Considerations in the Design of Software for Sparse Gaussian Elimination

S. C. Eisenstat, M. H. Schultz
and A. H. Sherman

Research Report #55

This research was supported in part by the Office of Naval Research, N0014-67-A-0097-0016.

1. Introduction

Consider the large sparse system of linear equations

$$Ax = b \tag{1.1}$$

where A is an N×N sparse nonsymmetric matrix and x and b are vectors of length N. Assume that A can be factored in the form

$$A = LDU \tag{1.2}$$

where L is a unit lower triangular matrix, D a nonsingular diagonal matrix, and U a unit upper triangular matrix. Then an important method for solving (1.1) is sparse Gaussian elimination, or, equivalently, first factoring A as in (1.2) and then successively solving the systems

$$Ly = b,\ Dz = y,\ Ux = z. \tag{1.3}$$

Recently, several implementations of sparse Gaussian elimination have been developed to solve systems like (1.1) (cf. Curtis and Reid [2], Eisenstat, Schultz, and Sherman [5], Gustavson [6], and Rheinboldt and Mesztenyi [7]). The basic idea of all of these is to factor A and compute x without storing or operating on zeroes in A, L, or U. Doing this requires a certain amount of storage and operational overhead, i.e. extra storage for pointers in addition to that needed for nonzeroes and extra nonnumeric "bookkeeping" operations in addition to the required arithmetic operations. All these implementations of sparse Gaussian elimination generate the same factorization of A and avoid storing and operating on zeroes. Thus, they all have the same costs as measured in terms of the number of nonzeroes in L and U or the number of arithmetic operations performed. The implementations do, however, have different overhead requirements, and thus their total storage and time requirements vary a great deal.

In this paper we discuss the design of sparse Gaussian elimination codes. We are particularly interested in the effects of certain flexibility and cost constraints on the design, and we examine possible tradeoffs among the design goals of flexibility, speed, and small size.

In Section 2 we describe a basic design due to Chang [1], which has been used effectively in the implementations referred to above. Next, in Section 3 we discuss the

It is no longer possible, however, to handle multiple righthand sides so efficiently. Then again, if all three steps are combined into one TRKSLV step, it is unnecessary to store even a description of the zero structure of L. But by combining steps in this way, we lose the ability to solve efficiently sequences of systems all of whose coefficient matrices have identical zero structure.

3. Storage of Sparse Matrices

In this section we describe two storage schemes that can be used to store A, L, and U. The schemes are designed specifically for use with sparse Gaussian elimination and they exploit the fact that random access of sparse matrices is not required.

We call the first storage scheme the uncompressed storage scheme. It has been used previously in various forms by Gustavson [6] and Curtis and Reid [2]. The version given here is a row-oriented scheme in which nonzero matrix entries are stored row by row, although a column-oriented version would work as well. Within each row, nonzero entries are stored in order of increasing column index. To identify the entries of any row, it is necessary to know where the row starts, how many nonzero entries it contains, and in what columns the nonzero entries lie. This extra information describes the zero structure of the matrix and is the storage overhead mentioned earlier.

Storing the matrix A with the uncompressed scheme requires three arrays (IA, JA, and A), as shown in Figure 3.1. The array A contains the nonzero entries of A stored row by row. IA contains N+1 pointers that delimit the rows of nonzeroes in the array A -- A(IA(I)) is the first stored entry of the I*th* row. Since the rows are stored consecutively, the number of entries stored for the I*th* row is given by IA(I+1) - IA(I). (IA(I+1) is defined so that this holds for the N*th* row.) The array JA contains the column indices that correspond to the nonzero entries in the array A -- if A(K) contains a_{IJ}, then JA(K) = J. The storage overhead incurred by using the uncompressed storage scheme for A is the storage for IA and JA. Since IA has N+1 entries and JA has one entry per entry of the array A, the storage overhead is approximately equal to

$$L = \begin{bmatrix} 1 & & & & & \\ \ell_{21} & 1 & & & & \\ 0 & \ell_{32} & 1 & & & \\ 0 & \ell_{42} & \ell_{43} & 1 & & \\ 0 & \ell_{52} & \ell_{53} & \ell_{54} & 1 & \\ 0 & 0 & 0 & \ell_{64} & \ell_{65} & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 1 & u_{12} & 0 & 0 & 0 & 0 \\ & 1 & u_{23} & u_{24} & u_{25} & 0 \\ & & 1 & u_{34} & u_{35} & 0 \\ & & & 1 & u_{45} & u_{46} \\ & & & & 1 & u_{56} \\ & & & & & 1 \end{bmatrix}$$

L:	1	ℓ_{21}	1	ℓ_{32}	1	ℓ_{42}	ℓ_{43}	1	ℓ_{52}	ℓ_{53}	ℓ_{54}	1	ℓ_{64}	ℓ_{65}	1
k:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
JL:	1	1	2	2	3	2	3	4	2	3	4	5	4	5	6
IL:	1	2	4	6	9	13	16								

U:	1	u_{12}	1	u_{23}	u_{24}	u_{25}	1	u_{34}	u_{35}	1	u_{45}	u_{46}	1	u_{56}	1
k:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
JU:	1	2	2	3	4	5	3	4	5	4	5	6	5	6	6
IU:	1	3	7	10	13	15	16								

Figure 3.2.

