Simulation de Propagation d'Incendie

Projet de Calcul Parallèle avec MPI

Réalisé par : CEYLON Nathanel et DIALLO Ibrahima

Professeur : Xavier JUVIGNY

M2 ISDS : ISUP — Sorbonne Université

Mars 2025

Introduction

Ce projet vise à modéliser la propagation d'un feu de forêt sur un terrain carré, en prenant en compte l'effet du vent (direction et intensité) ainsi que la densité de végétation. Le travail s'appuie sur un projet réalisé à l'ENS Paris-Saclay par Élise Foulatier [1], initialement développé en Python de manière séquentielle. Notre objectif a été de reprendre ce modèle et de le paralléliser progressivement avec MPI (*Message Passing Interface*).

Le terrain est discrétisé en une grille $N \times N$ dans laquelle deux cartes sont utilisées :
— La carte de végétation, où chaque cellule a une valeur entre 0 et 255 indiquant la densité végétale.
— La carte d'incendie, où chaque cellule indique l'intensité du feu (de 0 à 255).
Chaque cellule peut être :
— Saine : pas de feu, végétation intacte (intensité = 0).
— Brûlante : feu actif ou en extinction (valeur initiale 255, divisée par 2 à chaque itération).
— Brûlée : feu éteint, végétation détruite (végétation = 0).

La propagation du feu repose sur deux probabilités :
— p_1 : probabilité qu'une cellule voisine prenne feu, influencée par le vent et la végétation.
— p_2 : probabilité que le feu commence à s'éteindre sur une cellule brûlante.
Des valeurs pseudo-aléatoires contrôlées (dépendant de la position et du pas de temps) assurent la reproductibilité de la simulation.

Le projet a été mené en trois étapes :

1. Développement du code séquentiel et mesure des temps de calcul.
2. Mise en place de MPI avec séparation entre affichage et calcul.
3. Parallélisation complète du calcul avec découpage de la grille entre les processus.

Pour chaque phase, des tests ont été effectués, et les performances mesurées (temps par itération, total, accélérations, etc.) ont été analysées. Ce rapport retrace la progression du projet, détaille les choix de conception, et présente les résultats observés.

1 Étape 1 : Configuration matérielle et analyse des performances en mode séquentiel

1.1 Configuration matérielle

Les simulations ont été réalisées sur un ordinateur doté des caractéristiques suivantes :
— Processeur : AMD Ryzen 9 7940HS avec Radeon 780M Graphics
— Architecture : 64 bits (x86_64)
— Fréquence maximale : 4.0 GHz
— Nombre de cœurs physiques : 8
— Nombre de threads (logiques) : 16
— Cache L1 : 96 KiB (données) + 128 KiB (instructions)
— Cache L2 : 8192 KiB
— Cache L3 : 16384 KiB
— Système d'exploitation : Windows 64 bits

1.2 Paramètres de simulation

Les simulations en mode séquentiel ont été effectuées avec les paramètres suivants :
— Taille du domaine : 1.0 km × 1.0 km et 10.0 km × 10.0 km
— Discrétisation spatiale : 75 et 100 cellules par direction
— Vecteur du vent : $(1.0, 1.0)$ et $(2.0, 3.0)$
— Position initiale du feu : $(1, 50)$ et $(15, 10)$

1.3 Méthode de mesure des performances

Afin d'évaluer la performance du programme, nous avons mesuré les temps suivants à chaque itération de la simulation :
— Temps de calcul moyen par itération

1. Projet original : `https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/simulation-dela-propagation-dun-feu-de-foret`

— Temps d'affichage moyen par itération
— Temps total de la simulation
Les données ont été extraites automatiquement à l'aide du module `time` en Python.

Temps de calcul (s)	Temps d'affichage (s)	Total (s)
0.00012	0.00003	35.25404

TABLE 1 – Temps moyen observé en mode séquentiel

1.4 Analyse des résultats

L'analyse de ces résultats montre que le temps de calcul représente l'essentiel du coût d'exécution, tandis que l'affichage a un impact négligeable. L'intérêt d'une approche parallèle réside donc dans la répartition efficace des calculs entre plusieurs processus afin de réduire le temps total de simulation.

