

Slide 1

Parallélisme de données – MPL

Bernard Pottier

Mars 2005

Slide 2

Plan

1. Introduction,
2. Parallélisme de données et MPL,
3. Exemples de programmes.

Slide 3

1 Présentation

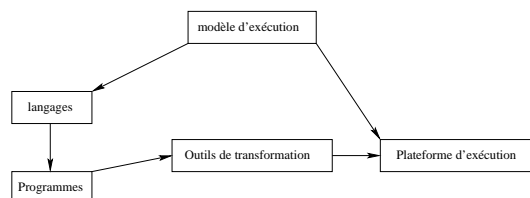
1.1 Définition du domaine

- le raisonnement humain est préférentiellement séquentiel : on aime décrire une solution sous forme de tâches consécutives (recette de cuisine, remplacement d'une pièce de moteur),
- la nature a un comportement à la fois parallèle (plusieurs évènements se produisent simultanément) et séquentiel (principe de causalité) : il pleut sur toute une surface, à une goutte succède une autre goutte.

Slide 4

1.2 Qu'est ce qu'un modèle d'exécution ?

- Les *modèles d'exécution* constituent une représentation abstraite, simplifiée, du fonctionnement d'un circuit, d'une machine, ou d'un logiciel.
- En aval, on crée des outils de traduction permettant de cibler une exécution effective, cablée, ou programmée.
- En amont, on construit des langages permettant l'expression d'algorithmes de traitement conformes au modèle.

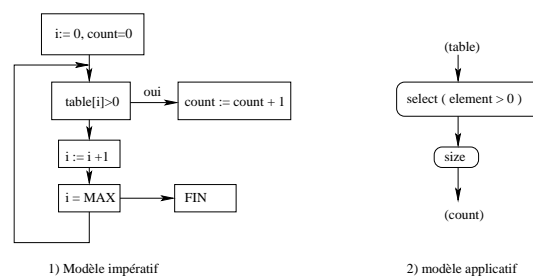


Slide 5

1.3 Modèles d'exécution séquentiel

- Il s'agit d'un graphe de tâches qui peut comporter des cycles dénotant les répétitions.
 - Le *grain* des tâches peut être défini comme le nombre d'opérations qu'elle induit (peut-être 1).
 - Dans le modèle *impératif* (C), ce graphe fait apparaître des variables et des affectations explicites,
 - Dans le modèle *applicatif* (Smalltalk), chaque étape consomme des données et en produit à destination d'étapes suivantes.
- Au plus une tâche est exécutée à un instant donné.

Slide 6



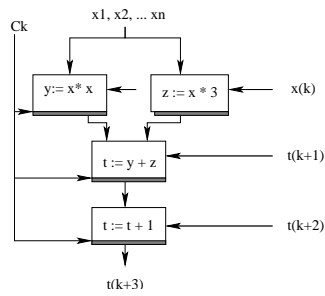
Deux modèles séquentiels : l'un où l'on explicite les variables (`i`, `count`, `table`) et les *affectations*, l'autre où les données circulent dans le graphe.

1.4 Quelques modèles parallèles

1.4.1 pipeline

Un seul graphe, mais plusieurs données présentes. Analogie : chaînes industrielles de montage.

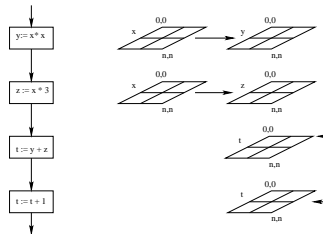
Slide 7



1.4.2 Parallélisme de données

Un seul graphe agit sur des collections de données à plusieurs dimensions. Analogie : particules gazeuses.

Slide 8



Slide 9

1.5 Modèle à parallélisme de données

Ce modèle met en jeu des ensembles de données sur lesquels on opère des traitements identiques de manière synchronisée.

A ce modèle d'exécution correspondent des modèles de machines :

- SIMD (Single Instruction Multiple Data) : machines historiques
Connection Machine, MasPar
- systolique : réseaux cablés pour la génomique, le traitement du signal,
- cellulaire : rétines intelligentes, automates cellulaires.
- PVM : émulation sur réseau de stations.

