SG6: High Performance Computing

Distributed algorithms on multi-dimensional topologies

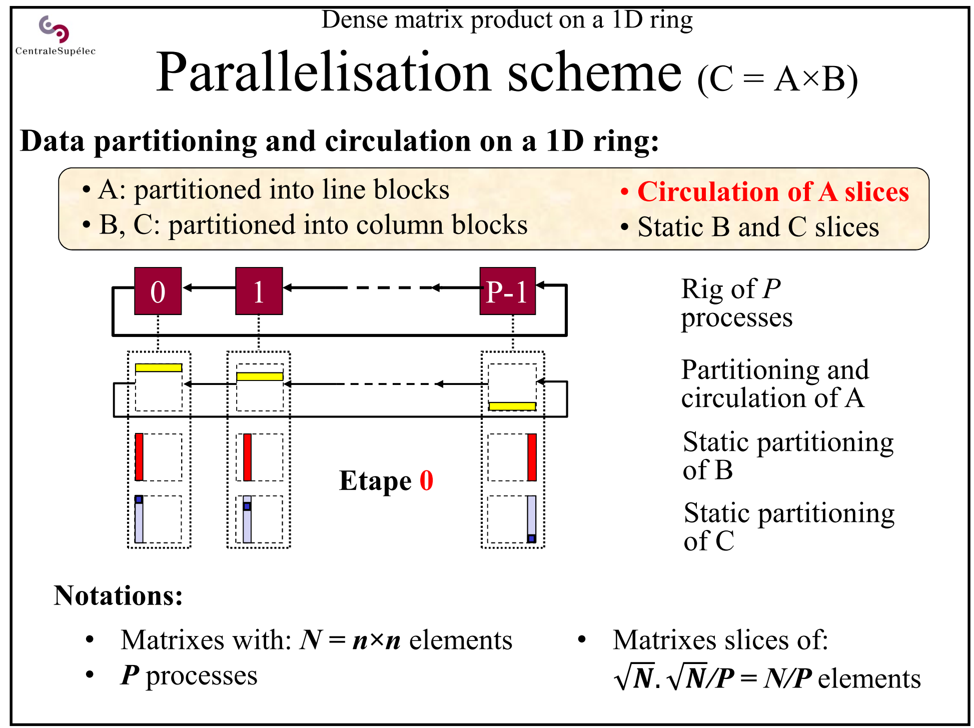
Stéphane Vialle

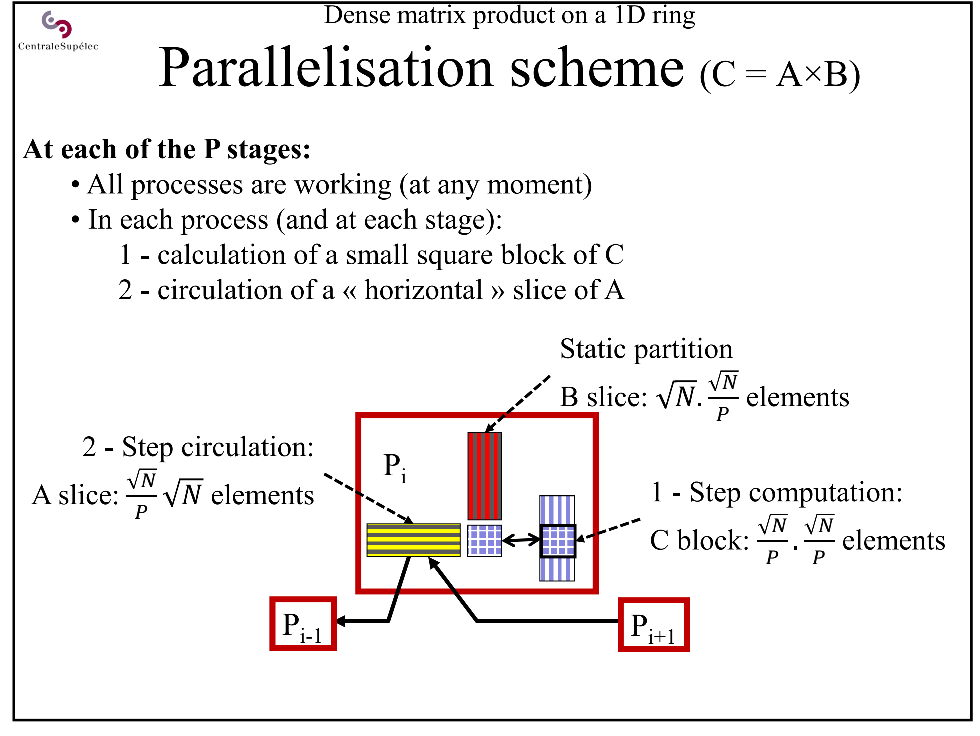
Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

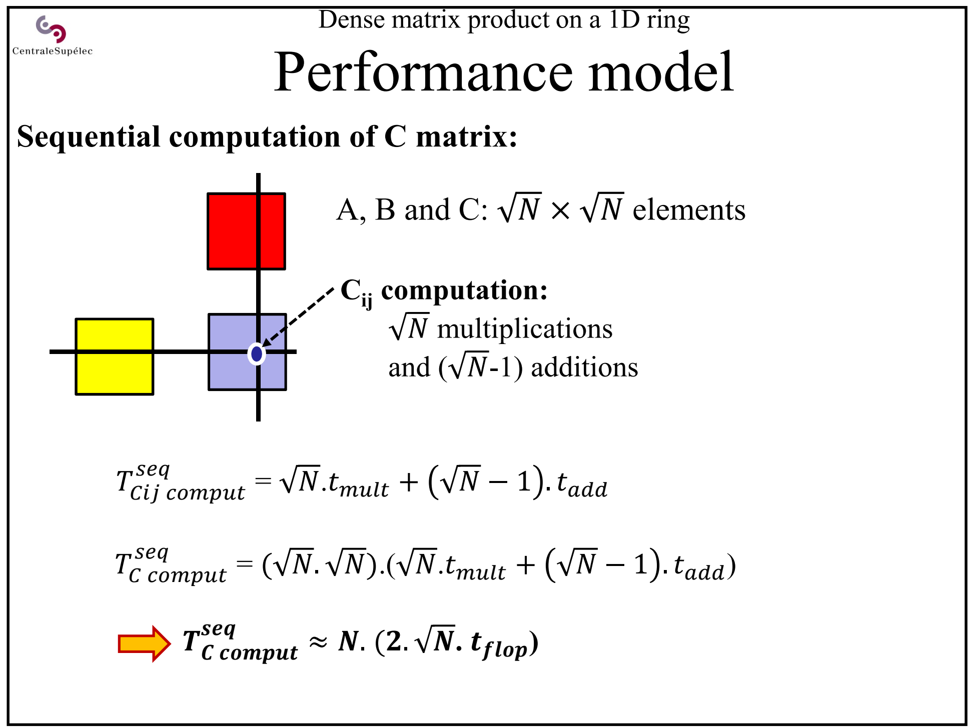
Distributed algorithms on multi-dimensional topologies

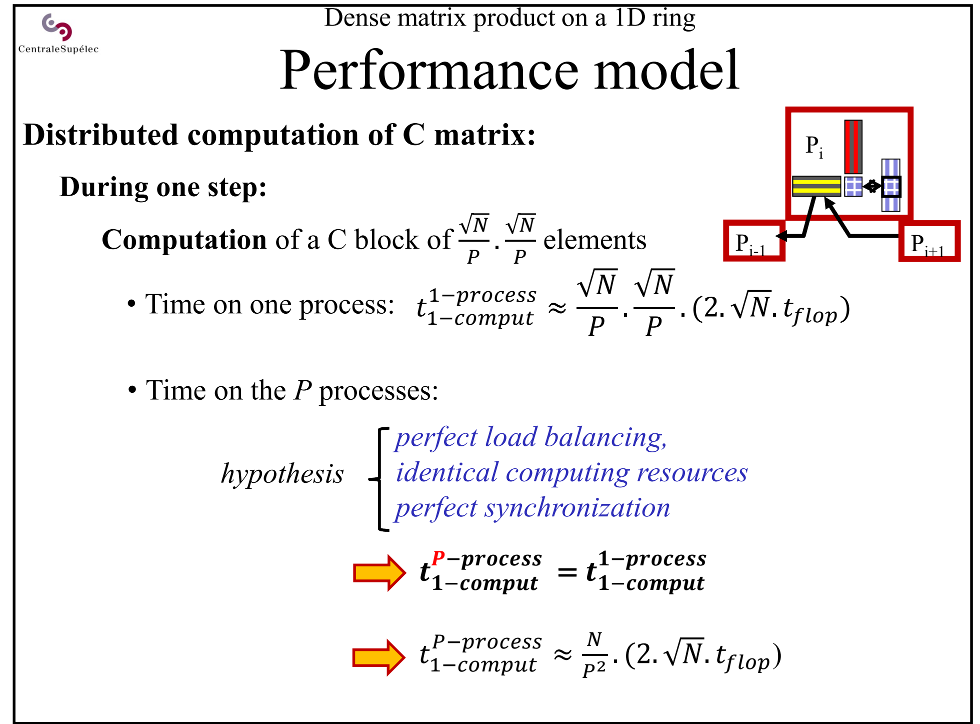
Dense matrix product on a 1D ring
- **Parallelisation scheme**
- **Performance model**

Dense matrix product on a 2D torus
Hyper-quicksort on a kD hypercube


Dense matrix product on a 1D ring
CentraleSupélec
Parallelisation scheme (C = A×B)
Data partitioning and circulation on a 1D ring:
• A: partitioned into line blocks
• B, C: partitioned into column blocks
• Circulation of A slices
• Static B and C slices
0
1
P-1
Rig of P processes
Partitioning and circulation of A
Static partitioning of B
Static partitioning of C
Etape 0
Notations:
• Matrixes with: N = n×n elements
• P processes
• Matrixes slices of: √N.√N/P = N/P elements

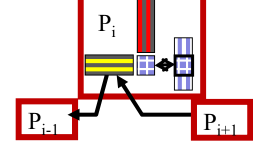

Dense matrix product on a 1D ring
CentraleSupélec
Parallelisation scheme (C = A×B)
At each of the P stages:
• All processes are working (at any moment)
• In each process (and at each stage):
1 - calculation of a small square block of C
2 - circulation of a « horizontal » slice of A
Static partition
B slice: √N.√N/P elements
2 - Step circulation:
A slice: √N/P √N elements
Pi
1 - Step computation:
C block: √N/P . √N/P elements
Pi-1
Pi+1


Dense matrix product on a 1D ring
CentraleSupélec
Performance model
Sequential computation of C matrix:
A, B and C: $\sqrt{N} \times \sqrt{N}$ elements
C_{ij} computation:
$\sqrt{N}$ multiplications
and ($\sqrt{N}$-1) additions
$T^{seq}_{Cij\ comput} = \sqrt{N}.t_{mult} + (\sqrt{N}-1).t_{add}$
$T^{seq}_{C\ comput} = (\sqrt{N}.\sqrt{N}).(\sqrt{N}.t_{mult} + (\sqrt{N}-1).t_{add})$
$T^{seq}_{C\ comput} \approx N.(2.\sqrt{N}.t_{flop})$


Dense matrix product on a 1D ring
CentraleSupélec
Performance model
Distributed computation of C matrix:
During one step:
Computation of a C block of $\frac{\sqrt{N}}{P}.\frac{\sqrt{N}}{P}$ elements
• Time on one process: $t^{1-process}_{1-comput} \approx \frac{\sqrt{N}}{P}.\frac{\sqrt{N}}{P}.(2.\sqrt{N}.t_{flop})$
• Time on the P processes:
hypothesis
perfect load balancing,
identical computing resources
perfect synchronization
$t^{P-process}_{1-comput} = t^{1-process}_{1-comput}$
$t^{P-process}_{1-comput} \approx \frac{N}{P^2}.(2.\sqrt{N}.t_{flop})$
P_i
P_{i-1}
P_{i+1}

Performance model

Distributed computation of C matrix:



During one step:

Circulation of A slices of $\sqrt{N}.\frac{\sqrt{N}}{P}$ elements

- Time of one communication: $P_{i-1} \leftarrow P_i$

$$t_{1-comm} = t_s + (\sqrt{N}.\frac{\sqrt{N}}{P}).t_w \quad \begin{cases} t_s : \text{setup time or latency time} \\ t_w = sizeof(1\ element)/B_w \end{cases}$$

- Time of all the communications accross the P processes (during 1 step)

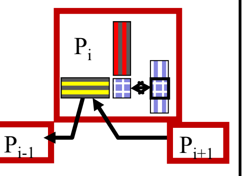
hypothesis
- *All communications are executed in parallel,*
- → *each process can send and receive in parallel…*
  …depends on the hardware & on the MPI comm routine!
- → *all communications cross the same « network »…*
  … depends on the deployment!

