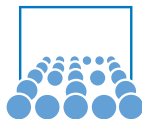


# HPC – Algorithms and Applications

## Dwarf # 2 – Sparse Linear Algebra

Michael Bader

Winter 2013/2014



# The Seven Dwarfs of HPC – Dwarf # 2

1. dense linear algebra
2. **sparse linear algebra**
3. spectral methods
4. N-body methods
5. structured grids
6. unstructured grids
7. Monte Carlo

# Part I

## Application Example: Page Rank



# Ranking Problem for Websites

- given:  $n$  websites connected by hyperlinks
- wanted: rank websites according to "importance"
- idea: rank depends on links to a website

## Quantitative approach – count the links:

- Graph model:  
websites  $\rightarrow$  nodes, links  $\rightarrow$  edges
- represented as *adjacency matrix*:  
 $A_{ij} = 1$  if an edge exists from  $i$  to  $j$ , (else  $A_{ij} = 0$ )
- ranking depends on number of edges to  $j$

$$r(j) := \sum_{i \neq j} A_{ij} \quad (\text{column sum})$$

# PageRank

## Qualitative approach:

- Goal: links from "important" website have higher impact
- Step 1: add weights (rank) instead of number of links
- Example: page 3 and 4 link to page 2  
⇒ impact  $x_2$  of page 2 is  $x_2 = x_3 + x_4$

## Modelled by adjacency matrix:

- leads to system of equations:

$$x_j = \sum_{i \neq j} A_{ij} x_i = \sum_{i \neq j} (A^T)_{ji} x_i$$

- in matrix-vector notation:  $x = A^T x$   
(search eigenvector for eigenvalue 1)

## PageRank (2)

- Goal: reduce influence of pages with many links
- Step 2: weights divided by number of outgoing links (each website has a "total impact"/sum of weights of 1)
- Example: page 3 (three outgoing links) and 4 (two links) link to page 2  
⇒ impact  $x_2$  of page 2 is  $x_2 = \frac{1}{3}x_3 + \frac{1}{2}x_4$

### Modelled by adjacency matrix:

- $n_i$ : number of outgoing links of page  $i$ ;  $n_i = \sum_j A_{ij}$
- Resulting system of equations:

$$x_j = \sum_{i \neq j} \frac{1}{n_i} A_{ij} x_i = \sum_{i \neq j} \left( \frac{1}{n_i} (A^T)_{ji} \right) x_i$$

# Page-Rank Matrix

- set  $B_{ji} := \frac{1}{n_i}(A^T)_{ji} \rightarrow$  leads to system of equations:

$$x_j = \sum_{i \neq j} \frac{1}{n_i} A_{ij} x_i = \sum_{i \neq j} B_{ji} x_i$$

- search eigenvector for eigenvalue 1:  $x = Bx$

## Properties of the page-rank matrix:

- all column sums are 1
- all  $B_{ji} \geq 0$ , diagonal elements  $B_{jj} = 0$   
(linking to your own page is not counted)
- $B$  is a so-called (left) **stochastic matrix**

# Stochastic Matrices – Properties

$B$  a stochastic matrix, then:

1.  $B$  has 1 as eigenvalue;  
all elements of the corresp. eigenvectors  $b^{(1)} \geq 0$   
→ normalise  $b^{(1)}$ , such that  $\sum b_j^{(1)} = 1$
2. element sum of  $y = Bx$  is equal to the element sum of  $x$ ;  
if  $x \geq 0$  (element-wise), then also  $y \geq 0$
3.  $v$  an eigenvector of  $B$  with eigenvalue  $\neq 1$ ,  
⇒ element sum of  $v$  equal to 0
4.  $\lambda$  eigenvalue of  $B$ , then  $|\lambda| \leq 1$

(without proofs → see a resp. textbook)



# Vector Iteration with Stochastic Matrices

- examine iteration  $x^{(m)} = Bx^{(m-1)}$ ,  
start vector  $x^{(0)} \geq 0$  has element sum 1
- use eigenvector decomposition of  $x^{(0)}$ :

$$x^{(0)} = \sum_j \gamma_j b^{(j)}$$

- then:  $x^{(m)} = B^m x^{(0)} = \sum_j \gamma_j \lambda_j^m b^{(j)}$
- if  $\lambda_1 = 1$  and all other  $0 < \lambda_j < 1$ , then:

$$x^{(m)} = \sum_j \gamma_j \lambda_j^m b^{(j)} \rightarrow \gamma_1 b^{(1)} \quad \text{for } m \rightarrow \infty$$

