Programmation Parallèle

Fiche 4 : Cuisine et dépendances

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/pap/

Cette fiche porte sur la recherche de composantes connexes dans une image en OpenMP. Nous utiliserons pour cela le noyau `max` dont une version séquentielle vous est fournie dans EASYPAP. Récupérez les dernières mises à jour à l'aide d'un discret `git pull`.

1 Introduction au problème

On souhaite identifier les objets non-transparents présents sur une image. On travaille ici en 4-connexité : deux pixels sont connexes s'ils sont adjacents par un bord (nord, sud, est, ouest). Deux pixels $\neq 0$ appartiennent au même objet s'ils sont connexes ou s'il existe une chaine de pixels (non nuls) connexes les reliant.

Pour identifier les objets on commence par attribuer une couleur unique à chaque pixel $\neq 0$. Ensuite, on propage l'identité maximale dans le voisinage des pixels colorés 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 couleur, celle du pixel de plus grande ouleur. La propagation a atteint un état stable, le calcul est terminé.

Pour réaliser cette propagation, il est naturel d'itérer jusqu'à stagnation un balayage de l'ensemble des pixels en attribuant à chaque pixel 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 de l'ensemble des pixels peut être sensiblement amélioré en remplaçant le balayage ordinaire par deux balayages :

```
// premier balaye les pixels dans le sens « habituel »
for (int i = 0; i < DIM; i++)
  for (int j = 0; j < DIM; j++)
    ...

// puis un second parcours dans le sens inverse
for (int i = DIM-1; i >= 0 ;i--)
  for (int 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 `max_compute_seq` (dans `max.c`).

Vous pouvez le tester ainsi :

```
./run -l images/spirale.png -k max
```

2 Parallélisation avec une clause omp for

Notons que la propagation peut être effectuée en parallèle sans grande précaution car l'ordre des calculs importe peu pourvu que l'on rejoigne un état stable.

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 `tile_down_right` et `tile_up_left` (**en les recopiant** pour conserver les versions séquentielles) de la façon la plus simple possible (ajout de deux directives).

— Vérifiez que le code le bon fonctionnement du code sur l'image `spirale.png` :

```
./run -l images/spirale.png -k max -v omp
```

— Mesurez les performances obtenues pour `DIM` = 2048 avec des spirales de 100 tours :

```
./run -s 2048 -k max -v omp -a 100 -n
```

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 opaques 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.

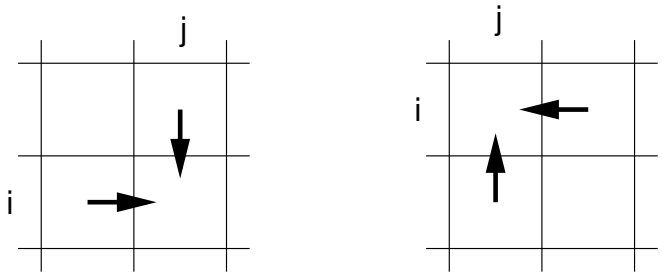
3 Parallélisation avec des tâches

Une piste pour améliorer l'efficacité de la parallélisation est de sacrifier un peu de parallélisme pour conserver la bonne transmission du max. Sur la figure 1 ci-dessous, chaque case représente une « tuile » de pixels. Le contenu de chaque tuile est traité de façon séquentielle.

Dans la phase descendante (Figure 1a), on traite d'abord la tuile en haut à gauche. Puis, d'une façon générale, on peut traiter une tuile après que ses voisines Nord et Ouest ont été traitées.

Symétriquement, dans la phase montante (Figure 1b), on traite d'abord la tuile en bas à droite. Puis, d'une façon générale, on peut traiter une tuile après que ses voisines Sud et Est ont été traitées.

La version « tuilée » de cet algorithme est implémentée par la fonction `max_compute_tiled` (dans `max.c`), le nombre de tuiles étant fixé par `NB_TILES_X` $\times$ `NB_TILES_Y`. Pour simplifier, on pourra utiliser des tuiles carrées en fixant un nombre identique horizontalement et verticalement à l'aide de l'option `--nb-tiles` (ou `-nt`). Exemple en utilisant 16×16 tuiles :



(a) descente (b) montée

FIGURE 1 – Dépendances entre tuiles lors de la descente (a) et lors de la montée (b). C'est curieux, ça rappelle un exercice de la fiche précédente non ?

```
./run -s 2048 -k max -v tiled -a 100 -nt 16
```

Parallélisez le code de la version tuilée (en créant une nouvelle variante `max_compute_task`) 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 pourra utiliser un tableau annexe qui ne servira que pour exprimer les dépendances :

```
int tuile[NB_TILES_Y][NB_TILES_X + 1] __attribute__ ((unused));
```

À nouveau, vérifiez le bon fonctionnement du code sur l'image `spirale.png` :

```
[my-machine] ./run -k max -v task -l images/spirale.png -nt 16 -n
Using kernel [max], variant [task]
Computation completed after 284 iterations
```

Vérifiez également, au moyen d'une trace d'exécution, que les dépendances entre tâches sont bien respectées. Vous pouvez par exemple utiliser cette séquence de commandes :

```
# generate thumbnails
[my-machine] ./run -k max -v task -l images/stars.png -nt 16 -tn -n
# trace
[my-machine] ./run -k max -v task -l images/stars.png -nt 16 -t -n
# observe
[my-machine ./view
```

En bougeant la souris au-dessus du diagramme de Gantt, de la gauche vers la droite, vous devriez observer un « front » de tâches progressant vers le coin en bas à droite (voir Figure 2, page 4), puis vers le coin en haut à gauche, etc.

— Mesurez les performances obtenues pour `DIM` = 2048 avec des spirales de 100 tours.
— Calculez une borne théorique maximale en fonction du nombre de tuiles selon une dimension (e.g. `NB_TILES_X`) 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 le nombre de tuiles.

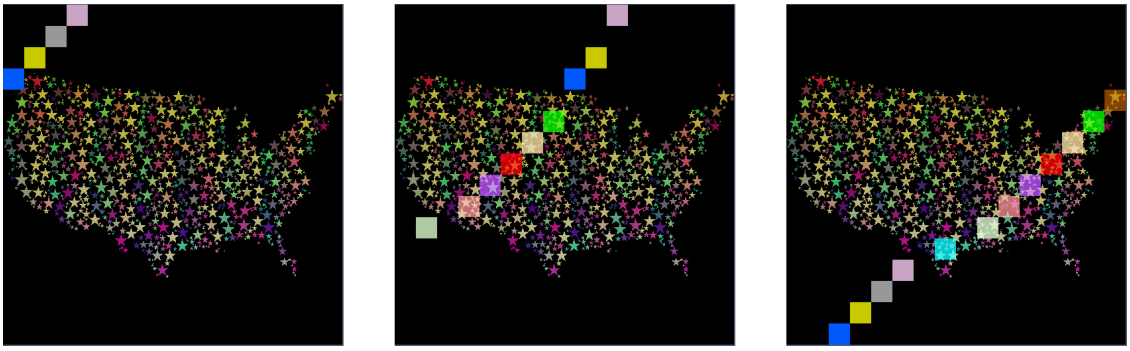


FIGURE 2 – Progression du front de tâches lors de la phase descendante de la propagation du maximum.