Parallelisation et modèles de programmation
Exercices

Caner Candan
M2 MIHP

25 novembre 2010

Table des matières

1 Produit matriciel

1.1 Enoncé

Matrice : Décrire un algorithme PRAM pour le modèle EREW qui utilise $O(n^3)$ processeurs pour multiplier deux matrices de taille $n * n$. [1]

1.2 Solution

Une solution consisterait à diffuser dans un premier temps toutes les données des matrices à multiplier à chaque processus puis de faire le produit des matrices en prenant le soin que chaque processus utilise les données qui leur est propre.

1.2.1 La diffusion

En respectant le modèle EREW, les matrices A et $B \in K_{n*n}$ sont copiées, dans un premier temps, n fois dans les cubes A' et $B' \in K_{n*n*n}$. Ceci permet de reserver une dimension du cube à un processus et éviter toute lecture concurrente.

L'algorithme 1 présente le pseudo-code de la fonction de diffusion qui prend comme donnée en entrée une matrice $M \in K_{n*n}$ et retourne un cube $M' \in K_{n*n*n}$. Elle est exprimée en 2 étapes. La première consiste à affecter, en utilisant $O(n^2)$ processeurs, chaque point de la matrice dans la première dimension du cube. La deuxième effectue la copie en diffusion [2] de chaque point de la matrice situé à la première dimension du cube vers les autres dimensions du cube en utilisant $O(logn)$ processeurs et pour une matrice entière en $O(n^2 * logn)$.

```
   Données: M ∈ K_{n*n}, n ∈ N
   Résultat: M' ∈ K_{n*n*n}
 1 début
 2     || pour i ← 0 à n faire
 3         || pour j ← 0 à n faire
 4             m'(i, j, 0) ← m(i, j)
 5     || pour s ← 0 à log_2 n faire
 6         || pour h ← 0 à s^2 faire
 7             || pour i ← 0 à n faire
 8                 || pour j ← 0 à n faire
 9                     m'(i, j, 2^s + h) ← m'(i, j, h)
10 fin
```

Algorithm 1: Copie de matrice en diffusion

1.2.2 Le produit

Toujours en respectant le modèle EREW, le produit matriciel prend en paramêtre les cubes A' et $B' \in K_{n*n*n}$ tel que chaque processus utilise sa dimension de cube respective pour lire les données de a_{ij} et b_{ij}.

1. $c_{ij} = \sum_{k=0}^{n} a_{ik} b_{kj}$: `http://fr.wikipedia.org/wiki/Produit_matriciel`
2. Hot potato routing : `http://en.wikipedia.org/wiki/Hot_potato_routing`

L'algorithme 2 présente le pseudo-code de la fonction de produit matriciel qui prend comme donnée en entrée les cubes A' et $B' \in K_{n*n*n}$ et retourne une matrice $C \in K_{n*n}$. L'affectation des valeurs dans la matrice C se fait en $O(n^2)$ et le produit de c_{ij} se fait en $O(n)$.

```
   Données: A, B ∈ K_{n*n*n}, n ∈ N
   Résultat: C ∈ K_{n*n}
1  début
2  |  ‖ pour i ← 0 à n faire
3  |  |  ‖ pour j ← 0 à n faire
4  |  |  |  c(i, j) ← Σ_{k=0}^{n} a(i, k, p) * b(k, j, p)
5  fin
```

Algorithm 2: Produit matriciel

1.2.3 L'implémentation

Après avoir décrit la fonction de diffusion et le produit matriciel respectant tous deux le modèle EREW, nous allons voir l'implementation d'un produit de deux matrices A et $B \in K_{n*n}$ produisant une troisième matrice $C \in K_{n*n}$.

L'algorithme 3 décrit l'implémentation.

```
   Données: A, B ∈ K_{n*n}, n ∈ N
   Résultat: C ∈ K_{n*n}
1  début
2  |  A' ← diffusion(A, n)
3  |  B' ← diffusion(B, n)
4  |  C ← produit(A', B')
5  fin
```

Algorithm 3: Implémentation du produit matriciel

2 Découpage d'un tableau binaire

2.1 Enoncé

Découpage d'un tableau : Soit A un tableau de longueur n dont les éléments sont soit des 0 *soit des* 1*. Concevez un algorithme EREW de complexité* $O(logn)$ *utilisant* $O(n)$ *processeurs pour ranger tous les éléments* 1 *à la droite du tableau tout en maintenant leur ordre relatif (propriété de stabilité des éléments).*

Hint : effectuer un calcul de prefixe pour déterminer quel devrait être la position de chaque élément.

