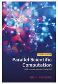
Sparse Matrices and Their Data Structures

Section 4.2 of Parallel Scientific Computation, 2nd edition

Rob H. Bisseling

Utrecht University

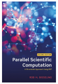


Basic sparse technique: adding two sparse vectors

- Problem: add a sparse vector **y** of length n to a sparse vector **x** of length n, overwriting **x**, i.e.,

$$\mathbf{x} := \mathbf{x} + \mathbf{y}.$$

- '**x** is a sparse vector' means that $x_i = 0$ for most i.
- The number of nonzeros of **x** is c_x and that of **y** is c_y.

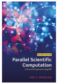


Example: store sparse vectors in compressed form

- Given are vectors $\mathbf{x}$, $\mathbf{y}$ of length $n = 8$ in a compressed vector data structure:

$x[j].a =$	2	5	1	
$x[j].i =$	5	3	7	
$y[j].a =$	1	4	1	4
$y[j].i =$	6	3	5	2

- The number of nonzeros of these vectors is $c_x = 3$ and $c_y = 4$.
- The jth nonzero in the array of $\mathbf{x}$ has
  - numerical value $x_i = x[j].a$,
  - index $i = x[j].i$.
- How to compute $\mathbf{x} + \mathbf{y}$?



Addition is easy for dense storage

- A compressed vector data structure for $\mathbf{x}$, $\mathbf{y}$ is:

$x[j].a =$	2	5	1
$x[j].i =$	5	3	7

$y[j].a =$	1	4	1	4
$y[j].i =$	6	3	5	2

- The dense vector data structure for $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z} = \mathbf{x} + \mathbf{y}$ is:

0	0	0	5	0	2	0	1

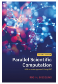
0	0	4	4	0	1	1	0

0	0	4	9	0	3	1	1

- A compressed vector data structure for $\mathbf{z} = \mathbf{x} + \mathbf{y}$ is:

$z[j].a =$	3	9	1	1	4
$z[j].i =$	5	3	7	6	2

- Conclusion: use an auxiliary dense vector!



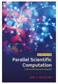
Location array

- The array *yloc* registers the location $j = yloc[i]$ where a nonzero vector component y_i is stored in the compressed array.
- It registers a dummy value -1 if y_i is not stored.
- *yloc* is similar to the inverse of a permutation:

$$yloc[\, y[j].i \,] = j.$$

$y[j].a =$	1	4	1	4
$y[j].i =$	6	3	5	2
$j =$	0	1	2	3

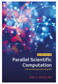
$y_i =$	0	0	4	4	0	1	1	0
$yloc[i] =$	-1	-1	3	1	-1	2	0	-1
$i =$	0	1	2	3	4	5	6	7



Algorithm for sparse vector addition: pass 0

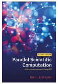
input: $\mathbf{x}$: sparse vector with $c_x \geq 0$ nonzeros, $\mathbf{x} = \mathbf{x}_0$,
$\mathbf{y}$: sparse vector with $c_y \geq 0$ nonzeros,
$yloc$: dense vector of length n,
$yloc[i] = -1$, for $0 \leq i < n$.
output: $\mathbf{x} = \mathbf{x}_0 + \mathbf{y}$, $yloc[i] = -1$, for $0 \leq i < n$.

```
{ Register location of nonzeros of $\mathbf{y}$ }
for $j := 0$ to $c_y - 1$ do
    $yloc[y[j].i] := j$;

...
```



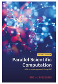
Algorithm for sparse vector addition: passes 1, 2

```
...
{ Add matching nonzeros of x and y }
for $j := 0$ to $c_x - 1$ do
    $i := x[j].i$;
    if $yloc[i] \neq -1$ then
        $x[j].a := x[j].a + y[yloc[i]].a$;
        $yloc[i] := -1$;

{ Append remaining nonzeros of y to x }
for $j := 0$ to $c_y - 1$ do
    $i := y[j].i$;
    if $yloc[i] \neq -1$ then
        $x[c_x].i := i$;
        $x[c_x].a := y[j].a$;
        $c_x := c_x + 1$;
        $yloc[i] := -1$;
```



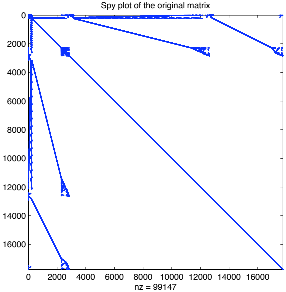
Analysis of sparse vector addition

- The total number of operations is $\mathcal{O}(c_x + c_y)$, since there are $c_x + 2c_y$ loop iterations, each with a small constant number of operations.
- The number of flops equals the number of nonzeros in the intersection of the sparsity patterns of **x** and **y**; 0 flops can happen.
- The initialization of array *yloc* to -1 costs n operations, which will dominate the total cost if only one vector addition has to be performed.
- *yloc* can be reused in subsequent vector additions, because each modified element $yloc[i]$ is reset to -1.
- If we add two $n \times n$ matrices row by row, we can amortize the $\mathcal{O}(n)$ initialization cost over n vector additions.



