ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2014)
3 crédits (3-1.5-4.5)

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Lundi le 3 novembre 2014

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) Une entreprise offre un service d'espace disque à distance aux usagers, AieNuage. L'entreprise veut offrir un service fiable mais ne veut pas assumer les coûts de prendre des copies de sécurité sur ruban. Elle utilise donc un système avec un grand nombre de noeuds avec redondance. Chaque noeud est constitué d'une unité de traitement et d'un disque de 1TB, pouvant accueillir les fichiers de 100 usagers. Les fichiers de chaque usager sont stockés en permanence sur 3 noeuds différents, n'importe lequel des 3 pouvant alors servir les requêtes pour cet usager. Si une unité de traitement est en panne, le noeud est indisponible temporairement. Si un disque est en panne, les données de cette copie sont irrémédiablement perdues. La probabilité qu'une unité de traitement soit en panne est de 0.001. La probabilité qu'un disque soit en panne est de .002. Quelle est la probabilité que les fichiers d'un usager à un moment donné soient indisponibles (perdus ou non)? Irrémédiablement perdus? Si le service compte 2 milliards d'usagers, combien ont perdu leurs fichiers irrémédiablement à un instant donné? **(2 points)**

Les fichiers d'un usager sont sur 3 noeuds spéficiques. Si les 3 disques correspondants sont en panne simultanément, les fichiers sont irrémédiablement perdus, soit une probabilité de $.002^3 = 8 \times 10^{-9}$*, ou* $10^{-9} \times 2000000000 = 16$ *usagers. Pour que les fichiers d'un usager soient disponibles sur un noeud, il faut que l'unité de traitement et le disque soient fonctionnels, une probabilité de* $(1 - .002)(1 - .001) = 0.997002$*. Pour que les fichiers soient non disponibles (temporairement ou irrémédiablement perdus), il faut que les 3 noeuds soient non disponibles* $(1 - 0.997002)^3 = 26.946 \times 10^{-9}$*.*

b) Sur les processeurs Intel, il est possible de varier la taille des pages en mémoire virtuelle, et d'utiliser soit des petites pages de 4Kio ou des grandes pages de 4Mio. Quel est l'impact de ce choix sur le taux de succès pour la cache de pré-traduction d'adresse (TLB)? Le taux de succès en mémoire cache? L'utilisation efficace de la mémoire physique? **(1 point)**

Les grandes pages demandent beaucoup moins d'entrées dans le TLB et améliorent donc beaucoup le taux de succès du TLB si elles sont le moindrement bien utilisées en étant assez pleines. La taille des pages a peu ou pas d'impact sur le taux de succès en mémoire cache puisque les blocs de cache sont beaucoup plus petits que même les petites pages. Les grandes pages peuvent causer une plus grande perte d'espace par fragmentation puisqu'on arrondit chaque segment d'un processus à la frontière de taille de page (4Mio) supérieure, et donc causer une utilisation moins efficace de la mémoire physique.

c) Expliquez ce qui peut causer un faux partage de données en cache. Quel en est l'impact? Comment peut-on se débarrasser d'un tel problème? Donnez un exemple d'allocateur en TBB qui permet d'allouer une donnée qui ne présentera pas de tel problème. **(1 point)**

Lorsque par malchance deux variables différentes se retrouvent dans la même ligne de cache, et que ces deux variables sont utilisées par deux fils d'exécution parallèles qui

exécutent sur deux coeurs avec caches séparées, ceci cause un conflit sur cette ligne de cache en raison du faux partage. En utilisant le cache_aligned_allocator de TBB, on peut s'assurer que deux objets ne seront pas alloués dans la même ligne de cache, puisque chacun sera aligné au début d'une ligne de cache. Ceci cause un peu de perte d'espace mémoire mais empêche le faux partage.

d) Vous venez de paralléliser une fonction F d'un programme qui était sériel. La nouvelle version de F peut ainsi utiliser efficacement les 64 processeurs de votre ordinateur. Les autres fonctions sont inchangées. Le programme s'exécute maintenant 10 fois plus vite au total. Quelle fraction du temps prenait initialement la fonction F? **(1 point)**

Soit fp la fraction parallélisable. Le programme prenait un temps t et prends maintenant $t/10 = t \times ((1 - fp) + fp/64)$ donc $1/10 = 1 - fp \times 63/64$ ou $0.9 \times 64/63 = fp$ donc $fp = 0.9143$.

