CentraleSupélec

Mineure CalHau1

# Algorithmique distribuée

**Stéphane Vialle**

CentraleSupélec université PARIS-SACLAY École doctorale Sciences et technologies de l'information et de la communication (STIC) RISEGrid lorraine Grand Est

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

CentraleSupélec

## Algorithmique Distribuée

1. **Produit de matrices sur un anneau de P processeurs**
2. Produit de matrices sur tore de $P = q \times q$ processeurs
3. N-corps sur un anneau de $P$ processeurs
4. Quick-sort sur hypercube de $P = 2^d$ processeurs

CentraleSupélec

Produit de matrices denses sur anneau

## Algorithme distribué

**Problème à résoudre :**

A, B, C : n x n = N éléments

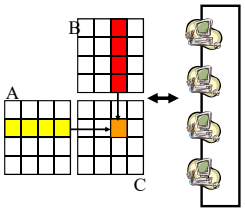
C = A . B $\quad c_{ij} = \sum_{k=1}^{n} (a_{ik} b_{kj})$ $\quad$ O(Nbr d'opérations) = $O(N^{3/2})$

**Comment répartir les données ?**

Pb du **partitionnement**

Calculs déterministes localisés :
- Partitionnement statique

Calculs déterministes NON localisés :
- Duplication ?
- **Circulation** ?



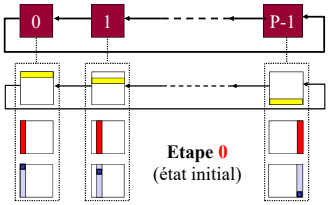
Produit de matrices denses sur anneau

## Algorithme distribué

**Partitionnement sur un anneau de processeurs :**

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes

- **Circulation de A**
- B et C statiques




Topologie

Partitionnement et circulation de A

Partitionnement statique de B

Partitionnement statique de C

**Etape 0**
(état initial)

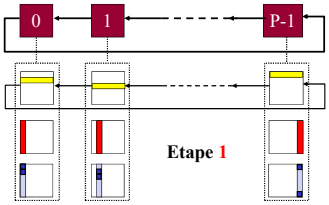
Produit de matrices denses sur anneau

## Algorithme distribué

**Partitionnement sur un anneau de processeurs :**

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes

- **Circulation de A**
- B et C statiques




Topologie

Partitionnement et circulation de A

Partitionnement statique de B

Partitionnement statique de C

**Etape 1**

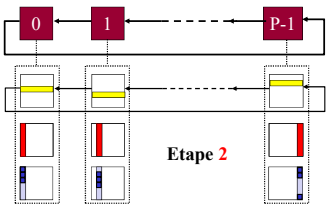
Produit de matrices denses sur anneau

## Algorithme distribué

**Partitionnement sur un anneau de processeurs :**

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes

- **Circulation de A**
- B et C statiques




Topologie

Partitionnement et circulation de A

Partitionnement statique de B

Partitionnement statique de C

**Etape 2**

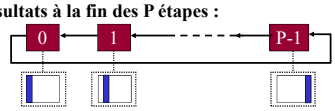
Produit de matrices denses sur anneau

## Algorithme distribué

**Partitionnement sur un anneau de processeurs :**

| | |
|---|---|
| • A partitionnée en blocs de lignes | • **Circulation de A** |
| • B et C partitionnées en blocs de colonnes | • B et C statiques |

**Résultats à la fin des P étapes :**




Topologie

Partitionnement statique de C

**Bilan :**

- Chaque PC a calculé un bloc de colonnes de C
- Les *P* PC ont travaillé en parallèle

→ **Calcul de tous les blocs de colonnes en parallèle**, en *P* étapes

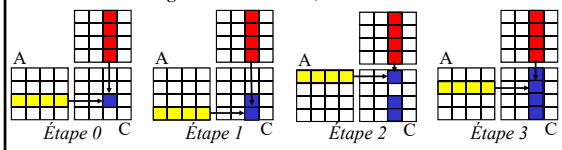
Produit de matrices denses sur anneau

## Algorithme distribué

**Partitionnement sur un anneau de processeurs :**

| | |
|---|---|
| • A partitionnée en blocs de lignes | • **Circulation de A** |
| • B et C partitionnées en blocs de colonnes | • B et C statiques |

**Déroulement de l'algorithme sur PE-2, avec *P* = *4* :**




Produit de matrices denses sur anneau

## Algorithme distribué

**Stratégies d'implantation sur un anneau de P processeurs :**

```
// Sans recouvrement
for (step=0; step<P; step++)
  calcul();
  barrier();   // si besoin
  circulation();
  barrier();   // si besoin
}
```

```
// Avec recouvrement
for (step=0; step<P; step++)
   thread{ calcul(); }
   thread{ circulation(); }
   joinThreads();
   permutBuff();
}
```

- Concevoir l'algorithme avec des barrières de (re)synchronisation potentielles
- Selon le mécanisme de communication utilisé :
  - Implanter des barrières explicites : synchronisation forte

  *ou bien*
  - Se contenter de la synchro des comms : synchronisation relaxée

Produit de matrices denses sur anneau

## Algorithme distribué

**Stratégies d'implantation sur un anneau de P processeurs :**

```
// Sans recouvrement
for (step=0; step<P; step++)
  calcul();
  barrier();    // si besoin
  circulation();
  barrier();    // si besoin
}
```

```
// Avec recouvrement
for (step=0; step<P; step++)
   thread{ calcul(); }
   thread{ circulation(); }
   joinThreads(); // barrier
   permutBuff();
}
```

- Concevoir l'algorithme avec une barrière de (re)synchronisation et implanter la barrière de (re)synchronisation !
- Permuter les buffers de calcul et de communication (quasi-obligatoire)

Produit de matrices denses sur anneau

## Modélisation de perfs sur anneau théorique

**Performances sans recouvrement :**

• Temps séquentiel :

$$T_{seq} = N.(2.\sqrt{N}-1)t_{flop}$$
$$T_{seq} \approx 2.N.\sqrt{N}t_{flop}$$

• Temps parallèle :

$$t_{1-calc} = \frac{\sqrt{N}}{P}.\frac{\sqrt{N}}{P}.(2.\sqrt{N}-1)t_{flop}$$
$$t_{1-calc} \approx 2.\frac{N\sqrt{N}}{P^2}t_{flop}$$

