

Introduction à la programmation des ordinateurs parallèles

Ronan.Keryell@enstb.org

—

Département Informatique

École Nationale Supérieure des Télécommunications de Bretagne

Master Recherche 2^{ème} année Informatique de Rennes 1 — ENSTBr

ISIA — ENSMP

octobre 2006–février 2007

Version 1.7

Extraire du parallélisme

2

Découpage en tâches (typique MIMD) \equiv faire des choses différentes sur des données différentes :

```

pour  $i = 0$  à  $n - 1$  faire
  tâches parallèles
     $x = a + b.v[i] + c.v[i]^2 + d.v[i]^3$ 
    ||
     $y = e. + f.v[i] + g.v[i]^2$ 
    ||
     $z = v[i]^4$ 
  fin tâches parallèles
   $vv[i] = x + z.y$ 
fin pour

```



Exemple de calcul de polynômes de vecteurs (livre « Initiation au parallélisme, concepts, architectures et algorithmes », Marc GENGLER, Stéphane Ubéda & Frédéric Desprez)

```

pour  $i = 0$  à  $n - 1$  faire
   $vv[i] = a + b.v[i] + c.v[i]^2 + d.v[i]^3 + e.v[i]^4 + f.v[i]^5 + g.v[i]^6$ 
fin pour

```

Calcul avec parallélisme de donnée (typique SIMD) \equiv faire en parallèle la même chose sur des données différentes :

```

pour  $i = 0$  à  $n - 1$  faire en parallèle
   $vv[i] = a + b.v[i] + c.v[i]^2 + d.v[i]^3 + e.v[i]^4 + f.v[i]^5 + g.v[i]^6$ 
fin pour

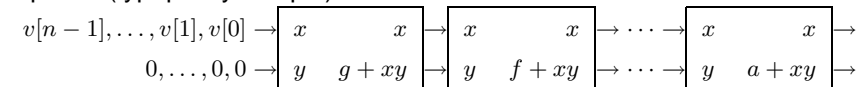
```



Extraire du parallélisme

3

Pipeline (typique systolique) \equiv travail à la chaîne :



7 étages de pipeline (7 processeurs) traitant 1 flux de plusieurs données.



Parallélisme de données :

- Régularité des données
- Même calcul à des données distinctes

Parallélisme de contrôle :

- Fait des choses différentes

Parallélisme de flux : pipeline

- Régularité des données
- Chaque donnée subit séquence de traitements



Degré du parallélisme

- Nombre d'opérations exécutées simultanément
- Idée du nombre de processeurs nécessaires
- Si moins de processeurs, repliement du parallélisme
- Peut varier au cours de l'exécution
- Si trop de processeurs : inactivité ↗



Taille moyenne des tâches élémentaires (en nombre d'instruction, taille mémoire, temps)

Gros grain → grain fin :

- ▶ Programmes
- ▶ Procédures & fonctions
- ▶ Instructions
- ▶ Expressions
- ▶ Opérateurs
- ▶ Bits

Choix en relation avec l'architecture cible

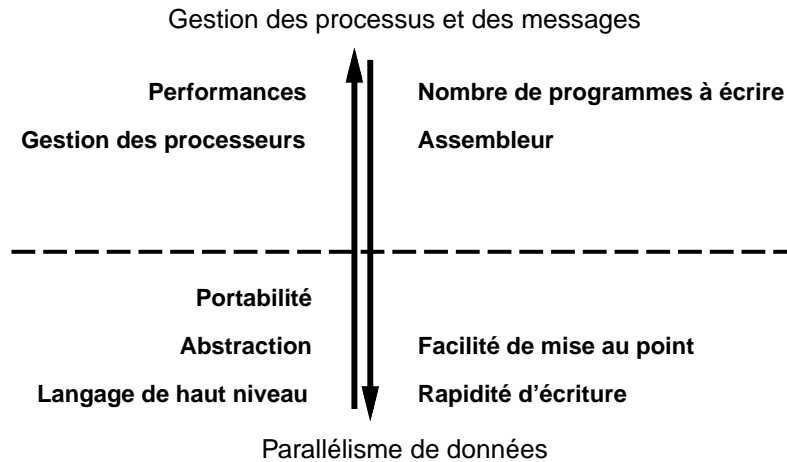


Programmation des machines parallèles

Principalement 2 niveaux :

