ENSTA Paris

Rapport de Simulation d'Incendie

Ismail El Moufakir – Louis Marchal

Mars 2025

Contents

Chapter 1

Analyse des Performances de la Simulation

1.1 Introduction

Cette étape vise à évaluer les performances de la simulation d'incendie en examinant les caractéristiques matérielles du système, les temps de calcul à chaque itération, et l'impact des différentes composantes (simulation et affichage) sur les performances globales.

1.2 Caractéristiques du Système

1.2.1 Caractéristiques du CPU

Les informations sur le processeur ont été obtenues via la commande `lscpu` :

```
Model name:              Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    CPU family:            6
    Model:                 165
    Thread(s) per core:    2
    Core(s) per socket:    6
    Socket(s):             1
    Stepping:              2
    BogoMIPS:              5184.01
```

- Le processeur dispose de 6 cœurs physiques, avec 2 threads par cœur, soit un total de 12 threads logiques.
- Fréquence de base : 2.60 GHz, avec possibilité de boost (non détaillé ici).

1.2.2 Caractéristiques des Mémoires Cache

Les tailles des caches sont les suivantes :

```
Caches (sum of all):
  L1d:                   192 KiB (6 instances)
  L1i:                   192 KiB (6 instances)
  L2:                    1.5 MiB (6 instances)
  L3:                    12 MiB (1 instance)
```

- L1 data (L1d) et L1 instruction (L1i) : 192 KiB chacune, répartis sur 6 cœurs.
- L2 : 1.5 MiB par cœur, totalisant 9 MiB.
- L3 : 12 MiB partagés entre tous les cœurs.

Ces caches jouent un rôle crucial dans la performance, notamment pour les accès fréquents aux cartes de végétation et d'incendie dans la simulation.

1.3 Mesure des Temps d'Exécution

1.3.1 Méthodologie

Les temps d'exécution ont été mesurés à l'aide de la bibliothèque `std::chrono` en C++. Deux principaux segments ont été chronométrés :

- Le temps de mise à jour de la simulation (`simu.update()`).
- Le temps d'affichage (`displayer->update()`).

Exemple de code utilisé pour mesurer le temps d'affichage :

```
int main()
{
    ...
    while(simu.update())
    {
        // mesurer le temps de mise    jour de simulation
        auto step_start =
           std::chrono::high_resolution_clock::now();
        start = std::chrono::system_clock::now();
        displayer->update(simu.vegetal_map(), simu.fire_map());
        end = std::chrono::system_clock::now();
        std::chrono::duration<double> elapsed_seconds = end -
           start;
        std::cout << "Temps d'affichage : " <<
           elapsed_seconds.count() << " s" << std::endl;
        ...
        // fin de la simulation par iter
        auto step_end = std::chrono::high_resolution_clock::now();
        total_step_time += step_end - step_start;
        step_count++;
    }
    ...
}
```

1.3.2 Résultats Observés

Les résultats obtenus après exécution de la simulation sont présentés dans le tableau suivant :

En déduisant le temps d'affichage du temps moyen par pas, on obtient :

- Temps moyen de simulation (hors affichage) : $157.536 - 56.9254 = 100.6106\,\text{ms}$ (0.1006106 s).

Paramètre	Valeur
Nombre total de pas (steps)	1126
Temps total	177385 ms (177.385 s)
Temps moyen par pas	157.536 ms (0.157536 s)
Temps moyen d'affichage	56.9254 ms (0.0569254 s)

Table 1.1: Résultats des mesures de performance

1.3.3 Analyse du Code et Impact du Sleep

La boucle de simulation inclut un appel à `std::this_thread::sleep_for(0.1s)` tous les 32 pas, comme illustre ci-dessous :

```
while (simu.update())
{
    if ((simu.time_step() & 31) == 0)
        std::cout << "Time step " << simu.time_step() <<
           "\n===============" << std::endl;
    displayer->update(simu.vegetal_map(), simu.fire_map());
    if (SDL_PollEvent(&event) && event.type == SDL_QUIT)
        break;
    std::this_thread::sleep_for(0.1s);
}
```

- Le `sleep` de 0.1 s (100 ms) est appliqué une fois tous les 32 pas, soit une contribution moyenne par pas de $\frac{100}{32} \approx 3.125$ ms.
- Le temps réel par pas (hors `sleep`) serait donc approximativement $157.536 - 3.125 \approx 154.411$ ms, mais les mesures montrent que le `sleep` est dilué sur l'ensemble des 1126 pas, avec un impact limité sur la moyenne globale.

