1989

Increasing the Efficiency of Vector ITPACK

Robert E. Lynch
Purdue University, rel@cs.purdue.edu

Report Number:
89-866

INCREASING THE EFFICIENCY OF VECTOR ITPACK

Robert E. Lynch

CSD-TR-866
February 1989

INCREASING THE EFFICIENCY OF VECTOR ITPACK

Robert E. Lynch
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

0. Preface. Except for some corrections of minor misprints, this is the same as a memo written in July, 1984, which had extremely limited distribution. A few members of the Purdue Computer Science Department received copies and several copies were left with the University of Texas Center of Numerical Analysis where the results were presented orally and discussed during a two-day visit of the author. The puns in Section 3 are intentional.

1. Introduction. Vector ITPACK is a collection of iterative schemes designed to solve large sparse systems of linear algebraic equations; it was developed by researchers at the Center for Numerical Analysis of the University of Texas at Austin.

Because the solution of very large systems of equations is expensive, it is important that packages such as ITPACK be as computationally efficient as possible. Reduction of execution time by a factor of 2 or 4 or 6 will produce very substantial reduction of cost to the user.

This report includes analyses of some features of ITPACK routines which lead to recommendations designed to reduce execution time. Although our analyses concentrate on improving efficiency on the Cyber 205, some of our recommendations also give improved efficiency on other vector or pipe-line computers, such as the Cray 1.

We restrict our discussion to the use of ITPACK for the solution of linear systems which arise from the finite difference discretization of a partial differential equation.

Excluding the time for parameter adaption and stopping tests, our theoretical analysis predicts reduction in execution time by a factor of 4.8 (see end of Section 5). We have only one experimental comparison: our program for Jacobi iteration with a natural ordering runs 6.15 times faster than ITPACK's; see the end of Section 6.

In Section 9 we compare timings of ITPACK and some direct methods.

2. Times for vector operations. Throughout, we use the timing for operations with vectors of length N on a 2-pipe Cyber 205 as available at Purdue (times for vector arithmetic operations are halved for a 4-pipe machine when $N \gg 1$). When comparing various alternatives, we do so for $N \gg 1$, and neglect the 'start-up' time.

We are primarily concerned with three-vector operations: addition of a pair of vectors, $a_i = b_i + c_i$, $i = 1, \ldots, N$; componentwise vector multiplication, $a_i = b_i * c_i$, $i = 1, \ldots, N$; and 'gather'. The 'gather' vector operation constructs a vector, **a**, from components (rearranged, duplicated, or eliminated) of a vector **b**, as specified by a 'control vector', c_i, $i = 1, \ldots, N$. The value of c_i is the component (subscript j) of **b** which is to be made the i-th component, a_i, of the vector **a**. The times for these operations on a 2-pipe Cyber 205, as measured by clock-cycles, are listed in Table 1 for 64-bit and 32-bit vector operations.

Table 1. Times in clock cycles for vectors of length N

	64-bit	32-bit
vector add, +	51 + N/2	51 + N/4
vector multiply, *	52 + N/2	52 + N/4
gather	69 + 5 N/4	

Each cycle on the Cyber 205 takes 0.02μsec = 0.02×10^{-6} seconds. Henceforth, we give times in microseconds; Table 2 lists these times for the vector operations we consider.

Table 2. Time in microseconds

	64-bit	32-bit
+	1.02 + 0.010N	1.02 + 0.005N
*	1.04 + 0.010N	1.04 + 0.005N
gather	1.38 + 0.025N	

Note that execution of half-precision (32-bit) *arithmetic* operations takes half the time ($N \gg 1$) of full word (64-bit) arithmetic operations. Also note that (neglecting start-up time) a 'gather' takes the time of 2.5 64-bit vector arithmetic operations and 5 32-bit arithmetic operations.

3. Word lengths. Each iteration increases the accuracy of an approximate solution of $A\mathbf{x}=\mathbf{b}$ by only a tiny bit. For example, if it takes 20 iterations to reduce the error by a factor of 10^{-5}, then one decimal digit of accuracy is obtained every 20/5=4 iterations and one binary bit in 1 to 2 iterations. If it takes 100 iterations, then 20 iterations are needed for every decimal digit of accuracy and about 6 per binary bit.

If one writes the iterative scheme as $\mathbf{x}^{(m+1)}=\mathbf{x}^{(m)}+\mathbf{p}^{(m)}$, where $\mathbf{p}^{(m)}$ is the computed pseudo-residual, then one might have the situation in which $x_i^{(m)}=0.12345\cdots$ has 3 digits of accuracy and $p_i^{(m)}=0.0004321....$ But, because only a tiny bit of increased accuracy is obtained, clearly, only the leading significant digit (the 4) in $p_i^{(m)}$ is of importance. Consequently, one needs to compute $\mathbf{p}^{(m)}$ accurate to only a very few significant digits.

It follows that use of 64-bit arithmetic cannot reduce the error any faster than 32-bit arithmetic. In contrast, use of 64-bit arithmetic actually *slows* the process because 64-bit arithmetic operations take twice the time of 32-bit arithmetic operations.

Our first recommendation is, therefore, that most of the arithmetic performed by ITPACK be done with Cyber HALF PRECISION variables. If ITPACK is given a matrix with 64-bit entries, then ITPACK could make a 32-bit copy of it. To obtain error reduction of a factor of 10^{-4} or 10^{-5}, probably no 64-bit arithmetic is necessary. If higher than, say, 10^{-5} accuracy is needed then, after obtaining 4 or 5 decimal digits of accuracy with 32 bit arithmetic, ITPACK could do a few 64-bit calculations to determine an accurate current residual and then revert to 32-bit arithmetic.

4. ITPACK data structure. Here we determine the time to carry out a matrix-vector multiplication, $\mathbf{y}=A\mathbf{x}$, with the data structure used by ITPACK, and compare it with the time when an alternative data structure is used.

Two arrays are involved: COEF contains the nonzero entries of A and JCOEF gives the subscripts j corresponding to $a_{i,j}$ as stored in COEF; i.e., JCOEF is, essentially, an incidence matrix for the graph of the matrix A. In many applications, COEF will be generated by a module which discretizes a partial differential equation. (If A arises from a network-flow problem it has a much more complicated graph.)