⇨ $$t_{1-circulation}^{P-processes} = t_{1-comm} \approx \frac{N}{P}.t_w$$

Dense matrix product on a 1D ring

Performance model

Distributed computation of C matrix:



During one step:

$$t_{1-step}^{P-process} = t_{1-comput}^{P-process} + t_{1-circulation}^{P-processes}$$

$$t_{1-step}^{P-process} \approx \frac{N}{P^2}.(2.\sqrt{N}.t_{flop}) + \frac{N}{P}.t_w$$

During the *P* steps of C distributed computation:

$$T_{C\ comput}^{distributed} = P.t_{1-step}^{P-process}$$

⇨ $$T_{C\ comput}^{distributed} \approx \frac{N}{P}.(2.\sqrt{N}.t_{flop}) + N.t_w$$ **with: *P > 1***

hypothesis
- *All computations identical & executed in parallel*
- *All communications identical & executed in parallel*

Performance model

Distributed computation of C matrix:

Execution times:

$$T_{C\ comput}^{seq} \approx N.(2.\sqrt{N}.t_{flop})$$

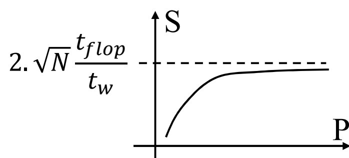
$$T_{C\ comput}^{distributed} \approx \frac{N}{P}.(2.\sqrt{N}.t_{flop}) + N.t_w \qquad \text{with: } P > 1$$

Speedup:

$$S(N,P) = \frac{T_{C\ comput}^{seq}}{T_{C\ comput}^{distributed}} = \frac{N.(2.\sqrt{N}.t_{flop})}{\frac{N}{P}.(2.\sqrt{N}.t_{flop}) + N.t_w}$$

$$\boldsymbol{S(N,P) = P.\frac{1}{1 + \frac{P}{2.\sqrt{N}} \times \frac{t_w}{t_{flop}}}}$$

With N = N_0



$2.\sqrt{N}\frac{t_{flop}}{t_w}$

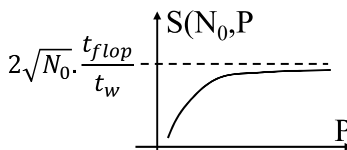
Dense matrix product on a 1D ring

Performance model

Distributed computation of C matrix:

$$\boldsymbol{S(N,P) = P.\frac{1}{1 + \frac{P}{2.\sqrt{N}} \times \frac{t_w}{t_{flop}}}}$$

- **When data size is fixed:** N = N_0



$2\sqrt{N_0}.\frac{t_{flop}}{t_w}$

- **When $P = P_0$ is fixed**, and data size increases:

$$\lim_{N\to\infty}(S(N, P_0)) = P_0$$

Speedup increases and tends to P_0 (ideal S)

In fact: $T_{C\ comput}^{distributed} \approx \underbrace{\frac{N}{P_0}.(2.\sqrt{N}.t_{flop})}_{O(N^{3/2})} + \underbrace{N.t_w}_{O(N)}$

When increasing the pb size, on a fixed number of nodes:

Computations increase faster than communications

→ Speedup will increase! …. Nice problem!

CentraleSupélec

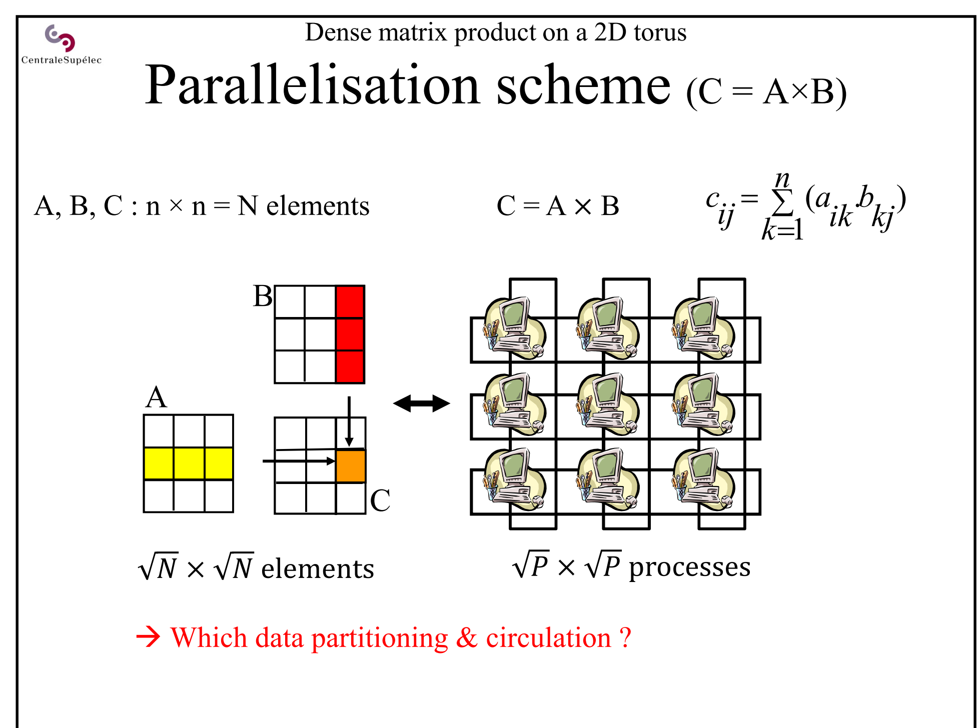
Distributed algorithms on multi-dimensional topologies

