ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2018)
3 crédits (3-1.5-4.5)

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Vendredi le 26 octobre 2018

HEURE: 9h30 à 11h20

DUREE: 1H50

NOTE: Toute documentation permise, calculatrice non programmable permise

Ce questionnaire comprend 4 questions pour 20 points

Question 1 (5 points)

a) Lors de votre premier travail pratique, vous avez analysé le comportement d'applications parallèles PThread et TBB. D'après les données obtenues, donnez une courte explication des différences entre PThread et TBB par rapport à leur fonctionnement propre. Combien de fils d'exécution sont démarrés? Comment se fait la répartition du calcul entre les fils d'exécution? Quels sont les appels systèmes impliqués dans la synchronisation des fils d'exécution? **(2 points)**

Avec la librairie PThread, la solution de base utilisée est de créer de nouveaux threads dans une boucle, au nombre des coeurs disponibles (ou le double), et d'avoir le thread principal qui attend la fin de ces threads avec un join. Le travail est divisé statiquement entre les threads.

TBB bâtit, par-dessus les facilités de base pour les threads, des fonctionalités plus avancées. Les threads sont créés dans une arborescence à partir du thread principal, chaque thread créant un second thread, récursivement, tant qu'il y a assez de travail et jusqu'à ce que le nombre maximal (e.g., deux fois le nombre de coeurs disponibles) de threads soit atteint. Par la suite, le travail est distribué dynamiquement entre les threads. Chaque thread commence avec une certaine quantité de travail et va en chercher plus dans une queue, lorsque son travail courant est terminé. Ceci minimise le débalancement entre les threads, qui ferait qu'un thread ayant terminé perdrait son temps à attendre après ceux encore actifs.

Dans les deux cas, divers appels système sont effectués comme sys_clone pour créer les threads, sys_wait pour attendre la fin d'un thread, sys_futex en cas de contention pour la synchronisation avec les mutex et sys_sched_yield pour coordonner les threads.

b) Vous disposez d'une grappe de calcul avec 64 noeuds. Chaque noeud est constitué d'une carte électronique et de deux disques en miroir (en redondance, il suffit que l'un des deux fonctionne). La probabilité de cesser de fonctionner pendant une période de 15 minutes est de .001 pour les cartes et de .01 pour les disques. Deux possibilités s'offrent. On peut avoir des tâches de 15 minutes ou d'une heure (tous les noeuds doivent être parfaitement fonctionnels pendant toute la période choisie, sinon le résultat est perdu). Etant donné le surcoût de mettre en place la tâche et de sauver les résultats à la fin, une tâche de 15 minutes fait 5 fois moins de travail qu'une tâche d'une heure. Quelle option permettra d'effectuer globalement plus de travail en moyenne? **(2 points)**

Les disques sont fonctionnels sauf si les deux sont en panne, une probabilité de $.01 \times .01 = .0001$. *Un noeud sera fonctionnel si la carte et les disques le sont, une probabilité de* $(1 - .001) \times (1 - .0001) = 0.9989001$.

Attention, on pourrait être tenté de simplement additionner les probabilités de panne ($.001 + .0001 = .0011$*), ce qui donne un résultat très près du résultat correct (*$1 - 0.9989001 = 0.0010999$*), mais néanmoins incorrect car il compte deux fois le cas où la carte et le système de disques sont simultanément en panne. L'équation correcte pour additionner les probabilités tient compte de la probabilité combinée, et la soustrait pour compenser le fait qu'elle soit présente deux fois (la carte en panne inclut le cas où le disque est aussi en panne, et le disque en panne inclut aussi le cas où la carte est en panne). L'équation est* $p(AorB) = p(A) + p(B) - p(AandB)$, *soit* $.001 + .0001 - .001 \times .0001 = 0.0010999$, *qui correspond effectivement à la réponse correcte.*

