**ECOLE POLYTECHNIQUE DE MONTREAL**

**Département de génie informatique et génie logiciel**

**Cours INF8601:** Systèmes informatiques parallèles (Automne 2013)
3 crédits (3-1.5-4.5)

---

**CORRIGÉ DU CONTRÔLE PÉRIODIQUE**

**DATE: Mardi le 22 octobre 2013**

**HEURE: 14h45 à 16h35**

**DUREE: 1H50**

**NOTE: Toute documentation permise, calculatrice non programmable permise**

**Ce questionnaire comprend 4 questions pour 20 points**

---

# Question 1 (5 points)

a) Une application utilise énormément de mémoire et le système de mémoire virtuelle doit ainsi régulièrement transférer certaines pages sur le disque. Une optimisation suggérée est d'avoir la possibilité de compresser des pages en mémoire. Lorsqu'une page est très utilisée elle est conservée telle quelle, lorsqu'une page est moins utilisée elle est compressée mais conservée en mémoire et peut être décompressée au prochain accès, et lorsqu'une page est très peu utilisée elle est transférée sur disque pour être relue au prochain accès. Cette technique permet de conserver jusqu'à deux fois plus de pages en mémoire lorsqu'elles sont compressées mais demande un certain effort de calcul pour la compression / décompression. En raison de ce compromis, cette technique est présentement marginalement utile. Pour chacun des paramètres suivants, dites s'il rend plus avantageux ou moins avantageux cette technique: i) augmentation de la vitesse du disque en passant à un disque à semi-conducteur (SSD), ii) augmentation de la vitesse du CPU, iii) augmentation du nombre de CPU, iv) augmentation de la taille de la mémoire? Justifiez? **(2 points)**

*Si le disque est plus rapide, le coût de transférer une page vers le disque est moindre et la technique est moins intéressante. Une augmentation de la vitesse du CPU ou du*

*nombre de CPU donne plus de ressources pour la compression / décompression, rend le disque plus lent en terme relatif et accroît l'intérêt de la technique. Si la mémoire est plus grande, les pages à transférer sont moins nombreuses et parmi celles les moins souvent utilisées, la technique est alors moins intéressante.*

b) Un centre de données contient 10 000 noeuds. Chaque noeud est connecté à une unité externe de disques en RAID contenant 5 disques dont au moins 4 doivent être fonctionnels pour opérer correctement. Chaque noeud en lui-même est en panne en moyenne 8 heures par année. Par ailleurs, chaque unité RAID est en panne lorsque 2 disques ou plus sont défectueux, avec chaque disque qui a une probabilité de panne de 0.001. Quelle est la probabilité de panne pour un noeud (noeud lui-même ou unité de disque en panne) et donc quel est le nombre moyen de noeuds en panne à un instant donné dans le centre de données. **(2 points)**

*Une unité de disque sera fonctionnelle s'il y a 0 ou 1 disque en panne:* $5!/5! \times (1 - .001)^5 + 5!/4! \times .001 \times (1 - .001)^4 = 0.99500999 + .0049800 = 0.99999$. *Un noeud sera fonctionnel* $1 - 8/(365.25 \times 24) = 0.999087$, *soit pour un noeud et ses disques* $0.99999 \times 0.999087 = 0.999077$. *Sur 10 000 noeuds, il y aurait donc à tout moment en moyenne 9990.77 noeuds fonctionnels, soit 9 ou 10 noeuds en panne.*

c) En une seconde, une application sur un processeur exécute 2 000 000 000 instructions et génère autant d'accès mémoire pour les instructions, en plus de générer 500 000 000 accès pour les données, dans le cas idéal où il n'y a aucune attente après la mémoire (pas de faute de cache). Le taux de succès en cache est de 99.9% pour les instructions et de 99.5% pour les données. La pénalité d'échec est de 80ns. Quelle est la vitesse de cette application sur ce processeur, en nombre d'instructions par seconde, en tenant compte des attentes après la mémoire? **(1 point)**

*Ce qui prendrait idéalement une seconde prendra en plus* $(2\ 000\ 000\ 000 \times (1-0.999) + 500\ 000\ 000 \times (1 - 0.995))80ns = 0.36s$. *Ceci donne donc 2 000 000 000 instructions en 1.36s ou 1470 MIPS.*

