TP10. Programmation concurrente

I. Parallélisation du produit matriciel (C)

Le fichier à compléter `mult_mat.c` définit le type suivant :

```
struct matrice_s {
  int nbl;  //nombre de lignes
  int nbc;  //nombre de colonnes
  int** tab; //tableau de nbl tableaux de nbc coeffs
};

typedef struct matrice_s* matrice;
```

ainsi que plusieurs fonctions utilitaires.

1. Définir une fonction `produit` qui calcule le produit de deux matrices.

2. Identifier dans la fonction `produit` quels calculs pourraient être faits en parallèle. Doit-on utiliser des verrous pour garantir que la parallélisation de ces opérations mène à un résultat correct ?

3. Identifier les données qu'il est nécessaire de passer à chaque fil d'exécution pour qu'il effectue ces calculs. Créer alors une structure stockant les informations nécessaires, puis un type pour les pointeurs vers une telle structure.

4. Définir une fonction prenant un seul argument de type `void*` et de type de retour `void*` qui effectuera les calculs que doit réaliser un fil d'exécution.

5. Définir une fonction `matrice produit_parallele(matrice m1, matrice m2)` qui calcule le produit de deux matrices en parallélisant les calculs à l'aide de deux threads.

6. **(Bonus)** Modifier la fonction précédente pour qu'elle prenne en entrée un entier `nb_threads` et parallélise le calcul à l'aide de `nb_threads` threads.
Indication : On pourra utiliser un tableau de threads stocké dans le tas.

II. Calcul du maximum par "Diviser pour régner" (OCaml)

Nous nous intéressons ici à l'algorithme de calcul du maximum d'un tableau suivant.

- Si le tableau est de taille 1, alors le maximum est la valeur contenue dans l'unique case.
- Si le tableau est de taille > 1, on le découpe en deux tableaux, dont on calcule les maximums par appel récursif, et on retourne alors le maximum de ces deux maximums.

Cet algorithme, de type diviser pour régner, suit l'idée d'un tournoi sportif. Une rapide étude de complexité indique que cet algorithme a une complexité en $\Theta(n)$ (où n est la taille du tableau), qui est aussi la complexité de l'algorithme de calcul du maximum par une simple boucle parcourant le tableau.

1. Implémenter l'algorithme décrit.

On va confier les appels récursifs à différents threads. Pour cela, on définit le type ci-dessous, permettant la représentation des entrée/sorties de la fonction récursive auxiliaire calculant le maximum.

```
type espace_recherche = {
    tab : int array;    (* tableau dont on cherche le max *)
    d : int;            (* indice de début de la recherche *)
    f : int;            (* indice de fin de la recherche *)
    mutable ret : int; (* valeur de retour *)
}
```

En passant une telle structure à un appel récursif, on lui indique non seulement quelle recherche de maximum effectuer, mais aussi où stocker le résultat : dans le champ `ret`.

2. Proposer une implémentation parallélisée de l'algorithme. On veillera à utiliser au plus n threads, où n est la taille du tableau.