La probabilité que tous les noeuds soient fonctionnels pendant 15 minutes est de $0.9989001^{64} = 0.931990795$. *La probabilité que tous les noeuds soient fonctionnels pendant 4 intervalles de 15 minutes consécutifs est de* $0.931990795^4 = 0.754477845$. *Si le travail accompli pendant un intervalle de 15 minutes est de t, la moyenne sera de 0.931990795t en tenant compte des intervalles où le travail sera perdu. Pour des tâches de 1 heure, le travail accompli sera de 5t mais la moyenne sera de*

$5t \times 0.754477845 = 3.772389223t$ pour 1 heure, soit l'équivalent de $3.772389223t/4 = 0.943097306t$ pour 15 minutes. Les tâches de 1 heure sont donc légèrement plus avantageuses en moyenne.

c) Vous analysez le temps passé dans les différentes fonctions d'un programme de calcul et obtenez les temps suivants: A 100s, B 80s, C 60s, D 120s, E 40s. Le temps passé dans les autres fonctions du programme est négligeable. Vous êtes en mesure de paralléliser les fonctions A, B et D. Quel sera le facteur d'accélération obtenu avec 8 threads roulant sur autant de coeurs? **(1 point)**

Le temps total est de $100+80+60+120+40 = 400s$. La fraction parallélisable est de $(100+80+120)/400 = 0.75$. Le facteur d'accélération sur 8 coeurs avec 0.75 du programme qui est accéléré, selon la loi d'Ahmdal, sera donc de $1/(0.75/8+(1-0.75)) = 2.9$.

Question 2 (5 points)

a) Un ordinateur 32 bits, dont la mémoire est adressable à l'octet, possède 2 coeurs (P0 et P1). Chaque coeur a une mémoire cache de données de 4096 octets avec des blocs de 256 octets et une fonction de correspondance directe. La mémoire cache est initialement vide. Les accès suivants (R pour lecture ou W pour écriture) ont lieu sur les coeurs spécifiés aux adresses données. Comment se décomposent les 32 bits d'adresse en: décalage dans le bloc, numéro de bloc, et étiquette? Quel est le taux de succès en cache pour chacun des 2 coeurs? Quel est l'état (M, O, E, S ou I) de chacun des blocs présents dans les caches à la fin de ces accès? **(2 points)**

```
P0 R 0x111; P1 R x111; P0 R 0x1F0; P0 R 0x222;
P1 W 0x211; P1 R 0x2200; P0 R 0x0F0; P1 W 0x000;
```

Le décalage dans le bloc pour des blocs de $256 = 2^8$ octets est de 8 bits. La mémoire cache contient 4096 octets / 256 octets/bloc = 16 blocs, soit 4 bits pour le numéro de bloc. Ainsi, les 32 bits d'adresse se décomposent, du plus au moins significatif, en 20 bits d'étiquette, 4 bits de numéro de bloc en cache et 8 bits de décalage. Pour les accès donnés, les blocs 0, 1 et 2 en cache (b0, b1, b2) seront affectés. Leur état est montré pour chaque processeur après chaque accès. On indique aussi s'il y a eu une faute de cache pour cet accès (un x en colonne F). On peut voir qu'il y a 4 accès de P0 et 3 fautes de cache, donc un taux de succès de 0.25. Pour P1, on a 4 accès et 4 fautes, pour un taux de succès de 0.

accès	P0				P1			
	F	b0	b1	b2	F	b0	b1	b2
P0 R 0x111	x		E					
P1 R 0x111			S		x		S	
P0 R 0x1F0			S				S	
P0 R 0x222	x		S	E			S	
P1 W 0x211			S	I	x		S	M
P1 R 0x2200			S	I	x		S	E
P0 R 0x0F0	x	E	S	I			S	E
P1 W 0x000		I	S	I	x	M	S	E

b) Un ordinateur 64 bits, dont la mémoire est adressable à l'octet, utilise des tables de pages à 4 niveaux (en fait un arbre de noeuds, chacun occupant une page, à 4 niveaux) et des pages de 4KiO. Les 16 bits les plus significatifs de l'adresse sont inutilisés. Un programme ajoute à son espace adressable

une zone de 12MiO calquée sur un fichier. La table de page doit donc être augmentée pour couvrir ces nouveaux 12MiO. Combien de noeuds devront être ajoutés à l'arbre pour couvrir ces 12MiO au minimum? Au maximum? **(2 points)**

