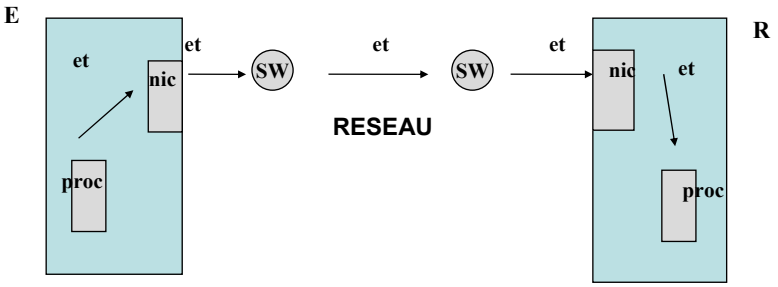
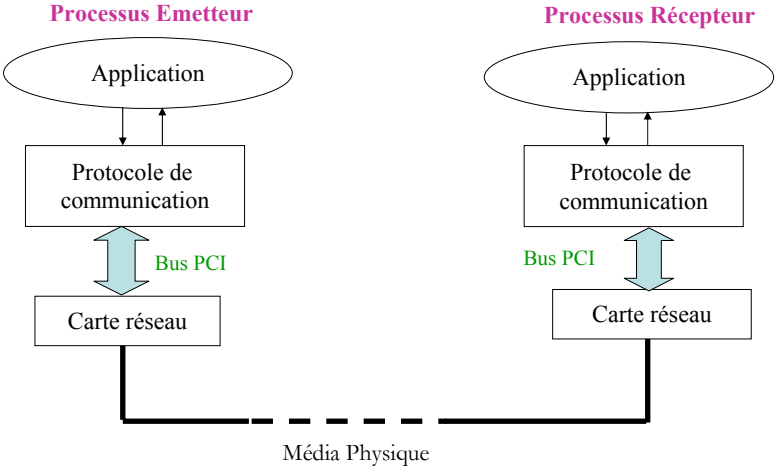
Eléments d'étude de performances d'une application distribuée

1) Synoptique d'une communication



2) Réseau d'Interconnexion



T(n)E-R = overhead + routage + latence canal + contention

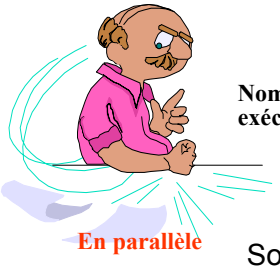
Définitions

Latence {

- Bande passante : Débit maximal pour propager l'information après son entrée sur le réseau
- Temps de transmission : Taille du message / bande passante (sans contention)
- Temps de transit : Temps pour que le premier bit arrive au récepteur
- Coût émission : Temps pour un processeur, pour délivrer un message sur le réseau
- Coût réception : Temps pour un processeur, pour retirer un message du réseau

