ECOLE NATIONALE SUPÉRIEURE DE TECHNIQUES AVANCÉES

OS202

Rapport Projet

Gabriel CORSI HONÓRIO et Diego FLEURY

Palaiseau, France 2025

Table des matières

1. Présentation générale

Parallélisation d'une Simulation de Feu de Forêt

Ce projet est inspiré du travail d'Elise FOULATIER, réalisé en 2021 à l'ENS de Paris-Saclay. L'objectif est de simuler la propagation d'un incendie en tenant compte de la densité de la végétation et des conditions de vent. La simulation utilise un modèle probabiliste basé sur des lois empiriques et sera parallélisée pour optimiser ses performances.

1.1. Modélisation de l'Incendie

La simulation repose sur une grille carrée discrétisée en N x N cellules. Chaque cellule possède deux cartes :

- **Carte d'incendie** : Indique l'intensité de l'incendie (0 à 255)
- **Carte de végétation** : Indique la densité de la végétation

1.2. États des Cellules

- **Saine** : Non touchée par le feu
- **Brûlante** : En train de brûler
- **Brûlée** : Incendie éteint et végétation épuisée

1.3. Probabilités Utilisées

- **p1** : Propagation du feu aux cellules voisines (fonction du vent et de la végétation)
- **p2** : Extinction du feu dans une cellule brûlante

1.4. Objectifs du Projet

L'objectif est d'optimiser la simulation d'incendie en parallélisant les calculs. Le travail est décomposé en plusieurs étapes, chacune disponible dans une branche spécifique du dépôt Git.

1.4.1. Organisation du Projet

Le projet est organisé comme suit :

- **Branch** `sequentiel` : Implémentation séquentielle fournie comme base du projet.
- **Branch** `premiere_partie` : Parallélisation avec OpenMP pour optimiser les calculs sur architectures à mémoire partagée.
- **Branch** `deuxieme_partie` : Séparation entre l'exibition et le traitement en utilisant MPI.
- **Branch** `troisieme_partie` : Combinaison des optimisations OpenMP et MPI des parties précédentes.

Note importante : La branche `main` ne doit pas être considérée pour l'évaluation du travail. Veuillez vous référer uniquement aux branches mentionnées ci-dessus.

Note sur les tests et l'évaluation : Des scripts de test ont été développés pour chaque partie du projet afin d'évaluer les performances :

- Scripts pour les tests de performance séquentielle
- Scripts pour les tests de parallélisation avec OpenMP
- Scripts pour les tests de performance avec MPI
- Scripts pour les tests combinant OpenMP et MPI

Note sur la documentation : Chaque branche contient un fichier README détaillant comment exécuter le programme et les tests correspondants. Veuillez vous référer à ces documents pour plus d'informations sur l'utilisation du code dans chaque partie du projet.

Note sur les fonctionnalités non implémentées : La partie bonus de la question 3 n'a pas été implémentée dans le cadre de ce projet.

2. Étape 1 : Parallélisation OpenMP

2.1. Identifier les caractéristiques matérielles de la machine

La commande suivante a été utilisée pour obtenir le nombre de cœurs et les tailles des caches :

```
$ lscpu | grep -E 'Socket|Core|Thread|L[1-3][di]? cache'
```

Et voici la sortie :

```
Model name:                      AMD Ryzen 5 7600X 6-Core Processor
Thread(s) per core:              2
Core(s) per socket:              6
Socket(s):                       1
L1d cache:                       192 KiB (6 instances)
L1i cache:                       192 KiB (6 instances)
L2 cache:                        6 MiB (6 instances)
L3 cache:                        32 MiB (1 instance)
```

2.2. Mesurer les temps d'exécution en mode séquentiel

Pour obtenir les résultats suivants, nous avons utilisé la version du code disponible sur la branche "sequentiel".

Nous avons affiché le temps d'afflichage ainsi que le temps total à des fins de comparaison. Dans les études suivantes, seul le temps d'avancement sera pris en compte.

