Introduction à la programmation des ordinateurs parallèles

Ronan.Keryell@enstb.org

—

Département Informatique

École Nationale Supérieure des Télécommunications de Bretagne

Master Recherche 2ème année Informatique de Rennes 1 — ENSTBr
ISIA — ENSMP
octobre 2006–février 2007
Version 1.7

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`

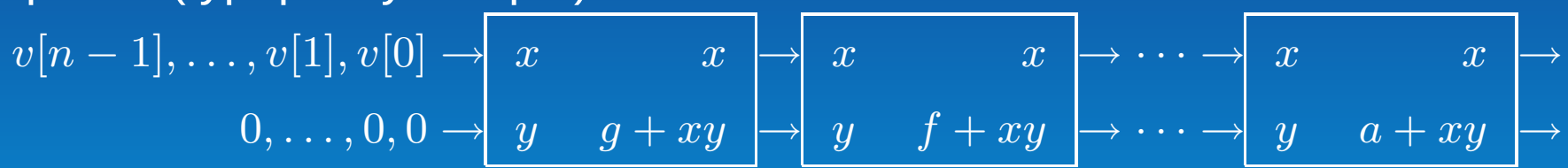
$$vv[i] = a + b.v[i] + c.v[i]^2 + d.v[i]^3 + e.v[i]^4 + f.v[i]^5 + g.v[i]^6$$

`fin pour`

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

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

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

Gros grain $\rightarrow$ grain fin :

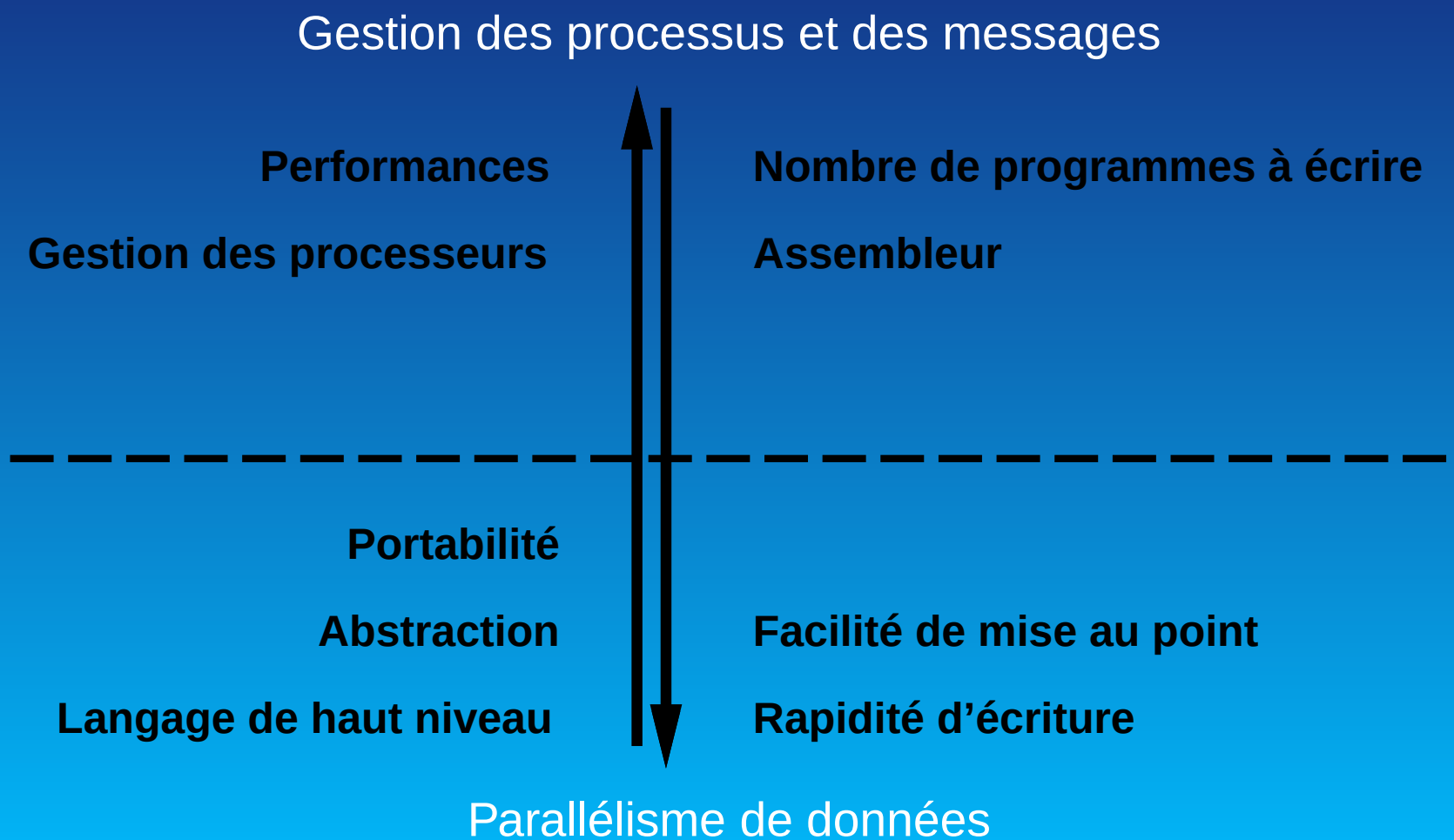
- Programmes
- Procédures & fonctions
- Instructions
- Expressions
- Opérateurs
- Bits

