ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2017)
3 crédits (3-1.5-4.5)

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Vendredi le 27 octobre 2017

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) Trois serveurs de données redondants, chacun contenant une unité de disque RAID de 5 disques, sont utilisés pour alimenter une grappe de calcul. Chaque serveur a une probabilité de panne de 0.1 pour son électronique, de 0.05 pour l'électronique de l'unité de disque RAID et de 0.15 pour chaque disque dans les unités de disque RAID. Un serveur fonctionne en autant que son électronique, l'électronique de l'unité de disque RAID et au moins 4 disques sur les 5 sont opérationnels. Quelle est la probabilité globale de panne pour 1 serveur? Pour les 3 serveurs simultanément? **(2 points)**

Une unité de disque RAID est fonctionnelle si 4 ou 5 disques sont fonctionnels. Pour 5 disques fonctionnels: $\frac{5!}{(5!\times 0!)} \times (1-0.15)^5 \times 0.15^0 = 0.443705313$. *Pour exactement 4 disques fonctionnels:* $\frac{5!}{(4!\times 1!)} \times (1-0.15)^4 \times 0.15^1 = 0.391504688$. *Le total est donc pour 4 ou 5 disques de:* $0.443705313 + 0.391504688 = 0.835210001$. *Un serveur est fonctionnel si ses 3 composantes sont fonctionnelles (électronique, électronique de l'unité RAID, disques) simultanément:* $(1-0.1) \times (1-0.05) \times 0.835210001 = 0.71410455$. *La probabilité de panne pour 1 serveur est donc de:* $(1-0.71410455) = 0.28589545$. *La probabilité que les 3 serveurs soient en panne simultanément est de:* $0.28589545^3 = 0.02336801$

b) Lors du premier travail pratique, vous avez calculé le ratio $(user + sys)/elapsed$ à partir des chiffres dans le répertoire *results* pour l'exécution de *dragonizer*. Que représente chacune de ces trois composantes? Que signifie une valeur plus faible 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 roule 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). Si ce ratio est plus faible, une partie du temps a été passé en attente, par exemple pendant que certains coeurs sont inutilisés en raison d'une mauvaise division du travail ou d'une partie de l'algorithme qui est sérielle.

c) Si vous aviez à paralléliser le calcul pour une fractale dont le temps de calcul par unité de surface est très variable (contrairement à la fractale du dragon, dont le temps de calcul est proportionnel à la taille des intervalles), quelle librairie utiliseriez-vous entre *pthreads* et *TBB*? Pourquoi? **(1 point)**

Lorsque le temps est très variable, il faut avoir une stratégie dynamique pour répartir le travail, ce qui est facilement disponible avec TBB. Il serait possible de réaliser la même chose avec pthreads mais cela demanderait de réimplémenter des mécanismes dynamiques similaires à ce qui existe déjà dans TBB.

d) Tous les thread d'exécution d'un même processus partagent le même espace mémoire. Dans ce contexte, comment le système peut-il fournir à chaque thread sa propre pile et ses propres variables spécifiques (*Thread Local Storage*)? **(1 point)**

Une portion de l'espace adressable du processus est réservée pour la pile de chaque thread. Cet espace serait accessible de chaque thread par des moyens indirects mais ils n'ont normalement qu'un pointeur vers leur propre pile, le registre pointeur de pile, qui est spécifique à chaque coeur. Il existe différents mécanismes pour avoir de l'espace pour des variables spécifiques à chaque thread autres que les variables locales, par exemple un vecteur dont les éléments sont indexés par le numéro de thread. Souvent, le plus simple est d'avoir un espace réservé à chaque thread avec un registre spécifique à chaque coeur qui y accède, tout comme pour la pile.

