**1.** We start leisurely by considering the following tasks.

(a) Don't use C-code for the answer.

```
1  // 1. Get the value of a and b as well as of *c, *d and **e.
2  // 2. What is the value of c, d and e?
3  int a = 2, b = 5, *c = &a, *d = &b;
4  int* e[1] = {&a};
5   a  =  *c * *d;
6   d  =   c;
7  *d *=  *c * *d;
8   a *= **e = 10;
9
```

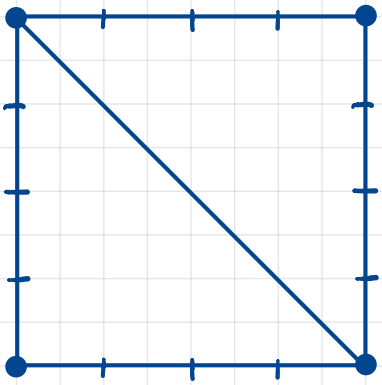
a=2, b=5, c=Zeigerauf(a), d=Zeigerauf(b)
e=Array von Pointern(Zeigerauf(a))
a=2*5=10
d=Zeigerauf(a)
a=2*2*5=20
a=10*10=100

(b) Have a look at the material and make yourself familiar with the `structs`, `functions`, `data types` and `utilities`.

(c) Read this brief introduction to `makefiles`.

(d) Run the `makefile` and explain the output.

(4+Karma-Point+Karma-Point+2 = 6 Punkte + 2 Karma-Points)



1. Compilieren und linken
2. führt die Datei hpc_demo0 aus für zwei Probleme.

   In den Problem wird ein grobes Netz geladen, welches anschließend verfeinert wird. Wobei die Matrizen im SED (Sparse Extracted Diagonal) Format gespeichert werden.

   Zu einer gewählten rechten Seite wird $Ax=b$ näherungsweise mit Gauß-Seidel gelöst und das Residuum berechnet.

**First impressions of storage formats, algorithms**

**2.** In this exercise we want to get a first impression why storage formats are crucial. We concern ourselves with the storage formats from the lecture. As an example, we introduce a first simple sparse matrix storage format, namely the `COO` (coordinate) format, which is also known as the *triplet format*. If a matrix $M$ is given in the `COO` format, then

- $M \to m$; denotes the number of rows.
- $M \to n$; denotes the number of columns.
- $M \to nz$; is the number of entries.
- $M \to nzmax$; is the number of numerical values which is greater or equal to $nz$. The reason is that we allocate a puffer to store at least $nz$ values.
- $Mx = M \to x$; is an array, which contains the numerical values of $M$ and has the length $nzmax$.
- $M \to p$; is an array, which contains the column indices for $Mx$ and has the length $nzmax$.
- $M \to ind$; is an array, which contains the row indices for $Mx$ and has the length $nzmax$.

The `COO` format is not ordered and permits duplicate entries. If the matrix gets larger during runtime, the new entries are appended to the data array. For instance, the data

$$\begin{aligned} m = 3, \quad n = 3, \quad nzmax = 8, \\ Mx = [1.0 \ 9.0 \ 2.0 \ 5.4 \ 5.5 \ * \ * \ *], \\ p = [0 \ 0 \ 1 \ 2 \ 0 \ * \ * \ *], \quad ind = [0 \ 1 \ 2 \ 1 \ 2 \ * \ * \ *], \quad nz = 5. \end{aligned}$$

---

* Generally: Results are to be explained and code is to be commented.

gives the matrix $M$

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 9 & 0 & 5.4 \\ 5.5 & 2 & 0 \end{pmatrix}. \tag{1}$$

In C, we combined the matrix formats `COO`, `CRS` and `CCS` in one type `cs`.

Explain the following sparse matrix storage formats:

(a) `CRS` format (*Compressed Row Storage*), `CCS` format (*Compressed Column Storage*),

(b) `SED` format (*Sparse Extracted Diagonal*), also called Modified Column Storage,

(c) `CDS` format (*Compressed Diagonal Storage*),

(d) `JDS` format (*Jagged Diagonal Storage*),

(e) `SKY` format (*Skyline Storage*).