3) La tyrannie de la loi d'Amdhal

En séquentiel



En parallèle

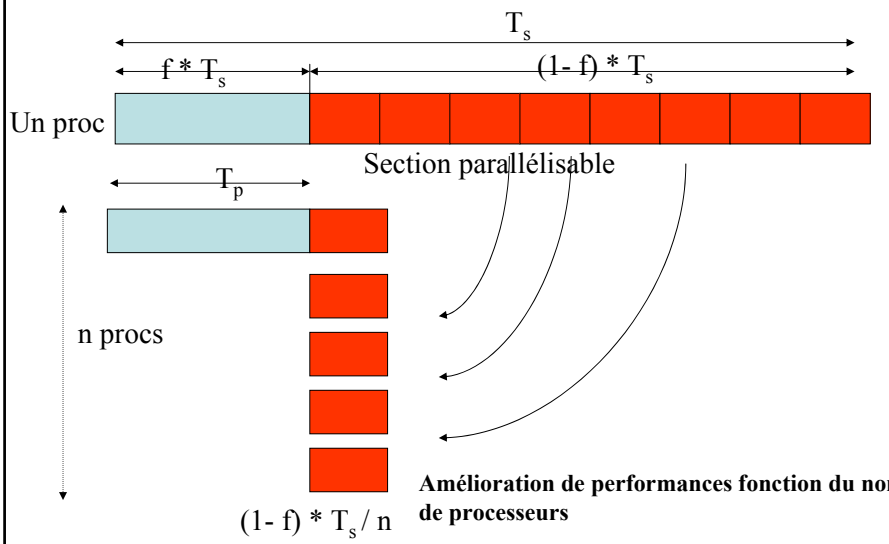
$$T = N * C * Tc$$

- N : Nombre d'instructions exécutées
- C : Nombre moyen de cycles de base par instruction
- Tc : Temps d'un cycle du processeur

Soit :
Ts = Temps d'exécution de la partie séquentielle
Tp = Temps d'exécution de la partie parallèle
n = Nombre de processeurs

Soit f la fraction du temps de la partie séquentielle

3.1) La loi d'Amdahl



Amélioration de performances fonction du nombre de processeurs

3.2) Speed-up : facteur d'accélération

- L'augmentation de performances attendue, en améliorant une partie de la machine, est limitée par la fraction du temps pendant laquelle cette partie est utilisée.

T = Ts + Tp/n

L'augmentation de performances est donnée par :

$$S(n)=\frac{Ts}{f*Ts+(1-f)*Ts/n}=\frac{n}{1+(n-1)*f}=\frac{1}{f+\frac{1-f}{n}}$$

Lorsque n -> infini l'accélération est limitée à 1/f

• **Exemple1: 80 : 20**
80% parallèle sur 80 PE
20% séquentiel

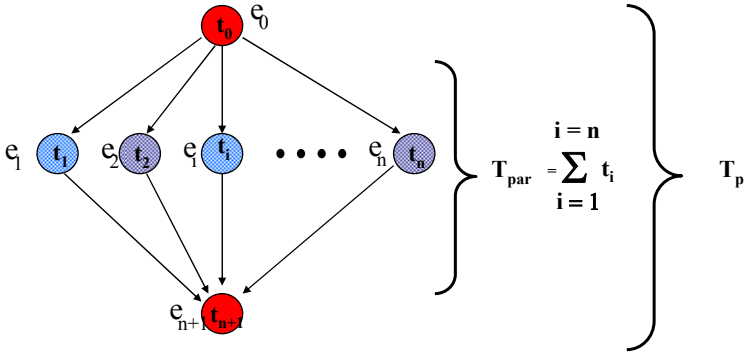
$$Speed\_up = \frac{1}{(1-0,80) + \frac{0,80}{80}} = 4,76$$

• **Exemple2: 99 : 1**
99% parallèle sur 100 PE
1% séquentiel

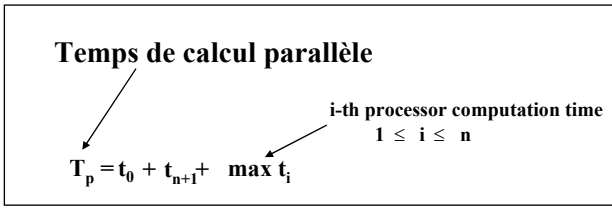
$$Speedup = \frac{1}{(1-0,99) + \frac{0,99}{100}} = 50,25$$

4) Un modèle analytique des performances

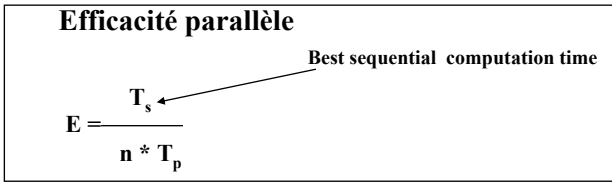
Soit un graphe de tâches à un seul niveau représentant un travail partionné en unions disjointes de tâches s'exécutant sur n calculateurs.



