



ISTY

Institut des Sciences et Techniques des Yvelines

CAMPUS DE MANTES EN YVELINES

CAMPUS DE SAINT-QUENTIN-EN-YVELINES

Rapport Master M1

Master Calcul Haute Performance Simulation (CHPS)

Optimisation du Code CNN pour la Reconnaissance des Plantes

Réalisé par: Aicha Maaoui, Bouchelga Abdeljalil, Mohamed Aitmhand,

Farhat Aitaider, Lydia Chouaki

Encadré par: Prof. Hugo Bolloré

Mai 2022

Institut des Sciences et Techniques des Yvelines (ISTY)

Contents

1	Introduction	1
1.1	Profilage	2
1.1.1	Gprof	2
1.1.2	Perf	3
2	Loop Tiling et OpenMP	5
2.1	Loop Tiling	5
2.2	Déroulement de la boucle (Loop unrolling)	7
2.3	OpenMP (Open Multiprocessing)	7
2.3.1	Utilisation d'OpenMP	8
2.3.2	Mode d'accès aux variables	10
2.3.3	Scheduling	11
2.4	Étude de la scalabilité	11
2.4.1	Avant optimisation	11
2.4.2	Scalabilité forte	12
2.4.3	Scalabilité faible	13
2.4.4	Après Optimisation	14
2.5	Conclusion	16
3	Optimisation du code en séquentiel	17
3.1	Avant propos	17
3.2	Flags de compilation: Options d'optimisation	17
4	Parallélisation du code avec <i>MPI</i>	19
4.1	Avant propos	19

4.2	Strucutre du code original	19
4.3	1ere tentative de parallélisation: Approche 1	20
4.3.1	Profilage du code avec <i>Gprof</i> et <i>Perf</i>	20
4.3.2	Approche 1: Parallélisation du calcul de convolution	21
4.3.3	Approche 1bis: Tentative d'amélioration de l'approche 1	23
4.4	Approche 2: Décomposition de l'image	25
4.5	Approche 3: Décomposition de l'ensemble des images	26
4.6	Conclusions	27
5	Etude de l'effet de <i>Push_back</i>	29
5.1	Conclusion	30
	References	32
A	Relative à l'implémentation <i>MPI</i>	33
A.1	Description de la machine utilisée dans la partie <i>MPI</i>	33
A.2	Branches d'implémentation <i>MPI</i>	33

List of Figures

1.1	Résultats de gprof profiler	2
1.2	Résultats de perf profiler	3
1.3	Graphe des appels	4
2.1	Loop Tiling	6
2.2	Modèle fork-join	8
2.3	Synchronisation des threads	10
2.4	Exemple de synchronisation des threads	10
2.5	Graphe de Scalabilité forte	12
2.6	Graphe Scalabilité faible	14
2.7	Graphe Scalabilité forte	15
2.8	Graphe de Scalabilité faible	16
3.1	Speed-up obtenu en fonction de flags de compilation.	18
4.1	Structure du code original.	19
4.2	Résultat de profilage du code original avec <i>Gprof</i>	20
4.3	Résultat de profilage du code original avec <i>Perf</i>	20
4.4	Illustration de la procédure de convolution.	21
4.5	Utilisation de 3 processeurs pour le calcul de la convolution.	22
4.6	Speed-up obtenu en fonction du nombre de processus et taille des images d'entrée	23
4.7	Comparison du temps d'exécution avec les approches 1 et <i>1bis</i>	24
4.8	Précision obtenue en fonction du nombre de processus, pour les approches 1 et 1bis, tailles d'images 50×50 et 200×200	25
4.9	Partie a paralléliser dans l'approche 2.	26

4.10	Partie a paralléliser dans l'approche 3.	26
4.11	Temps de Pré-processing en fonction du nombre de processus.	27
4.12	Analyse de scalabilité forte: Speed-up de la partie pre-processing obtenu en fonction de nombre de processus.	28
4.13	Evolution de la précision, perte et temps d'exécution en fonction de nombre de processus.	28
5.1	Structure du code original.	29
5.2	Comparison du temps de pre-processing avec et sans <i>push_back</i>	30
5.3	Comparison du temps d'exécution avec et sans <i>push_back</i>	30

List of Tables

2.1	Variation de l'exécution et efficacité en fonction du nombre de processus.	12
2.2	Variation de l'exécution et efficacité en fonction du taille du probleme.	13
2.3	Variation de l'exécution et efficacité en fonction du nombre de processus.	14
2.4	Variation de l'exécution et efficacité en fonction du nombre de processus et de la taille du probleme.	15
3.1	Ajout de flags de compilation pour optimiser le temps de calcul.	17
A.1	Caractéristiques de la machine utilisée.	33

Chapter 1

Introduction

Dans la premier partie du projet nous avons réalisé notre premier objectif qui consiste à implémenter un réseau de neurones capable de reconnaître les plantes, mais il est très coûteux en terme de performance et manque d'optimisation.

Notre défi maintenant est de l'optimiser et de le rendre plus efficace en augmentant les performances du code. Pour ce faire, nous allons analyser et identifier les différents problèmes du programme, puis nous l'ajusterons progressivement en appliquant plusieurs techniques d'optimisation pour le rendre plus efficace.

La démarche que nous avons suivie dans ce travail est la suivante :

Nous avons d'abord supprimé les fonctions et les fonctionnalités inutiles et nous l'avons réorganisé pour qu'il soit facile à modifier.

Ensuite nous avons utilisé le profilage pour savoir où le programme passe beaucoup de temps, sous peine d'optimiser la mauvaise fonction, nous avons utilisé gprof et perf pour identifier les points du programme qui posent problème. Après cette étape, nous avons penché dans l'optimisation en utilisant tous les outils et les armes qui permettent de réduire le temps d'exécution du programme, comme OpenMP, MPI, loop tiling, etc.

Et à la fin, nous avons fait un etude de scalabilité fortes et faibles pour savoir à quel point notre programme évolue.

- Notre indicateur de mesure c'est le temps en seconde.
- Les informations sur la machine utilisé sont sur le lien github au répertoire Profiling.

1.1 Profilage

Avant de commencer d'optimiser notre programme, il faut savoir quelles sont les fonctions dans lesquelles le programme passe le plus de temps, cela nous permet de connaître à quels endroits on doit concentrer notre attention pour rendre le code efficace. Pour ce faire, nous allons utiliser l'outils *gprof* et *perf*.

1.1.1 Gprof

Gprof est un logiciel GNU Binary Utilities qui permet d'effectuer du profilage de code, "*Profiler gprof*" 2004.

l'utilisation : Nous allons recompilé le programme en ajoutant l'option **-gp** au Makefile, par la suite on exécute le programme normalement, ce qui produit le fichier **gmon.out** qu'on va analyser avec *gprof*.

```
Flat profile :
Each sample counts as 0.01 seconds.
%   cumulative   self           self         total
time  seconds    seconds   calls   s/call   s/call   name
8.32    4.87    4.87    56768    0.00    0.00  Convolution_layer::convolution_process(std::vector<double,
std::allocator<double> > const&, int)
6.65    8.77    3.90  2870112124    0.00    0.00  std::vector<double, std::allocator<double> >::operator[](
5.40   11.93    3.16  2469283920    0.00    0.00  std::vector<std::vector<double, std::allocator<double> >,
std::allocator<std::vector<double, std::allocator<double> > > >::operator[](unsigned long)
4.38   14.49    2.56  255226129    0.00    0.00  void std::vector<double, std::allocator<double>
>::M_realloc_insert<double const&>(__gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double> >::const
double const&
3.00   16.25    1.75  511559237    0.00    0.00  double* std::__relocate_a<double*, double*, std::allocator
>(double*, double*, double*, std::allocator<double>&)
2.95   17.97    1.73  1307962870    0.00    0.00  std::vector<double, std::allocator<double> >::operator[](
long) const
2.55   19.46    1.49  511605373    0.00    0.00  std::vector<double, std::allocator<double>
>::S_max_size(std::allocator<double> const&)
2.24   20.77    1.31  1514140368    0.00    0.00  __gnu_cxx::__normal_iterator<double*, std::vector<double,
std::allocator<double> > >::base() const
2.12   22.01    1.24  1162534676    0.00    0.00  std::vector<double, std::allocator<double> >::size() cons
2.10   23.24    1.23  511559237    0.00    0.00  std::enable_if<std::__is_bitwise_relocatable<double, void>
double*>::type std::__relocate_a_1<double, double>(double*, double*, double*, std::allocator<double>&)
2.07   24.45    1.21    3548    0.00    0.00  Convolution_layer::BackPropagation(std::vector<std::vector<
std::allocator<double> >, std::allocator<std::vector<double, std::allocator<double> > > >, double)
2.05   25.65    1.20  565584956    0.00    0.00  void std::allocator_traits<std::allocator<double> >::const
double const&>(std::allocator<double>&, double*, double const&)
2.02   26.83    1.18  255779624    0.00    0.00  std::vector<double, std::allocator<double> >::_M_check_len
long, char const*) const
1.93   27.96    1.13  741522846    0.00    0.00  __gnu_cxx::__normal_iterator<double*, std::vector<double,
std::allocator<double> > >::_normal_iterator(double* const&)
1.88   29.06    1.10  1576122979    0.00    0.00  double* std::__niter_base<double*>(double*)
1.88   30.16    1.10  565584956    0.00    0.00  std::vector<double, std::allocator<double> >::push_back(do
1.78   31.20    1.04    3548    0.00    0.00  Softmax_layer::BackPropagation(std::vector<double,
std::allocator<double> > const&, double)
1.74   32.22    1.02  304976185    0.00    0.00  std::vector<double, std::allocator<double> >::begin()
1.70   33.22    0.99  1386396041    0.00    0.00  double const& std::forward<double const&>(std::remove_ref
```

Figure 1.1: Résultats de gprof profiler

les résultats que **gprof** nous a montré ne sont pas présentables et il se peut qu'ils ne soient pas très précis, pour cela nous pouvons utiliser un autre profiler comme **perf** pour avoir plus d'informations sur les fonctions qui prennent beaucoup de temps.

