Université de Bordeaux, Master Informatique
Examen du jeudi 16 avril 2015

IN8W16
Durée : 1h30 – Sans document

Programmation Multicœur et GPU

Barème indicatif : 2 - 1 - 1 - 5 - 2 - 3 - 2 - 1 - 3

MPI (2 pts)

Question 1 Quelles sont les différences entre les fonctions `MPI_Isend()` et `MPI_Send()` ? Décrire une situation où l'emploi `MPI_Isend()` est particulièrement avantageux.
Question bonus : donner un second cas d'utilisation sans rapport avec le premier.

Optimisation de code séquentiel (2 pts)

Le code suivant place dans `C` la somme de `A` et de la transposée de `B`.

```
int A[DIM][DIM];
int B[DIM][DIM];
int C[DIM][DIM];
...
for (int i = 0; i < DIM; i++)
   for (int j = 0; j < DIM; j++)
      C[i][j] = A[i][j] + B[j][i];
```

On suppose que le cœur dispose d'un cache de 128 ko dont les lignes font 64 octets (il y a donc 2048 entrées de 64 octets). Un entier est codé sur 4 octets, une ligne de cache contient donc 16 entiers. Ce cache est géré par la politique LRU (évincement de la ligne de cache la moins récemment utilisée). On s'intéresse au nombre de défauts de cache provoqués par l'exécution de cet algorithme à froid (aucune donnée n'est présente dans le cache). On note ce nombre *Défauts(DIM)* pour des tableaux `DIM` × `DIM`.

Question 2 Donner une estimation de Défauts(128) et Défauts(4096). Justifier soigneusement[1].
Question bonus : Quelle est la valeur la plus grande de `DIM` pour laquelle aucune donnée n'est chargée deux fois dans le cache ?

Question 3 Quelle technique algorithmique mettre en œuvre pour optimiser ce code ? On ne demande pas d'écrire le code.

Parallélisation du problème des n-dames via OpenMP (10 pts)

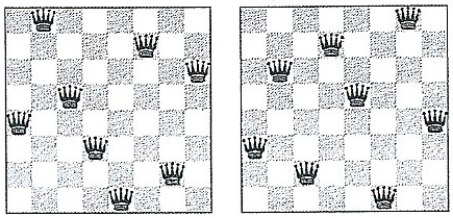

FIGURE 1 – Deux solutions au problème des 8-dames

Notre objectif est de paralléliser un programme, fort naïf, comptant le nombre de configurations de `DIM` dames sur un échiquier `DIM` × `DIM` sans qu'aucune d'elles ne se menacent. À titre d'illustration, la Figure 1 montre deux façons de poser 8 dames sur un échiquier de façon à ce que chaque dame soit seule à la fois sur sa rangée, sur sa colonne et sur sa diagonale. Le programme à paralléliser est présenté en Figure 2.

1. Une matrice carrée de dimensions 128 pèse 64 ko et occupe donc 1k entrées ; une matrice carrée de dimensions 4096 pèse 64Mo et occupe 1M entrées.

```
1  #define DIM 16

3  typedef bool echiquier[DIM][DIM];

5  static bool ok (int rangee, int colonne, echiquier e)
   {
7  ...
   }
9
   void
11 ndames (int rangee, echiquier e, int *cpt)
   {
13   for (int colonne = 0; colonne < DIM; colonne++)
       if (ok (rangee, colonne, e))          // la case n'est pas menacee
15     {
         if (rangee == DIM - 1)              // derniere rangee
17         (*cpt)++;
         else
19       {
           e[rangee][colonne] = true;        // placer une dame
21         ndames (rangee + 1, e, cpt);
           e[rangee][colonne] = false;       // retirer la dame
23       }
       }
25 }

27 int
   main ()
29 {
     echiquier e;
31   int cpt = 0;

33   memset (e, 0, sizeof(echiquier));       // echiquier vide
     ndames (0, e, &cpt);
35   printf ("%d\n",cpt);

37   return EXIT_SUCCESS;
   }
```

Ce programme repose sur l'algorithme du parcours en profondeur d'abord. On part d'un échiquier vide (ligne 33) puis on appelle la fonction `ndames()`. Celle-ci essaye de placer les dames rangée après rangée : lors d'un appel `ndames(rangee,e,cpt)` on explore en largeur (ligne 13) toutes les colonnes de la rangée `rangee` pour tester si l'on peut y disposer une dame et, le cas échéant, on y pose une dame (ligne 20) pour lancer en profondeur le calcul sur la rangée suivante (ligne 21). Au retour de l'appel récursif on nettoie la case (ligne 22) avant de passer à la colonne suivante. Le nombre de configurations découvertes est incrémenté lorsqu'on arrive à placer une dame sur la dernière rangée (ligne 17).

