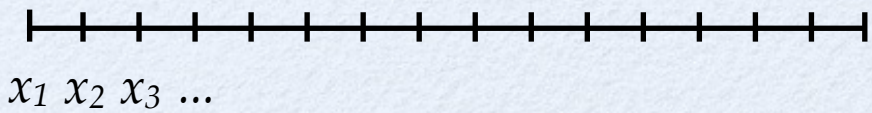
HPCSE II

Sparse linear algebra

1-d diffusion equation

- Recall the one dimensional diffusion equation $\frac{\partial f}{\partial t} = c\frac{\partial^2 f}{\partial x^2}$
- Discretize space on a regular mesh

$$x_i = i\Delta x$$



- Replace the spatial derivative by a finite difference stencil

$$\frac{\partial f(x_i)}{\partial t} = c\frac{\partial^2 f(x_i)}{\partial x^2} \approx c\frac{f(x_{i+1}) + f(x_{i-1}) - 2f(x_i)}{(\Delta x)^2}$$

- Discretize time and replace the temporal derivative by a finite difference

$$\frac{\partial f(x_i,t)}{\partial t} \approx \frac{f(x_i,t+\Delta t) - f(x_i,t)}{\Delta t} \approx c\frac{f(x_{i+1},t) + f(x_{i-1},t) - 2f(x_i,t)}{(\Delta x)^2}$$

- We get a forward-Euler integrator

$$f(x_i,t+\Delta t) \approx f(x_i,t) + \frac{c\Delta t}{(\Delta x)^2}\left[f(x_{i+1},t) + f(x_{i-1},t) - 2f(x_i,t)\right]$$

Rewriting as a matrix problem

- Interpret the function at time *t* as a vector ***f****(t)* with elements

$$f_i(t) = f(x_i, t)$$

- Then the integrator can be written as

$$f_i(t+\Delta t) = f_i(t) + \frac{c\Delta t}{(\Delta x)^2}\left[f_{i+1}(t) + f_{i-1}(t) - 2f_i(t)\right]$$

- Boundary conditions: let us fix them to keep it simple

$$f_1(t+\Delta t) = f_1(t) \qquad f_L(t+\Delta t) = f_L(t)$$

- This is just a matrix equation with a **tridiagonal matrix *M***

$$f_i(t+\Delta t) = \sum_{j=1}^{N} M_{ij} f_j(t)$$

$$\vec{f}(t+\Delta t) = M\vec{f}(t)$$

$$M = \begin{pmatrix} 1 & 0 & & & 0 \\ \frac{c\Delta t}{(\Delta x)^2} & 1-\frac{2c\Delta t}{(\Delta x)^2} & \frac{c\Delta t}{(\Delta x)^2} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{c\Delta t}{(\Delta x)^2} & 1-\frac{2c\Delta t}{(\Delta x)^2} & \frac{c\Delta t}{(\Delta x)^2} \\ 0 & & & 0 & 1 \end{pmatrix}$$

2-d diffusion equation

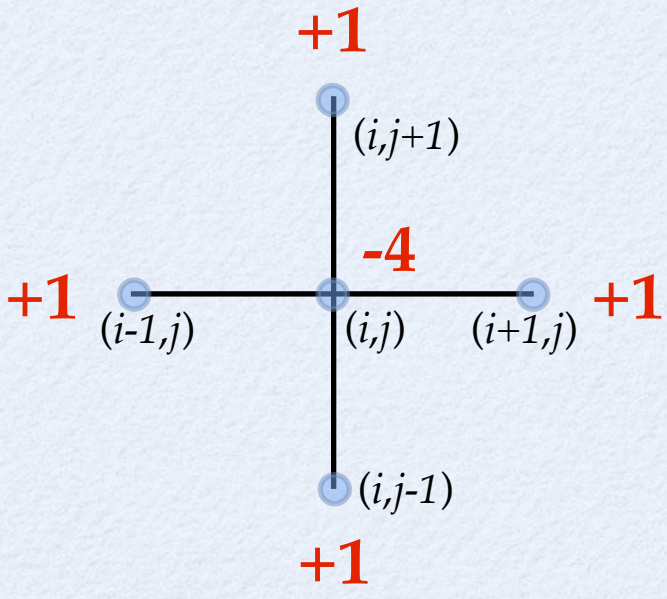
- Let's go to two dimensions $\frac{\partial f}{\partial t} = c\Delta f = c\frac{\partial^2 f}{\partial x^2} + c\frac{\partial^2 f}{\partial y^2}$
- Discretize space on a regular 2D mesh