- Gestion totale du parallélisme « à la main » au niveau des processeurs
- Langage de haut niveau abstrayant le parallélisme de la machine





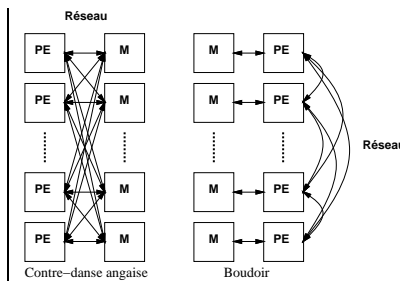
Mode de couplage avec la mémoire

10

Solutions possibles :

- exploiter la localité des accès
- couper la mémoire en morceau

Mais comment relier processeurs et mémoire(s) ?



Couplage fort contre couplage faible.



Rêve de l'utilisateur λ : ne voir qu'une grosse machine SISD
 \rightsquigarrow énorme mémoire globale.

Problèmes (à défaut de SISD...) :

- sortance des portes logiques bornées (temps en $\mathcal{O}(\mathcal{M})$) = : amplifications exponentielles nécessaires (temps en $\mathcal{O}(\log \mathcal{M})$)
- temps de propagation de l'ordre de $\frac{2}{3}c$ dans les fils
- on peut augmenter le débit mais pas diminuer la latence
- conflits d'accès à une même case par plusieurs PES

Couplage fort \equiv mémoire partagée

11

Avantages :

- plus simple à programmer (mais non sans surprise...)
- logiciel plus simple : chaque PE voit toutes les autres mémoires
- découplage entre nombre de PES et nombre de mémoires
 - beaucoup de bancs \rightsquigarrow fort débit
 - nombres premiers pour éviter des conflits, etc.
- mémoire maximale vue par **chaque** processeur, utile pour applications demandant beaucoup de mémoire



Inconvénients :

- optimisation du cas pire : « on ne sait rien et on fait le maximum »
- matériel complexe car débit élevé en continu
- cohérence mémoire délicate si cache
- difficile si beaucoup de PES

Mémoire souvent réalisée de manière structurée : bancs.

Pour grosses machines vectorielles ou les multiprocesseurs (mainframes ou stations de travail), BSP, à faible nombre de processeurs : CRAY Y-MP C916 : 1024 bancs pour 250 Go/s



Inconvénients :

- si accès très dispersés : lent
- placement (distribution et alignement) important
- temporaires pour stocker les données communiquées
- nécessite un logiciel plus complexe.

Toutes machines SIMD sauf BSP : CM-2, MP-1, ILLIAC IV, OPSILA.

Machines MIMD avec beaucoup de PES : DELTA, CM-5, TRANSPUTERS.



Avantages :

- optimisation du meilleur cas : « on sait tout et on fait le minimum », en ayant tout localement
- rapide si placement des données correct
- « compilation » des communications : cache logiciel, fusion de messages
- assez simple même si beaucoup de PES



Éviter les conflits sur des accès typiques (lignes, colonnes, diagonales si possible) :

- $(i, j) \rightarrow (ui + vj) \bmod N$ BSP
- $(i, j) \rightarrow i \oplus j \bmod N$ STARAN
- $(i, j) \rightarrow AI \oplus BJ \bmod N$ XOR scheme
- carrés magiques $N \times N$ avec des nombres de 1 à N

Problème : temps de calcul \rightsquigarrow faire simple...



MP : Multi-Processeur... à mémoire partagée ((MP)²).

Au niveau système :

SMP

- : symétrique
Chaque processeur peut accéder au matériel d'E/S (sémaphores...)

AMP

- : asymétrique
Seul 1 processeur accède au matériel. Les autres sous-traitent

SMP plus compliqué (SunOS 5) mais plus efficace que l'AMP (SunOS 4) pour les E/S.



Simulation d'une mémoire partagée

18

- Détourner chaque accès mémoire et éventuellement faire une communication : *run-time resolution*
Solution de dernier recours car très lente...
- Bien délimiter et isoler les zones à problèmes : insérer les communications pour les données qui manquent en local.
- Gérer le problème au niveau page du système d'exploitation (MMU pour faire de la mémoire virtuellement partagée) ~> transparent au programmeur
Problème lors du partage des pages en écriture : chaque PE va transférer la page lors de chaque écriture ~> ping-pong de page pour 1 octet...

De manière générale, plus le problème est irrégulier, les communications aussi...



