Compte rendue du projet OS202

Parallélisation d'une simulation de feu de forêt

Amine Maazizi
Ayoub Boufous
Wiam Benrguibi

15 mars 2025

Table des matières

1 Introduction

La simulation numérique d'un incendie permet d'en comprendre les dynamiques et d'optimiser les stratégies de prévention et d'intervention. Ce projet modélise **la propagation d'un feu de forêt** en fonction du vent et de la densité de végétation à partir d'une approche probabiliste. L'objectif est d'accélérer cette simulation en exploitant les capacités du **calcul parallèle** via `OpenMP` et `MPI`.

2 Caractéristiques Matérielles

La machine utilisée dispose de **12 cœurs physiques**. Le tableau ci-dessous présente les tailles des différentes mémoires caches :

Mémoire cache	Taille
L1d	576 KiB (12 instances)
L1i	384 KiB (12 instances)
L2	24 KiB (12 instances)
L3	30 MiB (1 instance)

Table 1 – Caractéristiques des mémoires caches.

```
Architecture:             x86_64
CPU op-mode(s):         32-bit, 64-bit
Address sizes:          39 bits physical, 48 bits virtual
Byte Order:             Little Endian
CPU(s):                   24
On-line CPU(s) list:    0-23
Vendor ID:                GenuineIntel
Model name:             Intel(R) Core(TM) i7-14650HX
  CPU family:           6
  Model:                183
  Thread(s) per core:   2
  Core(s) per socket:   12
  Socket(s):            1
  Stepping:             1
  BogoMIPS:             4838.39
```

3 Partie 1 : Parallélisation avec OpenMP

3.1 Modifications apportées

1. **Création d'un vecteur `fire_keys`**
   Ce vecteur est utilisé pour stocker les clés de `m_fire_front`.
2. **Ajout de `#pragma omp parallel for`**
   Cette directive parallélise la boucle sur les indices de `fire_keys`.
3. **Utilisation de `#pragma omp critical`**
   Cette directive sécurise les mises à jour de `m_fire_map` et de `next_front`, évitant ainsi les race conditions.

3.2 Validation de la parallélisation

Pour s'assurer que la simulation parallèle et séquentielle sont identiques, nous nous sommes basés sur deux critères :

1. **Indicateur initial :**
   Le nombre de pas de simulation est identique entre la version séquentielle et la version parallèle (544 pas).
2. **Analyse détaillée :**
   Nous avons constaté que le fait d'avoir 544 pas identiques ne garantit pas que les deux simulations évoluent de manière identique. En effet, nous avons comparé les valeurs de `m_fire_map` et `m_vegetation_map`, qui reflètent l'évolution détaillée du feu et de la végétation. Une parallélisation mal implémentée (par exemple, en cas de conditions de course ou de tirages pseudo-aléatoires désynchronisés) peut modifier ces cartes, car des mises à jour concurrentes ou désordonnées altèrent la propagation du feu par rapport à l'ordre strictement séquentiel. Pour comparer ces cartes de façon efficace (l'affichage complet étant impossible à cause du volume de données), nous avons suivi la suggestion d'un LLM : utiliser un **hash SHA-1**. Ce hash est calculé sur les cartes sérialisées dans un ordre fixe, et l'identité des hashes confirme, de manière rapide et fiable, l'équivalence des simulations.

Valeurs SHA-1 obtenues :

Code parallèle :

```
SHA-1 à t=1:   e351b2af36ea88cdf1f81379a92bfd56483e1b8a
SHA-1 à t=2:   02455e2cc0e2a18e4029740356af555a14efff66
SHA-1 à t=3:   5947dd2d99ababef3444ffe8fbf0d5fd56c8ee81
SHA-1 à t=4:   ab386c4a01054d0b9683236bd0a92214141c427b
SHA-1 à t=5:   764ee77c42b4a31e072af13d4352f1da71b084f4
...
SHA-1 à t=441: e84d09f6dc7f7fee8760220a6d7ccb92cbe88480
SHA-1 à t=442: e84d09f6dc7f7fee8760220a6d7ccb92cbe88480
SHA-1 à t=443: e84d09f6dc7f7fee8760220a6d7ccb92cbe88480
SHA-1 à t=444: e84d09f6dc7f7fee8760220a6d7ccb92cbe88480
SHA-1 à t=445: e84d09f6dc7f7fee8760220a6d7ccb92cbe88480
```

