Examen

20 Mai 2016 – Durée : 2 heures.
Documents autorisés : *polycopiés et notes de cours et de TDTP.*

- Les calculatrices, baladeurs et autres appareils électroniques sont interdits. Les téléphones portables doivent être éteints et rangés dans les sacs.
- Cet énoncé contient 5 pages.
- Merci d'éviter l'usage du crayon mine (crayon à papier).
- Le barème est donné à titre indicatif.

Exercice 1 – Echauffement MPI (2 points)

Voici un programme MPI lancé avec 5 processus :

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char *argv[]){
5    int my_rank, i, nb_procs;
6    MPI_Status status;
7    char s;
8    char r[20];
9
10   MPI_Init(&argc, &argv);
11   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
12   MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
13
14   switch (my_rank){
15   case 0 : s='h'; break ;
16   case 1 : s='e'; break ;
17   case 2 : s='l'; break ;
18   case 3 : s='l'; break ;
19   case 4 : s='o'; break ;
20   }
21
22   MPI_Send(&s, 1, MPI_CHAR, 0 /* destinataire */, 99 /* tag */, MPI_COMM_WORLD);
23
24   if (my_rank == 0) {
25     for (i=0; i<nb_procs; i++){
26       MPI_Recv(r+i, 1, MPI_CHAR, MPI_ANY_SOURCE /* emetteur */,
27                99 /* tag */, MPI_COMM_WORLD, &status);
28     }
29     r[nb_procs] = 0;
30     printf("%d a recu : %s\n", my_rank, r);
31   }
32
33   MPI_Finalize();
34   return 0;
35 }
```

1. Ce programme a-t-il un comportement déterministe ? Justifiez.

MPI ANY SOURCE
→ Non : l'ordre de réception n'est pas imposé (MPI ANY SOURCE) et les caractères reçus sont écrits dans la chaîne de caractères "r" dans les processus ∅ dans un ordre indéterminé suivant l'ordre de réception des msg

2. Qu'affiche ce programme ?

∅ a reçu hhelol" avec les lettres "hello" dans un ordre quelconque

3. Quelle communication collective MPI permet de réaliser le même schéma de communication (à savoir le même échange de données entre les processus) ?

opération Gather (MPI_GATHER())

Exercice 2 – Suite de l'échauffement MPI (2 points)

1. Quelle est la différence entre `MPI_Isend` et `MPI_Send` ?
Pourquoi cette différence a-t-elle été introduite dans le standard MPI ?

Différence : MPI_Isend() non bloquant (retourne immédiatement)
MPI_Send() bloquant (retourne quand on peut réécrire dans le buffer)

But : permet le recouvrement des communications par du calcul

Nom : Prénom: Numéro étudiant :

Exercice 3 – Borne théorique (2 points)

Un algorithme séquentiel requiert 38 tâches, chaque tâche ayant le même temps d'exécution (égal à une unité de temps). Son graphe de précédence a une hauteur de 18.

1. Quelle est le temps d'exécution de cet algorithme sur une infinité de processeurs ?

18 unités de temps (correspondant à la hauteur du graphe de précédence)

2. Donnez une borne maximum du temps d'exécution de cet algorithme sur une machine à 4 processeurs.

Théorème de Brent $t_4 \leq t_\infty - \frac{q-c}{p} = 18 + \frac{38-18}{4} = 23$

Exercice 4 – Transport de neutrons (14 points)

Nous souhaitons effectuer une simulation parallèle de transport de neutrons à l'aide d'une méthode de type Monte Carlo. Nous étudions ici un modèle simplifié en 2D (voir figure 1) [1]. Une source émet des neutrons contre une plaque homogène d'épaisseur H et de hauteur infinie (ce qui ramène ce problème 2D à un problème 1D). Un neutron peut être réfléchi par la plaque, absorbé dans celle-ci ou transmis à travers celle-ci. Nous souhaitons calculer les pourcentages d'occurrence de ces trois cas.

Deux constantes décrivent l'interaction des neutrons dans la plaque : la « section efficace » de capture C_c et la « section efficace » de diffusion C_s. La section efficace totale est $C = C_c + C_s$.

La distance L qu'un neutron parcourt dans la plaque avant d'interagir avec un atome est modélisée par une distribution exponentielle de moyenne $1/C$. Si u est un nombre aléatoire issu d'une distribution uniforme sur $[0.1[$, la formule $L = -(1/C)\ln u$ fournit un nombre aléatoire de la distribution exponentielle appropriée.

Quand un neutron interagit avec un atome dans la plaque, sa probabilité de « rebondir » sur l'atome est C_s/C, alors que la probabilité qu'il soit absorbé par l'atome est C_c/C. Nous pouvons utiliser un nombre aléatoire d'une distribution uniforme sur $[0, 1[$ pour déterminer le résultat d'une interaction neutron-atome.

Si un neutron est diffusé, il a la même probabilité de partir dans chaque direction. Ainsi son nouvel angle de direction D (mesuré en radians) peut être modélisé par une variable aléatoire distribuée uniformément dans $[0, \pi[$. Comme la plaque a une hauteur infinie, nous ne faisons en effet pas de distinction entre un rebond vers le haut et un rebond vers le bas. Pour un angle de direction D, la distance correspondante sur l'axe « x » que le neutron parcourt dans la plaque entre deux collisions est $L\cos(D)$.

La simulation d'un neutron se poursuit jusqu'à ce que

- le neutron soit absorbé par un atome ;
- ou que la position en « x » du neutron soit inférieure à 0, ce qui signifie que le neutron a été réfléchi par la plaque ;
- ou que la position en « x » du neutron soit supérieure à H, ce qui signifie que le neutron a été transmis à travers la plaque.

L'algorithme général est donné en algorithme 1. Les résultats de cette simulation sont les nombres de neutrons réfléchis, absorbés et transmis, ainsi que les positions de tous les neutrons absorbés.

1. D'après le livre « Parallel Programming in C with MPI and OpenMP », de M. Quinn.

Algorithme 1 Simulation de transport de neutrons.

Pré-condition : C, C_c, C_s : La distance moyenne entre les interactions neutron/atome est $1/C$. C_c et C_s sont les composantes absorbantes et diffusantes de C.
Pré-condition : H : épaisseur de la plaque
Pré-condition : L : distance parcourue par le neutron avant la collision
Pré-condition : d : direction du neutron ($0 \leq d \leq \pi$)
Pré-condition : x : position de la particule ($0 \leq x \leq H$)
Pré-condition : n : nombre de neutrons
Pré-condition : $r = 0, b = 0, t = 0$: nombre de neutrons réfléchis, absorbés et transmis
Pré-condition : $absorbes$: tableau d'au plus n éléments contenant les positions des neutrons absorbés
Pré-condition : $cont$: variable à $Vrai$ tant que le neutron est simulé