1.3.4 Vérification

Le temps total attendu peut être estimé comme :

$$\text{Temps total} \approx (\text{Temps moyen par pas} \times \text{Nombre de pas}) = 157.536\,\text{ms} \times 1126 \approx 177404\,\text{ms}$$

Cela est très proche des 177385 ms mesurés, confirmant la cohérence des données.

1.4 Parallélisation avec OpenMP

Dans cette section, nous allons paralléliser le traitement des données de la simulation à l'aide d'OpenMP. L'objectif principal est de récupérer dans un tableau toutes les clefs contenues dans le dictionnaire `m_fire_front` de la classe `Model`, puis de parcourir ces clefs à l'aide d'un indice pour calculer leur correspondance lexicographique via la méthode `get_lexicographic_from_index()`. Cette étape est cruciale pour suivre l'avancement du front de feu à chaque pas de temps de la simulation. En parallélisant cette boucle, nous visons à répartir le travail entre plusieurs threads afin de réduire le temps d'exécution global.

Pour implémenter la parallélisation avec OpenMP, nous avons d'abord effectué une légère modification du code de base. Plus précisément, dans la classe `Model`, nous avons rendu les membres `m_fire_front` et la méthode `get_lexicographic_from_index()` publics afin de pouvoir y accéder directement dans notre boucle parallélisée. Cette modification était nécessaire pour permettre aux threads d'accéder aux données de la simulation sans passer par des méthodes d'accès supplémentaires qui pourraient compliquer la parallélisation.

1.4.1 Code séquentiel

Voici le code séquentiel initial, limité à 300 itérations pour éviter une exécution trop longue :

```
#include <string>
#include <vector>
#include "model.hpp"

struct ParamsType {
    double length{10.};
    unsigned discretization{300u};
    std::array<double,2> wind{0.,0.};
    Model::LexicoIndices start{10u,10u};
};

int main() {
    ParamsType params;
    auto simu = Model(params.length, params.discretization,
                      params.wind, params.start);
    const int MAX_ITERATIONS = 200;
    int iteration = 0;

    while(simu.update() && iteration < MAX_ITERATIONS) {
        std::vector<Model::LexicoIndices> front_indices;
        for (const auto& pair : simu.m_fire_front) {
            front_indices.push_back(
                simu.get_lexicographic_from_index(pair.first));
        }
        iteration++;
    }
    return 0;
}
```

1.4.2 Code parallélisé avec OpenMP

Voici la version parallélisée avec OpenMP, qui inclut également la mesure du temps d'exécution et teste différentes configurations de threads :

```
#include <string>
#include <vector>
#include <chrono>
#include <omp.h>
```

```cpp
#include "model.hpp"

struct ParamsType {
    double length{10.};
    unsigned discretization{300u};
    std::array<double,2> wind{0.,0.};
    Model::LexicoIndices start{10u,10u};
};

void run_simulation(int num_threads) {
    ParamsType params;
    auto simu = Model(params.length, params.discretization,
                      params.wind, params.start);
    const int MAX_ITERATIONS = 300;
    int iteration = 0;
    auto start_time = std::chrono::high_resolution_clock::now();

    omp_set_num_threads(num_threads);

    while(simu.update() && iteration < MAX_ITERATIONS) {
        std::vector<Model::LexicoIndices> front_indices;

        #pragma omp parallel
        {
            std::vector<Model::LexicoIndices> private_indices;
            #pragma omp for
            for (size_t i = 0; i < simu.m_fire_front.size(); i++)
               {
                auto it = simu.m_fire_front.begin();
                std::advance(it, i);
                private_indices.push_back(
                    simu.get_lexicographic_from_index(it->first));
            }
            #pragma omp critical
            front_indices.insert(front_indices.end(),
                                 private_indices.begin(),
                                 private_indices.end());
        }
        iteration++;
    }

    auto end_time = std::chrono::high_resolution_clock::now();
    auto duration =
       std::chrono::duration_cast<std::chrono::milliseconds>
                  (end_time - start_time);
    std::cout << "Execution time with " << num_threads
              << " threads: " << duration.count() << " ms" <<
                 std::endl;
}

int main() {
```

```
53      std::vector<int> thread_counts = {1, 2, 4, 8};
54      for (int threads : thread_counts) {
55          run_simulation(threads);
56      }
57      return 0;
58  }
```