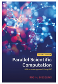
Accidental zero

`https://www.filmsite.org/greatestflops14.html`



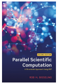
Matrix `memplus` with $n = 17\,758$ and $126\,150$ entries, including $27\,003$ accidental zeros

- An accidental zero is a matrix element that is numerically zero but still occurs as a nonzero pair $(i, 0.0)$ in the data structure.
- It may be created by an operation $y_i := x_i + (-x_i)$.
- Testing all operations in a sparse matrix algorithm for zero results is more expensive than computing with a few extra entries, so accidental zeros are usually kept.



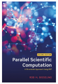
No abuse of numerics for symbolic purposes!

- Instead of using the symbolic location array, initialized at -1, we could have used an auxiliary array storing numerical values, initialized at 0.0.
- We could then add $\mathbf{y}$ into the numerical array, update $\mathbf{x}$ accordingly, and reset the array.
- Unfortunately, this would make the resulting sparsity pattern of $\mathbf{x} + \mathbf{y}$ dependent on the numerical values of $\mathbf{x}$ and $\mathbf{y}$: an accidental zero in $\mathbf{y}$ would not lead to a new entry.
- This dependence may prevent sparsity pattern reuse for repeated multiplication by a matrix with different numerical values but the same sparsity pattern.
- Reuse often speeds up subsequent program runs.



Sparse matrix data structure: coordinate scheme

- In the coordinate scheme or triple scheme, every nonzero element a_{ij} is represented by a triple (i, j, a_{ij}), where
  - i is the row index,
  - j the column index,
  - a_{ij} the numerical value.
- The triples are stored in arbitrary order in an array.
- This data structure is easiest to understand and is often used for input/output, e.g. in the Matrix Market format used by the SuiteSparse Matrix Collection, `https://sparse.tamu.edu`.
- It is suitable for input to a parallel computer, since all information about a nonzero is contained in its triple. The triples can be sent directly to the responsible processors.
- It is less suitable, however, for row-wise or column-wise matrix operations, because they would require a lot of searching.

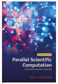


Data structure: Compressed Row Storage

- In the Compressed Row Storage (CRS) data structure, each matrix row i is stored as a compressed sparse vector consisting of pairs (j, a_{ij}) representing nonzeros.
- This data structure is also known as Compressed Sparse Row (CSR).
- In the data structure, $a[k]$ denotes the numerical value of the kth nonzero, and $j[k]$ its column index.
- Rows are stored consecutively, in order of increasing i.
- $start[i]$ is the address of the first nonzero of row i.
- The number of nonzeros of row i is

$$start[i+1] - start[i],$$

where by convention $start[n] = nz(A)$.



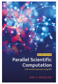
Example of CRS

$$A = \begin{bmatrix} 0 & 3 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 \\ 0 & 5 & 9 & 2 & 0 \\ 6 & 0 & 0 & 5 & 3 \\ 0 & 0 & 5 & 8 & 9 \end{bmatrix}, \; n = 5, \; nz(A) = 13.$$

The CRS data structure for A is:

$a[k] =$	3	1	4	1	5	9	2	6	5	3	5	8	9
$j[k] =$	1	4	0	1	1	2	3	0	3	4	2	3	4
$k =$	0	1	2	3	4	5	6	7	8	9	10	11	12

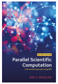
$start[i] =$	0	2	4	7	10	13
$i =$	0	1	2	3	4	5



Sparse matrix–vector multiplication using CRS

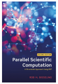
input: A : sparse $n \times n$ matrix,
$\mathbf{v}$: dense vector of length n.
output: $\mathbf{u}$: dense vector of length n, $\mathbf{u} = A\mathbf{v}$.

```
for $i := 0$ to $n - 1$ do
    $u[i] := 0;$
    for $k := start[i]$ to $start[i+1] - 1$ do
        $u[i] := u[i] + a[k] \cdot v[j[k]];$
```



Incremental Compressed Row Storage

- Incremental Compressed Row Storage (ICRS) is a variant of CRS proposed by Joris Koster in 2002.
- In ICRS, the location (i, j) of a nonzero a_{ij} is encoded as a 1D index $i \cdot n + j$.
- Instead of the 1D index itself, the difference with the 1D index of the previous nonzero is stored, as an increment in the array *inc*. This technique is sometimes called delta-indexing.
- The nonzeros within a row are ordered by increasing j, so that the 1D indices form a monotonically increasing sequence and the increments are positive.
- This is cache-friendly, because consecutively accessed vector components v_j will be closer together in memory.
- An extra dummy element $(n, 0)$ is added at the end.

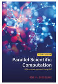


Example of ICRS

$$A = \begin{bmatrix} 0 & 3 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 \\ 0 & 5 & 9 & 2 & 0 \\ 6 & 0 & 0 & 5 & 3 \\ 0 & 0 & 5 & 8 & 9 \end{bmatrix}, \; n = 5, \; nz(A) = 13.$$

The ICRS data structure for A is:

$a[k] =$	3	1	4	1	5	9	2	...	0
$j[k] =$	1	4	0	1	1	2	3	...	0
$i[k] \cdot n + j[k] =$	1	4	5	6	11	12	13	...	25
$inc[k] =$	1	3	1	1	5	1	1	...	1
$k =$	0	1	2	3	4	5	6	...	13

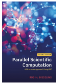


Sparse matrix–vector multiplication using ICRS

input: A : sparse $n \times n$ matrix,
$\mathbf{v}$: dense vector of length n.
output: $\mathbf{u}$: dense vector of length n, $\mathbf{u} = A\mathbf{v}$.

