times & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
\times & \times & 0 & \times & 0 & 0 & \times & \times & 0 & \times & \times & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & \times & \times & \times & 0 & \times & 0 & \times & \times & \times & \times\\
0 & \times & \times & \times & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & \times & 0 & 0 & 0 & \times & 0 & \times & \times & \times & \times & 0
\end{pmatrix}, \qquad (1.1)$$

then (after permuting into block order) the optimal T looks like

$$\begin{array}{c|cccccccccc}
 & 3 & 9 & 4 & 6 & 1 & 2 & 7 & 5 & 10 & 8\\
3 & \times & \times & \times & & & & & & & \\
9 & \times & \times & \times & & & & & & & \\
4 & \times & \times & \times & & & & & & & \\
6 & \times & \times & \times & \times & & & & & & \\
1 & \times & \times & \times & \times & \times & & & & & \\
2 & \times & \times & \times & & & \times & \times & \times & & \\
7 & \times & \times & \times & & & \times & \times & \times & & \\
5 & \times & \times & \times & & & \times & \times & \times & & \\
10 & & & & & & & & & \times & \\
8 & \times & \times & \times & & & \times & \times & \times & \times & \times
\end{array}. \qquad (1.2)$$

This block-triangular form is called the *SP decomposition* of T.

Rather than compute the rows of T one by one via the One-Row algorithm, HA uses the above structure of T to speed up the computations. A further speedup occurs because sizes of the submatrices passed to the One-Row matching routine are reduced; the description below uses the notation that $C(R)$ equals the set of columns with a nonzero in some row in R, and $C(i) \equiv C(\{i\})$. The first row discovered in each block is called a *block leader*. The other rows in a block are called the *associates* of the block leader. HA uses an array *ORDER* of length at most m to represent an ordered list of block leaders: $ORDER(k) = i$ means that row i is the leader of the kth block. We use a linked list *BMEM* of length m to store all associates: if the next associate in i's block is row j, then $BMEM(i) = j$, whereas if i is the last associate in its block, $BMEM(i) = 0$.

The linear order of the block leaders in the array *ORDER* is the same as the order of the corresponding diagonal blocks in T's block-triangular decomposition. Having this order on the rows will help execute numerical steps more efficiently in Pass 2. We obtain this (nonunique) linear order as HA progresses by recording the sequence of block leaders leaving the stack. We now can write the combinatorial part of HA as follows:

Combinatorial Hierarchical Algorithm (Pass 1):

Initialize the block counter k and the list *BMEM* to 0.
Let $R_0 = \{1, 2, \ldots, m\}$. Push 0 onto *STACK*.
While *STACK* is not empty, let i be the top element **do**
  while there exists an unprocessed row $j \in R_i$ **do**
    compute R_j by ORA on the submatrix $A_{R_i \setminus \{j\}, C(R_i) \setminus C(j)}$;
    if $|R_j| < |R_i|$, **then** {i's block further decomposes}
      push row j onto the stack; {j becomes the leader of a new block contained in R_i}
      save R_j data;
      set $i := j$;
    else
      insert j into BMEM with i pointing to j; {register j as an associate in i's block}
    endif
  done
  remove i from *STACK*;
  $k := k + 1$;
  $ORDER(k) := i$; {i's block is the kth and it will not further decompose}
done
end.

Let n_B denote the total number of blocks. In example (1.1), when HA stops, we will obtain

	1	2	3	4	5	n_B (= 6)
ORDER:	3	6	1	2	10	8

	1	2	3	4	5	6	7	8	9	m (= 10)
BMEM:	0	7	9	0	0	0	5	0	4	0

which tells us that the 10 rows of A are decomposed into six ordered blocks. The contents of each block can be sequenced easily by scanning the list BMEM starting from the block leader:

$$B_1 = \{3, 9, 4\}, \quad B_2 = \{6\}, \quad B_3 = \{1\}, \quad B_4 = \{2, 7, 5\}, \quad B_5 = \{10\}, \quad B_6 = \{8\}.$$

Note that this agrees with the block-triangular decomposition of T in (1.2). Later in Pass 2, we shall do numerical processing on blocks in reverse order, i.e., the bottom block of rows will get reduced first.

The set of rows used in the numerical processing of each block is given by the $\{R_i\}$ data; we use this data in both Pass 1 and Pass 2, so we need to allocate some space to store it, though the space can be gradually salvaged as Pass 2 proceeds. But in some applications, e.g., the Newton-Raphson method for nonlinear problems, the same sparsity pattern will be used over and over again with changing coefficients, so then it *is* necessary to keep $\{R_i\}$ data stored throughout the computation. In order to keep the storage of $\{R_i\}$ data compact and easily accessible, we append R_i to an array *TR* only for block leaders i. We use two pointer arrays to identify the beginning and the end of each R_i in *TR*.

In Pass 2 we use the sparsity pattern of T as represented by the R_i as a road map to do eliminations on A to get $\hat{A}$. The elimination is performed blockwise, thus is called *block elimination*. We are essentially doing blockwise partial Gaussian elimination of A.

Before we begin block elimination, we first find a well-conditioned basis of A; all the pivots of the block elimination are to be selected within the basis. This task is handled by `MA28`, a package of sparse matrix LU-factorization and linear-equation-solving routines written by Duff at Harwell (see Duff [5]), which can factorize a rectangular matrix. Let G denote the set of columns in the chosen basis. To understand what Pass 2 does, consider Figure 1.

Here "F" represents a full (dense) submatrix; "0" represents a zero submatrix; and "$*$" represents an arbitrary submatrix. Figure 1 assumes that the rows of A are permuted in the same order as the rows of T, and the columns of A in the same order as the pivot choices in G. The block eliminations consist of two types of operations.

(1) eliminate each subdiagonal block whose corresponding block in T is dense, and

(2) transform each diagonal block into an identity.

During the processing of each block of rows, operation (1) *must* precede operation (2); otherwise some nonzeros might fill in the places eliminated by operation (2) while performing operation (1).

The numerical processing starts from the last block, i.e., B_{n_B}, and proceeds backward. Let B_k be the current block being processed, and $j = ORDER(k)$. By scanning R_j once, we can easily find the set of rows U_k to be used for processing B_k. Now pass the submatrix $A_{U_k G}$ to `MA28` to find a subset of columns C_k such that the square submatrix $A_{U_k C_k}$ is well conditioned and nonsingular. For each $i \in B_k$, solve the system

$$\lambda^t A_{U_k C_k} = A_{i C_k}, \tag{1.3}$$

and set

$$\tilde{A}_{i\bullet} = A_{i\bullet} - \lambda^t A_{U_k \bullet}.$$

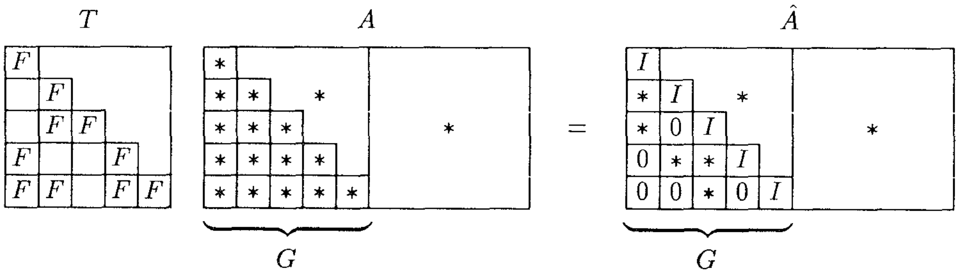

Figure 1

This is operation (1) for processing block B_k. Note that (1.3) is a single LU-factorization which $|B_k|$ solves. Next we pass the submatrix $\tilde{A}_{B_k, G \setminus C_k}$ to MA28 to find a subset of columns $\tilde{C}_k$ such that $\tilde{A}_{B_k \tilde{C}_k}$ is a well-conditioned and nonsingular submatrix. Then for each $i \in B_k$, if i is the $p(i)$th in the pivot sequence, we solve the system

$$\lambda^t \tilde{A}_{B_k \tilde{C}_k} = e_{p(i)}, \tag{1.4}$$

where $e_{p(i)}$ is the $p(i)$th unit vector of length $|B_k|$, and set

$$\hat{A}_{i\bullet} = \lambda^t \tilde{A}_{B_k \bullet}.$$

This completes operation (2) for B_k. In terms of submatrices, if A originally looks like

$$\begin{array}{c|c|c|c|} \multicolumn{1}{c}{} & \multicolumn{2}{c}{\overbrace{\hphantom{C_k \quad \tilde{C}_k}}^{G}} & \multicolumn{1}{c}{} \\ \multicolumn{1}{c}{} & \multicolumn{1}{c}{C_k} & \multicolumn{1}{c}{\tilde{C}_k} & \multicolumn{1}{c}{} \\ \cline{2-4} U_k & Q & R & W \\ \cline{2-4} & * & * & * \\ \cline{2-4} B_k & S & T & V \\ \cline{2-4} \end{array}$$

then the new B_k will become

$$\begin{array}{c|c|c|c|} \multicolumn{1}{c}{} & \multicolumn{1}{c}{C_k} & \multicolumn{1}{c}{\tilde{C}_k} & \multicolumn{1}{c}{} \\ \cline{2-4} B_k & 0 & I & (T - SQ^{-1}R)^{-1}(V - SQ^{-1}W) \\ \cline{2-4} \end{array}$$