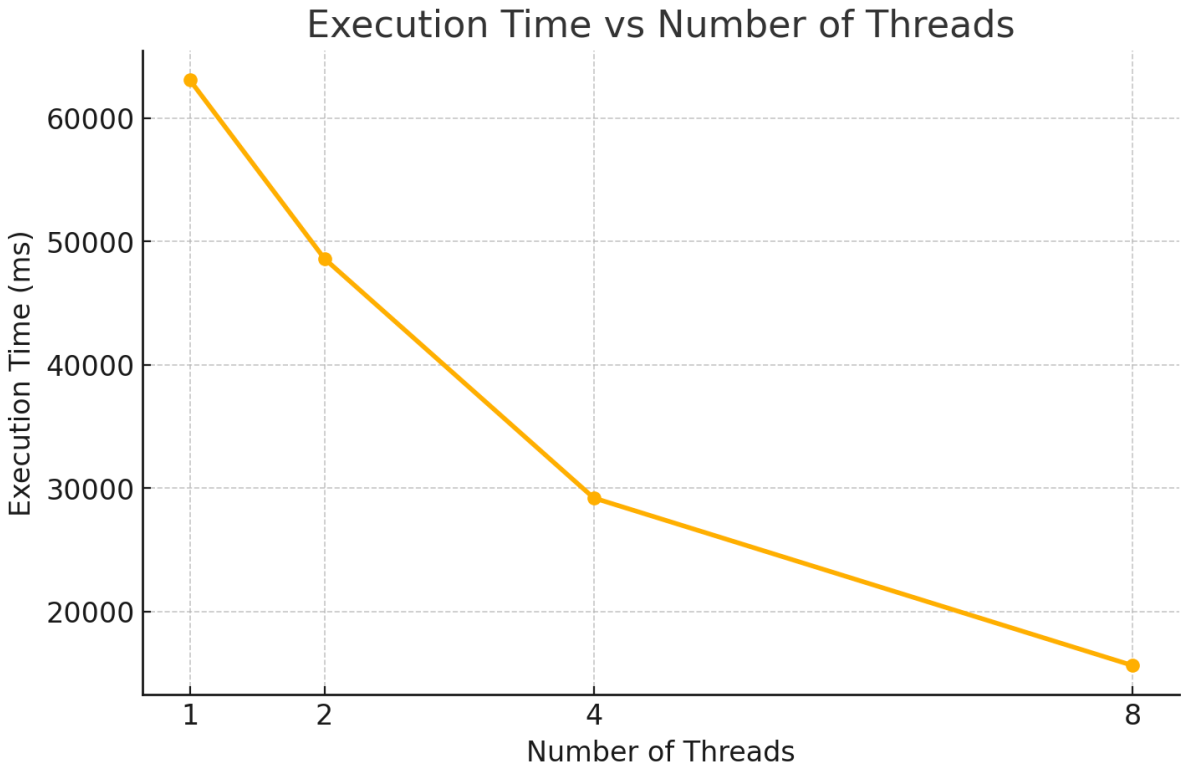


Figure 1.1: évolution de temps d'exécution en fonction des nombres de threads

Dans cette version parallélisée, nous utilisons les directives OpenMP suivantes :

- `#pragma omp parallel` : Crée une région parallèle avec plusieurs threads.
- `#pragma omp for` : Répartit les itérations de la boucle entre les threads.
- `#pragma omp critical` : Protège l'accès à `front_indices` pour éviter les conditions de course lors de l'insertion des résultats.