We begin with an example which appears in Kincaid-Oppe-Young [1982]:

$$A=\begin{bmatrix} a_{11} & a_{12} & 0 & a_{14} \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & 0 & a_{33} & 0 \\ 0 & a_{42} & 0 & a_{44} \end{bmatrix}, \quad \text{COEF}=\begin{bmatrix} a_{11} & a_{12} & a_{14} \\ a_{22} & a_{21} & a_{23} \\ a_{33} & a_{31} & 0 \\ a_{44} & a_{42} & 0 \end{bmatrix}, \quad \text{JCOEF}=\begin{bmatrix} 1 & 2 & 4 \\ 2 & 1 & 3 \\ 3 & 1 & 1 \\ 4 & 2 & 1 \end{bmatrix}.$$

To form $\mathbf{y}=A\mathbf{x}$ (for example when computing a residual), one forms $Y(I)=\sum_J \text{COEF}(I,J)*X(\text{JCOEF}(I,J))$; i.e.,

$$\mathbf{y}=\begin{bmatrix} a_{11} & x_1 \\ a_{22} & x_2 \\ a_{33} & x_3 \\ a_{44} & x_4 \end{bmatrix} + \begin{bmatrix} a_{12} & x_2 \\ a_{21} & x_1 \\ a_{31} & x_1 \\ a_{42} & x_2 \end{bmatrix} + \begin{bmatrix} a_{14} & x_4 \\ a_{23} & x_3 \\ 0 & x_1 \\ 0 & x_1 \end{bmatrix}.$$

The "x" vectors, such as $(x_2,x_1,x_1,x_2)^T$, are formed with gathers.

For an $N\times N$ matrix with at most 3 nonzero entries per row, ITPACK carries out three gathers, three *'s, and two +'s. The times in microseconds are given in Table 3.

Table 3: ITPACK time for forming $\mathbf{y}=A\mathbf{x}$, 3 entries per row

	64-bit	32-bit
3 gathers	4.14 + 0.075N	
3 *	3.12 + 0.030N	3.12 + 0.015N
2 +	2.04 + 0.020N	2.04 + 0.010N
total	9.30 + 0.125N	9.30 + 0.100N

Note that for $N \gg 1$, about 0.075/0.125 = 60 percent of the time is spent in gathering for 64-bits and about 0.075/0.100 = 75 percent for 32-bits – more time is spent gathering than in performing arithmetic.

Note also that changing from 64-bit arithmetic to 32-bit reduces execution time by only a factor of 0.125/0.100 = 1.25 instead of the theoretical optimum of 2.

As pointed out by Kincaid-Oppe-Young [1982], if the matrix has a 'nice' diagonal structure (and if this is known by ITPACK), then one can increase efficiency.

The matrix A above has 5 nonzero diagonals; suppose it is stored by diagonals:

$$\text{COEF}=\begin{bmatrix} a_{11} & 0 & 0 & a_{12} & a_{14} \\ a_{22} & 0 & a_{21} & a_{23} & 0 \\ a_{33} & a_{31} & 0 & 0 & 0 \\ a_{44} & a_{42} & 0 & 0 & 0 \end{bmatrix}$$

Then $y = Ax$ can be computed as

$$y = \begin{bmatrix} a_{11} & x_1 \\ a_{22} & x_2 \\ a_{33} & x_3 \\ a_{44} & x_4 \end{bmatrix} + \begin{bmatrix} & \\ a_{31} & x_1 \\ a_{42} & x_2 \end{bmatrix} + \begin{bmatrix} a_{21} & x_1 \\ 0 & x_2 \\ 0 & x_3 \end{bmatrix} + \begin{bmatrix} a_{12} & x_2 \\ a_{23} & x_3 \\ 0 & x_4 \end{bmatrix} + \begin{bmatrix} a_{14} & x_4 \\ & \\ & \end{bmatrix}$$

No gathers are needed; e.g., having the first term, one adds the next term with

```
DO I = 3, 4
    Y(I) = Y(I) + COEF(I,2) * X(I–2)
CONTINUE
```

which vectorizes.

Moreover, the array JCOEF is not needed, so that storage is reduced. JCOEF can be replaced by a pair of one-dimensional arrays which give the row (equation) number with the first nonzero entry of each column of COEF and the component (less 1) of x which multiplies this coefficient, for example

$$IAFRST = [1,3,2,1,1], \quad IXFRST = [0,0,0,1,3].$$

The DO-loop above becomes

```
DO I = IAFRST(2), N
    Y(I) = Y(I) + COEF(I,2) * X(I+IXFRST(2))
CONTINUE
```

Note that instead of doing operations with vectors all of length 4, here one does vector multiplications with vectors of lengths 4, 2, 3, 3, and 1, and vector additions with vectors of lengths 2, 3, 3, and 1.

The times for the $N \times N$ case with 5-diagonals are listed in Table 4 (we neglect the minor savings because shorter than N vectors are used).

Table 4: 5-diagonal matrix-vector product

	64-bits	32-bits
5 *	5.20 + 0.050N	5.20 + 0.025N
4 +	4.08 + 0.040N	4.08 + 0.020N
total	9.28 + 0.090N	9.28 + 0.045N

When $N \gg 1$, the total times in Table 3 are greater than the corresponding ones in Table 4 by factors of .125/.090 = 1.39 (64-bits) and .100/.045 = 2.2 (32-bits). Furthermore, and in contrast with Table 3, the change from 64-bit to 32-bit arithmetic does reduce execution total time in Table 4 by a factor of 0.090/0.045 = 2.

5. ITPACK and ELLPACK. During the evolution of ITPACK, it was closely associated with ELLPACK. Communication between ELLPACK segments is by means of variables and arrays of prescribed structure. ELLPACK allows 'any' discretization module and the only requirement is that such a module generate arrays COEF and JCOEF together with the values of several switches.

Consequently, ITPACK had to process 'any' COEF array given only that each row of COEF contained matrix entries of a single equation. Since no order of the entries in rows of COEF was

specified by ELLPACK, vector ITPACK resorted to the use of gathers as illustrated in Section 4.

However, the entries in COEF are not stored in a random order by a discretization module. COEF is constructed in some systematic way by each discretization module.

