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

Table des figures

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.

Données des différentes simulations Pour plus de transparence, nous avons fourni dans chaque dossier src, un fichier `results.txt` qui contient les résultats des différentes simulations que nous avons réalisé et qui nous ont permis de tracer les différentes courbes qui apparaissent dans ce rapport.

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.

2 Architecture matérielle

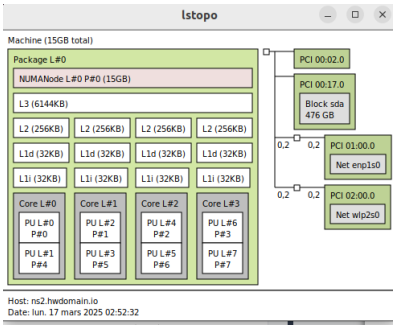


FIGURE 1 – Résultat de la commande lstopo : Nous pouvons visualiser les tailles des caches

FIGURE 2 – Résultat de la commande lscpu

2.1 Informations clés de l'ordinateur

Contexte d'exécution

— **Système d'exploitation** : Ubuntu installé en dual boot (natif).
— **Runtime** : Exécution native sur le matériel, sans virtualisation intermédiaire.
— **Avantages du natif** : Meilleure performance grâce à l'absence de surcharge liée à une couche de virtualisation.

Architecture et processeur

— **Architecture** : x86_64 (64 bits).
— **Modèle du processeur** : Intel(R) Core(TM) i5-10210U.
— **Fréquence** :
  — Base : 1.60 GHz.
  — Turbo Boost : 4.20 GHz.
— **Cœurs et threads** : 4 cœurs physiques, 8 threads (Hyper-Threading).
— **Importance** : Permet une parallélisation efficace avec jusqu'à 8 threads.

Cache

— **L1d** : 128 KiB (4 instances).
— **L1i** : 128 KiB (4 instances).
— **L2** : 1 MiB (4 instances).
— **L3** : 6 MiB (1 instance).
— **Importance** : Réduit la latence d'accès à la mémoire, crucial pour les simulations intensives en calcul.

3 Les étapes de la parallélisation

3.1 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.

3.2 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.

3.3 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.

3.4 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.

4 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.

4.1 Fichiers principaux

4.1.1 `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`.

4.1.2 `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.

4.1.3 `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).

4.2 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.

4.3 Simulation

Dans le but d'obtenir des temps de simulation appréciables (de l'ordre de 0.1 s), nous avons du effectuer les simulations sur des grilles très grandes (n = 1500 et 2000).

Paramètres	Temps Mise à Jour (s)	Temps Affichage (s)	Temps Total (s)
$l = 400, n = 1500, w = [20.0, 12.0], s = [12, 15]$	0.129754	0.167813	0.297567
$l = 400, n = 2000, w = [20.0, 12.0], s = [12, 15]$	0.124577	0.286312	0.410889

TABLE 1 – Résultats de l'exécution du code original

5 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.

5.1 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`)

```
// 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
        }
    }
}
```

5.2 Comment s'assurer que nous obtenons la même simulation ?

Nous avons défini une fonction log_grids qui permet d'écrire dans un fichier simulation$_log.txtlecontenudesdifférentesgrilles(fe$

```
//Dans le fichier Model.cpp

    void Model::log_grids(std::size_t step) const {
    std::ofstream file("simulation_log.txt", std::ios::app);
    file << "Step " << step << " - Fire Map:\n";
    for (std::size_t i = 0; i < m_geometry; ++i) {
        for (std::size_t j = 0; j < m_geometry; ++j) {
            file << std::setw(4) << static_cast<int>(m_fire_map[i * m_geometry + j]);
        }
        file << "\n";
    }
    file << "Vegetation Map:\n";
    for (std::size_t i = 0; i < m_geometry; ++i) {
        for (std::size_t j = 0; j < m_geometry; ++j) {
            file << std::setw(4) << static_cast<int>(m_vegetation_map[i * m_geometry + j
    ]);
        }
        file << "\n";
    }
    file << "\n";
}
```

Cette fonction peut alors être appelée dans la fonction update de la manière suivante :

```
    bool Model::update() {
        //...
        if (m_time_step == 1100) {
            log_grids(m_time_step);
        }
    }
```

5.3 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.

5.4 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.

— La planification statique nous garantit un bon **équilibrage des charges** entre les différents threads.