1.1.2 Perf

Perf est un outil d'analyse de performance sous Linux, il peut également être utilisé pour faire des traces "*Profiler perf*" 2007.

Nous exécutons le programme avec la commande : **perf record ./program**. A l'aide de cette commande le programme génère le fichier perf.data qui contient toutes les informations temporelle de notre programme.

Puis lorsqu'on exécute la commande suivantes : **perf report**. Cela donne la vue suivantes :

Overhead	Command	Shared Object	Symbol
8.03%	pe	pe	[.] Convolution_layer::convolution_process
5.29%	pe	pe	[.] std::vector<double, std::allocator<double>>::_M_realloc_insert<double const&>
4.73%	pe	pe	[.] std::vector<std::vector<double, std::allocator<double>>, std::allocator<std::vector<double, std::allocator<double>>>>::_M_check_len
3.66%	pe	pe	[.] std::vector<double, std::allocator<double>>::_operator[]
2.71%	pe	pe	[.] std::vector<double, std::allocator<double>>::_size
2.40%	pe	pe	[.] std::vector<double, std::allocator<double>>::_M_check_len
2.24%	pe	pe	[.] std::vector<double, std::allocator<double>>::_push_back
2.09%	pe	pe	[.] std::_relocate_a_1<double, double>
2.07%	pe	pe	[.] std::_relocate_a<double*, double*, std::allocator<double>> >
1.97%	pe	pe	[.] std::_niter_base<double*>
1.90%	pe	pe	[.] std::vector<double, std::allocator<double>>::_S_relocate
1.87%	pe	pe	[.] __gnu_cxx::new_allocator<double>::construct<double, double const&>
1.86%	pe	pe	[.] std::vector<double, std::allocator<double>>::_operator[]
1.85%	pe	pe	[.] Softmax_layer::BackPropagation
1.85%	pe	libc.so.6	[.] _int_free
1.83%	pe	pe	[.] __gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double>>>::_S_do_relocate
1.73%	pe	pe	[.] std::vector<double, std::allocator<double>>::_S_do_relocate
1.70%	pe	pe	[.] __gnu_cxx::__normal_iterator<double*, std::vector<double, std::allocator<double>>>::operator[]
1.70%	pe	libc.so.6	[.] malloc
1.64%	pe	pe	[.] Convolution_layer::BackPropagation
1.53%	pe	pe	[.] std::vector<double, std::allocator<double>>::_S_max_size
1.47%	pe	pe	[.] Pooling_layer::BackPropagation
1.43%	pe	pe	[.] std::allocator_traits<std::allocator<double>>::construct<double, double const&>
1.38%	pe	pe	[.] __gnu_cxx::new_allocator<double>::allocate
1.26%	pe	pe	[.] std::min<unsigned long>
1.23%	pe	pe	[.] __gnu_cxx::operator-<double*, std::vector<double, std::allocator<double>> > >
1.23%	pe	pe	[.] std::vector<double, std::allocator<double>>::_end
1.20%	pe	pe	[.] std::forward<double const&>
1.11%	pe	pe	[.] std::_Vector_base<double, std::allocator<double>>::_M_deallocate
1.04%	pe	pe	[.] std::vector<double, std::allocator<double>>::_max_size
1.04%	pe	pe	[.] std::vector<double, std::allocator<double>>::_begin
1.04%	pe	pe	[.] Pooling_layer::Pooling_process
0.94%	pe	pe	[.] __gnu_cxx::operator!=<double*, std::vector<double, std::allocator<double>> > >
0.88%	pe	libc.so.6	[.] __memmove_avx_unaligned_erms
0.88%	pe	pe	[.] std::_Vector_base<double, std::allocator<double>>::_M_get_Tp_allocator
0.85%	pe	libc.so.6	[.] cfree@GLIBC_2.2.5
0.82%	pe	pe	[.] std::_Vector_base<double, std::allocator<double>>::_M_allocate
0.81%	pe	pe	[.] std::allocator_traits<std::allocator<double>>::_allocate
0.80%	pe	pe	[.] std::allocator_traits<std::allocator<double>>::_max_size

Figure 1.2: Résultats de perf profiler

Par défaut, **perf** ne collecte que les informations temporelles. Pour ce faire, nous exécutons la commande suivante : **perf record -g ./program** ensuite **perf record** pour obtenir le graphe des appels.

Samples: 510K of event 'cycles:u', Event count (approx.): 317379096085

Children	Self	Command	Shared Object	Symbol
+ 94.24%	0.00%	program	libc.so.6	[.] __libc_start_call_main
+ 94.24%	0.00%	program	program	[.] main
+ 74.02%	0.00%	program	program	[.] output::Training_data
+ 45.94%	2.22%	program	program	[.] std::vector<double, std::allocator<double> >::push_
+ 40.02%	0.00%	program	program	[.] output::prediction
+ 39.16%	5.42%	program	program	[.] std::vector<double, std::allocator<double> >::_M_re
+ 23.82%	1.87%	program	program	[.] Softmax_layer::BackPropagation
+ 23.58%	0.00%	program	program	[.] Pooling_layer::Pooling_parameters
+ 23.25%	1.02%	program	program	[.] Pooling_layer::Pooling_process
+ 20.22%	0.00%	program	program	[.] output::Testing_data
+ 15.43%	1.47%	program	program	[.] Pooling_layer::BackPropagation
+ 14.82%	0.00%	program	program	[.] Convolution_layer::convolution_parameters
+ 13.60%	8.12%	program	program	[.] Convolution_layer::convolution_process
+ 13.25%	1.64%	program	program	[.] Convolution_layer::BackPropagation
+ 10.86%	2.35%	program	program	[.] std::vector<double, std::allocator<double> >::_M_ch
+ 10.03%	1.88%	program	program	[.] std::vector<double, std::allocator<double> >::_S_re
+ 8.46%	1.72%	program	program	[.] std::vector<double, std::allocator<double> >::_S_do
+ 6.96%	0.16%	program	program	[.] std::vector<std::vector<double, std::allocator<doub
+ 6.14%	2.00%	program	program	[.] std::__relocate_a<double*, double*, std::allocator<
+ 6.12%	1.06%	program	program	[.] std::vector<double, std::allocator<double> >::max_s
+ 5.62%	1.66%	program	program	[.] std::allocator_traits<std::allocator<double> >::con
+ 5.07%	4.68%	program	program	[.] std::vector<std::vector<double, std::allocator<doub
+ 4.82%	0.47%	program	program	[.] std::vector<double, std::allocator<double> >::vecto
+ 4.62%	0.74%	program	program	[.] std::vector<double, std::allocator<double> >::~vect
+ 4.41%	1.51%	program	program	[.] std::vector<double, std::allocator<double> >::_S_ma
+ 4.29%	3.64%	program	program	[.] std::vector<double, std::allocator<double> >::opera
+ 4.19%	0.79%	program	program	[.] std::_Vector_base<double, std::allocator<double> >::
+ 3.90%	0.13%	program	program	[.] std::max_element<__gnu_cxx::__normal_iterator<doubl
+ 3.52%	0.72%	program	program	[.] std::__max_element<__gnu_cxx::__normal_iterator<dou
+ 3.50%	0.01%	program	program	[.] std::vector<std::vector<double, std::allocator<doub
+ 3.43%	0.00%	program	program	[.] std::vector<std::vector<double, std::allocator<doub
+ 3.43%	0.00%	program	program	[.] std::vector<std::vector<double, std::allocator<doub

Figure 1.3: Graphe des appels

Après avoir pris une idée sur les parties dont le programme passe beaucoup de temps, nous nous penchons maintenant sur l'optimisation.

Chapter 2

Loop Tiling et OpenMP

2.1 Loop Tiling

Il s'avère que notre code n'utilise pas la mémoire cache pour un accès plus rapide à la mémoire. Une solution à cela consiste à utiliser une technique appelée **Loop Tiling**.