For example, a 5-point difference approximation requires coefficients C, L, R, B, T ('center', 'left', etc.). The matrix is

$$A = \left[\begin{array}{ccc|ccc|ccc} a_{11} & a_{12} & 0 & a_{14} & & & & & \\ a_{21} & a_{22} & a_{23} & 0 & a_{25} & & & & \\ 0 & a_{32} & a_{33} & 0 & 0 & a_{36} & & & \\ \hline a_{41} & 0 & 0 & a_{44} & a_{45} & 0 & a_{47} & & \\ & a_{52} & 0 & a_{54} & a_{55} & a_{56} & 0 & a_{58} & \\ & & a_{63} & 0 & a_{65} & a_{66} & 0 & 0 & a_{69} \\ \hline & & & a_{74} & 0 & 0 & a_{77} & a_{78} & 0 \\ & & & & a_{85} & 0 & a_{87} & a_{88} & a_{89} \\ & & & & & a_{96} & 0 & a_{98} & a_{99} \end{array}\right]$$

If the equations are constructed in a 'natural order', COEF might be the following for 3×3 interior points and Dirichlet boundary conditions (ELLPACK's 5 POINT STAR generates an array with these entries):

$$\text{COEF} = \begin{bmatrix} C & L & R & T & B \\ a_{11} & 0 & a_{12} & a_{14} & 0 \\ a_{22} & a_{21} & a_{23} & a_{25} & 0 \\ a_{33} & a_{32} & 0 & a_{36} & 0 \\ a_{44} & 0 & a_{45} & a_{47} & a_{41} \\ a_{55} & a_{54} & a_{56} & a_{58} & a_{52} \\ a_{66} & a_{65} & 0 & a_{69} & a_{63} \\ a_{77} & 0 & a_{78} & 0 & a_{74} \\ a_{88} & a_{87} & a_{89} & 0 & a_{85} \\ a_{99} & a_{98} & 0 & 0 & a_{96} \end{bmatrix},$$

i.e., columns of COEF contain *diagonals* of A. The product $y=Ax$ can be formed as in the example near the end of Section 4.

The times for $y=Ax$ with such a 5-diagonal matrix are given in Table 4 as 0.090N (64-bit) and 0.045N (32-bit). In addition, there is some savings because, for a square array of $\sqrt{N}\times\sqrt{N}=N$ mesh-points, the lengths of vectors involved in the calculation of $y=Ax$ are: N, $N-1$, $N-1$, $N-\sqrt{N}$, and $N-\sqrt{N}$. Table 5 lists the times when these savings are included.

Table 5: Square array of mesh-points, 5-point star, natural order

	64-bit	32-bit
1 *, length N	$1.04 + 0.010N$	$1.04 + 0.005N$
2 *, length $N-1$	$2.08 + 0.01(2N-2)$	$2.08 + 0.005(2N-2)$
2 *, length $N-\sqrt{N}$	$2.08 + 0.01(2N-2\sqrt{N})$	$2.08 + 0.005(2N-2\sqrt{N})$
2 +, length $N-1$	$2.04 + 0.010(2N-2)$	$2.04 + 0.005(2N-2)$
2 +, length $N-\sqrt{N}$	$2.04 + 0.010(2N-2\sqrt{N})$	$2.04 + 0.005(2N-2\sqrt{N})$
total	$9.28 + 0.010(9N-4\sqrt{N}-4)$	$9.28 + 0.005(9N-4\sqrt{N}-4)$

If one ignores the known structure and uses 5 gathers and operates on vectors all of length N, one has the times listed in Table 6. For $N \gg 1$ the savings in time by using a known diagonal structure (Table 5) instead of gathers (Table 6) are by factors of 0.215/0.090 = 2.4 (64-bit) and 0.170/0.045 = 3.8.

Table 6: Current ITPACK processing, 5-point, natural ordering

	64-bit	32-bit
5 gathers	$6.90 + 0.125N$	
5 *'s	$5.20 + 0.050N$	$5.20 + 0.025N$
4 +'s	$4.08 + 0.040N$	$4.08 + 0.020N$
total	$16.18 + 0.215N$	$16.18 + 0.170N$

Hence it seems worth the investment in analysis and programming time to include input parameters to ITPACK which would identify the structure of COEF, thereby reducing the user's cost. These new parameters would identify things such as

$m\times n$ mesh
5-point 2-d; 7-point 3-d; 9-point 2-d; etc.
natural by mesh-row; red-black; etc.

as well as switches for finite elements and also "no structure specially treated by ITPACK".

We observe that currently, ITPACK processes a 5-point natural ordered array COEF in time (Table 6) $16.18 + 0.215N$ (64-bits). If this structure were known to ITPACK and if ITPACK used 32-bit operations without gathers, then the time would be $9.28 + 0.045N$ (Table 5) for $N \gg 1$. This is a savings of a factor of .215/.045 = 4.8.

6. Experimental results. With $A=I-B$, $\mathrm{diag}\,B=0$, consider Jacobi iteration $\mathbf{x}^{(m+1)}=B\mathbf{x}^{(m)}+\mathbf{b}$, where B is a 4-diagonal matrix from a 5-point approximation on a square set of mesh-points and with natural ordering. Since 4 vector multiplications and 4 vector additions are required per iteration, $8.24 + 0.080N$ (64-bit) and $8.24 + 0.040N$ (32-bit) microseconds are needed. Table 7 lists observed times per unknown per iteration for a program we wrote (the main part of it is given at the end of this report).

As expected, there is only a little variation in the times for scalar operations (column (A)). For vector operations with short vectors, the start-up time makes a significant effect on the execution time. Column (E) shows the percent difference between the observed times in column (C) and the theoretically predicted times in column (D). We conclude that one can obtain execution times close to the optimal times. Garrett Birkhoff expressed this differently: for long vectors, one *can* obtain accurate time estimates from operation counts and the manufacture's published operation times.

Hockney (see Hockney-Jesshope [1981, pp. 51-59]) introduced the concept of "half-performance length $n_{1/2}$": the vector length required to achieve half the maximum performance, and the value given for the Cyber 205 (or 203) is $n_{1/2}=100$. As explained by Hockney-Jessope, the larger the value of $n_{1/2}$, the more "parallel" is a process.

Solving the equations

$$8.24+0.080/N=0.160, \quad 8.24+0.040/N=0.080,$$

for N, we find

$$n_{1/2}=103, \quad n_{1/2}=206,$$

for 64-bit and 32-bit Jacobi iteration, respectively.