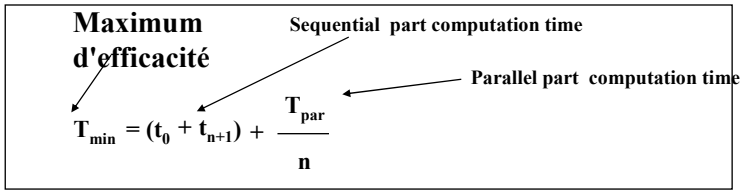
Temps de calcul parallèle



Efficacité parallèle



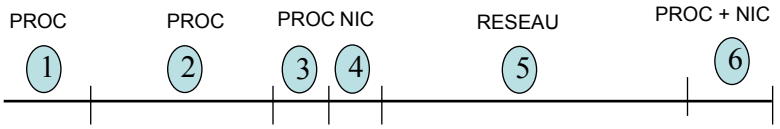
Maximum d'efficacité



5) Les trois paramètres clés des architectures distribuées

- **la latence** pour accéder à une donnée
  - latence de la hiérarchie mémoire (caches, mémoire centrale, disque)
  - latence réseau
- **la vitesse de transfert** des données d'une tâche s'exécutant dans un processeur vers une tâche d'un autre processeur
- **la performance** scalaire d'un processeur : le nombre d'instructions par cycle (IPC ou CPI)

5.1) Le coût d'une émission



1. Émetteur place le message dans un buffer de sortie
2. Préparation des paquets et transfert dans une file pour émission
3. Activation du handler de messages
4. Exécution du handler de messages
5. Temps d'occupation de la bande passante du réseau
6. Temps d'occupation du support hardware

Exercice

Soit :
- Un calculateur parallèle dont les accès mémoires distants coûtent 2.000ns
- Le processeur a une horloge à 2 Ghz et exécute une instruction par cycle CPI =1
- Seulement 0,5 % des instructions engendrent un accès distant

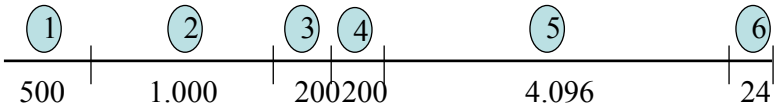
Calculer le CPI d'une telle machine :

Nombre d'instructions exécutées en 1 ns =
Coût en instructions des accès distants =

Solution ?

Exemple

Processeur 1 Ghz : une instruction/cycle
Réseau : BP 2 Gb./s.
Messages : 1 Ko.



Coût de la communication : 6.020 instructions dont 1.724 dans le PROC

5.2) Calcul du temps d'exécution en parallèle

T = Tseq + Tcom

Tseq est le temps d'exécution dans chaque processeur
Tcom le temps de LA MESSAGERIE.

T = n * g + Nc (Ts + Lc*Tb)

n = nombre de " paquets " d'instructions dans chaque processeur
g = nombre d'instructions par " paquets "
Nc = nombre de communications
Ts = temps d'établissement de la communication
Lc = longueur du message
Tb = temps d'émission par octets transmis

Si exécution et communication peuvent être recouvertes alors :

T = max (n*g + Nc*Ts, Nc*Lc*Tb)

• *Exemple IBM/SP2*

Ts = 39 micros.
Tb = 0,028 micros.
Perf. du proc. = 328 Mflops/s. => 328 inst.flot/micros.

• Intel Pentium IV 3 Ghz ou AMD Athlon 1.8 Ghz

Com ~ 60,000 instructions

En prenant 2 messages par flots (Nc =2) et Lc=1024 octets:
T = max (n*g + 78, 57) =
T = n*g + 78 micros. => latence équivaut à 328*78 = 25.000 instructions

6) Améliorer l'efficacité du calculateur parallèle

Limites physiques : Expédier 1 bit à 100 mètres prend un temps théorique équivalent à l'exécution locale de 10.000 instructions.

demain

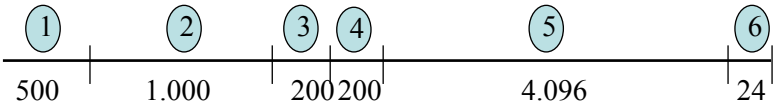
100.000 instructions voire plus.

