

# Rapport de Projet : Parallélisation de l'Algorithme de Darboux

---

|       |  |   |
|-------|--|---|
| 1.    | <b>Introduction</b> .....                                | 2 |
| 2.    | <b>Architecture de la solution</b> .....                 | 2 |
| 2.1   | Structure générale.....                                  | 2 |
| 2.2   | Organisation du code .....                               | 2 |
| 3.    | <b>Implémentation détaillée</b> .....                    | 3 |
| 3.1   | Parallélisation OpenMP .....                             | 3 |
| 3.2   | Parallélisation MPI.....                                 | 3 |
|       | Distribution initiale des données par blocs .....        | 3 |
|       | Échanges de frontières optimisés .....                   | 3 |
|       | Gestion de la convergence globale .....                  | 3 |
|       | Rassemblement final des résultats .....                  | 3 |
| 3.3   | Version hybride MPI+OpenMP.....                          | 3 |
| a.    | Un processus MPI par nœud de calcul: .....               | 3 |
| b.    | Multithreading OpenMP au sein de chaque processus: ..... | 3 |
| c.    | Optimisation des communications inter-nœuds:.....        | 4 |
| 4.    | <b>Analyse des performances</b> .....                    | 4 |
| 4.1   | Méthodologie d'évaluation .....                          | 4 |
| 4.2   | Résultats expérimentaux.....                             | 5 |
| 4.2.1 | Temps d'exécution séquentiel.....                        | 5 |
| 4.2.2 | Performances OpenMP .....                                | 5 |
| 4.2.3 | Performances hybrides MPI+OpenMP .....                   | 6 |
| 4.2.4 | Analyse des accélérations .....                          | 7 |
| 5.    | <b>Conclusion</b> .....                                  | 7 |

## 1. Introduction

---

Le présent rapport décrit notre travail de parallélisation de l'algorithme de Darboux, utilisé pour le calcul du remplissage des cuvettes dans un Modèle Numérique de Terrain.

L'objectif principal était d'optimiser les performances de l'algorithme en utilisant deux approches de parallélisation :

- OpenMP pour l'exploitation des architectures
- MPI pour la distribution des calculs sur un cluster
- Une approche hybride combinant les deux technologies.

## 2. Architecture de la solution

---

### 2.1 Structure générale

Notre implémentation repose sur une décomposition par domaine des données, où les matrices sont découpées en bandes horizontales. Cette approche permet :

- Une distribution équilibrée de la charge de calcul
- Une minimisation des communications entre processus
- Une exploitation efficace de la localité des données

### 2.2 Organisation du code

Le code est structuré autour de plusieurs composants clés :

- Une fonction principale de calcul (`darboux()`)
- Des routines d'échange de frontières (`exchange_boundaries()`)
- Des fonctions de distribution et rassemblement des données (`gather_data()`)

Nous avons utilisé la compilation conditionnelle pour générer trois versions différentes de l'exécutable :

- Version séquentielle : `main`
- Version OpenMP : `main_OMP`
- Version hybride MPI + OpenMP : `main_OMP_MPI`

Les directives de compilation conditionnelle (-DOMP et -DMPI) nous permettent de maintenir un code source unique tout en générant ces différentes versions.

### 3. Implémentation détaillée

#### 3.1 Parallélisation OpenMP

La parallélisation OpenMP se concentre sur la boucle de calcul principale :

```
#pragma omp parallel for reduction(|:modif)
for (int i = start; i < end; i++) {
    for (int j = 0; j < ncols; j++) {
        modif |= calcul_Wij(W, Wprec, m, i, j);
    }
}
```

#### 3.2 Parallélisation MPI

Distribution initiale des données par blocs

```
int blockSize = nrows / size;
int start = rank * blockSize;
int end = (rank == size - 1)? nrows : start + blockSize;
```

Échanges de frontières optimisés

```
void exchange_boundaries(float *W, int start, int end, int ncols, int rank, int size);
```

Gestion de la convergence globale

```
int global_modif = 0;
MPI_Allreduce(&modif, &global_modif, 1, MPI_INT, MPI_LOR, MPI_COMM_WORLD);
modif |= global_modif;
```