Rappelons que l'architecture de nos machines actuelles est comme suit :

- **Le cache L1** est de 32 Kio par cœur. C'est $32 \times 1024 = 32,768$ bytes par core.
- **Le cache L2** est de 512 Kio par cœur (1 Mio par tuile = 1024 Ko par tuile. Chaque tuile a 2 cœurs. Donc $1024 \text{ KiB par tiles} / 2 \text{ cores per tiles} = 512 \text{ KiB par core}$), c'est $512 \times 1024 = 524,288$ bytes par core.

Lorsque la taille de nos problèmes est petite, les données dans une boucle peuvent tenir dans le cache L1/L2 pour des performances élevées/moyennes. Lorsque la taille devient trop élevée, les données peuvent ne pas tenir dans le cache et "se répandre" dans la RAM et pas le cache, ce qui est beaucoup plus lent. Cela entraîne une dégradation des performances.

Il s'avère qu'en pratique, nous pouvons améliorer les performances de notre cache avec la technique **Loop Tiling**. C'est une technique conçue pour conserver votre jeu de données dans des caches pendant que nous travaillons avec, pour profiter de la latence de la mémoire.

l'image (2.1) est tirée de l'article "*Loop Tiling*" 2021.

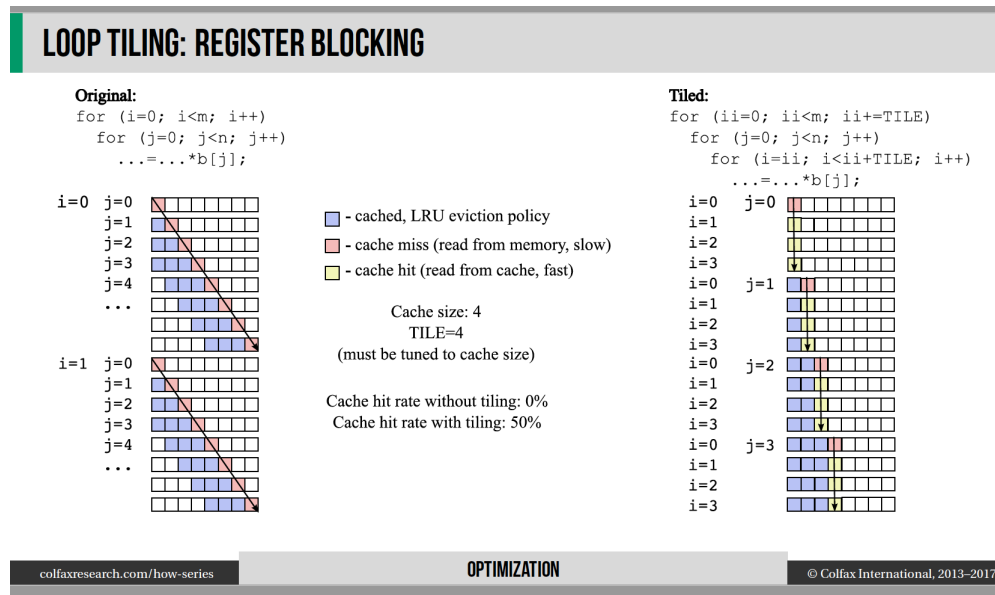


Figure 2.1: Loop Tiling

Ce qui se passe en pratique, c'est que l'utilisation de loop tiling permet de réorganiser les accès en blocs, de sorte que la répétition du même bloc peut atteindre le cache plusieurs fois. Étant donné que la taille du bloc est définie par exemple sur 32k, il s'adaptera probablement à la plupart des caches. Et cela peut facilement s'intégrer dans le cache L1 et on aura un taux d'accès au cache élevé (performance élevé) sinon si la taille est plus grand que 32,768, il se peut tenir dans le cache L2 (performance moyenne).

Application

Exemple de la boucle qui fait la produit de convolution dans la fonction **convolution_process**

Avant :

Listing 2.1: Avant: Exemple de boucle faisant le produit de convolution dans **convolution_process**.

```
1 for (int ii = 0; ii < ConvMat.height; ii++) {
2   //loop on the height of the convolution matrix
3   for (int jj = 0; jj < ConvMat.width; jj++) {
4     //loop on the width of the convolution matrix
5
6     double sum = 0; //initialization of the summation
7     .....
8     double image = (pixel[((ii + kk) * (ConvMat.width + 2) + (jj + hh))]);
9     sum += (image * filter_matrix[idx][kk * filter_width + hh]);
10    .....
```

Après :

Listing 2.2: Après: Exemple de boucle faisant le produit de convolution dans **convolution_process**.

```

1 for (int a = 0; a < ConvMat_height; a+=Tile) {
2   for (int jj = 0; jj < ConvMat_width; jj++) { //loop on the width of the
        convolution matrix
3     for(int ii = a; ii < a+Tile; ii++){
4       double sum = 0; //initialization of the summation
5       .....
6       double image = (pixel[(( ii + kk) * (ConvMat_width + 2) + (jj + hh))]);
7       sum += (image * filter_matrix[idx][kk * filter_width + hh]);
8       .....

```

La syntaxe **après** produit le même résultat que la syntaxe **avant**, mais il le fait simplement d'une manière plus efficace pour la machine (en tirant le meilleur parti du cache rapide).

2.2 Déroulement de la boucle (Loop unrolling)

Le déroulement de boucle est une technique de transformation de boucle qui permet d'optimiser le temps d'exécution d'un programme. Nous supprimons ou réduisons essentiellement les itérations. Le déroulement de la boucle augmente la vitesse du programme en éliminant les instructions de contrôle de boucle et les instructions de test de boucle.

La bonne chose à propos de la version déroulée est qu'elle implique moins de charge de traitement pour le processeur.

Nous allons ajouter le drapeau **#pragma unroll** aux boucles qui permet au compilateur de "dérouler" des boucles.

2.3 OpenMP (Open Multiprocessing)

Les ordinateurs personnels modernes sont équipés de processeurs multicœurs, ce qui permet l'exécution simultanée de deux flux d'instructions indépendants. Grâce à cette fonctionnalité, vous pouvez doubler les performances sur un processeur double cœur, quatre fois sur un processeur quadricœur, etc.

Il existe plusieurs technologies pour implémenter des threads dans une application. Pour cela nous avons la bibliothèque **OpenMP** fournit un ensemble limité de directives pour programmation multi-thread :

- Création de sections parallèles.
- Répartition des itérations de boucle entre les threads.
- Répartition de charge.

OpenMP signifie Open Multiprocessing, c'est une bibliothèque de programmation parallèle implémentée en tant qu'extensions de compilateur. La bibliothèque comprend une description des directives, des fonctions et des variables d'environnement.

2.3.1 Utilisation d'OpenMP

Le travail avec la bibliothèque OpenMP est comme suit, "*Optimisation et parallélisme avec OpenMP*" 2007 : Nous utilisons des directives du compilateur aux endroits requis pour dire au compilateur quoi faire avec le programme séquentiel. L'avantage c'est que cette approche garantit l'unicité du code, c'est à dire le programme série et parallèle se ressemblent. Et quand un compilateur qui ne prend pas en charge OpenMP ignorera les directives inconnues.

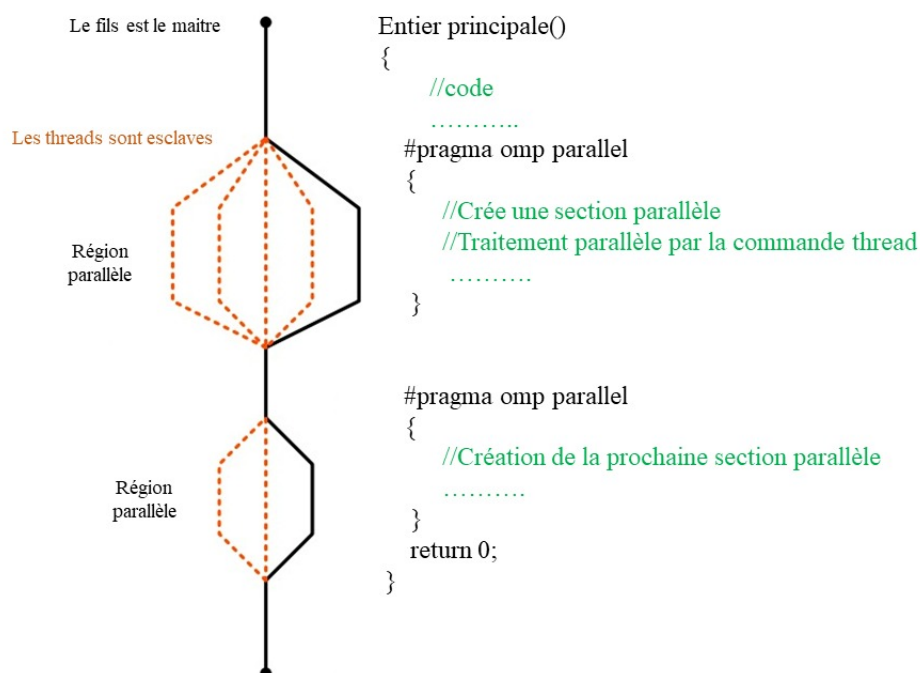


Figure 2.2: Modèle fork-join

OpenMP utilise le parallélisme **”pulsé” (fork-join)**: le nombre de threads exécutés en parallèle peut changer pendant l’exécution. La plupart des fonctionnalités de la bibliothèque sont réalisées via des directives au compilateur.

