UNIVERSITÉ D'ORLÉANS **M1 Info**

Informatique Année **2022 - 2023**

Contrôle : Programmation Parallèle - 2h

Exercice 1. Quelques questions (4pts)

1. A désigne un tableau d'entiers de longueur n que l'on souhaite distribuer par morceaux à $nprocs$ processeurs. Si $n = 22$ et $nprocs = 4$ donnez le contenu des deux tableaux $send\_counts$ et $displs$ paramètres de la fonction ci-dessous ainsi que la valeur du paramètre n_local pour chaque processeur

   ```
   MPI_Scatterv(A,send_counts,displs,MPI_INT,A_local, n_local, MPI_INT,root,MPI_COMM_WORLD)
   ```

2. Expliquez ce qu'est le modèle $Fork\_Join$ dans le contexte de la parallélisation avec OpenMP.

3. Soit un tableau de caractères (alphabet en minuscules) de taille n . On souhaite calculer le nombre d'apparition de chaque voyelle.

MPI : Dans le cadre du parallélisme de tâches, comment paralléliseriez vous cet algorithme pour une architecture à mémoire distribuée ?

OpenMP : En séquentiel si on suppose que le résultat est un tableau R de 6 éléments dont la case 0 est le nombre d'apparition de **a** et ainsi de suite il s'agit simplement d'une boucle sur le tableau pour incrémenter la case de R si le caractère est la voyelle associée à cette case. Quelle directive utiliseriez vous pour paralléliser cette boucle en OpenMP ? Utiliseriez vous des clauses spécifiques ? Justifiez.

Exercice 2. Encore un calcul sur un MNT (8pts)

Sur un terrain réel, l'eau qui tombe sur le sol s'écoule en grande partie vers le cours d'eau le plus proche. Pour calculer cet écoulement sur un MNT ce n'est pas facile car un terrain est composé d'une multitude de petites cuvettes dans lesquelles l'eau va se stocker. L'algorithme que vous avez utilisé lors du projet calculait justement l'ensemble de ses petites cuvettes et non pas les bassins versants. Il aurait fallu poursuivre les calculs pour finalement trouver les principaux bassins versants. Une autre possibilité est de conserver le même algorithme mais de modifier le MNT au préalable pour supprimer les petites cuvettes.

L'idée est d'appliquer un algorithme de remplissage des petites cuvettes. Cet algorithme consiste à travailler sur un MNT intermédiaire dans lequel on a tout rempli à partir de la hauteur max du terrain et qu'on va progressivement abaisser vers le MNT initial mais en gardant les cuvettes remplies.

Rappels

Le MNT se présente sous la forme d'un fichier texte constitué d'une entête suivie des valeurs de hauteur du MNT. L'entête contient les six lignes suivantes qui correspondent aux éléments explicités à droite

```
1025                                                    ncols
1025                                                    nrows
618360                                                  xllcorner
6754408                                                 yllcorner
1                                                       cellsize
-9999                                                   NODATA
95.7 98.2579 102.195 104.46 107.86 106.635 105.86
```

Ainsi la première ligne donne le nombre de colonnes du MNT, la seconde donne le nombre de lignes et ces deux nombres peuvent être différents. Les deux lignes suivantes indiquent les coordonnées du coin supérieur gauche mais cette information ne sera pas utilisée comme la taille du point indiquée par la cinquième ligne. Enfin la dernière ligne de l'entête donne la valeur associée à l'absence d'information sur un point du MNT. Les lignes suivantes contiennent les valeurs des hauteurs.

