EFFICIENT SPARSE MATRIX MULTIPLICATION SCHEME FOR THE CYBER 203

JULES J. LAMBIOTTE, JR.

NASA/LANGLEY RESEARCH CENTER

HAMPTON, VIRGINIA

Efficient Sparse Matrix Multiplication Scheme for the CYBER-203

Jules J. Lambiotte, Jr.
NASA/Langley Research Center
Hampton, Virginia

Abstract

Many important algorithms for solving problems in linear algebra require the repeated computation of the matrix-vector product $b = Ax$ where A is symmetric and sparse. Examples are the conjugate gradient and Lanczos methods.

This work has been directed toward the development of an efficient algorithm for performing this computation on the CYBER-203. The desire to provide software which gives the user the choice between the often conflicting goals of minimizing central processing (CPU) time or storage requirements has led to a diagonal-based algorithm in which one of three types of storage is selected for each diagonal. For each storage type, an initialization subroutine estimates the CPU and storage requirements based upon results from previously performed numerical experimentation. These requirements are adjusted by weights provided by the user which reflect the relative importance the user places on the two resources.

The three storage types employed were chosen to be efficient on the CYBER-203 for diagonals which are sparse, moderately sparse, or dense; however, for many densities, no diagonal type is most efficient with respect to both resource requirements. The user-supplied weights dictate the choice.

Introduction

Many of the important numerical techniques used today to solve linear equations require repeated computation of a symmetric matrix times a vector. Examples are the conjugate gradient method, with all its variants, for solving simultaneous linear equations (refs. 1 and 2) and the Lanczos algorithm for eigenvalue and eigenvector extraction (ref. 3). These methods are particularly attractive when the matrix is sparse since, unlike direct methods, they do not require storage of the entire matrix. The matrix is only used to multiply a vector and to do this one only needs to know the nonzero elements and their position within the matrix.

The primary objective of this work has been to develop software for the CYBER-203 that provides an efficient means for computing $b = Ax$ when A is an $n \times n$, symmetric, sparse matrix.

Because use of vector hardware instructions on a vector processor has very definite implications about the storage, a user's desire to minimize both the required central processing unit (CPU) time and the total storage needed to represent A are often conflicting goals. Thus, a more specific objective of the work has been to design the software so that it provides alternative storage/computational procedures for the matrix A and automatically selects the procedure which best reflects the users relative concerns about minimizing the two resources.

These objectives have led to the development of a diagonal-based storage and computation scheme in which a preprocessing subroutine, CMPACT, chooses one of three storage methods for each diagonal using CPU and storage estimates and user-provided resource weighting information. The subroutine, CMXV, can be called repeatedly to compute Ax using the compact form of matrix A.

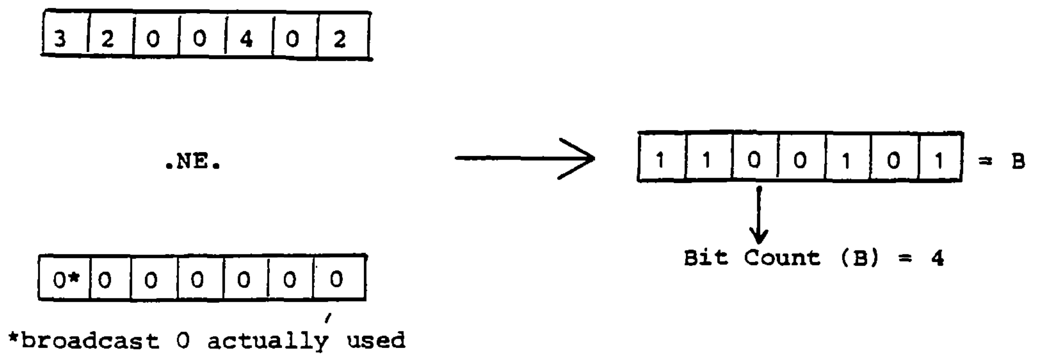
Subsequent sections of the paper will describe the relevant CYBER-203 instructions used, the diagonal-based algorithm with the tradeoffs between the methods, a description of the implementation used, and results for several sparse matrices.