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

5.5 Résultats :

Speedup Global en fonction du nombre de threads (Partie 1)

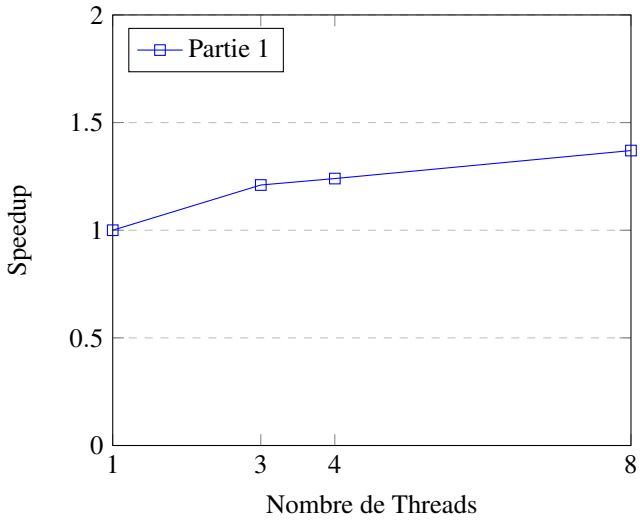


TABLE 2 – Speedup Global en fonction du nombre de threads (Partie 1)

Nombre de Threads	Temps Calcul (s)	Temps Affichage (s)	Temps Total (s)	Speedup
1 (Séquentiel)	0.129754	0.167813	0.297567	1.00
3	0.0256197	0.21956	0.24518	1.21
4	0.0269534	0.212806	0.23976	1.24
8	0.0247988	0.191629	0.21643	1.37

5.6 Interprétation des résultats

— Pour 3 threads : Speedup = 1.21

Interprétation : L'utilisation de 3 threads permet une accélération de 21 % par rapport à la version séquentielle. Cela montre que la parallélisation OpenMP est efficace, mais l'accélération est limitée par le temps d'affichage, qui ne peut pas être parallélisé.

— Pour 4 threads : Speedup = 1.24

— Interprétation : L'ajout d'un thread supplémentaire améliore légèrement le speedup à 24 %. Cela indique que la parallélisation commence à atteindre ses limites en termes de scalabilité.

— Pour 8 threads : Speedup = 1.37

Interprétation : Avec 8 threads, le speedup atteint 37 %. Cela montre que l'augmentation du nombre de threads permet une meilleure exploitation des ressources, mais l'accélération reste modérée en raison du temps d'affichage séquentiel.

6 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.

6.1 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);
    }
}
```

6.2 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 :

6.2.1 `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.

6.2.2 `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.

6.2.3 `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 :

```c
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.

6.2.4 `MPI_Reduce` pour mesurer les performances

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

```c
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.

6.3 Résultats obtenus

Speedup Global en fonction de la taille de la grille (Partie 2)

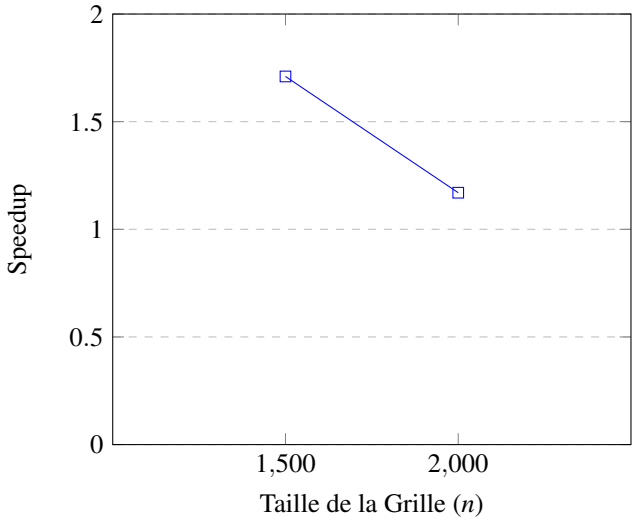


TABLE 3 – Speedup Global en fonction de la taille de la grille (Partie 2)

Taille de la Grille (n)	Temps Séquentiel (s)	Temps Parallèle (s)	Speedup
1500	0.297567	0.17396	1.71
2000	0.410889	0.349725	1.17

6.4 Interprétation du Speedup

— Pour n=1500 : Speedup = 1.71

  - Interprétation : La parallélisation MPI avec 2 processus permet d'obtenir une accélération significative de 71 % par rapport à la version séquentielle. Cela montre que la séparation des tâches (affichage sur le rank 0 et calcul sur le rank 1) est efficace pour cette taille de grille. Les communications entre les deux processus sont bien gérées et n'introduisent pas de goulot d'étranglement majeur.
- Pour n=2000 : Speedup = 1.17
  - Interprétation : Le speedup diminue à 1.17, ce qui correspond à une accélération de seulement 17 %. Cela indique que pour une grille plus grande, les temps de communication entre les processus deviennent plus importants par rapport au temps de calcul. La parallélisation MPI avec seulement 2 processus n'est pas suffisante pour exploiter efficacement les ressources disponibles pour des problèmes de grande taille.

6.5 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`.

7 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.

7.1 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.

7.2 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
        }
    }
}
```

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.

