Rapport Projet HPC :
Le modèle shallow water

Inès Benzenatti
Fadwa Alozade

Encadrants :
Constantinos Makassikis
Salli Moustafa

Polytech Sorbonne - MAIN4

20 mai 2018

Table des matières

1 Introduction

1.1 Le modèle shallow water

Le modèle *shallow water* permet de décrire l'écoulement d'un fluide homogène sur la verticale comme les ondes qui se forment à la surface de l'eau lorsque l'on y jette un caillou. C'est un modèle assez simple à poser mais qui représente des phénomènes très variés et complexes, d'où son importance. Une fois les équations posées, la résolution, elle, est plus difficile. C'est pour cela qu'aujourd'hui, elle se fait de façon numérique, les algorithmes de résolution numérique étant connus.

Dans le cadre de notre projet de MAIN4 du module HPC, un programme séquentiel qui effectue cette résolution nous a été fourni. Le but du projet étant d'accélérer son exécution, nous avons testé plusieurs parallélisations possibles que nous présentons tout au long de ce rapport.

1.2 Ce que fait le code séquentiel

Le programme fourni prend en entrée des données telles que la taille de la matrice, le nombre de pas de temps, etc, et produit en sortie une vidéo constituée d'une image par pas de temps. On y voit la propagation des ondes circulaires d'un fluide.

Le code séquentiel est divisé en plusieurs fonctions dont 5 principales qui sont appelées dans cet ordre :

1. **parse args :** permet de récupérer les arguments que l'utilisateur transmet en entrée, tels que la taille de l'image, le nombre de pas de temps, le fichier d'exportation.
2. **alloc :** alloue de la mémoire pour les 6 grilles qu'utilise le modèle en initialisant leurs valeurs à 0.
3. **gauss init :** initialise la grille hFil au temps $t = 0$.
4. **forward :** remplit les grilles à chaque pas de temps à partir de $t = 1$ et les exporte si l'utilisateur a précisé de le faire. C'est la fonction qui contient la plus grande partie des calculs.
5. **dealloc :** désalloue la mémoire de toutes les grilles.

2 La parallélisation avec MPI

Pour les tests, nous avons utilisé les ordinateurs de la salle 327 de Polytech dans le bâtiment Esclangon. L'exécution des programmes ne s'étant pas faite le même jour ni sur les mêmes machines pour toutes les parties, le temps en séquentiel est différent d'une partie à l'autre.

2.1 Ce que nous parallélisons

En utilisant MPI, nous avons fait le choix de paralléliser certaines des fonctions sur plusieurs processeurs :

- **parse args :** tous les processeurs font appel à cette fonction, ainsi ils possèdent tous toutes les données d'entrée.
- **alloc :** seul le processeur 0 alloue le tableau global hFil. Par ailleurs, tous les processeurs allouent des tableaux locaux pour les 6 grilles.
- **gauss init :** nous avons fait le choix d'attribuer l'initialisation au processeur 0 seulement. Ainsi, il remplit le tableau global hFil et le distribue aux autres processeurs grâce à une fonction Scatter.
- **forward :** tous les processeurs font les calculs de la fonction forward sur leurs portions de l'image en se communiquant les données voisines dont ils ont besoin. Ils envoient ensuite leurs portions de l'image hFil au processeur 0 qui les recolle dans l'image hFil globale.
  Si exportation il y a, seul le processeur 0 se charge d'exporter le hFil global. Dans la version avec MPI-IO (section 2.4), tous les processeurs contribuent à l'exportation en écrivant dans le fichier de sortie.
- **dealloc :** le processeur 0 désalloue le tableau de l'image globale et tous les processeurs désallouent les tableaux locaux.

2.2 Parallélisation par bandes

Pour la parallélisation par bandes, nous avons découpé l'image en autant de bandes que de processeurs.

Pour effectuer les calculs du forward, chaque processeur a besoin de données que ses voisins possèdent. Ainsi, lors de l'allocation des tableaux locaux, la taille des tableaux n'est pas égale à la taille de l'image divisée par le nombre de processeurs. En effet, on alloue une ou deux lignes supplémentaires pour accueillir les données des voisins.

