COMPTE RENDU MULTITHREADING

Binöme :

GIACOMONI Jessy
BROCHETTO Sylvain

Année :
2013 - 2014

Groupe :
I4 - Groupe 2

Expliquer le fonctionnement du schéma producteur / consommateur.

Le but de ce schéma est de faire en parallèle la producteur et le consommation, prenons un exemple. Nous avons une station météo nous devons donc collecter plusieurs informations (producteurs) puis ensuite les transferer aux clients (consommateurs de ces données). De plus, nous pouvons collecter de manière parallèles tous les capteurs météo (par exemple: capteur d'humiditié, pression etc.). Le consommateur n'a pas besoin que toutes les mesures soient terminées afin d'afficher ou de faire un traitement spécial par rapport à toutes les données. C'est ce que l'on nomme le schéma consommateur/producteur.

Prenons maintenant le code qui a été founit pour l'excercice 1 :

```
void *producteur(void *arg) {
   pthread_t id = pthread_self();
   int *parametre = (int *)arg;

   sleep(1);
   pthread_mutex_lock(&verrou);
   while(_index == TAILLE) {
         printf(«wait pour de la place\n»);
      pthread_cond_wait(&attente_de_la_place, &verrou);
   }
   buffer[_index] = *parametre;
   printf(«producteur : %d ecrit a __indexe %d\n», *parametre, _index);
   _index++;
   printf(«on signal qu'il y a un truc a lire\n»);
   pthread_cond_signal(&attente_quelque_chose_a_lire);
   // on reveil un lecteur car maintenant, il peut lire
   pthread_mutex_unlock(&verrou);
}
```

Explication de la fonction :

- On stocke dans des variables l'identificateur du thread ainsi que le numéro de paramètre.
- Suspension du thread courant.
- On rentre dans la section critique en bloquant l'accès au buffer.
- On pose le verrou pour laisser l'accès au producteur
- Mise en attente du thread tant que le tube est plein.
- Dès lors que le tube n'est plus plein, on écrit notre paramètre reçu dans la case du buffer d'indice (index)
- On incrémente _index pour que le producteur écrive dans la case suivante au prochain tour
- On envoit un signal qui permet d'avertir les consommateurs
- Déverouillage du mutex pour permettre la lecture au consommateur

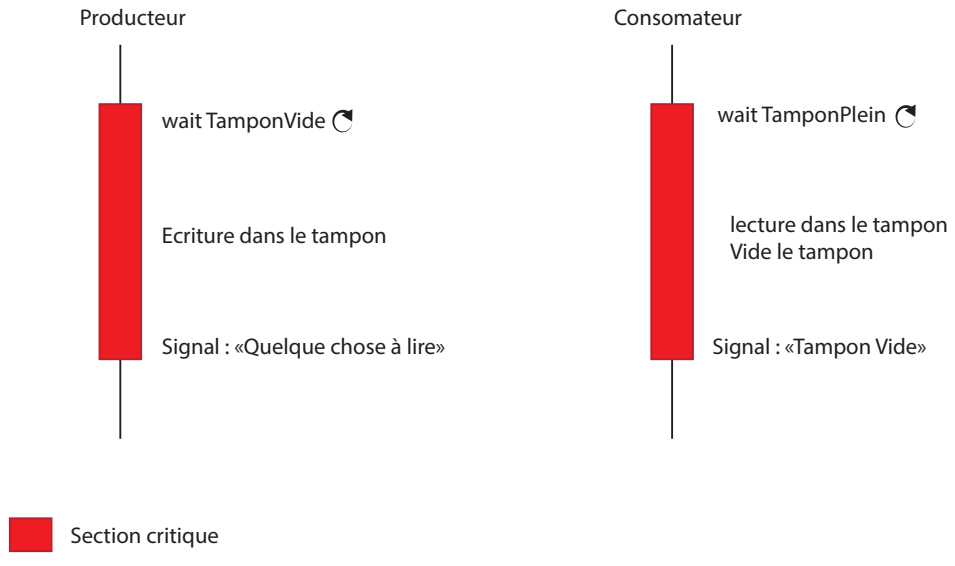
```
void *consommateur(void *arg) {
   pthread_t id = pthread_self();
   int *parametre = (int *)arg;


   pthread_mutex_lock(&verrou);
   while(_index == 0){
         printf(«wait pour avoir quelque chose a consommer\n»);
      pthread_cond_wait(&attente_quelque_chose_a_lire, &verrou);
   }
   _index--;
   printf(«consommateur %d lit %d a __indexe %d\n», *parametre, buffer[_index], _index);
   printf(«on signal qu'il ya de la place\n»);
   pthread_cond_signal(&attente_de_la_place);
   // On reveil un ecrivain car maintenant, il peut ecrire
   pthread_mutex_unlock(&verrou);
}
```

