INF7235 — Programmation parallèle haute performance
Examen final (Hiver 2010)

Durée: (3 heures) **Documentation autorisée:** Toute documentation personnelle.

Remarques :

- Lorsqu'il est demandé de **justifier** vos réponses, tant la clarté, la justesse que la pertinence des explications seront évaluées — donc si vous écrivez des choses fausses, qui n'ont pas rapport, vous serez pénalisés.
- Vous devez **obligatoirement** répondre aux questions 1, 2 et 3.
- Vous devez ensuite répondre **à l'une des questions 4 ou 5**.

  Si vous répondez à ces deux dernières questions, les points vous seront comptés **en bonus** — donc la note maximale possible est de 40 / 35!

---

1. Calcul des écarts à la moyenne (10 pts)

On veut écrire un programme MPI/C composé d'un groupe de processus où chaque processus possède une valeur entière qui lui est spécifique et où on calcule puis imprime l'écart entre chacune des valeurs et la *moyenne* de ces différentes valeurs, l'impression étant réalisée uniquement par le processus 0.

Par exemple, supposons cinq processus où la valeur conservée par chacun est simplement son numéro de processus. L'exécution produirait alors le résultat suivant (la moyenne est $2 = \lfloor \frac{0+1+2+3+4}{5} \rfloor$), imprimé par le processus 0 :

```
ecarts[0] =  -2
ecarts[1] =  -1
ecarts[2] =   0
ecarts[3] =   1
ecarts[4] =   2
```

Complétez le programme MPI 1 (page suivante) pour réaliser cette tâche, et ce de deux façons différentes :

a. En utilisant des opérations de communication *point à point* — i.e., uniquement avec `MPI_Send` et `MPI_Recv`.

b. En utilisant des opérations de communication *collectives* — donc *sans utiliser* `MPI_Send` ou `MPI_Recv`.

Notes :

- Pour la moyenne, utilisez simplement la division entière.
- Donnez uniquement la «partie à compléter» du code, donc inutile de répéter le reste du programme (prélude et impression).

```
int main( int argc, char *argv[] )
{
  int numProc;                // Numero du processeur.
  int nbProcs;                // Nombre de processeurs.

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &numProc );
  MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );

  int val = ...;              // Valeur specifique au processus.

  int *ecarts = (int *) malloc( nbProcs * sizeof(int) );

  ////////////////////////////////
  // Partie a completer...
  ////////////////////////////////









  ////////////////////////////////


  // On imprime les resultats.
  if ( numProc == 0 ) {
    for( int i = 0; i < nbProcs; i++ ) {
      printf( "\tecarts[%d] = %4d\n", i, ecarts[i] );
    }
  }

  MPI_Finalize();
  return( 0 );
}
```

Programme MPI 1: Programme MPI à compléter pour le calcul des écarts à la moyenne.

2. Distribution de la température dans une salle (10 pts)

Machine parallèle et langage

On dispose d'une machine parallèle de type *cluster* («grille de processeurs») avec mémoire distribuée, disposant d'un nombre limité de processeurs — donc une machine semblable à `trex_cluster`. On désire programmer cette machine en MPI/C.