Choix en relation avec l'architecture cible

- 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 ↗

Principalement 2 niveaux :

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


Gestion des processus et des messages
Performances
Gestion des processeurs
Nombre de programmes à écrire
Assembleur
Portabilité
Abstraction
Langage de haut niveau
Facilité de mise au point
Rapidité d'écriture
Parallélisme de données

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

Problèmes (à défaut de SISD...) :

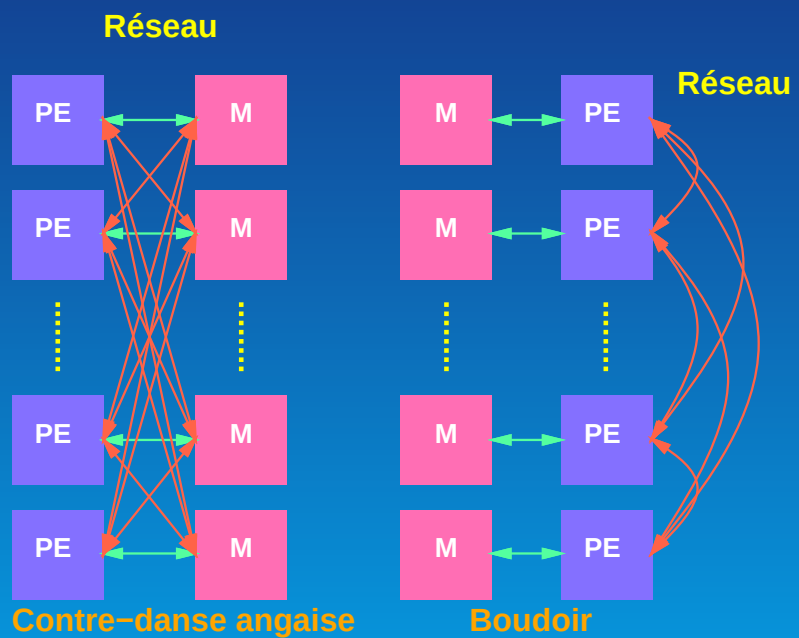
- sortance des portes logiques bornées (temps en $\mathcal{O}(\mathcal{M})$) $=:$ amplifications exponentielles nécessaires (temps en $\mathcal{O}(\log \mathcal{M})$)
- temps de propagation de l'ordre de $\frac{2}{3}c$ dans les fils
- on peut augmenter le débit mais pas diminuer la latence
- conflits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

Mode de couplage avec la mémoire

Solutions possibles :

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

Mais comment relier processeurs et mémoire(s) ?



