TP n°6 - Programmation concurrente

Notions abordées

- Utilisation de la librairie `pthread` en C
- Définition fonctions de type `void*`, transtypage, et création de struct en C
- Utilisation des modules `Thread` et`Mutex` en OCaml

Exercice 1 : Parallélisation pour le produit de deux matrices

Q. 1 Télécharger le fichier `compagnon_mult_mat.c` sur cahier de prépa. Prendre connaissance du type utilisé pour représenter les matrices et des fonctions disponibles. Définir ensuite une fonction `produit` qui calcule le produit de deux matrices.

Q. 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 ?

Q. 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 agrégeant les informations nécessaires, puis un type pour les pointeurs vers une telle structure.

Q. 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.

Q. 5 Définir finalement une fonction `produit_parallele` qui calcule le produit de deux matrices en parallélisant le plus de calculs possible. Tester la fonction obtenue. On peut la comparer à `produit`. À l'aide d'un affichage bien choisi, on peut observer que les calculs faits en parallèle ne sont pas toujours faits dans le même ordre.

Exercice 2 : Addition binaire en parallèle

Les questions 2, 5 et 6 sont à traiter sur machine, en C. Un fichier `compagnon_add_par_bloc.c` contenant quelques fonctions utiles est disponible sur cahier de prépa. On veillera à le compiler avec l'option `-pthread` nécessaire pour que la librairie du même nom soit bien incluse.

Q. 1 Donner une description précise de la fonction `mystere` définie ci-dessous.

```c
bool mystere(bool* t1, bool* t2, bool* tab, int n){
  // hyp : t1, t2 et tab sont de taille n
  // ???
  bool r = false;
  for (int i = 0; i < n; i++) {
    if (t1[i] && t2[i]) {
      tab[i] = r;
      r = true;
    } else if (t1[i] || t2[i]) {
      tab[i] = !r;
    } else {
      tab[i] = r;
      r = false;
    }
  }
  return r;
}
```

Q. 2 On souhaite adapter cette fonction pour qu'elle puisse être exécutée par un `pthread`.

- Se munir d'un type qui permet d'agréger des valeurs d'arguments et une valeur de sortie pour la fonction précédente.
- Définir alors une fonction de type de retour `void*` et prenant un seul argument de type `void*` équivalente à la fonction précédente.
- Proposer alors un jeu de quelques tests permettant de vérifier que les appels à ces deux fonctions coïncident.

Q. 3 Soit $n \in \mathbb{N}^*$. Soient $a_{n-1}a_{n-2}\dots a_0 \in \{0,1\}^n$ et $b_{n-1}b_{n-2}\dots b_0 \in \{0,1\}^n$. Soit $k \in [\![0,n]\!]$. On note♣ $a^g = a_{k-1}a_{k-2}\dots a_0$ et $a^d = a_{n-1}a_{n-2}\dots a_k$. On définit de même b^g et b^d. Si c^g (resp. c^d) désigne le mot composé des k (resp. $n-k$) bits de poids faible de la somme de a^g et b^g, et si $r^g \in \{0,1\}$ (resp. $r^d \in \{0,1\}$) indique si une retenue a été oubliée dans cette somme, comment peut-on calculer la somme de a et b ?

Q. 4 En s'inspirant de la question précédente, proposer un mécanisme d'addition par blocs qui puisse être parallélisé. À quelle condition cette parallélisation semble profitable en terme d'efficacité ?

Q. 5 Définir une fonction `add_par_bloc` qui additionne deux nombres codés dans des tableaux de booléens en deux blocs traités **séquentiellement** grâce à la fonction proposée dès la question 1. (On s'assure dans cette question que le découpage, les sous-appels et la gestion des retenues sont maîtrisés, la parallélisation à proprement parler est traitée à la question suivante).

♣. si les lettres g et d en exposant semblent inversées, ces notations sont cohérentes si on envisage le codage dans un tableau, où les bits de poids faible sont dans la case d'indice 0, usuellement représentée à gauche.

Q. 6 Définir une fonction `add_parrallele` qui additionne deux nombres codés dans des tableaux de booléens en deux blocs traités **parallèlement** grâce à la fonction proposée à la question 2.

Exercice 3 : Calcul du maximum par "Diviser pour régner"

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 de tailles $\geqslant 1$, dont on calcule les maximums par appel récursif, 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.

Q. 1 En utilisant plusieurs fils d'exécution, transformer l'algorithme précédent pour qu'il soit de complexité ♣ $\Theta(\log(n))$.

