ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2019)
3 crédits (3-1.5-4.5)

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Vendredi le 25 octobre 2019

HEURE: 9h30 à 11h20

DUREE: 1H50

NOTE: aucune documentation permise sauf un aide-mémoire, préparé par l'étudiant, qui consiste en une feuille de format lettre manuscrite recto verso

Ce questionnaire comprend 4 questions pour 20 points

Question 1 (5 points)

a) On vous demande de mettre en place un centre de données qui hébergera le service de prise de rendez-vous en clinique médicale «Bye Bye Bobo». Il est bien sûr impensable qu'un tel service soit hors-service pendant de longues périodes. Ainsi, on vous impose un objectif de disponibilité du service de $p = 0.995$.

Votre centre de données dispose de deux liens internet indépendants dont la probabilité de panne est de 0.05 chacun.

Le centre de données est composé de 15 serveurs dont l'électronique (i.e. tout le serveur sauf les disques) a une probabilité de panne de 0.001. Ces serveurs utilisent une stratégie de stockage RAID 0 sur 2 disques. Notez que bien que RAID 0 offre une performance accrue, aucune panne n'est tolérée: les deux disques doivent fonctionner pour que le serveur soit opérationnel. Les disques utilisés ont, individuellement, une probabilité de panne de 0.005.

Le service est considéré en panne si 2 serveurs ou plus ne sont plus utilisables de l'extérieur.

Calculez la disponibilité du service de réservation. Sera-t-il possible de rencontrer l'objectif de disponibilité de l'énoncé? (2 points)

Disponibilité du stockage d'un serveur:
Probabilité Stockage Disponible = (Probabilité Disque Disponible)^2

$$(1 - 0.005)^2 = 0.990025$$

Disponibilité d'un serveur:
*Probabilité Serveur Disponible = Probabilité Stockage Disponible * Probabilité Électronique Disponible*

$$0.990025 * 0.999 = 0.98903498$$

Probabilité que 14 serveurs soient disponibles:

$$(15!/(14! * 1!)) * 0.98903498^{14} * (1 - 0.98903498)^1 = 0.140949643$$

Probabilité que 15 serveurs soient disponibles:

$$(15!/(15! * 0!)) * 0.98903498^{15} * (1 - 0.98903498)^0 = 0.847568379$$

Probabilité qu'au moins 14 serveurs soient disponibles:

$$\textit{Probabilité au moins 14 serveurs} = 0.847568379 + 0.140949643 = 0.988518022$$

Probabilité qu'internet soit disponible:
Probabilité Internet Dispo = 1 - (Probabilité Panne Lien)^2

$$1 - (0.05)2 = 0.9975$$

Probabilité que le service soit disponible:
*Probabilité au moins 14 serveurs * Probabilité Internet Dispo*

$$0.9975 * 0.988518022 = 0.9860467274$$

L'objectif de disponibilité n'est pas rencontré

b) Vous développez un jeu vidéo et visez un taux de rafraîchissement constant de 60 images par seconde. Autrement dit, vous disposez d'environ 16ms (1/60) pour compléter le rendu de chaque image.

Malheureusement, vos tests vous indiquent que vous n'atteignez pas cette cible; le taux de rafraîchissement n'atteignant que 45 images par seconde.

Votre jeu s'exécute sur un système disposant d'un processeur à 4 coeurs et la simulation physique du jeu représente initialement 30% du temps de rendu d'une image.

Assumant que vous ne pouvez pas modifier l'algorithme de simulation de la physique, outre que pour le paralléliser, **quel est le facteur d'accélération minimal requis de cette portion du programme afin d'atteindre le taux de rafraîchissement souhaité? Cette cible est-elle atteignable sur cette plate-forme, et pourquoi? (2 points)**

Atteindre la cible de 60Hz requiert un facteur d'accélération global de $1.33\ (60/45)$.