Solutions:

•Réduire les latences en améliorant les localités :

•Tolérer les latences restantes en pipelinant communication et calcul

6.1) Identification des latences

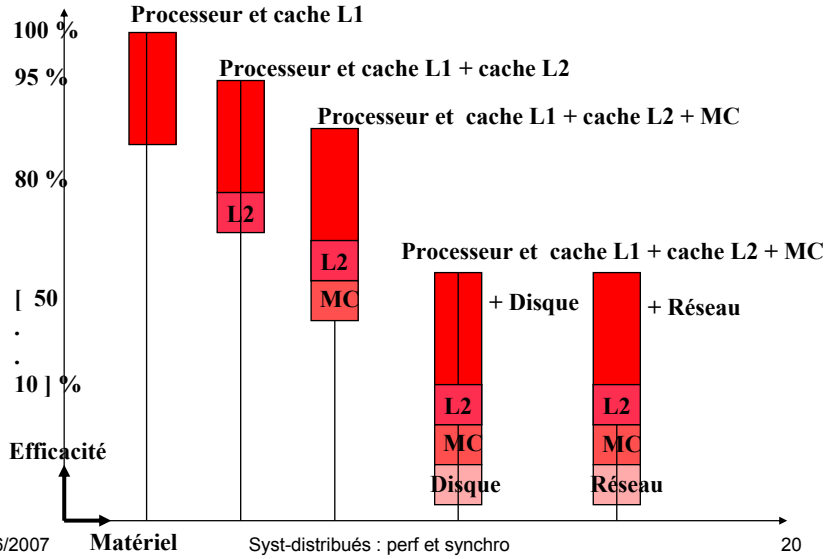


- **Accès mémoire ou communication**
  - temps processeur, temps interface, délai de transit, l'occupation de la bande passante, contention réseau
- **Latence de synchronisation**

Temps séparant l'exécution d 'une opération de synchronisation de la poursuite du calcul (opération lock, barrier)

While (!done) do
mydiff = diff = 0 ;
Barrier(bar1, nprocs) ;
For i <- mymin to mymax do

6.2) Réduire et tolérer les latences



- **Réduire les latences**

  - Réduire les temps d 'accès à chaque niveau de la hiérarchie mémoire - tendre vers les coûts matériels
  - Re-structurer l'organisation système pour réduire la fréquence des accès à latence importante
    - » Duplication des données - cache
    - » Amélioration de la localité des données
  - Re-structurer l'application
    - » Décomposer et allouer localement le calcul au processeur
    - » Re-structurer les données pour améliorer les localités spatiales et temporelles

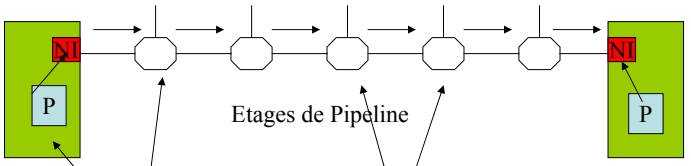
- **Tolérer les latences résiduelles : recouvrir calcul et communication**

  - Réduire les temps d'exécution à l'intérieur du même processus (concerne la mémoire et le réseau, pas le disque)

  - Latences relativement faibles et donc difficiles à recouvrir : le temps de commutation de contexte de processus classique n'est pas compatible avec ces temps de latences.

6.3) Solutions

• Latences des accès mémoires ou des communications :
Pipe-line sur les accès distants



Recouvrement communication avec communication.
Recouvrement calcul par communication.

1) Transfert par bloc

Regroupement de messages pour réaliser des messages plus longs (limité par la bande passante du réseau) pour amortir l'overhead de constitution des messages. Ne nécessite qu'un seul ack du receveur. La latence du processeur ou du NIC n'est pas impactée.

