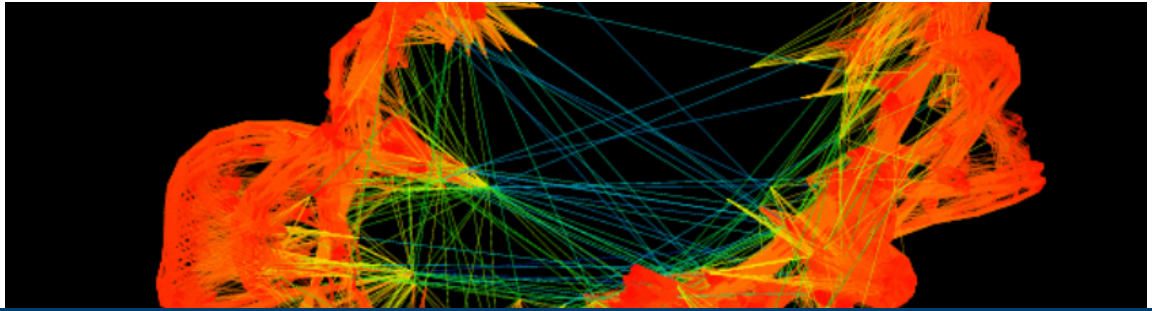


High-Performance Matrix Computations

Sparse Matrix Representations and Computations

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

RWTH AACHEN UNIVERSITY

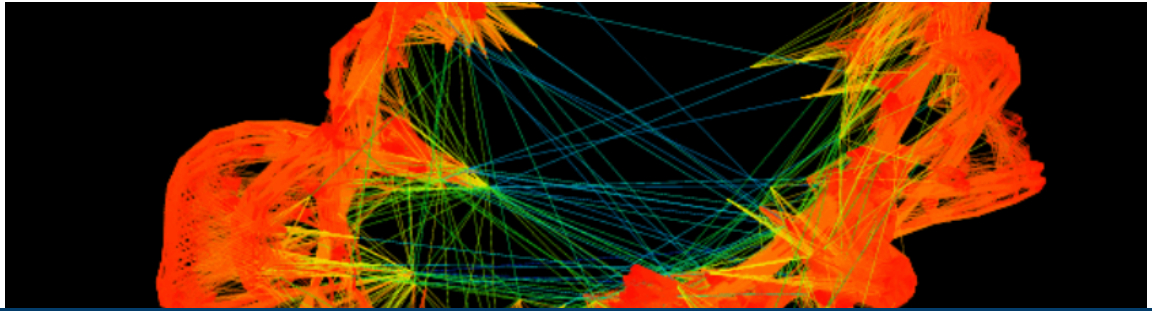
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Organisation of This Module

Topics: High-Performance Computations of Sparse Matrices

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

- Lectures based on slides
- Practical examples and exercises
  1. Part 1: C codes on Laptop and CLAIX
  2. Part 2: Jupyter notebooks with Julia on Laptop



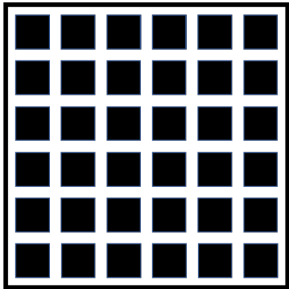
Part I: Sparse Matrix

RWTH AACHEN UNIVERSITY

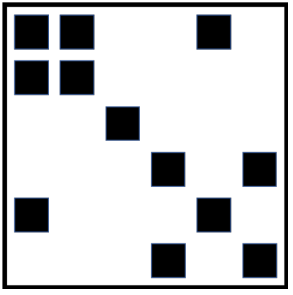
JÜLICH Forschungszentrum
JÜLICH SUPERCOMPUTING CENTRE

Sparse Matrices

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



Dense Matrix

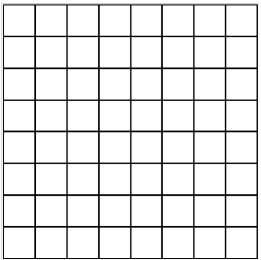


Sparse Matrix

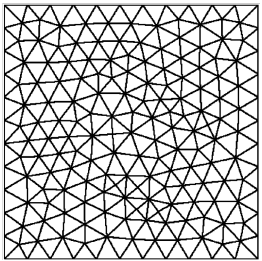
For a $N \times N$ sparse matrix A, the number of non-zeros elements (nnz) is $\mathcal{O}(N)$. The sparsity is defined as $\frac{nnz}{N^2}$.

