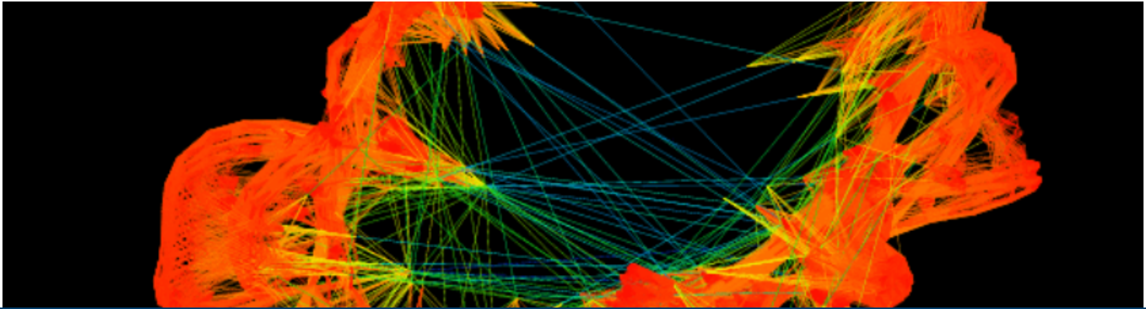


High-Performance Matrix Computations

Sparse Matrix Representations and Computations

Jan. 24, 2022 | Xinzhe Wu (xin.wu@fz-juelich.de) | Jülich Supercomputing Centre

RWTH AACHEN UNIVERSITY

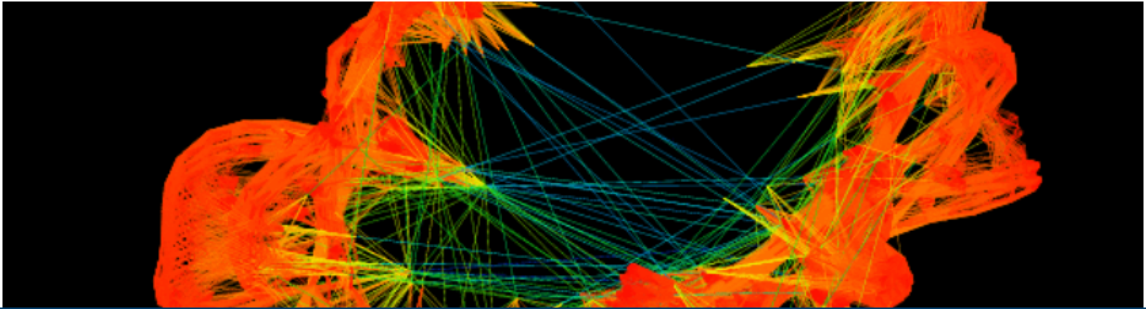
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Organisation

Topics: High-Performance Computations of Sparse Matrices

- Module 1 (Jan. 24): Sparse Matrix Representations and Computations
- Module 2 (Jan. 26): Applications of Sparse Matrix:
  - Iterative linear solver: Conjugate Gradient method (CG)
  - Graph analytics: PageRank algorithm to rank webpages (if we have time)

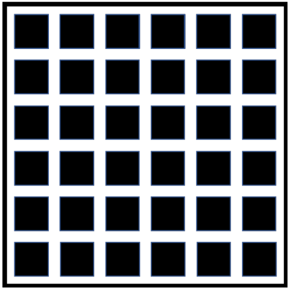
- Lectures based on slides
- Practical examples and exercises
  1. Module 1: C codes on Laptop and CLAIX
     - numerical kernel implementation
     - calling of high-performance libraries for sparse matrices
     - testing and benchmarking
  2. Module 2: Jupyter notebooks with Julia on Laptop
     - Questions in sequence during the execution of Jupyter notebooks



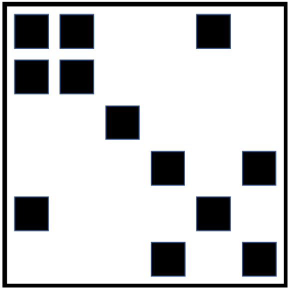
Part I: Sparse Matrix

Sparse Matrices

Sparse matrix is a matrix (real, complex) where most of the elements are zeros.



Dense Matrix

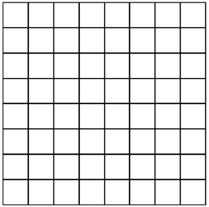


Sparse Matrix

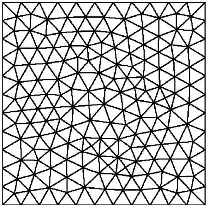
For a $N \times N$ sparse matrix A, the number of non-zeros elements (nnz) is $\mathcal{O}(N)$. The sparsity is defined as $\frac{nnz}{N^2}$.

