Polytech Sorbonne

Projet HPC : Le modèle shallow-water

OpenMP, MPI, AVX

Colette Voisembert, Romaric Kanyamibwa

MAIN4

Année 2017 - 2018

Table des matières

1 Introduction **2**

2 Première partie : Parallélisation MPI **2**
2.1 Décomposition par bande 2
2.2 Décomposition par Blocs 4
2.3 Décomposition par Blocs vs Décomposition Bandes 7
2.4 Mesure de Performance 7
2.5 MPI_IO 9

3 Partie deux **10**
3.1 Hiérarchie mémoire 10

4 Parallélisation MPI+OpenMP **11**

5 AVX **12**

6 AVX+OPENMP+MPI **13**

7 Conclusion **13**

1 Introduction

L'objectif de ce projet est la parallélisation d'un code séquentiel qui modélise les équation de Shallow-Water (S-W). Les équations S-W permettent la représentation de l'écoulement d'un fluide homogène sur la verticale.

En résolvant numériquement les équations de S-W sur une grille de Arakawa-C on arrive à avoir une modélisation qui est parallélisable. En effet, on peut découper la grille pour que différentes machines calculent différentes parties de la grille. De cette façon on arrive à accélérer le temps de calculs de notre programme. La parallélisation n'est pas seulement le fait d'utiliser plusieurs machines, c'est aussi utiliser la même machine pour effectuer le calcul parallèle (SIMD), c'est à dire vectoriser les différentes opérations arithmétiquement effectuées pour profiter de l'architecture de la machine.

Durant ce projet, nous avons exploré trois différents types de parallélisation. La parallélisation multithreads avec OpenMP, la parallélisation multicore avec MPI et la parallélisations SIMD avec AVX. A la fin du projet on compare ces trois méthodes et on fait un programme hybride qui utilise les trois types de parallélisation pour voir si on peut améliorer d'avantage la performance du programme.

Pour tester la performance de nos codes nous avons utilisé principalement la salle 327 du bâtiment Esclagon à Polytech Sorbonne[1].

2 Première partie : Parallélisation MPI

Dans la première partie du projet nous nous somme amener à concevoir 2 algorithmes de parallélisation. Le premier algorithme qu'on implémenté c'était un algorithme de décomposition par bande et le deuxième était une décomposition par bloc.

Ces algorithmes repose sur les faits que nos calculs sont sauvegarder sur des grilles et que on peut découper ces grilles pour effectuer des calculs parallèle.

2.1 Décomposition par bande

La décomposition par bandes consiste à découper en bandes les différents tableaux utilisés pour les calculs de la simulations. Si par exemple, nous avons NP processus et un tableau de largeur $size\_y$ et de hauteur $size\_x$. La décomposition par bandes consiste à découper le tableau en NP sous tableaux de hauteur $\frac{size_x}{NP}$ et de largeur $size_y$ comme illustré sur la figure ci-dessous.

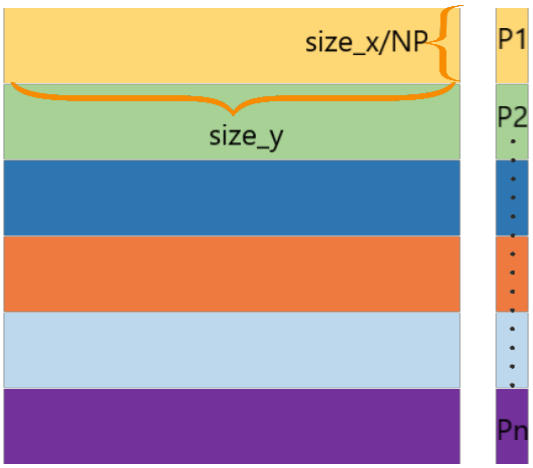

Figure 1 – *Décomposition par bande*

Dans un premier temps, l'algorithme doit initialiser le tableau de chaque processus avec la bonne partie de la grille. C'est à dire qu'il faut donner le bon hfil[2] à chaque processus. Pour $t = 0$ le k^{ieme} processus est initialisé

[1] Les derniers test de performance ont été effectués sur les ordinateurs pc4012, pc4013, pc4014, pc4022, pc4023, pc4015, pc4017 et pc4020 de la salle 327

[2] code de l'initialisation sur le fichier init.c

avec

$$HFIL(0,i,j) = exp\left(\frac{\left(\frac{(i+size_x/NP*my\_rank)*dx-gmx}{gsx}\right)^2}{2}\right) * exp\left(\frac{\left(\frac{j*dy-gmy}{gsy}\right)^2}{2}\right)$$

Après avoir initialisé proprement chaque bande on effectue les calculs comme dans le programme séquentiel. Toutefois, on remarque que dans les différents calculs l'élément de la i^{ieme} ligne et de la j^{ieme} colonne a besoin des éléments des colonnes $j+1$ et $j-1$ et des lignes $i+1$ et $i-1$. Donc pour que le processus P puisse calculer la dernière ligne de sa grille il doit récupérer la première ligne du processus $P+1$. De la même manière, pour calculer sa première ligne, le processus P a besoin de la dernière ligne du processus $P-1$. On voit donc que le processus P doit envoyer et recevoir des données du processus qui le précède et de celui qui le suit.

