Equilibrage et régulation de charge

Option PRCD – 2018-2019

Mathieu Faverge

Structure du cours

- 4 séances avec M. Faverge (Algèbre linéaire dense, distribution)
- 4 séances avec G. Aupy (Ordonnancement (partie 1))
- 4 séances avec P. Ramet (Ordonnancement (partie 2))
- **Epreuve**: 1h20 avec droit aux documents, basé principalement sur l'ordonnancement et le placement des données, avec l'algèbre linéaire dense comme trame de fond.

Architectures

Loi de Moore

Architectures

Loi de Moore

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.

Architectures

Loi de Moore

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.

- Evolution des architectures. (IT336: D. Barthou)
  - Fréquence des processeurs
  - Multiplication des chips pour palier au problème de dissipation d'énergie
  - Accélérateurs: GPU, Phi, ...

Langages de programmation

See PG305, O. Aumage

- Shared memory:

Langages de programmation

See PG305, O. Aumage

- Shared memory: OpenMP, Pthread,
- Distributed:

Langages de programmation

See PG305, O. Aumage

- Shared memory: OpenMP, Pthread,
- Distributed: MPI, PVM, HPF, hash-programming-environment (HPE), ...
- Global Array, Gasnet, UPC
- Accélerateurs:

Langages de programmation

See PG305, O. Aumage

- Shared memory: OpenMP, Pthread,
- Distributed: MPI, PVM, HPF, hash-programming-environment (HPE), ...
- Global Array, Gasnet, UPC
- Accélerateurs: Cuda, OpenCL, ...

Langages de programmation

See PG305, O. Aumage

- Shared memory: OpenMP, Pthread,
- Distributed: MPI, PVM, HPF, hash-programming-environment (HPE), ...
- Global Array, Gasnet, UPC
- Accélerateurs: Cuda, OpenCL, ...
- + récent: Charm++, OpenHMPP, OpenACC, Cilk, Intel TBB, ...
- PaRSEC, StarPU, StarSS, ...

Qu'est ce qu'il faut en retenir?

- Evolution des codes pour s'adapter et pour résoudre les problèmes de stockage mémoire, débit, temps de calculs
- 2 problèmes dans le parallélisme: distribution des données, et ordonnancement des calculs

Comment évaluer la performance d'un code parallèle? Amdhal's law

Comment évaluer la performance d'un code parallèle? Amdhal's law

$$T_{parallel} = (1 - s) * T_{seq} + \frac{s * T_{seq}}{a} \quad (1)$$

avec s la portion du code qui est accélérée et a, le facteur d'accélération

Comment évaluer la performance d'un code parallèle? Amdhal's law

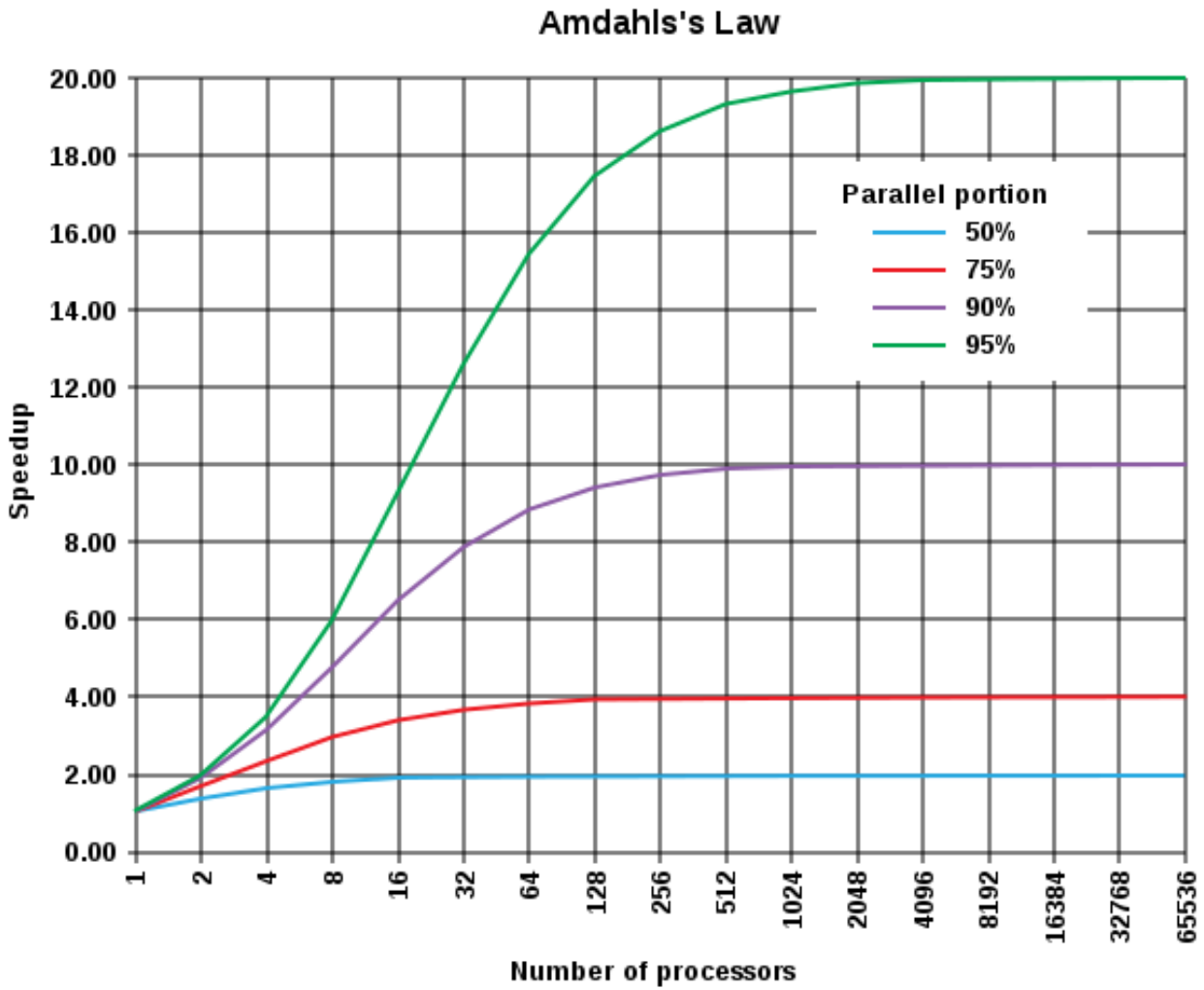
$$T_{parallel} = (1 - s) * T_{seq} + \frac{s * T_{seq}}{a} \quad (1)$$

avec s la portion du code qui est accélérée et a, le facteur d'accélération

$$Speedup = T_{seq} / T_{parallel} = \frac{1}{(1 - s) + \frac{s}{a}} \quad (2)$$

$$T_{parallel} = T_{communication} + T_{sequentielparallel} = \frac{1}{(1 - s) + \frac{s}{a}} \quad (3)$$

Comment évaluer la performance d'un code parallèle? Limits of Amdhal's law



Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce qu'un *x*flop/s?

Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce qu'un *x*flop/s?

- *x*flop/s représente une vitesse d'exécution, c'est le nombre d'opérations flottantes effectuées par seconde. *Quand ce terme est utilisé, il fait référence au nombre d'opérations sur des doubles (addition ou multiplication)*.

Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce qu'un *x*flop/s?

