# Introduction au parallélisme

Cécile Germain-Renaud
cecile.germain@lri.fr

## Organisation

- 12 séances cours et TD, 4h chacune
- Les slides ne suffisent pas
- 2 devoirs – presque tous largement commencés en TD
  - A rendre à la date prévue,
  - 1 jour de retard = 1 point en moins.
- Des références bibliographiques par chapitre



## 1. Introduction

## Applications

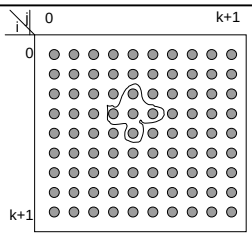
- Simulation numérique : expériences in silico
  - Trop grandes : météorologie
  - Trop petites : biologie, matériaux
  - Trop dangereuses : maintenance nucléaire
  - Trop coûteuses : crash-tests, conception aéronautique
  - Impossibles : climat, astrophysique

  À réaliser en réalité
- Serveurs traditionnels : Fichiers, Web, Vidéo, Base de données, …
- Serveurs de calcul : « on demand computing »
- Réalité virtuelle ou augmentée : médias, médical

  Peuvent consommer une puissance de calcul et demander une capacité mémoire supérieure à celle d'un processeur

4

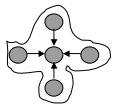
## Relaxation : un exemple de parallélisme à grain fin




- Spécification

  $X_{n+1} = f(X_n)$
- Réalisation séquentielle

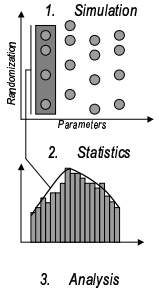
```
double X[k+1], Y[k+1];
for (i = 1; i <= k  ; i++)
  for (j = 1; j <= k  ; j++)
    Y[i,j] = X[i,j];
for (i = 1; i <= k  ; i++)
  for (j = 1; j <= k  ; j++)
    X[i,j] = 0.25*(Y[i+1,j] + Y[i-1,j] +Y[i, j-1] + Y[i, j+1]);
```



5

## Un autre type d'application : les simulations Monte-Carlo

- Nombre de simulations = taille de l'espace des paramètres x la randomisation interne




6

## Faisabilité et objectifs

- Beaucoup d'applications sont naturellement parallèles
  - Relaxation : grain fin
  - Monte-Carlo : grain moyen à gros
  - Couplage : gros grain
- Support matériel pour exploiter ce parallélisme
  - Plusieurs processeurs
- Objectifs
  - THP : traiter des grands problèmes en temps raisonnable
  - HP : améliorer le temps de traitement des problèmes ordinaires
- Questions
  - Performance réelle : Adéquation Architecture / Modèle de programmation / Application
  - Utilisabilité :
    - Modèles et environnements de programmation
    - Tout le reste – dont les I/O

7

# 2. Architectures parallèles

## Lectures – cours 1, 2, 3

- Recommandé
  - A.J. van der Steen et J.Dongarra. *Overview of Recent Supercomputers* http://www.top500.org/ORSC/2004/
- Références
  - D. E. Culler et J.P. Singh *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann 1998 (1100 pages !)

9

Plan

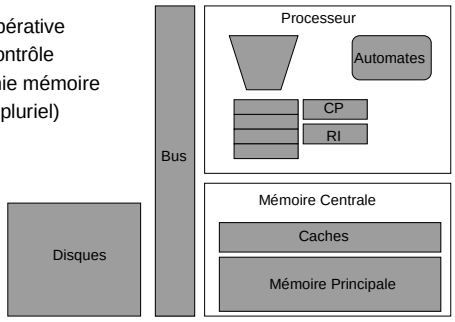
2.1 La classification de Flynn

2.2 Classification par la mémoire

10

Processeur séquentiel

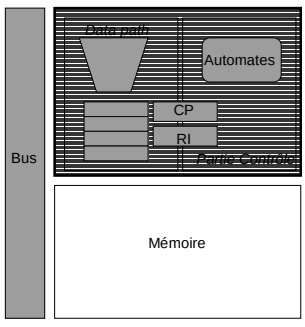
- Partie opérative
- Partie contrôle
- Hiérarchie mémoire
- Bus (au pluriel)




11

Processeur séquentiel et classification de Flynn

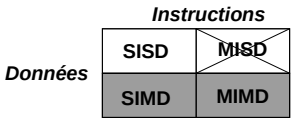
- Partie opérative
- Partie contrôle




12

## Classification de Flynn

- Degré de réplication de la partie opérative et de la partie contrôle



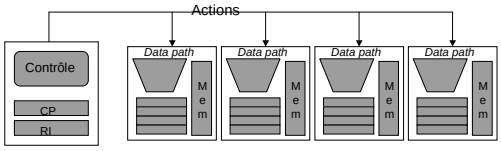

13

## Classification de Flynn

- SISD : Single Instruction Single Data
  - Microprocesseur : microparallélisme + instructions SIMD
  - Voir cours Architecture avancée
- SIMD : Single Instruction Multiple Data
  - Architectures SIMD
  - Processeurs vectoriels
- MIMD : Multiple Instruction Multiple Data
  - MPP
  - Clusters
  - Constellations
  - Toutes les machines parallèles généralistes actuelles

14

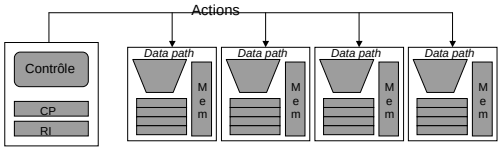
## SIMD




- Le silicium mémoire est sous-exploité

15

## SIMD



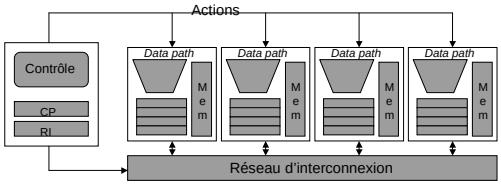

• Adapté aux opérations vectorielles élémentaires

| | | | | |
|---|---|---|---|---|
| = | X(1) | X(2) | X(3) | X(4) |
| + | Y(1) | Y(2) | Y(3) | Y(4) |
| | Z(1) | Z(2) | Z(3) | Z(4) |

• Unités fonctionnelles peu puissantes 1/4/8 bits
• Parallélisme à grain très fin

16

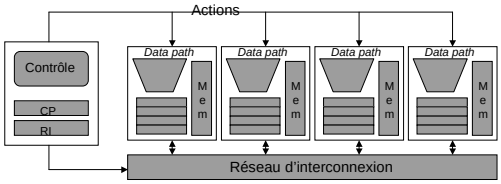
## SIMD




- Réseau
  - SIMD : communications structurées identiques – modèle de programmation systolique
  - Communications quelconques

17

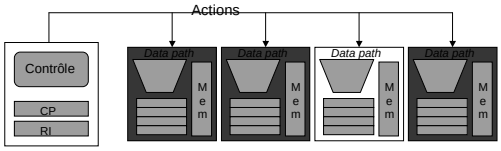
## SIMD : Problème de l'adressage indirect




| | | | | |
|---|---|---|---|---|
| = | X(1) | X(2) | X(3) | X(4) |
| + | Y(L(1)) | Y(L(2)) | Y(L(3)) | Y(L(4)) |
| | Z(1) | Z(2) | Z(3) | Z(4) |

18

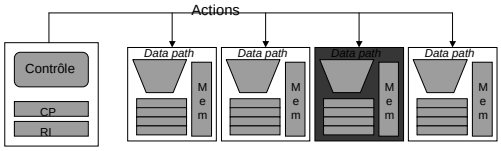
## SIMD : Problème des conditionnelles




```
if (X(i) == 0)
   Y(i) = 0
else
   Y(i) = 1/X(i)
```

19

## SIMD : Problème des conditionnelles




```
if (X(i) == 0)
   Y(i) = 0
else
   Y(i) = 1/X(i)
```

20

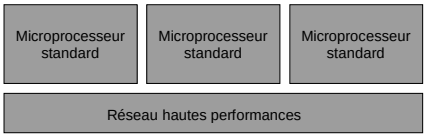
## SIMD

- Les premières machines parallèles commerciales (années 80)
  - Connection Machine 2 de Thinking Machines Corp.
  - MASPAR
- N'existent plus actuellement comme machines généralistes
  - CPU spécialisés
  - Fréquence d'horloge incompatible avec l'évolution technologique
  - Les synchronisations limitent le parallélisme exploitable
- Architectures spécialisées
  - ApeNext, coprocesseurs,…
  - Extensions SIMD des microprocesseurs

21

## MIMD non vectoriel

- Exploite les acquis des architectures de microprocesseurs
  - Hiérarchie mémoire
  - OS




22

## Plan

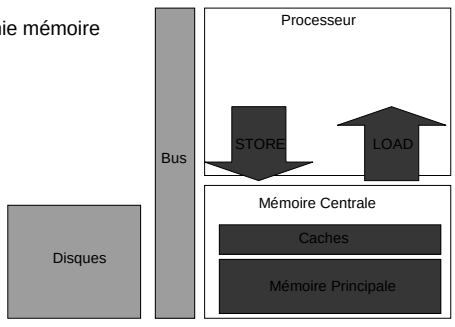
2.1 La classification de Flynn

2.2 Classification par la mémoire

23

## Processeur séquentiel

- Hiérarchie mémoire



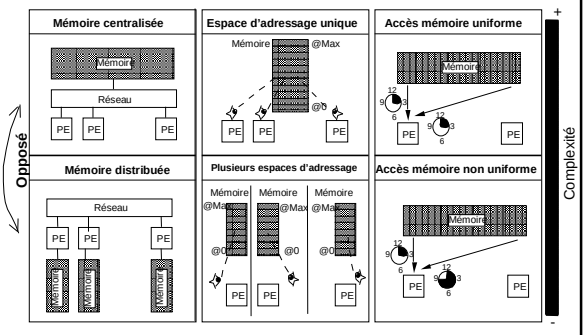

24

## Organisation mémoire

- Situation de la mémoire/des caches par rapport aux processeurs
  - Centralisée
  - Distribuée
- Organisation de l'espace d'adressage
  - Unique
  - Plusieurs
- Hiérarchie des temps d'accès mémoire
  - Uniforme
  - Non uniforme