Question 2 (5 points)

a) Un ordinateur multi-coeur 32 bits, dont la mémoire est adressable à l'octet, possède une cache de données de 32Kio, avec des ensembles de 4 blocs, pour chacun de ses 8 coeurs. Chaque bloc en cache contient 64 octets. Cette cache est initialement vide. Comment se décomposent les 32 bits d'adresse en décalage dans le bloc, numéro d'ensemble et étiquette? Les adresses suivantes sont accédées en séquence, en lecture (R) ou en écriture (W), sur les processeurs spécifiés (P0 à P7). Donnez l'état des cases non vides (M, E, S ou I) après chaque accès. **(2 points)**

```
P0 W 0x22B 7;P3 W 0x22B 5;P7 W 0x22B 9;P1 R 0x22B;P2 R 0x33B;
```

Pour des blocs de 64 octets, les 6 derniers bits d'adresse représentent le décalage dans le bloc. La cache contient $2^{15}/2^6 = 2^9 = 512$ *blocs de 64 octets soit* $512/4 = 128$ *ensembles de 4 blocs. Il faut donc 7 bits pour représenter l'ensemble en cache et le reste,* $32-6-7 = 19$ *constitue l'étiquette du bloc (qui distingue un bloc des autres qui peuvent se retrouver dans le même ensemble). Pour l'adresse 0x022B, cela donne 0b0000 0010 0010 1011 ou 0b000|0 0010 00|10 1011, soit étiquette 0, ensemble 0x08, décalage 0x2B. Pour l'adresse 0x033B, cela donne 0b0000 0011 0011 1011 ou 0b000|0 0011 00|11 1011, soit étiquette 0, ensemble 0xC, décalage 0x3B. Initialement, tous les blocs en cache ont le statut invalide (I). La première opération fait charger un bloc dans la cache de P0 (premier bloc de l'ensemble 8, étiquette 0, statut M). La seconde opération invalide le bloc de la cache de P0 (qui sera réécrit puisque modifié) et le charge dans la cache de P3 (premier bloc de l'ensemble 8, étiquette 0, statut M). La troisième opération concerne toujours le même bloc et invalide le bloc de la cache de P3 (qui sera réécrit puisque modifié) et le charge dans la cache de P7 (premier bloc de l'ensemble 8, étiquette 0, statut M). Ce même bloc est ensuite lu par P1, il est réécrit par P7 puisque modifié et son statut devient S avant d'être chargé par P1 (premier bloc de l'ensemble 8, étiquette 0, statut S). Finalement, un bloc différent est chargé dans la cache de P2 (premier bloc de l'ensemble 0xC, étiquette 0, statut E ou S).*

b) Un ordinateur multi-coeur 64 bits, dont la mémoire est adressable à l'octet, utilise des tables de pages à 3 niveaux et des pages de 16Kio. Chaque noeud de la table de pages entre dans une page. Montrez comment l'adresse se décompose en décalage dans la page, index pour chacun des 3 niveaux dans la table, et s'il y a lieu les bits inutilisés. Décomposez l'adresse virtuelle 0x0123456789ABCDEF en ces différentes composantes (décalage, index0, index1...). Décrivez l'information typique contenue dans chaque entrée de 64 bits de la table de page. **(2 points)**

Une page de 16Kio (2^{14}*) requiert 14 bits pour le décalage et peut contenir 2048 entrées de 8 octets (64 bits) lorsqu'utilisée comme noeud dans la table de pages. L'index dans un noeud de la table de pages requiert donc 11 bits pour indexer ces 2048 entrées. Le reste des bits* $64-14-3\times11 = 17$ *est inutilisé. Ces bits inutilisés limitent l'espace virtuel adressable par un processus à* 2^{47}*. L'adresse virtuelle se décompose donc ainsi: 17 bits inutilisés, 11 bits index0, 11 bits index1, 11 bits index2, 14 bits décalage. Ceci donne pour l'adresse 0x0132457689BACDFE: 0b0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 en binaire, ou 0b0000 0001 0010 0011 0 | 100 0101 0110 | 0111 1000 100 | 1 1010 1011 11 | 00 1101 1110 1111 en séparant les champs. On obtient ainsi champ par champ en hexadécimal: inutilisés 0x0246, index0 0x456, index1 0x3C4, index2 0x6AF, décalage 0x0DEF. Chaque entrée dans la table (en réalité un arbre) de pages contient l'adresse physique correspondante (feuilles) ou l'adresse physique du noeud de prochain niveau. Cette adresse occuperait normalement 64 bits mais, puisque les pages sont alignées sur un multiple de 16Kio, les 14 derniers bits sont implicitement à 0 et peuvent être utilisés pour des bits de statut (page accédée, modifiée, ou valide, permissions d'accès de la page...).*