- *x*flop/s représente une vitesse d'exécution, c'est le nombre d'opérations flottantes effectuées par seconde. *Quand ce terme est utilisé, il fait référence au nombre d'opérations sur des doubles (addition ou multiplication)*.
- kilo 10^3, mega 10^6, giga 10^9,

Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce qu'un *x*flop/s?

- *x*flop/s représente une vitesse d'exécution, c'est le nombre d'opérations flottantes effectuées par seconde. *Quand ce terme est utilisé, il fait référence au nombre d'opérations sur des doubles (addition ou multiplication)*.
- kilo 10^3, mega 10^6, giga 10^9, tera 10^{12},

Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce qu'un *x*flop/s?

- *x*flop/s représente une vitesse d'exécution, c'est le nombre d'opérations flottantes effectuées par seconde. *Quand ce terme est utilisé, il fait référence au nombre d'opérations sur des doubles (addition ou multiplication)*.
- kilo 10^3, mega 10^6, giga 10^9, tera 10^{12}, peta 10^{15},

Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce qu'un *x*flop/s?

- *x*flop/s représente une vitesse d'exécution, c'est le nombre d'opérations flottantes effectuées par seconde. *Quand ce terme est utilisé, il fait référence au nombre d'opérations sur des doubles (addition ou multiplication)*.
- kilo 10^3, mega 10^6, giga 10^9, tera 10^{12}, peta 10^{15}, exa 10^{18},

Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce qu'un *x*flop/s?

- *x*flop/s représente une vitesse d'exécution, c'est le nombre d'opérations flottantes effectuées par seconde. *Quand ce terme est utilisé, il fait référence au nombre d'opérations sur des doubles (addition ou multiplication)*.
- kilo 10^3, mega 10^6, giga 10^9, tera 10^{12}, peta 10^{15}, exa 10^{18}, zetta 10^{21},

Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce qu'un *x*flop/s?

- *x*flop/s représente une vitesse d'exécution, c'est le nombre d'opérations flottantes effectuées par seconde. *Quand ce terme est utilisé, il fait référence au nombre d'opérations sur des doubles (addition ou multiplication)*.
- kilo 10^3, mega 10^6, giga 10^9, tera 10^{12}, peta 10^{15}, exa 10^{18}, zetta 10^{21}, yotta 10^{24}

Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce que le *theoretical peak*?

Comment évaluer l'efficacité ou la performance d'un code?

Qu'est ce que le *theoretical peak*?

- Le pic de performance théorique n'est pas calculé à partir de l'exécution d'un benchmark, mais calculé à partir des données constructeur de l'unité de calcul exploitée.
- $\text{FLOPS} = \text{cores} \times \text{clock} \times \frac{\text{FLOPs}}{\text{cycle}}$
- Exemple
  - Intel Xeon 5570 quad core à 2.93 GHz peut effectuer 4 opérations flottantes en double précision par cycle, soit une performance théorique de 11.72 GFlop/s CPU, ou 46.88 GFlop/s pour le socket.
  - Intel Xeon(R) CPU E5-2680 à 2.5GHz (AVX2 + FMA)
    16 opérations par cycle $\rightarrow 40 GFlop/s$
  - Intel Xeon Phi à 1.7GHz (AVX512 + FMA) 32 opérations par cycle
    $\rightarrow 41.6 GFlop/s$

1

Algèbre linéaire dense

1.1

Algèbre linéaire dense

Introduction

Quel est l'intérêt des bibliothèques d'algèbre linéaire dense? (1/2)

- La plupart des codes de simulations numériques résolvent le problème suivant:
  - $Ax = b$
  - A est une matrice de taille M par N
  - x et b sont deux vecteurs (ou ensembles de vecteurs) de taille N, et M, respectivement.
- But: fournir aux utilisateurs des librairies pour faire cette opération le plus efficacement possible:
  - le plus rapidement possible
  - stable et précis numériquement
- Il y a deux types majeurs de problèmes:
  - A est dense. Toutes les entrées sont considérées comme non nulles
  - A est creux. Il y a un grand pourcentage d'entrées nulles.

Quel est l'intérêt des bibliothèques d'algèbre linéaire dense? (2/2)

Résoudre $Ax = b$

Quel est l'intérêt des bibliothèques d'algèbre linéaire dense? (2/2)

Résoudre $Ax = b$

- Est-ce qu'on peut calculer l'inverse de A?

Quel est l'intérêt des bibliothèques d'algèbre linéaire dense? (2/2)

Résoudre $Ax = b$

- Est-ce qu'on peut calculer l'inverse de A?
- Quelles sont les deux principales classes d'algorithmes?

Quel est l'intérêt des bibliothèques d'algèbre linéaire dense? (2/2)

Résoudre $Ax = b$

- Est-ce qu'on peut calculer l'inverse de A?
- Quelles sont les deux principales classes d'algorithmes?
  - Directes ou itératives

Quel est l'intérêt des bibliothèques d'algèbre linéaire dense? (2/2)

Résoudre $Ax = b$

- Est-ce qu'on peut calculer l'inverse de A?
- Quelles sont les deux principales classes d'algorithmes?
  - Directes ou itératives
- Il existe plusieurs méthodes de factorisation:
  - Matrice générale: $A = LU$
  - Matrice symétrique définie positive: $A = LL^t$
  - Matrice symétrique non définie positive: $A = LDL^t$
  - Autre solution: $A = QR$

Étapes de résolutions pour les méthodes directes

1. Factoriser A avec une des méthodes précédentes

$$Ax = (LU)x = b$$

Étapes de résolutions pour les méthodes directes

1. Factoriser A avec une des méthodes précédentes

$$Ax = (LU)x = b$$

2. Première étape de résolution (descente)

$$L(Ux) = Ly = b$$

Étapes de résolutions pour les méthodes directes

1. Factoriser A avec une des méthodes précédentes

$$Ax = (LU)x = b$$

2. Première étape de résolution (descente)

$$L(Ux) = Ly = b$$

3. Deuxième étape de résolution (remontée)

$$Ux = y$$

Un peu d'histoire

- High Performance Computers:
  - IBM 370/195, CDC 7600, Univac 1110, DEC PDP-10, Honeywell 6030
- Fortran 66
- Fournir des logiciels portables sur différentes architectures
- Atteindre une certaine efficacité
- BLAS (Level 1)
  - Opérations sur les vecteurs
- Release de la première version en 1979
  - Correspond au temps du Cray 1

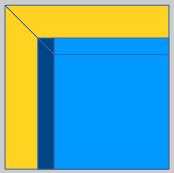
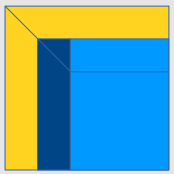
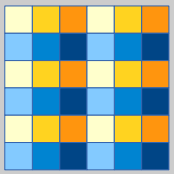
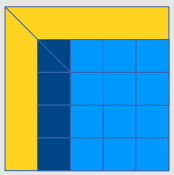
Un peu d'histoire

- Opérations vectorielles insuffisantes!!!
  - Considérons l'opération AXPY ($y = \alpha x + y$): $2n$ flops pour $3n$ lectures/écritures
  - Intensité de calcul $= (2n)/(3n) = 2/3$
  - Encore trop bas pour obtenir de bonnes performances
- Début des BLAS-2 (1984-1986)
  - Implémentation standard de 25 nouvelles operations opérants sur un couple matrice-vecteur
    - GEMV: $y = \alpha Ax + \beta y$, GER: $A = A + \alpha x * y^t$,
    - Début du support des 4 précisions: (S/D/C/Z), 66 routines, 18K LOC
  - Pourquoi les BLAS 2?
    - $O(n^2)$ operations pour $O(n^2)$ données
      $\rightarrow$ Meilleure intensité de calcul $(2n^2)/(n^2) = 2$
    - OK pour les machines vectorielles, mais insuffisant pour les machines à base de caches