Table 7: Microseconds per unknown per iteration; Jacobi; natural ordering

		VAX	Cyber				
1/*h*	N		(A)	(B)	(C)	(D)	(E)
4	9		4.68	1.41	1.37	0.955	43.5
8	49		5.13	0.333	0.285	0.208	37.0
16	225		5.34	0.242	0.0965	0.0766	26.0
20	361	56.30	5.77	0.121	0.0771	0.0628	22.7
32	961				0.0570	0.0485	17.5
64	3969				0.0478	0.0421	13.5

VAX:	short word
(A):	64-bit scalar mode
(B):	64-bit, vector mode, all DO-loops vectorized
(C):	32-bit, vector mode, all DO-loops vectorized
(D):	0.040 + 8.24/N, theoretical time
(E):	100[(C) – (D)]/(D)

Interpolating to the experimental results in columns (B) and (C), with the function $t=a+b/N$, and then solving for N when $t=0.160$ and $t=0.080$, we find

$$n_{1/2}=167, \quad n_{1/2}=331,$$

for 64-bit and 32-bit Jacobi iteration, respectively.

Kincaid-Oppe-Young [1984, p. 27] report observed times on the Cyber 205 for the basic iterative schemes in ITPACK with $N=3969$ $h=1/64$. They list only total times and do not discuss their results; nor do they point out some puzzling features about them. Table 8 lists their times, as well as time per iteration, and time per iteration per unknown. We do not know if their total times include time for

stopping tests.

Table 8: Basic ITPACK schemes

	Natural ordering				Red-black ordering			
	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
Jacobi	4380	5.112	1.17	.294	4380	5.189	1.18	.298
Gauss-Seidel	4982	56.066	11.25	2.835	4978	11.851	2.38	.600
SOR	174	1.971	11.33	2.854	151	.363	2.40	.606
SSOR	140	2.862	20.44	5.151	2477	5.730	2.31	.583
RS					2620	2.785	1.06	.268

(a),(e)	number of iterations
(b),(f)	total time in seconds
(c),(g)	(time per iteration) $\times 10^3$
(d),(h)	(time per iteration) $\times 10^6$ / 3969

We do find it curious that Gauss-Seidel took 602 more iterations than Jacobi – we would expect it to take about 2190 fewer iterations.

We would expect an iteration of SSOR to take about twice the time as an iteration of SOR, as in column (c). It seems strange that the times are about equal when the red-black ordering is used (column (g)).

As expected, the times for the successive methods are much greater than the simultaneous (and vectorizable) Jacobi scheme when the natural ordering is used (column (c)). But, these methods do vectorize when the red-black ordering is used, and we would expect the times to be about the same as for Jacobi; column (g) shows they take about twice the time as Jacobi. In particular, these results suggest that when the red-black ordering is used, the ITPACK Gauss-Seidel and SOR methods can be speeded up by a factor of 2, so that their time per iteration is close to that for Jacobi.

Finally, we find that the ITPACK Jacobi time in column (d) is larger by a factor of .294/.0478 = 6.15 than the corresponding time in column (C) of Table 7.

7. **Fictitious unknowns.** We now show that execution time can be reduced if one introduces fictitious unknowns and equations so that the unknowns are on a rectangular mesh. (It is well-known that this 'trick' works in other cases; e.g., for the Poisson equation, introducing such 'fictitious' points allows one to use FFT in the capacitance matrix method.) At the end of this section we show that this is true even if one has to *triple* the number of unknowns. We also show that for a red-black ordering, execution time is reduced if one adds (if necessary) a column of mesh-points so that in each mesh-row the number of red points and the number of black points are different.

Consider the following triangular array of mesh-points in a natural order:

1
2 3
4 5 6

The 6×6 matrix for the 5-point star has nonzero entries as indicated below; the matrix has 5 nonzero diagonals.

	1	2	3	4	5	6
1	x	x	o			
2	x	x	x	x		
3	o	x	x	o	x	
4		x	o	x	x	o
5			x	x	x	x
6				o	x	x

Each row and column of mesh-points added to the preceding triangular array to get a larger and similar triangular array, adds two more diagonals to the matrix. It would be an inefficient use of storage to store the matrix by diagonals because there would be so many diagonals and zeros. To form $y=Ax$, it would probably be most efficient to use the ITPACK storage mode and gathers.

However, the gathers can be eliminated by adding fictitious mesh-points to get the square array

1 2 3
4 5 6
7 8 9

One then gets a block tridiagonal matrix with tridiagonal diagonal blocks and diagonal off-diagonal blocks. The matrix-vector multiplication is done with 4 additions and 5 multiplications; no gathers are required. See Table 4 for times and see the end of this section for a comparison of efficiencies.

Next consider a red-black ordering for 2×6 mesh-points:

1 7 2 8 3 9
10 4 11 5 12 6

The 12×12 matrix is given below; it has 11 nonzero diagonals.

	1	2	3	4	5	6	7	8	9	10	11	12
1	x			o		o	x	o		x		
2		x			o		x	x	o		x	
3			x			o		x	x	o		x
4	o			x			x		o	x	x	
5		o			x			x		o	x	x
6	o		o			x			x		o	x
7	x	x		x			x			o		o
8	o	x	x		x			x			o	
9		o	x	o		x			x			o
10	x		o	x	o		o			x		
11		x		x	x	o		o			x	
12			x		x	x	o		o			x

In the preceding set of mesh-points, there are as many red as black points in each mesh row. If one adds another column of mesh-points, so that there are different numbers of colored points in each row:

1	8	2	9	3	10	4	
11	5	12	6	13	7	14	

then the number of diagonals is reduced from 11 to 9. One reduces the amount of work necessary for a multiplication, $y=Ax$, by about (11 - 9)/11 = 18 percent. The 14-by-14 matrix is indicated below.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	x				o		o	x			x			
2		x				o		x	x			x		
3			x				o		x	x			x	
4	o			x				o		x	o			x
5		o			x			x			x	x		
6			o			x			x			x	x	
7	o			o			x			x	o		x	x
8	x	x			x			x				o		o
9		x	x			x			x				o	
10			x	x			x			x				o
11	x			o	x			o			x			
12		x			x	x			o			x		
13			x			x	x			o			x	
14				x			x	o			o			x

When ITPACK process known structures, it could use gathers to rearrange COEF and add fictitious equations to obtain a new matrix with 'nice' structure; then no gathers are required when the many iterates are computed.

We now estimate the 'cross-over point' where use of the ITPACK data structure is as efficient as increasing the number of unknowns to get a rectangular array and a 'nice' diagonal structure.