Sparse Matrices

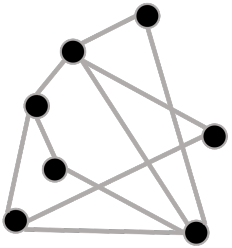
Non-zeros encode connectivity: Finite-Elements Meshes, Hyperlinks, Social Networks, ···



Structured Mesh



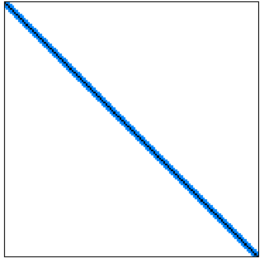
Unstructured Mesh



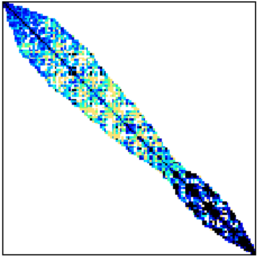
Indirected Graph

Sparsity Patterns

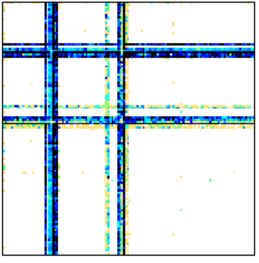
- Mesh type: Elements, structured, unstructured, $\cdots$
- Problem dimension (2D, 3D)
- Discretization method
- Graph (connections, directed, indirected, $\cdots$)



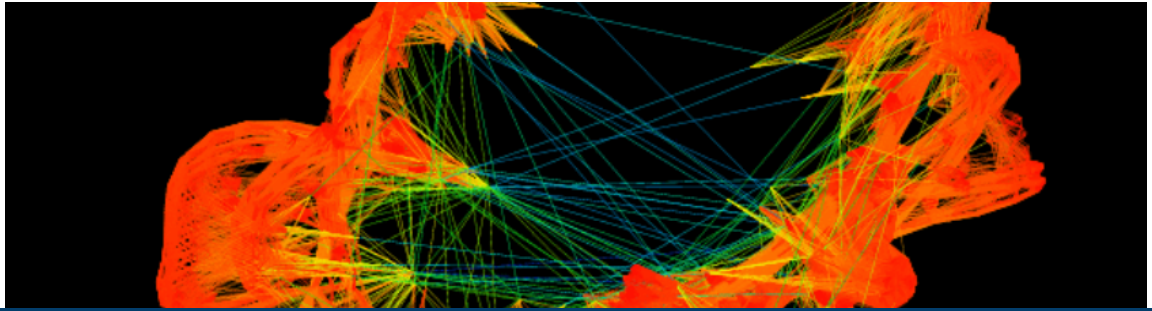
Laplace eqn 2D mesh (Link)



electromagnetic (Link)



Packet trace data (Link)



Part II: Sparse Matrix Storage Formats

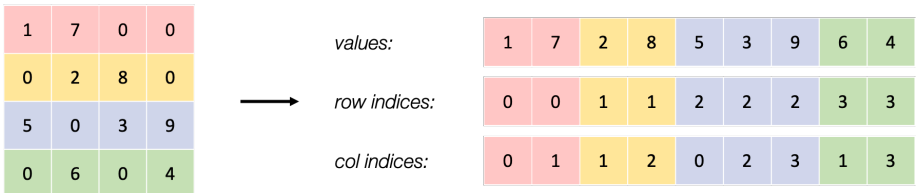
RWTH AACHEN UNIVERSITY

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Matrix Format: Coordinate (COO)