25

## Organisation mémoire



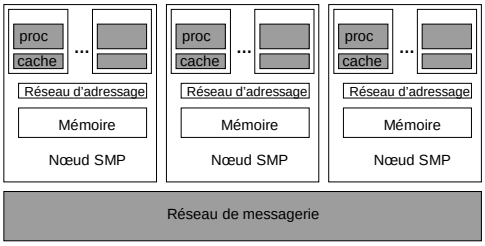

26

## Terminologie

- SMP : Symmetric Multiprocessors
  - Espace d'adressage unique, UMA
- ccNUMA : Cache Coherent NUMA
  - Espace d'adressage unique
- MPP : Massively Parallel Multiprocessors
  - Le plus souvent espace d'adressages multiples
  - Mais espace d'adressage unique ccNUMA existe
- Clusters :
  - La principale différence avec une MPP est dans le réseau (au sens large) non propriétaire

27

## Architectures hybrides




28

## Les architectures actuelles

- Architectures courantes
  - Serveur bas de gamme ou poste de travail haut de gamme
    Quelques (2-4) processeurs dans un seul PC
  - Serveurs multiprocesseurs (*SMP* – Symmetric Multi-Processor )
    Nombre moyen (4-64) de processeurs en un système - HP SuperDome, Sun SunFire, IBM Regatta
- *Clusters* of Workstations (COW - NOW)
  Ensemble de machines mono/bi/quadri processeur ou de petits SMP - HP AlphaServer, Beowulf Linux PCs
- Supercalculateurs
  - Massively Parallel Processors (*MPP*) : Multiprocesseurs mémoire distribuée – nœuds SMP
  - *Constellations* – systèmes parallèles à nœuds vectoriels

29

# 3. Performance

## Plan

- Objectifs
- Débit
- Accélération
- D'autres critères d'extensibilité

31

## Objectif (1)

- On veut résoudre un problème - produit de matrices –
- Choix de programmation
  - Plusieurs algorithmes : naïf, Cannon, …
  - Pour un algorithme, nombreux choix de parallélisation : distribution des données et du calcul, style de programmation,…
- Paramètres de l'architecture
  - Nombre de processeurs
  - Caractéristiques processeurs : vitesse, caches, architecture interne
  - Performances réseau d'interconnexion
  - Paramètres de configuration I/O OS

32

## Objectif (2)

- Définir des indicateurs objectifs qui mesurent le gain apporté par le parallélisme
- L'indicateur pertinent dépend du critère par rapport auquel on veut optimiser
  - Extensibilité d'une architecture parallèle : capacité à améliorer la performance en augmentant les ressources
  - Extensibilité d'un algorithme parallèle : capacité d'un algorithme à utiliser plus de ressources
  - Latence ou débit
    - Latence : temps d'exécution
    - Débit : nombre de transactions, de calculs,… par unité de temps

33

## Définitions

- Un algorithme A
- Souvent paramétrisé : *n*
  - Exemple : produit de matrices
- W(n) = Travail à effectuer
  - Exemple : opérations flottantes
- T(p, n) = Temps d'exécution de A sur p processeurs
- Ts(n) = Temps d'exécution sur 1 processeur
  - De A
  - Du meilleur algorithme équivalent à A
    - Exemple relaxation vs méthodes directes

34

## Plan

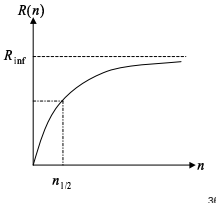
- Objectifs
- Débit
- Accélération
- D'autres critères d'extensibilité

35

## $R_{inf}$ et $n_{1/2}$

A est fixé

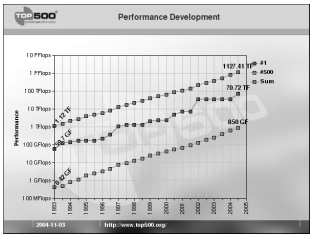
- Débit de travail, typiquement en FLOPS/s
  R(p,n) = W(n)/T(p,n)
- Débit asymptotique
  $R_{inf}(p)$ = limite de R(p,n) quand n tend vers l'infini
- Demi-performance $n_{1/2}$
  $R(p, n_{1/2}) = R_{inf}(p)/2$
- Paradigme : pipeline vectoriel
  - Temps d'exécution affine
    T(p,n) = s + cn
  - Travail linéaire
    W(n) = an




36

## Linpack et le Top 500

- Résultats depuis 1993 sur les 500 machines parallèles les plus puissantes
- Résolution d'un système linéaire
- Benchmark assez facile
- Volontariat
- Autres graphes




37

## Gordon Bell Prize: "peak performance"

| *Year* | *Type* | *Application* | *No. Procs* | *System* | *Gflop/s* |
|---|---|---|---|---|---|
| 1988 | PDE | Structures | 8 | Cray Y-MP | 1.0 |
| 1989 | PDE | Seismic | 2,048 | CM-2 | 5.6 |
| 1990 | PDE | Seismic | 2,048 | CM-2 | 14 |
| 1992 | NB | Gravitation | 512 | Delta | 5.4 |
| 1993 | MC | Boltzmann | 1,024 | CM-5 | 60 |
| 1994 | IE | Structures | 1,904 | Paragon | 143 |
| 1995 | MC | QCD | 128 | NWT | 179 |
| 1996 | PDE | CFD | 160 | NWT | 111 |
| 1997 | NB | Gravitation | 4,096 | ASCI Red | 170 |
| 1998 | MD | Magnetism | 1,536 | T3E-1200 | 1,020 |
| 1999 | PDE | CFD | 5,832 | ASCI BluePac | 627 |
| 2000 | NB | Gravitation | 96 | GRAPE-6 | 1,349 |
| 2001 | NB | Gravitation | 1,024 | GRAPE-6 | 11,550 |
| 2002 | PDE | Climate | 5,120 | Earth Sim | 26,500 |

**Four orders of magnitude in 13 years**

D'après www.siam.org/sciencepolicy/keyes.ppt

38

## Gordon Bell Prize: "price performance"

| *Year* | *Application* | *System* | *$ per Mflops* |
|---|---|---|---|
| 1989 | Reservoir modeling | CM-2 | 2,500 |
| 1990 | Electronic structure | IPSC | 1,250 |
| 1992 | Polymer dynamics | cluster | 1,000 |
| 1993 | Image analysis | custom | 154 |
| 1994 | Quant molecular dyn | cluster | 333 |
| 1995 | Comp fluid dynamics | cluster | 278 |
| 1996 | Electronic structure | SGI | 159 |
| 1997 | Gravitation | cluster | 56 |
| 1998 | Quant chromodyn | custom | 12.5 |
| 1999 | Gravitation | custom | 6.9 |
| 2000 | Comp fluid dynamics | cluster | 1.9 |
| 2001 | Structural analysis | cluster | 0.24 |

**Four orders of magnitude in 12 years**

D'après www.siam.org/sciencepolicy/keyes.ppt

39

## Plan

- Objectifs
- Débit
- Accélération
- D'autres critères d'extensibilité

40

## Accélération

- $S(p,n) = T_s(n)/T(p,n)$
- Bon comportement : croissante en p pour n fixé
- Application parfaitement parallélisable : $S(p,n) = p$
  - Somme de vecteurs
  - Parallélisme plaisant (ou trivial)
- Accélération linéaire : $S(p,n) = kp \quad ; k < 1$
- Limitations
  - Taille du problème
  - Processeurs oisifs – problème d'équilibrage de charge
  - Surcoûts en particulier communications

41

## Réduction

On néglige les coûts de communication

- 1 donnée par processeur n=p

  $Ts(p) = p$

  $T(p,p) = \log(p)$

  $S(p,p) = p/\log(p)$
- k données par processeur n=kp

  $Ts(p) = n$

  $T(p,n) = \log(p) + n/p$

  $S(p,n) = n/(\log(p) + n/p)$

42

## Loi d'Amdhal

- L'application comporte
  - une fraction séquentielle f : contrôle, calculs scalaires répliqués,…
  - Une fraction parfaitement parallélisable 1-f
- $T(p,n) = f + (1-f)/p$
- $S(p,n) = p/(pf + 1-f) < 1/f$
  - Inutile d'ajouter des processeurs au delà d'une certaine limite qui dépend de l'application
- S'applique au déséquilibre de charge en général
- Quelle est l'hypothèse implicite ?

43

## Surcoûts

- Communication sur mémoire distribuée
- Synchronisation sur mémoire partagée

44

## Facteurs pénalisants pour l'accélération

Quand p augmente

- La quantité de travail par processeur diminue
  *Mais le débit peut augmenter à cause d'une meilleure localité des données – en particulier utilisation du cache*
- La quantité de communication ou de synchronisation par processeur diminue moins vite que la quantité de travail par processeur
- Les déséquilibres de charge augmentent par manque de travail
- Le coût des communication peut augmenter

45

## Optimiser pour d'autres critères

- L'hypothèse implicite est que le problème est de taille fixe
- Ts (n) = a + nb ; f = a/(a + nb) ;
- Les problèmes de taille fixe existent : traitement d'images
- Mais d'autres contraintes peuvent être plus pertinentes par rapport à la demande de l'utilisateur : résoudre un problème, effectuer une simulation etc.
- En temps constant
- En encombrement mémoire constant

46

## La loi de Gustafson

- Partie séquentielle et partie parfaitement parallélisable
- La partie parfaitement parallélisable est indéfiniment extensible

$$T(p,n) = a + nb/p = 1 \text{ d'où } n = p(1-a)/b$$

$$Ts(n) = a + nb$$

$$S(p,n) = a + (1-a)p$$

- Accélération linéaire !
- Où est l'erreur ?