Slide 10

2 Exemple historique : MasPar et MPL

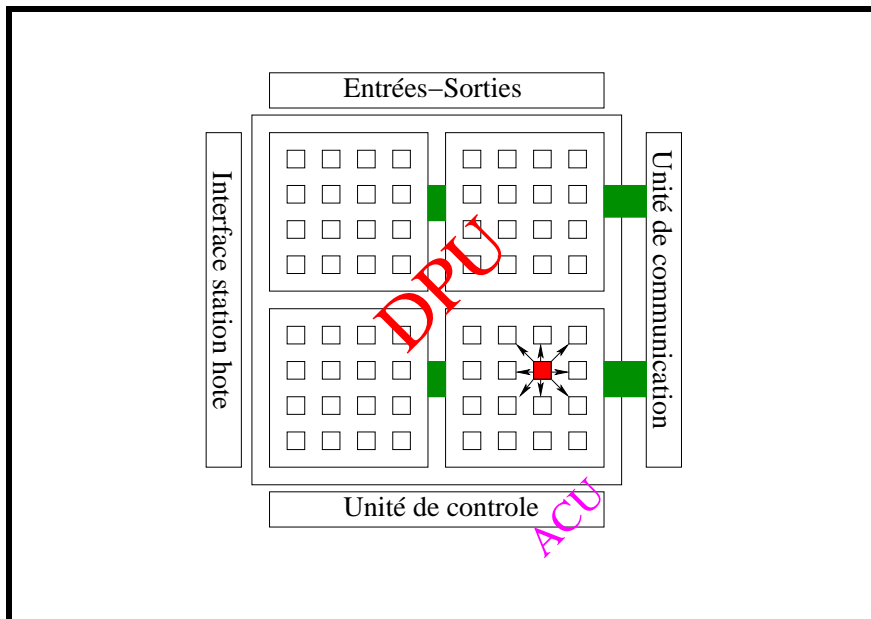
2.1 Architecture générale

MasPar est une machine *massivement parallèle* SIMD comportant de 1000 à 16000 processeurs élémentaires (PEs).

La machine est divisée en plusieurs unités fonctionnelles :

- le calculateur hôte, (station graphique sous Unix).
- l'unité de calcul parallèle : **DPU**,
- le réseau de communication du **DPU**,
- l'unité de contrôle : **ACU**,
- un système d'entrée-sorties.

Slide 11



Slide 12

2.1.1 Unité de contrôle (ACU)

Il s'agit d'un processeur comportant :

- 5 registres de 32 bits,
- 128Ko de mémoire vive pour les données,
- 1 Mo de RAM pour les instructions,
- 4Go de mémoire virtuelle sur disque.

Ce processeur est spécifique à la machine. Il lit les instructions d'un programme et émet des informations de contrôle vers le **DPU**.

Slide 13

2.1.2 Tableau de processeurs (DPU)

Chaque processeur élémentaire (PE) comporte :

- 16 registres de 32 bits et 16Ko de RAM.
- Un masque d'activité permettant d'inhiber temporairement les écritures en registre et en mémoire.

Les PEs sont regroupés en *grappes*^a de 4×4 PEs. Une machine de 1000 PEs comporte donc 64 grappes.

Un bus spécial permet d'échanger des informations avec l'unité de contrôle : les PEs exécutent tous le même code.

^aclusters

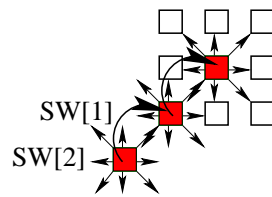
Slide 14

2.1.3 Communications

- Les *communications locales* sont menées via des liens directs reliant un PE à ses voisins N, NE, NW, E,W, S, SE et SW. Ce système local se nomme **X-Net**.
- les *communications globales* sont menées via un *routeur* permettant à un PE d'adresser n'importe quel autre PE. Le langage permet de spécifier des opérations d'échanges multiples (*un-vers-plusieurs* ou *plusieurs-vers-un*).

Slide 15

- Les communications locales permettent de faire passer des valeurs vers les voisins immédiats dans 8 directions cardinales.
- En itérant de proche en proche, on peut atteindre des PE distants.