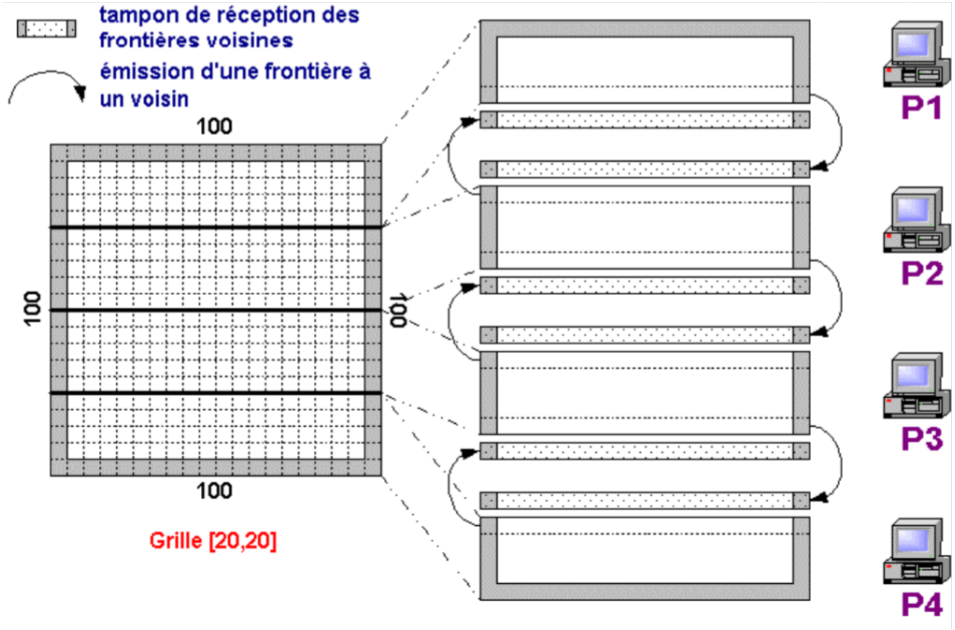

Figure 2 – *Send/Receive nécessaires pour le calcul du schéma*

Nous avons l'algorithme suivant pour la décomposition par bandes.

```
algo shallow_water_bande;
var size_x,size_y;  /* nombre de lignes et de colonnes de l'image */
var size_x_local;/**selon le processus est size_x/NP+1 ou size_x/NP+2*/
var hFil_local, uFil_local, vFil_local; /*taille 2*size_x/NP*size_y*/
var hPhy_local, uPhy_local, vPhy_local; /*taille 2*size_x/NP*size_y*/
var NP;              /*Nombre de processus*/
var my_rank;         /*Rang du processus*/


Si my_rank egal 0 ou egal NP-1
    size_x_local=(h/NP+1)
Sinon
    size_x_local=h/NP+2;
FinSi

allocation de 2*size_x_local*size_y memoire pour les tableaux

Pour i=0...,size_x_local-1
    Pour j = 0,...,size_y-1

    Si i<size_x/NP+1
        tmp=(size_x/NP*my_rank)
    Finsi

    HFIL_LOCAL(0, i, j) = height *
    (exp(- pow(((i+tmp) * dx - gmx) / gsx, 2) / 2.)) *
```

```
    (exp(- pow((j * dy - gmy) / gsy, 2) / 2.)) ;
    Finpour
Finpour


/* forward */
/*on note local_data les tableaux hFil_local, uFil_local, vFil_local,
hPhy_local, uPhy_local, vPhy_local*/
Pour i=0,...,nbiter

    calcul de hFil_local, uFil_local,
    vFil_local,hPhy_local, uPhy_local, vPhy_local

    Si my_rank different de 0

        Envoie de la 1ere ligne local_data+size_y au processus my_rank-1

        Reception de la derniere ligne de my_rank-1 dans au tableau

    FinSi

    Si my_rank different de NP-1

        Reception de la 1ere ligne de my_rank+1
        dans local_data+(size_x_local-1)*size_y

        Envoie de la derniere ligne local_data+(h_local-2)*w
        au processus  my_rank+1

    FinSi

    (Gather) Rassemblement de tous les hfil au prossesus 0
    /*export au fichier*/
    Si my_rank egal 0

        export(shalw_<size_y>x<size_y>_T<NP>.sav,HFILL);

    FinSi

Finpour
```

2.2 Décomposition par Blocs

La décomposition par blocs consiste à découper en blocs rectangulaires les matrices utilisées lors de la simulation. Cette fois on découpe un tableau selon $size_x$ mais aussi selon $size_y$. Notre algorithme peut prendre en entrée des matrices dont le nombre de lignes et le nombre de colonnes sont chacun des puissances de 2. Et le nombre NP de processeurs doit également être une puissance de 2.

En effet, voici comment nous avons procédé pour découper notre matrice. Si nous avons P processeurs, tant que $P > 1$ nous divisons tour à tour $size_x$ et $size_y$ par 2; et nous divisons P par 2.