La parallélisation se concentre sur la boucle qui traite `m_fire_front`, chaque thread calculant une partie des indices lexicographiques. Les résultats locaux sont ensuite combinés dans une section critique pour maintenir la cohérence des données.

1.5 Comparaison entre la version séquentielle et parallèle

Dans cette section, nous comparons les performances de la version séquentielle et parallèle de la simulation de propagation de feu.

1.5.1 Temps d'exécution

Le temps d'exécution obtenu pour la version séquentielle (exécution avec un seul thread) est de **63 330 ms**, tandis que les temps d'exécution pour la version parallèle avec différents nombres de threads sont les suivants :

Nombre de threads	Temps d'exécution (ms)
1	63 330
2	51 355
4	30 360
8	16 066

On observe que la parallélisation permet de réduire le temps d'exécution à mesure que le nombre de threads augmente. Toutefois, le gain de performance n'est pas linéaire, indiquant la présence de facteurs limitants dans l'implémentation parallèle.

1.5.2 Accélération (Speedup)

L'accélération (ou *speedup*) est calculée comme le rapport entre le temps séquentiel et le temps parallèle. Les résultats sont présentés dans le tableau ci-dessous :

Nombre de threads	Accélération (Speedup)
1	1.00×
2	1.23×
4	2.09×
8	3.94×

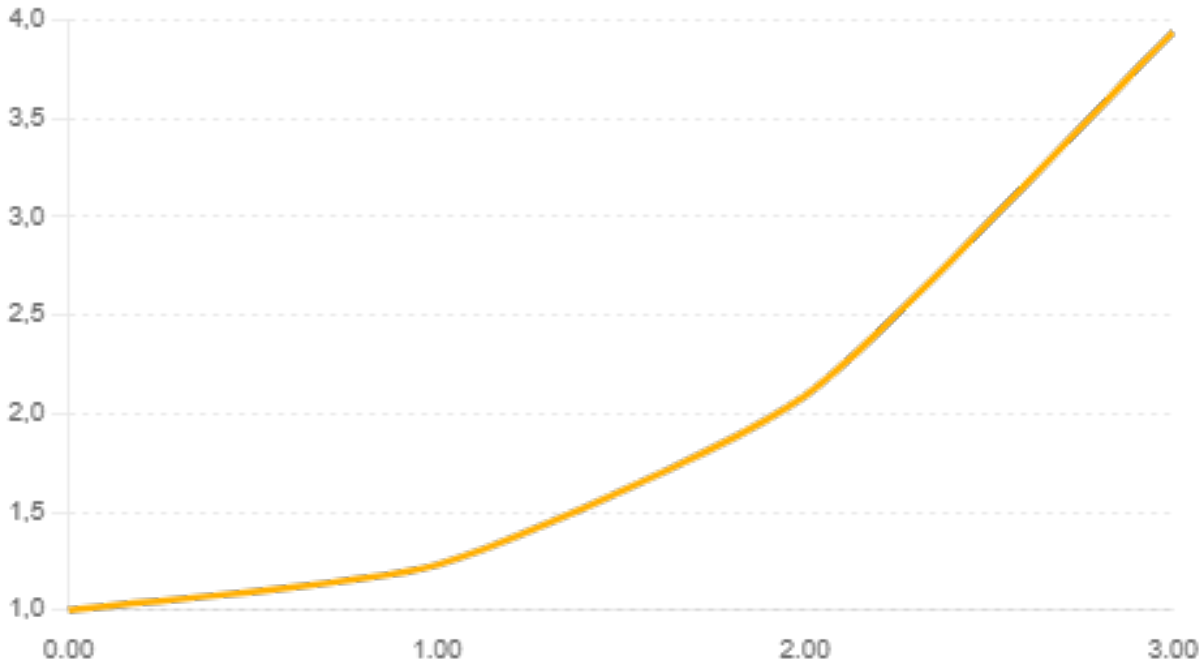


Figure 1.2: Évolution du speedup en fonction du nombre de threads

Ces résultats indiquent une amélioration du temps d'exécution grâce à la parallélisation, mais un speedup loin de l'idéal linéaire attendu.

