# Travaux dirigés n°3

Xavier JUVIGNY

November 25, 2019

## Contents



## 1 Produit scalaire

- À partir du fichier `dotproduct.cpp`, paralléliser le calcul du produit scalaire à l'aide de directives OPENMP;
- Calculer l'accélération du produit scalaire en faisant varier le nombre de threads à l'aide de la variable d'environnement `OMP_NUM_THREADS`. Comment expliquez-vous le résultat que vous obtenez pour l'accélération ?
- Écrire une deuxième version du produit scalaire mais cette fois ci parallélisée à l'aide des threads C++ 2011;
- Comparer les temps de calcul pour les deux approches;

## 2 Produit matrice–matrice

Soient $A$ et $B$ deux matrices définies à l'aide de deux couples de vecteurs $\{u_A, v_A\}$ et $\{u_B, v_B\}$ :

$$\left\{ \begin{array}{rcl} A & = & u_A.v_A^T \text{ soit } A_{ij} = u_{A_i}.v_{A_j} \\ B & = & u_B.v_B^T \text{ soit } B_{ij} = u_{B_i}.v_{B_j} \end{array} \right.$$

On calcule le produit matrice–vecteur C=A.B à l'aide d'un produit matrice–matrice plein ( complexité de $2.n^3$ opérations arithmétiques ) et on valide le résultat obtenu à l'aide de l'expression sous forme de produit tensoriel de $A$ et $B$ :

$$\begin{array}{rcccl} C & = & A.B & = & \left(u_A.v_A^T\right).\left(u_B.v_B^T\right) = u_A\left(v_A^T.u_B\right)v_B^T \\ & = & u_A\left(v_A|u_B\right)v_B^T & = & \left(v_A|u_B\right)u_A.v_B^T \end{array}$$

Soit

$$C_{ij} = \left(v_A|u_B\right)u_{A_i}.v_{B_j}$$

ce qui nécéssite en tout $2.n + 2.n^2$ opérations arithmétiques ( dont $2.n$ opérations pour le produit scalaire ).

On se propose par étape de paralléliser le produit matrice–matrice fourni dans le fichier `ProdMatMat.cpp` :

1. Mesurer le temps de calcul du produit matrice–matrice donné;

2. **Première optimisation cache :** Permutez les boucles en $i, j$ et $k$ jusqu'à obtenir un temps optimum pour le calcul du produit matrice–matrice ( et après vous être persuader que cela ne changera rien au calcul ). Expliquez pourquoi la permutation des boucles obtenues est bien la meilleurs façon d'ordonner les boucles.

3. **Première parallélisation :** À l'aide d'OpenMP, paralléliser le produit matrice–matrice. Mesurez le temps obtenu à variant le nombre de threads à l'aide de la variable d'environnement `OMP_NUM_THREADS`. Calculez l'accélération et le résultat obtenu en fonction du nombre de threads.

4. **Deuxième optimisation de la mémoire cache** : Pour pouvoir exploiter au mieux la mémoire cache, on se propose de transformer notre produit matrice–matrice "scalaire" en produit matrice–matrice par bloc ( on se servira pour le produit "bloc–bloc" de la meilleurs version **séquentielle** du produit matrice–matrice obtenu précédemment ).

   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–blocs possédant une taille fixée ( par le programmeur ).

   Le produit matrice–matrice se fait alors par bloc. Pour calculer le bloc $C_{IJ}$, on calcul

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

   Mettre en œuvre ce produit matrice–matrice en séquentiel puis faire varier la taille des blocs jusqu'à obtenir un optimum ( aux alentours de 128 ). Comparer le temps pris par rapport au produit matrice–matrice "scalaire". Comment interprétez vous le résultat obtenu ?

5. **Parallélisation du produit matrice–matrice par bloc** : À l'aide d'OpenMP, parallélisez le produit matrice–matrice par bloc puis mesurez l'accélération parallèle en fonction du nombre de threads. Comparez avec la version scalaire paralléliser. Comment expliquez vous ce résultat ?

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

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

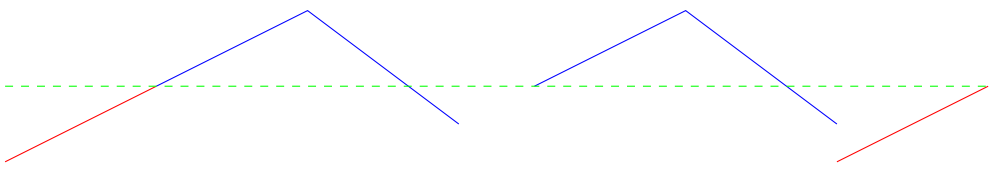


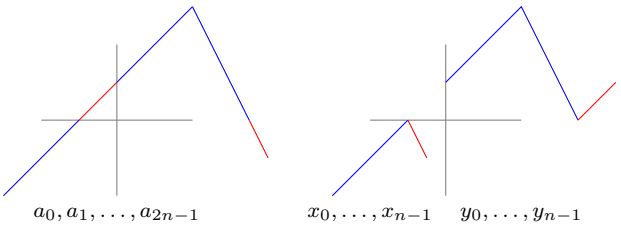
Figure 1: Exemples de suites bitoniques

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 $y_i$.*

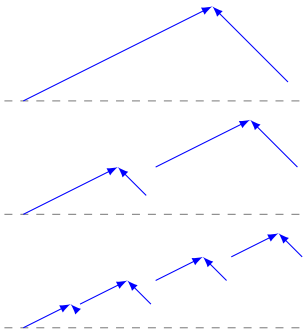



(a) Avant le split (b) Après le split

Figure 2: Exemple de split bitonique

### 3.1 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 $\frac{n}{2}$ premiers éléments dans l'ordre croissant et les derniers $\frac{n}{2}$ 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 :

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

### 3.2 Travail à faire

Utiliser la version du tri fourni 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 ?

# 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 nécessaires à la détection éventuelle de divergence de la suite, on augmente l'intensité de chaque pixel par lesquels une suite divergente est passée ( on ne fait rien pour les suites convergentes ). À l'aide d'OpenMP, paralléliser le code Bhudda donné dans le fichier `bhudda.cpp`