L'algorithme séquentiel

A partir du fichier texte il est donc possible de construire une matrice de réels Z de taille $nrows \times ncols$ qui contient toutes les hauteurs du terrain correspondant. Ainsi Z_{ij} est la hauteur du point (i, j) de la parcelle de terrain représenté. L'algorithme de remplissage est un algorithme itératif qui construit une matrice W initialisée par

1. $W_{ij} = NODATA$ si $Z_{ij} = NODATA$
2. $W_{ij} = Z_{ij}$ sinon et si $i = 0$ ou $j = 0$ ou $i = nrows - 1$ ou $j = ncols - 1$ (donc sur les bords)
3. $W_{ij} = Max$ sinon où Max désigne la plus haute hauteur du terrain

Les itérations peuvent se décrire par le pseudo code décrit par l'algorithme 1 où ϵ est une valeur représentant une petite pente pour l'écoulement de l'eau. Notons que pour effectuer ce calcul, il est nécessaire d'utiliser 3 matrices. La matrice Z du MNT initial et deux matrices W^{t-1} et W^t pour respectivement le remplissage calculé au temps précédent et pour le calcul courant. La matrice W^{t-1} est initialisée comme décrit ci-dessus et cette initialisation n'est pas exprimée dans l'algorithme 1.

Algorithm 1 Algorithme de Darboux pour le remplissage des cuvettes d'un MNT

```
1: modification=true
2: while modification do
3:     modification=false
4:     for all (i,j), 0 ≤ i ≤ nrows, 0 ≤ j ≤ ncols do
5:         if W_ij^{t-1} == NODATA then
6:             W_ij^t = NODATA
7:         else if W_ij^{t-1} > Z_ij then
8:             for all 8 neighbors ((i + k1), (j + k2)) = (n1, n2) 0 ≤ k1, k2 ≤ 1 do
9:                 if Z_{n1n2} ≥ W_{n1n2}^{t-1} + ε then
10:                    W_ij^t = Z_ij
11:                    modification=true
12:                else if W_ij^{t-1} > W_{n1n2}^{t-1} + ε then
13:                    W_ij^t = W_{n1n2}^{t-1} + ε
14:                    modification=true
15:                end if
16:            end for
17:        else
18:            W_ij^t = W_ij^{t-1}
19:        end if
20:    end for
21:    echange(W^t,W^{t-1})
22: end while
```

Les questions

A partir de la description de l'algorithme de remplissage d'un MNT répondez aux questions suivantes :

1. Décrivez la parallélisation de cet algorithme pour une architecture à mémoire distribuée. Vous pouvez faire des schémas et répondre également aux points suivants en explicitant les fonctions MPI que vous utiliseriez. Aucun code n'est demandé.
   (a) En considérant uniquement la matrice Z comment va-t-on distribuer les données ? Est ce que les distribuer par bloc a un intérêt ? Et si on suppose qu'on a $nprocs$ processeurs quelle sera la taille des données distribuées sur chaque processeur ? Comment prendre en compte les $NODATA$.
   (b) Si on suppose la partie distribution terminée, est ce que l'algorithme nécessite des phases de communication entre les itérations ? Si oui, comment seront-elles effectuées ?
   (c) L'algorithme s'arrête lorsque lors d'une itération aucune valeur de W^{t-1} n'a été modifiée ($W^t = W^{t-1}$). Comment gérer cette condition lorsque le calcul est réparti sur les $nprocs$ processeurs ?

2. Désormais le MNT est tellement grand qu'il est découpé en dalles et que désormais il est décrit par un ensemble de fichiers contenant les données de hauteur de chaque dalle. Comment généralisez vous votre algorithme pour qu'il s'applique à ces nouvelles données.

Exercice 3. Un jeu de Taquin par un algorithme de recherche par profondeur d'abord(8pts)

Un jeu de Taquin (Figure 1) peut se résoudre par un algorithme de **recherche par profondeur d'abord** ou **Depth-First Search (DFS)**.

En considérant un taquin 3×3 qui comprend tous les chiffres de 1 à 8 et une case libre le principe de l'algorithme est

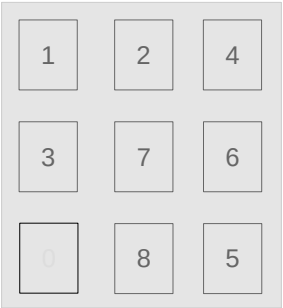

Figure 1: Jeu du taquin : Revenir à la grille ordonnée 1, 2, 3 etc par déplacements successifs d'une case grâce à la case libre.

de construire l'espace de recherche sous forme de graphe et de s'arrêter dès que la solution a été trouvée. On ne stocke pas l'espace de recherche en entier mais une pile permet de l'explorer.