Explication de la fonction :

- On stocke dans des variables l'identificateur du thread ainsi que le numéro de paramètre.
- On pose le verrou pour laisser l'accès au consommateur, on rentre dans la section critique.
- Mise en attente du thread tant que le tube est vide
- On décrémente _index pour que le prochain consommateur preleve la variable de la case precedente
- On signale au producteur qu'il dispose d'espace pour écire, en reveillant le thread en attente
- Déverouillage du mutex pour permettre l'écriture du producteur

Voici un schéma permettant de faire un récapitulatif

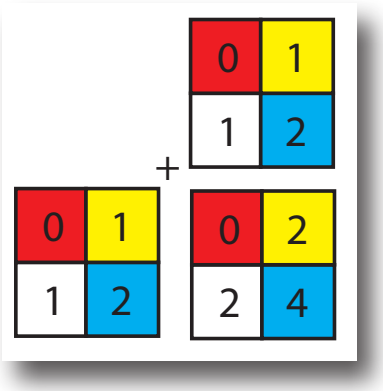


1) Addition entre deux matrices

Le but ici est d'addtionner deux matrices carrées ensemble. Afin de résoudre le plus rapidement possible une addition de deux matrices nous faisons appel aux threads. En effet, le calcul des lignes sont indépendantes donc nous pouvons additionner une ligne avec un thread puis une autre avec un autre thread etc.

Dans ce genre d'exercice il faut donc placer la valeur calculer dans un tableau et dans lequel celui-ci est rempli par tous les threads. Afin que tous les threads puissent placer les valeurs calculées il nous faut donc déclarer une matrice globale qui sera notre matrice de résultat. Sachant que les lignes sont indépendantes les unes aux autres nous n'avons pas besoin de synhroniser chaque threads, par besoin d'attendre un résultat précédent.