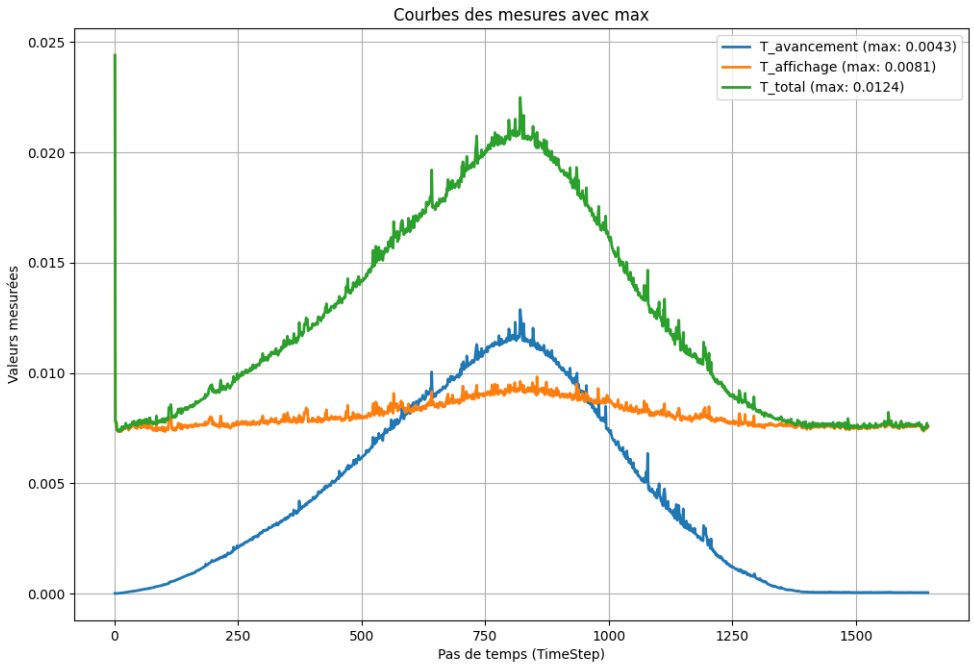

Figure 1: Tous les temps

Note : On peut constater que le temps d'affichage reste plus ou moins constant dans le temps et ne varie qu'en fonction de la taille de la grille. Par conséquent, il ne sera pas pris en compte dans les analyses futures.

2.3. Paralléliser l'avancement en temps avec OpenMP

Avant la modification, l'information sur les cellules en feu était stockée dans une table de hachage non ordonnée (unordered map ou hash table), ce qui permet un accès rapide mais complique une itération ordonnée.

Après la modification, ces informations ont été organisées sous forme de vecteur indexé. Cela permet de :

- ❖ Garder une structure plus simple (avec un index numérique au lieu d'une clé hachée).
- ❖ Parcourir efficacement les données grâce à un simple parcours de tableau, ce qui est plus adapté à la parallélisation avec OpenMP.

Note : Avant d'utiliser ‘schedule(dynamic, 256)‘, l'implémentation avec ‘pragma omp parallel for‘ posait un problème d'accès concurrent aux données globales, entraînant

des conflits dans la zone critique. Comme plusieurs threads tentaient de modifier simultanément les mêmes structures de données partagées (next_front), cela causait des incohérences (crash). La solution finale est décrite ci-dessous.

Résumé de la parallélisation OpenMP

Lorsqu'on crée une région parallèle avec 'pragma omp parallel', plusieurs threads exécutent leur code indépendamment, accélérant ainsi les calculs en répartissant les tâches entre les cœurs du processeur.

Le mot-clé 'pragma omp single' garantit qu'un seul thread redimensionne 'all_propagations' en fonction du nombre de threads actifs ('omp_get_num_threads()').

La structure 'FirePropagation' regroupe les informations sur la propagation du feu : position de la cellule ('cell_position'), intensité ('fire_intensity') et indicateur de nouvel incendie ('is_new_fire').

Chaque thread stocke temporairement ses mises à jour dans un vecteur 'propagations', puis elles sont fusionnées dans 'all_propagations' pour éviter les conflits d'accès mémoire. Une fois le calcul terminé, les mises à jour sont appliquées à 'm_fire_map' et 'next_front' pour assurer une évolution cohérente du front de feu.

Avec 'schedule(dynamic, 256)', chaque thread traite 256 cellules avant d'en récupérer d'autres, équilibrant ainsi la charge de travail, notamment lorsque certaines cellules nécessitent plus de calculs.

2.4. Mesurer l'accélération obtenue

Note : Pour garantir la même quantité de cellules pour tous les processus (cas du modèle de Gustafson), la taille de la grille a été ajustée selon le calcul montré ci-dessous. La longueur du côté du carré **n** a été fixée à 204 afin que, lorsqu'il y a 6 threads, le nombre total de cellules soit équivalent à celui du modèle d'Amdahl.

$n = 204 * \sqrt{p}$

Où p est le nombre de threads.

2.4.1. SpeedUp Amdahl

La loi théorique d'Amdahl donne le speedup maximal en fonction de la fraction séquentielle f du programme :

$$S_A(p) = \frac{1}{f + \frac{1-f}{p}}$$

où : - p est le nombre de processeurs, - f est la fraction du programme qui est séquentielle.

Définition pratique

Le speedup est défini comme le rapport entre le temps d'avancemment d'exécution en mode séquentiel T_s et le temps d'exécution en mode parallèle T_p :