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

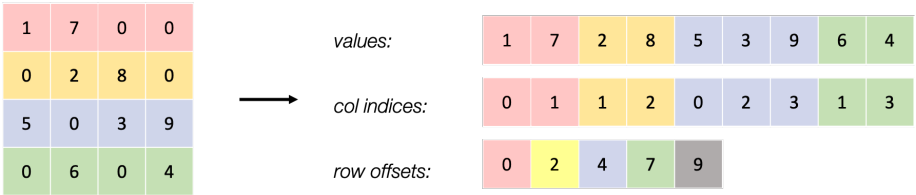
- Row index (int) (*nnz*)
- Column index (int) (*nnz*)
- Values (data type) (*nnz*)



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*)



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

values:

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

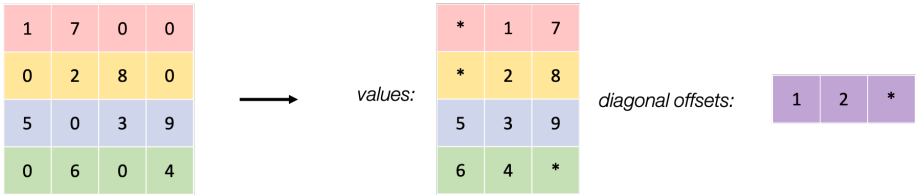
column indices:

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) ($N * M$)



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

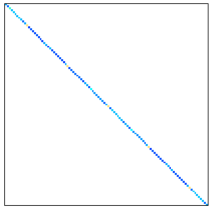
Format	Structure (words)	Values
Dense	-	N^2
COO	$2 \times nnz$	nnz
CSR	$N + 1 + nnz$	nnz
ELL	$M \times N$	$M \times N$
DIA	D	$D \times N$

Hands-on

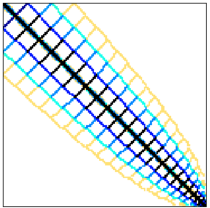
1. prepare a readme.txt
2. compile with CMake
3. show test matrix suite website and search engine
4. try

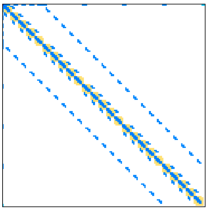
Storage Format Comparison

Bytes per Nonzero Entry



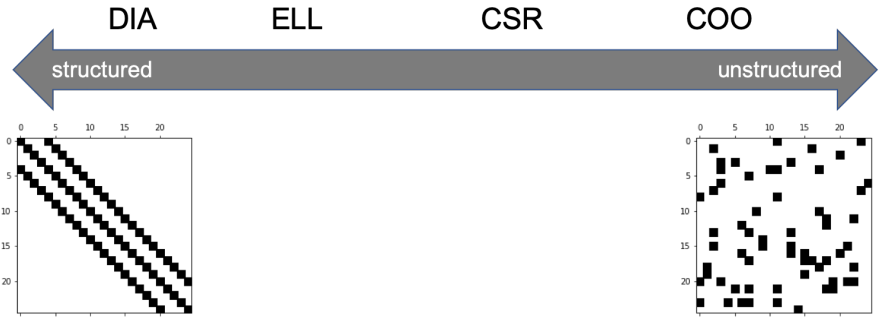
- COO: 16.00
- CSR: 16.01
- DIA: 8.01
- ELL: 12.00





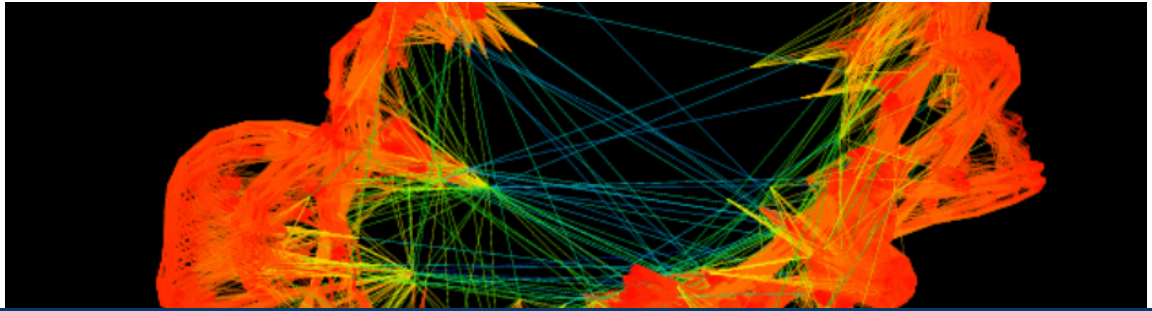
- COO: 16.00
- CSR: 12.93
- DIA: 586.72
- ELL: 145.27

