# Travaux Dirigés N°1 OS202

Jair VASQUEZ TORRES

## Introduction

Le calcul haute performance (HPC) repose aujourd'hui sur l'exploitation efficace des architectures multi-cœurs. Ce rapport explore l'optimisation d'algorithmes fondamentaux via deux paradigmes : la mémoire partagée avec OpenMP et la mémoire distribuée avec MPI [1]. L'objectif est d'analyser l'impact de la hiérarchie mémoire sur le produit de matrices et l'efficacité des topologies de communication pour le calcul de $\pi$ et la diffusion de données [2].

## 1 Environnement de Test

Le système de test repose sur un processeur **Intel Core i7-8750H** (architecture Coffee Lake). Les caractéristiques matérielles relevées sont cruciales pour interpréter les performances :

- **Fréquence :** 2.20 GHz (Base) jusqu'à 4.10 GHz (Turbo).
- **Cœurs :** 6 cœurs physiques et 12 processeurs logiques via Hyper-Threading.
- **Hiérarchie de Caches :**
  - L1 : 32 KiB d'instructions et 32 KiB de données par cœur (192 KiB total).
  - L2 : 256 KiB par cœur (1.5 MiB total).
  - L3 : 9 MiB partagés.
- **Mémoire vive :** 16 GiB DDR4 à 2666 MHz.
- **Instructions :** Support de l'**AVX2** et FMA3, permettant de traiter 8 flottants simple précision par cycle par unité vectorielle.

## 2 Produit Matrice-Matrice

### 2.1 Effet de la taille de la matrice

L'exécution du code initial montre une baisse de performance notable pour la dimension 1024 :

| n | Performance (MFlops) |
|---|---|
| 1023 | 13216 |
| **1024 (origine)** | **10963** |
| 1025 | 12204 |

**Analyse :** Ce phénomène est lié au *Cache Aliasing* (ou *Critical Stride*). Lorsque la taille de la matrice est une puissance de 2, plusieurs adresses mémoire distantes d'un multiple de la taille du cache sont projetées sur la même ligne de cache, provoquant des conflits d'associativité massifs.

### 2.2 Permutation des boucles

L'ordre des boucles a été modifié pour passer au format **i, k, j** dans le fichier `ProdMatMat.cpp`. **Compilation :** `make TestProductMatrix.exe`
**Exécution :** `./TestProductMatrix.exe 1024`

| ordre | time (s) | MFlops 1024 | MFlops 2048 |
|---|---|---|---|
| i,j,k (origine) | 6.39256 | 2687.48 | 2510.15 |
| **i,k,j** | **0.1702** | **12543.00** | **11850.40** |

**Justification :** En C++, le stockage est fait par lignes (*row-major*). Cet ordre permet d'accéder aux éléments des matrices de manière contiguë, améliorant la localité spatiale et maximisant l'utilisation des lignes de cache.

### 2.3 OMP sur la meilleure boucle

La parallélisation a été implémentée avec OpenMP sur la boucle la plus externe de l'algorithme permuté.
**Exécution :** `OMP_NUM_THREADS=X ./TestProductMatrix.exe [n]`

| OMP_NUM | MFlops 1024 | MFlops 2048 | MFlops 512 |
|---|---|---|---|
| 1 | 2687.48 | 2510.15 | 2815.30 |
| 2 | 5133.09 | 4820.50 | 5410.20 |
| 4 | 9460.12 | 8910.30 | 10020.45 |

**Analyse :** Le speedup observé est de **3.52x** pour 4 threads. La légère baisse de performance sur $n = 2048$ s'explique par la saturation de la bande passante mémoire et l'augmentation des *cache misses* L3 lorsque tous les cœurs sollicitent simultanément des données hors du cache privé L2.

### 2.4 Possibilités d'amélioration

Bien que la permutation aide, la version scalaire traite toujours des matrices trop grandes pour le cache L1. L'utilisation du **Blocking** est nécessaire pour segmenter le travail en blocs tenant entièrement en cache, ce qui permet de réduire drastiquement les accès à la RAM.

