Multiplication of a symmetric banded matrix by a vector on a vector multiprocessor computer

by R. Reuter
U. Scharffenberger
J. Schüle

This paper describes how to vectorize and parallelize the multiplication of a symmetric banded matrix by a vector, on a vector multiprocessor. The ideas presented involve two packed-band-storage schemes, and implementations for both schemes are studied. The best among the uniprocessor solutions proposed achieves a maximum of 37.1 Mflops on an IBM Enterprise System/3090™ 400E with Vector Facility (VF). For one of the schemes, a parallel implementation on an IBM 3090™ VF multiprocessor is presented, and time measurements are discussed.

Introduction

Large-scale simulations in science and engineering often require the solution of extremely large systems of linear equations, usually with several tens of thousands of unknowns. Fortunately, the vast majority of these systems arise from the solution of differential equations using finite difference or finite element methods [1], the matrices representing these linear systems being extremely sparse, with sometimes less than 1% of the entries nonzero. If the matrices were fully occupied, none of the computers currently available could satisfy the storage requirements of realistic problems.

On the other hand, it is a nontrivial task to treat systems of linear equations in true sparse format (storing nonzero entries only). When direct solvers, such as Gaussian elimination, are applied, the matrices usually suffer from fill-in during the elimination process, and only a skilled programmer can keep this fill-in small and provide an efficient implementation of *direct sparse solvers*.

Vectorizing direct solvers is an even more difficult task. Since there are only a few nonzero entries per row (and column), vector lengths are intrinsically small. In addition, these vectors have to be gathered and scattered using indirect addressing. References [2] and [3] give an overview of the difficulties met and of modern approaches to solving them.

For iterative solvers (such as the method of conjugate gradients) the situation is somewhat better. Here, the most time-consuming part is the multiplication of a sparse matrix by a vector, for which some fairly efficient (with respect to vectorization) schemes have been devised [4]. Nevertheless, these schemes require the matrices to be stored in specific formats, a requirement not met by most of the older finite difference and finite element packages.

When a storage scheme should not waste space yet be flexible enough to permit efficient implementations of both direct and iterative solvers, band-storage schemes (described in the following section) offer a good compromise. They do not suffer from the above-mentioned drawbacks, because

- Direct solvers for banded systems are easily programmed, since there is no fill-in outside the band [which means that Gaussian elimination can be performed in place (without the need for additional storage)]. Moreover, data can be accessed consecutively and indirect addressing avoided.
- The storage requirements for banded matrices are usually an order of magnitude smaller than those for a full-storage mode (a fully occupied matrix of the same dimensions). In addition, there are efficient algorithms that reorder the equations and unknowns of linear systems so as to minimize the band envelope of a sparse matrix [5, 6]. Thus the storage overhead of band-storage schemes, compared to true sparse-storage schemes, can be kept relatively moderate.

Moreover, many commercial software packages (such as the IBM Engineering and Scientific Subroutine Library (ESSL) [7]) and public-domain software packages (such as LINPACK [8]) offer good direct solvers for banded systems.

For the reasons given above, many programs from finite element applications employ band-storage schemes for stiffness matrices and use solvers suitable for these.

The following investigation addresses the question of how to multiply a symmetric banded matrix by a vector. The motivation for doing so comes from the method of conjugate gradients (CG), a widely used iterative solver for symmetric positive definite linear systems. Matrix-vector products are by far the most time-consuming kernel of CG and may easily take 95% of the total CPU time used.

The answer to this seemingly trivial problem is far from obvious, as we see below. This paper is a case study of how to efficiently implement a simple mathematical algorithm on a shared-memory vector multiprocessor computer such as the IBM ES/3090™ Vector Facility (VF). Here, "efficient" means that an implementation must

- Take into account the symmetry of the problem.
- Effectively utilize the storage hierarchy of the computer, especially to achieve a good reuse of data already residing in the cache or in vector registers.
- Make efficient use of the vector instructions offered by the hardware.
- Perform a coarse-grained decomposition into parallel tasks with well-balanced task sizes.

Although timing results and speedups are reported, they are not considered the main achievement of our work. The reader should consider it to be a case study in which difficulties were met at different stages of the implementation and ideas developed to overcome them.

Data structures for symmetric banded matrices

A symmetric $n \times n$ matrix $\mathbf{A} = (a_{ij})$ is called banded if its nonzero entries are concentrated along some consecutive diagonals (including the main diagonal):

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1m} & & & & \\ \vdots & & & \ddots & & \mathbf{0} & \\ a_{1m} & & & & \ddots & & \\ & \ddots & & & & a_{n-m+1,n} & \\ & & \ddots & & & \vdots & \\ & \mathbf{0} & & \ddots & & & \\ & & & & a_{n-m+1,n} & \cdots & a_{nn} \end{bmatrix}$$

(m columns, n rows)

With the above notation, $a_{ij} = 0$ whenever $|j - i| \geq m$. The number m is called the *half-bandwidth* of the banded matrix **A**. In total, $2m - 1$ diagonals of the matrix are *occupied*, but some (or even many) of the entries inside the band may be zero.