$$S(p) = \frac{T_s}{T_p}$$

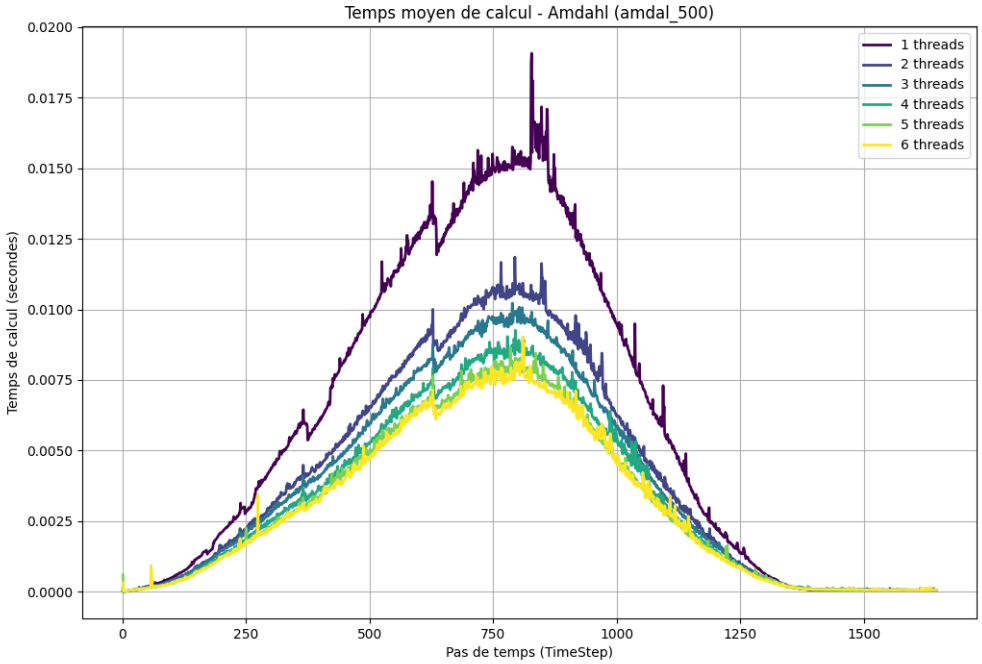

Figure 2: Timing curves Amdal

2.4.2. SpeedUp Gustafson

La loi théorique de Gustafson propose une autre vision, en considérant que l'augmentation du nombre de processeurs permet d'augmenter la taille du problème :

$$S_G(p) = p - f \cdot (p - 1)$$

où : - p est le nombre de processeurs, - f est la fraction du programme qui est séquentielle.

Définition pratique

Le speedup est défini comme le rapport entre le temps d'avancemment d'exécution en mode séquentiel T_s et le temps d'exécution en mode parallèle T_p :

$$S(p) = \frac{T_s}{T_p/p}$$

où :

- p est le nombre de processeurs.

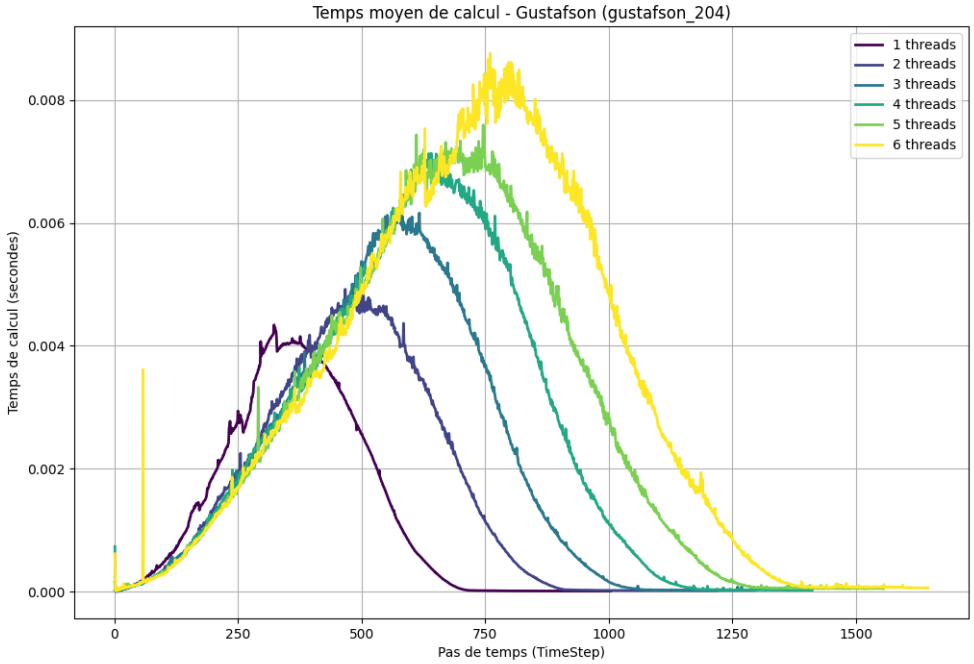

Figure 3: Timing curves Gustafson

2.4.3. Resultats

Dans la comparaison ci-dessous, on a deux figures : l'une sur le speed-up basé sur le temps parallélisé (nombre de threads égal à 1) (Figure 4) et l'autre basée sur le temps séquentiel (Figure 5). On peut voir que le résultat le plus pessimiste est donné par Amdahl, tandis que Gustafson fournit une estimation plus optimiste. L'accélération réelle se situe entre ces deux courbes.