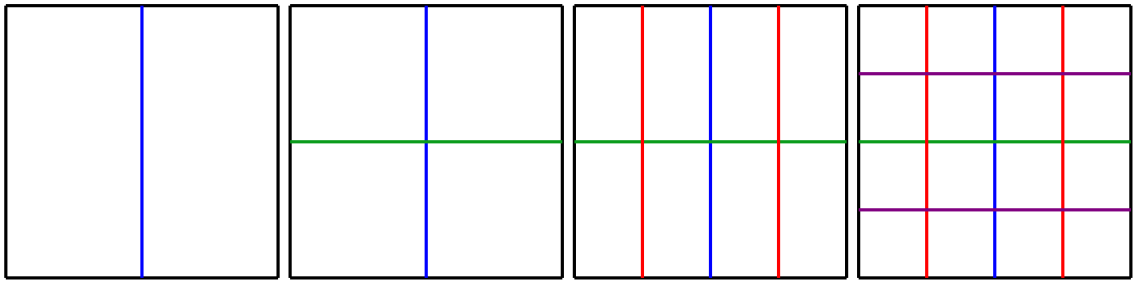

Figure 3 – *Découpage par bloc pour N = 2,4,8,16*

Après avoir initialisé proprement chaque bloc, on effectue les calculs comme dans le programme séquentiel. Toutefois, comme pour la décomposition par bandes, on remarque que certain blocs ont besoin d'informations appartenant à des blocs voisins.

On remarque que dans les différents calculs, l'élément de la i^{ieme} ligne et de la j^{ieme} colonne (noté e_{i_j} a besoin des éléments des colonnes $j+1$ et $j-1$ et des lignes $i+1$ et $i-1$. Donc pour que le processus P puisse calculer la dernière ligne de sa grille il doit récupérer la première ligne du processus $P + Nb\_Col\_P$ (avec Nb_Col_P qui est le nombre de colonnes de processus, 3 dans l'exemple ci-dessous). Le processus P a besoin de la dernière ligne du processus $P - Nb\_Col\_P$, pour calculer sa première ligne. Il en va de même pour les colonnes. Le processus P a besoin la colonne de droite du processus $P - Nb\_Col\_P$ pour calculer sa colonne de gauche et de la colonne de gauche du processus $NP + Nb\_Col\_P$, pour calculer sa colonne de droite. On remarque également, qu'en théorie, le processus P a besoin de quatre données supplémentaires pour finir de compléter le carré. Ainsi le processus P a besoin du coin en bas à droite du processus $P - Nb\_Col\_P - 1$. Il en va de même pour les trois autres coins (voir le schéma ci-dessous si besoin). Cependant dans cet exemple de modèle Shallow-Water, ces données dans les coins ne sont pas nécessaires, ainsi nous ne les échangerons pas. Pour conclure, on voit donc que le processus P doit envoyer et recevoir des données du processus qui le précède, de celui qui le suit, celui à sa droite et celui à sa gauche.

