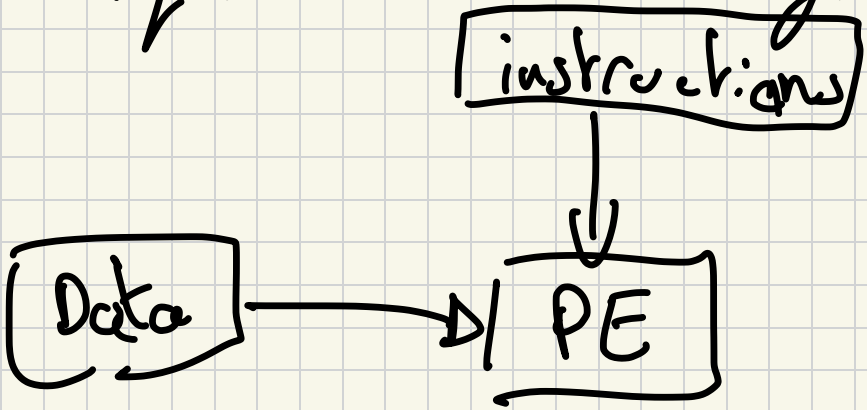
pierre @ jofiv.et

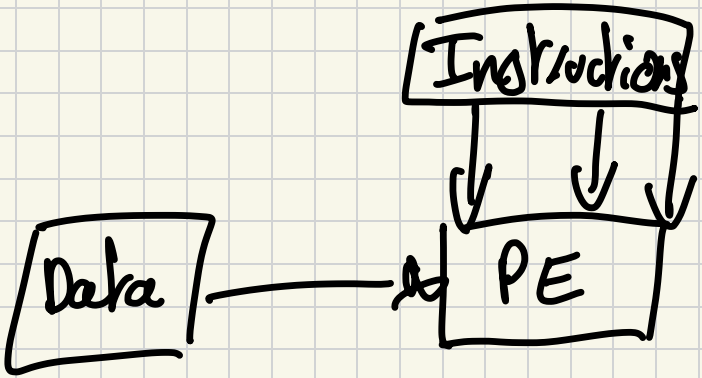
Flynn's taxonomy 1966



four kind of architectures:

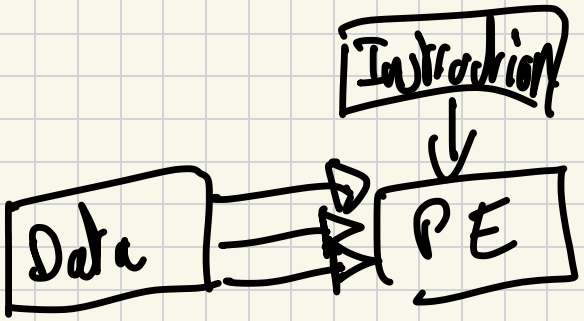
* SISD: single instruction, single data

* MISD : multiple instructions / single data



* SIMD : single instruction / multiple data

(vectorization)



* MIMD : multiple instructions, multiple data

very large and abstract of hardware, SPMD : single program multiple data (Message Passing Interface)

Hardware considerations:

* multiple PE, coordination:
  synchronous or asynchronous
* memory space:
  shared or distributed
* memory addressing:
  local or global
  (Remote Memory Addressing)

	shared-memory	distributed-memory
scaling	✗	✓
data integrity	✗	✓
performance optimisation	✗	✓
incremental parallelisation	✓	✗
automatic		

* Matrix storage
  - graphs
  - discrete systems

1) Dense matrix:
   * indexing (start at 0, or 1)
   * memory. layout (column-major or row-major)

$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Rightarrow$ float a[] = {1, 2, 3, 4};

2) Sparse matrix (an operator with a lot of zeros)

- coordinate format (coo)

  n (rows), m (columns),

  nnz (number of nonzero entries)

  col (columns indices)

row (row indices), v (numerical values)