Code séquentiel :

```
SHA-1 à t=1:   e351b2af36ea88cdf1f81379a92bfd56483e1b8a
SHA-1 à t=2:   02455e2cc0e2a18e4029740356af555a14efff66
SHA-1 à t=3:   3ab075debbdb1a0d745d5acbf957246b0b997ead
SHA-1 à t=4:   ecb8d32e7f9db66089fafb7fbe700382792a0f85
SHA-1 à t=5:   0734ef47085e38c79bb0a2e7a59b43f81f8e735e
...
SHA-1 à t=441: 16c0778f0115e0faf296b1cb60bb1c936e07364e
SHA-1 à t=442: 16c0778f0115e0faf296b1cb60bb1c936e07364e
SHA-1 à t=443: 16c0778f0115e0faf296b1cb60bb1c936e07364e
SHA-1 à t=444: 16c0778f0115e0faf296b1cb60bb1c936e07364e
SHA-1 à t=445: 16c0778f0115e0faf296b1cb60bb1c936e07364e
```

Conclusion de la validation :
Les valeurs SHA-1 des versions parallèle et séquentielle sont identiques jusqu'à t=2, mais divergent dès t=3 (par exemple, `5947dd2d...` pour la version parallèle contre `3ab075de...` pour la version séquentielle). Cette divergence révèle que la parallélisation perturbe l'évolution de la simulation, probablement à cause d'un ordre de mise à jour variable ou de tirages pseudo-aléatoires non synchronisés entre les threads. Bien que les deux versions atteignent un état stable (hash constant) à partir de t=441, elles ne suivent pas la même trajectoire, et plusieurs états stables sont observés en fin de simulation. Cela indique la nécessité d'une synchronisation plus rigoureuse des accès ou d'un alignement des graines aléatoires pour obtenir une équivalence parfaite avec la version séquentielle.

3.3 Problèmes rencontrés

1. **Initialement :**
   La parallélisation avec OpenMP a été implémentée sans utiliser `#pragma omp critical`. Avec un seul thread, tout fonctionnait correctement.
2. **Dès que 2 threads sont utilisés :**
   L'accès simultané de plusieurs threads aux mêmes éléments de `m_fire_map` ou `next_front` entraînait des écritures concurrentes non synchronisées, ce qui causait une race condition et provoquait immédiatement un segmentation fault.

3.4 Résultats et Analyse

Threads	Iteration Time	Simu Update Time	Display Update Time
1	0.0015163576	3.351614e-05	0.00147673
2	0.001053014	2.857406e-05	0.0010191752
4	0.0010958876	2.993814e-05	0.0010604654
8	0.0012948252	3.033716e-05	0.0012595776
16	0.0011116396	2.908858e-05	0.0010777414
24	0.0010113618	2.81815e-05	0.0009786056

TABLE 2 – Valeurs de performance en fonction du nombre de threads

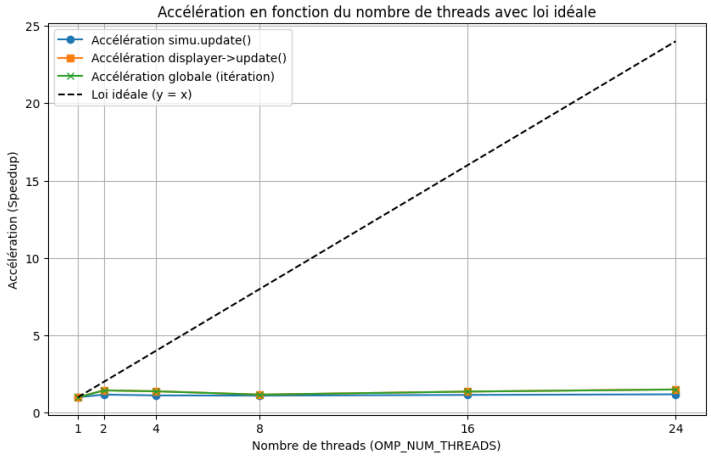

FIGURE 1 – Speed-Up Pour la parallélisation avec OpenMP.