*Modèle de comm :*
- $t_{com}(q) = t_s + q.t_w$
- *toutes les comms en parallèles*

$$t_{1-circ} = t_s + \sqrt{N}.\frac{\sqrt{N}}{P}t_w$$
$$t_{1-circ} \approx \frac{N}{P}t_w$$

$$T^{no}_{par} \approx P.(t_{1-calc} + t_{1-circ})$$
$$T^{no}_{par} \approx 2.\frac{N.\sqrt{N}}{P}t_{flop} + Nt_w$$

• Speed up :

$$S^{no} = \frac{T_{seq}}{T^{no}_{par}} \approx P.\frac{1}{1+\frac{Pt_w}{2.\sqrt{N}t_{flop}}}$$


Produit de matrices denses sur anneau

## Modélisation de perfs sur anneau théorique

**Comparaison calculs-communications (sans recouvrement) :**

• Calculs :

$$T^{calc}_{par} \approx 2.\frac{N^{3/2}}{P}t_{flop}$$
$$P \text{ fixé} \Rightarrow O(T^{calc}_{par}) = O(N^{3/2})$$

• Circulation :

$$T^{circ}_{par} \approx Nt_w$$
$$O(T^{circ}_{par}) = O(N)$$

$$O(T^{calc}_{par}) > O(T^{circ}_{par})$$

Avec $P$ fixé, quand $N \uparrow$ : les calculs deviennent prépondérants
→ le surcoût des comms devient négligeable

Et le speedup devient parfait :

$$S^{no} \approx P.\frac{1}{1+\frac{P.t_w}{2.\sqrt{N}.t_{flop}}} \xrightarrow[N\to\infty]{} P$$

*« Bon » problème pour un TP !*

Produit de matrices denses sur anneau

## Modélisation de perfs sur anneau théorique

**Performances avec recouvrement :**

• Temps séquentiel :

$$T_{seq} = N.(2.\sqrt{N} - 1)t_{flop}$$
$$T_{seq} \approx 2.N.\sqrt{N}\,t_{flop}$$

• Temps parallèle :

$$t_{1-calc} = \frac{\sqrt{N}}{P}.\frac{\sqrt{N}}{P}.(2.\sqrt{N} - 1)t_{flop}$$
$$t_{1-calc} \approx 2.\frac{N\sqrt{N}}{P^2}t_{flop}$$
$$t_{1-circ} = t_s + \sqrt{N}.\frac{\sqrt{N}}{P}t_w$$
$$t_{1-circ} \approx \frac{N}{P}t_w$$
$$T^{ov}_{par} \approx P.\max(t_{1-calc}, t_{1-circ}).\alpha$$

• A $P$ fixé :

$$\exists N_0 / N > N_0 \Rightarrow t_{1-calc} > t_{1-circ}$$

$$T^{ov}_{par} \approx P t_{1-calc}.\alpha$$
$$T^{ov}_{par} \approx 2.\alpha.\frac{N.\sqrt{N}}{P}t_{flop}$$

Produit de matrices denses sur anneau

## Modélisation de perfs sur anneau théorique

**Performances avec recouvrement :**

• Temps séquentiel :

$$T_{seq} = N.(2.\sqrt{N} - 1)t_{flop}$$
$$T_{seq} \approx 2.N.\sqrt{N}\,t_{flop}$$

• Temps parallèle :

$$\exists N_0 / N > N_0 \Rightarrow$$
$$T^{ov}_{par} \approx 2.\alpha.\frac{N.\sqrt{N}}{P}t_{flop}$$

• Speed up :

$$S^{ov} = \frac{T_{seq}}{T^{ov}_{par}} \approx P.\frac{1}{\alpha}$$

Remarque:

$$\alpha = \alpha(N,P) \xrightarrow[N\to\infty]{} 1$$

encore: $S(P) \xrightarrow[N\to\infty]{} P$

Rappel :

$$S^{no} = P.\frac{1}{1 + \frac{Pt_w}{2.\sqrt{N}t_{flop}}}$$

Si le recouvrement est techniquement possible, alors son exploitation doit améliorer les perfs.

## Algorithmique Distribuée

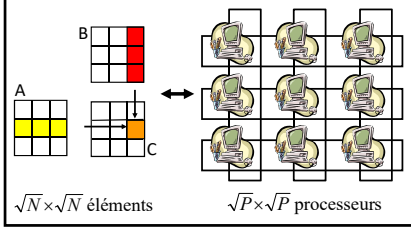
1. Produit de matrices sur un anneau de P processeurs
2. **Produit de matrices sur tore de $P = q \times q$ processeurs**
3. N-corps sur un anneau de $P$ processeurs
4. Quick-sort sur hypercube de $P = 2^d$ processeurs

Produit de matrices sur tore de P proc

## Algorithme distribué

A, B, C : n × n = N éléments

C = A . B $c_{ij} = \sum_{k=1}^{n} (a_{ik} b_{kj})$ O(Nbr d'opérations) = O(N^3/2)



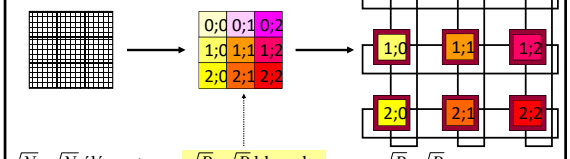

Quels partitionnement et circulation ?

$\sqrt{N} \times \sqrt{N}$ éléments $\sqrt{P} \times \sqrt{P}$ processeurs

Produit de matrices sur tore de P proc

## Algorithme distribué

**Partitionnement 2D unique**
(opérandes et résultats au même format)




$\sqrt{N} \times \sqrt{N}$ éléments

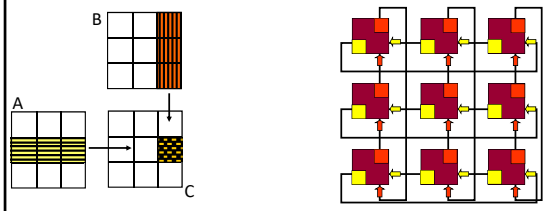
$\sqrt{P} \times \sqrt{P}$ blocs de $\frac{\sqrt{N}}{\sqrt{P}} \times \frac{\sqrt{N}}{\sqrt{P}}$ éléments