2) Prefetching

Anticiper la communication en la préparant au plus tôt :

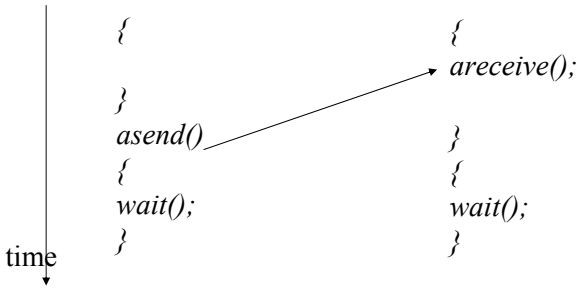
- à l'initiative de l'émetteur qui l'expédie au récepteur avant que celui-ci ne la demande
- à l'initiative du receveur : communication établie à la demande

3) Asynchronisme

Le calcul se poursuit par des opérations indépendantes sans attendre le ack du recepteur : - à l 'initiative de l'émetteur

Exemple de prefetching et d'asynchronisme

La precommunication engendre une communication avant le point ou l'opération apparaît dans le programme.



ProcA

For i<-0 to n-1 do
calcul de A[i] ;
write A[i] ;
a-send (A[i] to **ProcB) ;** */* envois asynch*/*
End for
/ le calcul est transféré ds une autre boucle */*
For i<-0 to n-1 do
calcul de f(x, i) ;
suite …

ProcB

a-receive (myA[0] from **ProcA) ;**
/* pipe-line comm-calcul*/
For i<-0 to n-2 do
a-receive (myA[i+1] from **ProcA)** **;**
while (!recv-probe(*myA[i]) itère ;*
use myA[i] ;
calcul de g(y, i) ;
suite ...
End for
while (!recv-probe**(***myA[n-1]) itère ;*
use myA[n-1] ;
calcul de g(y, n-1) ;

Sur le processeur A tous les sends sont émis avant le calcul de f.
Les messages sont envoyés au receveur aussitôt que possible =>

- pression sur les buffers d'entrée du receveur

Si l'overhead sur le NIC est important il est avantageux de maintenir un flot de calcul sur le processeur =>

- construction d 'un pipe-line logiciel

Le *a-receive* envoie la spec du receive au gestionnaire de message et autorise le processeur à continuer. Quand la donnée arrive, le NIC est averti et il transfère la donnée dans l'application qui l'a demandée.
L'application teste que la donnée est arrivée *recv-probe* avant son utilisation. Si elle est arrivée, alors un nouveau *a-receive* est émis pour préparer le suivant =>

- recouvrement du coût du NIC et éventuellement le coût du transit et du receive.

4) Mutithreading

- Communication asynchrone mais commutation vers un thread d 'un autre contexte.
- Un programme multithreadé est un programme parallèle décomposé en P processus, P > nbprocs et P/nbprocs threads sont mappés sur chaque processeur
- Nécessite un parallélisme additionnel pour tolérer les latences

Principes de base de synchronisation dans les systèmes distribués

La notion de transaction atomique

1) Synchronisation d'horloge

- **Horloges logiques**
  - Lamport 78 et 90

  Définition d'un ordre total dans lequel les événements se produisent .

  Hypothèse :

  a->b < a précède b> l'événement a se produit avant l'événement b

  Propriétés :

  - -> est une relation transitive
  - Si x et y sont indépendants alors x -> y est faux de même que y -> x
  - À chaque événement x, on associe une valeur t(x) sur laquelle toutes les machines doivent s'accorder :

    Si a -> b alors T(a) < T(b)

    T est une fonction linéaire croissante positive.

- **Algorithme :**

Chaque message transporte sa date d'émission calculée avec l'horloge de l'émetteur ; à l'arrivée si la date lue est < à la date du récepteur tr, le message reçoit la date tr+1 ; sinon, tr est retenue.

Si un processus envoie ou reçoit deux messages consécutifs, il doit avancer son horloge d'une unité de temps.

Ainsi tous les processus s'entendent sur l'ordre dans lequel les événements se produisent

2) La notion de transaction atomique

- **Définition**
  - Modèle issu de l'accès aux bases de données
  - Un ensemble d'opérations effectuant une seule fonction logique de façon atomique quel que soit l'état des systèmes et des réseaux.
  - Une transaction est effectuée (committed) ou abandonnée (aborted)
  - L'état du système après l'abandon d'une transaction peut être restitué et la transaction réitérée (rolled back)
- **Modèle de persistance**
  - Mémoire volatile : données non persistante (MC)
  - Mémoire non volatile : données persistantes (disques)
  - Mémoire stable : données résistantes (disques RAID)