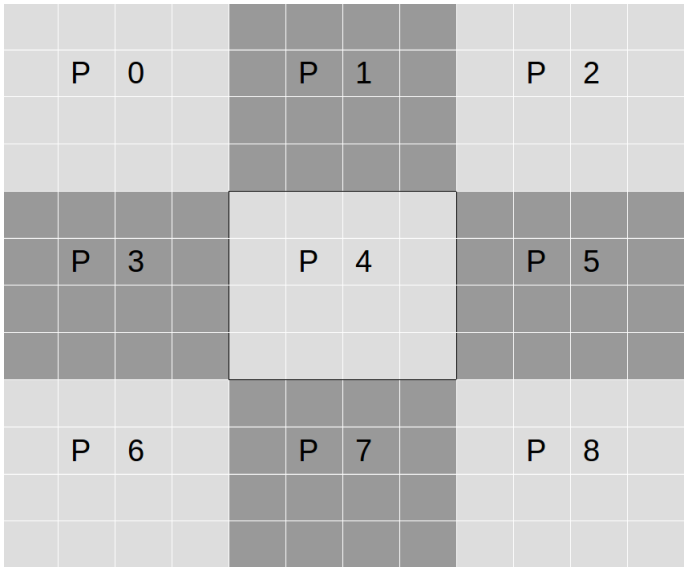

Figure 4 – *Décomposition par blocs*

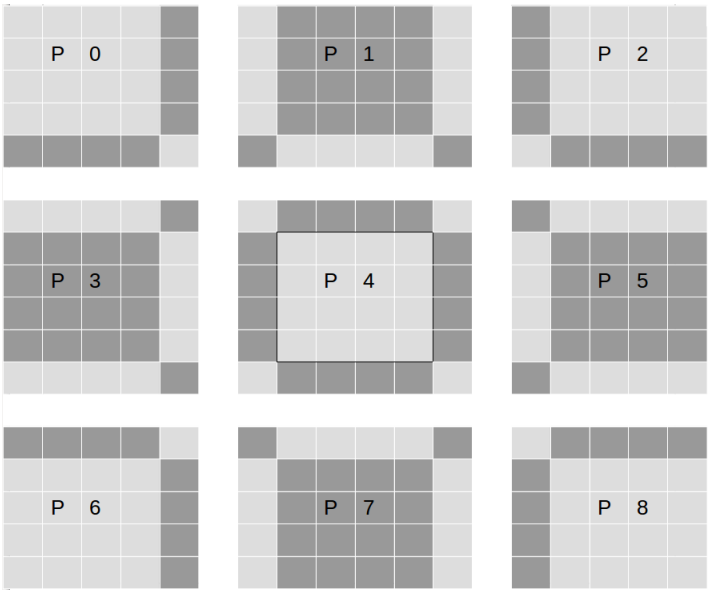

Figure 5 – *Send/Receive nécessaires pour le calcul du schéma*

```
algo shallow_water_bloc;
var size_x,size_y;  /* nombre de lignes et de colonnes de l'image */
var size_x_local, size_y_local;/**hauteur et largeur des blocs*/
var Nb_Col_P=1 /*nombre de colonnes de processus*/
var Nb_Lig_P=1 /*nombre de lignes de processus*/
var hFil_local, uFil_local, vFil_local; /*taille des blocs*/
var hPhy_local, uPhy_local, vPhy_local; /*taille des blocs*/
var NP;              /*Nombre de processus*/
var my_rank;         /*Rang du processus*/

size_x_local = size_x et size_y_local = size_y
alterne_Lig_Col = 0
Tant que NP > 1
    Si alterne_Lig_Col est egale a 0
        size_x_local = size_x_local/2
        alterne_Lig_Col = 1
        Nb_Col_P=2*Nb_Col_P
    else
        size_y_local = size_y_local/2
        alterne_Lig_Col = 0
        Nb_Lig_P=2*Nb_Lig_P
    FinSi
    NP = NP/2
FinTantque


allocation de 2*size_x_local*size_y_local memoire pour les tableaux

Pour i=0...,size_x_local-1
    Pour j = 0,...,size_y_local-1

    HFIL_LOCAL(0, i+(my_rank>=NbCol), j+(my_rank%NbCol!=0)) = height *
    (exp(-pow(((i+(my_rank/NbCol)*size_x/NbLi) *dx-gmx) / gsx,2)/2.))*
    (exp(-pow(((j+(my_rank%NbCol)*size_y/NbCol) * dy-gmy)/gsy,2)/ 2.)) ;
    Finpour

Finpour


/* forward */
Pour i=0,...,nbiter

    calcul de hFil_local, uFil_local,
    vFil_local,hPhy_local, uPhy_local, vPhy_local

    Si my_rank >= Nb_Col_P

        Envoie de la 1ere ligne local_data+w au processus my_rank-Nb_Col_P

        Reception de la derniere ligne de my_rank-Nb_Col_P dans local_data

    FinSi

    Si my_rank < Nb_Col_P*(Nb_Lig_P-1)

        Reception de la 1ere ligne de my_rank+Nb_Col_P
        dans local_data+(size_x_local-1)*size_y_local

        Envoie de la derniere ligne local_data+(h_local-2)*w
        au processus  my_rank+Nb_Col_P

    FinSi
    Si my_rank modulo Nb_Col_P different de 0
```

```
        Envoie de la 1ere colonne (a l aide d un buffer)
        au processus my_rank-1

        Reception de la derniere colonne de my_rank-1
        dans la deuxieme colonne (a l aide d un buffer)

    FinSi

    Si (my_rank+1) modulo Nb_Col_P different de 0

        Reception de la 1ere colonne de my_rank+1
        dans derniere colonne

        Envoie de la derniere colonne pour la premiere
        colonne du processus  my_rank+1

    FinSi

    (Gather) Rassemblement de tous les hfil au prossesus 0
    /*export au fichier*/
    Si my_rank egal 0

        export(shalw_<size_y>x<size_y>_T<NP>.sav,HFILL);

    FinSi

Finpour
```

2.3 Décomposition par Blocs vs Décomposition Bandes

La méthode de décomposition par blocs est plus intéressante que celle de décomposition par bandes. En effet, la taille des messages échangés dans la décomposition par blocs, diminue quand le nombre de processus NP augmente. Pour la décomposition par bandes, la taille des messages reste constante. L'avantage de la méthode par bande, est qu'elle est plus facile à coder, et qu'elle nécessite moins de Send/Receive différents (mais la taille totale des Send/Receive est plus élevée). Une faible latence pénalisera la décomposition par bloc et une faible bande passante pénalisera la décomposition par bande.

2.4 Mesure de Performance

Après avoir implémenté les deux algorithmes il est temps de comparer leurs performances avec celle de la version séquentielle. Pour faire cela nous allons travailler avec les paramètres $x = 8192$,$y = 8192$ et $t = 20$. Pour étudier le programme parallèle nous allons voir comment l'efficacité et le speed-up varient en fonction du nombre de processus qu'on utilise.

On commence nos mesures de performance avec la décomposition par bandes et par blocs en mode bloquante. C'est à dire que le programme attend la fin des réceptions et des envoies avant de continuer. Le temps d'exécution du programme en séquentielle est d'environ 5 minutes. On tient à remarquer que ce temps d'exécution est celui du code séquentiel fournit (sans prend en compte la hiérarchie mémoire). Par contre les programmes parallèles fonctionnent eux avec la bonne hiérarchie mémoire.