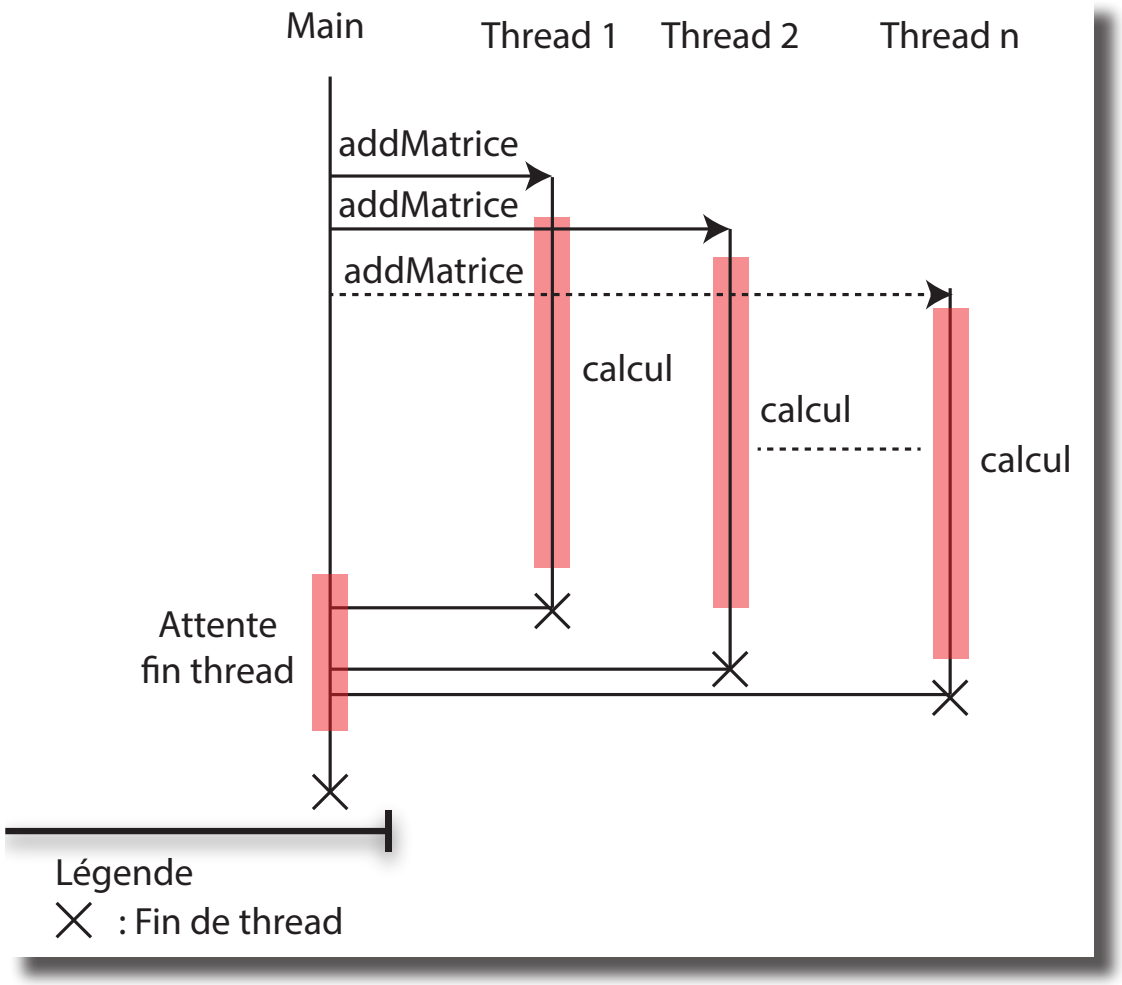
Voici un schéma permettant de préciser le principe de calcul avec les threads



Les différentes couleurs (rouge, bleu) permettent d'exprimer le calcul d'un thread. Nous avons donc deux matrice déjà construit [[0,1][1,2]] et nous les ajoutons ensemble et cela nous donne : [[0,2][2,4]].

Explication du main principal

Le but de l'exercice est de montrer qu'avec deux matrices carrées ont peut les additionner sans encombre.
On représente ci-après le déroulement de notre programme avec une dimension n.


Main
Thread 1
Thread 2
Thread n
addMatrice
addMatrice
addMatrice
calcul
calcul
calcul
Attente
fin thread
Légende
: Fin de thread

Algorithme :

Légende des couleurs algorithmes :
rose : entrée
bleu : variable globale
rouge : variable locale
vert : commentaire

Programme main :

```
//Nous supposons que nous avons 3 matrices déclaré comme global comme suit
_premiereMatrice
_secondeMatrice
_resultatMatrice

//On ajoute comme variable global la taille de chaque matrice
_tailleMatrice
```