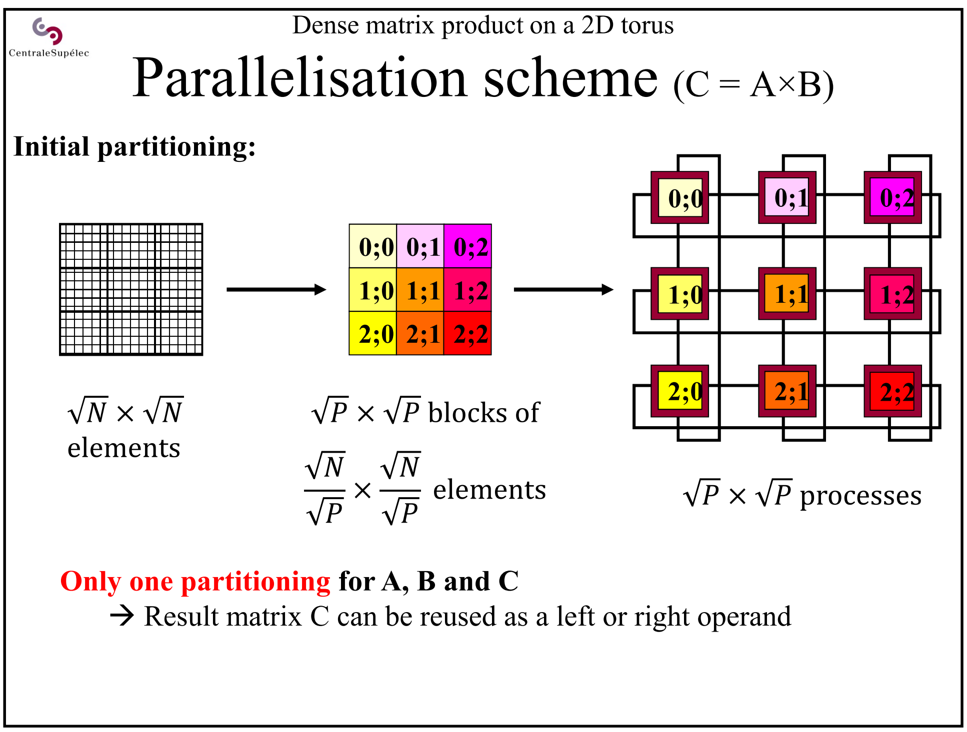
Dense matrix product on a 1D ring

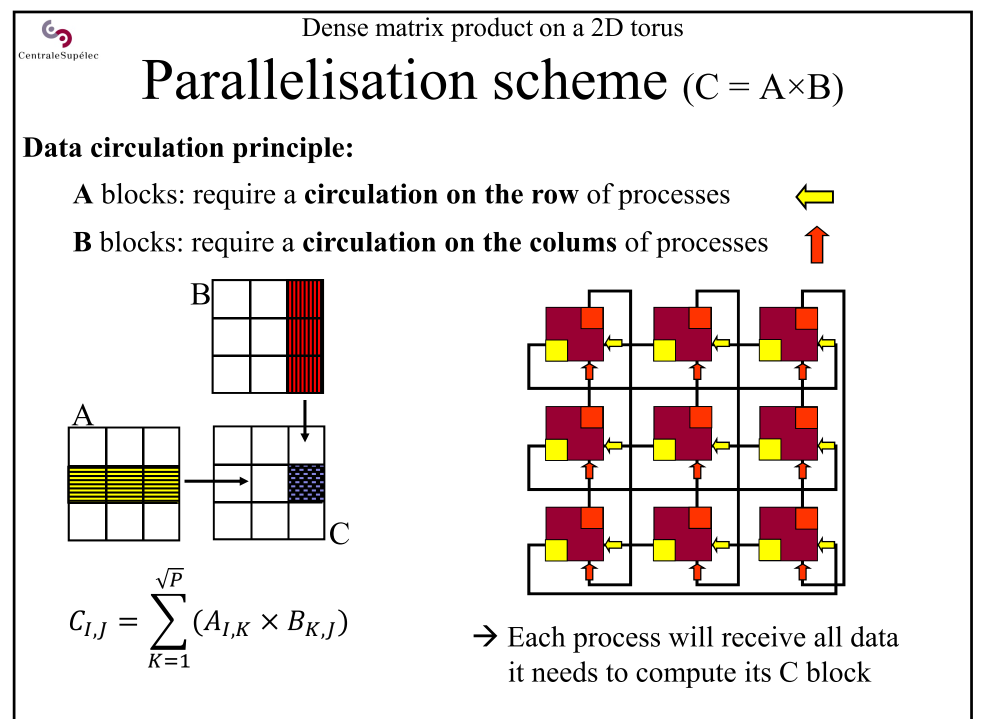
Dense matrix product on a 2D torus

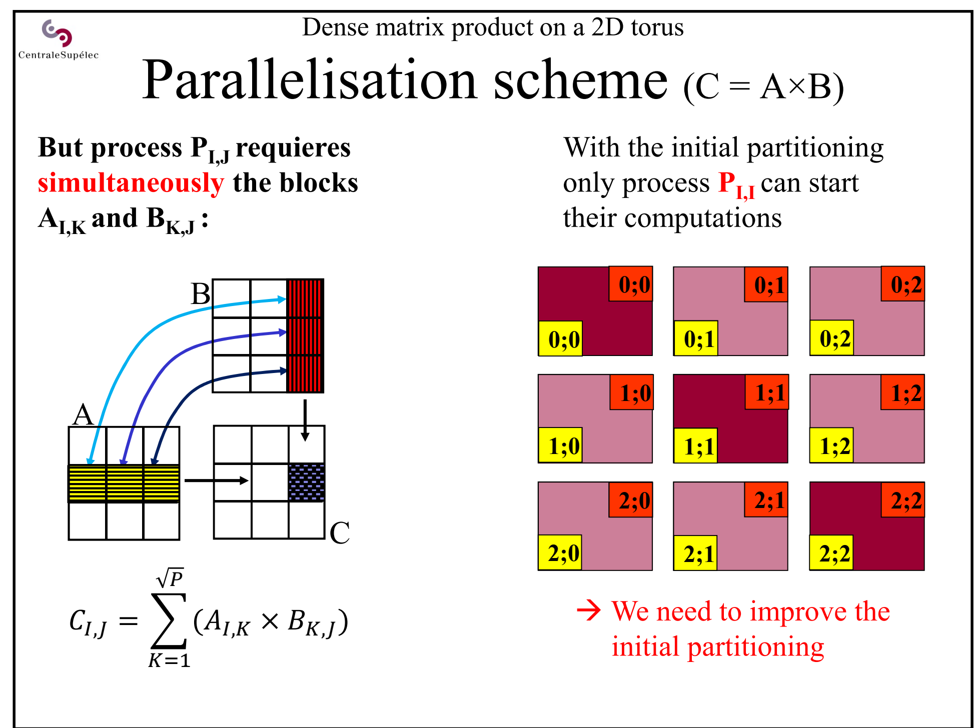
- **Parallelisation scheme**
- **Performance model**

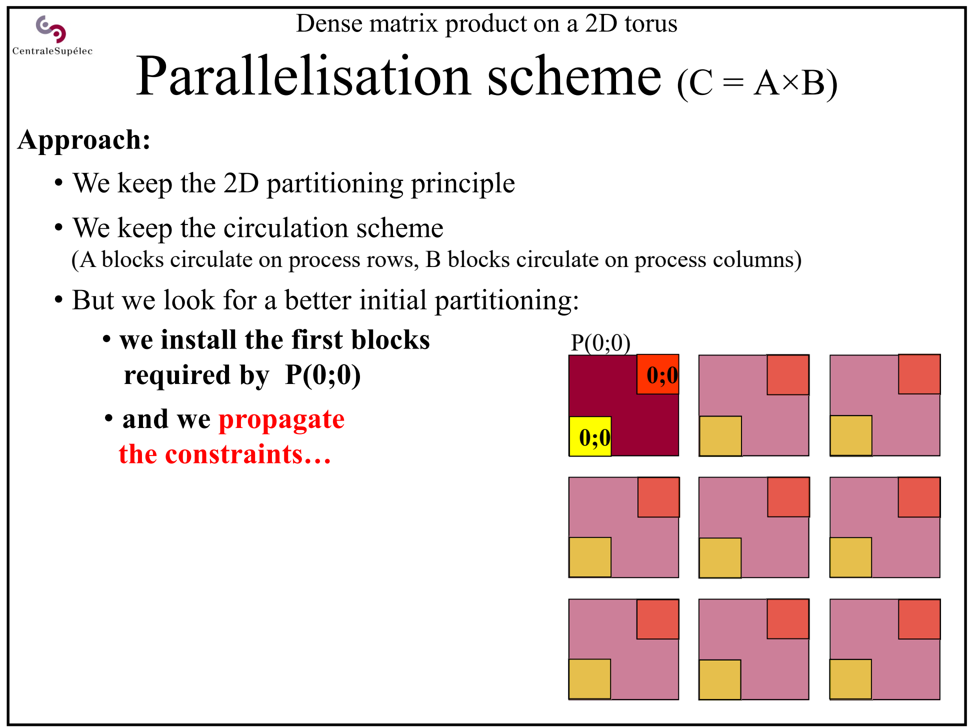
Hyper-quicksort on a kD hypercube

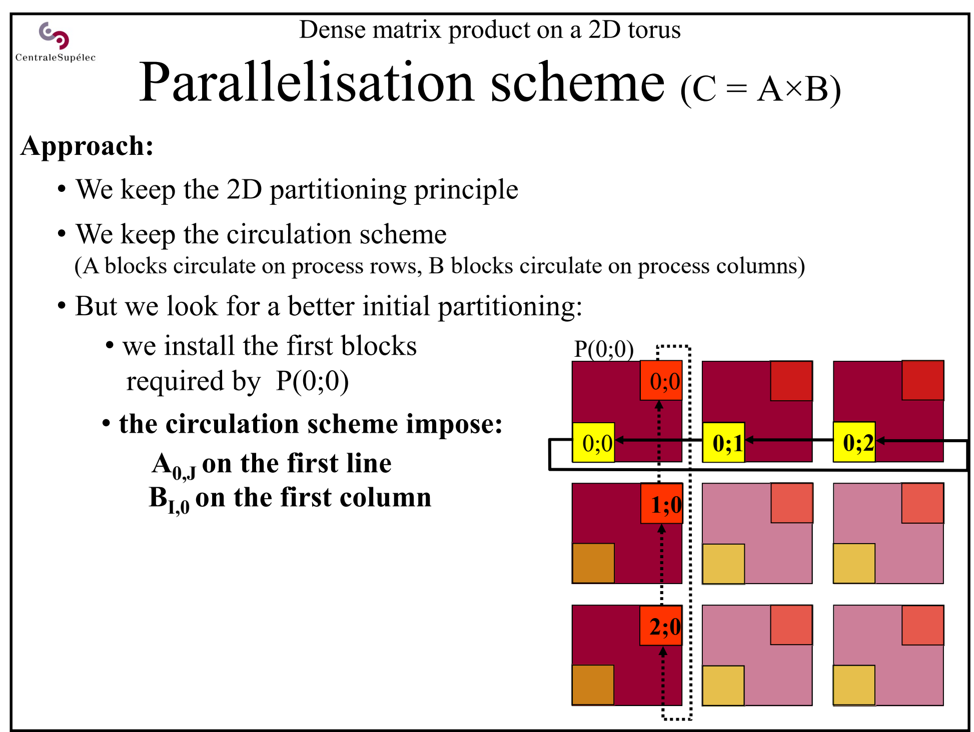


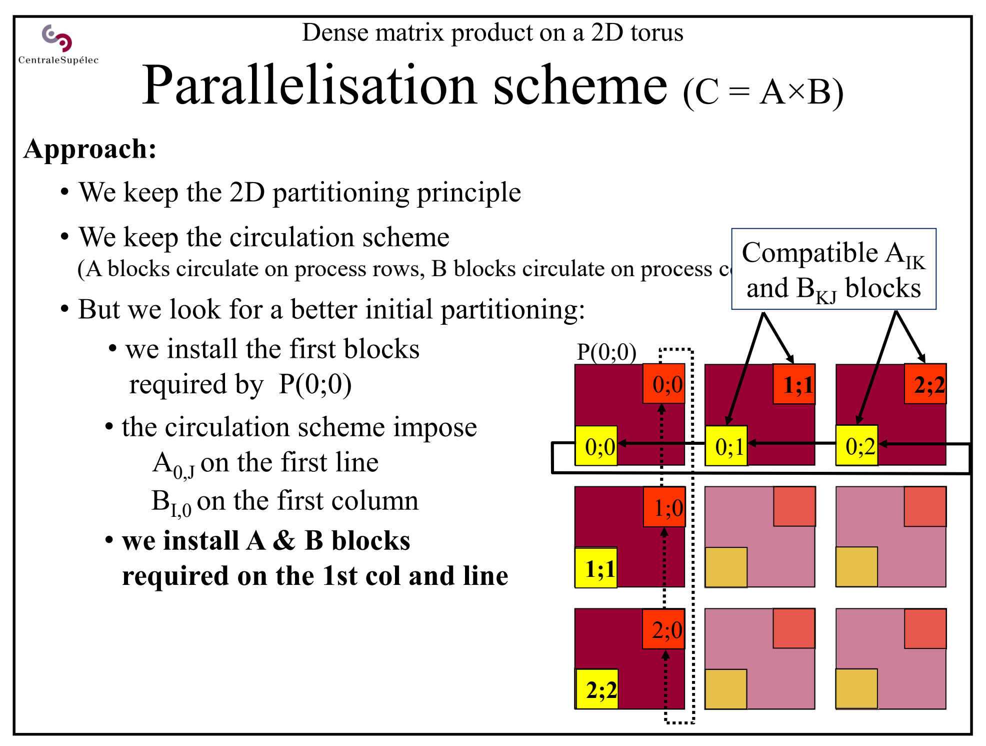

CentraleSupélec
Dense matrix product on a 2D torus
Parallelisation scheme (C = A×B)
Initial partitioning:
0;0 0;1 0;2
1;0 1;1 1;2
2;0 2;1 2;2
$\sqrt{N} \times \sqrt{N}$
elements
$\sqrt{P} \times \sqrt{P}$ blocks of
$\frac{\sqrt{N}}{\sqrt{P}} \times \frac{\sqrt{N}}{\sqrt{P}}$ elements
$\sqrt{P} \times \sqrt{P}$ processes
Only one partitioning for A, B and C
→ Result matrix C can be reused as a left or right operand

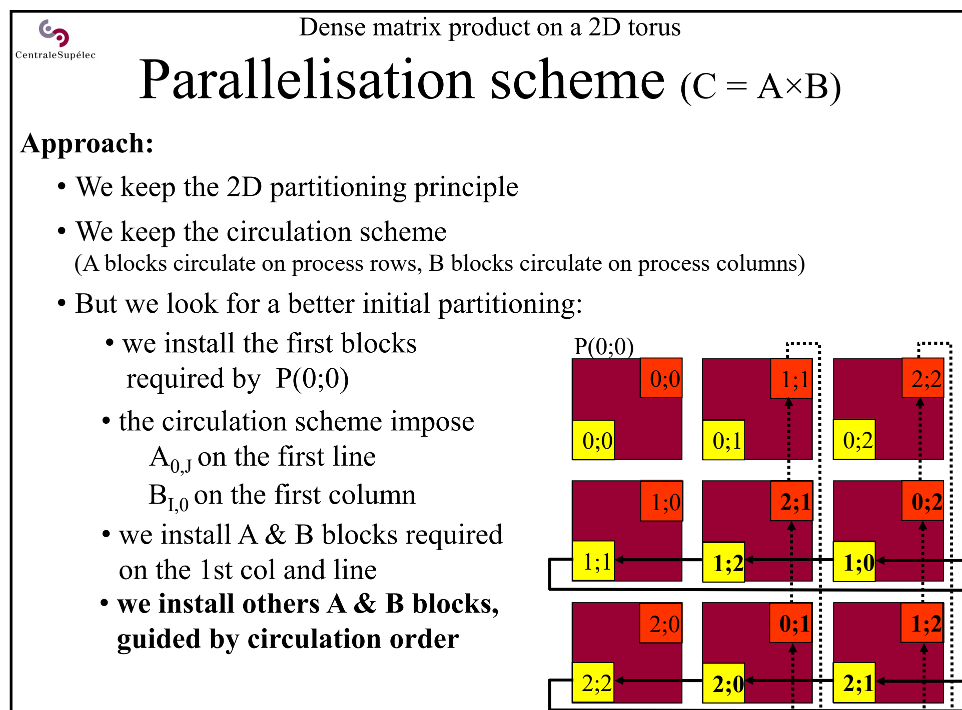

CentraleSupélec
Dense matrix product on a 2D torus
Parallelisation scheme (C = A×B)
Data circulation principle:
A blocks: require a circulation on the row of processes
B blocks: require a circulation on the colums of processes
B
A
C
$C_{I,J} = \sum_{K=1}^{\sqrt{P}} (A_{I,K} \times B_{K,J})$
→ Each process will receive all data
it needs to compute its C block