c) Plusieurs périphériques permettent un accès direct à la mémoire (DMA). Le système peut ainsi envoyer une commande à l'unité de disque pour lire certains secteurs et copier leur contenu à une certaine adresse physique en mémoire centrale qui correspond à un tampon d'entrée-sortie. Lorsque l'opération est terminée, l'unité de disque le signale au système d'exploitation par le biais d'une interruption. Que doit alors faire le système d'exploitation pour assurer la cohérence avant de laisser l'unité centrale de traitement accéder au contenu du tampon qui vient d'être rempli par l'unité de disque via un accès direct? **(1 point)**

Le problème est que le gestionnaire de cache n'est pas au courant de ces nouvelles valeurs placées directement en mémoire centrale. Le système d'exploitation doit donc invalider le contenu des mémoire cache pour tout l'espace correspondant au tampon d'entrée-sortie ainsi modifié par le périphérique, par exemple avec la fonction `invalidate_kernel_vmap_range`.

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. On veut compter pour une image les pixels qui sont plutôt rouges, verts ou bleus. Ainsi, si l'intensité rouge d'un pixel est plus grande que le vert ou le bleu, le pixel est compté comme rouge. En cas d'égalité de puissance pour 2 ou 3 couleurs pour un pixel, votre programme peut choisir comme il le veut entre ces 2 ou 3 couleurs. Ecrivez un programme, utilisant la librairie TBB, qui offre la fonction *void CompteCouleur(struct Pixel image[], int taille, int *rouge, int *vert, int *bleu)* permettant d'effectuer le calcul du nombre de pixels de chaque couleur. L'argument image contient un vecteur de pixels alors que l'argument taille contient la taille de ce vecteur. La fonction CompteCouleur doit calculer et retourner dans les arguments correspondants le nombre de pixels plutôt rouges, verts ou bleus. **(3 points)**

```
struct Compteur {
  int rouge, vert, bleu;
  struct Pixel *v;
  Compteur() : { rouge=0; vert=0; bleu=0; }
  Compteur(Compteur& c, split) { v = c.v; rouge=0; vert=0; bleu=0; }

  void operator()( const blocked_range<int>& r ) {
    int tmpr = rouge, tmpv = vert, tmpb = bleu;
    for(int i = r.begin(); i != r.end(); i++ ) {
      if(v[i].rouge > v[i].vert) {
        if(v[i].rouge > v[i].bleu) tmpr++;
        else tmpb++;
      } else {
        if(v[i].vert > v[i].bleu) tmpv++;
        else tmpb++;
      }
    }
    rouge = tmpr; vert = tmpv; bleu = tmpb;
  }
  void join( Count& rhs ) {rouge += rhs.rouge; vert += rhs.vert; bleu += rhs.bl
```

```
};

void CompteCouleur(struct Pixel image[], int taille, int *rouge, int *vert, int
  Compteur c;
  c.v = image;
  parallel_reduce(blocked_range<int>(0, taille), c);
  *rouge = c.rouge; *vert = c.vert; *bleu = c.bleu;
}
```