As mentioned before, there are algorithms that reorder the rows and columns of a matrix so as to reduce the bandwidth. Typically, in real-life applications, the half-bandwidth m is small compared to the size n of the matrix. Commonly, n is ten times m or even larger, so that storing **A** in a full-storage mode would result in a tremendous storage overhead.

In the case of banded matrices, there are two obvious storage schemes for exploiting the specific structure, which are discussed in the following sections.

- *Packed-band-storage scheme (PBS)*

In this scheme, an array AA of dimension (m, n), as shown below, is defined to consist of diagonals of **A**. The first row of AA is the main diagonal of **A**, and the jth off-diagonal of **A** comprises the $(j + 1)$th row of AA. Because of the symmetry of **A**, only the off-diagonals above (or

below) the main diagonal are needed. Note that the lengths of the off-diagonals decrease, so that trailing zeros must be inserted to fill the rows of AA.

$$AA = \begin{bmatrix} a_{11} & \cdots & a_{n-m+1,n-m+1} & a_{n-m+2,n-m+2} & \cdots & a_{n-1,n-1} & a_{nn} \\ a_{12} & \cdots & a_{n-m+1,n-m+2} & a_{n-m+2,n-m+3} & \cdots & a_{n-1,n} & 0 \\ \cdot & \cdots & \cdot & \cdot & \cdots & 0 & \cdot \\ \cdot & \cdots & \cdot & \cdot & \cdots & \cdot & \cdot \\ \cdot & \cdots & \cdot & \cdot & \cdots & \cdot & \cdot \\ \cdot & \cdots & \cdot & \cdot & \cdots & \cdot & \cdot \\ a_{1m} & \cdots & a_{n-m+1,n} & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

This scheme (and a slight variation thereof, storing the main diagonal in the last row and the last off-diagonal in the first row) is part of the LINPACK standard and is also required by the routines DPBF and DPBS in the IBM Engineering and Scientific Subroutine Library. It is especially suited when direct solvers are used to solve the linear system $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{A}$ banded; for this reason PBS has become quite popular in application programs.

- *Transposed packed-band-storage scheme (TPBS)*

In this scheme, diagonals are stored column-wise from left to right. As is indicated below, all columns but the first must be filled with trailing zeros (note that the array AAT in this scheme is just the transpose of the one from PBS, hence the name):

$$AAT = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{22} & a_{23} & \cdots & \cdot \\ a_{33} & a_{34} & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & a_{n-m+1,n} \\ \cdot & \cdot & \cdots & 0 \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & a_{n-1,n} & \cdots & \cdot \\ a_{nn} & 0 & \cdots & 0 \end{bmatrix}.$$

IBM ES/3090 Vector Facility hardware and software for high-performance computing

The IBM ES/3090 VF offers a set of registers and machine instructions, beyond those provided by the scalar IBM System/370™ architecture, which allows the user to perform several types of operations in a vector mode. The vector operations include

- LOAD from storage to a vector register, and STORE back to memory.
- ADD, SUBTRACT, MULTIPLY, DIVIDE, and ACCUMULATE ($b \leftarrow a_1 + \cdots + a_n$).
- The *compound instructions*
  - MULTIPLY AND ACCUMULATE.
  - MULTIPLY AND ADD.
  - MULTIPLY AND SUBTRACT.

  After a start-up time, these perform two floating-point operations per machine cycle. Application code should contain as many compound instructions as possible to achieve optimal performance.

All vector instructions operate on sets of up to 128 or 256 (depending on the model) elements. The maximum dimension of the set is known as its *vector section*.

An important feature of the arithmetic instructions is that one of the operands need not be loaded into a vector register prior to operation, but can be extracted from storage immediately. For example, in the FORTRAN loop

```
DO I = 1,N
   A(I) = A(I) + B(I) * C(I)
END DO
```

the vector C need not reside in a vector register prior to the multiply-and-add operation. The target vector A, however, must be loaded into a vector register before the vector operation is executed. For a more comprehensive overview of the vector architecture of the IBM 3090™, consult Reference [9].

The VS FORTRAN Version 2 compiler for the IBM 3090 VF [10] exploits many of the features provided by the hardware, thus permitting efficient coding of many problems from science and engineering. Two key techniques VS FORTRAN uses to accomplish this aim are

- *Outer-loop vectorization*. By vectorizing loops other than the innermost one, the compiler can keep values that remain unchanged over the entire inner loop in registers, thus saving load and store operations.
- *Scalar temporary*. In a vectorizable loop, under certain circumstances, the compiler is able to treat a FORTRAN scalar variable as if it were a vector (scalar temporary). In this case, no storage is allocated for the temporary, and different values (corresponding to different loop indices) assigned to the temporary are placed in a vector register for later use. The reader will find examples of scalar temporaries in the remainder of the text.

These possibilities help the programmer to control reuse of vector registers explicitly, thus avoiding superfluous load and store operations. A comprehensive overview of how these features are effectively used is in Reference [11].

Another important aspect that must be considered in high-performance computing is the cache mechanism of the IBM 3090 [12]. The cache is a small but very fast buffer located intermediately between central storage and CPU. Its mechanism, explained in detail in [12], cannot be

controlled directly by the programmer. As a general rule, application code should be written in such a way that data are loaded into the cache only once and then reused as often as possible (*data-locality*). Reference [11] discusses how this can be accomplished.