Now that block B_k has been settled, it will not be used for processing any other block above it, so it can be ignored in future computation. Also, since each column can only serve as a pivot once, $\tilde{C}_k$ will not be used either. Then set $k \leftarrow k - 1$ and repeat the same procedure with the smaller matrix.

Note that when row $i \in B_k$ is being processed, we can reuse the storage space for $A_{i\bullet}$ to store the updates $\tilde{A}_{i\bullet}$ and $\hat{A}_{i\bullet}$ (since we know that they have fewer nonzeros than $A_{i\bullet}$). Thus we do not need to keep a copy of the original input coefficient matrix, which saves some working space. The major computational effort of Pass 2 is spent in (1.3) and (1.4), namely, factoring and solving linear systems.

2. IMPLEMENTATION TRICKS

The analysis in Chang and McCormick [4] assumes that A has full row rank, that (MP) is satisfied, and that the constraints are in the form $Ax = b$ (i.e., all equalities) for convenience in deriving theoretical properties of the algorithm. However, most real data violate one or more of these assumptions. In this section we consider how to deal with such matrices when implementing HA. In addition, some practical techniques for speeding up the algorithm are also discussed.

Warm-start matching and restricted columns. These two techniques were first used in McCormick [11] and have proven useful in speeding up both the combinatorial and numerical processing. We have adopted them in the development of HA.

The major work in the combinatorial processing involves the computation of maximum cardinality matchings; a matching routine is called for each row. Warm-start matching speeds this up by first finding a one-time fixed matching on all the rows. When processing A_{RC}, the part of the fixed matching appearing in A_{RC} is used as an initial matching. Then it is augmented into a maximum matching in A_{RC}.

The major work in numerical processing involves computing *LU*-factorizations. Recall that an initial LU factorization on all of A gives us the set of **good** columns G such that $A_{\bullet G}$ is square, nonsingular, and well conditioned. The **restricted column option** is to restrict all future *LU*-factorizations for processing a block B_k (during operations (1) and (2)) to be chosen within the submatrix $A_{B_k G}$. If A has many more columns than rows, this option greatly reduces the size of the rectangular matrix within which a nonsingular submatrix is to be found. McCormick [12] found that this led to a large savings in time.

Relaxing the full row rank assumption. Dependent rows are of no use in the preprocessing step and in the optimization procedure that follows. Although their presence does not hinder our algorithm, from an efficiency point of view it seems to be a good idea to detect and remove dependent rows first. Indeed, removing dependent rows is a natural by-product of the initial LU-factorization for the restricted-column option anyway. Only equality rows need to be classified into dependent and independent rows. This is done by performing an initial LU factorization on the submatrix of equality rows. McCormick [12] shows that under (MP) the same number of final nonzeros will result no matter which subset of dependent rows is deleted.

Dealing with inequality rows and matrices without (*MP*). Inequality rows are certainly independent after adding slacks to them. But they (or any other row containing a nonzero which is the only nonzero in its column) will not be used by HA to reduce any other rows. This is because such a row can always be matched to its slack entry, and there are no other nonzeros in the slack column through which the inequality row could get labelled. Since such a row is never labelled, it never appears in any U_i. In particular, HA is incapable of reducing matrices without some equality rows.

Inequality rows will potentially be reduced but cannot be used by HA in processing other rows, while dependent rows will neither be processed nor be used. We want to exclude these two types of rows from being considered as possible used rows when processing a row i. Adding slacks before applying HA would be too slow, and we still need to identify dependent rows anyway. We handle this problem as follows. When the input stage finishes, we save the row-type information in an indicator *FIXRTC* by setting

$$FIXRTC(r) = \begin{cases} 0, & \text{if } r \text{ is an equality row;} \\ -1, & \text{if } r \text{ is an inequality row;} \\ -2, & \text{if } r \text{ is a free row.} \end{cases}$$

After an initial LU factorization is done, we further distinguish dependent rows by setting the *FIXRTC* values to -3. (The values of independent equality rows remain at 0.) When the one-time fixed matching is found for all independent equality rows, replace their *FIXRTC* values with the indices of their matched columns. Now *FIXRTC* serves two purposes: it identifies row types and also saves the fixed matching.

Dropping the (MP) assumption will not create any singularity problems in HA, since the pivots for Gauss-Jordan elimination are chosen by numerical considerations. Recall that when processing block B_k numerically, we need to find two nonsingular submatrices to form pivot blocks for operations (1) and (2). Note that U_k, as a set of used rows, must contain only independent equality rows. Also, U_k has not been processed before (remember that the numerical processing of HA is performed from the bottom up; once a block is processed, then it will not be used anymore). Thus, the existence of a nonsingular submatrix in $A_{U_k G}$ is assured. As for the existence of $\tilde{A}_{B_k \tilde{C}_k}$, we consider two cases: B_k is either a set of independent equality row(s) or a singleton inequality row. For the former, for the same reason as $A_{U_k G}$, $A_{B_k G}$ must have full row rank before operation (1). After operation (1), $\tilde{A}_{B_k G}$ still has full row rank, since operation (1) premultiplies $A_{B_k G}$ by a nonsingular matrix. Now $\tilde{A}_{B_k, G \setminus C_k}$ must have full row rank; otherwise $\tilde{A}_{B_k G}$ cannot have full row rank either, since $\tilde{A}_{B_k C_k}$ is 0. The second case (B_k containing only one row), does not need operation (2); hence we do not have to worry about finding a nonsingular $\tilde{A}_{B_k \tilde{C}_k}$.

A final trick for applying HA to practical problems concerns what we call **manual pivoting** in Pass 2. Suppose a block B_k is being processed, and it uses only one row r for elimination (i.e., $U_k = \{r\}$). Then we do not need a full-blown LU-factorization for finding a pivot block; a pivot element is all we need in this case. Any nonzero element in row r can be used as a pivot element, and no fill-in will occur when a multiple of row r is added to a row in B_k. We use as the pivot the first nonzero element whose absolute value is greater than a threshold parameter. Since (1) row r has not been processed (reduced) itself and (2) the rows processed by row r will not be used later, manual pivoting should not cause too much numerical instability. Note that the choice of pivot does not have to be consistent with the fixed-column technique, so that we do not need to spend time checking whether the chosen column is in G. Manual pivoting saves the considerable overhead of data moving and checking involved in an LU factorization.

Basic program modules. `HASP` consists of six major program modules: `ALLOC`, `MPSIN`, `SPINIT`, `PASS1`, `PASS2`, and `MPSOUT`. `ALLOC`, `MPSIN`, `SPINIT`, and `MPSOUT` are utility routines that were adopted from McCormick's `SPARSER` with minor modifications. The function of each module is described below.

(1) `ALLOC`. the driver routine for the whole system. It manages the following things:

(a) Reads and sets up the parameters that control the execution of other subroutines. These parameters should be provided by the user in a specification file.
(b) Allocates the core space passed to it from the main program according to the data types and sizes of the arrays used in each subroutine.
(c) Calls other subroutines.

(2) `MPSIN`. The input routine (originally adapted from `MINOS` 5.0). It reads data in the industry standard MPS format (with rows, columns, RHS, ranges, and bounds information).

(3) `SPINIT`. An initialization routine in the system. It uses `MA28` subroutines (see Duff [5]) from the HARWELL Library to find an initial LU factorization in the equality rows of the whole matrix, thus identifying dependent and independent rows. As mentioned before, the dependent rows are removed from the matrix. The set G of column indices of this initial basis is saved and will be used in `PASS2` (the numerical reduction step) as the range for choosing LU factors from rectangular systems.

