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

Université de Bordeaux 1 — Master 2 SDRP, 2006 – 2007

Corrigé du partiel

Alexandre DENIS

29 novembre 2006

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 : microprocesseurs et parallélisme (8 points)

1. Rappelez brièvement la définition de :
   (a) parallélisme

   Un système parallèle et un système dans lequel on a introduit de la concurrence dans le but d'accélérer l'exécution.

   (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 contexte 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 local (d'où le* non-uniform*).*

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

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 pendant 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.

Est-ce une tendance durable ? Justifiez ?

Même si IBM annonce pour bientôt un Power 6 à 5 GHz, les limites physiques de montée en fréquence sont presque atteintes. La tendance du parallélisme plutôt que la montée en fréquence dans les processeurs semble donc durable.

3. Énumérez et décrivez les formes de parallélisme qui existent 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 intégrés directement au sein d'une seule puce.*

4. 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.*

2 Portabilité des *threads* (7 points)

1. 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 sous-jacentes du type de ressource ressource.*
> *Ce principe mène par exemple à une organisation avec des* pilotes *présentant chacun la même interface au-dessus de ressources différentes.*

2. Qu'appelle-t-on la *portabilité* ?

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

3. Quels sont les aspects internes des bibliothèques de *threads* qui sont non-portables (ou difficilement portables) ? Par rapport à quels types de ressources la question de la portabilité se pose-t-elle pour les *threads* ?

> *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.*

4. Comment situez-vous les bibliothèques de *threads* dans les différents modèles d'abstraction ? Pourquoi a-t-on procédé de cette façon ? Que représente alors l'API `pthread` ?

> *Dans la majorité des cas, les bibliothèques de threads sont développées de façon* spécifique *pour un système d'exploitation donnée (car elles y sont intimement liées, surtout pour les cas noyau et hybrides), et utilisent un système de* pilotes *pour la portabilité par rapport au processeur (il suffit d'écrire le* spinlock *en assembleur pour chaque proceseur).*
> *L'API* `pthread` *est alors une interface* générique*, c'est-à-dire une interface assurant l'accès à une infrastructure de portabilité, et conçue non-pas pour une application donnée mais pour être réutilisable.*

3 Problème : multiplication de matrices parallèle (8 points)

Nous étudions dans ce problème la parallélisation à l'aide de *threads* d'une multiplication de matrices, telle qu'elle peut être mise en oeuvre à l'intérieur d'une bibliothèque d'algèbre linéaire comme LAPACK. La version séquentielle est donnée ci-dessous, qui calcule le produit de la matrice **`A`** par la matrice **`B`**, place le résultat dans la matrice **`C`**, toutes les matrices étant carrées de taille **`N`** fixe :

```
double A[N][N], B[N][N], C[N][N];
int i, j, k;
for(i = 0; i < N; i++)
{
  for(j = 0; j < N; j++)
  {
    double sum = 0.0;
    for(k = 0; k < N; k++)
      sum += A[i][k] + B[k][j];
    C[i][j] = sum;
  }
}
```

3.1 Parallélisation directe

1. Nous allons paralléliser la boucle la plus externe (sur **`i`**). Pourquoi ?

   La boucle sur `i` correspond au parcours des colonnes de la matrice `C` ; la boucle sur `j` correspond au parcours des lignes. Il est donc possible de paralléliser l'une ou l'autre de ces boucles car elles travaillent en écriture sur des données disjointes. La parallélisation de la boucle sur `k` demanderait l'utilisation d'un mutex ce qui serait inefficace.
   Ayant le choix entre `i` et `j`, on préfèrera paralléliser la boucle la plus externe pour générer des tâches de plus gros grain, ce qui a pour effet de réduire le surcoût engendré par la parallélisation (création des threads, synchornisation, etc.).