The IBM 3090 is also available as a multiprocessor with up to six independent CPUs. Each CPU contains its own cache, and all or some of the CPUs may be equipped with vector facilities.

The Parallel FORTRAN compiler [13] offers the user the possibility of employing several processors concurrently. However, it is the user's responsibility to exploit this potential. In this respect, four aspects deserve consideration:

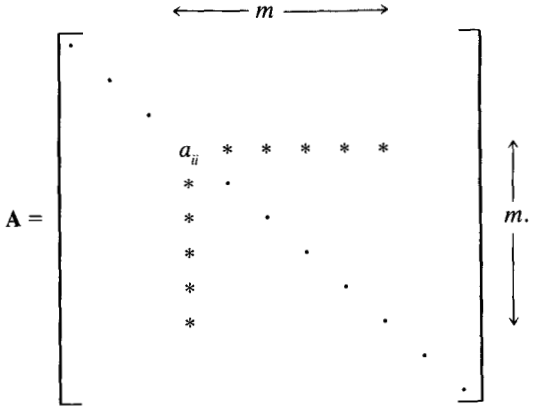
- Setting up parallel tasks is relatively costly. Consequently, the tasks that are assigned to each processor should consume considerably more time than the cost of the administrative overhead (*coarse-grain parallelism*).
- The tasks running concurrently should perform approximately the same amount of work (*load balancing*). Speedups obtained by running components of a program concurrently on a number of processors depend critically on the amount of parallelism inherent in the program (Amdahl's law). If significant portions of the program must run serially, all but one processor will be idle, thus causing a considerable degradation of the overall performance.
- All processors share the central memory of the system, and all private data are organized by the compiler. For a user, this means that special care must be taken for the integrity of data. Concurrent writes to a storage location will result in unpredictable (and, almost surely, wrong) results. Even if Parallel FORTRAN provides efficient data-locking and task-synchronizing mechanisms [13], write-access conflicts may nevertheless offset any performance gain due to parallelism. If possible, different tasks should not write to the same storage locations.
- Each CPU is equipped with its own cache. When tasks operate on identical data, the data must be loaded into the caches of all processors that perform these tasks. If one of these processors updates a datum in its cache, the corresponding data in the caches of the other processors are flagged as invalid. This procedure is necessary to ensure data integrity but prevents cache reuse by the processors with the invalid entries. As a rule, for parallelization, as much as possible different processors should operate on disjoint data.

Vectorized implementations of matrix-vector multiplication

It is clear that different storage schemes call for different implementations. However, one key observation is essential to understanding the following discussion: During the multiplication of a matrix by a vector, each entry of the matrix is referenced exactly once. For a symmetric banded matrix, this means that in storage schemes PBS and TPBS, all entries, except those in the first row or the first column, respectively, of the arrays AA and AAT are addressed exactly twice. If it is possible to load these data only once into a vector register and use them twice for arithmetic operations, a large amount of memory traffic can be avoided.

- *The WING scheme*

In the PBS storage scheme, the columns of the array AA contain *half-rows* (and *half-columns*) of the matrix **A**:



Among the entries marked by an asterisk, the half-row to the right of a_{ii} contains (because of symmetry) the same data as the half-column below a_{ii}. In the PBS scheme, these data constitute a column of the array AA. Thus, if they are stored according to FORTRAN convention, they reside consecutively in memory.

The idea of multiplying a matrix by a vector along half-rows and half-columns is not completely new. Before the use of vector computers became widespread, it was extensively used for multiplying sparse matrices stored in true sparse (symmetric) storage mode by vectors. For example, an early version of ITPACK [14] employed this scheme. In that kind of application, however, the WING scheme demonstrates poor performance [4].

The following piece of code performs the multiplication of a symmetric banded matrix **A**, stored according to the PBS scheme, by a vector **x** to produce a vector **y**:

```
VECT +--- DO 10 K=1,N
     |___    Y(K) = AA(1,K) * X(K)
RECR +--- DO 20 I=1,N
     |       T=0.D0
VECT |+--   DO 30 J=2,M
     ||        T        = T        + AA(J,I) * X(I+J-1)
     ||___     Y(I+J-1) = Y(I+J-1) + AA(J,I) * X(I)
     |___    Y(I) = Y(I) + T
```

Here, N denotes n, the size of the matrix (and **x** and **y**), and M denotes m, its half-bandwidth. Inserted in the code is the vector report listing provided by the VS FORTRAN compiler, which marks the loops that have been chosen for vectorization (VECT) as well as the one that carries data dependencies preventing vectorization (RECR). In these listings, CONTINUE statements closing a loop are suppressed. Note that the code is simplified and does not take into account the fact that the lengths of the last $m - 1$ half-rows (and half-columns) are less than m. In loop 10, Y is initialized as the product of the main diagonal of matrix **A** and vector **x**. The first statement of loop 30 accumulates in T the product of the Ith half-row of **A** and the corresponding part of **x**. The second statement multiplies the Ith half-column of **A** by the scalar $X(I)$ and accumulates the result into the corresponding part of the result vector **y**. The temporary T is added to $Y(I)$ only when the J-loop has terminated.

