ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2012)
3 crédits (3-1.5-4.5)

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Mardi le 30 octobre 2012

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) Vous avez accès à une grappe de 128 ordinateurs. Chaque ordinateur est constitué d'une partie électronique avec un taux de disponibilité de 0.999 et de deux disques redondants en miroir. Le taux de disponibilité de chaque disque est de 0.99. Vous avez conçu votre application parallèle pour décomposer le problème en 128 morceaux, faire calculer chaque morceau sur un ordinateur différent, et obtenir le résultat à la fin si tous les ordinateurs ont été opérationnels tout au long du calcul. Quelle est la probabilité que toute la grappe soit disponible à un instant donné? **(2 points)**

Le miroir de disques sera non disponible si les deux disques sont en panne simultanément, $(1 - 0.99)^2 = .0001$*, soit une probabilité de disponibilité pour le miroir de 0.9999. La probabilité qu'un ordinateur (disque et électronique) soit disponible est donc de 0.999 x 0.9999 = 0.9989. La probabilité que les 128 ordinateurs soient disponibles en même temps est conséquemment de* $0.9989^{128} = 0.8686$.

b) Une grappe de 1024 noeuds est utilisée pour une tâche de calcul qui se termine par une réduction (somme des résultats fournis par chacun des 1024 noeuds). Les 1024 noeuds sont connectés à un commutateur performant qui permet à n'importe quelle paire de

noeuds de communiquer en même temps. Toutefois, un noeud donné ne peut envoyer et recevoir qu'un seul message par unité de temps. Propposez une organisation efficace pour effectuer l'opération de réduction. Combien d'unités de temps sont nécessaires pour envoyer les messages requis pour cette opération de réduction? **(2 points)**

Une propagation en arbre fait le travail avec 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 noeuds, reste 1 noeud... Au premier tour, les 256 feuilles de gauche envoient leur résultat, puis les 256 de droite, ensuite de 256 vers 128, les 128 noeuds de gauche puis les 128 de droite... jusqu'à la racine qui reçoit la valeur de gauche puis celle de droite et les somme, pour un total de 18 unités de temps, auquel il faut ajouter 1 unité pour recevoir la valeur du noeud orphelin. Ceci demande 1023 messages.

Une propagation en papillon fonctionne différemment. Au premier cycle, les 512 noeuds impairs envoient leur valeur aux 512 noeuds pairs. Au deuxième cycle, les 256 noeuds pairs non multiples de 4 envoient leur valeur aux 256 noeuds multiples de 4. Ceci se poursuit pour 128, 64, 32, 16, 8, 4, 2 et 1 noeud, pour un total de 10 unités de temps. Cette solution est donc préférable. Ici encore, 1023 messages sont envoyés.

c) Vous analysez un programme à paralléliser et trouvez qu'il est constitué d'une phase a) de préparation du problème, une phase b) de calcul matriciel, et une phase c) de regroupement des résultats. En mode sériel, ces phases prennent respectivement a: 10s, b: 4000s c: 20s. En parallélisant le programme sur 128 noeuds, la phase b se retrouve raccourcie par un facteur 128 mais il faut ajouter 20s pour envoyer les données en multi-diffusion aux 128 noeuds et 1s par noeud pour récupérer les résultats. Quel est le temps de calcul sériel? En parallèle? Quel est le facteur d'accélération. **(1 point)**

Le temps était de 10s + 4000s + 20s = 4030s et devient 10s + 20s + 4000s / 128 + 20s + 128s = 209.25, pour un facteur d'accélération de 19.26.

Question 2 (5 points)

a) Dans votre premier travail pratique, vous avez utilisé la librairie TBB pour paralléliser le problème du dragon, et vous avez étudié comment la librairie TBB découpait le problème en intervalles et assignait les fils d'exécution. Décrivez et expliquez comment la librairie décomposait votre problème du dragon et assignait le travail à plusieurs fils d'exécution. **(2 points)**

La librairie TBB divise récursivement en 2 le problème en ajoutant des fils d'exécution jusqu'à atteindre 8 fils (nombre de processeurs logiques). Par la suite, une fraction de l'intervalle disponible au fil est traité, de manière à laisser une portion à assigner plus tard de manière plus flexible, permettant ainsi d'équilibrer la charge.

