**Université Bordeaux 1 Master Informatique** **IN8W16**
Examen du jeudi 19 avril 2012 Durée : 1h30 – Sans document

## Programmation Multicœur et GPU

*Le barème indicatif est sur 26 points – sélectionnez vos questions !*

Le sujet porte sur une sorte de jeu de la vie sauf qu'à chaque itération la valeur suivante d'une cellule est le max de cette cellule et ses 4 voisines (Nord, Sud, Est, Ouest). Si cette explication vous suffit, vous pouvez directement répondre à la question 1.

### Introduction au problème

*On souhaite identifier les objets blancs présents sur une image en noir et blanc. On travaille ici en 4-connexité : deux pixels sont connexes s'il sont adjacents par un bord (nord, sud, est ou ouest). Deux pixels blancs appartiennent au même objet s'ils sont connexes ou s'il existe une chaine de pixels connexes blancs les reliant. Dans le tableau ci-contre les pixels contenant 1 et ceux contenant 2 forment deux objets 4-connexes distincts.*

| i \ J | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 2 |
| 1 | 0 | 1 | 0 | 2 | 2 |
| 2 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |

*Pour identifier les objets on commence par attribuer l'entier 0 aux pixels noirs et un entier distinct à chaque pixel blanc. Ensuite, on propage l'identité maximale dans le voisinage des pixels blancs en itérant le calcul.*

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | | 1 | | | 2 |
| 1 | | 3 | | 4 | 5 |
| 2 | | 6 | 7 | | |
| 3 | | | 8 | | |
| 4 | | | | | |

*Au bout d'un nombre d'itérations (borné par le nombre de pixels) les pixels appartenant au même objet ont tous acquis la même identité, celle du pixel de plus grande identité. La propagation stagne, le calcul est terminé.*

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | | 8 | | | 5 |
| 1 | | 8 | | 5 | 5 |
| 2 | | 8 | 8 | | |
| 3 | | | 8 | | |
| 4 | | | | | |

*Pour réaliser cette propagation, il est naturel d'itérer jusqu'à stagnation un balayage de l'ensemble des pixels en attribuant à chaque pixel blanc (non nul) l'identité maximale entre celle du pixel considéré et celles des quatre pixels voisins. L'efficacité d'un tel algorithme basé sur une succession de parcours classiques (*for (i=0 ; i < MAX ; i++) for (j=0 ; j < MAX ; j++) *...) de l'ensemble des pixels peut être sensiblement amélioré en remplaçant le balayage ordinaire par deux balayages :*

- *le premier balaye les pixels dans le sens direct* for (i=0 ; i < MAX ; i++) for (j=0 ; j < MAX ; j++) *...*
- *le second dans le sens inverse :* for (i=MAX-1 ; i >=0 ;i--) for (j=MAX-1 ; j >=0 ;j--)*...*

*Il apparaît alors peu utile de calculer le maximum sur l'ensemble des 4 pixels voisins mais simplement sur ceux favorisant le plus la propagation du max.*

*Par convention posons que le sens descendant corresponde au parcours du tableau de gauche à droite puis de haut en bas ; pour faire descendre le max il suffit de comparer les identités de la cellule considérée à celles des cellules ouest et nord. Le sens montant correspond au parcours du tableau de droite à gauche puis de bas en haut, on fait remonter le max en consultant les cellules sud et est. Le code de ce programme est à pour titre* ***code 1****.*

| | | |
|---|---|---|
| | *N* | |
| *O* | *C* | |
| | | |

*descendant*

| | | |
|---|---|---|
| | | |
| | *C* | *E* |
| | *S* | |

*Montant*

*Notons que la propagation peut être effectuée en parallèle sans grande précaution car l'ordre des calculs importe peu pourvu qu'on arrive à la stagnation.*

## Partie OpenMP et MPI (code 1)

**Question 1.** (1pt) Soit un processeur disposant de caches L1 (taille = 32ko, associativité = 4), L2 (t = 256ko, a = 8) et L3 (t = 8Mo, a = 16). En supposant que l'algorithme d'évincement de cache est de type LRU (Least Recently Used) et que les lignes de caches font 64 octets, donner très grossièrement les nombres de défauts de cache L1, L2 et L3 pour la procédure identifier_objets() du **code 1** en supposant qu'on fasse *n* fois la boucle while.

NB. On supposera que le cache est initialement vide. L'image pèse 512 x 512 x 4o= 1 Mo.