(4) `PASS1`. The combinatorial computation routine described in Section 1. Given the sparsity pattern of an input matrix, `PASS1` hierarchically decomposes the rows into blocks and gathers necessary information about the SP-decomposition of the transformation matrix T. In particular, it identifies the rows that can be processed together as a block in `PASS2`, the rows to be used for reducing a block of rows, and, most importantly, the order of blocks in numerical processing.

Two specialized subroutines for computing maximum bipartite matchings are called by `PASS1`. `BP` is called only once to find an initial bipartite matching in the set of independent equality rows, which is used as a warm-start matching. Then, for each row processed by HA, subroutine `BP1` augments the induced part of the fixed matching to optimality and returns the set of labelled rows to `PASS1`. Both `BP` and `BP1` are adapted from the bipartite-matching code BCM described in Chang and McCormick [3]. This code is a modified depth-first search labeling algorithm with a lookahead technique which outperformed other matching codes in computational testing.

(5) PASS2. The numerical computation routine. Once PASS1 has figured out the combinatorial structure of the input sparsity pattern and produced the SP-decomposition of T, then PASS2 will process the matrix data to produce a sparser equivalent matrix using blockwise partial Gauss-Jordan elimination as described in Section 1. The sequence of blocks is provided by PASS1. The bottom block of rows in the SP-decomposition will get processed first, and once processed will not be touched again.

The major work involved in the elimination is again done by MA28 subroutines which can perform LU-factorizations in rectangular systems and then find solutions for different right-hand-side vectors. MA28 also monitors stability to ensure a reliable factorization, so that the square matrix found in a rectangular system is fairly well conditioned.

(6) MPSOUT. The output routine. It puts the reduced matrix data into MPS format and writes it to a disk file.

Control Parameters and Options

EPS. The zero tolerance for numerical calculations in MA28 and PASS2.

AIJTOL. The threshold for zero elements in MPSIN.

U. The MA28 factor that determines the trade-off between sparsity and stability. U $= 1.0$ gives partial pivoting for numerical stability, while U $= 0.0$ does not check multipliers at all with pivots chosen purely on the Markowitz sparsity criterion.

EXPAN. The storage expansion factor used in setting up the size of the work space for performing LU-factorizations.

The above 4 parameters can be specified in a specification file. In all tests reported in later sections, we use these values: EPS $= 1.0\text{D} - 8$, AIJTOL $= 1.0\text{D} - 6$, U $= 0.1$, EXPAN $= 2$, as recommended in Duff [5] and used in McCormick [12].

Another two MA28 options regarding how LU-factorizations should be performed are the following:

LBLOCK. With default value TRUE in MA28. If TRUE, the matrix is first permuted to block-lower-triangular form. This option was found to be inefficient by McCormick [12]; thus we set LBLOCK = FALSE in all test runs.

MTYPE. Controls whether $Bx = b$ or $x^tB = b^t$ is the system to be solved when calling MA28. The computational testing in McCormick [12] shows that factoring submatrices of A in their normal (as opposed to the transposed) form appears to be faster for running SPARSER. Thus, the same option was used for all HASP tests.

3. COMPUTATIONAL RESULTS

The experimental implementation of HA is a FORTRAN program called HASP. We first compare HASP to SPARSER to evaluate its efficiency. Then we run MINOS 5.0 (Murtagh and Saunders [15]), a state-of-the-art simplex method package, on both A and the reduced matrix $\hat{A}$ to see whether MINOS running times are reduced. The NETLIB linear-programming problems (see Gay [7]) were used as the test set. The computer experiments were all done on a Sun-3/60 machine.

3.1 Comparing the Hierarchical Algorithm to the Sequential Algorithm

Both HASP and SPARSER were coded in FORTRAN with double-precision arithmetic. The Sun f77 FORTRAN compiler was used with −O3 option and the default floating-point code generation option. The CPU times spent in major segments as well as the total time were recorded as separate items.

A total of 68 linear programs together with their characteristics are listed in Table I. Columns 2 and 3 are the numbers of relevant rows *NRR* and relevant columns *NRC*. We call the rows and columns in A the *relevant* rows and columns, since only they are relevant to the sparseness. Right-hand sides and objective functions are not relevant. Columns 4 and 5 list the number of (relevant) nonzeros *NRNZ* and the initial density *IDEN* of A, where $IDEN = 100 \times NRNZ/(NRR \times N)$. Columns 6 and 7 show the number of equality rows *NEQR* and the number of equality nonzeros *NEQNZ* in A. Column 8 shows the percentage of equality rows *PEQR* in A, i.e., $PEQR = 100 \times (NEQR/NRR)$. Lastly, column 9 gives the number of dependent rows *NDP*. The difference between *NEQR* and *NDP* then gives some indication of the potential for making the matrix sparser. The characteristics of these problems relevant to computational performance of linear-programming algorithms can be found in Lustig [10].

The two algorithms both delete the same number of nonzeros after the pure combinatorial processing is done on all test problems, and 51 problems do become sparser. Only 11 problems have different reductions by the two algorithms after the numerical processing (lucky cancellations often appear during numerical processing, which improves the combinatorial reduction, but in an unpredictable way). The difference is not significant and appears to favor neither code.

The distribution of density reductions is summarized in Table II. In each range the averages of *NRNZ*, *IDEN*, *NEQR*, *NEQNZ*, and *PEQR* for those problems processed by HASP are also listed.

It appears that those test problems with relatively smaller and denser coefficient matrices and with higher percentages of equality rows tend to have more density reduction. The correlation of coefficients of *IDEN* and *PEQR* with Density Reduction (or % Redn in NZ) were .30 and .19 respectively.

In Table III we compare the speeds of HASP and SPARSER in average time spent on each problem in five runs. The time spent in ALLOC + MPSIN + SPINIT is nearly identical for the two codes, so we give a single, combined time,