$\sqrt{P} \times \sqrt{P}$ processeurs

Produit de matrices sur tore de P proc

## Algorithme distribué

**Schémas de circulation :**
Circulation des blocs de **A en ligne** ⇦
Circulation des blocs de **B en colonne** ⇧




→ Chaque processeur voit passer toutes les données utiles à ses calculs

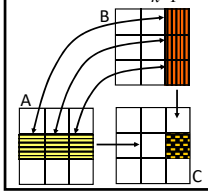
Produit de matrices sur tore de P proc

## Algorithme distribué

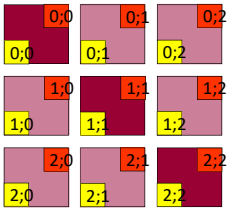
**Problème de séquencement :**

besoin **simultané** des blocs $A_{i,k}$ et $B_{k,j}$

C = A . B $\quad c_{ij} = \sum_{k=1}^{n} (a_{ik} . b_{kj})$




Seuls les processeurs de la première diagonale sont satisfaits !




→ Modifier le partitionnement initial

Produit de matrices sur tore de P proc

## Algorithme distribué

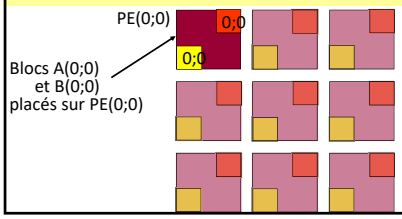
**Démarche :**

On conserve le principe du partitionnement en 2D

On conserve le schéma de circulation (A en ligne, B en colonne)

Mais on cherche un partitionnement initial adapté …

**… on place les données de PE(0;0), et on *propage les contraintes***

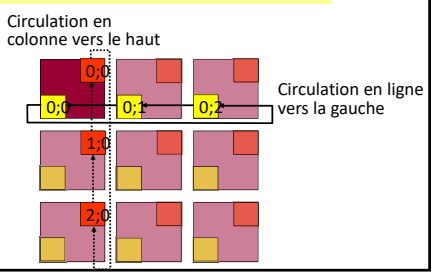



Produit de matrices sur tore de P proc

## Algorithme distribué

Les schémas de circulation imposent :

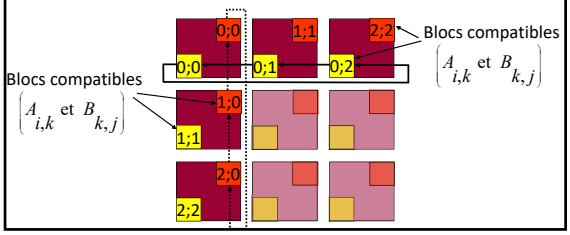
- les blocs de A sur la première ligne de processeurs
- les blocs de B sur la première colonne de processeurs



Produit de matrices sur tore de P proc

## Algorithme distribué

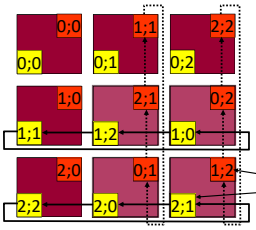
L'algorithme de produit matriciel impose :
- les blocs de B sur la première ligne de processeurs
- les blocs de A sur la première colonne de processeurs




Produit de matrices sur tore de P proc

## Algorithme distribué

Les schémas de circulation imposent :
- les blocs de A sur les lignes de processeurs restantes
- les blocs de B sur les colonnes de processeurs restantes




On vérifie si les nouveaux couples sont compatibles…

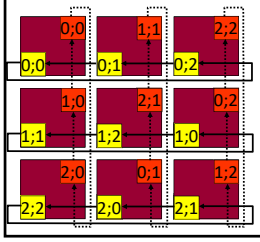
… Oui! On a toujours :

$\left(A_{i,k} \text{ et } B_{k,j}\right)$

Produit de matrices sur tore de P proc

## Algorithme distribué

**Finalement :**
- Partitionnement fixé sur tous les processeurs
- Tous les couples de blocs sont compatibles à la première étape
- Tous les couples de blocs sont compatibles après chaque circulation




Colonne 0 décalée de 0 cran vers le haut
Colonne 1 décalée de 1 cran vers le haut
Colonne 2 décalée de 2 cran vers le haut

Généralisation …

Puis Calcul complet en $\sqrt{P}$ étapes

Produit de matrices sur tore de P proc

## Algorithme distribué

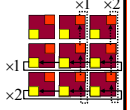
**Algorithme de Canon :**

Produit de matrices de $\sqrt{N}\times\sqrt{N}$ éléments sur un tore de $\sqrt{P}\times\sqrt{P}$ procs.

1 – Partitionnement des matrices en $\sqrt{P}\times\sqrt{P}$ blocs de $\frac{\sqrt{N}}{\sqrt{P}}\times\frac{\sqrt{N}}{\sqrt{P}}$ éléments


2 – Pré-décalages initiaux des blocs sur le tore de proc :

- ligne $i$ ($[0;\sqrt{P}-1]$) : décalage de $i$ crans ←,
- colonne $j$ ($[0;\sqrt{P}-1]$) : décalage de $j$ crans ↑.




3 - $\sqrt{P}$ étapes : $\sqrt{P}$ × { calculs : somme partielle de C(i,j) ;
circulations : A(i,j) 1 cran ←, et B(i,j) 1 cran ↑ ; }

4 – Dernière circulation ou post décalages de A et B si nécessaire.

Produit de matrices sur tore de P proc

## Algorithme distribué

**Pseudo-Code de l'algorithme de Canon sur chaque processeur :**

```
main()
{
 partitionnement();
 predecalage();
 barriere();
 for (step=0; step < SQRT_P-1; step++) {
    multiplication_locale();
    circulation_ligne_colone();
    barriere();
 }
 multiplication_locale();
 postdecalage();
}
```

Post-décalage :

- nécessaire si on souhaite réutiliser les matrices pour des opérations variées (**+**, **x**, …)

Produit de matrices sur tore de P proc

## Modélisation des performances

**1 - Performances avec messages bloquants et sans recouvrement :**

*pré-décalages; $\sqrt{P}$ étapes de calcul et $\sqrt{P}$-1 circulations; postdécalage*

Hyp : Toutes les communications peuvent se faire en parallèle
Modélisation en « $t_s$+Q.$t_w$ »

- Pré et Post décalages :