Ces valeurs serviront de référence de performance pour évaluer les gains obtenus grâce à la parallélisation dans les étapes suivantes.

2 Étape 2 : Mise en place de l'environnement MPI

2.1 Objectif

Cette deuxième étape a pour but de transformer progressivement le code séquentiel en une version parallèle à l'aide de la bibliothèque `mpi4py`, qui implémente l'interface MPI pour Python. L'enjeu principal est de séparer clairement les rôles des différents processus et de coordonner les échanges entre eux pour conserver la cohérence de la simulation.

La mise en place s'est déroulée en plusieurs phases :
— Duplication du code d'origine pour conserver une base séquentielle fonctionnelle.
— Intégration de `mpi4py` dans le code pour initialiser la communication entre processus.
— Répartition des tâches :
  — Le processus de rang `0` est chargé exclusivement de l'affichage.
  — Les autres processus (`rang` > 0) prennent en charge les calculs liés à la propagation du feu.
— Synchronisation des processus à l'aide de `comm.Barrier()` pour maintenir un déroulement cohérent entre affichage et calcul.

2.2 Structure du programme MPI

Une fois le parallélisme introduit, la structure du programme est la suivante :
— Processus 0 :
  — Récupère les sous-cartes de tous les processus de calcul.
  — Reconstruit les cartes complètes pour affichage avec `Pygame`.
  — Enregistre les temps d'exécution dans un fichier CSV pour analyse.
— Processus > 0 :
  — Gèrent l'évolution du feu sur un sous-domaine de la grille.
  — Envoient les données mises à jour au rang 0 pour affichage.

2.3 Protocole de test

Afin d'analyser les performances de cette version MPI, un script automatisé en Python a été utilisé pour tester différentes configurations :
— Taille du terrain (km) : 1, 5, 10
— Discrétisation de la grille : 75 et 100 cellules par direction
— Vecteurs de vent : $(1, 1)$ et $(2, 3)$
— Points de départ du feu : $(5, 5)$, $(15, 10)$ et centre de la grille
— Nombre de processus : 1 (séquentiel), 2 (test de base parallèle)

Pour chaque simulation, les métriques suivantes ont été enregistrées :
— Nombre d'itérations avant extinction du feu
— Temps moyen de calcul par itération
— Temps moyen d'affichage par itération
— Temps total d'exécution

2.4 Comparaison séquentiel vs MPI (2 processus)

L'extrait suivant illustre les résultats pour une configuration fixe : grille 100×100, d'un km de longeur, vent $(2, 3)$, feu initialisé au centre.

Mode	**Temps calcul (s)**	**Temps affichage (s)**	**Temps total (s)**
Séquentiel	0.000	0.000	43.15
MPI (2 processus)	0.001	0.000	50.48

TABLE 2 – Comparaison des temps par itération (séquentiel vs MPI)

Ce premier test en mode parallèle met en évidence plusieurs constats :
— Le temps de calcul diminue légèrement, mais la surcharge induite par la communication MPI est non négligeable avec seulement 2 processus.
— L'affichage, toujours pris en charge par le processus maître, reste constant.
— L'exécution MPI à 2 processus est légèrement plus lente que la version séquentielle, ce qui confirme qu'une telle configuration est trop modeste pour révéler pleinement les avantages du parallélisme.

Cette étape reste cependant cruciale : elle valide l'organisation du code et la communication entre processus, ce qui permettra dans l'étape suivante une véritable parallélisation du calcul par découpage du domaine.