$$\vec{r}_{i,j} = (i\Delta x, j\Delta x)$$

- And we get a two-dimensional version of the finite difference equation

$$f(\vec{r}_{i,j}, t+\Delta t) \approx f(\vec{r}_{i,j}, t) + \frac{c\Delta t}{(\Delta x)^2}\left[f(\vec{r}_{i+1,j}, t) + f(\vec{r}_{i-1,j}, t) + f(\vec{r}_{i,j+1}, t) + f(\vec{r}_{i,j-1}, t) - 4f(\vec{r}_{i,j}, t)\right]$$

- This uses the second order "stencil" for the two-dimensional Laplacian



2d diffusion equation as matrix equation

- We can again rewrite this as a matrix equation.
  On an L *x* L mesh use the indexing

$$f_{i+Lj}(t) = f(\vec{r}_{i,j}, t)$$

- We again get a matrix equation, but now with a banded matrix

$$f_i(t+\Delta t) = \sum_{j=1}^{N} M_{ij} f_j(t)$$

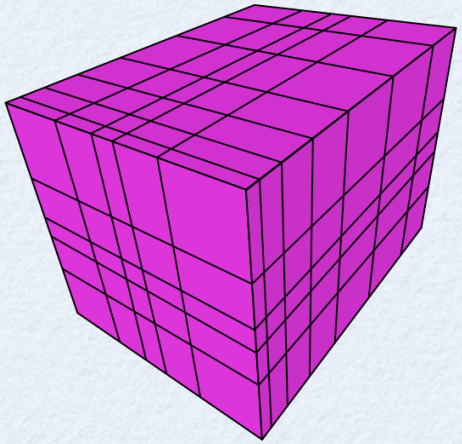
$$\vec{f}(t+\Delta t) = M\vec{f}(t)$$

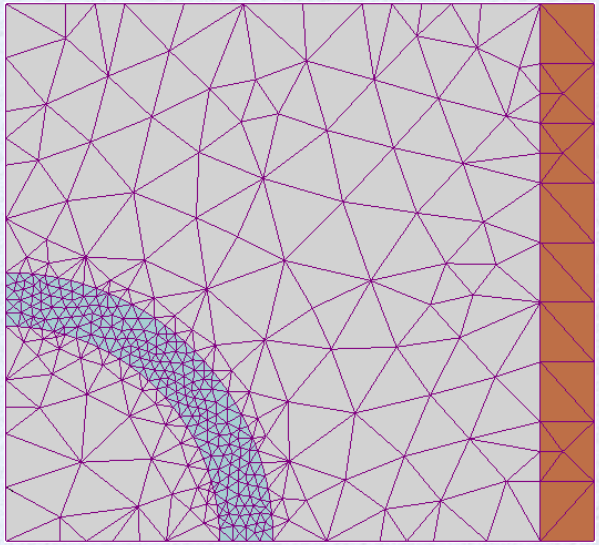
- nonzero entries only for
  i-j=0,±1,±*L*

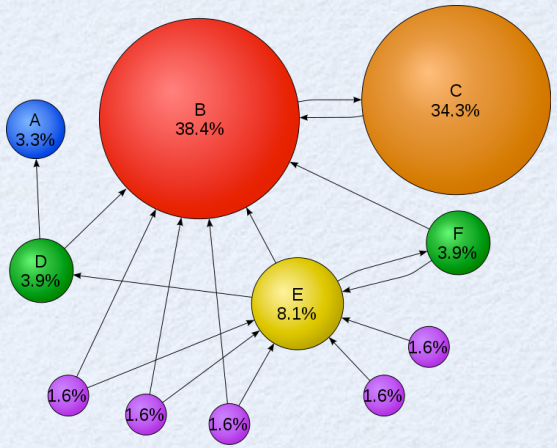
$$M = \begin{pmatrix} & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \end{pmatrix}$$

Differential equations as sparse matrix problems

- The mapping is much more general. We can map any iteration procedure for differential equations to a matrix problem
  - finite differences, finite elements or other finite basis sets
  - regular or unstructured grids
  - arbitrary graphs