- dans un vrai tore unidirectionnel :

$$t_{pre-decal} \approx 2.\left(\sqrt{P}-1\right).\frac{N}{P}t_w$$
$$t_{post-decal} \approx 2.\left(\sqrt{P}-1\right).\frac{N}{P}t_w$$

- dans un cluster :

$$t_{pre-decal} \approx 2.\frac{N}{P}t_w$$
$$t_{post-decal} \approx 2.\frac{N}{P}t_w$$

Produit de matrices sur tore de P proc

## Modélisation des performances

**1 - Performances avec messages bloquants et sans recouvrement :**

*pré-décalages; $\sqrt{P}$ étapes de calcul et $\sqrt{P}$-1 circulations; postdécalage*

- Pré et post décalage : $t_{pre-decal} \approx 2.\frac{N}{P} t_w$ et : $t_{post-decal} \approx 2.\frac{N}{P} t_w$
- Circulation : $t_{1-circ} \approx 2.\frac{N}{P} t_w$
- Calcul : $t_{1-calc} \approx 2.\frac{N^{3/2}}{P^{3/2}} t_{flop}$

Donc sur un cluster, avec des msgs bloquants : $T_{par}(P) \approx 2.\frac{N^{3/2}}{P} t_{flop} + 2\left(\sqrt{P}+1\right)\frac{N}{P} t_w$

Produit de matrices sur tore de P proc

## Modélisation des performances

**1 - Performances avec messages bloquants et sans recouvrement :**

*pré-décalages; $\sqrt{P}$ étapes de calcul et $\sqrt{P}$-1 circulations; postdécalage*

- Temps parallèle sur cluster : $T_{par}(P) \approx 2.\frac{N^{3/2}}{P} t_{flop} + 2\left(\sqrt{P}+1\right)\frac{N}{P} t_w$
- Temps séquentiel : $T_{seq} \approx 2.N^{3/2} t_{flop}$
- Speed up sur cluster avec msgs bloquants : $S(P) \approx P.\frac{1}{1+\left(\sqrt{P}+1\right) t_w / \sqrt{N}\, t_{flop}}$

Donc : $S(P) \xrightarrow[N \to \infty]{} P$ On retrouve l'asymptote idéale

Produit de matrices sur tore de P proc

## Modélisation des performances

**2 - Performances avec msgs non bloquants et sans recouvrement :**

*pré-décalages; $\sqrt{P}$ étapes de calcul et $\sqrt{P}$-1 circulations; postdécalage*

Hyp : Toutes les communications peuvent se faire en parallèle
Modélisation en « $t_s$+Q.$t_w$ »

- Pré et Post décalages :
  - dans un vrai tore unidirectionnel : $t_{pre-decal} \approx \left(\sqrt{P}-1\right)\frac{N}{P} t_w$ ; $t_{post-decal} \approx \left(\sqrt{P}-1\right)\frac{N}{P} t_w$
  - <u>dans un cluster :</u> $t_{pre-decal} \approx \frac{N}{P} t_w$ ; $t_{post-decal} \approx \frac{N}{P} t_w$

Produit de matrices sur tore de P proc

## Modélisation des performances

**2 - Performances avec msgs non bloquants et sans recouvrement :**

*pré-décalages; $\sqrt{P}$ étapes de calcul et $\sqrt{P}$-1 circulations; postdécalage*

- Pré et post décalage : $t_{pre-decal} \approx \frac{N}{P} t_w$ et : $t_{post-decal} \approx \frac{N}{P} t_w$
- Circulation : $t_{1-circ} \approx \frac{N}{P} t_w$
- Calcul : $t_{1-calc} \approx 2.\frac{N^{3/2}}{P^{3/2}} t_{flop}$

Donc sur un cluster, avec des msgs non-bloquants : $T_{par}(P) \approx 2.\frac{N^{3/2}}{P} t_{flop} + \left(\sqrt{P}+1\right)\frac{N}{P} t_w$

Produit de matrices sur tore de P proc

## Modélisation des performances

**2 - Performances avec msgs non bloquants et sans recouvrement :**

*pré-décalages; $\sqrt{P}$ étapes de calcul et $\sqrt{P}$-1 circulations; postdécalage*

- Temps parallèle sur cluster : $T_{par}(P) \approx 2.\frac{N^{3/2}}{P} t_{flop} + \left(\sqrt{P}+1\right)\frac{N}{P} t_w$
- Temps séquentiel : $T_{seq} \approx 2.N^{3/2} t_{flop}$
- Speed up sur cluster avec msgs non-bloquants : $S(P) \approx P.\frac{1}{1+\left(\sqrt{P}+1\right)t_w / 2.\sqrt{N}\, t_{flop}}$

Donc : $S(P) \xrightarrow[N \to \infty]{} P$ On retrouve l'asymptote idéale

Produit de matrices sur tore de P proc

## Modélisation des performances

**3 - Performances avec msgs non bloquants et avec recouvrement :**

*pré-déc; $\sqrt{P}$-1 calculs et $\sqrt{P}$-1 circulations; 1 calcul et postdec*

Hyp : Toutes les communications peuvent se faire en parallèle
Modélisation en « $t_s$+Q.$t_w$ »

- Pré et Post décalages :
  - dans un vrai tore unidirectionnel : $t_{pre-decal} \approx \left(\sqrt{P}-1\right).\frac{N}{P} t_w$ ; $t_{post-decal} \approx \left(\sqrt{P}-1\right).\frac{N}{P} t_w$
  - <u>dans un cluster :</u> $t_{pre-decal} \approx \frac{N}{P} t_w$ ; $t_{post-decal} \approx \frac{N}{P} t_w$

Produit de matrices sur tore de P proc

## Modélisation des performances

**3 - Performances avec msgs non bloquants et avec recouvrement :**

*pré-déc; $\sqrt{P}$-1 calculs et $\sqrt{P}$-1 circulations; 1 calcul et postdec*

- Pré et post décalage : $t_{pre-decal} \approx \frac{N}{P} t_w$ et : $t_{post-decal} \approx \frac{N}{P} t_w$
- Circulation : $t_{1-circ} \approx \frac{N}{P} t_w$
- Calcul : $t_{1-calc} \approx 2.\frac{N^{3/2}}{P^{3/2}} t_{flop}$

$$T_{par}(P) \approx t_{pre-dec} + \left(\sqrt{P}-1\right).\alpha.\max\left(t_{1-calc}; t_{1-circ}\right) + \alpha.\max(t_{1-calc}; t_{post-dec})$$

$$T_{par}(P) \approx \frac{N}{P} t_w + \sqrt{P}.\alpha.\max\left(2.\frac{N^{3/2}}{P^{3/2}} t_{flop}; \frac{N}{P} t_w\right)$$