D'où l'arrivée des BLAS-3

BLAS	Memory Refs	Flops	Ratio
Lvl 1 $y = y + \alpha x$	$3n$	$2n$	$2/3$
Lvl 2 $y = y + Ax$	n^2	$2n^2$	2
Lvl 3 $C = C + AB$	$4n^2$	$2n^3$	$n/2$

Évolution des bibliothèques d'algèbre linéaire dense

	**LINPACK** (70's) vector operations	• Level 1 BLAS
	**LAPACK** (80's) block operations	• Level 3 BLAS
	**ScaLAPACK** (90's) block cyclic data distribution	• PBLAS • BLACS (message passing)
	**PLASMA** (00's) tile operations	• tile layout • dataflow scheduling

Bibliothèques d'algèbre linéaire dense modernes

Repose sur deux piliers:

- Algorithmes à base de tuiles
- Support d'exécution

PLASMA-OpenMP Nouvelle version de Plasma basée sur OpenMP
`https://bitbucket.org/icl/plasma`

DPLASMA Version distribuée sur PaRSEC
`https://bitbucket.org/icldistcomp/parsec`

Chameleon Version distribuée sur supports d'exécutions multiples (StarPU, PaRSEC, Quark, ...)
`https://gitlab.inria.fr/solverstack/chameleon`

Slate Version MPI/OpenMP
`https://bitbucket.org/icl/slate`

2

Résolution de système triangulaire

2.1

Résolution de système triangulaire

Version scalaire

Résoudre $Lx = b$

$$\begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & & \vdots \\ \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

On en déduit l'égalité:

$$\sum_{k=1}^{i-1} l_{ik} x_k + l_{ii} x_i = b_i$$

Résoudre $Lx = b$

Version scalaire, example avec $N = 4$

$$\begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

Résoudre $Lx = b$

Version scalaire, example avec $N = 4$

$$\begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

$$b_1 = l_{11} x_1 \quad (1)$$

$$b_2 = l_{21} x_1 + l_{22} x_2 \quad (2)$$

$$b_3 = l_{31} x_1 + l_{32} x_2 + l_{33} x_3 \quad (3)$$

$$b_4 = l_{41} x_1 + l_{42} x_2 + l_{43} x_3 + l_{44} x_4 \quad (4)$$

Résoudre $Lx = b$

Version scalaire, example avec $N = 4$

$$\begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

$$\begin{aligned} b_1 &= l_{11} x_1 && (1) \\ b_2 &= l_{21} x_1 + l_{22} x_2 && (2) \\ b_3 &= l_{31} x_1 + l_{32} x_2 + l_{33} x_3 && (3) \\ b_4 &= l_{41} x_1 + l_{42} x_2 + l_{43} x_3 + l_{44} x_4 && (4) \end{aligned}$$

$$\begin{aligned} x_1 &= b_1 / l_{11} && (1) \\ x_2 &= (b_2 - l_{21} x_1) / l_{22} && (2) \\ x_3 &= (b_3 - l_{31} x_1 - l_{32} x_2) / l_{33} && (3) \\ x_4 &= (b_4 - l_{41} x_1 - l_{42} x_2 - l_{43} x_3) / l_{44} && (4) \end{aligned}$$

Résoudre $Lx = b$

Algorithm 1: Algorithme ligne

```
1:  for i = 1 to n do
2:
3:      for j = 1 to i − 1 do
4:
5:          b_i = b_i − l_ij x_j
6:
7:      end for
8:
9:      x_i = b_i / l_ii
10:
11: end for
```

$$\begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix}$$

$$\begin{aligned} x_1 &= b_1 / l_{11} \\ x_2 &= (b_2 - l_{21} x_1) / l_{22} \\ x_3 &= (b_3 - l_{31} x_1 - l_{32} x_2) / l_{33} \\ x_4 &= (b_4 - l_{41} x_1 - l_{42} x_2 - l_{43} x_3) / l_{44} \end{aligned}$$

Résoudre $Lx = b$

Algorithm 2: Algorithme colonne

```
1:  for j = 1 to n do
2:
3:      x_j = b_j / l_jj
4:
5:      for i = j + 1 to n do
6:
7:          b_i = b_i − l_ij x_j
8:
9:      end for
10:
11: end for
```

$$\begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix}$$

$$\begin{aligned} x_1 &= b_1 / l_{11} \\ x_2 &= (b_2 - l_{21} x_1) / l_{22} \\ x_3 &= (b_3 - l_{31} x_1 - l_{32} x_2) / l_{33} \\ x_4 &= (b_4 - l_{41} x_1 - l_{42} x_2 - l_{43} x_3) / l_{44} \end{aligned}$$

Résoudre $Lx = b$: Complexité

Algorithm 3: Algorithme ligne

1: **for** $i = 1$ to n **do**
2: **for** $j = 1$ to $i - 1$ **do**
3: $b_i = b_i - l_{ij} x_j$
4: **end for**
5: $x_i = b_i / l_{ii}$
6: **end for**

Résoudre $Lx = b$: Complexité

Algorithm 4: Algorithme ligne

```
1: for i = 1 to n do
2:     for j = 1 to i − 1 do
3:         b_i = b_i − l_ij x_j
4:     end for
5:     x_i = b_i / l_ii
6: end for
```

$$\sum_{k=1}^{n}(1 + 2 * k) \equiv n^2 + o(n)$$

2.2

Résolution de système triangulaire

Version par bloc

Résoudre $Lx = b$

$$\begin{pmatrix} L_{11} & 0 & \dots & 0 \\ L_{21} & L_{22} & & \vdots \\ \vdots & \vdots & \ddots & 0 \\ L_{n1} & L_{n2} & \dots & L_{nn} \end{pmatrix} * \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix}$$

On en déduit l'égalité:

$$\sum_{k=1}^{i-1} L_{ik} X_k + L_{ii} X_i = B_i$$

Résoudre $Lx = b$

Version scalaire, example avec $N = 4$

$$\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix} * \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{pmatrix}$$

Résoudre $Lx = b$

Version scalaire, example avec $N = 4$

$$\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix} * \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{pmatrix}$$

$$B_1 = L_{11}X_1 \quad (1)$$

$$B_2 = L_{21}X_1 + L_{22}X_2 \quad (2)$$

$$B_3 = L_{31}X_1 + L_{32}X_2 + L_{33}X_3 \quad (3)$$

$$B_4 = L_{41}X_1 + L_{42}X_2 + L_{43}X_3 + L_{44}X_4 \quad (4)$$

Résoudre $Lx = b$

Version scalaire, example avec $N = 4$

$$\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix} * \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{pmatrix}$$

$$\begin{aligned} B_1 &= L_{11}X_1 && (1) \\ B_2 &= L_{21}X_1 + L_{22}X_2 && (2) \\ B_3 &= L_{31}X_1 + L_{32}X_2 + L_{33}X_3 && (3) \\ B_4 &= L_{41}X_1 + L_{42}X_2 + L_{43}X_3 + L_{44}X_4 && (4) \end{aligned}$$

$$\begin{aligned} X_1 &= L_{11}^{-1}B_1 && (1) \\ X_2 &= L_{22}^{-1}(B_2 - L_{21}X_1) && (2) \\ X_3 &= L_{33}^{-1}(B_3 - L_{31}X_1 - L_{32}X_2) && (3) \\ X_4 &= L_{44}^{-1}(B_4 - L_{41}X_1 - L_{42}X_2 - L_{43}X_3) && (4) \end{aligned}$$