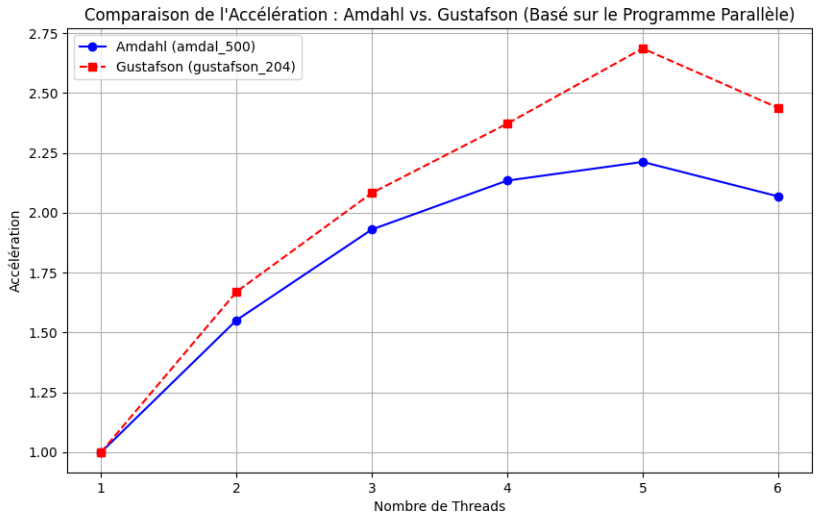

Figure 4: Comparaison des SpeedUp - basé Parallèle

On observe que l'accélération atteint son maximum avec 5 threads avant de diminuer légèrement à 6 threads. Cette diminution peut être attribuée à la gestion des ressources par le planificateur du système d'exploitation (scheduler), qui peut introduire des surcharges liées au partage des threads ou à la gestion du parallélisme.

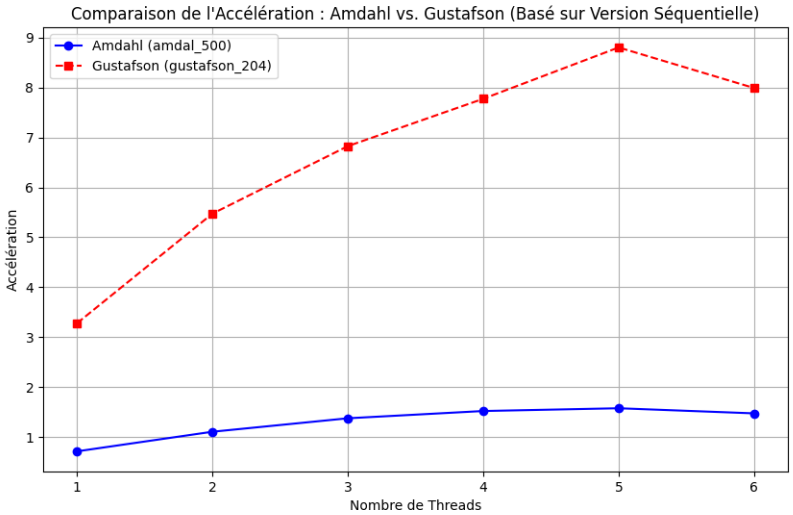

Figure 5: Comparaison des SpeedUp - basé Séquentielle

On observe que l'accélération de Gustafson augmente de manière significative, atteignant des valeurs superlinéaires (speed-up supérieur au nombre de threads). Ce comportement est attendu, car la loi de Gustafson prend en compte l'augmentation proporti-

onnelle de la charge de travail avec le nombre de processeurs, ce qui réduit l'influence de la partie séquentielle. De plus, des facteurs tels qu'une meilleure exploitation du cache et un impact réduit de la surcharge de parallélisation peuvent contribuer à cette accélération accrue.

3. Étape 2 : Parallélisation MPI

3.1. Intégrer MPI pour la gestion des processus

Nous avons modifié les fichiers Makefile, run_tests et les scripts Python afin d'utiliser MPI pour la compilation (MPI_CC) et l'exécution (MPI_exec). L'objectif était de répartir les calculs entre plusieurs processus afin d'améliorer les performances. Pour cette première implémentation, nous avons divisé l'exécution en deux processus distincts.

3.2. Déléguer l'affichage au processus principal

La communication est principalement réalisée via MPI_Send et MPI_Recv, avec l'utilisation de MPI_Irecv et MPI_Test pour éviter les blocages. Le processus 0 reçoit en continu les données de la simulation, y compris le temps de l'étape actuelle, les cartes de végétation et de feu, ainsi que les temps de calcul. Il détecte également les signaux d'arrêt et envoie une commande de terminaison au processus 1 via MPI_Isend. De son côté, le processus 1 vérifie s'il doit interrompre l'exécution grâce à MPI_Iprobe, effectue les calculs et envoie les résultats au processus 0. L'utilisation de MPI_Comm_dup garantit une copie sécurisée du communicateur MPI afin d'éviter toute interférence. À la fin, MPI_Comm_free libère le communicateur dupliqué, et MPI_Abort est appelé dans le processus 0 pour assurer une terminaison correcte.

Note : L'implémentation d'une communication dupliquée a été une initiative visant à prévenir d'éventuelles complications futures concernant la gestion de la communication entre plusieurs processus. Cependant, à ce stade, elle est totalement dispensable.