La figure 1 est une image de dimension 12x12. On peut voir sa séparation par bandes sur 4 processeurs sur la figure 2. Les lignes en jaunes sont les lignes allouées par le processeur courant et qui contiendront les données des voisins. Ces données seront utilisées pour le calcul mais ne seront pas mises à jour.

Pour cet échange de données, nous avons choisi une communication standard bloquante avec la fonction SendRecv.

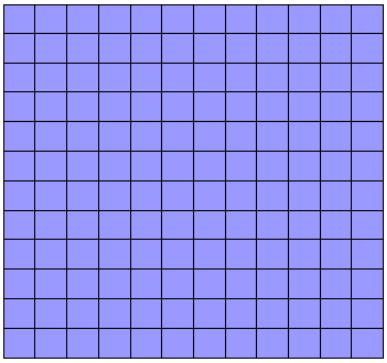


FIGURE 1 – Image 12x12

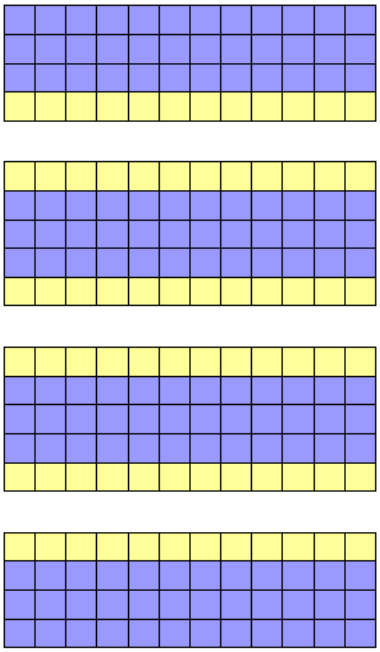


FIGURE 2 – Séparation par bandes

2.2.1 Les performances

Pour les valeurs x = 8192 , y = 8192 et t = 20 :

Nb Procs	*Tmps Séquentiel*	*Tmps parallèle*	*Speedup*	*Efficacité*
1	433.487s	633.763s	0.68	0.68
2	433.487s	659.48s	0.66	0.33
4	433.487s	307.736s	1.41	0.35
8	433.487s	163.113s	2.66	0.33
16	433.487s	109.997s	3.94	0.25

Conclusion :

Le cas le plus efficace est celui avec 4 processeurs.

2.3 Parallélisation par bandes avec recouvrement

Pour gagner encore plus de temps sur l'exécution du programme, nous avons utilisé des communication synchrones non bloquantes pour recouvrir les communications par le calcul en utilisant des fonction Issend et Irecv.

2.3.1 Les performances

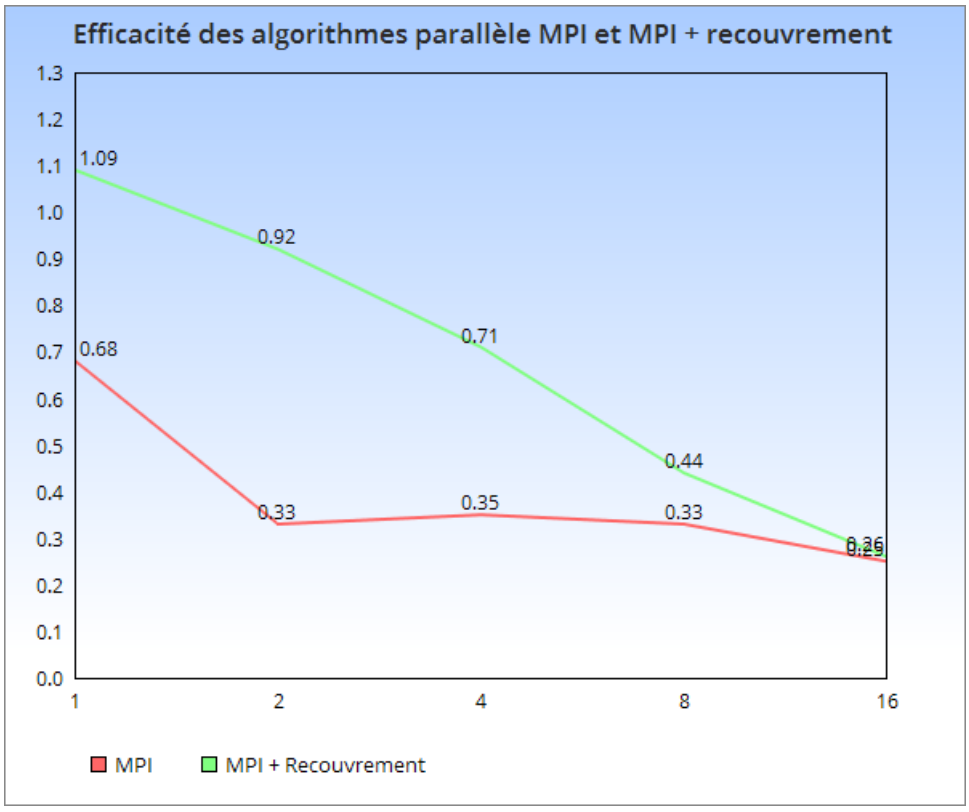
Pour les valeurs x = 8192 , y = 8192 et t = 20 :