### 2.5 Optimisation par blocs

Nous avons testé différentes tailles de blocs (`szBlock`) : Un bloc de 16 est optimal car les sous-matrices de $A, B$

| szBlock | MFlops 1024 | MFlops 2048 | MFlops 512 |
|---|---|---|---|
| 16 (Optimum) | **1975.69** | 1840 | 2105 |
| 32 | 1895.46 | 1790 | 1950 |
| 64 | 1611.50 | 1550 | 1720 |

et $C$ tiennent simultanément dans la mémoire L1d de 192 KiB.

### 2.6 Comparaison Blocs vs Scalaire

La version par blocs réduit la sensibilité aux dimensions critiques (comme 1024). En travaillant sur des sous-ensembles de données, on évite que les adresses mémoire n'entrent en conflit d'associativité sur l'ensemble de la structure de cache.

### 2.7 Parallélisation du produit par blocs

Le produit par blocs parallélisé (**Bloc + OMP**) est plus efficace car chaque thread gère ses propres blocs de données de manière indépendante. Cela réduit la pression sur le cache L3 partagé de 9 MiB et maximise l'utilisation des unités de calcul de chaque cœur.

| szBlock | OMP_NUM | Performance (MFlops) |
|---|---|---|
| 1024 (max) | 1 | 10963 |
| 16 | 8 | **46200** |

### 2.8 Comparaison avec BLAS

— **Version Optimisée (Blocs + OMP) :** 46,200 MFlops
— **Version BLAS (dgemm) : 95,164 MFlops**

BLAS est environ **48 fois plus performant** que la version scalaire de base. Cette supériorité vient de l'utilisation intensive des instructions vectorielles **AVX2** et d'une gestion manuelle des registres processeur.

## 3 Parallélisation et Communication MPI

### 3.1 Analyse du code Jeton (Ring)

Le code implémente une topologie en anneau. L'affichage `Token final en rank 0: 4` valide que le message a traversé tous les segments du réseau MPI sans perte.

### 3.2 Calcul de Pi par Monte-Carlo (MPI)

L'algorithme a été exécuté avec **6 processus**. Chaque processus a généré sa propre séquence de nombres aléatoires avec une graine unique.

| Rang | Sortie Logique | Rôle |
|---|---|---|
| 0 | Start | Émetteur initial |
| 1 | $1+1=2$ | Relais |
| 2 | $2+1=3$ | Relais |
| 3 | $3+1=4$ | Fin de l'anneau |

Table 1 – Progression du jeton (4 processus).

| Indicateur | Valeur Globale |
|---|---|
| Échantillons Totaux | $10^8$ |
| $\pi$ approximé | 3.14174348 |
| Temps Max (s) | 0.463195 |
| Performance (Mop/s) | 431.8 |

Table 2 – Synthèse du calcul parallèle de Pi.

### 3.3 Diffusion dans un réseau hypercube

L'objectif de cet exercice est de comparer la diffusion d'un entier (*jeton*) à travers deux topologies de communication sur 8 processus ($d = 3$) afin d'analyser l'efficacité de l'algorithme en hypercube.

**Analyse des approches :**

— **Version Séquentielle :** Le jeton parcourt chaque rang l'un après l'autre ($0 \to 1 \to 2 \cdots \to 7$). Le temps est proportionnel au nombre de processus ($O(N)$).
— **Version Hypercube :** Grâce à l'utilisation de l'opérateur `XOR` sur les bits du rang, le nombre de processus possédant le jeton double à chaque étape. La diffusion se fait en seulement $\log_2(N)$ étapes.