**Question 2.** (5 points) Paralléliser le **code 1** donné à l'aide d'OpenMP. Il s'agit de :

- réécrire la fonction identifier_objets en cherchant à créer une seule fois l'équipe de threads (attention à bien gérer la terminaison de la boucle, tous les threads doivent faire le même nombre de boucles) ;
- ajouter les directives nécessaires à la parallélisation des boucles for de la fonction descendre_max (la parallélisation de monter_max est identique) *;*
- justifier le fait d'utiliser ou pas sections critiques et opérations atomiques.

NB. Le code produit n'a pas à respecter les dépendances de données induites du code 1 car l'ordre des calculs importe peu pourvu qu'on arrive à la stagnation.

**Question 3.** (1pt) En supposant que l'on dispose de 8 cœurs ayant chacun leurs propres caches L2 mais partageant le cache L3, indiquer très grossièrement le nombre de défauts de cache obtenus par cœur pour la version parallèle pour n itérations. Peut on faire mieux ? Le cas échéant donner (sans coder) des indications d'optimisation.

**Question 4.** (5 points) Écrire un pseudo code MPI permettant de distribuer efficacement cet algorithme. Bien préciser les appels d'émission et de réception de message, bien faire apparaître la détection de la terminaison de l'algorithme.

## Partie GPU

*Pour propager le maximum selon une stratégie "proche" du programme séquentiel, on peut utiliser une propagation en quatre étapes successives (au lieu de deux) : vers_le_bas, vers_la_droite, vers_le_haut, vers_la_gauche. On suppose, pour simplifier, que les images sont carrées et de dimension NxN. On décide, pour chaque noyau, de lancer N threads selon une seule dimension. Voici une proposition pour le noyau vers_le_bas:*

```
__kernel void vers_le_bas(__global int *img,
                          unsigned int N)
{
  unsigned int x = get_global_id(0);
  int mx = 0, val;

  for(unsigned int y = 0; y < N, y++) {
     val = img[ y*N + x ];
     mx = (val ? (val > mx ? val : mx) : 0)
     img[ y*N + x ] = mx;
  }
}
```

**Question 6.** (1 point) Dans le code ci-dessus, vous noterez que, même lorsque qu'un pixel vaut 0 ou devient le nouveau maximum, l'écriture "img[ y*N + x ] = mx;" est tout de même effectuée. À votre avis pourquoi ?

**Question 7.** (5 points) En suivant le même principe, écrire le noyau vers_la_droite. Discutez de l'efficacité des accès mémoire de cette version. Proposez une version optimisée utilisant de la mémoire locale.

**Partie tâches (code 2)**

*Expérimentalement on s'aperçoit que si l'on se contente de paralléliser les boucles for, la version parallèle effectue significativement plus d'itérations que la version séquentielle en présence de « gros » objets. En effet, admettons que tous les pixels soient blancs alors la version parallèle nécessitera autant d'itérations qu'il y a de threads pour remonter le max au lieu d'une seule pour la version séquentielle. Une piste pour améliorer l'efficacité de la parallélisation est de sacrifier un peu de parallélisme pour conserver la bonne transmission du max.*

*Sur le schéma ci-contre chaque case représente un bloc de pixels. Le contenu de chaque macro-cellule est traité de façon séquentielle. À la première étape on traite la macro-cellule marquée 1, une fois traitée on peut traiter celles marquées 2 et ainsi de suite. De façon générale on peut traiter une macro-cellule dans le sens descendant (resp. montant) si ses voisines nord et ouest (resp. sud et est) ont été traitées. Le **code 2** implémente en séquentiel cet algorithme.*

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 |
| 3 | 4 | 5 | 6 |
| 4 | 5 | 6 | 7 |

Sens descendant

| | | | |
|---|---|---|---|
| 7 | 6 | 5 | 4 |
| 6 | 5 | 4 | 3 |
| 5 | 4 | 3 | 2 |
| 4 | 3 | 2 | 1 |

Sens montant

**Question 8.** (1 point) Ce schéma de calcul respecte bien la transmission du max mais est peu parallèle au départ et à la fin du traitement. Donner une borne maximale pour le speed-up lorsque l'on parallélise le traitement à l'aide de n x n macro-cellules. Expliquer.

**Question 9.** (5 points) Paralléliser le **code 2** en faisant exécuter le traitement chaque macro-cellule par une tâche OpenMP. Il s'agit de :

- réécrire la fonction identifier_objets _cell en cherchant à créer une seule fois l'équipe de threads ;
- réécrire la fonction lancer_descente en utilisant les directives nécessaires à sa parallélisation via des tâches OpenMP (la parallélisation de lancer_monte() est identique).

**Question 10.** (2 points) Quel est le principal défaut de l'implémentation via des tâches OpenMP ? Indiquer comment paralléliser ce même code avec des pthreads. Expliquez en particulier comment résoudre le travers observé par l'utilisation des tâches OpenMP ?