UNIVERSITÉ IBN TOFAIL

FACULTÉ DES SCIENCES - KENITRA

DÉPARTEMENT INFORMATIQUE

MASTER INFORMATIQUE ET INTELLIGENCE ARTIFICIELLE

RAPPORT

Optimisation des Boucles avec OpenMP

Réalisé par :

Anas BOUKHLIJA

Mohamed Yassine FAIDA

Chaimae RETBI

Omayma ELHARTI

Encadré par :

Nada FAQIR

ANNÉE UNIVERSITAIRE 2024-2025

Table des matières

1 Introduction

La performance des programmes parallèles peut souvent être significativement améliorée par l'optimisation des boucles, un aspect crucial dans le développement d'applications à haute performance. Dans ce contexte, OpenMP (Open Multi-Processing) se présente comme une solution efficace pour la parallélisation des boucles, offrant une interface simple pour exploiter les architectures multi-cœurs modernes.

Ce projet se concentre sur l'étude et la mise en œuvre des techniques d'optimisation des boucles en utilisant OpenMP. L'objectif principal est de comparer les performances des programmes avant et après l'application de diverses techniques d'optimisation telles que **le déroulage de boucle (loop unrolling)** et **la fusion de boucles (loop fusion)**.

Objectif de ce projet se concentre sur l'étude et l'implémentation des techniques d'optimisation des boucles en utilisant OpenMP. L'objectif principal est de :

- **Étudier les Techniques d'Optimisation :** Examiner et mettre en œuvre diverses techniques d'optimisation des boucles telles que le déroulage de boucle (loop unrolling) et la fusion de boucles (loop fusion).
- **Comparer les Performances :** Évaluer l'impact de ces techniques sur les performances des programmes parallèles en mesurant et en comparant les temps d'exécution avant et après optimisation.
- **Fournir des Recommandations :** Identifier les meilleures pratiques basées sur les résultats expérimentaux pour guider l'optimisation des boucles dans les applications parallèles.

2 Optimisation des Boucles dans les Programmes Parallèles avec OpenMP

OpenMP (Open Multi-Processing) est une API de programmation parallèle pour les architectures à mémoire partagée, qui permet de paralléliser les boucles et autres sections de code de manière simple et efficace. Voici une vue d'ensemble des principales directives et fonctions d'OpenMP utilisées pour l'optimisation des boucles.

2.1 Directives pour la Parallélisation des Boucles

2.1.1 Directive #pragma omp parallel for

Fonctionnement : Cette directive est utilisée pour diviser les itérations d'une boucle for entre plusieurs threads. OpenMP crée un groupe de threads (une "team") qui exécute le travail en parallèle. Chaque thread se voit attribuer une portion des itérations de la boucle à exécuter. Le programmeur peut contrôler la répartition du travail à l'aide de clauses telles que schedule.

Avantages :

- **Simple à utiliser :** Il suffit d'ajouter la directive pour paralléliser la boucle.
- **Optimisation automatique :** OpenMP distribue les itérations de manière efficace entre les threads.

Scénarios d'utilisation : Idéal pour les boucles indépendantes où chaque itération peut être exécutée de manière autonome.

Exemple :

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

int main() {
    int N = 1000;
    int array[N];

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        array[i] = array[i] * 2;
    }
    return 0;
}
```

2.1.2 Directive #pragma omp for

Fonctionnement : Utilisée à l'intérieur d'une région parallèle (générée avec #pragma omp parallel), cette directive permet de distribuer les itérations d'une boucle entre les threads créés par la région parallèle. Contrairement à #pragma omp parallel for, elle n'initialise pas la création de threads mais se contente de distribuer les itérations dans les threads déjà en cours d'exécution.

Avantages :

- **Flexibilité :** Permet de combiner des instructions parallèles personnalisées avant ou après la boucle.

Scénarios d'utilisation : Utile lorsque vous avez besoin de gérer manuellement la création de threads et de spécifier exactement quelles portions de code seront parallélisées.

Exemple :

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

int main() {
    int N = 1000;
    int array[N];

    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < N; i++) {
            array[i] = array[i] * 2;
        }
    }
    return 0;
}
```

2.1.3 Directive #pragma omp parallel for collapse(n)

Fonctionnement : La directive collapse permet de paralléliser des boucles imbriquées en les fusionnant en une seule boucle pour la parallélisation. En spécifiant le nombre de boucles imbriquées à "fusionner" avec collapse(n), où n représente le nombre de niveaux, OpenMP les traite comme une seule grande boucle.

Avantages :

- **Meilleure parallélisation :** Augmente le nombre d'itérations distribuées entre les threads, ce qui peut améliorer l'efficacité de la parallélisation pour les boucles imbriquées.
- **Simplicité :** Facilite l'écriture de boucles parallélisées imbriquées sans avoir à gérer manuellement la distribution des itérations.

Scénarios d'utilisation : Utile dans les situations où plusieurs niveaux de boucles imbriquées doivent être parallélisés efficacement. Particulièrement efficace lorsque les boucles extérieures ont un faible nombre d'itérations, mais que les boucles intérieures ont un nombre important d'itérations.

Exemple :

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

int main() {
    int N = 1000;
    int M = 500;
    int matrix[N][M];

    #pragma omp parallel for collapse(2)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            matrix[i][j] = i + j;
        }
    }
    return 0;
}
```

2.2 Directives pour la Gestion des Exécutions

2.2.1 Directive #pragma omp master

Fonctionnement : La directive master spécifie que le bloc de code suivant ne sera exécuté que par le thread maître (le thread principal). Contrairement à single, aucun point de synchronisation implicite n'est créé à la fin de master, ce qui permet aux autres threads de continuer leur travail sans attendre.

Avantages :

- **Exécution spécifique :** Utilisé lorsque certaines parties du code doivent être exécutées uniquement par le thread maître.
- **Pas de synchronisation :** Permet aux autres threads de poursuivre leur exécution sans être bloqués.

Scénarios d'utilisation : Utilisé pour les sections de code qui ne nécessitent qu'une seule exécution, comme l'initialisation de variables globales ou l'affichage de résultats.

Exemple :

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

int main() {
    #pragma omp parallel
    {
        #pragma omp master
        {
            printf("This is executed by the master thread.\n");
        }
    }
    return 0;
}
```

2.2.2 Directive #pragma omp ordered

Fonctionnement : La directive ordered est utilisée pour assurer qu'une section spécifique du code dans une boucle parallèle soit exécutée dans l'ordre des itérations. Elle est généralement utilisée avec des boucles parallélisées via #pragma omp for et la clause ordered.

Avantages :

- **Sécurité de l'ordre :** Permet de paralléliser tout en garantissant que certaines opérations soient effectuées dans l'ordre strict des itérations.

Scénarios d'utilisation : Utile lorsque certaines parties d'une boucle doivent conserver un ordre spécifique, même dans un environnement parallèle.

