**Semestre 2**

| |
|---|
| **Matière :** Algorithmique et architectures parallèles …….. |
| **Enseignants : Sami Achour / Tahar Alimi** |
| **Classe :** ING-A2-GL |

# Calcul parallèle de la somme d'un vecteur en OpenMP : Méthodes de synchronisation

## 1) Introduction : Le problème de la somme en parallèle

- Présentation du problème : **Calculer la somme des éléments d'un grand vecteur**.
- Pourquoi paralléliser ce calcul ?
- Définition de la **course critique** et des **problèmes d'accès concurrent**.

## 2) Première implémentation : Une solution erronée

On commence avec une **implémentation naïve**, qui n'utilise que #pragma omp parallel et #pragma omp for.

***Version naïve (FAUX RÉSULTAT):***

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 1000000
#define THREADS 4

int main() {
    int *vecteur = (int*)malloc(N * sizeof(int));
    long long somme = 0;  // Variable partagée

    // Initialisation
    for (int i = 0; i < N; i++) {
        vecteur[i] = 1;
    }

    // Version incorrecte
    #pragma omp parallel num_threads(THREADS)
    {
        #pragma omp for
        for (int i = 0; i < N; i++) {
            somme += vecteur[i];  // Problème ici !
        }
    }

    printf("Somme erronée: %lld\n", somme);

    free(vecteur);
    return 0;
}
```

**Explication du problème :**

- somme est une **variable partagée**.
- Plusieurs threads **lisent et modifient simultanément** cette variable, ce qui provoque des **écritures perdues**.
- Résultat : la somme **ne sera pas correcte** et variera d'une exécution à l'autre.

## 3) Solutions correctes et explication de chaque méthode

### a) Utilisation de #pragma omp critical

- On protège l'accès à somme avec une **section critique**.
- Cette directive garantit que **seul un thread à la fois** exécute la section critique.
- **Tous les autres threads attendent** leur tour.
- Lorsqu'on doit **protéger plusieurs opérations** sur une variable partagée.
- Lorsqu'on modifie **plusieurs variables partagées en même temps**.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 1000000
#define THREADS 4

int main() {
   int *vecteur = (int*)malloc(N * sizeof(int));
   long long somme = 0;  // Variable partagée
   // Initialisation
   for (int i = 0; i < N; i++) {
      vecteur[i] = 1;    }
   #pragma omp parallel num_threads(THREADS)
   {  #pragma omp for
      for (int i = 0; i < N; i++) {
         #pragma omp critical
         somme += vecteur[i];  // Problème Résolu ! -> Section Critique
      }   }
   printf("Somme: %lld\n", somme);
   free(vecteur);
   return 0;}
```

***Avantages et inconvénients***

✓ Corrige le problème d'accès concurrent.

✗ **Lente** : Ralentit le programme à cause du **verrou global** qui force les threads à s'exécuter un par un. → **Forte contention**

### b) Utilisation de #pragma omp atomic

- On utilise #pragma omp atomic pour une **mise à jour atomique** plus efficace.
- Applique **une protection uniquement sur une opération unique** (+=, -=, etc.).
- Plus rapide que critical, mais limité aux **opérations simples**.
- Pour une **modification rapide** d'une seule variable partagée.
- Idéal pour **des incrémentations ou des additions** concurrentes.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 1000000
#define THREADS 4

int main() {
   int *vecteur = (int*)malloc(N * sizeof(int));
   long long somme = 0;  // Variable partagée
   // Initialisation
   for (int i = 0; i < N; i++) {
      vecteur[i] = 1;    }
   #pragma omp parallel num_threads(THREADS)
   {   #pragma omp for
      for (int i = 0; i < N; i++) {
         #pragma omp atomic
         somme += vecteur[i];  // Problème Résolu ! -> Mise-à-jour atomique
      }   }
   printf("Somme: %lld\n", somme);
   free(vecteur);
   return 0;}
```

***Avantages et inconvénients***

✓ Plus rapide que critical (optimisé par le processeur).

✗ Moins efficace que d'autres méthodes pour un grand nombre d'opérations.

**c) Somme locale par thread**

Chaque thread accumule sa somme localement avant de la fusionner.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 1000000
#define THREADS 4

int main() {
   int *vecteur = (int*)malloc(N * sizeof(int));
   long long somme = 0;  // Variable partagée
   // Initialisation
   for (int i = 0; i < N; i++) {
      vecteur[i] = 1;    }
   #pragma omp parallel num_threads(THREADS)
{  long long somme_locale = 0;
    #pragma omp for
   for (int i = 0; i < N; i++) {
      somme_locale += vecteur[i];    }
   #pragma omp atomic
   somme += somme_locale;}
   printf("Somme : %lld\n", somme);
   free(vecteur);
   return 0;}
```

***Avantages et inconvénients***

✓ Diminue le nombre d'opérations atomiques.