Dans l'objectif d'une implémentation en OCaml, 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   ;            (* le tableau dont on cherche le max *)
  g   : int         ;            (* la borne gauche, au sens large *)
  d   : int         ;            (* la borne droite, au sens large *)
  mutable ret : int ;            (* la valeur de retour *)
}
```

En passant une telle structure à un appel récursif, on lui indique non seulement quelle recherche de maximum éffectuer (grâce aux champs `tab`,`g` et `d`), mais aussi où stocker le résulat : dans le champ `ret`.

Q. 2 Proposer une implémentation en OCaml de l'algorithme proposé ci-avant.

Q. 3 Si ce n'est pas le cas, modifier votre algorithme pour qu'il utilise au plus n fils d'exécution, où n est la taille du tableau à trier.

♣. La notion de complexité n'a jamais été définie dans le cas d'algorithme manipulant plusieurs fils d'exécution, la complexité sera donc ici le "temps d'exécution". On supposera que la machine dispose d'un nombre non borné de fils d'exécution, et que ces fils d'exécution s'exécutent en parallèle, même si cette hypothèse n'est pas très réaliste.

Exercice 4 : Un très mauvais algorithme de tri

On se propose d'implémenter un algorithme de tri inspiré du tri bulle, mais mettant à profit plusieurs fils d'exécution. Dans toute la suite T désigne un tableau d'entiers de taille n. L'idée est la suivante : un tableau T de taille n est trié (de manière croissante) lorsque :

$$\forall i \in [\![0, n-2]\!], T[i] \leqslant T[i+1]$$

Aussi on utilise $n-1$ fils d'exécution, les "trieurs" : le $i^{\text{ème}}$ étant chargé d'assurer que $T[i] \leqslant T[i+1]$. S'il constate que ce n'est pas le cas, il range $T[i]$ et $T[i+1]$ en les inversant. Chacun de ces $n-2$ fils d'exécution a une vision très locale de l'état de tri global du tableau, aussi on charge un autre fil d'exécution, le "vérifieur", de parcourir le tableau pour vérifier si celui-ci est globalement trié ou non. Si c'est le cas il peut arrêter tous les autres fils d'exécution, si ce n'est pas le cas il recommence sa vérification du tableau.

Q. 1 Proposer un mécanisme de protection assurant qu'aucune valeur du tableau ne soit perdue, et ce malgré les éventuels entrelacements des fils d'exécutions. Expliciter l'utilisation faite de ces protections par les différents fils d'exécution.

Q. 2 Montrer qu'il est nécessaire d'accorder une attention particulière aux ordonnancements des différents mécanismes de protection afin d'éviter les inter-blocages. Justifier que votre choix d'ordonnancement ne conduit pas à un inter-blocage.

Afin de réaliser l'implémentation en OCaml, les différents fils d'exécutions échangeront au moyen de variables globales :

- `tab: int array` le tableau à trier ;
- `est_trie: bool ref` un booléen initialisé à `false` indiquant si le tableau est trié, qui sera notamment sera utile pour permettre au fil d'exécution vérifieur d'arrêter les fils d'exécution trieurs ;
- les objets mis en place pour les mécanismes de protection.

Q. 3 Proposer une fonction OCaml `trieur (i: int): unit` qui est le code du fil d'exécution trieur d'indice i.

Q. 4 Proposer une fonction OCaml `verifieur (): unit` qui est le code du fil d'exécution vérifieur.

Q. 5 En déduire une fonction `tri : unit -> unit` permettant de trier le tableau `tab` par l'algorithme concurrent présenté ci-avant.

On pourra modifier les fonctions ci-dessus pour faire apparaître l'inter-blocage mentionné en **Q. 2**.

Lancer deux fils d'exécution en parallèle en OCaml

Pour gérer des fils d'exécution en OCaml on utilise le module `Thread` de la librairie `Thread.Posix`. On peut charger ce module sur `utop` en tapant `#require "threads.posix";;`. Afin de compiler un programme OCaml utilisant ce module on pourra utiliser la ligne de commande ci-dessous.

```
ocamlc -thread unix.cma threads.cma main.ml -o main
```

Les fils d'exécution sont des objets de type `Thread.t`. Ils doivent être créés par appel à la fonction `Thread.create : ('a -> 'b) -> 'a -> Thread.t` qui prend en arguments :

- une fonction qui est celle que le fil d'exécution doit exécuter ;
- l'argument sur lequel ce fil d'exécution doit exécuter la fonction.

La fonction `Thread.create` retourne alors le fil d'exécution ainsi créé.

De même que pour la création de fils d'exécution en C, on peut se munir d'une structure permettant la gestion des entrées/sorties de telles fonctions.
On peut demander à attendre qu'un fil d'exécution ait terminé son exécution grâce à la fonction `Thread.join : Thread.t -> unit` qui prend en argument le fil d'exécution en question.