The code shown above exhibits several important features:

1. The J-loop consists of vector compound instructions: one vector multiply-and-accumulate and one vector multiply-and-add.
2. Data in this loop are accessed consecutively.
3. Columns of AA must be loaded into the cache exactly once. Once an iteration of the I-loop is finished, the Ith column is never touched again.

For banded matrices with sufficiently large bandwidth m, the work done in this routine is dominated by the off-diagonals (i.e., the work done in the J-loop), while the multiplication of the main diagonal by **x** and the load and store operations at the beginning and at the end of the I-loop are negligible. Furthermore, there is little scalar work (address arithmetic) compared to the vector operations contained in the J-loop. For these two reasons, we consider the *number of vector operations (including loads and stores) per pair of off-diagonals* as a sensitive gauge of the efficiency of a piece of code.

In the proposed code for the PBS storage scheme, the following five vector operations are necessary for each vector section of the J-loop:

1. Load one section of $AA(J,I)$ into a vector register.
2. Perform a multiply-and-accumulate with the corresponding section of $X(I+J-1)$ taken from storage immediately. Place the result into the scalar variable T.
3. Load the corresponding section of $Y(I+J-1)$ into a vector register.
4. Perform a multiply-and-add with (the scalar) $X(I)$ taken from storage and $AA(J,I)$ already residing in a vector register.
5. Store the updated section of $Y(I+J-1)$ back to memory.

Note that apart from the multiply-and-accumulate, another operation not mentioned above (sum-partial-sums) is needed to complete the first statement of the J-loop. However, this operation is extremely cheap compared to the multiply-and-accumulate, and in light of the above discussion it is reasonable to ignore it.

• *The DIAG scheme*

In the TPBS storage scheme, the columns of AAT contain diagonals of the banded matrix **A**, which leads to vectorizing along diagonals of **A**. For each diagonal, a section of length equal to the vector section size is multiplied by the corresponding section of the vector **x**, and the result is added to the corresponding section of the vector **y**. Due to the symmetry of **A**, the entries of a diagonal can be kept in a vector register and used twice. However, the two results corresponding to a diagonal and its symmetric counterpart in **A** do not contribute to the same entries of the vector **y**.

The following code performs the same matrix-vector product **y** = **Ax**:

```
VECT +--- DO 10 K=1,N
     |___    Y(K) = AAT(K,1) * X(K)
RECR +--- DO 20 J=2,M
VECT |+--    DO 30 I=1,N
     ||         Y(I)     = Y(I)     + AAT(I,J) * X(I+J-1)
     ||___      Y(I+J-1) = Y(I+J-1) + AAT(I,J) * X(I)
     |___
```

As in the WING scheme, N denotes the size of the matrix and M its half-bandwidth. This code, too, is simplified and ignores the fact that the lengths of the diagonals in the TPBS scheme decrease (thus, the I-loop becomes shorter with increasing J).

Multiplying a matrix by a vector along diagonals is also a well-known idea. An early reference is [15], and a similar idea has been implemented in some routines of the ESSL [7, 16].

In this scheme the following five vector operations are required to process a pair of off-diagonals:

1. Load one section of $AAT(I,J)$ into a vector register.
2. Perform a multiply-and-add with $X(I+J-1)$ loaded from storage. Place the result in the vector register that already contains $Y(I)$.
3. Load a section of $Y(I+J-1)$ into a vector register.
4. Perform a multiply-and-add with $X(I)$ taken from storage and $AAT(I,J)$ already residing in a vector register.
5. Store $Y(I+J-1)$ back to memory.

```
      NP = NPROCS()
      NWORK = N/NP
      PARALLEL LOOP 40 K=1,NPROCS
        DO 20 I=(K-1)*NWORK+1,K*NWORK
          T = 0.D0
          DO 30 J=2,M
            T =
              T              + AA(J,I) * X(I+J-1)
            YY(I+J-1,K) =
              YY(I+J-1,K) + AA(J,I) * X(I)
30        CONTINUE
          YY(I,K) = YY(I,K) + T
20      CONTINUE
40    CONTINUE
      DO 50 I=1,N
        Y(I) = AA(1,I) * X(I)
        DO 60 K=1,NP
          Y(I) = Y(I) + YY(I,K)
60      CONTINUE
50    CONTINUE
```

Figure 1

Synchronized parallel implementation for the WING scheme.

The code presented above is very similar to that implementing the WING scheme, if the roles of the *I*- and *J*-loops are interchanged. However, there is the important difference that the *I*-loop contains two vector multiply-and-adds, while in the WING scheme the *J*-loop contains a vector multiply-and-accumulate operation.