3 Étape 3 : Parallélisation complète de la simulation

3.1 Principe de parallélisation

Dans cette dernière étape, la parallélisation de la simulation a été étendue à l'ensemble du processus d'évolution du feu à l'aide de `MPI`. Le domaine de simulation, structuré sous forme d'une grille $N \times N$, est divisé horizontalement en tranches (sous-domaines), chacune étant affectée à un processus de calcul. Cette stratégie permet de répartir efficacement la charge de travail entre les différents processus.
— Le processus de rang 0 conserve un rôle central, en charge de l'affichage graphique avec `Pygame`, ainsi que de la collecte et de l'enregistrement des performances.
— Les processus de rang supérieur à 0 effectuent les calculs sur leur sous-grille respective. Chaque processus dispose d'une ou plusieurs lignes de cellules fantômes, récupérées de ses voisins, afin de gérer correctement les bords lors des mises à jour.
— Des `MPI_Barrier()` sont utilisées pour synchroniser les étapes clés, notamment la propagation du feu et l'affichage.

3.2 Accélération : définitions et calculs

L'accélération a été évaluée pour mesurer les gains liés à la parallélisation :

$$A_{\text{globale}} = \frac{T_{\text{seq}}}{T_p}$$

où T_{seq} est le temps total de la version séquentielle et T_p celui obtenu avec p processus.

Nb processus	**Temps calcul (s)**	**Temps total (s)**	A_{globale}
1	0.00000	43.20	1.00
2	0.00000	50.55	0.85
4	0.00000	24.70	1.75
6	0.00100	18.29	2.36
8	0.00114	16.74	2.58

TABLE 3 – Accélérations obtenues pour différents nombres de processus (Longueur ; 1km, Discretisation : 100×100, Vent : (2, 3), Feu : Centre de la grille)

3.3 Visualisation des performances

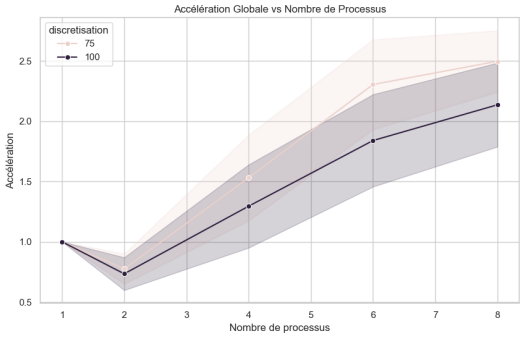

FIGURE 1 – Accélération globale en fonction du nombre de processus

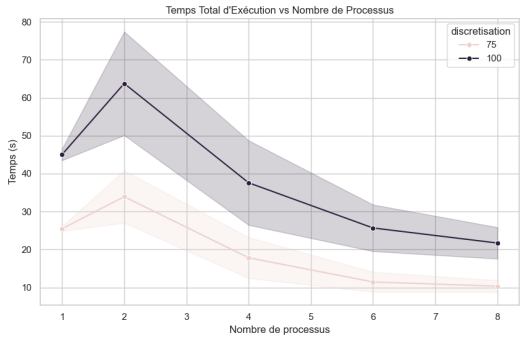

FIGURE 2 – Temps total d'exécution en fonction du nombre de processus

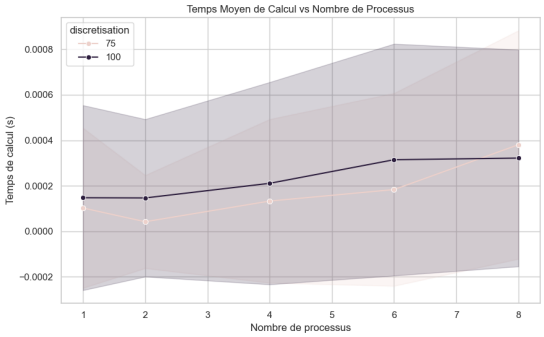

FIGURE 3 – Temps moyen de calcul par itération

3.4 Analyse des Performances

— Accélération Globale : - L'accélération globale augmente significativement jusqu'à 6 ou 8 processus, validant ainsi l'intérêt de la parallélisation. - Une saturation est visible au-delà de 6 processus, principalement due à la surcharge de communication MPI et à la difficulté d'équilibrer les sous-domaines entre les processus. Cette saturation peut être quantifiée en calculant l'efficacité :

$$\text{Efficacité} = \frac{\text{Accélération réelle}}{\text{Nombre de processus}}$$

Une efficacité proche de 1 indique une bonne scalabilité, tandis qu'une valeur inférieure suggère des inefficacités dans la parallélisation.