Le facteur d'accélération minimal requis de la portion simulation physique du jeu est de:

$$60/45 = 1/(0.70 + 0.30/s)$$

Isolant pour s *:* $s = 6$*, soit le facteur minimal d'accélération de la portion simulation physique.*

La cible n'est pas atteignable. Avec un parallélisme parfaitement efficace, on atteindrait seulement un facteur d'accélération de 4 pour cette section (4 coeurs).

c) Les fils d'exécution (threads) d'un même processus partagent généralement un ensemble de données. Le partage de données, et plus généralement les mécanismes d'exclusion mutuelle, sont souvent un obstacle à l'échelonnabilité (scalability) des système multi-coeurs. Il est cependant facile pour un développeur d'identifier ces points de contention que ce soit intuitivement ou par l'utilisation d'outils de profilage.

Cependant, le faux partage (false sharing) est un autre phénomène via lequel divers coeurs peuvent mutuellement se nuire, dégradant ainsi la performance du système.

Décrivez, dans vos mots, le phénomène de faux partage. Comment un développeur peut-il se protéger contre les problèmes de faux partage dans un logiciel comportant plusieurs fils d'exécution?

(1 point)

Le faux partage est un phénomène qui se produit lorsque plusieurs coeurs se retrouvent à accéder, en lecture et en écriture, à des données qui ne sont pas logiquement partagées, mais qui résident dans la/les mêmes lignes/blocs d'antémémoire.

Ces accès provoquent l'invalidation des lignes et leur relecture et réécriture d'un niveau de cache partagé moins performant (ou de la mémoire centrale), ralentissant ainsi les coeurs impliqués dans l'échange.

On peut se prémunir contre ce phénomène en allouant les données des divers threads dans des régions disjointes alignées sur la taille d'une ligne/bloc d'antémémoire.

Question 2 (5 points)

a) Soit un système 32-bits à 4 coeurs dont chaque coeur dispose d'une cache L1 de 32KiB d'associativité à 2 voies (2 blocs par ensemble) dont les blocs (cache lines) sont d'une longueur de 32 octets. La cohérence des caches de ce système est assurée par un circuit de contrôle des caches utilisant le protocole de cohérence MESI.
Donnez l'état MESI correspondant à chaque bloc affectées par les accès suivants:

- Processeur 1, Lecture à l'adresse 0x13CF
- Processeur 1, Écriture de la valeur 42 à l'adresse 0x1ACD
- Processeur 0, Écriture de la valeur 42 à l'adresse 0x1AD3
- Processeur 3, Lecture à l'adresse 0x1AD3

(2 points)

Décomposition des adresses:
Blocs de 32 octets, il faut 5 bits pour exprimer le décalage

$$\textit{Bits de décalage} : [4,0]$$

Les ensembles sont de 2 blocs (64 octets au total), il faut donc 512 ensembles pour couvrir les 32KiB de la cache L1. 9 bits sont nécessaires pour exprimer l'index (numéro d'ensemble).

$$\textit{Bits d'index} : [13,5]$$

Les 18 bits restant représentent l'étiquette: $[31,14]$

Décomposons les trois adresses utilisées dans le problème

```
0x1ACD: [...0] [011010110] [01101]
    Étiquette: 0
    Ensemble: 214 ou 0b01101011
    Décalage: Inutile
0x1AD3: [...0] [011010110] [10011]
    Étiquette: 0
    Ensemble: 214 ou 0b01101011
    Décalage: Inutile
0x13CF: [...0] [010011110] [01111]
    Étiquette: 0
    Ensemble: 158 ou 0b01001111
    Décalage: Inutile
```

Évolution de l'état MESI des blocs

Tous les blocs commencent dans l'état Invalid

Processeur 1, Lecture à l'adresse 0x13CF

- *Processeur 1, Ensemble 158, Bloc 0: I -> E*

Processeur 1, Écriture de la valeur 42 à l'adresse 0x1ACD

- *Processeur 1, Ensemble 158, Bloc 0: E*
- *Processeur 1, Ensemble 214, Bloc 0: I -> M*

Processeur 0, Écriture de la valeur 42 à l'adresse 0x1AD3

- *Processeur 1, Ensemble 158, Bloc 0: E*
- *Processeur 1, Ensemble 214, Bloc 0: M -> I*
- *Processeur 0, Ensemble 214, Bloc 0: I -> M*