# Question 2 (5 points)

a) Dans le premier travail pratique, vous avez utilisé l'outil Valgrind / Callgrind. Quelle information en avez-vous tiré? Comment avez-vous pu avec cette information prédire le facteur d'accélération de votre application en multi-processeur? Est-ce que cette prédiction est plutôt une borne inférieure ou supérieure de l'accélération qui sera réellement obtenue? **(1.5 point)**

*Cet outil permet de voir la proportion du temps passé dans chaque fonction. Il peut ainsi montrer les fonctions sur lesquelles il faut se concentrer pour paralléliser ou optimiser le programme. Dans le meilleur des cas, en parallélisant une fonction sur un processeur avec n coeurs, on obtiendra un facteur d'accélération pour cette fonction de n. En connaissant la fraction du temps consacré à cette fonction et le facteur d'accélération, il est facile de calculer le gain global résultant avec la loi d'Amdahl.*

*Cette prédiction est une borne supérieure car souvent on ne réussit pas à accélérer par un facteur n, en raison de temps perdu en contention pour la mémoire cache, en synchronisation, ou en préparation du travail parallèle.*

b) Toujours pour le premier travail pratique, vous avez utilisé LTTng afin de mieux comprendre le comportement de TBB en comparaison avec la librairie PThread. Combien de fils d'exécution sont utilisés dans chaque cas? Comment se fait la répartition du calcul entre les fils d'exécution? Quelles sont les données écrites dans la trace qui permettent ainsi d'afficher une vue temporelle de l'exécution? **(1.5 point)**

*Dans les deux cas, un fil par processeur logique est démarré pour faire le travail, soit 8. Avec PThread, la décomposition est explicite et chaque thread fait le travail demandé, qui est divisé dès le départ et espère-t-on correspond à environ 1/8 du travail, afin de bien équilibrer la charge. Dans le cas de TBB, il découpe le travail récursivement en 2, créant un second fil pour prendre la seconde moitié, jusqu'à avoir un fil par processeur. Rendu là, il prend une fraction du travail à faire afin de compléter l'ensemble en quelques itérations. L'avantage est qu'un fil qui termine tout son travail plus vite peut aller chercher du travail restant auprès des autres fils qui seraient rendus moins loin.*

*La trace contient tous les événements noyaux importants, en particulier tous les ordonnancements (changements de contexte) pour chaque processeur, en plus des appels systèmes (entrée et sortie) pour chaque processus. De là, il est facile de calculer quel est le processus courant sur chaque processeur en fonction du temps et même de montrer pour chaque processus s'il est en mode usager ou système.*

c) Un serveur est responsable de fournir des images en différents formats et résolutions. Chaque requête pour une image demande en moyenne 150ms de temps de processeur, 50ms de disque et 300ms d'attente après un client lors de la communication réseau. Chaque fil d'exécution traite une requête à la fois, séquentiellement. L'unité centrale de traitement contient 8 processeurs (physiques sans hyperthread). En supposant que les disques et le réseau sont toujours disponibles, combien de fils d'exécution devrait-on avoir au minimum pour toujours occuper les 8 processeurs? Combien de disques devrait-on avoir afin de pouvoir fournir au même débit que les 8 processeurs. **(2 points)**

*Une requête complète prend* $150ms + 50ms + 300ms = 500ms$*, soit une répartition de 0.3 CPU, 0.1 disque et 0.6 réseau. Il faudra avoir des fils d'exécution en proportion avec 8 pour les 8 processeurs et donc* $8 \times 0.1/0.3 = 2.66$ *donc 3 pour les disques et* $8 \times 0.6/0.3 = 16$ *pour l'attente après les clients, ce qui fait un total de 27 au minimum. Il faut 3 disques puisque chacun des 3 fils réservés aux disques correspond à la charge requise pour utiliser un disque.*

# Question 3 (5 points)

a) Votre ordinateur contient 8 processeurs. Cependant, vous ne savez pas si ce sont en fait 4 processeurs physiques avec hyperthread ou 8 processeurs physiques. De plus,

vous ne savez pas si certains processeurs partagent on non leur cache de niveau L1. Cependant, on vous a bien mis en garde contre le fait de mettre deux tâches temps réel sur deux processeurs qui partagent la même cache L1. Sachant que vous pouvez facilement spécifier au système d'exploitation quel fil exécuter sur quel processeur, proposez une méthode (description d'un programme d'essai à exécuter) pour déterminer les processeurs physiques et logiques ainsi que les processeurs physiques qui partagent leur cache L1. **(2 points)**