— Temps Total d'Exécution :

  - Le temps total d'exécution diminue rapidement jusqu'à 4 processus, puis la baisse ralentit progressivement.
  - Une augmentation anormale est observée pour 2 processus, probablement due à des coûts de communication supplémentaires ou à un déséquilibre initial des charges de travail.
  - Après 6 processus, le temps total converge vers une valeur similaire pour les deux niveaux de discrétisation (75 et 100), indiquant que la taille du problème influence moins les performances globales au-delà de ce seuil.
- Temps Moyen de Calcul par Itération :
  - Le temps moyen de calcul par itération augmente légèrement avec le nombre de processus, surtout pour la discrétisation 100. Cela peut être attribué à la fragmentation du travail lorsque chaque processus traite une portion plus petite de la grille.
  - Pour la discrétisation 75, le temps de calcul reste relativement stable, montrant une meilleure résistance à la fragmentation. Cela suggère que des grilles plus petites sont mieux adaptées à la parallélisation.

3.5 Pistes d'amélioration

Pour améliorer davantage les performances du système, plusieurs optimisations peuvent être envisagées, en ciblant à la fois les aspects algorithmiques, les communications inter-processus, et l'utilisation des ressources matérielles :

- Optimisation de l'affichage : L'affichage Pygame constitue un goulot d'étranglement majeur, car il n'est pas parallélisé et reste constant quel que soit le nombre de processus. Pour atténuer cet impact :
  - Désactiver l'affichage lors des tests de performance pour mesurer précisément les gains de calcul.
  - Utiliser des techniques de visualisation distribuée, où chaque processus gère une portion de l'affichage.
  - Réduire la fréquence de rafraîchissement de l'affichage (par exemple, afficher une image toutes les n itérations au lieu de chaque itération).
- Équilibrage des charges entre processus : Une répartition inégale des sous-domaines peut entraîner des déséquilibres de charge, avec certains processus terminant leur travail plus tôt que d'autres. Pour y remédier :
  - Implémenter un découpage dynamique de la grille, où chaque processus reçoit une portion adaptée à sa charge actuelle.
  - Étudier des stratégies de partitionnement plus avancées, telles que l'utilisation d'algorithmes de graphes pour minimiser les communications entre sous-domaines adjacents.
- Approche hybride MPI + OpenMP : Une approche hybride combinant MPI (pour la communication inter-nœuds) et OpenMP (pour l'exploitation intra-nœud via multi-threading) pourrait maximiser l'utilisation des ressources matérielles :
  - Chaque processus MPI pourrait tirer parti des threads OpenMP pour accélérer les calculs locaux.
  - Cette approche serait particulièrement efficace sur des architectures modernes avec plusieurs cœurs par nœud.

3.6 Conclusion

Cette analyse met en lumière les bénéfices significatifs de la parallélisation pour le calcul intensif, tout en soulignant ses limites inhérentes liées aux coûts de communication et à l'affichage non parallélisé. Bien que l'accélération globale soit notable jusqu'à 6 ou 8 processus, elle se heurte à des contraintes pratiques qui limitent son efficacité au-delà de ce seuil.

Pour dépasser ces limitations, une optimisation ciblée des deux principaux facteurs critiques — les communications MPI et l'affichage Pygame — s'avère indispensable. De plus, l'exploration d'une approche hybride MPI + OpenMP pourrait offrir des gains supplémentaires en exploitant pleinement les architectures modernes. Enfin, une mesure rigoureuse des performances permettrait d'affiner les stratégies d'optimisation et d'identifier les zones prioritaires d'amélioration.

En conclusion, cette étude valide l'intérêt de la parallélisation pour les simulations complexes, tout en ouvrant des perspectives concrètes pour améliorer encore davantage les performances. Ces optimisations pourraient non seulement augmenter l'efficacité du système, mais aussi renforcer sa scalabilité pour des problèmes de taille croissante.