FIGURE 2 – Code à paralléliser

Le code donné en Figure 2 est, par essence, similaire à celui du *voyageur de commerce* étudié lors de nos TP. Nous allons donc appliquer les mêmes techniques de parallélisation : parallélisation du seul premier niveau de récursivité puis parallélisation imbriquée via l'utilisation d'équipes de threads puis de tâches parallèles. On portera une attention toute particulière aux variables partagées, on les traitera au moyen des techniques vues en cours (duplication des données, mutex, instruction atomique, réduction).

Question 4 Écrire une fonction `ndames_par_1(int *cpt)` qui parallélise à l'aide de directives OpenMP le traitement de la première rangée. On aura intérêt à réutiliser astucieusement la fonction `ndames()` sans la modifier.
Note : l'étudiant sûr de lui - i.e. maitrisant la parallélisation du voyageur de commerce - pourra répondre directement à la Question 5 qui généralise cette question.

Exemple de programme utilisant `ndames_par_1()` :

```
int
main ()
{
  int cpt = 0;
  ndames_par_1 (&cpt);
  printf ("%d\n", cpt);

  return EXIT_SUCCESS;
}
```

Question 5 Écrire une fonction `ndames_par_prof (int prof, int rangee, echiquier e, int *cpt)` qui parallélise le calcul jusqu'à la profondeur de récursivité `prof` en créant récursivement des équipes de threads imbriquées et en distribuant le travail à l'aide de la directive OpenMP `for` (il ne s'agit pas d'utiliser la clause `collapse`). On pourra :

— supposer que `prof < DIM-1` ;
— utiliser `memcpy(dest,src,sizeof(echiquier))` pour recopier le contenu d'un échiquier ;
— réutiliser astucieusement la fonction `ndames()` sans la modifier.

Exemple de programme utilisant `ndames_par_prof()` :

```
int
main ()
{
  echiquier e;
  int cpt = 0;

  omp_set_nested (1);
  memset (e, 0, sizeof(echiquier));
  ndames_par_prof (4, 0, e, &cpt);      // parallelisation sur 4 niveaux
  printf ("%d\n",cpt);

  return EXIT_SUCCESS;
}
```

Question 6 Écrire une fonction `ndames_par_task (int prof, int rangee, echiquier e, int *cpt)` qui parallélise le calcul à l'aide de tâches OpenMP imbriquées jusqu'à la profondeur de récursivité `prof`. Donner le programme principal contenant la création des threads et le lancement du calcul.

OpenCL (6 pts)

Question 7 Dans certains noyaux OpenCL, il est fait appel à la primitive :
`barrier(CLK_LOCAL_MEM_FENCE)`
Que fait cette instruction ? Quelle est sa portée (i.e. quels sont les *work items*, appelés aussi *threads*, impliqués) ? Indiquez, en donnant un exemple, dans quelles situations il est impératif d'y faire appel.

On s'intéresse à un noyau qui calcule, pour chaque ligne d'une matrice, la somme des éléments de la ligne. Le résultat est donc un vecteur, dont la $i^{\text{ème}}$ composante contient la somme des éléments de la ligne i.
La matrice est de dimension $[LINES][COLS]$, et le vecteur de dimension $LINES$.
Voici un noyau OpenCL qui tente d'implémenter ce calcul. Il est exécuté par $TILE \times LINES$ *work items*, regroupés en *workgroups* de taille $TILE \times 1$. Autrement dit, un workgroup de $TILE$ threads s'occupe de chaque ligne de la matrice.

```
__kernel void sumline(__global float *matrix,
                      __global float *vector)
{
  int x = get_global_id(0);
  int y = get_global_id(1);
  int xloc = get_local_id(0);
  __local float tile[TILE];

  tile[xloc] = 0.0;

  for (int c = xloc; c < COLS; c += TILE)
    tile[xloc] += matrix[y * COLS + c];

  if (x == 0) {
    float sum = 0.0;

    for (int i = 0; i < TILE; i++)
      sum += tile[i];

    vector[y] = sum;
  }
}
```

Question 8 Il manque une instruction dans le noyau pour que celui-ci soit correct. Expliquez laquelle et indiquer à quel endroit il faut la placer.

Question 9 Expliquez en quoi la seconde partie du code (à partir de `if (x == 0)`) est assez maladroite. Donnez une version plus efficace de cette portion de code.