7.3 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 :

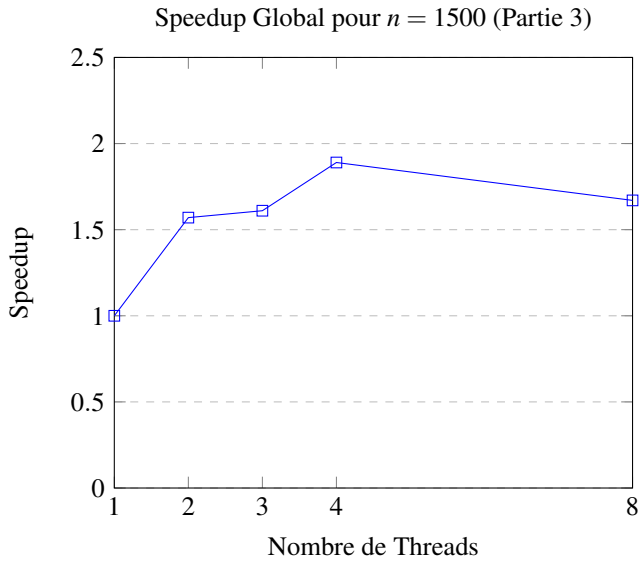
```
// Réduction du nombre de messages MPI grâce à OpenMP
MPI_Isend(vegetation.data(), vegetation.size(), MPI_UINT8_T, 0, 0, MPI_COMM_WORLD, &
    reqs[0]);
MPI_Isend(fire.data(), fire.size(), MPI_UINT8_T, 0, 1, MPI_COMM_WORLD, &reqs[1]);
MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
```

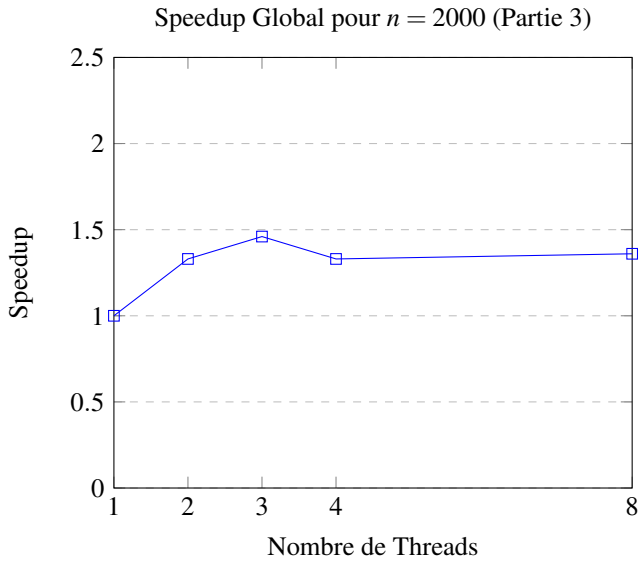
Avec cette optimisation :

— 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.

7.4 Résultats obtenus





7.5 Interprétation des résultats obtenus

- Pour n=1500 :
  - Le speedup augmente avec le nombre de threads, atteignant un maximum de 1.89 pour 4 threads.
  - Pour 8 threads, le speedup diminue légèrement à 1.67, ce qui est du à l'utilisation de oversubscribe. En effet, notre machine ne présente que 4 coeurs et pour aller au dela de ces 4 coeurs physiques, il faut utilise cette option qui a cependant des effets indésirables sur le speedup
- Pour n=2000 :
  - Le speedup est plus faible que pour n=1500, avec un maximum de 1.46 pour 3 threads.
  - Pour 4 et 8 threads, le speedup reste autour de 1.33-1.36, et ainsi la parrallélisation semble moins efficace pour une grille plus grande
- Analyse Globale :
  - La combinaison OpenMP + MPI montre une amélioration modérée des performances, avec un speedup maximal de 1.89 pour n=1500 et 1.46 pour n=2000.
  - La diminution du speedup pour un grand nombre de threads (8) est due au fait que notre ordinateur ne possède que 4 coeurs physiques.

7.6 Conclusion de l'implémentation

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

- Meilleure exploitation des ressources CPU : chaque processus MPI utilise plusieurs cœurs grâce à OpenMP.
- Amélioration des performances : calculs plus rapides .

8 Étape 4 : Parallélisation complète avec MPI

Dans cette dernière étape, la parallélisation est entièrement réalisée avec MPI, sans recours à OpenMP. Désormais, chaque processus MPI gère une sous-région spécifique de la grille de simulation et échange uniquement les informations nécessaires avec ses voisins.

8.1 Décomposition de la grille entre processus

Jusqu'à présent, chaque processus MPI manipulait la totalité de la grille ou envoyait l'ensemble des données au processus maître. Avec cette nouvelle approche, chaque processus ne gère qu'une portion de la grille. La grille est divisée en régions approximativement égales (égales lorsque la taille du terrain est un multiple du nombre de processus), chacune étant attribuée à un processus différent. Cette décomposition permet de favoriser un très bon équilibrage de charges entre les processus de calcul.

