AMAR LECHANI

Projet ExPResSO

January 6, 2026

1 Introduction

Dans le contexte des architectures modernes multi-cœurs, l'exploitation du parallélisme est indispensable pour la performance des applications. Cependant, la manipulation directe des fils d'exécution (via `pthreads`) est complexe et sujette aux erreurs.

L'objectif de ce projet est de concevoir et implémenter **ExPResSO**, une bibliothèque de haut niveau inspirée d'OpenMP. Elle permet au programmeur de soumettre des tâches sans gérer manuellement le cycle de vie des threads. Ce rapport détaille les choix d'architecture de la bibliothèque, les stratégies de parallélisation appliquées à plusieurs algorithmes classiques, et une analyse des performances obtenues.

2 Architecture et Implémentation de la Bibliothèque

L'implémentation de la bibliothèque repose sur le modèle du *Thread Pool* (équipe de travailleurs) et du paradigme *Fork-Join*.

2.1 Gestion de l'équipe et des tâches

La structure centrale du système (`expresso_system_t`) maintient l'état global.

- **Stockage des tâches :** Nous avons choisi d'utiliser une **liste chaînée simple** (`task_t *next`) pour la file d'attente. Ce choix se justifie par la nature dynamique de la soumission de tâches : le nombre de tâches n'est pas connu à l'avance, et une liste chaînée permet une allocation flexible sans redimensionnement coûteux de tableau.

- **Synchronisation :** L'accès à la file d'attente est protégé par un unique `pthread_mutex_t`. Deux variables de condition sont utilisées : `cond_work` pour réveiller les travailleurs lorsqu'une tâche est ajoutée, et `cond_done` pour réveiller le thread principal lorsque le travail est terminé.

2.2 La barrière de synchronisation (`expresso_wait`)

Un choix d'implémentation crucial a été fait au niveau de la fonction `expresso_wait`. Plutôt que de laisser le thread principal (celui qui a lancé le `main`) en attente passive (bloqué sur un `cond_wait`), nous l'avons intégré à l'équipe de calcul.

Justification : Tant qu'il reste des tâches dans la file, le thread principal en récupère une et l'exécute. Cela permet d'utiliser 100% des ressources processeur disponibles, sans gaspiller le cœur du thread principal.

2.3 Politiques d'Ordonnancement (Version Avancée)

Pour la version avancée, nous avons implémenté trois politiques distinctes, gérées via des files d'attente spécifiques (une file commune pour le dynamique, des files privées pour le statique/équilibré) :

1. **SCHEDULE_DYNAMIC :** Politique par défaut. Toutes les tâches vont dans une file commune (FIFO). Le principe est "premier arrivé, premier servi". C'est la politique la plus robuste pour des tâches de durées inégales.

2. **SCHEDULE_STATIC :** Distribution *Round-Robin* (tourniquet) au moment de la soumission. La tâche i est assignée au travailleur $i \pmod N$. Cette politique minimise la contention sur le mutex (chaque worker a sa file), mais risque de déséquilibrer la charge si les tâches n'ont pas la même durée.

3. **SCHEDULE_BALANCED :** Utilise la fonction `expresso_weighted_task`. Nous maintenons un tableau de "charge cumulée" (`worker_loads`). Chaque nouvelle tâche est assignée au travailleur ayant la charge la plus faible.

3 Parallélisation des Applications

L'utilisation de la bibliothèque a été validée sur quatre applications, en adoptant une stratégie de décomposition de domaine (Data Parallelism).

3.1 Produit Scalaire (dot) et Multiplication Matrice-Vecteur (vecmul)

Ces deux algorithmes parcourent des tableaux linéaires.

- **Stratégie :** Découpage des vecteurs en blocs contigus de taille L (paramètre `block`).
- **Justification :** Le traitement par blocs contigus favorise la localité spatiale des données et améliore l'efficacité du cache processeur.
- **Réduction :** Pour `dot`, chaque tâche calcule une somme partielle locale. La somme finale est réalisée séquentiellement par le thread principal après le `expresso_wait`. Ce surcoût séquentiel est négligeable par rapport au calcul parallèle.

