Systèmes et supports d'exécution pour le calcul parallèle et distribué

Université de Bordeaux 1 — Master 2 SDRP, 2008 – 2009

Corrigé du Partiel

Alexandre DENIS

28 novembre 2008

Durée : 2 h. **Documents interdits.**
Les exercices sont indépendants les uns des autres. Le barème est donné à titre indicatif.
Les parties de code demandées doivent être rédigées en langage C.

1 Questions de cours (7 points)

1. Définissez brièvement :
   (a) système distribué

   Un système distribué est un système regroupant plusieurs ressources de calcul sans mémoire commune.

   (b) processeur superscalaire

   Un processeur superscalaire est un processeur doté de plusieurs unités (arithmétiques, logiques, flottantes, etc.) travaillant simultanément, permettant une exécution en pipeline *ce qui lui permet de commencer l'exécution d'une instruction alors que la précédente n'est pas terminée.*

   (c) processeur multithread

   Un processeur multithread est un processeur qui gère les changements de contexte en matériel, réduisant ainsi le surcoût du multithreading. On distingue les processeur multithreads à gros grain (CMT) fournissant une instruction de changement de contexte rapide (fenêtrage de registres, etc.) et les processeurs multithreads à grain fin (FMT) changeant de contexte automatiquement à chaque instruction.
   On classe parfois également dans cette catégorie les processeurs doté d'une technologie SMT, c'est-à-dire des processeurs superscalaires dotés de plusieurs unités de contrôle, gérant ainsi les contextes concurremment.

(d) machine SMP, machine NUMA

*Une machine SMP (*Symmetric Multi-Processor*) est une machine dotée de plusieurs processeurs identiques, accédant à une mémoire commune.*
*Une machine NUMA (*Non-Uniform Memory Access*) est une machine doté de plusieurs processeurs, où un banc mémoire est attaché à chaque processeur (ou à un groupe de processeur), et équipée d'un dispositif matériel permettant à chaque processeur d'accéder à la mémoire des autres processeurs, mais moins rapidement qu'à sa mémoire locale (d'où le* non-uniform*).*

2. Il existe actuellement de nombreux processeurs qui utilisent du parallélisme en interne.

(a) Pourquoi a-t-on introduit du parallélisme à l'intérieur des processeurs ? Justifiez.

Pendant longtemps, le facteur principal d'augmentation des performances des processeurs reposait sur l'augmentation de la fréquence d'horloge, rendue possible par l'amélioration de la finesse de gravure du silicum (loi de Moore). Cependant, depuis que les finesses de gravure sont passées sous les 110 nm, l'amélioration des finesses de gravure entraine une augmentation des courants de fuite, donc une augmentation du dégagement thermique. Ceci a entrainé une stagnation depuis plusieurs années des fréquences maximales entre 3 et 4 GHz.
Les fondeurs de processeurs ont donc décidé de se tourner vers le parallélisme pour améliorer malgré tous les performances de leurs processeurs.

(b) Quelles formes peut prendre le parallélisme à l'intérieur des microprocesseurs ?

processeurs vectoriels — *ce sont des processeurs qui travaillent directement sur des vecteurs de nombres, permettant par exemple l'addition de deux vecteurs en une seule opération.*

processeurs superscalaires — *ce sont des processeurs capables de démarrer l'exécution d'une instruction avant la fin de l'instruction précédente.*

processeurs SMT — *ce sont des processeurs superscalaires dotés de plusieurs unités de contrôle, gérant donc plusieurs contextes concurremment.*

processeurs multicore — *ce sont en réalité plusieurs processeurs complets intégrés directement au sein d'une seule puce.*

(c) Comment utilise-t-on, au niveau logiciel, ces différentes formes de parallélisme offertes par les microprocesseurs ?

processeurs vectoriels — *les applications doivent être vectorisées, soit en programmant les unités de calcul vectorielles en assembleur, soit par une vectorisation automatique par le compilateur (voir l'option* `-msse` *de* `gcc`*).*

processeurs superscalaires — *les applications exécutées par les processeurs superscalaires sont des applications séquentielles classiques, non modifiées.*

processeurs SMT et processeurs multicore — *les applications doivent être parallélisées à l'aide de threads, soit explicitement, soit automatiquement par un compilateur tel que OpenMP.*

3. Qu'appelle-t-on une construction d'intergiciel par abstraction ?

La construction d'intergiciels par abstraction est un principe de construction d'intergiciels, typiquement en couches logicielles superposées, où chaque couche présente une interface abstraite*, unifiée par rapport aux différentes variantes du type de ressource sous-jacente. Ce principe mène par exemple à une organisation avec des* pilotes *présentant chacun la même interface au-dessus de ressources différentes.*

4. Qu'appelle-t-on la *portabilité* ?

La portabilité est la capacité d'un code à utiliser différentes variantes d'un type donné de ressource.