La courbe obtenue montre une accélération loin de l'idéal : le speedup plafonne rapidement et présente même des fluctuations au-delà de 4 threads. Cela indique que l'augmentation du nombre de threads n'apporte pas une amélioration linéaire des performances. Cela peut être justifié par :

— Overhead de création et synchronisation des threads : la gestion des threads introduit un coût supplémentaire, notamment lors de leur création, de la répartition des tâches et de la synchronisation entre eux.

— Usage excessif de `#pragma omp critical` à plusieurs reprises lors des accès, ce qui entraîne une forte contention et limite la parallélisation efficace.

---

4 Partie 2 : Répartition des tâches avec MPI

4.1 Modifications apportées

1. **Séparation de la boucle principale**
   La boucle est divisée en deux parties distinctes :
   (a) **Affichage :** Géré par le processus 0.
   (b) **Calcul :** Géré par les autres processus.
2. **Échange initial de la géométrie**
   Le processus 1 envoie, de manière bloquante, la géométrie de la simulation qui est reçue également de manière bloquante par le processus 0. Même si la géométrie est accessible via les arguments de la ligne de commande pour tous les processus, cette approche permet de gérer un cas de géométrie variable dans la simulation (même si ce n'est pas le cas dans ce projet).
3. **Échanges pendant la boucle**
   La boucle principale comporte un envoi bloquant et une réception non bloquante des cartes de la végétation et du feu.

4.2 Problèmes rencontrés

1. **Séparation inadéquate des boucles**
   Initialement, une seule boucle combinait affichage et calcul sur les processus, ce qui entraînait l'initialisation de deux fenêtres d'affichage. Cette approche s'est vite avérée inefficace.
2. **Mismatch des types de données**
   Lors des envois MPI, un décalage entre les types de données a été constaté, problème qui a été rapidement corrigé.
3. **Réception non bloquante inadaptée**
   Lors de l'initialisation de la géométrie dans le processus 0, l'utilisation d'une réception non bloquante faisait que le processus ne patientait pas pour la géométrie. Cela conduisait à l'initialisation des tailles des cartes avec des valeurs aléatoires, provoquant ainsi des dysfonctionnements.

4.3 Commentaire

Dans cette partie, nous avons testé une approche non bloquante avec `MPI_Irecv` et `MPI_Waitall` afin de permettre un chevauchement entre calcul et communication. Cependant, pour ce problème précis, un récepteur bloquant (`MPI_Recv`) serait plus simple et tout aussi efficace, puisque les données sont attendues immédiatement sans travail intermédiaire. Le choix des appels non bloquants a été fait en anticipation d'une possible extension dans la Partie 4.

4.4 Résultats et Analyse

1. **Version parallèle (-n 2) :**
   Temps global moyen par itération : **0.00145327 secondes**.
2. **Version séquentielle :**
   Temps global moyen par itération : **0.00121068 secondes**.

La surcharge induite par la communication et la synchronisation entre les processus dans la version parallèle dépasse les gains obtenus par la répartition de la charge, rendant l'exécution séquentielle plus rapide.

---

5 Partie 3 : Combinaison d'OpenMP et MPI

5.1 Modifications apportées

1. Intégration du parallélisme OpenMP dans le code MPI de la Partie 2, en ajoutant une parallélisation de la boucle avec `#pragma omp parallel for`.

5.2 Résultats

1. **Exécution sur 2 cœurs et 4 threads :**
   Temps global moyen par itération : **0.0256515 secondes**.

5.3 Bonus : Double Buffering

Pour rendre l'affichage asynchrone par rapport à l'avancement de la simulation, nous avons mis en œuvre la technique du double buffering. Cette méthode consiste à alterner entre deux buffers : pendant que l'un est utilisé pour l'affichage, l'autre reçoit les nouvelles données de la simulation. Nous prévoyons ainsi une accélération globale grâce à la réduction des temps d'attente, même si celle-ci reste limitée par les surcoûts liés à la gestion des communications non bloquantes.
Temps global moyen par itération avec double buffering : 0.00146456 secondes.