Images from Wikipedia

The Poisson equation $\Delta\phi = f$

- A widely used partial differential equation
  - It relates the potential of gravity to the mass distribution

$$\Delta\phi = 4\pi G\rho$$

$G = 6.673\times10^{-11} m^3 kg^{-1} s^{-2}$ is the gravitational constant

ρ is the mass density

  - It relates the electric potential to the charge distribution

$$\Delta V = -\frac{\rho}{\varepsilon}$$

$\varepsilon =$ permitivity of the medium ($8.85\times10^{-12} F/m$ in vacuum)

ρ is the charge density

- We can put it on a mesh like the diffusion equation and write it as a matrix equation

$$\sum_j M_{ij}\phi_j = 4\pi G\rho_j$$

$$M\vec{\phi} = 4\pi G\vec{\rho}$$

Iteratively solving the Poisson equation

- The Poisson equation can be solved easily by iteration, after realizing that the central value of the solution is the average of the surrounding values minus the local density:

$$\frac{1}{(\Delta x)^2}\left[\phi(\vec{r}_{i+1,j})+\phi(\vec{r}_{i-1,j})+\phi(\vec{r}_{i,j+1})+\phi(\vec{r}_{i,j-1})-4\phi(\vec{r}_{i,j})\right]=4\pi G\rho(\vec{r}_{i,j})$$

$$\Rightarrow \phi(\vec{r}_{i,j})=\frac{1}{4}\left[\phi(\vec{r}_{i+1,j})+\phi(\vec{r}_{i-1,j})+\phi(\vec{r}_{i,j+1})+\phi(\vec{r}_{i,j-1})\right]-\pi G(\Delta x)^2\rho(\vec{r}_{i,j})$$

  - Start with a random guess $\vec{\phi}^{(0)}$
  - Iterate the fixed point equation $\vec{\phi}^{(n+1)}=M\vec{\phi}^{(n)}-\pi G(\Delta x)^2\vec{\rho}$

- Speed up convergence with successive overrelaxation (SOR) apply the proposed change multiplied by a factor

$$\vec{\phi}^{(n+1)}=(1-\alpha)\vec{\phi}^{(n)}+\alpha\left[M\vec{\phi}^{(n)}-\pi G(\Delta x)^2\vec{\rho}\right]$$

$$1\le\alpha<2$$

Sparse and dense matrices

- A **sparse vector** of length N has $m << N$ non-zero entries
  - vector operations can be performed with $O(m)$ instead of $O(N)$ effort
  - storage requirements are $O(m)$ instead of $O(N)$
    store the indices and values of non-zero entries in two vectors of size m

- A **sparse matrix** of size NxN has $m = O(N)$ or $m = O(N \log N)$ often
  - storage requirements are $O(m)$ instead of $O(N^2)$
  - many operations are faster

- Sparse matrix examples:
  - tridiagonal matrix for 1d diffusion equation: **$3N-4$ nonzero entries**
  - band matrix for 2d diffusion equation: **less than $5N$ non-zero entries**

Complexity of matrix operations

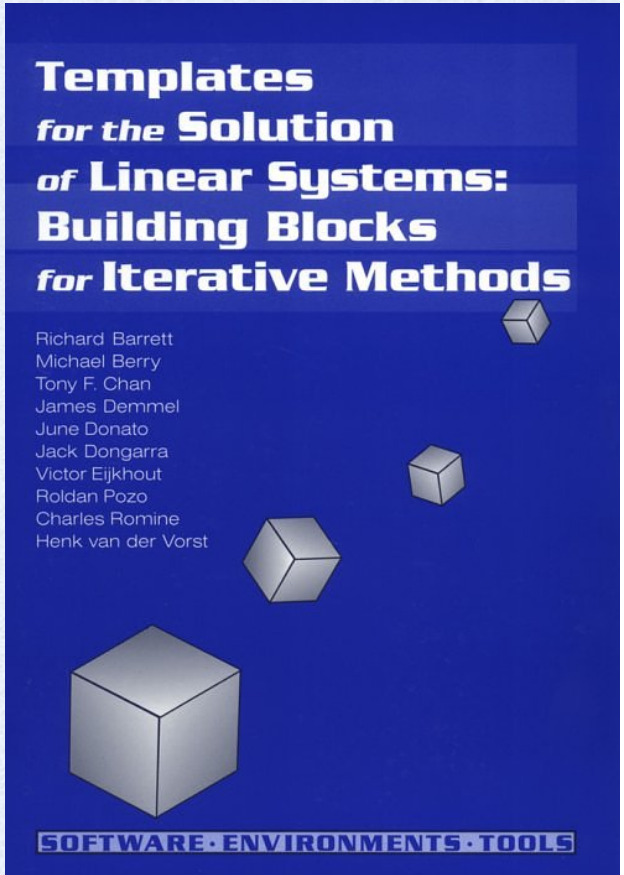
Operation	Dense matrix	Sparse matrix with O(*N*) non-zero entries
Matrix additions	$O(N^2)$	$O(N)$
Matrix-vector multiplications	$O(N^2)$	$O(N)$
Linear equation solvers	$O(N^3)$	$O(N)$
Calculate some eigenvalues	$O(N^2)$	$O(N)$