Format directif :

```
# pragma omp commande [option [option] ... ]
{
    // bloc
    . . .
}
```

La directive s’applique uniquement à l’instruction ou au bloc suivant. Une directive peut avoir plusieurs options. Les directives sont sensibles au forme de lettre. Ils doivent être écrits en lettres minuscules.

La commande parallèle est la principale directive OpenMP. L’exécution parallèle du programme commence par lui. Le compilateur remplace la directive par un code spécial pour créer un groupe de threads. Le thread qui crée d’autres threads est appelé **”maître (Master)”**, les threads créés sont appelés **”esclaves (slave)”**. Chaque thread (y compris le thread maître ainsi que les threads esclaves) exécute un bloc de code en parallèle. Les threads sont numérotés de 0 à N, où 0 est le thread maître.

Format directif :

```
# pragma omp parall le [option[ [, ] option] ...]
{
    // bloc
    . . .
}
```

Les threads sont synchronisés en fin de bloc (synchronisation implicite de la barrière). Le programme ne poursuivra pas son exécution tant que tous les threads du bloc n’auront pas terminé leur travail. Toutes les variables sont partagées entre les threads.

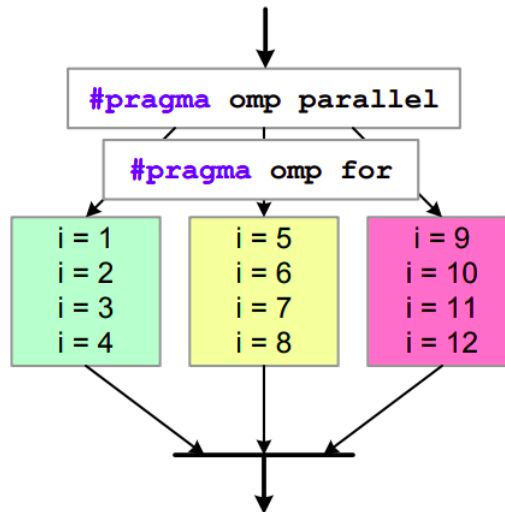


Figure 2.3: Synchronisation des threads

Exemple :

```
#pragma omp parallel for
for (i = 0; i < 12; i++)
c[i] = a[i] + b[i];
```

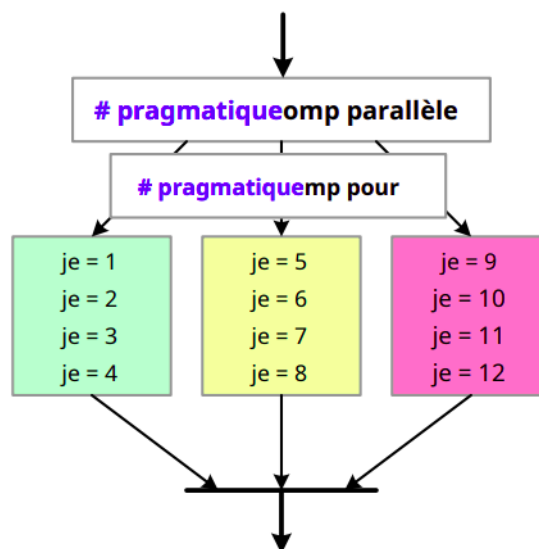


Figure 2.4: Exemple de synchronisation des threads

2.3.2 Mode d'accès aux variables

Il est possible d'influencer la visibilité des variables à l'aide d'options privé et partagé:

- *Mode privé* - toutes les variables de la liste deviennent privées dans la région parallèle.

- *Mode partagé* - toutes les variables de la liste deviennent communes au thread.

2.3.3 Scheduling

OpenMP a la capacité de contrôler la planification des itérations entre les threads. La première option est **statique**. Une distribution uniforme statique des itérations est effectuée. chaque thread décide indépendamment quel morceau de la boucle il traitera.

La deuxième option est **dynamique**, dans ce cas toutes les itérations sont divisées en blocs, mais l'affectation aux threads ne se produit pas. Au cours de son exécution, le thread demande la prochaine "portion" de travail.

Les pragmas que nous avons utilisé pour notre programme :

```
#pragma omp parallel for
#pragma omp for private(n)
#pragma omp for schedule(dynamic)
```

2.4 Étude de la scalabilité

Parmi nos objectif dans ce présent travail est de rendre notre programme scalable, "*Scaling tutorial*" 2021 "*Scalabilité*" 2022 "*Scalability: strong and weak scaling*" 2016.

2.4.1 Avant optimisation

Un algorithme parallèle est dit scalable si , avec une augmentation du nombre de processeurs, il permet une augmentation de l'accélération tout en maintenant un niveau constant d'efficacité dans l'utilisation des processeurs. Nous avons deux types de scalabilité, la scalabilité forte et la scalabilité faible.

- **la scalabilité forte** signifie que le temps total d'exécution des tâches diminuera de manière linéaire lorsqu'on augmente le nombre de processus. Cependant, conformément à la loi d'Amdahl [1] pour l'exécution parallèle d'un algorithme, le temps total d'exécution d'un programme ne peut être réduit que sur un segment de code optimisé de manière appropriée.
- **la scalabilité faible** signifie que lorsque la taille du problème augmente proportionnellement au nombre de processus ajoutés, le temps d'exécution reste stable.

2.4.2 Scalabilité forte

On garde la taille de notre problème fixe qui représente la taille de l'image d'entrée (la matrice de pixels) et on augmente le nombre de coeurs.

Calcul de l'efficacité : L'efficacité se définit par l'expression suivante :

$$E(p) = \frac{S(p)}{p} \quad (2.1)$$

avec:

$$S(p) = \frac{T_1}{T_p} \quad (2.2)$$

où:

S(p) : Représente l'accélération.

p: le nombre de processus.

T_1 : temps d'exécution pour un seul processus.

Les résultats sont présentés dans le tableau suivant et la figure (2.5).

nombre de procussus	durée d'exécution (secondes)	efficacité (%)
1	128.963256	.
2	128.828312	50.05
4	128.005621	25.18
8	129.793375	12.42
16	132.667213	6.07

Table 2.1: Variation de l'exécution et efficacité en fonction du nombre de processus.

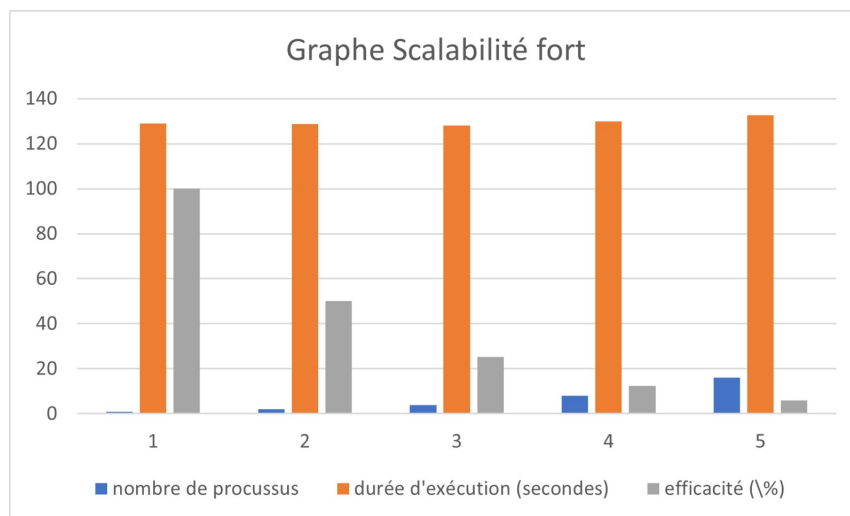


Figure 2.5: Graphe de Scalabilité forte

La courbe nous montre que lorsque le nombre de processus augmente jusqu'à 8, le temps est légèrement changé, mais il augmente fortement lorsque le nombre de processus devient trop important, c'est à dire que le programme ralentit avec l'ajout de processus.

Avec 16 processus, la durée d'exécution est plus grande qu'avec 1 seule processus (128 secondes contre 132) et l'efficacité est faible à 6.07%.

Donc l'efficacité diminue lorsque le nombre de processus augmente, ce qui prouve que *nous n'avons pas une forte scalabilité*.

Si nous passons de 2 à 4 processus et que le temps d'exécution diminue de moitié, dans ce cas nous aurons une efficacité de 100%, pour la suite de notre travail, après l'optimisation nous privilégions une efficacité de 75% ou plus.

2.4.3 Scalabilité faible

Maintenant nous allons augmenter la taille de notre problème avec le nombre de processus.

Remarque : Pour augmenter la taille de notre problème, nous allons multiplier par 2 la taille c'est à dire notre matrice de pixels. par exemple pour une premier taille de 50×50 (2500 pixels), si nous multiplions par 2 nous trouvons 5000, où cela donne une taille de dimension presque égale à 70×70 .