Multiplication de matrice :

$$B = A \times A$$

Modèle d'exécution et couplage mémoire

A & B



- Mémoire partagée :

- Mémoire distribuée sur les PEs :

Temporaires



Communication dans 1 temporaire (voire 2...) nécessaire pour des calculs répartis aussi par bloc.



Programmation SPMD pour mémoire distribuée

19

- But : programme qui s'exécute sur plusieurs processeurs
- Pas d'accès simple aux données des autres processeurs
- Distribuer les données et faire suivre les calculs pour exprimer du parallélisme
- Simplification possible : règle des écritures locales (*owner-compute rule*)
 $A = F(B, C)$
F est calculé sur le nœud de la machine où est stocké A



```
forall( $i, \tilde{L}i \leq 0$ )
     $X(\tilde{S}_Xi) = f(Y(\tilde{S}_Yi), Z(\tilde{S}_Zi), \dots)$ 

devient

prog(p)
forall( $i, \tilde{L}i \leq 0$ )
    send $_{Y(\tilde{S}_Yi)}(p)$ 
    send $_{Z(\tilde{S}_Zi)}(p)$ 
    receive $_{Y(\tilde{S}_Yi)}(p)$ 
    receive $_{Z(\tilde{S}_Zi)}(p)$ 
    if own $_{X(\tilde{S}_Xi)}(p)$ 
         $X(\tilde{S}_Xi) = f(Y(\tilde{S}_Yi), Z(\tilde{S}_Zi), \dots)$ 
```



PIPS read and write array regions

22

- Array elements described as integer polyhedra
- Integer polyhedra: compromise between expressivity and easy mathematical management
- Not all the memory accessed can be sum up with polyhedra
- \rightsquigarrow Array regions can be exact (the elements accessed are exactly described) or inexact (with more points than really accessed, to be conservative)
- Regions are built up bottom-up through hierarchical statements

```
// N-r-exact{}
double b[N], a[N]; int i; ...
```

```
// i-w-exact{}
// i-r-exact{}
```



Automatic distribution with PIPS source-to-source transformation tool
from École des Mines de Paris

- Many semantics analysis and code transformations available in PIPS
- Parallelization transformations usable here to extract parallel tasks
- Interprocedural dependence graph usable to feed the Kaapi scheduler
- PIPS compute “Regions” that define storage areas used by a piece of code : use them to generate communications
- Parallelization is in the general case not decidable : need to think about SAFESCALE directives to help the compiler too



PIPS read and write array regions

23

```
// N-r-exact{}
// a[ $\varphi_1$ ]-r-exact{ $0 \leq \varphi_1 \wedge \varphi_1 \leq N$ }
// b[ $\varphi_1$ ]-r-inexact{ $0 \leq \varphi_1 \wedge \varphi_1 \leq N$ }
// Code to execute somewhere else :
{
    // s-w-exact{}
    double s = 0;
    // i-w-exact{}
    // i-r-exact{}
    // N-r-exact{}
    // a[ $\varphi_1$ ]-r-exact{ $0 \leq \varphi_1 \wedge \varphi_1 \leq N$ }
    // b[ $\varphi_1$ ]-r-inexact{ $0 \leq \varphi_1 \wedge \varphi_1 \leq N$ }
    for(i = 0; i < N; i++) {
        // i-r-exact{}
```



```

// a[ $\varphi_1$ ]-r-exact{ $\varphi_1 == i$ }
// s-r-exact{ }
// s-w-exact{ }
s += a[ i ];
// s-r-exact{ }
// i-r-inexact{ }
// a[ $\varphi_1$ ]-r-inexact{ $\varphi_1 == i$ }
// b[ $\varphi_1$ ]-w-inexact{ $\varphi_1 == i$ }
if (s > 0)
    // i-r-exact{ }
    // a[ $\varphi_1$ ]-r-exact{ $\varphi_1 == i$ }
    // b[ $\varphi_1$ ]-w-exact{ $\varphi_1 == i$ }
    b[ i ] = 2*a[ i ];
}

```



PIPS in and out array regions

26

- Use the dependence graph to compute elements that are *really* used by a statement (*in* regions) and that are written and will *really* be needed by a future statement (*out* regions)

```

// N-in-exact{ }
double b[N], a[N]; int i; ...

// N-in-exact{ }
// a[ $\varphi_1$ ]-in-exact{ $0 \leq \varphi_1 \wedge \varphi_1 \leq N$ }
// b[ $\varphi_1$ ]-out-inexact{ $0 \leq \varphi_1 \wedge \varphi_1 \leq N$ }
// Code to execute somewhere else:
{
    // s-out-exact{ }
    double s = 0;
    // N-in-exact{ }

```



```

}
// b is used later
...