Exemple :

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

int main() {
```

```
    int N = 1000;

    #pragma omp parallel for ordered
    for (int i = 0; i < N; i++) {
        #pragma omp ordered
        {
            printf("Thread %d, iteration %d\n", omp_get_thread_num(), i
   );
        }
    }
    return 0;
}
```

2.2.3 Directive #pragma omp nowait

Fonctionnement : Cette directive empêche les threads d'attendre à la fin d'une boucle avant de passer à l'instruction suivante dans le code. Normalement, OpenMP impose une synchronisation implicite à la fin de chaque boucle parallèle, mais nowait peut être utilisé pour contourner cela.

Avantages :

- **Réduction des attentes inutiles :** Améliore les performances lorsque l'attente des threads à la fin de la boucle n'est pas nécessaire.
- **Fluidité d'exécution :** Permet aux threads de poursuivre leur exécution immédiatement après avoir terminé leurs itérations.

Scénarios d'utilisation : Utile lorsque les itérations suivantes dans une boucle ne dépendent pas des autres itérations ou des résultats des autres threads.

Exemple :

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

int main() {
    int N = 1000;
    int array[N];

    #pragma omp parallel for nowait
    for (int i = 0; i < N; i++) {
        array[i] = i * 2;
    }
```

```
12      // Pas de synchronisation implicite ici
13
14      return 0;
15  }
```

2.3 Directives pour la Répartition des Boucles

2.3.1 Directive #pragma omp parallel for schedule

Fonctionnement : Cette directive permet de contrôler précisément comment les itérations d'une boucle sont réparties entre les threads à l'aide de la clause schedule. Les options les plus courantes sont :

- **Répartition Statique (schedule(static)) :** Avec schedule(static), OpenMP divise de manière fixe et égale les itérations de la boucle entre les threads dès le début de l'exécution. Chaque thread reçoit un nombre prédéfini d'itérations et les exécute sans attendre.
  Quand utiliser static ?
- Lorsque chaque itération a un temps d'exécution similaire.
- Pour minimiser la surcharge de gestion des threads, car les itérations sont attribuées une seule fois, sans redistribution dynamique.
  Avantages :
- Faible surcharge de gestion.
- Bonne pour des itérations homogènes en termes de coût.
  Inconvénients :
- Déséquilibre potentiel si certaines itérations prennent plus de temps à exécuter que d'autres, car la répartition ne s'ajuste pas dynamiquement.
- **Répartition Dynamique (schedule(dynamic)) :** Avec schedule(dynamic), les threads ne reçoivent pas toutes leurs itérations au début. Au lieu de cela, un thread prend un bloc d'itérations, les exécute, puis prend un autre bloc une fois qu'il a terminé. Cela permet de mieux équilibrer la charge lorsque certaines itérations sont plus coûteuses que d'autres.
  Quand utiliser dynamic ?
- Lorsque les itérations ne prennent pas toutes le même temps à exécuter.
- Pour mieux équilibrer la charge lorsque la complexité ou la durée des itérations varie.
  Avantages :

- Meilleur équilibrage de la charge lorsque les itérations ont des durées d'exécution variables.
- Réduit le risque que certains threads restent inactifs en attendant que d'autres finissent.

Inconvénients :

- Une surcharge supplémentaire due à la gestion dynamique des itérations (chaque thread doit demander de nouvelles itérations une fois son bloc terminé).

• **Répartition Guidée (schedule(guided)) :** Avec schedule(guided), les premiers blocs d'itérations assignés aux threads sont grands, puis leur taille diminue progressivement à mesure que les threads avancent. Cela permet de commencer avec une charge importante pour chaque thread, puis d'affiner l'équilibrage lorsque le programme approche de la fin de la boucle.

Quand utiliser guided ?

- Lorsque tu veux un compromis entre une faible surcharge et un bon équilibrage de la charge.
- Pour les boucles avec un travail initial important, suivi d'un travail moins intense vers la fin.

Avantages :

- Combinaison d'une répartition initiale rapide et d'un bon équilibrage de charge sur la fin.
- Peut minimiser les déséquilibres sans imposer trop de surcharge.

Inconvénients :

- Comme dynamic, il y a une certaine surcharge de gestion dynamique des itérations.
- Moins prévisible que static en termes de répartition de charge.

• **Répartition Auto (schedule(auto)) :** Avec schedule(auto), la répartition des itérations est laissée à la discrétion du compilateur ou du runtime d'OpenMP. Cela permet au compilateur de décider de la meilleure stratégie en fonction des caractéristiques du programme et de la machine cible.

Quand utiliser auto ?

- Lorsque tu fais confiance au compilateur pour prendre des décisions intelligentes concernant la répartition des boucles.
- Lorsque tu veux déléguer les détails de l'optimisation.

Avantages :

- Simplicité : tu laisses le compilateur ou le runtime gérer la répartition.

- Possibilité d'adaptation aux architectures matérielles spécifiques.

Inconvénients :

- Moins de contrôle sur la façon dont les itérations sont réparties.
- Les décisions du compilateur ne sont pas toujours optimales pour tous les cas.

2.4 Directives pour la Réduction

2.4.1 Directive #pragma omp parallel for reduction(operator : variable)

Fonctionnement : Cette directive permet d'effectuer des opérations de réduction (comme des sommes, des produits, des maximums) sur des variables partagées en toute sécurité dans un contexte parallèle. Chaque thread maintient une copie privée de la variable, puis les résultats sont combinés à la fin de la boucle.

Avantages :

- **Sécurité des données :** Évite les conflits d'accès aux données partagées en combinant les résultats de manière ordonnée.
- **Simplicité :** Réduction automatique des variables sans avoir à écrire du code supplémentaire pour gérer manuellement les résultats des threads.

Scénarios d'utilisation : Utile pour les boucles où une variable globale doit être mise à jour par tous les threads, comme lors du calcul d'une somme totale.

Exemple :

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

int main() {
    int N = 1000;
    int sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < N; i++) {
        sum += i;
    }
    return 0;
}
```

2.5 Directives pour la Gestion de la Concurrence

2.5.1 Directive #pragma omp atomic

Fonctionnement : La directive atomic assure que les opérations sur des variables partagées se déroulent de manière atomique, c'est-à-dire sans interruption par d'autres threads. Cela empêche les conditions de concurrence pour ces opérations simples.

Avantages :

- **Performance :** atomic est plus performant que critical car il ne protège qu'une seule opération sur une variable partagée, limitant la surcharge.
- **Simplicité :** Idéal pour les opérations arithmétiques sur une seule variable partagée.

Scénarios d'utilisation : atomic est utilisé pour des opérations simples comme l'incrémentation, l'accumulation ou la mise à jour d'une variable partagée dans une boucle parallélisée.

Exemple :

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

int main() {
    int N = 1000;
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        #pragma omp atomic
        sum += i;
    }

    printf("Sum = %d\n", sum);

    return 0;
}
```

2.5.2 Directive #pragma omp critical

Fonctionnement : La directive critical garantit que seulement un thread à la fois peut exécuter une section spécifique du code. Elle est utile dans des cas où plusieurs threads doivent accéder à des ressources partagées, mais de manière exclusive pour éviter les conditions de concurrence.

Avantages :

- **Sécurité des données :** Protège les sections critiques qui doivent être exécutées par un seul thread à la fois.
- **Facilité d'utilisation :** Permet d'ajouter une gestion des sections critiques sans avoir à gérer explicitement des verrous.

Scénarios d'utilisation : Utile pour les boucles parallélisées où certaines sections de code doivent accéder à des données partagées de manière sécurisée.

Exemple :

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

int main() {
    int N = 1000;
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        int temp = i; // Calcul intermediaire
        #pragma omp critical
        {
            sum += temp;  // Section critique protegee
        }
    }
    return 0;
}
```

2.6 Directives pour les Variables Privées

2.6.1 Directive #pragma omp parallel for private

Fonctionnement : Cette directive garantit que certaines variables sont privées à chaque thread, ce qui signifie que chaque thread dispose de sa propre copie de la variable. Cela permet d'éviter les problèmes liés à l'accès concurrent aux mêmes variables.

Avantages :

- **Protection des données :** Évite les conflits liés à l'accès simultané aux variables partagées.

Scénarios d'utilisation : Utile lorsque des variables temporaires sont utilisées dans une boucle, mais que chaque thread doit avoir sa propre copie pour éviter des comportements imprévisibles.

Exemple :

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

int main() {
    int N = 1000;

    #pragma omp parallel for private(temp)
    for (int i = 0; i < N; i++) {
        int temp = i * 2;
        // Traitement utilisant temp
    }

    return 0;
}
```

2.7 Directives de Synchronisation

2.7.1 Directive #pragma omp barrier

Fonctionnement : La directive barrier impose une synchronisation entre tous les threads. Lorsque la directive est rencontrée, chaque thread doit attendre que tous les autres atteignent ce point avant de continuer. Cela garantit que certaines sections de code ne commencent à s'exécuter qu'une fois que toutes les itérations précédentes ont été terminées.