- **Primitives de base d'une transaction**
  - Begin_Transaction // initialise une transaction
  - End_Transaction // termine une transaction et initie la validation
  - Abort_Transaction // interrompt la transaction et restitue les valeurs présentes avant le début de la transaction
  - Read // lecture de données persistantes
  - Write // écriture persistante de données
- **Propriétés**
  - Atomicité // L'ensemble des opérations constituant une transaction est indivisible
  - Consistance // une transaction ne peut s'affranchir des contraintes d'intégrité
  - Sérialité // l'exécution de transactions simultanées s'effectue séquentiellement dans un ordre arbitraire
  - Durable // les changements sont définitifs après validation

ACID (Atomicity, Consistency, Isolation, Durability)

La notion d'ordonnancement

```
Begin_Transaction
 x = 0 ;
 x = x + 1 ;
End_Transaction
```

```
Begin_Transaction
 x = 0 ;
 x = x + 2 ;
End_Transaction
```

```
Begin_Transaction
 x = 0 ;
 x = x + 3 ;
End_Transaction
```

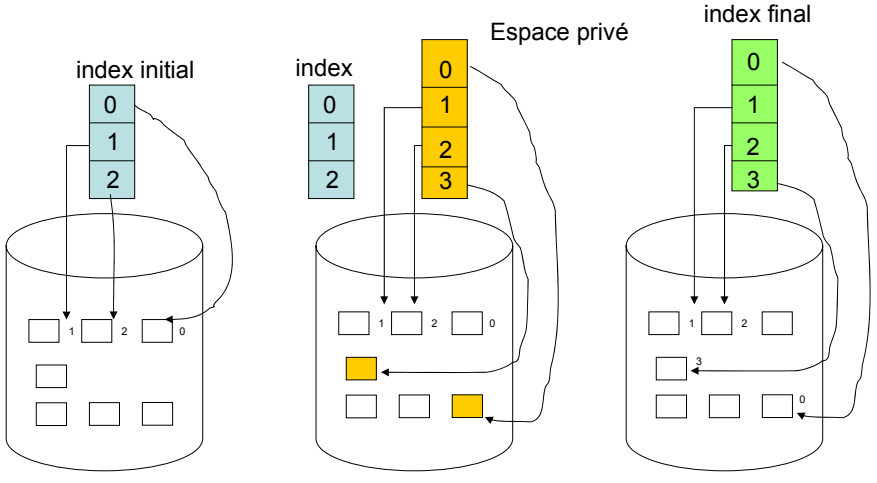
01	02	03
x = 0 ;	x = 0 ;	x = 0 ;
x = x + 1 ;	x = 0 ;	x = 0 ;
x = 0 ;	x = x + 1 ;	x = x + 1 ;
x = x + 2 ;	x = x + 2 ;	x = 0 ;
x = 0 ;	x = 0 ;	x = x + 2 ;
x = x + 3 ;	x = x + 3 ;	x = x + 3 ;
Correct	Correct	Incorrect

Le système doit garantir un entrelacement valide des transactions.

3) Implémentation

- **3.1) Espace de travail privé**
  - Allocation d'un espace de travail privé au processus contenant une copie des ressources nécessaires à la transaction
  - En lecture : copie non nécessaire ; allocation d'un pointeur vers l'espace de travail du processus source
  - En écriture : copie de la table de références désignant l'espace de travail (et non l'espace en entier) (Ex. I-nodes en Unix pour des fichiers)

Exemple



Implémentation (suite)