47

## Amdhal et Gustafson

- Il n'y a pas d'erreur
- Amdhal : n et p sont indépendants
- Gustafson : on calcule S(p, n(p))

48

Plan

- Objectifs
- Débit
- Accélération
- D'autres critères d'extensibilité

49

Critères économiques

- Par processeur
- Vitesse V(p,n) = R(p,n)/p
- Efficacité E(p,n) = S(p,n)/p < 1
- Iso-efficacité : Taille nécessaire pour atteindre une efficacité donnée
  - Peut on atteindre une certaine efficacité sur un système donné ?
  - De combien faut-il augmenter la taille du problème ?
  - Existe-t-il une taille maximale du système telle qu'on ne puisse plus atteindre l'efficacité requise ?
  - Comparaison des applications : la meilleur isoefficaité est la plus petite

50

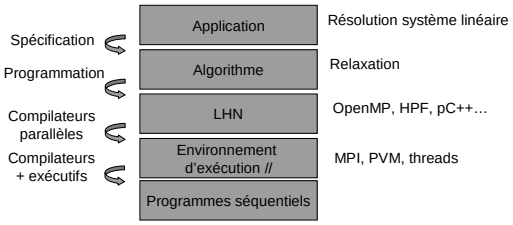
# 4. Introduction à la programmation parallèle

## Plan

- Modèles de programmation et modèles d'exécution d'exécution
- Ordonnancement et placement

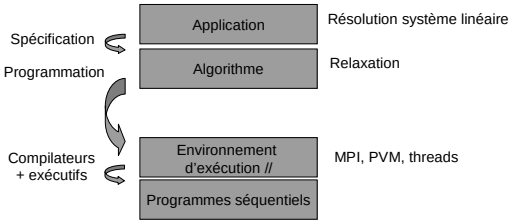
52

## Modèles de programmation et modèles d'exécution




53

## Modèles de programmation et modèles d'exécution




54

## Modèles

- D'exécution
  - Expriment – ou sont une abstraction de – l'architecture matérielle
    - mémoire partagée /mémoire distribuée
    - SIMD/MIMD
- De programmation
  - Expriment – ou sont une abstraction de – sources du parallélisme : données/contrôle

55

## Modèles de programmation : Djikstra/Flynn

- Parallélisme de contrôle
  Composition parallèle de processus séquentiels
  PAR(SEQ) - MIMD

```
pardo i = 1, n
    a(i) = b(i) +c(i)
    x(i) = y(i) + z(i)
end do
```

- Parallélisme de données
  Composition séquentielle de processus parallèles
  SEQ(PAR) - SIMD

```
forall i = 1, n
    a(i) = b(i) +c(i)
forall i = 1, n
    x(i) = y(i) + z(i)
```

56

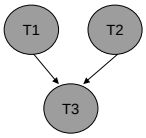
## Modèles d'exécution

- Espaces d'adressage multiples
  - $p$ processus
  - Communiquent par messagerie
  - Le programmeur ou le compilateur doit définir le placement des données et l'ordonnancement des calculs
  - Le placement des données peut définir celui des calculs : ex. OCR
- Espaces d'adressage unique
  - $p$ threads
  - Communiquent à travers des variables partagées
  - Le programmeur ou le compilateur doit définir le placement et l'ordonnancement des calculs

57

## Ordonnancement

- Programme = Graphe de tâches
- Une tâche est une unité de traitement séquentiel
- Un arc (T1, T2) correspond à un transfert d'information à la fin de t2 vers le début de t1
- Un graphe de tâches sans cycle définit un ordre partiel
  - T1 << T2 s'il existe un chemin de T1 vers T2
  - Ordre total : programme séquentiel
  - Ordre partiel : programme parallélisable




T1 // T2
T3

58

## Ordonnancement et modèle de programmation

- La définition de l'ordonnancement peut être
  - Réalisée dans un programme parallèle
  - Extraite automatiquement d'une spécification plus ou moins contrainte : parallélisation automatique
- L'expression de l'ordonnancement dans les langages parallèles
  - Par structures de contrôle : SEQ(PAR)
  - Par synchronisation : PAR(SEQ)

59

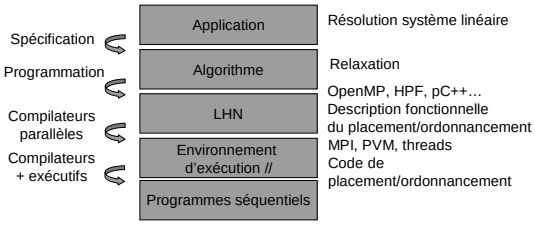
## Placement

- L'ordonnancement ne prend pas en compte le nombre de processeurs disponibles
- Le placement replie le parallélisme illimité sur les ressources réelles
- Contrainte : minimisation des surcoûts

**Parallélisation = placement + ordonnancement**

60

## Modèles de programmation et modèles d'exécution




61

## Plan

- Modèles de programmation et modèles d'exécution
- Ordonnancement et placement

62

## Placement statique

- Off-line, éventuellement paramétré par le nombre de processus et le numéro de processus
- Les temps de calcul doivent être connus
- Cas régulier : Bloc, cyclique, cyclique (k), …
- Cas irrégulier : ad hoc – par exemple bisection récursive

63

## Décalage, distribution bloc

```
double A[N], B[N];
for (I=0; I<N-1; I++)
  A[I] = B[I+1];
```

*À la HPF*

```
double A(N), B(N)
!HPF$ distribute (block) :: A, B
forall I=1,N-1
    A(I) = B(I+1)
```

*À la OpenMP*

```
#define M=N/P
double A[N], B[N];
#pragma omp for schedule(static, M)
for (I=0; I<N-1; I++)
  A[I] = B[I+1];
```

*À la MPI*

```
#define M=N/P
double a[M], B[M];
<group definition>
if (me != 0)
  <send b[0] to me-1>
for (i=0; i<M-1; i++)
  a[i] = b[i+1];
if (me != P-1)
 <rcv(& a[M-1] from me+1>
```

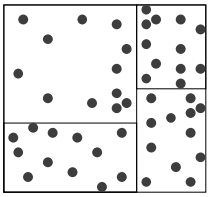
*À la pthreads*

```
#define M=N/P
double A[N], B[N];
pthread_t thread[P];
for (i=0; i < P; i++)
  pthread_create (&thread[i], myfunc, i);
for (i=0; i < P; i++)
 pthread_join (thread[i]);

void myfunc (int k)
  ideb = M*k;
  ifin = (k <P-1 ? M*(k+1) : M*(k+1) –1);
  for (i=ideb; i < ifin -1; i++)
     A[i] = B[i+1];
```

64

## Bisection récursive

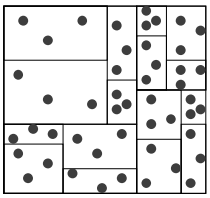


Les éléments de calcul sont caractérisés par une donnée nD

Souvent position spatiale

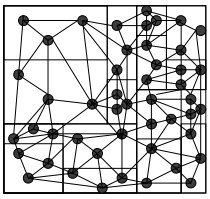
65

## Bisection récursive



66

## Bisection récursive



Applicable aussi à un maillage

67

## Placement/ordonnancement dynamique

- Motivation:
  - La durée des calculs n'est pas prévisible
  - La puissance des machines n'est pas connue
- Placement/ordonnancement on-line : décidé à l'exécution
- Questions
  - Surcoûts : gestion des processus ou des threads, des échanges d'information
  - Robustesse à l'irrégularité des temps d'exécution

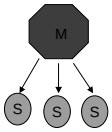
68

## Placement/ordonnancement dynamique

- Centralisé: Maître-Esclave

  *Maître*
  - Implémente un algorithme de partitionnement, par exemple bloc, avec possibilité de taille variable du bloc
    - Taille fixe : équivalent à statique
    - Guided self-scheduling: chaque esclave prend 1/P de ce qui reste
    - Factoring: chaque esclave prend 1/P de la moitié du batch restant
  - Effectue les opérations globales par exemple réduction

  *P Esclaves*
  - Exécutent un bloc, et redemandent du travail dès que terminé

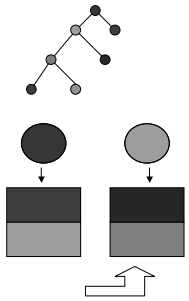



69

## Placement/ordonnancement dynamique

- Réparti
  - Vol de travail
    - Un processus pousse sur la pile des travaux en attente les tâches les plus longues
    - Les processus ou threads inactifs sélectionnent un autre processus auquel ils prennent du travail au fond de la pile
    - Peut être prouvé optimal pour une large classe d'applications
    - Implémentation délicate : gestion de verrous en espace d'adressage unique, protocole en espaces d'adressages multiples

  http://supertech.lcs.mit.edu/cilk/



70

# 4. Architectures de communication

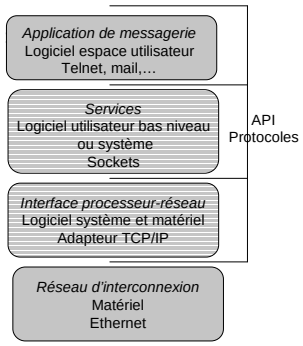
## Lectures

- Recommandé
  - A.J. van der Steen et J.Dongarra. *Overview of Recent Supercomputers* section networks http://www.top500.org/ORSC/2004/networks.html#networks
  - D.E Culler et al. LogP: Towards a Realistic Model of Parallel Computation
- Références
  - Tom Leighton *Introduction to Parallel Algorithms and Architectures* Morgan Kaufmann, 1992. (> 800 pages)

72

## Introduction

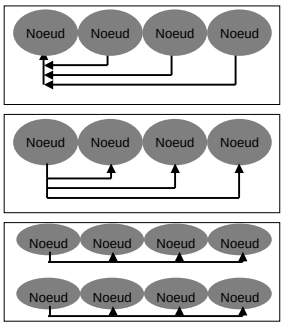
- Contexte pour ce chapitre mémoire distribuée, espaces d'adressages multiples
- Mais la partie réseau d'interconnexion a des applications pour d'autres architectures




73

## Fonctionnalités

- Obligatoire
  - Connexion point-à-point entre nœuds
- Optionnelle
  - Support communications collectives
  - Réduction
  - Diffusion 1-to-many
  - Multi-diffusion mainy-to-mainy




74

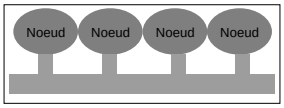
## Plan

- Topologie

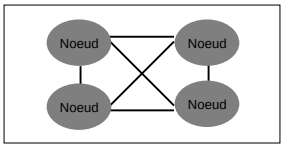
75

## Réseaux élémentaires

- Bus
  - Exclusion mutuelle
  - Point à point et diffusion
  - Extensibilité limitée



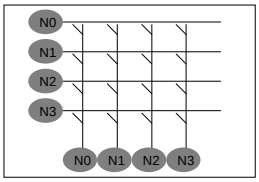

- Connexion complète
  - $O(P^2)$ liens
  - O(P) ports/noeud
  - Non extensible




76

## Crossbar

- Relativement extensible
  - $P^2$ points de croisement
  - 2P liens
  - 1 port/nœud
  - Typiquement –Fin 2004 – 16 ports