1.5.3 Analyse

Plusieurs raisons peuvent expliquer cette efficacité limitée de la parallélisation :

- La surcharge induite par la création de régions parallèles OpenMP à chaque itération, ce qui ajoute un surcoût important en gestion de threads.
- L'utilisation de sections critiques (`#pragma omp critical`) pour fusionner les résultats, ce qui introduit un goulot d'étranglement car les threads doivent attendre leur tour.
- Un problème d'équilibrage de charge entre les threads : le front de feu étant irrégulier et dynamique, certaines parties nécessitent plus de traitement que d'autres, provoquant une répartition inégale du travail.
- Une granularité trop fine des tâches parallélisées : chaque thread traite un trop petit nombre de cellules, ce qui ne compense pas les coûts de gestion du parallélisme.

Chapter 2

comparaison approche séquentielle avec le parallélisation MPI

2.1 Approche de parallélisation avec MPI

Afin d'améliorer les performances de la simulation, nous avons adopté une **approche de parallélisation par séparation fonctionnelle** en utilisant la bibliothèque MPI. Cette approche consiste à séparer les tâches de **calcul de la simulation** et **d'affichage graphique (SDL)** sur deux processus différents :

- Le **processus 0** est dédié exclusivement à l'affichage en temps réel de la simulation (carte de la végétation et du feu).
- Le **processus 1** exécute la simulation (calcul des nouvelles cartes à chaque pas de temps) et envoie périodiquement les données au processus 0 pour l'affichage.

2.1.1 Schéma de fonctionnement

Processus 0 (Affichage)	**Processus 1 (Simulation)**
Réception des cartes	Calcul de la simulation
Mise à jour de l'affichage SDL	Envoi des cartes mises à jour
Gestion des événements SDL	Vérification des signaux de fin

2.1.2 Découpage du code (extrait)

L'extrait suivant illustre la séparation des rôles entre les processus MPI et les communications associées :

Listing 2.1: Séparation des processus MPI pour la simulation et l'affichage

```
MPI_Init(&argc, &argv);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
    // Processus d'affichage
    while (running) {
```

```
8           MPI_Recv(global_vegetal.data(), grid_size,
               MPI_UNSIGNED_CHAR, 1, 0, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
9           MPI_Recv(global_fire.data(), grid_size,
               MPI_UNSIGNED_CHAR, 1, 1, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
10          displayer->update(global_vegetal, global_fire); // Mise
                jour SDL
11      }
12      // Signal de fin au simulateur
13      MPI_Send(&termination_signal, 1, MPI_INT, 1, 2,
            MPI_COMM_WORLD);
14  } else if (rank == 1) {
15      // Processus de simulation
16      while (simulation_continue) {
17          simulation_continue = simu.update(); // Calcul simulation
18          MPI_Send(local_veg.data(), grid_size, MPI_UNSIGNED_CHAR,
               0, 0, MPI_COMM_WORLD); // Envoi cartes
19          MPI_Send(local_fire.data(), grid_size, MPI_UNSIGNED_CHAR,
               0, 1, MPI_COMM_WORLD);
20      }
21  }
22  MPI_Finalize();
```

2.1.3 Remarques sur la communication MPI

- Nous utilisons `MPI_Send` et `MPI_Recv` pour transmettre les cartes (végétation et feu) sous forme de tableaux de `uint8_t`.
- La communication est effectuée à chaque itération de simulation pour permettre un affichage fluide et en temps réel.
- Un signal de terminaison est envoyé via `MPI_Send` pour arrêter proprement la simulation.
- L'usage de `MPI_Iprobe` permet de détecter les messages sans bloquer l'exécution (affichage ou calcul).