Avantages :

- **Coordination stricte :** Assure que tous les threads progressent ensemble à des points critiques dans l'exécution du programme.

Scénarios d'utilisation : Utilisé pour synchroniser les threads dans une boucle ou entre différentes sections parallélisées d'un programme.

Exemple :

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

int main() {
```

```
    int N = 1000;
    int array[N];

    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < N; i++) {
            array[i] = i;
        }

        #pragma omp barrier  // Synchronisation des threads ici

        #pragma omp for
        for (int i = 0; i < N; i++) {
            array[i] *= 2;
        }
    }
    return 0;
}
```

2.8 Directives pour les Tâches et les Sections Critiques

2.8.1 Directive #pragma omp task

Fonctionnement : La directive task permet de spécifier une tâche à exécuter de manière asynchrone par les threads. Cela est utile pour paralléliser des blocs de code dont la répartition entre threads ne peut pas être déterminée statiquement à l'avance, comme les boucles irrégulières ou les calculs récursifs.

Avantages :

- **Asynchronie :** Permet de créer des tâches indépendantes qui peuvent être exécutées en parallèle de manière flexible.
- **Parallélisme fin :** Adapté pour les tâches dynamiques ou les structures de données irrégulières.

Scénarios d'utilisation : Parfait pour les calculs récursifs ou les algorithmes dynamiques où les tâches sont créées au fur et à mesure.

Exemple :

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

```c
#include <omp.h>

void example_task(int i) {
    printf("Thread %d: Task %d\n", omp_get_thread_num(), i);
}

int main() {
    int N = 100;

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < N; i++) {
                #pragma omp task
                example_task(i);  // Creation de taches paralleles
            }
        }
    }
    return 0;
}
```

2.8.2 Directive #pragma omp single

Fonctionnement : La directive #pragma omp single permet de spécifier qu'un seul thread (peu importe lequel) exécutera le bloc de code qui suit. Les autres threads dans la région parallèle attendront que le bloc de code soit terminé, sauf si l'option nowait est utilisée. Cette directive est souvent utilisée pour effectuer des initialisations ou des opérations qui doivent être réalisées une seule fois dans une région parallèle.

Avantages :

- **Exécution exclusive :** Garantit que le bloc de code est exécuté par un seul thread, évitant les conflits ou la duplication des efforts pour des opérations spécifiques.
- **Simplicité de synchronisation :** Les autres threads attendent automatiquement la fin de la section single, simplifiant la gestion de la synchronisation.
- **Optimisation des ressources :** Évite les surcharges de synchronisation lorsque le bloc de code doit être exécuté une seule fois.

Scénarios d'utilisation : Utilisé pour initialiser des variables globales ou effectuer des préparations nécessaires avant le début des calculs parallèles.

Exemple :

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

int  main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            // Cette section sera executee par un seul thread
            printf("This block is executed by thread %d\n",
   omp_get_thread_num());

            // Creation de taches paralleles
            #pragma omp task
            {
                printf("Task 1 executed by thread %d\n",
   omp_get_thread_num());
            }

            #pragma omp task
            {
                printf("Task 2 executed by thread %d\n",
   omp_get_thread_num());
            }
        }

        // Attente de la fin des taches
        #pragma omp taskwait
    }

    return 0;
}
```

3 Techniques d'Optimisation des Boucles

3.1 Déroulage de Boucle (Loop Unrolling)

Fonctionnement : Le déroulage de boucle consiste à augmenter le travail effectué par chaque itération d'une boucle en "déroulant" certaines itérations. Cela réduit le nombre total d'ité-

rations et diminue les coûts associés aux instructions de contrôle de boucle comme l'incrémentation de compteur et la vérification de la condition de sortie.

Avantages :

- **Amélioration des performances :** Réduit la surcharge liée au contrôle des boucles et permet une meilleure utilisation du pipeline du processeur.

Scénarios d'utilisation : Utile pour des boucles où les calculs à chaque itération sont simples et répétitifs. Le déroulage peut être particulièrement efficace en combinaison avec OpenMP, surtout sur des systèmes avec de nombreux cœurs.

Exemple :

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

int main() {
    int N = 1000;
    int array[N];

    #pragma omp parallel for
    for (int i = 0; i < N; i += 4) {
        array[i] = array[i] * 2;
        array[i + 1] = array[i + 1] * 2;
        array[i + 2] = array[i + 2] * 2;
        array[i + 3] = array[i + 3] * 2;
    }
    return 0;
}
```

3.2 Fusion de Boucles (Loop Fusion)

Fonctionnement : La fusion de boucles consiste à combiner plusieurs boucles indépendantes itérant sur la même plage d'indices en une seule boucle. Cela peut améliorer la localité des données et réduire le nombre total d'instructions de boucle exécutées, optimisant ainsi les performances.

Avantages :

- **Réduction de la surcharge :** Diminue la gestion des structures de contrôle de boucle en combinant plusieurs boucles en une seule.

- **Optimisation de la mémoire cache :** Peut améliorer l'efficacité de la mémoire en réduisant le nombre d'allers-retours à la mémoire.

Scénarios d'utilisation : Utile pour les applications où plusieurs boucles distinctes doivent traiter les mêmes ensembles de données, ou lorsque la mémoire cache peut être mieux exploitée en combinant les calculs.

Exemple :

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

int main() {
    int N = 1000;
    int array1[N], array2[N];

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        array1[i] = array1[i] * 2;
        array2[i] = array2[i] + 5;
    }
    return 0;
}
```

4 Métriques de performance

Dans un programme parallèle, la mesure des performances est essentielle pour évaluer l'efficacité et la pertinence des optimisations. Les programmes parallélisés avec OpenMP nécessitent une analyse approfondie pour comprendre comment ils utilisent les ressources et où se trouvent les opportunités d'amélioration. Quatre métriques principales permettent d'évaluer ces performances : **Speedup**, **Efficacité**, **Travail** (Work) et **Overhead**.

4.1 Speedup

Le speedup mesure l'amélioration des performances d'un programme parallèle par rapport à sa version séquentielle :

$$S = \frac{T_s}{T_p}$$

— S est le Speedup,
— T_s est le temps d'exécution séquentiel,
— T_p est le temps d'exécution parallèle.

Un speedup plus élevé indique que la parallélisation est efficace. Idéalement, le speedup devrait être proportionnel au nombre de threads utilisés.

4.2 Efficacité

L'efficacité mesure dans quelle mesure les ressources (threads) sont utilisées par rapport à l'optimal

$$E = \frac{S}{P} = \frac{T_s}{P \times T_p}$$

— E est l'efficacité,
— S est le Speedup,
— P est le nombre de threads (processeurs),
— T_s est le temps d'exécution séquentiel,

— T_p est le temps d'exécution parallèle.

Une efficacité élevée signifie que le programme tire bien parti de la parallélisation, tandis qu'une efficacité faible peut signaler un surcoût ou une mauvaise gestion des ressources.

4.3 Travail (Work)

Le travail représente la quantité totale de travail effectuée par le programme, mesurée en termes d'opérations élémentaires ou en temps CPU.

$$W = P \times T_p$$

— W est le travail total,
— P est le nombre de threads (processeurs),
— T_p est le temps d'exécution parallèle.

Comprendre le travail total aide à évaluer le potentiel d'optimisation. Un travail faible peut limiter les bénéfices de la parallélisation, même avec plusieurs threads.

4.4 Overhead

L'overhead est le temps ou les ressources supplémentaires nécessaires pour gérer la parallélisation, y compris le temps de création et de destruction des threads, et le temps consacré à la synchronisation.

$$O = P \times T_p - T_s$$

— O est l'overhead,
— P est le nombre de threads (processeurs),
— T_p est le temps d'exécution parallèle,
— T_s est le temps d'exécution séquentiel.

Un overhead élevé peut réduire l'efficacité d'un programme parallèle. Minimiser l'overhead est crucial pour améliorer les performances globales.

Ces quatre métriques fournissent une base solide pour évaluer et optimiser les performances des programmes OpenMP.