Dense matrix product on a 2D torus
CentraleSupélec
Parallelisation scheme (C = A×B)
But process $P_{I,J}$ requieres simultaneously the blocks $A_{I,K}$ and $B_{K,J}$:
B
A
C
$C_{I,J} = \sum_{K=1}^{\sqrt{P}} (A_{I,K} \times B_{K,J})$
With the initial partitioning only process $P_{I,I}$ can start their computations
0;0
0;1
0;2
1;0
1;1
1;2
2;0
2;1
2;2
→ We need to improve the initial partitioning

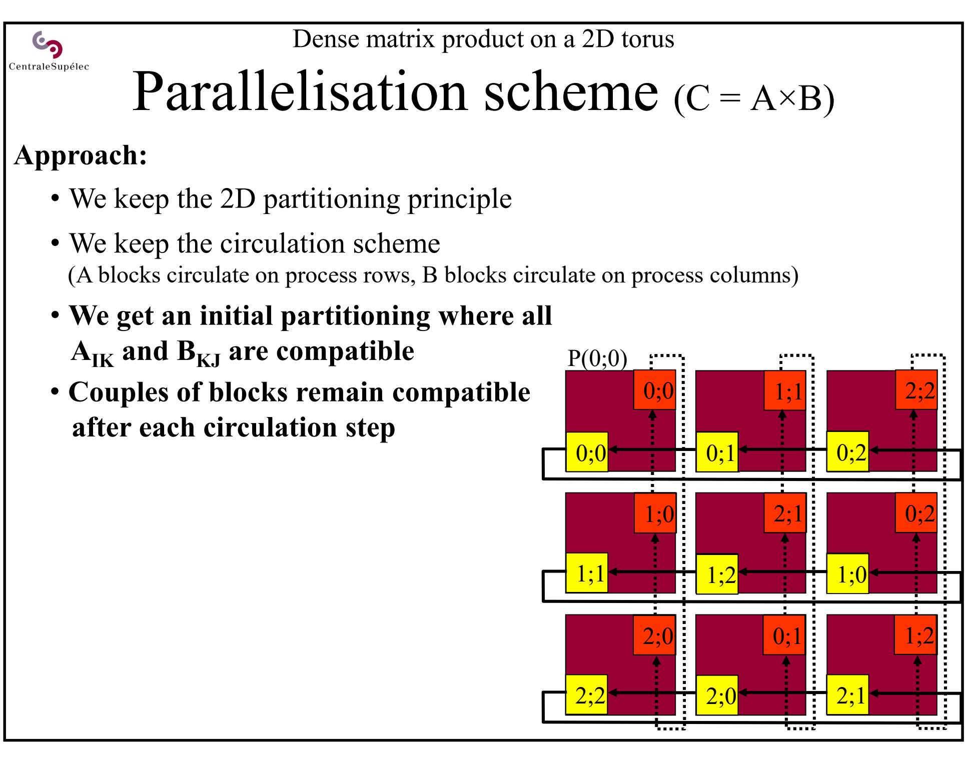

Dense matrix product on a 2D torus
CentraleSupélec
Parallelisation scheme (C = A×B)
Approach:
• We keep the 2D partitioning principle
• We keep the circulation scheme
(A blocks circulate on process rows, B blocks circulate on process columns)
• But we look for a better initial partitioning:
• we install the first blocks required by P(0;0)
• and we propagate the constraints…
P(0;0)
0;0
0;0


Dense matrix product on a 2D torus
CentraleSupélec
Parallelisation scheme (C = A×B)
Approach:
• We keep the 2D partitioning principle
• We keep the circulation scheme
(A blocks circulate on process rows, B blocks circulate on process columns)
• But we look for a better initial partitioning:
• we install the first blocks required by P(0;0)
• the circulation scheme impose:
A0,J on the first line
BI,0 on the first column
P(0;0)
0;0
0;0
0;1
0;2
1;0
2;0


Dense matrix product on a 2D torus
CentraleSupélec
Parallelisation scheme (C = A×B)
Approach:
• We keep the 2D partitioning principle
• We keep the circulation scheme
(A blocks circulate on process rows, B blocks circulate on process c
• But we look for a better initial partitioning:
• we install the first blocks required by P(0;0)
• the circulation scheme impose
A0,J on the first line
BI,0 on the first column
• we install A & B blocks required on the 1st col and line
Compatible AIK and BKJ blocks
P(0;0)
0;0
1;1
2;2
0;0
0;1
0;2
1;0
1;1
2;0
2;2



CentraleSupélec
Dense matrix product on a 2D torus
Parallelisation scheme (C = A×B)
Approach:
• We keep the 2D partitioning principle
• We keep the circulation scheme
(A blocks circulate on process rows, B blocks circulate on process columns)
• But we look for a better initial partitioning:
• we install the first blocks required by P(0;0)
• the circulation scheme impose
$A_{0,J}$ on the first line
$B_{I,0}$ on the first column
• we install A & B blocks required on the 1st col and line
• we install others A & B blocks, guided by circulation order
P(0;0)
0;0
1;1
2;2
0;0
0;1
0;2
1;0
2;1
0;2
1;1
1;2
1;0
2;0
0;1
1;2
2;2
2;0
2;1

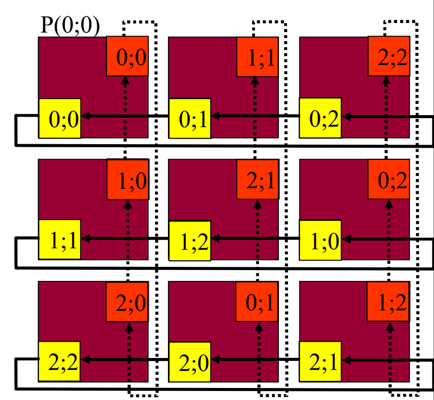


CentraleSupélec
Dense matrix product on a 2D torus
Parallelisation scheme (C = A×B)
Approach:
• We keep the 2D partitioning principle
• We keep the circulation scheme
(A blocks circulate on process rows, B blocks circulate on process columns)
• We get an initial partitioning where all A_{IK} and B_{KJ} are compatible
• Couples of blocks remain compatible after each circulation step
P(0;0)
0;0
1;1
2;2
0;0
0;1
0;2
1;0
2;1
0;2
1;1
1;2
1;0
2;0
0;1
1;2
2;2
2;0
2;1

Parallelisation scheme (C = A×B)

Approach:

- We keep the 2D partitioning principle
- We keep the circulation scheme
  (A blocks circulate on process rows, B blocks circulate on process columns)
- We get an initial partitioning where all A_{IK} and B_{KJ} are compatible
- Couples of blocks remain compatible after each circulation step



Finally:

- **Row *I* has been pre-shifted of *I* steps to the left**
- **Column *J* has been pre-shifted of *J* steps to the top**

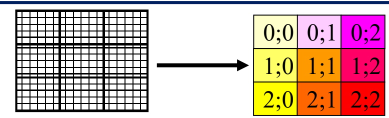
Dense matrix product on a 2D torus

Parallelisation scheme (C = A×B)

Canon algorithm: C = A×B

Product of dense matrixes of $\sqrt{N} \times \sqrt{N}$ elements
on a 2D torus of $\sqrt{P} \times \sqrt{P}$ processes

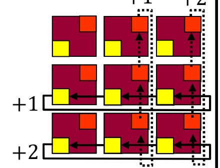
1 : Matrix partitioning: block I,J
on process I,J



0;0	0;1	0;2
1;0	1;1	1;2
2;0	2;1	2;2

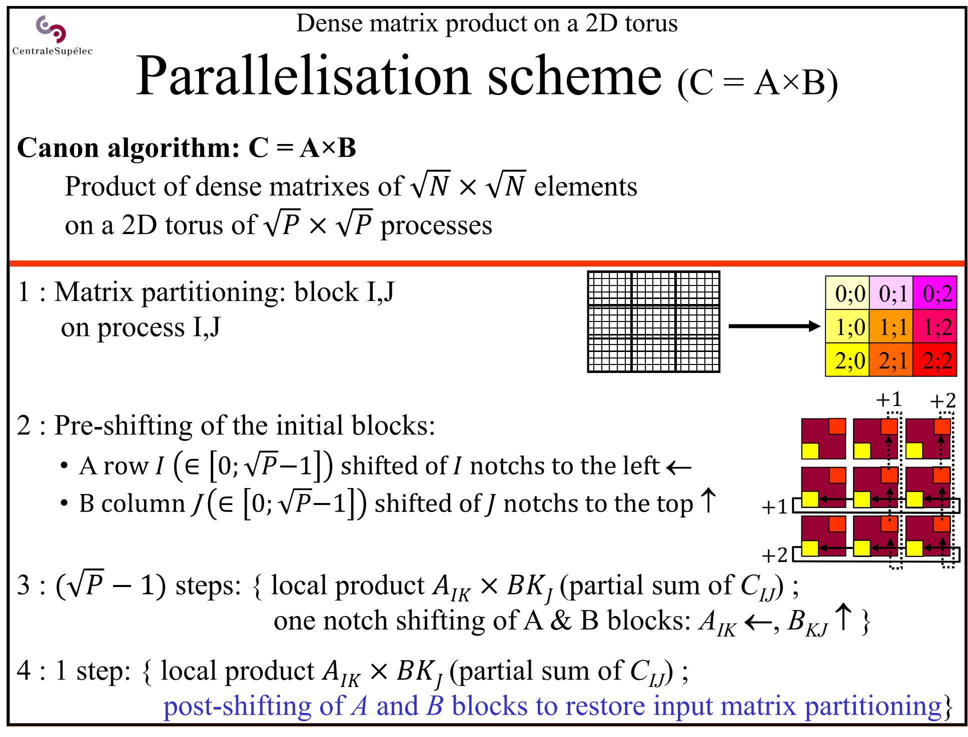
2 : Pre-shifting of the initial blocks:

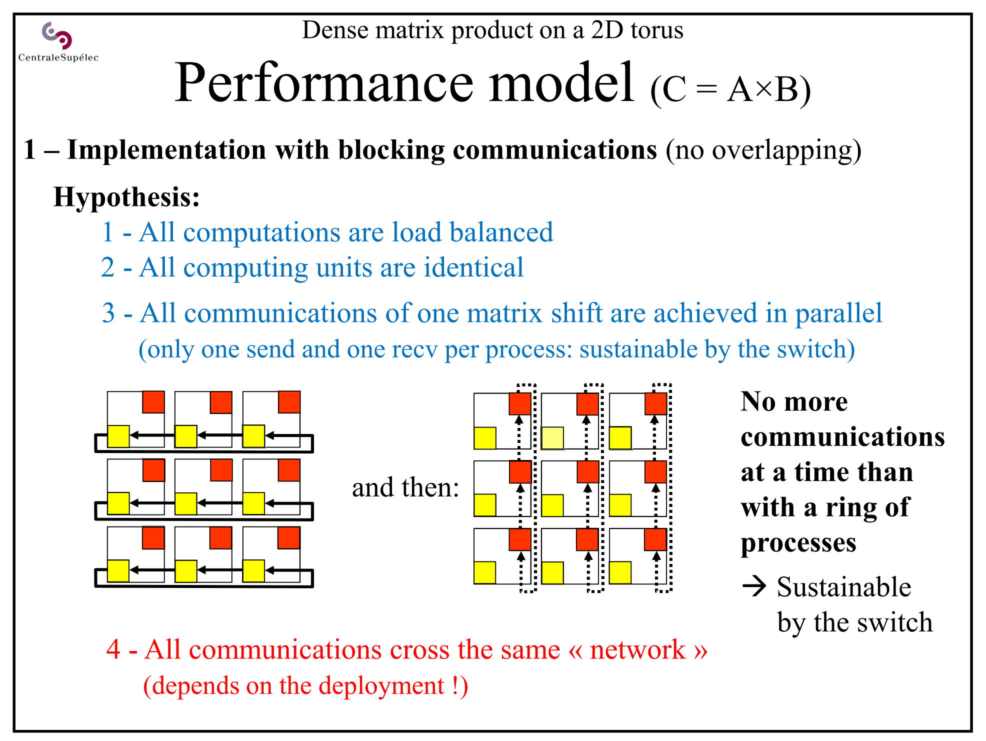
- A row I ($\in [0; \sqrt{P}-1]$) shifted of I step to the left ←
- B column J ($\in [0; \sqrt{P}-1]$) shifted of J step to the top ↑



3 : $\sqrt{P}$ steps: { local product $A_{IK} \times BK_J$ (partial sum of C_{IJ}) ;
1 notch shifting of A & B blocks: A_{IK} ←, B_{KJ} ↑ }

4 : Post-shifting of A and B blocks to restore input matrix partitioning


Dense matrix product on a 2D torus
CentraleSupélec
Parallelisation scheme (C = A×B)
Canon algorithm: C = A×B
Product of dense matrixes of $\sqrt{N} \times \sqrt{N}$ elements
on a 2D torus of $\sqrt{P} \times \sqrt{P}$ processes
1 : Matrix partitioning: block I,J
on process I,J
0;0 0;1 0;2
1;0 1;1 1;2
2;0 2;1 2;2
2 : Pre-shifting of the initial blocks:
• A row I ($\in [0;\sqrt{P}-1]$) shifted of I notchs to the left ←
• B column J ($\in [0;\sqrt{P}-1]$) shifted of J notchs to the top ↑
+1 +2
+1
+2
3 : ($\sqrt{P}-1$) steps: { local product $A_{IK} \times BK_J$ (partial sum of C_{IJ}) ;
one notch shifting of A & B blocks: A_{IK} ←, B_{KJ} ↑ }
4 : 1 step: { local product $A_{IK} \times BK_J$ (partial sum of C_{IJ}) ;
post-shifting of A and B blocks to restore input matrix partitioning}


Dense matrix product on a 2D torus
CentraleSupélec
Performance model (C = A×B)
1 – Implementation with blocking communications (no overlapping)
Hypothesis:
1 - All computations are load balanced
2 - All computing units are identical
3 - All communications of one matrix shift are achieved in parallel
(only one send and one recv per process: sustainable by the switch)
and then:
No more communications at a time than with a ring of processes
→ Sustainable by the switch
4 - All communications cross the same « network »
(depends on the deployment !)

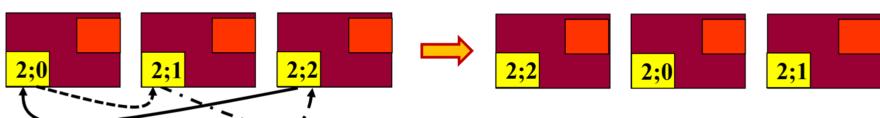
Dense matrix product on a 2D torus

Performance model (C = A×B)

1 – Implementation with blocking communications (no overlapping)

Pre- and post-shifting implementation on a cluster:

Each block can be moved with only one send and one recv
(no need to concatenate several elementary shifts on a cluster network)



3 parallel send-&-recv ≡ 2 notch shifting

$\Rightarrow t_{pre-shifting-A} = t_s + (\frac{\sqrt{N}}{\sqrt{P}} \times \frac{\sqrt{N}}{\sqrt{P}}).t_w \approx \frac{N}{P}.t_w$

Considering blocking communications:

$\Rightarrow \boldsymbol{t_{pre-shifting-A-B} \approx 2.\frac{N}{P}.t_w}$ **Same result for post-shifting**

Dense matrix product on a 2D torus

Performance model (C = A×B)

1 – Implementation with blocking communications (no overlapping)

- Pre-shifting (A & B): $2.\frac{N}{P}t_w$
- 1 step computations: $\frac{\sqrt{N}}{\sqrt{P}}.\frac{\sqrt{N}}{\sqrt{P}}.(2.\frac{\sqrt{N}}{\sqrt{P}}.t_{flop}) = 2.\left(\frac{N}{P}\right)^{\frac{3}{2}}.t_{flop}$
  (*P* processes in parallel)
- 1 step circulation: $2.(\frac{\sqrt{N}}{\sqrt{P}}.\frac{\sqrt{N}}{\sqrt{P}}).t_w = 2.\frac{N}{P}t_w$
  (blocking communications)
- Post-shifting (A & B): $2.\frac{N}{P}t_w$

Complete distributed computation:

$$T_{C\ comput}^{distributed} \approx t_{pre-shifting} + (\sqrt{P}\text{-}1).(t_{1-step-comput} + t_{1-step-circulation}) + (t_{1-step-comput} + t_{post-shifting})$$

Performance model (C = A×B)

1 – Implementation with blocking communications (no overlapping)

- Pre-shifting (A & B): $2.\frac{N}{P}t_w$
- 1 step computations: $\frac{\sqrt{N}}{\sqrt{P}}.\frac{\sqrt{N}}{\sqrt{P}}.(2.\frac{\sqrt{N}}{\sqrt{P}}.t_{flop}) = 2.\left(\frac{N}{P}\right)^{\frac{3}{2}}.t_{flop}$
  (P processes in parallel)
- 1 step circulation: $2.(\frac{\sqrt{N}}{\sqrt{P}}.\frac{\sqrt{N}}{\sqrt{P}}).t_w = 2.\frac{N}{P}t_w$
  (blocking communications)
- Post-shifting (A & B): $2.\frac{N}{P}t_w$

Complete distributed computation:

$$T_{C\ comput}^{distributed} \approx 2.\frac{N}{P}t_w + (\sqrt{P}\text{-}1).(2.\left(\frac{N}{P}\right)^{\frac{3}{2}}.t_{flop} + 2.\frac{N}{P}t_w) + (2.\left(\frac{N}{P}\right)^{\frac{3}{2}}.t_{flop} + 2.\frac{N}{P}t_w)$$

Dense matrix product on a 2D torus

Performance model (C = A×B)

1 – Implementation with blocking communications (no overlapping)

- Pre-shifting (A & B): $2.\frac{N}{P}t_w$
- 1 step computations: $\frac{\sqrt{N}}{\sqrt{P}}.\frac{\sqrt{N}}{\sqrt{P}}.(2.\frac{\sqrt{N}}{\sqrt{P}}.t_{flop}) = 2.\left(\frac{N}{P}\right)^{\frac{3}{2}}.t_{flop}$
  (P processes in parallel)
- 1 step circulation: $2.(\frac{\sqrt{N}}{\sqrt{P}}.\frac{\sqrt{N}}{\sqrt{P}}).t_w = 2.\frac{N}{P}t_w$
  (blocking communications)
- Post-shifting (A & B): $2.\frac{N}{P}t_w$

Complete distributed computation:

$$T_{C\ comput}^{distributed} \approx 2.\frac{N^{3/2}}{P}t_{flop} + 2.(\sqrt{P}+1).\frac{N}{P}t_w$$ **with: $P > 1$**

Dense matrix product on a 2D torus

Performance model (C = A×B)

1 – Implementation with blocking communications (no overlapping)

Complete distributed computation:

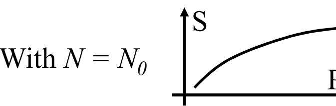
$$T_{C\,comput}^{distributed} \approx 2.\frac{N^{3/2}}{P} t_{flop} + 2.(\sqrt{P}+1).\frac{N}{P} t_w$$

Speedup:

$$S(N,P) = \frac{T_{C\,comput}^{Seq}}{T_{C\,comput}^{distributed}} \approx \frac{2.N^{3/2}.t_{flop}}{2.\frac{N^{3/2}}{P} t_{flop} + 2.(\sqrt{P}+1).\frac{N}{P} t_w}$$

$$S(N,P) \approx P.\frac{1}{1+\frac{\sqrt{P}+1}{\sqrt{N}} \times \frac{t_w}{t_{flop}}}$$

With $N = N_0$



Again: $\lim_{N\to\infty} S(N,P_0) = P_0$

When pb size increase, speedup increases and tends to ideal speedup

Dense matrix product on a 2D torus

Performance model (C = A×B)

2 – Implementation with non-blocking communications

If we assume the switch can achieve A and B communications in parallel:

- Pre-shifting (A & B): $2.\frac{N}{P} t_w \rightarrow \frac{N}{P} t_w$
- 1 step computations: $2.\left(\frac{N}{P}\right)^{\frac{3}{2}}.t_{flop}$ (unchanged)
  (*P* processes in parallel)
- 1 step circulation: $2.\frac{N}{P} t_w \rightarrow \frac{N}{P} t_w$
  (blocking communications)
- Post-shifting (A & B): $2.\frac{N}{P} t_w \rightarrow \frac{N}{P} t_w$

Complete distributed computation:

$$T_{C\,comput}^{distributed} \approx 2.\frac{N^{3/2}}{P} t_{flop} + 2.(\sqrt{P}+1).\frac{N}{P} t_w$$

$$\rightarrow 2.\frac{N^{3/2}}{P} t_{flop} + (\sqrt{P}+1).\frac{N}{P} t_w$$

Dense matrix product on a 2D torus

Performance model (C = A×B)

2 – Implementation with non-blocking communications

Complete distributed computation:

$$T_{C\ comput}^{distributed} \approx 2.\frac{N^{3/2}}{P}t_{flop} + (\sqrt{P}+1).\frac{N}{P}t_w$$

Speedup:

$$S(N,P) = \frac{T_{C\ comput}^{Seq}}{T_{C\ comput}^{distributed}} \approx \frac{2.N^{3/2}.t_{flop}}{2.\frac{N^{3/2}}{P}t_{flop} + (\sqrt{P}+1).\frac{N}{P}t_w}$$

$$\boldsymbol{S(N,P) \approx P.\frac{1}{1+\frac{\sqrt{P}+1}{2.\sqrt{N}} \times \frac{t_w}{t_{flop}}}}$$

Speedup increases faster

Again: $\lim_{N\to\infty} S(N,P_0) = P_0$

When pb size increase, speedup increases and tends to ideal speedup

Dense matrix product on a 2D torus

Performance model (C = A×B)

Comparison 1D Ring / 2D Torus solutions:

	1D ring	**2D torus**
Blocking comm	$T_{par}(P) \approx 2.\frac{N^{3/2}}{P}.t_{flop} + N.t_w$	$T_{par}(P) \approx 2.\frac{N^{3/2}}{P}.t_{flop} + 2.(\sqrt{P}+1)\frac{N}{P}.t_w$
Non-blocking: A & B circ. in parallel		$T_{par}(P) \approx 2.\frac{N^{3/2}}{P}.t_{flop} + (\sqrt{P}+1)\frac{N}{P}.t_w$