Résoudre $Lx = b$

Algorithm 5: Algorithme ligne

1: **for** $i = 1$ to n **do**
2: **for** $j = 1$ to $i - 1$ **do**
3: $B_i = B_i - L_{ij} X_j$
4: **end for**
5: Solve $L_{ii} X_i = B_i$
6: **end for**

$$\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix}$$

$$\begin{aligned} X_1 &= L_{11}^{-1} B_1 \\ X_2 &= L_{22}^{-1} (B_2 - L_{21} X_1) \\ X_3 &= L_{33}^{-1} (B_3 - L_{31} X_1 - L_{32} X_2) \\ X_4 &= L_{44}^{-1} (B_4 - L_{41} X_1 - L_{42} X_2 - L_{43} X_3) \end{aligned}$$

Résoudre $Lx = b$

Algorithm 6: Algorithme ligne

1: **for** $i = 1$ to n **do**
2: **for** $j = 1$ to $i - 1$ **do**
3: $B_i = B_i - L_{ij} X_j$
4: **end for**
5: Solve $L_{ii} X_i = B_i$ (TRSM)
6: **end for**

$$\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix}$$

$$\begin{aligned} X_1 &= L_{11}^{-1} B_1 \\ X_2 &= L_{22}^{-1} (B_2 - L_{21} X_1) \\ X_3 &= L_{33}^{-1} (B_3 - L_{31} X_1 - L_{32} X_2) \\ X_4 &= L_{44}^{-1} (B_4 - L_{41} X_1 - L_{42} X_2 - L_{43} X_3) \end{aligned}$$

Résoudre $Lx = b$

Algorithm 7: Algorithme ligne

1: **for** $i = 1$ to n **do**
2: **for** $j = 1$ to $i - 1$ **do**
3: $B_i = B_i - L_{ij} X_j$ (GEMM)
4: **end for**
5: Solve $L_{ii} X_i = B_i$ (TRSM)
6: **end for**

$$\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix}$$

$$\begin{aligned} X_1 &= L_{11}^{-1} B_1 \\ X_2 &= L_{22}^{-1} (B_2 - L_{21} X_1) \\ X_3 &= L_{33}^{-1} (B_3 - L_{31} X_1 - L_{32} X_2) \\ X_4 &= L_{44}^{-1} (B_4 - L_{41} X_1 - L_{42} X_2 - L_{43} X_3) \end{aligned}$$

Résoudre $Lx = b$

Algorithm 8: Algorithme colonne

```
1: for j = 1 to n do
2:     Solve L_jj X_j = B_j
3:     for i = j + 1 to n do
4:         B_i = B_i − L_ij X_j
5:     end for
6: end for
```

$$\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix}$$

$$\begin{aligned} X_1 &= L_{11}^{-1} B_1 \\ X_2 &= L_{22}^{-1} (B_2 - L_{21} X_1) \\ X_3 &= L_{33}^{-1} (B_3 - L_{31} X_1 - L_{32} X_2) \\ X_4 &= L_{44}^{-1} (B_4 - L_{41} X_1 - L_{42} X_2 - L_{43} X_3) \end{aligned}$$

Résoudre $Lx = b$

Algorithm 9: Algorithme colonne

1: **for** $j = 1$ to n **do**
2: Solve $L_{jj}X_j = B_j$ (TRSM)
3: **for** $i = j + 1$ to n **do**
4: $B_i = B_i - L_{ij}X_j$ (GEMM)
5: **end for**
6: **end for**

$$\begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix}$$

$$\begin{aligned} X_1 &= L_{11}^{-1} B_1 \\ X_2 &= L_{22}^{-1}(B_2 - L_{21}X_1) \\ X_3 &= L_{33}^{-1}(B_3 - L_{31}X_1 - L_{32}X_2) \\ X_4 &= L_{44}^{-1}(B_4 - L_{41}X_1 - L_{42}X_2 - L_{43}X_3) \end{aligned}$$

2.3

Résolution de système triangulaire

Mémoire distribuée

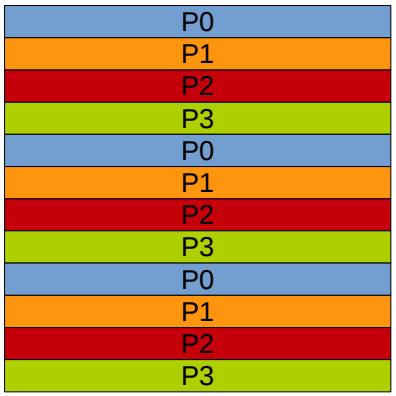
Définitions

- np processeurs
- $dloc = Bcast(r, d)$ est le Broadcast de la donnée d par la source r vers la destination $dloc$ sur chaque noeud
- $dloc = Reduce(r, d)$ est la réduction de la donnée d sur la source r et le résultat est stockée dans $dloc$ sur r
- $dloc = Scatter(r, d)$ est la diffusion de la donnée d par la source r vers la donnée locale $dloc$

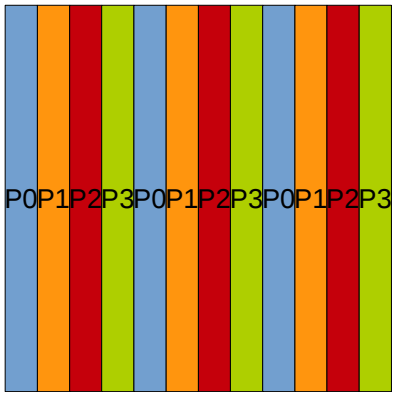
Nécessite de choisir une distribution des données

Nécessite de choisir une distribution des données

Cyclic en lignes



Cyclic en colonnes



Chaque processeur exécute le même algorithme avec son propre rang $r \in [1, np]$

$Lx = b$, Algorithme ligne

Algorithm 10: Row cyclic

```
1: for i = 1 to n do
2:     if r == i%np then
3:         for j = 1 to i − 1 do
4:             b_i = b_i − l_ij xloc_j
5:         end for
6:         x_i = b_i / l_ii
7:     end if
8:     xloc_i = Bcast( i%np, x_i )
9: end for
```

Algorithm 11: Column cyclic

```
1: for i = 1 to n do
2:     sumloc = 0
3:     for j = 1 to i − 1 do
4:         if r == j%np then
5:             sumloc = sumloc − l_ij xj
6:         end if
7:     end for
8:     sum = Reduce(i%np, sumloc)
9:     if r == i%np then
10:        b_i = b_i − sum
11:        x_i = b_i / l_ii
12:    end if
13: end for
```

$Lx = b$, Algorithme ligne

Distribution row-cyclic

- **PAS de parallélisme**
- Besoin de stocker une copie de x sur chaque noeud
- Opération AXPY

Distribution column-cyclic

- Mise à jour **parallèle**
- Possibilité d'utiliser DOT pour la mise à jour
- Échange d'une seule valeur à chaque étape

$Lx = b$, Algorithme colonne

Algorithm 12: Row cyclic

```
1:  for j = 1 to n do
2:      if r == j%np then
3:          x_j = b_j / l_jj
4:      end if
5:      xtmp = Bcast( j%np, x_j )
6:      for i = j + 1 to n do
7:          if r == i%np then
8:              b_i = b_i − l_ij xtmp
9:          end if
10:     end for
11: end for
```

Algorithm 13: Column cyclic

```
1:  for j = 1 to n do
2:      if r == j%np then
3:          x_j = b_j / l_jj
4:          for i = j + 1 to n do
5:              xtmp_i = l_ij x_j
6:          end for
7:      end if
8:      Scatter( j%np, xtmp )
9:      for i = j + 1 to n do
10:         if r == i%np then
11:             b_i = b_i − xtmp_i
12:         end if
13:     end for
14: end for
```

$Lx = b$, Algorithme colonne

Distribution row-cyclic

- Mise à jour **parallèle**
- Mise à jour par AXPY

Distribution column-cyclic

- **PAS de parallélisme** dans la première phase
- Nécessite de stocker une copie de x sur chaque noeud
- 2 boucles sur i
- Utilisation potentielle d'AXPY

Équilibrage de charge

Considérons une matrice de taille $N = n \times p$, avec l'algorithme par colonne sur une distribution *row-cyclic*, et r le rang de chaque processeur.