Chaque noeud de 4KiO dans la table de pages contient des entrées de 8 octets (64 bits), soit 4096/8 = 512 entrées. L'espace de 12MiO représente 12MiO / 4KiO = 3x1024 = 3072 pages de mémoire physique. Ceci requiert autant d'entrées qui occupent environ 3072/512= 6 noeuds au niveau L1 dans la table des pages dans le meilleur des cas. Dans le meilleur cas, le noeud parent au niveau L2 était pré-existant et les 6 entrées étaient donc libres. Ceci donne donc un minimum de 6 noeuds à ajouter.

Dans le pire cas, seulement la racine de l'arbre existe pour cette région de mémoire virtuelle, au niveau L4, et les entrées se répartissent au niveau L1 sur 7 noeuds. Les 7 entrées correspondantes pourraient se répartir sur 2 noeuds au niveau L2 et encore 2 noeuds au niveau L3. Le nombre total de noeuds ajoutés serait donc de 7 + 2 + 2 = 11.

c) La section de code suivante s'exécute sur un ordinateur avec ordonnancement faible. Les variables f1, f2, f3, valid1 et valid2 sont initialement à 0. Quelles barrières devrait-on insérer, et où, pour avoir comme seules sorties possibles `0 0 0`, `12 24` ou `12 24 42`?

```
Processeur 0                    Processeur 1

f1=12;                          if(valid1) {
f2=24;                            if(valid2) {
valid1=1;                           printf("%d %d %d\n", f1, f2, f3);
f3=42;                            } else {
valid2=2;                           printf("%d %d\n", f1, f2);
                                } else {
                                  printf("0 0 0\n");
                                }
```

(1 point)

La barrière d'écriture, cmm_smp_wb(), assure que ce qui a été écrit avant sera propagé avant en mémoire. Ainsi, les nouvelles valeurs de f1 et f2 (f3) seront propagées de P0 vers la mémoire centrale avant valid1=1 (valid2=1). De l'autre côté, pour la lecture sur P1, il faut s'assurer, avec une barrière de lecture, cmm_smp_rmb(), que si valid1 (valid2) n'est plus 0, les autres variables, f1 et f2 (f3), auront aussi eu leurs anciennes valeurs invalidées en cache, afin que tout accès obtienne les nouvelles valeurs.

```
Processeur 0                    Processeur 1

f1=12;                          if(valid1) {
f2=24;                            if(valid2) {
cmm_smp_wmb()                       cmm_smp_rmb()
valid1=1;                           printf("%d %d %d\n", f1, f2, f3);
f3=42;                            } else {
cmm_smp_wmb()                       cmm_smp_rmb()
valid2=2;                           printf("%d %d\n", f1, f2);
                                } else {
                                  printf("0 0 0\n");
                                }
```

Question 3 (5 points)

a) Le programme suivant utilise la librairie TBB et est exécuté avec la commande `./qtbb 2014`. Donnez la sortie pour la ou les lignes qui commencent par a). Expliquez le patron des lignes en sortie qui commencent par b). **(2 points)**

```
// programme qtbb.c
#include "tbb/tbb.h"
using namespace tbb;

struct Process {
  float value;
  Process() : value(0) {}
  Process(Process& p, split) { value = 0; }
  void operator()(const blocked_range<float*>& v) {
    float m = 0, *pf;
    for(pf=v.begin(); pf!=v.end(); pf++) m = *pf > m ? *pf : m;
    value = m > value ? m : value;
    printf("b) %f\n", m);
  }
  void join(Process& r) {value =  r.value > value ? r.value : value;}
};

int main(int argc, const char* argv[]) {
  Process p;
  int i, n = atoi(argv[1]);
  float m, *v = new float[n];
  for(i = 0; i < n; i++) v[i] = (float)(i % 256 + i / 256);
  parallel_reduce( blocked_range<float*>(v, v+n), p);
  printf("a) %f\n", p.value);
}
```