5 Impact des Techniques d'Optimisation des Boucles sur les Performances

Dans cette section, nous analyserons l'impact des différentes techniques d'optimisation des boucles, notamment le déroulage de boucle (loop unrolling) et la fusion de boucles (loop fusion), sur les performances des programmes parallèles. L'objectif est de comparer les temps d'exécution avant et après l'application de ces optimisations et d'évaluer leur influence sur l'efficacité de la parallélisation.

5.1 Implémentation Techniques d'Optimisation des Boucle

Pour cette étude, plusieurs scénarios de tests seront mis en place :

5.1.1 Boucle Séquentielle Simple

Ce test exécute une boucle séquentielle simple qui traite un tableau en effectuant plusieurs opérations mathématiques sur ses éléments. Chaque élément du tableau est manipulé à l'aide d'opérations telles que la racine carrée, le logarithme, le sinus et le cosinus, et les résultats sont stockés dans le même tableau. Ce test sert de référence pour évaluer les performances des boucles parallèles dans les tests suivants.

```
// ------------------------------------------------------------------
// Simple Loops
// ------------------------------------------------------------------
for (int i = 0; i < N; i++) {
    double temp = array1[i] * 1.5;
    double sqrt_value = sqrt(temp);
    double log_value = log(sqrt_value);
    double sin_value = sin(log_value);
    double result = cos(sin_value);

    array1[i] = (int)((result * 100) + array1[i] % 100);
}
for (int i = 0; i < N; i++) {
    double temp = array2[i] * 1.5;
    double sqrt_value = sqrt(temp);
    double log_value = log(sqrt_value);
    double sin_value = sin(log_value);
    double result = cos(sin_value);

```

```
    array2[i] = (int)((result * 100) + array2[i] % 100);
}
```

5.1.2 Déroulement de Boucle Séquentielle

Ce test introduit l'optimisation du déroulement de boucle pour une exécution séquentielle. En regroupant plusieurs itérations d'une boucle dans un seul bloc d'instructions, on cherche à réduire le nombre de sauts conditionnels et à améliorer les performances. Les calculs mathématiques réalisés dans le test précédent sont optimisés par le déroulement des itérations, et l'impact sur les performances est mesuré.

```
// ------------------------------------------------------------------
// Loop Unrolling Applied
// ------------------------------------------------------------------
for (int i = 0; i < N; i += 4) {

    // I
    double temp = array1[i] * 1.5;
    double sqrt_value = sqrt(temp);
    double log_value = log(sqrt_value);
    double sin_value = sin(log_value);
    double result = cos(sin_value);
    array1[i] = (int)((result * 100) + array1[i] % 100);

    // I + 1
    temp = array1[i + 1] * 1.5;
    sqrt_value = sqrt(temp);
    log_value = log(sqrt_value);
    sin_value = sin(log_value);
    result = cos(sin_value);
    array1[i + 1] = (int)((result * 100) + array1[i + 1] % 100);

    // I + 2
    temp = array1[i + 2] * 1.5;
    sqrt_value = sqrt(temp);
    log_value = log(sqrt_value);
    sin_value = sin(log_value);
    result = cos(sin_value);
    array1[i + 2] = (int)((result * 100) + array1[i + 2] % 100);

    // I + 3
    temp = array1[i + 1] * 1.5;
```

```
    sqrt_value = sqrt(temp);
    log_value = log(sqrt_value);
    sin_value = sin(log_value);
    result = cos(sin_value);
    array1[i + 3] = (int)((result * 100) + array1[i + 3] % 100);
}

// ------------------------------------------------------------------
// Loop Unrolling Applied
// ------------------------------------------------------------------
for (int i = 0; i < N; i += 4) {

    // I
    double temp = array2[i] * 1.5;
    double sqrt_value = sqrt(temp);
    double log_value = log(sqrt_value);
    double sin_value = sin(log_value);
    double result = cos(sin_value);
    array2[i] = (int)((result * 100) + array2[i] % 100);

    // I + 1
    temp = array2[i + 1] * 1.5;
    sqrt_value = sqrt(temp);
    log_value = log(sqrt_value);
    sin_value = sin(log_value);
    result = cos(sin_value);
    array2[i + 1] = (int)((result * 100) + array2[i + 1] % 100);

    // I + 2
    temp = array2[i + 2] * 1.5;
    sqrt_value = sqrt(temp);
    log_value = log(sqrt_value);
    sin_value = sin(log_value);
    result = cos(sin_value);
    array2[i + 2] = (int)((result * 100) + array2[i + 2] % 100);

    // I + 3
    temp = array2[i + 1] * 1.5;
    sqrt_value = sqrt(temp);
    log_value = log(sqrt_value);
    sin_value = sin(log_value);
    result = cos(sin_value);
```

```
74      array2[i + 3] = (int)((result * 100) + array2[i + 3] % 100);
75  }
```

5.1.3 Fusion de Boucles Séquentielles

Dans ce test, plusieurs boucles séquentielles indépendantes sont fusionnées en une seule. Cette technique vise à réduire les frais généraux liés à l'organisation des boucles en minimisant le nombre d'itérations et en permettant une exécution plus efficace. Les calculs précédemment effectués dans des boucles séparées sont combinés dans une seule boucle itérative, et les performances sont évaluées par rapport à l'exécution séquentielle simple.

```
1   // ------------------------------------------------------------------
2   // Loop Fusion Applied
3   // ------------------------------------------------------------------
4   for (int i = 0; i < N; i++) {
5       double temp1 = array1[i] * 2;
6       double sqrt_value1 = sqrt(temp1);
7       double log_value1 = log(sqrt_value1);
8       double sin_value1 = sin(log_value1);
9       double result1 = cos(sin_value1);
10      array1[i] = (int)((result1 * 100) + array1[i] % 100);
11
12      double temp2 = array2[i] + 5;
13      double sqrt_value2 = sqrt(temp2);
14      double log_value2 = log(sqrt_value2);
15      double sin_value2 = sin(log_value2);
16      double result2 = cos(sin_value2);
17      array2[i] = (int)((result2 * 100) + array2[i] % 100);
18  }
```

5.1.4 Optimisations Combinées Séquentielles

Ce test combine les techniques de déroulement de boucle et de fusion de boucles séquentielles. L'objectif est d'optimiser au maximum l'exécution séquentielle en réduisant les frais généraux associés à la gestion des boucles tout en augmentant l'efficacité des calculs. Les performances de cette approche combinée sont mesurées et comparées aux autres tests séquentiels.

```
1   for (int i = 0; i < N; i += 4) {
2
3       // Processing for array1
```

```
    if (i < N) {
        double temp = array1[i] * 1.5;
        double sqrt_value = sqrt(temp);
        double log_value = log(sqrt_value);
        double sin_value = sin(log_value);
        double result = cos(sin_value);
        array1[i] = (int)((result * 100) + array1[i] % 100);
    }
    if (i + 1 < N) {
        double temp = array1[i + 1] * 1.5;
        double sqrt_value = sqrt(temp);
        double log_value = log(sqrt_value);
        double sin_value = sin(log_value);
        double result = cos(sin_value);
        array1[i + 1] = (int)((result * 100) + array1[i + 1] % 100);
    }
    if (i + 2 < N) {
        double temp = array1[i + 2] * 1.5;
        double sqrt_value = sqrt(temp);
        double log_value = log(sqrt_value);
        double sin_value = sin(log_value);
        double result = cos(sin_value);
        array1[i + 2] = (int)((result * 100) + array1[i + 2] % 100);
    }
    if (i + 3 < N) {
        double temp = array1[i + 3] * 1.5;
        double sqrt_value = sqrt(temp);
        double log_value = log(sqrt_value);
        double sin_value = sin(log_value);
        double result = cos(sin_value);
        array1[i + 3] = (int)((result * 100) + array1[i + 3] % 100);
    }

    // Processing for array2
    if (i < N) {
        double temp = array2[i] * 1.5;
        double sqrt_value = sqrt(temp);
        double log_value = log(sqrt_value);
        double sin_value = sin(log_value);
        double result = cos(sin_value);
        array2[i] = (int)((result * 100) + array2[i] % 100);
    }
```

```
    if (i + 1 < N) {
        double temp = array2[i + 1] * 1.5;
        double sqrt_value = sqrt(temp);
        double log_value = log(sqrt_value);
        double sin_value = sin(log_value);
        double result = cos(sin_value);
        array2[i + 1] = (int)((result * 100) + array2[i + 1] % 100);
    }
    if (i + 2 < N) {
        double temp = array2[i + 2] * 1.5;
        double sqrt_value = sqrt(temp);
        double log_value = log(sqrt_value);
        double sin_value = sin(log_value);
        double result = cos(sin_value);
        array2[i + 2] = (int)((result * 100) + array2[i + 2] % 100);
    }
    if (i + 3 < N) {
        double temp = array2[i + 3] * 1.5;
        double sqrt_value = sqrt(temp);
        double log_value = log(sqrt_value);
        double sin_value = sin(log_value);
        double result = cos(sin_value);
        array2[i + 3] = (int)((result * 100) + array2[i + 3] % 100);
    }
}
```