2. Donnez l'idée générale d'une parallélisation de ce code en OpenMP.

   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 (`A`, `B`, `C`) et les variables locales aux threads (`i`, `j`, `k`).*

3. Soit **`P`** le nombre de processeurs. On supposera que **`N`** est multiple de **`P`**. Donnez une version de ce code parallélisée à l'aide de `pthread`.

```
double A[N][N], B[N][N], C[N][N];
struct task { int begin; int end; };
/* calcule un bloc de lignes de la matrice C */
void*worker(struct task*arg)
{
  int i, j, k;
  for(i = arg->begin; i < arg->end; i++)
  {
    for(j = 0; j < N; j++)
    {
      double sum = 0.0;
      for(k = 0; k < N; k++)
        sum += A[i][k] + B[k][j];
      C[i][j] = sum;
    }
  }
  free(arg);
  return NULL;
}
/* code principal : lance et attend les threads */
int i;
const int block = N / P; /* nombre de lignes par processeur */
pthread_t threads[P];
for(i = 0; i < P; i++)
{
  struct task*t = malloc(sizeof(struct task));
  t->begin = i * block;
  t->end   = (i + 1) * block;
  pthread_create(&threads[i], NULL, &worker, t);
}
for(i = 0; i < P; i++)
{
  pthread_join(&threads[i], NULL);
}
```

4. Quelle est l'accélération attendue ? Justifiez.

Étant donné que les threads travaillent sur des parties disjointes de la matrice `C` *avec un travail équitablement réparti, on peut s'attendre à une accélération de l'ordre de P, si on néglige le coût de création des threads.*

3.2 Équilibrage de charge

Nous allons ajouter à notre code une gestion de l'équilibrage de charge.

1. Qu'est-ce que l'équilibrage de charge ? Dans quel cas est-il important ?

L'équilibrage de charge consiste en une répartition équitable de la charge de travail entre les différents processeur, de façon à ce qu'à tout moment tous les processeurs soient utilisés.
Il est important de s'en préoccuper dans les applications irrégulières, pour lesquelles on ne peut pas répartir la charge équitablement a priori *par un simple découpage des données.*

2. Est-il crucial de gérer l'équilibrage de charge dans le cas présent ?

Dans le cas présent, il n'y a pas absolument besoin de gérer dynamiquement l'équilibrage de charge car l'application concernée est parfaitement régulière. Un simple découpage en parts égales de la matrice `C` *mène à une répartition équitable des tâches.*
Le problème peut néanmoins se poser sur une machine NUMA où le temps de calcul ne dépend pas uniquement du nombre d'opérations mais aussi de la localisation des données (matrices `A`*,* `B`*, et* `C`*).*

3. Proposez une version parallèle `pthread` qui gère l'équilibrage de charge. On pourra considérer comme granularité de tâche une ligne de la matrice **`C`**.

Chaque thread traite une ligne à la fois. Ainsi, si les processeurs calculent à des vitesses différentes, le temps où le déséquilibre se produit sera inférieur au temps de calcul d'une ligne.

```
double A[N][N], B[N][N], C[N][N];
int nextline = 0; /* prochaine ligne à calculer */
pthread_mutex_t mutex;
/* calcule des lignes de la matrice C */
void*worker(void*arg)
{
  while(1)
  {
    int i, j, k;
    pthread_mutex_lock(&mutex);
    i = nextline++;
    pthread_mutex_unlock(&mutex);
    if(i >= N)
      return NULL;
    for(j = 0; j < N; j++)
    {
      double sum = 0.0;
      for(k = 0; k < N; k++)
        sum += A[i][k] + B[k][j];
      C[i][j] = sum;
    }
  }
  return NULL;
}
/* code principal : lance et attend les threads */
pthread_t threads[P];
pthread_mutex_init(&mutex, NULL);
for(i = 0; i < P; i++)
{
  pthread_create(&threads[i], NULL, &worker, NULL);
}
for(i = 0; i < P; i++)
{
  pthread_join(&threads[i], NULL);
}
```