Cependant, cette décomposition de la grille sera iimiter par la présence de

- Dépendances Voisines : Chaque cellule de la grille dépend de ses 4 voisins directs (Nord, Sud, Est, Ouest). Les cellules Est et Ouest sont locales à la sous-grille, tandis que les cellules Nord et Sud peuvent appartenir à une autre sous-grille, nécessitant l'utilisation de cellules fantômes.

Pour faire face à cette difficulté, il est alors nécessaire d'introduire la notion d'introduire un échange de cellules "Fantômes".

- Cellules Fantômes : Les cellules fantômes sont des copies des cellules situées à la frontière des sous-grilles adjacentes. Elles permettent à chaque processus de connaître l'état des cellules voisines sans avoir à accéder directement à la mémoire d'un autre processus. Ces cellules sont mises à jour avant chaque itération de la simulation pour assurer une propagation correcte du feu entre les sous-grilles. Ainsi lorsque le feu se propage jusqu'à la limite entre deux regions de la grille detenues par des processus différents, nous pouvons mettre la grille globale à jour de façon cohérente comme ci la grille n'avait pas été diviser.

8.2 Introduction des cellules fantômes

L'un des défis de cette approche est la propagation correcte du feu entre les régions gérées par différents processus. Pour cela, des **cellules fantômes** sont introduites. Ces cellules permettent de stocker temporairement l'état des bords de chaque sous-région afin d'assurer une propagation du feu entre des régions différentes de la grille qui ne sont pas detenues par le meme processus.

L'échange des cellules fantômes se fait via une communication inter-processus :

```
//   change   des cellules fantômes avec les processus voisins
void update_ghost_cells(std::vector<std::uint8_t>& local_fire,
                        std::vector<std::uint8_t>& local_vegetation,
                        int local_rows, int cols,
                        MPI_Comm newComm, int comp_rank, int comp_size)
{
    // Echange pour le FEU ---------------------------------------------
    MPI_Request fire_requests[4];
    int fire_req_count = 0;

    // Réception Nord / Envoi Nord
    if (comp_rank > 0) {
        MPI_Irecv(local_fire.data(), cols, MPI_UINT8_T, comp_rank-1, 0, newComm, &
    fire_requests[fire_req_count++]);
        MPI_Isend(local_fire.data() + cols, cols, MPI_UINT8_T, comp_rank-1, 1, newComm, &
    fire_requests[fire_req_count++]);
    }

    // Réception Sud / Envoi Sud
    if (comp_rank < comp_size-1) {
        MPI_Irecv(local_fire.data() + (local_rows+1)*cols, cols, MPI_UINT8_T, comp_rank
    +1, 1, newComm, &fire_requests[fire_req_count++]);
        MPI_Isend(local_fire.data() + local_rows*cols, cols, MPI_UINT8_T, comp_rank+1, 0,
     newComm, &fire_requests[fire_req_count++]);
    }

    MPI_Waitall(fire_req_count, fire_requests, MPI_STATUSES_IGNORE);

    //   change   pour la VEGETATION ---------------------------------------
    MPI_Request veg_requests[4];
    int veg_req_count = 0;

    // Réception Nord / Envoi Nord (tags différents: 2/3)
    if (comp_rank > 0) {
        MPI_Irecv(local_vegetation.data(), cols, MPI_UINT8_T, comp_rank-1, 2, newComm, &
    veg_requests[veg_req_count++]);
        MPI_Isend(local_vegetation.data() + cols, cols, MPI_UINT8_T, comp_rank-1, 3,
    newComm, &veg_requests[veg_req_count++]);
    }

    // Réception Sud / Envoi Sud
    if (comp_rank < comp_size-1) {
        MPI_Irecv(local_vegetation.data() + (local_rows+1)*cols, cols, MPI_UINT8_T,
    comp_rank+1, 3, newComm, &veg_requests[veg_req_count++]);
        MPI_Isend(local_vegetation.data() + local_rows*cols, cols, MPI_UINT8_T, comp_rank
    +1, 2, newComm, &veg_requests[veg_req_count++]);
    }

    MPI_Waitall(veg_req_count, veg_requests, MPI_STATUSES_IGNORE);

    // Conditions aux limites NEUMANN ----------------------------------
    // Pour le FEU
    if (comp_rank == 0) {
        std::copy(local_fire.begin() + cols, local_fire.begin() + 2*cols, local_fire.
    begin());
    }
    if (comp_rank == comp_size-1) {
        std::copy(
            local_fire.begin() + (local_rows) * cols,
            local_fire.begin() + (local_rows + 1) * cols,
            local_fire.begin() + (local_rows + 1) * cols
        );
```

```
    }

    // Pour la VEGETATION
    if (comp_rank == 0) {
        std::copy(local_vegetation.begin() + cols,
                  local_vegetation.begin() + 2*cols,
                  local_vegetation.begin());
    }
    if (comp_rank == comp_size-1) {
        std::copy(local_vegetation.begin() + local_rows*cols,
                  local_vegetation.begin() + (local_rows+1)*cols,
                  local_vegetation.begin() + (local_rows+1)*cols);
    }
    MPI_Barrier(newComm);
}
```