Le problème

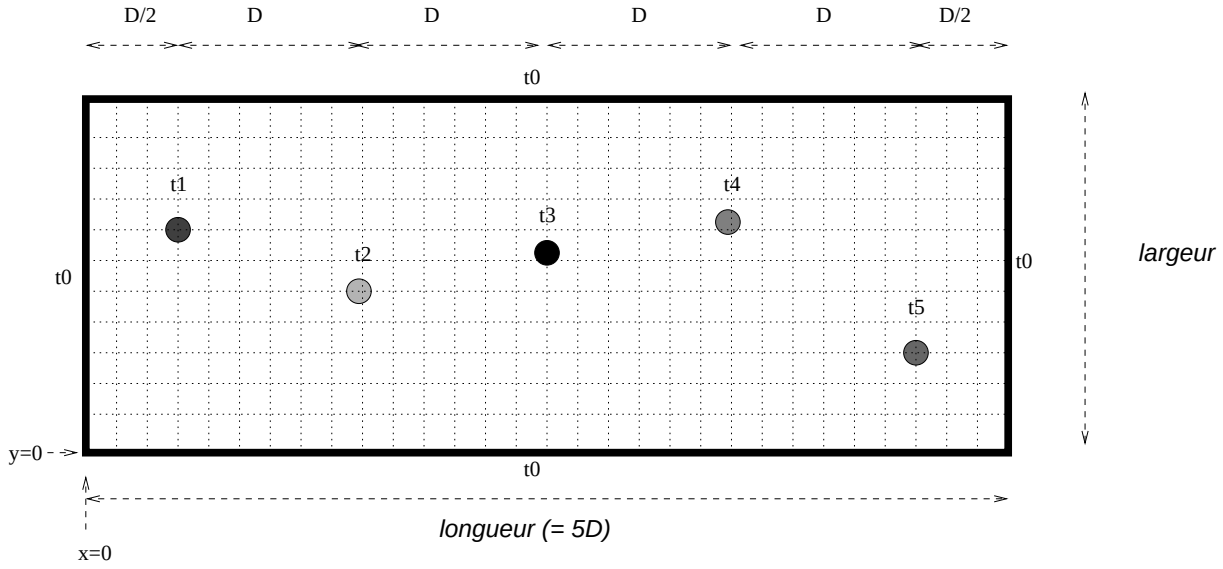

Figure 1: Salle rectangulaire avec sources de chaleur.

La figure 1 présente la configuration générale d'une salle rectangulaire ($longueur \times largeur$) avec des sources *ponctuelles* de chaleur — les petits cercles, de dimension négligable (point) — disposées à différents endroits dans la salle. Les lignes pointillées à l'intérieur du rectangle servent uniquement à illustrer la «discrétisation de l'espace» et ne font pas partie de la configuration de la salle (voir plus bas).

Les caractéristiques de la configuration d'une telle salle sont les suivantes :

- La salle est de forme rectangulaire et mesure $longueur \times largeur$.
- La salle possède n sources de chaleur — ici, $n = 5$. Ces sources ont des températures possiblement différentes les unes des autres ($t1, t2, \ldots$, illustrées aussi par la «couleur» des différentes sources). Il s'agit de sources *ponctuelles*, donc sans «dimension».
- En termes de position *horizontale*, les sources de chaleur sont réparties uniformément dans la salle. Si l'on note par $x = 0$ le coté gauche de la salle, alors les sources de chaleurs sont positionnées aux coordonnées $x = \frac{1}{2}D, \frac{3}{2}D, \frac{5}{2}D, \frac{7}{2}D, etc.$, où D est la distance entre les positions horizontales des sources de chaleur — et, donc, la longueur de la salle est nD.
- Les positions *verticales* peuvent varier d'une source à l'autre.
- La température sur la frontière de la salle est fixe.

On veut écrire un programme parallèle pour une machine à mémoire distribuée, en utilisant MPI/C, qui permet de simuler (approximer) la diffusion et la distribution de la chaleur pour une

telle salle avec sources de chaleur. Pour ce faire, on va utiliser une approche semblable à celle vue en cours pour la diffusion de la chaleur dans un cylindre, c'est-à-dire approximation de la solution exacte par une discrétisation de l'espace (ici, une grille de points) et du temps (itération de la simulation). Toutefois, à la différence de l'exemple vu en cours, il s'agit ici d'une simulation sur un espace à deux (2) dimensions (rectangle) plutôt qu'à une seule dimension (cylindre $\approx$ ligne). Dans un tel cas, l'équation à utiliser est celle dite *de Jacobi*, définie comme suit, qui donne la température au point i, j de la grille au temps $k + 1$ en fonction de la température des voisins immédiates au temps k :

$$T^{k+1}[i,j] = \frac{T^k[i-1,j] + T^k[i+1,j] + T^k[i,j-1] + T^k[i,j+1]}{4}$$

En d'autres mots, la valeur pour un point $[i, j]$ de la grille (espace 2D) est définie par la moyenne de ses *quatres voisins immédiats* (nord, sud, est, ouest, ou encore gauche, droit, haut, bas) — contrairement à la simulation du cylindre où on utilisait la moyenne des deux voisins immédiats (espace 1D).

Le programme reçoit les arguments suivants sur la ligne de commande :

1. La longueur de la salle ;
2. La largeur de la salle ;
3. La température sur la frontière de la salle (`t0`).
4. Le nombre de sources chaleur (par exemple, 5) ;
5. Pour chaque source de chaleur, sa position verticale ($0 < y < largeur$) et sa température (les `t1`, ..., `t5` dans l'exemple).
6. La distance entre les points de simulation (discrétisation de l'espace), tant au niveau vertical qu'horizontal, i.e., la distance entre les lignes pointillées sur la figure — pour simplifier, on suppose que la longueur et la largeur sont des multiples exacts de cette distance ;
7. La durée de la simulation, i.e., le nombre d'itérations durant lesquelles on veut simuler l'évolution de la température.