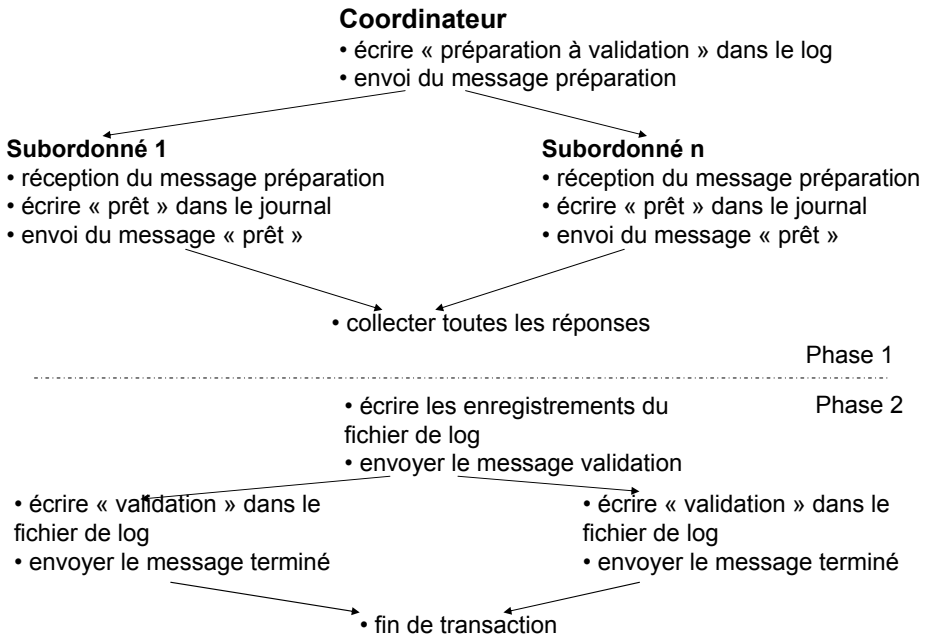
- **3.2) Journalisation (fichier de log)**
  - Maintien d'un fichier temporaire dont chaque enregistrement mémorise :
    - L'identification de la transaction
    - Le nom de l'item modifié
    - L'ancienne valeur
    - La nouvelle valeur
    - un status (begin, aborted, committed)
  - Ecriture en mémoire stable
  - Transfert en mémoire persistante
  - Validation du fichier de log
  - Undo (Ti)
  - Redo (Ti)
  - Rollback sur le fichier de log si la transaction est avortée pour annuler ou terminer après une procédure de reprise

- **3.3) La validation d'une transaction**

Protocole de validation à 2 phases « two phase commit ».

Nécessité d'un protocole pour assurer l'atomicité sur des ressources distribuées et diverses.

- Le processus qui exécute la transaction joue le rôle de « coordinateur »
- Les subordonnés associés à la validation écrivent dans le journal leur décision



4) Contrôle de concurrence

• 4.1) Verrouillage

Réalisé par un gestionnaire de verrous centralisé, ou un gestionnaire local pour les fichiers locaux.

Le gestionnaire gère une liste des fichiers verrouillés, ce qui empêche tout utilisateur d'accéder à un fichier verrouillé.

– **Optimisation**

Distinction entre verrous d'écriture (bloquant) et verrous de lecture (non bloquant en lecture)

– **Granularité**

Unité de verrouillage (fichier, enregistrement, page…). Compromis entre nombre de transactions parallèles et nombre de verrous.

– **Implémentation**

Two-phase locking (verrouillage en 2 phases) garantit la sérialisation des accès.

– **Algorithme « two phase locking »**

En phase 1 le processus acquiert tous les verrous dont il a besoin et ne peut modifier de fichiers dans cette phase.

En phase 2, le processus relâche tous les verrous acquis.

Extension :

Le système ne passe en phase 2 que lorsque la transaction est terminée avec succès ou non ; ainsi la consistance est assurée car une transaction ne lit que des valeurs écrites par une transaction validée.

Toutes les acquisitions et tous les relâchements sont gérés par le système sans que la transaction ait à s'en occuper.

Interblocage possible peut-être gérée par un watchdog »

- **4.2) Estampillage**

– Algorithme

Associer une estampille à chaque BEGIN de transactions : l'algo de Lamport assure que chaque estampille est unique.

Chaque fichier possède une estampille de lecture et une estampille de lecture, dates auxquelles la dernière lecture ou la dernière écriture a eu lieu.

Ces estampilles permettent de définir un ordre complet sur les accès.

La transaction la plus ancienne s'exécute d'abord ; une transaction « en retard » est abandonnée.

Solution sans interblocage.