Question 2 (5 points)

a) Un très grand vecteur v contient des entiers signés qui représentent les entrées et les sorties de participants pendant *la fête de la plage* à Polytechnique. Par exemple, +1 indique une entrée et -2 la sortie de deux personnes. Sachant que la salle était initialement vide, il faut calculer le vecteur t qui contient le nombre de participants dans la salle après avoir traité la donnée du vecteur v à la position correspondante. On veut effectuer ce calcul avec la librairie TBB. Quelle structure (parallel for, parallel reduce, parallel scan ou parallel do) suggérez-vous d'utiliser? Ecrivez le coeur de ce programme, comme dans les exemples utilisés pour les dispositives du cours expliquant TBB. **(2 points)**

Il faut avoir calculé la somme de toutes les entrées et sorties afin d'avoir le nombre de participants présents à un instant donné. Ceci peut se faire efficacement avec un parallel scan qui permet de calculer en parallèle la variation pour différentes sections du vecteur v et ensuite de combiner cette information pour calculer, dans une deuxième passe, de nouveau en parallèle, le nombre total de participants.

```
class Body {
  T sum; T* const t; const T* const v;

  Body( T t_[], const T v_[] ) : sum(0), v(v_), t(t_) {}

  T get_sum() const {return sum;}

  template<typename Tag>
  void operator()( const blocked_range<int>& r, Tag ) {
    T temp = sum;
    for( int i=r.begin(); i<r.end(); ++i ) {
      temp = temp + v[i];
```

```
      if( Tag::is_final_scan() ) t[i] = temp;
    }
    sum = temp;
  }

  Body( Body& b, split ) : v(b.v), t(b.t), sum(0) {}

  void reverse_join( Body& a ) { sum = a.sum + sum;}

  void assign( Body& b ) {sum = b.sum;}
};

float DoParallelScan( T t[], const T v[], int n ) {
  Body body(t,v);
  parallel_scan(blocked_range<int>(0,n), body );
  return body.get_sum();
}
```

b) Dans le premier travail pratique, vous avez examiné et comparé le travail de PThread et TBB. Quels sont les appels système impliqués dans la synchronisation des fils d'exécution de PThread et TBB? **(1 point)**

Avec PThread, les nouveaux fils d'exécution sont créés avec l'appel système sys_clone. Par la suite, la synchronisation des fils d'exécution, par exemple avec des mutex ou pthread_join, se fait avec sys_futex lorsqu'une attente est nécessaire. La situation avec TBB est assez semblable. La principale différence est que le fil original participe au calcul et que la création de nouveaux fils se fait récursivement, en arbre. Il est amusant de noter que TBB fait de nombreux appels système sched_yield. TBB utilise probablement une approche graduelle pour l'acquisition d'un mutex ou d'un sémaphore: quelques essais avec une boucle active, quelques essais avec une boucle de sched_yield et finalement un appel aux mutex du système d'exploitation avec futex. Sur Linux, il est douteux que les appels à sched_yield soient très utiles.

c) Un ordinateur 64 bits fonctionne avec une table de pages à 4 niveaux. Seul un bit de son espace adressable n'est pas utilisé, laissant 63 bits pour ses adresses virtuelles. Toutes les pages ont la même taille et sont utilisées autant pour peupler l'espace virtuel des processus que pour contenir les différents niveaux de la table de page. Quelle est la taille d'une page? **(1 point)**

Une page de 2^n octets prend n bits pour le décalage dans la page et permet de stocker $2^n/8$ entrées de 64 bits (8 octets) dans un morceau de la table de pages. Pour choisir parmi ces $2^n/8$ entrées, il faut $n-3$ bits. L'adresse se décompose en 1 bit inutilisé, n bits pour le décalage et 4 champs de n - 3 bits (un par niveau de la table de pages). Nous avons donc $64 = 1 + n + 4 \times (n-3) = 5n - 13$ donc $64 - 1 + 12 = 5n$ et $n = 15$. La taille d'une page doit donc être de 2^{15} octets ou 32Kio.

d) Un processeur 64 bits possède une cache L1 de 256Kio avec des blocs de 256 octets. La fonction de correspondance est directe-associative avec un degré d'associativité de 4, ce qui donne des ensembles de 4 blocs chacun. Si on vous donne l'adresse suivante en hexadécimal `0x000000000A0B0C0D`. Donnez le numéro d'ensemble en cache, le décalage dans le bloc et l'étiquette associés à cette adresse. **(1 point)**