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
- **Packet (PKT):**
  - Reorders rows and columns to concentrate nonzeros into roughly diagonal submatrices
  - This improves cache performance as nearby rows access nearby x elements
- **Jagged Diagonal Storage (JDS):**
  - Group similarly dense rows into evenly-sized partitions, and represent each section independently using either CSR or ELL
  - It can be naturally mapped to CUDA blocks, which ensures the same amounts of computation of threads within the same block.
- **Other Hybrid methods (HYB):**
  - It is used for the irregular sparse matrices, e.g., ELL handles *typical* entries and COO handles *exceptional* entries



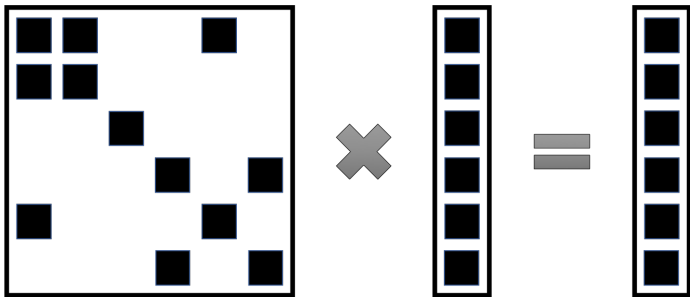
Part III: Sparse Matrix-Vector Multiplication (SpMV)

RWTH AACHEN UNIVERSITY

JÜLICH Forschungszentrum
JÜLICH SUPERCOMPUTING CENTRE

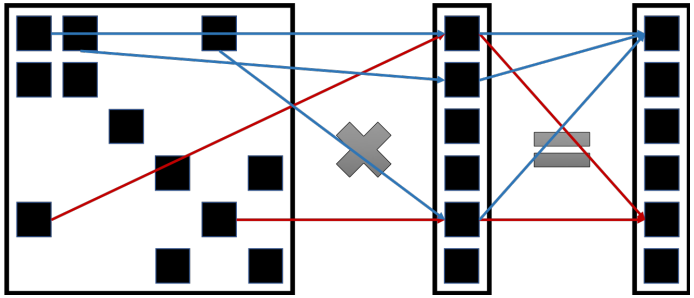
Sparse Matrix-Vector Multiplication (SpMV)

- 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: COO

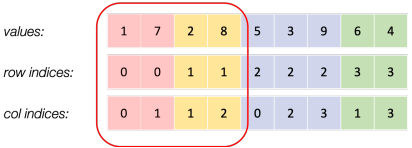
```
1  struct SparseMatrixCOO {
2    double * values;
3    int * col_indices;
4    int * row_indices;
5    int N;
6    int nnz; };
7
8  void spmv_coo(SparseMatrixCOO m, double *x, double *y){
9
10   for (int i=0; i<m.nnz; ++i){
11     y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
12   }
13 }
```

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.

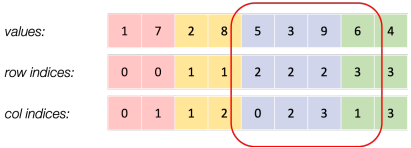
Parallel SpMV on CPUs: COO

```
1  struct SparseMatrixCOO {
2    double * values;
3    int * col_indices;
4    int * row_indices;
5    int N;
6    int nnz; };
7
8  void spmv_coo(SparseMatrixCOO m, double *x, double *y){
9  #pragma omp parallel for
10   for (int i=0; i<m.nnz; ++i){
11     y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
12   }
13 }
```