Let the domain contain M interior mesh-points and suppose this is embedded in a rectangle of N mesh-points. When the ITPACK data structure and 32-bit arithmetic are used, then the time required to form Ax when A has at most k nonzero entries per row is

$$(k-1)(51+M/4)+k(52+M/4)+k(60+5M/4)=-51+163k+7kM/4-M/4.$$

When full diagonals are stored, the gathers are not needed, and the time is

$$(k-1)(51+N/4)+k(52+N/4)=-51+103k+2kN/4-N/4.$$

Thus for $N \ll 1$, the ITPACK data structure is more efficient when

$$\frac{M}{N} < \frac{2k-1}{7k-1}$$

For $k=1$, 2, 3, 4, 5, and ∞, the approximate 'cross-over points' are N=6M, 4.3M, 4M, 3.8M, 3.7M, and 3.5M, respectively. Thus, storage by full diagonals is more efficient even when one has to more than

triple the number of unknowns.

8. Other savings. In addition to improving the efficiency of residual calculations and the calculations in all of the basic iterative schemes, other improvements in efficiency of ITPACK can be made.

For example, routine SCAL rearranges entries in COEF so that the main diagonal is in the first column of COEF and SCAL also scales to get $D^{-1/2}AD^{-1/2}$. Currently this routine is strictly scalar. Use of bit-vectors, WHERE blocks, and so on, could reduce the execution time of SCAL. Moreover, the study and discovery necessary to vectorize SCAL (and other ITPACK subprograms) will be beneficial for learning how to vectorize.

The following is a sketch of a vectorized SCAL:

```
Form vector with entries 1, 2, ··· , N (equation numbers)
IDAG(1;N) = Q8VINTL(1,1;IDIAG(1;N))
Locate main diagonal entries in 1st column of COEF
BIT(1;N) = JCOEF(1,1;N) .EQ. IDIAG(1;N)
See if all of main diagonal is in 1st column (count the 1's in BIT)
ISUM = Q8SCNT (BIT(1;N))
IF(ISUM .EQ. N) THEN check other columns
    DO J=2, MAXNZ
        BIT(1;N) = JCOEF(1,J;N) .EQ. IDIAG(1;N)
        ISUM = ISUM + Q8SCNT (BIT(1;N))
    CONTINUE
    IF(ISUM .GT. N) error exit: too many main diagonals entries
ELSE have to rearrange COEF
    DO J=2, MAXNZ
        Find main diagonal entries in column J
        BIT(1;N) = JCOEF(1,J;N) .EQ. IDIAG(1;N)
        JSUM = Q8SCNT (BIT(1;N))
        IF(SUM .NE. 0) THEN
            ISUM = ISUM + JSUM
            WHERE (BIT(1;N)) interchange
                TEMP(1;N) = COEF(1,J;N)
                COEF(1,J;N) = COEF(1,I;N)
                COEF(1,1;N) = TEMP(1;N)
                ITEMP(1;N) = JCOEF(1,J;N)
                JCOEF(1,J;N) = JCOEF(1,1;N)
                JCOEF(1,1;N) = ITEMP(1;N)
            END WHERE
        END IF
    CONTINUE
    IF(ISUM .LT. N .OR. N .LT. ISUM) error exit: can't find full main diagonal
    check to see if two main diagonals entries are in one equation
    BIT(1;N) = JCOEF(1,1;N) .EQ. IDIAG(1;N)
```

```
      ISUM = Q8SCNT (BIT(1;N))
      IF(ISUM .NE. N) error exit
   END IF
   set zero values in JCOEF to unity
   DO J=2, MAXNZ
      BIT(1;N) = JCOEF(1,J;N) .EQ. 0
      WHERE (BIT(1;N))
         JCOEF(1,J;N) = 1
      END WHERE
   CONTINUE
   check for zero diagonal entry
   BIT(1;N) = COEF(1,1;N) .EQ. 0.
   ISUM = Q8SCNT(BIT(1;N))
   IF(ISUM .GT. 0) error exit
   change sign of equations with negative main diagonal
   BIT(1;N) = COEF(1,1;N) .LT. 0.
   ISUM = Q8SCNT(BIT(1;N))
   IF(ISUM .GT. 0) THEN have to change signs
      WHERE (BIT(1;N))
         RGHTSD(1;N) = – RGHTSD(1;N)
      END WHERE
      DO J=1, MAXNZ
         WHERE (BIT(1;N))
            COEF(1,J;N) = – COEF(1,J;N)
         END WHERE
      CONTINUE
   END IF
   Store reciprocal square root of main diagonal
   COEF(1,1;N) = 1. / SQRT (COEF(1;N))
   multiply to get diag( – 1/2)A
   DO J=2, MAXNZ
      COEF(1,J;N) = COEF(1,1;N) * COEF(1,J;N)
   CONTINUE
   multiply to get diag( – 1/2)A diag( – 1/2)
   DO J=2, MAXNZ
      TEMP(1;N) = G8VGATHR(COEF(1,1;N), JCOEF(1,J,N); TEMP(1;N))
      COEF(1,J;N) = COEF(1,J;N) * TEMP(1;N)
   CONTINUE
```

9. Comparison of ITPACK and direct methods. Kincaid-Oppe-Young [1984] give times to solve the linear system (to accuracy 10^{-5}) for the standard 5-point approximation of

$$u_{xx}+2u_{yy}=0 \quad \text{on } \Omega, \quad u=1+xy \quad \text{on } \partial\Omega,$$

where Ω is a unit square. We compare these with times given for direct methods in the CDC MAGEV

(Math/Geophysical Vector library) manual. Table 9 list N, t=time, t/N= time per unknown, and

$$\nu = \frac{\log(\frac{t_j/N_j}{t_{j-1}/N_{j-1}})}{\log(\frac{N_j}{N_{j-1}})},$$

where ν is the observed value of ν in $t/N = O(N^{\nu})$.

The MAGEV routines GEL and HGEL use scaled Gauss Elimination with partial pivoting for full matrices using 64-bit and (H) 32-bit arithmetic, respectively. Since asymptotically, $N^3/3$ operations (* and +), one expects $\nu = 2$.

Routines BDGEL and HBDGEL (64-bit and 32-bit) use unscaled Gauss elimination (no pivoting) for Band matrices. The data given in the MAGEV manual are for matrices with half-bandwidth $\beta = N/10$. Asymptotically, band elimination uses $N\beta^2$ operations; hence one expects $\nu = 2$.

TRID and HTRID use cyclic reduction to solve TRIDiagonal systems and the number of operations is asymptotically proportional to N; one expects $\nu = 0$. The efficiencies of these increase when they solve a collection of (different) tridiagonal systems all with the same number of unknowns. This is because the coefficient matrices can be regarded as the square blocks of a block diagonal matrix; if there are k blocks, then one processes vectors of length kN, and efficiency increases as the vector length increases. Two-dimensional red-black line methods for 5- and 9-point difference approximations might make use of these routines.