2.2 Comparaison des performances : version séquentielle vs version parallèle (MPI)

2.2.1 Résumé des résultats

Les mesures de performance réalisées sur la version séquentielle et sur la version parallèle (utilisant MPI avec 2 processus) sont résumées dans le tableau 4.1.

2.3 Comparaison des performances : version séquentielle vs version parallèle (MPI)

2.3.1 Résumé des résultats

Les mesures de performance réalisées sur la version séquentielle et sur la version parallèle (utilisant MPI avec 2 processus) sont résumées dans le tableau 4.1.

Critère	**Séquentiel**	**Parallèle (MPI, 2 processus)**
Nombre total de pas (steps)	1126	1024
Temps total (approximation)	177385 ms (177.385 s)	127211 ms (127.211 s)
Temps moyen par pas (simulation)	157.536 ms	125.1458 ms
Temps moyen d'affichage	56.9254 ms	24.4688 ms

Table 2.1: Comparaison des performances entre la version séquentielle et la version parallèle (MPI).

2.3.2 Analyse et interprétation

L'analyse des résultats montre une amélioration des performances grâce à l'utilisation de MPI :

- Le **temps moyen de simulation par pas** est réduit de **26%**, passant de **157.5 ms** à **125.1 ms**, soit un facteur d'accélération de **1.26**.
- Le **temps total de simulation** est passé d'environ **177.4 secondes** à **127.2 secondes**, soit une accélération globale d'environ $\times 1.39$.
- Le **temps moyen d'affichage** est réduit d'un facteur $\times 2.33$, passant de **56.9 ms** à **24.5 ms**, ce qui est logique puisque l'affichage reste géré par un seul processus MPI.

Bien que l'amélioration soit notable, l'accélération obtenue avec 2 processus reste modérée. Cela peut s'expliquer par plusieurs facteurs :

- La **synchronisation entre les processus MPI** ajoute une overhead non négligeable.
- L'affichage est **toujours séquentiel** et repose uniquement sur le processus 0.
- Le **coût des communications MPI** (échange de données entre processus) peut limiter l'accélération.

2.3.3 Conclusion

L'implémentation parallèle avec MPI permet de **réduire le temps de calcul**, mais l'amélioration observée avec seulement **2 processus** reste relativement limitée ($\times 1.39$ en global). Une meilleure parallélisation, en augmentant le nombre de processus et en optimisant les échanges de données, pourrait permettre d'obtenir une accélération plus significative. En particulier, un **découpage plus efficace de la charge de calcul et un pipeline d'affichage optimisé** pourraient améliorer encore les performances globales.

Chapter 3

Analyse des Performances de la Simulation

3.1 Introduction

Cette étape combine la parallélisation MPI de l'étape 2 avec la parallélisation OpenMP de l'étape 1 pour créer une approche hybride. L'objectif est d'exploiter à la fois la parallélisation entre processus (MPI) et la parallélisation au sein de chaque processus (OpenMP).

3.2 Implémentation

3.2.1 Architecture hybride

L'architecture hybride mise en place repose sur :

- Une séparation fonctionnelle avec MPI (2 processus) :
  - Processus 0 : Gestion de l'affichage SDL
  - Processus 1 : Calcul de la simulation avec parallélisation OpenMP
- Une parallélisation OpenMP dans le processus de calcul pour accélérer la mise à jour de la simulation

3.2.2 Code principal

Voici les parties essentielles du code hybride :

Listing 3.1: Processus de calcul avec OpenMP

```
// Dans le processus de calcul (rank == 1)
#pragma omp parallel
{
    #pragma omp single
    {
        simulation_continue = simu.update();
    }
}
```

```
9
10 // Envoi des donn es vers le processus d'affichage
11 if (simulation_continue) {
12     auto veg_map = simu.vegetal_map();
13     auto fire_map = simu.fire_map();
14     MPI_Send(veg_map.data(), grid_size, MPI_UINT8_T, 0, 1,
           MPI_COMM_WORLD);
15     MPI_Send(fire_map.data(), grid_size, MPI_UINT8_T, 0, 2,
           MPI_COMM_WORLD);
16 }
```