La configuration du taquin, c'est-à-dire l'état de la grille, va être représentée par un `unsigned int` grâce à un codage qui permet de connaître

1. la position de la case libre
2. la valeur du chiffre de toutes les autres cases
3. si une grille est enfant ou parent dans l'espace de recherche.

Le codage n'est pas plus détaillé ici (il s'agit d'une représentation de toutes les informations en utilisant les 32 bits et de garder l'entier non signé correspondant). Pour l'exercice, il suffit de considérer qu'une grille est représentée par un `unsigned int`.

L'arbre de recherche, va consister à passer d'une configuration à l'autre du taquin, en déplaçant une case adjacente de la case libre, vers cette case libre. Donc un nœud a au plus 4 enfants.

D'un point de vue informatique on va stocker les nœuds du graphe en utilisant le codage de la grille en `unsigned int`. La figure 2 illustre l'évolution d'un graphe et la pile correspondante. Le bit de poids fort du codage de la grille indique si une grille est *parent* ou pas. L'ensemble des grilles *parent* correspond à la branche en cours d'exploration.

Autrement dit les étapes de l'algorithme sont les suivantes. On initialise une `pile LIFO` avec la grille initiale dont on cherche la solution et on indique que cette grille est un parent.

1. Génération des enfants de la grille `parent` en tête de la pile LIFO. Les enfants sont générés en remplissant la case libre par un déplacement de l'élément de gauche, de droite, au-dessus ou en-dessous. Empiler les enfants uniquement s'ils ne sont pas déjà dans le graphe ou s'ils n'ont pas déjà été présents dans le graphe ou s'ils ne sont pas une solution.
2. Si à partir du parent aucun nouvel enfant n'a été généré, dépiler la pile LIFO tant que la tête est un parent.
3. Si la tête est un enfant, choisir parmi les enfants qui précédent, celui qui va être exploré et qui va devenir `parent`.

Lorsqu'en générant les enfants, on trouve l'enfant solution l'algorithme s'arrête et la solution est constituée de tous les éléments `parent` dans la pile représentant le graphe de recherche.