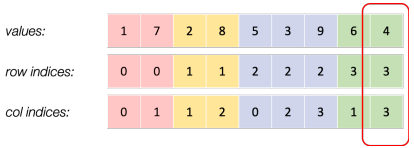
Parallel SpMV on CPUs: COO

```
1  struct SparseMatrixCOO {
2    double * values;
3    int * col_indices;
4    int * row_indices;
5    int N;
6    int nnz; };
7
8  void spmv_coo(SparseMatrixCOO m, double *x, double *y){
9  #pragma omp parallel for
10   for (int i=0; i<m.nnz; ++i){
11     y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
12   }
13 }
```



Parallel SpMV on CPUs: COO

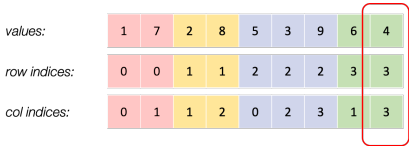
```
1  struct SparseMatrixCOO {
2    double * values;
3    int * col_indices;
4    int * row_indices;
5    int N;
6    int nnz; };
7
8  void spmv_coo(SparseMatrixCOO m, double *x, double *y){
9  #pragma omp parallel for
10   for (int i=0; i<m.nnz; ++i){
11     y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
12   }
13 }
```



Oops, race condition appears because of output interference.

Parallel SpMV on CPUs: COO

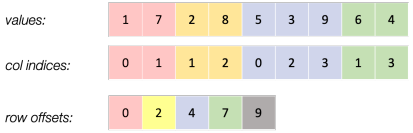
```
1  struct SparseMatrixCOO {
2    double * values;
3    int * col_indices;
4    int * row_indices;
5    int N;
6    int nnz; };
7
8  void spmv_coo(SparseMatrixCOO m, double *x, double *y){
9  #pragma omp parallel for
10   for (int i=0; i<m.nnz; ++i){
11     y[m.row_indices[i]] += m.values[i] * x[m.col_indices[i]];
12   }
13 }
```



Switching to an atomic addition will make the output of this kernel correct...
with a potentially large number of serialized writes...
Anyway, this format is better suited to sequential hardware...

Sequential SpMV: CSR

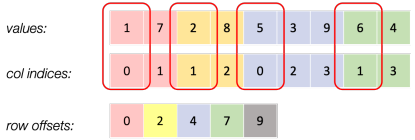
```
1  struct SparseMatrixCSR {
2    double * values;
3    int * col_indices;
4    int * row_offsets;
5    int N;
6    int nnz; };
7
8  void spmv_csr(SparseMatrixCSR m, double *x, double *y){
9
10   for (int i=0; i<m.N; ++i){
11     for (int j=m.row_offsets[i]; i<m.row_offsets[i+1]; ++j){
12       y[i] += m.values[j] * x[m.col_indices[j]];
13     }
14   }
15 }
```



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

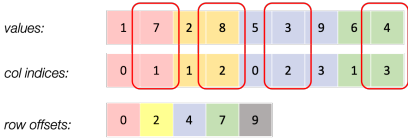
Parallel SpMV on CPUs: CSR

```
1  struct SparseMatrixCSR {
2    double * values;
3    int * col_indices;
4    int * row_offsets;
5    int N;
6    int nnz; };
7
8  void spmv_csr(SparseMatrixCSR m, double *x, double *y){
9  #pragma omp parallel for
10   for (int i=0; i<m.N; ++i){
11     for (int j=m.row_offsets[i]; i<m.row_offsets[i+1]; ++j){
12       y[i] += m.values[j] * x[m.col_indices[j]];
13     }
14   }
15 }
```



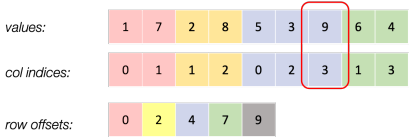
Parallel SpMV on CPUs: CSR

```
1  struct SparseMatrixCSR {
2    double * values;
3    int * col_indices;
4    int * row_offsets;
5    int N;
6    int nnz; };
7
8  void spmv_csr(SparseMatrixCSR m, double *x, double *y){
9  #pragma omp parallel for
10   for (int i=0; i<m.N; ++i){
11     for (int j=m.row_offsets[i]; i<m.row_offsets[i+1]; ++j){
12       y[i] += m.values[j] * x[m.col_indices[j]];
13     }
14   }
15 }
```