Algorithme fonction : main
Entree : entier DimensionDesMatrices

```
algorithme :
entier i,j,tailleMatrice, nombreThreads <- 0
thread threads

Si DimensionDesMatrices = null alors
        ecrire «Information : vous devez renseigner la taille de la matrice»
        exit
finSi

_tailleMatrice <- DimensionDesMatrices
nombreThreads <- _tailleMatrice

_premiereMatrice <- allocateMatrice()
_secondeMatrice <- allocateMatrice()
_resultatMatrice <- allocateMatrice()

Pour i allant de _tailleMatrice faire
        Pour j allant de _tailleMatrice faire
                _premiereMatrice [i][j] <- i + j
                _secondeMatrice [i][j] <- i + j
        finPour
finPour
```

```
Pour i allant de nombreThreads faire
        worker(hanlde)  //Appel de l'Handler pour faire le calcul de la matrice
finPour

Pour i allant de numbreThreads faire
        Attendre fin handler //On attend que chaque threads est terminé leur tâche
finPour

ecrire «Résultat de la mutliplication»
ecrire _resultatmatrice
```

Algorithme handler : worker

```
Entrée : numéroLigne

algorithme :
        additionMatrice(numéroLigne)
```

Algorithme fonction : additionMatrice

```
Algorithme :
Entier i,k <- 0

Pour k allant de _tailleMatrice faire
        _resultatMatrice[i][k] <- _premiereMatrice[i][k] + _secondeMatrice[i][k];
finPour
```

Approximation d'une intégrale

Pour faire l'approximation d'une intégrale nous avons d'abord chercher qu'est ce qu'il pouvait se faire en parallèle. C'est à dire quels sont les calculs qui sont indépendants dans ce calcul. Voici les différentes étapes.

$$T_n = \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right)$$

(1) (2)

Nous avons ici deux calculs indépendant: Celui en dehors de la parenthèse et celui dedans la parenthèse. Allons plus loin et étudions l'intérieur de la parenthèse.

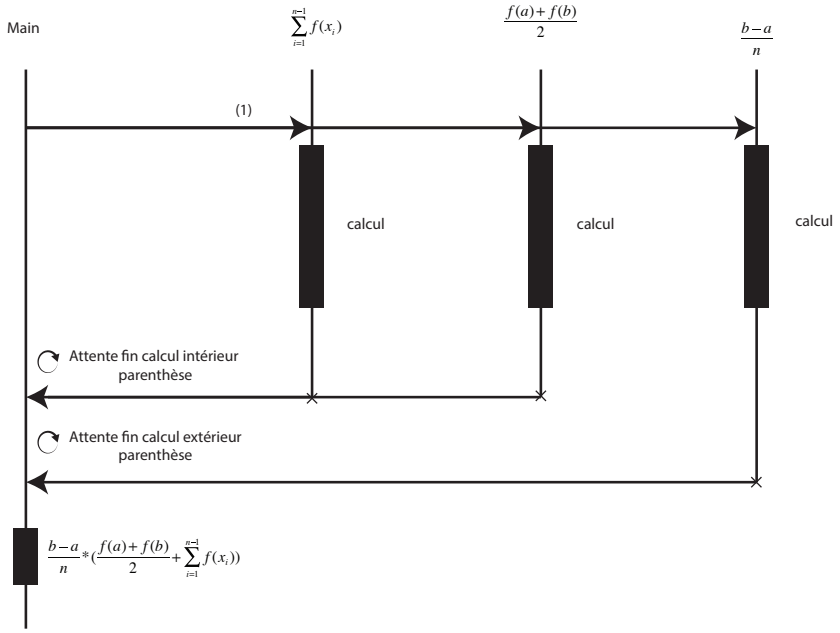
$$\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right)$$

(3) (4)

Dans cette parenthèse nous avons encore deux calculs dépendants.

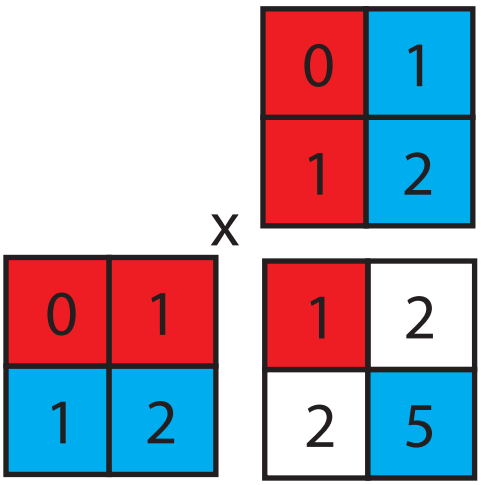
Ainsi pour résoudre cet excercice nous avons besoin de 4 threads indépendant. Nous allons calculer le résultat final dans le thread principal c'est à dire dans le main.