Table I. NETLIB Problem Characteristics

Problem name	Relevant rows (NRR)	Relevant columns (N)	Relevant nonzeros (NRNZ)	Initial density (IDEN)	Equality rows (NEQR)	Equality nonzeros (NEQNZ)	% Eq rows (PEQR)	Depend rows (NDP)
25FV47	821	1571	10400	0 81	516	5908	62 85	1
ADLITTLE	56	97	383	7 05	15	173	26 79	
AFIRO	27	32	83	9 61	8	34	29 63	
AGG	488	163	2410	3 03	36	288	7 38	
AGG2	516	302	5284	3 39	60	518	11 63	
AGG3	516	302	4300	2 76	60	534	11 63	
BANDM	305	472	2494	1 73	305	2494	100 00	
BEACONFD	173	262	3375	7 45	140	3309	80 92	
BLEND	74	83	491	7 99	43	298	58 11	
BOEING1	350	384	3485	2 59	9	168	2 57	
BOEING2	166	143	1196	5 04	4	56	2 41	
BORE3D	233	315	1429	1 95	214	1370	91 85	2
BRANDY	220	249	2148	3 92	166	1784	75 45	27
CAPRI	271	353	1767	1 85	142	1072	52 40	
CZPROB	929	3523	10669	0 33	890	7024	95 80	
E226	223	282	2578	4 10	33	938	14 80	
ETAMACRO	400	688	2409	0 88	272	1374	68 00	
FFFFF800	524	854	6227	1 39	350	4775	66 79	
FINNIS	497	614	2310	0 76	47	134	9 46	
FORPLAN	161	421	4563	6 73	90	3775	55 90	
GANGES	1309	1681	6912	0 31	1284	6612	98 09	
GFRD-PNC	616	1092	2377	0 35	548	2182	88 96	
GREENBEA	2392	5405	30877	0 24	2199	22598	91 93	3
GREENBEB	2392	5405	30877	0 24	2199	22598	91 93	3
GROW15	300	645	5620	2 90	300	5620	100 00	
GROW22	440	946	8252	1 98	440	8252	100 00	
GROW7	140	301	2612	6 20	140	2612	100 00	
NESM	662	2923	13288	0 69	480	12708	72 51	
PEROLD	625	1376	6018	0 70	495	4388	79 20	
PILOT	1441	3652	43159	0 82	233	3689	16 17	
PILOT JA	940	1988	14698	0 79	661	8746	70 32	
PILOT WE	722	2789	9126	0 45	583	7856	80 75	
PILOT4	410	1000	5141	1 25	287	2577	70 00	
PILOTNOV	975	2172	13057	0 62	701	10225	71 90	
RECIPE	91	180	663	4 05	67	351	73 63	
SC105	105	103	280	2 59	45	122	42 86	
SC205	205	203	551	1 32	91	249	44 39	
SCAGR25	471	500	1554	0 66	300	1334	63 69	
SCAGR7	129	140	420	2 33	84	362	65 12	
SCFXM1	330	457	2389	1 58	187	1467	56 67	
SCFXM2	660	914	5183	0 86	374	2939	56 67	
SCFXM3	990	1371	7777	0 57	561	4411	56 67	
SCRS8	490	1169	3182	0 56	384	2576	78 37	
SCSD1	77	760	2388	4 08	77	2388	100 00	
SCSD6	147	1350	4316	2 17	147	4316	100 00	
SCSD8	397	2750	8584	0 79	397	8584	100 00	
SCTAP1	300	480	1692	1 17	120	360	40 00	
SCTAP2	1090	1880	6714	0 33	470	1410	43 12	
SCTAP3	1480	2480	8874	0 24	620	1860	41 89	
SEBA	515	1028	4352	0 82	507	4330	98 45	
SHARE1B	117	225	1151	4 37	89	891	76 07	
SHARE2B	96	79	694	9 15	13	84	13 54	
SHELL	536	1775	3556	0 37	534	3550	99 63	1
SHIP04L	402	2118	6332	0 74	354	4158	88 06	42
SHIP04S	402	1458	4352	0 74	354	2838	88 06	42
SHIP08L	778	4283	12802	0 38	698	8411	89 72	66
SHIP08S	778	2387	7114	0 38	698	4619	89 72	66
SHIP12L	1151	5427	16170	0 26	1045	10635	90 79	109
SHIP12S	1151	2763	8178	0 26	1045	5307	90 79	109
SIERRA	1227	2036	7302	0 29	528	3973	43 03	10
STAIR	356	467	3856	2 32	209	1374	58 71	
STANDATA	359	1075	3031	0 79	160	2128	44 57	
STANDGUB	361	1184	3139	0 73	162	2236	44 88	1
STANDMPS	467	1075	3679	0 73	268	2776	57 39	
STOCFOR1	117	111	447	3 44	63	273	53 85	
STOCFOR2	2157	2031	8343	0 19	1143	4929	52 99	
STOCFOR3	16675	15695	64875	0 02	8829	38403	52 95	
VTP BASE	198	203	908	2 26	55	500	27 78	

Table II. Density Reduction Distribution and Problem Attributes

Density reduction range (in %)	$[0,1)$	$[1,5)$	$[5,10)$	$[10,20)$	$[20, max]^{\dagger}$
Problems processed by SPARSER	31	16	12	7	2
Problems processed by HASP	31	16	13	6	2
Average NRNZ	8398 5	9271 1	3076 5	3384 8	2934 5
Average IDEN	1 91	1 53	2 38	3 13	4 59
Average NEQR	576 0	639 4	250 5	473 5	222 5
Average NEQNZ	4613 1	6285 1	1938 5	2710 3	2901 5
Average PEQR	56 99	68 76	57 07	81 73	90 46

† Note maximum density reduction = 65 45% in BEACONFD

Table III. CPU Times Comparison (Sun-3/60 sec)

Problem name	AMS	COMB_PA	PASS1	SPARSR	PASS1+2	SA adj total	HA adj total	Ratio_1 (%)	Ratio_2 (%)	Ratio_+ (%)	Ratio_A (%)
25FV47	20 10	4 38	2 47	20 06	8 50	21 62	10 01	56 39	38 46	42 37	46 30
ADLITTLE	1 13	0 06	0 01	0 11	0 06	0 19	0 10	16 67	100 00	54 54	52 63
AFIRO	0 41	0 01	0 01	0 02	0 01	0 04	0 02	100 00	-	-	50 00
AGG	5 46	0 90	0 16	0 97	0 19	1 16	0 39	17 78	43 86	19 59	33 62
AGG2	8 91	1 28	0 25	1 64	0 29	1 96	0 59	19 53	11 11	17 68	30 10
AGG3	8 96	1 24	0 26	1 39	0 28	1 74	0 61	20 97	13 33	20 14	35 06
BANDM	6 65	1 00	0 45	10 26	7 92	11 90	9 55	45 00	80 67	77 19	80 25
BEACONFD	6 64	0 80	0 21	3 92	2 97	4 67	3 70	26 25	88 46	75 76	79 23
BLEND	1 19	0 10	0 05	0 39	0 16	0 50	0 27	50 00	37 93	41 03	54 00
BOEING1	6 78	0 40	0 30	0 46	0 36	0 56	0 49	75 00	100 00	85 71	87 50
BOEING2	2 58	0 08	0 07	0 10	0 10	0 17	0 20	87 50	150 00	100 00	117 65
BORE3D	4 78	0 54	0 29	3 61	2 33	5 54	4 25	53 70	66 45	64 54	76 71
BRANDY	4 77	0 62	0 21	3 75	2 34	4 70	3 30	33 87	68 05	62 40	70 21
CAPRI	4 39	0 60	0 32	3 17	2 52	3 98	3 35	53 33	85 60	79 50	84 17
CZPROB	25 82	3 96	4 15	8 18	7 31	10 37	9 49	104 80	74 88	89 36	91 51
E226	6 11	0 38	0 16	0 66	0 42	1 80	1 58	42 10	92 86	63 64	87 78
ETAMACRO	4 99	0 92	0 64	1 04	0 71	1 50	1 19	69 57	58 33	68 27	79 33
FFFFF800	13 84	2 28	1 12	11 28	6 22	13 96	8 83	49 12	56 67	55 14	63 25
FINNIS	5 52	0 62	0 62	0 63	0 62	0 81	0 77	100 00	-	-	95 06
FORPLAN	8 78	0 38	0 17	0 87	0 61	1 61	1 32	44 74	89 80	70 11	81 99
GANGES	15 48	9 26	6 34	69 21	41 40	72 49	44 74	68 47	58 48	59 82	61 72
GFRD-PNC	23 16	1 98	1 59	4 84	2 66	21 09	18 88	80 30	37 41	54 96	89 52
GREENBEA	92 55	30 32	24 65	188 73	82 72	230 57	124 59	81 30	36 66	43 83	54 04
GREENBEB	93 74	30 34	25 29	193 56	84 80	235 20	126 45	83 36	36 46	43 81	53 76
GROW15	10 95	0 88	0 59	0 88	0 59	2 08	1 76	67 04	-	-	84 61
GROW22	16 02	1 68	1 14	1 68	1 14	3 43	2 90	67 86	-	-	84 55
GROW7	5 25	0 28	0 18	0 28	0 18	0 83	0 74	64 29	-	-	89 16
NESM	41 78	3 26	2 46	3 23	2 46	20 44	19 62	75 46	-	-	95 99
PEROLD	36 93	2 54	1 72	7 95	1 99	33 51	27 50	67 72	4 99	25 03	82 07
PILOT	76 36	8 36	6 82	9 40	6 88	13 98	11 46	81 58	5 77	73 19	81 97
PILOT JA	38 29	5 98	3 57	20 97	7 43	33 00	19 65	59 70	25 75	35 43	59 54
PILOT WE	26 76	3 80	2 74	7 39	2 79	16 40	11 77	72 11	1 39	37 75	71 77
PILOT4	19 38	1 56	0 78	8 65	1 63	18 13	11 26	50 00	12 00	18 84	62 11
PILOTNOV	40 55	6 02	3 93	14 40	4 42	31 61	21 53	65 28	5 85	30 69	68 11
RECIPE	1 63	0 10	0 04	1 02	0 10	1 19	0 27	40 00	6 52	9 80	22 69
SC105	0 85	0 08	0 05	0 18	0 24	0 27	0 31	62 50	190 00	133 33	114 82
SC205	1 48	0 30	0 17	0 63	0 88	0 76	1 02	56 67	215 15	139 68	134 21
SCAGR25	4 25	1 20	0 80	4 63	3 76	5 06	4 17	66 67	86 30	81 21	82 41
SCAGR7	1 35	0 16	0 08	0 52	0 36	0 68	0 50	50 00	77 78	69 23	73 53
SCFXM1	5 23	0 80	0 40	3 59	1 61	4 01	2 04	50 00	43 36	44 85	50 87
SCFXM2	10 20	2 52	1 46	13 01	5 54	13 84	6 38	57 94	38 89	42 58	46 10
SCFXM3	14 93	5 30	3 12	28 24	11 63	29 50	12 85	58 87	37 10	41 18	43 56
SCRS8	8 95	1 66	1 08	10 21	5 46	11 65	6 92	65 06	51 23	53 48	59 40
SCSD1	6 75	0 10	0 06	0 11	0 06	0 66	0 63	60 00	-	-	95 46
SCSD6	12 22	0 32	0 21	0 31	0 21	1 29	1 19	65 62	-	-	92 25
SCSD8	23 52	1 66	1 16	1 62	1 16	3 58	3 18	69 88	-	-	88 83
SCTAP1	4 25	0 36	0 30	0 36	0 30	0 58	0 51	83 33	-	-	87 93
SCTAP2	16 12	3 98	3 91	3 97	3 91	4 78	4 67	98 24	-	-	97 70
SCTAP3	20 71	7 50	6 90	7 30	6 89	8 35	7 86	92 00	-	-	94 13
SEBA	13 31	1 74	1 33	1 75	1 33	6 90	6 58	76 44	-	-	95 36
SHARE1B	3 02	0 24	0 08	1 76	1 30	2 47	2 02	33 33	80 26	73 86	81 78
SHARE2B	1 50	0 12	0 03	0 38	0 13	0 45	0 17	25 00	38 46	34 21	37 78
SHELL	217 22	1 60	1 53	1 61	1 53	209 81	209 72	95 62	-	-	99 96
SHIP04L	14 79	0 96	0 86	1 46	0 90	2 66	2 07	89 58	8 00	61 64	77 82
SHIP04S	10 15	0 90	0 62	2 94	2 68	3 82	3 52	68 89	100 98	91 16	92 15
SHIP08L	28 92	3 64	3 24	9 66	3 39	12 04	5 75	89 01	2 49	35 09	47 76
SHIP08S	16 57	2 60	1 90	10 72	9 80	12 20	11 26	73 08	97 29	91 42	92 29
SHIP12L	37 11	6 90	5 91	24 71	18 97	27 83	22 01	85 65	73 33	76 77	79 09
SHIP12S	19 57	4 54	3 18	22 28	20 49	24 07	22 26	70 04	97 58	91 97	92 48
SIERRA	167 45	6 48	4 24	8 41	4 28	158 24	154 98	65 43	2 07	50 89	97 94
STAIR	7 19	0 88	0 55	0 88	0 55	1 30	0 99	62 50	-	-	76 15
STANDATA	6 30	0 68	0 56	0 68	0 56	1 29	1 19	82 35	-	-	92 25
STANDGUB	6 71	0 70	0 58	0 68	0 58	1 32	1 20	82 86	-	-	90 91
STANDMPS	7 53	1 10	0 83	1 32	1 08	2 08	1 87	75 46	113 64	81 82	89 90
STOCFOR1	1 17	0 14	0 06	0 42	0 22	0 54	0 31	42 86	57 19	52 38	57 41
STOCFOR2	22 85	23 12	14 15	33 27	20 69	39 05	26 43	61 20	64 43	62 19	67 68
STOCFOR3	430 64	1359 64	827 00	1686 23	1119 78	1904 12	1337 56	60 83	89 65	66 41	70 25
VTP BASE	2 20	0 20	0 13	0 22	0 15	0 55	0 45	65 00	100 00	68 18	81 82
Total Time	1831 65	1569 03	980 24	2488 76	1535 53	3318 48	2365 70				
Ratios of Total Time								62 47	60 38	61 70	71 29