Sparse Matrices
Encoding connectivity

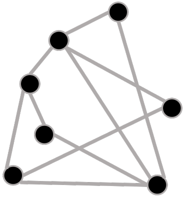
Finite-Elements Meshes, Hyperlinks, Social Networks, Neural Networks, $\cdots$



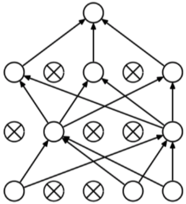
Structured Mesh



Unstructured Mesh



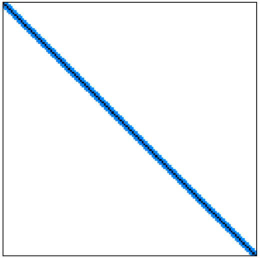
Indirected Graph



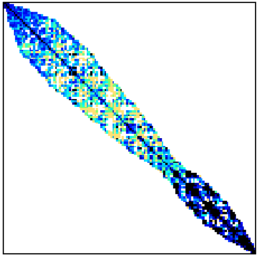
DNN with dropout

Sparsity Patterns

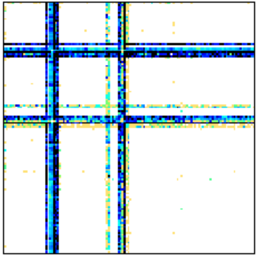
- Mesh type: Elements, structured, unstructured, $\cdots$
- Problem dimension (2D, 3D)
- Discretization method
- Graph (connections, directed, indirected, $\cdots$)



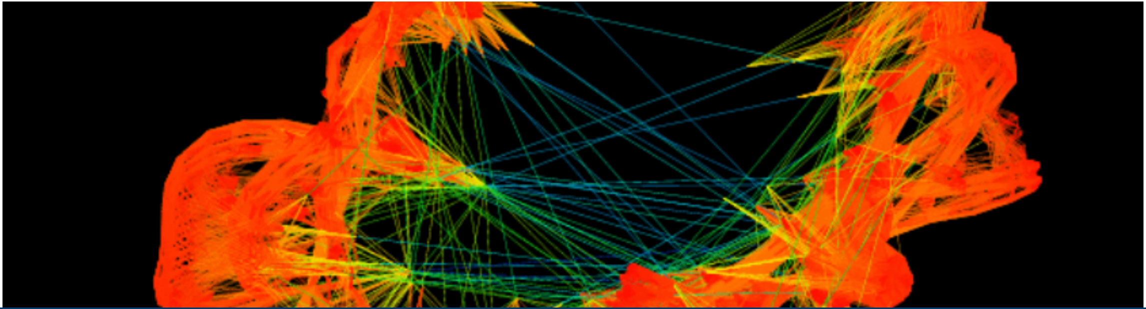
Laplace eqn 2D mesh (Link)



electromagnetic (Link)



Packet trace data (Link)



Part II: Sparse Matrix Storage Formats

Matrix Format: Coordinate (COO)

Idea: store both the column index & row index for every nonzero element

- Row index (int) (*nnz*)
- Column index (int) (*nnz*)
- Values (data type) (*nnz*)

1	7	0	0
0	2	8	0
5	0	3	9
0	6	0	4

→

values:	1	7	2	8	5	3	9	6	4
row indices:	0	0	1	1	2	2	2	3	3
col indices:	0	1	1	2	0	2	3	1	3

Matrix Format: Compressed Sparse Row (CSR)

Idea: store the column index for every nonzero & row offsets for each row

- Row offset (int) (*N*)
- Column index (int) (*nnz*)
- Values (data type) (*nnz*)

1	7	0	0
0	2	8	0
5	0	3	9
0	6	0	4

→

values:	1	7	2	8	5	3	9	6	4
col indices:	0	1	1	2	0	2	3	1	3
row offsets:	0	2	4	7	9				

Matrix Format: ELLPACK (ELL)

Idea: store the values and column indices with padding.

- max nb of el per row (M)
- Column index (int) ($N * M$)
- Values (data type) ($N * M$)

1	7	0	0
0	2	8	0
5	0	3	9
0	6	0	4

$\longrightarrow$

values:

1	7	*
2	8	*
5	3	9
6	4	*

column indices:

0	1	*
1	2	*
0	2	3
1	3	*

Matrix Format: Diagonal (DIA)

Idea: store the values and column indices with padding.