2 Questions de réflexion (6 points)

1. Quels sont les aspects internes des bibliothèques de *threads* qui sont intrinsèquement non-portables (ou difficilement portables) ? Par rapport à quels types de ressources la question de la portabilité se pose-t-elle pour les *threads* ? Vous pourrez distinguer le cas des *threads* utilisateurs et des *threads* noyau.

   Les aspects non-portables dans l'implémentation de threads sont :
   – *les instructions atomiques nécessaires pour construire les* spinlocks*, dépendantes du processeur ;*
   – *les changements de contexte (pour un ordonnanceur noyau), les mécanismes d'attente (pour les ordonnanceurs utilisateur), la disposition de la pile en mémoire, dépendent du système d'exploitation.*

2. La bibliothèque de *threads* Marcel dispose d'une API qui lui est propre mais également d'une API `pthread`.

   (a) Que représente cette API `pthread` pour *Marcel* ?

   L'API `pthread` est conçue au départ comme une interface générique*, c'est-à-dire non-pas pour une application donnée mais pour être réutilisable. Le support de l'API `pthread` au-dessus de l'API propre de Marcel est une incarnation du mécanisme de* personnalité*.*

   (b) Quelle en est l'utilité ?

   La personnalité `pthread` de Marcel permet de porter facilement sur Marcel des applications multithread existantes, sans avoir besoin de modifier leur code source ni même de les recompiler.

3. La bibliothèque `GNUPth` (GNU Portable Thread) de Ralf S. Engelschall est une bibliothèque de threads écrite en C et 100 % portable. Elle a la double particularité de fonctionner uniquement en espace utilisateur et d'être non préemptive. À quoi peut-elle servir ? Expliquez en quelques lignes l'intérêt, l'utilité de threads utilisateurs non préemptifs.

   Les threads utilisateurs, même s'ils n'apportent pas de concurrence réelle, apportent un semblant de concurrence. Ceci est particulièrement utile pour les applications où il est naturel de raisonner en tâches indépendantes (par exemple : un thread pour le réseau, un thread d'affichage graphique, etc.). Par rapport à un entrelacement qui serait fait à la main, la programmation en est facilitée, et l'ordonnancement par nécessité assure une utilisation optimale du temps CPU disponible.

4. La notion de thread et la notion de tâche se distinguent par le fait que la création et l'instanciation d'un thread sont une seule et même étape alors que la création et l'intanciation d'une tâche sont deux étapes distinctes. Commentez en quelques lignes les bénéfices qu'offrent les tâches en dissociant ces deux opérations

   En dissociant la création et l'instanciation d'une tâche, le temps de création peut être inférieur au temps de création d'un thread et il est possible de disposer de tâches préallouées. Il en résulte que les tâches permettent d'exprimer le parallélisme de façon plus fine, en contrepartie d'un code plus verbeux.

3 Problème : compilateur OpenMP (7 points)

Nous étudions dans ce problème la parallélisation d'un code à l'aide d'OpenMP et la transormation de code faite par le compilateur.

Soit le code suivant calculant la transformée en ondelettes (DWT — *Discrete Wavelet Tranform*) d'un vecteur `input` de taille `n`. Le résultat est dans `output`. (*Note :* le contenu de `input` est détruit par cette fonction !)

```
void DWT(int*input, int*output, int n)
{
  int l;
  for(l = n / 2; l > 1 ; l = l / 2)
  {
    int i;
    for(i = 0; i < l; i++)
    {
      output[i]   = input[i*2] + input[i*2+1];
      output[i+l] = input[i*2] - input[i*2+1];
    }
    memcpy(input, output, l*2*sizeof(int));
  }
}
```

3.1 Parallélisation

1. Quelle(s) boucle(s) `for` peut être parallélisée(s) en OpenMP. Pourquoi ?

   Par raison d'efficacité, habituellement on parallélise la boucle la plus externe. Cependant, dans le code `DWT`, la boucle externe réprésente un calcul itératif, où chaque itération dépend du résultat de l'itération précédente. Pour respecter la dépendance de données, la seule boucle parallélisable est donc la boucle interne en `i`, où chaque itération écrit sur une zone disjointe des autres.

2. Donnez l'idée générale d'une parallélisation de ce code en OpenMP. Quelles variables doivent être partagées entre les *threads* et lesquelles doivent être locales ?