We emphasize that *these results are for four different types of matrices.* (H)GEL solve full-matrix systems. The data for (H)BGEL are for matrices with half-bandwidth $\beta = N/10$. The ITPACK results are for 5-diagonal matrices with half-band width $\beta = \sqrt{N}$. (H)TRID solves tridiagonal systems.

Figure 1 shows graphs of $\log(t/N)$ versus $\log(N)$. Figure 2 shows graphs of ν versus $\log(N)$. In these figures the following abbreviations are used

64-bit				32-bit		
F	(Full)	=	GEL;	f	=	HGEL;
B	(Band)	=	BGEL;	b	=	HBGEL;
T	(Tridiagonal)	=	TRID;	t	=	HTRID;
				e	=	(time for b)*100/N [see below].

As usual, $\kappa = O(h^{-2}) = O(N) = O(h^2)$ denotes the condition number and $R = -\rho(A)$ is the asymptotic rate of convergence. CG (Conjugate Gradient) methods. Except for SOR, asymptotically the error reduction factor, $\|e^{(m)}\| / \|e^{(0)}\|$, is e^{-Rm}; for optimum SOR it is $m\rho(A)^{m-1}$.

The values of κ in the table above are taken from Birkhoff-Lynch [1984, Table 1, p. 165]. The values of κ and R for the Chebyshev acceleration methods (the ITPACK "SI" [Semi-Iterative] schemes) also apply to the corresponding CG (Conjugate Gradient) methods. Except for SOR, asymptotically the error reduction factor, $\|e^{(m)}\| / \|e^{(0)}\|$, is e^{-Rm}; for optimum SOR it is $m\rho(A)^{m-1}$.

If the observed ν for the direct method is, asymptotically, the theoretical ν, then Figure 2 suggests that one is close to the asymptotic value at N about 10,000 for the full-matrix routines (H)GEL and N about 50,000, or perhaps much larger, for the band-matrix routines (H)BGEL ($\beta = N/10$). For the

Natural	Red-black	Method	expected ν	R = Rate
JCG	jcg	= Jacobi, CG	0.5	–
JSI	jsi	= Jacobi, SI	0.5	$2/\kappa^{1/2}$
SOR	sor	= SOR	0.5	$4/\kappa^{1/2}$
	rcg	= Reduced System CG	0.5	–
	rsi	= Reduced System SI	0.5	–
SCG	scg	= SSOR CG	0.25	–
SSI	ssi	= SSOR SI	0.25	$2^{3/2}/\kappa^{1/4}$ natural

tridiagonal solvers (H)TRID, it appears to be about $N = 100{,}000$.

We now give our interpretation of the graphs in Figures 1 and 2.

Consider (H)TRID: The time per unknown decreases as the vector length increases because the efficiency of Cyber 205 arithmetic increases as N increases. The time per unknown decreases by more than a factor of 10 when N changes from 16 to 8192. The number of arithmetic operations is proportional to N, and, as expected, ν tends to a constant.

Also shown on Figure 1, is the time per unknown for solving 2, 3, 8, and 64 (different) tridiagonal systems with 256 and 512 unknowns. The time per unknown for a pair of systems with 256 unknowns is just about equal the the time per unknown for a single system with 512 unknowns. Note also, that the time per unknowns decreases very rapidly when the numbers of systems is small and then reaches an asymptotic value: only a small number of systems gives a large portion of the potential savings.

Consider the graphs on Figure 2 for (H)GEL and (H)BGEL: Since the observed values of ν are considerably smaller than the asymptotic value, one gaining significantly by using Cyber 205 arithmetic instead of scalar arithmetic. For (H)GEL, the observed ν are between 1.2 and 1.7; i.e., these routines behave as if their operation counts were $O(N^{2.2})$ to $O(N^{2.7})$, instead of the $O(N^3)$, which one gets from and operation count for Gauss elimination.

For (H)BGEL, the 'effective operation count' is even smaller. For the range of values we have, these routines behave as if their operations counts were between $O(N^{1.9})$ and $O(N^{2.2})$ -- an order of magnitude smaller than that for conventional Gauss elimination's $O(N^3)$.

Consider the ITPACK routines: In the natural ordering SOR, SSOR CG, and SSOR SI are not vectorizable and one would not expect any effect on them due to the Cyber 205 vector arithmetic. The observed values of ν for these routines are close the the expected values. On the other hand, the vectorizable routines have values of ν increasing as N increases; for these the Cyber vector arithmetic is significantly increasing their efficiencies, as it does with the direct solvers we consider.

For $N = 65025$ ($h = 1/256$), Kincaid-Oppe-Young [1984, p. 18] list values of iteration time as 29.369 for Jacobi SI and, in the entry immediately below, 32.334 for SOR. Figure 2 and the entries for ν in Table 9 for these methods suggests that these times should be interchanged in their table on p. 18.

The "Estimated" Band Solver. Since $\sqrt{N} = N/10$ when $N = 100$, the results for (H)BGEL for $N = 100$ are directly comparable with those for ITPACK. Theoretical, values for solve time, t, from (H)BGEL for matrices with half-band width $\beta = \sqrt{N}$ would be less than those given in the Tables by a

factor of of about N/100, for N > 100; however, we do not know how the Cyber 205 arithmetic affects this factor. The values of (time for HBGEL)*100/N are indicated by the graphs marked "e" (= estimated) in Figure 1. These give *estimates* for the times that HBGEL would solve systems like those solved by ITPACK. The graph is almost horizontal -- this is consistent with BGEL having an 'effective operations count' of about $O(N^2)$, because the time per unknown is about $O(N)$ for the range of data we have. From the behavior of the observed v for HBGEL, we expect the routine to be close to its asymptotic performance for N about 100,000.

Except for RS CG, SSOR CG, and RS SI, all in the red-black ordering, the ITPACK routine are less efficient than this (theoretical) band solver HBGEL.

We emphasize that the graph labeled "e" is *not* from experimental data. We suggest that some experiments be performed with HBEL solving the same linear system that ITPACK solved for N between 100 and 2000.

10. References.

Birkhoff, G., and R. E. Lynch [1984], *Numerical Solution of Elliptic Problems*, SIAM Publications, 1984.

Concus, P., G. H. Golub, and D. P. O'Leary [1976], "A generalized conjugate gradient method for the numerical solution of partial differential equations," in *Sparse Matrix Computations*, J. R. Bunch and D. J. Rose (eds.), 309mi332, Academic Press.