Couplage fort contre *couplage faible*.

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 ⇝ 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

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

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.

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

- $(i, j) \longrightarrow (ui + vj) \mod N$ BSP
- $(i, j) \longrightarrow i \oplus j \mod N$ STARAN
- $(i, j) \longrightarrow AI \oplus BJ \mod N$ *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

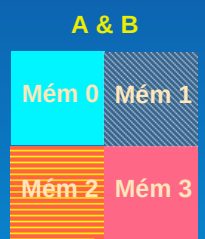
- : asymétrique
  Seul 1 processeur accède au matériel. Les autres sous-traitent

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

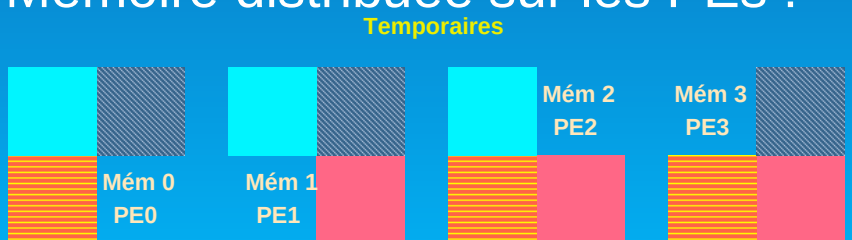
Multiplication de matrice :

$$B = A \times A$$

Modèle d'exécution et couplage mémoire



- Mémoire partagée :
- Mémoire distribuée sur les PEs :



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

- 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 bour 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...

- 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($i, \widetilde{L}i \leq 0$)

X($\widetilde{S}_{\mathtt{X}}i$)=f(Y($\widetilde{S}_{\mathtt{Y}}i$),Z($\widetilde{S}_{\mathtt{Z}}i$),...)

devient

prog(p)

forall($i, \widetilde{L}i \leq 0$)

send$_{\mathtt{Y}(\widetilde{S}_{\mathtt{Y}}i)}(p)$

send$_{\mathtt{Z}(\widetilde{S}_{\mathtt{Z}}i)}(p)$

receive$_{\mathtt{Y}(\widetilde{S}_{\mathtt{Y}}i)}(p)$

receive$_{\mathtt{Z}(\widetilde{S}_{\mathtt{Z}}i)}(p)$

if *own*$_{\mathtt{X}(\widetilde{S}_{\mathtt{X}}i)}(p)$

X($\widetilde{S}_{\mathtt{X}}i$)=f(Y($\widetilde{S}_{\mathtt{Y}}i$),Z($\widetilde{S}_{\mathtt{Z}}i$),...)

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

- 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
- ⇝ 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{}
```

```
// N-r-exact{}
// a[φ1]-r-exact{0 ≤ φ1 ⋀ φ1 ≤ N}
// b[φ1]-r-inexact{0 ≤ φ1 ⋀ φ1 ≤ N}
// Code to execute somewhere else:
{
   // s-w-exact{}
   double s = 0;
   // i-w-exact{}
   // i-r-exact{}
   // N-r-exact{}
   // a[φ1]-r-exact{0 ≤ φ1 ⋀ φ1 ≤ N}
   // b[φ1]-r-inexact{0 ≤ φ1 ⋀ φ1 ≤ N}
   for(i = 0; i < N; i++) {
      // i-r-exact{}
```

```
// a[φ1]-r-exact{φ1 == i}
// s-r-exact{}
// s-w-exact{}
s += a[i];
// s-r-exact{}
// i-r-inexact{}
// a[φ1]-r-inexact{φ1 == i}
// b[φ1]-w-inexact{φ1 == i}
if (s > 0)
   // i-r-exact{}
   // a[φ1]-r-exact{φ1 == i}
   // b[φ1]-w-exact{φ1 == i}
   b[i] = 2*a[i];
}
```