• *Parallelization of WING and DIAG*

Since WING and DIAG vectorize the inner loop, the canonical way to parallelize is to assign a number of iterations of the outer loop to each processor. If we denote by *NPROCS* the number of processors that are to be employed and by *NWORK* the number of iterations per task (*NWORK* ≃ *N*/*NPROCS*), a Parallel FORTRAN implementation would look as follows [13]:

```
      PARALLEL LOOP 40 K=1,NPROCS
        DO 20 I=(K-1)*NWORK+1, K*NWORK
          ...
          Inner loop
          ...
20      CONTINUE
40    CONTINUE
```

Parallel FORTRAN permits one to specify the number of parallel tasks at run-time, and the routine NPROCS from the FORTRAN library passes this information to the FORTRAN program. For this reason, the above code provides a general-purpose parallel implementation for both schemes, WING and DIAG.

However, both schemes suffer from one important drawback. As mentioned before, a serious performance degradation may occur if write accesses by different tasks to the same storage location have to be synchronized in order to ensure data integrity. An analysis of the code for the WING and DIAG schemes reveals that in the above parallel implementation, different tasks will write to overlapping parts of the vector **y**.

Since the parts of code that must be synchronized are the time-intensive parts of the whole application, it can be expected that synchronization will almost surely result in significant idle time. One solution is to provide copies of **y** for each task and sum up their contributions in a serial mode. The code in **Figure 1** shows a rough implementation of this idea for the WING scheme.

We did not carry out a parallel implementation of the WING and DIAG schemes because time measurement for parallel jobs is a very complex task on multi-user shared-memory machines, and the STRIP scheme, which is described in the following section, proved superior in uniprocessor implementations.

• *An improvement: The STRIP scheme*

The DIAG and WING schemes exploit the symmetry of the matrix **A** by loading matrix data into the vector registers only once and using them twice. However, a price must be paid for this. In both schemes, in the inner loop, one load and one store operation must be performed on each section of the result vector **y**.

For the TPBS storage scheme, there is one computational scheme that overcomes this drawback, at the cost of neglecting the symmetry of the problem. Matrix **A** is divided into several horizontal strips of height equal to the vector section size. **Figure 2** shows such a strip for a half-bandwidth of 3 and a vector section size of 4 (for this example). The asterisks indicate the five diagonal bands inside the strip.

In the STRIP scheme, the sections of the diagonals are multiplied by corresponding sections of the vector **x**, and the results are accumulated in a section of the vector **y**. The particular sectioning of the STRIP scheme permits the accumulation to be performed in a vector register as shown in the following code:

```
VECT  +---  DO 10 I=1,N
      |       T = AAT(I,1) * X(I)
RECR  |+--   DO 20 J=2,M
      ||        T = T + AAT(I,J)       * X(I+J-1)
      ||___     T = T + AAT(I+J-1,J) * X(I-J+1)
      |____   Y(I) = T
```

Note that now the I-loop (outer loop) is vectorized. The multiplication of the diagonals by their corresponding sections of the vector **x** is performed in the J-loop, the results being accumulated in the scalar temporary T. In the machine code generated by the compiler, T is replaced by a vector register. Its contents are stored in **y** only after the J-loop has terminated. Also, note that the inner loop now carries a recurrence (RECR) detected by the compiler and cannot be vectorized.

This piece of code requires the following vector operations per pair of off-diagonals (not the main diagonal):

1. Load the right off-diagonal $AAT(I,J)$ into a vector register.
2. Perform a multiply-and-add with $X(I+J-1)$ taken immediately from storage.
3. Load the left off-diagonal $AAT(I+J-1,J)$ into a vector register.
4. Perform a multiply-and-add with $X(I-J+1)$ taken immediately from storage.

The price that must be paid for the savings in vector operations is the violation of the principle of data locality. In the STRIP scheme, the symmetry of the matrix is not exploited, as can be seen from the code segment. Different sections of the array AAT are processed in the two statements of the J-loop, and all data of the array must be loaded into a vector register twice. The code presented above has some degree of data locality, insofar as left and right diagonals are processed alternately. For most iterations of the J-loop, the two sections have some data in common, and these must be loaded into the cache only once.

Since loss of data locality may completely destroy any performance gain due to improved vectorization, only comparative time measurements can determine which scheme is preferable.

Note that the code presented above does not work correctly for the *prolog* (the first $m-1$ rows of **A**) and the *epilog* (the last $m-1$ rows of **A**). There, the lengths of the off-diagonals on the left or the right, respectively, depend upon the row, a fact that prevents the FORTRAN compiler from vectorizing the outer loop. Thus, we wrote the program in IBM System/370 assembly language, so as to be able to keep the accumulated results for each vector section in the vector register denoted by T and simultaneously change the lengths of the vector operations, according to the diagonal being processed. This fact, however, is not the main reason for the performance differences found during the timing measurements.

Parallelization of STRIP

Splitting the STRIP scheme among several processors is very easy. Each task operates on its own strip of **A** and computes a group of consecutive entries of the result **y**. This ensures that no write-access conflicts occur, and the user need not be concerned about data integrity.

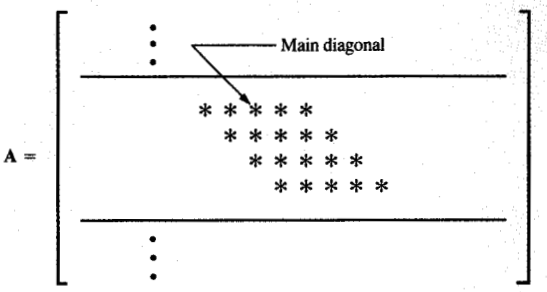


Figure 2

STRIP scheme.