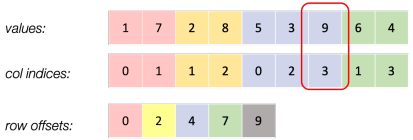
Parallel SpMV on CPUs: CSR

```
1  struct SparseMatrixCSR {
2    double * values;
3    int * col_indices;
4    int * row_offsets;
5    int N;
6    int nnz; };
7
8  void spmv_csr(SparseMatrixCSR m, double *x, double *y){
9  #pragma omp parallel for
10   for (int i=0; i<m.N; ++i){
11     for (int j=m.row_offsets[i]; i<m.row_offsets[i+1]; ++j){
12       y[i] += m.values[j] * x[m.col_indices[j]];
13     }
14   }
15 }
```



Parallel SpMV on CPUs: CSR

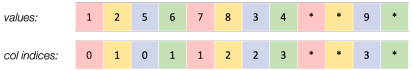
```
1  struct SparseMatrixCSR {
2    double * values;
3    int * col_indices;
4    int * row_offsets;
5    int N;
6    int nnz; };
7
8  void spmv_csr(SparseMatrixCSR m, double *x, double *y){
9  #pragma omp parallel for
10   for (int i=0; i<m.N; ++i){
11     for (int j=m.row_offsets[i]; i<m.row_offsets[i+1]; ++j){
12       y[i] += m.values[j] * x[m.col_indices[j]];
13     }
14   }
15 }
```



Control flow divergence: each row involves a variable amount of computation.

Sequential SpMV: ELL

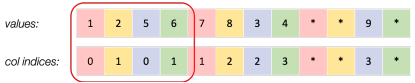
```
1  struct SparseMatrixELL {
2    double * values;
3    int * col_indices;
4    int N;
5    int max_row; };
6
7  void spmv_ell(SparseMatrixELL m, double *x, double *y){
8
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; j<m.max_row; ++j){
11       const int idx = i + j * m.N;
12       y[i] += a.values[idx] * x[m.col_indices[idx]];
13     }
14   }
15 }
```



Two options for padding the empty elements: Place zeros in values OR place an invalidating indicator into either array.

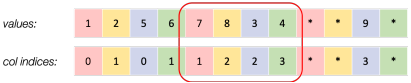
Parallel SpMV on CPUs: ELL

```
1  struct SparseMatrixELL {
2    double * values;
3    int * col_indices;
4    int N;
5    int max_row; };
6
7  void spmv_ell(SparseMatrixELL m, double *x, double *y){
8  #pragma omp parallel for
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; j<m.max_row; ++j){
11       const int idx = i + j * m.N;
12       y[i] += a.values[idx] * x[m.col_indices[idx]];
13     }
14   }
15 }
```



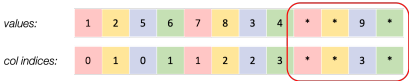
Parallel SpMV on CPUs: ELL

```
1  struct SparseMatrixELL {
2    double * values;
3    int * col_indices;
4    int N;
5    int max_row; };
6
7  void spmv_ell(SparseMatrixELL m, double *x, double *y){
8  #pragma omp parallel for
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; j<m.max_row; ++j){
11       const int idx = i + j * m.N;
12       y[i] += a.values[idx] * x[m.col_indices[idx]];
13     }
14   }
15 }
```



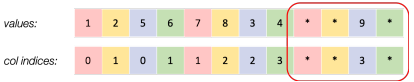
Parallel SpMV on CPUs: ELL

```
1  struct SparseMatrixELL {
2    double * values;
3    int * col_indices;
4    int N;
5    int max_row; };
6
7  void spmv_ell(SparseMatrixELL m, double *x, double *y){
8  #pragma omp parallel for
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; j<m.max_row; ++j){
11       const int idx = i + j * m.N;
12       y[i] += a.values[idx] * x[m.col_indices[idx]];
13     }
14   }
15 }
```



Parallel SpMV on CPUs: ELL

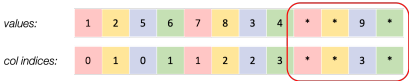
```
1  struct SparseMatrixELL {
2    double * values;
3    int * col_indices;
4    int N;
5    int max_row; };
6
7  void spmv_ell(SparseMatrixELL m, double *x, double *y){
8  #pragma omp parallel for
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; j<m.max_row; ++j){
11       const int idx = i + j * m.N;
12       y[i] += a.values[idx] * x[m.col_indices[idx]];
13     }
14   }
15 }
```



It performs well for matrices with similarly-dense rows.