3.2 Multiplication de Matrices (matmul)

- **Stratégie :** Découpage "gros grains" par blocs de lignes de la matrice résultat C.
- **Justification :** Chaque tâche est responsable de calculer N lignes complètes. Cela crée des tâches suffisamment lourdes pour amortir le coût de gestion des threads, ce qui est essentiel pour la performance.

3.3 Liste Chaînée (list)

- **Stratégie :** Une tâche par nœud de la liste.
- **Problème :** Chaque nœud possède une valeur simulant un temps de calcul hétérogène (`sleep(node->value)`).
- **Solution :** Utilisation de `expresso_weighted_task` avec le poids égal à la valeur du nœud. Cela permet à la politique `BALANCED` de répartir équitablement les "longues" et les "courtes" tâches, là où une approche statique aurait échoué.

4 Analyse des Performances

Les tests ont été réalisés sur une machine virtuelle Ubuntu multi-cœurs. Nous avons comparé les temps d'exécution séquentiels (référence) et parallèles (ExPResSO) sur deux scénarios : faible charge et forte charge.

4.1 Analyse du surcoût (Overhead)

Sur des données de petite taille (paramètres par défaut : matrices 32x32, vecteurs 1024), nous observons systématiquement que la version ExPResSO est plus lente que la version séquentielle.

- *Exemple (dot) :* 10 μs (Seq) vs 450 μs (Parallèle).

- **Interprétation :** Ce phénomène est normal. Le temps de gestion de la bibliothèque (allocation mémoire des tâches, verrouillage des mutex, réveil des threads via signaux système) est supérieur au temps de calcul lui-même, qui est quasi-instantané. Le parallélisme ne devient bénéfique que lorsque le temps de calcul domine le temps de gestion.

4.2 Analyse du gain (Speedup) en forte charge

Lorsque la taille du problème augmente, l'efficacité de la bibliothèque devient évidente. Voici les résultats obtenus sur de gros volumes de données :

Application	Taille du problème	Temps Séquentiel	Temps ExPResSO	Accélération
DOT	100 Millions	0.306 s	0.301 s	$\sim \times 1.01$
MATMUL	1000 x 1000	4.024 s	1.556 s	$\sim \times 2.58$
VECMUL	50 Millions	0.240 s	0.051 s	$\sim \times 4.70$
LIST	5 Nœuds (1s à 5s)	15.001 s	5.000 s	$\sim \times 3.00$

- **Matmul et Vecmul :** On observe des accélérations significatives (x2.5 à x4.7). Le calcul parallèle exploite efficacement les cœurs disponibles. Pour `vecmul`, le gain est particulièrement impressionnant car les tâches sont indépendantes et sans partage de données critique.
- **Dot :** Le gain est minime. Cela s'explique par le fait que l'opération est bornée par la bande passante mémoire (*Memory Bound*) et non par la puissance de calcul CPU. Ajouter des threads ne permet pas de lire la mémoire plus vite.
- **List :** Le résultat est théoriquement parfait. En séquentiel, le temps est la somme des attentes ($1+2+3+4+5 = 15s$). En parallèle avec suffisamment de travailleurs, le temps est déterminé par la tâche la plus longue ($5s$). Le gain observé confirme le bon fonctionnement de l'ordonnancement.

5 Conclusion

Le projet ExPResSO a permis de mettre en œuvre les mécanismes fondamentaux d'un moteur d'exécution parallèle : gestion du cycle de vie des threads, synchronisation par exclusion mutuelle et variables de condition.

L'implémentation répond aux exigences fonctionnelles : la version de base et la version avancée (ordonnancement statique et équilibré) sont opérationnelles. Les tests sur les benchmarks montrent que la bibliothèque est robuste (fonctionne même sans tâche) et performante, offrant des gains de temps substantiels sur les calculs lourds (`matmul`, `vecmul`) et les tâches hétérogènes (`list`).