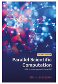
```
$j := inc[0]$;
$k := 0$;
for $i := 0$ to $n - 1$ do
    $u[i] := 0$;
    while $j < n$ do
        $u[i] := u[i] + a[k] \cdot v[j]$;
        $k := k + 1$;
        $j := j + inc[k]$;
    $j := j - n$;
```

- ICRS is slightly faster than CRS because the increments translate well into C pointer arithmetic.
- No indirect addressing like $v[j[k]]$ is needed.



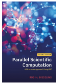
A few other data structures

- Compressed column storage (CCS), similar to CRS.
- Gustavson's data structure: both CRS and CCS, but storing numerical values only once. Offers row-wise and column-wise access to the sparse matrix.
- The two-dimensional doubly linked list: each nonzero is represented by i, j, a_{ij}, and links to a next and a previous nonzero in the same row and column.



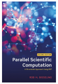
Two-dimensional doubly linked list

- Advantage: it offers maximum flexibility: row-wise and column-wise access are easy and elements can be inserted and deleted in $\mathcal{O}(1)$ operations.
- Useful for parallel sparse LU decomposition with pivoting, where rows or columns have to move frequently from one set of processors to another.
- Disadvantages:
  - $7nz + 2n$ memory space needed, compared to only $2nz + n$ for CRS;
  - following the links causes arbitrary jumps in the computer memory, often incurring cache misses.

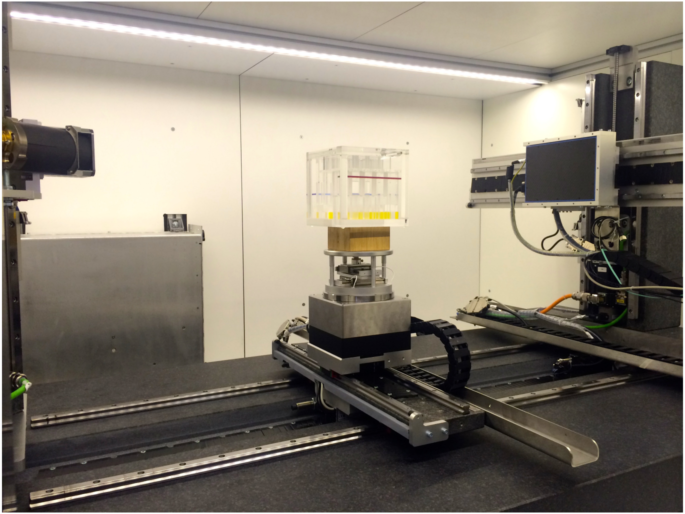


Matrix-free storage

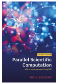
- Matrix-free storage: sometimes it may be too costly to store the matrix explicitly. Instead, each matrix element is recomputed when needed.
- This may enable the solution of otherwise unsolvable huge problems.
- Example: the weblink matrix of the whole World Wide Web is not explicitly stored. Instead the behaviour of a random surfer is simulated.
- Example: the sparse system matrix of a Computed Tomography (CT) scan is recomputed one row at a time.



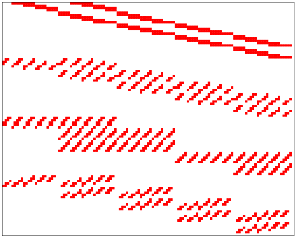
Flexible CT scanner at CWI Amsterdam



Left: X-ray source. Middle: object to be scanned. Right: detector.



Solving a sparse rectangular linear system from CT

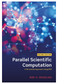


4 projections (angles)
5×5 detector pixels
$5 \times 5 \times 5$ object voxels

$m \times n$ sparse matrix
$m = 100$, $n = 125$
$nz = 1394$

$$b_i = \sum_{j=0}^{n-1} a_{ij} x_j, \quad 0 \leq i < m.$$

- a_{ij} is the weight of ray i in voxel j,
- x_j is the density of voxel j,
- b_i is the detector measurement for ray i.
- Not every ray hits every voxel: the system is sparse.
- Usually $m < n$, so system is underdetermined.



Summary

- Sparse matrix algorithms are more complicated than their dense equivalents, as we saw for sparse vector addition.
- Still, using sparsity can save large amounts of CPU time and memory space.
- We learned an efficient way of adding two sparse vectors by using a dense initialized auxiliary array. You will be surprised to see how often you can use this trick.
- Compressed row storage (CRS) and its variants are useful data structures for sparse matrices.
- CRS stores the nonzeros of each row together, but does not sort the nonzeros within a row. ICRS sorts by increasing index.
- Sorting is a mixed blessing: it may help, but it also takes time.