Ce programme calcule la valeur maximale dans un vecteur. Le vecteur de 2014 éléments contient 0..255, 1..256, 2..257, 3..258, 4..259, 5..260, 6..261, 7..229. La valeur maximale est donc de 261, ce qui est imprimé à la toute fin: `a) 261.000000`. *Il est à noter que le calcul* `(i % 256 + i / 256)` *est effectué avec des entiers et que les fractions sont perdues avant la conversion en* `(float)` *pour ranger dans le vecteur* `v`. *Les lignes b) sont imprimées pour chaque appel à la fonction* `operator()`. *Chaque appel reçoit un intervalle différent et calcule et imprime la valeur maximale dans cet intervalle. Lors d'un test, le patron suivant a été obtenu. On voit que les intervalles ont une longueur d'environ 8 éléments; chaque thread reçoit donc plusieurs intervalles, au fil du déroulement du calcul, étant donné la répartition dynamique du travail effectuée par TBB.*

```
b) 6.000000
b) 14.000000
b) 22.000000
b) 30.000000
...
```

```
b) 242.000000
b) 250.000000
b) 255.000000
b) 10.000000
b) 18.000000
b) 26.000000
b) 34.000000
b) 42.000000
b) 50.000000
...
```

b) Le programme suivant est utilisé pour estimer la valeur de certains paramètres d'un ordinateur. Deux exécutions sont faites (commande 1 et commande 2) en faisant varier à chaque fois un des paramètres. La commande et les temps d'exécution (temps CPU en mode usager) sont donnés en commentaire. A la lueur de ces résultats, que pouvez-vous conclure sur la taille des blocs en cache? Sur la taille de la mémoire cache (de quel niveau L1, L2 ou L3)? Sur la taille de la mémoire centrale? **(2 points)**

```
# commande 1 et résultat
$ for i in 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26;
    do /usr/bin/time --format="%U" ./cache $i 0 32; done
1.97 1.96 1.96 1.95 1.97 1.96 1.97 1.97 2.06 2.13 2.15 2.15 2.18 2.19 2.19

# commande 2 et résultat
$ for i in 0 1 2 3 4 5 6 7 8 9 10;
    do /usr/bin/time --format="%U" ./cache 24 $i 32; done
1.92 2.48 4.15 8.24 17.61 25.36 32.61 35.63 38.31 41.90 38.33

# cache.c:
int main(int argc, char **argv)
{ uint64_t i, j, inc, sum = 0, one = 1;
  static uint32_t *data;
  uint64_t size = one<<atoi(argv[1]),
    stride = one<<atoi(argv[2]), iter = one<<atoi(argv[3]);

  data = malloc(size * sizeof(uint32_t));
  for(i = 0; i < size; i++) data[i] = i ^ 16777215;

  for(i = 0 ; i < iter; i += size / stride)
    for(j = 0; j < size; j += stride) sum += data[j];
  return(sum);
}
```

Un vecteur est accédé à répétition un nombre constant de fois. Avec la commande 1, la taille du vecteur (donnée en puissance de 2 du nombre d'éléments de 4 octets) est variée de $2^{12} \times 4 = 16KiO$ *à* $2^{26} \times 4 = 256MiO$*. On remarque que le temps reste plutôt constant, jusqu'à une taille de* $2^{20} \times 4 = 4MiO$ *et monte légèrement pour les tailles suivantes. Ceci indique que la mémoire cache a une taille d'environ 4MiO. L'augmentation du temps d'exécution demeure modeste car l'accès au vecteur n'est*