3.3. Mesurer les temps obtenue avec deux processus

Nous avons ensuite évalué le temps obtenue en exécutant la simulation sur cinq scénarios différents, en faisant varier la taille de la grille de 100x100 à 500x500. Pour chaque scénario, nous avons effectué cinq tests et mesuré le temps d'exécution moyen.

Les résultats sont présentés sous forme de graphiques :

- Un premier graphique montre l'évolution du temps moyen en fonction de la taille de la grille (Figure 6).
- Deux autres graphiques décomposent le temps total en calcul, communication et affichage, l'un en échelle linear (Figure 7) et l'autre en échelle logarithmique (Figure 8), pour une meilleure visualisation.

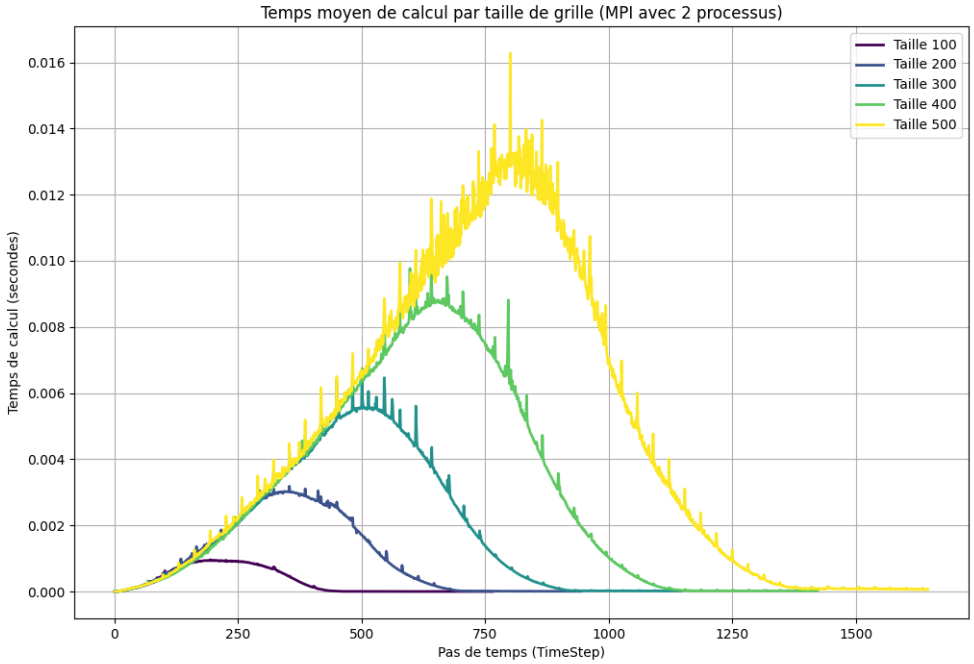

Figure 6: Courbes de temps par la taille de grille

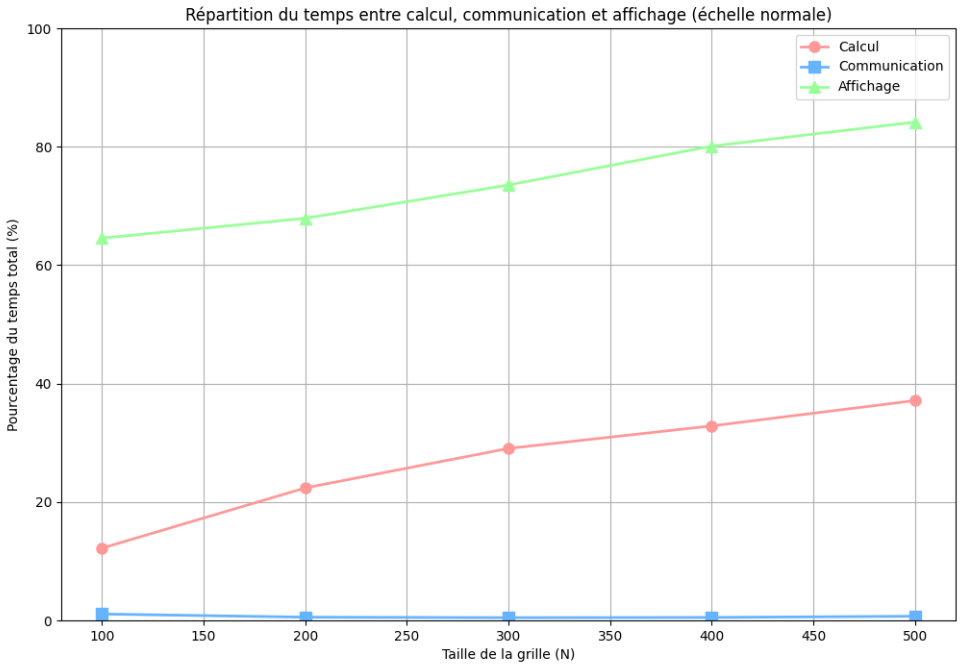

Figure 7: Courbes de les temps - linear

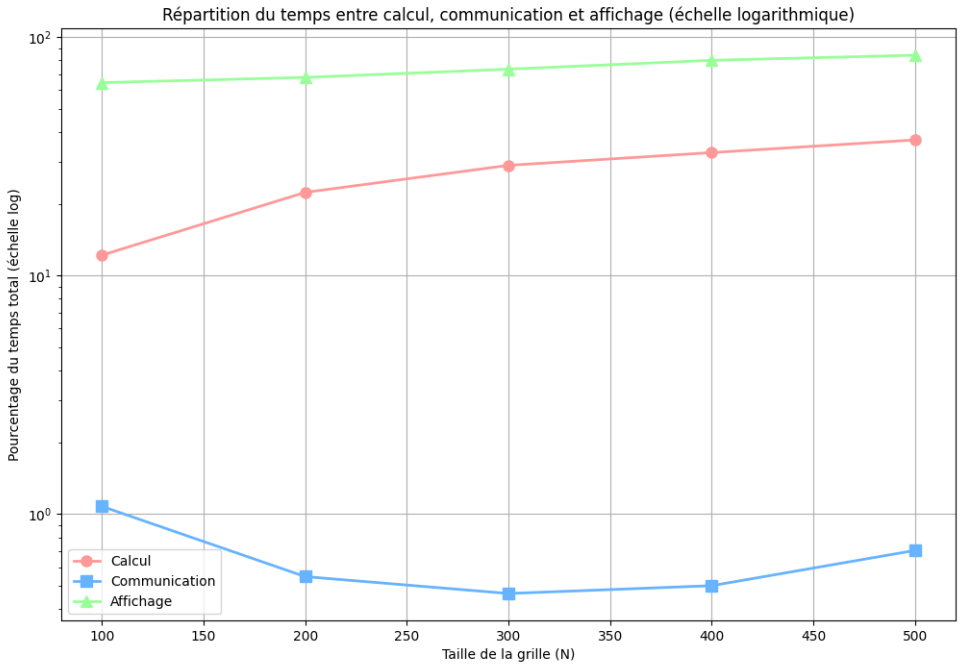

Figure 8: Courbes de les temps - monolog

On peut observer que la proportion du temps total consacrée à la communication diminue à mesure que la taille du problème (c'est-à-dire la taille de la grille) augmente. Cela s'explique par le fait que le temps de communication ne croît pas aussi rapidement que le temps de calcul. En parallèle, le temps d'affichage augmente lentement, tandis que le temps de traitement croît de manière plus significative, car la charge de calcul devient plus importante avec une grille plus grande.

