INF7235 — Programmation parallèle haute performance
Examen final (Hiver 2008)

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

1. Calcul de l'ensemble de Mandelbrot (20 pts)

L'ensemble de Mandelbrot est un exemple bien connu de fractal. En voici une définition [1] :

> *The Mandelbrot set is a set of points in the complex plane, the boundary of which forms a fractal. Mathematically, the Mandelbrot set can be defined as the set of complex c-values for which the orbit of 0 under iteration of the complex quadratic polynomial* $x_{n+1} = x_n^2 + c$ *remains bounded.*
>
> *Eg.* $c = 1$ *gives the sequence* $0, 1, 2, 5, 26, \ldots$ *which leads to infinity. As this sequence is unbounded, 1 is not an element of the Mandelbrot set.*
>
> *On the other hand,* $c = i$ *gives the sequence* $0, i, i-1, -i, i-1, -i, \ldots$ *which is bounded, and so it belongs to the Mandelbrot set.*[2]
>
> *When computed and graphed on the complex plane, the Mandelbrot Set is seen to have an elaborate boundary, which does not simplify at any given magnification. This qualifies the boundary as a fractal.*

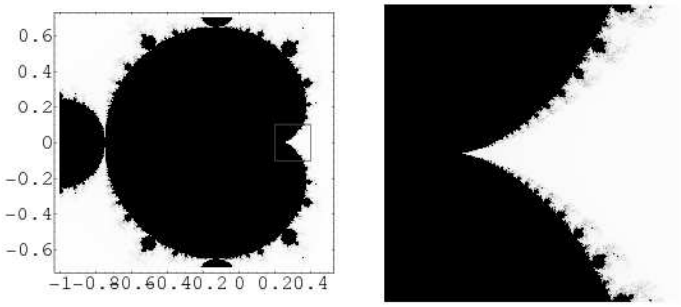

Figure 1: Une représentation graphique de l'ensemble de Mandelbrot.

La figure 1 présente une version graphique de l'ensemble de Mandelbrot. Les différents points du plan qui sont en noir représentent des points qui font partie de l'ensemble de Mandelbrot — la figure de droite, qui est un agrandissement de la petite partie encadrée de la figure de gauche, montre une propriété typique des fractales, à savoir, *l'auto-similarité*.

Un point (x, y) fait partie de l'ensemble lorsqu'on est (presque) assuré que *l'application itérée et répétitive* d'une certaine fonction fait en sorte que l'on reste toujours à l'intérieur d'un cercle de rayon 2 (centré à l'origine). Mathématiquement, étant donné un point de départ (x, y), la fonction itérée à appliquer est déterminée par la récurrence suivante :

$$\begin{aligned} (x_0, y_0) &= (x, y) \\ (x_{k+1}, y_{k+1}) &= (x_k^2 - y_k^2 + x, 2x_k y_k + y) \end{aligned}$$

Dès que (x_k, y_k) représente un point à l'extérieur du cercle de rayon 2, alors on considére que (x, y) *ne fait pas partie de l'ensemble de Mandelbrot.*

Il est possible d'obtenir des représentations *en couleur* de l'ensemble de Mandelbrot. Dans ce cas, lorsqu'un point fait partie de l'ensemble, il est coloré en noir ; autrement, on choisit une couleur

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

[2] Rappel : i représente le nombre complexe $\sqrt{-1}$.

qui dépend du nombre d'itérations requis pour faire en sorte que l'application de la récurrence aille à l'extérieur du cercle de rayon 2. L'extrait de code MPD 1 (voir page suivante) présente, entre autres, une fonction MPD qui permet de déterminer la couleur associée à un point (`x`, `y`), et ce en fonction d'un paramètre `maxIterations`.

On veut écrire, en MPD, un programme qui permet de déterminer la `couleur` d'un ensemble de points compris entre que -2.0 et $+2.0$. Plus précisément, le programme reçoit, sur la ligne de commande, les arguments suivants :

- Argument 1 : Un nombre réel h qui indique la distance entre les différents points, répartis uniformément entre -2.0 et $+2.0$ — on suppose que h divise l'intervalle $[-2.0, 2.0]$ de façon appropriée. Par exemple, $h = 0.5$ implique qu'il faudra déterminer la couleur de chacun des points satisfaisant la condition suivante :

$$\{(x, y) \mid x, y \in \{-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0\}\}$$

- Argument 2 : La valeur `maxIterations`, utilisée par la fonction `couleur`, à partir de laquelle on considère qu'un point fait partie de l'ensemble de Mandelbrot.