b) Vous devez compiler un très gros logiciel constitué de 5000 fichiers. La compilation de chaque fichier prend 0.5s de temps de processeur et 0.4s d'attente après les serveurs de fichiers en réseau. Votre ordinateur contient 8 processeurs. Puisqu'il est facile de paralléliser une telle compilation en répartissant les fichiers entre les processeurs, combien de processus parallèles (ou fils d'exécution) de compilation devrait-on utiliser

au minimum pour accélérer le plus possible la compilation? Est-ce qu'il y a un problème à créer beaucoup plus de processus parallèles que ce minimum requis? **(2 points)**

Pendant 1 seconde de temps CPU de compilation, il y aura 0.4s x 1s / .5s = 0.8s de temps d'attente pour le serveur de fichier en réseau. En planifiant 8 fils d'exécution pour les 8 processeurs, il suffirait d'avoir 8 x .8s / 1s = 6.4, soit 7 fils d'exécution supplémentaires pour bloquer et attendre après les accès aux fichiers. Le problème toutefois avec cela est qu'il y aurait toujours un fil disponible pour occuper un CPU mais ce fil pourrait avoir exécuté la fois précédente sur un autre CPU, forçant ainsi une migration de CPU inutile avec les fautes de cache et de TLB associées. Il serait donc préférable d'avoir 8 + 8 = 16 fils d'exécution de manière à tenir tous les CPU occupés et à minimiser la migration de fils d'exécution entre les CPU. Avoir beaucoup plus de fils d'exécution que nécessaire ajoute un peu de coût pour créer et ordonnancer les fils puis pour les changements de contexte. Toutefois, cela est normalement moins coûteux que l'inverse, avoir trop peu de fils et laisser des processeurs inoccupés. Pour cette raison, il est préférable d'avoir un peu trop de fils que pas assez.

c) Les ordinateurs du laboratoire bénéficient de la technologie hyperthread. Comment cela fonctionne-t-il et quel en est le bénéfice? Est-ce que cela affecte le choix du nombre de processus parallèles recommandés pour accélérer une tâche parallèle? **(1 point)**

Les processeurs physiques contiennent un double ensemble de registres et apparaissent chacun comme deux processeurs logiques. Lorsqu'une faute de cache survient sur un processeur logique, l'autre processeur logique devient actif. Ceci permet donc, pour un seul processeur physique, d'accomplir plus de travail car les latences des fautes de pages ne sont pas perdues puisque deux fils (un sur chaque processeur logique) sont disponibles pour utiliser le processeur physique. Le gain de performance est de 10 à 30% environ. Pour une application parallèle, il faut donc s'assurer d'avoir assez de fils d'exécution pour occuper tous les processeurs logiques.

Question 3 (5 points)

a) Le programe suivant s'exécute sur un ordinateur du laboratoire utilisé pour les travaux pratiques du cours INF8601. La variable sums est un vecteur d'entiers 32 bits de 64 entrées. Le temps d'exécution de ce programme (temps réel écoulé) pour nb_iter = 1 000 000 000 est, pour nb_thread = (1, 2, 4, 8, 16), de respectivement (2.57s, 3.41s, 4.60s, 3.67s, 2.85s). Comment expliquez-vous que le temps augmente puis diminue en fonction du nombre croissant de fils d'exécution parallèles? Ces ordinateurs contiennent 4 processeurs et une cache L1 avec des blocs de longueur 64 octets. **(2 points)**

```
  omp_set_num_threads(nb_thread);
  #pragma omp parallel for shared(sums)
  for(i = 0; i < nb_iter; i++) {
    sums[omp_get_thread_num()] += i % 2 + 1;
  }
}
```

Les blocs en cache contiennent 16 mots de 4 octets. Avec un seul processeur, il n'y a pas de faute de cache. Avec deux processeurs, les deux fils d'exécution souffrent du faux partage puisque les entrées sums[0] et sums[1] tombent dans la même ligne de cache. Le coût associé aux deux processeurs qui se battent pour avoir cette ligne de cache dépasse tout gain qu'il pourrait y avoir avec deux processeurs. A quatre fils d'exécution, le problème est exacerbé. Rendu à 8 fils d'exécution, deux fils partagent chaque processeur physique et peuvent chacun faire un tour de boucle en utilisant le même bloc en cache. Il y a donc autant de contention entre les 4 processeurs mais à chaque fois qu'un processeur a le bloc, deux fils d'exécution peuvent en profiter. Rendu à 16 fils, il y a 4 fils par processeur et donc moins d'échanges pour chaque bloc de cache et la performance s'améliore encore par rapport au pire cas à 4 processeurs.