*Pour déterminer si la technologie hyperthread est utilisée, on peut exécuter un petit programme qui ne fait que des calculs sur des données prenant peu d'espace, il devrait y avoir très peu de fautes de cache. En exécutant un tel programme sur un processeur puis sur deux processeurs en parallèle, la performance de chacun devrait être identique si ce sont deux processeurs physique différents. Par contre, si les deux processeurs sont en fait virtuels et partagent le même processeur physique, la performance sera environ la moitié pour chacun.*

*Pour déterminer si deux processeurs physiques partagent la même cache L1, différentes stratégies sont possibles. Une stratégie est d'accéder en boucle un vecteur de grande taille (nécessairement plus grande que la taille d'une cache L1). Si le même programme est exécuté sur deux processeurs sans ralentissement, la cache n'est pas partagée. Par contre, si la performance est sensiblement dégradée, l'impact peut venir de la contention sur la cache L1 ou de la contention à un autre niveau du système de mémoire.*

*Une deuxième stratégie plus fiable est de premièrement mesurer la taille de la cache L1. Ceci peut se faire en accédant en boucle un vecteur et en variant la taille de ce vecteur tout en mesurant la performance pour chaque cas. La performance (nombre d'accès par seconde) devrait être assez stable jusqu'à s'approcher de la taille de la cache L1 et par la suite diminuer rapidement. En prenant la plus grande taille avant que la performance ne se dégrade, on utilise au maximum la cache L1. Ensuite, avec cette taille, on peut rouler ce programme sur deux processeurs en parallèle. Si les caches L1 sont séparées, chacun s'exécutera avec la bonne performance. Si les caches L1 sont partagées, chacun se retrouvera avec la moitié de la cache à sa disposition environ et la taille du vecteur sera alors plus grande que l'espace disponible et les deux programmes verront une performance dégradée.*

*Une troisième stratégie est d'avoir deux fils d'exécution parallèles qui lisent et écrivent en boucle une variable partagée. Si la performance est beaucoup meilleure pour une paire de processeurs que pour une autre, cette première paire partage vraisemblablement leur cache L1.*

b) Un jeu multi-fil s'exécute sur deux processeurs à mémoire partagée. C'est un jeu classique à plusieurs joueurs, chacun ayant le choix entre plusieurs outils, le but du jeu étant de construire quelque chose. Une section de code montrée plus bas effectue la sélection d'outil sur le processeur 0 alors que les informations sur cet outil peuvent être accédées sur le processeur 1. Pour que l'exécution soit cohérente, il faut que les informations sur le nouvel outil choisi soient disponibles avant d'être accédées. Quelles sont les barrières mémoire minimales à insérer (et où) afin d'assurer un comportement correct du programme dans tous les cas? **(2 points)**

```
(initialement outil = &marteau; marteau.debut = 0)
Processeur 0                    Processeur 1

tournevis.debut = 1;            outil_courant = outil;
outil = &tournevis;             debut = outil_courant->debut;
```

*Ne sachant le modèle de mémoire de l'ordinateur cible, il faut insérer des barrières mémoire qui effectueront le travail requis selon l'ordinateur sur lequel le programme s'exécute. Sur le processeur 0, il faut que tournevis.debut soit disponible avant d'être lu (avant que outil = &tournevis), ce qui requiert une barrière mémoire en écriture entre les deux. Sur le processeur 1, il faut que la mise à jour de tournevis.debut arrive avant que le nouvel outil ne soit vu. Pour ce faire, il faut donc une barrière de lecture qui assure que si on a lu le nouvel outil, toute autre mise à jour antérieure (tournevis.debut) nous soit aussi parvenue. Cependant, le compilateur ou le pipeline super-scalaire ne vont pas causer de réordonnancement entre les lectures de outil_courant et outil_courant-¿debut car la deuxième dépend de la première. Cependant, sur de rares processeurs (DEC Alpha), des queues d'invalidation multiples parallèles pourraient causer un problème. La barrière minimale requise est donc une barrière de lecture dépendante. Une barrière de lecture donnerait une exécution cohérente mais serait un peu moins rapide.*

```
(initialement outil = &marteau; marteau.debut = 0)
Processeur 0                    Processeur 1

tournevis.debut = 1;            outil_courant = outil;
smp_wmb();                      smp_read_barrier_depends();
outil = &tournevis;             debut = outil_courant->debut;
```

c) Un système de cache utilise le protocole MOESI et un maintien de la cohérence par répertoire. Expliquez quelle information doit être conservée avec cette organisation pour chaque bloc en mémoire cache? **(1 point)**

