Module: Le Parallélisme

Chapitre 1 :

Introduction à la Programmation Parallèle

1. Introduction

La programmation appelée simultanée,
- où la machine n'effectue pas un *unique* traitement séquentiel,
- mais utilise des *processus concurrents*,
    * soit grâce à un seul processeur (parallélisme simulé ou concurrence)

    * soit grâce à plusieurs processeurs (parallélisme)

◊ Nécessite des techniques et des outils de mise en œuvre de ces processus concurrents

La *programmation parallèle* peut permettre une expression de solution très proche de la logique des problèmes à traiter.

La synchronisation et la communication entre les différents processus sont les problèmes majeurs qui se posent dans ce genre de programmation

Application :

Cette programmation a longtemps été liée aux systèmes d'exploitation des gros ordinateurs: ils doivent partager leurs ressources pour le compte de nombreux utilisateurs,

Il divise son temps et spécifie des tâches qui :
- communiquent,
- coopèrent,
- ou entrent en compétition :

    ◊ C'est ce qu'on appelle la programmation simultanée.

Puis, évolution oblige, les techniques de la programmation concurrente ne sont plus utilisés exclusivement dans le cadre de l'écriture des systèmes d'exploitation, mais s'imposent dans beaucoup de domaines tels que :

- le contrôle des processus industriels,
- les centrales téléphoniques,
- en temps réel (prise en compte du temps : contrainte temporelle)
- en intelligence artificielle (volume des informations important),
- en traitement de la parole et de l'image,
- etc…

Et s'est généralisé actuellement dans beaucoup d'axes en informatique à cause du gain considérable du temps, offert par le parallélisme.

Des *langages* se sont alors imposés où le multitâche est réalisé grâce à des instructions bien spécifiques, tels que ADA, l'un des premiers dans le domaine à accentuer cette tendance, depuis voilà plus d'une vingtaine d'années.

La programmation simultanée représente une part importante de la recherche informatique actuelle.

Le parallélisme ou la concurrence n'est pas toujours imposé pour la résolution d'un problème, mais est commode pour spécifier élégamment une solution.

Le parallélisme est véritablement apparu (le pure) lorsque des machines composées de multiples processeurs sont apparues, et il est devenu alors nécessaire de construire des applications pouvant exploiter au mieux le parallélisme potentiel de ces machines, voici plus d'une vingtaine d'année.

Le projet d'une 5è génération lancé en 1982 par les Japonais :
- a mis à l'avant garde des machines, les multiprocesseurs, et
- les a promis à un avenir brillant dans ce qu'ils ont appelés, les machines « intelligentes », qui ne peuvent être performantes que s'ils sont multiprocesseurs avec un nombre important de processeurs (de l'ordre de 256) appelés massivement parallèles.

Les architectures en réseau, (ayant fait un bond en avant avec l'apparition des autoroutes de l'information et Internet), ont également beaucoup à devoir au parallélisme et partage des tâches, sans qui, ils n'existeraient pas actuellement.

Bref, en un mot, « la ***communication*** »

- est un mot de la fin du siècle passé et de ce début de ce siècle, le plus important qui se doit,
- et est une technique, si ce n'est la plus importante en parallélisme ;

sans communication, le parallélisme n'existerait pas car les différents processus qui s'exécutent simultanément ont besoin à un moment ou à un autre, de communiquer.

2. Qu'est-ce que la programmation parallèle ?

2.1. Les raisons d'être du parallélisme

Le *parallélisme* (= multitâche = temps partagé = concurrence = simultanéité …)
A plusieurs raisons d'être, qui se résument :

-1- Calcul des expressions indépendantes:

En calcul numérique, nous avons souvent à évaluer des expressions indépendantes dans un même calcul, ce qui, en séquentiel, coûte un temps précieux alors qu'ils peuvent l'être simultanément, avec un temps record.