```
}
// b is used later
...
```

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

- 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[φ1]-in-exact{0 ≤ φ1 ⋀ φ1 ≤ N}
// b[φ1]-out-inexact{0 ≤ φ1 ⋀ φ1 ≤ N}
// Code to execute somewhere else:
{
   // s-out-exact{}
   double s = 0;
   // N-in-exact{}
```

```
// a[φ1]-in-exact{0 ≤ φ1 ⋀ φ1 ≤ N}
// b[φ1]-out-inexact{0 ≤ φ1 ⋀ φ1 ≤ N}
// s-in-exact{}
for(i = 0; i < N; i++) {
   // i-in-exact{}
   // a[φ1]-in-exact{φ1 == i}
   // s-in-exact{}
   // s-out-exact{}
   s += a[i];
   // s-in-exact{}
   // i-in-inexact{}
   // a[φ1]-in-inexact{φ1 == i}
   // b[φ1]-out-inexact{φ1 == i}
   if (s > 0)
```

```
        // i-in-exact{}
        // a[φ1]-in-exact{φ1 == i}
        // b[φ1]-out-exact{φ1 == i}
        b[i] = 2*a[i];
    }
}
// b is used later
...
```

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

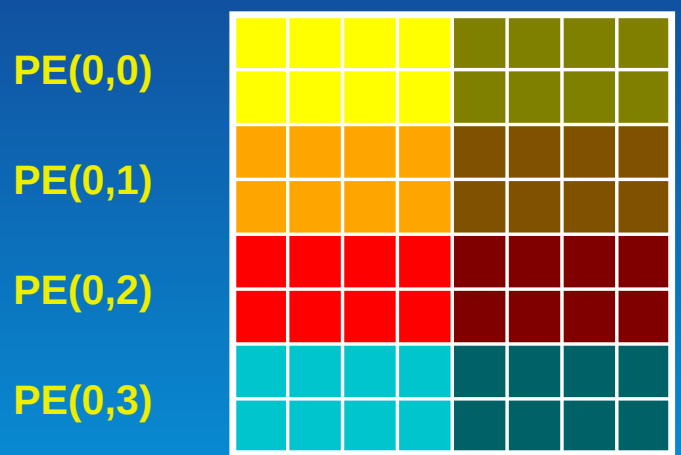
- $\texttt{receive}_{\mathcal{N}}(e \in \mathrm{InExact}(\mathcal{S}) \bigcup \mathrm{InInexact}(\mathcal{S}))$
- $\texttt{receive}_{\mathcal{N}}(e \in \mathrm{OutInexact}(\mathcal{S}))$
- $\texttt{executeTask}_{\mathcal{N}}(\mathcal{S})$
- $\texttt{send}_{\mathcal{N}}(e \in \mathrm{OutExact}(\mathcal{S})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

- 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. . .)

