

# **PROJET HPC**

## **Le modèle Shallow water**

**Diab Nabil**

# PARTIE 1 : Parallélisation MPI

## Remarque :

Tous les tests et analyse de performances de cette partie ont été fait avec les dimensions suivantes :

-x 8196 -y 8196 -t 20

## 1 – Choix de décomposition

N'ayant pas eu le temps d'implémenter les deux solutions de décomposition, j'ai décidé d'implémenter uniquement la **décomposition par bandes**.

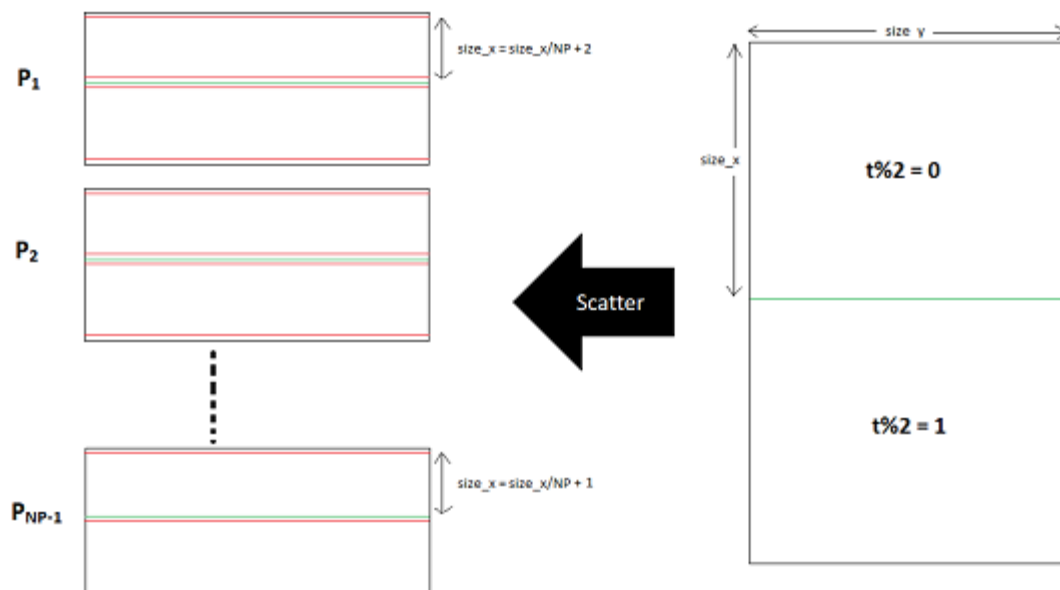
En plus d'être beaucoup plus rapide et simple à implémenter, la décomposition par bande représentera au fil de l'implémentation de ce projet beaucoup plus d'avantages que celle par bloc car les données sur le projet global sont des données contiguës qui sera favorable au partage des ressources et sur la vectorisation par exemple.

Cette décomposition est aussi plus avantageuse sur le matériel utilisé car chaque processus communique avec au plus deux autres processus contre 8 dans la décomposition par bloc, ce qui implique que nous avons une chance sur deux de communiquer avec uniquement des processus se trouvant sur la même machine, une chance qui est nulle dans le cas d'une décomposition par bloc.

Pour **Recouvrir les communications de calculs**, il suffisait de remplacer les communications bloquantes en non bloquantes, ce qui a pu nous donner un gain d'environ **10%** sur le test de référence.

## 2- Structure de données

Voici la structure de données utilisé lors de la première partie :

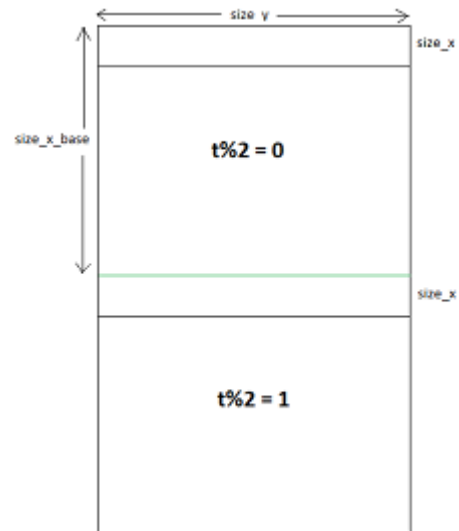


Les tableaux sont initialement chargés dans le processus 0 et ensuite un scatter est réalisé afin de partager les ressources à tous les processus.