2.2 Solution

Notre tableau étant binaire, contenant que des 0 et 1, il devient facile de compter le nombre de 1 en sommant toutes les valeurs du tableau pour ensuite déterminer à quel index placer les 0 et 1.

Toute fois pour respecter l'énoncé et avoir une complexité en $O(logn)$, nous allons devoir utiliser la fonction calculant la somme des prefixes.

2.2.1 Déterminer l'index

L'algorithme 4 permet de déterminer l'index qui consituera la frontière entre les 0 et 1 dans le tableau. Pour cela il prend en paramètre le tableau binaire $B \in K_n$ ainsi que la taille du tableau $n \in N$.

Données: $B \in K_n, n \in N$
Résultat: $i \in K$
début
  $i \leftarrow n - calcul\_somme\_prefix(B)$
fin

Algorithm 4: Trouver l'index

2.2.2 Créer le tableau binaire

L'algorithme 5 prend en paramètre l'index et la taille et crée un tableau binaire trié en fonction de l'index, 0 à gauche et 1 à droite.

Données: $i, n \in N$
Résultat: $B' \in K_n$
début
  pour $j \leftarrow 0$ **à** i **faire**
    $B'(i) \leftarrow 0$
  pour $j \leftarrow i$ **à** n **faire**
    $B'(i) \leftarrow 1$
fin

Algorithm 5: Création du tableau binaire trié

2.2.3 Implémentation

L'algorithme 6 illustre la solution.

Données: $B \in K_n$
Résultat: $B' \in K_n$
1 **début**
2 | $i \leftarrow trouver\_index(B)$
3 | $B' \leftarrow creer\_tableau\_binaire(i)$
4 **fin**