Cette adresse peut être exprimée en binaire ainsi: 0000 0000 0000 0000 0000 0000 0000 0000 0000 1010 0000 1011 0000 1100 0000 1101. Le décalage dans le bloc est ce chiffre modulo 256, soit les 8 derniers bits ou 0000 1101 (0x0D). Un ensemble contient $4 \times 256 = 1024$ *octets. La cache contient donc* 256×1024 *octets ou 256 ensembles de 1024 octets. Le numéro d'ensemble en cache est donné par le modulo 256 du reste de l'adresse, soit les 8 bits qui précèdent les 8 derniers ou 0000 1100 (0x0C). L'étiquette est ce qui distingue deux blocs différents qui peuvent se retrouver en cache dans le même ensemble, soit le reste de l'adresse, ou 0x000000000A0B.*

Question 3 (5 points)

a) Un des modèles étudiés de cohérence en mémoire partagée est l'ordre partiel des écritures (PSO). Expliquez lesquels réordonnancements des accès mémoire (lecture ou écriture versus lecture ou écriture) sont possibles? Donnez un exemple de structure matérielle qui pourrait expliquer de tels réordonnancements. **(2 points)**

Avec PSO, les écritures peuvent non seulement être retardées par rapport aux lectures mais l'ordre des écritures n'est pas respecté. Les écritures sont typiquement retardées par rapport aux lectures en présence de queues d'écritures qui permettent de ne pas bloquer et de laisser les écritures se faire en différé. Lorsqu'on a des bancs de mémoire cache parallèles, on peut utiliser une queue d'écriture par banc et il peut alors arriver qu'une queue soit plus pleine que l'autre. Certaines écritures seraient alors retardées par rapport à d'autres qui ont été faites en même temps ou même un peu avant.

b) La section de code suivante s'exécute sur un ordinateur avec ordonnancement faible. Quelles sont les sorties possibles produites par printf? Quelles barrières devrait-on insérer, et où, pour avoir comme seules sorties possibles `1 2 3` ou `0 0 0`? **(2 points)**

```
(initialement valid, a, b et c sont 0)
Processeur 0                    Processeur 1

a=1;                            if(valid) printf("%d %d %d\n", a, b, c);
b=2;                            else printf("0 0 0\n");
c=3;
valid=1;
```

Initialement, tout est possible, la mise à jour de chacune des valeurs a, b et c peut ou non avoir atteint le processeur 1. Il est donc possible d'avoir 0 0 0, 0 0 3, 0 2 0, 0 2 3, 1 0 0, 1 0 3, 1 2 0, 1 2 3. En mettant une barrière d'écriture avant valid = 1, nous

sommes certains que la mémoire centrale verra les écritures de a, b et c avant de voir celle de valid. A l'inverse, en mettant une barrière de lecture après la lecture de valid, nous sommes certains que toute mise à jour retardée de a, b et c sera prises en compte. A ce moment, soit que valid est à 0 et que les valeurs de a, b et c peuvent ou non être en train de se faire mettre à jour, soit que valid est à 1 et a, b et c ont été mises à jour et ces mises à jour sont rendues au processeur 1.

```
Processeur 0                    Processeur 1

a=1;                            if(valid) {
b=2;                              smp_rmb();
c=3;                              printf("%d %d %d\n", a, b, c);
smp_wmb();                      }
valid=1;                        else printf("0 0 0\n");
```

c) Vous devez programmer une application avec plusieurs fils d'exécution de manière à maximiser l'utilisation de toutes les ressources disponibles sur le serveur (processeurs, disques...). Chaque fil traite une requête à la fois qui peut nécessiter du calcul, des accès réseau et des accès disque. Puisque ce logiciel sera installé sur une grande variété de serveurs, vous ne pouvez savoir à l'avance quel sera le nombre de processeurs ou de disques, ni la fraction du temps passée à attendre après les requêtes réseau. Par ailleurs, il faut éviter d'avoir beaucoup trop de fils d'exécution, puisque cela encombre les queues du système d'exploitation, et il ne faut pas non plus continuellement créer et détruire des fils d'exécution, puisqu'il y a un coût non négligeable associé à cela. Proposez une organisation efficace pour décider combien de fils d'exécution créer et pour gérer l'utilisation et l'évolution de ces fils d'exécution. **(1 point)**

Le plus simple est de commencer avec un nombre de fils d'exécution qui est le double du nombre de processeurs. Ensuite, à chaque nouvelle requête reçue, on vérifie la charge du système. Si le CPU ou les disques sont surchargés, (longue queue d'attente, load average beaucoup supérieur au nombre de processeurs), on peut mettre la requête en queue ou même la refuser puisqu'ajouter des fils d'exécution n'aidera en rien. Si la charge n'est pas très forte, on peut refiler la requête à un fil d'exécution inactif (fil en attente sur la queue des requêtes à traiter) ou on peut créer un nouveau fil, s'il n'en reste plus de libre. Lorsqu'un fil termine le traitement d'une requête, on peut le mettre en attente, ou il peut être détruit si un grand nombre de fils disponibles sont déjà en attente.