8.3 Échange des Cellules Fantômes

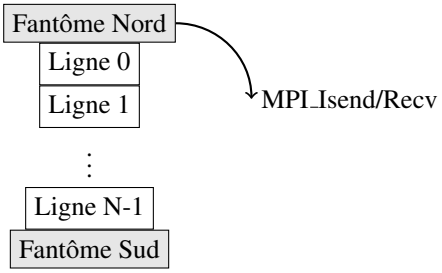
Cette fonction synchronise les cellules fantômes entre processus voisins pour deux grilles : **feu** et **végétation**.

8.3.1 Paramètres

- `local_fire/local_vegetation` : Grilles locales (incluant les fantômes)
- `local_rows` : Nombre de lignes *réelles* (sans fantômes)
- `cols` : Nombre de colonnes (identique pour tous)
- `newComm` : Communicateur MPI pour les processus de calcul
- `comp_rank` : Rang du processus dans `newComm`
- `comp_size` : Nombre total de processus de calcul

8.3.2 Architecture des Données

Structure mémoire :



8.3.3 Étapes d'Échange

1. **Communication Non-Bloquante** :
   — Utilisation de `MPI_Isend` et `MPI_Irecv`
   — 4 requêtes maximum par grille (Nord/Send+Recv et Sud/Send+Recv)
2. **Synchronisation** :
   — `MPI_Waitall` garantit la complétion des échanges
   — Barrière MPI finale pour synchroniser tous les processus

8.3.4 Conditions aux Limites

Pour les processus extrêmes (`comp_rank == 0` ou `comp_size-1`) :

Fantômes ← Copie de la première/dernière ligne réelle

`std::copy(begin+cols, begin+2*cols, begin)` → Neumann Nord

`std::copy(end-cols, end, end)` → Neumann Sud

8.3.5 Optimisations

— Chevauchement calcul/communication via MPI non-bloquant
— Séparation des canaux de communication (tags différents)
— Copie mémoire optimisée pour les bords

8.4 Répartition des Taches

— Processus d'Affichage (Rank 0) : Le processus de rang global 0 est responsable de l'affichage de la simulation. Il collecte les sous-grilles locales des autres processus et les affiche à l'écran.
— Processus de Calcul (Rangs 1 à N) : Les autres processus sont responsables du calcul de la propagation du feu sur leurs sous-grilles respectives. Ils mettent à jour les cellules fantômes et effectuent les calculs locaux.
— Processus de Relais (Rank 1) : Le processus de rang global 1 joue un rôle particulier en tant que relais entre le processus d'affichage et les autres processus de calcul. Il collecte les données des autres processus de calcul et les transmet au processus d'affichage.

8.5 Optimisation des échanges et synchronisation

Avec cette nouvelle organisation, l'envoi des données mises à jour ne se fait plus après chaque itération. À la place, une synchronisation globale permet de vérifier si la simulation doit continuer :

```
// Vérification de l'activité globale du feu
int local_active = local_continue ? 1 : 0;
int global_active = 0;
MPI_Allreduce(&local_active, &global_active, 1, MPI_INT, MPI_SUM, newComm);
simulation_running = (global_active > 0);
```

Grâce à cette approche :

— Tous les processus décident ensemble de l'arrêt de la simulation.
— La charge de communication est équilibrée entre les processus. Le processus de rang 1 a néanmoins plus de communications à faire. Il échange des informations cruciales avec le processus de l'affichage comme par exemple la grille globale rassemblée auprès de tous les processus de calculs.