Le calcul de l'efficacité : la durée d'exécution de référence (1 processus) est divisée par la durée d'exécution avec n processus et le résultat est converti en pourcentage.

Taille du problème	nombre de procussus	durée d'exécution (secondes)	efficacité (%)
50×50	1	126.884265	.
70×70	2	126.204934	100.53
100×100	4	126.591202	100.23
140×140	8	127.397960	99.59
200×200	16	127.851052	99.24

Table 2.2: Variation de l'exécution et efficacité en fonction du taille du probleme.

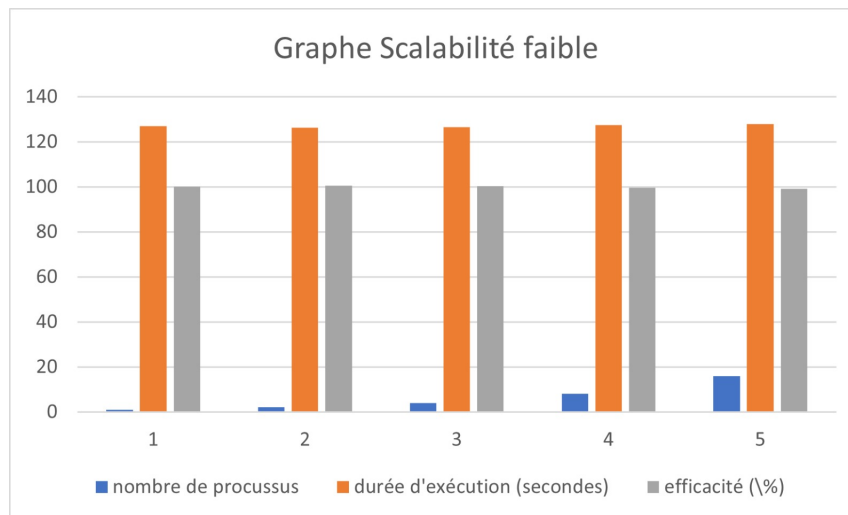


Figure 2.6: Graphe Scalabilité faible

Selon les résultats obtenus, nous remarquons que le temps d'exécution a peu changé, ce qui prouve que nous avons *une scalabilité faible*.

2.4.4 Après Optimisation

Après tous les implémentations d'optimisation que nous appliqués, nous avons étudié une deuxième fois la scalabilité du programme.

Scalabilité forte :

nombre de procussus	durée d'exécution (secondes)	efficacité (%)
1	18.963256	.
2	17.029014	55.67
4	14.507663	32,68
8	15.511376	15,28
16	16.567202	7,15

Table 2.3: Variation de l'exécution et efficacité en fonction du nombre de processus.

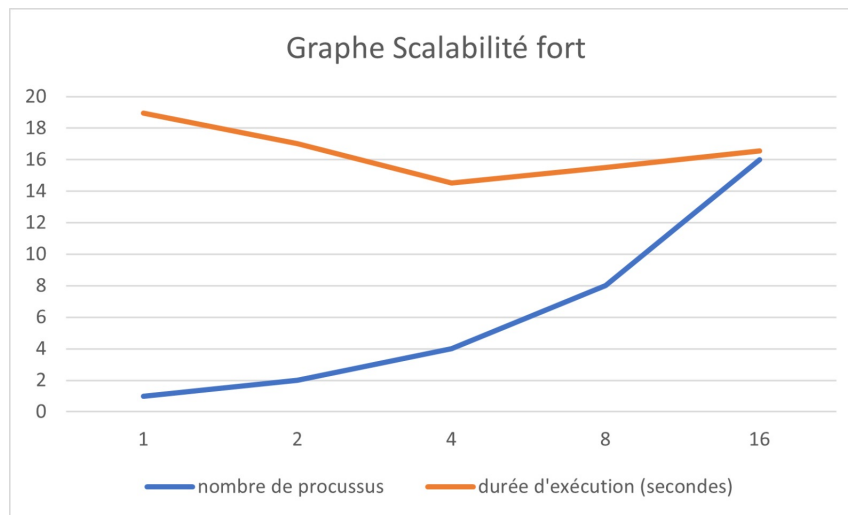


Figure 2.7: Graphe Scalabilité forte

On remarque que le temps d'exécution diminue un peu avec l'augmentation du nombre de processus jusqu'à 8, contrairement à la première étude qui nous a montré que la performance globale se dégrade avec l'augmentation des processus. Mais elle ne diminue pas linéairement, ce qui est imprévu, et cela peut être dû à plusieurs possibilités :

Soit notre programme nécessite encore une parallélisation ce qui est fortement possible, ou la communication entre les processus n'est pas bien faite.

Scalabilité faible :

Taille du problème	nombre de procussus	durée d'exécution (secondes)	efficacité (%)
50×50	1	16.874266	.
70×70	2	18.211939	92.65
100×100	4	19.541012	86.35
140×140	8	23.998967	70.31
200×200	16	25.791957	65.42

Table 2.4: Variation de l'exécution et efficacité en fonction du nombre de processus et de la taille du problème.

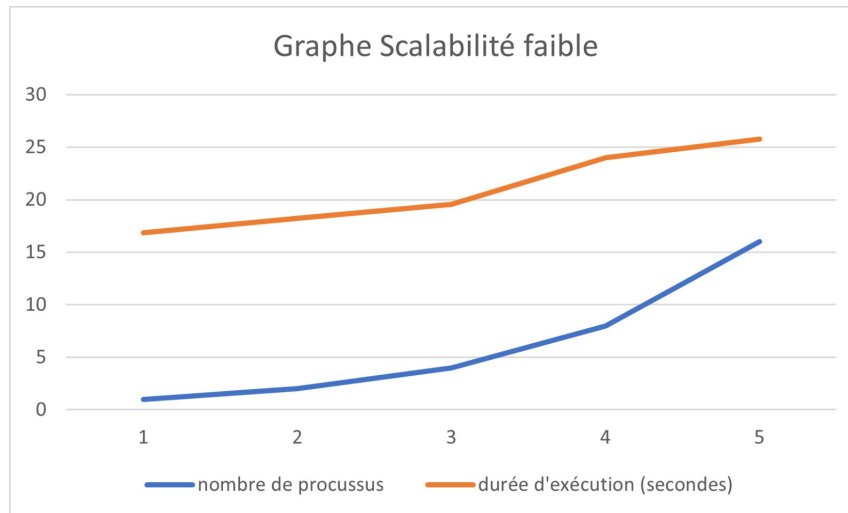


Figure 2.8: Graphe de Scalabilité faible

2.5 Conclusion

Dans la premier partie du projet, nous sommes partis d'un code exécuté sur un seul noeud et tres coûteux en terme de performance et manque d'optimisation.

Puis nous avons appliqué plusieurs piste d'optimisation, comme le parallélisme, le réglage de la mémoire, l'auto vectorisation, etc. pour faire réduire le temps d'exécution du code de 135.99812s d'origine à 11.211290s par itération. Il y a encore des opportunités d'optimisation qu'on peut apporter en parallélisation par MPI, le réglage de la vectorisation et d'autre chose qu'on peut le faire pour une future opportunité.

Finalement, ce projet nous a permis de mettre en pratique les techniques d'analyse et d'optimisation que nous avons vues au deuxième semestre, ainsi nous a fait connaître le but d'utilisation du calcul parallèle pour réduire le temps d'exécution et aussi fournir la possibilité de résoudre beaucoup de problème complexe dont la solution n'est pas possible lors d'utilisation d'un système informatique à processeur unique.

Chapter 3

Optimisation du code en séquentiel

3.1 Avant propos

Ce chapitre est une initiation des deux chapitres suivants qui sont consacrés à l'implémentation des approches *MPI* (dans le branches *Aicha_Optimization_MPI_Approachi*, avec *i* l'approche étudiée). Ainsi, les flags menant à une optimisation considérable dans le temps d'exécution vont être ajoutés dans le Makefile lors du test du fonctionnement de la partie *MPI*.

3.2 Flags de compilation: Options d'optimisation

On se propose dans cette partie de diminuer le temps d'exécution en ajoutant des flags de compilation pour le compilateur *g++*.

Pour se faire, on fixe le nombre d'Epochs et le taux d'apprentissage à 1 et 0.003, respectivement. La taille de chaque image du Data est égal à 50×50 (en cm^2). Le calcul étant séquentiel, l'exécution du code est faite sur un seul processus.

Le tableau (3.1) regroupe les flags de compilation ajoutés au fichier *Makefile*, dans *CFLAGS*.

Flag	Description	Temps Exec.
Default (gcc -Wall)	-	82.12 sec
-O2	Recommandé par Intel pour utilisation générale	11.76 sec
-O3	Optimisation (calcul intensif en virgule flottante)	10.95 sec
-Ofast	-O3 plus quelques extras	10.94 sec
-ffast-math	optimisation agressives en virgule flottante	13.08 sec
-ftree-vectorize	vectorisation automatique	11.65 sec

Table 3.1: Ajout de flags de compilation pour optimiser le temps de calcul.

Le speed-up suite à l'ajout des flags de compilation est illustré dans la figure (3.1).