b) Un programme OpenMP est utilisé par une compagnie qui produit du matériel promotionnel et change très souvent de logo. Ainsi, un des thread ajoute régulièrement un nouveau logo alors que tous les thread doivent utiliser le plus récent logo pour générer du matériel. Il en résulte un accès concurrent à la variable *nb_version* et au vecteur *logo*[] entre le thread 0 et les autres thread. Parmi les possibilités pour synchroniser correctement ces accès concurrents, vous avez le choix entre des sections critiques, des opérations atomiques ou un accès sans synchronisation mais avec des barrières mémoire. Quel est le nom de la directive OpenMP à utiliser pour chacune de ces 3 possibilités? Parmi ces 3 possibilités, lesquelles sont applicables ici et laquelle devrait permettre la meilleure parallélisation? Expliquez. **(2 points)**

```
/* Normalement, créer un nouveau logo */
int CreateLogo() { return 42; }

int main(int argc, char **argv)
{ int N = 5000, nb_version = 0;
  int logo[N+1];

  logo[nb_version] = CreateLogo();
  nb_version++;

  #pragma omp parallel for shared(nb_version,logo)
  for(int i = 0; i < N; i++) {
    if(omp_get_thread_num() == 0) {
      logo[nb_version] = CreateLogo();
      nb_version++;
    }
    int v = nb_version - 1;
    int current_logo = logo[v];
    /* Faire quelque chose avec le logo */
  }
}
```

Il est intéressant de constater que les valeurs existantes de logo continuent à être valides puisqu'elles ne sont pas enlevées. De nouvelles entrées dans logo sont simplement ajoutées. L'important est donc que les nouvelles entrées ne soient pas accédées avant d'être assignées, et donc que la nouvelle entrée dans logo soit toujours visible avant que l'incrément de nb_version ne soit visible. Le pragma omp critical, sur le bloc `logo[nb_version] = CreateLogo(); nb_version++;` *et sur le bloc* `int v = nb_version - 1; int current_logo = logo[v];` *permettrait de s'assurer*

que l'accès ne puisse avoir lieu au milieu de la modification et ajoute les barrières mémoires requises pour assurer la cohérence. Le pragma omp atomic est utilisé pour les opérations atomiques mais, étant donné que l'opération inclut un incrément et un accès à un vecteur, cela dépasse ce qui peut être supporté par ce pragma. Le pragma omp flush pourrait être utilisé pour s'assurer que la valeur de `logo[nb_version]` *est disponible avant la nouvelle valeur de* `nb_version`*. Ceci est plus performant que de procéder avec une section critique.*

```
// Pour une boucle de 100000 * 5000 le temps est de
// 4.826s / 18.928s sans cohérence
// 13.128s / 51.500s avec omp flush
// 64.102s / 252.120s avec omp critical
#pragma omp parallel for shared(nb_version,logo)
for(int i = 0; i < N; i++) {
  if(omp_get_thread_num() == 0) {
    //#pragma omp critical
    //{
    logo[nb_version] = CreateLogo();
    #pragma omp flush
    nb_version++;
    #pragma omp flush(nb_version)
    //}
  }
  //#pragma omp critical
  //{
  #pragma omp flush(nb_version)
  int v = nb_version - 1;
  #pragma omp flush
  int current_logo = logo[v];
  //}
  /* Faire quelque chose avec le logo */
```

Question 4 (5 points)