5.1.5 Boucle Parallèle Simple avec OpenMP

Ce test implémente une boucle simple qui est parallélisée à l'aide d'OpenMP. L'objectif est de comparer les performances d'une exécution séquentielle (sans OpenMP) avec celles d'une exécution parallèle. Les itérations de la boucle effectuent des calculs mathématiques sur les éléments d'un tableau, et les résultats sont stockés.

```
// ------------------------------------------------------------------
// Simple parallel loop with OpenMP
// ------------------------------------------------------------------
for (int test = 0; test < totalTestCounts; test++) {

    // Set the number of threads
    int threadsCount = threadCounts[test];
    omp_set_num_threads(threadsCount);

```

```
    // Initialize thread execution times to zero
    double thread_times[threadsCount];
    for (int i = 0; i < threadsCount; i++) {
        thread_times[i] = 0.0;  // Initialize to zero
    }

    #pragma omp parallel
    {

        #pragma omp for schedule(guided)
        for (int i = 0; i < N; i++) {
            double thread_start_time = omp_get_wtime();

            double temp = array1[i] * 1.5;
            double sqrt_value = sqrt(temp);
            double log_value = log(sqrt_value);
            double sin_value = sin(log_value);
            double result = cos(sin_value);

            array1[i] = (int)((result * 100) + array1[i] % 100);

            double thread_end_time = omp_get_wtime();

            // Accumulate time
            #pragma omp atomic
            thread_times[omp_get_thread_num()] += thread_end_time -
thread_start_time;
        }

        #pragma omp for schedule(guided)
        for (int i = 0; i < N; i++) {
            double thread_start_time = omp_get_wtime();

            double temp = array2[i] * 1.5;
            double sqrt_value = sqrt(temp);
            double log_value = log(sqrt_value);
            double sin_value = sin(log_value);
            double result = cos(sin_value);

            array2[i] = (int)((result * 100) + array2[i] % 100);

            double thread_end_time = omp_get_wtime();
```

```
51
52
53
54              // Accumulate time
55              #pragma omp atomic
56              thread_times[omp_get_thread_num()] += thread_end_time -
    thread_start_time ;
57          }
58      }
59  }
```

5.1.6 Déroulement de Boucle Parallèle avec OpenMP

Ce test applique l'optimisation du déroulement de boucle dans un contexte parallèle avec OpenMP. En exécutant plusieurs itérations en parallèle et en les regroupant, ce test vise à réduire les sauts conditionnels et à maximiser les performances.

```
1  // ------------------------------------------------------------------
2  // Loop Unrolling
3  // ------------------------------------------------------------------
4  for (int test = 0; test < totalTestCounts; test++) {
5
6      // Set the number of threads
7      int threadsCount = threadCounts[test];
8      omp_set_num_threads(threadsCount);
9
10     // Initialize thread execution times to zero
11     double thread_times[threadsCount];
12     for (int i = 0; i < threadsCount; i++) {
13         thread_times[i] = 0.0;  // Initialize to zero
14     }
15
16     #pragma omp parallel
17     {
18         // Loop Unrolling Applied
19         #pragma omp for schedule(guided)
20         for (int i = 0; i < N; i += 4) {
21             double thread_start_time = omp_get_wtime();
22
23             // I
24             double temp = array1[i] * 1.5;
25             double sqrt_value = sqrt(temp);
```

```
            double log_value = log(sqrt_value);
            double sin_value = sin(log_value);
            double result = cos(sin_value);
            array1[i] = (int)((result * 100) + array1[i] % 100);

            // I + 1
            temp = array1[i + 1] * 1.5;
            sqrt_value = sqrt(temp);
            log_value = log(sqrt_value);
            sin_value = sin(log_value);
            result = cos(sin_value);
            array1[i + 1] = (int)((result * 100) + array1[i + 1] % 100)
    ;

            // I + 2
            temp = array1[i + 2] * 1.5;
            sqrt_value = sqrt(temp);
            log_value = log(sqrt_value);
            sin_value = sin(log_value);
            result = cos(sin_value);
            array1[i + 2] = (int)((result * 100) + array1[i + 2] % 100)
    ;

            // I + 3
            temp = array1[i + 1] * 1.5;
            sqrt_value = sqrt(temp);
            log_value = log(sqrt_value);
            sin_value = sin(log_value);
            result = cos(sin_value);
            array1[i + 3] = (int)((result * 100) + array1[i + 3] % 100)
    ;

            double thread_end_time = omp_get_wtime();

            // Accumulate time
            #pragma omp atomic
            thread_times[omp_get_thread_num()] += thread_end_time -
    thread_start_time;
        }

        // Loop Unrolling Applied
        #pragma omp for schedule(guided)
```

```
64          for (int i = 0; i < N; i += 4)
65          {
66              double thread_start_time = omp_get_wtime();
67
68              // I
69              double temp = array2[i] * 1.5;
70              double sqrt_value = sqrt(temp);
71              double log_value = log(sqrt_value);
72              double sin_value = sin(log_value);
73              double result = cos(sin_value);
74              array2[i] = (int)((result * 100) + array2[i] % 100);
75
76              // I + 1
77              temp = array2[i + 1] * 1.5;
78              sqrt_value = sqrt(temp);
79              log_value = log(sqrt_value);
80              sin_value = sin(log_value);
81              result = cos(sin_value);
82              array2[i + 1] = (int)((result * 100) + array2[i + 1] % 100)
    ;
83
84              // I + 2
85              temp = array2[i + 2] * 1.5;
86              sqrt_value = sqrt(temp);
87              log_value = log(sqrt_value);
88              sin_value = sin(log_value);
89              result = cos(sin_value);
90              array2[i + 2] = (int)((result * 100) + array2[i + 2] % 100)
    ;
91
92              // I + 3
93              temp = array2[i + 1] * 1.5;
94              sqrt_value = sqrt(temp);
95              log_value = log(sqrt_value);
96              sin_value = sin(log_value);
97              result = cos(sin_value);
98              array2[i + 3] = (int)((result * 100) + array2[i + 3] % 100)
    ;
99
100             double thread_end_time = omp_get_wtime();
101
102             // Accumulate time
```

```
            #pragma omp atomic
            thread_times[omp_get_thread_num()] += thread_end_time -
    thread_start_time;
        }
    }
}
```