Ce que vous devez faire

Décrivez et comparez (avantages et désavantages, forces et faiblesses) diverses façons de concevoir un programme MPI pour résoudre ce problème. *Vous n'avez pas à écrire de code MPI.* Vous devez uniquement décrire les grandes lignes des approches proposées, en termes de décomposition et agglomération en tâches, de communications entre les tâches, etc. Vous devez aussi conclure en indiquant *l'approche que vous suggéreriez d'utiliser, de façon à bien exploiter les ressources de la machine.*

3. Transformation d'images (10 pts)

Machine parallèle et langages

On dispose d'une machine parallèle de type «grille de multiprocesseurs» avec mémoire distribuée et système de fichiers en réseau. Cette machine possède donc les caractéristiques suivantes :

- La machine compte 32 noeuds, connectés par un réseau haute vitesse qui permet à plusieurs processeurs, mais pas tous, de communiquer entre eux en même temps.
- Chaque noeud est un petit multiprocesseur, avec quatre (4) processeurs qui se partagent une mémoire locale par l'intermédiaire d'un bus.
- Le système de fichiers est en réseau, donc tous les noeuds ont accès aux mêmes fichiers et mêmes structures de répertoires. (Ce qui n'était pas le cas sur `trex_cluster`!)

Cette machine comporte des compilateurs pour MPI/C et OpenMP/C. En fait, ces compilateurs permettent d'écrire du code qui, comme cela se fait de plus en plus, combinent MPI et OpenMP *dans un même programme* : MPI pour les communications entre les noeuds, OpenMP pour le traitement parallèle sur les processeurs (ou coeurs) locaux à un noeud.

Le problème

On désire écrire un programme pour faire du traitement d'images. Chaque image est conservée dans un fichier et est représentée sous forme de *pixmap* :[1]

> *In computer graphics, a bitmap or pixmap is a type of memory organization or image file format used to store digital images. The term bitmap comes from the computer programming terminology, meaning just a map of bits, a spatially mapped array of bits. Now, along with pixmap, it commonly refers to the similar concept of a spatially mapped array of pixels. Raster images in general may be referred to as bitmaps or pixmaps, whether synthetic or photographic, in files or memory.*
>
> *In some contexts, the term bitmap implies one bit per pixel, while pixmap is used for images with multiple bits per pixel.*

La procédure de traitement d'une image consiste simplement à effectuer les opérations suivantes :

1. Charger le fichier en mémoire dans une matrice à deux dimensions d'entiers (type `short int`) ;
2. Appliquer une fonction de transformation à chacun des entiers de la matrice pour produire une matrice résultante — la fonction est simple et directe, en ce sens que la valeur produite pour un point donné ne dépend que de ce point, pas des voisins ;
3. Écrire dans un fichier de sortie le contenu de la matrice résultante.

Les images sont de tailles variables — certaines images sont très petites, d'autres très grandes.

Le seul argumet fourni à l'appel du programme est le nom d'un répertoire contenant les divers fichiers à traiter.

Ce que vous devez faire

Décrivez et comparez (avantages et désavantages, forces et faiblesses) diverses façons de concevoir et écrire un programme pour résoudre ce problème. *Vous n'avez pas à écrire de code.* Vous devez uniquement décrire les grandes lignes des approches proposées, y compris le langage suggéré pour les diverses parties du programme. Vous devez aussi conclure en indiquant *l'approche que vous suggéreriez d'utiliser, de façon à bien exploiter les ressources de la machine.*

[1] `http://en.wikipedia.org/wiki/Bitmap`

4. Parallélisation OpenMP d'un problème de programmation dynamique (5 pts)

La «programmation dynamique» peut être définie comme suit :[2]

> La programmation dynamique est une technique algorithmique qui permet de résoudre une catégorie particulière de problèmes d'optimisation sous contrainte. [...] Elle s'applique à des problèmes d'optimisation dont la fonction objectif se décrit comme «la somme de fonctions monotones non-décroissantes des ressources».
>
> La programmation dynamique s'appuie sur une relation entre la solution optimale du problème et celles d'un nombre fini de sous-problèmes. Concrètement, cela signifie que l'on va pouvoir déduire la solution optimale d'un problème à partir d'une solution optimale d'un sous problème. Généralement, cette relation est utilisée pour évaluer les solutions des problèmes «de bas en haut», c'est-à-dire qu'on calcule les solutions des sous-problèmes les plus petits pour ensuite déduire petit à petit les solutions de tous les sous-problèmes.

Plus concrètement, les problèmes de programmation dynamique sont typiquement résolus en remplissant une matrice de valeurs, où chaque entrée de la matrice représente une solution à un sous-problème et où la solution globale est obtenue en appliquant une opération appropriée sur la matrice résultante.