Slide 16

2.2 MPL

- *Massive Parallel Language* (MPL) permet d'exprimer des algorithmes conformes au modèle parallélisme de données.
- La syntaxe de ce langage peut être interprétée comme un sur-ensemble de C dans lequel on introduit la notion de variables parallèles, localisées dans les PEs.

2.2.1 Allocation de variables

```
int i;  
/* variable scalaire implantée dans l'ACU */  
plural int x,y,z,t;  
/* variables parallèles implantées dans le DPU */
```

Slide 17

2.2.2 Exemple 1

```
int i;
plural int x,y,z,t;

main()
{ /*... initialisation de x dans les PEs */
  for (i = 0; i < 10; i++) { /* scalaire = 1 calcul */
    y = x * x; /* parallèle = 1000 calculs */
    z = x * 3;
    t = z + y + 1;
    x = t; }
}
```

Slide 18

2.2.3 Types simples et opérations élémentaires

+,-	int, short, char	1
	plural char	6
	plural short	12
	plural int	24
	plural float	120
<<,>>	int	shift+5
	plural	10+shift/4 +size/2
×	plural char, char	38
	plural short, short	90
	plural float, float	240

Slide 19

2.2.4 Mouvements de données

La désignation d'une variable distante provoque des copies de données de, ou vers cette variable :

```
x=xnetSW[1].x; y =racinede(xnetSW[2].x); xnetSW[1].x=0;
```

xnet[]	dist > 1	$(size + 2) \times dist + size/4 + 11$
	dist == 1	$size \times 5/4 + 12$
xnet[] 32 bits	dist > 1	$34 \times dist + 19$
	dist ==1	52
router	global <i>pire des cas</i>	5000

Slide 20

2.2.5 Test, exécutions conditionnelles

```
if (condition) bloc1; else bloc2;
```

- Lors que la condition est scalaire (ACU), la signification est celle du test séquentiel (vrai ? bloc1, faux ? bloc2).
- Lorsqu'elle est parallèle, on exécute bloc1 sur les processeurs actifs où la condition est vraie. *Puis*, on exécute bloc2 sur les processeurs actifs où la condition est fausse.
- La durée de l'exécution est la somme durée(bloc1)+durée(bloc2) !

Slide 21

2.2.6 Masque d'activité

- Le masque d'activité détermine si un PE va être actif ou pas.
- Ce masque se propage dans le flot de contrôle lorsque l'on imbrique les instructions de test.
- On peut forcer une activité globale en utilisant le mot-clé `all`.
- Plus le niveau d'imbrication est grand et moins le nombre de PE actifs est élevé.

Slide 22

2.2.7 Bouclages

```
while (condition) action;
```

Comme pour les tests, la classe d'allocation associée à la condition détermine la signification du code.

- **condition scalaire** : la boucle est une simple répétition qui s'achève lorsque `condition == 0`
- **condition parallèle** : la boucle se répète *tant qu'elle est vraie sur au moins 1 PE!*

Slide 23

2.2.8 Conversion et compatibilités des types simples

La localisation des variables est un élément *très important* des algorithmes :

- (scalaire) = (scalaire) : affectation sur l'ACU.
- (plural) = (plural) : affectation en parallèle, sur tous les PEs actifs.
- (plural) = (scalaire) : diffusion d'une valeur de l'ACU à tous les PEs actifs du DPU
- (scalaire) = (plural) : interdit (!!)

Slide 24

2.2.9 Tests globaux

Une forme de controle originale permet de savoir si une condition est vraie à au moins un endroit, ou si elle n'est vraie nulle part.

La variable est *parallèle*, mais le résultat est *scalaire*.

```
if (globalOr(parallelCondition)) { ... }
```

Alternativement `reduceOr32(parallelCondition)` pour évaluer un test sur des variables de 32 bits.

Slide 25

2.2.10 Localisation des processeurs

Il est rendu possible grâce à un numérotage des PEs.

Il existe 3 variables, `ipro` `ixproc` `iyproc` qui permettent de repérer l'index global (lignes, puis colonnes), la position horizontale, ou verticale.

Le langage permet d'accéder à une variable (plural !) chez un voisin lointain ou immédiat en adressage absolu.

```
proc[n].valeur = x ;  
/* range x dans valeur du processeur n */  
proc[y][x]. valeur = x ;  
/* idem mais chez le PE (x,y) */
```