TD/TP nº 2 : Multithreading vs Multiprocessing

— Partie TD —

Question 1 : Somme et produit de matrices en séquentiel

Notations : $A = (a_{ij})$ est une matrice de type (m, q) (m lignes et q colonnes). Nous ne considérons que des matrices de nombres dits flottants (type `double`).

a) Somme de matrices

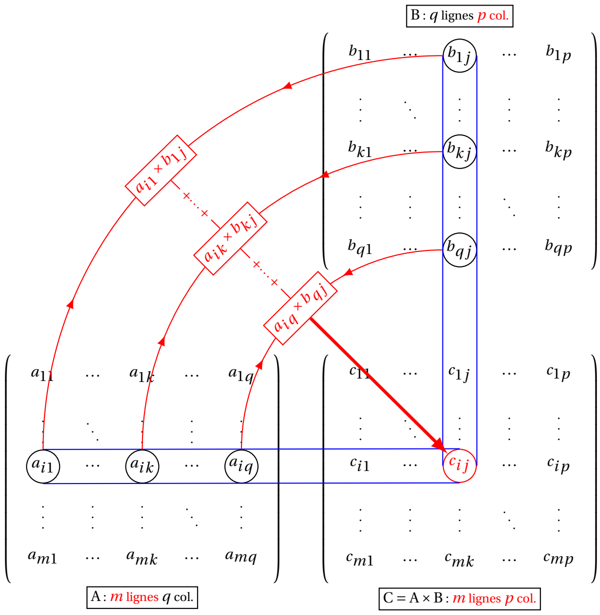
La somme de matrices est l'exemple type d'algorithme (dans sa version de base) qui se parallélise facilemment.

Rappel : pour $C = A + B$, si $A = (a_{ij})$ est une matrice de type (m, q) et $B = (b_{ij})$ une matrice (m, q) alors $C = (c_{ij})$ est une matrice (m, q) telle que

$$\forall i \in \{1, \ldots, m\},\ \forall j \in \{1, \ldots, q\} \ :\ c_{ij} = a_{ij} + b_{ij}$$

Écrivez sur papier la fonction `sommeMat(A,B,C,m,q)` en C. Attention, en C, les indices commencent par 0 !

b) Produit de matrices



Source : adapté de texample.net/tikz/examples/matrix-multiplication

Rappel : pour $C = AB$, si $A = (a_{ij})$ est une matrice de type (m, q) (m lignes et q colonnes), $B = (b_{ij})$ une matrice (q, p) alors $C = (c_{ij})$ est une matrice (m, p) tel que

$$\forall i \in \{1, \ldots, m\},\ \forall j \in \{1, \ldots, p\} \ :\ c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$$

Écrivez sur papier le programme C. Attention, en C, les indices commencent par 0 !

c) Quelle est la **complexité** de ces algorithmes dans le cas de matrices carrées (m, m) ?

d) Stockage des matrices en mémoire

Le stockage classique sous la forme de `double matrice[][]` n'est ni pratique ni très efficace pour des raisons d'allocation, libération etc. Nous allons utiliser une allocation linéaire et une fonction/macro pour convertir les coordonnées (ligne,colonne) en une "adresse" dans la représentation linéaire.

Voici l'idée avec une matrice A(2,3) stockée sous forme `double A[6]` (au lieu de `double matA[2][3]`) :

$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix} \Longrightarrow$

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}

Comment calculer x en fonction de i et j pour passer de a_{ij} à A[x] sachant que A a m lignes et q colonnes ?

$x =$ ______

Si la notation A[i][j] est vraiment souhaitable, il possible d'allouer un tableau de pointeurs supplémentaire (rappel : `pteur+entier = &(ptr[entier])`) et le remplir avec les adresses correspondantes dans le bloc de données de l'allocation linéaire.

Ecrire le code C de l'allocation d'une matrice (n, m) stockée sous forme linéaire avec en plus un tableau de pointeurs pour l'accès directe à une case. On suppose que les éléments de la matrice sont des `double`.

— Partie TP —

Objectif : comparer la programmation multi processus et la programmation multi-threads en termes de performances. On commence par se donner une API pour se faciliter le travail.

Commencez par recopier TOUS les fichiers du répertoire `/home/usager/TPINFO/butelle/M4102C/TP2` *dans un répertoire personnel. Nous allons utiliser un découpage propre en fichier include, fichier de fonctions et un fichier contenant un* `main` *(ex.* `multi.h, multi.c` *et* `testmulti.c`*).*

Question 2 : Programmation Multi-processus :

Codez vos fonctions dans le fichier fourni `multi.c` *en respectant les prototypes*

a) Ecrire une fonction `proc_start` qui prend deux paramètres : f (le nom d'une fonction) et id un entier qui sera le numéro de tâche

Cette fonction doit appeler f(id) par un nouveau processus fils créé par fork, le fils doit s'arrêter après par `exit(0)`. La fonction `proc_start` doit retourner le PID du ps créé.