What are the advantages of each format? (2+4+2+2+2 = 12 Punkte)

a) CRS (Compressed Row Storage) / CCS (Compressed Column Storage)

Bsp.: CRS

$$\begin{pmatrix} 1 & 0 & 0 \\ 9 & 0 & 5,4 \\ 5,5 & 2 & 0 \end{pmatrix} \rightarrow$$

| 1 | 9 | 5,4 | 5,5 | 2 |
|---|---|---|---|---|

| 0 | 0 | 2 | 0 | 1 |
|---|---|---|---|---|

| 0 | 1 | 3 |
|---|---|---|

Speichere 3 Arrays

1. Array - speichert Zeilen-weise alle nicht Null Einträge

2. Array - speichert die Spalten-Indices zu den Einträgen aus dem 1. Array

3. Array - speichert die Row-Offsets

Pro:

- Umwandlung kann in linearer Komplexität durchgeführt werden

## b) SED (Sparse Extracted Diagonal)

Bsp.: $\begin{pmatrix} 11 & 12 & 0 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 0 & 33 & 0 \\ 41 & 0 & 0 & 44 \end{pmatrix} \longrightarrow$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| values | 11 | 22 | 33 | 44 | * | 21 | 31 | 41 | 12 | 23 | 14 | 24 | □ |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| ptr_ind | 5 | 8 | 9 | 11 | 12 | 1 | 2 | 3 | 0 | 1 | 0 | 1 | |

Speichere zwei Vektoren

1. Vektor - real/double Vektor für Matrix Einträge
2. Vektor - index Vektor

Pro:

- Umwandlung kann in linearer Komplexität durchgeführt werden.
- Diagonal Einträge sind direkt zugänglich (Jacobi-Method)

# c) CDS (Compressed Diagonal Storage)

Bsp.:

$$\begin{pmatrix} 11 & 12 & 0 & 0 & 0 \\ 21 & 22 & 23 & 0 & 0 \\ 0 & 32 & 33 & 34 & 0 \\ 0 & 0 & 43 & 44 & 45 \\ 0 & 0 & 0 & 54 & 55 \end{pmatrix} \longrightarrow \begin{array}{|ccc|} \hline * & 11 & 12 \\ 21 & 22 & 23 \\ 32 & 33 & 34 \\ 43 & 44 & 45 \\ 54 & 55 & * \\ \hline \end{array}$$

| -1 | 0 | 1 |
|---|---|---|

Speicherung der Diagonalen und Nebendiagonalen mit Indizes:

-1 für erste Subdiagonale, -2 für zweite usw.

0 für Diagonale

1, 2 für obere Diagonalen

Pro:

- Dieses Format ist für nicht symmetrische, diagonal konstruierte Matrizen vor gesehen.
  z.B. für strukturierte Matrizen von der finite Elemente Diskretisierung eines Tensor-Produkt-Gitter

# d) JDS (Jagged Diagonal Storage)

**Bsp.:**

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

→

shift left

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

Column pos.

| | | |
|---|---|---|
| 1 | 3 | 4 |
| 5 | 0 | 0 |
| 2 | 5 | 0 |
| 3 | 4 | 5 |
| 4 | 8 | 0 |

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

→

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| val | 6 | 3 | 2 | 4 | 3 | 9 | 2 | 7 | 8 | 5 | 1 |
| col_ind | 1 | 3 | 2 | 4 | 5 | 3 | 4 | 5 | 8 | 4 | 5 |
| index | 1 | 6 | 10 | 12 | | | | | | | |
| permutation | 1 | 4 | 3 | 5 | 2 | | | | | | |

Erst werden alle Einträge nach links geschoben und Spalten Position Notiert.
Dann wird der Länge nach Permutiert.
Zum Schluss wird alles in 4 Arrays gespeichert.