Processeur 3, Lecture à l'adresse 0x1AD3

- *Processeur 1, Ensemble 158, Bloc 0: E*
- *Processeur 1, Ensemble 214, Bloc 0: I*
- *Processeur 0, Ensemble 214, Bloc 0: M -> S*
- *Processeur 3, Ensemble 214, Bloc 0: I -> S*

b) Dans le cadre de votre travail, vous devez réviser le code suivant provenant d'un collègue. Celui-ci sera exécuté sur des machines dotées de 16 coeurs physiques exposant chacun 2 coeurs logiques (pour un total de 32 coeurs logiques visibles) ainsi que d'assez d'espace pour contenir l'ensemble de données à traiter en mémoire vive. De plus, la version de GCC disponible permet l'utilisation d'opération atomique: atomic_exchange(obj,value), atomic_fetch_add(obj, value), atomic_fetch_sub(obj, value) etc.

Le résultat du code suivant est-il déterministe pour un même ensemble de données? Expliquez. Si il ne l'est pas, proposez une solution.

Vous ne pouvez vous empêcher de remarquer que l'exécution est lente.

Modifiez *void *tache(void *input)* en optimisant sa performance tout en préservant le déterminisme. Expliquez vos changements.
(2 points)

weeb.c

```
struct Travail {
  uint32_t *donnees_partiel;
  uint32_t taille;
};

uint64_t total_modulo = 0;
uint64_t total_operation = 0;

pthread_mutex_t modulo_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t thread_barrier;

void *tache(void *input) {
  /* Ne pas modifier */
  pthread_barrier_wait(&thread_barrier);
  struct Travail *travail = (struct Travail *)input;
  /* Fin: Ne pas modifier */

  /* .... Travail quelconque ... */
  for (int i = 0; i < travail->taille; i++) {
    pthread_mutex_lock(&modulo_mutex);
    if (travail->donnees_partiel[i] % 10) {
      travail->donnees_partiel[i] =
      travail->donnees_partiel[i] % 10;
      total_modulo++;
    } else {
      travail->donnees_partiel[i] = 11;
    }
    pthread_mutex_unlock(&modulo_mutex);
    total_operation +=2;
  }

  /* .... Travail quelconque ....*/
  pthread_exit(NULL);
}

int main() {
  int nb_thread = 138;
  pthread_t *tid = NULL;
  struct Travail *travail_par_thread = NULL;
  uint32_t *donnees = NULL;

  /* Initialization */
  nb_thread = atoi(argv[1]);
  tid = calloc(sizeof(pthread_t), nb_thread);
  travail_par_thread = calloc(sizeof(struct Travail), nb_thread);
  donnees = calloc(sizeof(uint32_t), TAILLE_DONNEES);
```

```
  pthread_barrier_init(&thread_barrier, NULL, nb_thread);

  /* Acquisition de données */
  /* ... */

  /* Division du travail entre les threads en structure Travail */
  /* ... */

  for (int i = 0; i < nb_thread; i++) {
    pthread_create(&tid[i], NULL, tache,
    (void *)&travail_par_thread[i]);
  }

  for (int i = 0; i < nb_thread; i++) {
    pthread_join(tid[i], NULL);
  }

  printf("Compte modulos %" PRIu64 "\n", total_modulo);
  printf("Compte operations %" PRIu64 "\n", total_operation);

  pthread_barrier_destroy(&thread_barrier);

  /* Nettoyage */
  /* ... */
  return 0;
}
```

Non, l'exécution n'est pas déterministe. Malgré que "total_modulo" soit protégée par modulo_mutex, total_operation, elle, n'est pas à l'abri de modifications concurrentes. Une solution simple ici est de déplacer "pthread_mutex_unlock(&modulo_mutex);" après l'incrément de total_operation

Plusieurs modifications sont possible ici. L'utilisation du mutex à l'intérieur de la boucle représente un coût important de synchronisation. Une solution de base serait l'utilisation de deux variables locales à "tache()" permettant l'accumulation locale lors du parcours de la boucle, suivi de la mise à jour des compteurs globales:

```
uint64_t local_modulo = 0;
uint64_t local_operation = 0;
for (int i = 0; i < travail->taille; i++) {
  if (travail->donnees_partiel[i] % 10) {
    travail->donnees_partiel[i] = travail->donnees_partiel[i] % 10;
    local_modulo++;
  } else {
    travail->donnees_partiel[i] = 11;
  }
  local_operation+=2;
}
pthread_mutex_lock(&modulo_mutex);
```

```
total_modulo += local_modulo;
total_operaton += local_operation;
pthread_mutex_unlock(&modulo_mutex);
```