2D torus solution *should be* faster when

- P ≥ 3 *with blocking comm* → P ≥ 2^2
- P ≥ 2 *with blocking comm* → P ≥ 2^2

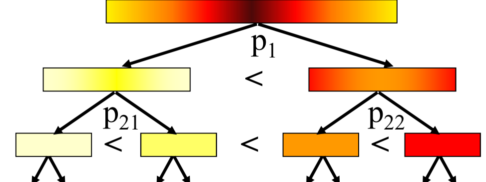
Il the assumptions on the network switch are right…

Distributed algorithms on multi-dimensional topologies

Dense matrix product on a 1D ring
Dense matrix product on a 2D torus
Hyper-quicksort on a kD hypercube
- **Parallelisation scheme**

Principe séquentiel

Principe :
- Tri récursif
- Diviser-pour-régner
- Séparation des données selon des valeurs *pivots*

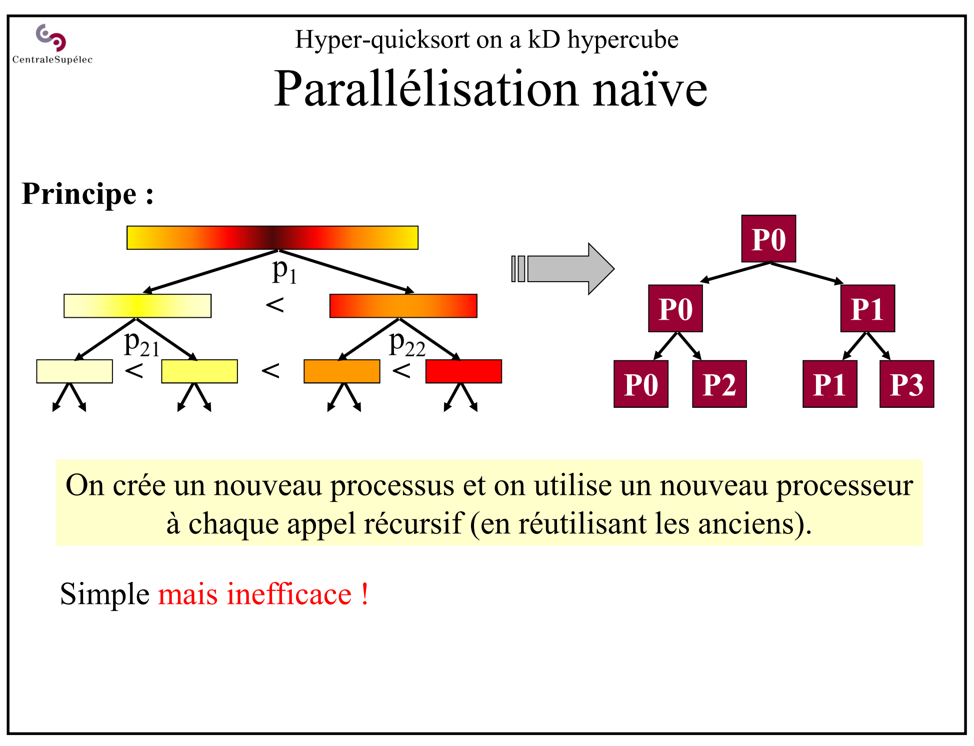


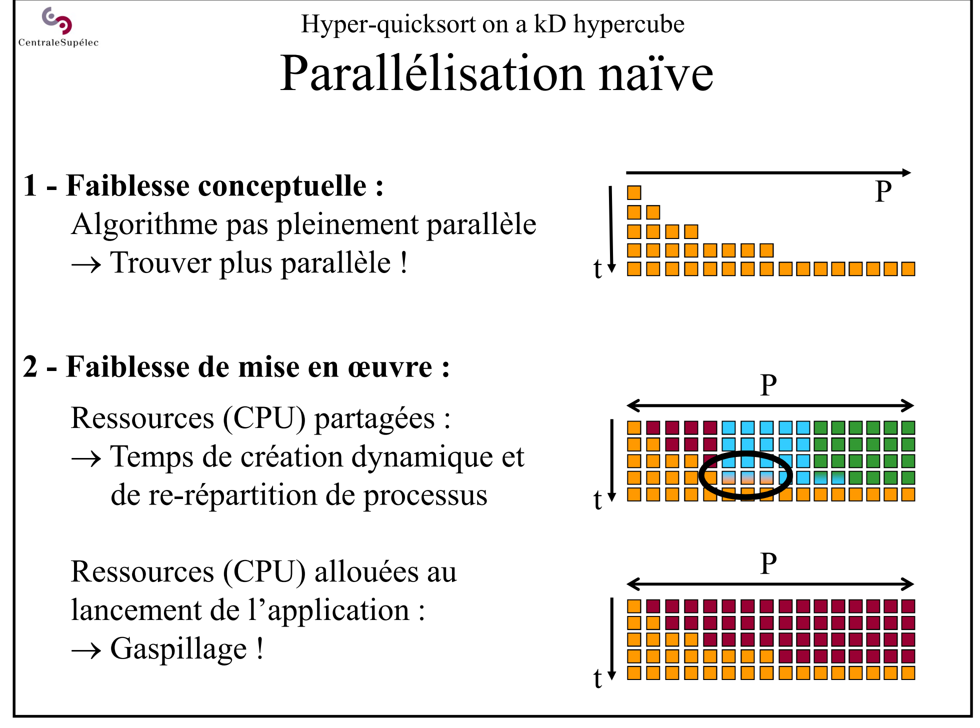
Performances :

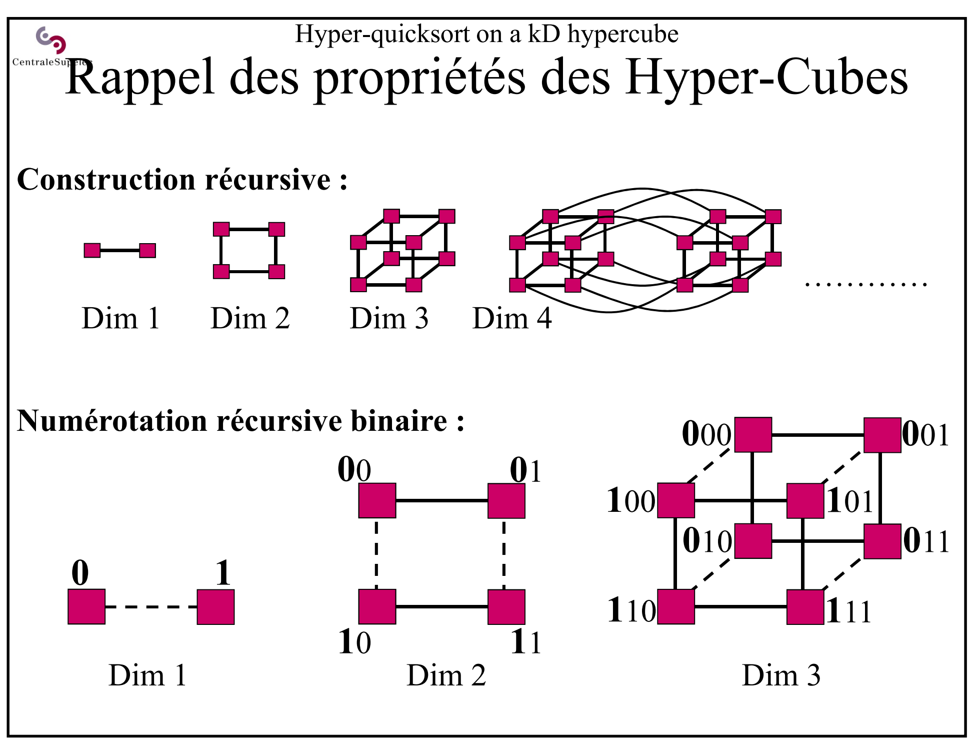
pire cas : $\left(\frac{N.(N-1)}{2}\right).t_{comp} \rightarrow O(N^2)$

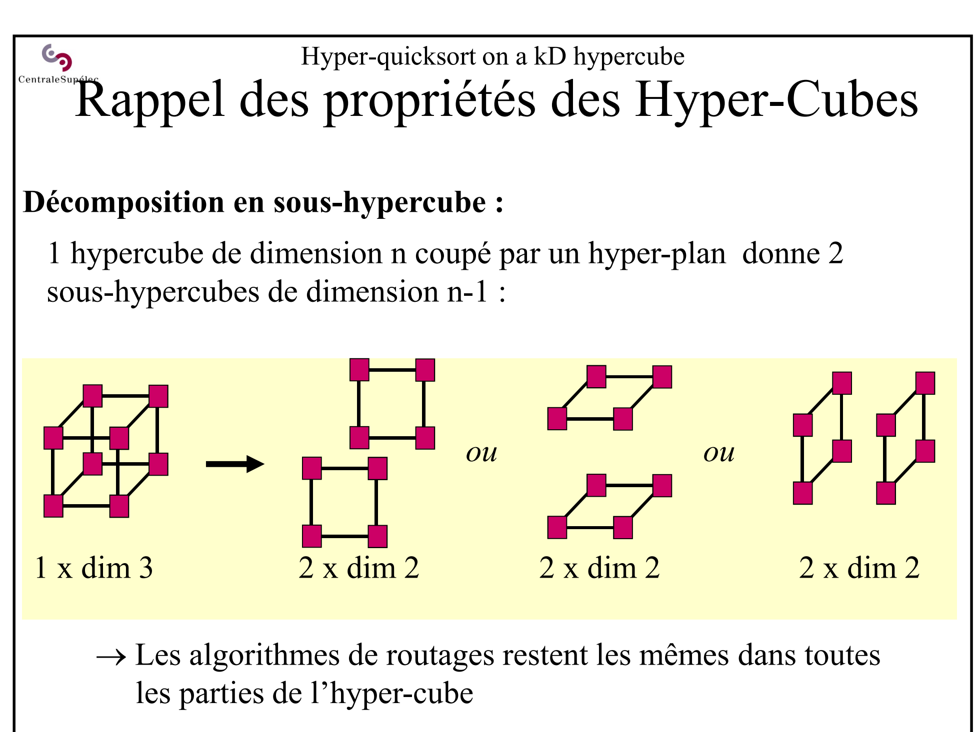
moyen cas (Knuth) : $\rightarrow O(2.N.\log(N))$

meilleur cas : $\left(N.\log_2(N) - 2.N + 1\right)t_{comp} \rightarrow O(N.\log(N))$


Hyper-quicksort on a kD hypercube
Parallélisation naïve
Principe :
p1
<
p21
p22
P0
P0
P1
P0
P2
P1
P3
On crée un nouveau processus et on utilise un nouveau processeur à chaque appel récursif (en réutilisant les anciens).
Simple mais inefficace !