5.1.7 Fusion de Boucles Parallèles avec OpenMP

CDans ce test, plusieurs boucles parallèles indépendantes sont fusionnées en une seule boucle. Cela permet de minimiser le coût de gestion des threads et d'optimiser l'utilisation des ressources. Les performances de cette approche sont mesurées pour déterminer l'efficacité de la fusion des boucles parallèles par rapport la fusion des boucles séquentielles.

```
// ------------------------------------------------------------
// Loop Fusion Applied with OpenMP
// ------------------------------------------------------------
for (int test = 0; test < totalTestCounts; test++) {

    // Set the number of threads
    int threadsCount = threadCounts[test];
    omp_set_num_threads(threadsCount);

    // Initialize thread execution times to zero
    double thread_times[threadsCount];
    for (int i = 0; i < threadsCount; i++) {
        thread_times[i] = 0.0;  // Initialize to zero
    }

    #pragma omp parallel
    {
        #pragma omp for schedule(guided)
        for (int i = 0; i < N; i++) {
            double thread_start_time = omp_get_wtime();

            double temp1 = array1[i] * 2;
            double sqrt_value1 = sqrt(temp1);
            double log_value1 = log(sqrt_value1);
            double sin_value1 = sin(log_value1);
            double result1 = cos(sin_value1);
            array1[i] = (int)((result1 * 100) + array1[i] % 100);

```

```
            double temp2 = array2[i] + 5;
            double sqrt_value2 = sqrt(temp2);
            double log_value2 = log(sqrt_value2);
            double sin_value2 = sin(log_value2);
            double result2 = cos(sin_value2);
            array2[i] = (int)((result2 * 100) + array2[i] % 100);

            double thread_end_time = omp_get_wtime();

            // Accumulate time
            #pragma omp atomic
            thread_times[omp_get_thread_num()] += thread_end_time -
    thread_start_time;
        }
    }

}
```

5.1.8 Optimisations Combinées Parallèles avec OpenMP

Ce test combine le déroulement de boucle et la fusion de boucles parallèles, utilisant OpenMP pour la parallélisation. L'objectif est d'optimiser au maximum les calculs en exploitant plusieurs threads tout en minimisant les frais généraux associés. Les performances sont évaluées pour déterminer l'efficacité de cette approche combinée.

```

// ------------------------------------------------------------------
// Combination of Loop Unrolling and Loop Fusion with OpenMP
// ------------------------------------------------------------------
for (int test = 0; test < totalTestCounts; test++) {

    // Set the number of threads
    int threadsCount = threadCounts[test];
    omp_set_num_threads(threadsCount);

    // Initialize thread execution times to zero
    double thread_times[threadsCount];
    for (int i = 0; i < threadsCount; i++) {
        thread_times[i] = 0.0;  // Initialize to zero
    }

    #pragma omp parallel
```

```
18      {
19          #pragma omp for schedule(guided)
20          for (int i = 0; i < N; i += 4) {
21
22              double thread_start_time = omp_get_wtime();
23
24              if (i < N) {
25                  double temp = array1[i] * 1.5;
26                  double sqrt_value = sqrt(temp);
27                  double log_value = log(sqrt_value);
28                  double sin_value = sin(log_value);
29                  double result = cos(sin_value);
30                  array1[i] = (int)((result * 100) + array1[i] % 100);
31              }
32              if (i + 1 < N) {
33                  double temp = array1[i + 1] * 1.5;
34                  double sqrt_value = sqrt(temp);
35                  double log_value = log(sqrt_value);
36                  double sin_value = sin(log_value);
37                  double result = cos(sin_value);
38                  array1[i + 1] = (int)((result * 100) + array1[i + 1] %
    100);
39              }
40              if (i + 2 < N) {
41                  double temp = array1[i + 2] * 1.5;
42                  double sqrt_value = sqrt(temp);
43                  double log_value = log(sqrt_value);
44                  double sin_value = sin(log_value);
45                  double result = cos(sin_value);
46                  array1[i + 2] = (int)((result * 100) + array1[i + 2] %
    100);
47              }
48              if (i + 3 < N) {
49                  double temp = array1[i + 3] * 1.5;
50                  double sqrt_value = sqrt(temp);
51                  double log_value = log(sqrt_value);
52                  double sin_value = sin(log_value);
53                  double result = cos(sin_value);
54                  array1[i + 3] = (int)((result * 100) + array1[i + 3] %
    100);
55              }
56
```

```
57              // Processing for array2
58              if (i < N) {
59                  double temp = array2[i] * 1.5;
60                  double sqrt_value = sqrt(temp);
61                  double log_value = log(sqrt_value);
62                  double sin_value = sin(log_value);
63                  double result = cos(sin_value);
64                  array2[i] = (int)((result * 100) + array2[i] % 100);
65              }
66              if (i + 1 < N) {
67                  double temp = array2[i + 1] * 1.5;
68                  double sqrt_value = sqrt(temp);
69                  double log_value = log(sqrt_value);
70                  double sin_value = sin(log_value);
71                  double result = cos(sin_value);
72                  array2[i + 1] = (int)((result * 100) + array2[i + 1] %
    100);
73              }
74              if (i + 2 < N) {
75                  double temp = array2[i + 2] * 1.5;
76                  double sqrt_value = sqrt(temp);
77                  double log_value = log(sqrt_value);
78                  double sin_value = sin(log_value);
79                  double result = cos(sin_value);
80                  array2[i + 2] = (int)((result * 100) + array2[i + 2] %
    100);
81              }
82              if (i + 3 < N) {
83                  double temp = array2[i + 3] * 1.5;
84                  double sqrt_value = sqrt(temp);
85                  double log_value = log(sqrt_value);
86                  double sin_value = sin(log_value);
87                  double result = cos(sin_value);
88                  array2[i + 3] = (int)((result * 100) + array2[i + 3] %
    100);
89              }
90
91              double thread_end_time = omp_get_wtime();
92
93              // Accumulate time
94              #pragma omp atomic
```

```
95              thread_times[omp_get_thread_num()] += thread_end_time -
    thread_start_time;
96          }
97      }
98
99  }
```

5.2 Tests de Performance : Temps d'exécution

Cette section présente une analyse comparative des temps d'exécution des différents programmes implémentés, tant en version séquentielle qu'en version parallèle. Le diagramme ci-dessous illustre les performances des programmes en fonction du nombre de threads utilisés.

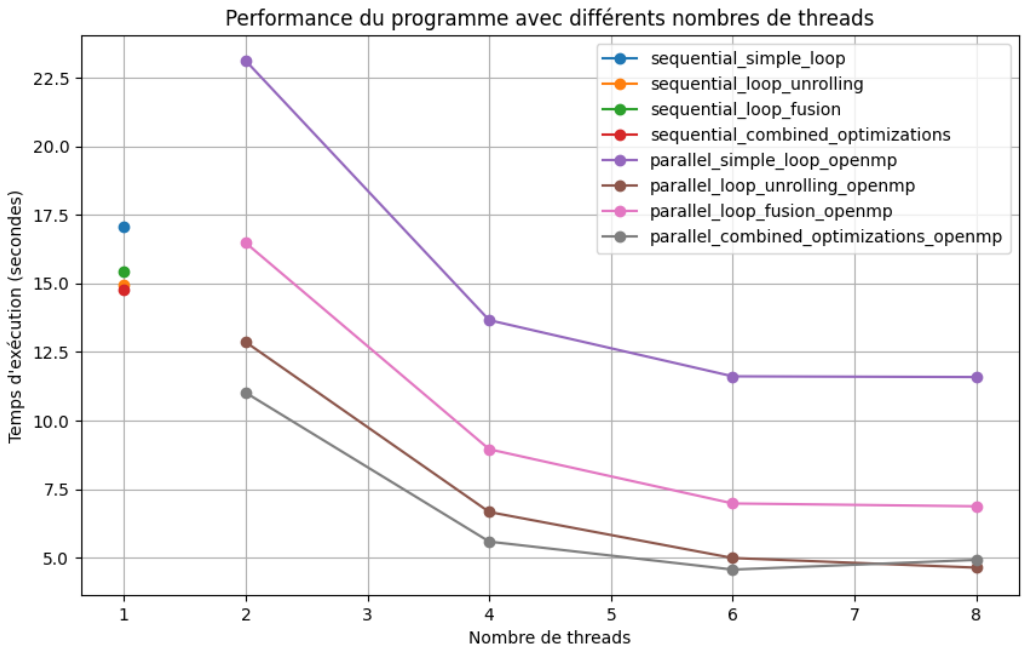

FIGURE 1 – Tests de Performance : Temps d'exécution

- **Performance des Programmes Séquentiels :** Les programmes séquentiels affichent des temps d'exécution plus longs, ce qui souligne les limitations de cette approche, notamment pour des charges de travail importantes.
- **Impact de la Parallélisation :** Les temps d'exécution des programmes parallèles diminuent de manière significative avec l'augmentation du nombre de threads. Cela

indique une répartition efficace de la charge de travail entre les threads.

