Travaux dirigés n°3

Xavier JUVIGNY

5 février 2017

Table des matières

1 Produit scalaire

À partir du fichier `dotproduct.cpp`, paralléliser le produit scalaire à l'aide des threads de C++ 2011 puis en OpenMP.

Comparer les temps de calculs des deux approches.

Calculer l'accélération (en OpenMP ou avec les pthreads, au choix) du produit scalaire parallèle avec deux, trois ou quatres threads. Comment interprétez-vous le résultat ?

2 Tri bitonique

Le tri bitonique est un des tris les plus performants dans un contexte parallèle. Il se base sur une suite dite *bitonique*.

Définition 1 *Une suite $a_0, a_1, \ldots, a_{n-1}$ est dite **bitonique** si il existe un élement $a_i, 0 < i < n-1$ tel qu'une des conditions suivantes est satisfaite :*

- $a_0 \leq a_1 \leq \ldots \leq a_i \geq a_{i+1} \geq \ldots \geq a_{n-1}$ *ou*
- $a_0 \geq a_1 \geq \ldots \geq a_i \leq a_{i+1} \leq \ldots \leq a_{n-1}$ *ou*
- *un décalage d'indice devrait satisfaire une des deux relations ci–dessus.*

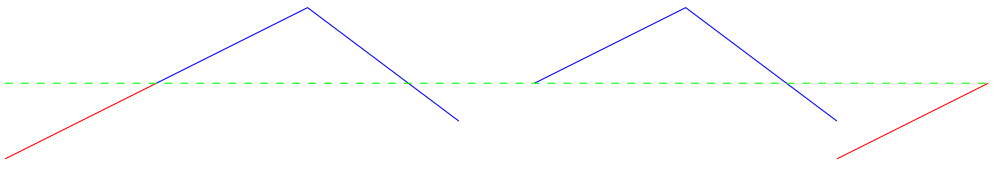


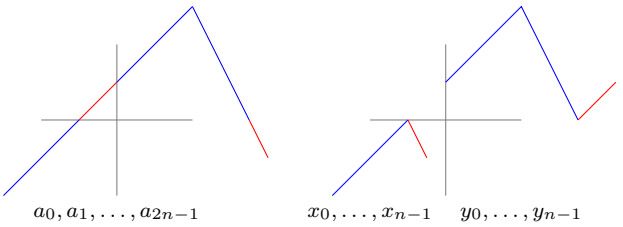
FIG. 1 : Exemples de suites bitoniques

L'algorithme de tri se base sur le théorème de division bitonique :

Théorème 1 *Soit une suite bitonique $a_0, a_1, \ldots, a_{2n-1}$. On définit les sous–suites :*

$$\begin{aligned} x_i &= \min(a_i, a_{i+n}) \text{ pour } i = 0, \ldots, n-1 \\ y_i &= \max(a_i, a_{i+n}) \text{ pour } i = 0, \ldots, n-1 \end{aligned}$$

Alors les deux suites $x_0, x_1, \ldots, x_{n-1}$ et $y_0, y_1, \ldots, y_{n-1}$ sont des suites bitoniques et chaque éléments de la suite x_i sont plus petits que les éléments de la suite y_i.

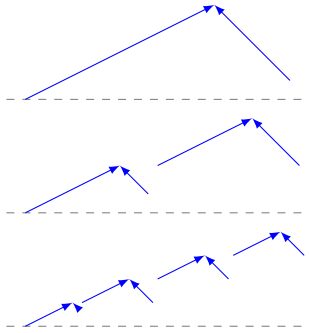


(a) Avant le split (b) Après le split

FIG. 2 : Exemple de split bitonique

2.1 Tri d'une suite bitonique

Soit une suite bitonique de n éléments. Si on applique le théorème récursivement :



Après $\log(n-1)$ pas, chaque suite bitonique possédera seulement deux éléments qui pourront être triés trivialement.

L'algorithme complet de tri consistera donc à :

1. Trier les $fracn2$ premiers éléments dans l'ordre croissant et les derniers $fracn2$ éléments dans l'ordre décroissant
2. Trier la suite bitonique résultante en $\log n$ étapes.

Comment trier $\frac{n}{2}$ éléments ? $\Rightarrow$ Récursivement

La complexité de l'algorithme de tri est de :

- $\log(n)$ étapes ;
- Chaque pas i demande i sous-pas

Donc le nombre de pas est donc :

$$\text{Nombre de pas} = \sum_{i=1}^{log(n)} i = \frac{1+\log(n)}{2} log(n)$$

2.2 Travail à faire

Utiliser la version du tri fournie en séquentiel pour trier un tableau d'entier puis un tableau de vecteurs selon leurs normes L2.