Plus précisément, on veut écrire en MPD une procédure ayant l'interface suivante :

```
procedure calculerCouleurs( res Couleur couleurs[*,*], real h, int maxIterations )
# PRECONDITION
#   On suppose que le tableau couleurs est suffisamment grand
#   pour contenir tous les points á distance h les uns des autres,
#   compris entre BORNE_INF et BORNE_SUP (inclusivement).
```

Pour exécuter ce programme, on a à notre disposition une machine multi-processeurs à mémoire partagée comportant `NB_PROCS` processeurs.

Vous devez écrire et comparez (avantages/désavantages, forces/faiblesses) trois (3) versions différentes de cette procédure (communications par variables partagées) :

a. Parallélisme itératif à granularité (très) fine — aussi fine que le permet le langage.

b. Parallélisme itératif à granularité grossière (allocation statique).

c. Parallélisme de style «sac de tâches» (allocation dynamique).

En conclusion de votre analyse comparative des diverses versions de la procédure, vous devez indiquer celle qui, d'après vous, devrait avoir les meilleures performances sur une machine telle qu'`arabica`, en justifiant brièvement votre choix.

Note : Vous pouvez écrire du pseudo-MPD, en d'autres mots, vous n'avez pas à écrire du code qui serait nécessairement accepté par le compilateur MPD. Par contre, soyez aussi clair et précis que possible. Entre autres, si un bout de code ne peut pas être réalisé par des opérations primitives, alors introduisez des procédures ou fonctions auxiliaires, pour lesquelles vous devez alors préciser l'interface (l'en-tête, la signature), mais sans nécessairement donner les détails de mise en oeuvre — entre autres, pour le sac de tâches, vous n'avez pas à donner la mise en oeuvre du sac de tâches lui-même, bien que vous deviez indiquer clairement ce qu'est une tâche et comment elles sont obtenues par les processus.

```
# Programme MPD pour calculer l'ensemble de Mandelbrot.

const int NB_PROCS = ...

# Bornes inferieure et superieure de l'espace a l'interieur duquel
# on examine les points pour determiner leur "couleur".
const real BORNE_INF = -2.0
const real BORNE_SUP = +2.0


# Couleurs possibles pour les differents points.
type Couleur = enum( BLANC, ..., NOIR )

procedure couleurPour( int nbIterations, int maxIterations ) returns Couleur c
# Retourne une couleur qui depend du nombre d'iterations:  plus nbIterations est grand,
# plus la couleur est foncee. Si nbIterations = maxIterations, alors c = NOIR.

procedure distance( real x, real y ) returns real d
{ d = sqrt( x**2 + y**2 ) }

procedure couleur( real x, real y, int maxIterations ) returns Couleur c
{
  real xk = x, yk = y;
  int nbIterations = 0;
  while( distance(xk, yk) < 2.0 & nbIterations < maxIterations ) {
    real xkp1 = xk*xk -  yk*yk + x;
    real ykp1 = 2 * xk * yk  + y;
    xk = xkp1;
    yk = ykp1;
    nbIterations += 1
  }
  c = couleurPour( nbIterations, maxIterations )
}


...

######################################################################
# Programme principal.
######################################################################

if (numargs() < 2) {
  printf( "Pas assez d'arguments?!\n" );
  stop( -1 );
}
real h;            getarg(1, h);
int maxIterations; getarg(2, maxIterations);

...

# Appel a la procedure de calcul des couleurs.
calculerCouleurs( couleurs, h, maxIterations );

# On affiche la matrice des couleurs.
afficher( couleurs );
```

Extrait de code MPD 1: Programme MPD pour calcul de l'ensemble de Mandelbrot et fonction pour déterminer la couleur d'un point.

2. Problème des mots clés (sac de tâches en MPI) (5 pts)

Soit le programme de recherche de mots-clés fait au labo il y a quelques semaines — voir le listing distribué en classe.

Supposons qu'on ait un *très grand* nombre de *petits fichiers* à analyser. Est-ce que ce programme, dans son état actuel, aurait de bonnes performances sur une machine comme `trex_cluster` (*cluster Beowulf* à 16 processeurs interconnectés par un réseau Ethernet haute vitesse)? Si ce n'est pas le cas, expliquez de quelle façon on pourrait/devrait modifier le programme pour améliorer ses performances.