Hyper-quicksort on a kD hypercube
Parallélisation naïve
1 - Faiblesse conceptuelle :
Algorithme pas pleinement parallèle
→ Trouver plus parallèle !
2 - Faiblesse de mise en œuvre :
Ressources (CPU) partagées :
→ Temps de création dynamique et de re-répartition de processus
Ressources (CPU) allouées au lancement de l'application :
→ Gaspillage !
P
t

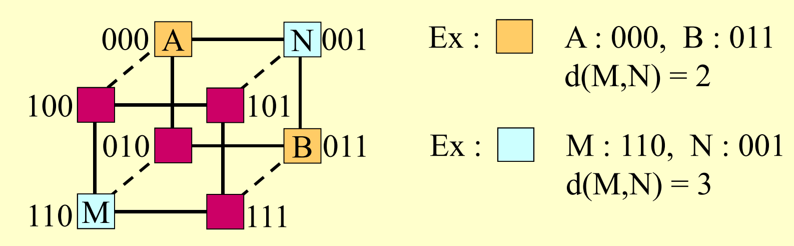

Hyper-quicksort on a kD hypercube
CentraleSupélec
Rappel des propriétés des Hyper-Cubes
Construction récursive :
Dim 1
Dim 2
Dim 3
Dim 4
....
Numérotation récursive binaire :
0
1
Dim 1
00
01
10
11
Dim 2
000
001
100
101
010
011
110
111
Dim 3


Hyper-quicksort on a kD hypercube
CentraleSupélec
Rappel des propriétés des Hyper-Cubes
Décomposition en sous-hypercube :
1 hypercube de dimension n coupé par un hyper-plan donne 2
sous-hypercubes de dimension n-1 :
ou
ou
1 x dim 3
2 x dim 2
2 x dim 2
2 x dim 2
→ Les algorithmes de routages restent les mêmes dans toutes
les parties de l'hyper-cube

Rappel des propriétés des Hyper-Cubes

Distance entre nœuds :

Le nombre de bits différents dans les adresses de deux nœuds donne leur distance (si la numérotation a suivi la construction récursive)



→ Les algorithmes de routages seront à base de calculs simples (et rapides)

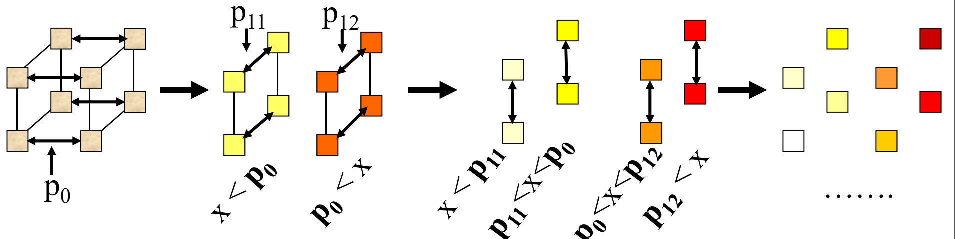
Principe de parallélisation sur Hyper-Cube

Objectif :

- **Tous les processeurs doivent travailler tout le temps !**
- S'inspirer d'un tri séquentiel rapide (*O(N.log(N))*) : quick-sort

→ Trouver un schéma de parallélisation avec comm. optimisées (peu de comm, ou comm locales)

Idée : Quick-sort : algo récursif ↔ topologie récursive : Hyper-cube



CentraleSupélec

Principe de parallélisation sur Hyper-Cube

Init : Chaque nœud charge N/P données en local

Etape 0 :

- Choix de 2^0 pivots pour les 2^0 hypercubes de dimension d-0
- Séparation selon le pivot sur chaque nœud de l'hypercube
- Echange des listes inférieures et supérieures en dimension d-0
- Fusions locales des listes conservées et des listes reçues