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

Exemple de tableau 2D

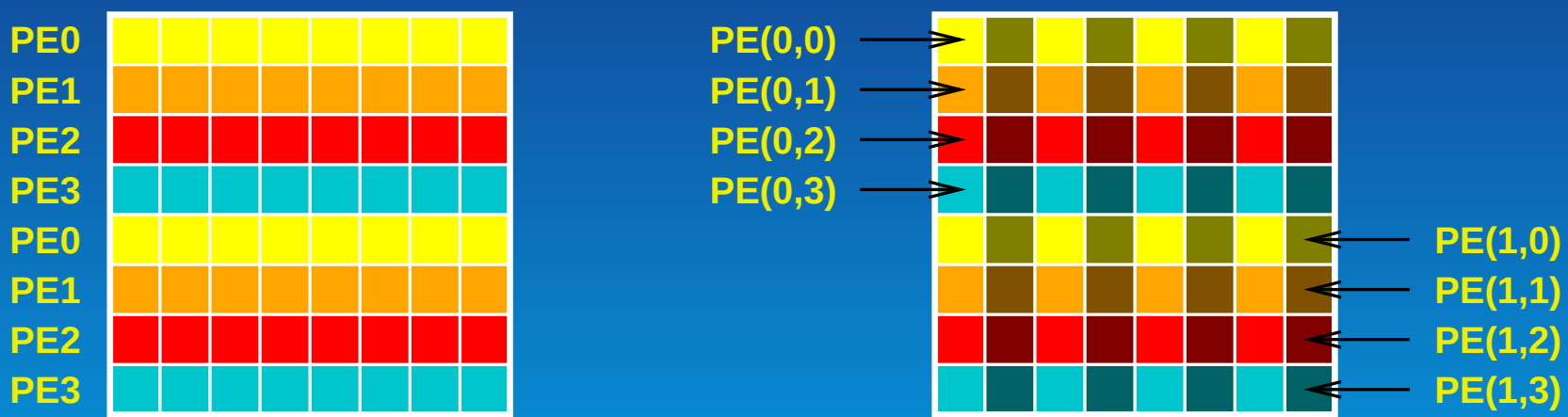
Bloc 1D



Bloc 2D

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

Exemple de tableau 2D

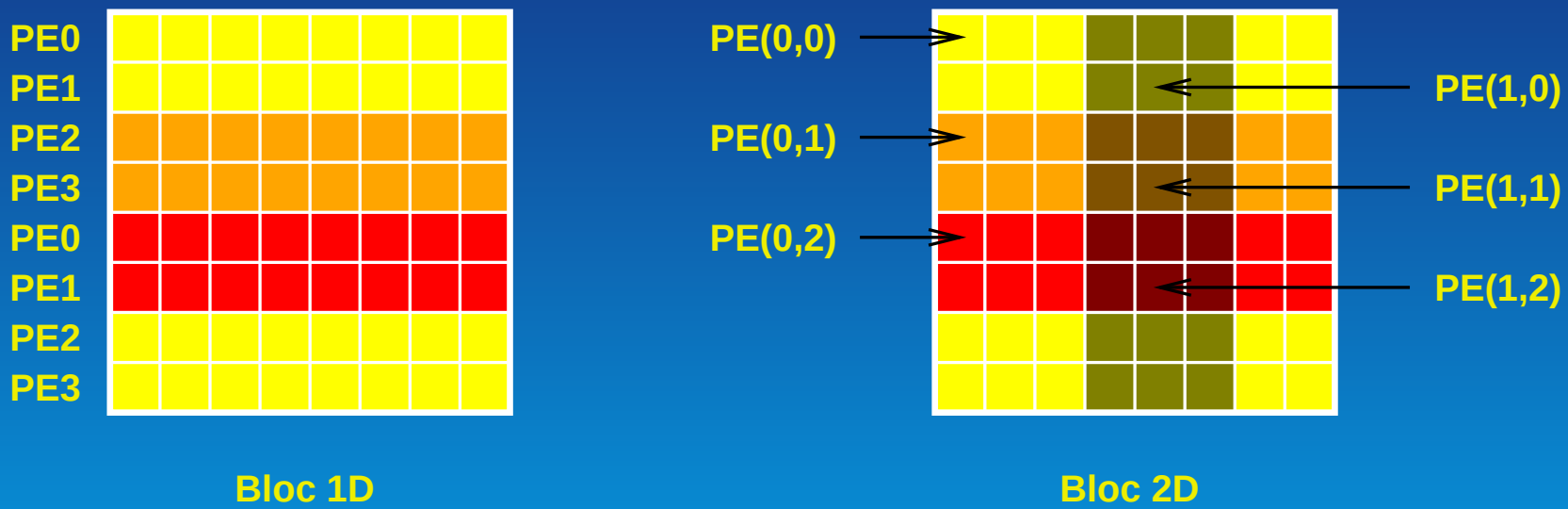


Bloc 1D

Bloc 2D

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

Exemple de tableau 2D



Bloc 1D

Bloc 2D