Note : Vous n'êtes pas obligés de donner les détails du code ; il suffit plutôt d'expliquer les principales modifications qu'il faudrait apporter au programme.

3. Calculs «*embarassingly parallel*» (5 pts)

Wilkinson et Allen donnent la définition suivante d'un calcul parallèle idéal : «*[The] "ideal" computation from a parallel computing standpoint [is] a computation that can be divided into a number of completely independent parts, each of which can be executed by a separate processor. This is known as an* embarassingly parallel *computation.*»

a. Donnez un (ou deux) exemple(s) de problème qui est *embarassingly parallel*.

b. Si on ignore la lecture des données de départ, ainsi que l'écriture des résultats, est-ce qu'un problème qui est *embarassingly parallel* conduit nécessairement à une accélération linéaire? Justifiez brièvement votre réponse.

4. Mesures de performances (5 pts)

a. Pour un problème d'une certaine taille N, on a déterminé que 5 % des instructions d'un programme étaient des instructions d'entrée/sortie devant être exécutées sur un unique processeur. Par contre, toutes les autres instructions peuvent s'exécuter en parallèle (problème *embarassingly parallel*).

   Quel est le nombre mimimum de processeurs requis de façon à obtenir un programme parallèle s'exécutant avec une accélération d'au moins 10? Quelle est alors l'efficacité du programme sur cette machine?

b. Un programme s'exécute en 250 secondes sur une machine à 16 processeurs. À partir de *benchmarks* et de profilage du code, on détermine que 25 secondes du temps d'exécutions sont consacrées à l'initialisation (par ex., ouverture des fichiers, lecture des données) et finalisation (par ex., fermeture des fichiers) du programme. Durant les 225 autres secondes, on a que les 16 processeurs sont tous actifs, sans temps mort. Quelle est l'accélération de type Gustafson-Barsis (*scaled speed-up*) de ce programme? Quelle est alors l'efficacité du programme sur cette machine?

5. Automates cellulaires et jeu de la vie (20 pts)

Voici un extrait d'un mémoire de maîtrise[3] qui explique ce que sont les automates cellulaires :

> Un automate cellulaire peut être vu comme un modèle où l'espace, le temps et les grandeurs physiques prennent des valeurs discrètes. L'espace est représenté par une matrice de cellules, avec une, deux, ou plusieurs dimensions selon le problème à traiter. Chaque cellule peut, à un instant donné, être dans un nombre fini d'états. Ces états sont habituellement associés et représentés graphiquement par des couleurs. L'état d'une cellule au temps t est fonction de l'état, au temps $t-1$, d'un nombre fini de cellules appelé son voisinage. À chaque nouvelle unité de temps, les mêmes règles sont appliquées pour toutes les cellules de la grille, produisant une nouvelle génération de cellules dépendant entièrement de la génération précédente.
>
> De tels systèmes sont appelés *automates cellulaires* à la condition que ceux-ci rencontrent les trois propriétés fondamentales suivantes :
>
> a. Le parallélisme : L'ensemble des constituants de l'automate évoluent simultanément.
> b. La localité : Le nouvel état d'une cellule ne dépend que de son état actuel et de l'état des cellules de son voisinage immédiat.
> c. L'homogénéité : Les lois sont universelles, c'est-à-dire communes à l'ensemble de l'espace de l'automate cellulaire.

Les automates cellulaires ont été popularisés grâce au «Jeu de la vie» (de John Conway)[4] :

> *Life is played on a grid of square cells[...]. A cell can be live or dead. A live cell is shown by putting a marker on its square. A dead cell is shown by leaving the square empty. Each cell in the grid has a neighborhood consisting of the eight cells in every direction including diagonals.*
>
> *To apply one step of the rules, we count the number of live neighbors for each cell. What happens next depends on this number.*
>
> - *A dead cell with exactly three live neighbors becomes a live cell (birth).*
> - *A live cell with two or three live neighbors stays alive (survival).*
> - *In all other cases, a cell dies or remains dead (overcrowding or loneliness).*
>
> *Note: The number of live neighbors is always based on the cells before the rule was applied. In other words, we must first find all of the cells that change before changing any of them.*

La figure 1 présente un exemple de simulation du jeu de la vie pour une grille 7×7 : les «.» dénotent des cellules mortes (vides) alors que les «`X`» dénotent des cellules vivantes. Ici, la notion de «voisinage entre cellules» est basée sur une topologie en forme de «tore» (cylindre), c'est-à-dire que la première et la dernière lignes sont considérées comme étant adjacentes, de même que la première et dernière colonnes.