5.4 Résultats et Analyse

5.4.1 Suivant la loi d'Amdhal

NUM_THREADS	Avg_step	Global_display	Global_sim
1	2.2507459999999998e-05	0.524455	0.8303746
2	2.2571540000000002e-05	0.5743192	0.8941092
4	2.1637675e-05	0.443273	0.6621305
8	2.179964e-05	0.5251876	0.9301234
16	2.044686e-05	0.4781602	0.685039
24	2.236216e-05	0.525885	0.7479326

TABLE 3 – Temps moyen (secondes) en fonction du nombre de threads pour trois mesures différentes

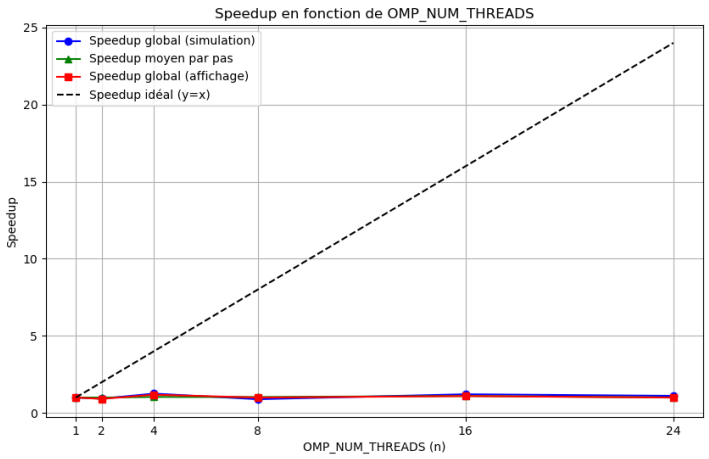

FIGURE 2 – Speed-Up Pour la parallélisation MPI et OpenMP.

Malgré la parallélisation, l'accélération globale reste modeste en raison de la part séquentielle du code et des surcoûts associés à la gestion des threads, conformément aux prédictions d'Amdahl.
Ce qui est justifié par :
- Fraction séquentielle non négligeable : La loi d'Amdahl prédit qu'une portion même minime de code séquentiel limite fortement le speedup global.

- Surcharges de parallélisation : La gestion des threads, la synchronisation et la communication entre ceux-ci engendrent des surcoûts. Ces surcoûts, qui augmentent avec le nombre de threads, compensent en partie les bénéfices du parallélisme.

5.4.2 Suivant la loi de Gustafson

NUM_THREADS	Global_sim	Avg_step	Global_display
1	0.7152846	2.2092e-05	0.5018012
2	0.8731488	8.390534e-05	0.664849
4	1.768892	0.0002694434	1.548338
8	4.428484	0.0009272818	4.209666
16	17.14542	0.002854734	16.91764
24	39.96224	0.005376924	39.76388

TABLE 4 – Comparaison des temps moyens (secondes) en fonction du nombre de threads

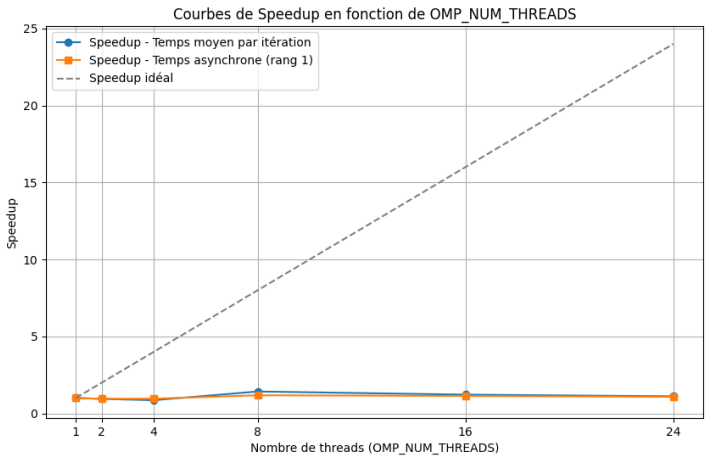

FIGURE 3 – Speed-Up Pour la parallélisation MPI et OpenMP suivant une loi de Gustafson $\sqrt{N} = 20 \cdot n$.

Pour la loi de Gustafson, les valeurs mesurées montrent une augmentation de l'accélération globale avec le nombre de threads (par exemple, passant de 0.7152846 pour 1 thread à 39.96224 pour 24 threads). Toutefois, cette amélioration reste bien inférieure au gain théorique idéal.
Une raison à cela pourrait être l'effet de la partie séquentielle, l'affichage dans ce cas.