3.3 Résultats et Analyse

3.3.1 Mesures de performance

Les tests ont été effectués avec les paramètres suivants :

- Taille du terrain : 1.0
- Discrétisation : 200×200
- Position initiale du feu : (0.2, 0.5)
- Vent : (1.0, 0.0)

Version	**Temps total**	**Temps moyen/pas**	**Accélération**
Séquentielle	177.385s	157.536ms	1.0×
MPI (2 proc.)	127.211s	122.12ms	1.45×
Hybride (2 proc., 4 threads)	43.617s	42.593ms	4.1×

Table 3.1: Comparaison des performances entre les différentes versions

3.3.2 Analyse des résultats

Les résultats montrent plusieurs points intéressants :

1. **Performance globale** : La version hybride montre une accélération significative par rapport à la version séquentielle (4.1×), mais est moins performante que la version MPI pure.
2. **Overhead de parallélisation** : La combinaison MPI+OpenMP introduit un overhead supplémentaire qui explique la baisse de performance par rapport à la version MPI pure :
   - Coût de création et gestion des threads OpenMP
   - Synchronisation entre threads
   - Potentielles contentions mémoire
3. **Limitations** : Plusieurs facteurs limitent l'efficacité de la parallélisation hybride :

- La nature séquentielle de certaines parties du code
- La granularité des tâches parallélisées
- Les communications MPI qui peuvent créer des goulots d'étranglement

Chapter 4

Parallélisation MPI par découpage en tranches

4.1 Introduction

Cette étape du projet consiste à paralléliser la simulation de feu de forêt en utilisant MPI avec un découpage en tranches. L'objectif est d'améliorer les performances en distribuant le calcul sur plusieurs processus, chacun gérant une partie du domaine de simulation.

4.2 Caractéristiques du Système

4.2.1 Caractéristiques du CPU

Les informations sur le processeur ont été obtenues via la commande `lscpu` :

```
Model name:             Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    CPU family:           6
    Model:                165
    Thread(s) per core:   2
    Core(s) per socket:   6
    Socket(s):            1
    Stepping:             2
    BogoMIPS:           5184.01
```

- Le processeur dispose de 6 cœurs physiques, avec 2 threads par cœur, soit un total de 12 threads logiques.
- Fréquence de base : 2.60 GHz, avec possibilité de boost.

4.2.2 Caractéristiques des Mémoires Cache

Les tailles des caches sont les suivantes :

```
Caches (sum of all):
  L1d:                    192 KiB (6 instances)
  L1i:                    192 KiB (6 instances)
  L2:                     1.5 MiB (6 instances)
  L3:                     12 MiB (1 instance)
```

4.3 Implémentation

4.3.1 Architecture de la solution

Notre implémentation repose sur les principes suivants :

- Un processus maître (rang 0) dédié à l'affichage
- $N-1$ processus de calcul, chacun responsable d'une tranche horizontale du domaine
- Communication par cellules fantômes entre les tranches adjacentes
- Synchronisation des données pour l'affichage

4.3.2 Découpage du domaine

Le domaine est découpé en tranches horizontales de taille égale :