b) Les programmes suivants s'exécutent sur deux processeurs à mémoire partagée afin de jouer à Roche, Papier, Ciseaux. Chaque processeur choisit un chiffre entre 0 et 2, attend que le choix de l'autre processeur soit prêt et vérifie s'il a gagné, auquel cas il émet un son pour signifier qu'il a gagné. Est-ce que ce programme donnera le résultat attendu sur un ordinateur à cohérence séquentielle? Avec ordonnancement total des écritures? Avec un ordonnancement partiel des écritures? Avec un ordonnancement faible? Quelles sont les barrières mémoire à insérer (et où) afin d'assurer un comportement correct du programme dans tous les cas? **(2 points)**

```
(initialement choix0 = -1; pret0 = 0, choix1 = -1; pret1 = 0)
Processeur 0                        Processeur 1

choix0 = choisir(0,2);              choix1 = choisir(0,2);
pret0 = 1;                          pret1 = 1;
while(pret1 == 0);                  while(pret0 ==0);
if(gagne(choix0, choix1)            if(gagne(choix1, choix0)
{ jouer_musique();                  { jouer_musique();
}                                   }
```

Avec un ordinateur à cohérence séquentielle ou même avec un ordonnancement total des écritures, le protocole fonctionnera tel qu'attendu. Il faut toutefois s'assurer que le compilateur ne réordonnance pas les écritures, soit en empêchant l'optimisation, en déclarant certaines variables volatiles ou en ajoutant des barrières mémoire de compilateur. Avec un ordonnancement partiel ou faible, il peut arriver que pret0 (ou pret1) soit écrit en mémoire centrale et visible à l'autre processeur avant choix0 (ou choix1). Ceci briserait le protocole et empêcherait le fonctionnement correct. Ceci peut être corrigé avec une barrière d'écriture qui force pret0 à être propagé en mémoire avant choix0, et une barrière de lecture qui force une synchronisation de toutes les lectures, assurant que la mise à jour de choix0 soit nécessairement lue si pret0 est lu à 1.

Une alternative à envisager serait de mettre une barrière mémoire complète smp_mb plutot que smp_wmb et de laisser tomber smp_rmb. Dans ce cas, si la barrière sur le processeur 0 est rencontrée en premier, il transmettrait choix0 à la mémoire centrale

et viderait sa queue de mise à jour de la cache; choix1 et pret1 pourraient ne pas avoir encore été changés. Si maintenant le processeur 1 progresse et passe sa barrière mémoire, il transmettrait choix1 à la mémoire centrale et obtiendrait la mise à jour pour choix0. Pour la suite, le processeur 1 fonctionnera correctement. Par contre, sur le processeur 0, choix1 bien que rendu en mémoire centrale peut encore être en queue de mise à jour et, par malchance, il pourrait arriver que pret1, bien qu'arrivé em mémoire centrale après choix1, arrive avant choix1 au processeur 0 en raison de queues de mises à jour multiples pouvant causer des réordonnancements. Il serait donc possible selon ce scénario que le processeur 0 obtienne un mauvais résultat, et cette alternative n'est donc pas acceptable.

```
(initialement choix0 = -1; pret0 = 0, choix1 = -1; pret1 = 0)
Processeur 0                        Processeur 1

choix0 = choisir(0,2);              choix1 = choisir(0,2);
smp_wmb();                          smp_wmb();
pret0 = 1;                          pret1 = 1;
while(pret1 == 0);                  while(pret0 ==0);
smp_rmb();                          smp_rmb();
if(gagne(choix0, choix1)            if(gagne(choix1, choix0)
{ jouer_musique();                  { jouer_musique();
}                                   }
```

c) Dans les systèmes de mémoire cache avec cohérence par répertoire, il faut stocker pour chaque bloc en cache la liste des processeurs qui en possèdent une copie, par exemple pour leur envoyer un message d'invalidation en cas d'écriture. Une manière naïve de réaliser cela serait d'avoir une matrice d'octets avec une rangée pour chaque bloc de mémoire centrale et une colonne pour chaque processeur. Chaque octet permet d'indiquer si un bloc se trouve sur un processeur donné. Est-ce organisé de cette manière dans les processeurs courants? Comment peut-on réaliser ce répertoire de manière à prendre beaucoup moins d'espace? **(1 point)**

Les processeurs réservent normalement un bit plutôt qu'un octet par processeur pour indiquer la présence ou non d'un bloc. Ensuite, il suffit de stocker cette information pour les blocs en cache et non pas pour tous les blocs de mémoire centrale. C'est exactement ce que fait le i7 de Intel qui ajoute cette information à celle des étiquettes des blocs présents au niveau de sa cache L3.