- The first two only need to iterate over non-zero elements
- Linear equations and eigenvalue problems can be solved by iterative methods using only matrix-vector multiplications, such as in

$$\vec{\phi}^{(n+1)} = (1-\alpha)\vec{\phi}^{(n)} + \alpha\left[M\vec{\phi}^{(n)} - \pi G\vec{\rho}\right]$$

Iterative linear solvers

- Sparse linear equations can be solved by iterative methods that only need the matrix in the form of matrix-vector products, and hence have linear scaling for sparse matrices!
- SOR for the Poisson equation was a very simple example. but there are more and better ones
  - Conjugate Gradient (CG)
  - BiCG
  - GMRES
  - ...
- All methods nicely explained including pseudo-code in the templates book
  - http://www.netlib.org/linalg/html_templates/Templates.html
  - PDF version http://www.netlib.org/templates/templates.pdf



Unstructured grids: PageRank

- The page rank matrix, used to rank web pages is a prime example of an unstructured sparse matrix problem.
- It is a diffusion matrix on the graph of all web pages, mimicking a random surfer. The simplest version is
  - pick one of the links on a page at random. Jump to a random page from all pages if there is no link on a page
  - the matrix row for a given page *s* contains an entry $1/L(s)$ in every column corresponding to one of the $L(s)$ pages that it links to.
  - since all entries are positive and the row sums are 1, this is a Markov transition matrix.
  - The equilibrium distribution gives the page rank. Recall that this is the largest left eigenvector of the matrix:

$$p_y = \sum_x W_{x,y} p_x \Leftrightarrow \vec{p}^T = \vec{p}^T W \Leftrightarrow \vec{p} = W^T \vec{p}$$

The power method

- The power method is the simplest iterative eigensolver. Just multiply the vector many times with the matrix.
- Algorithm from the template book http://web.eecs.utk.edu/~dongarra/etemplates/

ALGORITHM 4.1: **Power Method for HEP**

(1) *start with vector* $y = z$, *the initial guess*
(2) **for** $k = 1, 2, \ldots$
(3) $\quad v = y/\|y\|_2$
(4) $\quad y = Av$
(5) $\quad \theta = v^* y$
(6) $\quad$ **if** $\|y - \theta v\|_2 \leq \epsilon_M |\theta|$, **stop**
(7) **end for**
(8) *accept* $\lambda = \theta$ *and* $x = v$

Why the power method works

- Proof of the power method for Hermitian matrices
  - decompose the starting vector into a sum of eigenvectors

$$\vec{y} = \sum_i c_i \vec{u}_i$$

$$\text{where } A\vec{u}_i = \lambda_i \vec{u}_i \text{ and } |\lambda_1| > |\lambda_2| > \ldots$$