5.4.3 BONUS : avec double-buffering

NUM_THREADS	Temps_global_moyen (seconds)	Temps_global_asynchrone (seconds)
1	0.00177044	1.28357
2	0.00188076	1.35756
4	0.00206624	1.34484
8	0.00124538	1.09135
16	0.00145464	1.13354
24	0.00159627	1.19319

TABLE 5 – Comparaison des temps globaux

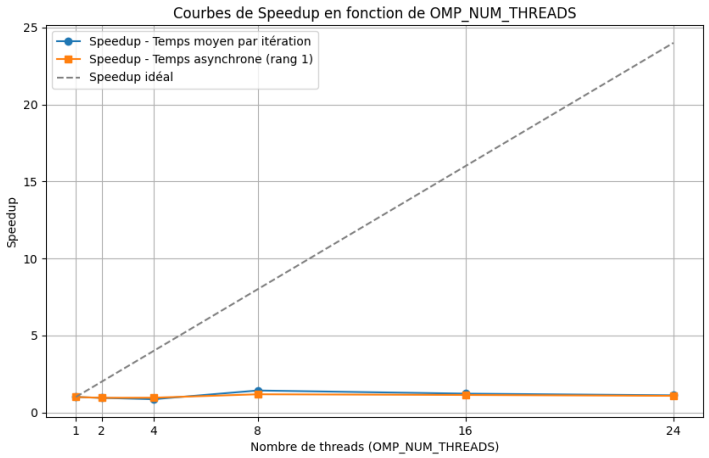

FIGURE 4 – Speed-Up Pour la parallélisation MPI et OpenMP avec double-buffering.

Même avec la technique du double buffering qui permet de rendre asynchrone l'affichage et l'avancement en temps, il semble que le speedup reste aussi médiocre que pour les deux approches précédentes. Deux raisons pouvant expliquer ce constat sont :

— Le surcoût induit par le mécanisme de double buffering, en particulier lors du basculement entre les tampons, qui introduit des délais supplémentaires.

— Une charge de travail par thread insuffisante, amplifiant l'impact des surcoûts de synchronisation et de gestion des threads.

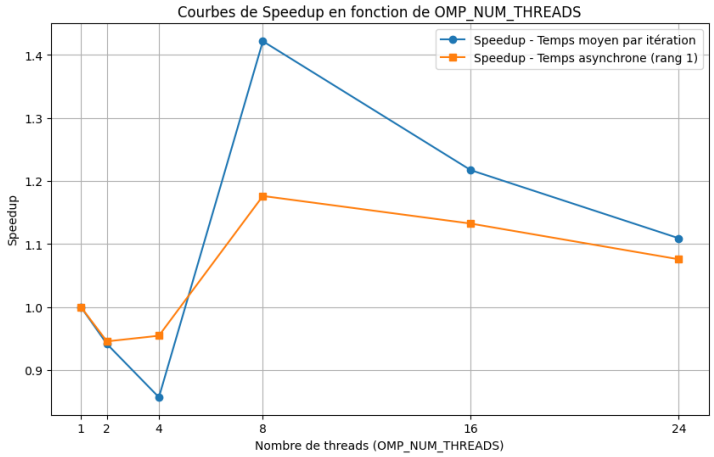

FIGURE 5 – Speed-Up Pour la parallélisation MPI et OpenMP avec double-buffering sans la loi idéal.

La courbe du temps global moyen présente une tendance non linéaire, avec un pic à 4 threads (environ 0.00207 s) suivi d'une nette amélioration à 8 threads (0.00125 s), ce qui suggère qu'une augmentation modérée du nombre de threads permet une meilleure répartition de la charge avant que les surcoûts ne se manifestent aux niveaux supérieurs. Par ailleurs, la courbe du temps global asynchrone, systématiquement plus élevée, atteint un pic à 2 threads (1.35756 s), diminue à 8 threads (1.09135 s) puis remonte légèrement, indiquant que les mécanismes d'asynchronisme sont fortement impactés par les coûts de gestion et de synchronisation.

---

6 Partie 4 : Décomposition de domaine avec recouvrement

6.1 Modifications apportées

Par rapport au code obtenu à l'étape 2, voici les changements effectués pour implémenter la décomposition des domaines.