Exemple : traitement des vecteurs, matrices, etc,…on a à effectuer la même opération, souvent, aux composantes d'un vecteur ou aux éléments d'une matrice, d'une manière totalement indépendante, par exemple, une translation d'un vecteur v, de n :

V + n = (v1 + n , v2 + n , … , vk + n) : on a à réaliser k additions.
Si traitement séquentiel : **k** unités de temps
Si traitement parallèle : **1** unité de temps
(k processeurs : chacun effectue une addition)
Mais ce n'est pas toujours facile car : parfois, ces calculs ne sont pas toujours indépendants, mais partagent par exemple une variable, calculée par un processeur et communiquée à un autre qui attend cette valeur pour commencer ou continuer un calcul.

Exemple : $\pi := 3.14$ $n := 4$ $x := \pi + n$
P1 P2 P3

- P1 et P2 peuvent être exécutés en parallèle,
- mais P3 ne peut pas l'être, il attendra que P1 et P2 soient terminés pour que leur valeur soit connue et qu'il puisse s'exécuter.

-2- « Diviser pour régner » : souvent, dans les problèmes complexes, le fait de les diviser ou partitionner en plusieurs sous problèmes, plus simples, permet de trouver une solution au problème complexe.
Ceci permet de ramener la solution du problème compliquée à celles de ces problèmes plus simples.
De plus, si ces problèmes plus simples peuvent être traités simultanément, le traitement n'en sera que plus élégant et plus performant,

==> C'est souvent le but des algorithmes utilisant le parallélisme :

Si un problème P peut être divisé en N sous problèmes P*i* peu ou pas dépendants, ces N sous problèmes sont traités simultanément pour résoudre le problème P.

-3- « Gain de performance » : c'est le but essentiel du parallélisme, en effet, les VLSI ont atteint le maximum des performances des machines, depuis, les seules améliorations ont visé la multiplicité des processeurs et le développement des langages parallèles.

De nombreux travaux ont eu ou ont lieu en vue :
- de trouver de nouveaux algorithmes dits parallèle, pour solutionner des problèmes classiques de tri, recherche, etc…,
- et de mettre en œuvre de façon très performante ces algorithmes sur des machines multiprocesseurs.

2.2. Comparaison de la programmation Séquentielle / Parallèle

Les **programmes séquentiels** sont écrits grâce à une succession ordonnée d'instructions, compilés en langage machine et ensuite exécutés selon l'ordre imposé.

- Pour les mêmes données, l'ordinateur exécutera toujours la même séquence d'instructions et donnera les mêmes résultats : on dit qu'ils sont déterministes.
Si on constate une *erreur* dans le programme, on peut la trouver par pistage (en imprimant les instructions exécutées), ou grâce à des points d'arrêt et affichage des valeurs des variables.

- Les programmes séquentiels se *terminent* toujours pour retourner un résultat.

- Le facteur *temps* est peu important lors de l'exécution des instructions, chacune mettant généralement une unité de temps.

En **programmation parallèle**, l'ordre n'est pas imposé dans l'exécution des instructions : si celles ci ne partagent aucune variable (elles sont indépendantes), elles s'exécuteront simultanément, mais si elles ont des variables partagées, un problème de synchronisation apparaît, alors l'une doit attendre que l'autre instruction ait évalué la variable partagée, puis le lui a communiquée, pour qu'elle puisse enfin s'exécuter : l'ordre intervient alors, mais d'une manière non explicite syntaxiquement.

- Pour de mêmes *données*, un programme parallèle ne garantit pas les mêmes résultats : on dit qu'ils sont non déterministes.

Exemple (x := 1) // (x := 2)

Pour cet exemple très simplifié, on ne sait pas si à l'arrêt, la valeur de x sera 1 ou 2

- Les programmes parallèles ne sont pas obligés de se terminer pour donner un résultat, mais souvent, destinés à être exécutés infiniment comme dans le cas du contrôle des processus industriels, des centrales téléphonique, etc…

- Le facteur *temps* est très important au niveau de l'exécution des programmes parallèles, il intervient au niveau de l'exécution de chaque tâche, avec des instants éventuellement d'attente (on attend une valeur d'un autre processus), etc…

La programmation simultanée ou programmation parallèle est le *nom* donné aux notations et aux techniques de programmation exprimant que le parallélisme est possible

Elle résout les problèmes :
- de communication et
- de synchronisation.