L'utilisation d'opération atomiques est aussi une bonne option.

```
for (int i = 0; i < travail->taille; i++) {
  if (travail->donnees_partiel[i] % 10) {
    travail->donnees_partiel[i] = travail->donnees_partiel[i] % 10;
    atomic_fetch_add(total_modulo, 1);
  } else {
    travail->donnees_partiel[i] = 11;
  }
  atomic_fetch_add(total_operation, 2);
}
```

*On note aussi que la valeur ajoutée à total_operation peut être calculée une seul fois: total operation += 2 * travaille->taille*
Bien sûr, nous pouvons combiner les solutions précédentes:

```
uint64_t local_modulo = 0;
for (int i = 0; i < travail->taille; i++) {
  if (travail->donnees_partiel[i] % 10) {
    travail->donnees_partiel[i] = travail->donnees_partiel[i] % 10;
    local_modulo++;
  } else {
    travail->donnees_partiel[i] = 11;
  }
}
atomic_fetch_add(total_modulo, local_modulo);
atomic_fetch_add(total_operation, travail->taille * 2);
```

Nous pourrions aussi déplacer la phase "reduce" vers le parent en ajoutant des champs de résultats dans l'objet Travail passé en paramètre. Cependant, ceci va plus loin que la question.

c) **Expliquez, en vos mots, la technologie Hyperthread.**
Êtes-vous en mesure d'affirmer qu'Hyperthread joue un rôle important pour le programme (weeb.c) de la question précédente (Q2.b)? (1 point)

Hyperthread est une technologie physique pour lequel un contexte d'exécution supplémentaire est présent (ensemble de registre) permettant l'entrelacement du flot d'instruction lors d'une faute de cache.

Notons que nb_thread est assigné à 138. Étant donné que les machines exécutant le code possèdent 16 coeurs physiques et 16 logiques, cette valeur pourrait ne pas être optimale en fonction du travail effectué par le reste du code car elle dépasse 32/16.

Dans notre cas, nous ne pouvons affirmer/infirmer si Hyperthread joue un rôle important. Nous ne somme pas au courant du travail complet réalisé par "tache". De plus, nous n'avons aucune données sur les temps d'exécution. Des banc d'essais seraient nécessaires.

Question 3 (5 points)

a) Une image en couleur est représentée par un vecteur de ***taille*** pixels. Chaque pixel est une structure de donnée contenant 3 octets, les champs rouge, vert et bleu chacun représentant l'intensité pour la couleur primaire correspondante (0, 255). On veut compter, pour une image, les pixels pour lesquels le pixel précédent est noir (0,0,0) ainsi que les pixels ayant une composante rouge plus grande que la somme des composantes verte et bleu. Ecrivez un programme, utilisant la librairie TBB, qui offre la fonction:

void compteStatistique(struct Pixel image[], int taille, int *voisin_noir, int* dominant_rouge)