Question 4 (5 points)

a) Le programme suivant calcule la distribution des couleurs dans une grande image. Convertissez ce programme sériel en programme OpenMP efficace. **(2 points)**

```
void ColorDistribution(unsigned char image[2048][2048],
```

```
    unsigned int d[256])
{
  for(int i = 0; i < 2048; i++)
    for(int j = 0; j < 2048; j++) d[image[i][j]]++;
}
```

On serait tenté de tout simplement convertir la première boucle en parallel for. Toutefois, l'histogramme stocké dans le vecteur d est partagé pour les différentes itérations exécutées en parallèle et il en résulterait de la corruption. Une opération atomique permettrait de le mettre à jour correctement mais serait plutôt inefficace.

```
void ColorDistribution(unsigned char image[2048][2048],
    unsigned int d[256])
{
  int i, j;

  # pragma omp parallel for private(i,j)
  for(i = 0; i < 2048; i++) {
    for(j = 0; j < 2048; j++) {
      #pragma omp atomic
      (d[image[i][j]])++;
    }
  }
}
```

Une manière efficace d'obtenir le bon résultat est de faire les sommes dans des copies privées, en ne faisant les sommes dans le vecteur global qu'à la fin du travail. Ici on crée un vecteur local pour chaque fil, et son contenu est copié dans le vecteur global à la fin.

```
void ColorDistribution(unsigned char image[2048][2048],
    unsigned int d[256])
{
  int i, j;

  # pragma omp parallel private(i,j)
  { unsigned int *d_private = new unsigned int[256];
    for(i = 0; i < 256; i++) d_private[i] = 0;

    # pragma omp for
    for(i = 0; i < 2048; i++) {
      for(j = 0; j < 2048; j++) {
        (d_private[image[i][j]])++;
      }
```

```
    }

    #pragma omp critical
    for(i = 0; i < 256; i++) d[i] += d_private[i];
    delete d_private;
  }
}
```

b) Lorsqu'une boucle parallel for est rencontrée en OpenMP, la librairie de support à l'exécution doit répartir les itérations entre les différents fils d'exécution. En supposant que, dans le court programme montré ici, la valeur de `img->width` est de 1024, que le nombre de processeurs disponibles est 8, et que la durée de la fonction f1 est assez variable, expliquez comment le travail pourrait être réparti efficacement sur plusieurs fils d'exécution par OpenMP, en spécifiant bien quel fil traitera quelles itérations à quel moment. **(2 points)**

```
int encode(struct image *img)
{
    int i, j;

    #pragma omp parallel for private(i,j)
    for (i = 0; i < img->width; i++) {
        for (j = 0; j < img->height; j++) {
            img->data[i][j] = f1(img->data[i][j]);
        }
    }
}
```

Lorsqu'on ne spécifie rien, OpenMP va typiquement allouer deux fois plus de fils que le nombre de processeurs et utiliser une stratégie guidée pour répartir les itérations. Il pourrait donc partir 16 fils et donner à chacun un morceau assez gros pour commencer, par exemple 32 itérations. Le fil 0 traiterait i de 0 à 31, le fil 1 de 32 à 63... et le fil 15 de 480 à 511. Au fur et à mesure que les fils terminent, ils peuvent obtenir un second morceau, plus petit, par exemple de 16. A chaque fois, la taille choisie est proportionnelle au nombre d'itérations restantes fois le nombre de fils d'exécution.

c) Dans une boucle OpenMP, vous avez besoin de calculer une somme globale. Vous pourriez prendre une variable globale protégée par un verrou (section critique ou mutex), déclarer une variable comme une réduction pour la boucle, ou utiliser une opération d'incrément atomique sur une variable globale. Qu'est-ce qui sera le plus performant? Pourquoi? **(1 point)**

Une opération atomique sera généralement plus efficace qu'un mutex puisque la prise et le relâchement de mutex sont eux-mêmes réalisés à partir d'une opération atomique.

Par contre, la réduction sera beaucoup plus efficace puisque le gros des opérations sera effectué sur une copie locale de la somme.

Le professeur: Michel Dagenais