- Quel est le nombre d'opérations effectuées par le processeur de rang $r \in [1, p]$?

Équilibrage de charge

Considérons une matrice de taille $N = n \times p$, avec l'algorithme par colonne sur une distribution *row-cyclic*, et r le rang de chaque processeur.

- Quel est le nombre d'opérations effectuées par le processeur de rang $r \in [1, p]$?

$$nbops(r) = p * \sum_{0}^{n-1} k + (r-1) * n$$

Équilibrage de charge

Considérons une matrice de taille $N = n \times p$, avec l'algorithme par colonne sur une distribution *row-cyclic*, et r le rang de chaque processeur.

- Quel est le nombre d'opérations effectuées par le processeur de rang $r \in [1, p]$?

$$nbops(r) = p * \sum_{0}^{n-1} k + (r-1) * n$$

  - Quelle est la différence entre les processeurs de rang 1 et p?

Équilibrage de charge

Considérons une matrice de taille $N = n \times p$, avec l'algorithme par colonne sur une distribution *row-cyclic*, et r le rang de chaque processeur.

- Quel est le nombre d'opérations effectuées par le processeur de rang $r \in [1, p]$?

$$nbops(r) = p * \sum_{0}^{n-1} k + (r-1) * n$$

- Quelle est la différence entre les processeurs de rang 1 et p?

$$nbops(p) - nbops(1) = (p-1) * n$$

Équilibrage de charge

Considérons une matrice de taille $N = n \times p$, avec l'algorithme par colonne sur une distribution *row-cyclic*, et r le rang de chaque processeur.

- Quel est le nombre d'opérations effectuées par le processeur de rang $r \in [1, p]$?

$$nbops(r) = p * \sum_{0}^{n-1} k + (r-1) * n$$

  - Quelle est la différence entre les processeurs de rang 1 et p?

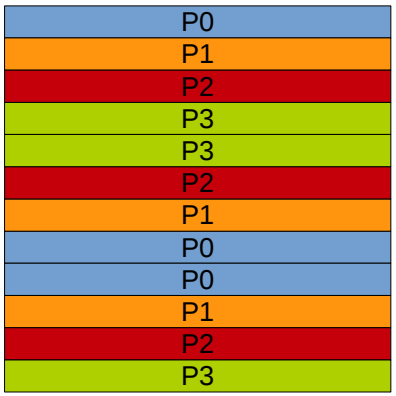
$$nbops(p) - nbops(1) = (p-1) * n$$

Comment réduire ce problème d'équilibrage?

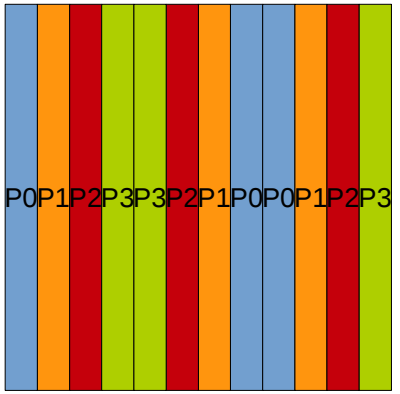
Comment réduire ce problème d'équilibrage?

Comment réduire ce problème d'équilibrage?

Ligne



Colonne



2.4

Résolution de système triangulaire

Ordonnancement de l'algorithme par blocs

Ordonnancement

Considérons une matrice de taille $N = NT \times b$, où b est la taille de chaque bloc, et NT le nombre de blocs, et une architecture à mémoire partagée de p coeurs.

Ordonnancement

Considérons une matrice de taille $N = NT \times b$, où b est la taille de chaque bloc, et NT le nombre de blocs, et une architecture à mémoire partagée de p coeurs.

Proposez un ordonnancement pour $NT = 4$ et $P = 2$ de l'algorithme $LX = B$.
Rq: En mémoire partagée par blocs, les deux algorithmes par colonnes et par lignes sont identiques.

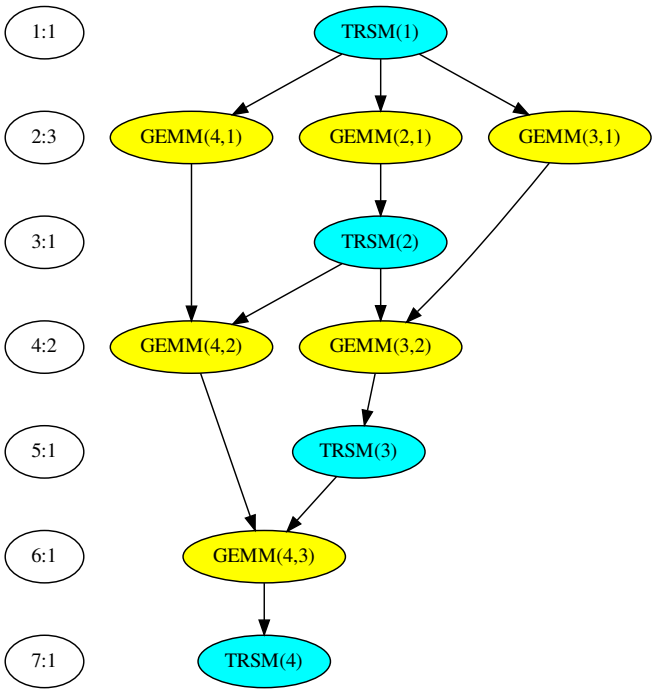
Rappel de l'algorithme par bloc

Algorithm 14: Algorithme ligne

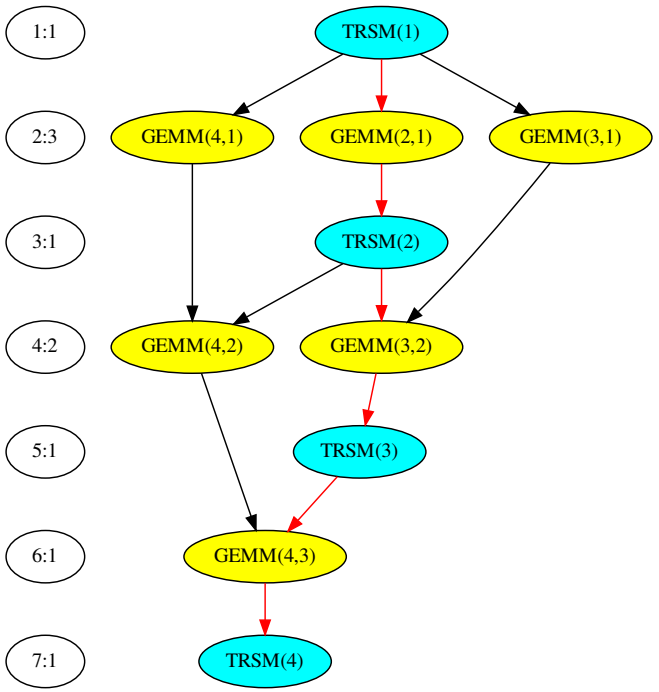
1: **for** $i = 1$ to n **do**
2: **for** $j = 1$ to $i - 1$ **do**
3: $B_i = B_i - L_{ij} X_j$ GEMM(i, j)
4: **end for**
5: Solve $L_{ii} X_i = B_i$ TRSM(i)
6: **end for**