Ainsi, on en déduit que:

- Le flag `-Ofast` est équivalent au flag `-O3` avec quelques extras. Cependant, le speed-up obtenu en utilisant chaque flag d'eux est presque comparable pour notre code. On conservera dans ce cas le flag de compilation `-O3`,
- Le flag `-ffast-math` peut être activé en utilisant le flag `-Ofast`,
- L'auto vectorisation est obtenue en utilisant le flag de compilation `-ftree-vectorize`, mais elle est incluse dans le flag de compilation `-O3`.

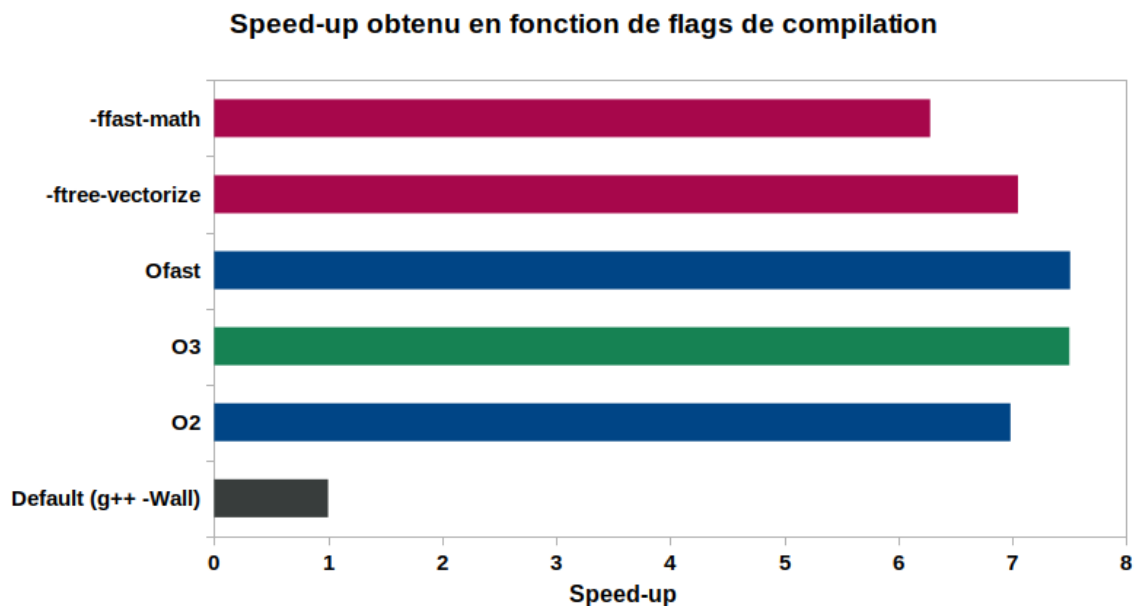


Figure 3.1: Speed-up obtenu en fonction de flags de compilation.

On conclut qu'on peut utiliser le flag de compilation `-O3` pour avoir un speed-up considérable du code d'environ 7.5%.

Chapter 4

Parallélisation du code avec *MPI*

4.1 Avant propos

Dans ce chapitre, l'analyse de performance (avec *OB1*, appendix A) est effectuée sur 3 nombre d'épochs pour des images de tailles 50×50 et 200×200 et un taux d'apprentissage égal à 0.003.

4.2 Structure du code original

Le code de la partie training est décrit par la figure (4.1).

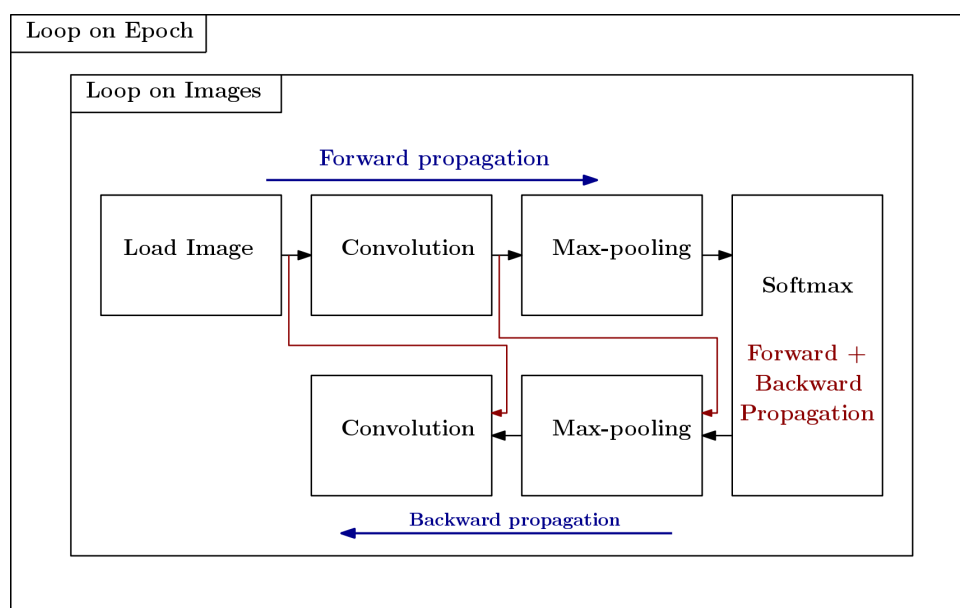


Figure 4.1: Structure du code original.

On a d'abord une première boucle sur le nombre d'épochs. A chaque epoch, on a une deuxième

boucle sur l'ensemble des images disponibles. Pour chaque image, on commence par la charger et réduire le nombre de canaux de 3 à 1. L'image réduite passe ensuite à la partie convolution. L'image convoluée passe ensuite à la partie max-pooling pour finir dans la couche softmax. Ces étapes sont effectuées ensuite dans le sens inverse dans la back-propagation. Il est important ici de noter que les parties max-pooling et convolution dans la back-propagation prenant en entrée non seulement les données issues de la couche softmax, mais aussi des entrées initiales comme l'indiquent les flèches rouges.

4.3 1ere tentative de parallélisation: Approche 1

4.3.1 Profilage du code avec *Gprof* et *Perf*

D'après les résultats de *Gprof* (illustrées dans la figure (4.2)), la partie convolution consomme environ 16% du temps d'exécution.

analysis_convolution_code_structuring.txt

1 Flat profile:

2

3 Each sample counts as 0.01 seconds.

4	%	cumulative	self	calls	self	total	name
5	time	seconds	seconds	s/call	s/call		
6	37.17	2.13	2.13	217175159	0.00	0.00	void std::vector<double, std::allocator<double>>::M_realloc_insert<double const&>(_gnu_cxx::__nor
7	15.88	3.04	0.91	6038	0.00	0.00	Convolution_layer::convolution_parameters(std::vector<double, std::allocator<double>> const&, int, i
8	13.79	3.83	0.79	3019	0.00	0.00	Softmax_layer::BackPropagation(std::vector<double, std::allocator<double>> const&, double)
9	10.65	4.44	0.61	6038	0.00	0.00	Softmax_layer::Softmax_start(std::vector<std::vector<double, std::allocator<double>>>, std::allocat
10	7.50	4.87	0.43	3019	0.00	0.00	Pooling_layer::BackPropagation(std::vector<std::vector<double, std::allocator<double>>>, std::allocat
11	5.93	5.21	0.34	6038	0.00	0.00	Pooling_layer::Pooling_parameters(std::vector<std::vector<double, std::allocator<double>>>, std::all
12	3.32	5.40	0.19	3019	0.00	0.00	Convolution_layer::BackPropagation(std::vector<std::vector<double, std::allocator<double>>>, std::all
13	2.79	5.56	0.16	117767	0.00	0.00	void std::vector<std::vector<double, std::allocator<double>>>, std::allocator<std::vector<double, std
14	1.75	5.66	0.10	6038	0.00	0.00	Data::create_canal(cv::Mat*)
15	1.22	5.73	0.07				_init
16	0.00	5.73	0.00	313976	0.00	0.00	void std::vector<double, std::allocator<double>>::M_realloc_insert<double>(_gnu_cxx::__normal_iter
17	0.00	5.73	0.00	54342	0.00	0.00	Data::loadImage(std::::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>> const
18	0.00	5.73	0.00	45311	0.00	0.00	void std::vector<int, std::allocator<int>>::M_realloc_insert<int const&>(_gnu_cxx::__normal_iterat
19	0.00	5.73	0.00	18114	0.00	0.00	std::vector<double, std::allocator<double>>::M_default_append(unsigned long)
20	0.00	5.73	0.00	6038	0.00	0.00	Pooling_layer::Hidden(std::vector<std::vector<double, std::allocator<double>>>, std::allocator<std::v
21	0.00	5.73	0.00	6038	0.00	0.00	Softmax_layer::Hidden()
22	0.00	5.73	0.00	6038	0.00	0.00	Convolution_layer::Hidden(std::vector<double, std::allocator<double>> const&)
23	0.00	5.73	0.00	6038	0.00	0.00	output::prediction(int, int&, int&)
24	0.00	5.73	0.00	6038	0.00	0.00	Data::get_fusion_canal() const
25	0.00	5.73	0.00	6038	0.00	0.00	std::vector<std::vector<double, std::allocator<double>>>, std::allocator<std::vector<double, std::all
26	0.00	5.73	0.00	26	0.00	0.00	void std::vector<std::::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>, std

Figure 4.2: Résultat de profilage du code original avec *Gprof*.