   La parallélisation à l'aide d'un compilateur OpenMP se fait en instrumentant le code à l'aide de `#pragma`. qui :
   - *délimitent la section parallèlé ;*
   - *déclarent quelle boucle paralléliser ;*
   - *déclarent quelles sont les variables globales (`input`, `output`, `l`) et les variables locales aux threads (`i`).*

3. Sur `P` processeurs, quelle est l'accélération attendue d'une version OpenMP ? Justifiez.

   Étant donné que les threads travaillent sur des parties disjointes du vecteur `output` avec un travail équitablement réparti, on peut s'attendre à une accélération de l'ordre de `P`. Cependant, le coût de synchronisation à chaque itération de la boucle en `l` et la partie non-parallélisée (`memcpy`) rendront l'accélération plus faible en pratique.

3.2 Transformation OpenMP → C

Le pragma OpenMP `for` peut recevoir un attribut optionnel `schedule` pour guider la répartition de la charge, pouvant prendre entre autres les valeurs suivantes :

`static` — pour une boucle allant de 0 à N-1, chaque *thread* reçoit le travail correspondant à `N/P` itérations de la boucle.

`dynamic` — chaque itération est considérée comme une tâche, les *threads* piochent du travail dans la liste de tâches jusqu'à ce que tout le travail soit effectué.

1. Proposez une version `pthread` du code `DWT` correspondant à ce qu'un compilateur OpenMP génère pour un ordonnancement `static`.

```
struct task_s {
  pthread_t tid;
  int*input;
  int*output;
  int b_num;
  int b_size;
  int l;
};
void*calc(void*arg)
{
  struct task_s t = (struct task_s*)arg;
  int i;
  for(i = t->b_num*t->b_size; i < (t->b_num+1)*t->b_size; i++) {
    t->output[t->i]      = t->input[t->i*2] + t->input[t->i*2+1];
    t->output[t->i+t->l] = t->input[t->i*2] - t->input[t->i*2+1];
  }
  return NULL;
}
void DWT(int*input, int*output, int n)
{
  int l;
  for(l = n / 2; l > 1; l = l / 2) {
    int max = (l > P) ? P : l;
    int j;
    struct tasks_s tasks[P] =
     { .input = input, .output = output, .l = l, .b_size = l/P };
    for(j = 0; j < max; j++) {
      tasks[j]->b_num = j;
      pthread_create(&tasks[j].tid, &calc, NULL, &tasks[j]);
    }
    for(j = 0; j < max; j++)
      pthread_join(&tasks[j].tid, NULL);
    memcpy(input, output, l*2*sizeof(int));
  }
}
```

2. Proposez une version `pthread` du code `DWT` correspondant à ce qu'un compilateur OpenMP génère pour un ordonnancement `dynamic`.

```
struct dwt_status_s {
  pthread_mutex_t lock;
  int*input;
  int*output;
  int l;
  int i;
};
void*worker(void*arg)
{
  struct dwt_status_s s = (struct dwt_status_s*)arg;
  int i;
  do {
    pthread_mutex_lock(&s->lock);
    i = s->i++;
    pthread_mutex_unlock(&s->lock);
    if(i < s->l) {
      s->output[i]      = s->input[i*2] + s->input[i*2+1];
      s->output[i+s->l] = s->input[i*2] - s->input[i*2+1];
    }
  }  while(i < s->l);
  return NULL;
}
void DWT(int*input, int*output, int n)
{
  int l;
  for(l = n / 2; l > 1; l = l / 2) {
    int j;
    struct pthread_t tids[P];
    struct dwt_status_s status =
     { .input = input, .output = output, .l = l, .i = 0 };
    pthread_mutex_init(&status->lock, NULL);
    for(j = 0; j < P; j++)
      pthread_create(&tids[j], &worker, NULL, status);
    for(j = 0; j < P; j++)
      pthread_join(&tids[j], NULL);
    pthread_mutex_destroy(&status->lock);
    memcpy(input, output, l*2*sizeof(int));
  }
}
```

3. Quelle est l'utilité du mode `dynamic` d'un point de vue général ?

L'ordonnancement `dynamic` est utile pour gérer convenablement l'équilibrage de charge sur des applications irrégulières, c'est-à-dire des applications pour lesquelles il n'est pas possible de prévoir dès la compilation le volume de données et la quantité de calcul à effectuer.

4. L'ordonnancement `dynamic` est-il nécessaire dans le cas du code `DWT` ? Justifiez.

La parallélisation du code `DWT` porte sur une seule boucle dont les bornes sont connues. Il n'y a donc pas besoin d'ordonnancement `dynamic`. Un ordonnancement `static` avec des blocs de taille `l/P` donnera déjà un bon équilibrage.