Ordonnancement et DAG

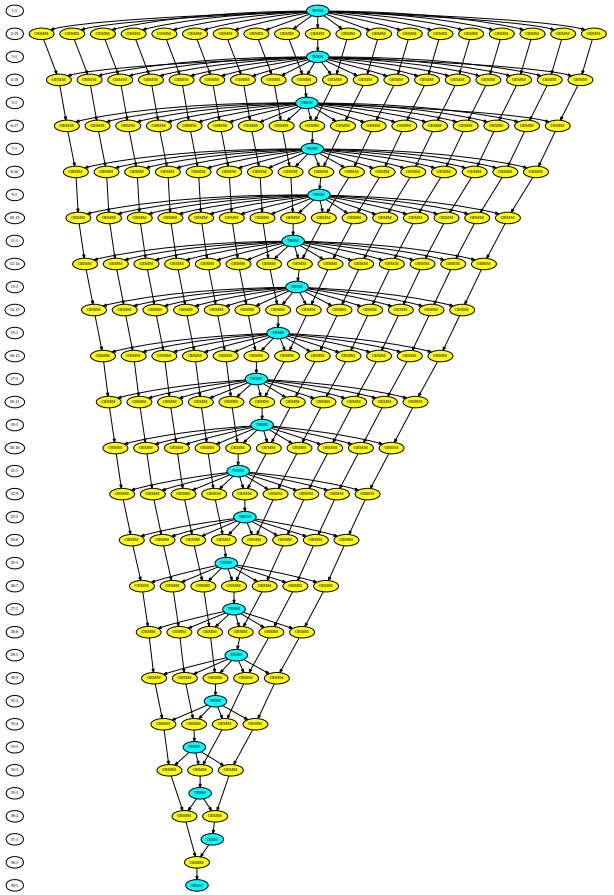
Ordonnancement et DAG



Ordonnancement et DAG



Exemple de DAG avec une matrice plus grande



3

Factorisation LU

3.1

Factorisation LU

Algorithme scalaire

Factorisation $A = LU$

Factorisons une matrice générale A sous la forme LU, tel que L soit triangulaire inférieure avec une diagonale unitaire, et U triangulaire supérieure.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{pmatrix} * \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix}$$

Factorisation $A = LU$

Factorisons une matrice générale A sous la forme LU, tel que L soit triangulaire inférieure avec une diagonale unitaire, et U triangulaire supérieure.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{pmatrix} * \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix}$$

$$a_{ij} = \sum_{k=1}^{min(i,j)} l_{ik} * u_{kj}$$

Algorithme scalaire

Algorithm 15: Factorisation LU

1: **for** $k = 1$ to n **do**
2: $\quad u_{k*} = a_{k*}$
3: $\quad$ **for** $i = k + 1$ to n **do**
4: $\quad\quad l_{ik} = a_{ik} / a_{kk}$
5: $\quad\quad$ **for** $j = k + 1$ to n **do**
6: $\quad\quad\quad a_{ij} = a_{ij} - l_{ik} * u_{kj}$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end for**

Algorithme scalaire

Algorithm 16: Factorisation LU

1: **for** $k = 1$ to n **do**
2: $\quad u_{k*} = a_{k*}$
3: $\quad$ **for** $i = k + 1$ to n **do**
4: $\quad\quad l_{ik} = a_{ik} / a_{kk}$
5: $\quad\quad$ **for** $j = k + 1$ to n **do**
6: $\quad\quad\quad a_{ij} = a_{ij} - l_{ik} * u_{kj}$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end for**

Quel est le problème numérique de cet algorithme?

Algorithme scalaire

Algorithm 17: Factorisation LU

```
1: for k = 1 to n do
2:     u_{k*} = a_{k*}
3:     for i = k + 1 to n do
4:         l_{ik} = a_{ik} / a_{kk}
5:         for j = k + 1 to n do
6:             a_{ij} = a_{ij} − l_{ik} ∗ u_{kj}
7:         end for
8:     end for
9: end for
```

Quel est le problème numérique de cet algorithme?

a_{kk} **doit être différent de** 0

Algorithme scalaire: complexité

Algorithm 18: Factorisation LU

1: **for** $k = 1$ to n **do**
2: $\quad u_{k*} = a_{k*}$
3: $\quad$ **for** $i = k + 1$ to n **do**
4: $\quad\quad l_{ik} = a_{ik} / a_{kk}$
5: $\quad\quad$ **for** $j = k + 1$ to n **do**
6: $\quad\quad\quad a_{ij} = a_{ij} - l_{ik} * u_{kj}$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end for**

Algorithme scalaire: complexité

Algorithm 19: Factorisation LU

```
1: for k = 1 to n do
2:     u_{k*} = a_{k*}
3:     for i = k + 1 to n do
4:         l_{ik} = a_{ik} / a_{kk}
5:         for j = k + 1 to n do
6:             a_{ij} = a_{ij} − l_{ik} ∗ u_{kj}
7:         end for
8:     end for
9: end for
```

$$\sum_{k=1}^{n}((n-k)+2*(n-k)^2 \equiv 2/3n^3 + o(n^2)$$

3.2

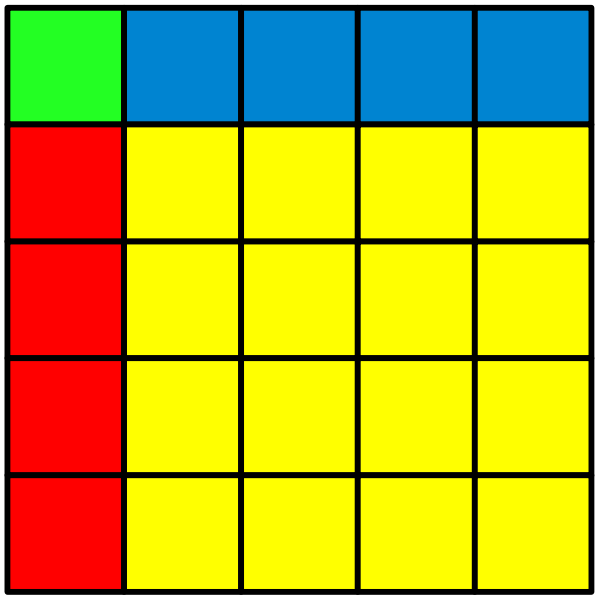
Factorisation LU

Block algorithm

Algorithme block

Algorithm 20: Factorisation LU

1: **for** $k = 1$ to n **do**
2: Factorize $A_{kk} = L_{kk} U_{kk}$
3: **for** $i = k + 1$ to n **do**
4: Solve $A_{ik} = L_{ik} * U_{kk}$
5: **end for**
6: **for** $j = k + 1$ to n **do**
7: Solve $A_{kj} = L_{kk} * U_{kj}$
8: **end for**
9: **for** $i = k + 1$ to n **do**
10: **for** $j = k + 1$ to n **do**
11: $A_{ij} = A_{ij} - L_{ik} * U_{kj}$
12: **end for**
13: **end for**
14: **end for**