The price paid for easy parallelization is the violation of the "different data to different caches" principle. Different strips of **A** share a considerable overlap of data in the array AAT. As a consequence, these data must be loaded into the caches of at least two processors, thus increasing the total number of machine operations executed. Only time measurements can show whether the performance degradation due to this effect is pronounced or (we hope) not.

Load balancing

The m rows of the prolog (and the epilog) require fewer floating-point operations than m rows of the body of **A**, and the corresponding pieces of code are thus faster. However, since prolog and epilog perform more complicated operations, this effect is much less pronounced than expected. For this reason, we decided to employ the simplest task-distribution algorithm, namely to distribute an equal number of rows to each processor. If the matrix size n is not exactly divisible by the number of processors, half of the remaining rows are given to the processor working on the prolog and the other half to that working on the epilog.

Comparative measurements

Figures 3 and 4, shown later, compare the performance of the three schemes WING, DIAG, and STRIP, run on a uniprocessor. Note that we did not measure the symbolic codes given in the text above; rather, the true codes for the multiplication of a symmetric banded matrix by a vector were compared. Recall that WING and DIAG were programmed in FORTRAN, while STRIP was coded in assembly language.

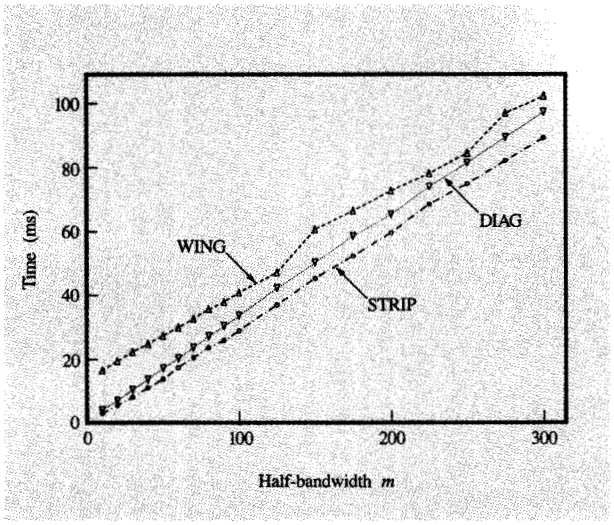


Figure 3

Multiplication times for constant matrix size (n = 2500) as a function of half-bandwidth.

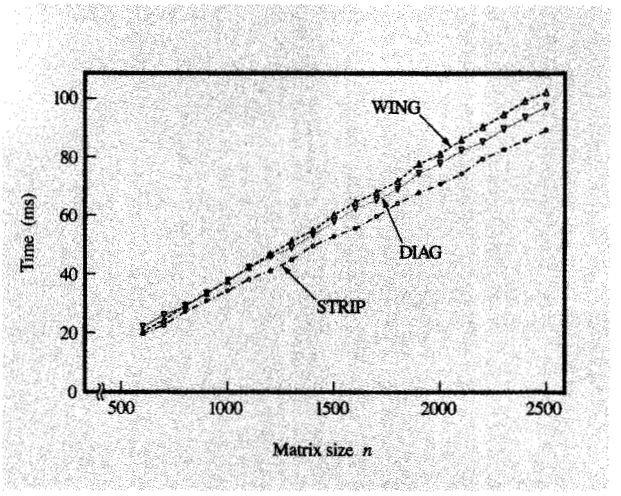


Figure 4

Multiplication times for constant half-bandwidth (m = 300) as a function of matrix size.

In multi-user environments, CPU-time measurements are not completely reliable but provide results that are slightly greater than the true values. This effect, while negligible for long runs, may cause a relatively considerable loss of accuracy for jobs that run only for a few milliseconds. For this reason, all measurements were repeated five times, and the minimum of the five results was used. This procedure gives more reliable results than just averaging over the number of runs.

The measurements covered a parameter range of matrix size n from 600 to 2500 in steps of 100 and half-bandwidth m from 10 to 100 in steps of 10 and from 100 to 300 in steps of 25. Results were obtained on an ES/3090 400E running under the MVS/ESA™ operating system. The machine has a vector section size of 128 and a cache size of 64 kilobytes. Each program was compiled and run using the VS FORTRAN Version 2 Release 4 compiler [10].

Figures 3 and **4** show multiplication times for the three schemes. Figure 3 gives results for constant matrix size n = 2500 with variable half-bandwidth m, while Figure 4 shows results for constant half-bandwidth m = 300 and variable matrix size n.

The measurements show that STRIP performs better than the two other schemes over the whole parameter range. The speedup of STRIP compared to DIAG ranges from 1.06 (for n = 900 and m = 275) to 1.38 (for n = 2500 and m = 10). The speedup of STRIP compared to WING ranges from 1.03 (for n = 600 and m = 300) to 5.50 (for n = 2200 and m = 10). STRIP performed at a maximum speed of 37.1 Mflops (for n = 2000 and m = 20), while WING achieved 28.4 Mflops (for n = 2200 and m = 300) and DIAG achieved 29.0 Mflops (for n = 2300 and m = 60).