Produit de matrices sur tore de P proc

## Modélisation des performances

**3 - Performances avec msgs non bloquants et avec recouvrement :**

*pré-déc; $\sqrt{P}$-1 calculs et $\sqrt{P}$-1 circulations; 1 calcul et postdec*

Rappel : Calculs prépondérants, donc :

- Temps parallèle sur cluster : $\exists N_0 / N > N_0 \Rightarrow T_{par} \approx 2.\alpha.\frac{N^{3/2}}{P} t_{flop} + \frac{N}{P} t_w$
- Temps séquentiel : $T_{seq} \approx 2.N^{3/2} t_{flop}$
- Speed up sur cluster avec msgs non-bloquants et recouvrement : $\exists N_0 / N > N_0 \Rightarrow S(P) \approx P.\frac{1}{\alpha + t_w / 2\sqrt{N} t_{flop}}$

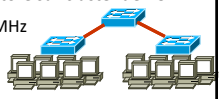
$\text{Hyp} : \alpha = \alpha(N,P) \xrightarrow[N \to \infty]{} 1$ Donc : $S(P) \xrightarrow[N \to \infty]{} P$ On retrouve l'asymptote idéale

Produit de matrices sur tore de P proc

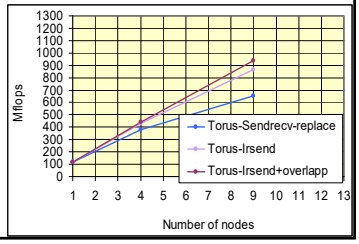
## Modélisation des performances

**Expérimentation du produit de matrices en tore sur cluster de PC**

- Cluster « Beowulf »
- 3 implants. en MPI (voir chap MPI)
- 12 x P3 - 700MHz
- **Fast Ethernet**
- 3 switchs



- Performances correctes
- L'envoi de messages non-bloquant est efficace.
- Le recouvrement est encore plus efficace, il gomme le passage à 3 switchs.



Produit de matrices sur tore de P proc

## Modélisation des performances

**Comparaison Anneau/Tore pour le produit de matrices :**

| | Anneau | Tore |
|---|---|---|
| Bloquant | $T_{par}(P) \approx 2.\frac{N^{3/2}}{P} t_{flop} + N t_w$ | $T_{par}(P) \approx 2.\frac{N^{3/2}}{P} t_{flop} + 2\left(\sqrt{P}+1\right)\frac{N}{P} t_w$ |
| Non-bloquant Recouvrement | $T_{par} \approx 2.\alpha.\frac{N^{3/2}}{P} t_{flop}$ | $T_{par} \approx 2.\alpha.\frac{N^{3/2}}{P} t_{flop} + \frac{N}{P} t_w$ |

**Bilan :**

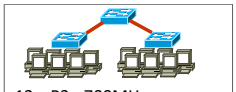
bloquant → P = 4 : tore plus lent, P = 9 : tore plus rapide
Le tore est plus rapide pour P > 7

non bloquant → L'anneau est toujours plus rapide !
Même temps de boucle calcul-circulation, mais prédécalage à faire en plus sur le tore!

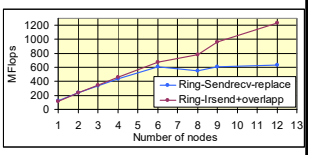
Produit de matrices sur tore de P proc

## Modélisation des performances

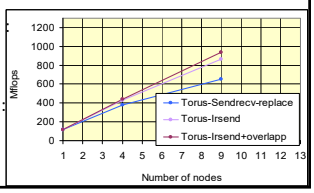
**Tests sur cluster « Beowulf » :**



12 x P3 - 700MHz
**Fast Ethernet & 3 switchs**




- non-bloquant + recouvrement : (presque) pas de différence
  → normal !
- bloquant (sans recouvrement) :
  P= 4 : tore plus lent
  P= 9 : tore plus rapide
  → normal !




## Algorithmique Distribuée

1. Produit de matrices sur un anneau de P processeurs
2. Produit de matrices sur tore de $P = q \times q$ processeurs
3. **N-corps sur un anneau de *P* processeurs**
4. Quick-sort sur hypercube de $P = 2^d$ processeurs

Problème des N-Corps sur anneau de processeurs

## Définition

**Calcul des trajectoires de N *corps* en attraction mutuelle:**

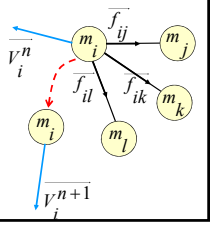
- N molécules en interaction électrostatique
- N planètes en interaction gravitationnelle
- …

$$\text{A } t_n : N \text{ triplets } (\vec{Pos}_i^n, \vec{V}_i^n, \vec{a}_i^n)$$

$$\text{Avec : } \vec{a}_i^n = \sum_{j \neq i} \left( \frac{G.m_j}{(d_{ij}^n)^2} . \frac{\vec{Pos}_j^n - \vec{Pos}_i^n}{d_{ij}^n} \right)$$

$$\text{D'où : } \vec{V}_i^{n+1} = \vec{V}_i^n + \vec{a}_i^n . \delta t$$

$$\vec{Pos}_i^{n+1} = \vec{Pos}_i^n + \vec{V}_i^n . \delta t + \vec{a}_i^n . (\delta t)^2$$




Problème des N-Corps sur anneau de processeurs

## Définition

**Dans sa forme initiale :**

- Tous les corps interagissent avec tous les autres
  → *Long range data interaction*
  → *O(N²)* : plus complexe qu'un produit de matrices (*O(N^3/2)*)

**Dans sa forme usuelle :**

- On ignore les influences trop lointaines (trop faibles) : on se limite à un *rayon d'influence*

→ perte de précision, mais : $\alpha.N^2.t_{1\text{-influence}} \rightarrow \alpha.n^2.t_{1\text{-influence}}$
avec : n << N

**Dans tous les cas :**

Le problème des N-corps reste un problème très lourd (*O(N²)*) qui sature très vite les ordinateurs → besoin de parallélisation

Problème des N-Corps sur anneau de processeurs

## Algorithme distribué

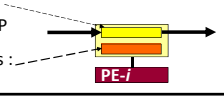
**Principe de parallélisation :**

- Chaque processeur « gère » l'évolution de N/P corps.
- On fait circuler les coordonnées courante des N corps sur l'anneau.
- A chaque étape : chaque processeur accumule l'influence des N/P corps dont il reçoit les coordonnées sur ses N/P corps.
- Après P étapes, chaque processeur calcule les nouvelles coordonnées de ses N/P corps.