denoted "AMS." (We do not count time spent in MPSOUT anywhere since in practice an SP code would be integrated into an optimizer rather than running standalone.) Column "COMB_PA" reports combinatorial time in SPARSER, which we compare with "PASS1" time in HASP. Column "SPARSR" reports total combinatorial plus numerical time in SPARSER (exclusive of ALLOC + MPSIN + SPINIT), which we compare to "PASS1 + 2" in HASP. The "adj. total" columns include the ALLOC + SPINIT time, excluding ALLOC time relating to MPS input (since this needs to be done by the optimizer anyway). We further compute HASP / SPARSER time ratios for combinatorial processing

("Ratio_1"), numerical processing ("Ratio_2"), combinatorial plus numerical processing ("Ratio_+"), and for adjusted total processing ("Ratio_A"), defined by

$$\text{Ratio\_1} = 100 \times \frac{\texttt{PASS1}\text{ time}}{\texttt{COMB\_PA}\text{ time}},$$

$$\text{Ratio\_2} = 100 \times \frac{\texttt{PASS1}+2\text{time} - \texttt{PASS1}\text{ time}}{\texttt{SPARSR}\text{ time} - \texttt{COMB\_PA}\text{ time}},$$

$$\text{Ratio\_+} = 100 \times \frac{\texttt{PASS1}+2\text{ time}}{\texttt{SPARSR}\text{ time}},$$

$$\text{Ratio\_A} = 100 \times \frac{\text{HASP adjusted total time}}{\text{SA adjusted total time}}.$$

If a problem was not reduced in Pass 1, then Pass 2 was skipped, and its Ratio_2 and Ratio_+ entries are marked by a "-". We also cumulate total times over the 68 problems at the bottom of Table III and compute the values of the ratios using these total times.

PASS1 of HASP was always faster than or equal to the combinatorial part of SPARSR for all 68 test problems except CZPROB. Using total times over all 68 problems, Ratio_1 and Ratio_2 are respectively 62.47% and 60.38%, i.e., the combinatorial and numerical computations of HASP are 1.60 and 1.66 times faster than their counterparts in SPARSER. But Ratio_2 varies a lot: from 1.39% to 215.15%, with 8 problems less than 10% and 8 problems greater than or equal to 100%. Ratio_1 and Ratio_2 do not seem to be related to each other. A problem with great speed in finding a combinatorial solution may be slow in the numerical counterpart, and vice versa. This implies that the improvement in the total speed by using HASP does not solely rely on the improvement in one part (either combinatorial or numerical) of computation. Ratio_+ compares the sum of the combinatorial and numerical solution times of the two algorithms; other parts of computation common to both are not included. Totalled over all 68 problems, PASS1 and PASS2 together were about 1.62 times faster than SPARSER. As measured by the overall Ratio_A, HASP ran 1.40 times faster than SPARSER did. The apparent discrepancy between the overall Ratio_A figure of 71.29% and the other ratio figures is due to the fact that AMS time is included in Ratio_A, but not in the other ratios. These routines take about a third of total HASP and SPARSER time, and are the same for both, which dilutes Ratio_A. HASP was slower in only 3 out of 68 problems in adjusted time.

We calculated correlations between various problem attributes and running time. We found that the number of equality rows predicted running times best, with a correlation coefficient of about 0.95 with all times in Table III except "AMS." The number of relevant nonzeros and nonzeros in equality rows both had correlation coefficients of about 0.75 with these times.

The distribution of processing time between combinatorial processing and numerical processing was roughly the same between HASP and SPARSER: both routines spent about 2.5 times longer in numerical processing than in combinatorial. However, since HASP is faster in both these components whereas HASP and SPARSER are the same on SPINIT, the proportion of time spent in SPINIT went up from 26.8% of the total in SPARSER to 37.9% in HASP. We also ran various regressions to see if we could see which sorts of problems are better for HASP than for SPARSER, but we found no conclusive evidence. HASP appears to be generically faster than SPARSER.

It is advantageous to design a fast procedure for finding the combinatorial gain, not only because it reduces the whole preprocessing time, but also because of the following conservative consideration: What if the whole preprocessing step was not worth doing because it only deleted a small number of nonzeros (when the net savings in the total processing time, i.e., preprocessing plus optimization, is the major concern)? If the amount of combinatorial gain is quickly obtainable, it can serve as an indicator to predict whether a positive net savings in total processing time will be achieved. If the prediction says "no," we can skip PASS2 and stick to the original LP, without losing much time in running PASS1. But how much of the adjusted total time is consumed by PASS1? The average ratio (in %) of PASS1 time/HA total time for the test problems with positive combinatorial gains is 19.05%. Later, after running MINOS on A, we shall see that HA total time itself is a small portion of the total processing time for solving an LP (on average only 2.71%, when tested on problems achieving at least a 1% density reduction). Thus, PASS1 is very worthwhile to do; we shall see that overall this cost is more than compensated for by the time saved in optimization.

