UNIVERSITÉ D'ORLÉANS **M1 Info**
Informatique **2012 - 2013**

Session 1: Programmation Parallèle - 2h

Exercice 1. PRAM : Fréquence d'apparition (3pts)

Donnez un algorithme PRAM pour une machine *CREW* qui permet de calculer la fréquence d'apparition des éléments d'un tableau T de taille n connue lorsque ces éléments sont compris entre 0 et une valeur $\max$ également supposée connue. Par exemple soit $T = \{0, 5, 7, 7, 5, 0, 7, 2\}$ et $\max = 10$ alors votre algorithme aura pour résultat un tableau F de taille 10 tel que $F = \{0.25, 0, 0.125, 0, 0, 0.25, 0, 0.375, 0, 0, 0\}$.

Exercice 2. MPI : Fréquence d'apparition (4pts)

Soit T un tableau distribué sur les processeurs d'une architecture à mémoire distribuée, écrivez la fonction suivante utilisant la librairie MPI pour calculer la fréquence d'apparition de chaque élément du tableau :

`void Frequence(int* T, int n, int max, float* F)`.

Par exemple si T est le tableau distribué de la manière suivante

pid	0	1	2	3
T distribué	$T = \{0, 0, 4, 10\}$	$T = \{1, 2, 3, 4\}$	$T = \{3, 3, 5, 10\}$	$T = \{4, 10, 9, 7\}$

alors à la sortie de l'appel `Frequence(T,16,10,F)` chaque processeur a en local le tableau de taille 10 tel que $F = \{\frac{2}{16}, \frac{1}{16}, \frac{1}{16}, \frac{3}{16}, \frac{3}{16}, \frac{1}{16}, 0, \frac{1}{16}, 0, \frac{1}{16}, \frac{3}{16}\}$.

Exercice 3. MPI : Suites bitoniques (8pts)

Une suite de nombres $\{a_0, a_1, \ldots, a_{n-1}\}$ est dite bitonique si elle vérifie la propriété (1)

$$\exists k \; tel \; que \; a_0 \leq a_1 \leq \ldots \leq a_k \geq a_{k+1} \geq \ldots \geq a_{n-1}$$

à une permutation cyclique près.

Par exemple $\{1, 3, 7, 8, 4, 2, 1\}$ est une suite bitonique avec $k = 3$. De même $\{4, 5, 6, 8, 10, 7, 1, 2, 3\}$ est également une suite bitonique car $\{2, 3, 4, 5, 6, 8, 10, 7, 1\}$ avec $k = 6$ respecte la propriété (1).

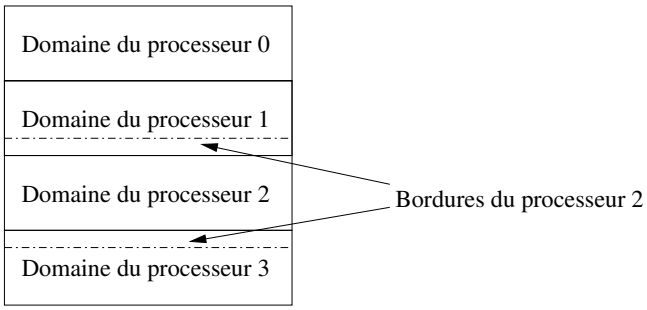
Ecrivez le programme MPI qui permet de déterminer si une suite de nombres est bitonique ou pas. Vous devrez compléter le programme ci-dessous et écrire les fonctions dont vous avez besoin. Vous êtes libre de définir la méthode choisie pour tester si la suite est bitonique ou pas mais seul le processeur de pid 0 aura le résultat final.

```
int main ( int argc , char **argv ) {
  int pid, nprocs;
  MPI_Init (&argc , &argv) ;
  MPI_Comm_rank(MPI_COMM_WORLD, &pid ) ;
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs ) ;

  int taille = atoi(argv[1]);
  int n = taille*nprocs;
  int suite[taille];
  srand(time(NULL));
  for (int i=0; i<taille; i++)
    suite[i] = rand()%100;

  // le tableau suite réparti sur les processeurs est-il bitonique ou non ?
  // Appel des fonctions à définir et à écrire

  MPI_Finalize() ;
  return 0 ;}
```

Exercice 4. Communications via RMA (5pts)

L'objectif de cet exercice est d'implémenter des fonctions de communication RMA pour un calcul itératif parallèle classique sur une matrice avec recouvrement.

La matrice est découpée en sous-matrices distribuées sur chacun des processeurs. Chaque processeur effectue un calcul sur la sous-matrice qui lui a été attribuée. Afin d'obtenir un calcul équivalent au calcul séquentiel, chaque processeur utilisera une bordure (classiquement appelée *ghost*) qui contiendra les valeurs calculées par ses voisins du haut et du bas (voir schéma).



Dans cet exercice on va considérer une matrice carrée de taille $N \times N$ et on utilisera toujours un nombre de processeurs m tel que m est un diviseur de N. On considéra une grille torique. Sur l'exemple du schéma, cela signifie que la bordure du haut du processeur 0 se trouve dans le domaine du processeur 3 et la bordure du bas du processeur 3 se trouve dans le domaine du processeur 0.

Par conséquent, chaque processeur possèdera une matrice locale de taille $N \times (\frac{N}{m} + 2)$ dont la ligne 0 et $\frac{N}{m} + 1$ contiendront les bordures des voisins du haut et du bas respectivement. De plus, le calcul itératif doit s'arrêter quand l'écart maximum de la valeur d'une même cellule de la matrice complète entre deux itérations ne dépasse pas un certain seuil τ. On considère que la fonction de calcul local nous fournit en retour cette valeur du plus grand écart entre deux itérations du calcul pour la sous matrice.

1. Ecrire une fonction
   `void init_bordures(int localMat[], int N, int nbProc, int rang, MPI_Win *b1,MPI_Win *b2)`
   qui va créer les fenêtres MPI nécessaires à la gestion des bordures. `localMat` est la matrice qui contient à la fois le domaine du processeur + les bordures hautes et basses, `N` est le coté de la matrice globale, `nbProc` est le nombre de processeurs participant au calcul et `rang` est le rang du processeur courant. Enfin `b1` et `b2` sont les deux fenêtres de gestion des bordures (à vous de définir leurs rôles respectifs).

2. Ecrire une fonction `update_bordures` qui a les mêmes paramètres que la précédente et qui permet à un processeur `rang` d'écrire les valeurs qu'il a calculées dans la bordure de ses deux voisins.

3. Ecrire une fonction
   `void init_ecart(int localEcart, int nbProc, int rang, MPI_Win *ecart )`
   où `localEcart` est l'écart calculé en local lors d'une itération et `ecart` la fenêtre qui va gérer les écarts. Les autres paramètres ont la même signification que `init_bordures`.

4. Ecrire une fonction `update_ecart` qui a les mêmes paramètres que la précédente et qui permet au processeur `rang` de mettre à jour l'écart des autres processeurs pour qu'ils prennent son écart local.