Algorithm 6: Découpage d'un tableau binaire

3 Premier élément non-nul

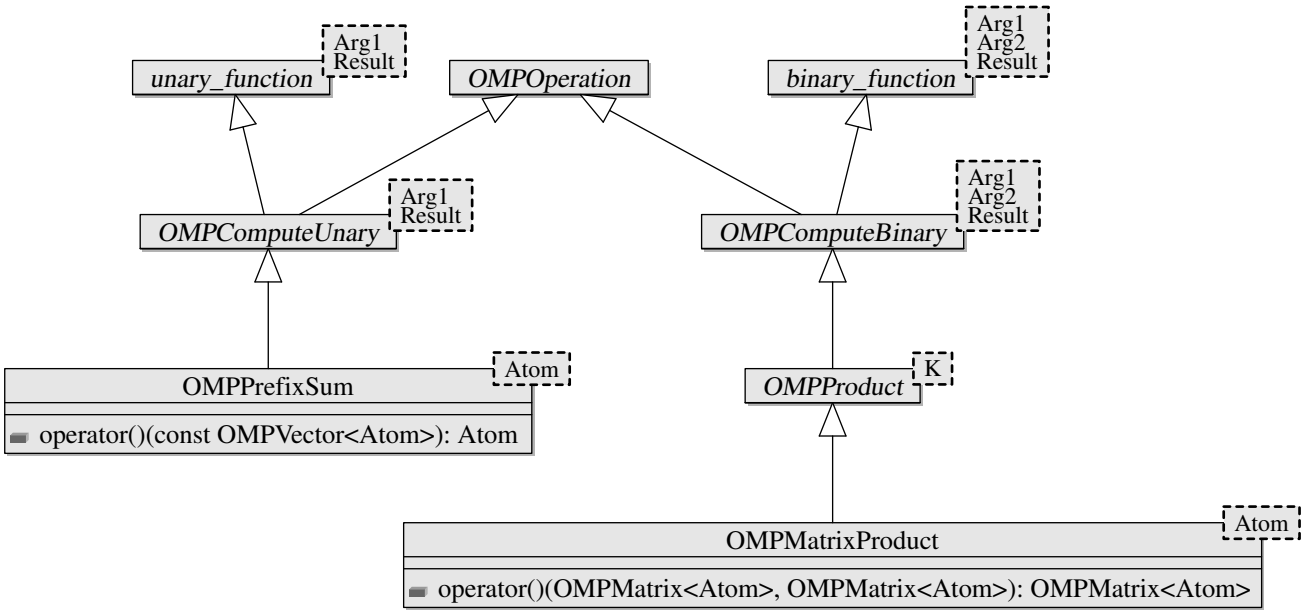
3.1 Enoncé

Premier élément non-nul : Soit A un tableau de longueur n dont les éléments sont soit des 0 *soit des* 1. *Concevez un algorithme CRCW de complexité $O(1)$ qui utilise $O(n)$ processeurs.*

4 Design

Tous les exercices ont été developpés en C++ en utilisant le paradigme de programmation objet et le design qui suit a été élaboré pour regrouper les différents composants de calculs. Néanmoins une recherche a été effectué pour determiner les limites de OpenMP et C++. Il semblerait que les conteneurs fournit par la STL [3] ne soient pas threadsafe [4]. Une étude [5] donne une liste non-exaustive des précautions à prendre lors de l'utilisation de OpenMP en C++. Cependant la version OpenMP 3.0 apporte des concepts nouveaux permettant l'utilisation notemment des iterateurs. On peut ainsi, par exemple, s'affranchir de simple indexeur de tableau et étendre la parallelisation sur des listes chainées.

4.1 Diagramme de classes



Deux grandes familles d'operations coexistent. Les operations de calcul à paramêtre unique "OMPComputeUnary" et celles à deux paramêtres "OMPComputeBinary". Tous deux renvoient un résultat. Elles sont préfixés de "OMP" pour OpenMP [6]. Le calcul de somme des prefixes prenant un seul paramêtre en entrée "OMPVector" et renvoyant un "Atom", cette classe se situe dans la famille "OMPComputeUnary". Le produit matriciel se situe, quant à lui, dans la famille "OMPComputeBinary", cette classe prend deux paramêtres "OMPMatrix" et retourne le même type. On peut imaginer d'autres opérations comme, par exemple, un produit vectoriel.

4.2 Surcharge d'opérateur

Compte tenu des possibilités offertes par le langage C++, comme la surcharge des opérateurs, l'opérateur $*$ se voit ainsi surchargé dans la classe "OMPMatrix" afin d'appeler implicitement la classe "OMPMatrixProduct" pour le calcul du produit matriciel. Ainsi en écrivant le code $C = A * B$ ceci revient à écrire $C = OMPMatrixProduct()(A, B)$.

3. Standard Template Library : `http://www.sgi.com/tech/stl/`
4. On dit qu'un programme ou qu'une portion de code est thread-safe s'il fonctionne correctement durant une exécution simultanée par plusieurs threads (processus légers). `http://fr.wikipedia.org/wiki/Threadsafe`
5. C++ and OpenMP : `http://www.compunity.org/events/pastevents/parco07/parco_cpp_openmp.pdf`
6. Open Multi-Processing : `http://openmp.org/`

5 Mesure de performance

Je profite de ce rapport pour parler d'un travail qui m'a été demandé en entreprise. Nous utilisons le framework EO [7] pour développer nos algorithmes évolutionnaires [8]. Le sujet consistait à implémenter, au framework EO, un parellisme à mémoire partagée en utilisant OpenMP.

5.1 Préambule

Avant de commencer il est important de préciser que les EA [9] travaillent sur une population d'individus aussi appelé échantillon. Un individu étant représenté par un point dans l'échantillon, la complexité d'un problème est définit par le nombre de dimensions pour chaque individu.

5.2 Identifier les ressources les plus utilisées

Un test de profiling a été executé afin d'identifier les ressources les plus utilisées dans le framework EO. Le test nous a permi d'identifier une fonction qui est utilisée par une grande majorité des opérateurs [10] EO. Il s'agit de la fonction "apply". Elle prend en paramètre une population d'individus et un opérateur. Cette fonction va itérer sur tous les individus de la population et appliquer l'opérateur. Il peut être intéressant d'optimiser cette fonction. Nous allons nous limiter à transformer le code sequentiel en parallèle.

