Projet : Parallélisation d'une simulation de feu de forêt

Cédric DADA
Nicolas RINCON

cedric-darel.dada@ensta.fr
nicolas.rincon@ensta.fr

École Nationale Supérieure de Techniques Avancées - ENSTA
Systèmes parallèles et distribués - OS02
Palaiseau, France

Année académique 2024-2025

Table des matières

Remarque préalable

Ce travail a été réalisé sur la base des résultats obtenus à partir des exécutions disponibles sur le dépôt GitHub suivant : https://github.com/CedricDada/Cours_Ensta_2025 ou https://github.com/Nrinconv/OS02_ENSTA.

1 Introduction

Ce projet a pour objectif de simuler la propagation d'un incendie de forêt en tenant compte de la densité de la végétation et des conditions de vent. La simulation repose sur un modèle probabiliste qui détermine la propagation du feu à partir de deux probabilités principales : la probabilité qu'un feu se propage à une cellule voisine et la probabilité que le feu s'éteigne progressivement.

Le programme est d'abord implémenté en mode séquentiel, puis il est parallélisé progressivement en plusieurs étapes afin d'améliorer ses performances. La parallélisation est réalisée en utilisant OpenMP pour l'exécution sur un seul nœud multi-cœurs, puis en utilisant MPI pour répartir la charge sur plusieurs processus, et enfin en combinant les deux approches.

1.1 Les étapes de la parallélisation

1.1.1 Étape 1 : Parallélisation avec OpenMP

Dans cette première phase, l'évolution de l'incendie est parallélisée à l'aide d'OpenMP. L'objectif est d'exécuter les calculs en parallèle sur plusieurs threads d'un même processeur en répartissant les cellules enflammées entre les threads.

1.1.2 Étape 2 : Parallélisation avec MPI

Dans cette deuxième étape, l'affichage et le calcul de la simulation sont séparés. Un processus MPI est dédié à l'affichage, tandis qu'un autre processus est responsable des calculs. Cette approche permet une exécution distribuée sur plusieurs machines.

1.1.3 Étape 3 : Combinaison de MPI et OpenMP

Dans cette phase, on combine les méthodes précédentes : MPI est utilisé pour répartir le travail entre plusieurs processus, et chaque processus utilise OpenMP pour paralléliser les calculs sur plusieurs cœurs. Cela permet d'exploiter pleinement les architectures hybrides.

1.1.4 Étape 4 : Parallélisation complète avec MPI

La dernière étape consiste à paralléliser complètement l'exécution en utilisant uniquement MPI. Le domaine de simulation est divisé entre plusieurs processus MPI, chaque processus traitant une sous-région du terrain. Des cellules fantômes sont utilisées pour synchroniser les zones adjacentes. Un processus reste responsable de l'affichage, tandis que les autres effectuent les calculs en parallèle.

2 Étape 0 : Programme séquentiel

Avant d'ajouter la parallélisation, une première version du programme est développée en mode séquentiel. Cette version exécute la simulation sur un seul cœur, sans utiliser OpenMP ni MPI.

Le programme est divisé en plusieurs fichiers, chacun ayant un rôle spécifique.

2.1 Fichiers principaux

2.1.1 `simulation.cpp` : contrôle principal de la simulation

Ce fichier contient la boucle principale qui fait avancer la simulation. Il réalise les actions suivantes :

— Initialise les paramètres (taille de la grille, probabilités, conditions initiales).

— Met à jour le modèle à chaque itération.

— Mesure le temps d'exécution pour analyser les performances.

— Affiche l'état actuel avec les fonctions du module `display`.

2.1.2 `model.cpp` : propagation du feu

Ce fichier contient les règles de propagation du feu :

— Applique les probabilités pour décider si une cellule prend feu ou s'éteint.

— Met à jour les cellules en fonction de ces règles.

— Enregistre l'état du système toutes les 100 itérations pour pouvoir analyser les résultats.

2.1.3 `display.cpp` : affichage graphique

Ce fichier utilise la bibliothèque SDL pour afficher la simulation :

— Initialise la fenêtre de rendu.

— Affiche la grille avec des couleurs représentant les différents états des cellules.

— Met à jour l'affichage à chaque itération.

— Gère les interactions avec l'utilisateur (fermeture de la fenêtre, pause).

2.2 Problèmes de la version séquentielle

Dans cette version, tous les calculs sont faits sur un seul cœur. Cela pose des problèmes de performance :

— Temps d'exécution long : quand la grille est grande, la simulation prend beaucoup de temps.

