Adrien Cambillau – Corentin Raoult

8INF856
Programmation sur architectures parallèles
Devoir 2

Adrien Cambillau - Corentin Raoult

1.(a)
P-Square-Matrix-Multiply(A,B,n)

```
parallèle for i=1 to n
    parallèle for j=1 to n
        C[i,j]=0
        parallèle for k=1 to n réduction(+:C)
            C[i,j] = C[i,j] + A[i,k]*B[k,j]
return C
```

1.(b)

Pour le travail on a 3 parcours à $\Theta(n)$ ce qui donne donc une durée de :

$T_1 = \Theta(n) + \Theta(n) + \Theta(n)$

soit $T_1 = \Theta(n^3)$

2. Analyse de l'implémentation de l'algorithme récursif du tri par fusion

Pour paralléliser cet algorithme nous avons dans un premier temps créer les threads à l'aide de la directive compilateur « *#pragma omp parallel* ». On utilise cette directive juste une fois avant le premier appel de la foncton *«triFusionParallele() »*. Ensuite, dans cette fonction, on utilise les directives *« #pragma omp single nowait »* et *« #pragma omp task »* pour créer une tâche à chaque fois qu'on appelle la fonction *«triFusionParallele() »* en divisant le tableau initial par 2 à chaque itération. Grâce à l'utilisation des « task » on n'as pas besoin de créer des threads à chaque fois et la charge de travail est équilibrée.

Ci-dessous le tableau des résultats en secondes de la durée de l'algorithme de tri par fusion en séquentiel et en parallèle en fonction de la taille du tableau n.

n	Séquentiel	Parallèle
2	0.000004	0.001785
4	0.000006	0.001462
8	0.000009	0.001293
16	0.000016	0.001288

32	0.000029	0.001187
64	0.000058	0.001203
128	0.000178	0.001185
256	0.000274	0.001185
512	0.000493	0.001184
1024	0.001049	0.001444
2048	0.002171	0.001446
4096	0.003343	0.001433
8192	0.007130	0.001498
16384	0.016486	0.001915
32768	0.029183	0.002804
65536	0.053000	0.005904
131072	0.109439	0.009344
262144	0.226159	0.016392
524288	0.460941	0.027386
1048576	0.957917	0.051272
2097152	1.943377	0.098930
4194304	3.971822	0.183063
8388608	8.289898	0.360697
16777216	16.732428	0.748042

Grâce à ce tableau comparatif, on voit que l'algorithme parallélisé est plus efficace que le séquentiel quand la taille du tableau excède les 2048 éléments (environ).

Cette différence doit venir du temps de création des threads, lorsque n est suffisamment petit, la création des threads prend plus de temps que le traitement par la version non parallèle.

Lorsque n double, le temps non parallèle double, ce qui fait qu'avec un n assez grand, l'algorithme parallèle devient plus efficace.

3.(a)

```
SP2(T, n)
    parallèle for i = 0 to n
        somme = 0
        parallèle for j = 0 to i réduction(+:somme)
            somme += T[j]
            S[i] = somme
    return S
```

Adrien Cambillau – Corentin Raoult

n = 1024
SP1 :
moyenne 0.0005
min 0.0004
max 0.0006

SP2 :
moyenne 0.015
min 0.006
max 0.017

n = 102400
SP1 :
moyenne 0.0011
min 0.0008
max 0.0013

SP2 :
moyenne 9.25
min 9.20
max 9.30

Comme on peut le constater la version récursive de l'algorithme est plus performante quel que soit le n choisis alors que la version itérative montre une grande baisse de performance lorsque n devient grand.

Cela vient sûrement du fait que la version récursive découpe le tableau, chaque morceau de tableau ainsi découpé est traité par un thread, alors que pour la version itérative il n'y a qu'un seul thread qui fait la somme et met le résultat dans le tableau, rendant ainsi l'algorithme peu efficace lorsque n est grand.

Pour accéder au code :

utilisateur : 8inf856-16

mot de passe : HVfiLx