4. Étape 3 : Combinaison OpenMP + MPI

4.1. Intégration d'OpenMP et MPI dans le projet

Dans cette étape, nous avons repris la parallélisation OpenMP déjà effectuée à la première étape et l'avons utilisée pour paralléliser l'avancement en temps du code obtenu à la deuxième étape. Pour cela, nous avons modifié le Makefile ainsi que les scripts d'exécution afin de prendre en charge l'intégration conjointe d'OpenMP et MPI.

De plus, la note mentionnée dans l'énoncé a été prise en compte : nous avons ajouté l'option –bind-to none lors du lancement du programme avec OpenMPI afin d'assurer un bon fonctionnement de la parallélisation OpenMP.

4.2. Analyse de l'accélération et impact d'OpenMP sur la performance

Nous avons évalué l'accélération globale et l'accélération spécifique de l'avancement en temps en fonction du nombre de threads afin d'analyser l'efficacité de la parallélisation. En comparant ces deux métriques, nous pouvons déterminer dans quelle mesure OpenMP contribue aux gains de performance et quel est son impact par rapport à l'exécution globale du programme. L'interprétation des résultats met en évidence que l'accélération est principalement due à l'avancement en temps, tandis que l'influence de MPI reste limitée.

Dans la comparaison ci-dessous, on a deux figures : l'une sur le speed-up basé sur le temps parallélisé (nombre de threads égal à 1) (Figure 9) et l'autre basée sur le temps séquentiel (Figure 10).