Nb Procs	*Tmps Séquentiel*	*Tmps parallèle*	*Speedup*	*Efficacité*
1	433.487 s	397.179 s	1.09	1.09
2	433.487 s	234.326 s	1.85	0.92
4	433.487 s	153.293 s	2.83	0.71
8	433.487 s	124.12 s	3.49	0.44
16	433.487 s	106.09 s	4.09	0.26

Conclusion :

Le cas le plus efficace est observé avec 2 processeurs.

Comparaison des efficacités :



Conclusion :

Nous observons que les efficacités du programme avec recouvrement des calculs sont meilleures que celles sans recouvrement.

2.4 Parallélisation par bandes avec MPI-IO

MPI-IO permet de simuler des entrées et sorties de données parallèles. Nous l'avons utilisé pour permettre à tous les processeurs d'écrire sur un même fichier de sortie pour l'exportation de l'image.

Pour cela, nous avons repris le code de la parallélisation par bandes et l'avons modifié pour que le processeur 0 ne soit plus le seul à appeler les fonctions export. Aussi, nous avons supprimé le gather pour que le processeur 0 ne récupère plus l'image globale à chaque étape du calcul.

2.4.1 Les performances

Pour les valeurs x = 512 , y = 512, t = 40 et 4 processeurs :

Tmps Séquentiel	*Tmps parallèle bandes*	*Tmps parallèle MPI-IO*
2.49 s	3.23 s	2.8 s

Conclusion :

Le temps avec MPI-IO est meilleur que celui avec MPI seul car la phase de mise en commun est inexistante pour MPI-IO, ainsi tous les processeurs procèdent à l'exportation de leurs résultats.

2.5 Parallélisation par blocs

Pour la parallélisation par blocs, nous avons découpé l'image en autant de blocs que de processeurs.

Comme au découpage par bandes, pour effectuer les calculs du forward, chaque processeur a besoin de données que ses voisins possèdent. Ainsi, lors de l'allocation des tableaux locaux, la taille des tableaux n'est pas égale à la taille de l'image divisée par le nombre de processeurs mais on alloue des lignes et des colonnes supplémentaires pour accueillir les données des voisins.

La figure 3 est une image de dimension 12x12. On peut voir sa séparation par blocs sur 16 processeurs sur la figure 4. Les lignes et colonnes en jaunes sont les lignes et colonnes allouées par le processeur courant et qui contiendront les données de ses voisins. Ces données seront utilisées pour le calcul mais ne seront pas mises à jour.

Pour cet échange de données, nous avons choisi de garder une communication standard bloquante avec la fonction SendRecv.

