

# **On Efficient Storage of Sparse Matrices**

**Shahadat Hossain**

Department of Mathematics and Computer Science

University of Lethbridge, Canada

*Computing by the Numbers: Algorithms, Precision, and Complexity*

*Matheon Workshop 2006 in honor of the 60th birthday of Richard Brent*

*Weierstrass Institute for Applied Analysis and Stochastics*

*Berlin*

## Outline

- Introduction and background
- Sparse matrix storage schemes: Compressed Row, Jagged Diagonal
- Performance Issues
- Concluding remarks and future directions

## Challenges in Large-scale Numerical Computing

### High Performance Computing Challenges

- Climate Models e.g. Community Climate System Model
- High-temperature Superconductor model calculation

### Supercomputer Systems

- Superscalar e.g., SGI Altix, IBM Power and Power4
- Parallel vector supercomputers e.g., Earth Simulator, Cray X1

**Widening gap between sustained and peak performance!!**

## Krylov Subspace Methods

Solve for  $x \in R^n$

$$Ax = b$$

where  $A \in R^{n \times n}$  is large and sparse.

Main computational tasks in a Krylov subspace method (e.g. Conjugate Gradient)

1. **sparse matrix times vector (MVP):**  $w = Av$
2. dot product:  $c = x^T y$
3. vector update (saxpy):  $y = y + \alpha x, \alpha \in R$
4. preconditioning operation

## Exploiting Sparsity

Target architecture: Vector supercomputers e.g. The Earth Simulator, CRAY X1

1. **Regular structure:** e.g., banded: Store the diagonals; can be implemented efficiently with excellent performance (flops).
2. **General structure:** Either adapt the diagonal storage scheme or reorder the matrix to obtain a diagonal structure.

Observation:

**On vector machines long vectors can improve the performance of MVP calculation significantly**

## Compressed Row Storage (CRS)

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & 0 & 0 & 0 \\ a_{20} & 0 & a_{22} & 0 & 0 \\ a_{30} & 0 & 0 & a_{33} & 0 \\ a_{40} & 0 & 0 & 0 & a_{44} \end{pmatrix} \quad \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & 0 & 0 & 0 \\ a_{20} & a_{22} & 0 & 0 & 0 \\ a_{30} & a_{33} & 0 & 0 & 0 \\ a_{40} & a_{44} & 0 & 0 & 0 \end{pmatrix}$$

(a) (b)

value	$a_{00}$	$a_{01}$	$a_{02}$	$a_{03}$	$a_{04}$	$a_{10}$	$a_{11}$	$a_{20}$	$a_{22}$	$a_{30}$	$a_{33}$	$a_{40}$	$a_{44}$
colind	0	1	2	3	4	0	1	0	2	0	3	0	4
rowptr	0	5	7	9	11	13							

(c)

**Figure 1:** (a) Sparsity pattern of the  $6 \times 6$  “arrow-head” matrix. (b) The “arrow-head” matrix after CRS compression. (c) Compressed Row Storage (CRS) data structure for the “arrow-head” matrix.

## MVP in CRS

```
for (int i = 0; i < m ; i++) {  
    upper = rowptr[i+1]; // fetch the upper index  
    // loop over row i  
    for (int j = rowptr[i]; j < upper; j++) {  
        b[i] = b[i] + value[j] * v[colind[j]];  
    }  
}
```

Figure 2: Matrix-vector multiplication in CRS scheme

**Most rows contain only few nonzero entries ( $\rho_i$ ) compared with the matrix dimension –  $\rho_i \ll n$  implies shorter vector in the inner for-loop.**

### Jagged Diagonal Storage(JDS)

$$\begin{pmatrix} a_{00} & 0 & 0 & a_{03} & 0 \\ a_{10} & a_{11} & 0 & 0 & 0 \\ a_{20} & 0 & a_{22} & 0 & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & 0 & 0 & 0 & a_{44} \end{pmatrix} \quad \begin{pmatrix} a_{00} & a_{03} & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 & 0 \\ a_{20} & a_{22} & 0 & 0 & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{44} & 0 & 0 & 0 \end{pmatrix}$$

(a) (b)

value	$a_{30}$	$a_{10}$	$a_{20}$	$a_{00}$	$a_{40}$	$a_{31}$	$a_{11}$	$a_{22}$	$a_{03}$	$a_{44}$	$a_{32}$	$a_{33}$	$a_{34}$
colind	0	0	0	0	1	1	2	3	4	2	3	4	
jdptr	0	5	10	11	12	13							
perm	3	1	2	0	4								

(c)

**Figure 3:** (a) Sparsity pattern of the  $6 \times 6$  row permuted “arrow-head” matrix. (b) The row-permuted “arrow-head” matrix after JDS compression. (c) Jagged diagonal storage (JDS) data structure for the “arrow-head” matrix.

