Informatique - Concours blanc - MPI -2026

1 Exercice : Synchronisation

1.1 Rendez-vous et barrière de synchronisation

Deux threads T_1 et T_2 exécutent les fonctions suivantes :

```
void* T1(void* arg) {
    a1();
    a2();
    return NULL;
}
```

```
void* T2(void* arg) {
    b1();
    b2();
    return NULL;
}
```

1. Si on lance deux threads en parallèle qui exécutent respectivement `T_1` et `T_2`, quels sont les entrelacements possibles des exécutions de `a_1,a_2,b_1` et `b_2` (on supposera que ces quatres fonctions s'exécutent de manière atomique) ?
2. À l'aide de deux sémaphores initialisés à 0, écrire une version modifiée des fonctions `T1` et `T2` garantissant simultanément :
   — a_2 ne peut pas être exécutée avant que b_1 ne soit terminée ;
   — b_2 ne peut pas être exécutée avant que a_1 ne soit terminée.

On considère maintenant $n \geq 2$ threads exécutant tous le code suivant :

```
void* worker(void* arg) {
    step1();
    step2();
    return NULL;
}
```

La contrainte que l'on souhaite garantir est la suivante : aucun thread ne doit exécuter `step2()` tant que tous les n threads n'ont pas terminé d'exécuter `step1()`.

Les variables partagées sont :

```
int n;
int count = 0;
pthread_mutex_t mutex;
sem_t barrier; /*initialise \‘a 0*/
```

Nous allons considérer une première solution :

```
void* worker(void* arg) {
    step1();

    pthread_mutex_lock(&mutex);
    count++;
    pthread_mutex_unlock(&mutex);

    if (count == n) {
        sem_post(&barrier);
    }

    sem_wait(&barrier);

    step2();
    return NULL;
}
```

3. Décrire un ordonnancement concret (par exemple pour $n = 3$) qui met en évidence que cette solution n'est pas fonctionnelle.
4. Modifier la solution proposée afin de fournir une solution correcte au problème. Expliquer brièvement le principe.
5. Une fois que les n threads ont franchi la barrière, quelle est la valeur du sémaphore `barrier` ?
6. Que se passerait-il si un $(n + 1)$eme thread cherchait à franchir la barrière ?

Pour la suite du sujet on notera `barrier_wait` la portion de code corrigée précédemment qui se trouve entre `step1()` et `step2` dans la fonction `worker`.

On considère $n \geq 2$ threads exécutant une boucle :

```
void* worker(void* arg) {
    while(true) {
        step();
        barrier_wait();
    }
}
```

La fonction `barrier_wait()` doit permettre de garantir que tous les n threads ont terminé leur tour de boucle (l'exécution de la fonction `step()`) avant qu'un d'entre eux soit en mesure de recommencer une nouvelle exécution de `step()` et donc un nouveau tour de boucle.

7. Donner un ordonnancement montrant que cette solution permet un mélange entre deux itérations consécutives du `while`. On proposera un ordonnancement concret en se contentant par exemple de 3 threads pour simplifier la description.

Voici une tentative de solution fonctionnelle utilisant une nouveau sémaphore `newpassage` en plus des variables déjà utilisées jusqu'ici :

```
void barrier_wait() {

    pthread_mutex_lock(&mutex);
    count++;
    if (count == n) {
        sem_wait(&newpassage);
        sem_post(&barrier);
    }
    pthread_mutex_unlock(&mutex);

    sem_wait(&barrier);
    sem_post(&barrier);

    pthread_mutex_lock(&mutex);
    count--;
    if (count == 0) {
        sem_post(&newpassage);
    }
    pthread_mutex_unlock(&mutex);

    sem_wait(&newpassage);
    sem_post(&newpassage);
}
```

8. Donner l'initialisation correcte de `newpassage` pour que cela fonctionne. En sortie d'un tour de boucle, quelles sont les valeurs des deux sémaphores `barrier` et `newpassage` ? Rajouter une instruction dans le code qui permet d'avoir ces valeurs correctes pour permettre d'effectuer correctement le prochain tour de boucle.