```

Read and write regions are overkill for us because some statements may write elements that are not used later...



PIPS in and out array regions

27

```

// a[ $\varphi_1$ ]-in-exact{ $0 \leq \varphi_1 \wedge \varphi_1 \leq N$ }
// b[ $\varphi_1$ ]-out-inexact{ $0 \leq \varphi_1 \wedge \varphi_1 \leq N$ }
// s-in-exact{ }
for (i = 0; i < N; i++) {
    // i-in-exact{ }
    // a[ $\varphi_1$ ]-in-exact{ $\varphi_1 == i$ }
    // s-in-exact{ }
    // s-out-exact{ }
    s += a[ i ];
    // s-in-exact{ }
    // i-in-inexact{ }
    // a[ $\varphi_1$ ]-in-inexact{ $\varphi_1 == i$ }
    // b[ $\varphi_1$ ]-out-inexact{ $\varphi_1 == i$ }
    if (s > 0)

```



```

// i-in-exact{
// a[ $\varphi_1$ ]-in-exact{ $\varphi_1 == i$ }
// b[ $\varphi_1$ ]-out-exact{ $\varphi_1 == i$ }
b[i] = 2*a[i];
}
}
// b is used later
...

```



Optimized SAFESCALE generation

30

- For inexact out regions, may use combining write instead of read-then-write
 - ▶ Need to track modified elements with a run-time resolution
 - ▶ May use inspector-executors
 - ▶ Need to detect loop invariant region patterns
- In the generation sketch up, guess that variables are allocated in the same way on each task using them
 - ▶ Quite inefficient if a task use only few element
 - ▶ Use a more compact allocation
- What about the general pointers
- Some science fiction: reorganize globally variables for more efficient communications and data access (spitting red-black relaxation data into red and black in 2 different arrays...)



To distribute a statement S on a node \mathcal{N} :

- $\text{receive}_{\mathcal{N}}(e \in \text{InExact}(\mathcal{S}) \cup \text{InInexact}(\mathcal{S}))$
- $\text{receive}_{\mathcal{N}}(e \in \text{OutInexact}(\mathcal{S}))$
- $\text{executeTask}_{\mathcal{N}}(S)$
- $\text{send}_{\mathcal{N}}(e \in \text{OutExact}(\mathcal{S}) \cup \text{OutInexact}(\mathcal{S}))$



Optimized SAFESCALE generation

31

- Big collaboration needed here in SAFESCALE : everything still to do ☺



Exemple de tableau 2D

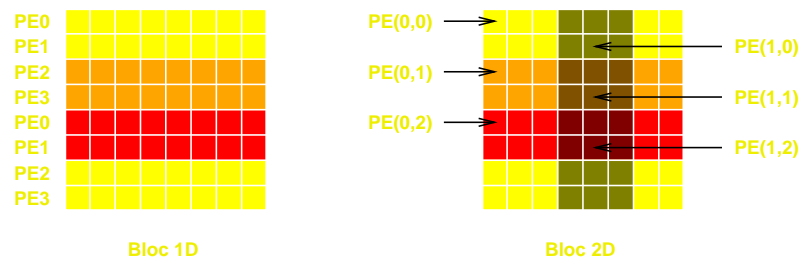


- Minimise les communications si accès de voisinage : calcul de différences finies,...



Distribution bloc-cyclique

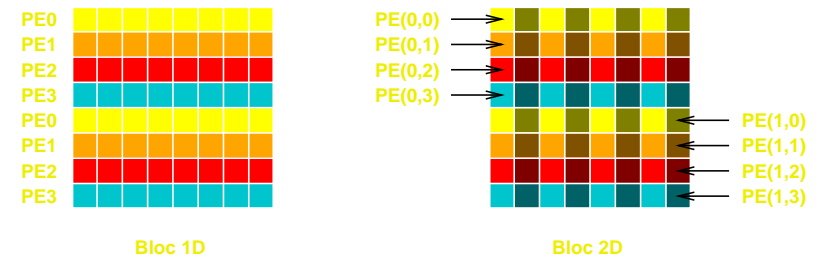
Exemple de tableau 2D



- Compromis entre les 2 approches précédentes : calcul sur des matrices triangulaires,...
- Tous les mélanges sont possible par dimension de processeur, y compris la non distribution



Exemple de tableau 2D



- Favorise la répartition de la charge au dépend des communications de voisinage : calcul de fractales,...