- max nb of el per row (M)
- Column index (int) ($N * M$)
- Values (data type) ($N * M$)

1	7	0	0
0	2	8	0
5	0	3	9
0	6	0	4

$\longrightarrow$

values:

*	1	7
*	2	8
5	3	9
6	4	*

diagonal offsets:

1	2	*

Matrix Format: Memory footprint

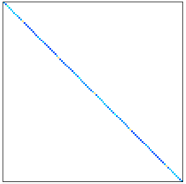
- N - number of rows and columns in the matrix
- nnz - number of non-zeros elements in the matrix
- M - number of nonzero entries in the densest row
- D - number of non-null diagonal

Format	Structure (words)	Values
Dense	-	N^2
COO	$2 \times nnz$	nnz
CSR	$N + 1 + nnz$	nnz
ELL	$M \times N$	$M \times N$
DIA	D	$D \times N$

Storage Format Comparison

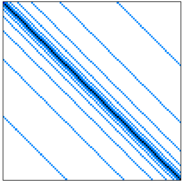
Bytes per Nonzero Entry (`double` & `int`)

`bcsstm06`



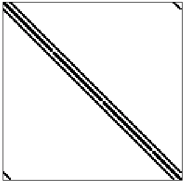
- COO: 16.00
- CSR: 16.01
- DIA: 8.01
- ELL: 12.00

`Trefethen_200`



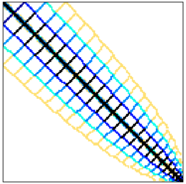
- COO: 16.00
- CSR: 12.28
- DIA: 9.44
- ELL: 13.29

`G13`



- COO: 16.00
- CSR: 13.00
- DIA: 16.01
- ELL: 12.00

`benezene`

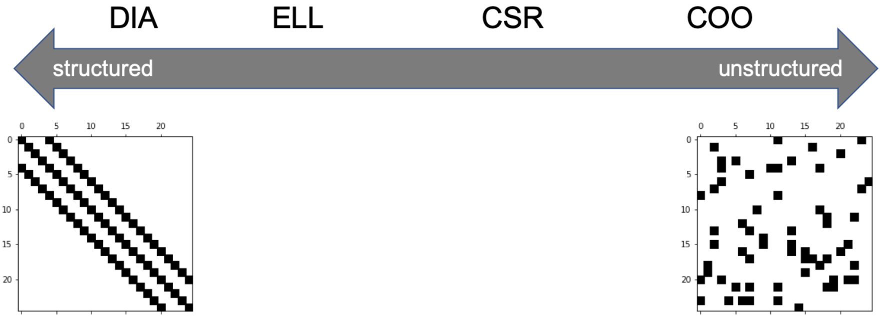


- COO: 16.00
- CSR: 12.14
- DIA: 1550.76
- ELL: 15.04

`G21`

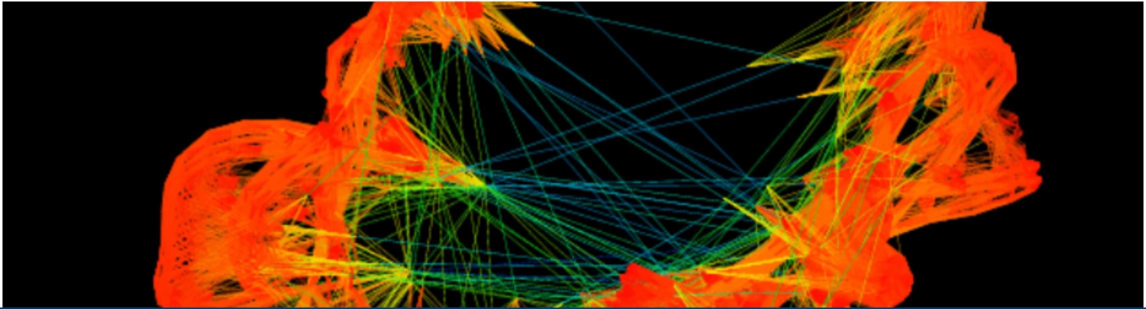
- COO: 16.00
- CSR: 12.34
- DIA: 1041.08
- ELL: 147.08

Summary



Other Sparse Matrix Formats

- **Compressed Sparse Column (CSC):**
  - Like CSR, but stores a dense set of sparse column vectors
  - Useful for when column sparsity is much more regular than row sparsity
- **Blocked CSR:**
  - the matrix is divided into blocks stored using CSR with the indices of the upper left corner
  - Useful for block-sparse matrices
- **Hybrid methods (HYB):**
  - It is used for the irregular sparse matrices, e.g., ELL handles *typica*l entries and COO handles *exceptional* entries
- ...