For small half-bandwidths, the performance of WING is extremely poor. This is because the vector lengths are very short, and in essence the vector overhead is what is being measured in this case. This does not hold for the DIAG scheme, which vectorizes along diagonals independently of the band size. WING is slightly better than DIAG only for large half-bandwidths and small matrix sizes (Figure 4).

In Figure 3, the measurements for the WING scheme clearly show performance degradation when the half-bandwidth m passes multiples of the vector section size (*loop sectioning*). The same effect, although not as pronounced, can be seen in Figure 4 in the measurements for the DIAG scheme.

The figures demonstrate that multiplying a symmetric banded matrix by a vector is better accomplished using the TPBS storage scheme and the STRIP method, over a wide range of matrix sizes and bandwidths. In some cases, however, a user may be forced to employ PBS, either because other CPU-intensive parts rely heavily on this scheme, or the application consists of old "dusty-deck" code that cannot be changed in a reasonable amount of time. Also, in this case the code for the WING scheme exhibits reasonable performance, which is not much worse than for the other schemes.

Time measurements for the parallel runs

The following timing results were measured on an IBM ES/3090 600J running under VM/XA™. Note that they are not comparable to the ones in Figures 3 and 4 (running on

an IBM ES/3090 400E), because the J models have shorter cycle times and advanced design features. The measurements were not performed on a completely dedicated machine. (VM/XA offers the possibility of dedicating specific processors to a single user. This does not mean, however, that the operating system allows the user to exploit all dedicated processors all of the time.)

Figure 5 shows the wall-clock times necessary to execute the parallelized code for STRIP on one, two, three, and four processors, for a typical set of parameters (n = 3920 and m = 290). The results clearly indicate that the efficiency of the code stays far below ideal expectations. With four processors, the execution time is not much smaller than with one processor alone, and the best speedup, obtained with two processors, is less than 1.25.

Familiarity with the *task concept* of Parallel FORTRAN [13] helps one understand this behavior. In Parallel FORTRAN, a task has to be set up using the ORIGINATE statement before it can be assigned work using either the DISPATCH or the SCHEDULE statement. This provides the user with high flexibility for data sharing between different tasks or data copying from one task to another. However, the time to execute the necessary ORIGINATE statements is usually relatively long, growing with the size of the load module and the number of processors employed. This time, while negligible for long-running production code, causes considerable overhead for small parallel jobs that run for only a few tenths of a second. Separate measurements of the times necessary to execute the ORIGINATE statements indicate that it is mainly this overhead that causes the deviation from the ideal of the results shown in Figure 5.

There are many application programs that do not require the full flexibility of the SCHEDULE statement. For this reason the VS FORTRAN Version 2 Release 5 compiler [17] contains a language construct (PARALLEL CALL) that distributes the execution without requiring an ORIGINATE. With this compiler, a user can expect to obtain much better speedups for our application. A rough estimate, based on the results from Figure 5 and the timings of the ORIGINATE statements, gave about 160 Mflops on a four-processor IBM 3090 VF model J.

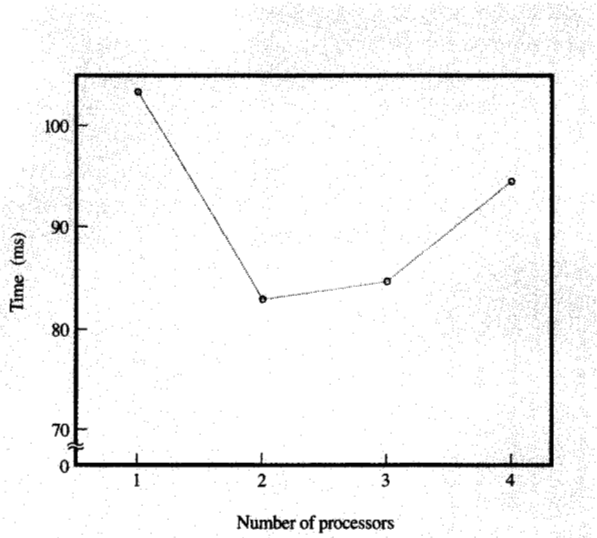


Figure 5

Execution times with one to four processors (n = 3920, m = 290).

Conclusions

Any implementation of a matrix-vector product for a symmetric banded matrix must take into account the specific data structure used. Since the matrices arising from real-life applications are usually extremely large, they must be stored in some packed format in order to fit into memory. Two of these are described above: the packed-band-storage scheme (PBS) and the transposed packed-band-storage scheme (TPBS). These schemes are not restricted to symmetric matrices, but have obvious extensions for nonsymmetric ones. They exploit the band structure of banded matrices almost optimally, insofar as only a small number of zeros must be filled in.

Both storage schemes are suitable for developing efficient implementations of the matrix-vector product on a vector computer, but TPBS appears to be better for the following reasons:

- Both multiplication implementations presented for TPBS produce better timing results than the one presented for PBS, over most of the parameter range considered.
- Since the multiplication schemes for TPBS use diagonals of the matrix as vectors, their performance is not sensitive to the number of off-diagonals, as is the case for the WING scheme presented for the PBS storage mode.