We also note that the "combinatorial gain" in nonzeros after Pass 1 is often augmented through lucky cancellations to get a much larger "total gain" in nonzeros after Pass 2. We performed a regression in order to predict total gain based on combinatorial gain and found

$$(\%\text{ Redn in NZ after Pass 2}) = 1.44(\%\text{Redn in NZ after Pass 1}) - 1.19,$$

with an R^2 of 0.8661.

Lustig [10] provides pictures of the nonzero pattern for all NETLIB test problems obtainable at that time. Every NETLIB problem with name beginning with "GROW" or "SC" has a staircase structure (see Fourer [6] and Ho and Loute [8]). Including STAIR, there are a total of 17 problems, i.e., 1/4 of the whole test set, having staircase patterns. It is interesting that only 7 of the 17 had positive combinatorial gains in HASP (and SPARSER).

Table IV provides some insight into the sizes of the blocks in T that are encountered in practice. Column "Calls to ORA" counts the total number of calls to ORA over all rows of the matrix. Column "real blocks" reports the number of blocks encountered with more than 1 row (i.e., where manual pivoting does not apply), and "rows in real blocks" counts the total number of rows in such blocks. Column "Sparser rows %" tells what percent of total rows were made sparser, and "Length of TR" tells how much of array TR was actually used during HASP. The two "Rows used" columns report $\sum_i U_i$ and $\max_i U_i$.

Table IV. Blocks and Used-Rows Information

Problem name	Calls to ORA	Real blocks	Rows in real blocks	Sparser rows %	Length of TR	Rows used max	Rows used total
25FV47	154	1	2	18 90	446	57	294
ADLITTLE	9	1	2	17 86	20	2	12
AGG	6			1 23	13	1	6
AGG2	9			1 74	19	1	9
AGG3	9			1 74	19	1	9
BANDM	141			46 23	1195	42	1053
BEACONFD	69			39 88	1274	64	1024
BLEND	23			31 08	68	6	44
BOEING1	5			1 43	16	3	10
BOEING2	1			0 60	6	4	4
BORE3D	64	1	2	28 14	286	36	222
BRANDY	92	1	2	48 19	441	50	351
CAPRI	89			32 84	368	8	280
CZPROB	22			2 37	399	45	376
E226	21			9 42	115	9	93
ETAMACRO	2			0 50	6	2	3
FFFFF800	126			24 05	451	22	324
FORPLAN	20			12 42	101	5	80
GANGES	222	1	12	17 80	1414	12	1312
GFRD-PNC	26			4 22	65	3	38
GREENBEA	376	1	2	15 78	1087	14	711
GREENBEB	376	1	2	15 78	1087	14	711
PEROLD	55			8 80	113	2	57
PILOT	22			1 53	45	1	22
PILOT JA	117			12 45	265	3	147
PILOT WE	31			4 29	63	1	31
PILOT4	123			30 00	259	2	135
PILOTNOV	63			6 46	131	3	67
RECIPE	18			19 78	37	1	18
SC105	8			7 62	81	16	72
SC205	17			8 29	324	34	306
SCAGR25	56			11 89	164	3	107
SCAGR7	20			15 50	56	3	35
SCFXM1	64	4	8	20 61	222	12	164
SCFXM2	128	8	16	20 61	442	12	327
SCFXM3	192	12	24	20 61	662	12	490
SCRS8	110			22 45	446	15	335
SHARE1B	48	9	24	53 85	178	6	164
SHARE2B	48			50 00	97	1	48
SHIP04L	8			2 22	17	1	8
SHIP04S	32			8 89	217	11	184
SHIP08L	48			6 74	97	1	48
SHIP08S	64			8 99	657	12	592
SHIP12L	96			9 21	505	10	408
SHIP12S	96			9 21	1249	15	1152
SIERRA	20			1 64	41	1	20
STANDMPS	1			0 21	110	108	108
STOCFOR1	21			17 95	63	8	41
STOCFOR2	49			2 27	156	8	106
STOCFOR3	213			1 28	780	8	566
VTP BASE	1			0 51	3	1	1

We can draw several conclusions from Table IV. First, only 40 blocks of size larger than 1 were seen in all 51 problems; thus manual pivoting is well worth it. Also, even when real blocks occur, they tend to be quite small; the largest block seen has only 12 rows (for GANGES). Indeed, even the U_i's (which can be the unions of many blocks) tend to be quite small. Thus, it seems likely that HASP is largely taking advantage of relatively few fairly dense rows and also pairs of rows i, k where the nonzeros in row k are a subset of those in row i (so that row k can be used to reduce row i without causing fill-in).

We also collected statistics on the number of calls to MA28 that HASP and SPARSER made for LU factorizations and the time taken up by those calls. We found that over all 51 problems, HASP made 1801 calls to MA28, to SPARSER's 3646 (i.e., fewer than half), largely due to skipping MA28 for 1×1

systems (manual pivoting). The time per call was essentially the same for HASP and SPARSER. Thus we give credit to manual pivoting for saving time in HASP's numerical processing.

3.2 Solving the Original and Reduced LPs with MINOS

Is it really worthwhile transforming the constraint matrix A to a sparser $\hat{A}$ before solving the corresponding optimization problem? The following computational experiments will show that it *is* worthwhile.

Let (A) denote the original linear program:

$$\text{minimize } cx \text{ s.t. } Ax \le b,\ x \ge 0,$$

and $(\hat{A})$ the resulting linear program after being reduced by HASP:

$$\text{minimize } cx \text{ s.t. } \hat{A}\, x \le \hat{b},\ x \ge 0.$$

We used MINOS 5.0 to solve both of them using the 33 NETLIB problems with density reduction of at least 1% as the test set. Note that $(\hat{A})$ is output to a disk file before being input to MINOS.

The MINOS processing times for the two linear programs are denoted by MINOS(A) and MINOS$(\hat{A})$ respectively, and the HASP processing time on (A) is denoted by HASP. I/O times are not included. Besides raw CPU times, the two ratios below are also informative about the "before/after" comparisons:

(1) The percentage reduction in MINOS solution time:

$$100 \times \frac{\text{MINOS}(A) - \text{MINOS}(\hat{A}\ )}{\text{MINOS}(A)},$$

(2) The percentage net savings in MINOS solution time:

$$100 \times \frac{\text{MINOS}(A) - [\text{HASP} + \text{MINOS}(\hat{A}\ )]}{\text{MINOS}(A)}.$$

McCormick [12] describes two kinds of experiments for testing the time savings in running MINOS. They are adopted here. In Experiment I we ran MINOS on (A) and $(\hat{A})$ starting with their own default crash bases (often referred to as a *cold start*). That means no starting basis was specified in advance, and MINOS selects a triangular basis from all columns of the standard-form constraint matrix $(A \quad I)$.

But such comparison may not reveal the true worth of the preprocessing step. The computational report by McCormick [12] describes the difficulty in comparing LP solution times when using a cold start: Although A is equivalent to $\hat{A}$ in Phase 2, with Phase 1 artificial variables $(A \quad I)$ are *not* equivalent to $(TA \quad I)$. This will result in different pivot sequences in solving (A) and $(\hat{A})$ from a cold start, which will produce different numbers of iterations. Thus the difference in run times can be quite independent of the sparseness issue.

In some applications of linear programming, a problem may need to be solved many times with only changes in b or c. A feasible basis can be saved in a file, and when solving this problem again one only needs to run Phase 2. This is often called a *warm start* and is used in Experiment II. Here, equivalence does hold, so that nearly identical pivot paths are taken (numerical perturbations introduced by reduction can cause pivot paths to diverge despite the theoretical equivalence), and we can better judge how matrix reduction contributes to savings in MINOS running time.

Experiment I. *Solving* (A) *and* $(\hat{A})$ *from a Cold Start.* We summarize the cost and savings of CPU time in Table V.

Twenty out of 33 problems have positive reduction in MINOS running time, and 18 of them have positive net savings in total processing time (HASP plus MINOS($\hat{A}$)). That means a little more than half of the test problems are worthy of preprocessing by HASP. The percentage net savings range from the maximum 35.84% of BEACONFD to the minimum −198.76% of PEROLD. These two problems happen to have the highest and the second lowest percentage reduction in nonzeros respectively. But overall there is no strong relationship between "% Net Savings" (or "% Redn in MINOS") and "% Redn in NZ."