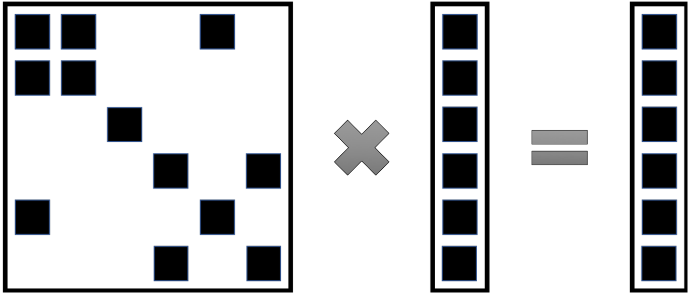
Part III: Sparse Matrix-Vector Multiplication (SpMV)

RWTH AACHEN UNIVERSITY

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

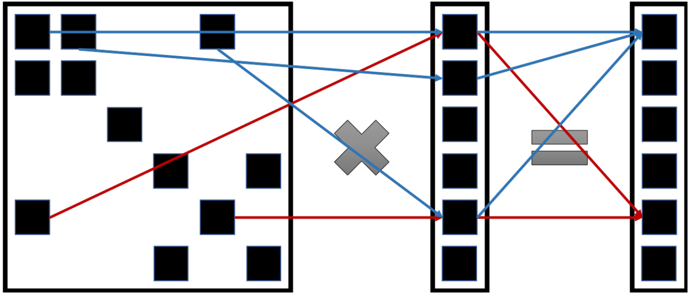
Sparse Matrix-Vector Multiplication (SpMV)

- SpMV is to compute $u = Av$ in which A is sparse matrix, and u and v are dense vectors
- A is stored in compressed format.



Sparse Matrix-Vector Multiplication (SpMV)

- SpMV is to compute $u = Av$ in which A is sparse matrix, and u and v are dense vectors
- A is stored in compressed format.



Applications of SpMV

- In many applications, variables are connected to only a few others, leading to sparse matrices.
- Sparse matrices occur in various application areas:
  - transition matrices in Markov models;
  - finite-element matrices in numerical simulations;
  - linear programming matrices in optimisation;
  - weblink matrices in Google PageRank computation;
  - Deep Neural Network (DNN) for deep learning;
  - $\cdots$
- More generally, SpMV is the main computation step in iterative methods for linear systems or eigenproblems:
  - **Linear system** $Ax = b$, solved by the conjugate gradient (CG), MINRES, GMRES, QMR, BiCGStab, $\cdots$
  - **Eigenproblem** $Ax = \lambda x$ solved by power method, Lanczos method, Jacobi–Davidson, $\cdots$

Sequential SpMV: DIA

```
struct SparseMatrixDIA {
double * values;
int * diag;
int N;
int ndiag; };

void spmv_dia(SparseMatrixDIA m, double *x, double *y){

  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; i<m.ndiag; ++j){
      int col = i + m.diag[j];
      double val = m.val[j*m.N+i];
      if(col >= 0 && col < m.N)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	*	*	5	6	1	2	3	4	7	8	9	*

DIA requires to pad the empty elements: Place zeros in values OR place an invalidating indicator into either array.

Parallel SpMV on CPUs: DIA

```c
struct SparseMatrixDIA {
  double * values;
  int * diag;
  int N;
  int ndiag; };

void spmv_dia(SparseMatrixDIA m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; i<m.ndiag; ++j){
      int col = i + m.diag[j];
      double val = m.val[j*m.N+i];
      if(col >= 0 && col < m.N)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	*	*	5	6	1	2	3	4	7	8	9	*

Parallel SpMV on CPUs: DIA

```
struct SparseMatrixDIA {
  double * values;
  int * diag;
  int N;
  int ndiag; };

void spmv_dia(SparseMatrixDIA m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; i<m.ndiag; ++j){
      int col = i + m.diag[j];
      double val = m.val[j*m.N+i];
      if(col >= 0 && col < m.N)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	*	*	5	6	1	2	3	4	7	8	9	*

Parallel SpMV on CPUs: DIA

```c
struct SparseMatrixDIA {
  double * values;
  int * diag;
  int N;
  int ndiag; };

void spmv_dia(SparseMatrixDIA m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; i<m.ndiag; ++j){
      int col = i + m.diag[j];
      double val = m.val[j*m.N+i];
      if(col >= 0 && col < m.N)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	*	*	5	6	1	2	3	4	7	8	9	*

Parallel SpMV on CPUs: DIA

```c
struct SparseMatrixDIA {
  double * values;
  int * diag;
  int N;
  int ndiag; };

void spmv_dia(SparseMatrixDIA m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; i<m.ndiag; ++j){
      int col = i + m.diag[j];
      double val = m.val[j*m.N+i];
      if(col >= 0 && col < m.N)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

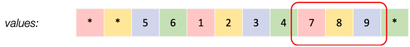
values:												
	*	*	5	6	1	2	3	4	7	8	9	*

- Pros: (1) avoid storing col/row indices; (2) continuous memory access;
- Cons: potentially waste storage for padding and zero values on occupied diagonals.

Parallel SpMV on CPUs: DIA

