Recherche des composantes connexes dans une image
en OpenMP

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'ils 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.

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.

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.

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 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 < DIM; i++) for (j=0 ; j < DIM; j++) *...) de l'ensemble des pixels peut être sensiblement amélioré en remplaçant le balayage ordinaire par deux balayages :*

	0	1	2	3	4
0		8			5
1		8		5	5
2		8	8		
3			8		
4					

- *le premier balaye les pixels dans le sens habituel :*

```
for (i=0; i < DIM; i++)
    for (j=0; j < DIM; j++) ...
```

- *le second dans le sens inverse :*

```
for (i=DIM-1; i >=0 ;i--)
    for (j=DIM-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. Cet algorithme est implémenté par la fonction propager_max_v0 *(dans* compute.c*).*

	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.

Étape 1 : OpenMP FOR

Parallélisez le premier code donné à l'aide de boucles OpenMP sans forcément respecter les dépendances de données induites par le code séquentiel : l'ordre des calculs importe peu pourvu qu'on arrive à la stagnation. Ici, il s'agit de paralléliser les fonctions descendre_max et monter_max de la façon la plus simple possible (ajout de deux directives).

- Vérifiez que le code le bon fonctionnement du code avec l'option –l images/spirale.png
- Mesurez les performances obtenues pour DIM = 2048 avec des spirales de 100 tours (option -t 100)

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) après que ses voisines nord et ouest (resp. sud et est) ont été traitées.

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

La version séquentielle de cet algorithme est implémentée par la fonction propager_max_v2 *(dans* compute.c*), le nombre de macro-cellules étant fixé par la constante GRAIN (il y a GRAIN² macro-cellules).*

Étape 2 : Tâches OpenMP

Parallélisez le code (version v2) en faisant en sorte que chaque macro cellule soit réalisée par une tâche OpenMP. On utilisera alors la clause depend pour contraindre l'ordre d'exécution des tâches afin de conserver l'ordre séquentiel de la propagation du max. Pour simplifier l'expression des dépendances on utilisera les tableaux cellulem[GRAIN][GRAIN] et celluled[GRAIN][GRAIN].

- À nouveau, vérifiez que le code le bon fonctionnement du code avec l'option –l images/spirale.png
- Mesurez les performances obtenues pour DIM = 2048 avec des spirales de 100 tours.
- Calculez une borne théorique maximale en fonction du GRAIN et en supposant que l'on dispose d'un nombre non borné de processeurs.
- Mesurez les accélérations obtenues par rapport à la version séquentielle en faisant varier la taille du grain.

Étape 3 : Placement des tâches sur machine NUMA

Les machines du CREMI (salle 203 par exemple) possèdent une architecture à accès mémoire non uniforme (NUMA). Il est possible de vérifier le nombre de bancs mémoires à l'aide de la commande lstopo. On s'intéresse désormais à l'influence du placement des tâches dans notre programme sur le temps d'exécution. Il s'agit de comparer les performances obtenues par la parallélisation à l'aide des tâches OpenMP (version v3) à celles obtenues par un ordonnanceur ad hoc (version v4) qui permet de placer explicitement les tâches sur les différents cœurs. En particulier, le cœur choisi pour la tâche en charge de la macro-cellule (i, j) est choisi dans la fonction ajouter_job. Dans l'état actuel, le cœur est choisi de la façon suivante :

int cœur = i % P ;

À quoi correspond cette politique de distribution ? Est-elle stupide ?

Influence du cache L3

Relevez les différents temps d'exécution des différentes versions pour une taille d'image 2048x2048, un grain de 32, et un twist assez grand (500). Pour les versions parallèles, utilisez OMP_NUM_THREADS=12.

Comparer les versions parallèles avec et sans first-touch (option –ft). Est-ce que cela a une influence sur les temps d'exécution ?

Pour mettre en valeur l'influence du cache L3, on pourra modifier le choix du cœur en adoptant un placement différent à la montée et à la descente :

cœur = ((sens == 1) ? i + 6: i) % P;

Influence sur les bancs mémoire

Reprendre le placement initial pour les taches de calcul (cœur = i % P). Mesurer le temps d'exécution pour la taille d'image 4096x4096, un grain de 32, et un twist assez grand (500). Pour les versions parallèles, utilisez OMP_NUM_THREADS=12.

Utilisez le script shell et le script R pour produire une courbe de speed up fonction du nombre de threads.

```
Rscript speedUp.R <fichier de données> <temps séquentiel>
```

Reprendre les mesures uniquement pour la version pthreads avec le paramètre -ft après avoir modifier la stratégie de placement du first-touch (ajouter la chaine anti-ft dans le nom du fichier de sortie) :

cœur = ((sens == 2) ? i + 6 : i) % P;