1. **Découpage de la zone en tranches**
   La zone de simulation (carte de la végétation) est découpée en tranches. Nous avons opté pour une stratégie *One Direction Splitting*, en raison de sa simplicité d'implémentation, avec un découpage horizontal en fonction du nombre de processus. Le processus de rang 0 calcule :
   — Le nombre de lignes de base par processus (`base_rows`) ;
   — Le nombre de lignes supplémentaires (`extra_rows`) à répartir équitablement ;
   — Pour chaque processus, la première ligne (`first_row`) et le nombre de lignes locales (`local_rows`) qui constituent le sous-domaine.
   Chaque processus se voit ainsi attribuer une tranche locale de la zone globale.
2. **Parallélisation avec cellules fantômes**
   Pour permettre des calculs locaux cohérents sur les bords, des cellules fantômes

sont ajoutées en haut et en bas de chaque tranche locale. Avant chaque itération, les processus échangent leurs lignes de bord avec leurs voisins à l'aide de `MPI_Sendrecv` :

— Le processus envoie sa première ligne de données (hors fantôme) au voisin supérieur et reçoit la ligne correspondante pour mettre à jour son bord supérieur.

— De même, il échange sa dernière ligne de données avec le voisin inférieur pour mettre à jour son bord inférieur.

Ce mécanisme garantit que les calculs tiennent compte de l'état des cellules voisines situées dans un autre processus.

6.2 Problèmes rencontrés

Le problème observé avec ce code est que, bien que le découpage de la zone de simulation en tranches soit correctement réalisé pour un nombre donné de processus, seule la sous-section associée à un processus (par exemple, le processus de rang n) évolue réellement. Les autres sections restent affichées en vert, indiquant qu'elles ne sont pas mises à jour comme attendu.

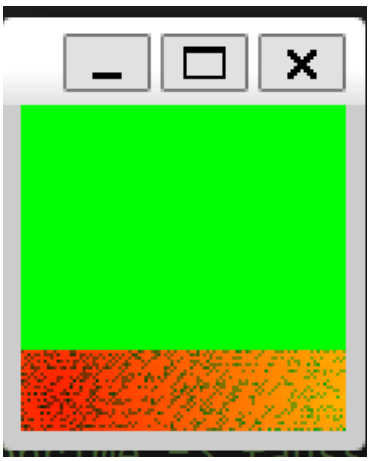

FIGURE 6 – Le problème rencontré.

6.3 Explications

On peut conjecturer que ce comportement est dû à plusieurs facteurs potentiels :

— **Gestion incorrecte des cellules fantômes :**
L'échange des cellules fantômes entre les processus, via `MPI_Sendrecv`, pourrait être mal synchronisé ou mal implémenté, empêchant ainsi la mise à jour correcte des bords de chaque tranche.

— **Erreur dans le découpage de la zone :**
Une mauvaise répartition des lignes entre les processus (calcul incorrect des indices de début et de fin ou des tailles de sous-domaines) peut entraîner le non-traitement de certaines sections de la carte.

— **Problème lors de la collecte des données :**
L'utilisation de `MPI_Gatherv` pour rassembler les sous-domaines pourrait être erronée (mauvais décalages ou tailles), de sorte que seules les données d'un processus sont effectivement intégrées à l'affichage global.

6.4 Résultats et Analyse

Malgré le fait que la simulation ne fonctionne pas parfaitement, nous avons opté pour calculer le speedup de cette approche. En effet, un sous-domaine fonctionne correctement et, en théorie, le programme est parallélisé. Ainsi, si les autres domaines se comportaient de la même manière, tous traiteraient leur charge en un temps identique, compte tenu de la répartition équitable que nous avons mise en place. Par conséquent, les résultats du speedup demeurent représentatifs d'un code pleinement opérationnel.