Hockney, R. W., and C. R. Jesshope [1981], *Parallel Computers*, Adam Hilger Ltd, Bristol.

Kincaid, D. R., T. C. Oppe, and D. M. Young [1982], "Adapting ITPACK routines for use on a vector computer", Report CNA-177, Center for Numerical Analysis, Univ. of Texas.

Kincaid, D. R., T. C. Oppe, and D. M. Young [1984], "Vector computations for sparse linear systems", Report CNA-189, Center for Numerical Analysis, Univ. of Texas.

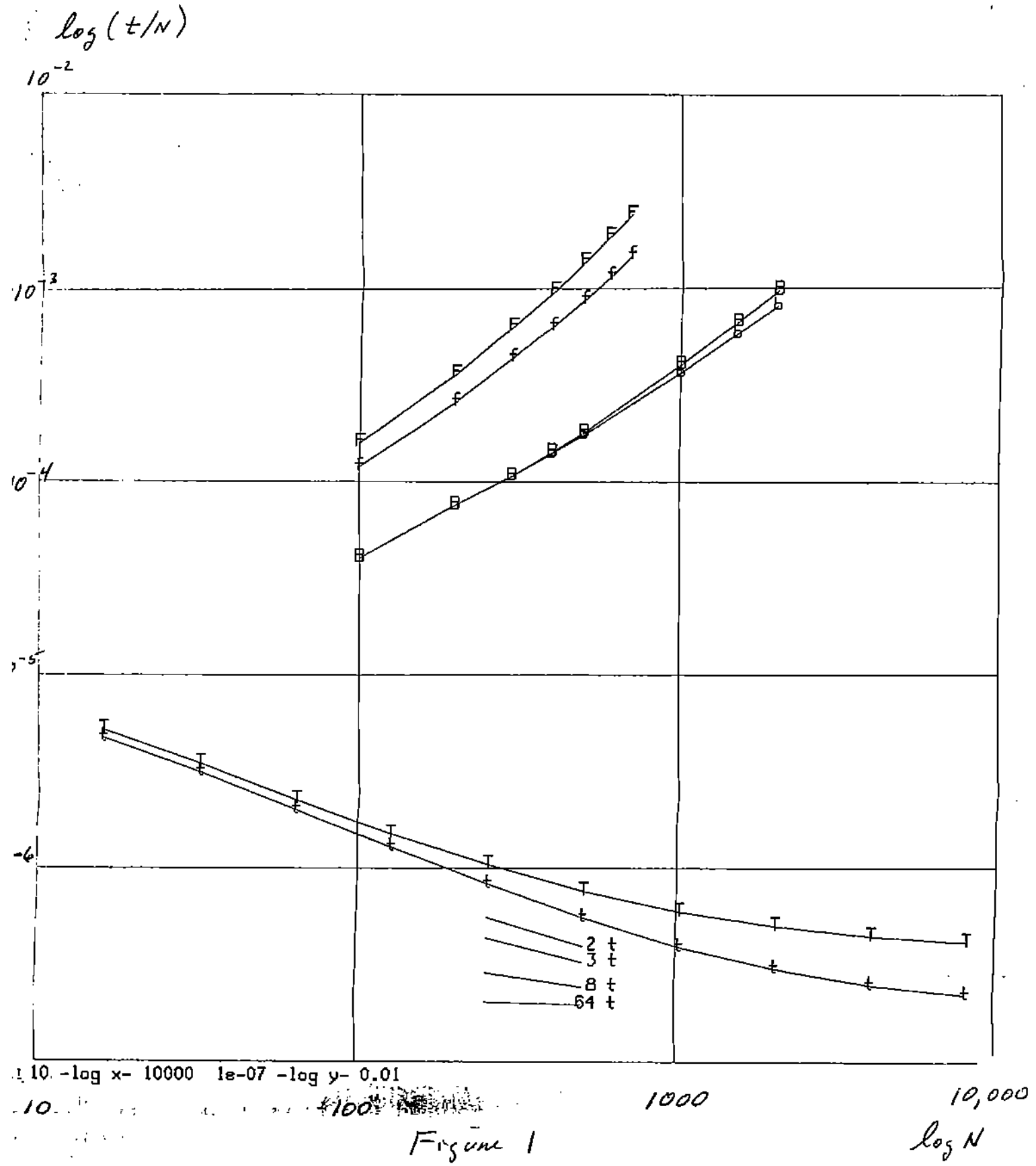
log (t/N)
10^-2
10^-3
10^-4
10^-5
-6
2 t
3 t
8 t
64 t
10 -log x- 10000 1e-07 -log y- 0.01
10
100
1000
10,000
log N

Figure 1

$$\nu = \frac{-\log(t_{j-1}/N_{j-1}) + \log(t_j/N_j)}{-\log(N_{j-1}) + \log(N_j)}$$

