Master d'informatique | **module PC2R** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2006-2007**

Partiel du 20 novembre 2006

Exercice 1 : Threads Posix

Dans cet exercice, le choix du langage est libre.

Multiplication matrices-vecteurs et matrices-matrices

En Calcul Scientifique, les matrices contenant des `int`, des `double` ou des entiers multi-précision apparaissent fréquemment et optimiser toute opération concernant la multiplication matrice-vecteur ou la multiplication matrice-matrice est crucial. On voit ici comment utiliser les threads POSIX pour *paralléliser* ces opérations.

Un vecteur est codé comme un tableau d'entiers de taille N. Une matrice de M lignes et N colonnes est un tableau de M vecteurs de taille N.

Par exemple, la matrice

$$\begin{matrix} 1 & 3 \\ 5 & 7 \\ 2 & 4 \end{matrix}$$

est représentée par un tableau contenant les vecteurs de taille 2 ($[1,3]$, $[5,7]$ et $[2,4]$).

Dans la suite, on ne considère plus que des matrices ayant le même nombre de lignes et de colonnes (on parle alors de matrices carrées). L'élément situé à la i-ème ligne et à la j-ième colonne d'une matrice $\mathbf{a}$ est noté $\mathbf{a}_{i,j}$. Le i-ème élément d'un vecteur $\mathbf{v}$ est noté $\mathbf{v}_i$.

Une multiplication entre une matrice carrée $\mathbf{a}$ de taille N et un vecteur $\mathbf{v}$ de taille N renvoie un vecteur de taille N dont le i-ième élément est donné par la formule :

$$\sum_{j=1}^{N} \mathbf{a}_{i,j}\mathbf{v}_i$$

Une multiplication entre deux matrices carrées $\mathbf{a}$ et $\mathbf{b}$ de taille N renvoie une matrice carrée de taille N dont l'élément situé à la k-ième ligne et la ℓ-ième colonne est donné par la formule :

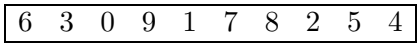
$$\sum_{i=1}^{N} \mathbf{a}_{k,i}\mathbf{b}_{i,\ell}$$

1. Définir les types de données `matrice` et `vecteur` et programmer les fonctions de multiplication matrice-vecteur et matrice-matrice.
2. Donner une version multi-threadée de la multiplication matrice-matrice qui utilise la fonction de multiplication matrice-vecteur que vous avez définie dans la question précédente.
3. Donner une version multi-threadée de la multiplication matrice-vecteur.
4. Donner une fonction prenant en argument trois matrices $\mathbf{a}$, $\mathbf{b}$ et $\mathbf{c}$ (on suppose tous les éléments de $\mathbf{c}$ égaux à 0) et écrivant le résultat de la multiplication de $\mathbf{a}$ par $\mathbf{b}$ dans $\mathbf{c}$. La multiplication doit être ici multi-threadée et faite en place (dans $\mathbf{c}$) : deux threads doivent calculer $\sum_{i=1}^{N/2} \mathbf{a}_{k,i}\mathbf{b}_{i,\ell}$ d'une part et $\sum_{N/2+1}^{N} \mathbf{a}_{k,i}\mathbf{b}_{i,\ell}$ d'autre part et d'autre threads doivent faire la somme de ces deux valeurs calculées.

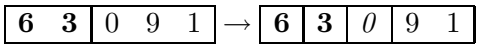
Tri-fusion

On cherche maintenant à utiliser le multi-threading pour implanter efficacement un algorithme de tri-fusion dont on rappelle le fonctionnement ci-dessous.

Les listes d'entiers sont codés comme des tableaux d'entiers.



La liste est scindée en deux, et l'algorithme de tri continue à faire le scindage



... jusqu'à ce que les sous-tableaux soient de longueur 1. L'algorithme procède alors à la fusion des sous-tableaux. On donne en annexe les codes `C`, `Caml` et `Java` de cet algorithme.

1. Rappeler l'utilisation des Mutex en Threads POSIX (ou de leur équivalent) en `C`, `Java` et `Caml`.
2. Dans le langage de votre choix et en partant des implantations données ci-après, multi-threader le tri-fusion dans les appels récursifs.
3. Dans le même langage, multi-threader l'algorithme de fusion.