## MVP in JDS

```
for (int j = 0; j < rho_max ; j++) {  
    upper = jdp[tr][j+1]; // fetch the upper index  
    // loop over jagged diagonal j  
    for (int i = jdp[tr][j]; i < upper; i++) {  
        col = colind[i];  
        b[i] = b[i] + value[i] * v[col];  
    }  
}
```

Figure 4: Matrix-vector multiplication in JDS scheme

- **Needs an  $m$ -vector to store the original row order.**
- **Extra work to restore the correct order in the result .**

## Transposed Jagged Diagonal Storage (TJDS)

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{22} & a_{33} & a_{44} \\ a_{20} & 0 & 0 & 0 & 0 \\ a_{30} & 0 & 0 & 0 & 0 \\ a_{40} & 0 & 0 & 0 & 0 \end{pmatrix}$$

(a)

value	$a_{00}$	$a_{01}$	$a_{02}$	$a_{03}$	$a_{04}$	$a_{10}$	$a_{11}$	$a_{22}$	$a_{33}$	$a_{44}$	$a_{20}$	$a_{30}$	$a_{40}$
rowind	0	0	0	0	1	1	2	3	4	2	3	4	
tjdptr	0	5	10	11	12	13							

(b)

Figure 5: (a) The “arrow-head” matrix after the columns have been compressed. (b) Transposed jagged diagonal storage (TJDS) data structure for the “arrow-head” matrix.

## MVP in TJDS

```
for (int i = 0; i < col-max ; i++) {  
    upper = tjdp[tr[i+1]]; // fetch the upper index  
    ind = 0; //index to iterate over v's elements  
    // loop over jagged diagonal j  
    for (int j = tjdp[tr[i]]; j < upper; j++) {  
        row = rowind[j];  
        b[row] = b[row] + value[j] * v[ind];  
        ind++;  
    }  
}
```

Figure 6: Matrix-vector multiplication in TJDS scheme

**No need for a permutation vector in TJDS scheme**

## **Bi Jagged Diagonal Storage (Bi-JDS) – New Sparse Matrix Storage Scheme**

1. JDS and TJDS are suitable for MVP calculation on vector machines.
2. The number of nonzero entries in each row may vary considerably for general sparse matrices e.g. the arrow-head matrix under JDS or TJDS is stored as many short jagged diagonals.

The **Bi-JDS scheme** partitions a sparse matrix into two segments and compresses the matrix in both column and row directions.

### Bi-JDS Data Structure

$$\begin{matrix}
 \left( \begin{array}{ccccc}
 a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\
 a_{10} & a_{11} & 0 & 0 & 0 \\
 a_{20} & 0 & a_{22} & 0 & 0 \\
 a_{30} & 0 & 0 & a_{33} & 0 \\
 a_{40} & 0 & 0 & 0 & a_{44}
 \end{array} \right) & \begin{matrix} \uparrow \\ \text{column compression} \end{matrix} & \left( \begin{array}{ccccc}
 a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\
 a_{10} & a_{11} & 0 & 0 & 0 \\
 a_{20} & a_{22} & 0 & 0 & 0 \\
 a_{30} & a_{33} & 0 & 0 & 0 \\
 a_{40} & a_{44} & 0 & 0 & 0
 \end{array} \right) \\
 \leftarrow \text{row compression} & & 
 \end{matrix}$$

(a) (b)

value	$a_{00}$	$a_{10}$	$a_{20}$	$a_{30}$	$a_{40}$	$a_{01}$	$a_{11}$	$a_{22}$	$a_{33}$	$a_{44}$	$a_{02}$	$a_{03}$	$a_{04}$
row_col_ind	0	0	0	0	0	1	1	2	3	4	0	0	0
tjdptr	11	14											
row_min	2												
rcol_max	1												
rcol_min	1												

(c)

**Figure 7:** (a) Bi-JDS partition of the  $6 \times 6$  “arrow-head” matrix. (b) The “arrow-head” matrix after column and row compression. (c) Bi-jagged diagonal (Bi-JDS) data structure for the “arrow-head” matrix