Voici, un schéma global permettant d'expliquer le fonctionnement de l'ensemble de notre programme.



Produit de deux matrices.

Pour mettre en place la programmation pour le calcul de la multiplication de deux matrices il nous faut regarder le fonctionnement de la multiplication de deux matrices de dimension n, ici 2.



Element clé pour la résolution de l'excercice

Nous voyons comme pour l'addition le calcul des lignes sont indépendantes. Ainsi nous allons faire un programe assez similaire à celui de l'exercice 2. Nous allons calculer chaque ligne avec un thread différent. Nous pouvons aller encore plus loin. Nous pouvons créer une thread par case de la matrice. En effet, le calcul de chaque case sont indépendant. Il faut juste un accès aux deux matrices et une matrice résultat qui sera utilisé par tous les threads. Comme chaque case est utilisée par un thread différent alors nous n'aurons pas de problème réecriture dans cette matrice.

Nous allons donc définir 3 matrices globales. Dans les 3 matrices 2 seront les matrices dîtes «entrantes» et la 3ieme sera la matrice résultat.

Pour cet excercice les valeurs dans les matrices ne sont pas importantes donc nous mettons des valeurs qui sont directement rempli par le programme et non par l'utilisateur dans le programme main.

Algorithme :

Légende des couleurs algorithmes :
rose : entrée
bleu : variable globale
rouge : variable locale
vert : commentaire

Programme main :

```
//Nous supposons que nous avons 3 matrices déclaré comme global comme suit
_premiereMatrice
_secondeMatrice
_resultatMatrice

//On ajoute comme variable global la taille de chaque matrice
_tailleMatrice

Fonction : Main
Entree : entier DimensionDesMatrices

algorithme :

entier i,j,tailleMatrice, nombreThreads <- 0
thread threads

Si DimensionDesMatrices = null alors
        ecrire «Information : vous devez renseigner la taille de la matrice»
        exit
finSi

_tailleMatrice <- DimensionDesMatrices
nombreThreads <- tailleMatrice

_premiereMatrice <- allocateMatrice()
_secondeMatrice <- allocateMatrice()
_resultatMatrice <- allocateMatrice()

Pour i allant de _tailleMatrice faire
        Pour j allant de _tailleMatrice faire
                _premiereMatrice [i][j] <- i + j
                _secondeMatrice [i][j] <- i + j
```

```
        finPour
finPour

Pour i allant de nombreThreads faire
        worker(hanlde)  //Appel de l'Handler pour faire le calcul de la matrice
finPour

Pour i allant de numbreThreads faire
        Attendre fin handler //On attend que chaque threads est terminé leur tâche
finPour

ecrire «Résultat de la mutliplication»
ecrire _resultatmatrice
```

ALGORITHME HANDLER : WORKER

```
Entrée : numéroLigne

algorithme :
        multiplicationMatrice(numéroLigne)
```

ALGORITHME FONCTION : MULTIPLICATIONMATRICE

```
Entrée : Entier numéroLigne

Algorithme :

Entier i,j,k, resultat <- 0
Entier activeLigne <- numéroLigne

Pour i allant de _tailleMatrice faire
        resultat <- 0.0
        Pour k allant de _tailleMatrice faire
                résultat <- resultat + _premiereMatrice[i][k] * _secondeMatrice[k][j];
        finPour
        _resultatMatrice[i][j] <- resultat
finPour
```

Le cycle de vie des threads est représenté ci-contre. On trouvera un grand approchement que celui que nous avions pour l'addition. Pour plus de clarter on ne met la phase de transition du handler.