```
type args =
  {
    nb: int           ;
    mutable res : int ;
  }

let au_carre (args: args): unit =
  args.res <- args.nb * args.nb
```

```
let () =
  let arg1 = {nb = 2; res = -1} in
  let arg2 = {nb = 9; res = -1} in
  let pa = Thread.create au_carre arg1 in
  let pb = Thread.create au_carre arg2 in
  Thread.join pa;
  Thread.join pb;
  assert (arg1.res = 4 && arg2.res = 81)
```

Lancer deux fils d'exécution en parallèle en C

Pour gérer des fils d'exécution en C on utilise la bibliothèque `pthread`. Ainsi, on veillera à indiquer `include <pthread.h>` en tête du fichier `main.c`, et à le compiler avec l'option `-pthread` :

```
gcc -pthread main.c -o main
```

Les fils d'exécution sont des objets de type `pthread_t`. Ils doivent d'abord être déclarés puis sont lancés par la fonction `pthread_create` qui prend 4 arguments :

- l'adresse du fil d'exécution, de type `pthread_t*` donc ;
- une adresse qui ne nous est pas utile, on mettra donc `NULL` ;
- une fonction `void* todo(void* arg);`, qui est celle que le fil d'exécution doit exécuter ;
- un pointeur vers les arguments de type `void*`.

Il est donc nécessaire de formater les instructions à effectuer dans une fonction `void* todo(void* arg)`. Pour cela on crée d'abord une structure `struct arg_s` qui rassemble les données utiles à `todo` dans un même objet `a` dont l'adresse sera passée au fil d'exécution comme 4-ème argument de `pthread_create`. La fonction `todo` commence par transtyper son argument `arg` pour y reconnaître un pointeur de type `(struct arg_s)*`. Les calculs peuvent donc être codés en accédant aux données par `a->champ`. Enfin le résultat ne peut être retourné en sortie de `todo`, mais doit être enregistré dans un objet existant hors de la fonction (on ajoute parfois son adresse dans la structure `arg_s`). Autrement dit la fonction `todo` doit non seulement prendre ses entrées à travers un unique pointeur, mais doit aussi agir par effet de bord uniquement.

On attend que plusieurs fils d'exécution aient terminé leur exécution grâce à la fonction `pthread_join` qui prend 2 arguments :

- le processus, de type `pthread_t` donc ;
- une adresse qui ne nous est pas utile, on mettra donc `NULL`.

```c
#include <pthread.h>
#include <assert.h>

struct args_s {
  int nb;
  int* res;
};
typedef struct args_s arg_carre;

void* au_carre(void* args) {
  arg_carre* a = (arg_carre*) args;
  *(a->res) = a->nb * a->nb;
  return NULL;
}

int main(){
  int resA, resB;
  arg_carre argsA = {2, &resA};
  arg_carre argsB = {9, &resB};
  pthread_t pA, pB;
  pthread_create(&pA, NULL, au_carre, &argsA);
  pthread_create(&pB, NULL, au_carre, &argsB);
  pthread_join(pA, NULL);
  pthread_join(pB, NULL);
  assert((resA == 4) && (resB == 81));
  return 0;
}
```

Lancer une kyrielle de fils d'exécution en parallèle en C

Afin de lancer un grand nombre ♡de fils d'exécution en parallèle, il faut les déclarer non pas explicitement un à un, mais dans un tableau à l'aide d'un `malloc`. De même les structures qui servent à passer les arguments aux fils d'exécution doivent être stockées dans des tableaux. Il doit y avoir autant de structures différentes physiquement (placées à des endroits différents en mémoire) que de fils d'exécution. En particulier déclarer les arguments dans une structure qu'on modifie à chaque tour de boucle ne marche pas : l'espace alloué pour les arguments du premier fil est un espace dans la pile qui est modifié dès le deuxième tour de boucle, et potentiellement avant que le premier fil n'ait pu lire ses arguments, et/ou écrire son résultat. On donne ci-dessous un exemple type de lancement d'un grand nombre de fils d'exécution.

```c
#include <pthread.h>
#include <stdlib.h>
#include <assert.h>

struct arg_s {
  int i;      //indice de la case
  int* t_in;  //tableau des entrées
  int* t_out; //tableau des résulats
};
typedef struct arg_s arg;

void* moins_un(void* ptr_args) {
  arg* a = (arg*) ptr_args;
  a->t_out[a->i] = a->t_in[a->i] - 1;
  return NULL;
}

int main(){ //remplit res tq pour i\in[1..n], res[i] = val[i]-1
  int n = 10;
  int* val = (int*) malloc(n*sizeof(int));
  int* res = (int*) malloc(n*sizeof(int));

  //1-préparer les arguments
  arg* args = (arg*) malloc(n*sizeof(struct arg_s));
  for(int k = 0; k < n; k++){
    args[k] = (arg) {k,val,res};
  }
  //2-lancer les threads
  pthread_t* threads = (pthread_t*) malloc(n*sizeof(pthread_t));
  for(int k = 0; k < n; k++){
    pthread_create(threads+k, NULL, moins_un, args+k);
  }
  //3-attendre la fin des threads
  for(int k = 0; k < n; k++){
    pthread_join(threads[k], NULL);
  }
  //4-lire les résultats
  assert((res[0] == val[0]-1) && (res[n-1] == val[n-1]-1));
  //5-libérer la mémoire
  free (val);
  free (res);
  free (args);
  free (threads);
  return 0;
}
```

Cette notice fait suite à la notice "Lancer **deux** fils d'exécution en parallèle en C ".