Par exemple, soit un problème de programmation dynamique dont la matrice M (de taille $n \times n$) est définie comme suit, où V_1 et V_2 sont des constantes, et f est une fonction auxiliaire *pure* (fonction mathématique pure, i.e., sans effet de bord et sans accès à des variables globales) :

$$
\begin{array}{ll}
M[i,0] = V_1, & \text{pour } i = 0, \ldots, n-1 \\
M[0,j] = V_2, & \text{pour } j = 1, \ldots, n-1 \\
M[i,j] = f(\ M[i-1,j-1], M[i-1,j], M[i,j-1]\ ), & \text{pour } i,j = 1, \ldots, n-1
\end{array}
$$

Les valeurs sur la première ligne et la première colonne de M représentent les valeurs associées aux solutions des sous-problèmes les plus simples (cas de base définis par des constantes). Plus on s'éloigne vers la droite et vers le bas, plus la valeur de la matrice représente la solution à un sous-problème complexe. On remarque que la valeur pour une solution $M[i,j]$ ne dépend que des valeurs pour des sous-problèmes plus simples. Ainsi, si on considère la position $[0,0]$ comme étant située dans le coin gauche supérieur, alors $M[i,j]$ dépend des positions adjacentes suivantes : juste en haut : $M[i-1,j]$, immédiatement à gauche : $M[i,j-1]$, et en diagonale en haut à gauche : $M[i-1,j-1]$.

Les extraits de programmes C 1 et 2 (page suivante) présentent deux versions d'une mise en oeuvre en C pour ce problème de programmation dynamique.

a. Parallélisez ces extraits de programme C en ajoutant, si approprié, des directives OpenMP. Assurez-vous évidemment que le résultat produit sera correct si le programme est exécuté de façon parallèle. Indiquez aussi, de façon explicite, le type de `schedule` qu'il serait préférable d'utiliser.

b. Si vous n'avez pas pu paralléliser certaines parties, *expliquez brièvement pourquoi.*

Note : Vous pouvez détacher la feuille et répondre directement sur celle-ci.

[2] `http://fr.wikipedia.org/wiki/Programmation_dynamique`

```
for( int i = 0; i < n; i++ ) {
  M[i][0] = V1;
}


for( int j = 1; j < n; j++ ) {
  M[0][j] = V2;
}


for( int i = 1; i < n; i++ ) {

  for( int j = 1; j < n; j++ ) {
    M[i][j] = f( M[i-1][j-1], M[i-1][j], M[i][j-1] );
  }

}
```

Extrait de programme C 1: Première version d'un algorithme séquentiel pour un problème de programmation dynamique.

```
for( int i = 0; i < n; i++ ) {
  M[i][0] = V1;
}


for( int j = 1; j < n; j++ ) {
  M[0][j] = V2;
}


for ( int diag = 2; diag <= (n-1)+(n-1); diag++ ) {
  int kinf = MAX(1, diag-(n-1));
  int ksup = MIN(diag, n);

  for( int k = kinf; k < ksup; k++ ) {
    M[diag-k][k] = f( M[diag-k-1][k-1], M[diag-k-1][k], M[diag-k][k-1] );
  }

}
```

Extrait de programme C 2: Deuxième version d'un algorithme séquentiel pour un problème de programmation dynamique.

5. Parallélisation OpenMP d'une recherche de palindromes (5 pts)

L'extrait de programme C 3 présente une mise en oeuvre séquentielle pour une routine `trouverPalindromes` qui reçoit en argument un tableau de chaînes de caractères et la taille de ce tableau, puis qui retourne en résultat les chaînes qui sont des *palindromes*, ainsi que le nombre de palindromes trouvés. Une routine auxiliaire `estPalindrome` est utilisée pour déterminer si une chaîne `s` est bien un palindrome — une chaîne qui est son propre inverse, donc qui se lit de la même façon de gauche à droite ou de droite à gauche, par exemple, `"kayak"`, `"cc"`, etc.

Parallélisez ces routines C en ajoutant, si approprié, des directives OpenMP. Assurez-vous évidemment que le résultat produit sera correct une fois le programme exécuté de façon parallèle, et ce peu importe le nombre de processeurs. Indiquez aussi, de façon explicite, le type de `schedule` qu'il serait préférable d'utiliser.

Note : Vous pouvez détacher la feuille et répondre directement sur celle-ci.

```
int estPalindrome( char* s )
{
  int n = strlen(s);
  int res = 1;


  for( int i = 0; i < n/2; i++ ) {

    res = res &&  (s[i] == s[n-i-1]);

  }

  return( res );
}


void trouverPalindromes( char* M[], int n, char* res[], int *nb )
{
  int k = 0;


  for( int i = 0; i < n; i++ ) {

    if ( estPalindrome(M[i]) )

       { res[k] = M[i]; k += 1; }

  }

  *nb = k;
}
```

Extrait de programme C 3: Routines pour la recherche des palindromes dans un tableau de chaînes.