CYBER-203 Characteristics

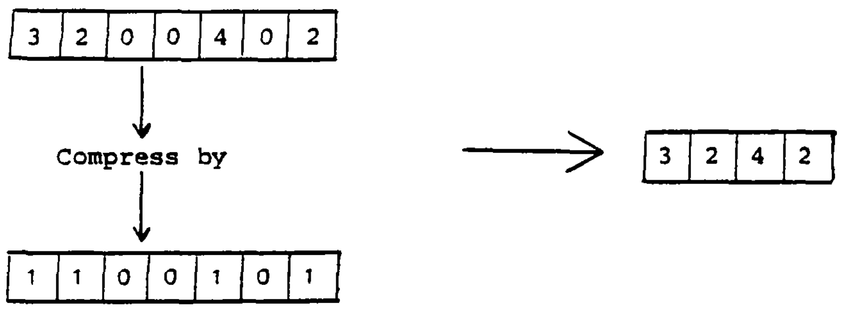
The CYBER-203 at Langley Research Center is a vector processing computer capable of producing 50 million floating point results (64 bit) for a vector addition and 25 million for a vector multiplication. It has one million words of bit addressable central memory in a virtual memory architecture.

The high CPU rates are achieved by operations on long vectors whose components, by definition, are consecutively stored in memory. However, if vector lengths are short (say, 50 or less), the fast scalar capability makes serial computation superior.

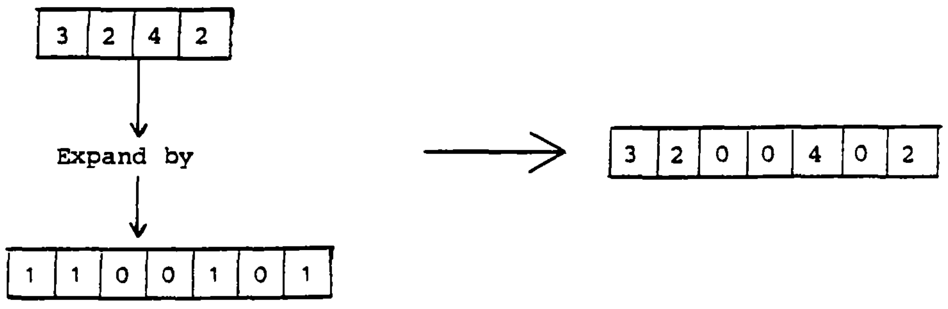
In addition to the usual arithmetic operations (+, -, *, and ÷), several nontypical hardware instructions exist which proved useful in this work. These were the vector compare, compress, expand, and bit count. Figure 1 demonstrates their use.



(a) Compare vector not equal to 0; result to bit vector, B; count "on" bits in B.



(b) Compress vector by bit vector.



(c) Expand compressed vector by bit vector.

Figure 1. CYBER-203 nontypical vector instructions.

Diagonal-Based Matrix Multiplication

It is possible to describe the multiplication process $b = Ax$ for a matrix A in terms of elements of each diagonal. Let $A(\ell)$ denote the ℓ^{th} superdiagonal (also the ℓ^{th} subdiagonal since A is symmetric) and let $A_k(\ell)$ be the k^{th} component. That is, $A_k(\ell) = a_{k,k+\ell} = a_{k+\ell,k}$. The procedure for computing $b = Ax$ for the $n \times n$ matrix A is

```
    $b_k \leftarrow A_k(o)\ x_k \qquad k = 1,2,\dots,n.$
    For $\ell = 1,2,\dots,n-1.$
        $b_k \leftarrow b_k + A_k(\ell)\ x_{k+\ell} \qquad$ for $k = 1,2,\dots,n-\ell$        (1)
        $b_{k+\ell} \leftarrow b_{k+\ell} + A_k(\ell)\ x_k \qquad$ for $k = 1,2,\dots,n-\ell$        (2)
    End F
```

Note that if A is banded, ℓ need only go from 1 to the bandwidth β and that if any diagonals are identically zero, they can be easily identified and all computation for them in (1) and (2) can be omitted.

The diagonal-based scheme has been selected as the foundation for this work for several reasons:

a. Nonzero structure of real problems - Many matrices arising from finite difference or finite element formulations naturally lead to a sparsity pattern in which most of the nonzeros lie along a few of the diagonals. The 5 diagonal matrix arising from central differencing of Poisson's equation is an extreme example. Of course, there the pattern is so predictable that special storage techniques are not needed; but for irregular grids, or more complex equations with more complicated differencing, the sparsity is not so easily specified. This is especially true in finite element formulations where one of the strengths of the method is the ability to use nonuniform elements.

b. Vectorization - The $n - \ell$ multiplications and additions in equations (1) and (2) can be carried out by vector operations of length $n - \ell$.

c. Symmetry of diagonals - The ℓth subdiagonal is also the ℓth superdiagonal. Since equations (1) and (2) are identical in form, the storage and computation most appropriate for the subdiagonal is also most appropriate for the superdiagonal.