Rassemblement final des résultats

```
void gather_data(float *W, int blockSize, int ncols, int nrows, int rank, int size);
```

#### 3.3 Version hybride MPI+OpenMP

Notre approche hybride combine MPI et OpenMP pour exploiter efficacement les ressources matérielles disponibles. Cette implémentation s'articule autour de trois axes principaux :

a. Un processus MPI par nœud de calcul:

Nous utilisons MPI pour distribuer le calcul entre les différents nœuds du cluster. Chaque nœud reçoit un processus MPI qui est responsable d'une portion spécifique des données. Cette approche permet de minimiser les communications inter-nœuds tout en maximisant l'utilisation des ressources locales.

b. Multithreading OpenMP au sein de chaque processus:

À l'intérieur de chaque processus MPI, nous utilisons OpenMP pour paralléliser les calculs sur les cœurs disponibles du nœud. Cette parallélisation est appliquée sur la boucle principale de calcul, avec une réduction pour la variable de modification. Nous avons également implanté un système de mesure du temps d'exécution pour chaque thread afin d'analyser la répartition de la charge de travail.

c. Optimisation des communications inter-nœuds:

Les communications entre les processus MPI sont optimisées pour réduire leur impact sur les performances. Après chaque itération de calcul, les processus échangent uniquement les données de frontière nécessaires et synchronisent leur état de convergence. Cette synchronisation est réalisée de manière efficace grâce à une opération de réduction collective MPI.

La combinaison de ces trois aspects nous permet d'exploiter efficacement à la fois le parallélisme à mémoire partagée au sein de chaque nœud et le parallélisme à mémoire distribuée entre les nœuds du cluster. Cette approche est particulièrement efficace pour les grands jeux de données où le coût des communications peut être amorti par le gain en puissance de calcul.

## 4. Analyse des performances

---

### 4.1 Méthodologie d'évaluation

Les tests ont été réalisés sur le cluster OpenStack avec :

- Différentes tailles de données (small, medium, large)
- Variation du nombre de threads OpenMP
- Variation du nombre de processus MPI
- Mesures répétées pour assurer la fiabilité 🚨 🚨
- L'exécution MPI+OMP est faite avec : `mpirun -hostfile host.txt --map-by ppr:1:node ./executable [arguments]`

🚨 🚨 En raison de la latence de connexion avec les machines virtuelles, nous avons effectué un minimum de 10 exécutions pour chaque configuration et retenu les meilleurs temps d'exécution. Cette approche nous permet d'obtenir des mesures plus représentatives des performances réelles en minimisant l'impact des variations de latence réseau et des perturbations système.