Paralléliser à l'aide des threads de C++ 2011 l'algorithme de tri puis calculer l'accélération obtenue pour le tri sur les entiers puis pour le tri sur les vecteurs.

Comment interprétez-vous la différence d'accélération entre le tri sur les entiers et le tri sur les vecteurs ?

3 Produit matrice–matrice

Dans le fichier `TestProduct.cpp`, on définit deux matrices A et B à partir de deux couples de vecteurs u_A, v_A, u_B, v_B comme : $A = u_A.v_A^T$ et $B = u_B.v_B^T$ soit $A_{ij} = u_{Ai} \times v_{Aj}$ et $B_{ij} = u_{Bi} \times v_{Bj}$.

On calcule le produit matrice–matrice $C = A.B$ à l'aide d'un produit matrice–matrice classique (ce qui demande $2n^3$ opérations arithmétiques) et on vérifie si le résultat est bon à l'aide de la formule : $C = A.B = u_A.v_A^T.u_B.v_B^T = (v_A|u_B)u_A.v_B^T$ soit $C_{ij} = (v_A|u_B)u_{Ai}.v_{Bj}^T$, ce qui nécessite $2n+2.n^2$ opérations arithmétiques (dont $2n$ pour le produit scalaire)

On se propose d'optimiser le produit matrice–matrice "classique".

3.1 Optimisation du produit matrice–matrice "naïf"

Après avoir mesurer le temps nécessaire à calculer le produit matrice–matrice à l'aide du code d'origine de la fonction `prodSubBlocks` dans le fichier `ProdMatMat.cpp`, permutez y les boucles en i,j et k (en vous persuadant avant que cela ne change rien au calcul) jusqu'à obtenir un temps optimum pour le produit matrice–matrice.

Sachant que les cœfficients des matrices A, B et C sont stockés par colonnes (l 'indice de ligne est l'indice variant le plus vite dans le tableau linéaire en mémoire), pouvez vous expliquer le résultat obtenu ?

3.2 Parallélisation du produit matrice–matrice classique

En conservant dans la fonction `prodSubBlocks` le code ayant donné le meilleurs temps de calcul, recopier ce code dans `ompnaive_prodMatMat` (en l'adaptant) et paralléliser le à l'aide d'OpenMP (ou en thread C++ 2011 pour les plus courageux).

Calculer l'accélération obtenue sur 2, 4 et 8 threads. Comment interprétez vous le résultat ?

3.3 Optimisation par bloc du produit matrice–matrice

Afin d'exploiter au mieux la mémoire cache, on se propose de transformer notre produit matrice–matrice naïf à un produit par bloc.

L'idée est de décomposer les matrices A, B et C en sous–blocs matriciels :

$$A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ A_{N1} & & & A_{NN} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} & \dots & B_{1N} \\ B_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ B_{N1} & & & B_{NN} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1N} \\ C_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ C_{N1} & & & C_{NN} \end{pmatrix}.$$

où A_{IJ}, B_{IJ} et C_{IJ} sont des sous–matrices.

On fait le produit par bloc pour calculer le bloc matriciel C_{IJ} :

$$C_{IJ} = \sum_{K=1}^{N} A_{IK}.B_{KJ}$$

Mettre en œuvre ce produit dans la fonction `block_prodMatMat` (en utilisant pour le produit la fonction `prodSubBlocks` que vous avez optimisé dans la première question).

Calculez et comparez le temps obtenu avec le temps obtenu dans la question 1.

3.4 Parallélisation du produit matrice–matrice par bloc

Paralléliser le produit matrice–matrice par bloc en parallèle à l'aide d'OpenMP ou des threads C++ 2011.

Calculer l'accélération obtenue avec 2,4 et 8 threads. Comment expliquez vous ce résultat ?

4 Ensemble de Bhudda

L'ensemble de Bhudda est un ensemble dérivé de l'ensemble de Mandelbrot. Au lieu de dessiner des pixels en fonction du nombre d'itérations qu'il a fallu pour diverger, on augmente de un chaque pixel par lesquels passent les valeurs successives de chaque suite dont on a détecté la divergence.

Paralléliser à l'aide d'OpenMP ou des threads C++ 2011 le calcul et le tracé de l'ensemble de Bhudda à partir du code `bhudda.cpp`.