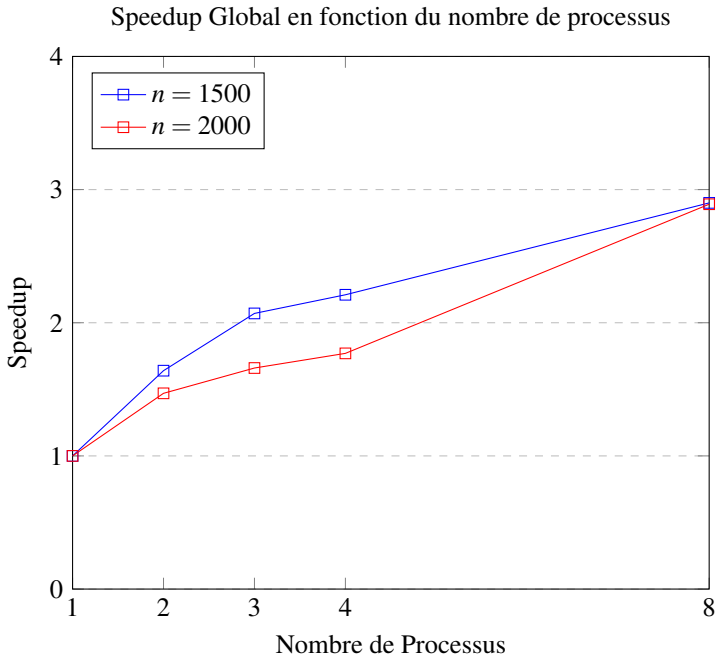
8.6 Récupération finale des données et affichage

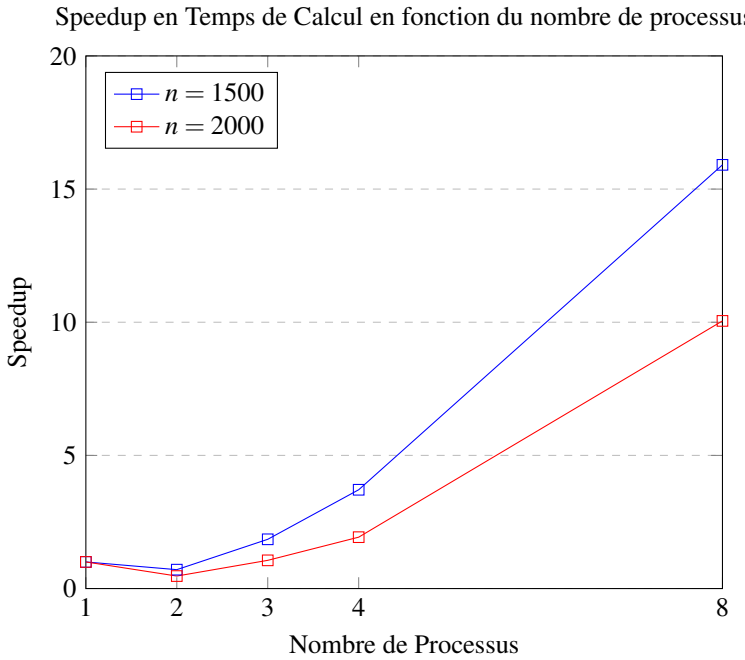
Une fois la simulation terminée, les données de chaque processus sont rassemblées pour être affichées. Contrairement aux versions précédentes où un seul processus gérait l'affichage, ici, les données sont récupérées de manière ordonnée grâce à `MPI_Gatherv` :

```
// Rassemblement des données finales
MPI_Gatherv(local_fire_data.data(), local_fire_data.size(), MPI_UINT8_T,
            global_fire.data(), counts.data(), displs.data(), MPI_UINT8_T,
            0, newComm);
```

Cette méthode permet d'optimiser la collecte des informations sans surcharger le processus maître.

8.7 Résultats obtenues





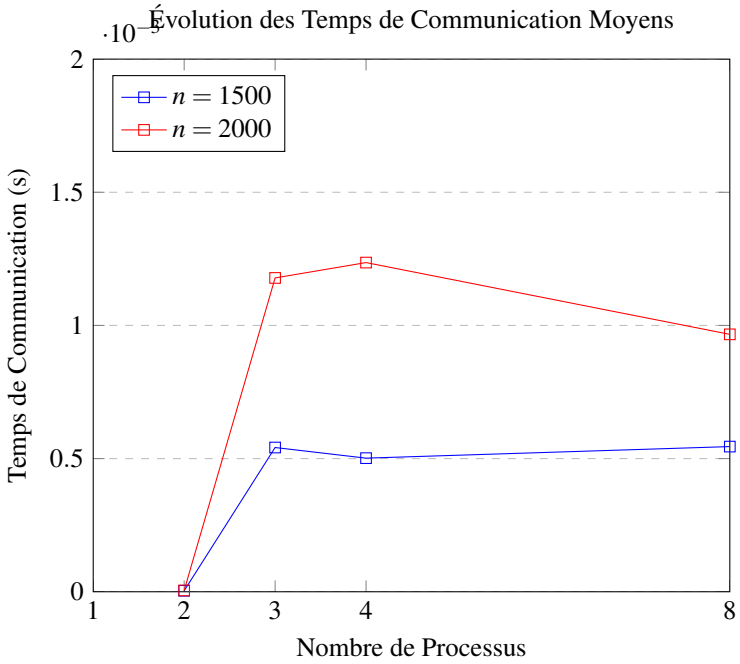


TABLE 4 – Temps d'exécution et Speedup pour $n = 1500$

Nombre de Processus	Temps Calcul (s)	Temps Affichage (s)	Temps Total (s)	Speedup (Global)	Speedup (Calcul)
1 (Séquentiel)	0.129754	0.167813	0.297567	1.00	1.00
2	0.181874	0.172482	0.181874	1.64	0.71
3	0.0701699	0.143596	0.143596	2.07	1.85
4	0.0349828	0.134875	0.134875	2.21	3.71
8	0.00815504	0.102482	0.102482	2.90	15.91

