

# Rapport - Laboratoire 06:

## Multiplication Matricielle

### Multithreadée

#### Table des matières

1. Introduction
2. Architecture de la solution
3. Implémentation détaillée
4. Tests et vérification
5. Analyse de performance

**Cours:** Programmation Concurrente (PCO)

**Semestre:** Automne 2025

**Date:** 18 Décembre 2025

**Auteur:** Jonatan Perret et Adrien Marcuard

## 1. Introduction

Ce laboratoire consiste à implémenter une multiplication matricielle optimisée pour architectures multi-coeurs, utilisant une décomposition par blocs et des mécanismes de synchronisation basés sur des moniteurs de Hoare.

### Objectifs

- Décomposer un problème mathématique pour le paralléliser

- Utiliser les moniteurs de Hoare pour la synchronisation
- Garantir la réentrance de la fonction `multiply()`
- Maximiser le parallélisme

## 2. Architecture de la solution

---

### ComputeParameters

Structure contenant les paramètres nécessaires pour calculer un bloc:

- Pointeurs vers les matrices A, B, et C
- Indices du bloc (blockI, blockJ)
- Nombre de blocs par ligne/colonne
- ID de computation (pour la réentrance)

### Buffer

Moniteur de Hoare gérant la communication entre threads:

- File de jobs à traiter
- Compteurs de jobs complétés par computation
- Conditions de synchronisation (jobAvailable, jobDone)
- Flag de terminaison

### ThreadedMatrixMultiplier

Classe principale gérant:

- Pool de threads workers
- Décomposition en blocs
- Coordination des calculs

## 3. Implémentation détaillée

---

### 3.1 Décomposition par blocs

Pour une matrice  $n \times n$  divisée en  $b \times b$  blocs:

$$C[i][j] = \sum_{k=0 \text{ to } b-1} A[i][k] \times B[k][j]$$

Chaque bloc  $C[i][j]$  est calculé indépendamment par un thread, ce qui garantit l'absence de conflits d'écriture.

#### Algorithme de calcul d'un bloc:

```

Pour chaque élément (i,j) dans le bloc C[blockI][blockJ]:
    sum = 0
    Pour chaque bloc K de 0 à nbBlocksPerRow-1:
        Pour chaque k dans le bloc K:
            sum += A[k][i] × B[j][k]
    C[j][i] = sum

```

## 3.2 Moniteur de Hoare - Buffer

### Méthodes principales:

#### **sendJob(params)**

```

monitorIn()
jobQueue.push(params)
signal(jobAvailable) // Réveille un worker en attente
monitorOut()

```

#### **getJob(parameters)**

```

monitorIn()
while (jobQueue.empty() && !isTerminating)
    wait(jobAvailable) // Attend un job
if (isTerminating && jobQueue.empty())
    return false // Terminaison propre
parameters = jobQueue.front()
jobQueue.pop()
monitorOut()
return true

```

#### **jobCompleted(computationId)**

```

monitorIn()
nbJobFinished++
jobsFinishedPerComputation[computationId]++
signal(jobDone) // Réveille le thread principal en attente
monitorOut()

```

## 3.3 Gestion de la réentrance

Pour supporter les appels concurrents à `multiply()`, chaque computation reçoit un ID unique:

```
int startNewComputation(int totalJobs) {
    monitorIn()
    int id = nextComputationId++
    jobsFinishedPerComputation[id] = 0
    totalJobsPerComputation[id] = totalJobs
    monitorOut()
    return id
}
```

Le thread appelant attend spécifiquement ses propres jobs:

```
void waitAllJobsDone(int computationId) {
    monitorIn()
    while (jobsFinishedPerComputation[computationId] < totalJobs)
        wait(jobDone)
    // Nettoyage
    jobsFinishedPerComputation.erase(computationId)
    monitorOut()
}
```

## 3.4 Terminaison propre

Le destructeur utilise un mécanisme en deux étapes:

### 1. Signaler la terminaison:

```
buffer->terminate() // Set isTerminating = true
```

### 2. Réveiller tous les threads:

```
for (int i = 0; i < 100; ++i) {
    monitorIn()
    signal(jobAvailable)
    monitorOut()
}
```

Cette approche compense l'absence de `broadcast` dans les moniteurs de Hoare, où `signal()` ne réveille qu'un seul thread.