L'implantation du parallélisme sur un ordinateur ne concerne pas la programmation simultanée : celle ci ne fait qu'établir un contexte théorique à étudier le parallélisme, sans se préoccuper des détails d'implantation.

Ce modèle s'est révélé si utile et si pratique dans l'écriture claire et correcte des logiciels, que maintenant, certains langages permettent la programmation simultanée (**ADA**, l'un des premiers, suivi d'une multitudes), ou l'enrichissement des langages classiques des techniques parallèles.

3- Exemple

Soit le problème classique de **tri** d'une liste d'entiers par ordre croissant de ses éléments.
Il est évident qu'un problème de ce genre, résolu en programmation séquentielle, sera très coûteux.

Soient les procédures :

Partition (u, list, list1, list2) :
à partir
de la liste *list* et
du pivot *u* ,
donne
list1, la liste des éléments de *list* < u ,
et *list2* , celle des éléments de *list* ≥ u

Append (list1, list2, list3) :
à partir des listes en entrée *list1* et *list2*,
construit la liste *list3* en concaténant *list1* avec *list2*

On définit alors :

Sort (list, slist) :
à partir de la liste d'entrée *list* ,
donne cette liste triée par ordre croissant *slist*

Soit *tête* (list) = 1er élément de list
Queue (list) = list sans son 1er élément

On peut alors écrire cette procédure comme suit :

Sort (list , slist) :

P1 : Partition (tête (list) , queue (list) , list1 , list2)

P2 : Sort (list1 , slist1)

P3 : Sort (list2 , slist2)

P4 : Append ([tête (list)] , slist2 , slist'2)

P5 : Append (slist1 , slist'2 , slist)

Il y a donc 5 procédures à exécuter pour le traitement de Sort (list , slist)

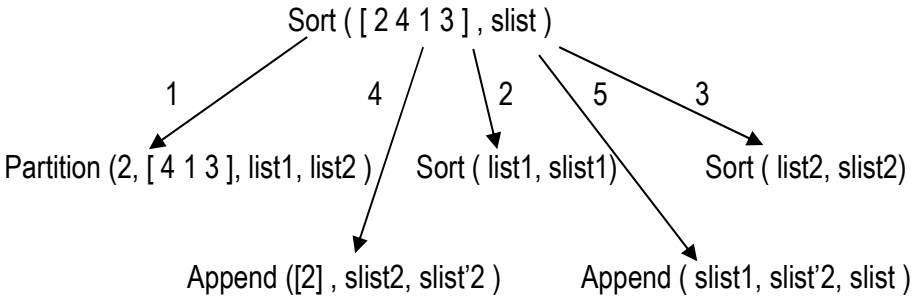
Le problème a donc été partitionné, ou divisé en problème (lots) plus simples

Réaliser ceci en parallèle est possible, vu que P2 et P3 sont totalement indépendants, donc peuvent être exécutés en parallèle, mais seulement après que P1 ait terminé vu que list1 et list2 sont élaborés par lui, qui les lui communiquera

P4 ne peut s'exécuter que lorsque P3 aura terminé
et P5 ne pourra s'exécuter que lorsque P2 et P4 auront terminés.

En résumé, on aura l'ordre suivant :

P1, puis P2 // P3, puis P4, puis P5
Application à Sort ([2 4 1 3], slist)



(1) ==> Elaboration list1 = [1]
list2 = [4 3]

(2) ==> Sort ([1] , slist1) //

Partition (1, nil , list1, list2)

Sort (list1, slist1)

Sort (list2, slist2)

Append ([1] , slist2 , slist'2)

Append (slist1, slist'2, slist)

→ Elaboration slist1 = [1]

(3) ==> Sort ([4 3] , slist2)

Partition (4, [3], list1, list2)

Sort (list1, slist1)

Sort (list2, slist2)

Append ([4] , slist2 , slist'2)

Append (slist1, slist'2, slist)

↳ Elaboration slist2 = [3 4]

(4) ==> Append ([2] , [3 4], slist'2)

↳ Elaboration slist'2 = [2 3 4]

(5) ==> Append ([1] , [2 3 4], slist)

↳ Elaboration slist = [1 2 3 4]