UNIVERSITÉ D'ORLÉANS **M1 Info**
Informatique **2016 - 2017**

Session 1: Programmation Parallèle - 2h

Exercice 1. Le jeu de Taquin (5pts)

Cet exercice fait suite au projet du 16 décembre.

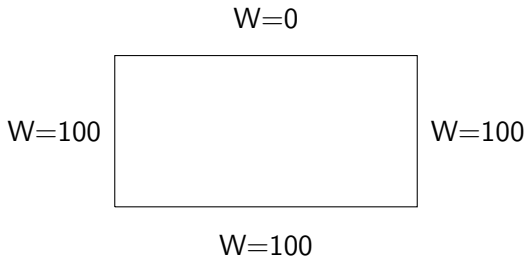
1. Quels sont les paramètres qui peuvent influencer la performance de la parallélisation ? Justifiez le choix que vous feriez pour ces paramètres.
2. La version proposée pour le projet était une version simplifiée. Pour les architectures distribuées (donc la version MPI) quel algorithme plus performant proposeriez vous ?
3. Même question pour une implémentation sur une architecture à mémoire partagée (donc la version OpenMP) ?

Exercice 2. Équation de la chaleur en régime permanent dans un rectangle 2D (9pts)

Dans cet exercice, il vous est demandé de paralléliser un code séquentiel existant résolvant l'équation (simplifiée) de la chaleur en régime permanent dans une région rectangulaire.

On considère une région physique rectangulaire ou les frontières (les bords) ont une température (`W`) constante. Pour simplifier le calcul, on considère qu'à chaque instant, la valeur à un point du rectangle correspond à la moyenne de ses 4 voisins : `W[Central] <= (1/4) * ( W[North] + W[South] + W[East] + W[West])`.

On cherche à évaluer l'état de la région lorsque la température est stabilisée, c'est-à-dire lorsque la plus grande modification d'un point, entre deux états, est inférieure à une borne (`Epsilon`) passée en paramètre. Les conditions initiales sont représentées sur la figure suivante :



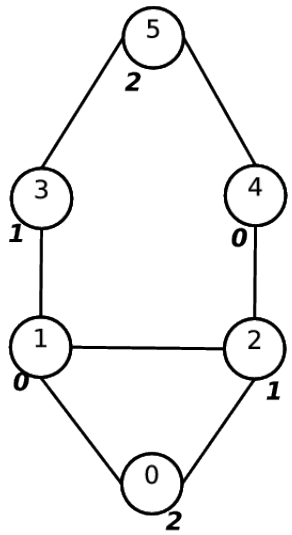
On ne vous demande pas d'écrire le programme résolvant l'équation de la chaleur. La version séquentielle de ce programme est fournie (voir celene). Il vous est demandé, avec OpenMP, de paralléliser ce programme **en modifiant le moins possible** le code existant. Reportez sur votre feuille les modifications réalisées dans le programme **(en indiquant le numéro de ligne où vous réalisez la modification)**. Les modifications sont à faire entre les lignes 99 et 199 du programme. Un `makefile` vous est fourni pour compiler/tester le programme.

- `make` compile les versions séquentielles et OpenMP du programme
- `make test` compile et exécute la version openmp du programme avec des paramètres par défaut (modifiables dans le makefile)
- `make check` vérifie que le résultat parallèle correspond au résultat séquentiel (avec des paramètres modifiables dans le makefile). Attention, il est possible que des erreurs liées à l'ordre des opérations flottantes qui change en OpenMP affichent un résultat faux alors que le calcul est correct (regardez si les valeurs sont vraiment différentes ou non).

N'oubliez pas que seul le rendu papier sera pris en compte pour l'évaluation! Ne passez donc pas tout votre temps sur la machine.

Exercice 3. Coloriage de Graphe, Algorithme de Brélaz (6pts)

L'objectif de cet exercice est de proposer une parallélisation pour les architectures à mémoire distribuée de l'algorithme de Brélaz qui permet de colorier un graphe. Pour cela, un graphe sera représenté par une matrice d'adjacence qui sera distribuée à chaque processeur impliqué dans le calcul.



$$\begin{matrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{matrix}$$

Comme illustré par l'exemple précédent la matrice d'adjacence M est définie par $M_{ij} = 1$ s'il existe une arête entre le nœud i et le nœud j et $M_{ij} = 0$ sinon. On ne considérera que les graphes non dirigés et connexes. Pour le problème de la coloration le coût de l'arête n'a pas d'importance donc on le considérera toujours égal à 1.

Le problème de la coloration consiste à affecter une couleur à chaque nœud du graphe de telle manière que deux nœuds adjacents soient coloriés par des couleurs différentes. Si on suppose qu'on numérote les couleurs de 0 à $n_c - 1$ où n_c est le nombre de couleurs nécessaires, les numéros en bold-italic sur le graphe précédent illustrent une coloration du graphe.

L'algorithme de Brélaz est un algorithme itératif. Dans une phase d'initialisation, on choisit le nœud de plus haut degré où le degré est le nombre d'arêtes connectées au nœud. On affecte la couleur 0 à ce premier nœud (en cas d'égalité on choisit au hasard parmi les nœuds de plus haut degré). Ensuite les étapes sont

1. calculer pour chaque nœud non colorié le nombre n_{vc} de voisins coloriés,
2. choisir le nœud n_{ac} ayant le plus grand n_{vc} et en cas d'égalité choisir le nœud ayant le plus fort degré,
3. affecter au nœud n_{ac} la plus petite couleur disponible et non utilisée par ses voisins.

Pour paralléliser cet algorithme on va distribuer les lignes de la matrice d'adjacence afin de répartir le calcul de n_{vc} à chaque itération de l'algorithme. Il est à noter qu'en distribuant par ligne la matrice d'adjacence, chaque processeur gère une partie des nœuds du graphe mais que pour chaque nœud il connaît les arêtes auxquelles ce nœud est connecté.

Vous disposez sous Celene d'un package contenant un programme principal et des fonctions utiles déjà implémentées pour vous aider. Vous devez écrire la fonction de signature

```
void Brelaz(int n, int taille, int* graphe_local, int* noeuds_colores);
```

où

1. `taille` est le nombre de nœuds du graphe complet
2. `n` est le nombre de nœuds en local sur un processeur.
3. `graphe_local` est la matrice d'adjacence en local sur un processeur. Elle est de taille $n \times taille$.
4. `noeuds_colores` est le tableau de longueur `taille` construit par cette fonction et qui contient au final le numéro de la couleur affectée à chaque nœud du graphe.

Reportez le code écrit sur votre copie. Attention, le package et le travail sur machine sont destinés à vous aider. Mais ne passez pas trop de temps à débugger et reportez bien votre fonction sur votre copie !!

Enfin parmi les fonctions proposées, vous trouverez des fonctions pour générer les graphes en .dot. De cette manière le programme se lance par

```
mpirun -np a ./main b c d e
```

où

- `a` est le nombre de processeurs utilisés
- `b` est le paramètre `taille`
- `c` est le processeur root
- `d` est le nom du fichier (par exemple graphe.dot) contenant le graphe initial
- `e` est le nom du fichier (par exemple graphe_res.dot) contenant le graphe coloré

Pour visualiser les graphes vous pourrez les convertir en .pdf (convert graphe.dot graphe.pdf).