  - after n iterations the vector is

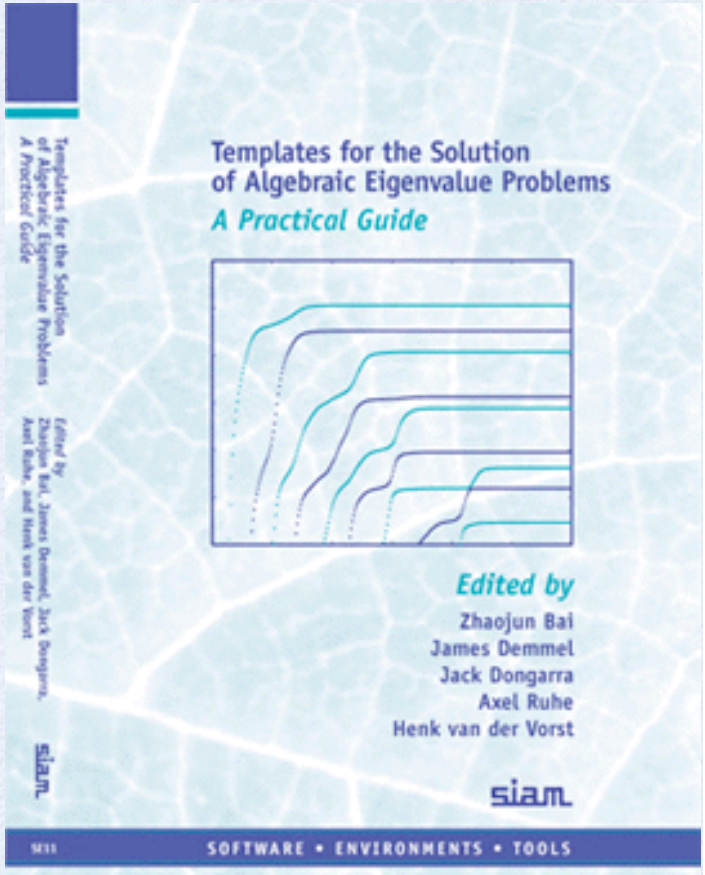
$$A^n \vec{y} = A^n \sum_i c_i \vec{u}_i = \sum_i c_i A^n \vec{u}_i = \sum_i c_i \lambda_i^n \vec{u}_i$$

  - now normalize and take the limit

$$\frac{A^n \vec{y}}{\| A^n \vec{y} \|} = \frac{\sum_i c_i \lambda_i^n \vec{u}_i}{\sqrt{\sum_i |c_i|^2 \lambda_i^{2n}}} = \frac{\sum_i c_i \frac{\lambda_i^n}{\lambda_1^n} \vec{u}_i}{\sqrt{\sum_i |c_i|^2 \frac{\lambda_i^{2n}}{\lambda_1^{2n}}}} \xrightarrow{n \to \infty} \frac{c_1 \frac{\lambda_1^n}{\lambda_1^n} \vec{u}_1}{\sqrt{|c_1|^2 \frac{\lambda_1^{2n}}{\lambda_1^{2n}}}} = \frac{c_1}{|c_1|} \vec{u}_1$$

Iterative eigensolvers solvers

- Sparse eigenproblems can be solved by iterative methods that only need the matrix in the form of matrix-vector products, and hence have linear scaling for sparse matrices if only a few eigenvectors are needed!
- Power method was a very simple example but there are more and better ones
  - Lanczos
  - Arnoldi
  - Jacobi-Davidson
  - ...
- All methods nicely explained including pseudo-code in the eigenvalues templates book
  - http://www.cs.ucdavis.edu/~bai/ET/contents.html



Variants of PageRank

- The PageRank matrices actually used are a bit more complicated
  - The "surfer" gets bored after a while:
    - with probability *d* the surfer follows a link
    - with probability (1-*d*) the surfer randomly jumps to a new page.
  - This will raise the weight of not so popular pages
  - it means that all the zeros get replaces by a small finite probability (*1-d*)/N
- Making all zero entries finite makes the matrix dense:

$$M = \text{ConstantMatrix}((1-d)/N) + dW^T$$

- A better way is to incorporate it explicitly in the multiplication function:
  - multiplying a vector by a constant matrix gives a constant vector

$$\vec{p}' = M\vec{p} = \text{ConstantMatrix}((1-d)/N)\vec{p} + dW^T\vec{p}$$
$$= \text{ConstantVector}((1-\text{d})/\text{N}) + dW^T\vec{p}$$

  - we perform a sparse matrix-vector product and add a constant vector

Sparse matrix problems

- Many problems in CSE can be mapped to sparse matrix problems
  - Explicit integrators for time integration of PDEs, such as the diffusion equation

$$\vec{f}(t+\Delta t) = M\vec{f}(t)$$

  - Implicit time integrators require solving a sparse linear systen of equations

$$M\vec{f}(t+\Delta t) = \vec{f}(t)$$

  - Solving PDEs by mapping to sparse linear systems of equations

$$M\vec{\phi} = 4\pi G\vec{\rho}$$

  - Sparse eigenproblems, such as the equilibrium state of diffusion or page rank

$$W^T\vec{p} = \lambda\vec{p}$$

- All of these problems boil down to
  - sparse matrix-vector multiplication, either directly or through iterative solvers
  - dense vector operations

Sparse matrix storage

- Discussion: how would you store a sparse matrix?