5.3 Pseudo-code de la fonction apply

L'algorithme 7 prend en paramètre une population ainsi qu'un opérateur à appliquer à chaque individu.

```
   Données: P ∈ K_n, F ∈ Operateur
1  début
2  |  pour i ← 0 à n faire
3  |  |  F(P(i))
4  fin
```

Algorithm 7: La fonction apply

5.4 La fonction en parallèle

L'algorithme 8 transforme la fonction "apply" en parallèle en utilisant le modèle PRAM CREW [11] et $O(n)$ processeurs pour parcourir tous les individus de la population.

```
   Données: P ∈ K_n, F ∈ Operateur
1  début
2  |  || pour i ← 0 à n faire
3  |  |  F(P(i))
4  fin
```

Algorithm 8: La fonction omp_apply

7. Evolving Object, `http://eodev.sf.net`
8. Algorithme évolutionnaire : `http://fr.wikipedia.org/wiki/Algorithme_evolutionnaire`
9. Algorithmes évolutionnaires
10. opérateurs de sélection et variation
11. Concurantial Read Exclusive Write

5.5 Speed-up

Après avoir crée la fonction alternative employant le parallisme à mémoire partagée, appelé "omp_apply", nous allons étudier une solution de mesure du speed-up.

L'équation suivante présente une méthode de mesure du speed-up [12] et est implémentée dans l'algorithme 9.

$$Mesure\ du\ Speedup = r \sum_{k=0,l=0}^{P,D} S_{p_{kl}}$$

Nous considérons les paramètres suivants :
n : le nombre de processus
p : la taille minimum de la population
P : la taille maximum de la population
$popStep$: le pas d'iteration de la population
d : la taille minimum de la dimension
D : la taille maximum de la dimension
$dimStep$: le pas d'iteration de la dimension
r : le nombre d'exécution pour chaque combinaison de p et d

```
   Données: p, P, popStep, d, D, dimStep, r ∈ N
 1 début
 2     pour k ← p à P faire
 3         pour l ← d à D faire
 4             pour m ← 0 à r faire
 5                 Ts ← 0
 6                 Tp ← 0
 7                 début
 8                     t1 ← omp_get_wtime()
 9                     ... code sequentiel avec k et l exécuté m fois ...
10                     apply( ... )
11                     t2 ← omp_get_wtime()
12                     Ts ← t2 − t1
13                 fin
14                 début
15                     t1 ← omp_get_wtime()
16                     ... code parallel avec k et l exécuté m fois ...
17                     omp_apply( ... )
18                     t2 ← omp_get_wtime()
19                     Tp ← t2 − t1
20                 fin
21                 ... on conserve le speed-up Ts/Tp pour k et l ...
22 fin
```

Algorithm 9: La fonction de mesure du speedup

12. $S_p = \frac{T_1^*}{T_p}$: http ://en.wikipedia.org/wiki/Speedup

5.6 Mesures

En prenant en compte les paramètres décrits précédement, nous allons lancer les tests sur deux architectures matérielles différentes.

Les jeux de paramètres suivants ont été utilisés :
n : en fonction du nombre de coeurs disponible
p : 1
P : 1000
$popStep$: 100
d : 1
D : 1000
$dimStep$: 100
r : 100

5.6.1 Graphique

Pour visualiser l'évolution du speed-up, nous utilisons un outil de génération de graphique [13], avec les données produits par les tests.

5.6.2 Double coeurs

Pour ce premier test, un ordinateur personnel équipé de 2 coeurs [14] a été utilisé.

FIGURE 1 – Test sur 2 coeurs

5.6.3 8 coeurs

Pour ce deuxième test, un serveur équipé d'un processeur i7 utilisant l'hyperthreading [15], permettant d'avoir 8 coeurs virtuel au lieu de 4, a été utilisé.

5.7 Dynamicité

5.8 Optimisation du compilateur

5.8.1 Auto paralleliseur

13. Utilisation de matplotlib en python pour générer des boites à moustache
14. Intel Centrino vPro cadensé à 2.40GHz
15. Hyperthreading : `http://en.wikipedia.org/wiki/hyperthreading`