# PageRank in Practice

- use a start vector  $x^{(0)}$  with element sum 1  
(then:  $\gamma_1 = 1$ )
- vector iteration  $x^{(m)} = Bx^{(m-1)}$  converges to ranking vector  $\gamma_1 b^{(1)} = b^{(1)}$  (with element sum 1)
- as every page has only few outgoing links  
→  $B$  a sparse matrix
- $n$  pages with an average of  $k$  links per page:  
→  $kn$  mult/add operations per iteration
- convergence faster for smaller values of the largest eigenvalue (except  $\lambda_1 = 1$ )

# Vector Iteration in Practice

## Problem: two separate partitions

- consider the following page-rank matrix:

$$B = \begin{pmatrix} B_I & 0 \\ 0 & B_{II} \end{pmatrix}$$

(web divided into two non-linked partitions)

- $B_I$  and  $B_{II}$  are stochastic matrices, each with eigenvectors  $b_I$  and  $b_{II}$  for eigenvalue 1
- $(b_I \ b_{II})^T$ , but also  $(b_I \ 0)^T$  and  $(0 \ b_{II})^T$  are eigenvectors of  $B$  (for eigenvalue 1)
- consequences for convergence and ranking?

## Vector Iteration in Practice (2)

### Problem: slow convergence

- happens, if at least one  $\lambda \approx 1$  (but  $\neq \lambda_1 = 1$ )
- modify page-rank matrix  $B$ :

$$\tilde{B} \rightsquigarrow \alpha B + (1 - \alpha) \frac{1}{n} \begin{pmatrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ 1 & \dots & 1 \end{pmatrix}$$

- new system of equations  $x = \tilde{B}x$ ,  
or:  $x = \alpha Bx + (1 - \alpha) \frac{1}{n} ee^T x$ , with  $e = (1, \dots, 1)^T$
- equivalent to:  $x - \alpha Bx = (1 - \alpha) \frac{1}{n} ee^T x$
- $\frac{1}{n} ee^T$  stochastic, therefore  $\tilde{B}$  a stochastic matrix, as well

# Vector Iteration in Practice (3)

## Regularisation

- $x = \tilde{B}x$  iff  $x - \alpha Bx = (1 - \alpha)\frac{1}{n}ee^T x$
- as  $e^T x = 1$  (element sum = 1):

$$(I - \alpha B)x = \frac{1}{n}(1 - \alpha)e \quad \text{where } 0 < \alpha < 1$$

- eigenvalues of  $\alpha B$  are  $\leq \alpha$   
 $\Rightarrow (I - \alpha B)$  not singular (all eigenvalues  $\geq 1 - \alpha$ )
- leads to **unique solution**

# Vector Iteration in Practice (4)

## Convergence

- compute solution via vector iteration:

$$x^{(m)} = \alpha Bx^{(m-1)} + (1 - \alpha) \frac{1}{n} e$$

- corresponds to iteration for error vector  $\epsilon(m) = x^{(m)} - x$ :

$$\epsilon(m) = \alpha B\epsilon(m-1)$$

- now: all eigenvalues of  $\alpha B$  are  $\leq \alpha$
- therefore  $\|\epsilon(m)\| \sim \alpha^m \|\epsilon(0)\|$   
(convergence faster for smaller  $\alpha$ )

# Vector Iteration in Practice (5)

## Regularisation vs. Convergence:

- Vector iteration converges faster for smaller  $\alpha$
- solution is better, the closer  $\alpha$  is to 1  
(then  $\tilde{B} \approx B$ )
- task: find an optimal  $\alpha$   
(common page-rank choice:  $\alpha = 0.85$ )
- regularisation parameter balances between exact solution and “well-behaved” problem
- regularisation therefore a frequent technique for ill-posed problems

# PageRank as a Population Model: Random Surfer

- each website “populated” by  $x_i$  web surfers
- total population:  $\sum x_i = 1$  (normalised)
- population corresponds to page rank: how many surfer are expected to be on each site?

## PageRank: ”Random Surfer”

- surfers randomly follow a link from the current page (and change to a different website)
- thus:  $\frac{1}{n_i} A_{ij} x_i$  surfers change to website  $j$
- regularisation: with probability  $(1 - \alpha)$ , a surfer will jump to another (random) page in the internet
- vector iteration  $\rightarrow$  population evolves towards an equilibrium



## Part II

# Data Structures for Sparse Matrices

*A matrix is called sparse, if it contains that many zero elements (i.e., that few non-zeros), such that it becomes worthwhile to change to special data structures and algorithms to store and process them.*

(following a definition by Wilkinson)

## Coordinate Scheme (aka Triple Scheme)

- store a triple  $(a_{ij}, i, j)$  for each non-zero  $a_{ij}$  (not necessarily sorted)
- implementation as “array of struct”:

```

struct sparseElement {
    int i; int j; int value
};
typedef struct sparseElement[] sparseMatrix;
  