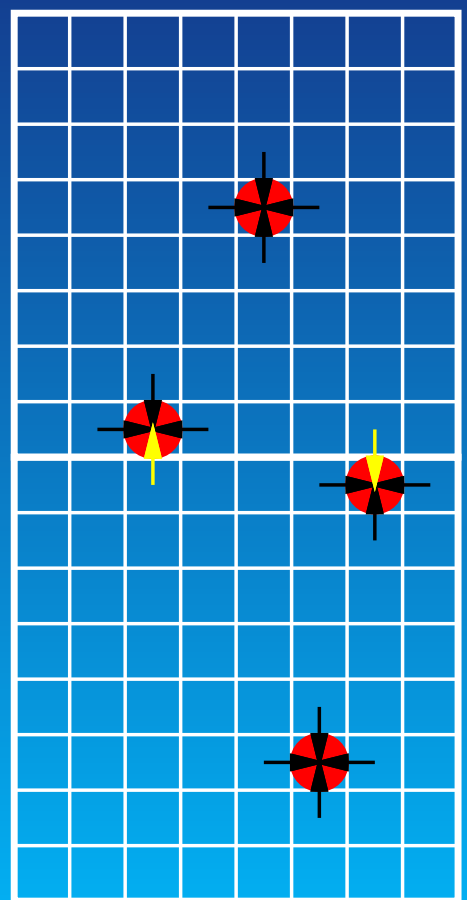
- Compromis entre les 2 approches précédentes : calcul sur des matrices triangulaires,...
- Tous les mélanges sont possible par dimension de processeur, y compris la non distribution

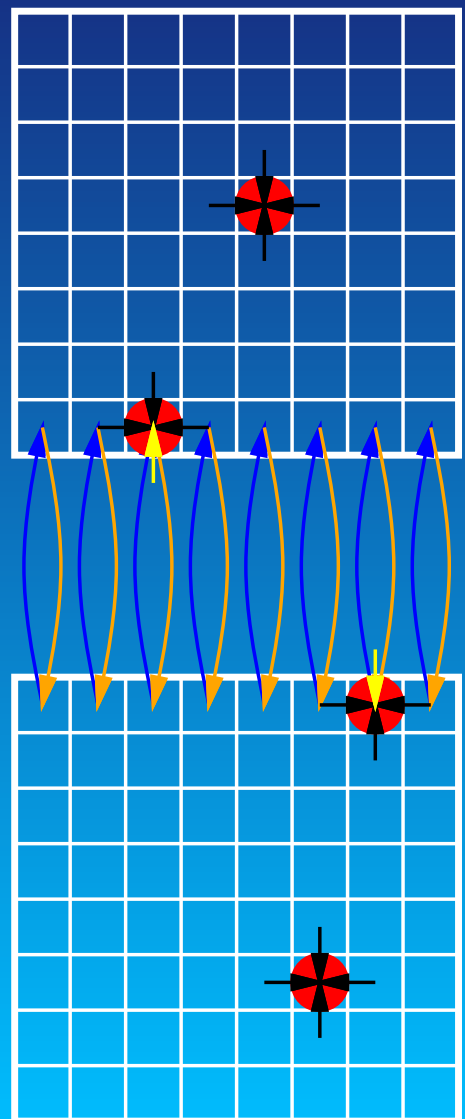
Problème : faire la différence entre une donnée locale et une donnée distante $\rightsquigarrow$ code compliqué et lent