Exemple : Myrinet switch
http://www.myri.com/

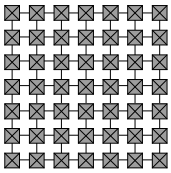
77

## Réseaux incomplets

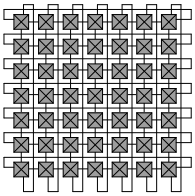
- Pour le parallélisme massif, réseaux incomplets
- Il n'existe pas de lien physique entre deux nœuds quelconques
- Le matériel doit assurer le routage
  - Transferts point à point entre deux processeurs quelconques
  - Eventuellement collectifs
  - Avec des performances élevées
    - Géométrie régulière
    - Routeurs = switch + contrôle
- Vision duale : la topologie des communication d'un algorithme

78

## Réseaux directs : grilles et tores



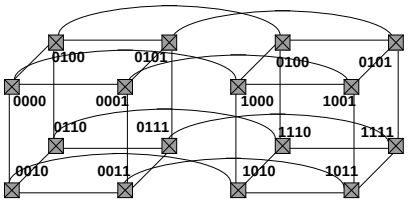
- Grille
  - Extensible
  - Non isotrope



- Tore
  - Extensible
  - Isotrope

79

## Réseaux directs : hypercubes




- Hypercube
  - Extensible – par doublement
  - Isotrope

80

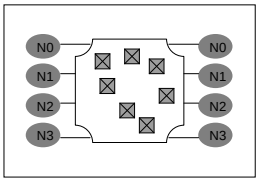
## Comparaison des réseaux directs

- Peu significatifs pour les architectures parallèles
  - Diamètre : plus grande distance
  - Distance moyenne
- Très important pour les performances des algorithmes
  - Bisection
  - Mais à pondérer par les contraintes technologiques

81

## Réseaux indirects

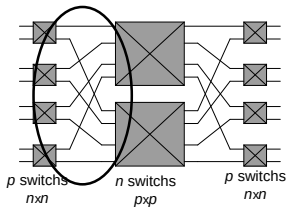
- Les processeurs sont en entrée et sortie du réseau seulement
- *Réarrangeable* si peut réaliser toute permutation des entrées sur les sorties
- Non réarrangeable = *bloquant*




82

## Reséaux indirects : le réseau de Clos $C(n,p)$

- Connexion shuffle
  - La sortie i du switch d'entrée j est connectée à l'entrée i du switch j




- Le réseau de Clos $C(n,p)$ est réarrangeable
- Réseau de Benes : n = 2 et décomposition récursive des switchs intermédiaires
- Exemple : Myrinet switch

83

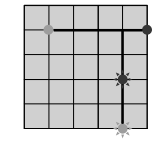
## Plan

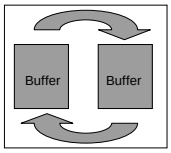
- Topologie
- Routage

84

## Routage

- Algorithme de choix des chemins
  - Câblé : routage par dimension
    - Grilles, hypercubes
  - Tabulé
    - Réseaux réarrangeables
  - Déterministe/aléatoire – fixé/adaptatif
    plus court chemin
- Traitement des conflits
  - Tamponnement ou déroutement
    hot-potatoe
  - Résolution d'interblocage par réseau virtuel
- Contrôle de flot
  - Paquet
  - Wormhole






85

## Plan

- Topologie
- Routage
- Performances

86

## Caractéristiques de la performance

- Des concepts identiques appliqués dans des contextes différents
- Latence isolée : pour un message court
  - Vue du matériel
  - Vue du programme
- Débit
  - Pour un ping-pong
  - En situation de charge

87

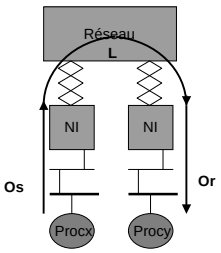
## Composantes de la performance

- Internes au réseau : matériel
  - Barebones latency
- Interface processeur-réseau
  - Matériel : débit critique – problème du bus I/O
  - Logiciel :
    - Couches de protocoles
    - Nombre de copies
- Exemple :
  - Myrinet  protocole propriétaire
    http://www.myri.com/myrinet/performance/index.html
  - Myrinet TCP
    http://www.myri.com/myrinet/performance/ip.html

88

## Le modèle LogP (1)

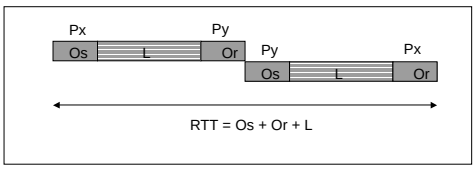
- L = latence interne au réseau
  - Délai de transmission d'interface à interface
  - Processeur disponible
- Os, Or = Overhead
  - Gestion du message par les processeur
  - Processeur non disponible => irréductible




89

## Ping-Pong

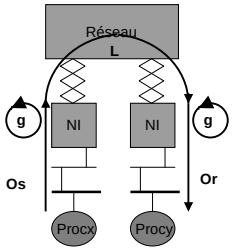
- Deux processeurs seulement communiquent
- Schéma message-réponse
- La latence décrite dans les benchmarks est la moitié du Round-trip time (RTT)
- L dépend essentiellement
  - Du débit des liens et des switchs
  - Du volume de communication




90

## Le modèle LogP (2)

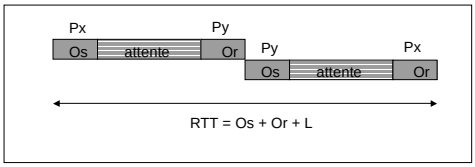
- g = Gap
  - Délai séparant deux communications sur un processeur
  - Etage le plus long du pipeline de communication
  - Composantes
    - Statique : ensemble du système
    - Dynamique : congestion
  - 1/g = débit effectif disponible par processeur




91

## Réseau saturé

- L'émission d'un nouveau message exige la réception d'un autre
- Modélisation : systèmes de file d'attente



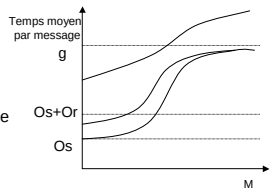

92

## Détermination expérimentale

- Sur des réseaux où l'overhead n'est pas dominant
- Mesurer g, Os, Or, L

```
do i= 1,M
   Emettre
   Calculer (C)
   Recevoir
end do
```

- Os est mesuré pour M petit, sans réception, et C = 0
- g est mesuré en faisant varier C pour situer les courbes qui convergent vers une asymptote commune




93

Plan

- Topologie
- Routage
- Performances
- Optimisations logicielles

94

Méthodes d'optimisation

- Déséquilibre entre les performances de communication et celles de calcul
  - En latence : pénalise les accès isolés
  - En débit, à un degré moindre, mais irrécupérable
  - Rien de nouveau !
- Diminuer le nombre de communications
  - A décomposition constante : Vectorisation des communications
  - Modifier la décomposition : Vs équilibrage de charge
- Diminuer le volume de communication
  - Agrégation: vs simplicité
- Recouvrement communication/calcul
  - Si l'overhead n'est pas dominant

95

# 5. Analyse de dépendances - Introduction à la parallélisation automatique

## Lectures

- La référence
  - Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann. 2002.
  - Plus particulièrement chapitres 2, 5, 6.
- Mais les aspects avancés (“méthodes polyédriques”) ne sont pas traités.
  - Paul Feautrier. Automatic Parallelization in the Polytope Model
    http://www.prism.uvsq.fr/rapports/1996/document_1996_8.ps
    Synthèse + bibliographie

97

## Avertissements

- Un cours raisonnablement approfondi demanderait 6 séances.
- TOUS LES EXEMPLES SONT DANS LE TD 5

98

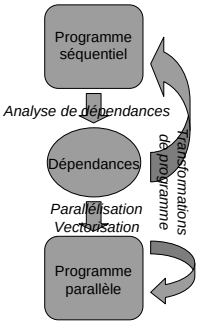
## Plan

- Introduction

99

## Introduction

- Méthodes systématiques et automatisables
  - Analyse d'un programme séquentiel, fonctionnel, parallèle
  - Pour la définition d'un ordonnancement compatible avec la sémantique du programme
  - Exprimable dans une syntaxe parallèle
- Aide à la parallélisation manuelle.
- Les paralléliseurs automatiques et leurs limites.




100

## Programme séquentiel et nids de boucles

- Programme séquentiel
  - Boucles et séquences : pas de conditionnelles, ni de procédures
  - Instruction : occurrence textuelle
  - Opération : occurrence dynamique. Exemple A(1, 3, 2)=….

```
DO i=1, N
  DO j=i, N+1
    DO k=j-i, N
(s)       A(i, j,k) = ….
(t)       B(i,j,k) = …
    ENDDO
  ENDDO
  DO r=1, N
(u)   C(i,r) = …
  ENDDO
ENDDO
```

*Nid de boucles parfaitement imbriqué*

Instruction (statique)

*Nid de boucles imparfaitement imbriqué*

101

## Introduction

- Contexte
  - Nids de boucles
- Grain très fin
  - Vectorisation
  - SIMD
  - Machines parallèles si grands vecteurs
- Grain moyen
  - Transcription directe en espace d'adressage unique

```
do i=1,N
    do j =1, N
        A(i,j) = B(i,j) +1
    end do
end do

do i=1,N
    A(i,1:N) = B(i,1:N) +1
end do

doall i=1,N
    do j =1, N
        A(i,j) = B(i,j) +1
    end do
end do
```

102

## Plan

- Introduction
- Dépendances

103

## Dépendances

- Soient s et t deux opérations d'un programme P.
- Collision : s et t sont en collision si
  - s et t accèdent au même emplacement mémoire et l'une au moins écrit.
- Dépendance : il existe une dépendance de s vers t si
  - s et t sont en collision

    ET
  - s avant t dans l'exécution de P

  Notation : s -> t
- L'ensemble des dépendances définit le *graphe de dépendances développé* du programme

104

## Typologie

s -> t

- Dépendance de flot - RAW - dépendance vraie
  s écrit, t lit
- Anti-dépendance - WAR
  s lit, t écrit
- Dépendance de sortie - WAW
  s et t écrivent

105

## Parallélisation par tri topologique

- TD 5.1
- Pas directement utilisable pour les boucles

106

## Repères d'instructions et prédicat de séquencement

- Repère d'instruction
  - décrit une opération dans un nid de boucles
  - (nom_inst, i1, i2, …, in) où i1, i2, …, in sont les indices des boucles qui englobent l'instruction
- Prédicat de séquencement

  s, t deux instructions englobées dans n boucles

  L' ordre d'exécution des opérations est :