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 4 & 5 & 0 & 7 \end{bmatrix}$$

$n = 4$
$m = 4$
$nnz = 6$

col = { 0, 0, 1, 1, 2, 3 };

row = { 1, 3, 1, 3, 0, 3 };

v = { 1, 4, 2, 5, 1, 7 };

* How to compute $y = Ax$?

$$y_i = \sum_{j=1}^{m} A_{ij} x_j$$

In the dense case, need to perform $n \times m$ operations

But here we can exploit the sparsity of A

single for loop:

```
for (int i = 0; i < n; ++i)
    y[i] = 0;
for (int i=0; i < nnz; ++i)
    y[ row[i] ] += v[i] · x[col[i]];
```

- compressed sparse row (CSR)
  n (rows), m (columns)
  row_displ (displacement of rows)
  col (column indices), v (numerical values)

row_displ is built recursively :

$\forall i \in [\![0; n-1]\!]$ row_displ[0] = 0 ;

row_displ[i+1] = row_displ[i]

+ number of nonzero of the $(i+1)^{th}$ row

assert (nnz == row_displ[n]);

$$A = \begin{bmatrix} 0 & 0 & 5 & 0 \\ 0 & 2 & 3 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$n = 4 \quad m = 4$

$nnz = 4$

$col = \{ 2, 1, 2, 3 \};$

$v = \{ 5, 2, 3, 1 \};$

$row\_displ = \{ 0, 1, 3, 4, 4 \};$

* How to compute $y = Ax$?

The number of operations is still nnz

Use a nested loop

```
for(int i=0; i<n; ++i){
  y[i] = 0;
  for(int j=row_disp[i]; j<row_disp[i+1]; ++j)
    y[i] += v[j] * x[col[j]];
}
```

Again, this is difficult to parallelize (or even make efficient) because of cache misses and non-contiguous memory-access

- diagonal format (DIA)
  n (rows), m (cols), v (numerical values)
  ndiag (number of diagonals),
  idiag (shifts w.r.t. the main diagonal)

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7 & 2 & 0 & 6 \\ 0 & 8 & 3 & 0 \\ 10 & 0 & 9 & 4 \end{bmatrix}$$

(two-dimensional array with padding)

$n = 4, \quad m = 4$

$ndiag = 4$

$idiag = \{-3, -1, 0, 2\}$

$$V = \begin{bmatrix} 0 & 0 & 0 & 10 \\ 0 & 7 & 8 & 9 \\ 1 & 2 & 3 & 4 \\ 0 & 6 & 0 & 0 \end{bmatrix}$$

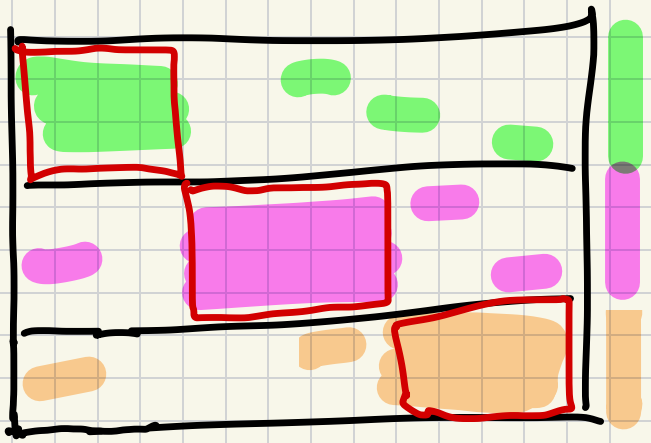
* How to compute $y = Ax$?

```
for (int i = 0; i < n; ++i)
    y[i] = 0.;
for (int k = 0; k < ndiag; ++k)
  for (int j = max(0, -idiag(k));
       j < min(n, n - idiag(k));
       ++j)
    y[j] += v[k][j] * x[j + idiag[k]];
```

Memory accesses in the inner-most loop are this time contiguous. vectorization, OpenMP should perform OK.