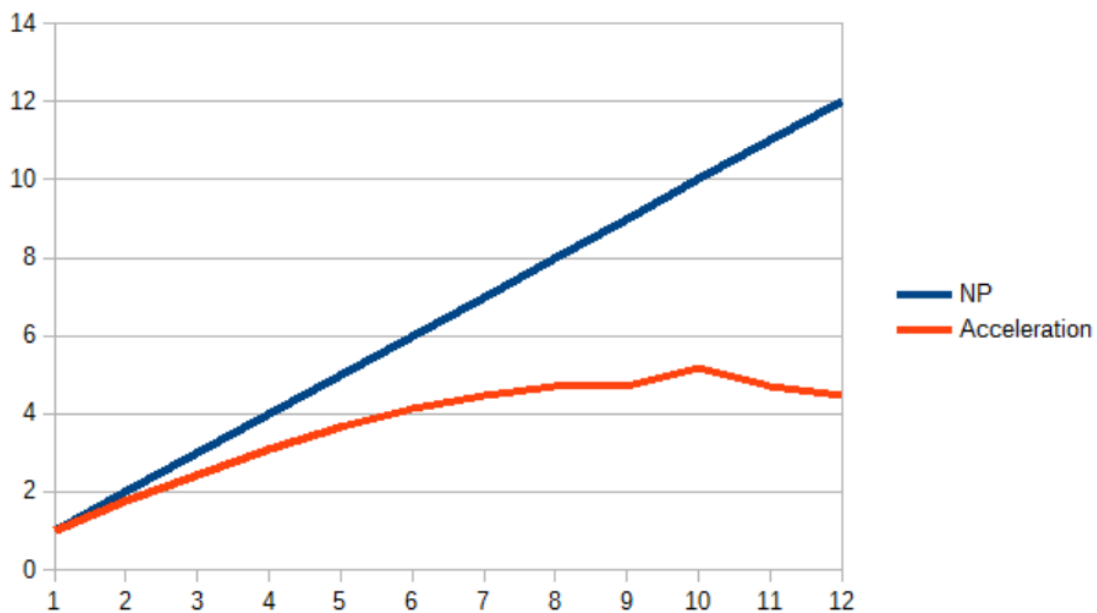
Quant au processus 0 dont la représentation ci-contre a une structure différente de tous les autres processus, c'est dans celui-ci que sera initialisé toutes les données et servira aussi à récupérer les données dans le cas d'un export.

Cette structure impliquera que tous les processus ne sont pas identiques et rendra les indices de partages de tableau bien plus complexes.

Pour ce qui est de l'exportation des données, dans le cas de la première partie, je n'ai pas utilisé MPI-IO pour exporter mes résultats. Mais un gather fait à chaque t dont la racine sera le processus 0. Ceci ne sera pas retenu comme une solution d'export car le temps augmente exponentiellement lors de l'export dans ce cas.