Parallel SpMV on CPUs: ELL

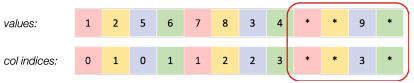
```
1  struct SparseMatrixELL {
2    double * values;
3    int * col_indices;
4    int N;
5    int max_row; };
6
7  void spmv_ell(SparseMatrixELL m, double *x, double *y){
8  #pragma omp parallel for
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; j<m.max_row; ++j){
11       const int idx = i + j * m.N;
12       y[i] += a.values[idx] * x[m.col_indices[idx]];
13     }
14   }
15 }
```



A worst-case: 1000 $\times$ 1000 matrix with sparsity 0.01, it requires 1000 * 1000 * 0.01=10,000 multiply/adds. If the densest row has 200 non-zeros values, then SpMV with ELL format performs 1000 * 200 = 200,000 multiply/adds.

Parallel SpMV on CPUs: ELL

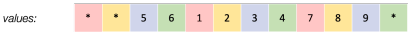
```
1  struct SparseMatrixELL {
2    double * values;
3    int * col_indices;
4    int N;
5    int max_row; };
6
7  void spmv_ell(SparseMatrixELL m, double *x, double *y){
8  #pragma omp parallel for
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; j<m.max_row; ++j){
11       const int idx = i + j * m.N;
12       y[i] += a.values[idx] * x[m.col_indices[idx]];
13     }
14   }
15 }
```



20× MORE computation and memory requirement!!!

Sequential SpMV: DIA

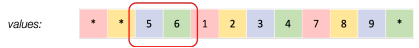
```
1  struct SparseMatrixDIA {
2    double * values;
3    int * diag;
4    int N;
5    int ndiag; };
6
7  void spmv_dia(SparseMatrixDIA m, double *x, double *y){
8
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; i<m.ndiag; ++j){
11       int start = max(-a->diag[j], 0);
12       int end = a->diag[j] > 0 ?  a->m - a->diag[j] : a->m;
13       if((i >= start) && (i < end)){
14         y[i] += a->val[j * a->m + i] * x[i + a->diag[j]];
15       }
16     }
17   }
18 }
```



Similar as ELL, two options for padding the empty elements: Place zeros in values OR place an invalidating indicator into either array.

Parallel SpMV on CPUs: DIA

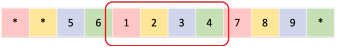
```
 1  struct SparseMatrixDIA {
 2    double * values;
 3    int * diag;
 4    int N;
 5    int ndiag; };
 6
 7  void spmv_dia(SparseMatrixDIA m, double *x, double *y){
 8  #pragma omp parallel for
 9    for (int i=0; i<m.N; ++i){
10      for (int j=0; i<m.ndiag; ++j){
11        int start = max(-a->diag[j], 0);
12        int end = a->diag[j] > 0 ?  a->m - a->diag[j] : a->m;
13        if((i >= start) && (i < end)){
14        y[i] += a->val[j * a->m + i] * x[i + a->diag[j]];
15        }
16      }
17    }
18  }
```



Parallel SpMV on CPUs: DIA

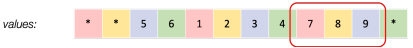
```
 1  struct SparseMatrixDIA {
 2    double * values;
 3    int * diag;
 4    int N;
 5    int ndiag; };
 6
 7  void spmv_dia(SparseMatrixDIA m, double *x, double *y){
 8  #pragma omp parallel for
 9    for (int i=0; i<m.N; ++i){
10      for (int j=0; i<m.ndiag; ++j){
11        int start = max(-a->diag[j], 0);
12        int end = a->diag[j] > 0 ?  a->m - a->diag[j] : a->m;
13        if((i >= start) && (i < end)){
14        y[i] += a->val[j * a->m + i] * x[i + a->diag[j]];
15        }
16      }
17    }
18  }
```

values:



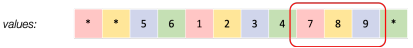
Parallel SpMV on CPUs: DIA

```
1  struct SparseMatrixDIA {
2    double * values;
3    int * diag;
4    int N;
5    int ndiag; };
6
7  void spmv_dia(SparseMatrixDIA m, double *x, double *y){
8  #pragma omp parallel for
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; i<m.ndiag; ++j){
11       int start = max(-a->diag[j], 0);
12       int end = a->diag[j] > 0 ?  a->m - a->diag[j] : a->m;
13       if((i >= start) && (i < end)){
14       y[i] += a->val[j * a->m + i] * x[i + a->diag[j]];
15       }
16     }
17   }
18 }
```



Parallel SpMV on CPUs: DIA

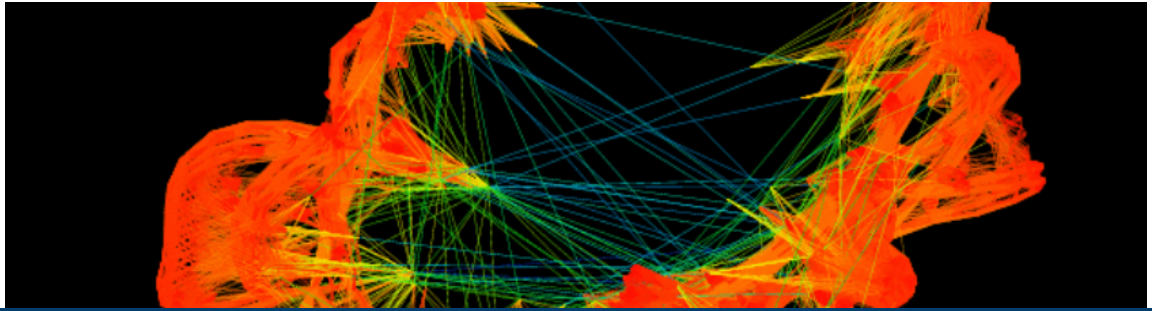
```
1  struct SparseMatrixDIA {
2    double * values;
3    int * diag;
4    int N;
5    int ndiag; };
6
7  void spmv_dia(SparseMatrixDIA m, double *x, double *y){
8  #pragma omp parallel for
9    for (int i=0; i<m.N; ++i){
10     for (int j=0; i<m.ndiag; ++j){
11       int start = max(-a->diag[j], 0);
12       int end = a->diag[j] > 0 ?  a->m - a->diag[j] : a->m;
13       if((i >= start) && (i < end)){
14       y[i] += a->val[j * a->m + i] * x[i + a->diag[j]];
15       }
16     }
17   }
18 }
```



It only performs well for diagonal matrices with: (1) limited number of diagonals; (2) dense diagonals.

Hands-on

- OpenMP parallelization and test with multiple matrices.



Part IV: High-Performance Libraries

RWTH AACHEN UNIVERSITY

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

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

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

```
1 mkl_sparse_create_d_csr ( &A, SPARSE_INDEX_BASE_ZERO, rows, cols, rowsStart, rowsEnd, colIndx, values );
2
3 mkl_sparse_d_mv ( SPARSE_OPERATION_NON_TRANSPOSE, alpha, A, SPARSE_FULL, x, beta, y );
4
5 mkl_sparse_destroy ( A );
```

The sparse matrix formats currrently supported are listed below:

- CSR
- CSC
- COO
- BSR

cuSPARSE

Key features[2]:

- Support for dense, COO, CSR, CSC, and Blocked CSR sparse matrix formats
- 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

```
1 //The function cusparseSpMV_bufferSize() returns the size of the workspace needed by cusparseSpMV()
2 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);
3
4 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

Homework

TO DO Test different matrices with intel MKL and cuSPARSE on CLAIX. Codes will be provided.

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