```
1:  Initialisation du générateur de nombres aléatoires par l'appel : srand48(0)
2:  Pour i = 0 à n − 1 faire
3:    d = 0
4:    x = 0
5:    cont = Vrai
6:    Tant que cont faire
7:      L = −1/C ∗ ln(drand48())
8:      x = x + L ∗ cos(d)
9:      Si x < 0 Alors      /* (réfléchi) */
10:       r = r + 1
11:       cont = Faux
12:     Sinon si x ≥ H Alors      /* (tramsmis) */
13:       t = t + 1
14:       cont = Faux
15:     Sinon si drand48() < C_c/C Alors      /* (absorbé) */
16:       absorbes[b] = x
17:       b = b + 1
18:       cont = Faux
19:     Sinon
20:       d = drand48() ∗ π
21:     Fin si
22:   Fin tant que
23: Fin pour
```


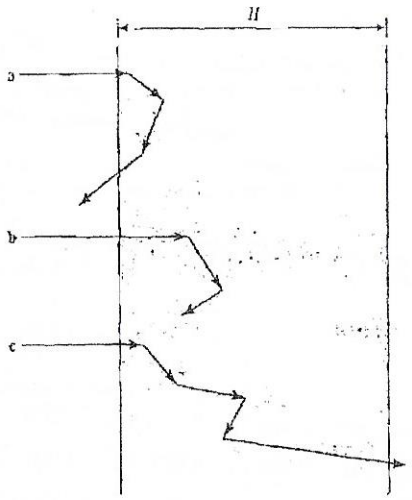

FIGURE 1 – Un neutron rencontrant un milieu homogène de longueur H peut être (a) réfléchi, (b) absorbé, ou (c) transmis (figure extraite du livre « Parallel Programming in C with MPI and OpenMP », de M. Quinn).

On rappelle que la fonction drand48() retourne un nombre à virgule flottante généré pseudo-aléatoirement et uniformément dans $[0.0; 1.0[$, et que la fonction srand48(s) prend en argument un entier s qui permet d'initialiser (de façon différente pour chaque valeur de s) la suite de nombres aléatoires générés par drand48().

On vise ici à simuler un grand nombre de neutrons, par exemple $n = 500000000$.

1. (6 points) On s'intéresse tout d'abord à une parallélisation sur plusieurs nœuds CPU, en mode multi-processus avec le standard MPI, à l'aide d'un équilibrage de charge statique. Les positions des neutrons absorbés seront collectées, de façon contigue mais sans ordre particulier entre les processus MPI, par le processus de rang ROOT. Ce processus de rang ROOT récupérera aussi les nombres totaux de neutrons réfléchis, absorbés et transmis. On veillera par ailleurs à ce que les nombre aléatoires générés par chaque processus MPI ne soient pas tous identiques.

   Ecrire l'algorithme parallèle correspondant en précisant les éventuelles difficultés à prendre en compte. Pour les fonctions MPI, on précisera la fonction employée et ses paramètres significatifs mais on pourra omettre la syntaxe en C.
2. (1 point) Quel peut être l'inconvénient d'un tel équilibrage de charge ? Comment devrait évoluer ce risque lorsque n devient « très grand » ?
3. (5 points) On souhaite désormais introduire du parallélisme de type multi-thread, à l'aide d'OpenMP, dans cette implémentation MPI en considérant qu'on dispose de plusieurs nœuds CPU multi-cœurs.

   Indiquez les éventuelles difficultés à prendre en compte pour mettre en œuvre cette implémentation parallèle MPI-OpenMP, détaillez les directives OpenMP à utiliser ainsi que leur emplacement dans l'algorithme, précisez le niveau de support des threads nécessaire pour votre implémentation MPI et présentez les possibles avantages et inconvénients de cette implémentation hybride MPI-OpenMP par rapport à la version MPI.