Sparse matrix storage

- **Matrix-free:**
  - just code the matrix-vector multiplication instead of storing the matrix
- **Packed** band matrices with *u* upper and *l* lower subdiagonals
  - store the diagonals only. a_{ij} stored in packed format in $p_{u+1+i-j,j}$

Dense storage of matrix *a*	Packed storage as a matrix *p*
$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

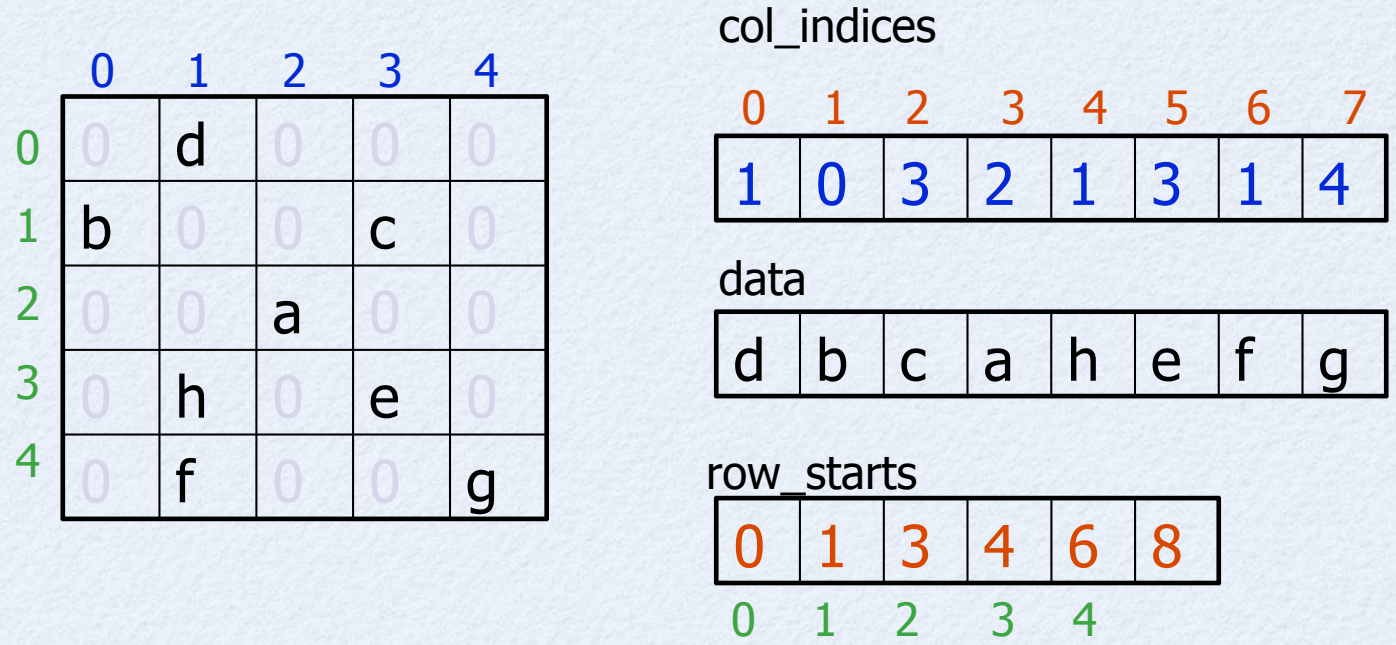
- **Sparse** storage formats:
  - store indices and values of non-zero elements
  - many options exist. What do you prefer?

Compressed storage formats

- Dictionary of keys (DOK)
  - an associative array mapping an index pair (i,j) to a value
  - stored as a tree or hash map of non-zero values
  - fast for iteratively building a matrix, slow access later
- List of lists (LIL)
  - stores one list per row, containing column index and value of the non-zero entries
  - the “list” can be a linked list, array, or vector, sorted by column index
  - fast for iteratively building a matrix
- Coordinate list (COO)
  - a list of triples (column, row, value), sorted by column and row
  - fast for iteratively building a matrix, slow access later

Compressed storage formats (cont.)

- Compressed sparse row (CSR)
  - stores the matrix in three arrays: column indices, values, and row starts



- Compressed sparse column (CSC) is similar but with row indices and column starts
- These are space-saving and efficient once the matrix is constructed. It make sense to change matrix storage format to CSR or CSC after building the matrix if another format is used for efficient construction.