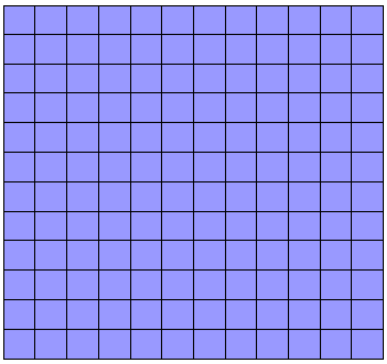


FIGURE 3 – Image 12x12

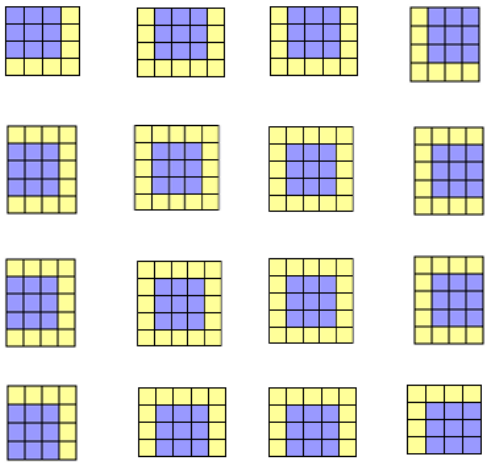


FIGURE 4 – Séparation par blocs

La parallélisation par blocs devrait en théorie être plus efficace que la parallélisation par bandes car les tailles des données échangées sont plus petites.

Malheureusement, nous n'avons pas pu l'observer car, même si notre programme compile et s'exécute sans erreur, la vidéo produite lors de l'exportation ne correspond pas à la vidéo attendue.

3 La parallélisation hybride MPI+OpenMP

3.1 Optimisation des accès mémoire

Avec le code que nous avions jusque là avec MPI, nous avons demandé à l'outil Valgrind de faire un état des lieux des accès mémoire. Le taux de miss dans le cache L1 était alors de 13%.

Nous avons ensuite relu le code pour essayer d'optimiser ses accès mémoire par rapport à l'architecture des machines. La partie qui prend le plus de temps à s'exécuter étant la double boucle du forward qui remplit les tableaux de données, nous avons analysé les accès mémoire de cette partie spécifiquement.

Jusqu'ici, les itérations se faisaient d'abord sur les indices j des colonnes, ensuite, pour chaque colonne, sur l'indice i des lignes. Or, le tableau est stocké en mémoire de telle sorte que les lignes soient mises l'une à la suite de l'autre.
Étant donné que le cache ne peut pas contenir toute l'image, lorsque nous accédons à une valeur, il récupère toutes les valeurs adjacentes, donc des bouts de la même ligne. Dans l'état où était notre code, il devait à chaque itération aller chercher un bout de la ligne suivante dans la mémoire pour le charger dans le cache.

La solution que nous avons appliquée est de simplement inverser les deux boucles pour que les itérations se fassent d'abord sur les lignes et ensuite, pour chaque ligne, sur les colonnes.
Notre code actuel va donc chercher un bout de la ligne en cours lors la première itération pour le mettre dans le cache, puis à la suivante, les valeurs dont il a besoin seront déjà dans le cache. Cela constitue un gain de temps et de performances considérable.

Pour nous assurer de l'efficacité de cette mesure prise, nous avons redemandé à Valgrind un état des lieux des accès mémoire et, cette fois, nous n'avions plus que $1,9\%$ de cache miss.

3.2 La parallélisation avec OpenMP

Afin de paralléliser à l'aide des threads, nous avons tout d'abord identifié les boucles FOR ainsi que les tâches pouvant s'exécuter en parallèle.

Dans le fichier forward.c, nous avons utilisé les directives :

pragma omp parallel for schedule(static) firstprivate(t)
— Schedule(static) : car toutes les opérations à chaque itération ont le même temps d'exécution.
— firstprivate(t) : pour que les différents threads s'exécutent à chaque pas de temps.

pragma omp parallel ... pragma omp single ... pragma omp task Ces directives sont introduites pour les communications, afin que chaque communication soit exécutée par un seul thread.

Dans le fichier init.c, nous avons utilisé les directives :

pragma omp parallel for schedule(static)
— Schedule(static) : car, là aussi, toutes les opérations à chaque itération ont le même temps d'exécution.