## 4.2 Résultats expérimentaux

### 4.2.1 Temps d'exécution séquentiel

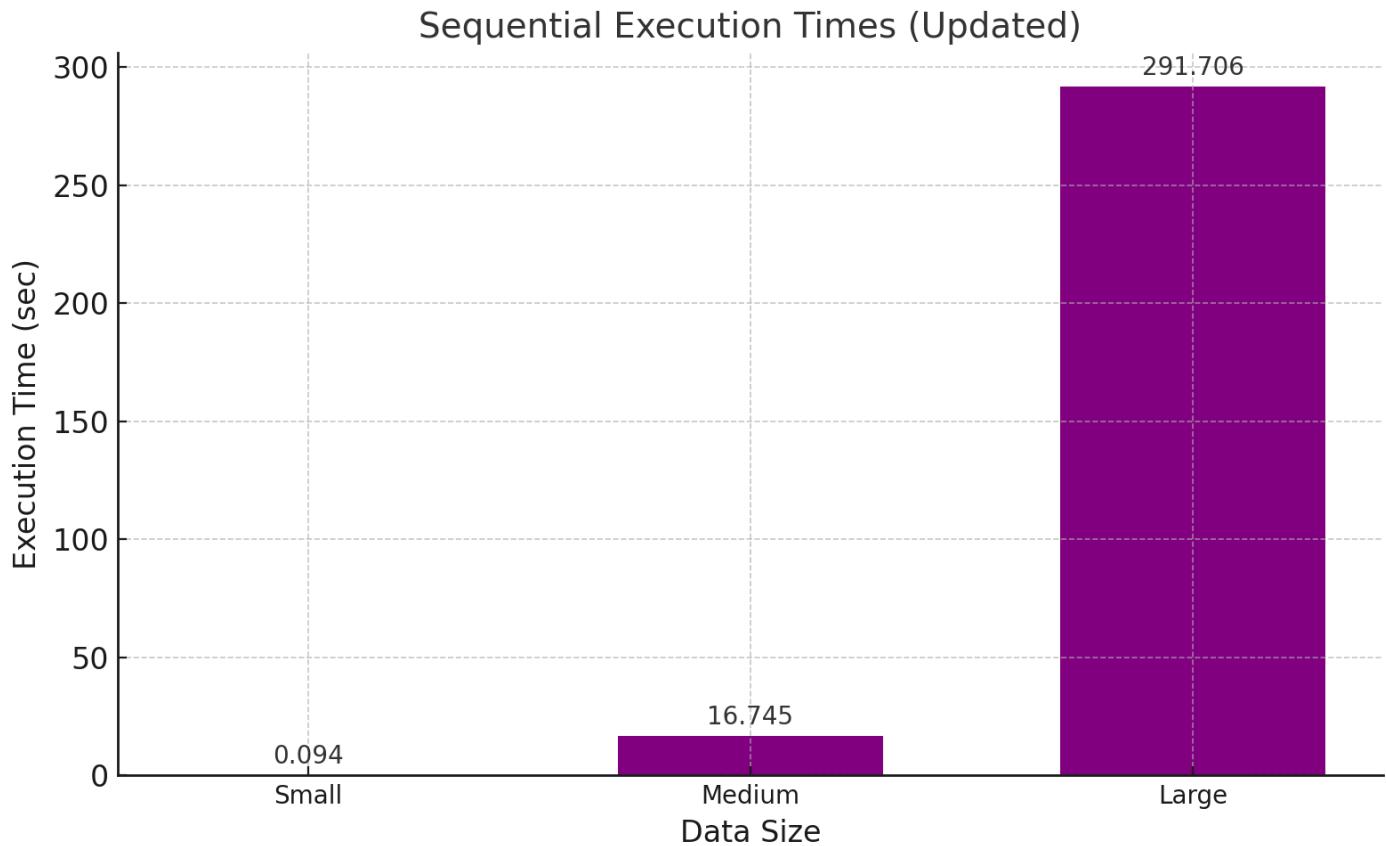


Figure 1 : Évolution du temps d'exécution séquentiel selon la taille des données