1.2 Application : construire des molécules H_2O

On considère deux types de threads :

— threads oxygène (O),

— threads hydrogène (H).

Une molécule d'eau requiert exactement **2 H et 1 O**.

Nous allons considérer deux types de threads : Oxygen et Hydrogen, ils s'exécutent de manière concurrentes mais doivent tous exécuter simmultannément une opération `lier()` lorsque deux threads de type hydrogène et un de type oxygène sont réunis au même point de synchronisation. Plus précisemment, il vont chacun appeler la fonction `lier()` et on souhaite garantir à l'aide d'une barrière de synchronisation qu'il aient tous les trois terminé leur opération de liaison avant de procéder à la suite du programme et qu'aucun autre thread ne commence à invoquer la fonction `lier()` tant que la barrière n'est pas franchie.

— Si un thread de type oxygène arrive à la barrière c'est qu'il a terminé l'appel à `lier()` et si on n'a pas deux threads hydrogène présents, il doit attendre.

— Si un tread de type hydrogène arrive à la barrière il doit attendre qu'il y ait un thread de type oxygène et un autre thread de type hydrogène.

Formellement, si l'on note S la suite des threads qui exécutent `lier()`, et si l'on partitionne S en blocs consécutifs de taille 3, alors pour tout bloc B_i : il contient un thread de type oxygène et deux de type hydrogène et il n'existe pas d'exécution où un élément de B_{i+1} exécute `lier()` avant que tous les éléments de B_i aient exécuté `lier()`.

Voici les variables partagées par l'ensemble des threads :

```
int oxygen = 0;
int hydrogen = 0;

pthread_mutex_t mutex;

sem_t oxyQueue;
sem_t hydroQueue;

barrier_t barrier3; // barriere de taille 3
```