**Réalisation :** On utilise une partition fixe et une partition circulante

Coord. courantes des N corps : partition circulante par bloc de N/P

Influences subies par les N/P corps : partition fixe par bloc de N/P



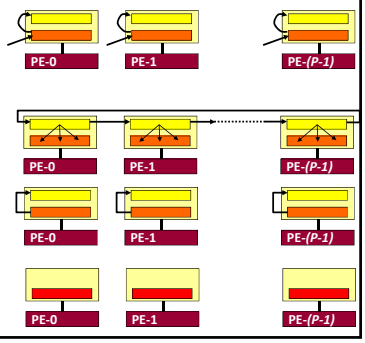
Problème des N-Corps sur anneau de processeurs

## Algorithme distribué

- Partitionnement initial
  - N/P corps par PE
  - duplication locale
- Calculs partiels
- **Circulation**
- Calculs partiels
- Permutation des buffers
- Sauvegarde des derniers calculs

P-1 étapes

Cycles




Problème des N-Corps sur anneau de processeurs

## Modélisation

**Performances avec msgs bloquants (sans recouvrement) :**

Hyp : Toutes les communications peuvent se faire en parallèle ;
Modélisation en « $t_s+Q.t_w$ »

- Temps séquentiel : $T_{seq}(N) \approx Cycle.N^2 k t_{flop}$
- Temps parallèle :

$$t_{1-calc} \approx (\frac{N}{P})^2 k t_{flop}$$

$$t_{1-circ} \approx \frac{N}{P} t_w$$

$$T_{par}(P,N) = Cycle.(P t_{1-calcul} + (P-1) t_{1-circ})$$

Donc, avec des msgs bloquants : $T_{par}(P,N) \approx Cycle.(\frac{N^2}{P} k t_{flop} + \frac{P-1}{P} N t_w)$

Problème des N-Corps sur anneau de processeurs

## Modélisation

**Performances avec msgs bloquants (sans recouvrement) :**

- Accélération :

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

$$\text{Donc} : S(P) \xrightarrow[N \to \infty]{} P$$

On retrouve l'asymptote idéale

- Rapport Calculs/Communications :

$$\frac{Calculs}{Comms} = \frac{O(N^2)}{O(N)} = O(N)$$

Très bon rapport ! Encore meilleur que celui du produit de matrices

En augmentant N les performances finiront toujours par être bonnes !

Problème des N-Corps sur anneau de processeurs

## Modélisation

**Performances avec msgs non-bloquants et avec recouvrement :**

Hyp : Toutes les communications peuvent se faire en parallèle ;
Modélisation en « $t_s$+Q.$t_w$ »

- Temps séquentiel : $T_{seq}(N) \approx Cycle.N^2.k.t_{flop}$
- Temps parallèle :

$$T_{par}(P,N) = Cycle.\left(\alpha.(P-1).Max(t_{1-calcul}, t_{1-circ}) + t_{1-calcul}\right)$$

$$\exists N_0 / N > N_0 \Rightarrow T_{par}(P,N) \approx Cycle.(\alpha.(P-1)+1).\frac{N^2}{P^2}.k.t_{flop}$$

Donc, avec des msgs non-bloquants et du recouvrement : $\exists N_0 / N > N_0 \Rightarrow T_{par}(P,N) \approx Cycle.\alpha.\frac{N^2}{P}.k.t_{flop}$

Problème des N-Corps sur anneau de processeurs

## Modélisation

**Performances avec msgs non-bloquants et avec recouvrement :**

- Accélération :

$$S(P,N) \approx \frac{Cycle.N^2.k.t_{flop}}{Cycle.\frac{N^2}{P}.k.t_{flop}}$$

$$S(P,N) \approx P.\frac{1}{\alpha}$$

$$\text{Hyp} : \alpha = \alpha(N,P) \xrightarrow[N \to \infty]{} 1 \qquad \text{Donc} : S(P) \xrightarrow[N \to \infty]{} P$$

On retrouve (encore) l'asymptote idéale

## Algorithmique Distribuée

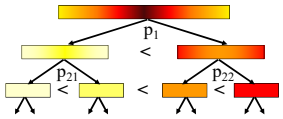
1. Produit de matrices sur un anneau de P processeurs
2. Produit de matrices sur tore de $P = q \times q$ processeurs
3. N-corps sur un anneau de $P$ processeurs
4. **Quick-sort sur hypercube de $P = 2^d$ processeurs**

Quick-sort sur hypercube de processeurs

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

Quick-sort sur hypercube de processeurs

## Algorithme séquentiel

**Implantation séquentielle :**

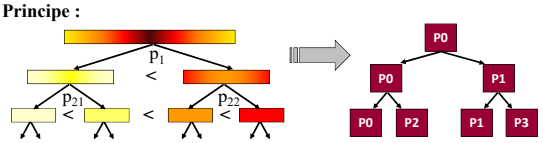
```
void QuickSort(double *A, int q, int r)
{
 int s, i;
 double pivot;

 if (q < r) {          /* Partitionement   */
   pivot = A[q];       /* Choix du pivot … */
   s = q;
   for (i = q+1; i <= r; i++) {
      if (A[i] <= pivot) {
        s = s+1;
        exchange(A,s,i);
      }
   }
   exchange(A,q,s);
   QuickSort(A,q,s-1); /* Appels recursifs */
   QuickSort(A,s+1,r);
 }
}
```

Quick-sort sur hypercube de processeurs

## Parallélisation naïve

**Principe :**




On crée un nouveau processus et on utilise un nouveau processeur à chaque appel récursif (en réutilisant les anciens).

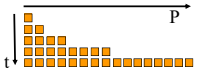
**Bilan :**

- Simple, facile à implanter en mémoire partagée
- Mais inefficace !