qu'un élément de l'exécution de la boucle. Le saut conditionnel associé à la boucle peut causer un délai assez long pendant lequel une faute de cache peut se résoudre en parallèle. On ne peut savoir facilement s'il s'agit de la taille de la mémoire cache L1, L2 ou L3. Toutefois, le saut le plus important est probablement entre la cache L3 et la mémoire centrale. De plus, une taille de 4MiO est de l'ordre de grandeur d'une mémoire cache L3. Les tests ne permettent pas de déduire la taille de la mémoire centrale, outre le fait qu'elle est supérieure à 256MiO. En effet, la taille de vecteur testée n'est pas assez grande et le saut, entre le temps d'accès en mémoire centrale et celui pour le débordement de mémoire virtuelle sur disque, causerait une augmentation bien plus considérable que ce qui est vu ici.

La commande 2 permet de déduire la taille des blocs de cache. En effet, le temps double avec le doublement du pas (stride) des accès, jusqu'à un pas de $2^4 \times 4 = 64$ octets. Par la suite le temps augmente beaucoup moins. Ceci s'explique par le fait que, tant que le pas est inférieur à la taille d'un bloc de cache, chaque augmentation diminue le nombre d'accès dans le même bloc en cache et augmente le nombre de fautes d'autant. Par la suite, lorsque le pas est supérieur à la taille d'un bloc, chaque accès cause une faute en cache dans un bloc différent et le temps change moins.

Voici un extrait de l'information sur le processeur testé, donnée dans `/proc/cpuinfo`, qui confirme ces déductions:

```
processor : 0
model name : Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz
cpu MHz : 1541.866
cache size : 4096 KB
physical id : 0
siblings : 4
core id : 0
cpu cores : 2
cache_alignment : 64
```

c) Vous possédez un ordinateur à 4 coeurs. Vous programmez une application de calcul parallèle avec la librairie pthread. Combien de thread devriez-vous utiliser pour paralléliser le calcul?

(1 point)

Il faut utiliser un petit multiple (habituellement 1 ou 2) du nombre de coeurs, soit 4 ou 8 threads, pour utiliser tous les coeurs, avoir une charge équilibrée entre les coeurs, et éviter d'avoir trop de threads par coeur et perdre du temps dans les changements de contexte. Il peut être utile d'avoir 2 threads par coeur si un thread peut bloquer sur un accès aux disques par exemple, permettant à l'autre thread d'utiliser le coeur, qui serait autrement inutilisé pendant ce temps.

Question 4 (5 points)

a) Le programme OpenMP suivant est exécuté. Donnez les lignes qui seront générées en sortie. **(2 points)**

```
int main(int argc, char **argv)
{ int nb_thread, thread, p, s;
  omp_set_num_threads(2);
  #pragma omp parallel private(thread)
```

```
  { printf("a)\n");
    for(int i = 0; i < 4; i++) {
      nb_thread = omp_get_num_threads();
      thread = omp_get_thread_num();
      printf("b) thread %d / %d, i = %d\n", thread, nb_thread, i);
    }
    printf("c) thread %d / %d\n", thread, nb_thread);
    #pragma omp for schedule(static,2)
    for(int i = 0; i < 4; i++) {
      nb_thread = omp_get_num_threads();
      thread = omp_get_thread_num();
      printf("d) thread %d / %d, i = %d\n", thread, nb_thread, i);
    }
  }
}
```

Ce programme génère la sortie suivante. L'ordre entre les sorties produites par les différents threads, entre deux barrières de rendez-vous, peut changer d'une exécution à l'autre.

```
a)
b) thread 0 / 2, i = 0
b) thread 0 / 2, i = 1
b) thread 0 / 2, i = 2
b) thread 0 / 2, i = 3
c) thread 0 / 2
d) thread 0 / 2, i = 0
d) thread 0 / 2, i = 1
a)
b) thread 1 / 2, i = 0
b) thread 1 / 2, i = 1
b) thread 1 / 2, i = 2
b) thread 1 / 2, i = 3
c) thread 1 / 2
d) thread 1 / 2, i = 2
d) thread 1 / 2, i = 3
```