3.2.1 Les performances

Pour les valeurs x = 8192 , y = 8192 et t = 20 :

Nb Procs	*Tmps Séquentiel*	*MPI*	*OpenMP 2TH*	*OpenMP 4TH*
4proc / 2th+2proc / 4th+1proc	276.456 s	137.23 s	70.436 s	42.795 s
8 / 2th+4proc / 4th+2proc	276.456 s	110.257 s	79.437 s	68.815 s
16 / 2th+8proc / 4th+4proc	276.456 s	108.186 s	76.926 s	76.946 s

2 Threads :

Nb procs	*Speedup*	*Efficacité*
2	3.92	0.98
4	3.48	0.43
8	3.59	0.22

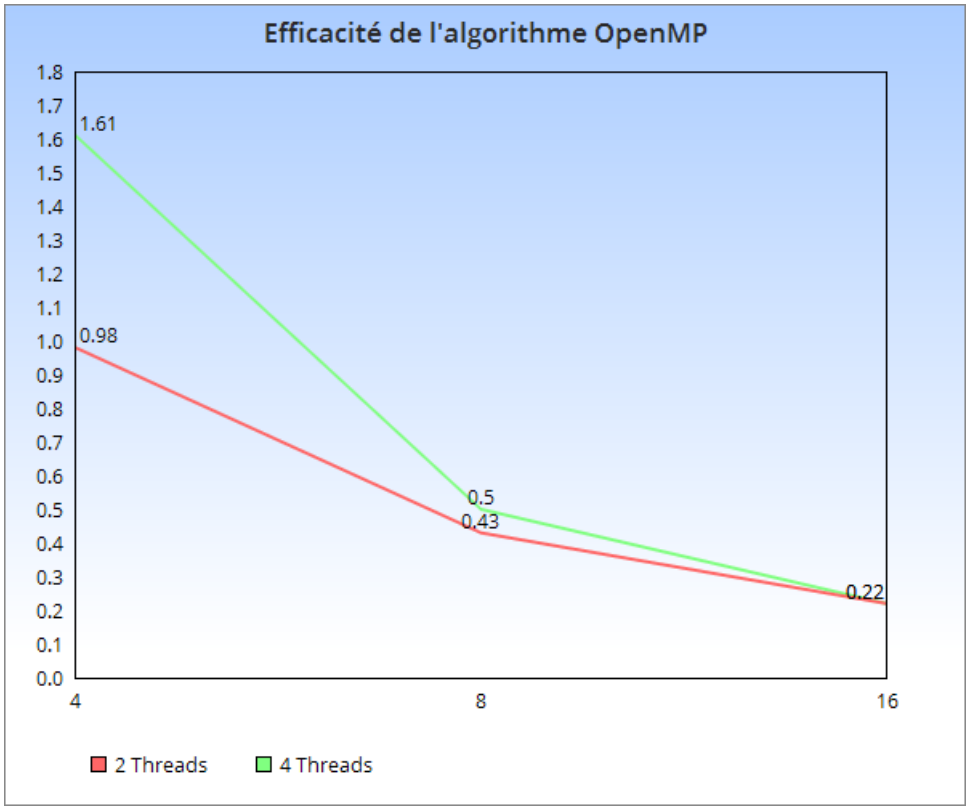
4 Threads :

Nb procs	*Speedup*	*Efficacité*
1	6.46	1.61
2	4.01	0.50
4	3.59	0.22

Conclusion :

Combiner MPI et OpenMP améliore grandement le temps d'exécution du programme. En utilisant 2 threads, la meilleure efficacité est observée en utilisant 2 processeurs, et en utilisant 4 threads, la meilleure efficacité est observée pour 1 processeur. Parmi les deux, il est nettement plus efficace d'utiliser 4 threads sur un processeur plutôt que 2 threads sur deux processeurs. Cela s'explique par la suppression du temps de communications MPI entre les processeurs.