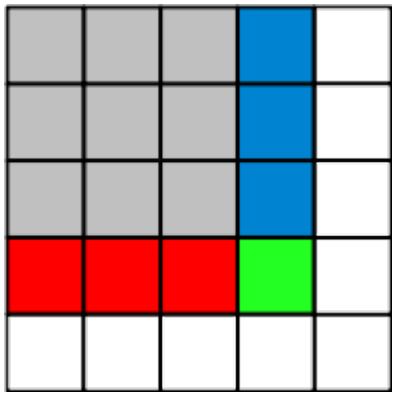
Première version: *Left-looking*

Considérons une matrice A, générale, de taille $n = NT \times b$, et tel que les i premières lignes et colonnes soient déjà factorisées $A_i = L_i U_i$ ou $A_{11}^{(i+1)} = L_{11}^{(i+1)} U_{11}^{(i+1)}$.
On veut ajouter b lignes/colonnes à la matrice factorisée tel qu'après NT étapes, l'intégarlité de l matrice A soit factorisée. $A_{11}^{(i+1)}$ est de taille $(i * b) \times (i * b)$, et $A_{22}^{(i+1)}$ est de taille $b \times b$.

$$\begin{pmatrix} A_{11}^{(i+1)} & A_{12}^{(i+1)} \\ A_{21}^{(i+1)} & A_{22}^{(i+1)} \end{pmatrix} = \begin{pmatrix} L_{11}^{(i+1)} & 0 \\ \mathbf{L_{21}^{(i+1)}} & \mathbf{L_{22}^{(i+1)}} \end{pmatrix} * \begin{pmatrix} U_{11}^{(i+1)} & \mathbf{U_{12}^{(i+1)}} \\ 0 & \mathbf{U_{22}^{(i+1)}} \end{pmatrix}$$

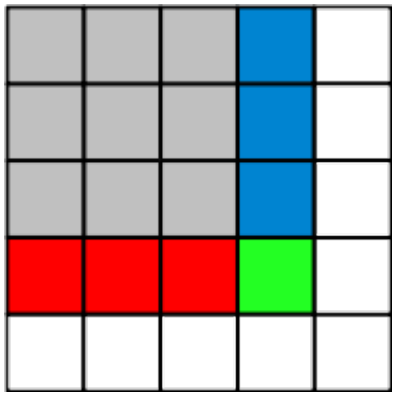
Algorithme *Left looking* (étapes)

- $A_{11}^{(i+1)} = L_{11}^{(i+1)} U_{11}^{(i+1)}$
- $A_{21}^{(i+1)} = L_{21}^{(i+1)} U_{11}^{(i+1)}$
- $A_{12}^{(i+1)} = L_{11}^{(i+1)} U_{12}^{(i+1)}$
- $A_{22}^{(i+1)} = L_{21}^{(i+1)} U_{12}^{(i+1)} + L_{22}^{(i+1)} U_{22}^{(i+1)}$



Algorithme *Left looking* (étapes)

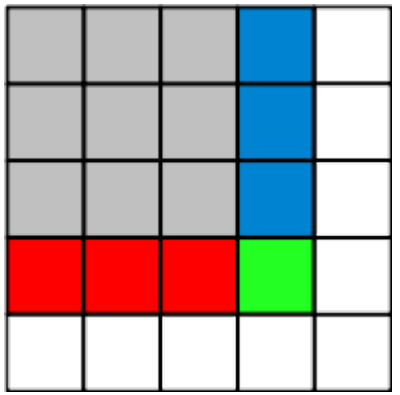
- $A_{11}^{(i+1)} = L_{11}^{(i+1)} U_{11}^{(i+1)}$
- $A_{21}^{(i+1)} = L_{21}^{(i+1)} U_{11}^{(i+1)}$
- $A_{12}^{(i+1)} = L_{11}^{(i+1)} U_{12}^{(i+1)}$
- $A_{22}^{(i+1)} = L_{21}^{(i+1)} U_{12}^{(i+1)} + L_{22}^{(i+1)} U_{22}^{(i+1)}$



1. Résoudre $A_{21}^{(i+1)} = L_{21}^{(i+1)} U_{11}^{(i+1)}$

Algorithme *Left looking* (étapes)

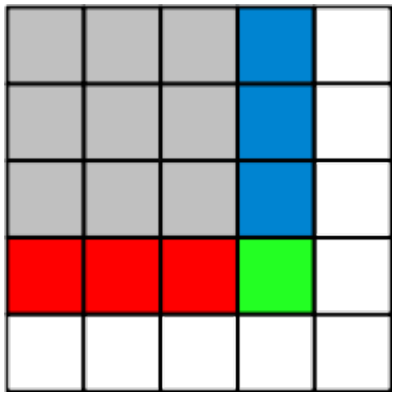
- $A_{11}^{(i+1)} = L_{11}^{(i+1)} U_{11}^{(i+1)}$
- $A_{21}^{(i+1)} = L_{21}^{(i+1)} U_{11}^{(i+1)}$
- $A_{12}^{(i+1)} = L_{11}^{(i+1)} U_{12}^{(i+1)}$
- $A_{22}^{(i+1)} = L_{21}^{(i+1)} U_{12}^{(i+1)} + L_{22}^{(i+1)} U_{22}^{(i+1)}$



1. Résoudre $A_{21}^{(i+1)} = L_{21}^{(i+1)} U_{11}^{(i+1)}$
2. Résoudre $A_{12}^{(i+1)} = L_{11}^{(i+1)} U_{12}^{(i+1)}$

Algorithme *Left looking* (étapes)

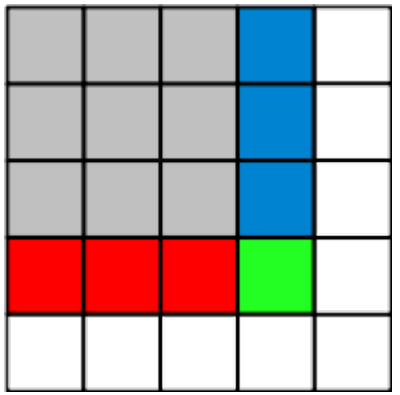
- $A_{11}^{(i+1)} = L_{11}^{(i+1)} U_{11}^{(i+1)}$
- $A_{21}^{(i+1)} = L_{21}^{(i+1)} U_{11}^{(i+1)}$
- $A_{12}^{(i+1)} = L_{11}^{(i+1)} U_{12}^{(i+1)}$
- $A_{22}^{(i+1)} = L_{21}^{(i+1)} U_{12}^{(i+1)} + L_{22}^{(i+1)} U_{22}^{(i+1)}$



1. Résoudre $A_{21}^{(i+1)} = L_{21}^{(i+1)} U_{11}^{(i+1)}$
2. Résoudre $A_{12}^{(i+1)} = L_{11}^{(i+1)} U_{12}^{(i+1)}$
3. Calculer $A_{22}^{(i+1)} -= L_{21}^{(i+1)} U_{12}^{(i+1)}$

Algorithme *Left looking* (étapes)

- $A_{11}^{(i+1)} = L_{11}^{(i+1)} U_{11}^{(i+1)}$
- $A_{21}^{(i+1)} = L_{21}^{(i+1)} U_{11}^{(i+1)}$
- $A_{12}^{(i+1)} = L_{11}^{(i+1)} U_{12}^{(i+1)}$
- $A_{22}^{(i+1)} = L_{21}^{(i+1)} U_{12}^{(i+1)} + L_{22}^{(i+1)} U_{22}^{(i+1)}$