Decomposition par Bande - Block Mode			
size_x:8192 , size_y:8192 , nbsteps:20			
Nombre de Processus	t [en sec]	Speed-up	Efficacité
1	304.945	-	-
2	104.686	2.912 93	145.6475%
4	91.4753	3.333 62	83.3407%
8	88.9888	3.426 77	42.8348%
16	84.2532	3.619 38	22.6212%

Decomposition par Block - Block Mode			
size_x:8192 , size_y:8192 , nbsteps:20			
Nombre de Processus	t [en sec]	Speed-up	Efficacité
1	304.945	-	-
2	198.674	1.5349	76.7451%
4	93.6439	3.256 42	81.4108%
8	92.7495	3.287 83	41.0980%
16	85.4941	3.566 85	22.2929%

On voit qu'en général la décomposition par bandes est plus performante que celle par blocs. La décomposition par blocs envoie deux fois plus de messages que celle par bandes donc on peut en déduire que les ordinateurs utilisés ont une faible latence. En général on s'attend à ce que la décomposition par blocs soit la méthode la plus rapide.

Pour valider d'avantage nos résultats, nous allons mesurer la performance de deux méthodes en mode non-bloquant[3]

Decomposition par Bande - Non Block Mode			
size_x:8192 , size_y:8192 , nbsteps:20			
Nombre de Processus	t [en sec]	Speed-up	Efficacité
1	304.945	-	-
2	93.1449	3.2739	163.6940%
4	90.7309	3.360 98	84.0246%
8	89.159	3.420 24	42.7530%
16	84.3341	3.6159	22.5995%

Decomposition par Block - Non Block Mode			
size_x:8192 , size_y:8192 , nbsteps:20			
Nombre de Processus	t [en sec]	Speed-up	Efficacité
1	304.945	-	-
2	101.259	3.011 54	150.5768%
4	96.983	3.1443	78.6078%
8	90.5597	3.367 34	42.0917%
16	86.0522	3.543 72	22.1482%

En mode non-bloquant on arrive à accélérer nos calculs de façon très significative. Pour $NP = 2$ on a un gain en efficacité très important surtout pour la décomposition par blocs. Néanmoins, pour NP supérieur à 2 le mode non bloquant ne nous permet pas de gagner grand chose.

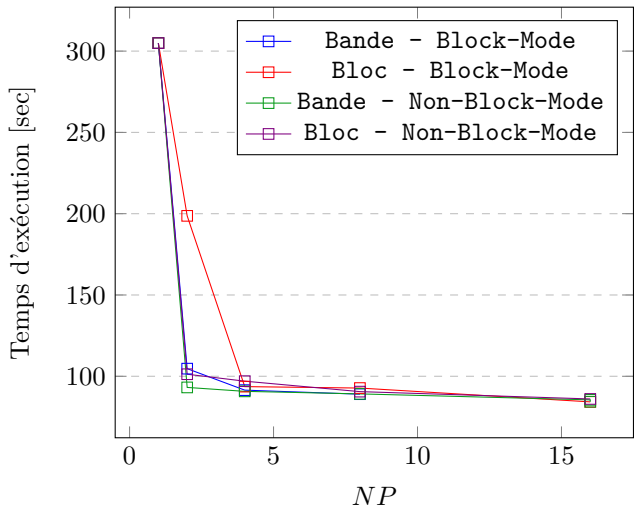

Figure 6 – *Temps d'exécution en fonction du nombre de processeurs*

[3]On utilise de Isent/Irecv avec un Waitall à la fin.

2.5 MPI_IO

Dans cette partie nous allons étudier les entrées/sorties parallèles. Jusqu'à maintenant, dans nos programmes parallèles, nous n'avions qu'un processus capable d'écrire sur le fichier généré avec l'export. Avec MPI nous pouvons autoriser aux différents processus à participer à l'écriture du fichier.

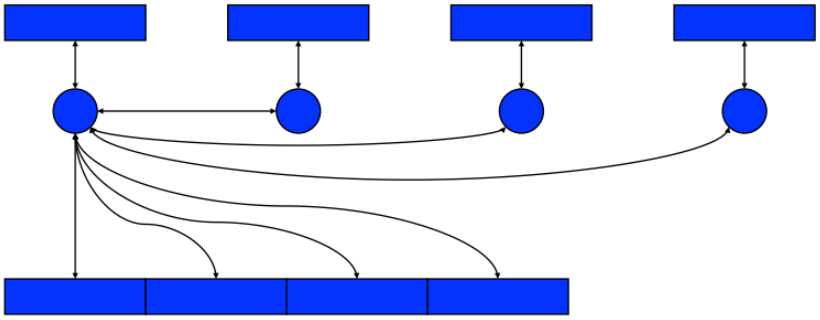

Figure 7 – *E/S non parallèle*

Pour paralléliser l'écriture nous allons nous servir de MPI_IO. Chaque processus va écrire sur un endroit prédéfinit par un offset pour qu'à la fin on puisse avoir le bon fichier. Si on est au processus de rang my_rank à l'instant t avec NP processus l'offset sera

$$offset = my\_rank \times \frac{size\_x}{NP} \times size\_y \times sizeof(double) + (t) \times size\_x \times size\_y \times sizeof(double)$$