Comparaison des efficacités :



4 La vectorisation

4.1 Ce que nous vectorisons

La vectorisation SIMD (Single Instruction Multiple Data) permet d'effectuer des calculs sur des vecteurs de données. Étant donné que nous travaillons avec des nombres flottants en double précision, nous avons choisi une vectorisation à 256bits, ce qui représente 4 données de 8 octets traitées simultanément.

Puisque les ordinateurs à notre disposition ne disposait pas de la version gcc qui supporte OpenMP4.0, nous avons effectué une vectorisation avec AVX2.

Nous l'avons d'abord appliquée au code séquentiel, puis nous l'avons transposée à nos codes parallèles précédemment décrits.

4.2 La vectorisation du programme séquentiel

Pour vectoriser le code séquentiel, nous avons changé notre manière d'allouer les tableaux des grilles et ce, en utilisant la fonction posix_memalign pour qu'elle aligne les données selon des paquets de 32 octets.

Ensuite, nous avons redéfini les fonctions qui calculent les pixels de chaque grille pour qu'elles renvoient des vecteurs alignées de type *inline* $__mm256d$. Nous y avons fait appel sur toutes les lignes de l'image mais seulement une fois toutes les 4 colonnes. Une fois le vecteur récupéré, nous faisions un stockage de ses données dans la grille correspondante.

4.2.1 Les performances

Pour les valeurs x = 8192 , y = 8192 et t = 20 :

Tmps Séquentiel	*Tmps SIMD*
281.526	27.873

Conclusion :

La vectorisation montre une réduction du temps d'exécution très importante.

4.3 La vectorisation du programme parallèle avec MPI

Pour vectoriser le programme parallèle, nous avons récupéré notre code de parallélisation par bandes avec MPI (section 2.2) et y avons apporté les mêmes modifications que pour vectoriser le code séquentiel.

4.3.1 Les performances

Pour les valeurs x = 8192 , y = 8192 et t = 20 :

Nb Procs	*Tmps Séquentiel SIMD*	*Tmps parallèle*	*Speedup*	*Efficacité*
1	27.873 s	183.044 s	0.15	0.15
2	27.873 s	59.768 s	0.47	0.23
4	27.873 s	76.529 s	0.36	0.09
8	27.873 s	75.477 s	0.37	0.04
16	27.873 s	75.547 s	0.37	0.02

4.4 La vectorisation du programme parallèle avec MPI et OpenMP

Pour vectoriser le programme parallèle MPI-OpenMP, nous avons récupéré notre code de parallélisation avec MPI et OpenMP (section 3.2) et y avons apporté les mêmes modifications que pour vectoriser le code séquentiel.

4.4.1 Les performances

Pour les valeurs x = 8192 , y = 8192 et t = 20 et 4 Threads :

Nb Procs	*Tmps Séquentiel SIMD*	*Tmps parallèle*	*Speedup*	*Efficacité*
1 + 4TH	27.873 s	242.393 s	0.11	0.02
2 + 4TH	27.873 s	57.07 s	0.49	0.06
4 + 4TH	27.873 s	74.327 s	0.37	0.023

Conclusion :

En utilisant la vectorisation, le temps en séquentiel est plus rapide qu'avec la parallélisation MPI ou MPI combiné à OpenMP, cela est peut-être dû au fait que les communications sont longues à cause de la taille des messages qui est probablement plus importante en utilisant des vecteurs.

4.5 La vectorisation du programme parallèle avec MPI-IO

Pour vectoriser le programme parallèle, nous avons récupéré notre code de parallélisation avec MPI-IO (section 2.4) et y avons apporté les mêmes modifications que pour vectoriser le code séquentiel.