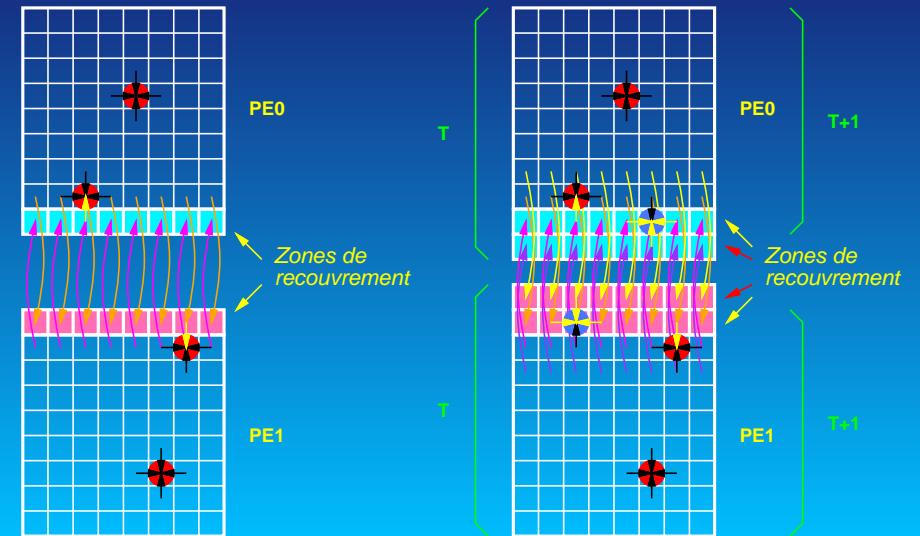
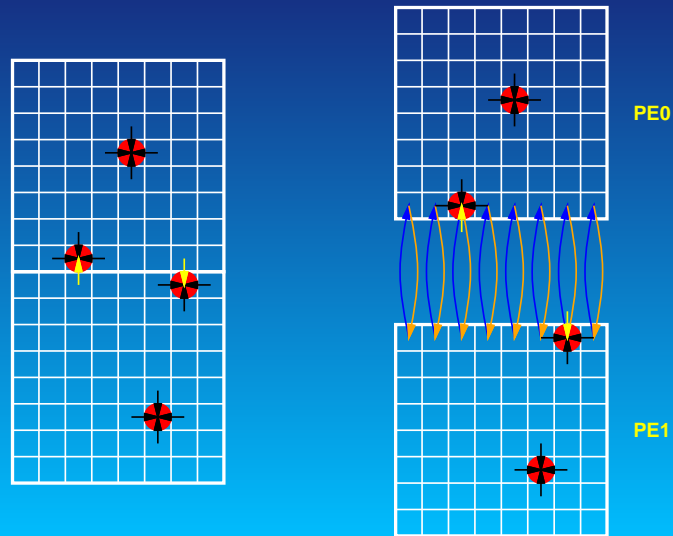
Recouvrements (overlaps)

Problème : faire la différence entre une donnée locale et une donnée distante \rightsquigarrow code compliqué et lent

- Idée : agrandir chaque zone locale pour stocker les données distantes nécessaires
- Rajouter des communications pour récupérer les valeurs des bords
- Éventuellement recouvrir calcul sur l'« intérieur » avec communication sur les bords
- Possibilité de diminuer les communications en faisant du calcul redondant : étendre le domaine de recouvrement d'autant

Exemple : $x_{i,j} = (y_{i-1,j} + y_{i+1,j} + y_{i,j-1} + y_{i,j+1})/4$ distribué en bloc sur 2 processeurs





Exemples de langages à parallélisme de contrôle

39

Langages à passage de messages

- Définition de processus communicants
- Bibliothèques pour échanger des données entre processus
- Parallélisme de contrôle

Langages à parallélisme de données

- Opérations qui s'appliquent à un ensemble de données (addition de matrice, produits scalaires,...)
- plus haut niveau d'abstraction (pas de processus,...)
- Manipulation d'objets familiers : tableaux