N	Temps moyen par itération (secondes)	Temps global (secondes)
1	0.00379497	2.93669
2	0.00232788	1.74177
3	0.00196415	1.48089
4	0.00153274	1.16282
5	0.00156447	1.20621
6	0.00137788	1.08967
7	0.00150174	1.21222
8	0.00220971	1.54684

TABLE 6 – Temps moyen par itération et temps global en fonction de N

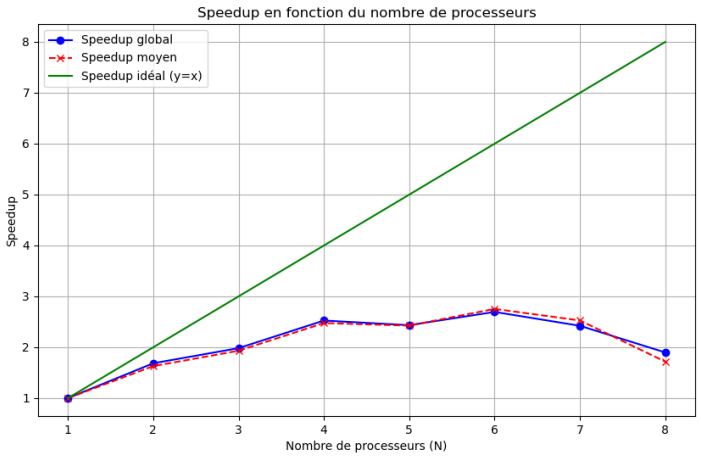

FIGURE 7 – Speed-Up Pour la parallélisation MPI avec décomposition de domaine.

Comparativement aux tentatives antérieures, l'implémentation par décomposition des domaines a permis d'obtenir un speedup nettement supérieur. En effet, le découpage en tranches, associé à l'utilisation efficace des cellules fantômes, assure une répartition équilibrée de la charge de calcul et minimise les échanges de données superflus. Ainsi, chaque processus peut traiter sa sous-section de manière autonome tout en maintenant une synchronisation optimale avec les autres, ce qui se traduit par une réduction significative des temps d'itération et des temps globaux (voir Tableau 6 et Figure 7).

Cette performance accrue s'explique principalement par :

— Une répartition optimisée de la charge de travail, permettant à chaque processus de travailler efficacement sur une tranche bien définie.

— La réduction des échanges inutiles grâce à l'utilisation de cellules fantômes, qui limite les surcoûts de communication entre les processus.

7 Conclusion

Dans ce projet, nous avons exploré et implémenté plusieurs stratégies de parallélisation pour la simulation d'un incendie de forêt, en combinant notamment OpenMP et MPI. Les résultats obtenus montrent que l'approche par décomposition de domaine, associée à l'utilisation de cellules fantômes, améliore significativement le speedup par rapport aux tentatives initiales. Cette méthode a permis une répartition équilibrée de la charge de calcul et une réduction notable des échanges superflus entre processus.

Cependant, plusieurs aspects restent à améliorer. Nous avons notamment constaté des divergences entre la simulation séquentielle et la simulation parallèle, en raison d'un ordre de mise à jour variable et de tirages pseudo-aléatoires non synchronisés. De plus, certaines sections de la simulation ne sont pas mises à jour correctement, comme l'indique l'affichage partiel des sous-domaines.

Pour compléter le projet, nous conjecturons que les améliorations suivantes pourraient être envisagées :

— Mettre en place une synchronisation plus stricte lors des échanges de données (notamment pour l'ordre des mises à jour et l'alignement des graines aléatoires) afin de garantir une équivalence parfaite avec la version séquentielle.

— Réviser notre stratégie de décomposition du domaine.

En conclusion, bien que les résultats actuels soient prometteurs, le projet révèle encore des défis importants en matière de parallélisation et de synchronisation. Des améliorations supplémentaires dans ces domaines pourraient permettre de tirer pleinement parti des architectures parallèles modernes et d'obtenir une simulation d'incendie à la fois robuste et efficace.