Among the problems with positive reduction in MINOS times, we can find only two—SIERRA and GFRD-PNC—that have more time spent in the preprocessing step than the time saved afterwards in running MINOS on the reduced LP's. Thus HASP seems to take only a small amount of time in reducing matrices as compared to the time required for solving the corresponding LP's by MINOS. Indeed, except for BEACONFD and SIERRA, most test problems spent only a small portion (on average, 2.71%) of CPU time in the preprocessing step comparing to the large amount of time consumed by MINOS. As a whole, for the 33 problems tested, the time spent in running HASP is only 0.27% of the time spent in solving these LP's. We also computed the "cost to saving" ratios as another way to assess the work of the preprocessing step, where the "cost" is represented by the time used in HASP and "saving" is the time saved purely in MINOS when $\hat{A}$ instead of A is being used. For the 20 problems with positive reduction in MINOS solution time, the overall "cost to saving" ratio is only 26.87%, quite an encouraging result.

On the other hand, the total time spent in MINOS on the reduced problems was 2.35 times the time spent on the original problems. However, a disproportionate part of this negative result is due to the three hardest problems, GREENBEA, 25FV47, and PEROLD. Without these three outliers, the reduced problems took 1.06 times as long as the original problems, which is still not good.

The reason why HASP looks bad here is that many of the problems used a lot more pivots in their reduced form than in their original form. An outstanding example of this is that the original GREENBEA took 25,983 iterations, but the reduced GREENBEA took 65,634 iterations. Overall, each iteration on a reduced problem costs only 0.93 of an iteration in an original problem, but the increase in number of iterations more than offsets this

Table V. (Exp. I) MINOS Solution Time Reduction

Problem name	% Redn in nonzeros	MINOS(A)	MINOS($\hat{A}$)	HASP	% Redn in MINOS	% Net Savings
BEACONFD	65 45	26 84	13 52	3 70	49 63	35 84
BANDM	25 90	271 96	225 24	9 55	17 18	13 67
GANGES	18 26	920 19	785 62	44 74	14 62	9 76
SHIP12S	14 09	846 74	663 04	22 26	21 69	19 07
BORE3D	13 79	86 98	70 54	4 25	18 90	14 01
BRANDY	13 64	175 40	224 64	3 30	-28 07	-29 95
SHARE1B	13 55	60 98	46 06	2 02	24 47	21 15
RECIPE	11 31	5 32	4 38	0 27	17 67	12 59
BLEND	10 79	13 34	17 66	0 27	-32 38	-34 41
E226	9 39	183 99	210 92	1 58	-14 64	-15 50
SCRS8	8 89	411 50	434 22	6 92	-5 52	-7 20
SHARE2B	8 50	20 86	14 08	0 17	32 50	31 69
SCAGR7	8 33	18 02	19 08	0 50	-5 88	-8 66
SHIP08S	8 32	714 84	661 44	11 26	7 47	5 90
CAPRI	8 09	85 38	80 42	3 35	5 81	1 89
SCAGR25	6 86	426 46	408 60	4 17	4 19	3 21
STOCFOR1	6 71	13 24	8 88	0 31	32 93	30 59
SCFXM1	5 90	198 32	191 72	2 04	3 33	2 30
SCFXM3	5 45	1477 36	1581 72	12 85	-7 06	-7 93
SCFXM2	5 36	686 56	650 76	6 38	5 21	4 29
SHIP04S	4 23	166 46	159 28	3 52	4 31	2 20
CZPROB	3 52	2656 30	2271 86	9 49	14 47	14 12
ADLITTLE	3 39	11 18	14 48	0 10	-29 52	-30 41
SC205	3 06	30 12	33 02	1 02	-9 63	-13 01
STANDMPS	3 07	133 20	149 80	1 87	-12 46	-13 87
GREENBEA	3 07	159533 52	402083 47	124 59	-152 04	-152 12
FORPLAN	3 05	168 44	136 08	1 32	19 21	18 43
25FV47	2 76	12479 92	17594 66	10 01	-40 98	-41 06
PILOT4	2 72	1796 02	3701 00	11 26	-106 07	-106 69
SHIP12L	2 52	1685 38	1579 58	22 01	6 28	4 97
SIERRA	1 37	1157 14	1025 75	154 98	11 35	-2 04
PEROLD	1 28	6448 54	19238 06	27 50	-198 33	-198 76
GFRD-PNC	1 09	397 86	391 32	18 88	1 64	-3 10
Total Time		193308 36	454690 90	526 44	-135 22	-135 49
without outliers		14846 38	15774 71	364 34	-6 25	-8 71

savings. We are unsure why HASP processing appears on average to cause more iterations from a cold start. This point bears further investigation.

Experiment II. *Solving* (A) *and* $(\hat{A})$ *from a Warm Start.* For each of the same 33 test problems we ran MINOS on the LP (A) first, stopped when Phase 1 finished, then used the first feasible basis obtained to start Phase 2 runs to solve both (A) and $(\hat{A})$ to optimality.

We found that the iteration counts in Phase 2 for the two LP's are not all the same. There is no bias favoring either (A) or $(\hat{A})$ regarding iteration counts. In general, their iteration counts are very close; the average ratio of the two for all 33 test problems is nearly 1 : 1. Therefore, the solution times consumed by them are suitable for comparison.

In 30 out of 33 problems the average CPU time used per iteration in Phase 2 for solving $(\hat{A})$ is less than that used for solving (A). Problem BEACONFD has the lowest ratio 0.44; SC205 has the highest ratio 1.04. The mean ratio is 0.91, and standard deviation is 0.1051.

We summarize the cost and savings of CPU time in Table VI. But note that MINOS times now only include solution times in Phase 2, since the two LP's were solved from a warm start. As in Table V, we have also computed totals

Table VI. (Exp. II) MINOS Solution Time Reduction in Phase 2

Problem name	% Redn in nonzeros	MINOS(A)	MINOS($\hat{A}$)	HASP	% Redn in MINOS	% Net Savings
BEACONFD	65.45	19.82	8.70	3.70	56.10	37.44
BANDM	25.90	224.32	184.72	9.55	17.65	13.40
GANGES	18.26	388.86	323.30	44.74	16.86	5.35
SHIP12S	14.09	484.92	396.16	22.26	18.30	13.71
BORE3D	13.79	23.40	21.24	4.25	9.23	-8.93
BRANDY	13.64	76.14	66.76	3.30	12.66	8.35
SHARE1B	13.55	52.76	46.74	2.02	11.41	7.58
RECIPE	11.31	2.64	2.24	0.27	15.15	4.92
BLEND	10.79	9.78	9.74	0.27	0.41	-2.35
E226	9.39	260.42	241.98	1.58	7.08	6.47
SCRS8	8.89	404.84	387.82	6.92	4.20	2.49
SHARE2B	8.50	7.86	6.78	0.17	13.74	11.58
SCAGR7	8.33	6.26	6.05	0.50	3.35	-4.63
SHIP08S	8.32	233.70	208.26	11.26	10.89	6.07
CAPRI	8.09	54.38	50.16	3.35	7.76	1.60
SCAGR25	6.86	264.30	256.64	4.17	2.90	1.32
STOCFOR1	6.71	5.33	4.82	0.31	9.57	3.75
SCFXM1	5.90	101.50	93.40	2.04	7.98	5.97
SCFXM3	5.45	796.62	750.66	12.85	5.77	4.16
SCFXM2	5.36	386.72	365.82	6.38	5.40	3.75
SHIP04S	4.23	85.62	83.58	3.52	2.38	-1.73
CZPROB	3.52	2104.22	1770.76	9.49	15.85	15.40
ADLITTLE	3.39	7.86	7.84	0.10	0.25	-1.02
SC205	3.06	23.60	22.35	1.02	5.30	0.97
STANDMPS	3.07	59.30	56.72	1.87	4.35	1.20
GREENBEA	3.07	76884.77	68482.13	124.59	10.93	10.77
FORPLAN	3.05	125.46	122.64	1.32	2.25	1.20
25FV47	2.76	13322.42	13564.82	10.01	-1.82	-1.89
PILOT4	2.72	4134.14	3999.02	11.26	3.27	3.00
SHIP12L	2.52	1332.88	1294.76	22.01	2.86	1.21
SIERRA	1.37	665.30	652.96	154.98	1.85	-21.44
PEROLD	1.28	10619.58	10243.12	27.50	3.54	3.29
GFRD-PNC	1.10	261.88	263.78	18.88	-0.73	-7.93
Total Time		113431.90	103996.47	526.44	8.32	7.85
without outliers		12605.13	11706.40	364.34	7.13	4.24

overall, and totals excluding the three outliers GREENBEA, 25FV47, and PEROLD.