Cependant, *Gprof* donne comme résultat le temps passé des fonctions en mode exclusif, différemment à *Perf* qui est en mode inclusif. Le profilage du code avec *Perf* est montré dans la figure (4.3).

Samples: 75K of event 'cycles', Event count (approx.): 76698207298

Overhead	Command	Shared Object	Symbol
15,47%	program	program	[.] std::vector<double, std::allocator<double>>::M_realloc_insert<double const&>
14,04%	program	libc-2.31.so	[.] _int_free
11,66%	program	libc-2.31.so	[.] malloc
8,14%	program	program	[.] Convolution_layer::convolution_parameters
6,41%	program	program	[.] Softmax_layer::BackPropagation
5,67%	program	libc-2.31.so	[.] _int_malloc
4,94%	program	libc-2.31.so	[.] __memmove_avx_unaligned_erms
4,67%	program	libc-2.31.so	[.] cfree@GLIBC_2.2.5
4,23%	program	program	[.] Pooling_layer::BackPropagation
3,42%	program	program	[.] Pooling_layer::Pooling_parameters
2,80%	program	program	[.] Convolution_layer::BackPropagation
2,41%	program	libstdc++.so.6.0.28	[.] operator new
2,06%	program	libc-2.31.so	[.] malloc_consolidate
1,49%	program	program	[.] Softmax_layer::Softmax_start
1,20%	program	libc-2.31.so	[.] unlink_chunk@lsr.0
1,16%	program	program	[.] Data::create_canal

Figure 4.3: Résultat de profilage du code original avec *Perf*.

D'après la figure (4.3), on remarque que la couche de convolution consomme 8.14% du temps d'exécution.

4.3.2 Approche 1: Parallélisation du calcul de convolution

Suite aux résultats obtenus lors du profilage du code, on a essayé de commencer par améliorer le calcul dans cette couche en le parallélisant. Cette première approche est implémentée dans la branche *Aicha_Optimization_MPI_Approach1*.

Initialement, cette partie applique sur une image de taille initiale 8 filtres de taille 3×3 chacun. A chaque étape, le filtre est superposé à l'image. On multiplie les nombres dans la matrice représentant l'image d'entrée et ceux du filtre, on les somme et on place le résultat dans la matrice convoluée. Ensuite, le filtre se déplace par le *stride* (ici égal à 1) et on refait cette procédure, jusqu'au remplissage de la matrice convoluée de sortie, comme illustré dans l'image (4.4).

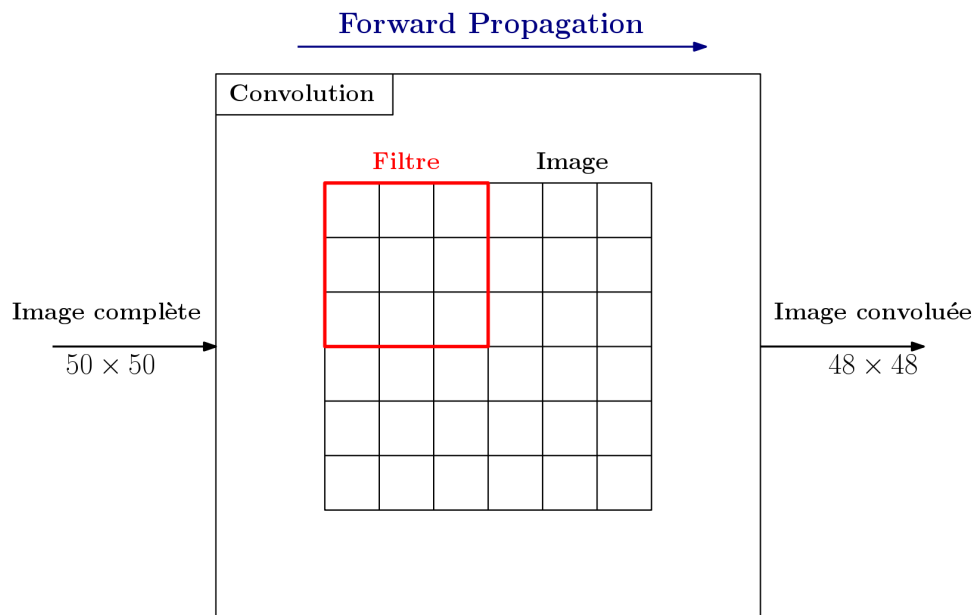
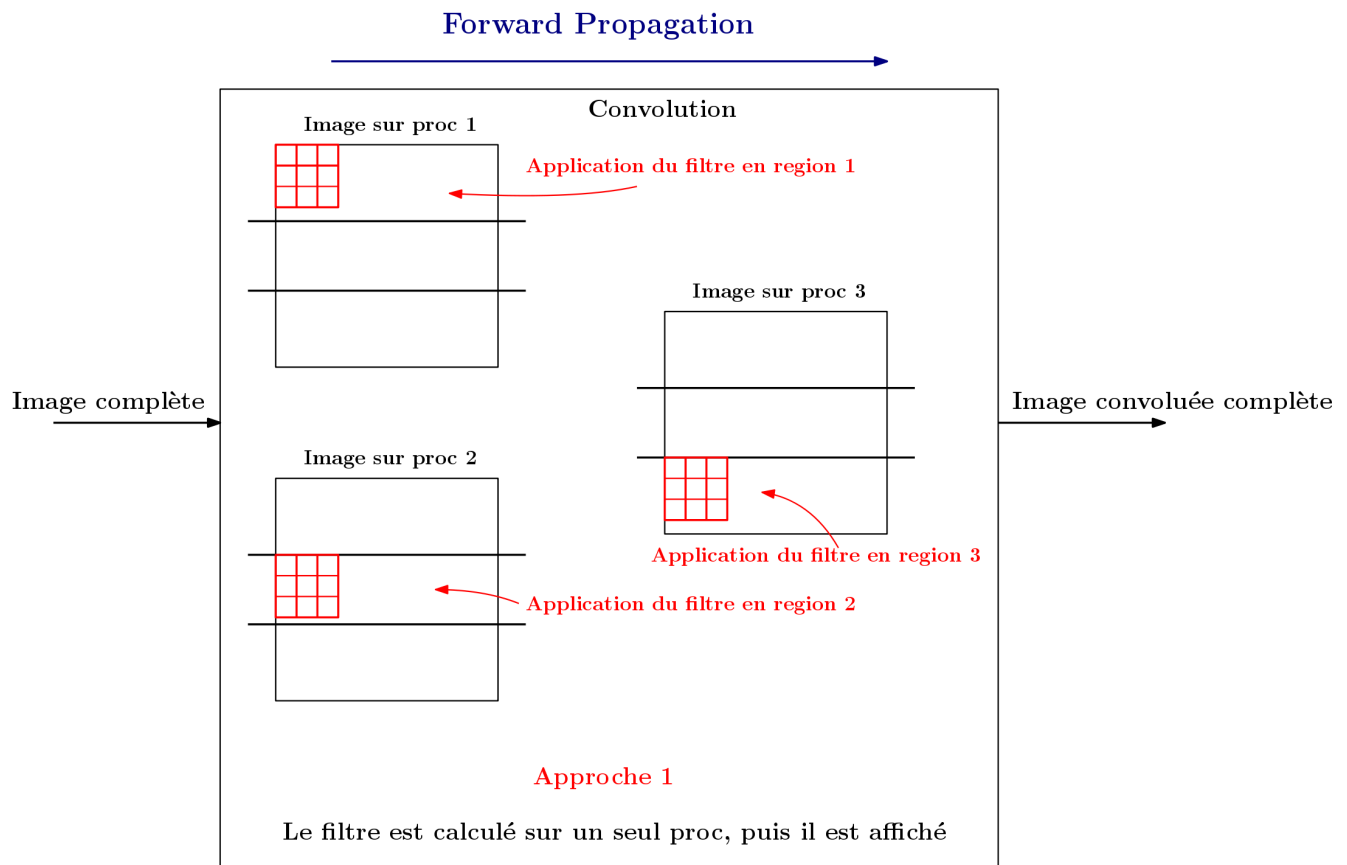


Figure 4.4: Illustration de la procédure de convolution.

Afin de paralléliser le calcul, on a lancé le code sur plusieurs processeurs. Donc, chaque processeur exécute la même chose que les autres et charge l' image entière. Le filtre cependant a été calculé sur un seul processeur (rank 0), ensuite partagé avec les autres (en utilisant *MPI_Bcast*). Avant de commencer le calcul de convolution, chaque processeur lui a été attribué une partie spécifique de l'image sur laquelle il va appliquer le filtre. La figure (4.5) illustre un exemple d'utilisation de 3 processeurs.



$$\text{Nombre de lignes dans chaque region} = \frac{\text{Nombre de lignes de l'image convoluée}}{\text{Nombre de processus}} + 2$$

Figure 4.5: Utilisation de 3 processeurs pour le calcul de la convolution.