Quick-sort sur hypercube de processeurs

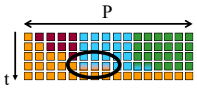
## Parallélisation naïve

**1 - Faiblesse conceptuelle :**
Algorithme pas pleinement parallèle
→ Trouver plus parallèle !

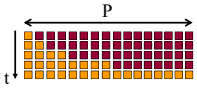



**2 - Faiblesse de mise en œuvre :**

Ressources (CPU) partagées :
→ Temps de création dynamique et de re-répartition de processus



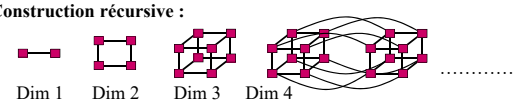

Ressources (CPU) allouées au lancement de l'application :
→ Gaspillage !



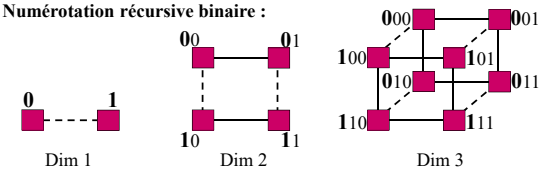

Quick-sort sur hypercube de processeurs

## Propriétés des hypercubes

• **Construction récursive :**




• **Numérotation récursive binaire :**

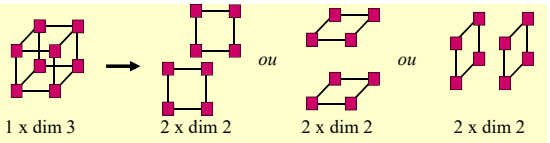



Quick-sort sur hypercube de processeurs

## Propriétés des hypercubes

• **Décomposition en sous-hypercube :**

1 hypercube de dimension n coupé par un hyper-plan donne 2 sous-hypercubes de dimension n-1 :



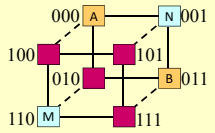

→ Les algorithmes de routages restent les mêmes dans toutes les parties de l'hyper-cube

Quick-sort sur hypercube de processeurs

## Propriétés des hypercubes

• **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)




Ex : A : 000, B : 011
d(M,N) = 2

Ex : M : 110, N : 001
d(M,N) = 3

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

Quick-sort sur hypercube de processeurs

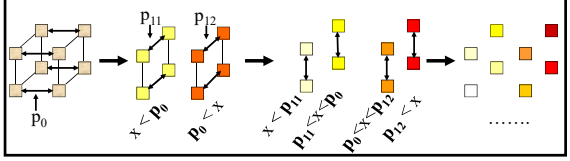
## Algorithme sur hypercube

**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



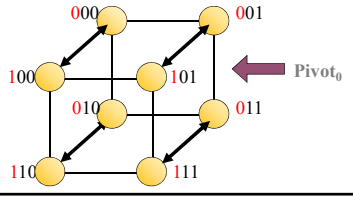

Quick-sort sur hypercube de processeurs

## Algorithme sur hypercube

**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

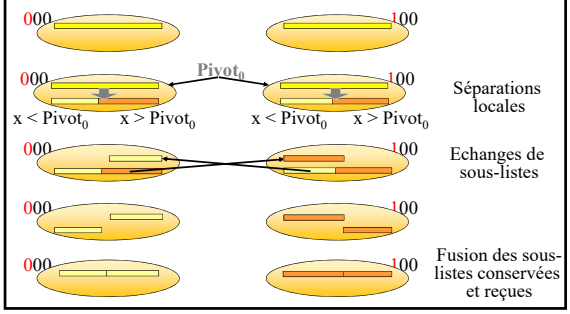



Communication avec voisin en dimension d-0 : seul le bit d-0 diffère

Quick-sort sur hypercube de processeurs

## Algorithme sur hypercube

**Détails de l'étape 0 sur PE-000 et PE-100 :**

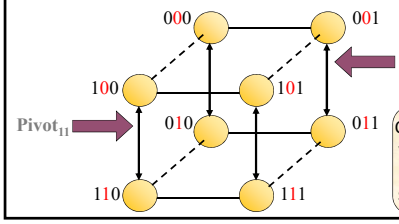



Quick-sort sur hypercube de processeurs

## Algorithme sur hypercube

**Etape 1 :**

- Choix de $2^1$ pivots pour les $2^1$ hypercubes de dimension d-1
- Séparation selon un pivot sur chaque nœud des $2^1$ hypercubes
- Echange des listes inférieures et supérieures en dimension d-1
- Fusions locales des listes conservées et des listes reçues



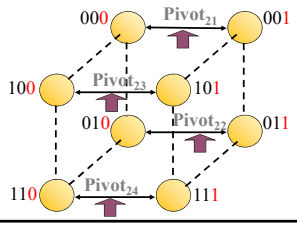

Communication avec voisin en dimension d-1 : seul le bit d-1 diffère

Quick-sort sur hypercube de processeurs

## Algorithme sur hypercube

**Etape 2 :**

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




Communication avec voisin en dimension d-2 : seul le bit d-2 diffère

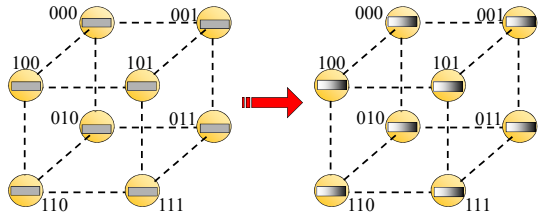
Quick-sort sur hypercube de processeurs

## Algorithme sur hypercube

**Etape finale :**

Chaque processeur tri en local sa liste finale

→ ex : quick-sort local et séquentiel sur chaque processeur




Quick-sort sur hypercube de processeurs

## 1ère implantation

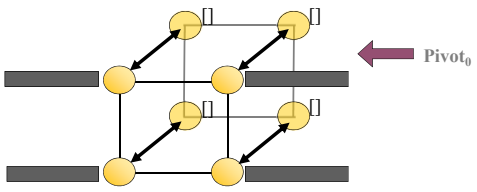
```
HcubeQuickSort(double *A, int d)
{
 int i;                          /* Compteur de dimension. */
 double *Ainf, *Asup, *Abuf;     /* Tables de données.     */
 double x;                       /* Pivot.                 */
 for (i = d-1; i >= 0; i--) {    /* Pour chaque dimension: */
   x = choix_pivot(me,i);        /* - 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                */
```