---

they are always equal to 1 and occur as the first stored entry of each row.

Figure 3.3b shows the data structures required when the diagonal entries are omitted. We now note that the indices in JU for certain rows of U are actually final subsequences of the indices for previous rows. For example, the indices for row 3 are 4,5, while those for row 2 are 3,4,5. Instead of storing the indices for row 3 separately, we can simply make use of the last two indices stored for row 2. All that is required is a pointer to locate the indices for row 3.

In general, the indices in JU can be compressed by deleting the indices for any row if they already appear as a final subsequence of some previous row (see Figure 3.4). It is possible to compress the indices in certain other cases as well, but tests have shown that very little is gained by so doing.

Since the compressed indices in JU do not correspond directly to the nonzeroes stored in

Before compaction:

k:	1	2	3	4	5	6	7	8	9
JU:	2	3	4	5	4	5	5	6	6
IU:	1	2	5	7	9	10	10		

After compaction:

k:	1	2	3	4	5	6	7
JU:	2	3	4	5	5	6	
IU:	1	2	5	7	9	10	10
ISU:	1	2	3	5	6	6	

Locations of column indices:	Before compaction	After compaction
row 1	JU(1)	JU(1)
row 2	JU(2) - JU(4)	JU(2) - JU(4)
row 3	JU(5) - JU(6)	JU(3) - JU(4)
row 4	JU(7) - JU(8)	JU(5) - JU(6)
row 5	JU(9)	JU(6)
row 6	-	-

Figure 3.4.

4. Three Implementation Designs

In this section we describe three specific implementation designs, which illustrate some of the tradeoffs mentioned earlier. Designs other than these three can also be derived, but these indicate the broad spectrum of implementations that are possible.

The first implementation (SGE1) is designed for speed. It uses the uncompressed storage scheme for A, L, and U because of the smaller operational overhead associated with it. Furthermore, we combine the NUMFAC and SOLVE steps to avoid saving the numeric entries of L, so that the computation consists of a SYMFAC step followed by the NUMSLV step.

The second implementation (SGE2) is designed to reduce the storage requirements. The entire computation is performed in a TRKSLV step to avoid storing either the description or the numerical entries of L. Moreover, U is stored with the compressed storage scheme to reduce the storage overhead. This design incurs more operational overhead than SGE1; the total storage requirements, however, are much smaller.

Finally, the third implementation (SGE3) attempts to balance the design goals of speed and small size. It splits the computation as in SGE1 to avoid storing the numerical entries of L and it uses the compressed storage scheme as in SGE2 to reduce storage overhead.

n	SGE1 (NUMSLV)	SGE2 (TRKSLV)	SGE3 (NUMSLV)	n	SGE1 (NUMSLV)	SGE2 (TRKSLV)	SGE3 (NUMSLV)
20	.58	1.15	.71	60	18.04	31.55	20.31
25	1.14	2.23	1.40	65	--	--	25.73
30	2.05	3.86	2.45	70	--	--	31.86
40	5.00	9.25	5.91	80	--	78.15	--
50	10.22	18.07	11.52				

Table 5.2.
Timings for the Model Problem (in seconds).

using the FORTRAN IV Level H Extended compiler.* For n = 60, SGE1 is the fastest implementation, requiring 40-45% less time than SGE2 and 10-15% less time than SGE3. On the other hand, SGE2 requires less storage, using 35-40% less than SGE1 and 15-20% less than SGE3. Furthermore, we see that SGE2 can solve larger problems (n = 80) than either SGE1 (n = 65) or SGE3 (n = 70). Evidently, then, the qualitative comparisons suggested in Section 4 are borne out in practice.

6. Conclusion

In this paper we have considered the design of implementations of sparse Gaussian elimination in terms of the competing goals of flexibility, speed, and small size. We have seen that by varying certain aspects of the design, it is possible to vary the degree to which each of these goals is attained. Indeed, there seems to be almost a continuous spectrum of possible designs -- SGE1 and SGE2 are its endpoints, while SGE3 is just one of many intermediate points. There is no single implementation that is always best; the particular implementation that should be used in a given situation depends on the problems to be solved and the computational

* We are indebted to Dr. P. T. Woo of the Chevron Oil Field Research Company for running these experiments for us.