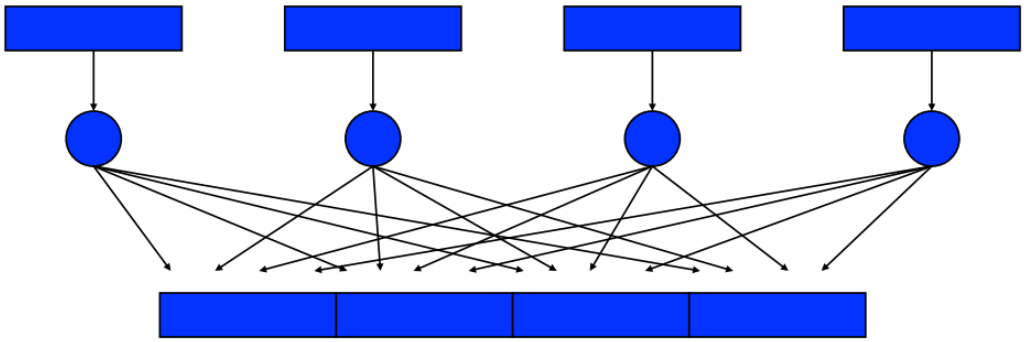

Figure 8 – *MPI_IO: E/S parallèle*

Dans nos codes nous avons utilisé les E/S collectives pour être thread-safe et pour ne pas avoir de problème de concurrence entre les différents processus.

Pour étudier MPI_IO nous l'avons exécuté avec les paramètres proposés dans l'énoncer. Avec NB-IO on note le mode non bloquant des E/S MPI. Avec BMode ou Block-Mode on note le mode bloquant des Send/Recv. Et enfin quand on ne met rien à la fin c'est le mode parallèle normal avec un processus qui écrit et MP_IO et les E/S MPI bloquant.

MPI_IO			
size_x:512 , size_y:512 , nbsteps:80 ,NP:4			
Mode d'exécution	t [en sec]	Speed-up	Efficacité
NP:1 - Block-Mode	9.62589	-	-
Block-Mode	5.09481	1.889 36	47.23%
NB Mode	4.7561	2.0239	50.60%
BMode MP_IO	4.28357	2.247 15	56.18%
BMode NB-MP_IO	4.22961	2.275 82	56.90%
NBMode MP_IO	3.78488	2.543 23	63.58%
NBMode NB-MP_IO	3.64451	2.641 19	66.03%

On voit qu'une parallélisation des entrées/sorties peut nous faire gagner en temps et en efficacité puisque que l'on ne fait pas de gather et que tous les processus se chargent eux-mêmes d'écrire leur partie sur le fichier.

3 Partie deux

3.1 Hiérarchie mémoire

Quand on charge les données dans un ordinateur, pour accélérer le chargement de données, le système d'exploitation charge une partie des données sur la mémoire cache. La mémoire cache étant très proche du CPU, elle permet un chargement 10 à 100 fois plus rapide que des données sur la RAM ou sur le disque dur. Ainsi on dit qu'on a un *cache hit* (réponse positive du cache) lorsqu'une application ou un logiciel demande des données qui se trouvent dans la mémoire cache. L'unité centrale de traitement (CPU) recherche les données dans sa mémoire la plus proche, qui est généralement la mémoire cache principale. Si les données demandées sont trouvées dans le cache, on considère qu'il s'agit d'une réponse positive.

Une réponse positive au cache sert les données plus rapidement, car les données peuvent être récupérées en lisant la mémoire cache. Le succès des caches peut également être dans des caches de disque où les données demandées sont stockées et accédées à la première requête.

Dans le cas contraire nous avons un *cache mis*.Le *cache mis* se produit dans les modes et méthodes d'accès à la mémoire cache. Pour chaque nouvelle requête, le processeur cherche dans le cache primaire pour trouver ses données. Si les données ne sont pas trouvées, on considère qu'il s'agit d'un oubli de cache.

Chaque absence de cache ralentit le processus global parce qu'après une absence de cache, l'unité centrale de traitement (CPU) cherchera un cache de niveau supérieur, comme L1, L2, L3 et la mémoire vive (RAM) pour ces données. De plus, pour chacune de ces recherches, une nouvelle entrée est créée et copiée dans le cache avant que le processeur ne puisse y accéder.