– **version JAVA**

```
// ------------------------------------------------------------------------
public static void triFusion(int tableau[]) {
  int longueur=tableau.length;
  if (longueur>0) {
    triFusion(tableau,0,longueur-1);
  }
}
private static void triFusion(int tableau[],int deb,int fin) {
  if (deb!=fin) {
    int milieu=(fin+deb)/2;
    triFusion(tableau,deb,milieu);
    triFusion(tableau,milieu+1,fin);
    fusion(tableau,deb,milieu,fin);
  }
}
private static void fusion(int tableau[],int deb1,int fin1, int fin2) {
  int deb2=fin1+1;
  int table1[]=new int[fin1-deb1+1];
  for(int i=deb1;i<=fin1;i++) {
    table1[i-deb1]=tableau[i];
  }
  int compt1=deb1;
  int compt2=deb2;

  for(int i=deb1;i<=fin2;i++) {
    if (compt1==deb2) {
//c'est que tous les éléments du premier tableau ont été utilisés
       break; //tous les éléments ont donc été classés
    }
    else if (compt2==(fin2+1)) {
//c'est que tous les éléments du second tableau ont été utilisés
     tableau[i]=table1[compt1-deb1];
//on ajoute les éléments restants du premier tableau
```

```
      compt1++;
    }
    else if (table1[compt1-deb1]<tableau[compt2]) {
      tableau[i]=table1[compt1-deb1];
//on ajoute un élément du premier tableau
      compt1++;
    }
    else {
      tableau[i]=tableau[compt2];
//on ajoute un élément du second tableau
      compt2++;
    }
   }
}
```

– **version C**

```
// -------------------------------------------------------------------
void fusion(int tableau[],int deb1,int fin1,int fin2)  {
  int *table1;
  int deb2=fin1+1;
  int compt1=deb1;
  int compt2=deb2;
  int i;
  table1=malloc((fin1-deb1+1)*sizeof(int));

  //on recopie les éléments du début du tableau
  for(i=deb1;i<=fin1;i++) {
    table1[i-deb1]=tableau[i];
  }
  for(i=deb1;i<=fin2;i++) {
    if (compt1==deb2) {
    //c'est que tous les éléments du premier tableau ont été utilisés
      break; //tous les éléments ont donc été classés
    }
    else if (compt2==(fin2+1)) {
    //c'est que tous les éléments du second tableau ont été utilisés
      tableau[i]=table1[compt1-deb1];
    //on ajoute les éléments restants du premier tableau
      compt1++;
    }
    else if (table1[compt1-deb1]<tableau[compt2]) {
      tableau[i]=table1[compt1-deb1]; //on ajoute un élément du premier tableau
      compt1++;
    }
    else {
     tableau[i]=tableau[compt2]; //on ajoute un élément du second tableau
     compt2++;
    }
  }
  free(table1);
 }
void tri_fusion_bis(int tableau[],int deb,int fin) {
  if (deb!=fin) {
    int milieu=(fin+deb)/2;
    tri_fusion_bis(tableau,deb,milieu);
    tri_fusion_bis(tableau,milieu+1,fin);
```

```
    fusion(tableau,deb,milieu,fin);
  }
}
void tri_fusion(int tableau[],int longueur) {
  if (longueur>0) {
    tri_fusion_bis(tableau,0,longueur-1);
  }
}
```

– **version Objective Caml**

```
(* ------------------------------------------------------------------- *)
let fusion tableau deb1 fin1 fin2=
  let deb2=fin1+1 and tableau_bis=(make_vect (fin1-deb1+1) 0) in
  let compt1=ref(deb1) and compt2=ref(deb2) in
  for i=deb1 to fin1 do
    (* recopie de la premiÃ©re partie du tableau *)
    tableau_bis.(i-deb1)<-tableau.(i)
  done;
  for i=deb1 to fin2 do
    if (!compt1)=deb2 then
      ()
    else if (!compt2)=(fin2+1) then
      begin
        tableau.(i)<-tableau_bis.(!compt1-deb1);
        compt1:=(!compt1)+1
      end
    else if tableau_bis.(!compt1-deb1) < tableau.(!compt2) then
      begin
        tableau.(i)<-tableau_bis.(!compt1-deb1);
        compt1:=(!compt1)+1
      end
    else
      begin
        tableau.(i)<-tableau.(!compt2);
        compt2:=(!compt2)+1;
      end
  done;;
(* fusion : int vect -> int -> int -> int -> unit = <fun> *)

let tri_fusion tableau=

  let rec tri_fusion_bis tableau debut fin=
    if debut<>fin then
      begin
        let milieu=(debut+fin)/2 in
        tri_fusion_bis tableau debut milieu;
        tri_fusion_bis tableau (milieu+1) fin;
        fusion tableau debut milieu fin;
      end;

  in let longueur=(vect_length tableau) in
    if (longueur>0) then
      tri_fusion_bis tableau 0 (longueur-1);;
```

Exercice 2 : Fair threads

Le but de cet exercice est de fournir une simulation *réaliste* de ce qui se passe lors d'inscription d'élèves en Master.