— Utilisation inefficace du processeur : seul un cœur est utilisé, alors que plusieurs sont disponibles.

— Affichage lent : quand les calculs sont trop longs, l'affichage n'est pas fluide.

Ces limites montrent pourquoi la parallélisation est nécessaire.

3 Étape 1 : Parallélisation avec OpenMP

La première étape de parallélisation consiste à utiliser OpenMP pour accélérer les calculs de propagation du feu et l'affichage de la simulation. Cela permet d'exploiter les architectures multi-cœurs d'un même processeur.

3.1 Parallélisation du calcul de la propagation du feu

Dans la version séquentielle, la mise à jour du feu était effectuée en parcourant la grille cellule par cellule dans le fichier `model.cpp`. Cette opération a été parallélisée en utilisant OpenMP avec la directive `#pragma omp parallel for`, ce qui permet de répartir les calculs entre plusieurs threads. Un double buffer a été introduit pour éviter les conflits d'accès aux données partagées. (modification dans `model.cpp`)

```cpp
// Avant (séquentiel) dans model.cpp
for (int i = 0; i < m_geometry; ++i) {
    for (int j = 0; j < m_geometry; ++j) {
        // Calcul de la propagation du feu
    }
}

// Après (parallélisé avec OpenMP) dans model.cpp
#pragma omp parallel for schedule(static)
for (int i = 0; i < static_cast<int>(m_geometry); ++i) {
    for (int j = 0; j < static_cast<int>(m_geometry); ++j) {
        std::size_t key = i * m_geometry + j;
        if (m_fire_map[key] > 0) {
            // Mise à jour du feu en fonction des cellules voisines
        }
    }
}
```

3.2 Pourquoi utiliser `schedule(static)` ?

Dans OpenMP, `schedule` contrôle comment les itérations de la boucle sont réparties entre les threads. Plusieurs stratégies existent :

- **static** : chaque thread reçoit un bloc fixe d'itérations à traiter.
- **dynamic** : les tâches sont distribuées dynamiquement, un thread prend une nouvelle tâche dès qu'il a terminé la précédente.
- **guided** : comme `dynamic`, mais avec des tailles de blocs qui diminuent progressivement.

Nous avons choisi `schedule(static)` pour plusieurs raisons :

- Uniformité du coût de calcul : chaque cellule a un coût de traitement similaire, donc une répartition fixe est efficace.
- Réduction des surcharges : contrairement à `dynamic`, la planification statique ne nécessite pas de synchronisation entre les threads à chaque nouvelle tâche.
- Meilleure exploitation du cache : les données de la grille sont stockées en mémoire contiguë, et une répartition fixe permet à chaque thread d'accéder à des régions proches en mémoire, ce qui améliore l'efficacité du cache.

Si la charge de travail par cellule avait été plus variable, une planification dynamique (`schedule(dynamic)`) aurait pu être plus adaptée. Toutefois, dans ce cas précis, la répartition statique est plus performante.

3.3 Optimisation de l'affichage

Dans la version séquentielle, un seul thread était responsable de l'affichage de toute la grille dans `display.cpp`. Cela entraînait des blocages lorsque les calculs prenaient trop de temps. Pour éviter cela, chaque processus de calcul met maintenant à jour sa propre portion de la grille grâce à la nouvelle fonction `update_region()`. (modification dans `display.cpp`)

```cpp
// Nouvelle fonction permettant d'afficher seulement une région spécifique (display.cpp)
void Displayer::update_region(int x, int y, int width, int height,
                              std::vector<std::uint8_t> const & vegetation_sub_map,
                              std::vector<std::uint8_t> const & fire_sub_map)
{
    std::lock_guard<std::mutex> lock(m_display_mutex);
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            int global_y = y + i;
            SDL_SetRenderDrawColor(m_pt_renderer,
                                   fire_sub_map[j + width * i],
```

```
                                    vegetation_sub_map[j + width * i],
                                    0, 255);
            SDL_RenderDrawPoint(m_pt_renderer, x + j, m_window_height - global_y - 1);
        }
    }
}
```

3.4 Gestion de la concurrence avec des mutex

Puisque plusieurs threads peuvent essayer d'accéder en même temps au moteur de rendu SDL, l'utilisation d'un mutex (`std::lock_guard<std::mutex>`) a été ajoutée dans `display.cpp` pour éviter des problèmes de synchronisation. (modification dans `display.cpp`)

```
// Protection des accès concurrents à l'affichage (display.cpp)
void Displayer::clear_renderer()
{
    std::lock_guard<std::mutex> lock(m_display_mutex);
    SDL_SetRenderDrawColor(m_pt_renderer, 0, 0, 0, 255);
    SDL_RenderClear(m_pt_renderer);
}
```

3.5 Conclusion de l'implémentation

Grâce à ces améliorations :

- Le calcul de la propagation du feu (`model.cpp`) est réparti entre plusieurs threads, réduisant ainsi le temps d'exécution.
- L'affichage (`display.cpp`) est optimisé en mettant à jour uniquement certaines parties de la grille.
- L'utilisation de mutex dans `display.cpp` empêche les conflits d'accès aux ressources partagées.
- La boucle principale de `simulation.cpp` intègre maintenant les versions parallélisées des modules.