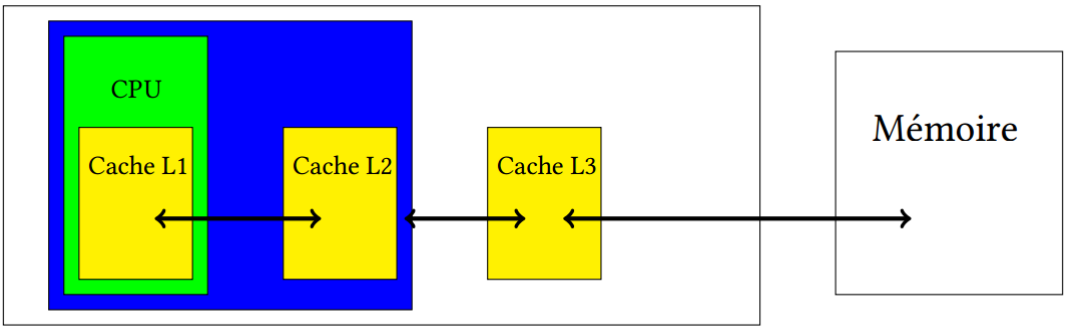

Figure 9 – *Architecture d'un ordinateur*

L'objectif de cette partie est de s'assurer le plus souvent possible d'aboir un cache hit et pas mis. Pour assurer cela, il faut que les boucles sur les tableaux respectent la hiérarchie mémoire des objets. Par exemple si on a une boucle sur une tableau il ne faut pas faire un parcourt en colonne mais un parcourt en ligne. Dans le programme séquentiel du début, nous avions un parcourt en colonne de nos tableaux. Ce qui fait qu'on fait à chaque itération de i, des sauts de $size\_y$ éléments. Avec de tels sauts on maximise le cache mis du programme et par conséquence on le ralentit. En revanche, si on veut maximiser le cache hit et accélérer le programme on doit faire:

```
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);
      }
    }
```

En interchangeant la boucle sur les i avec celle sur les j nous avons une accélération de 5. De 10 secondes on passe à 2.2 secondes dans le cas par défaut.

Cache hit-Cache Miss		
Mode d'exécution	t [en sec]	Speed-up
size_x:256 , size_y:256 , nbsteps:1000		
Mode-Cache Miss	9.92629	-
Mode-Cache Hit	2.08167	4.7684
size_x:512 , size_y:512 , nbsteps:1000		
Mode-Cache Miss	51.864	-
Mode-Cache Hit	9.72505	5.333

4 Parallélisation MPI+OpenMP

Dans ce partie nous allons injecter dans notre code parallèle, des pragma OpeMP pour le paralléliser d'avantage. La principale différence entre la parallélisation MPI et OpenMp est que MPI est une parallélisation multiprocesseurs tandis que OpenMP est une parallélisation multithread. Autrement dit on n'a pas besoin d'échanger des messages sur OpenMP car les différents threads générés partagent la même mémoire.

Comme dans la parallélisation OpenMP il y a un partage de mémoire, on peut paralléliser presque tous nos boucles for. La seule boucle qu'on ne peut pas paralléliser est la boucle for sur les t.

```
//Boucle non-parallelisable
for (t = 1; t < nb_steps; t++)
{
   #pragma omp parallel
   for (int i = ...; i < ...; i++) {
       for (int j = ...; j < ...; j++) {
       /*Calcul de HFIL etc*/
       }
    }
}
```

Dans nos mesures de performance nous utilisons toujours quatre threads car les machines utilisées sont des machines à deux cœurs de chacun deux threads (quatre threads au total).

Decomposition par Bande - Block Mode			
size_x:8192 , size_y:8192 , nbsteps:20			
Nombre de Processus	t [en sec]	Speed-up	Efficacité
1 et 0 threads (cache miss)	304.945	-	-
1 et 0 threads (cache hit)	86.3005	3.533 51	353.3526%
1 et 4 threads (cache hit)	43.2039	7.058 27	705.8279
2 et 4 threads (cache hit)	77.1119	3.954 57	197.7287%
4 et 4 threads (cache hit)	79.9965	3.811 98	95.2995%
8 et 4 threads (cache hit)	82.5966	3.691 99	46.1498%

Très étonnement on voit que OpenMP+CacheHit arrive à accélérer le programme beaucoup plus que n'importe quel autre combinaison. OpenMP est beaucoup plus efficace que la combinaison OpenMP+CacheHit+MPI. Ceci nous permet de comprendre que les communications MPI ont une influence très significative sur le temps de calcul.

Decomposition par Bande - Non Block Mode			
size_x:8192 , size_y:8192 , nbsteps:20			
Nombre de Processus	t [en sec]	Speed-up	Efficacité
1 et 0 threads (cache miss)	304.945	-	-
1 et 0 threads (cache hit)	86.3005	3.533 51	353.3526%
1 et 4 threads (cache hit)	43.2039	7.058 27	705.8279
2 et 4 threads (cache hit)	74.8714	4.0729	203.6460%
4 et 4 threads (cache hit)	81.2771	3.7519	93.7979%
8 et 4 threads (cache hit)	83.0803	3.670 49	45.8811%

Même avec des communications non-bloquantes on n'arrive pas à atteindre l'accélération qu'OpenMP pur nous permet d'avoir. Une autre observation encore plus étonnante qui découle des tableaux ci-dessus, est le fait qu'avoir une bonne hiérarchie mémoire sur un programme est presque équivalent à une parallélisation MPI.

5 AVX

En regardant la fonction $forward$ du fichier $forward.c$, on remarque que le code sur la boucle imbriquée de i et j se prête bien à une vectorisation SIMD avec les intrinsèques AVX.