Storage Tradeoffs

The vector computations implied in equations (1) and (2) assume $A(\ell)$ is available as a vector of length $n - \ell$. However, if the diagonal is relatively sparse, one might not want to store the entire diagonal with all its zeros. In fact, if the diagonal is very sparse, neither vector storage nor vector computation is likely to be very efficient.

Described below are three types of diagonal storage and their associated computation to execute equations (1) and (2).

Full Vector (Type 1) - Here the entire diagonal is stored including any zeros. Vectors of length $n - \ell$ are used. This mode will be most efficient when $A(\ell)$ is very dense.

Compressed Vector Plus Bit Pattern (Type 2) - Here only the nonzeros are stored along with a bit vector to give positional information within the diagonal. The computation is identical to that with type 1 diagonals after an expand is performed to generate the full diagonal $A(\ell)$. The extra expand makes type 2 CPU requirements always exceed type 1, but the storage can be considerably less.

Compressed Vector Plus Row Pointers (Type 3) - Here the assumption is that $A(\ell)$ is so sparse that it will be inefficient to expand the compressed vector. Equations (1) and (2) are executed serially making use of the row indices stored for positional information.

Figures (2) and (3) show the CPU and storage requirements for a diagonal of length 1000 as a function of density. A comparison of the two figures shows that, unfortunately, one cannot identify intervals of density where a particular diagonal type is most efficient with respect to both resources. For instance type 3 CPU is least for $d < 0.11$ but has a greater storage requirement than type 2 for $d > 0.02$. Even in those regions where one diagonal type is most efficient for both resources (type 1 for very dense and type 3 for very sparse), the boundaries of these regions vary with the length of the diagonal.

Since the minimization of both resources is frequently not possible, and since different users may attach different importances to the two resources, it was decided to let the user influence the storage selection through resource weighting factors. To implement this the initialization subroutine, CMPACT, does the following for each diagonal:

(1) Estimates the CPU and storage requirements for each of the three candidate types.

(2) Applies a user-supplied weight to compute the weighted resource requirement for each method.

(3) Selects the storage type that minimizes the sum of the two weighted resource requirements.

That is, denoting the predicted storage and CPU requirements for the j^{th} diagonal type by s_j and c_j respectively, their minimum by s_m and c_m, the users specified weighting by s_w and c_w, then the normalized and weighted resource, r_j, for the j^{th} diagonal type is computed as

$$r_j = \frac{s_j}{s_{min}} s_w + \frac{c_j}{c_{min}} c_w \qquad j = 1,2,3$$

Subroutine CMPACT computes r_j and selects the diagonal type which yields the minimum value of r.

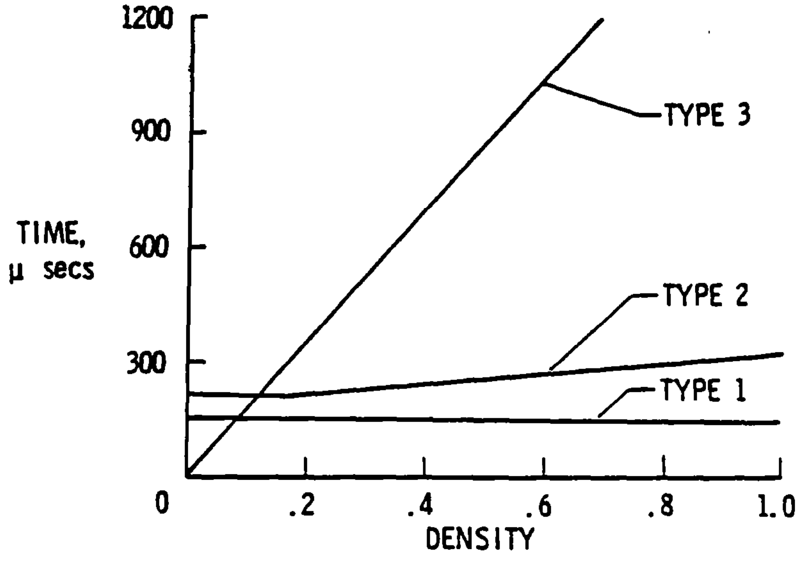


FIGURE 2. CPU TIME FOR DIAGONAL WITH LENGTH 1,000.