### 3. Attendre la fin:

```
for (auto& thread : threads)
    thread->join()
```

## 4. Tests et vérification

### 4.1 Tests fournis

#### SingleThread

- 1 thread, matrice 500×500, 5×5 blocs
- Vérifie le bon fonctionnement de base
- Gain de performance: ~17%

#### Simple

- 4 threads, matrice 500×500, 5×5 blocs
- Vérifie le parallélisme de base
- Gain de performance: ~290-320%

#### Reentering

- 2 computations parallèles, 4 threads
- Vérifie la réentrance de multiply()
- Tests critiques pour la robustesse

### 4.2 Tests additionnels implémentés

Nous avons implémenté **10 tests supplémentaires** pour valider différents aspects de l'implémentation:

#### Test 1: ManyThreads

```
constexpr int MATRIXSIZE = 500;
constexpr int NBTHREADS = 16;           // Plus de threads que de blocs
constexpr int NBBLOCKSPERROW = 5;      // 25 blocs seulement
```

**Objectif:** Vérifier que l'algorithme gère correctement un surplus de threads. Certains threads resteront inactifs une fois tous les jobs distribués.

**Résultat attendu:** L'implémentation doit gérer gracieusement les threads en surplus sans deadlock.

## Test 2: ManyBlocks

```
constexpr int MATRIXSIZE = 500;
constexpr int NBTHREADS = 4;
constexpr int NBBLOCKSPERROW = 10;    // 100 blocs au total
```

**Objectif:** Tester avec une granularité plus fine (plus de petits jobs). Améliore la distribution de charge et maximise le parallélisme.

**Résultat attendu:** Gain de performance optimal grâce à la meilleure répartition du travail.

## Test 3: MultipleReentering

```
constexpr int MATRIXSIZE = 500;
constexpr int NBTHREADS = 4;
constexpr int NBBLOCKSPERROW = 5;
MultiplierThreadedTester<ThreadedMultiplierType> tester(4); // 4 computations
                                                               parallèles
```

**Objectif:** Pousser la réentrance avec 4 appels simultanés à `multiply()`. Chaque computation a ses propres 25 jobs, soit 100 jobs concurrents.

**Résultat attendu:** Tous les calculs se terminent correctement avec des résultats exacts. Prouve la robustesse du mécanisme de réentrance.

## Test 4: MinimalBlocks

```
constexpr int MATRIXSIZE = 500;
constexpr int NBTHREADS = 4;
constexpr int NBBLOCKSPERROW = 2;    // Seulement 4 blocs au total
```

**Objectif:** Tester avec un nombre minimal de blocs. Limite fortement le parallélisme possible (4 threads pour 4 blocs).

**Résultat attendu:** Gain de performance limité mais correct. Valide le fonctionnement avec peu de jobs.

## Test 5: LargeMatrix

```
constexpr int MATRIXSIZE = 800;
constexpr int NBTHREADS = 8;
constexpr int NBBLOCKSPERROW = 8; // 64 blocs
```

**Objectif:** Tester avec une matrice plus grande nécessitant plus de calculs. Timeout étendu à 60 secondes.

**Résultat attendu:** Gain de performance significatif démontrant la scalabilité avec des données volumineuses.

## Test 6: NoDecomposition

```
constexpr int MATRIXSIZE = 500;
constexpr int NBTHREADS = 4;
constexpr int NBBLOCKSPERROW = 1; // Un seul bloc = pas de parallélisme
```

**Objectif:** Cas limite où il n'y a qu'un seul bloc. Aucun parallélisme possible, tous les threads sauf un restent inactifs.