  *(s, i1, i2, …, in) << (t, i'1, i'2, …, i'n) ssi*

  *(i1, i2, …, in) < (i'1, i'2, …, i'n) dans l'ordre lexicographique*

  *OU*

  *{(i1, i2, …, in) = (i'1, i'2, …, i'n) ET s avant t dans l'ordre syntaxique de P}*
- Exemples : TD 5.2.1

107

## Niveau des dépendances et GDR

- Niveau de la dépendance

  La dépendance

  (s, i1, i2, …, in) -> (t, i'1, i'2, …, i'n)

  est de niveau k si k est le premier indice tel que ik < i'k
  - Dépendance inter-itération - loop carried dependency

  Est de niveau $\propto$ si (i1, i2, …, in) = (i'1, i'2, …, i'n)
  - Dépendance intra-itération - loop-independent dependency
- Graphe de dépendance réduit (GDR)
  - Multi-graphe
  - Nœuds = instructions (statiques)
  - Arcs = dépendances étiquetées par leur niveau
- Exemples : TD 5.2.2

108

## Plan

- Introduction
- Dépendances
- Parallélisation et transformations

109

## Equivalence

- *Définition* Deux programmes sont équivalents s'ils produisent le même résultat
- Transformation
  - Ré-ordonnancement, ré-indexation, parallélisation,…
  - Les instances dynamiques ne changent pas : pas d'instructions ajoutées ou supprimées
  - La structure de contrôle et les indices peuvent changer
- « *Proposition* » Une transformation d'un programme P qui préserve le graphe de dépendances développé fournit un programme équivalent à P
- La réciproque est fausse TD 5.4.1

110

## Syntaxe parallèle

- Constructions séquentielles + doall
  - Par(seq)
    doall i = 1, n
      <Bloc d'instructions séquentiel (i)>
    end do
  - Exécutions parallèle de B1, …, Bn
  - Chaque Bi conserve son séquencement
  - Synchronisation à la fin du doall
  - Traduction immédiate en espace d'adressage unique
- Instructions vectorielles

111

## Instructions vectorielles

- Pour ce cours on se limitera à
  X(a:b:c) = OP ( Y1(a:b:c),…, Yk(a:b:c))

Équivalent à

```
doall i=a, b
    X(i) = OP (Y1(i), …, Yk(i))
end do
```

- Opération « par coordonnées » - $\alpha$–extension

112

## Doall et prédicat de séquencement

- Filtrer l'ordre lexicographique.
- Voir TD 5 - appendice

113

## Parallélisation de boucles

- *Une boucle do de profondeur k est équivalente à une boucle doall si elle ne porte pas de dépendance de niveau k.*
- Preuve :
  - Les ensembles Read et Write des opérations ne sont pas modifiés
  - Le prédicat de séquencement n'est modifié qu'au niveau k
  - Donc le GDD n'est pas modifié (voir TD 5 -appendice)

114

## Tranformations de boucles

- Une dépendance inter-itérations interdit la parallélisation.
- Des transformations (valides) peuvent faire apparaître de nouvelles opportunités de parallélisation.

115

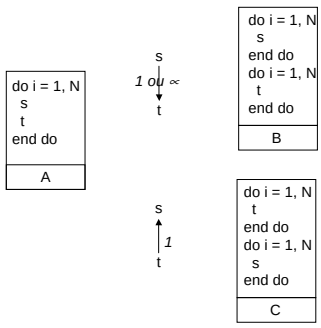
## Distribution de boucles

- Boucle à un seul niveau :
- Exemples TD 5.3

  *Une boucle contenant au moins deux instructions est équivalente à la séquence de deux boucles s'il n'existe pas de cycle de dépendances dans le GDR.*
- Le séquencement doit suivre la direction des dépendances

116

## Distribution de boucles




117

## Distribution de boucles

- En distribuant la boucle, le séquencement correct des opérations en dépendance de niveau 1 est garantie. S'il existe (s,i) -> (t,i') au niveau 1 ou $\propto$ dans A, on a i < i' ou i = i'. Dans B, (s,i) s'exécute avant (t,j) quels que soient i et j.
- La distribution ne crée pas de dépendance nouvelle.
  Les seules dépendances nouvelles possibles dans B sont (s,i) -> (t,i') avec i > i'. Si une telle dépendance existait, la dépendance inverse (flot et anti échangés) existerait dans A de t vers s, donc le GDR de A comporterait un cycle.
- Même raisonnement pour C.

118

## L'algorithme de Kennedy-Allen

Préliminaires

- G un graphe. Une composante fortement connexe de G est un ensemble de sommets tel qu'il existe un chemin entre deux quelconques de ces sommets.
- Il existe des algorithmes pour trouver les composantes fortement connexes maximales (CFCM) $\{\pi_1, \pi_2, \ldots, \pi_m\}$ - Tarjan
- Les arcs de G induisent naturellement un graphe sur $\{\pi_1, \pi_2, \ldots, \pi_m\}$ noté $G_\pi$

119

## L'algorithme de Kennedy-Allen

Procedure codegen (G, k)
Soient $\{\pi_1, \pi_2, \ldots, \pi_m\}$ les CFCM de G énumérées dans un ordre consistant avec le tri topologique de $G_\pi$
*pour* i= 1 à m *faire*
  *si* $G_\pi$ contient un cycle *alors*
  *début*
    générer un DO (pour l'indice adéquat)
    Soit $G_i$ le graphe induit par G sur $\pi_i$ où on ne conserve que les dépendances de niveau supérieur ou égal à k+1
    **codegen ($G_i$, k+1)**
    générer un ENDDO
  *fin*
  *sinon*
    générer une instruction vectorielle pour $\pi_i$ en $\rho(\pi_i)$ - k +1 dimensions, où $\rho(\pi_i)$ est le nombre de boucles contenant $\pi_i$

120

## L'algorithme de Kennedy-Allen

- C'est un algorithme de vectorisation :
  - Les boucles sont distribuées progressivement.
  - À chaque niveau, les instructions individuelles atteintes sont examinées pour leur vectorisation éventuelle.
  - Le parallélisme ne peut provenir que du tri topologique : multi-vectoriel.
- Exemple : TD 5.3.2

121

## Autres transformations

- Echange de boucles - TD 5.4
  - But : Modifier la granularité du parallélisme
  - Moyen : Modifier le parcours de l'espace d'itération pour faire apparaître du parallélisme
- Torsion de boucles (Loop skewing) _ TD 5.5
  - But : faire apparaître du parallélisme.
  - Moyen : Modifier le parcours de l'espace d'itération pour faire apparaître du parallélisme
- Et autres modifications du parcours de l'espace d'itération…
- Agrégation de boucles
  - But : diminuer le surcoût de synchronisation
- Transformations du code séquentiel
  - But : faire apparaître du parallélisme.
- Problèmes
  - Critères d'optimalité : nombre de boucles parallèles, accélération
  - Atteindre l'optimum

122

## Limites de l'analyse de dépendances

- L'analyse automatique implique
  - La résolution de systèmes linéaires en nombres entiers
  - D'où des approximations, qui doivent être conservatives
  - C'est une limite réelle pour les méthodes de transformation avancées
- La compilation séparée bloque l'analyse de dépendances inter-procédurales
- Les accès par indirection interdisent toute analyse de dépendances
  - Pointeurs
  - Indices tableaux : A(L(i))

123

# 6. Parallélisme de données

## Lectures

- **The Data Parallel Programming Model**
  Foundations, HPF Realization, and Scientific Applications
  Lecture Notes in Computer Science, Vol. 1132
  Perrin, Guy-Rene; Darte, Alain (Eds.). 1996
- Nombreux cours et exercices HPF sur le Web

125

## Plan

- Introduction aux langages data parallèles
  - Principe
  - Collections
  - Opérateurs

126

## Principes

- Modèle de programmation SEQ(PAR)
  - Flot de contrôle séquentiel
  - Le parallélisme est décrit par des types de données collections
- Collections
  - Type décrivant un ensemble (au sens non-technique) de données
  - Munies d'opérateurs
    - Parallèles génériques
    - spécifiques

```
real A(N), B(N), C(N)
do i = 0 , n
   A(i) = B(i) + C(i)
end do
```

```
real A(N), B(N), C(N)
A = B + C
```

127

## Collections : typologie

| | Répétition valeurs | Pas de répétition valeurs |
|---|---|---|
| Accès | Tableau | Dictionnaire (Map) |
| Pas d'accès | Liste | Ensemble (Set) |

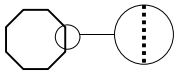
- Exemple : les collections java
  http://java.sun.com/docs/books/tutorial/collections

⚠ – Pas d'exécution parallèle

128

## Collections : typologie

- Les éléments peuvent-ils être des collections ?
  - Décrit plusieurs niveaux de parallélisme



  - Listes de listes
  - Tableau irrégulier (ragged array)

129

## Opérateurs parallèles

- $\alpha$-extension : étendre à la collection un opérateur valide pour le type des éléments

  $\alpha$+{1,2,3}{4,5,6} => {5,7,9}

  – Contraintes de cohérence : tableaux conformes, éléments de même clé,…
- Syntaxe
  – Explicite : *Lisp : !! (+!! A(!!1))
  – Surcharge d'opérateurs A = B+ C
  – Polymorphisme : Fonctions
  – Itérateurs : énumèrent les éléments de la collection auxquels s'applique l' $\alpha$-extension

130

## Opérateurs parallèles

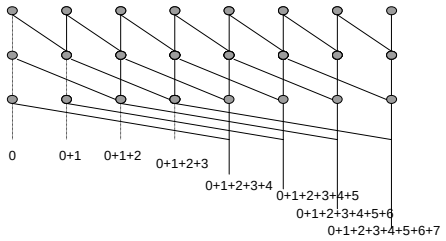
- Réduction
- Préfixe parallèle

131

## Préfixe parallèle

```
pour j := 0 to lg(n-1)
  pour i := 2^j to n-1 (en parallèle)
    s[i] := f(s[i-2^j], s[i])
```

f est associative et commutative




132

## Opérateurs spécifiques

- Point de vue collection pur
  - insertion, suppression
  - Listes : tri
  - Dictionnaires : accès par clé
- Tableaux : permutation, combinaison
  - Gather :
    B = gather (A,L) : pour tout i, B(i) = A(L(i))
  - Scatter :
    B = scatter (A,L) : pour tout i, B(M(i)) = A(i)
    - Contraintes de conformité
    - Collision : sens si M(i) = M(j) ?
    - Si M est injective, trie A suivant M