[3] Martin Lalonde, «Un simulateur de brèches hydro-thermales basé sur les automates cellulaires», Maîtrise en informatique, UQAM, Décembre 2005.

[4] `http://www.math.com/students/wonders/life/life.html`

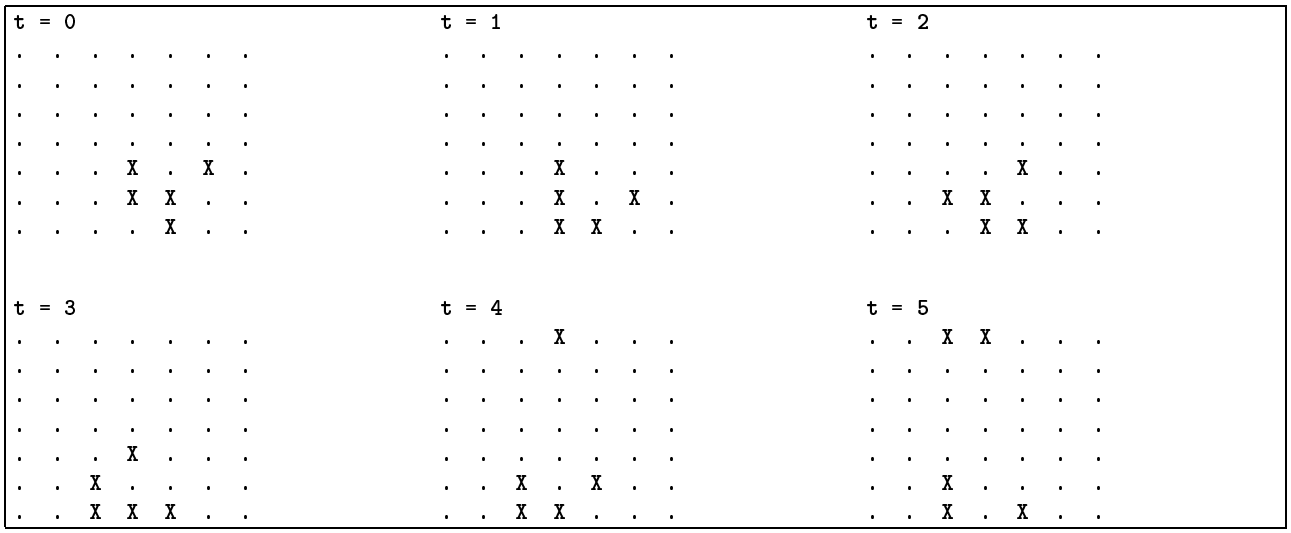

Figure 1: Un exemple de simulation du jeu de la vie pour une grille 7×7 — la valeur de t indique l'écoulement du temps, $t = 0$ indiquant la configuration initiale.

On veut écrire, en MPI, un programme qui permet de simuler le jeu de la vie, et ce pour un grand nombre de cellules et un grand nombre d'itérations. On suppose que le programme reçoit, sur la ligne de commande, les arguments suivants :

- Argument 1 : Nom du fichier contenant la configuration initiale de la grille, c'est-à-dire, nombre de lignes et nombre de colonnes de la grille, ainsi que l'état initial de chacune des cellules.
- Argument 2 : Durée de la simulation = Nombre d'itérations durant lesquelles on veut simuler l'évolution de l'automate cellulaire.

On veut exécuter ce programme sur une machine à mémoire distribuée avec 16 processeurs (style `trex_cluster`).

Décrivez et comparez (avantages et désavantages, forces et faiblesses) diverses façons d'écrire un programme MPI pour résoudre ce problème. Entre autres, vous devez évidemment comparer *différentes façons de répartir les données entre les processeurs*, tant en termes du *nombre de communications requises* que de la quantité de données à transmettre. Vous n'avez pas à écrire de code MPI, uniquement à décrire, de façon claire et précise, les grandes lignes de chacune des approches.