Le schéma général de la procédure d'inscription comportera 4 étapes, i.e. 4 administrations :

- tout commencera dans l'administration `C` qui modèlisera le responsable de l'EFU de licence et qui est la première personne à qui les étudiants vont s'adresser.
- une fois que l'étudiant se sera declaré candidat à l'inscription, l'étudiant ira voir le secrétariat de l'EFU, nommé `L`.
- une fois que le secrétariat de `L` aura donné sa bénédiction, l'étudiant ira voir le secrétariat de `M`
- une fois le sésame obtenu au secrétariat `M` l'étudiant retournera voir `C` pour confirmer son inscription puis ira dans `Bar` où il se mettra en attente d'un évènement *happy_ our* à chaque réception d'un tel évenement chaque étudiant dans `Bar` écrira le message `hourra je suis inscrit` (*numero_ etudiant*), les évènements `happy_our` seront générés à intervalles réguliers.

Pour décrire les informations relatives à un etudiant on utilisera la struct suivante :

```
struct eleve {
int numero_etudiant;
char *nom;
char *prenom;
int nb_ue_acquises;
int *notes;
char **intitules;
char *universite_dorigine;
char *nationalite;
char *mention_licence;
double moyenne;
}
```

On disposera de plus d'un compteur `int nbetudiant_encours`.

On commencera par remarquer que les traitements effectués dans chacune des administrations n'ont aucune raison d'être synchrones, il se peut par exemple qu'il y ait une personne plus rapide en `M` qu'en `L` ou l'inverse, ceci prévaut aussi pour les traitements effectués dans `C` et `Bar`. On notera aussi qu'au plus un élève à la fois par administration peut être traité, mais que les traitements effectués dans les administrations peuvent s'exécuter en parallèle.

1. Modelisation

Compte tenu des informations ci-dessus expliquer quelle modélisation vous allez faire de ce problème (indication : pensez à ce qui pourrait être les schedulers). Pensez aussi à definir un moyen pour pouvoir garder un contact avec chaque élève.

2. Le générateur d'élèves

Tous les élèves seront crées par un unique thread `generator` qui :

- s'exécutera dans l'administration `C`
- créera un nouveau thread élève qui ne possède que la moitié des informations administratives.
- pour chaque élève créé, `generator` incrémente le compteur `nbeleve à inscrire`

Par souci de réalisme ce thread ne créera que 400 élèves avant de se terminer.

Ecrire une fonction `void generator_fun(void *)` qui réalise cette tâche.

3. Le générateur d'événements

Dans l'administration `Bar` un thread particulier `beer` engendre un événement `happy_our` que tous les élèves actuellement dans `Bar` recoivent. Pour ce faire, ce thread va procéder en tirant un nombre aléatoire entre 0 et 10 et si ce nombre est 0, un événement est génére sinon il rend la main.

Ecrire une fonction `void generator_fun(void *)` qui réalise cette tâche.

4. Les élèves

Pour des contingences de temps, les élèves ne feront vérifier aléatoirement qu'un quart des informations administratives par chaque secretariat, si d'aventure, parmi les les informations vérifiées il en manque une, l'élève devra retourner dans l'administration d'où il vient et récuperer cette information.

Pour vérifier une information on utilisera la fonction `int checkinfo(struct eleve el,int numero_info)` et pour acquérir une information la fonction `void getinfo(struct eleve*, int numero_info)`.

Une fois la procédure d'inscription terminée, et avant d'aller dans l'administration `Bar` les étudiants retournent dans l'administration `C` pour y décrémenter le compteur du nombre d'élèves à inscrire. Si ce compteur tombe à 0 un événement `dernier inscrit` est engendré.

Ecrire une fonction `void eleve_fun(void *)` qui réalise cette tâche.

5. Le killer

Afin d'arrêter au bon moment tout le processus d'inscription on va utiliser un thread de nettoyage `Leon`. Ce thread va se lier à l'administration `C` et va attendre un événement `dernier inscrit` qui sera déclenché lorsque plus aucun étudiant n'aura à s'inscrire.