### 4.2.2 Performances OpenMP

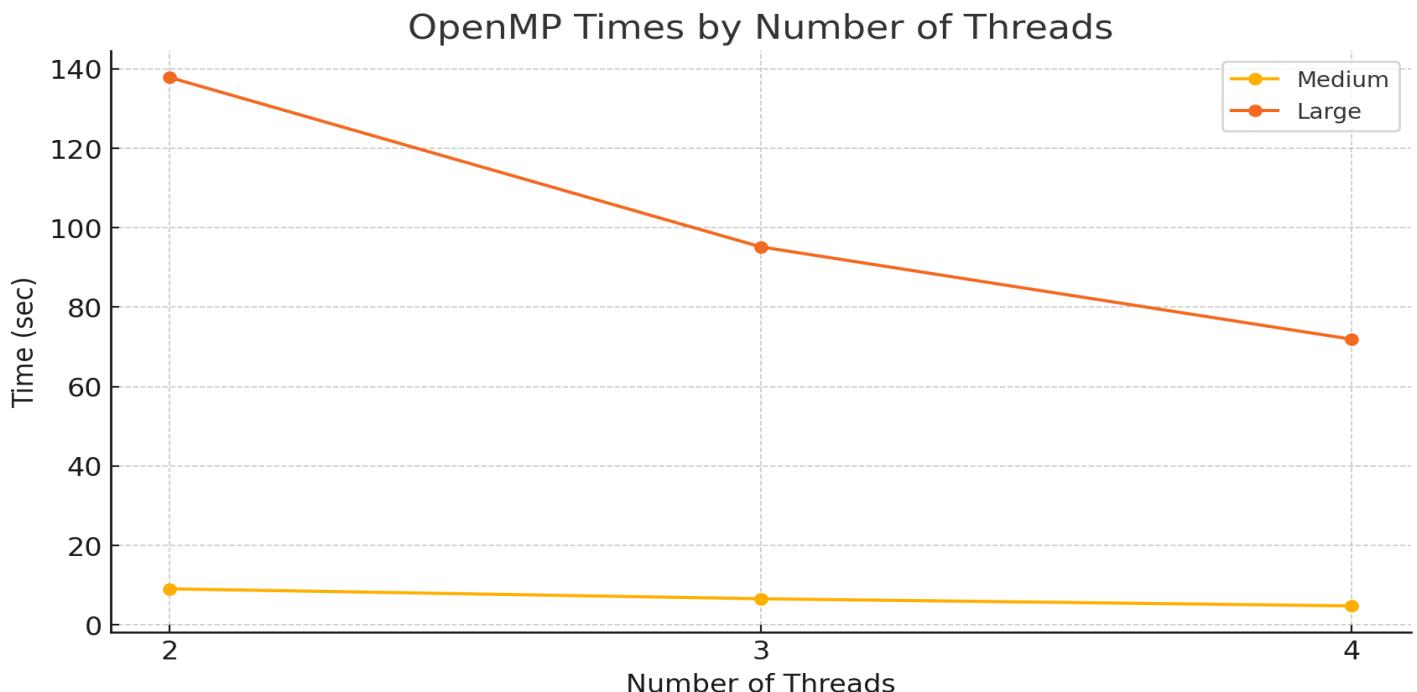


Figure 2 : Impact du nombre de threads sur les performances OpenMP

Voici le speed-up:

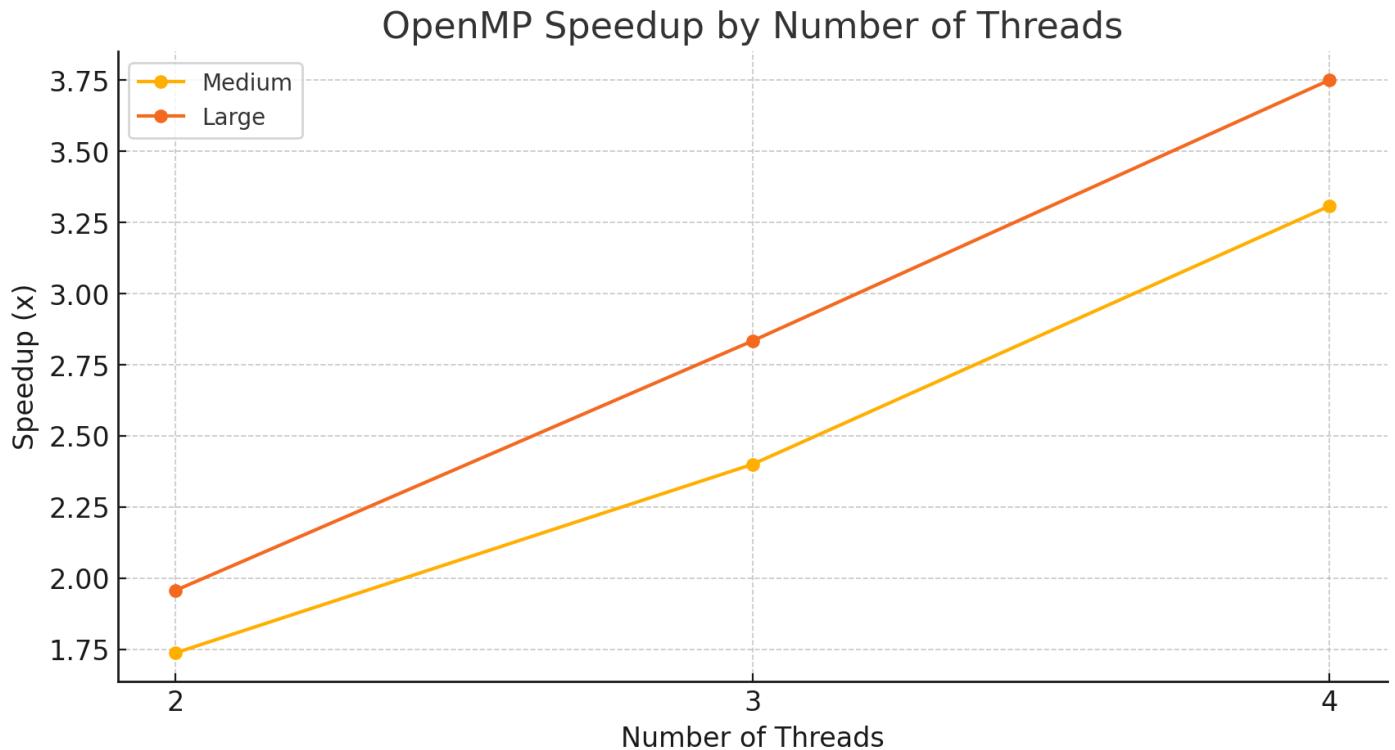


Figure 3 : Comparaison des accélérations obtenues avec OMP

#### 4.2.3 Performances hybrides MPI+OpenMP (tous sur 4 threads (--map-by ppr:1:node))

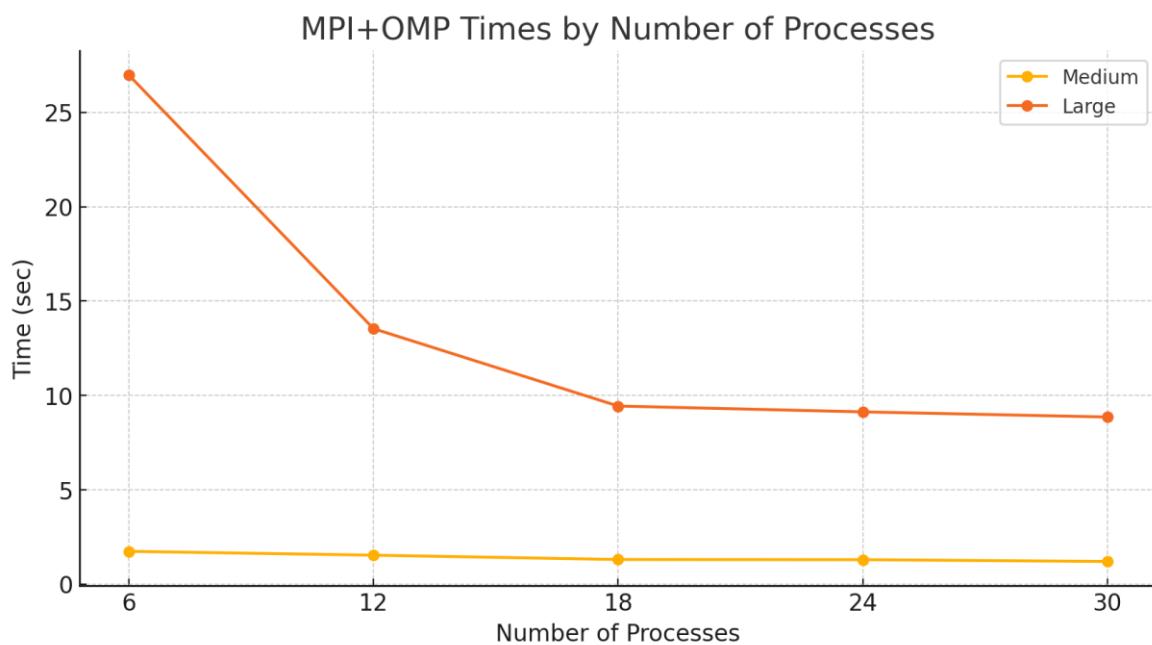


Figure 4 : Temps d'exécution de la version hybride selon le nombre de processus

#### 4.2.4 Analyse des accélérations

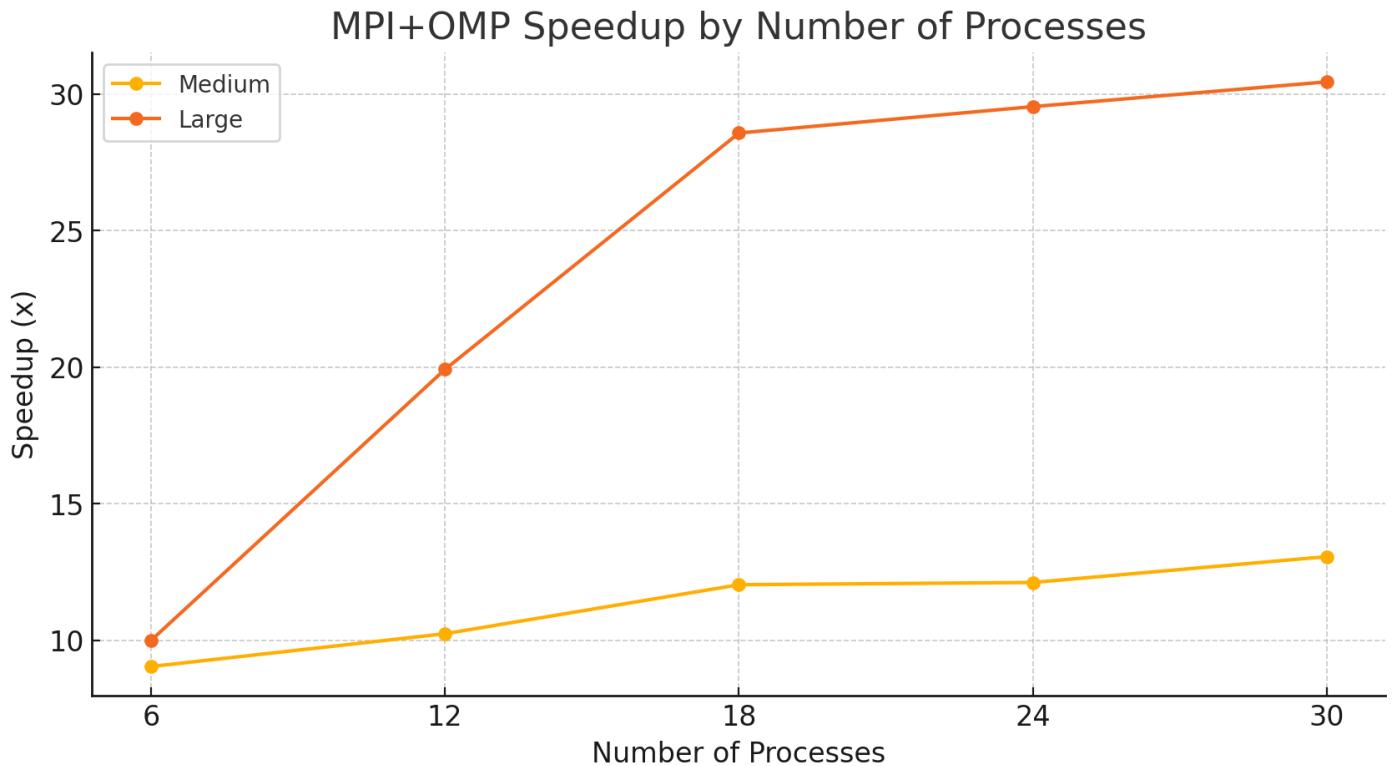


Figure 5 : Comparaison des accélérations obtenues avec OMP et MPI

#### 4.3 Discussion des résultats

Les résultats montrent que:

- La version OpenMP offre une accélération **quasi-linéaire**
- L'approche hybride MPI+OpenMP permet d'obtenir les meilleures performances

Les temps de référence pour l'exécution séquentielle sont :

```
SEQ_EXEC_TIME_LARGE 291.706 sec
SEQ_EXEC_TIME_MEDIUM 16.745 sec
```

## 5. Conclusion

Ce projet a démontré l'efficacité de la parallélisation hybride MPI+OpenMP pour l'algorithme de Darboux. Les principales réalisations incluent :

- Une implémentation robuste et efficace
- Des gains de performance significatifs sur les grands jeux de données
- Une bonne scalabilité jusqu'à un certain nombre de processus