```c
struct SparseMatrixDIA {
  double * values;
  int * diag;
  int N;
  int ndiag; };

void spmv_dia(SparseMatrixDIA m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; i<m.ndiag; ++j){
      int col = i + m.diag[j];
      double val = m.val[j*m.N+i];
      if(col >= 0 && col < m.N)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```



- it is applicable to the applications of stencils applied to regular grids
- many matrices have sparsity patterns that are inappropriate for DIA

Sequential SpMV: ELL

```c
struct SparseMatrixELL {
  double * values;
  int * col_indices;
  int N;
  int max_row; };

void spmv_ell(SparseMatrixELL m, double *x, double *y){

  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; j<m.max_row; ++j){
      int col = m.col_indicies[m.N*j+i];
      double val = m.val[m.N * j + i];
      if (val != 0)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	1	2	5	6	7	8	3	4	*	*	9	*
col indices:	0	1	0	1	1	2	2	3	*	*	3	*

Similar as DIA, ELL requires to pad the empty elements.

Parallel SpMV on CPUs: ELL

```c
struct SparseMatrixELL {
  double * values;
  int * col_indices;
  int N;
  int max_row; };

void spmv_ell(SparseMatrixELL m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; j<m.max_row; ++j){
      int col = m.col_indicies[m.N*j+i];
      double val = m.val[m.N * j + i];
      if (val != 0)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	1	2	5	6	7	8	3	4	*	*	9	*
col indices:	0	1	0	1	1	2	2	3	*	*	3	*

Parallel SpMV on CPUs: ELL

```
struct SparseMatrixELL {
  double * values;
  int * col_indices;
  int N;
  int max_row; };

void spmv_ell(SparseMatrixELL m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; j<m.max_row; ++j){
      int col = m.col_indicies[m.N*j+i];
      double val = m.val[m.N * j + i];
      if (val != 0)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	1	2	5	6	7	8	3	4	*	*	9	*
col indices:	0	1	0	1	1	2	2	3	*	*	3	*

Parallel SpMV on CPUs: ELL

```c
struct SparseMatrixELL {
  double * values;
  int * col_indices;
  int N;
  int max_row; };

void spmv_ell(SparseMatrixELL m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; j<m.max_row; ++j){
      int col = m.col_indicies[m.N*j+i];
      double val = m.val[m.N * j + i];
      if (val != 0)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	1	2	5	6	7	8	3	4	*	*	9	*
col indices:	0	1	0	1	1	2	2	3	*	*	3	*

Parallel SpMV on CPUs: ELL

```
struct SparseMatrixELL {
  double * values;
  int * col_indices;
  int N;
  int max_row; };

void spmv_ell(SparseMatrixELL m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; j<m.max_row; ++j){
      int col = m.col_indicies[m.N*j+i];
      double val = m.val[m.N * j + i];
      if (val != 0)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	1	2	5	6	7	8	3	4	*	*	9	*
col indices:	0	1	0	1	1	2	2	3	*	*	3	*

- nearly identical to the DIA with explicit column indices
- non-continuous access to *x*

Parallel SpMV on CPUs: ELL

```c
struct SparseMatrixELL {
  double * values;
  int * col_indices;
  int N;
  int max_row; };

void spmv_ell(SparseMatrixELL m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    for (int j=0; j<m.max_row; ++j){
      int col = m.col_indicies[m.N*j+i];
      double val = m.val[m.N * j + i];
      if (val != 0)
        dot += val * x[col];
    }
    y[i] += dot;
  }
}
```

values:	1	2	5	6	7	8	3	4	*	*	9	*
col indices:	0	1	0	1	1	2	2	3	*	*	3	*

- most efficient when the maximum number of nonzeros per row does not substantially differ from the average, e.g., matrices obtained from semi-structured meshes and well-behaved unstructured meshes

Sequential SpMV: CSR

```
struct SparseMatrixCSR {
  double * values;
  int * col_indices;
  int * row_offsets;
  int N;
  int nnz; };

void spmv_csr(SparseMatrixCSR m, double *x, double *y){

  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    int row_start = m.rowoffsets[i];
    int row_end = m.rowoffsets[i+1];
    for (int j=start; i<m.end; ++j)
      dot += m.val[j] * x[m.col_indices[j]];
    y[i] += dot;
  }
}
```

values:	1	7	2	8	5	3	9	6	4
col indices:	0	1	1	2	0	2	3	1	3

row offsets:	0	2	4	7	9

The iterate times of its inner loop depends on density of each row.

Parallel SpMV on CPUs: CSR