Cette version reste limitée à l'utilisation d'un seul nœud avec plusieurs cœurs.

4 Étape 2 : Séparation du calcul et de l'affichage avec MPI

Dans cette étape, nous avons utilisé MPI pour séparer la simulation en deux processus distincts :

- **Le processus maître (rank 0)** : responsable de l'affichage.
- **Le processus esclave (rank 1)** : effectue les calculs de propagation du feu et envoie les données mises à jour au maître.

Cette séparation permet d'améliorer la fluidité de l'affichage et d'éviter que les calculs ralentissent la mise à jour visuelle.

4.1 Organisation des processus dans `simulation.cpp`

Dans la version séquentielle, un seul processus réalisait à la fois les calculs et l'affichage. Maintenant, grâce à MPI, chaque processus a un rôle spécifique :

```
// Processus maître (rank 0) : affichage
if (rank == 0) {
    auto displayer = Displayer::init_instance(params.discretization, params.discretization);
    while (true) {
        MPI_Recv(vegetation.data(), vegetation.size(), MPI_UINT8_T, 1, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        MPI_Recv(fire.data(), fire.size(), MPI_UINT8_T, 1, 1, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        displayer->update(vegetation, fire);
```

```
    }
}

// Processus esclave (rank 1) : calculs
else if (rank == 1) {
    Model simu(params.length, params.discretization, params.wind, params.start);
    while (simu.update()) {
        MPI_Isend(simu.vegetal_map().data(), vegetation.size(), MPI_UINT8_T, 0, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Isend(simu.fire_map().data(), fire.size(), MPI_UINT8_T, 0, 1, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
}
```

4.2 Choix des communications MPI

L'un des aspects clés de cette implémentation est la gestion efficace de la communication entre les processus. Nous avons utilisé plusieurs méthodes MPI adaptées aux besoins de la simulation :

4.2.1 `MPI_Irecv` et `MPI_Waitall` pour l'affichage asynchrone

Le processus maître (`rank 0`) ne doit pas bloquer inutilement l'affichage en attendant les données du processus esclave. Pour éviter cela, nous utilisons des réceptions asynchrones avec `MPI_Irecv` :

```
// Réception asynchrone des données envoyées par l'esclave
MPI_Irecv(vegetation.data(), vegetation.size(), MPI_UINT8_T, 1, 0, MPI_COMM_WORLD, &reqs[0])
    ;
MPI_Irecv(fire.data(), fire.size(), MPI_UINT8_T, 1, 1, MPI_COMM_WORLD, &reqs[1]);

// Attente de la réception complète avant mise à jour de l'affichage
MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
displayer->update(vegetation, fire);
```

L'utilisation de `MPI_Irecv` permet au processus maître de continuer à exécuter d'autres tâches pendant que les données arrivent.

4.2.2 `MPI_Isend` pour l'envoi asynchrone des données

De même, pour éviter que le processus esclave ne soit bloqué à chaque envoi, nous utilisons `MPI_Isend`, qui permet d'envoyer les tableaux de simulation sans arrêter immédiatement l'exécution :

```
MPI_Isend(vegetation.data(), vegetation.size(), MPI_UINT8_T, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
MPI_Isend(fire.data(), fire.size(), MPI_UINT8_T, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Cela permet d'optimiser la communication et d'éviter une surcharge inutile sur le processus esclave.

4.2.3 `MPI_Iprobe` pour détecter la fin de la simulation

Le processus maître doit pouvoir détecter quand la simulation est terminée. Pour cela, nous utilisons `MPI_Iprobe`, qui vérifie s'il existe un message en attente sans bloquer le programme :

```
int flag = 0;
MPI_Status status;
MPI_Iprobe(1, 2, MPI_COMM_WORLD, &flag, &status);
if (flag) {
    int term;
    MPI_Recv(&term, 1, MPI_INT, 1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    keep_running = false;
}
```

Si un message est disponible, le maître le récupère avec `MPI_Recv`, ce qui lui permet de quitter proprement la boucle d'affichage.

4.2.4 `MPI_Reduce` pour mesurer les performances

Pour analyser l'impact des communications, nous avons ajouté un suivi du temps d'exécution à chaque itération :