9. Compléter le code suivant pour un thread oxygène :

```
void* oxygen_thread(void* arg) {
    pthread_mutex_lock(&mutex);
    oxygen++;

    if (/* condition */) {
        /* sem_post(...)




        */
    } else {
        pthread_mutex_unlock(&mutex);
    }

    sem_wait(&oxyQueue);
    lier();

    barrier_wait(&barrier3);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

10. Ecrire de manière analogue le code d'un thread hydrogène et indiquer pourquoi ce thread ne libère pas le mutex après le passage de la barrière.
11. Montrer que l'utilisation de la barrière de taille 3 et du mutex garantit l'absence de mélange entre deux molécules consécutives.

2 Partie II : Algorithmes de couplage

Cette partie comporte des questions nécessitant un code en C.

On s'intéresse ici à un problème d'appariement entre deux groupes d'individus U et V, que l'on suppose de même cardinalité n. On cherche à apparier les éléments de V aux éléments de U, chaque élément devant être apparié à exactement un élément de l'autre ensemble. Chaque élément de V (respectivement de U) a, de plus, un ordre de préférence total entre tous les éléments de U (respectivement de V).

Définitions et notations

Définition 1 (Couplage, couplage parfait). Un ensemble de paires $A \subseteq V \times U$ est un couplage si tout $x \in V$ et tout $y \in U$ apparaissent dans au plus un élément de A. Le couplage est dit parfait si tout $x \in V$ et tout $y \in U$ apparaissent dans un et un seul élément de A.

Notations. Dans toute la suite on notera :

— $|E|$ le cardinal de l'ensemble E,

— $>_x$ l'ordre associé à l'élément $x \in V \cup U$: si $u_1 \in U$ préfère strictement $v_1 \in V$ à $v_2 \in V$ alors $v_1 >_{u_1} v_2$. Par extension, on définit de même la notion de préférence $\geq_x$,

— $m_A(x)$ le partenaire de $x \in V \cup U$ dans le couplage parfait A.

Définition 2 (Couplage parfait stable). Un couplage parfait A est dit stable si, pour tous couples (v_1, u_1) et (v_2, u_2) de A, il est impossible que l'on ait $u_2 >_{v_1} u_1$ et $v_1 >_{u_2} v_2$.

Définition 3 (Pareto-optimalité). Soient A et A' deux couplages parfaits stables et $E \subseteq V \cup U$. On dit que A Pareto-domine A' sur E si et seulement si pour tout $x \in E$, $m_A(x) \geq_x m_{A'}(x)$ et il existe $x \in E$ tel que $m_A(x) >_x m_{A'}(x)$. Un couplage est Pareto-optimal s'il est stable et n'est pas Pareto dominé sur $V \cup U$.

En économie, la Pareto-optimalité d'un système exprime le fait que l'on ne peut améliorer la satisfaction d'un individu du système sans diminuer la satisfaction d'un autre.

II.1 — Recherche de couplages stables

L'algorithme de Gale–Shapley (algorithme 1) permet de montrer qu'il existe toujours au moins un couplage parfait stable et d'en trouver un de façon efficace. L'idée de l'algorithme est tout d'abord de choisir un des deux ensembles (ici pour illustration V). Les éléments de V font des propositions aux éléments de U. Quand un élément de V fait une proposition à un élément de U, celui-ci peut soit la refuser définitivement, soit l'accepter provisoirement en attendant une éventuelle meilleure offre. Toute nouvelle acceptation vaut rejet définitif des propositions précédemment acceptées.

Algorithm 1 Algorithme de Gale–Shapley

Require: $n = |V| = |U|$, les listes de préférences des éléments de V et U
Ensure: Un couplage parfait stable A

$A \leftarrow \varnothing$
while il existe un élément de V non apparié **do**
  Choisir un tel v
  $u \leftarrow$ élément de U que v préfère parmi ceux à qui il n'a pas encore proposé
  if u n'est pas apparié dans A **then**
    $A \leftarrow A \cup \{(v, u)\}$
  else
    u est apparié dans A à v'
    if $v >_u v'$ **then**
      $A \leftarrow A \setminus \{(v', u)\}$
      $A \leftarrow A \cup \{(v, u)\}$
    end if
  end if
end while
return A

Soit le jeu de données D suivant : $V = \{v_1, v_2, v_3\}$, $U = \{u_1, u_2, u_3\}$ et :

V	**U**
$u_2 >_{v_1} u_1 >_{v_1} u_3$	$v_1 >_{u_1} v_3 >_{u_1} v_2$
$u_1 >_{v_2} u_2 >_{v_2} u_3$	$v_2 >_{u_2} v_1 >_{u_2} v_3$
$u_1 >_{v_3} u_2 >_{v_3} u_3$	$v_2 >_{u_3} v_1 >_{u_3} v_3$

Q1. Donner la trace de l'algorithme 1 sur ces données. Le choix de v dans l'algorithme se fera par ordre croissant des indices.

Q2. Montrer que l'algorithme 1 termine.

Q3. Donner, sans le prouver, un invariant de boucle permettant de prouver que l'algorithme 1 retourne un couplage parfait.

Q4. Montrer que le couplage parfait calculé est stable.

Q5. Montrer que cet algorithme effectue moins de n^2 itérations de boucles.