Annexe

Nous avons écrit plusieurs scripts en Python pour mesurer les temps d'exécution, calculer les accélérations obtenues en fonction du nombre de threads et générer des courbes illustrant ces résultats. Ces scripts étaient adaptés aux différentes parties du projet. Voici un exemple de deux scripts utilisés pour collecter les temps d'exécution et analyser l'évolution du speedup :

```
import os
import subprocess
import csv
import statistics

n_values = [1, 2, 4, 8, 16, 24]

NUM_RUNS = 5

def parse_time(line):
    """Extracts the time (second last word) from a line."""
    parts = line.split()
    if len(parts) >= 2:
        try:
            return float(parts[-2])  # Time is the second-to-last word before 'seconds'
        except ValueError:
            return None
    return None

with open('simu_update_times.csv', 'w', newline='') as f1, \
     open('display_update_times.csv', 'w', newline='') as f2, \
     open('iteration_times.csv', 'w', newline='') as f3:
```

```
    writer1 = csv.writer(f1)
    writer2 = csv.writer(f2)
    writer3 = csv.writer(f3)

    writer1.writerow(['threads', 'simu_update_time'])
    writer2.writerow(['threads', 'display_update_time'])
    writer3.writerow(['threads', 'iteration_time'])

    for n in n_values:
        simu_times = []
        display_times = []
        iter_times = []

        my_env = os.environ.copy()
        my_env['OMP_NUM_THREADS'] = str(n)

        for _ in range(NUM_RUNS):
            result = subprocess.run(['./simulation'], env=my_env, text=True,
                capture_output=True)
            output = result.stdout
            lines = output.splitlines()

            if len(lines) < 3:
                print(f"Unexpected output for n={n}: {output}")
                continue

            simu_time = parse_time(lines[0])    # simu.update() time
            display_time = parse_time(lines[1]) # displayer->update() time
            iter_time = parse_time(lines[2])    # iteration time

            if simu_time is not None and display_time is not None and iter_time is not
                None:
                simu_times.append(simu_time)
                display_times.append(display_time)
                iter_times.append(iter_time)
            else:
                print(f"Error parsing times for n={n}: {output}")

        if simu_times and display_times and iter_times:
            avg_simu_time = statistics.mean(simu_times)
            avg_display_time = statistics.mean(display_times)
            avg_iter_time = statistics.mean(iter_times)

            writer1.writerow([n, avg_simu_time])
            writer2.writerow([n, avg_display_time])
            writer3.writerow([n, avg_iter_time])
        else:
            print(f"No valid runs completed for n={n}")
```

Listing 1 – Script de mesure des temps

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

simu_df = pd.read_csv('simu_update_times.csv')
display_df = pd.read_csv('display_update_times.csv')
iter_df = pd.read_csv('iteration_times.csv')

t1_simu = simu_df[simu_df['threads'] == 1]['simu_update_time'].values[0]
t1_display = display_df[display_df['threads'] == 1]['display_update_time'].values[0]
t1_iter = iter_df[iter_df['threads'] == 1]['iteration_time'].values[0]

simu_df['speedup_simu'] = t1_simu / simu_df['simu_update_time']
display_df['speedup_display'] = t1_display / display_df['display_update_time']
iter_df['speedup_iter'] = t1_iter / iter_df['iteration_time']

print("Tableau d'acc l ration pour simu.update():")
print(simu_df[['threads', 'simu_update_time', 'speedup_simu']].to_string(index=False))
print("\nTableau d'acc l ration pour displayer->update():")
print(display_df[['threads', 'display_update_time', 'speedup_display']].to_string(index=
    False))
```

```
print("\nTableau d'acc l ration globale (it ration):")
print(iter_df[['threads', 'iteration_time', 'speedup_iter']].to_string(index=False))

plt.figure(figsize=(10, 6))

plt.plot(simu_df['threads'], simu_df['speedup_simu'], marker='o', label='Acc l ration
    simu.update()')
plt.plot(display_df['threads'], display_df['speedup_display'], marker='s', label='
    Acc l ration displayer->update()')
plt.plot(iter_df['threads'], iter_df['speedup_iter'], marker='x', label='Acc l ration
    globale (it ration)')

max_threads = max(simu_df['threads'].max(), display_df['threads'].max(), iter_df['threads
    '].max())
x_ideal = np.arange(1, max_threads + 1)
plt.plot(x_ideal, x_ideal, linestyle='--', color='black', label='Loi id ale (y = x)')

plt.xlabel('Nombre de threads (OMP_NUM_THREADS)')
plt.ylabel('Acc l ration (Speedup)')
plt.title('Acc l ration en fonction du nombre de threads avec loi id ale')
plt.legend()
plt.grid(True)
plt.xticks(simu_df['threads'])

plt.savefig('speedup_plot.png')
plt.show()
```

Listing 2 – Script de calcul et de traçage du speedup