```
double iter_start = MPI_Wtime();
...
double iter_end = MPI_Wtime();
double local_iter_time = iter_end - iter_start;

// Récupération du temps maximal de l'itération parmi tous les processus
double global_iter_time = 0.0;
MPI_Reduce(&local_iter_time, &global_iter_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

Cette opération garantit que nous mesurons le temps maximum pris par un processus à chaque itération, ce qui est utile pour identifier d'éventuels goulots d'étranglement.

4.3 Conclusion de l'implémentation

Grâce à cette nouvelle architecture :

- L'affichage et le calcul sont désormais exécutés en parallèle, ce qui évite les blocages.
- Les communications sont optimisées grâce à l'utilisation de messages asynchrones (`MPI_Irecv`, `MPI_Isend`).
- Le programme peut détecter automatiquement la fin de la simulation sans interruption grâce à `MPI_Iprobe`.
- L'impact des communications est suivi en temps réel avec `MPI_Reduce`.

5 Étape 3 : MPI + OpenMP

Dans cette étape, nous combinons MPI et OpenMP afin d'exploiter au mieux les architectures multi-nœuds et multi-cœurs :

- **MPI** est utilisé pour répartir la simulation en plusieurs sous-domaines entre différents processus.
- **OpenMP** est intégré à l'intérieur de chaque processus MPI pour paralléliser les calculs sur plusieurs cœurs.

Cette hybridation permet d'optimiser la répartition de la charge de travail et de limiter les échanges de données entre processus.

5.1 Ajout d'OpenMP dans les processus MPI (`simulation.cpp`)

L'unique modification dans `simulation.cpp` est l'inclusion du fichier d'en-tête OpenMP :

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

Cela permet aux processus MPI d'exploiter OpenMP pour exécuter leurs calculs en parallèle. Cependant, la logique d'initialisation MPI et la gestion des communications restent inchangées par rapport à l'étape 2.

5.2 Parallélisation interne des calculs

L'ajout d'OpenMP a été réalisé dans la mise à jour du modèle de propagation du feu. Chaque processus MPI gère un sous-domaine, et OpenMP est utilisé pour traiter ce sous-domaine en parallèle. (`model.cpp`)

```
// Mise à jour parallèle de la propagation du feu
#pragma omp parallel for schedule(static)
for (int i = 0; i < static_cast<int>(m_geometry); ++i) {
    for (int j = 0; j < static_cast<int>(m_geometry); ++j) {
        std::size_t key = i * m_geometry + j;
        if (m_fire_map[key] > 0) {
            // Mise à jour de l'état du feu en parallèle
```

```
8           }
9       }
10  }
```

Cette modification permet :

— Une accélération du traitement au sein de chaque processus MPI.

— Une charge de travail plus équilibrée entre les cœurs d'un même nœud.

— Une réduction du nombre de processus MPI nécessaires, limitant ainsi la surcharge de communication.

5.3 Optimisation des communications MPI

Une conséquence directe de l'introduction d'OpenMP est la diminution du nombre total de processus MPI. Chaque processus est maintenant capable d'effectuer plus de travail indépendamment, ce qui réduit le besoin de communication inter-processus.

L'envoi des cartes mises à jour reste basé sur des transmissions asynchrones, mais il est maintenant effectué par un plus petit nombre de processus MPI :

```
1 // Réduction du nombre de messages MPI grâce à OpenMP
2 MPI_Isend(vegetation.data(), vegetation.size(), MPI_UINT8_T, 0, 0, MPI_COMM_WORLD, &reqs[0])
    ;
3 MPI_Isend(fire.data(), fire.size(), MPI_UINT8_T, 0, 1, MPI_COMM_WORLD, &reqs[1]);
4 MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
```

Avec cette optimisation :

— Moins de processus MPI envoient des messages, réduisant la latence des communications.

— Les échanges entre processus sont moins fréquents, car chaque processus traite plus de données localement.

5.4 Conclusion de l'implémentation

L'introduction d'OpenMP dans les processus MPI permet plusieurs améliorations :

— Réduction du nombre de processus MPI : moins de communication inter-processus, donc une exécution plus efficace.

— Meilleure exploitation des ressources CPU : chaque processus MPI utilise plusieurs cœurs grâce à OpenMP.

— Amélioration des performances : calculs plus rapides et communication optimisée.