```

- implementation as  $3 \times K$  matrix ( $K$  non-zeros):

$a_{ij}$	2.	-1.	-1.	2.	-1	-1.	2.	-1.	-1.	2.
$i$	1	1	2	2	2	3	3	3	4	4
$j$	1	2	1	2	3	2	3	4	3	4

- used, for example, in Matlab, or as input format

# Compressed Row Storage (CRS)

- two arrays of size  $K$  with  $a_{ij}$  and  $j$ :

a[k]	2.	-1.	-1.	2.	-1	-1.	2.	-1.	-1.	2.
j[k]	1	2	1	2	3	2	3	4	3	4

(sorted by increasing row)

- third array points to start of a row in a[k]:

start[i]	0	2	5	8	10
----------	---	---	---	---	----

- implementation of sparse-matrix-vector mult. (SpMV):

```

for(i=0; i<n; i++) {
    for(k=start[i]; k<start[i+1]; k++)
        y[i] += a[k] * x[ j[k] ];
};

```

# Variants of CRS

## Compressed Column Storage (CCS)

- same as CRS, but with rows and columns exchanged
- used in Harwell-Boeing / Rutherford Boeing collection

## Gustavson: Combine CRS and CCS

- compressed row storage format plus:
- column starts ( $\text{start}[j]$ ) and row indices  $i[k]$  as in CCS
- improves column-wise access to sparse matrices

## Block CRS

- store small matrix blocks (e.g.:  $2 \times 2$ ,  $4 \times 4$ ) instead of single elements
- improves exploitation of SIMD capabilities of CPUs

# Incremental CRS

- use  $\tilde{j} = (i - 1) \cdot n + (j - 1)$  as element index
- store increments of  $\tilde{j}$  in array `inc[k]`

<code>a[k]</code>	2.	-1.	-1.	2.	-1	-1.	2.	-1.	-1.	2.
<code>inc[k]</code>	0	1	3	1	1	3	1	1	3	1
$\tilde{j}[k]$	0	1	4	5	6	9	10	11	14	15

- implementation of sparse-matrix-vector mult. (SpMV):

```

for(k=0,i=0, jj =inc [0]; i<n; i += j/n)
    for(j = j%n; j<n; j+=inc[k],k++)
        y[i] += a[k] * x[j];
  
```

- C impl. of SpMV found to be faster than for CRS (Bisseling)
- avoids array `start[i]` (esp. attractive, if  $K < n$ )

## ELLPACK Format – Rectangular Storage

- scenario: at most  $k_{\max}$  non-zeros per row
- store values in  $n \times k_{\max}$  matrix/array  $a_{ik}$
- store column indices in  $n \times k_{\max}$  matrix/array  $j_{ik}$

$$a_{ik} : \begin{pmatrix} 2. & -1. & 0 \\ -1. & 2. & -1 \\ -1. & 2. & -1. \\ -1. & 2. & 0 \end{pmatrix} \quad j_{ik} : \begin{pmatrix} 1 & 2 & * \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & * \end{pmatrix}$$

- very regular data structure
- supports vectorisation

# ELLPACK Format – SpMV Implementation

- row-oriented implementation:

```
for(i=0; i<n; i++)  
  for(k=0; k<kmax; k++)  
    y[i] += a[i][k] * x[ j[i][k] ];
```

- column-oriented implementation:

```
for(k=0; k<kmax; k++)  
  for(i=0; i<n; i++)  
    y[i] += a[i][k] * x[ j[i][k] ];
```

→ second for-loop leads to vector access to y, a and j

## Jagged Diagonal Storage (JDS)

- sort rows according to number of non-zeros
- consider matrices  $a_{ik}$  and  $j_{ik}$  as in ELLPACK
- store  $a_{ik}$  and  $j_{ik}$  sequentially and in column-major order
- but do not store  $a_{ik}, j_{ik}$  for zero elements  
 $\Rightarrow$  requires additional array  $\text{start}[k]$  (start of  $k$ -th column)
- SpMV implementation (column-oriented):

```

for(k=0; k<kmax; k++ )
    for(i=0; i < (start[k+1]-start[k]); i++)
        y[i] += a[start[k]+i] * x[ j[start[k]+i] ];
  
```

- Note: element in the result vector  $y$  are re-ordered, as well (according to number of non-zeros per row)



# Reference/Literature (Sparse Linear Algebra)

Rob H. Bisseling:  
*Parallel Scientific Computing –  
A structured approach using BSP and MPI.*  
Oxford University Press, 2004.