133

## Opérateurs spécifiques

- Tableaux : sélection
  T(a:b:c) est le tableau des
  T(a + kc) pour 0 <= k <= (b-a)/c
- Et beaucoup d'autres primitives

134

## Plan

- Introduction aux langages data parallèles
  - Principe
  - Collections
  - Opérateurs
- High Performance Fortran (HPF)
  - Itérateurs
  - Directives de placement
  - Introduction à la compilation
  - Extensions

135

## Histoire

- Fortran 77 : séquentiel
- Fortran 90 : Vectoriel + encapsulation, polymorphisme
- Proposition de spécification réalisée par le HPF forum
  - HPF 1 : 94
  - HPF 2 : 97

136

## L'instruction FORALL

- Syntaxe
  FORALL (*index-spec-list* [, *mask-expr*]) *forall-assignment*
  - *index-spec-list* : a1:b1:c1,a2:b2:c2, …, an:bn:cn
  - *forall-assignment* : affectation
- Sémantique : itérateur sur un sous-ensemble d'un pavé de $Z^n$
  - Evaluer l'ensemble des indices valides : ceux sur lesquels le masque est vrai
  - Pour tous les indices valides, évaluer le membre droit
  - Pour tous les indices valides, réaliser l'affectation

137

## L'instruction FORALL

Forall (i=2:5:2, j=1:2)
A(i,j) = A(i+2,j) + B(i, j)

***synchronisation***

2,1
2,2
4,1
4,2

4,1
4,2
6,2
6,2

```
Do i=2:5:2
  Do j= 1,2
    Tmp(i,j) = A(i+2,j) + B(i,j)
  End do
End do
Do i=2:5:2
  Do j= 1,2
    A(i,j) = tmp(i,j)
  End do
End do
```

Code séquentiel équivalent

138

## Exemples

| | |
|---|---|
| Affectation<br><br>Forall (i=1:2,j=1:3) A(i,j) = i+j | Multidiffusion<br><br>Forall (i=1:2, j=1:4) Y(i) = A(i,i) |
| Permutation<br><br>Forall (i=2:5) X(i) = X(i-1)<br><br>Forall (i=1:4) Y(i, L(i)) = Z(i)<br>L = /1, 2, 2, 4/ | Sélection<br><br>Forall (i=1:3, X(i) != 0) Y(i) = 1/X(i) |

139

## Collisions

- En relation avec la vision espace d'adressage unique
- Les affectations multiples ne sont pas admises – le comportement est non-prédictible
- La correction ne peut pas être vérifiée par le compilateur
  - Incorrect : forall (i=1:n, j=1:n) A(i+j) = B(i,j)
  - Peut être correct : forall (i=1:n) A(L(i)) = B(i)
  - Toujours correct : forall (i=1:n) A(i) = B(L(i))

140

## Jacobi

**SEQ(PAR) :**

```
u = 0.0
unew = 0.0
err = tol + 1.0
iter = 0
DO WHILE (err > tol .and. iter < NITER)
  iter = iter + 1
  FORALL ( i=1:nx-1, j=1:ny-1 )
    unew(i,j) = (u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)+dx*dx*f(i,j)) / 4
  END FORALL
  err = MAXVAL( ABS(unew-u) )
  u = unew
END DO
```

141

## Fonctions

- Quelles conditions doit vérifier la fonction FOO pour que l'appel

  Forall (i=1:n) A(i) = FOO(i)

  soit cohérent avec la sémantique du forall ?

- FOO ne doit pas produire d'effet de bord
  - Vérifié par le compilateur
  - Contraintes syntaxiques sur-restrictives
    - Pas de pointeurs
    - Pas de paramètres OUT

142

## La construction FORALL

- Syntaxe

  FORALL (*index-spec-list* [, *mask-expr*])
  *forall-body-list*
  END FORALL

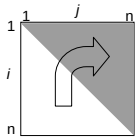
  - *forall-body* : f*orall-assignment* ou FORALL ou WHERE
- Sémantique
  - Imbrication de forall : espace d'itération non rectangulair
  - Exécution dans l'ordre syntaxique des instructions du *forall-body*

143

## La construction FORALL

- Imbrication

```
Forall i=1:n
   Forall j= i+1:n
     A(i,j) = A(j,i)
   End Forall
End Forall
```




- Séquence

```
Forall i=2:4
   A(i) = A(i-1) + A(i+1)
   C(i) = B(i) + A(i+1)
End Forall
```

*synchronisation*

*synchronisation*

*synchronisation*

144

## Placement des données

- Architecture matérielle cible : espaces d'adressages multiples
- Le placement des données détermine
  - Le placement des calculs : par exemple Owner Computes Rule
  - Donc la distribution de charge
  - Le volume et le nombre de communications : surcoût
- Choix HPF :
  - Placement explicite : le compromis localité/équilibrage de charge/surcoût est contrôlé par l'utilisateur
  - Par directives : pas un type – difficultés d'interprétation pour le passage des paramètres
  - Alignement + Distribution

145

## Alignement

- Position relative des tableaux
  - Indépendante du nombre de processeurs
- Syntaxe

!HPF$ ALIGN array(*source-list*) WITH target (*subscript-list*)

  - *Subscript* est
    - Une fonction d'UNE variable de source-list
    - Un triplet a:b:c
    - *
- Sémantique
  - Les éléments alignés sont placés sur le même processeur
  - * en destination indice la réplication
  - * en source indique la séquentialisation

146

## Alignement

*Individuel*

```
REAL A(3,3), B(3,3), X(2)
!HPF$ ALIGN A(i,j) WITH B(j,i)
!HPF$ ALIGN X(i) WITH A(i+1,2)
```

*Séquentialisé*

```
REAL A(3,3), Y(3)
!HPF$ ALIGN A(i,j) WITH Y(i)
Ou
!HPF$ ALIGN A(i,*) WITH Y(i)
```

*Répliqué*

```
REAL A(3,3), Y(3)
!HPF$ ALIGN Y(i) WITH A(i,*)
```

147

## Distribution : la directive PROCESSORS

- Syntaxe

  !HPF$ PROCESSORS array-decl
- Sémantique

  Définit une géométrie rectangulaire de processeurs limitée par le nombre de processeurs physiquement disponibles
- Exemple A 64 processeurs

  !HPF$ PROCESSORS PC(4,4,4) ou PC(2,4,8) etc.

  !HPF$ PROCESSORS PP(8,8) ou P(4,16) etc.

  !HPF$ PROCESSORS PL(64)

148

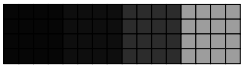
## Distribution : la directive DISTRIBUTE

- Syntaxe

  !HPF$ DISTRIBUTE array (*dist-format-list*) [ONTO procs]
- Sémantique
  - Chaque dimension du tableau est distribuée suivant le schéma correspondant de dist-format-list
  - *dist-format* est
    - BLOCK : fragments contigus de taille égale
    - CYCLIC : circulaire
    - BLOCK(k) : fragments contigus de taille k
    - CYCLIC(k) : circulaire des blocks de taille k
    - * : séquentialisé
  - *procs* est un tableau de processeurs (calculé par le compilateur si absent)
- Parallélisation possible des calculs associés aux données distribuées sur des processeurs différents

149

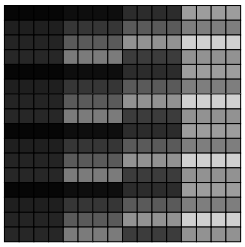
## Distribution : la directive DISTRIBUTE

| | |
|---|---|
| REAL A(16)<br>!HPF$ PROCESSORS P(4)<br>!HPF$ DISTRIBUTE A (*block)*<br>  | REAL A(16)<br>!HPF$ PROCESSORS P(4)<br>!HPF$ DISTRIBUTE A (*cyclic)*<br>  |
| REAL A(16)<br>!HPF$ PROCESSORS P(4)<br>!HPF$ DISTRIBUTE A (*cyclic(2))*<br>  | REAL B(4,16)<br>!HPF$ PROCESSORS P(4)<br>!HPF$ DISTRIBUTE B(*,*block)*<br>  |

150

## Distribution : la directive DISTRIBUTE

```
REAL B(16,16)
!HPF$ PROCESSORS P(4,4)
!HPF$ DISTRIBUTE B(cyclic,block)
```



Remarque



## Alignement et distribution

- Chaque tableau a une cible d'alignement ultime
- Seules les cibles ultimes peuvent être distribuées
- Des templates peuvent être déclarés pour simplifier l'alignement

!HPF$ TEMPLATE *array-decl*



## Alignement et distribution

- Les distributions ne sont pas fermées vis-à-vis
  - De l'alignement
  - Des sections de tableaux

```
REAL A(12), B(6)
!HPF$ PROCESSORS P(4)
!HPF$ ALIGN B(i) WITH A(2*i –1)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
```

Ou A(1:12:2)

## Conclusion

- Echec en tant que normalisation : n'a jamais été intégré dans le standard Fortran
- Concurrent OpenMP
- Explication socio-économique
  - L'essentiel des résultats pour une compilation efficace existe
  - Pas de marché
  - Problème de leadership
- Une niche au Japon, dans le programme Earth Simulator
  - Gordon Bell prize 2002 !

154

# 7. Parallélisme de contrôle

## Lectures


- Al Geist, A. Beguelin, J. Dongarra et al. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press 1994. Version en ligne : http://www.netlib.org/pvm3/book/pvm-book.html
- M.J. Quinn. Parallel Programming in C with MPI and OpenMP. McGrawHill 2003.


156

## Plan

- Introduction

157

## Introduction

Le seul point commun : PAR(SEQ)

| Espaces d'adressage multiples | Espace d'adressage unique |
|---|---|
| • Programme = processus<br>• Fonctionnalités<br>• API bas niveau<br>• Langages de programmation | • Programme = thread<br>• Fonctionnalités<br>• API bas niveau<br>• Langages de programmation |

158

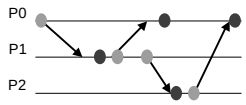
## Parallèle ou Réparti ?