- 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

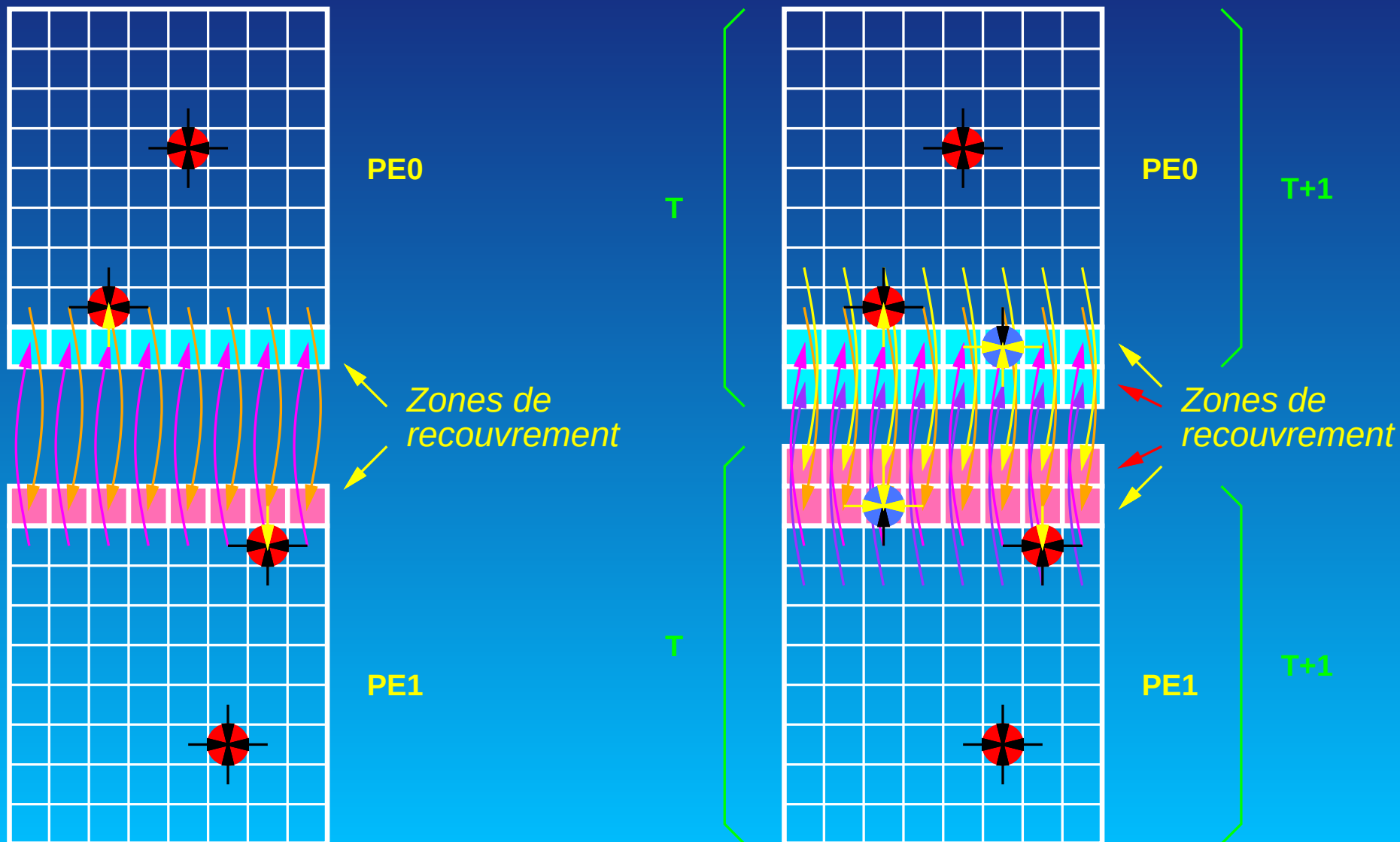
Exemple : $x_{i,j} = (y_{i-1,j} + y_{i+1,j} + y_{i,j-1} + y_{i,j+1})/4$ distribué en bloc sur 2 processeurs

Distribution sans recouvrement





Distribution avec recouvrement



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 ≡ envoyer un objet `Serializable` ou `Externalizable` dans un `Stream` avec *pipe* ou *socket*

- 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

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,...).

- Réseaux de la recherche rapide (RENATER, VTHD++, ABILENE,...)
- Mondialisation des infrastructures
- Exploiter les « jachères de calcul » disponibles
- Protocoles plus sexy et interopérables (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,...

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
- Décrypthon : bio-informatique
- Concours de cassage d'algorithmes de cryptographie

- 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,...)

Liste des transparents