b) Le programe suivant s'exécute sur un ordinateur avec 2 coeurs physiques hyperthread qui donnent ainsi 4 coeurs logiques. La variable s est un vecteur d'entiers 32 bits de 64 entrées. La commande indiquée en commentaire a été utilisée pour rouler ce programme avec un nombre croissant de threads de 1 à 32. Le temps d'exécution de ce programme (temps réel écoulé) imprimé en sortie est fourni à la suite de la commande en commentaire. Comment expliquez-vous cette évolution du temps d'exécution en fonction du nombre de threads, et en particulier que 2 threads parallèles prennent plus de temps que 1 thread? Est-ce que ce programme comporte un bogue de performance? Comment pourrait-on le corriger? **(2 points)**

```
// $ for i in 1 2 4 8 16 32; do /usr/bin/time --format="%E" ./qomp2018-b $i; do
// 10.44 13.66 25.81 27.71 22.38 8.13
```

```
int main(int argc, char **argv)
{ uint32_t s[64], pos;
  omp_set_num_threads(atoi(argv[1]));
  #pragma omp parallel private(pos) shared(s)
  { pos = omp_get_thread_num();
    #pragma omp for
    for(int i = 0; i < 2000000000; i++) {
      s[pos] += (i * pos) % (pos + 1);
    }
  }
}
```

Avec 2 threads, les variables s[0] et s[1] sont dans le même bloc de cache et cela cause un faux partage. A chaque tour de boucle, chaque thread cause une faute de cache et une invalidation du bloc partagé sur la cache de l'autre coeur. Il y a donc un bogue de performance. Pour le corriger, il faut que chaque variable utilisée par un thread différent soit dans un bloc différent en cache. On pourrait aligner chaque entrée dans le vecteur sur une ligne de cache ou avoir une variable privée pour faire le calcul et la reporter dans le vecteur à la fin de la boucle. Il est aussi possible de faire une réduction mais ceci représente un travail inutile par rapport à une variable privée. En effet, il n'y a pas d'entrée partagée entre les threads, à réduire par sommation à la fin, il faut simplement s'assurer que les entrées respectives soient dans des blocs en cache différents.

Avec 4 threads, le problème est exacerbé. Avec 8 threads, on voit que le temps change peu. Pourtant, les 8 threads accèdent des variables qui sont dans le même bloc en cache, si les blocs ont une taille de 64 octets soit 16 mots. L'explication vient du fait que cet ordinateur possède 4 coeurs et donc 4 threads qui roulent vraiment en même temps, et qui s'échangent les coeurs avec les 4 autres threads de temps en temps (e.g. tous les 20ms). A 16 threads, il y a plus de changements de contexte, ce qui laisse pendant cet intervalle les threads sur les autres coeurs moins en contention. Ceci pourrait expliquer que le temps diminue. Finalement, à 32 threads, on se retrouve avec deux blocs différents en contention entre les threads. Ceci, couplé aux changements de threads encore plus nombreux, peut expliquer que le temps soit encore un peu amélioré mais à peine mieux qu'avec 1 seul thread.

c) Une variable est accédée en écriture par plusieurs threads en parallèle et doit être protégée. Deux versions de code sont proposées afin d'éviter la corruption de cette variable. Est-ce que les deux versions sont correctes? Laquelle sera la plus efficace? **(1 point)**

```
// Version 1
#pragma omp critical
global_sum += local_sum;
// Version 2
#pragma omp atomic update
global_sum += local_sum;
```

Les deux versions sont correctes. Les processeurs offrent souvent nativement des instructions atomiques pour les opérations simples comme l'addition. Ceci est donc plus efficace qu'une section critique qui demande de prendre un verrou. La prise de verrou demande quelques opérations (dont une

opération atomique) et cause de la contention sur la variable utilisée pour le verrou, en plus de la contention en cache pour la variable à incrémenter.

Le professeur: Michel Dagenais