✗ Plus rapide, mais encore limité par l'opération atomic.

**d) Utilisation de reduction(+:somme)**

- OpenMP fournit un mécanisme optimisé pour l'addition.
- Chaque thread calcule une **somme locale**, puis OpenMP fusionne **automatiquement** les résultats.
- **La solution la plus rapide** pour des opérations comme somme, produit, max, min, etc.
- **Idéal pour les sommes, produits, min, max en parallèle**.
- Lorsque chaque thread peut accumuler ses valeurs **sans conflit**.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 1000000
#define THREADS 4

int main() {
   int *vecteur = (int*)malloc(N * sizeof(int));
   long long somme = 0;  // Variable partagée
   // Initialisation
   for (int i = 0; i < N; i++) {
      vecteur[i] = 1;    }
  #pragma omp parallel for reduction(+:somme)
      for (int i = 0; i < N; i++) {
               somme += vecteur[i];    }
   printf("Somme : %lld\n", somme);
   free(vecteur);
   return 0;}
```

***Avantages et inconvénients***

✓ **Meilleure solution** en général.

✓ OpenMP effectue la somme efficacement en utilisant une stratégie optimisée.

✗ Peut être légèrement plus lent que local_sum + atomic sur certaines architectures.

✗ Moins flexible que critical (ne protège **que certaines opérations**).

✗ Ne fonctionne pas si la mise à jour de la variable est **complexe**.

**e) Utilisation de simd**

L'utilisation de **SIMD (Single Instruction, Multiple Data)** permet d'exploiter la **vectorisation matérielle** du processeur pour exécuter plusieurs opérations en parallèle sur un même ensemble de données. Cela optimise l'utilisation des **unités vectorielles** du CPU.

- **SIMD divise les données en blocs** et effectue plusieurs opérations en **une seule instruction**.
- **Le compilateur optimise les accès mémoire** pour utiliser les **registres vectoriels**.
- **Idéal pour les boucles sur de grands tableaux**, où chaque itération peut être effectuée indépendamment.
- **Évite les conflits d'accès mémoire** en utilisant des instructions spécialisées.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 1000000
#define THREADS 4

int main() {
   int *vecteur = (int*)malloc(N * sizeof(int));
   long long somme = 0;  // Variable partagée
      double start_time, end_time;
   // Initialisation avec OpenMP + SIMD
   #pragma omp parallel for simd
   for (int i = 0; i < N; i++) {
      vecteur[i] = 2;
   }
      start_time = omp_get_wtime();
      // Parallélisation avec OpenMP et SIMD
   #pragma omp parallel num_threads(THREADS) reduction(+:somme)
   {
      #pragma omp for simd
      for (int i = 0; i < N; i++) {
         somme += vecteur[i];  // Accumulation avec SIMD
      }
   }
      end_time = omp_get_wtime();
      printf("Somme : %lld\n", somme);
      printf("Execution time = %f", end_time - start_time);
      free(vecteur);
   return 0;}
/*
1- Ajout de #pragma omp for simd : Active SIMD sur la boucle pour un traitement vectorisé.
2- Utilisation de reduction(+:somme_locale) :
** Évite les accès atomiques coûteux à chaque itération.
** Chaque thread accumule localement, puis atomic met à jour somme.
3- Initialisation optimisée avec #pragma omp parallel for simd :
** Utilisation de SIMD pour remplir rapidement vecteur
--------------------
Pourquoi cette version est plus rapide ?
✅ Réduction des accès concurrents : Somme locale par thread avant mise à jour atomique.
✅ Optimisation mémoire : Chargement SIMD réduit le nombre d'accès.
✅ Parallélisme efficace : Threads OpenMP + SIMD vectorisation.
```