a) On vous fournit la section de code suivante en OpenMP. Dans un premier temps, vérifiez que le code est correct (déterministe en présence de thread parallèles), et suggérez au besoin un correctif. Ensuite, vous devez proposer plusieurs améliorations qui permettront d'obtenir le même résultat mais plus efficacement. Fournissez une nouvelle version de cette section de programme et expliquez en une ou deux lignes de texte chaque amélioration apportée. **(3 points)**

```
int i, j, k;
double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB], d[ND];

int main(int argc, char **argv) {
  #pragma omp parallel shared(a,b,c,d) private(i,j,k)
  {
    #pragma omp for
    for(i=0; i<NRA; i++)
```

```
      for(j=0; j<NCA; j++) a[i][j]=fn(i,j);
    #pragma omp for
    for(i=0; i<NCA; i++)
      for(j=0; j<NCB; j++) b[i][j] = fn(i,j);
    #pragma omp for
    for(i=0; i<NRA; i++)
      for(j=0; j<NCB; j++) c[i][j] = 0;
    #pragma omp for
    for(i=0; i<ND; i++) d[i] = 0;

    #pragma omp for
    for(j=0; j<NCB; j++) {
      for (i=0; i<NRA; i++) {
        for (k=0; k<NCA; k++) {
          c[i][j] += a[i][k] * b[k][j];
          d[(int)(c[i][j]) % ND]++;
} } } } }
```

L'accès dans d n'est pas synchronisé et n'est donc pas déterministe. Certains incréments pourraient être perdus. On pourrait en faire une région critique (pragma omp critical) mais il est plus efficace d'utiliser une opération d'incrément atomique. On peut aussi utiliser une réduction sur d. Pour le reste, la partie initialisation est négligeable (boucles de profondeur 2) par rapport à la dernière boucle (profondeur 3). Pour cette dernière boucle, le choix de l'ordre des indices est très important, pour que l'indice le plus à droite dans une matrice varie le plus vite (boucle la plus imbriquée). On veut ainsi i avant j pour c, i avant k pour a, et k avant j pour b. Le meilleur choix est donc d'avoir i, k, j.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>
#include <omp.h>
#include <math.h>

double fn(int n1, int n2) {
  return fabs(sin((double)(n1 * n2)));
}

#define NRA 1000
#define NCA 2000
#define NCB 1500
#define ND 128

int i, j, k;
double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB], d[ND];

int main(int argc, char **argv)
{
  int test = atoi(argv[1]);
```

```
#pragma omp parallel shared(a,b,c,d) private(i,j,k)
{
  // initialisation ~ 0.1s
  #pragma omp for
  for(i=0; i<NRA; i++)
    for(j=0; j<NCA; j++) a[i][j] = fn(i,j);
  #pragma omp for
  for(i=0; i<NCA; i++)
    for(j=0; j<NCB; j++) b[i][j] = fn(i,j);
  #pragma omp for
  for(i=0; i<NRA; i++)
    for(j=0; j<NCB; j++) c[i][j] = 0;
  #pragma omp for
  for(i=0; i<ND; i++) d[i] = 0;

  // boucles j, i, k, sans atomic (course), 8.26s / 32.86s
  // sans optimisation, 20.45s / 78.91s
  if(test == 0) {
    #pragma omp for
    for(j=0; j<NCB; j++) {
      for(i=0; i<NRA; i++) {
        for(k=0; k<NCA; k++) {
          c[i][j] += a[i][k] * b[k][j];
          d[(int)(c[i][j]) % ND]++;
  } } } }
  // boucles j, i, k, 8.15s / 32.43s
  if(test == 1) {
    #pragma omp for
    for(j=0; j<NCB; j++) {
      for(i=0; i<NRA; i++) {
        for(k=0; k<NCA; k++) {
          c[i][j] += a[i][k] * b[k][j];
          #pragma atomic
          d[(int)(c[i][j]) % ND]++;
  } } } }
  // boucles i, j, k, 9.88s / 39.45s
  if(test == 2) {
    #pragma omp for
    for(i=0; i<NRA; i++) {
      for(j=0; j<NCB; j++) {
        for(k=0; k<NCA; k++) {
          c[i][j] += a[i][k] * b[k][j];
          #pragma atomic
          d[(int)(c[i][j]) % ND]++;
  } } } }
  // boucles i, k, j, 6.86s / 27.06s
  if(test == 3) {
```