## MVP with Bi-JDS

```
//The jagged diagonals are full-length; so jptr array is not needed.
for (int j = 0; j < rho_min ; j++) {
    ind = 0;
    for (i = m*j; i < m*(j+1); i++) {
        col = row_colind[i];
        b[ind] = b[ind] + value[i]*v[col];
        ind++;
    }
}
// Now update the product with the contribution from transposed jagged diagonals.
k = 0;
for (int i = 0; i < rcol_max; i++){
    ind=minrow+1;
    for (int indtjd=tjdptr[i]; indtjd < tjdptrr[i+1]; indtjd++){
        row = row_colind[k];
        b[row] = b[row] + val[indtjd]*v[ind];
        ind=ind+1;
        k=k+1;
    }
}
```

## Storage Requirements

$$\text{memory}_{\text{JDS}}: \quad 2nnz(A) + \rho_{max} + 1 + m$$

$$\text{memory}_{\text{TJDS}}: \quad 2nnz(A) + \kappa_{max} + 1$$

$$\text{memory}_{\text{BiJDS}}: \quad 2nnz(A) + n_{\text{tjd}} + 1$$

where

$$n_{\text{tjd}} = (\kappa_{max} - \kappa_{min})$$

$\rho_{max}$  = maximum number of nonzero entries in any row of  $A$ ,

$\kappa_{max}$  = maximum number of nonzero entries in any column of  $A$ , and

$\kappa_{min}$  = minimum number of nonzero entries in any column of  $A$ .

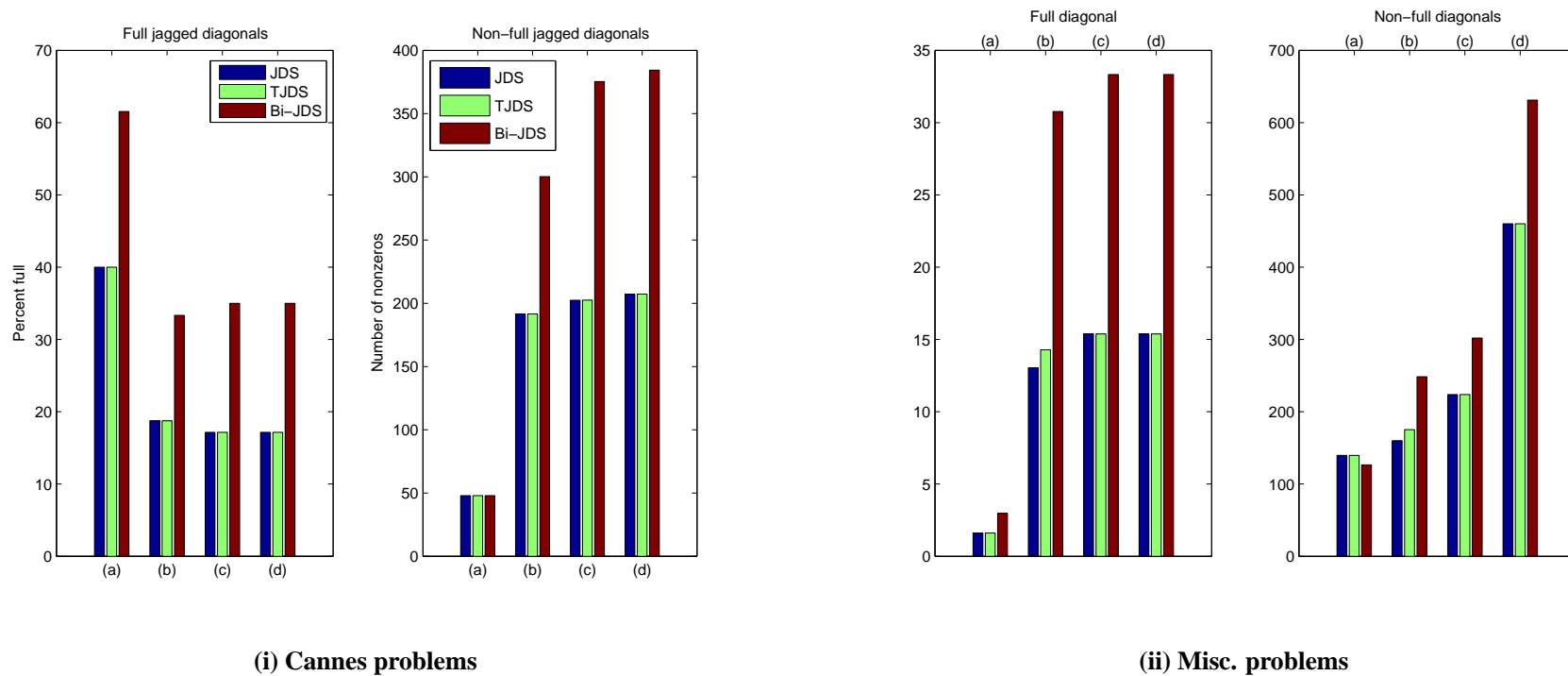
## Computational Experiments

Table 1: The Harwell-Boeing Test Problems Info.