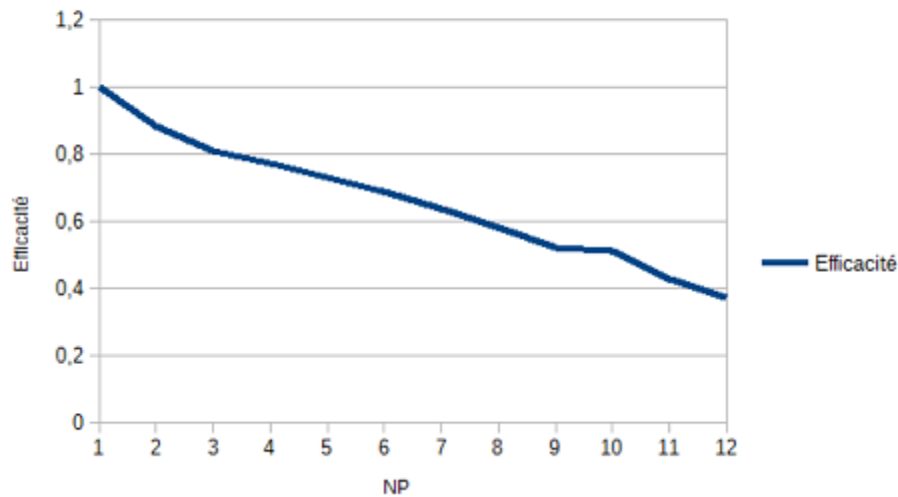


### 3- Résultats obtenus



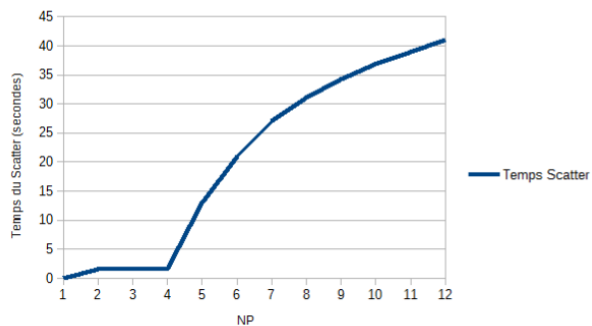
Les résultats obtenus seront très peu satisfaisants car nous ne dépasserons pas la barre d'une accélération x6 malgré le test lancé jusqu'à 12 processus et nous pouvons aussi remarquer une décélération et une grande baisse d'efficacité.

# RESULTATS OBTENUS

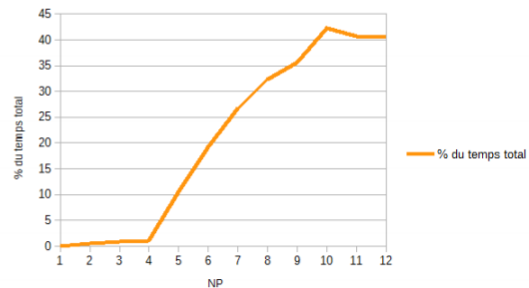


## 4 – Explication des résultats

### Temps du scatter



### Proportionnalité par rapport au temps total



Ces mauvais résultats s'expliquent par l'utilisation du scatter pour distribuer les ressources en début d'exécution, nous pouvons voir sur le graphique de gauche le temps en secondes que prend le scatter en fonction du nombre de processus et sur la droite la proportion (en pourcent) de temps passé dans le scatter par rapport au temps total de l'exécution et nous pouvons remarquer que cela peut aller jusqu'à plus de 40% ce qui est énorme.

Nous pouvons néanmoins qu'avec moins de 4 processus, le temps du scatter est correct. Cela s'explique par le fait que nous avons des machines avec des CPU quad core et que donc le partages de données se fait sur la même machine et il n'y a aucune communication à l'extérieur de cette machine. Nous pourrions donc considérer que le scatter peut être une solution dans le cas d'un calcul sur une seule machine.