Une fois réveillé ce thread sort de l'adminstration `C` et stoppe tous les étudiants avant de se terminer.

Ecrire une fonction `void Leon(void *)` qui réalise cette tâche.

6. Le main

Ecrire une fonction main qui crée les schedulers, les threads generator leon et beer, et les lance dans cet ordre. Expliquer pourquoi les 400 élèves sont obligatoirement engendrés en utilisant les fair threads.

Exercice 3 : Esterel

Le but de cet exercice est de simuler une chaîne de production comprenant deux machines M1 et M2 et un tapis roulant permettant d'acheminer à M2 pour finition les pièces fabriquées par M1.
Pour simplifier, on suppose que

- la fabrication, le transport et la finition se font pièce par pièce,
- l'ensemble est suspendu à l'instant où le signal `SUSPENSION` est émis,
- seulement à l'instant où le signal `REPRISE` est émis, chacun reprend son activité à partir de son point de suspension,

1. Ecrire le module `suspendre` contenant une boucle qui
   - attend le signal `SUSPENSION`,
   - à la réception de ce signal, fait une pause et émet ensuite en continu le signal `SUSPENSION`,
   - à la réception du signal `REPRISE` recommence la boucle.

   Ce module permet donc d'entretenir l'émission du signal `SUSPENSION` à chaque tick jusqu'à la réception du signal `REPRISE`.

   Pour la suite de l'exercice, on suppose connue la fonction C

   ```
   int alea_min_max (int min, int max);
   ```

   qui permet de générer un nombre aléatoire compris entre le `min` et le `max`.
2. Ecrire le module `M1` contenant une boucle qui
   - se met en suspension à la réception du signal `SUSPENSION`,
   - attend un nombre aléatoire (compris entre 2 et 3) de tick (c'est une durée supposée pour fabriquer une pièce),
   - émet en continu, à l'attention du `transporteur`, le signal `PRODUIT_PRET_T` pour lui indiquer qu'une pièce est prête pour être acheminée vers `M2`,
   - attend le signal `T_PRODUIT_PRIS` émis par le transporteur qui a pris en charge la pièce et recommence à fabriquer une nouvelle pièce.
3. Ecrire le module `M2` contenant une boucle qui
   - se met en suspension à la réception du signal `SUSPENSION`,
   - attend le signal `T_PRODUIT_PRET` émis par le `transporteur` pour lui indiquer qu'une pièce est arrivée,
   - fait une pause avant de prendre la pièce en émettant le signal `PRODUIT_PRIS_T` à l'attention du `transporteur` pour lui indiquer que la pièce est prise,
   - attend un nombre aléatoire (compris entre 2 et 3) de tick (c'est une durée supposée pour la finition d'une pièce),
   - émet, à l'attention de l'`usine` le signal valué `COMPTEUR` avec une valeur aléatoire comprise entre 0 et 19.
4. Ecrire le module `transporteur` contenant une boucle qui
   - se met en suspension à la réception du signal `SUSPENSION`,
   - attend le signal `M1_PRODUIT_PRET` émis par `M1` pour lui indiquer qu'une pièce est prête à être acheminée,
   - fait une pause avant de prendre la pièce en émettant le signal `PRODUIT_PRIS_M1` à l'attention du `M1` pour lui indiquer que la pièce est prise,
   - attend un nombre aléatoire (compris entre 2 et 4) de tick (c'est une durée pour acheminer une pièce à M2),

– émet en continu, à l'attention de `M2` le signal `PRODUIT_PRET_M2` pour lui indiquer qu'une pièce est arrivée,
– attend le signal `M2_PRODUIT_PRIS` émis par `M2` qui a pris la pièce pour recommencer sa boucle.

La valeur 0 (respectivement <> 0) associée au signal `COMPTEUR` émise par `M2` indique que la pièce est bonne (respectivement mauvaise).

5. Ecrire le module `usine` qui met en marche et en concurrence `M1`, `M2`, `transporteur` et des instructions permettant d'afficher, à chaque signal `COMPTEUR` reçu, le nombre de bonnes et de mauvaise pièces fabriquées depuis le début.

Voici un exemple du fonctionnement de l'usine :

```
usine> ;;;;;;;;;;;;;;;;;;;;SUSPENSION;;;;;REPRISE;;;;;;
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output: BON(1) MAUVAIS(0)
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output: BON(2) MAUVAIS(0)
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output:
--- Output: BON(3) MAUVAIS(0)
--- Output:
```