Attention, le prototype est un peu spécial `pid_t proc_start (void *(*f) (int), int id)`

Le premier paramètre indique que l'on passe un nom de fonction qui prend comme paramètre un seul entier et retourne un `void *`. Exemple de prototype : `void *lafonc(int id);`

b) Ecrire une fonction `proc_join` qui attend la fin d'un processus

Un seul paramètre : le PID du ps à attendre. Quel est l'appel système à utiliser ?

c) **compilation et exécution**

La compilation se fait par `make testmulti` et l'exécution par `./testmulti 3 ps` : cela doit permettre de lancer 3 ps qui vont chacun afficher un message.

Question 3 : Programmation Multi-threads

Codez vos fonctions dans le fichier fourni `multi.c` *en respectant les prototypes*

a) Ecrire une fonction `thread_start` avec les mêmes param. que `proc_start`, sauf qu'il retourne un identifiant de thread de type `pthread_t`

Cette fonction doit créer un thread qui exécute la fonction avec comme param. id (encore un numéro de tâche). Quel est l'appel système à utiliser ?

Pour "simplifier" on va se permettre de convertir l'entier id en un pointeur, Vous allez être obligé d'utiliser un cast spécial : `(void *) (intptr_t)` sur id et un cast `(thread_fn_t)` sur `f` peut aussi aider.

b) Ecrire une fonction `thread_join` avec comme param. un identifiant de thread de type `pthread_t`

Quel est l'appel système à utiliser pour attendre la fin d'un thread ?

c) **compilation et exécution**

De même compilez avec `make testmulti` et lancez l'exécution avec `./testmulti 3 th`

Question 4 : Implémentation des matrices

Remplissez les fonctions prédéfinies dans `matrices.c`*, la lecture et l'affichage de matrice sont fournis*

Pour un code plus clair, nous utiliserons un type `matrice_t` intégrant les dimensions de la matrice et son contenu :

```
typedef struct {
        int dim1,dim2;
        double *val;
} matrice_t;
```

a) **Programmez** les algorithmes séquentiels de somme et produit de matrices

b) **Compilation et exécution**

La compilation se fait par `make testmatrices`

Vous pourrez ensuite exécuter `./testmatrices add mat1.mat mat2.mat`

puis `./testmatrices mult mat1.mat mat2.mat`

Leur somme doit être égale à `somme.mat` et leur produit doit être égal à `produit.mat`.

Question 5 : Parallélisme

a) Observation

Lancez un navigateur, visitez la page de l'université : `www.univ-paris13.fr`.

Combien de processus sont lancés sur votre machine ? Combien vous appartiennent ? Combien de threads sont lancés par le navigateur ? (voir cours).

b) Préparation du calcul

On va paralléliser les calculs précédents sur les matrices carrées ($m \times m$) en découpant la boucle externe (attention ce n'est pas la meilleure technique : il vaudrait mieux découper en sous-matrices pour bénéficier de moins de *défauts de cache (page faults)*).

L'idée est la suivante : si on dispose de p tâches (processus ou threads avec $p \leq m$), on divise l'intervalle $[0, m-1]$ en p intervalles, soit $h = \left\lfloor \frac{m}{p} \right\rfloor$ (partie entière par défaut de la division de m par p).

- tâche 0 : calcul pour les lignes de 0 à $h-1$ incluse.
- tâche 1 : calcul pour les lignes de h à ______ incluse.
- tâche i : calcul pour les lignes de ______ à ______ incluse.
- Attention, la dernière tâche ($p-1$) doit éventuellement faire un peu plus de travail que les autres si $\frac{m}{p}$ ne tombe pas juste, elle doit aller dans tous les cas jusqu'à $m-1$ inclus.

c) Version multiprocessus

Découvrez combien de cœurs dispose le CPU de votre ordinateur : `grep proc /proc/cpuinfo | wc -l`

Notons que si l'Hyper-Threading est activé au niveau du BIOS cela vous donnera en fait le nombre de cœurs virtuels, c'est à dire deux fois le nombre de cœurs physiques mais peu importe.

Remplissez les fonctions prédéfinies dans multimat.c en respectant les prototypes

Programmez la somme partielle effectuée par la tâche *i* (comme vu au dessus) puis le produit partiel de matrices. Vous devez remplir les fonctions sommeChunk et produitChunk de multimat.c qui prennent comme paramètre le numéro de tâche `i`.

Compilation et Exécution Compilez avec `make testmultimat`

L'exécution se fait par `./testmultimat -p 3 add mat1.mat mat2.mat`

ou encore `./testmultimat -p 4 -d 1000 mult`

pour voir le temps de calcul de la multiplication de deux matrices carrées aléatoires 1000×1000 par 4 taches.

Utiliser plus de processsus que le nombre de cœurs ne sera pas très utile, testez!

A partir de quelle taille de matrice a-t-on intérêt de faire du multiProcessus ou du multiThreading? Calculez le speedup.