1. Résoudre $A_{21}^{(i+1)} = L_{21}^{(i+1)} U_{11}^{(i+1)}$
2. Résoudre $A_{12}^{(i+1)} = L_{11}^{(i+1)} U_{12}^{(i+1)}$
3. Calculer $A_{22}^{(i+1)} - = L_{21}^{(i+1)} U_{12}^{(i+1)}$
4. Factoriser $A_{22}^{(i+1)} = L_{22}^{(i+1)} U_{22}^{(i+1)}$

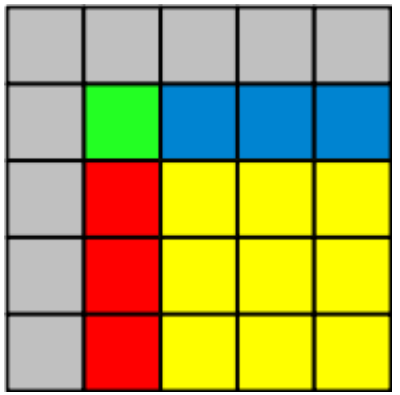
Deuxième version: *right-looking*

Considérons une matrice A, générale, de taille $n = NT \times b$, et tel que les $k-1$ premiers blocs lignes et colonnes soient déjà factorisés. On veut factoriser le premier bloc de b lignes et colonnes de la matrice restante.

$$\begin{pmatrix} A_{11}^{(k)} & A_{12}^{(k)} \\ A_{21}^{(k)} & A_{22}^{(k)} \end{pmatrix} = \begin{pmatrix} L_{11}^{(k)} & 0 \\ \mathbf{L_{21}^{(k)}} & \mathbf{L_{22}^{(k)}} \end{pmatrix} * \begin{pmatrix} U_{11}^{(k)} & \mathbf{U_{12}^{(k)}} \\ 0 & \mathbf{U_{22}^{(k)}} \end{pmatrix}$$

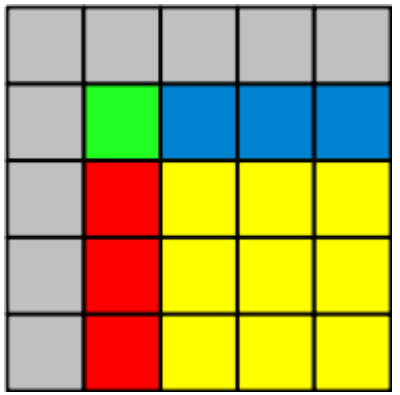
Algorithme right-looking (étapes)

- $A_{11}^{(k)} = L_{11}^{(k)} U_{11}^{(k)}$
- $A_{21}^{(k)} = L_{21}^{(k)} U_{11}^{(k)}$
- $A_{12}^{(k)} = L_{11}^{(k)} U_{12}^{(k)}$
- $A_{22}^{(k)} = L_{21}^{(k)} U_{12}^{(k)} + A_{22}^{(k+1)}$



Algorithme right-looking (étapes)

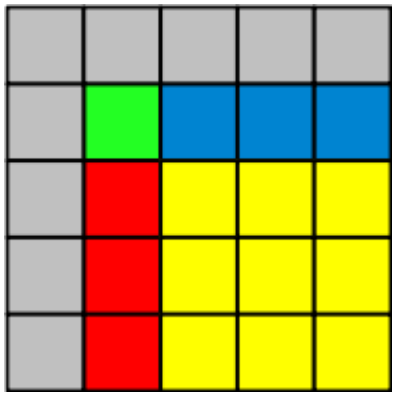
- $A_{11}^{(k)} = L_{11}^{(k)} U_{11}^{(k)}$
- $A_{21}^{(k)} = L_{21}^{(k)} U_{11}^{(k)}$
- $A_{12}^{(k)} = L_{11}^{(k)} U_{12}^{(k)}$
- $A_{22}^{(k)} = L_{21}^{(k)} U_{12}^{(k)} + A_{22}^{(k+1)}$



1. Factorize $A_{11}^{(k)} = L_{11}^{(k)} U_{11}^{(k)}$

Algorithme right-looking (étapes)

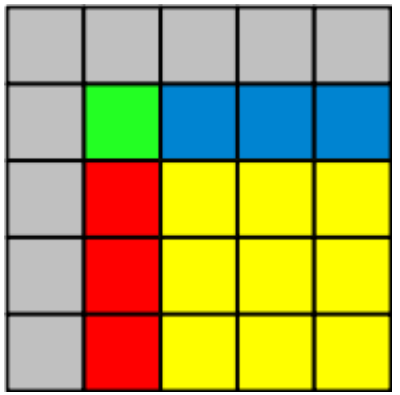
- $A_{11}^{(k)} = L_{11}^{(k)} U_{11}^{(k)}$
- $A_{21}^{(k)} = L_{21}^{(k)} U_{11}^{(k)}$
- $A_{12}^{(k)} = L_{11}^{(k)} U_{12}^{(k)}$
- $A_{22}^{(k)} = L_{21}^{(k)} U_{12}^{(k)} + A_{22}^{(k+1)}$



1. Factorize $A_{11}^{(k)} = L_{11}^{(k)} U_{11}^{(k)}$
2. Solve $A_{21}^{(k)} = L_{21}^{(k)} U_{11}^{(k)}$

Algorithme right-looking (étapes)

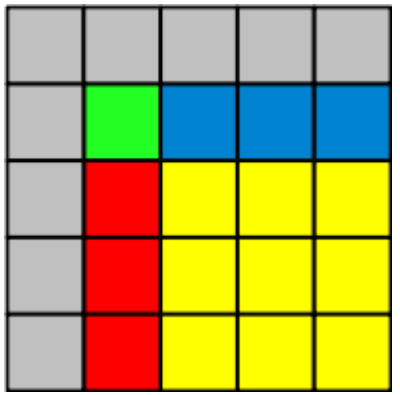
- $A_{11}^{(k)} = L_{11}^{(k)} U_{11}^{(k)}$
- $A_{21}^{(k)} = L_{21}^{(k)} U_{11}^{(k)}$
- $A_{12}^{(k)} = L_{11}^{(k)} U_{12}^{(k)}$
- $A_{22}^{(k)} = L_{21}^{(k)} U_{12}^{(k)} + A_{22}^{(k+1)}$



1. Factorize $A_{11}^{(k)} = L_{11}^{(k)} U_{11}^{(k)}$
2. Solve $A_{21}^{(k)} = L_{21}^{(k)} U_{11}^{(k)}$
3. Solve $A_{12}^{(k)} = L_{11}^{(k)} U_{12}^{(k)}$

Algorithme right-looking (étapes)

- $A_{11}^{(k)} = L_{11}^{(k)} U_{11}^{(k)}$
- $A_{21}^{(k)} = L_{21}^{(k)} U_{11}^{(k)}$
- $A_{12}^{(k)} = L_{11}^{(k)} U_{12}^{(k)}$
- $A_{22}^{(k)} = L_{21}^{(k)} U_{12}^{(k)} + A_{22}^{(k+1)}$



1. Factorize $A_{11}^{(k)} = L_{11}^{(k)} U_{11}^{(k)}$
2. Solve $A_{21}^{(k)} = L_{21}^{(k)} U_{11}^{(k)}$
3. Solve $A_{12}^{(k)} = L_{11}^{(k)} U_{12}^{(k)}$
4. Compute $A_{22}^{(k+1)} = A_{22}^{(k)} - L_{21}^{(k)} U_{12}^{(k)}$