Pour des préférences données, il peut exister plus d'un couplage parfait stable. Soient A et A' deux tels couplages pour un problème donné. On dira qu'un élément de $V \cup U$ préfère A à A' s'il n'a pas le même partenaire dans les deux couplages et s'il préfère celui de A à celui de A'.

Q6. Montrer que si v préfère le couplage A, alors son partenaire u dans A préfère A'.

La correction de l'algorithme de Gale–Shapley ne dépend pas de l'élément $v \in V$ qui fait sa proposition à chaque tour. En fait, on montre dans la suite que, quel que soit l'élément $v \in V$ non encore apparié qui fait une proposition à chaque tour, l'algorithme calcule toujours le même couplage parfait stable.

Dans la suite, on dira qu'un élément $u \in U$ est *réalisable* pour un élément $v \in V$ s'il existe un couplage parfait stable qui contient (v, u).

Q7. Montrer qu'au cours de l'algorithme 1, chaque $v \in V$ n'est rejeté que par les éléments $u \in U$ qui ne sont pas réalisables pour v.

Soit $M(v) \in U$ l'élément réalisable le mieux classé sur la liste des préférences de v et $P(u) \in V$ l'élément réalisable le moins bien classé sur la liste des préférences de u. On note enfin $A^* = \{(v, M(v)),\, v \in V\}$.

Q8. Montrer que lors de l'exécution de l'algorithme 1 :

1) Tout $v \in V$, s'il est apparié, l'est avec $u \in U$ supérieur ou égal à $M(v)$ dans sa liste de préférences.
2) Tout $v \in V$, s'il n'est pas apparié, n'a fait des propositions qu'à des $u \in U$ strictement supérieurs à $M(v)$ dans sa liste de préférences.

Q9. En déduire que quel que soit l'élément $v \in V$ choisi dans la boucle *tant que* de l'algorithme 1, l'algorithme termine en produisant l'ensemble A^*.

Ainsi, l'algorithme calcule le meilleur couplage parfait stable possible du point de vue des éléments de V. Il s'avère également que l'algorithme de Gale–Shapley fait correspondre u avec $P(u)$, pour tout $u \in U$. L'algorithme avantage donc de manière claire le groupe ayant l'initiative (ici V) : le couplage stable produit est dit Pareto-V-optimal en ce sens qu'il n'est pas Pareto-dominé sur V.

II.2 — Couplages Pareto-optimaux

Dans cette sous-partie, on utilise également le jeu de données D.

L'algorithme de Gale–Shapley ne retourne pas nécessairement un couplage Pareto-optimal. On s'intéresse ici à un algorithme susceptible de produire cette propriété.

Pour construire un couplage Pareto-optimal, on représente les données à l'aide d'un graphe orienté $G = (S, E)$ où :

— les sommets S sont les éléments de $V \cup U$,

— les arcs E représentent les premiers choix : $(v, u) \in E$ si et seulement si u est le premier choix de v dans sa liste de préférences et $(u, v) \in E$ si et seulement si v est le premier choix de u.

Q10. Montrer que G contient au moins un circuit, c'est-à-dire un chemin dont les deux sommets extrémités sont identiques.

Q11. Justifier par l'absurde que tout $x \in V \cup U$ est dans au plus un circuit.

On propose alors l'algorithme 2 pour calculer un couplage, dont on admettra qu'il est Pareto-optimal pour V.

Algorithm 2 Algorithme de couplage par graphe orienté

Require: $n = |V| = |U|$, les listes de préférences des éléments de V et U
Ensure: Un couplage A

$A \leftarrow \varnothing$
while il existe un élément de V non apparié **do**
    Construire le graphe G sur les données courantes
    for all arc (v, u) présent dans un circuit de G **do**
        $A \leftarrow A \cup \{(v, u)\}$
    end for
    Supprimer du problème tous les $v \in V$ et les $u \in U$ appariés
    Mettre à jour les listes de préférences des individus restants
end while
return A

Q12. Donner les graphes et la construction itérative de A produits par l'algorithme 2 sur D.

Q13. En utilisant les données D, montrer que l'algorithme 2 ne produit pas toujours un couplage stable.

On s'intéresse alors à une autre forme de stabilité pour assurer à la fois une propriété de Pareto et une stabilité dans les couplages.

Définition 4 (P-stabilité). Un couplage A est P-stable s'il n'existe aucune paire $(v, v') \in V^2$ telle que v préfère $m_A(v')$ et v' préfère $m_A(v)$.

Q14. Montrer par l'absurde que l'algorithme 2 produit un couplage P-stable.

Q15. Montrer que l'algorithme 1 ne produit pas nécessairement un couplage P-stable.

II.3 — Implémentation

Dans la suite, les individus sont numérotés de 0 à $n-1$. Ainsi $V = \{0, \ldots, n-1\}$ et $U = \{0, \ldots, n-1\}$.

On suppose disposer de deux matrices de taille $n \times n$ L_v et L_u donnant respectivement les listes de préférences des éléments de V et U : ainsi, $L_u[i][j]$ est le j-ème élément de V préféré par l'élément $i \in U$.

On définit ces tableaux en C par l'intermédiaire du code suivant :