Parallelizing sparse matrix operations

- Discussion: how would you parallelize a sparse matrix-vector multiplication?

Parallelizing sparse matrix operations

- A CSR matrix class

```
// an (incomplete) CSR class

template <class ValueType, class SizeType=std::size_t>
class csr_matrix
{
  typedef ValueType value_type;
  typedef SizeType size_type;

  csr_matrix(size_type s = 0)
  : n_(s)
  , row_starts(s+1)
  {}

  // we are missing functions to actually fill the matrix

  size_type dimension() const { return n_;}

  std::vector<value_type> multiply(std::vector<value_type> const& x) const;

private:
  size_type n_;
  std::vector<size_type> col_indices;
  std::vector<size_type> row_starts;
  std::vector<value_type> data;
};
```

Parallelizing sparse matrix operations

- Matrix-vector multiplication in CSR representation

```
template <class ValueType, class SizeType>
std::vector<ValueType>
csr_matrix<ValueType,SizeType>::multiply(std::vector<value_type> const& x) const
{
  assert( x.size()== dimension());
  std::vector<value_type> y(dimension());

  // loop over all rows
  #pragma omp parallel for
  for (size_type row = 0 ; row < dimension() ; ++ row)
     // loop over all non-zero elements of the row
    for (size_type i = row_starts[row] ; i != row_starts[row+1] ; ++i)
      y[row] += data[i] * x[col_indices[i]];

  return y;
}
```

- The loop over all rows can easily be parallelized
- There are no race conditions since each iteration writes into a different variable `y[row]`

Parallelizing sparse matrix operations

- Matrix-vector multiplication in CSC representation

```
template <class ValueType, class SizeType>
std::vector<ValueType>
csc_matrix<ValueType,SizeType>::multiply(std::vector<value_type> const& x) const
{
  assert( x.size()== dimension());
  std::vector<value_type> y(dimension());

  // loop over all columns
  #pragma omp parallel for
  for (size_type col = 0 ; col < dimension() ; ++ col) {
    // loop over all non-zero elements of the column
    for (size_type i = col_starts[col] ; i != col_starts[col+1] ; ++i)
      #pragma omp atomic
      y[row_indices[i]] += data[i] * x[col];
  }
  return y;
}
```

- The loop over all columns can also be parallelized
- But there are potential race conditions since different iteration may write into the same variable `y[row_indices[i]]`
- An atomic update is needed and makes the code inefficient!

Parallelizing sparse matrix operations

- Matrix-vector multiplication with a **transposed** matrix in CSR

```
template <class ValueType, class SizeType>
std::vector<ValueType>
csr_matrix<ValueType,SizeType>::multiply(std::vector<value_type> const& x) const
{
  assert( x.size()== dimension());
  std::vector<value_type> y(dimension());

  // loop over all rows
  #pragma omp parallel for
  for (size_type row = 0 ; row < dimension() ; ++ row)
    // loop over all non-zero elements of the row
    for (size_type i = row_starts[row] ; i != row_starts[row+1] ; ++i)
      #pragma omp atomic
      y[col_indices[i]] += data[i] * x[row];

  return y;
}
```

- **Potential race conditions!**

Parallelizing sparse matrix operations

- Matrix-vector multiplication with a **transposed** matrix in CSC

```
template <class ValueType, class SizeType>
std::vector<ValueType>
csc_matrix<ValueType,SizeType>::multiply(std::vector<value_type> const& x) const
{
  assert( x.size()== dimension());
  std::vector<value_type> y(dimension());

  // loop over all columns
  #pragma omp parallel for
  for (size_type col = 0 ; col < dimension() ; ++ col)
    // loop over all non-zero elements of the column
    for (size_type i = col_starts[col] ; i != col_starts[col+1] ; ++i)
      y[col] += data[i] * x[row_indices[i]];

  return y;
}
```

- **All is safe!**

Summary of sparse matrix operations

- If possible use a matrix-free method and hard-code the matrix-vector multiplication. This uses less memory and is faster.
- To iteratively build a matrix use DOK, LIL, COO or similar formats unless the data comes in the right order for CSR or CSC
- To use the matrix
  - prefer CSR format or similar for matrix-vector multiplication
  - prefer CSC format or similar for matrix-vector multiplication with the transposed matrix
- We might have to copy the matrix into a new format
- What do we do if we need to multiply vectors with both the original and the transposed matrix?