- PAR(SEQ) peut aussi décrire le calcul distribué
- Les spécificités du parallèle
  - Architecture fortement couplée, permettant un parallélisme à grain fin et la prise en compte des performances
  - Modèle de programmation SPMD : Single Process Multiple Data
    Code unique paramétré par un identifiant de tâche
  - Pas de prise en compte des défaillances = pas de tolérance aux fautes

159

## Modèle de programmation à passage de messages

- Espaces d'adressage multiples
  - Toutes les données sont privées
  - Toutes les tâches peuvent communiquer
  - La communication et la synchronisation sont explicites
- La réalisation d'un modèle de calcul processus+ canaux
  - Identique : Messages typés-délimités et non flot comme en sockets SOCK_STREAM
  - Différence essentielle : pas de réception implicite




160

## Plan

- Introduction
- Passage de messages

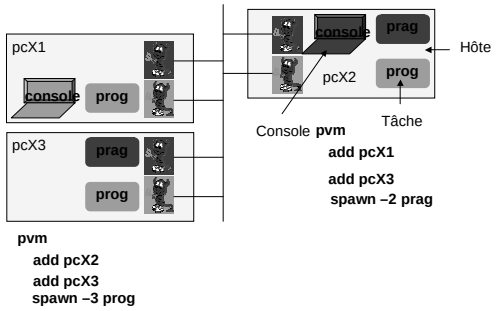
161

## Parallel Virtual Machine

- Composants
  - Une librairie de passage de messages
  - Un environnement de gestion d'une machine parallèle virtuelle dynamique
- Une API et des implémentations
  - Domaine public : protocole sous-jacent TCP
  - Propriétaires : protocole sous-jacent propriétaire
  - Pour la plupart des machines parallèles et les grappes
- Histoire
  - Sous la direction de Jack Dongarra -> Top 500
  - PVM 1.0 : 1989, interne Oak Ridge National Laboratory
  - PVM 2.0 : 1991, University of Tennessee
  - PVM 3.4.5 : 2004

162

## La machine virtuelle : version interactive




163

## La machine virtuelle

- Ajout automatique de machine :
  - pvmd <hostfile>
  - Une machine par ligne
- L'ajout d'une machine crée les fichiers
  - /tmp/pvmd.<uid>
    - numéro de port
    - son existence interdit de relancer le démon
  - /tmp/pvml.<uid>
    - stdin et stdout des tâches lancées par la console

164

## Création dynamique de tâches : pvm_spawn

int numt = pvm_spawn( char *task, char **argv, int flag, char *where, int ntask, int *tids

- task : nom de l'exécutable . Path par défaut $HOME/pvm3/bin/$PVM_ARCH/
- argv : arguments de l'exécutable
- flag : options de placement somme de
  - PvmTaskDefault 0 : quelconque
  - PvmTaskHost 1 : *where* spécifie une machine hôte
  - …
- ntask : Nombre de copies de l' exécutable à démarrer
- tids : Tableau[*ntask*] retourne les tids des processus PVM spawnés.
- numt Nombre de tâches spawnées.

165

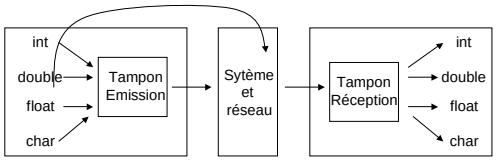
## Identification

int tid = pvm_mytid(void)
- Identifiant unique dans la MV
- « enrôle » la tâche dans la MV

int tid = pvm_parent(void)
- Identifiant du parent dans la MV
- Eventuellement PvmNoParent

166

## Communication




Formattage et copie *Marshalling* | Emission | Réception | Formattage et copie *Marshalling*

167

## Sérialisation

- Quel support pour la communication
  - de types complexes
  - d'objets pour les appels de méthodes à distance
  - pointeurs sous-jacents

168

## Emission : fonctions de base

- Allocation d'un tampon
  Int bufid = pvm_initsend(int encoding)
  - Encoding : méthode de formattage des données
  - PvmDataInPlace : pas de copie, les données ne doivent pas être modifiées jusqu'au retour de l émission correspondante
- Formattage – sérialisation
  int info = pvm_pckxxx(<type> *p, int cnt, int std)
  - Limité aux sections de tableau de types primitifs
  - Un même tampon peut recevoir plusieurs pck sur des types différents
- Emission
  int info = pvm_send(int tid, int msgtag)
  - tid : identifiant destinataire
  - msgtag : un typage utilisateur
  - L'émission est non-bloquante : transfert vers l'agent de communication
  - info < 0 indique une erreur.
- Pour le formattage et l'émission, le tampon est implicite

169

## Réception : fonctions de base

- Réception
  int bufid = pvm_recv(int tid, int msgtag)
  - tid : identifiant émetteur ; -1 = non spécifié
  - msgtag : un typage utilisateur; -1 = non spécifié
  - bloquante : attente sur un message émis par tid avec le tag msgtag.
- Formattage – désérialisation
  int info = pvm_upckxxx(<type> *p, int cnt, int std)
  - Copie dans le tableau p les données du buffer, avec le pas std

170

## Réception

- Canaux FIFO
  - Si A envoie deux messages successifs de même tag à B, les réceptions s'effectuent dans l'ordre d'émission
  - Rien n'est garanti sur l'ordre de réception de messages provenant de processeurs différents
  - Le non déterminisme est possible
  - Le blocage en attente infinie est possible
  - Compromis
    - Programmation plus simple : Identifier les messages par tid et msgtag
    - Introduit des synchronisations inutiles

| A | B | C |
|---|---|---|
| envoyer 2 à B | recevoir(-1) | envoyer 3 à B |

Peut être 2 ou 3

171

## Réceptions

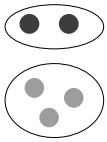
- pvm_probe
  - Crée un tampon
  - >0 si un message est arrivé
  - Évite l'attente, ou permet de ne pas recevoir
- pvm_bufinfo
  - Information sur le message arrivé
- pvm_nrecv : réception non bloquante

```
tant que (pvm_probe (…) == -1)
    travail utile
Ftq
info = pvm_bufinfo (…)
flag = attendu(info)
traitement suivant flag
```

172

## Synchronisation

- Groupes de tâches
  - Syntaxe : voir group operations
  - Numéros unique séquentiels
  - Fonctions de conversion tid <-> inum
  - Seule géométrie prédéfine anneau
    - Conversion en géométrie nD triviale
    - En 2D : xCoord = inum/N, yCoord= inum%N



- Barrière

  int info = pvm_barrier (char*group, int count)
  - Toutes les tâches appelantes attendent que count tâches aient appelé la fonction

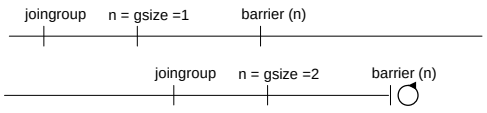
173

## Synchronisation

- Un piège classique

  Size = pvm_gsize(mygroup)

  Info = pvm_barrier(gsize)
- Peut aboutir à une attente infinie




174

## Passage de message vs Parallélisme de données

- Optimisation fine des communications
  - Recouvrement communication-calcul
  - Minimiser le volume et le nombre de communications
  - Distributions irrégulières des données
- Mais
  - Le parallélisme n'est pas explicite
  - Indéterminisme
    - Non reproductibilité des erreurs
    - Debug et maintenance difficiles
  - Code fastidieux

175

## MPI

- Le standard de facto en librairies de communication
- Sérialisation de types complexes
- Géométries intégrées
- Uniquement une librairie de communication
  - Pas de création dynamique de processus
  - Toute la gestion de processus est reportée sur l'exécutif (environnement) : mpi-run
  - Pas de tolérance aux fautes. La réalisation de la tolérance aux fautes fait l'objet de nombreux projets.

http://www.mpi_forum.org

176

## D'autres modèles de communication

- Contexte passage de message
  - Messages Actifs : réception implicite, activation d'un handler
- Contexte processus concurrents
  - Remote Procedure Call : http://www.faqs.org/rfcs/rfc1050.html
  - Remote Method Invocation : Java RMI
- Rendez-vous : Communicating Sequential Process (Hoare)

177

## Retour sur l'introduction

Le seul point commun : PAR(SEQ)

Espaces d'adressage multiples

- Programme = processus
- Fonctionnalités
  - Identification
  - Communication
  - Synchronisation
- API bas niveau
  - PVM, MPI
- Langages de programmation
  - Pas d'usage courant

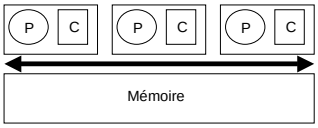
178

## Plan

- Introduction
- Passage de messages
- Mémoire partagée

179

## Les multiprocesseurs symétriques - principe




- Tout processus peut accéder à toute adresse dans l'espace partagé
  - Accès = chargement/rangement (LD/ST)
- Problèmes :
  - Sémantique des accès concurrents
    - D'autres opérations, liées à l'ordonnancement des accès.
  - Réalisation de la cohérence de cache
  - Atomicité

180

## Motivation pour les architectures SMP

- Economiques
  - Intégrer des systèmes matériels standard pour fournir des performances de milieu de gamme
  - Evolutions actuelle
    - Intégration de noeuds SMP dans des architectures très hautes performances.
    - Multiprocesseur en un circuit et circuits embarqués multiprocesseurs
- Simplicité de programmation
  - Il n'est pas obligatoire de partitionner les données/les calculs
  - Séduisant pour des applications irrégulières
- Objectif du cours
  - Montrer les limites de cette approche

181

## Consistance séquentielle

- Comment établir un ordre entre les instructions de lecture-écriture effectuées par différents processeurs ?

```
PX      PY
A=1     Lire B
B=2     Lire A
```

Résultats conformes à la consistance séquentielle

X1 X2 Y1 Y2 1, 2
X1 Y1 Y2 X2 1, 0
Y1 X1 X2 Y2 1, 0
Y1 Y2 X1 X2 0, 0

Résultat non conforme à la consistance séquentielle : (0,2)

0 : Y2 < X1
2 : X2 < Y1
Ordre séquentiel pour chaque code :
X1 < X2 et Y1 < Y2
Contradiction

182

## Consistance séquentielle

- Comment établir un ordre entre les instructions de lecture-écriture effectuées par différents processeurs ?
- "A multiprocessor is *sequentially consistent* if the result of any execution is the same as if

L'ordonnancement dans le temps physique n'est pas observable

183

## Consistance séquentielle

- Comment établir un ordre entre les instructions de lecture-écriture effectuées par différents processeurs ?
- "A multiprocessor is *sequentially consistent* if the result of any execution is the same as if
  - the operations of all the processors were executed in some sequential order

L'entrelacement n'est pas déterminé

184

## Consistance séquentielle

- Comment établir un ordre entre les instructions de lecture-écriture effectuées par différents processeurs ?
- "A multiprocessor is *sequentially consistent* if the result of any execution is the same as if
  - the operations of all the processors were executed in some sequential order
  - and the operations of each individual processor appear in this sequence in the order specified by its program."