**Pro:**

- Optimiert für sparse Matrix Vektor multiplikation
- Optimiert für Vektor-Prozesse und allgemeiner SIMD-Maschinen

## e) SKY (Skyline Storage)

**Bsp.:**

$$\begin{pmatrix} 11 & 4 & 12 & 0 & 0 \\ 4 & 22 & 0 & 0 & 41 \\ 12 & 0 & 33 & 89 & 0 \\ 0 & 0 & 89 & 44 & 0 \\ 0 & 41 & 0 & 0 & 55 \end{pmatrix} \longrightarrow$$

Diagonal

| 11 | 22 | 33 | 44 | 55 |
|---|---|---|---|---|

Off-Diagonal

| 4 | 12 | 0 | 89 | 41 | 0 | 0 |
|---|---|---|---|---|---|---|

First Column Index

| 0 | 1 | 3 | 4 | 7 |
|---|---|---|---|---|

Speichere 3 Arrays

1. Array – speichert die Diagonal Einträge ab

2. Array – speichert Zeilen-weise alle Einträge ab dem ersten nicht-Null-Eintrag bis zur Diagonalen ab

3. Array – speichert die Spalten-Offsets

**Pro:**

- Bei Cholesky-Zerlegung vorteilhaft (Form wird beibehalten)
- effizienter Zugriff auf Diagonal Einträge

**Parallel matrix vector product**

**3.** In this exercise we will implement the parallel matrix vector product and in the course of this we revise some mpi functions, e.g. `MPI_Bcast` or `MPI_Scatterv`. In order to do so, let $A \in \mathbb{R}^{m \times n}$ be a dense (row major) matrix, $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ vectors. We want to compute

$$y \leftarrow \beta y + \alpha A x, \quad \alpha, \beta \in \mathbb{R}.$$

To develop our parallel algorithm, we have to think about data decomposition. There are a variety of ways to partion the matrix and vector elements. Each data decomposition results in a different parallel algorithm. We will use a straightforward way: the rowwise block striping. With this strategy, each of the $p$ processes is responsible for a contiguous group of either $\lfloor m/p \rfloor$ or $\lceil m/p \rceil$ rows of the matrix $A$. The vectors $y$ and $x$ are replicated, meaning all the vector elements are copied on all of the nodes. Each process then calculates inner products of their respective rows of $A$ and $x$, multiplies the results with $\alpha$ and add it to the scaled entries of $y$. After the inner product computation each process has some elements of the result $y$. However, the vectors are supposed to be replicated so that each process has to communicate its results to all other processes (Hint: there is a mpi function for this task: openmpi doc).

Parallel matrix vector multiplication:

(a) Why is it acceptable to store the vectors $x$ and $y$ in their entirety on each node, but not the matrix $A$?

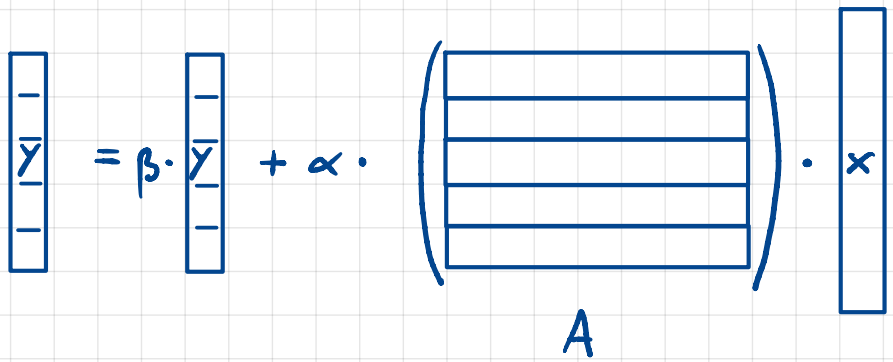
(b) Implement the parallel matrix vector multiplication explained above. Assume that $m$ is dividable by $p$, so that `MPI_Scatter` can be used.

(c) Extend your code such that $m$ do not need to be dividable by $p$. Therefore `MPI_Scatter` can not be used anymore and a function `slices` is necessary, which calculates for the input `problem_size`, `number_of_processes` and `process_rank` the `offset` and `size` of the slice (for process `process_rank`).

(d) Test your functions.

(1+5+6+4 = 16 Punkte)

a)




x muss im ganzen in jedem Knoten abgespeichert werden, da ganz x für die Berechnung in jedem Knoten gebraucht wird.

Es ist ok x und y in Gänze in jedem Knoten abzuspeichern, da sie bei Skalierung des Problems weniger anwachsen als A.