PCO: Rapport labo 6

Auteur: Anthony Pfister, Santiago Sugranes
Date: 19.12.2025
Cours: PCO2025 lab06, HEIG

Choix de conception

Architecture: Modèle de délégation

Les threads sont créés une seule fois dans le constructeur et restent actifs pendant toute la duré de vie de l'objet. Le thread principal délègue tout le travail aux threads workers via un buffer partagé.

Justification: évite le coût de création/destruction de threads à chaque appel à `multiply()`

Décomposition en blocs (i, j, k)

Le calcul suit la formule de la donnée du labo avec une décomposition en (i, j, k):

- chaque job calcule une contribution partielle `Aik * Bkj` pour un bloc k spécifique
- plusieurs jobs (différents k) contribuent au même élément `C[i][j]`
- nombre total de jobs : $\texttt{nbBlocksPerRow}^3$

Justification: respecte l'algorithme suggéré dans les instructions

Synchronisation: Moniteur de Hoare + Mutex

Buffer (moniteur de Hoare): gère la distribution des jobs et le suivi de l'avancement par `jobId` pour la réentrance.

Mutex pour les écritures: les écritures dans `C` sont protégés par `PcoMutex resultMutex` car plusieur threads écrivent dans le même `C[i][j]`. Les mises à jour sont groupées par bloc (tous les éléments d'un bloc dans une seule section critique).

Justification:

- le moniteur assure la synchronisation thread-safe de la file de jobs
- le mutex est nécessaire car plusieurs jobs contribuent au même élément de résultat
- le groupement par bloc réduit le nombre d'acquisitions du mutex

---

Gestion de la réentrance

Chaque appel à `multiply()` obtient un `jobId` unique via `registerComputation()`. Les threads signalent la fin de leurs jobs avec `notifyJobFinished(jobId)`, et le thread principal attend avec `waitForCompletion(jobId)` jusqu'à ce que tous les jobs de cette computation soient terminés.

Justification: permet à plusieurs appels à `multiply()` de s'executer concurremment sans problèmes

Terminaison des threads

Dans le destructeur:

1. signal de terminaison envoyée à tous les threads (`signalTermination()`)
2. attente de la fin de tous les threads (`join()`)
3. libération de la mémoire

Tests effectués

Test	Matrice	Threads	Blocs	Appels concurrents	Performance	Temps
SingleThread	500×500	1	5	1	-35.29%	2309 ms
Simple	500×500	4	5	1	+144.57%	1290 ms
Reentering	500×500	4	5	2	+141.96%, +25.84%	1703 ms
SmallMatrix	100×100	2	4	1	0%	23 ms
ManyBlocks	400×400	8	10	1	+316.22%	583 ms
HighReentrancy	300×300	4	6	4	+132.14%, +22.22%, -14.29%, -35.33%	585 ms
SingleBlock	200×200	4	1	1	-36.67%	152 ms
FewBlocksManyThreads	200×200	16	2	1	+235.29%	79 ms
ExtremeReentrancy	200×200	4	4	8	+81.82% à -55.21%	339 ms

Tous les tests passent sans erreur de calcul.

Analyse des performances

Pourquoi certains tests sont plus lents ?

SingleThread (-35%): overhead de synchronisation (moniteur, mutex) supérieur au bénéfice avec un seul thread.

SmallMatrix (0%): matrice trop petite pour amortir le coût de synchronisation.

HighReentrancy / ExtremeReentrancy (performances négatives): avec plusieurs calculs qui s'exécutent en même temps, tous les threads doivent attendre leur tour pour écrire dans la matrice résultat. Cette attente devient un goulot d'étranglement qui ralentit l'ensemble du traitement et annule les bénéfices du parallélisme.

Performances positives: les cas avec matrices moyennes/grandes, plusieurs threads et blocs (Simple, ManyBlocks, FewBlocksManyThreads) ont des gains significatifs car le parallélisme compense largement l'overhead de synchronisation.