Quick-sort sur hypercube de processeurs

## 1ère implantation

**Faiblesse de ce premier hyper-quicksort :**

**Si *mauvais choix* de pivot : déséquilibre définitif !**
→ processeurs surchargés et processeurs à vide

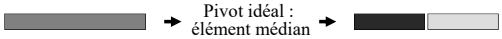



**→ Soigner le choix du pivot.**

Quick-sort sur hypercube de processeurs

## Algorithme optimisé

**Choix de pivot optimisé :**



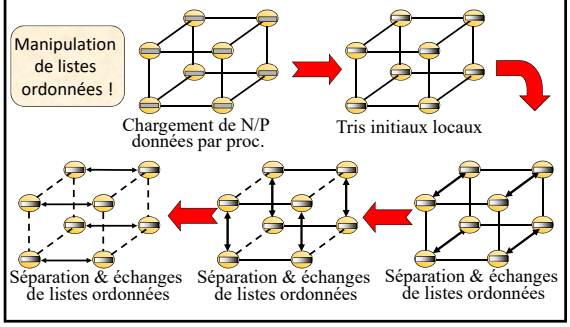

**En séquentiel** : on approxime l'élément médian, ex :
- l'élément médian des 5 premiers,
- l'élément médian d'un échantillonnage, ….

**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 !

Quick-sort sur hypercube de processeurs

## Algorithme optimisé

**Principe de l'hyper-quicksort optimisé :**




Quick-sort sur hypercube de processeurs

## 2ème implantation

**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);
   }
 }
}
```

Quick-sort sur hypercube de processeurs

## Modélisation simple des perfs.

*Etapes de l'algorithme final :*

1 - Tris initiaux-locaux

2 - Calcul de pivot : A[N/(2P)]

3 - Diffusion d'un pivot à un hypercube de dimension d-i

4 - Partitionnements locaux : dichotomie selon pivot

5 - Echange de listes ordonnées

6 - Fusion de listes ordonnées

$\log_2(P)$ itérations

*Complexité de chaque étape :*

$O(\frac{N}{P}.\log(\frac{N}{P}))$

$O(1)$

$O(d-i) = O(\log(P)-i)$

$O(\log(\frac{N}{P}))$

$O(\frac{N}{2P})$

$O(\frac{N}{P})$

Quick-sort sur hypercube de processeurs

## Modélisation simple des perfs.

$$O(T_{par}(N,P)) = O(\frac{N}{P}.\log(\frac{N}{P})) + O(\log(P)) + O\left(\sum_{i=0}^{\log_2(P)-1} (\log_2(P)-i)\right) + O(\log(P).\log(\frac{N}{P})) + O(\log(P).\frac{N}{2.P}) + O(\log(P).\frac{N}{P})$$

$$O(T_{par}(N,P)) = O(\frac{N}{P}.\log(\frac{N}{P})) + O(\log(P)) + O(\frac{\log^2(P)}{2}) + O(\log(P).\log(\frac{N}{P})) + O(\log(P).\frac{N}{P})$$

Quick-sort sur hypercube de processeurs

## Modélisation simple des perfs.

$$O(T_{par}(N,P)) = O(\frac{N}{P}.\log(\frac{N}{P})) + O(\log(P)) + O(\frac{\log^2(P)}{2}) + O(\log(P).\log(\frac{N}{P})) + O(\log(P).\frac{N}{P})$$

**Pour *P* fixé** (sur une machine parallèle donnée) :

$$O(T_{par}(N)) = O(N.\log(N)) + O(\log(N)) + O(N)$$

$$O(T_{par}(N)) = O(N.\log(N))$$

**Idem tri séquentiel !!**

Quick-sort sur hypercube de processeurs

## Modélisation simple des perfs.

**Bilan de la modélisation rapide de l'hyper-quick-sort:**

$$O(T_{par}(N)) = O(N.\log(N))$$

- Le tri séquentiel initial des données locales masque encore tout !
  → Tri séquentiel ≅ Hyper-Quicksort !
- Raisonner plus finement sur des ordres de grandeurs est délicat !
  ex : $O(N/P.\log(N/P)) + O(\log(P))$ <?> $O(N.\log^2(P))$

→ **Faire une modélisation plus précise – ne pas se contenter des ordres de grandeur : calculer les TEMPS d'exécution**

***Rappel :*** *Une parallélisation réelle sur une machine à P processeurs :*
*- Ne change pas la complexité du problème*
*- Divise juste le temps par une valeur fixe (le speed up!)*

Quick-sort sur hypercube de processeurs

## Modélisation détaillée des perfs.

$$T_{par}(N,P) = \alpha.\frac{N}{P}.\log(\frac{N}{P}).t_{comp} + \frac{(\log(P)-1).\log(P)}{2}.(t_s + t_w) + \log(P).\log(\frac{N}{P}).t_{comp} + \log(P).\left(t_s + \frac{N}{2.P}.t_w\right) + \log(P).\frac{N}{P}.t_{comp}$$

$$T_{par}(N,P) = \alpha.\frac{N}{P}.\log(\frac{N}{P}).t_{comp} + \log(P).\log(\frac{N}{P}).t_{comp} + \log(P).\frac{N}{P}.t_{comp} + \frac{(\log(P)-1).\log(P)}{2}.(t_s + t_w) + \log(P).\left(t_s + \frac{N}{2.P}.t_w\right)$$

Quick-sort sur hypercube de processeurs

## Bilan de la modélisation des perfs.

**Démarche :**

**1 - Modélisation du temps d'exécution de l'hyper-quicksort :**
- hyp sur le réseau d'interconnexion des processeurs
- modélisation de base des communications
- hyp sur la distribution des données et la qualité des pivots

$T^{hqs}_{par}(N,P) = ...$

**2 - Comparaison aux temps d'exécution d'autres tris parallèle :**

$T^{hqs}_{par}(N,P) <?> T^{bs}_{par}(N,P)$

**3 - Confrontation à l'expérimentation**

**Problèmes :** beaucoup d'hypothèses → manque de précision
- **Modéliser pour classifier les solutions**,
- Expérimenter pour connaître les temps d'exécution

Algorithmique distribuée

Fin