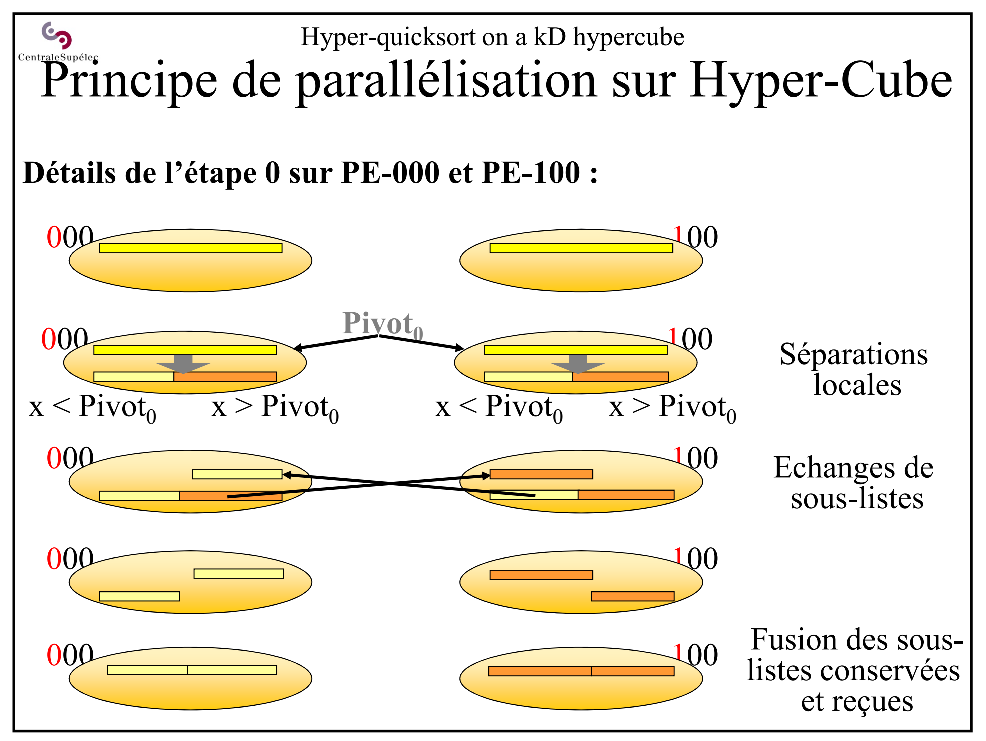
000 001

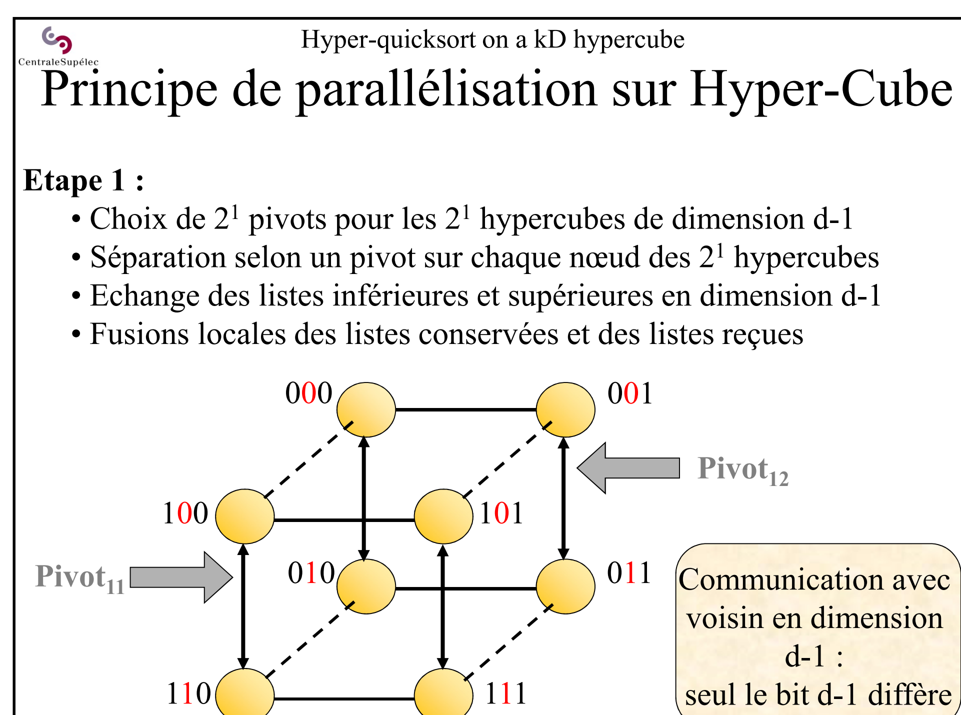
100 101 $Pivot_0$

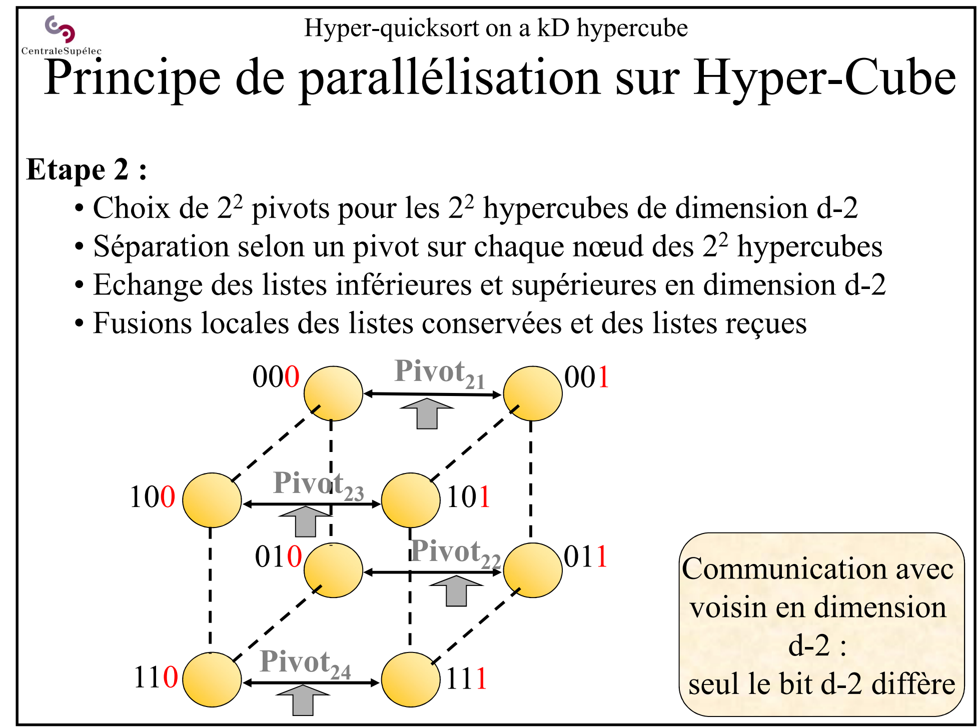
010 011

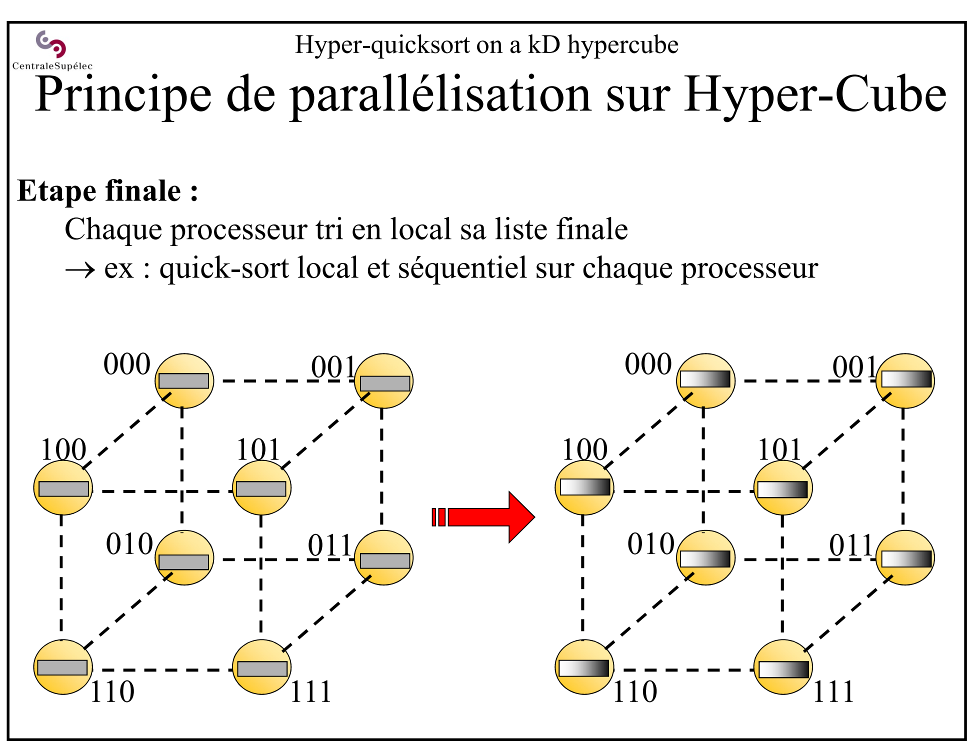
110 111

Communication avec voisin en dimension d-0 : seul le bit d-0 diffère



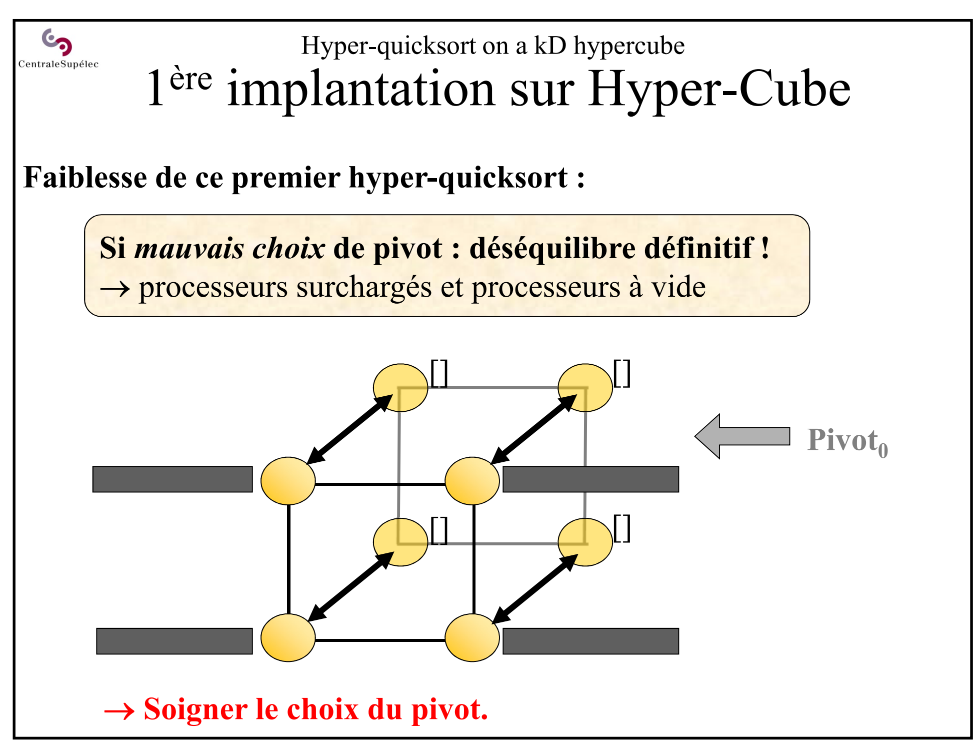

Hyper-quicksort on a kD hypercube
CentraleSupélec
Principe de parallélisation sur Hyper-Cube
Etape 1 :
• Choix de 2¹ pivots pour les 2¹ hypercubes de dimension d-1
• Séparation selon un pivot sur chaque nœud des 2¹ hypercubes
• Echange des listes inférieures et supérieures en dimension d-1
• Fusions locales des listes conservées et des listes reçues
000
001
Pivot12
100
101
Pivot11
010
011
110
111
Communication avec voisin en dimension d-1 :
seul le bit d-1 diffère

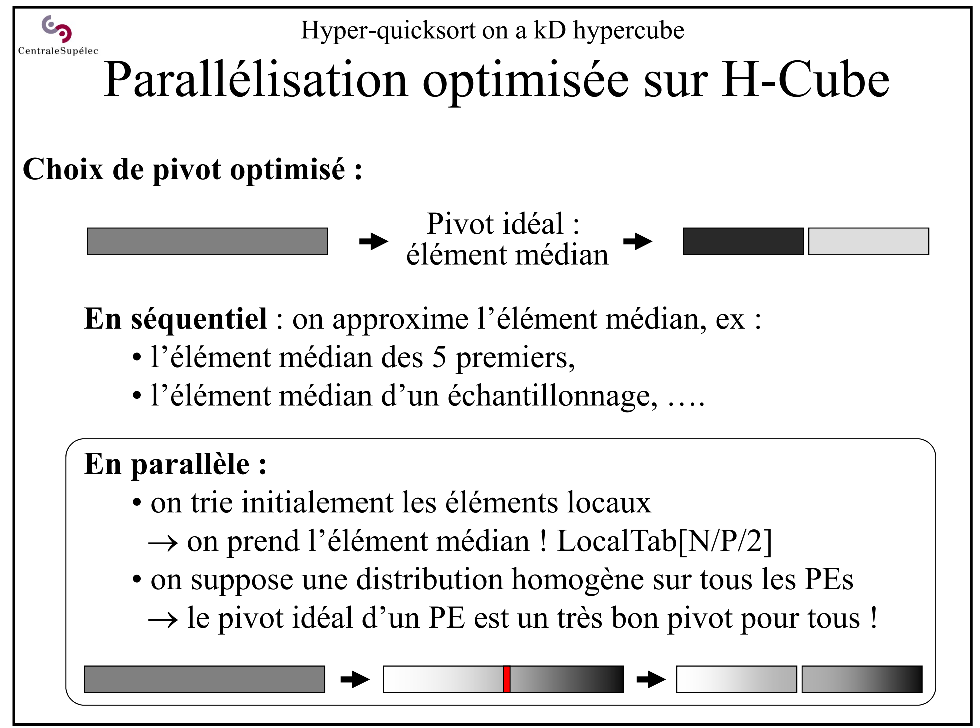

Hyper-quicksort on a kD hypercube
CentraleSupélec
Principe de parallélisation sur Hyper-Cube
Etape 2 :
• Choix de 2² pivots pour les 2² hypercubes de dimension d-2
• Séparation selon un pivot sur chaque nœud des 2² hypercubes
• Echange des listes inférieures et supérieures en dimension d-2
• Fusions locales des listes conservées et des listes reçues
000
Pivot21
001
100
Pivot23
101
010
Pivot22
011
110
Pivot24
111
Communication avec voisin en dimension d-2 :
seul le bit d-2 diffère



$1^{ère}$ implantation sur Hyper-Cube

```
HcubeQuickSort(double *A, int d)
{
 int i;                          /* Compteur de dimension. */
 double *Ainf, *Asup, *Abuf;     /* Tables de données.     */
 double x;                       /* Pivot.                 */
 for (i = d-1; i >= 0; i--) {  /* Pour chaque dimension: */
   x = choix_pivot(me,i);        /* - Partitionnement local*/
   partitioner(A,x,Ainf,Asup);
   if (me & exp2(i) == 0) {      /* - Communications       */
     asend(Asup,me | exp2(i));
     recv(Abuf,me | exp2(i));
     union(Ainf,Abuf,&A);
   } else {
     asend(Ainf,me & ~exp2(i));
     recv(Abuf,me & ~exp2(i));
     union(Abuf,Asup,&A);
   }
 }
 QuickSortSequentiel(A);         /* Tri final des donnes   */
}                                /* locales                */
```

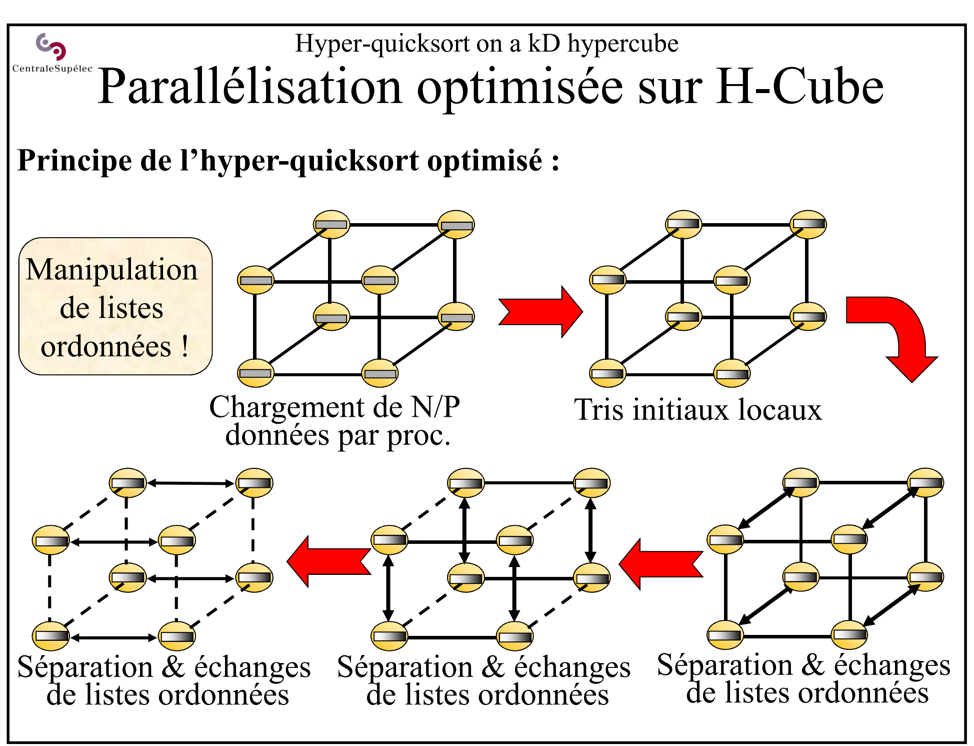

CentraleSupélec
Hyper-quicksort on a kD hypercube
1ère implantation sur Hyper-Cube
Faiblesse de ce premier hyper-quicksort :
Si mauvais choix de pivot : déséquilibre définitif !
→ processeurs surchargés et processeurs à vide
[]
[]
[]
[]
Pivot0
→ Soigner le choix du pivot.


CentraleSupélec
Hyper-quicksort on a kD hypercube
Parallélisation optimisée sur H-Cube
Choix de pivot optimisé :
Pivot idéal :
élément médian
En séquentiel : on approxime l'élément médian, ex :
• l'élément médian des 5 premiers,
• l'élément médian d'un échantillonnage, ….
En parallèle :
• on trie initialement les éléments locaux
→ on prend l'élément médian ! LocalTab[N/P/2]
• on suppose une distribution homogène sur tous les PEs
→ le pivot idéal d'un PE est un très bon pivot pour tous !

Hyper-quicksort on a kD hypercube

Parallélisation optimisée sur H-Cube

Principe de l'hyper-quicksort optimisé :



Hyper-quicksort on a kD hypercube

2ième implantation sur Hyper-Cube

Implantation de l'hyper-quicksort optimisé :

```
HcubeQuickSort(double *A, int d)
{
 int i;                       /* Compteur de dimension.  */
 double *Ainf, *Asup, *Abuf;  /* Tables de données.      */
 double x;                    /* Pivot.                  */
 QuickSortSequentiel(A);      /* Tri initial des donnes */
                              /* locales.                */
 for (i = d-1; i >= 0; i--) { /* Pour chaque dimension: */
   x = choix_pivot(me,i);     /* - Partitionnement local*/
   partitioner(A,x,Ainf,Asup);
   if (me & exp2(i) == 0) {   /* - Communications        */
     asend(Asup,me | exp2(i));
     recv(Abuf,me | exp2(i));
     union_ordonne(Ainf,Abuf,&A);
   } else {
     asend(Ainf,me & ~exp2(i));
     recv(Abuf,me & ~exp2(i));
     union_ordonne(Abuf,Asup,&A);
   }
 }
}
```

Distributed algorithms on
multi-dimensional topologies

End