```
struct SparseMatrixCSR {
  double * values;
  int * col_indices;
  int * row_offsets;
  int N;
  int nnz; };

void spmv_csr(SparseMatrixCSR m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    int row_start = m.rowoffsets[i];
    int row_end = m.rowoffsets[i+1];
    for (int j=start; i<m.end; ++j)
      dot += m.val[j] * x[m.col_indices[j]];
    y[i] += dot;
  }
}
```

values:	1	7	2	8	5	3	9	6	4
col indices:	0	1	1	2	0	2	3	1	3
row offsets:	0	2	4	7	9				

Parallel SpMV on CPUs: CSR

```
struct SparseMatrixCSR {
  double * values;
  int * col_indices;
  int * row_offsets;
  int N;
  int nnz; };

void spmv_csr(SparseMatrixCSR m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    int row_start = m.rowoffsets[i];
    int row_end = m.rowoffsets[i+1];
    for (int j=start; i<m.end; ++j)
      dot += m.val[j] * x[m.col_indices[j]];
    y[i] += dot;
  }
}
```

values:	1	7	2	8	5	3	9	6	4
col indices:	0	1	1	2	0	2	3	1	3

row offsets:	0	2	4	7	9

Parallel SpMV on CPUs: CSR

```
struct SparseMatrixCSR {
  double * values;
  int * col_indices;
  int * row_offsets;
  int N;
  int nnz; };

void spmv_csr(SparseMatrixCSR m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    int row_start = m.rowoffsets[i];
    int row_end = m.rowoffsets[i+1];
    for (int j=start; i<m.end; ++j)
      dot += m.val[j] * x[m.col_indices[j]];
    y[i] += dot;
  }
}
```

values:	1	7	2	8	5	3	9	6	4
col indices:	0	1	1	2	0	2	3	1	3

row offsets:	0	2	4	7	9

Parallel SpMV on CPUs: CSR

```
struct SparseMatrixCSR {
  double * values;
  int * col_indices;
  int * row_offsets;
  int N;
  int nnz; };

void spmv_csr(SparseMatrixCSR m, double *x, double *y){
#pragma omp parallel for
  for (int i=0; i<m.N; ++i){
    double dot = 0.0;
    int row_start = m.rowoffsets[i];
    int row_end = m.rowoffsets[i+1];
    for (int j=start; i<m.end; ++j)
      dot += m.val[j] * x[m.col_indices[j]];
    y[i] += dot;
  }
}
```

values:	1	7	2	8	5	3	9	6	4
col indices:	0	1	1	2	0	2	3	1	3
row offsets:	0	2	4	7	9				

- Pros: CSR storage format permits a variable number of nonzeros per row without wasted space
- Cons: (1) non-continuous memory access to data; (2) thread divergence

Sequential SpMV: COO

```
struct SparseMatrixCOO {
  double * values;
  int * col_indices;
  int * row_indices;
  int N;
  int nnz; };

void spmv_coo(SparseMatrixCOO m, double *x, double *y){

  for (int i=0; i<m.nnz; ++i){
    y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
  }
}
```

values:	1	7	2	8	5	3	9	6	4
row indices:	0	0	1	1	2	2	2	3	3
col indices:	0	1	1	2	0	2	3	1	3

This is a very satisfyingly simple function.

Parallel SpMV on CPUs: COO

```
struct SparseMatrixCOO {
  double * values;
  int * col_indices;
  int * row_indices;
  int N;
  int nnz; };

void spmv_coo(SparseMatrixCOO m, double *x, double *y){
???
  for (int i=0; i<m.nnz; ++i){
    y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
  }
}
```

values:	1	7	2	8	5	3	9	6	4
row indices:	0	0	1	1	2	2	2	3	3
col indices:	0	1	1	2	0	2	3	1	3

Parallel SpMV on CPUs: COO

```
struct SparseMatrixCOO {
  double * values;
  int * col_indices;
  int * row_indices;
  int N;
  int nnz; };

void spmv_coo(SparseMatrixCOO m, double *x, double *y){
???
  for (int i=0; i<m.nnz; ++i){
    y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
  }
}
```

values:	1	7	2	8	5	3	9	6	4
row indices:	0	0	1	1	2	2	2	3	3
col indices:	0	1	1	2	0	2	3	1	3

Parallel SpMV on CPUs: COO