L'argument *image* contient un vecteur de pixels alors que l'argument *taille* contient la taille de ce vecteur. La fonction ***compteStatistique*** doit calculer et retourner dans les arguments correspondants le nombre de pixel ayant un voisin précédent noir ainsi que le nombre de pixel ayant une composante rouge dominante **(3 points)**

```
#include <tbb/tbb.h>

struct Pixel {
  uint8_t rouge;
  uint8_t vert;
  uint8_t bleu;
};

bool estNoir(struct Pixel *pixel) {
  return pixel->rouge == 0 && pixel->vert == 0 && pixel->bleu == 0;
}

struct Statistique {
  int voisin_noir;
  int dominant_rouge;
  struct Pixel *image;
```

```
  // default constructor
  Statistique() {
    voisin_noir = 0;
    dominant_rouge = 0;
  }

  // split constructor
  Statistique(Statistique &stat, tbb::split) {
    image = stat.image;
    voisin_noir = 0;
    dominant_rouge = 0;
  }

  void operator()(const tbb::blocked_range<int> &range) {
    int local_voisin_noir = 0;
    int local_dominant_rouge = 0;

    for (int i = range.begin(); i < range.end(); i++) {
      if (i > 0 && estNoir(&image[i - 1])) {
        local_voisin_noir++;
      }
      if (image[i].rouge > image[i].vert + image[i].bleu) {
        local_dominant_rouge++;
      }
    }

    voisin_noir += local_voisin_noir;
    dominant_rouge += local_dominant_rouge;
    return;
  }

  void join(Statistique &right) {
    voisin_noir += right.voisin_noir;
    dominant_rouge += right.dominant_rouge;
  }
};

void compteStatistique(struct Pixel *image, int taille, int *voisin_noir, int *
  Statistique stat;
  stat.image = image;
  tbb::parallel_reduce(tbb::blocked_range<int>(0, taille), stat);
  *voisin_noir = stat.voisin_noir;
  *dominant_rouge = stat.dominant_rouge;
}
```

b) Lors du premiers laboratoire vous avez accompli la même tâche à l'aide de deux technologies différentes: Intel TBB et Pthread. **Discutez de leur différence en ce qui concerne la répartition et**

gestion du travail dans le cadre du laboratoire. Dans quel contexte choisiriez-vous une approche par rapport à l'autre? Intel TBB et Pthread sont-elles des technologie équivalente? Expliquez. (1 points)

Dans le cadre du laboratoire, la division du travail au niveau de pthread est statique (manuelle) tandis que TBB lui, grâce aux mécanismes interne (thread pool, vol de travail), est dynamique. (0.5 pts)

Pthread se prête bien au parallélisme de tâche tandis que TBB (parallel_*) se prête bien au parallélisme de données. Lorsque le temps de calcul est variable entre les intervalles traité TBB devrait normalement sortir gagnant de par sa nature dynamique. (0.25 pt)

Intel TBB est une abstraction de plus haut niveau construisant sur les fondations offertes par pthread (linux). Ce ne sont pas des technologies équivalentes. Elles sont complémentaires. Intel TBB est une grosse boîte à outils donnant accès à une panoplie d'algorithmes/concepts parallels. (0.25 pt)

c) Lors du premier travail pratique, vous avez calculé le ratio ***(user +sys)/elapsed*** à partir des données dans le répertoire results pour l'exécution de dragonizer. **Que représente chacune de ces trois composantes? Que signifie une valeur égale au nombre de coeur disponible pour ce ratio?**

(1 point)

La partie elapsed est le temps total réel écoulé. La partie user est le temps CPU en mode usager alors que sys est le temps CPU en mode système. Si le processus s'exécute sur un seul coeur, la composante elapsed est plus grande que la somme des deux autres car cela inclut le temps passé en attente. Sur un système multi-coeur, un programme parallèle dans le meilleur cas aura un ratio qui est égal au nombre de coeurs (tous les coeurs travaillent en parallèle sur le problème). Cela signifie qu'aucune attente n'as eu lieu lors de l'exécution lorsque ce ratio est égal au nombre de coeur disponible.

Question 4 (5 points)

a) Un compositeur fait le rendu d'une image à partir d'une image de base et d'un ensemble d'images à y appliquer en couches successives. Par exemple, lorsqu'un gestionnaire de fenêtres effectue le rendu d'une image à afficher, une image de base est utilisée (votre fond d'écran, par exemple) et chaque fenêtre y est par la suite superposé. Considérant que plusieurs fenêtres peuvent se chevaucher, il est important de respecter l'ordre d'application des couches.
Considérons l'implémentation suivante d'un compositeur, assumant que les couches ne débordent jamais de l'image de base.
On vous demande d'accélérer ce compositeur en utilisant OpenMP. **Proposez des modifications qui permettraient d'obtenir le même résultat que l'implémentation fournie, mais en bénéficiant du parallélisme via OpenMP. Expliquez brièvement l'intérêt de chaque modification (1-2 lignes). (2 points)**