- **Optimisations Combinées :** Les programmes qui intègrent des optimisations combinées (le déroulement de boucle et la fusion de boucles) montrent des temps d'exécution nettement inférieurs, illustrant l'importance d'utiliser plusieurs techniques d'optimisation en tandem.
- **Diminutions de Performance :** Il est important de noter que, pour certains programmes, l'augmentation du nombre de threads peut ne pas toujours conduire à une amélioration des performances, ce qui peut être dû à la surcharge de gestion des threads ou à la contention des ressources.

5.3 Tests de Performance : SpeedUp

Nous avons mesuré le SpeedUp de nos programmes parallèles par rapport aux versions séquentielles. Le SpeedUp est un indicateur essentiel qui permet d'évaluer l'amélioration de la performance grâce à la parallélisation. Il est défini comme le rapport entre le temps d'exécution du programme séquentiel et celui du programme parallèle.

Les résultats de ces mesures sont illustrés dans le diagramme ci-dessous :

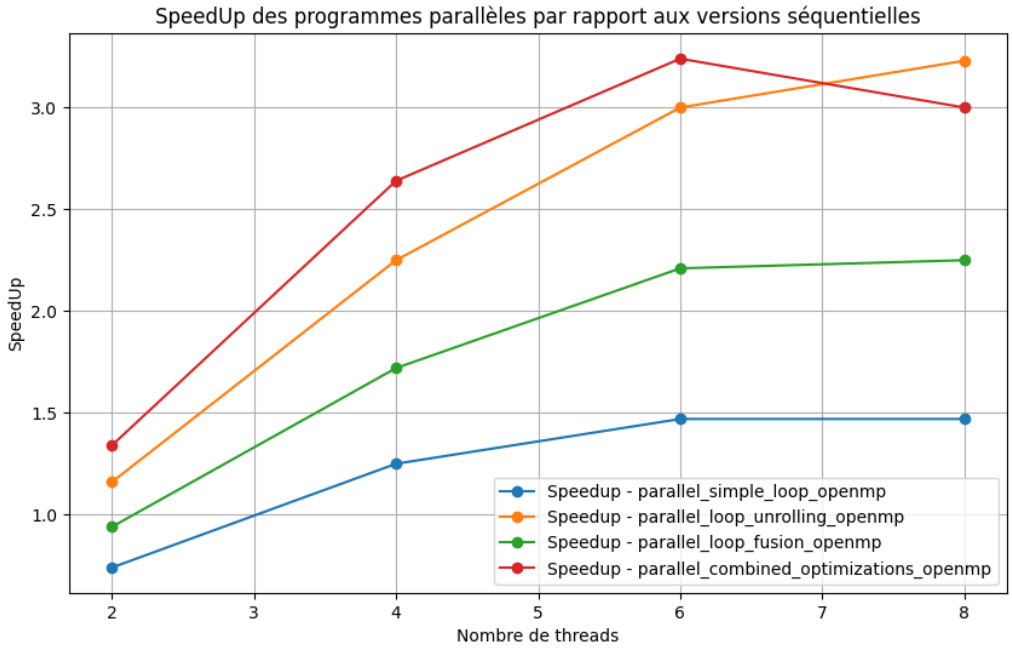

FIGURE 2 – Tests de Performance : Temps d'exécution

5.4 Tests de Performance : Efficacité

Pour compléter notre analyse, nous avons également évalué l'efficacité des programmes parallèles. L'efficacité représente le rapport entre le SpeedUp et le nombre de threads utilisés, offrant une indication précieuse sur l'utilisation des ressources.

Les résultats de ces mesures sont illustrés dans le diagramme ci-dessous :

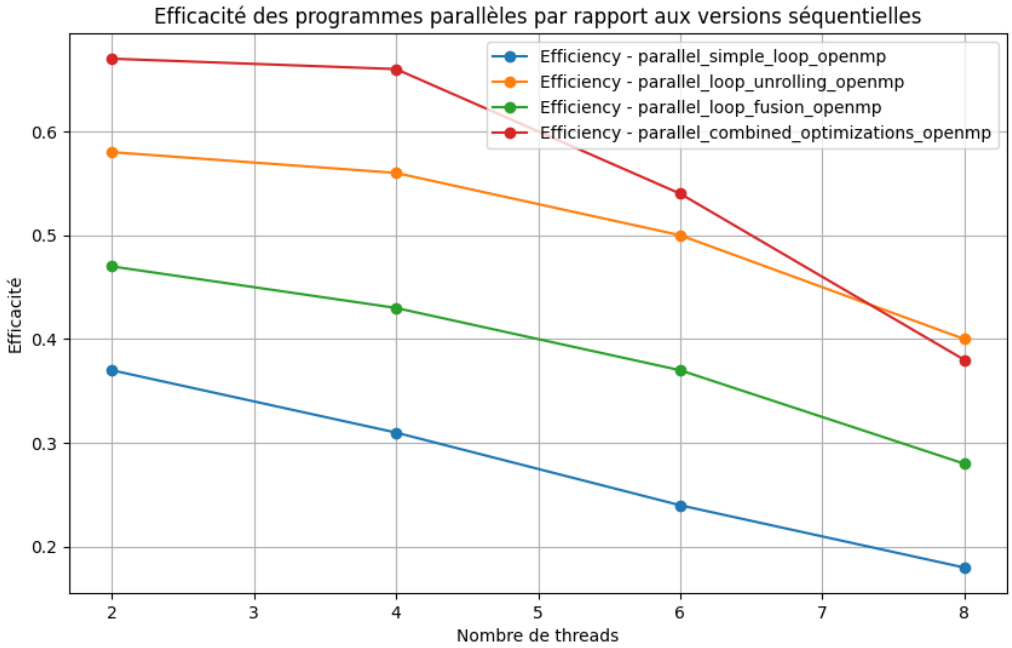

FIGURE 3 – Tests de Performance : Efficacité

5.5 Tests de Performance : Work

Nous avons également mesuré le "Work", qui correspond à la charge de travail totale effectuée par chaque programme. Cela se traduit par le produit du temps d'exécution et du nombre de threads. Cette mesure nous aide à comprendre la charge de travail réelle de chaque technique.

Les résultats de ces mesures sont illustrés dans le diagramme ci-dessous :