[Lamport, 1979]

Les accès mémoire de chaque programme sont rendus visibles dans son ordre syntaxique

185

## Consistance séquentielle

- Comment établir un ordre entre les instructions de lecture-écriture effectuées par différents processeurs ?
- "A multiprocessor is *sequentially consistent* if the result of any execution is the same as if
  - the operations of all the processors were executed in some sequential order
  - and the operations of each individual processor appear in this sequence in the order specified by its program."

[Lamport, 1979]

L'ordonnancement dans le temps physique n'est pas observable

L'entrelacement n'est pas déterminé

Les accès mémoire de chaque programme sont rendus visibles dans son ordre syntaxique

**Modèle minimal pour l'analyse et la programmation**

186

## Consistance séquentielle et microprocesseurs

- Non conformité des ordres
  a. Syntaxique du programme
  b. De lancement de instructions – vis à vis du processeur
  c. D'accès mémoire – visibilité vis-à-vis des autres processeurs
- La combinaison compilateur/microprocesseur
  - Garantit le respect des dépendances données et contrôle pour chaque programme séquentiel du point de vue du processeur
  - Ne garantit rien sur l'ordre d'accès à des variables différentes
    - Le réordonnancement dynamique est fondamental pour les performances architectures RISC et superscalaires
  - Ne garantit rien sur l'ordre de visibilité à l'extérieur
    - Tampons et caches

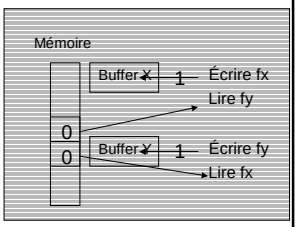
187

## Exemples

- Exclusion mutuelle

```
PX          PY
fx = 1      fy = 1
If (!fy)    If (!fx)
  code        code
```

- Tampon d' écriture
- Ordre séquentiel = ordre de lancement != ordre de visibilité
- PX et PY exécutent code




188

## Exemples

- Producteur-consommateur

```
PX          PY
data=2      while (!head);
head=1      val=data
```

lancer lire head
lancer lire data
retour data =0
écrire data
retour head=1

- Temps d'accès non uniforme
- Ordre séquentiel = ordre de lancement != ordre de visibilité
- Ou bien ordre séquentiel != ordre de lancement
  - Pas de dépendance de données !

```
PX          PY
head=1      val=data
data=2      while (!head);
```

189

## Consistance et microprocesseurs

- La CS est incompatible avec les performances sur les µprocs actuels
- Consistance relâchée
  - La consistance forte est inutile
    - Dans une section critique
    - Pour l'accès aux données privées
    - ….
  - Documenter précisément quel(s) modèle(s) implémente un µproc
- Support matériel pour la CS : instructions de « synchronisation »
  - Synchroniser la vue de la mémoire
  - Typiquement tous les accès mémoire pendants sont terminés et rendus visibles à l'extérieur avant d'exécuter les instructions suivantes dans l'ordre syntaxique. Ex. PowerPC SYNC
  - N'implique pas que les lignes de cache sont recopiées en mémoire !
  - Coût élevé : vidange du pipeline + éventuellement transactions bus +… au pire accès disques

190

## Cohérence de cache – caches des processeurs

- Les caches sont déterminants pour les performances monoprocesseurs, à fortiori pour les multiprocesseurs
  - Le temps d'accès des DRAM évolue beaucoup moins vite que la fréquence des processeurs + le micro-parallélisme contribue à la pression sur le débit mémoire
  - L'accès par blocs d'adresse contiguës est beaucoup plus efficace que l'accès aléatoire
  - ➢ Les caches ne sont utiles que lorsqu'il existe de la localité spatiale et temporelle

La valeur visible est celle du cache

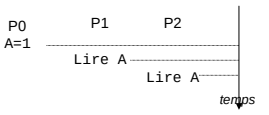
| | Succès | Echec |
|---|---|---|
| R | Aucune | • Acquérir ligne = recopier en mémoire la ligne éliminée si modifiée<br>• Action succès |
| W | Marquer la ligne à modifiée | |

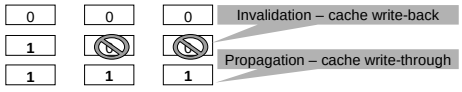
*Ecriture allouée, réécriture (write-back)*

191

## Objectif de la cohérence de cache

- Gestion des copies multiples d'une même adresse mémoire

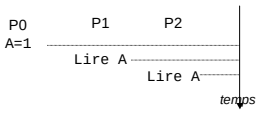



192

## Relation avec la consistance

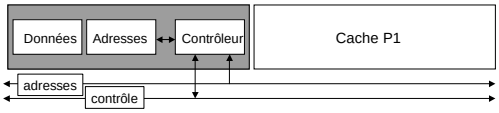
- La cohérence de cache assure la réalisation de l'ordre prescrit par la consistance pour les accès à une même variable
- Le programme de haut niveau + la compilation + les options de consistance assurent que P0 écrit avant que P1 et P2 ne lisent
- La cohérence de cache assure que P1 et P2 voient 1
- Mais un autre ordre séquentiellement consistant est toutes les lectures avant les écritures, ou P1 lit puis P0 écrit puis P2 lit.




193

## Réalisation de la cohérence de cache

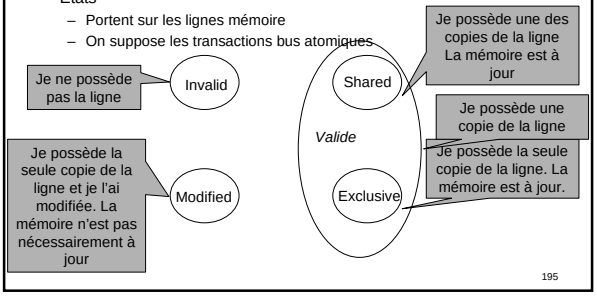
- Entièrement en matériel : invisible au niveau de la programmation en langage-machine
- Architectures à bus : espionnage de bus
  - Extension de la cohérence I/O
  - Les transactions bus sont visibles par tous les processeurs.
  - Les contrôleurs de cache peuvent écouter les transactions bus et effectuer les actions nécessaires sur les évènements importants




194

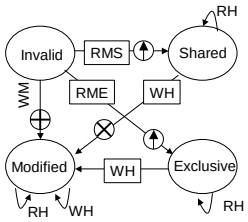
## Le protocole MESI

- Entièrement en matériel => automate
- Objectif : minimiser les transactions bus
- Etats
  - Portent sur les lignes mémoire
  - On suppose les transactions bus atomiques




195

## Le protocole MESI : conditions processeur




*Condition*

RH = read hit
RMS = read miss shared
RME = read miss excl.
WM = write miss
WH = write hit

*Transaction bus*

Cache block fill

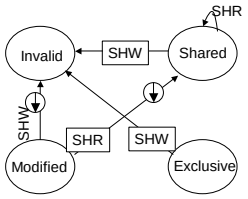
Read with-intent-to modify

Invalidate

•Succès en écriture :
- Si partagé, invalidation des autres copies
- Etat futur modifié : les écritures suivantes ne provoquent pas de transaction bus

196

## Le protocole MESI : conditions en espionnage




*Condition*

SHR = snoop hit on read
SHW = snoop hit on write
or
read with-intent-to modify

*Transaction bus*

Snoop push

• Soop Hit = détection sur la ligne d'adresse d'un accès à une ligne que je possède
• SHR : un autre lit la ligne
• SHW : un autre écrit la ligne que je possède ou invalidation

197

## Cohérence et performances

- Trafic de cohérence
  - L'écriture d'une donnée partagée invalide toutes les copies. Tous les processeurs qui lisent la donnée doivent effectuer une transaction de lecture.
- Faux partage
  - Un processeur écrit dans une partie d'une ligne de cache.
  - Un autre processeur écrit dans une autre partie d'une ligne de cache
  - Même si chaque processeur ne modifie que sa « partie » de la ligne de cache, toute écriture dans un cache provoque une invalidation de « toute » la ligne de cache dans les autres caches.

198

## Synchronisation

- Problème de l'atomicité
- P0 : s = s+1 ; P1 : s = s+2 ne produit pas nécessairement s = 3 même sur une machine séquentiellement consistante car l'opération lecture/écriture n'est pas atomique (LD puis ST)
- Implémentation efficace de l'accès exclusif à une variable

199

## Atomicité

- Code incorrect

```
tant que (verrou != 0) ;
verrou = 1
<section critique>
verrou = 0
```

- Sur une mémoire séquentiellement consistante avec cohérence de cache

- Code correct

```
tant que
   !(TestAndSet(verrou));
<section critique>
```

- TestAndSet (v)

```
resultat= faux
si v==0
    v=1
    resultat=vrai
fsi
```

- ET : un seul processeur voit le résultat vrai

200

## Implémentation des transactions atomiques (1)

- Une transaction bus atomique n'est pas possible. Instructions spécifiques réparties. Exemple PPC
- Lwarx ad, Rd :
  - Rd <- Mem(ad) et positionne l'unique RESERVE bit du processeur
  - Toute écriture par un processeur à cette adresse (parallélisme) et tout stwcx à une adresse quelconque (concurrence) positionnent RESERVE à faux
- Stwcx op, ad
  - Ecrit l'opérande immediat ou registre si RESERVE. Le succès où l'échec sont récupérables

201

## Implémentation des transactions atomiques (2)

- Implémentation du TestAndSet

```
lwarx    verrou, R1
cmp      R1, 0
bfalse   echec
stwcx    1, verrou
bfalse   echec
….
```

202

## SMP Conclusion 1

- Correction et performances pour la programmation par threads
- **Utiliser les primitives des librairies de threads : verrous, sémaphores, barrières,…**
- **S'informer sur leurs performances**

203

## SMP Conclusion 2

- La programmation utilisateur doit gérer le conflits d'accès au cache vrais et de faux partage
- Pour les problèmes irréguliers, le partitionnement explicite des données est nécessaire
- La programmation en espace d'adressage unique n'est que marginalement simplifiée par rapport à celle en espaces d'adressages multiples

204