# PARTIE 2 : MPI, OpenMP et SIMD

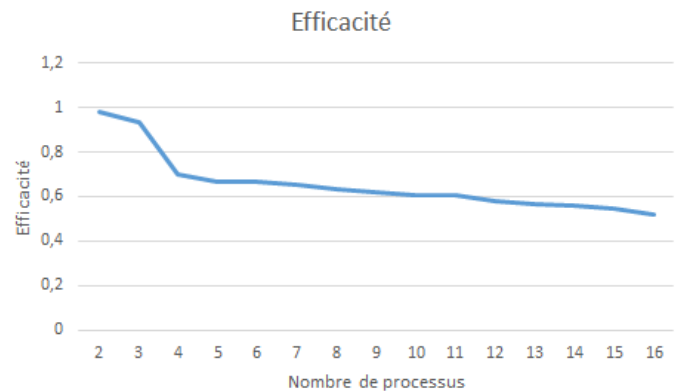
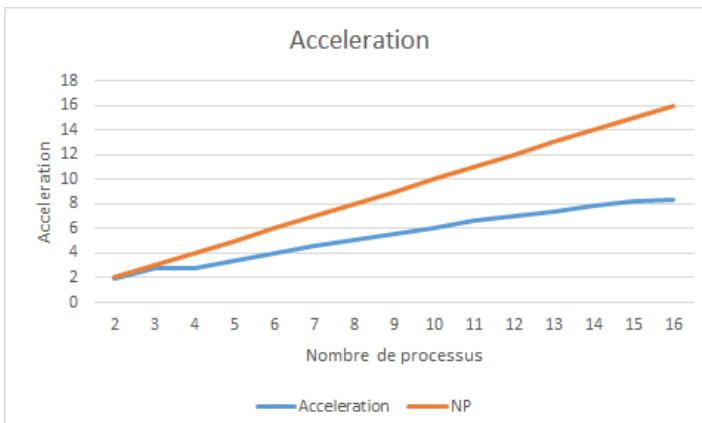
Remarque :

Tous les tests et analyse de performances de cette partie ont été fait avec les dimensions suivantes :

-x 8196 -y 8196 -t 100 dans la salle 14-15-403

## 1 – Ce qu’il fallait changer de la partie 1

Dans un premier temps, il fallait impérativement remplacer le scatter par une autre solution. Pour cela j’ai parallélisé la fonction gauss\_init et alloc. Désormais chaque processus est indépendant dès l’allocation de l’espace mémoire, le processus 0 a récupéré la même structure que tous les autres processus, ce qui a énormément simplifié le code. Après ces modifications les résultats se sont montrés bien plus satisfaisants :




L’efficacité obtenue n’est pas très satisfaisante mais nous pouvons remarquer que l’accélération ne décroît jamais en augmentant le nombre de processus, contrairement à notre précédente architecture. C’est à partir du code donnant ces résultats que nous allons travailler sur la seconde partie.

## 2 – Prise en compte de la hiérarchie mémoire

Après analyse des accès mémoire dans le fichier shalw.h et la boucle dans la fonction forward(), nous pouvons remarquer que nos accès ne sont pas contiguës, nous ne parcourons pas correctement notre tableau. Pour résoudre ce problème il suffisait d’inverser les boucles de cette manière :

```
for (int j = 0; j < size_y; j++) {  
    for (int i = 0; i < size_x; i++) {  
        HPHY(t, i, j) = hPhy_forward(t, i, j);  
        UPHY(t, i, j) = uPhy_forward(t, i, j);  
        VPHY(t, i, j) = vPhy_forward(t, i, j);  
        HFIL(t, i, j) = hFil_forward(t, i, j);  
        UFIL(t, i, j) = uFil_forward(t, i, j);  
        VFIL(t, i, j) = vFil_forward(t, i, j);  
    }  
}
```



```
for (int i = 0; i < size_x; i++) {  
    for (int j = 0; j < size_y; j++) {  
        HPHY(t, i, j) = hPhy_forward(t, i, j);  
        UPHY(t, i, j) = uPhy_forward(t, i, j);  
        VPHY(t, i, j) = vPhy_forward(t, i, j);  
        HFIL(t, i, j) = hFil_forward(t, i, j);  
        UFIL(t, i, j) = uFil_forward(t, i, j);  
        VFIL(t, i, j) = vFil_forward(t, i, j);  
    }  
}
```

Le léger changement nous permettra de gagner en moyenne **80%** de vitesse !