```
struct SparseMatrixCOO {
  double * values;
  int * col_indices;
  int * row_indices;
  int N;
  int nnz; };

void spmv_coo(SparseMatrixCOO m, double *x, double *y){
???
  for (int i=0; i<m.nnz; ++i){
    y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
  }
}
```

values:	1	7	2	8	5	3	9	6	4
row indices:	0	0	1	1	2	2	2	3	3
col indices:	0	1	1	2	0	2	3	1	3

Parallel SpMV on CPUs: COO

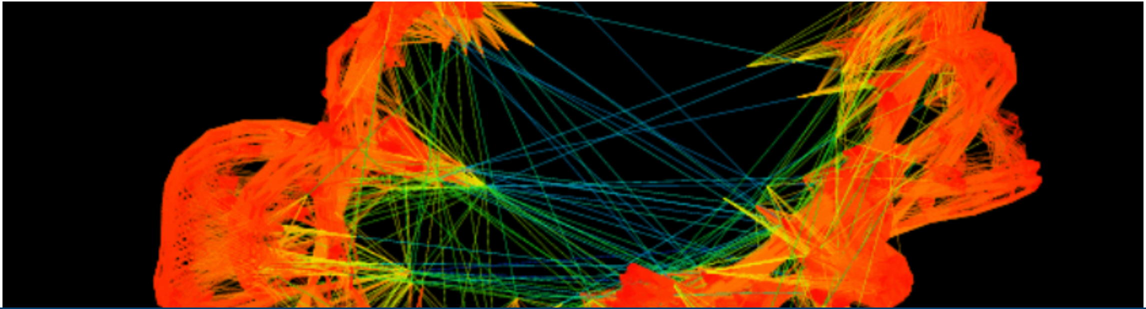
```
struct SparseMatrixCOO {
  double * values;
  int * col_indices;
  int * row_indices;
  int N;
  int nnz; };

void spmv_coo(SparseMatrixCOO m, double *x, double *y){
???
  for (int i=0; i<m.nnz; ++i){
    y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
  }
}
```

values:	1	7	2	8	5	3	9	6	4
row indices:	0	0	1	1	2	2	2	3	3
col indices:	0	1	1	2	0	2	3	1	3

Oops, race condition appears because of output interference.

non-trivial solution: Segmented/Prefix scan..



Part IV: High-Performance Libraries

Make better use of libraries

If I'm never going to implement my own sparse matrix multiplication, who cares?

- Dealing with data-dependent performance and avoiding irregularity are common issues in massively-parallel programming
- If it's hard for you to write sparse matrix algorithms that work efficiently in all cases, it's hard for library implementers as well!
- Knowing the tradeoffs can help you make better use of sparse matrix libraries

List of Libraries (Opensource)

- SuiteSparse, a suite of sparse matrix algorithms, geared toward the direct solution of sparse linear systems.
- PETSc, a large C library, containing many different matrix solvers for a variety of matrix storage formats.
- Trilinos, a large C++ library, with sub-libraries dedicated to the storage of dense and sparse matrices and solution of corresponding linear systems.
- Eigen3 is a C++ library that contains several sparse matrix solvers. However, none of them are parallelized.
- MUMPS (**MU**ltifrontal **M**assively **P**arallel sparse direct **S**olver), written in Fortran90, is a frontal solver.
- deal.II, a finite element library that also has a sub-library for sparse linear systems and their solution.
- DUNE, another finite element library that also has a sub-library for sparse linear systems and their solution.
- PaStix.
- SuperLU.
- Armadillo provides a user-friendly C++ wrapper for BLAS and LAPACK.
- SciPy provides support for several sparse matrix formats, linear algebra, and solvers.
- SPArse Matrix (spam) R and Python package for sparse matrices.
- Wolfram Language Tools for handling sparse arrays
- ALGLIB is a C++ and C# library with sparse linear algebra support
- ARPACK Fortran 77 library for sparse matrix diagonalization and manipulation, using the Arnoldi algorithm
- SPARSE Reference (old) NIST package for (real or complex) sparse matrix diagonalization
- SLEPc Library for solution of large scale linear systems and sparse matrices
- Sympiler, a domain-specific code generator and library for solving linear systems and quadratic programming problems.
- Scikit-learn A Python package for data analysis including sparse matrices.
- sprs implements sparse matrix data structures and linear algebra algorithms in pure Rust.

https://en.wikipedia.org/wiki/Sparse_matrix

List of Libraries

Two libraries support high-performance sparse matrix computations on CLAIX:

- Intel MKL: `https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html`
- Nvidia cuSPARSE: `https://developer.nvidia.com/cusparse`

Intel MKL: Inspector-executor Sparse BLAS Routines

Supports[1]:

- Sparse matrix-vector multiplication
- Sparse matrix-matrix multiplication with a sparse or dense result
- Solution of triangular systems
- Sparse matrix addition
- supported formats are:
  - CSR
  - CSC
  - COO
  - BSR

It divides operations into two stages:

- analysis: inspecting the matrix sparsity pattern and applies matrix structure changes
- execution: subsequent routine calls reuse this information in order to improve performance

[1] https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/blas-and-sparse-blas-routines/inspector-executor-sparse-blas-routines.html

Intel MKL: API for SpMV

Single call