**Résultat attendu:** Performance similaire au mode single-thread. Valide le bon fonctionnement sans décomposition.

## Test 7: StressTest

```
constexpr int MATRIXSIZE = 500;
constexpr int NBTHREADS = 20; // Beaucoup de threads
constexpr int NBBLOCKSPERROW = 10; // 100 petits blocs
```

**Objectif:** Test de charge avec forte concurrence (20 threads, 100 jobs). Stress sur le moniteur et la file de jobs.

**Résultat attendu:** Système stable sans deadlock malgré la forte contention.

## Test 8: StressReentering

```
constexpr int MATRIXSIZE = 500;
constexpr int NBTHREADS = 8;
constexpr int NBBLOCKSPERROW = 5;
MultiplierThreadedTester<ThreadedMultiplierType> tester(8); // 8 computations
    parallèles
```

**Objectif:** Test de réentrance extrême avec 8 multiplications simultanées ( $8 \times 25 = 200$  jobs concurrents).

**Résultat attendu:** Tous les calculs corrects dans le temps imparti (45s). Valide la robustesse sous charge maximale.

## Test 9: PerfectSquare

```
constexpr int MATRIXSIZE = 400; // 400 = 20 × 20
constexpr int NBTHREADS = 4;
constexpr int NBBLOCKSPERROW = 20; // 400 blocs de 20x20 chacun
```

**Objectif:** Tester avec une décomposition très fine (400 petits blocs). Optimise la distribution mais augmente l'overhead de synchronisation.

**Résultat attendu:** Performance correcte malgré le grand nombre de jobs. Teste l'efficacité du buffer.

## Test 10: FewThreadsManyBlocks

```
constexpr int MATRIXSIZE = 500;
constexpr int NBTHREADS = 2; // Peu de threads
constexpr int NBBLOCKSPERROW = 10; // 100 blocs
```

**Objectif:** Situation inverse: peu de threads mais beaucoup de travail. Teste l'équilibrage de charge avec file d'attente importante.

**Résultat attendu:** Gain de performance proche de 100% (2 threads  $\approx 2\times$  plus rapide), démontrant l'équilibrage efficace.

## 4.3 Vérification anti-interblocage

La ligne `#define CHECK_DURATION` active des assertions de durée:

```
ASSERT_DURATION_LE(30, ({ ... })) // Maximum 30 secondes
```

Tous les tests passent sous cette limite, confirmant l'absence d'interblocage.

## 4.4 Méthodologie de vérification

1. **Exactitude:** Comparaison avec une implémentation séquentielle (`SimpleMatrixMultiplier`)
2. **Performance:** Mesure des gains de temps avec `std::chrono`
3. **Robustesse:** Tests de réentrance avec threads concurrents
4. **Absence de deadlock:** Timeout de 30 secondes via `CHECK_DURATION`

## 5. Analyse de performance

### 5.1 Résultats obtenus

Test	Threads	Blocs	Gain de temps	Temps (ms)
SingleThread	1	5×5	~17%	1211
Simple	4	5×5	~287%	797
Reentering (x2)	4	5×5	~284%	941
ManyThreads	16	5×5	~712%	715
ManyBlocks	4	10×10	~316%	782
MultipleReentering (x4)	4	5×5	~288%	1248

### 5.2 Observations

#### Scalabilité:

- Avec 4 threads: gain de ~3×
- Avec 16 threads: gain de ~7× (saturation des coeurs)

### **Granularité:**

- Blocs  $5 \times 5$  (25 jobs): Bon équilibre
- Blocs  $10 \times 10$  (100 jobs): Légèrement meilleur pour distribution de charge

### **Réentrance:**

- Performance diminue légèrement avec contention élevée
- Mais reste fonctionnelle et correcte

## **5.3 Facteurs limitants**

1. **Overhead de synchronisation:** Entrée/sortie du moniteur pour chaque job
2. **Contention mémoire:** Accès concurrent aux matrices A et B (lecture)
3. **Granularité des blocs:** Trop petits → overhead, trop grands → déséquilibre de charge

## **6. Améliorations possibles**

1. **Cache blocking:** Optimiser l'accès mémoire pour réduire les cache misses
2. **Granularité adaptative:** Ajuster automatiquement la taille des blocs

## **7. Utilisation de l'IA**

Nous avons utilisé de l'intelligence artificielle dans le cadre ce rapport pour la relecture et la correction grammaticale du texte, ainsi que pour la génération de certaines sections explicatives. Ils ont également aidé à structurer le document de manière claire et cohérente.