Question 4 (5 points)

a) Le programme suivant effectue la multiplication de deux matrices contenant des valeurs calculées par les fonctions fn1 et fn2. Le temps requis pour exécuter fn1 et fn2 varie beaucoup selon la valeur de leurs deux arguments. Ce programme ne fonctionne pas à la vitesse espérée et on fait appel à vos services d'expert. Proposez des améliorations

qui permettent d'accélérer le plus possible ce programme parallèle sans changer le résultat final dans la matrice c. **(2 points)**

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

#pragma omp parallel shared(a,b,c) private(i,j,k)
{ #pragma omp for
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= fn1(i,j);
  #pragma omp for
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= fn2(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(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];
```

Pour la dernière boucle, l'ordre des indices est mauvais pour la localité de référence. Il faut que j varie plus vite que i et k (pour faire des accès séquentiels dans le même bloc de cache dans c et b), et que k varie plus vite que i (pour faire des accès séquentiels dans a). Il devient alors possible d'intégrer le calcul de a à cette boucle car a est justement de dimension [NRA][NCA]. Il est facile de mettre nowait sur la première boucle restante, permettant en parallèle de finir le calcul de b et commencer la mise à 0 de c.

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

#pragma omp parallel shared(a,b,c) private(i,j,k)
{ #pragma omp for nowait
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= fn2(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<NRA; i++)
  for (k=0; k<NCA; k++) {
    a[i][k]= fn1(i,k);
    for(j=0; j<NCB; j++)
      c[i][j] += a[i][k] * b[k][j];
  }
```

b) Pour chacun des 4 outils suivants expliquez leur utilité, leur coût en performance, et un exemple de problème pour lequel il pourrait poser un diagnostic efficacement: profileur de fonction Valgrind/Callgrind, outil de vérification Valgrind/Helgrind, profileur de compteur de performance OProfile, traceur LTTng. **(2 points)**

Valgrind/Callgrind permet de mesurer la proportion du temps passé dans chaque fonction et donc de voir laquelle a le plus besoin d'optimisation. Chaque entrée et sortie de fonction doit être instrumentée à un coût non négligeable.

L'outil Valgrind/Helgrind permet de vérifier l'utilisation des primitives de synchronisation dans un programme, par exemple pour détecter un ordonnancement incohérent de la prise des verrous ou une course sur une variable partagée en raison de l'oubli de protéger l'accès par la prise d'un verrou. Chaque accès en mémoire et chaque action de synchronisation doit être instrumentée impliquant un coût très important.

L'outil Oprofile permet d'obtenir pour un programme un profil des divers événements pour lesquels il existe des compteurs matériel. Par exemple, un profil des fautes de cache permet de voir rapidement si une section d'un programme cause beaucoup de fautes en cache par exemple en raison d'un faux partage. Oprofile procède par échantillonnage et ajoute un surcoût minime.

Finalement, l'outil LTTng permet de montrer l'état de chaque processus en fonction du temps. Il permet donc facilement de voir tout ce qui retarde un processus, CPU, accès disques, attente après un autre processus un l'expiration d'un délai. Quelques événements clé du noyau, qui sont déjà instrumentés statiquement, sont typiquement activés pour un surcoût faible.

c) Un programme parallèle sur 64 processeurs doit trouver le chemin le plus court pour un voyageur de commerce qui doit visiter un grand nombre de villes. Chaque fil d'exécution, après avoir évalué la longueur d'un chemin, vérifie si ce chemin est plus court que le meilleur chemin trouvé jusqu'à présent. Pour ce faire, le programme peut utiliser des mutex de lecture-écriture ou utiliser la technique RCU afin de lire et éventuellement remplacer la valeur du chemin le plus court jusqu'à présent. Lequel serait le plus performant? Pourquoi? Proposez une troisième solution possiblement plus intéressante? **(1 point)**

Dans ce cas, la valeur courante de chemin le plus court est lue très souvent mais rarement changée. La technique RCU est très appropriée pour ces cas. Toutefois, il serait aussi possible de simplement avoir une valeur de chemin le plus court par fil

d'exécution et de prendre le meilleur choix entre ceux trouvés par chaque fil à la toute fin. Dans ce cas, il n'y aurait plus de contention.

Le professeur: Michel Dagenais