```
mkl_sparse_create_d_csr ( &A, SPARSE_INDEX_BASE_ZERO,
                         rows, cols, rowsStart, rowsEnd, colIndx, values );

mkl_sparse_d_mv ( SPARSE_OPERATION_NON_TRANSPOSE,
                 alpha, A, SPARSE_FULL, x, beta, y );

mkl_sparse_destroy ( A );
```

(source)

Intel MKL: API for SpMV

```
Iterative method:
mkl_sparse_create_d_csr ( &A, SPARSE_INDEX_BASE_ZERO, rows, cols, rowsStart, rowsEnd,
colIndx, values );
mkl_sparse_set_mv_hint ( A, SPARSE_OPERATION_NON_TRANSPOSE, SPARSE_FULL, n_iter );
mkl_sparse_set_memory_hint ( A, SPARSE_MEMORY_AGRESSIVE );
mkl_sparse_optimize ( A );

for (int i=0;i<n_iter;i++) {
      mkl_sparse_d_mv ( SPARSE_OPERATION_NON_TRANSPOSE,  alpha, A, SPARSE_FULL, x, beta,
      y  );
      ...
}
mkl_sparse_destroy( A );
```

(source)

cuSPARSE

Key features[2]:

- Full suite of sparse routines covering sparse vector x dense vector operations, sparse matrix x dense vector operations, and sparse matrix x dense matrix operations.
- Routines for sparse matrix x sparse matrix addition and multiplication
- Generic high-performance APIs for sparse-dense vector multiplication (SpVV), sparse matrix-dense vector multiplication (SpMV), and sparse matrix-dense matrix multiplication (SpMM)

It provides GPU-accelerated basic linear algebra subroutines for sparse matrices that perform significantly faster than CPU-only alternatives.

[2]https://developer.nvidia.com/cusparse

cuSPARSE: API for SpMV

```
//The function cusparseSpMV_bufferSize() returns the size of the workspace needed by cusparseSpMV()
cusparseStatus_t cusparseSpMV_bufferSize(cusparseHandle_t handle, cusparseOperation_t opA, const void*     alpha,
        cusparseSpMatDescr_t matA, cusparseDnVecDescr_t vecX, const void* beta, cusparseDnVecDescr_t vecY, cudaDataType computeType,
        cusparseSpMVAlg_t alg, size_t* bufferSize);

cusparseStatus_t cusparseSpMV(cusparseHandle_t handle, cusparseOperation_t opA, const void* alpha, cusparseSpMatDescr_t matA,
        cusparseDnVecDescr_t vecX, const void* beta, cusparseDnVecDescr_t vecY, cudaDataType computeType, cusparseSpMVAlg_t alg, void
        * externalBuffer);
```

https://docs.nvidia.com/cuda/cusparse/index.html#cusparse-generic-function-spmv

The sparse matrix formats currrently supported are listed below:

- CUSPARSE_FORMAT_COO
- CUSPARSE_FORMAT_CSR

Hands-on

1. Checkout the structure of assignments
2. Checkout the matrix files in the `./data` and understand the MatrixMarket format
3. try to compile the example within `./hands-on`
   - `cd tasks`
   - `mkdir build`
   - `cd build`
   - `cmake ..`
   - `make`
4. print out the memory requirement per nonzero element for different matrices
   - `./hands-on/getMemSize.exe ../data/YourMatrixMarketFile.mtx`
5. Checkout the SuiteSparse Matrix Collection and its search engine

Homework 1

Implement SpMV for different format by hand

- Implement sequential SpMV for `COO`, `CSR`, `DIA` and `ELL`
- Naïve parallelization with OpenMP
- Test with different matrices and number of threads

Homework 2

Implement SpMV based on MKL

- complete the implementation of SpMV based on MKL for both `COO` and `CSR` formats.
- fill in the missing input arguments when calling the MKL routines.
  - mkl_sparse_?_create_coo: →API←
  - mkl_sparse_?_create_csr: →API←
  - mkl_sparse_?_mv: →API←
- test with different matrices and compare their performance by considering the diversity of sparsity pattern

Bonus

First try of SpMV based on cuSPARSE on single GPU for both COO and CSR

The codes are already completed, the tasks are:

- Run them on supercomputer CLAIX with different matrices (sparsity pattern, size, etc)
- Identify the time cost of memory transfers between GPU and CPU as a fraction of total time of execution

Takeaways:

- Sparse matrices are hard!
- There are a lot of ways to represent sparse matrices with different storage requirements
- Storage requirements depends differently on the sparsity pattern
- There is sometimes a need to safeguard against worst-case input
- There is often a trade-off between regularity and efficiency

Next Lectures:

- Conjugate Gradient method (CG)
- PageRank algorithm based on power iteration method