```
      #pragma omp for
      for(i=0; i<NRA; i++) {
        for(k=0; k<NCA; k++) {
          for(j=0; j<NCB; j++) {
            c[i][j] += a[i][k] * b[k][j];
            #pragma atomic
            d[(int)(c[i][j]) % ND]++;
  } } } }
  // ajout de collapse(2), 6.84s / 27.08s
  // peut introduire une course
  if(test == 4) {
    #pragma omp for collapse(2)
    for(i=0; i<NRA; i++) {
      for(k=0; k<NCA; k++) {
        for(j=0; j<NCB; j++) {
          c[i][j] += a[i][k] * b[k][j];
          #pragma atomic
          d[(int)(c[i][j]) % ND]++;
  } } } }
  // boucles i, k, j, sans atomic (course), 6.93s / 27.28s
  if(test == 5) {
    #pragma omp for
    for(i=0; i<NRA; i++) {
      for(k=0; k<NCA; k++) {
        for(j=0; j<NCB; j++) {
          c[i][j] += a[i][k] * b[k][j];
          d[(int)(c[i][j]) % ND]++;
  } } } }
  // boucles i, k, j, reduction de d, 4.96s / 19.77s
  // sans optimisation, 16.55s / 65.97s
  if(test == 6) {
    #pragma omp for reduction(+:d)
    for(i=0; i<NRA; i++) {
      for(k=0; k<NCA; k++) {
        for(j=0; j<NCB; j++) {
          c[i][j] += a[i][k] * b[k][j];
          d[(int)(c[i][j]) % ND]++;
  } } } }
} }
```

b) Le programme OpenMP suivant s'exécute et produit une sortie sur stdout. Donnez une sortie possible produite par l'exécution de ce programme. Est-ce que la sortie peut changer d'une exécution à l'autre? Les valeurs de *p* et *s* affichées sur la ligne c) sont-elles toujours égales? Expliquez. **(2 points)**

```
int main(int argc, char **argv)
{ int nb_thread, thread, p, s;
  printf("a)\n");
  omp_set_num_threads(3);
```

```
  #pragma omp parallel private(thread, p) shared(s)
  { printf("b)\n");
    #pragma omp for schedule(static,1) ordered
    for(int i = 0; i < 8; i++) {
      nb_thread = omp_get_num_threads();
      thread = omp_get_thread_num();
      p = s = i;
      printf("c) thread %d, p = %d, s = %d\n", thread, p, s);
    }
    #pragma omp single
    printf("d) thread %d / %d\n", thread, nb_thread);
    #pragma omp master
    printf("e) thread %d / %d\n", thread, nb_thread);
    #pragma omp critical
    printf("f) thread %d / %d\n", thread, nb_thread);
  }
  printf("g) thread %d / %d\n", thread, nb_thread);
}
```

La sortie peut en effet changer car l'ordre entre les threads peut changer. Ainsi, a et g sont hors de la section parallèle et ne changeront pas d'ordre. La valeur de thread, variable privée, affichée par g n'est pas définie et peut changer. Les autres printf peuvent changer d'ordre. La clause ordered a un effet difficile à réaliser car la synchronisation associée est à la fin de l'itération, après le printf. De plus, la sortie d n'est effectuée que par un seul thread mais celui-ci peut changer d'une fois à l'autre (pragma omp single), ce qui changera le numéro de thread associé à d. La valeur affichée pour s peut changer puisque cette variable est partagée et est accédée sans synchronisation. Voici une sortie possible:

```
a)
b)
b)
c) thread 0, p = 0, s = 0
c) thread 0, p = 3, s = 3
b)
c) thread 1, p = 1, s = 1
c) thread 1, p = 4, s = 4
c) thread 2, p = 2, s = 2
c) thread 2, p = 5, s = 5
c) thread 0, p = 6, s = 6
c) thread 1, p = 7, s = 7
d) thread 0 / 3
f) thread 2 / 3
f) thread 1 / 3
e) thread 0 / 3
f) thread 0 / 3
g) thread 21942 / 3
```

Le professeur: Michel Dagenais