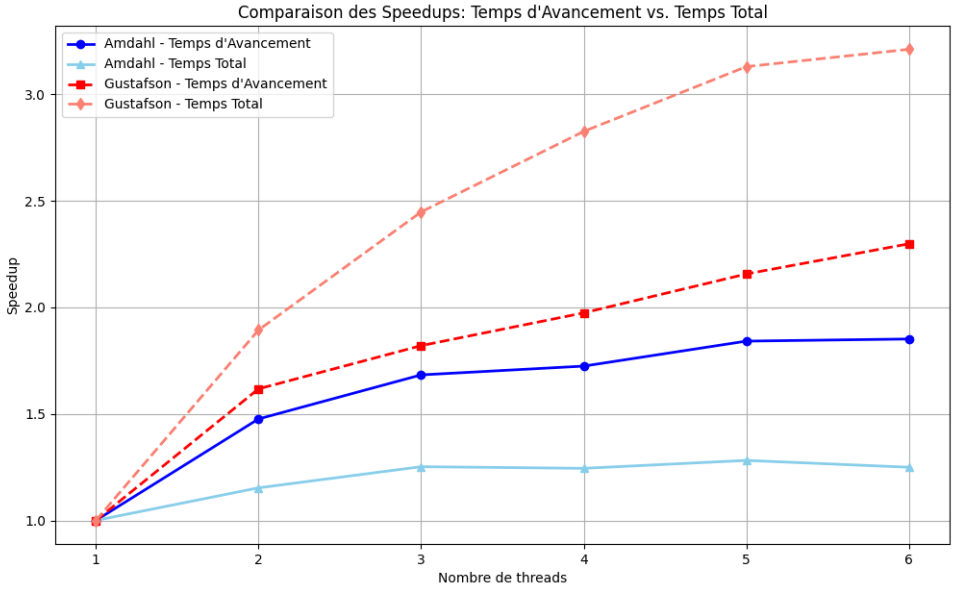

Figure 9: Comparaison des SpeedUp - basé Parallèle

On observe que les performances entre les courbes centrales (temps d'avancement) et les courbes externes (temps total) sont moins bonnes dans le modèle d'Amdahl. Cela signifie que, dans le cas d'Amdahl, l'ajout de threads entraîne une augmentation des coûts de communication via MPI, ce qui dégrade les performances globales. En effet, comme Amdahl suppose une partie séquentielle fixe, l'accroissement du nombre de threads met en évidence les limitations dues aux échanges entre processus, réduisant ainsi l'efficacité

globale.

En revanche, dans le cas de Gustafson, où la charge de travail totale augmente avec le nombre de threads, l'impact relatif des communications devient moins significatif, ce qui explique pourquoi le temps total continue de s'améliorer avec l'augmentation des threads. Ainsi, l'OpenMP conserve son efficacité dans les deux cas, mais le MPI devient un facteur limitant uniquement pour Amdahl, soulignant l'importance du modèle de parallélisation adopté.

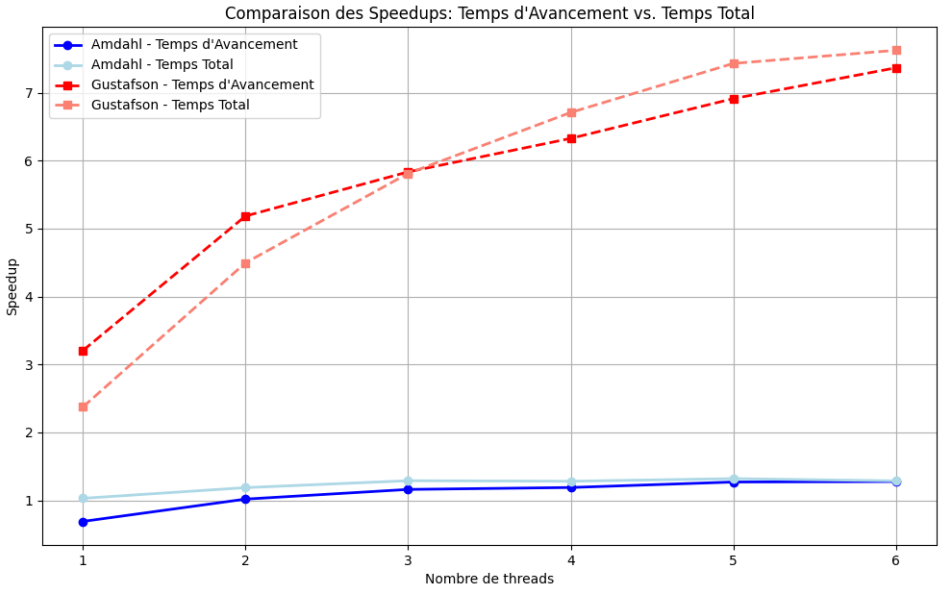

Figure 10: Comparaison des SpeedUp - basé Séquentielle

Dans cette comparaison, on observe que le speedup d'Amdahl reste pratiquement constant, avec une très faible amélioration, tandis que celui de Gustafson continue d'augmenter. La différence entre les courbes montre que le temps d'avancement (OpenMP) est le principal facteur influençant la performance, car le temps total (incluant MPI) reste proche. Ainsi, le MPI a peu d'impact sur l'amélioration globale, et l'OpenMP est le moteur principal du gain de vitesse observé.

4.3. Analyse de l'efficacité et impact d'OpenMP sur la performance

Nous avons analysé l'efficacité en fonction du nombre de threads afin d'évaluer l'impact d'OpenMP sur la performance du programme. En comparant les modèles d'Amdahl

et de Gustafson, nous pouvons identifier comment la parallélisation influe sur les gains de performance.

Dans la comparaison ci-dessous, on a deux figures : l'une sur le speed-up basé sur le temps parallélisé (nombre de threads égal à 1) (Figure 11) et l'autre basée sur le temps séquentiel (Figure 12).