When parallelization is considered, TPBS is even more favorable. There is one implementation (STRIP) that provides a simple, efficient implementation on a shared-memory multiprocessor. It requires no synchronization, and if the load balancing is well-chosen, considerable concurrency may be achieved.

One obvious disadvantage of the STRIP scheme is that it does not exploit the symmetry of the matrix. Nevertheless, in the uniprocessor version, it performs better than the other two schemes presented. This advantage will be even more pronounced if the schemes are extended to treat nonsymmetric matrices.

In the Introduction, we mentioned that the PBS storage scheme is especially suited for direct solvers. This paper indicates that for iterative solvers, TPBS is to be preferred. Unfortunately, there is at present no universal storage scheme for banded matrices that supports both types of solvers. For finite-element packages, for example, this means that the solver for the linear system must be chosen at the time the assembly of the stiffness matrix is designed.

References

1. P. G. Ciarlet, *Introduction to Numerical Linear Algebra and Optimization*, Cambridge University Press, Cambridge, England, 1989.
2. I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, England, 1986.
3. J. A. George and J. W. Liu, *Computer Solution of Large Sparse Matrices*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.
4. A. Peters, "Sparse Matrix Vector Multiplication Techniques on the IBM 3090 VF," *Technical Report 90.10.014*, IBM Scientific Center, Heidelberg, Germany, 1990.
5. E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," *Proceedings of the 24th National Conference of the ACM*, Brandon Press, Princeton, NJ, 1969, pp. 157–172.
6. J. Liu and A. Sherman, "Comparative Analysis of the Cuthill–McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices," *SIAM J. Numer. Anal.* **13**, 198–213 (1976).
7. *IBM Engineering and Scientific Subroutine Library Guide and Reference*, Order No. SC23-0184; available through IBM branch offices.
8. J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1984.
9. W. Buchholz, "The IBM System/370 Vector Architecture," *IBM Syst. J.* **25**, 51–62 (1986).
10. *IBM VS FORTRAN Version 2 Release 4 Language and Library Reference*, Order No. SC26-4221-4; available through IBM branch offices.
11. B. Liu and N. Strother, "Programming in VS FORTRAN on the IBM 3090 for Maximum Vector Performance," *IEEE Computer* **21**, 65–76 (1988).
12. S. G. Tucker, "The IBM 3090 System," *IBM Syst. J.* **25**, 4–19 (1986).
13. L. J. Toomey, E. C. Plachy, R. G. Scarborough, R. J. Sahulka, J. F. Shaw, and A. W. Shannon, "IBM Parallel FORTRAN," *IBM Syst. J.* **27**, 416–435 (1988).
14. D. R. Kincaid, J. R. Respess, and D. M. Young, "ITPACK 2C: A FORTRAN Package for Solving Large Sparse Systems by Adaptive Accelerated Iterative Methods," *ACM Trans. Math. Software* **8**, 302–322 (1982).
15. N. K. Madsen, G. H. Rodrigue, and J. I. Karush, "Matrix Multiplication by Diagonals on a Vector Parallel Processor," *Info. Proc. Lett.* **5**, 41–45 (1976).
16. G. Radicati di Brozolo and M. Vitaletti, "Conjugate Gradient Subroutines for the IBM 3090 Vector Facility," *IBM J. Res. Develop.* **33**, 125–135 (1989).
17. *IBM VS FORTRAN Version 2 Release 5 Language and Library Reference*, Order No. SC26-4221-5; available through IBM branch offices.

Received October 12, 1990; accepted for publication January 14, 1991

Richard Reuter *IBM Germany, Heidelberg Scientific Center, Tiergartenstrasse 15, D-6900 Heidelberg, Germany.* Dr. Reuter received his doctorate in mathematics from the Technical University of Aachen in 1982. He joined IBM in 1984 and initially worked in a joint research project with the University of Heidelberg on interactive statistical transplant data analysis on large databases. Since 1986, he has participated in the design and implementation, on the IBM 3090 VF, of NIC kernels required in engineering applications. Dr. Reuter's current research interests include the study of algorithms in the field of linear algebra for parallel and vector architectures.

Ulrich Scharffenberger *IBM Germany, Heidelberg Scientific Center, Tiergartenstrasse 15, D-6900 Heidelberg, Germany.* Dr. Scharffenberger received his doctorate in mathematics from the University of Frankfurt in 1988. That same year he joined IBM, and since then has worked on joint research projects with German universities. His current research focuses on kernel algorithms from linear algebra and Monte Carlo algorithms for vector and parallel architectures.

Joseph Schüle *Computing Center, Technical University of Braunschweig, Hans-Sommer Strasse 65, D-3300 Braunschweig, Germany.* Dr. Schüle received his doctorate in chemistry from the University of Münster in 1986. Between 1986 and 1989, he held postdoctoral positions in theoretical physics and physical chemistry at the universities of Stockholm and Berlin (FU), working on large-scale computations in theoretical chemistry. He assumed his current research position as staff member of the Computing Center of the Technical University in 1989. His current interests are in vectorization and parallelization of linear system solvers.