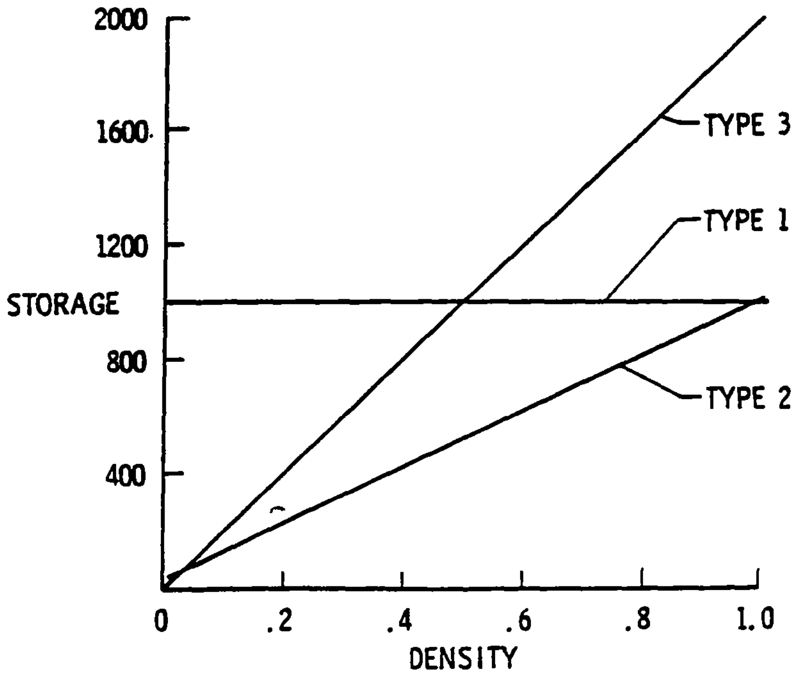


FIGURE 3. STORAGE REQUIREMENTS FOR DIAGONAL WITH LENGTH 1,000.

For this approach, CMPACT must be able to estimate s_j and c_j for all n and d. The storage estimates are easily made in terms of a diagonal of length n having z nonzeros.

$$s_1 = n$$

$$s_2 = z + w$$

$$s_3 = 2z$$

where w is the least number of 64-bit words needed to hold n bits.

The CPU estimates were obtained by timing the computation for a range of n and density d. For type 1 and 3 diagonals, single formulas were obtained, but the complexity of the expand used in type 2 diagonal computation required a table of values. The time in microseconds to perform the computations implied in equations (1) and (2) for a single diagonal can be estimated by

$$C_1 = 29 + 0.122\ n$$

$$C_2 = \text{See Table I}$$

$$C_3 = 7 + 1.74\ z$$

Since these values are used only in a selection process, their accuracy to a percent or two is sufficient.

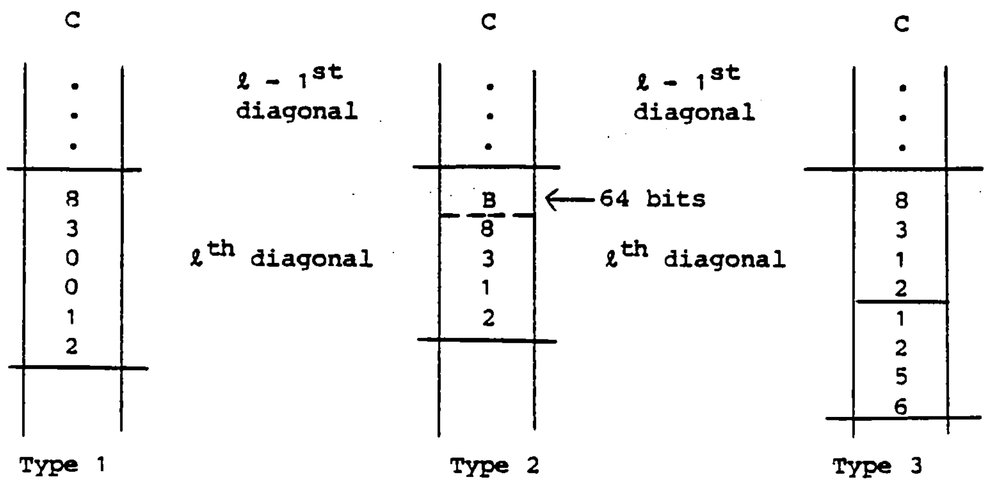
Table I.- Type 2 diagonal CPU times (microseconds) as a function of diagonal length n and density d.

n	d						
	0.	.1	.2	.4	.6	.8	1.0
100	53	53	53	57	60	63	68
500	123	123	124	141	160	176	197
5000	901	901	918	997	1134	1280	1429

Implementation

The matrix is received in subroutine CMPACT in its expanded form as an N by IB array. Each of the IB diagonals is treated individually as the compact representation, array C, is formed. C is a linear array in which the pertinent data for the L th diagonal is stored behind that for the L - 1 st diagonal. As illustrated in figure 4, this can be, for types 1, 2, or 3 respectively, either the entire diagonal, the nonzero bit pattern for the diagonal followed by the nonzeros, or the nonzeros and index data. A vector compare with broadcast zero generates the bit pattern and provides the number of nonzeros and density. If the weighting procedure determines that the diagonal should be type 2 or 3, a compress is performed. In addition, two integers for each diagonal are stored in a separate array. The first identifies the diagonal type and the second the number of nonzeros in the diagonal.