- UNIX : |, ssh, socket, NFS, mémoire partagée, mmap(),...
- CSP, OCCAM (processus communicants)
- ADA
- Java
- Langages classiques (C, Fortran) avec bibliothèques de communication (MPI, PVM, P4, NX/2,...)
- Classes C++ pour camoufler les messages



- Langage + beaucoup de *packages* incluant le parallélisme entre autre
- Parallélisme de tâche/contrôle

```
new class (...).start()
```

```
Runtime.getRuntime().exec(...)
```

 RMI : Remote Method Invocation
 Ex : les threads en JDK1.2 avec Solaris 2.6 s'exécutent sur différents processeurs
- Mémoire partagée : accès à des variables visibles en commun
- Mémoire distribuée : communication \equiv envoyer un objet
 Serializable ou Externalizable dans un Stream avec *pipe* ou *socket*



Facilitent la vie du programmeur :

- Vectoriseur : transforme un programme « séquentiel » (Fortran) en programme avec des instructions vectorielles
- Paralléliseur : transforme un programme « séquentiel » (Fortran) en programme(s) pour machine parallèle

Évite le parallélisme explicite mais pas toujours efficace.
Nécessite de l'aide... Directives (Cray, HPF,...).



- Fortran 90 : extension aux opérations sur les tableaux
- CM-Fortran (TMC), MPP-Fortran (Cray), HPF,... : directives de placement
- C* (TMC), MPL (MasPar), PompC, HyperC,... : directives de placement



- Réseaux de la recherche rapide (RENATER, VTHD++, ABILENE,...)
- Mondialisation des infrastructures
- Exploiter les « jachères de calcul » disponibles
- Protocoles plus sexy etinteropérables (XML)
- Mode des Web services

⇒ Grilles de calcul



- Besoins de stocker et partager des données sur plusieurs sites
- Recherche et indexation
- Multiprotocole (HTTP vu comme un pareparefeu...)
- Identifiants uniques indépendant des protocoles ou adresses (NAT,...)
- Construction d'une infrastructure entre pairs au dessus du réseau réel
 - ▶ Nœuds de routage
 - ▶ Nœuds de stockage
 - ▶ Mandataires (*proxy*)
- Exemple : JXTA de Sun en Java, OceanStore,...



Grilles de calcul

46

- Idée offrir des ressources de calcul semblable au service d'électricité
- Faire des requêtes de ressources à un supercalculateur virtuel
- Mutualisation des ressources pour utiliser des puissances de calcul inabordable autrement
- Exemple : Globus2 vu comme un service Web
- Problèmes : latences considérables, débits souvent faibles, identifications globales (PKI),...
- Besoin d'une algorithmique adaptée à très gros grain : couplage de code différents (transport + modèle biologique,...)



Majorité des ordinateurs de la planète ne font qu'attendre l'utilisateur... ☺→ Des PFLOPS potentiels gratuits

- SETI@home : remplace les économiseurs (useurs ?) d'écran pour rechercher de signaux extra-terrestres
- Décrypton : bio-informatique
- Concours de cassage d'algorithmes de cryptographie



Liste des transparents

0 Titre
1 Extraire du parallélisme
0 Introduction
4 Type de parallélisme
5 Grain du parallélisme
6 Degré du parallélisme
7 Programmation des machines parallèles
6 Programmation
8 Compromis à trouver...
9 Mode de couplage avec la mémoire
8 La mémoire
11 Couplage fort \equiv mémoire partagée
13 Couplage faible \equiv mémoire distribuée
15 Placement en mémoire
16 SMP & AMP
17 Le programmeur & la mémoire
18 Simulation d'une mémoire partagée
19 Programmation SPMD pour mémoire distribuée

21 Distribution concepts in PIPS
22 PIPS read and write array regions
26 PIPS in and out array regions
29 SAFESCALE generation blueprint
30 Optimized SAFESCALE generation
32 Distribution par bloc
33 Distribution cyclique
34 Distribution bloc-cyclique
35 Recouvrements (overlaps)
36 Distribution sans recouvrement
37 Distribution avec recouvrement
38 Langages parallèles
38 Langages à passage de messages
38 Langages à parallélisme de données
37 Langages
39 Exemples de langages à parallélisme de contrôle
40 Java
41 Exemples de langages à parallélisme de données
42 Traducteurs
43 Passage à l'échelle
44 Stockage pair à pair
45 Calcul distribué mondial en client-serveur
46 Grilles de calcul