Nr.	Name	$m$	$n$	$nnz$	$\rho_{max}$	$\rho_{min}$	$\kappa_{max}$	$\kappa_{min}$
a.	can144	144	144	720	15	6	15	6
b.	can838	838	838	5424	32	6	32	6
c.	can1054	1054	1054	6625	35	6	35	6
d.	can1072	1072	1072	6758	35	6	35	6
e.	add20	2395	2395	17319	124	2	124	2
f.	bfw398a	398	398	3678	23	3	21	3
g.	bfw398b	398	398	2910	13	2	13	2
h.	bfw782a	782	782	5982	13	2	13	2

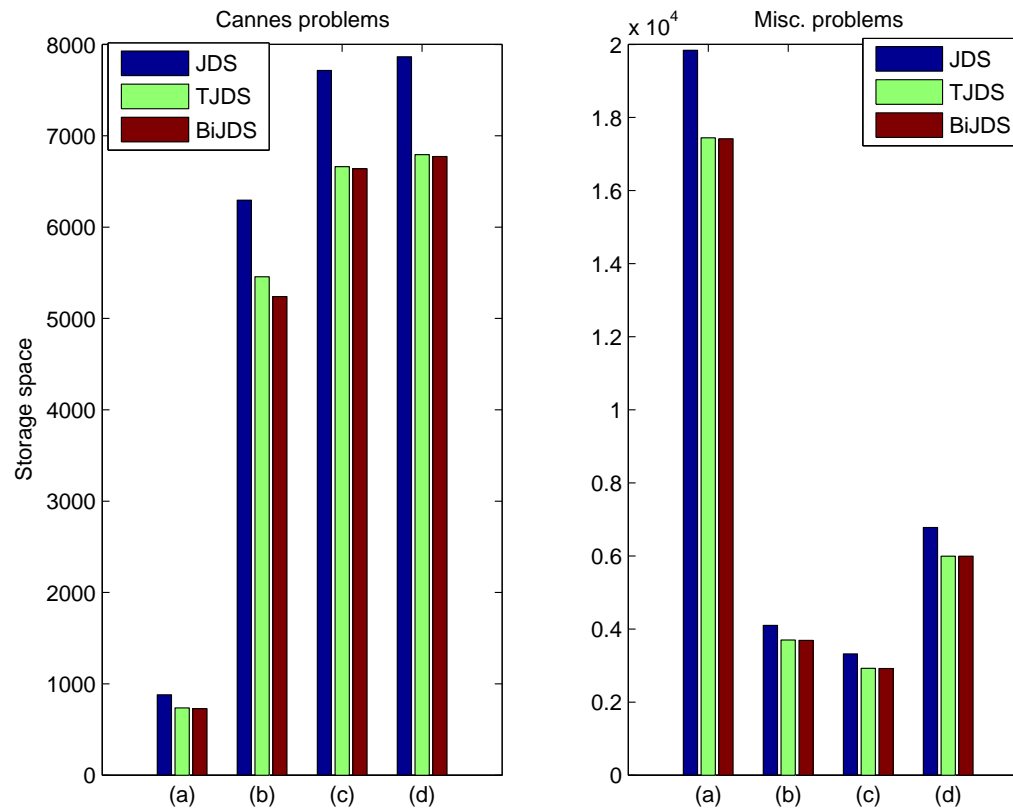
## Computational Experiments

Figure 8: Properties of the jagged diagonals in sparse storage schemes.



## Computational Experiments

Figure 9: Estimate of storage requirements in sparse storage schemes.



## Concluding Remarks

1. The BiJDS scheme produces more full-length jagged diagonals as well as longer non-full jagged diagonals compared with JDS or TJDS scheme.
2. The BiJDS scheme is also space efficient and does not need to store the permutation order of the rows of the matrix.
3. Can be combined with blocking strategy to reduce indirect memory access.
4. Extensive numerical tests on practical problems.