TABLE 5 – Temps d'exécution et Speedup pour $n = 2000$

Nombre de Processus	Temps Calcul (s)	Temps Affichage (s)	Temps Total (s)	Speedup (Global)	Speedup (Calcul)
1 (Séquentiel)	0.124577	0.286312	0.410889	1.00	1.00
2	0.265587	0.278707	0.278707	1.47	0.47
3	0.117127	0.247817	0.247817	1.66	1.06
4	0.0645298	0.232627	0.232627	1.77	1.93
8	0.0123973	0.142226	0.142226	2.89	10.05

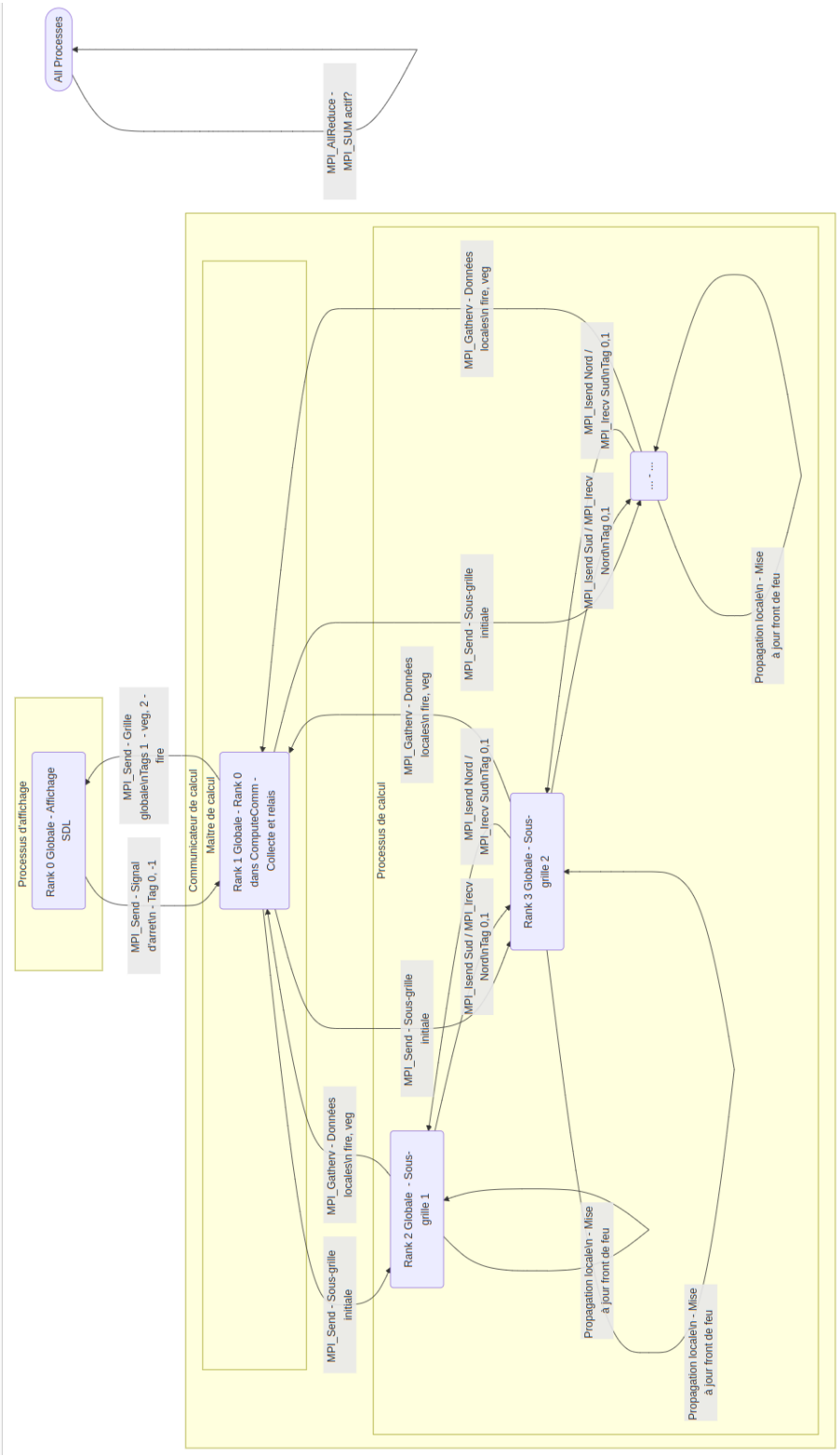


FIGURE 3 – Architecture des communications.