## 3 – Parallélisation OpenMP et SIMD

### 1 – OpenMP

Après une série de tests, j'ai choisi de diviser chaque processus en deux threads au niveau de la boucle de calcul. Je pense les meilleures performances avec ce nombre de threads sont dû aux types de processeurs installés sur les machines à notre disposition.

En effet, chaque processeur détient 4 cœurs physiques et 8 cœurs logiques, sachant que MPI se distribue sur les 4 cœurs physiques, il reste donc 4 autres cœurs logiques libres, ce qui permet à chacun de nos processus de se doubler en deux threads, si plus en sont créés, alors des chargements et écritures des états des registres des threads se font ce qui cause un ralentissement de notre exécution.

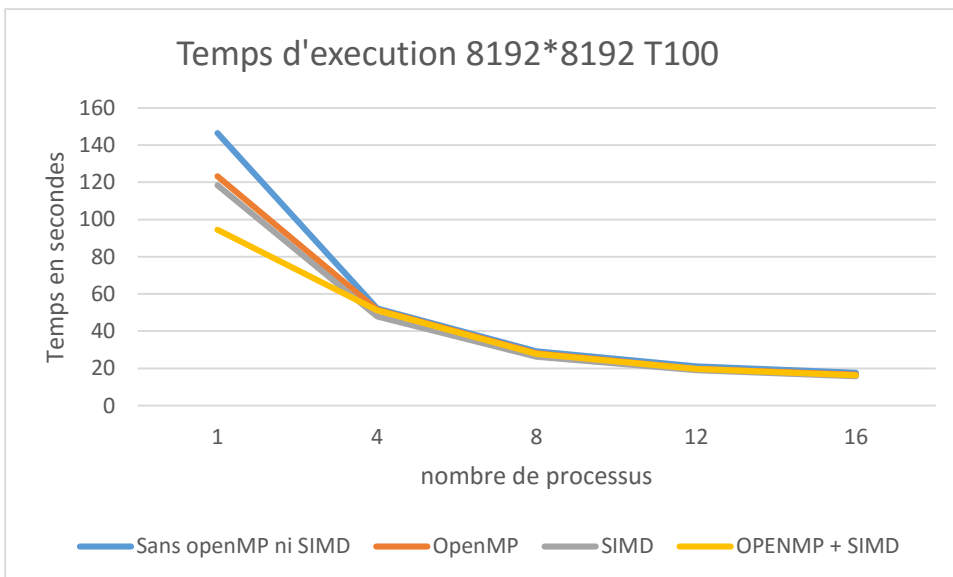
### 2 - SIMD

L'implémentation d'une vectorisation par SIMD a demandé beaucoup de travail et de modification de codes, c'est en fait toutes les fonctions du fichier forward.c qui ont été modifiées.

Ayant fait le choix de travailler avec des intrinsèques, j'ai pu également modifier certains calculs par des instructions de type fma pour essayer d'avoir les meilleures performances possibles.

Vous retrouverez ci-joint en annexe une brève explication sur la vectorisation du code et l'utilisation de fma pour le calcul.