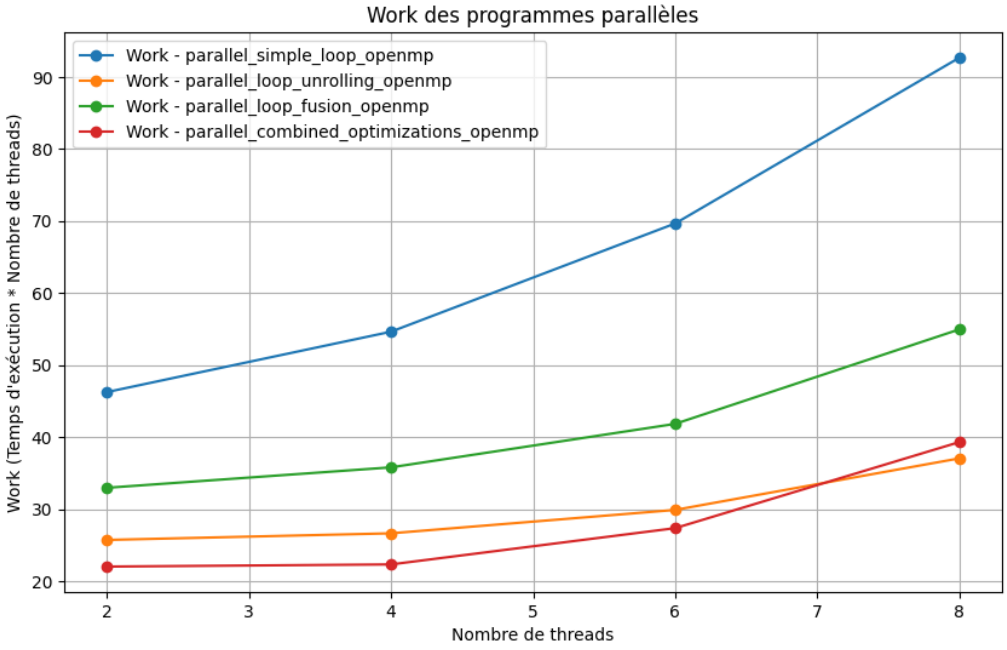

FIGURE 4 – Tests de Performance : Work

5.6 Tests de Performance : Overhead

Enfin, nous avons évalué le "Overhead", qui représente les coûts associés à la gestion des threads et à la synchronisation. Un overhead trop élevé peut réduire les gains de performance escomptés. Nous examinerons les résultats d'overhead pour chaque technique et leur impact sur la performance globale.

Les résultats de ces mesures sont illustrés dans le diagramme ci-dessous :

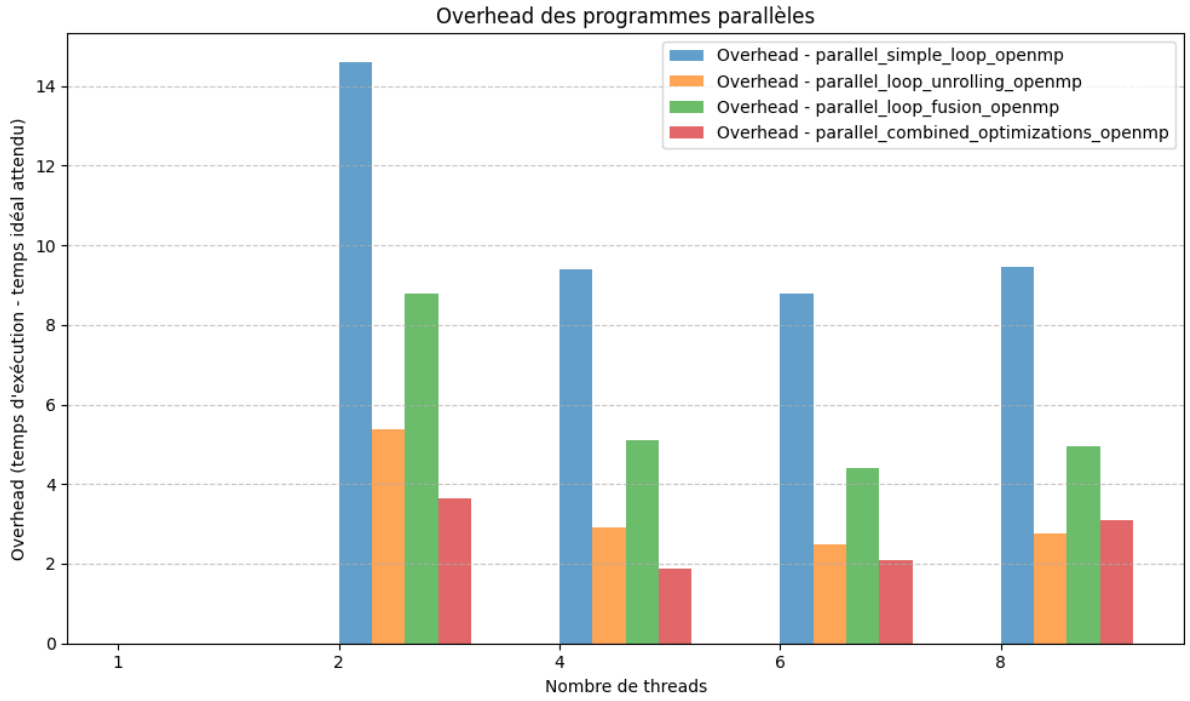

FIGURE 5 – Tests de Performance : Overhead

5.7 Résumé des Tests de Performance

Voici un tableau résumant les métriques clés pour chaque programme testé.

program_name	threads_count	execution_time	speedup	efficiency	work	overhead
sequential_simple_loop	1	17.058041	nan	nan	nan	nan
sequential_loop_unrolling	1	14.961129	nan	nan	nan	nan
sequential_loop_fusion	1	15.442099	nan	nan	nan	nan
sequential_combined_optimizations	1	14.756591	nan	nan	nan	nan
parallel_simple_loop_openmp	2	23.132451	0.74	0.05	46.26	14.6
parallel_simple_loop_openmp	4	13.662971	1.25	0.13	54.65	9.4
parallel_simple_loop_openmp	6	11.613298	1.47	0.17	69.68	8.77
parallel_simple_loop_openmp	8	11.5877	1.47	0.16	92.7	9.46
parallel_loop_unrolling_openmp	2	12.865141	1.16	0.22	25.73	5.39
parallel_loop_unrolling_openmp	4	6.66279	2.25	0.77	26.65	2.92
parallel_loop_unrolling_openmp	6	4.983059	3.0	1.2	29.9	2.49
parallel_loop_unrolling_openmp	8	4.632218	3.23	1.17	37.06	2.76
parallel_loop_fusion_openmp	2	16.485042	0.94	0.11	32.97	8.77
parallel_loop_fusion_openmp	4	8.952679	1.72	0.34	35.81	5.09
parallel_loop_fusion_openmp	6	6.974568	2.21	0.5	41.85	4.4
parallel_loop_fusion_openmp	8	6.870879	2.25	0.46	54.97	4.94
parallel_combined_optimizations_openmp	2	11.017359	1.34	0.37	22.03	3.64
parallel_combined_optimizations_openmp	4	5.582242	2.64	1.4	22.33	1.89
parallel_combined_optimizations_openmp	6	4.561367	3.24	1.54	27.37	2.1
parallel_combined_optimizations_openmp	8	4.915408	3.0	0.97	39.32	3.08

FIGURE 6 – Résumé des Tests de Performance

6 Conclusion

Dans ce rapport, nous avons exploré les différentes techniques d'optimisation des boucles dans les programmes parallèles à l'aide d'OpenMP. Nous avons tout d'abord présenté les différentes directives fournies par OpenMP pour faciliter la parallélisation des boucles, notamment celles permettant de diviser les itérations entre plusieurs threads et de gérer les exécutions de manière efficace. La gestion des sections critiques, des tâches, ainsi que la synchronisation entre threads a été abordée pour garantir une bonne coordination entre les threads parallèles.

L'optimisation de la répartition des boucles, à travers des directives comme schedule, permet d'adapter la répartition des itérations en fonction de la charge de travail, améliorant ainsi les performances globales du programme. De plus, les techniques de déroulage de boucles et de fusion de boucles, que nous avons étudiées dans le détail, jouent un rôle crucial dans la réduction du temps d'exécution en minimisant les surcoûts de contrôle et en améliorant l'exploitation des caches.

Les tests de performance que nous avons menés démontrent que ces techniques d'optimisation, appliquées judicieusement, permettent de significatives améliorations en termes de vitesse d'exécution. L'analyse comparative entre les boucles non optimisées, les boucles parallélisées sans optimisation et celles optimisées avec OpenMP a confirmé les gains en efficacité grâce à l'utilisation de ces techniques.

En conclusion, l'utilisation d'OpenMP pour l'optimisation des boucles dans les applications parallèles permet non seulement de simplifier la parallélisation des tâches complexes, mais également d'augmenter les performances des programmes de manière notable. La bonne maîtrise des directives et des optimisations proposées par OpenMP est essentielle pour exploiter pleinement les capacités des systèmes multicœurs modernes.