*Pour chaque bloc en mémoire cache, il faut connaître l'état du bloc (M, O, E, S ou I) ainsi que la liste des copies. L'état pour chaque bloc dans chaque cache peut être représenté sur 3 bits, par exemple (valid, modified, shared). Le répertoire des blocs n'a pas à être recopié dans chaque cache. Par exemple, il peut se trouver dans la cache L3 partagée. Il contient un vecteur de bits (un bit par processeur) pour chaque bloc qui indique pour chaque processeur si ce bloc est en cache ou non, pour savoir si un message d'invalidation doit être envoyé.*

# Question 4 (5 points)

a) La fonction suivante décale la valeur de chaque pixel d'une image. La variable KEY est une constante et l'image reçue en argument n'est pas accédée par d'autres fonctions

en parallèle. Proposez plusieurs améliorations qui devraient permettre d'accélérer significativement cette fonction tout en obtenant le même résultat à la fin. Justifiez. **(2 points)**

```
int encode(struct image *img)
{
    int i, j, index;
    int checksum = 0;

    #pragma omp parallel for private(i,j,index)
    for (i = 0; i < img->width; i++) {
        for (j = 0; j < img->height; j++) {
            index = i + j * width;
            img->data[index] = img->data[index] + KEY;
            #pragma omp atomic
            checksum += img->data[index];
        }
    }
    return checksum;
}
```

*Plusieurs optimisations sont possibles. Premièrement, la matrice de points de l'image doit être accédée par rangées pour augmenter la localité de référence. Ensuite, il est moins efficace de faire une opération atomique que d'utiliser une réduction pour checksum. Par ailleurs, il faut éviter d'accéder à répétition des variables qui ne peuvent être mises dans des registres car elles ne sont pas locales (tous les champs de img). Finalement, on peut simplifier légèrement le calcul de l'index. On pourrait aussi fusionner les deux boucles.*

```
int encode(struct image *img)
{
    int i, j;
    char *index;
    int checksum = 0;
    int width = img->width;
    int height = img->height;
    char *data = img->data;

    #pragma omp parallel for private(i,j,index) reduction(+:checksum)
    for (i = 0; i < height; i++) {
        index = data + i * width;
        for (j = 0; j < width; j++) {
            *index += KEY;
            checksum += *index;
            index++;
```

```
        }
    }
    return checksum;
}
```

b) Quelle est la différence entre les stratégies *static*, *dynamic*, *guided* et *auto* pour partitionner les itérations des boucles et les ordonnancer sur les fils d'exécution en OpenMP? Expliquez dans quelle situation chacune peut être particulièrement intéressante. **(2 points)**

*Auto choisit automatiquement une stratégie alors que static décompose au départ, de manière prévisible, le travail. Dynamic distribue des blocs d'itérations au fur et à mesure que les fils d'exécution deviennent libres. Les blocs ont tous la même taille (sauf les derniers) mais le nombre de blocs exécutés peut varier d'un fil à l'autre. Avec guided, la taille des blocs varie, celle-ci étant proportionnelle au nombre d'itérations restantes divisé par le nombre de fils d'exécution. Dans certains cas, une partition statique permet d'utiliser l'option nowait entre des sections parallèles, ce qui peut améliorer la performance. Static permet aussi de réduire la variabilité, ce qui peut être utile pour le déboguage. Dynamic peut être intéressant lorsqu'on veut spécifier une très petite taille de bloc d'itération, lorsque la durée de chaque itération varie énormément. Autrement, guided est généralement une meilleure stratégie. Auto laisse le système choisir, ce qui sera le meilleur choix lorsqu'on a pas de raison particulière d'insister sur le choix de l'une ou l'autre stratégie.*

c) Quel est l'effet de la clause *nowait* sur le déroulement d'une boucle *for* en OpenMP? Est-ce que ceci peut faire une différence appréciable? Expliquez? **(1 point)**

*La clause nowait retire la barrière (rendez-vous) qui serait insérée à la fin de la boucle. Certains fils d'exécution termineront donc possiblement avant d'autres et tous les calculs de la boucle ne seront donc pas encore finis. Si cela n'est pas un problème, par exemple parce que la boucle suivante utilise les mêmes données avec les mêmes fils, le retrait de cette barrière enlève le coût de la barrière elle-même. Plus important encore, ce retrait enlève aussi l'attente de chaque fil jusqu'à ce que le dernier fil ait terminé. L'impact peut donc être significatif.*

Le professeur: Michel Dagenais