In 31 out of 33 problems there were positive reductions in MINOS solution time, and overall the reduction in MINOS time was 8.32% (7.13% without outliers). The only two negative reduction problems are 25FV47 and GRFD-PNC; the former needs 242 extra iterations to solve $(\hat{A})$, while the latter has a "% Redn in MINOS" value very close to 0. Overall, "% Redn in MINOS" has a strong and positive correlation with "% Redn in NZ." The correlation coefficient between them is 0.915.

When the "cost factor" (HASP time) is also taken into consideration, column "% Net Savings" in Table VI shows that 25 out of 33 problems have positive net savings in total processing time (with HASP time included). Overall, the net savings was 7.13% (4.24% without outliers). The maximum "% Net Savings" is 37.44% of BEACONFD; the minimum is -21.44% of SIERRA. We could find no common characteristics of either the very good or the very bad problems for "% Net Savings." The warm-start case had results with "hard problems" (the three outliers) which were opposite to the cold-start case: For warm-start, performance increased significantly with the outliers included (% Net Savings went up), whereas for cold-start performance decreased drastically.

Density reduction is also correlated with "% Net Savings," although not as much as "% Redn in MINOS time," since HASP time is involved. In Table VII below, the whole range of the density reduction of the 33 problems is again divided into 5 intervals. In each interval the average percentage reduction in MINOS solution time and the average percentage net savings in total processing time are calculated. The correlation coefficient between "% Redn in NZ" and "% Net Savings" is 0.709, and it again shows a quite strong relationship between the two.

We would like to predict "% Net Savings" in MINOS based on "% Redn in NZ" through a regression on the data in Table VI. We would expect HASP time to contribute negatively to "% Net Savings," so we also include it in the regression. Problem BEACONFD has an anomalously high "% Redn in NZ," so we exclude it as an outlier. Our results are

$$\%\text{ Net Savings} = 0.44\ (\%\text{ Redn in NZ}) - 0.05(\text{HASP time}) + 0.33, \qquad R^2 = 0.2086, \text{ and}$$

$$\%\text{ Net savings} = 0.42(\%\text{ Redn in NZ after Pass 1}) - 0.06(\text{HASP time}) + 0.98, \qquad R^2 = 0.1961.$$

Thus each 1% decrease in nonzeros leads to about 0.43% net decrease in MINOS solution time.

We also computed the two ratios HASP/MINOS(A) and HASP/(MINOS(A) − MINOS($\hat{A}$)) for each problem. Note that the MINOS time now represents the time spent solving Phase 2 only. As a whole, for the 33 problems tested, the time spent in running HASP is only 0.46% of the time spent in all Phase 2 iterations. Except for the two problems, 25FV47 and GFRD-PNC (where the reductions in MINOS are negative), the overall "cost to saving" ratio, as represented by HASP/(MINOS(A) − MINOS($\hat{A}$)), is 5.14%.

4. RECOMMENDATIONS AND FURTHER WORK

In summary, using HASP to make the constraint matrix sparser often helps reduce MINOS running time as well as total processing time when solving the reduced LP by the simplex method. The "% Redn in NZ" (either total, or only after Pass 1) could be used as a rule of thumb for deciding whether to optimize using A or $\hat{A}$ when HASP ends. If the total "% Redn in NZ" is 3% or higher, solve the LP using the reduced $\hat{A}$. If a user wants finer control, the "% Redn in NZ" after Pass 1 can be observed (recall that Pass 1 takes only about 20% of total HASP time); if this is 2% or greater, then continue with Pass 2 of HASP, and decide as above. Note however that a simpler strategy is often preferable: just use $\hat{A}$ no matter what. In some cases HASP will incur a net time penalty, but it is apt to be small compared to total solution time.

The above strategy is based on our computational experience with the NETLIB test set which contains a large number of staircase LP's and may have some bias affecting these and other computational results. It is difficult

Table VII. (Exp. II) Distribution of Density Reduction and MINOS Solution Time Reduction (in Phase 2).

Density reduction range (in %)	[1, 3)	[3, 5)	[5, 10)	[10, 14)	[14, 65 45]
Number of problems	6	7	11	5	4
Average % Rcdn in MINOS	1 19	5 90	7 15	9 77	27 23
Average % Net Savings	−3 96	3 83	3 87	1 91	17 17

to propose any meaningful classification of which LPs are "good" or "bad" for HASP, or even to point to gross characteristics of LPs that are favorable (other than having relatively many equality rows). More experience needs to be accumulated from production use of HASP on a variety of applications to generate more refined guidelines. Such experience would also pin down whether the increase in iterations seen for some large problems using a cold start (Table V) is merely an anomaly or is instead a persistent phenomenon that needs to be addressed.

We also intend to test HASP in other situations: It would be interesting to see how much HASP speeds up an interior-point code (as Adler, et al. [1] have done). However, with interior-point codes the sparsity of AA^T is more important than the sparsity of A, so we believe that other approaches would be better (see McCormick and Chang [14]). It would also be interesting to see how much increased sparsity helps out in nonlinear optimization with linear constraints.

REFERENCES

1. ADLER, I., KARMARKAR, N., RESENDE, M., AND VEIGA, G. Data structures and programming techniques for the implementation of Karmarkar's algorithm, Tech Rep. Dept of Industrial Engineering and Operations Research, Univ of Calif., Berkeley, 1987 A shorter version appeared in *ORSA J. Comput. 1*, 2 (1989), 84–106.
2. CHANG, S. F. Increasing sparsity in matrices for large scale optimization—Theoretical properties and implementational aspects. Ph D. Thesis, Columbia Univ., New York, 1989
3. CHANG, S. F., AND MCCORMICK, S. T. A faster implementation of a bipartite cardinality matching algorithm. Univ. of British Columbia Tech Rep UBC 90-MSC-005, 1990.
4. CHANG, S. F., AND MCCORMICK, S. T A hierarchical algorithm for making sparse matrices sparser. *Math Program. 56*, (1992), 1–30.
5. DUFF, I S. MA28—a set of FORTRAN subroutines for sparse unsymmetric linear equations. A.E.R.E. Harwell Rep. 8730, 1977.
6. FOURER, R. Solving staircase linear programs by the simplex method, 2: Pricing. *Math. Program. 25*, (1983), 251–292.
7. GAY, D. M. Electronic mail distribution of linear programming test problems *Math. Program Soc. Comm. Algorithms Newsl. 13* (1985), 10–12.
8. HO, J. K., AND LOUTE, E. A set of staircase linear programming test problems *Math. Program. 20* (1981), 245–250.
9. HOFFMAN, A. J., AND MCCORMICK, S. T. A Fast Algorithm That Makes Matrices Optimally Sparse. In *Progress in Combinatorial Optimization*, W. R. Pulleyblank, Ed Academic Press, 1984, 185–196.
10. LUSTIG, I. J. An analysis of an available set of linear programming test problems Tech. Rep. SOL 87-11, Systems Optimization Laboratory, Dept. of Operations Research, Stanford Univ , Stanford, Calif., 1987. A shorter version appears in *Comput. Oper. Res. 16*, 2 (1989), 173–184.

11. McCormick, S. T. A combinatorial approach to some sparse matrix problems. Ph.D. Thesis, Stanford Univ., Stanford, Calif., 1983.
12. McCormick, S. T. Making sparse matrices sparser: Computational results. To appear in *Math. Program.*, 1990.
13. McCormick, S. T., and Chang, S. F. Weighted sparsity problem: Complexity and algorithms. UBC Faculty of Commerce Working Paper 90-MSC-007, Vancouver, BC, 1990.
14. McCormick, S. T., and Chang, S. F. Making AA^T Sparser for interior-point algorithms: Complexity and heuristics. In preparation, 1990.
15. Murtagh, B. A., and Saunders, M. A. MINOS 5.0 User's Guide. Tech. Rep. SOL 83-20, Systems Optimization Laboratory, Dept. of Operations Research, Stanford Univ., Stanford, Calif., 1983.

Received September 1990; accepted April 1992