What about distributed-memory parallelism?

We stick to the CSR format



$A =$

we could split according to cols but this doesn't fit the CSR format ($\Rightarrow$ CSC)

$y = Ax$

there is an issue w.r.t. the distribution of x

If A is block-diagonal, we can distribute x like we distribute A, and there is no additional cost.

Solving linear systems of equations

$$Ax = f$$

* Gaussian elimination

- For dense systems, complexity of order n^3
- For sparse systems, complexity can be lowered but it's most often superlinear (n^2 or more)
- In sequential, it's straightforward to implement

there are some implementations for.

distributed - memory parallelism:

- MUMPS
- Pastix
- PARDISO
- SuperLU_DIST

* Iterative methods

- much cheaper than direct methods
- But may not converge fast enough

We make the assumption that A^{-1} is too expensive to compute.

Instead we want to derive an iterative scheme:

$$x = x^{(k)} + e^{(k)}$$

- x: exact solution
- $x^{(k)}$: approximate solution
- $e^{(k)}$: error
- k: at iteration #k

What does $e^{(k)}$ satisfy ?

$$Ax = Ax^{(k)} + Ae^{(k)}$$
$$= f \Rightarrow Ae^{(k)} = \boxed{f - Ax^{(k)}}$$

We introduce $r^{(k)} = f - Ax^{(k)} \Rightarrow r^{(k)} = Ae^{(k)}$

In the end, we have $x = x^{(k)} + A^{-1} r^{(k)}$

Problem : we cannot compute $A^{-1} r^{(k)}$

Instead, introduce an approximation of A^{-1}, that we will denote M^{-1}

M^{-1} is what we call a preconditioner

There are various ways of building either M or M^{-1}

Basic preconditioners:

* Jacobi: we pick M as the diagonal of A, then, it is much cheaper to compute M^{-1}

In practice, the previous formula

$x = x^{(k)} + A^{-1} r^{(k)}$ becomes

the following iterative scheme:

$x^{(0)}$ is given

$$x^{(k+1)} = x^{(k)} + M^{-1} r^{(k)}$$

Alternatively, we can write:

$$x^{(k+1)} = x^{(k)} + M^{-1}(f - Ax^{(k)})$$

$$= (I - M^{-1}A)x^{(k)} + M^{-1}f$$

What about Jacobi preconditioning?

$$A = L + D + U$$

$$x^{(k+1)} = (I - D^{-1}A)x^{(k)} + D^{-1}f$$

$$\Rightarrow \quad = (I - D^{-1}(L+U) - I)x^{(k)} + D^{-1}f$$

$$= -D^{-1}\left[(L+U)x^{(k)} - f\right]$$

We need to update all the rows of $x^{(k+1)}$:

$\forall i \in [\![1; n]\!]$ $\quad x_i^{(k+1)} = \frac{1}{A_{ii}} \left[\sum_{\substack{j=1 \\ j \neq i}}^{n} A_{ij} x_j^{(k)} - f_i \right]$

↑ no diagonal coefficient

(k+1) on the LHS, (k) on the RHS, so we can update coefficient simultaneously.

if $A_{ii} = 0 \Rightarrow \frac{1}{A_{ii}} = 1$

The convergence is not guaranteed but one can check $\|r^{(k)}\|$

At Gauss-Seidel method

$$M = (L+D)^{-1}$$

$$x^{(k+1)} = (I - M^{-1}A)x^{(k)} + M^{-1}f$$

$$= \left(I - (L+D)^{-1}(L+D+U)\right)x^{(k)} + (L+D)^{-1}f$$

$$= -(L+D)^{-1}\left[Ux^{(k)} - f\right]$$

$$(L+D)\,x^{(k+1)} = f - Ux^{(k)}$$

Let's pick a row number $i \in [\![1; n]\!]$

$$\left((L+D)x^{(k+1)}\right)_i = f_i - \left(Ux^{(k)}\right)_i$$

$$\sum_{j=1}^{i} A_{ij} x_j^{(k+1)} = f_i - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)}$$

$$\sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} + A_{ii} x_i^{(k+1)} = f_i - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)}$$