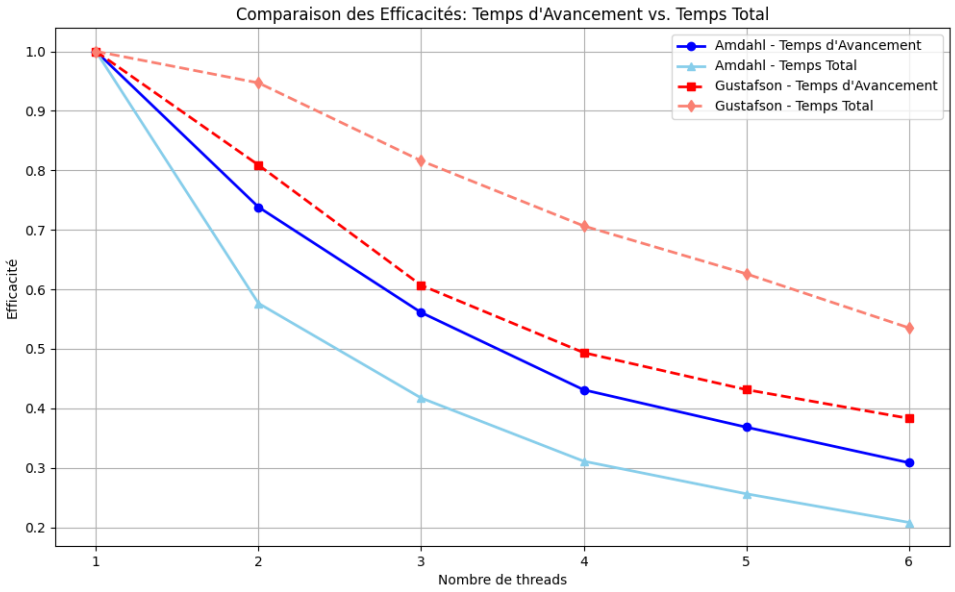

Figure 11: Comparaison des Efficacites - basé Parallèle

L'image montre l'efficacité du programme en fonction du nombre de threads, en considérant l'accélération basée sur le temps séquentiel du code parallélisé pour un thread. Nous observons qu'à mesure que le nombre de threads augmente, l'efficacité diminue significativement pour les deux modèles (Amdahl et Gustafson). Cette baisse s'explique par le surcoût de la parallélisation, notamment en raison de la communication et de la synchronisation entre les threads, ce qui réduit les gains attendus. Dans le cas du modèle d'Amdahl, l'efficacité diminue plus fortement, reflétant la limitation imposée par la fraction séquentielle du code. En revanche, dans le modèle de Gustafson, la différence est moindre.

Comme la différence entre les courbes d'Amdahl et de Gustafson reste constante, aussi bien pour les temps d'avancement que pour les temps totaux, la baisse d'efficacité est principalement influencée par le MPI.

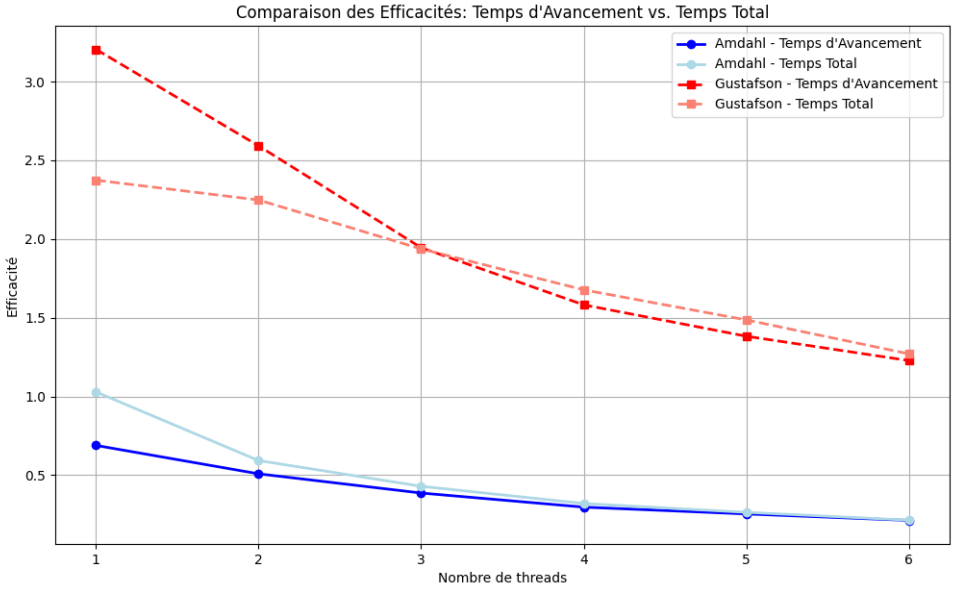

Figure 12: Comparaison des Efficacites - basé Séquentielle

Comme précédemment, nous constatons une diminution de l'efficacité avec l'augmentation du nombre de threads, mais avec une dynamique différente.

Le modèle de Gustafson maintient une efficacité plus élevée, notamment pour les premiers threads, soulignant son hypothèse d'expansion du problème parallélisable. En revanche, le modèle d'Amdahl subit une chute plus rapide, en raison de la fraction séquentielle qui limite les gains de parallélisation. Par ailleurs, la différence entre les courbes de temps d'avancement et de temps total reste visible, confirmant que l'impact du MPI joue un rôle important dans la dégradation de l'efficacité.