area51-naruto-run.cpp:

```
#include <vector>

struct Pixel {
  char r, g, b;
};

struct Image {
  int hauteur, largeur;
  struct Pixel *pixels;
};

struct Couche {
  int x, y;           // Position
  struct Image image; // Contenu de la couche
};

void ComposeImage(struct Image &image, const std::vector<Couche> &couches)
{
  // Application successive des couches
  for (auto couche = couches.begin(); couche < couches.end(); ++couche) {
    int ligne, colonne;

    for (ligne = 0; ligne < couche->image.hauteur; ligne++) {
      int index_image = (image.largeur * (couche->y + ligne)) + couche->x;
      int index_couche = couche->image.largeur * ligne;

      for (colonne = 0; colonne < couche->image.largeur; colonne++) {
        image.pixels[index_image++] = couche->image.pixels[index_couche++];
      }
    }
  }
}
```

On peut annoter les boucles itérant sur ligne et colonne avec #pragma omp parallel for [...]. Il faut bien prendre garde à utiliser la variable colonne plutôt que de simplement incrémenter index_image et index_couche. En effet, index_image et index_couche se retrouveraient partagés entre tous les threads exécutant la boucle sur "colonne". On pourrait aussi déplacer tout le calcul des index au sein de la boucle la plus imbriquée, mais cela serait inefficace.

```
void ComposeImage(struct Image &image, const std::vector<Couche> &couches)
{
  // Application successive des couches
  for (auto couche = couches.begin(); couche < couches.end(); ++couche) {
    int ligne, colonne;

    #pragma omp parallel for
```

```
    for (ligne = 0; ligne < couche->image.hauteur; ligne++) {
      const int index_image = (image.largeur * (couche->y + ligne)) + couche->x
      const int index_couche = couche->image.largeur * ligne;

      #pragma omp parallel for
      for (colonne = 0; colonne < couche->image.largeur; colonne++) {
        image.pixels[index_image + colonne] =
         couche->image.pixels[index_couche + colonne];
      }
    }
  }
}
```

À noter qu'il serait erroné d'annoter la boucle itérant sur les couches puisque celles-ci doivent impérativement être appliquées dans l'ordre exprimé.

On pourrait aussi fusionner les deux boucles, que ce soit en utilisant la clause collapse ou en remaniant le code explicitement.

```
void ComposeImage(struct Image &image, const std::vector<Couche> &couches)
{
  // Application successive des couches
  for (auto couche = couches.begin(); couche < couches.end(); ++couche) {
    int ligne, colonne;

    #pragma omp parallel for collapse(2)
    for (ligne = 0; ligne < couche->image.hauteur; ligne++) {
      const int index_image = (image.largeur * (couche->y + ligne)) + couche->x
      const int index_couche = couche->image.largeur * ligne;

      for (colonne = 0; colonne < couche->image.largeur; colonne++) {
        image.pixels[index_image + colonne =
         couche->image.pixels[index_couche + colonne];
      }
    }
  }
}
```

b) Soit le programme OpenMP suivant, **quelles sont les sorties possibles sur la console? (2 points)**

```
int main(int argc, char **argv)
{
  omp_set_num_threads(3);

  #pragma omp parallel
  {
    printf("%d ", omp_get_thread_num());
  }
  return 0;
}
```

0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0

c) **Complétez** le code du processeur 0 et **ajoutez** les barrières mémoires strictement nécessaires à P0 et P1 qui assureraient que la condition suivante soit vraie suite à l'exécution des opérations suivantes sur deux processeurs assumant un modèle mémoire faible (weak).

$$x == 1 || (y == 42 \;\&\&\; z == 10)$$

Le code de P1 ne doit pas être modifié au-delà de l'ajout de barrières mémoires. Toutes les variables utilisées sont initialement à 0. **(1 point)**

```
P0                P1
                  if (pret) {
                    y = a;
                    z = b;
                  } else {
                    x = 1;
                  }
```

P0

```
a = 42;
b = 10;
cmm_smp_wmb(); // Assure l'ordre des écritures de a et b par rapport à pret
pret = 1;
```

P1

```
if (pret) {
cmm_smp_rmb(); // Ordonner lectures de a et b par rapport à pret
y = a;
z = b;
} else {
x = 1;
}
```

Les chargés de cours: Jérémie Galarneau et Jonathan Rajotte-Julien