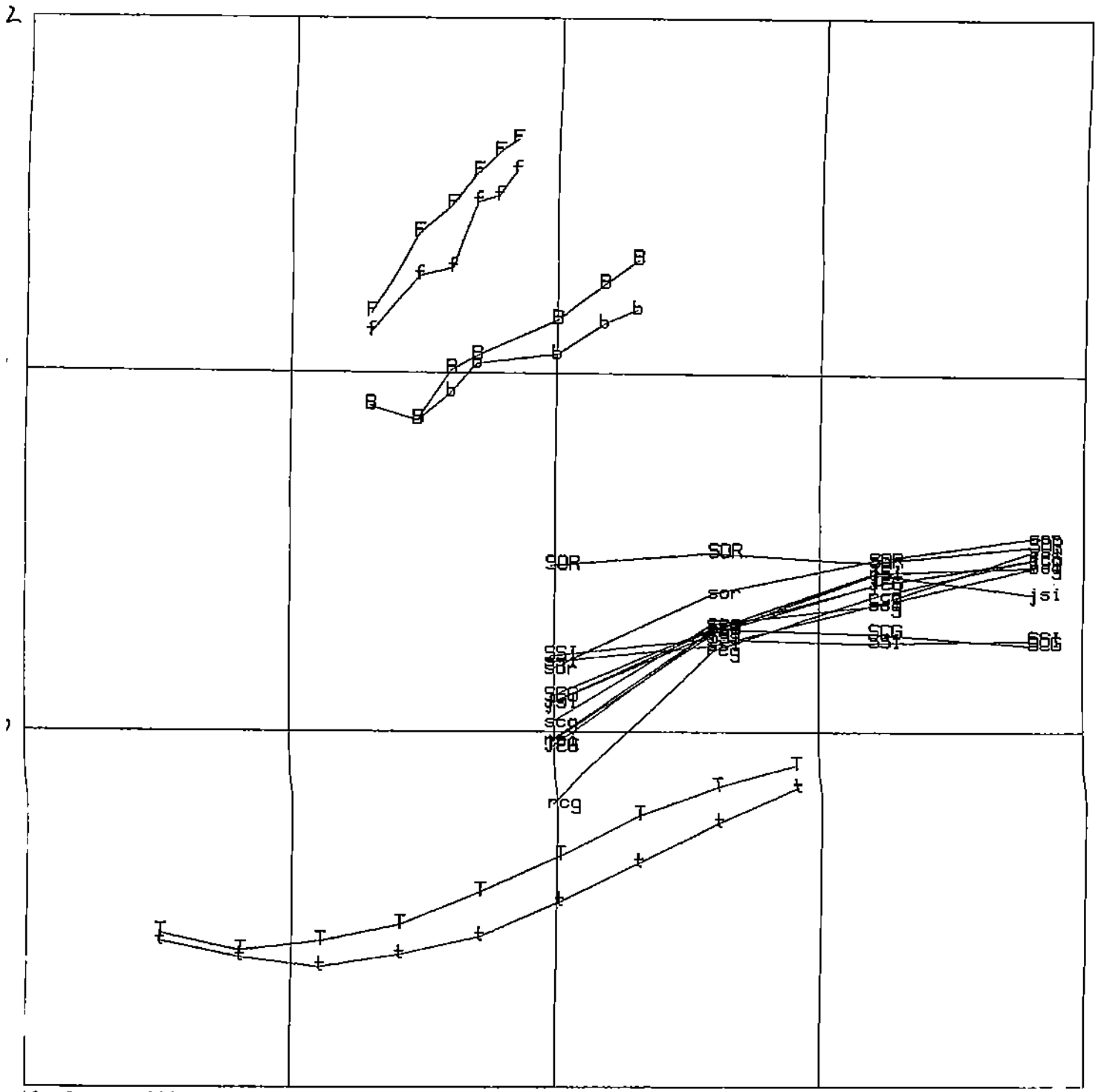

Figure 2

Table 9. Experimental results

N	*t* seconds	*t/N* micro seconds	ν	*t* seconds	*t/N* micro seconds	ν
		F			f	
100	0.016	160.000	0.	0.012	120.000	0.
200	0.072	360.000	1.170	0.052	260.000	1.115
300	0.190	633.333	1.393	0.131	436.667	1.279
400	0.387	967.500	1.473	0.254	635.000	1.302
500	0.686	1372.000	1.565	0.442	884.000	1.483
600	1.107	1845.000	1.625	0.697	1161.667	1.498
700	1.668	2382.857	1.660	1.036	1480.000	1.571
		B			b	
100	0.004	40.000	0.	0.004	40.000	0.
200	0.015	75.000	0.907	0.015	75.000	0.907
300	0.032	106.667	0.869	0.032	106.667	0.869
400	0.057	142.500	1.007	0.056	140.000	0.945
500	0.090	180.000	1.047	0.088	176.000	1.026
1000	0.400	400.000	1.152	0.365	365.000	1.052
1500	0.999	666.000	1.257	0.870	580.000	1.142
2000	1.950	975.000	1.325	1.630	815.000	1.182
		T			t	
16	0.000083	5.188	0.	0.000076	4.750	0.
32	0.000112	3.500	-0.568	0.000101	3.156	-0.590
64	0.000146	2.281	-0.618	0.000130	2.031	-0.636
128	0.000194	1.516	-0.590	0.000164	1.281	-0.665
256	0.000266	1.039	-0.545	0.000212	0.828	-0.630
512	0.000389	0.760	-0.452	0.000284	0.555	-0.578
1024	0.000610	0.596	-0.351	0.000407	0.397	-0.481
2048	0.001034	0.505	-0.239	0.000630	0.308	-0.370
4096	0.001851	0.452	-0.160	0.001052	0.257	-0.260
8192	0.003458	0.422	-0.098	0.001880	0.229	-0.162
		JCG			jcg	
225	0.010	44.444	0.	0.010	44.444	0.
961	0.040	41.623	-0.045	0.041	42.664	-0.028
3969	0.247	62.232	0.284	0.250	62.988	0.275
16129	1.789	110.918	0.412	1.809	112.158	0.411
65025	14.114	217.055	0.482	14.304	219.977	0.483
		JSI			jsi	
225	0.019	84.444	0.	0.019	84.444	0.
961	0.091	94.693	0.079	0.091	94.693	0.079
3969	0.554	139.582	0.274	0.562	141.597	0.284
16129	4.169	258.479	0.439	4.244	263.129	0.442
65025	28.730	441.830	0.385	29.369	451.657	0.388

N	t seconds	t/N micro seconds	ν	t seconds	t/N micro seconds	ν
	SOR			sor		
225	0.035	155.556	0.	0.012	53.333	0.
961	0.292	303.850	0.461	0.066	68.678	0.174
3969	2.442	615.268	0.497	0.471	118.670	0.386
16129	19.271	1194.804	0.473	3.749	232.438	0.479
65025	160.399	2466.728	0.520	32.334	497.255	0.545
	SCG			scg		
225	0.027	120.000	0.	0.007	31.111	0.
961	0.133	138.398	0.098	0.031	32.258	0.025
3969	0.827	208.365	0.288	0.196	49.383	0.300
16129	4.950	306.901	0.276	1.305	80.910	0.352
65025	28.156	433.003	0.247	10.004	153.849	0.461
	SSI			ssi		
225	0.028	124.444	0.	0.020	88.889	0.
961	0.162	168.574	0.209	0.113	117.586	0.193
3969	0.966	243.386	0.259	0.655	165.029	0.239
16129	5.579	345.899	0.251	4.379	271.499	0.355
65025	32.250	495.963	0.258	35.820	550.865	0.508
				rcg		
225				0.006	26.667	0.
961				0.019	19.771	-0.206
3969				0.108	27.211	0.225
16129				0.748	46.376	0.380
65025				5.935	91.273	0.486
				rsi		
225				0.008	35.556	0.
961				0.033	34.339	-0.024
3969				0.204	51.398	0.284
16129				1.544	95.728	0.444
65025				11.870	182.545	0.463