8.8 Conclusion

Avec cette version entièrement parallélisée avec MPI :

— L'utilisation de cellules fantômes assure la propagation correcte du feu entre sous-domaines.

— Cette approche favorise également un bon équilibrage de charges entre les processus de calcul.

Cette approche finalise l'optimisation de la simulation, assurant une exécution efficace sur des architectures massivement parallèles.

8.9 Interprétation des résultats

— Speedup Global : Le speedup global augmente avec le nombre de processus, mais il est limité par le temps d'affichage, qui ne peut pas être parallélisé. Pour n=1500, le speedup atteint 2.90 avec 8 processus, tandis que pour n=2000, il atteint 2.89.

— Speedup en Temps de Calcul : Le speedup en temps de calcul montre une amélioration significative, en particulier pour n=1500, où il atteint 15.91 avec 8 processus. Cela indique que la parallélisation est très efficace pour le calcul, mais que les communications et l'affichage limitent le speedup global.

— Temps de Communication : Les temps de communication augmentent avec le nombre de processus, mais restent relativement faibles par rapport au temps de calcul. Cela montre que les communications ne sont pas un goulot d'étranglement majeur dans cette simulation.

8.10 Comparaison avec la partie 3

— Partie 3 (OpenMP + MPI) : Le speedup est modéré, avec une amélioration significative pour 4 threads, mais une légère baisse pour 8 threads. Cela peut être dû à la surcharge de gestion des threads.

— Partie 4 (MPI seul) : Le speedup est nettement meilleur, en particulier pour 8 processus, où il atteint 2.90 pour n=1500 et 2.89 pour n=2000. Cela montre que la parallélisation MPI est plus efficace pour cette simulation.

— Comparaison : La partie 4 (MPI) surpasse la partie 3 (OpenMP + MPI) en termes de speedup, en particulier pour un grand nombre de processus. Cela suggère que MPI est mieux adapté pour ce type de simulation distribuée.

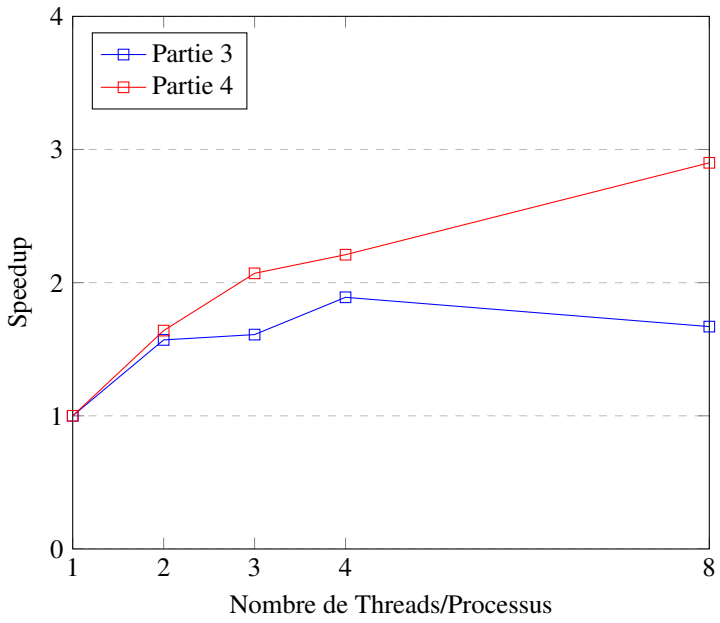
TABLE 6 – Speedup Global pour $n = 1500$

Nombre de Threads/Processus	Temps Partie 3 (s)	Speedup Partie 3	Temps Partie 4 (s)	Speedup Partie 4
1 (Séquentiel)	0.297567	1.00	0.297567	1.00
2	0.18941	1.57	0.181874	1.64
3	0.184722	1.61	0.143596	2.07
4	0.157255	1.89	0.134875	2.21
8	0.178025	1.67	0.102482	2.90

TABLE 7 – Speedup Global pour $n = 2000$

Nombre de Threads/Processus	Temps Partie 3 (s)	Speedup Partie 3	Temps Partie 4 (s)	Speedup Partie 4
1 (Séquentiel)	0.410889	1.00	0.410889	1.00
2	0.308398	1.33	0.278707	1.47
3	0.281419	1.46	0.247817	1.66
4	0.309858	1.33	0.232627	1.77
8	0.30248	1.36	0.142226	2.89

Comparaison des Speedups Globaux pour $n = 1500$



Comparaison des Speedups Globaux pour $n = 2000$