Les instructions AVX améliorent les performances d'une application, en traitant de gros morceaux de valeurs en même temps au lieu de traiter les valeurs individuellement. Ces morceaux de valeurs sont appelés vecteurs. Les vecteurs AVX peuvent contenir jusqu'à 256 bits de données. Les vecteurs AVX courants contiennent quatre doubles (4 x 64 bits = 256 bits), huit nombres flottants (8 x 32 bits = 256 bits), ou huit entiers (8 x 32 bits = 256 bits). Avec une parallélisation AVX on s'attend à avoir un programme au moins quatre fois plus rapide que en séquentiel.

La première étape de vectorisation est l'alignement de la mémoire et sa bonne allocation. Pour faire cela sur le fichier *memory.c*, on se sert de la fonction $aligned_alloc$. Ensuite, avec les intrinsèques AVX on crée nos vecteurs d'éléments et on fait les calculs des différentes fonctions forward.

Cache hit-Cache Miss		
Mode d'exécution	t [en sec]	Speed-up
size_x:2048 , size_y:2048 , nbsteps:40		
Mode-Cache Miss	37.9193	-
Mode-Cache Hit	15.6316	2.4258
AVX+Cache Hit	2.95811	12.8187
size_x:4096 , size_y:4096 , nbsteps:20		
Mode-Cache Miss	67.6261	-
Mode-Cache Hit	25.2386	2.6794
AVX+Cache Hit	5.9329	11.398

Avec OpenMP+Cachehit on est arrivé à une accélération de 7, ce qui était très étonnant. Maintenant avec la vectorisation SIMD on va encore plus loin avec une accélération de 11.

Vectorisation AVX			
size_x:8192 , size_y:8192 , nbsteps:20			
Type de Parallisation	t [en sec]	Speed-up	Efficacité
1 Process (cache miss)	304.945	-	-
1 Process (cache hit)	86.3005	3.533 51	353.3526%
1 Process et 4 threads (cache hit)	43.2039	7.058 27	705.8279
AVX(cache hit)	25.6119	11.906 37	1190.6392%

Donc on voit qu'AVX est un outil de parallélisation très puissant qui peut vraiment améliorer notre code s'il est parallélisable.

AVX est aussi rapide parce qu'il prend en compte l'architecture de l'ordinateur et la structure de la mémoire. Grâce à ces considérations on arrive à des performances surréalistes. Contrairement à MPI et éventuellement à OpenMP, avec AVX la parallélisation est efficace car elle ne perd pas de temps à créer des threads/processus[4] (appel système).

Malheureusement AVX est une parallélisation qui ne marche que si le code se prête à être vectorisé et si on a des opérations arithmétiques à effectuer. Dans les autres cas on préfère OpenMP ou/et MPI qui sont plus génériques.

[4]Les créations de threads et de processus sont faites par des appels systèmes. Les appels systèmes sont des opérations coûteuses en temps parce qu'ils nécessitent un changement de contexte.

6 AVX+OPENMP+MPI

Maintenant nous allons prendre notre meilleure parallélisation et nous allons voir comment elle se comporte avec AVX.

Decomposition par Bande - Block Mode + AVX			
size_x:8192 , size_y:8192 , nbsteps:20			
Nombre de Processus	t [en sec]	Speed-up	Efficacité
1 et 0 threads (cache miss)	304.945	-	-
1 et 0 threads (cache hit)	86.3005	3.533 51	353.3526%
1 et 4 threads (cache hit)	43.2039	7.058 27	705.8279
2 et 4 threads (cache hit)	91.4862	3.333 24	166.6618%
2 et 4 threads (cache hit)	87.5377	3.483 58	87.0897%
8 et 4 threads (cache hit)	86.9132	3.5086	43.8577%

Malheureusement il semble que comme OpenMP était pénalisé par MPI, AVX se retrouve également pénalisé par par la présence d'OpenMP et de MPI. On n'arrive pas à améliorer plus qu'avant le programme MPI+OpenMP. Ceci signifie que l'on a atteint une borne inférieure avec OpenMP et MPI. Cela confirme d'ailleurs la loi d'Amdahl. A partir d'un certain moment il est impossible de paralléliser le code parce qu'on peut jamais paralléliser la latence ou le débit.

7 Conclusion

Ce projet de HPC s'est avéré très intéressant et pédagogique. Il nous a permis d'explorer légèrement le vaste monde de HPC et surtout il nous a aidé à voir à quel point on peut optimiser notre code si on suit la hiérarchie mémoire de l'ordinateur et l'architecture de la machine. En effet plus on se rapproche de l'architecture machine ou de l'assembleur, plus on est efficace. AVX est un exemple très caractéristique de cela. Les données sur notre machine sont souvent stockées en mode vectoriel, si on exploite cela on peut augmenter considérablement l'efficacité et la performance du code.

En général la parallélisation avec des threads est plus efficace mais plus pénibles à gérer à cause des accès concurrents. Utiliser plusieurs types de parallélisation ne donne pas forcement de meilleurs résultats que des parallélisation individuelles.

Enfin, AVX est la parallélisation la plus compliquée à mettre en place, mais si on y arrive elle est aussi la plus performante. En effet, elle ne fait pas intervenir d'appels systèmes ou de communication.