```
/* La directive #define est utilisee pour definir une valeur pour la constante n qui
   servira \'a declarer des tableaux de taille fixe. */
#define n 4

int main(void) {
  int Lv[n][n], Lu[n][n];
  (...)
}
```

II.3.1 — Algorithme de Gale–Shapley

Pour connaître rapidement le classement des éléments de V dans les listes de préférences des éléments de U, on définit un tableau `Rang` de taille $n \times n$, tel que `Rang[i][j]` donne le rang de $j \in V$ dans la liste de préférences de l'élément $i \in U$. Par convention, les rangs commencent à 0.

Q16. Écrire une fonction de prototype

```
void calcule_rang(int Lu[n][n], int Rang[n][n]);
```

qui construit le tableau `Rang`.

Afin de pouvoir ajouter ou supprimer rapidement des couples (v, u) au couplage parfait stable A, on définit deux tableaux `CoupleV` et `CoupleU` tels que `CoupleV[i]` ou `CoupleU[i]` est le partenaire actuel de i. On définit enfin un dernier tableau `USuiv` de taille n stockant pour chaque élément de V le rang du prochain élément de U à qui il peut faire une proposition.

Q17. Proposer des valeurs d'initialisation pour ces trois tableaux.

Q18. Avec ces structures de données, écrire une fonction de prototype

```
void gale_shapley(int Lv[n][n], int Lu[n][n], int CoupleU[n], int CoupleV[n]);
```

qui réalise l'algorithme 1. On ne retournera pas le couplage parfait stable, on remplira seulement les variables `CoupleU` et `CoupleV` passées en argument. Pour le choix de v, on prendra les éléments de V par ordre croissant de leur numéro en choisissant à chaque fois le premier libre.

II.3.2 — Algorithme de couplage par graphe orienté

Sans explicitement coder tout l'algorithme 2 qui requiert de rechercher les circuits dans G, on se propose de coder quelques fonctions utiles à la réalisation de l'algorithme.

Pour pouvoir numéroter les sommets du graphe de manière continue, les individus sont maintenant numérotés de 0 à $2n-1$ de la manière suivante : $V = \{0, \dots, n-1\}$ et $U = \{n, \dots, 2n-1\}$.

On code le graphe orienté G par listes d'adjacence. On propose les types suivants :

```
struct noeud {
  int individu; // Element de U union V
  struct noeud* suivant;
};
typedef struct noeud* liste;

struct graphe_s {
  int nb_sommets;
  liste* liste_adjacence;
};
typedef struct graphe_s graphe;
```

Q19. Écrire une fonction de prototype

```
void ajoute_arc(graphe *g, int i, int j);
```

qui ajoute l'arc (i, j) au graphe G.

Q20. Écrire une fonction de prototype

```
void supprime_arc(graphe *g, int i, int j);
```

qui supprime l'arc (i, j) du graphe G.

Q21. Écrire une fonction de prototype

```
void supprime_sommet(graphe *g, int s);
```

qui supprime le sommet s du graphe G. Pour ce faire, on supprimera uniquement tous les arcs dont ce sommet est origine ou destination.

Q22. Écrire une fonction de prototype

```
graphe *construit_graphe(int Lv[n][n], int Lu[n][n]);
```

qui construit le graphe G utilisé dans l'algorithme 2 et dont la définition est donnée plus haut. On prendra bien garde à la numérotation des éléments de U.