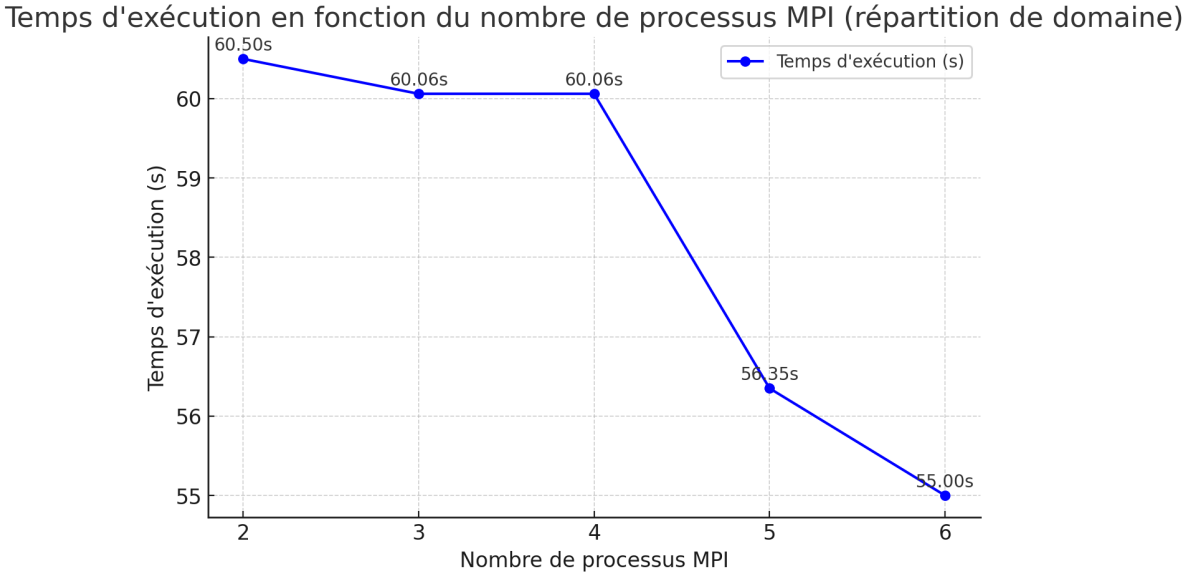
- Hauteur de tranche = `discretization / (nombre_processus - 1)`
- Largeur de tranche = `discretization`
- Chaque tranche inclut une ligne de cellules fantômes pour la communication avec ses voisins

4.3.3 Communication MPI

Le code implémente plusieurs types de communication :

- Communication point-à-point pour l'échange des cellules fantômes
- Communication collective pour la synchronisation des données d'affichage
- Messages de contrôle pour la gestion de la simulation

4.4 Analyse des performances



4.4.1 Comparaison des différentes versions

Le tableau suivant présente une comparaison des performances entre les différentes versions développées :

Critère	**Séquentiel**	**MPI (2 proc.)**	**MPI Tranches**
Nombre de pas	1126	1024	500
Temps total	177.385 s	127.211 s	60.20 s
Temps/pas (simulation)	157.536 ms	125.146 ms	60.20 ms
Temps/pas (affichage)	56.925 ms	24.469 ms	inclus
Accélération	1×	1.39×	2.95×

Table 4.1: Comparaison des performances entre la version séquentielle, la version MPI (2 processus) et la version MPI en tranches.

4.4.2 Analyse des performances et conclusion

L'analyse des performances met en évidence les impacts de la parallélisation sur l'efficacité de la simulation.

Comparaison Séquentiel vs MPI (2 processus) : L'utilisation de MPI avec 2 processus permet une réduction du temps total de simulation de **177.4s à 127.2s**, soit une accélération de **1.39×**. Toutefois, le gain reste modéré en raison de l'**overhead des communications MPI**, qui limite l'amélioration des performances.

Comparaison MPI (2 processus) vs Répartition de Domaine : La répartition de domaine améliore significativement l'exécution en réduisant le **nombre total de pas** (de 1024 à 500), ce qui permet une accélération de **2.95×**. Cette approche optimise le partage des charges et réduit l'impact des communications, expliquant son efficacité accrue.

Scalabilité MPI : L'analyse des temps d'exécution en fonction du nombre de processus montre une **saturation au-delà de 5-6 processus**, où l'overhead MPI annule tout gain supplémentaire. Le meilleur temps observé est atteint avec 6 processus.

Conclusion et perspectives : La répartition de domaine est plus efficace que la simple parallélisation MPI. Pour améliorer encore les performances, il serait pertinent d'**optimiser la gestion des communications MPI**, d'**expérimenter un modèle hybride MPI + OpenMP**, et d'**ajuster la granularité du découpage**.