## 4- Résultats

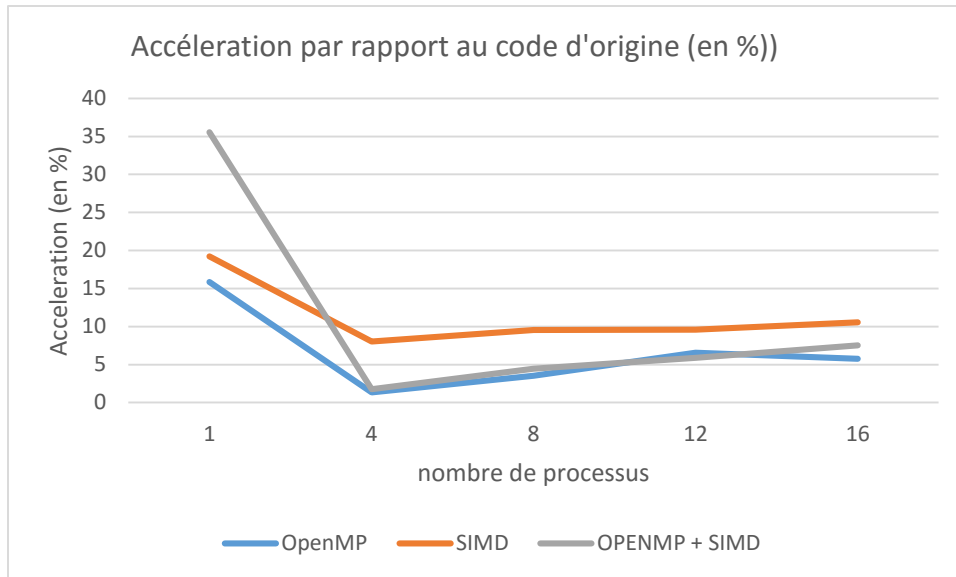


Nous pouvons remarquer sur les résultats obtenus ci-contre, que lors d'une exécution séquentielle, les parallélisations avec open MP et SIMD se démarquent face au code d'origine, et la combinaison des deux est encore plus marquée.

Ensuite en y ajoutant MPI, ces implémentations seront toujours plus performantes mais cet

écart est beaucoup plus faible

Les courbes étant très proches et difficiles à distinguer sur ce dernier graphique, je propose d'examiner ces résultats sous un autre angle :



Ce dernier graphique nous montre plus clairement l'efficacité de chacune des implémentations, nous pouvons y retrouver les très bon résultats en séquentiel de l'union de OpenMP et SIMD, aussi montrer quelle implémentation est la meilleure lorsqu'elle est combinée avec une parallélisations MPI. Dans ce dernier cas, c'est la parallélisations SIMD qui l'emporte.

## 5- Problèmes rencontrés

Deux problèmes assez importants ont été rencontrés lors de l'implémentation de la vectorisation SIMD, les deux portent sur l'alignement de la mémoire :

- Premièrement j'ai dû modifier la fonction d'allocation de mémoire en modifiant les `calloc` par des `_mm_malloc`, pour aligner nos tableaux en mémoire. Ce problème m'a coûté plusieurs jours de réflexion.
- Le second problème est lui aussi dû à l'alignement de la mémoire :
  - Certaines fonctions de `forward()` nécessitent un accès à des éléments qui ne sont pas alignés avec nos vecteurs de 4 éléments, par exemple sur petit bout de code

```
if (j < size_y - 4){
    tmp_tab[0] = VPHY(t - 1, i, j + 1);
    tmp_tab[1] = VPHY(t - 1, i, j + 2);
    tmp_tab[2] = VPHY(t - 1, i, j + 3);
    tmp_tab[3] = VPHY(t - 1, i, j + 4);
    e = _mm256_load_pd(&tmp_tab[0]);
}
```

nous pouvons remarquer que le chargement des éléments en mémoire ne s'est pas fait de façon vectorielle car ce n'était pas aligné et je n'ai toujours pas trouvé de solution vectorielle à ce problème.

## 6 – Conclusion

J'ai finalement décidé de tenter d'améliorer la solution donnant les meilleurs résultats (c'est-à-dire avec la vectorisation SIMD). La seule et meilleure amélioration que j'ai pu trouver est un déroulage de boucle x 4, soit 16 éléments par tableau accédé à chaque itération (car  $4 * 4$  éléments du vecteur) ce qui a donné un gain d'environ 12% sur 16 processus par rapport à la vectorisation SIMD de base, cette solution sera la meilleure performance et calculera en 14 secondes ce que calculait le programmes initial donné au début du projet en plus de 20 minutes en utilisant uniquement 4 machines. Ce qui implique qu'il est environ 100 fois plus rapide.

La remise contient le rapport, une annexe expliquant les grandes lignes de la vectorisation et l'inclusion de fma, le code de la première partie, le code de la partie OpenMP et enfin dans le dossier SIMD le meilleur code contenant openMP et le déroulage de boucle.