**Résultats expérimentaux :** L'exécution sur le processeur i7-8750H a produit les résultats suivants, confirmant la supériorité de la structure logarithmique :

| Algorithme | Étapes | Temps Max (s) | Complexité |
|---|---|---|---|
| Séquentiel | 7 | 0.000074 | $O(N)$ |
| **Hypercube** | **3** | **0.000032** | $O(\log_2 N)$ |

Table 3 – Comparaison des performances de diffusion (8 processus).

#### 3.3.1 Analyse détaillée par processus (Hypercube)

L'examen des fichiers de sortie individuels (`Output_Hyper_XX.tx` confirme que le jeton a été diffusé avec succès à l'ensemble des 8 processus. Bien que l'algorithme soit logarithmique, on observe une légère gitter (variation) dans les temps locaux due à l'ordonnancement du système d'exploitation :

**Observation :** Le temps maximum observé (Rang 5 : 0.000032 s) est celui utilisé pour calculer la performance globale du réseau. Cette mesure inclut non seulement le temps de calcul du XOR mais aussi la latence de communication inter-processus gérée par Open MPI.

**Conclusion technique :** L'accélération d'un facteur **2.3x** démontre que la topologie en hypercube minimise la latence de communication. Cette optimisation est indispensable pour les opérations collectives de synchronisation

| Rang MPI | Jeton reçu | Temps local (s) |
|---|---|---|
| Rang 0 | 1 | 0.000023 |
| Rang 1 | 1 | 0.000028 |
| Rang 2 | 1 | 0.000025 |
| Rang 3 | 1 | 0.000026 |
| Rang 4 | 1 | 0.000026 |
| Rang 5 | 1 | 0.000032 |
| Rang 6 | 1 | 0.000025 |
| Rang 7 | 1 | 0.000025 |

TABLE 4 – Données extraites des fichiers de sortie du simulateur d'hypercube.

dans les systèmes distribués.

# 4 Conclusion Générale

Ce TP a permis de quantifier l'impact de l'architecture matérielle sur l'efficacité logicielle à travers trois axes majeurs :

1. **Gestion de la mémoire (Hiérarchie de Cache) :** Le produit de matrices a démontré que la puissance de calcul brute est inutile sans une localité des données optimale. La permutation des boucles et le *blocking* ont permis de stabiliser les performances, mais c'est l'utilisation de **BLAS (dgemm)** qui a révélé le potentiel réel du i7-8750H. Avec un gain de **48x**, BLAS prouve que l'optimisation de bas niveau (vectorisation AVX2 et registres) est indispensable pour franchir le mur de la mémoire.
2. **Modèles de Parallélisme :**
   — **OpenMP** s'est avéré efficace pour réduire le temps de calcul sur un seul nœud, bien qu'il soit limité par la bande passante mémoire lors de l'augmentation du nombre de threads.
   — **MPI** a offert une alternative robuste pour la mémoire distribuée. Le calcul de $\pi$ a illustré un parallélisme parfait (*embarrassingly parallel*) où le passage à 6 processus offre une scalabilité quasi-linéaire sans conflit de cache.
3. **Topologies de Communication :** Les tests sur le jeton et l'hypercube ont souligné l'importance de l'organisation réseau. Le passage d'une topologie en anneau ($O(N)$) à un **hypercube** ($O(\log_2 N)$) a réduit le temps de diffusion de plus de **50%**, démontrant que pour les systèmes à grande échelle, le choix de la topologie est aussi critique que l'algorithme de calcul lui-même.

En conclusion, l'optimisation haute performance nécessite une approche hybride : une gestion rigoureuse de la mémoire locale, une exploitation des unités vectorielles et une stratégie de communication minimisant la latence réseau.

# Ressources et Reproductibilité

L'intégralité du code et du Makefile est accessible sur : `https://github.com/JairVasquezT/OS202-2026.git`

# Références


[1] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, et A. Goldberg. (2009). *Quincy : Fair Scheduling for Distributed Computing Clusters*. Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09). Association for Computing Machinery, Inc.

[2] Dongarra, J., et al. (2003). *The Sourcebook of Parallel Computing*.