4.5.1 Les performances

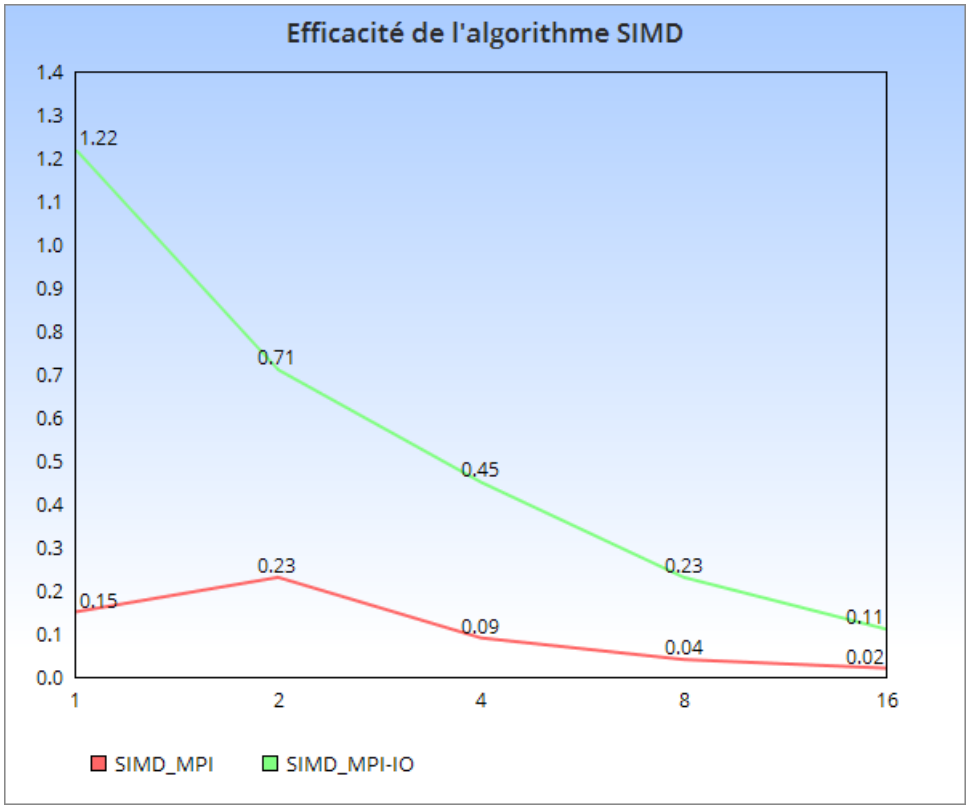
Pour les valeurs x = 8192 , y = 8192 et t = 20 :

Nb Procs	*Tmps Séquentiel SIMD*	*Tmps parallèle*	*Speedup*	*Efficacité*
1	27.873 s	22.74 s	1.22	1.22
2	27.873 s	19.58 s	1.42	0.71
4	27.873 s	15.56 s	1.79	0.45
8	27.873 s	15.29 s	1.82	0.23
16	27.873 s	15.13 s	1.84	0.11

Conclusion :

Nous observons une diminution du temps d'exécution avec l'utilisation de MPI-IO par rapport au séquentiel. Nous pouvons en conclure que lorsque le nombre de communication diminue, le temps d'exécution est nettement amélioré.

Comparaison des efficacités :



5 Conclusion

À l'issu de ce projet nous avons consolidé nos aquis sur de nombreuses méthodes pour la parallélisation et l'optimisation du temps d'exécution. Nous nous sommes aperçu que le temps d'exécution ne diminue pas toujours quand on augmente le nombre de processeurs, cela est dû au fait que plus il y a de processeurs plus le nombre de communications augmente ce qui ralentit le temps d'exécution.

Pour la première partie du projet nous avons constaté que le recouvrement des communications par le calcul améliore l'efficacité et après plusieurs recherches nous avons aussi trouvé que la décomposition par bloc est plus efficace car la taille des messages diminue bien que leur nombre augmente, ce qui crée des communications très rapides.

Lors de la deuxième partie, la parallélisation avec MPI et OpenMP a montré une grande réduction du temps d'exécution. La vectorisation s'est aussi avérée très efficace en séquentiel ainsi qu'avec le code parallèle MPI-IO, cependant elle n'a pas été efficace avec l'utilisation de MPI seul ou de MPI combiné à OpenMP. La différence entre les codes précités est l'absence du Gather pour MPI-IO, nous en concluons que le fait de devoir mettre tout en commun à chaque itération ralentit considérablement l'exécution du programme.