The subroutine returns to the user the CPU and storage estimates for the user provided weights. In addition the estimates for combinations $s_w = 1$, $c_w = 0$ and $s_w = 0$, $c_w = 1$ are returned to aid the user to adjust his weights in subsequent computations.



$$A(\ell) = [8 \quad 3 \quad 0 \quad 0 \quad 1 \quad 2]$$

$$B = [\underbrace{1\ 1\ 0\ 0\ 1\ 1\ 0\ .\ .\ .}_{64 \text{ bits}}\ 0]$$

Figure 4 - Storage for $A(\ell)$ $(n - \ell = 6)$.

Results

Results from two test matrices are presented here to demonstrate the effect and control the user has on the matrix storage and computational requirements by giving the statistics for different combinations of s_w and c_w. Refer to Tables II and III.

Case 1 - This is a randomly generated matrix with 400 equations and a bandwidth of 21. The densities are approximately uniformly distributed between 0. and 1. The average density is 55.7%. The storage selection that minimizes the CPU time (1.57 msec; mostly type 1) yields the largest storage requirement. The selection to minimize storage (4713 words; mostly type 2) yields the largest computation time.

Case 2 - This is a sparse matrix resulting from a finite element formulation with triangular elements and 3 degrees of freedom at each node. The matrix has 1086 equations, a bandwidth of 81, and an average density of 7.8%. Most of the diagonals are sparse. Of the 81 diagonals, 57 are less than 5% dense and approximately half of the nonzeros are on the four diagonals closest to the main diagonal. Because of the relatively few dense diagonals, most of the diagonals are type 2 (to minimize storage) or type 3 (to minimize CPU).

Both examples demonstrate the conflicting goals of minimizing both resources. They also show that use of the weighting factors can give the user a rather wide range of resource distributions. For instance, in the second example a weighting of 1 for c_w leads to a CPU time that is minimum but a storage requirement which is 1.73 times that if one set $s_w = 1$. However, setting $s_w = 1$ yields a CPU time which is 2.6 times the minimum. A reasonable middle ground occurs when $s_w = c_w = 0.5$. In this case, the CPU is 1.09 times the minimum and the storage is 1.2 times the minimum.

Table II.- Case 1; 21 × 400 random matrix.

Weights		Resources		Diagonal Selection		
c_w	s_w	CPU (Secs)	Storage	1	2	3
0	1	.00271	4713	1	20	0
.3	.7	.00217	4950	7	13	1
.5	.5	.00193	5481	11	9	1
.7	.3	.00174	6053	14	5	2
1	0	.00157	7495	19	0	2

Table III.- Case 2; 81 × 1086 finite element matrix.

Weights		Resources		Diagonal Selection		
c_w	s_w	CPU (Secs)	Storage	1	2	3
0	1	.01680	8032	1	72	8
.3	.7	.00800	9200	3	17	61
.5	.5	.00703	9622	3	8	70
.7	.3	.00682	9820	3	4	74
1	0	.00646	13883	8	0	73

Summary

This paper has described a computational and storage algorithm for sparse matrix multiplication on the CYBER-203. The multiplication is performed using diagonals of the matrix as the candidate vectors since this is where nonzero patterns predominate in many scientific applications. Three types of diagonal sparsity patterns are identified (roughly speaking, either dense, moderately sparse, or sparse) and storage and computational procedures developed for each.

Since, for most densities, no single diagonal type minimizes both storage and CPU requirements, an initialization subroutine selects the most "efficient" type for the diagonal based on estimated resource requirements and user-provided weights which indicate the relative importance the user attaches to each resource.

Examples are given which illustrate that, for a given matrix, the weights can be used to achieve minimal CPU time (at the expense of storage) or minimal storage (at the expense of CPU time) or some compromise between the two.

References

1. Hestenes, M. R. and Steifel, G., "Methods of Conjugate Gradients for Solving Linear systems", NBS Journal of Research, 49, 1952.

2. Kershaw, D. S., "The ICCG Method for the Iterative Solution of Systems of Linear Equations", J. Computational Physics, 26 (1978), pp. 43-65.

3. Wilkinson, J. H., The Algebraic Eigenvalue Problem, p. 388, Oxford University Press (Clarendon), London and New York, 1965.