***Avantages et inconvénients***

- ✅ **Exploitation maximale des registres vectoriels** du CPU.
- ✅ **Réduction du nombre d'instructions** exécutées.
- ✅ **Très efficace pour les opérations arithmétiques simples** comme l'addition, la multiplication, le min et le max.
- ✅ **Compatible avec OpenMP reduction**, permettant une parallélisation **multi-core + vectorielle**.
- ❌ **Nécessite que les données soient bien alignées en mémoire** pour être efficaces.
- ❌ **Moins efficace si les données ont des dépendances** (par ex., si chaque itération d'une boucle dépend des précédentes).
- ❌ **Le compilateur peut ignorer la vectorisation** si les accès mémoire ne sont pas optimaux.

## 4) Étude comparative des performances

| Méthode | Correcte ? | Temps d'exécution | Commentaire |
|---|---|---|---|
| Naïve (sans protection) | ✗ NON | Très rapide | Faux résultat |
| critical | ✓ OUI | Très lent | Bloque tous les threads |
| atomic | ✓ OUI | Moyen | Optimisé, mais pas idéal |
| somme locale + atomic | ✓ OUI | Rapide | Meilleure que atomic seul |
| reduction | ✓ OUI | Très rapide | Bien optimisée |
| simd | ✓ OUI | Ultra-rapide | Meilleur solution |

## 5) Tableau Comparatif des Directives OpenMP

| Critère | #pragma omp critical | #pragma omp atomic | #pragma omp reduction | #pragma omp simd |
|---|---|---|---|---|
| **Flexibilité** | Très flexible (peut protéger plusieurs opérations et variables) | Moins flexible (uniquement pour des opérations simples : +=, -=, etc.) | Très limité (uniquement pour certaines opérations : somme, min, max, etc.) | Très limité (uniquement pour des opérations indépendantes sur des tableaux) |
| **Performance** | **Lent** (bloque toute la section de code, attente entre threads) ⇨ (attente entre threads) | **Plus rapide** que critical (protège uniquement l'opération mémoire) ⇨ (accès mémoire fréquent) | **Le plus rapide** (chaque thread accumule localement avant la fusion finale) ⇨ (accumulation locale + fusion finale) | Extrêmement rapide (vectorisation, exécute plusieurs opérations en parallèle) ⇨ (vectorisation + multi-threading) |
| **Parallélisme** | **Faible** (goulot d'étranglement si plusieurs threads attendent leur tour) | **Meilleur** (évite les attentes longues) | **Excellente scalabilité** (chaque thread travaille indépendamment) | Très efficace (utilise les unités SIMD du processeur) |
| **Complexité d'utilisation** | Facile (bloque un bloc de code complet) | Facile (protège une seule opération) | Peut être contraignant (ne fonctionne qu'avec certaines opérations) | Contraignant (nécessite des données bien alignées et indépendantes) |
| **Cas d'utilisation** | - Protection de **plusieurs opérations** dans un bloc de code.<br>- Modification de **plusieurs variables partagées**. | - Addition ou incrémentation d'une seule variable partagée.<br>- Simple mise à jour (+=, -=, etc.). | - Somme, produit, min, max en parallèle.<br>- Idéal pour **accumulations massives**. | - Opérations vectorisables sur de grands tableaux (somme, produit, etc.).<br>- Calculs indépendants sur des boucles. |
| **Exemple d'utilisation** | c#pragma omp critical { somme += vecteur[i]; } | #pragma omp atomic somme += vecteur[i]; | #pragma omp parallel for reduction(+:somme) somme += vecteur[i]; | #pragma omp for simd reduction (+:somme) somme+=vecteur[i]; |

## 6) Conclusion et recommandations

- **Il ne faut jamais utiliser une variable partagée sans protection** pour éviter les **conditions de concurrence**.
- critical est **correct** mais **très lent** en raison des attentes entre threads..
- atomic est **plus rapide** mais reste **limité** lorsque l'on traite un grand nombre d'itérations.
- L'optimisation recommandée dépend du contexte :
  - **Somme locale + atomic** : Offre un **contrôle fin** et réduit la contention mémoire.
  - **reduction(+:somme)** : Solution **simple et efficace**, offrant **une excellente scalabilité**.
  - **simd + reduction(+:somme)** : **Meilleure performance** grâce à la **vectorisation SIMD**, idéale pour **grands tableaux** et **accès mémoire optimisé**.