4. (1 point) Que pensez-vous de l'efficacité de la vectorisation d'un tel code sur CPU ?
5. (1 point) Quel(s) paramètre(s) va(vont) influer sur l'intensité de calcul de cette application ?

 19 avril 2017

Ex 4 :

Q1] Variables locales :
nl : nombre local de neutron
nl = 0, rl = 0, tl = 0 nombre locaux de neutron reflechis, transmis et absorbés
absorbes : tableau des positions des neutrons absorbés
NP : nombre total de processus MPI
p : rang du processus MPI courant
status : de type MPI_Status

Q3] → (1)
```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &P);
MPI_Comm_size(MPI_COMM_WORLD, &NP);
Srand48(p)
nl = (n/NP) + (p < n%NP ? 1 : 0)
```
Allocation mémoire du tableau absorbés (nl elts, buff pour root n elts)

Q3] → (2)
```
For i=0 à nl-1 faire
  de = 0
  x = 0
  cont = Vrai
  Tant que cont faire
    L = -1/C x ln(drand48())
    x = x + L x cos(de)
    si x<0 Alors
      nl = nl + 1
      cont = Faux
    Sinon si x > H alors
      tl = tl + 1
      cont = Faux
    Sinon si drand48() < Cc/C Alors
```
Q3] → (3)
```
      absorbes[bl] = x
      bl = bl + 1
      cont = Faux
    Sinon
      de = drand48() x π
  Fin si
  Fin tant que
Fin For
```

MPI_Reduce (&nl, &n, 1, MPI_INT, MPI_SUM, ROOT, MPI_COMM_WORLD);
↳ MUL, MIN, MAX

MPI_Reduce (&tR, &t, 1, MPI_INT, MPI_SUM, ROOT, MPI_COMM_WORLD);
Si r ≠ ROOT Alors
  MPI_Send (absorbes, bl, MPI_FLOAT, ROOT, 0 /* tag */, MPI_COMM_WORLD);
Sinon
  b = bl
  Pour i = 0 à N-2 Faire
    MPI_Recv (absorb+b, n/NP+1 /* max */, MPI_FLOAT, MPI_ANY_SOURCE
    , 0 /* tag */, MPI_COMM_WORLD, &status);
    MPI_Get_count (&status, MPI_FLOAT, &bR);
    b += bl
  Fin Pour
Fin si
Libération mémoire du tableau 'absorbés'
MPI_Finalize()

Q2) Désynchronisation des threads sur les CPU ne faisant pas le même calcul, chaque machine prend une trajectoire différente
Plus on augmente, plus ce risque diminue

Q3)
① MPI_Init_thread (& argc, & argv, MPI_THREAD_FUNNELED, & provided);
Si provided < MPI_THREAD_FUNNED ALORS
  Erreur "MPI_THREAD_FUNNED not provided by the MPI implementation"
  exit(1);
Fin si
[...]

② [#pragma omp parallel for private(X, cnt, bll, d, b) schedule(dynamic, taille_chunk_a_paramétrer) reduction (+:nl, +:tl)
  ↳ variable temporaire (privée en OpenMP)
[...]

③ [#pragma omp atomic capture
  bll = bl++
  absorbes[bll] = X
  cnt = Faux

Q4) Très compliqué car il ne faut pas utiliser de masques sectorisés sur plusieurs particules

Q5)
- si Ce/C ↓ chacun ↓ d'être absorbé donc augmente le temps calculé
- si MFP chacun ↓ d'être transmit