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left[f_i - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)} - \sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} \right]$$

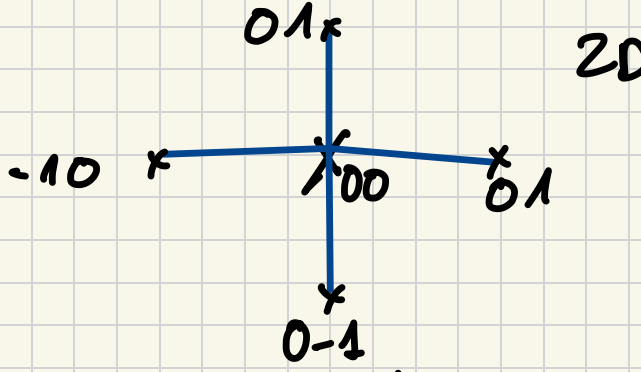
* Parallelizing the Gauss-Seidel method is much more difficult

With some assumptions, we can proceed as for the Jacobi method:

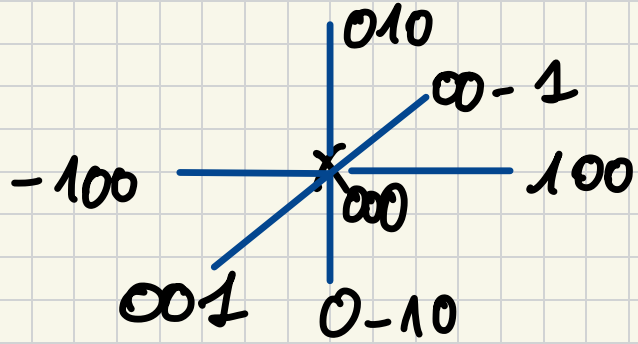
* diagonal A
* upper triangular A

We can still make the preconditioner parallel using graph coloring techniques

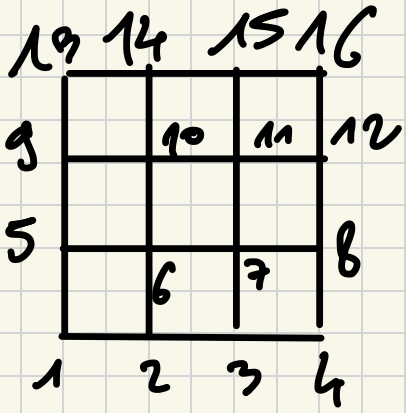
Let us assume that A comes from a stencil-based discretization:

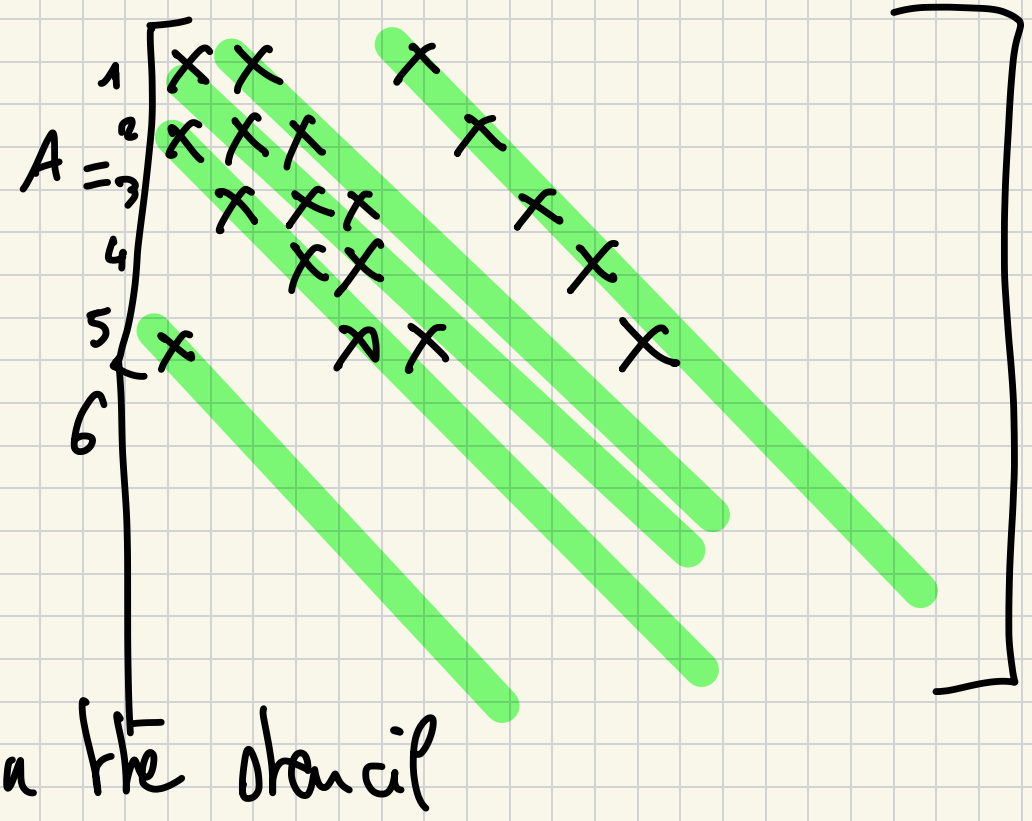


2D 5-point stencil



3D 7-point stencil



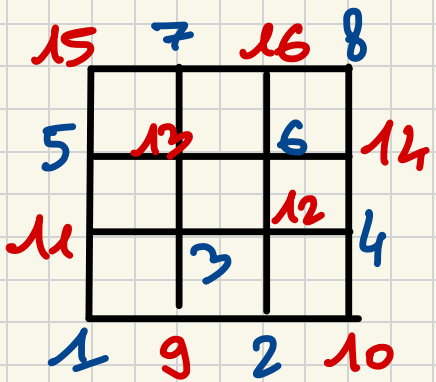


Same number of diagonals as the number of point in the stencil

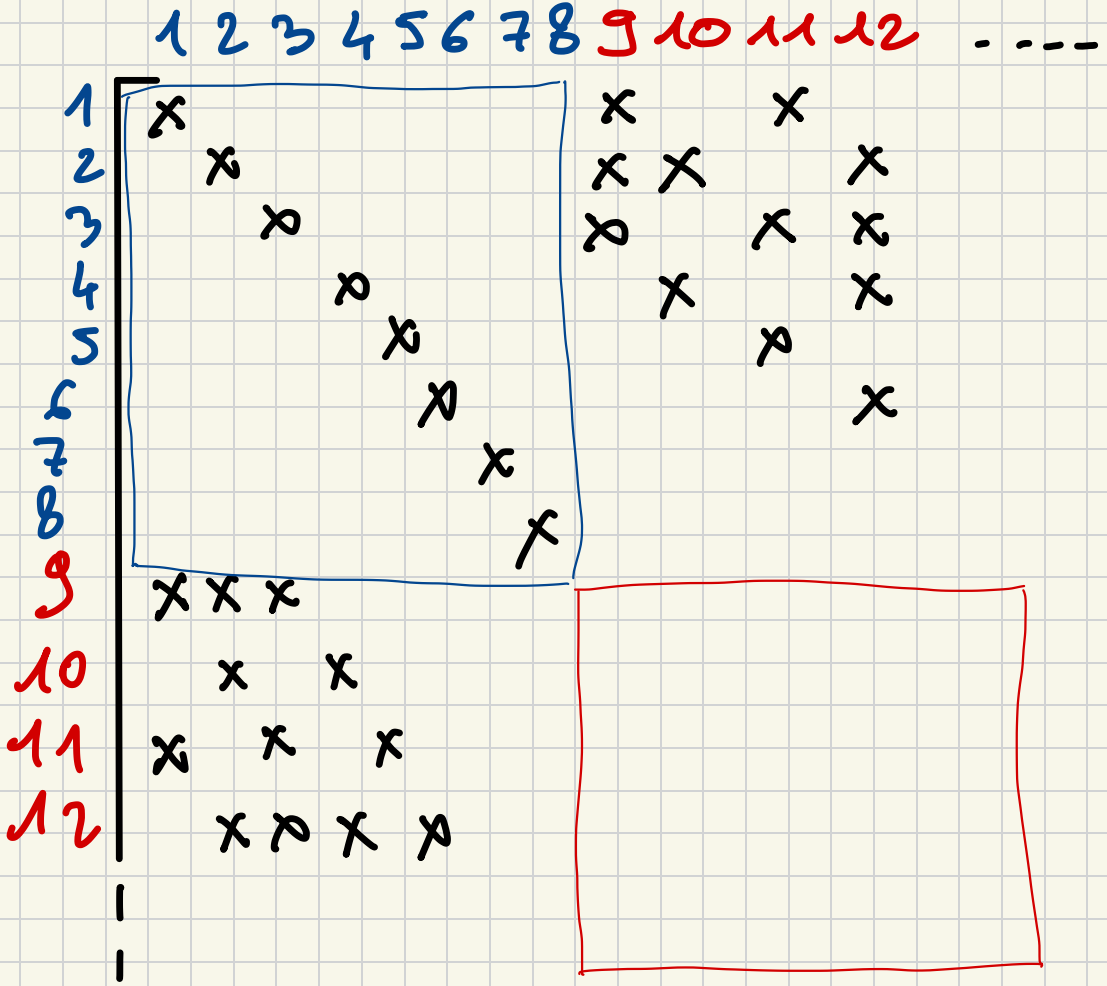
We need to renumber the unknowns to exhibit possible parallelism:

red-black renumbering

$\Rightarrow$ we want to avoid having two grid points in the same color (red or black) interacting through the stencil



thanks to the RB numbering diagonal blocks are diagonal



The update formula for Gauss-Seidel is

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left[f_i - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)} - \sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} \right]$$

* First we update all $x_i^{(k+1)}$ for i in blue:

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left[f_i - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)} \right]$$

This can be computed in parallel (just as for Jacobi)

* Then, we can update the red unknowns

$$x_i^{(k+1)} = \frac{1}{A_{ii}}\left[f_i - \sum_{j=1}^{i-1} A_{ij}\, x_j^{(k+1)} \right]$$

Keep in mind that the blue unknows were updated in the previous step

So again, we can do thing in parallel

* There are more advanced
  - iterative methods
  - preconditioners

For iterative methods, we most often use Krylov methods which are a specific kind of polynomial methods.

Remember the Cayley-Hamilton theorem

The characteristic polynomial of A

$$\chi(\lambda) = \det(\lambda I - A)$$

$$\chi(A) = 0$$

It is useful here because we can show that

$$\chi(X) = (-1)^{n+1} \det(A) + X q(X)$$

with $q \in \mathbb{K}_{n-1}[X]$

With the theorem, we get

$$\chi(A) = 0 = (-1)^{n+1} \det(A) I + A q(A)$$

$$A\, q(A) = (-1)^{n} \det(A)\, I$$

$$A \cdot \frac{q(A)}{(-1)^{n} \det(A)} = I \qquad \det A \neq 0$$

$$A^{-1} = \frac{q(A)}{(-1)^{n} \det(A)} \Rightarrow A^{-1} \text{ can be expressed as a polynom of } A$$