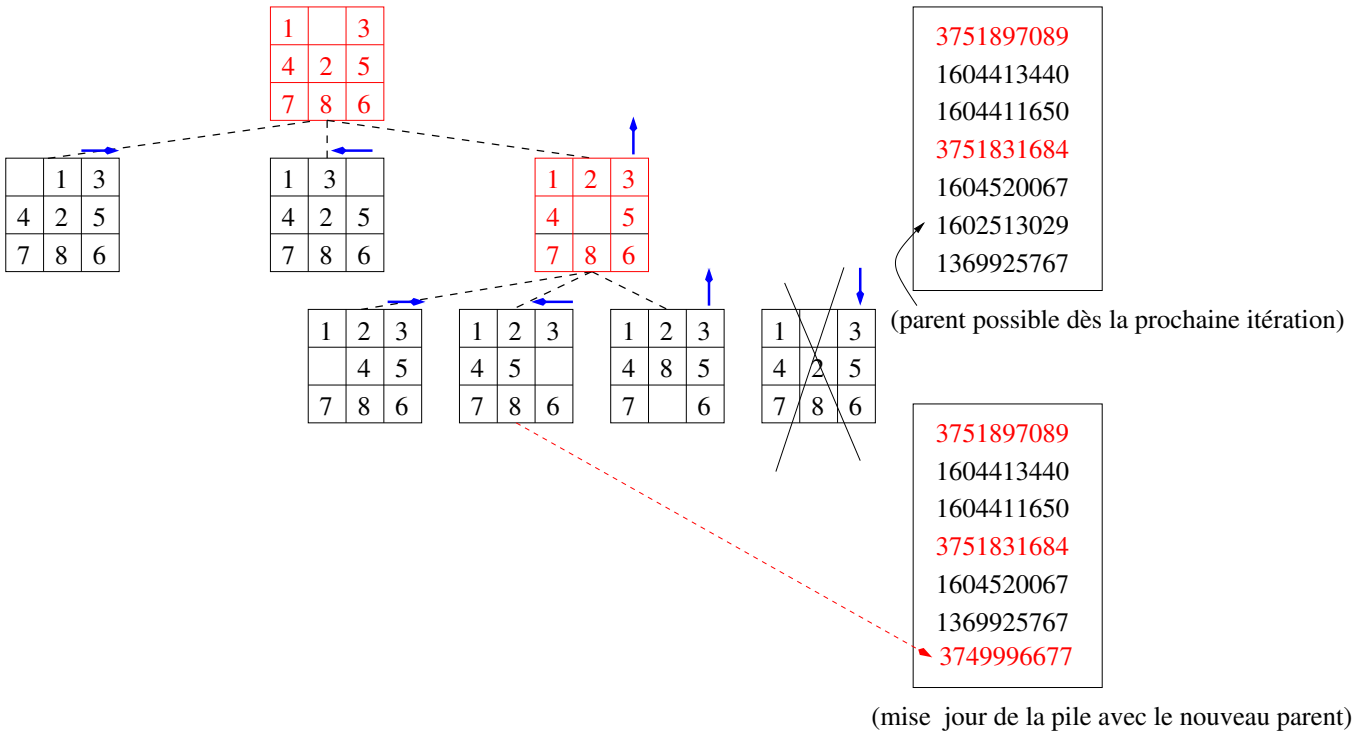

Figure 2: A gauche, exemple d'évolution du graphe où la dernière grille sera oubliée car déjà présente dans le graphe. La flèche indique le déplacement effectué pour passer de la configuration parent à son enfant. A droite, la pile LIFO contenant les `unsigned int` associés aux grilles du graphe, est représentée tête en bas. En rouge la branche en cours d'exploration.

Les questions

Vous disposez des fonctions suivantes qui permettent de manipuler les grilles à partir de leur codage en `unsigned int`.

```
unsigned int * generation(); // Pour générer la grille initiale
bool est_final(unsigned int *g); // test si la grille g est la solution
unsigned int* move_up(unsigned int *g); // génère une nouvelle grille par application
                          //d'un déplacement vers le haut sur la grille courante
                          //g=NULL si le déplacement n'est pas possible
 unsigned int* g move_down(unsigned int *g); //idem pour un déplacement vers le bas
 unsigned int* g move_left(unsigned int *g); // idem pour un déplacement vers la gauche
 unsigned int* g move_right(unsigned int *g); // idem pour un déplacement vers la droite
 bool est_parent(unsigned int *g); // test si la grille g est parent
 void devient_parent(unsigned int *g); // modifie le bit de poids fort pour que la grille
                                       // soit indiquée comme parent
```

N'implémentez pas les fonctions ci-dessus mais si nécessaire vous pouvez compléter ces déclarations par des fonctions supplémentaires sans les implémenter mais en indiquant clairement leur rôle.

Pour une architecture à mémoire distribuée

Décrivez la parallélisation possible de cet algorithme pour une architecture à mémoire distribuée. Vous pouvez illustrer vos explications par des schémas ou des exemples et également décrire votre parallélisation avec du pseudo code. Les questions suivantes peuvent vous aider :

1. Quels sont les calculs effectués par chaque processeur ?
2. Comment et quand peut-on partager l'exploration de l'espace de recherche ?
3. Comment ne pas rajouter à la pile, des enfants déjà traités en local ou par un autre processus ?

4. Comment obtenir un équilibrage de charges ?
5. Quelles seront les communications nécessaires ? Et à quel moment ?
6. Comment gérer la fin de l'exécution quand une solution a été trouvée ?

Pour représenter les piles vous pouvez utiliser `std::vector<unsigned int*>` et considérer qu'elles pourront être transmises via les fonctions de communication MPI.

Pour une architecture à mémoire partagée

Dans le contexte d'une exécution sur une architecture à mémoire partagée, l'accès aux données est simplifié mais l'exploration de l'espace de recherche doit être réparti sur les différents threads.

Décrivez votre parallélisation pour une telle architecture. Vous pouvez expliciter les directives que vous utiliseriez en justifiant leur rôle dans votre parallélisation.