Le découpage se fait toujours d'une manière horizontale. Ce découpage est adapté à notre structure de données car toute l' image est stockée horizontalement dans un seul vecteur à 1 dimension.

Le nombre de lignes attribuées à chaque processus est calculé par la formule suivante:

$$\text{Nombre de lignes dans chaque région} = \frac{\text{Nombre de lignes de l'image convoluée}}{\text{Nombre de processeurs}} + 2 \quad (4.1)$$

Il faut toujours laisser une ligne de plus en haut et une ligne de plus en bas pour l'application du filtre à l'image. Donc, on peut penser que cette méthode est plus adaptée aux grandes images qu'aux petites images. Par exemple, si on décompose une image 50×50 sur 16 procs, chaque processeur va prendre en charge 3 lignes sur lesquelles il va appliquer le filtre, plus 2 autres lignes. Ainsi, les résultats des différents processeurs sont rassemblées pour former l'image convoluée totale. On utilise pour ça la commande *MPI_Gatherv* qui va rassembler toutes les données dans le processus rank 0. Le calcul continue ensuite de manière normale sur le rank 0. On utilise

MPI_Gatherv et non pas *MPI_Gather* car cette commande rend possible un découpage qui n'est pas semblable pour tous les processeurs.

La figure (4.6) illustre le speed-up obtenu en utilisant cette première approche sur des images de tailles 50×50 et 200×200 .

Approche 1: Speed-up obtenu en fonction de la taille d'image et le nombre de processus

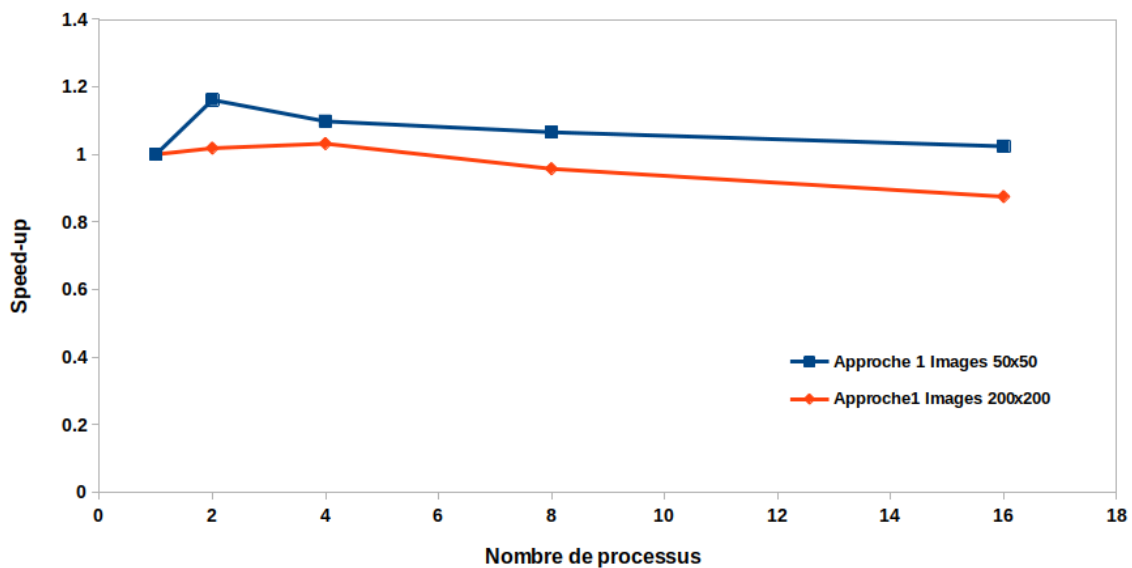


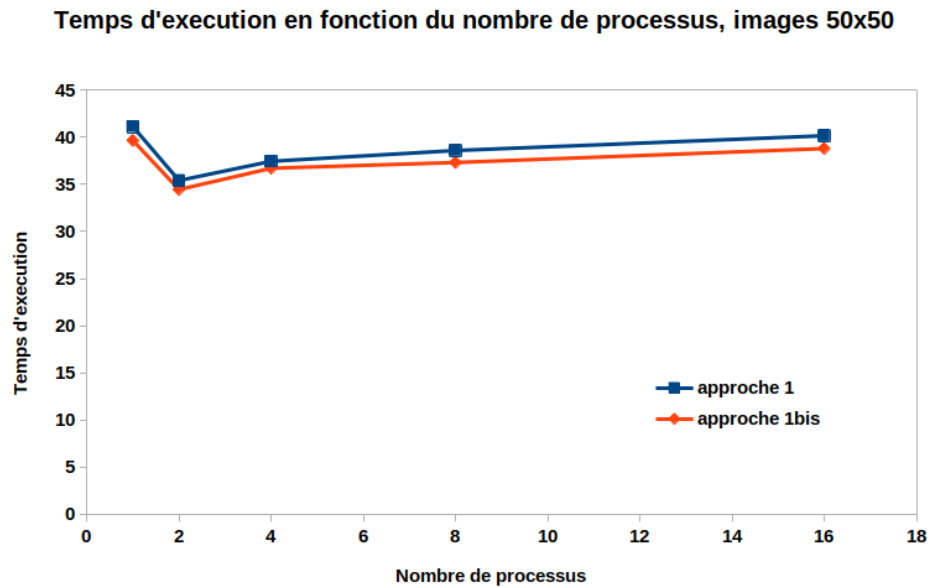
Figure 4.6: Speed-up obtenu en fonction du nombre de processus et taille des images d'entrée

On remarque que le speed-up obtenu dans le cas d'une image de taille 50×50 est plus grand que celle de taille 200×200 , ce qui est contraire à notre attentes. Parmi les hypothèses qui peuvent expliquer ce résultat inattendu, on peut citer:

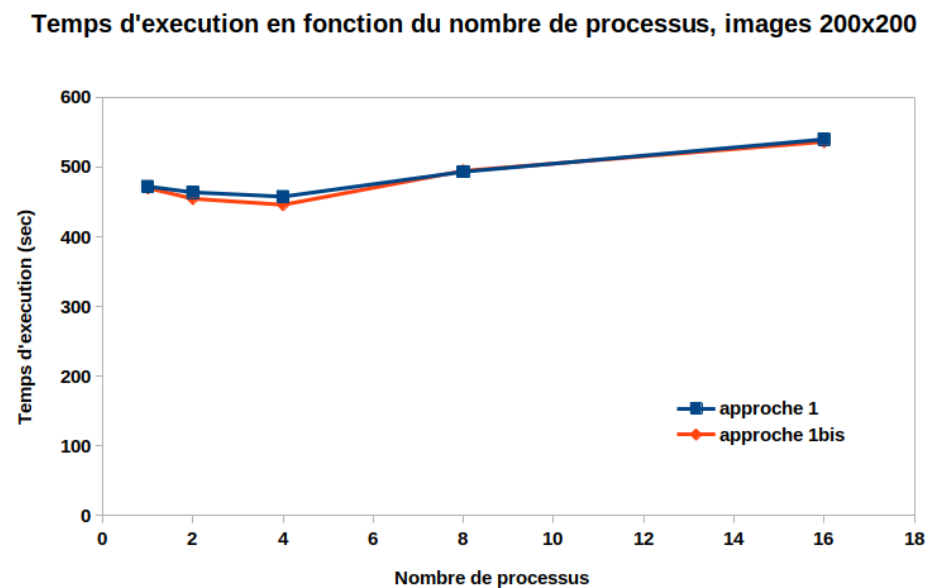
- le coût d'échange de données volumineux masque le coût gagné pendant le calcul,
- Il aussi probable que l'utilisation d'images plus grandes est accompagné par un surcoût dans d'autres parties du code qui masque le gain dans la couche de convolution

4.3.3 Approche 1bis: Tentative d'amélioration de l'approche 1

On a encore essayé d'optimiser cette approche. En réalité le calcul de filtre se fait 8 fois avec 8 filtres différents et donc le rassemblement de données avec *MPI_Gatherv* se fait de même 8 fois. Afin de réduire les communications MPI, on a essayé d'envoyer les résultats des 8 filtres en 1 seul bloc. La figure (4.7) montre le résultat du speed-up et on ne constate pas une grande amélioration au niveau temps d'exécution.



(a) Temps d'exécution en fonction du nombre de processus, 3 epochs, images 50×50 .



(b) Temps d'exécution en fonction du nombre de processus, 3 epochs, images 200×200 .

Figure 4.7: Comparaison du temps d'exécution avec les approches 1 et 1bis.

On peut constater d'après la figure (4.7) une légère amélioration du temps d'exécution. On peut aussi vérifier que tout au long de ces implémentations, la précision pour la même taille d'images varie peu, comme l'indique la figure (4.7). Mais pour le même nombre d'epochs, la précision est meilleure pour l'image 200×200 . Ceci peut s'expliquer par la présence de plus de détails. On note qu'initialement, le code d'origine avait une précision de 42.0338% pour 3 epochs et une taille d'image 50×50 .

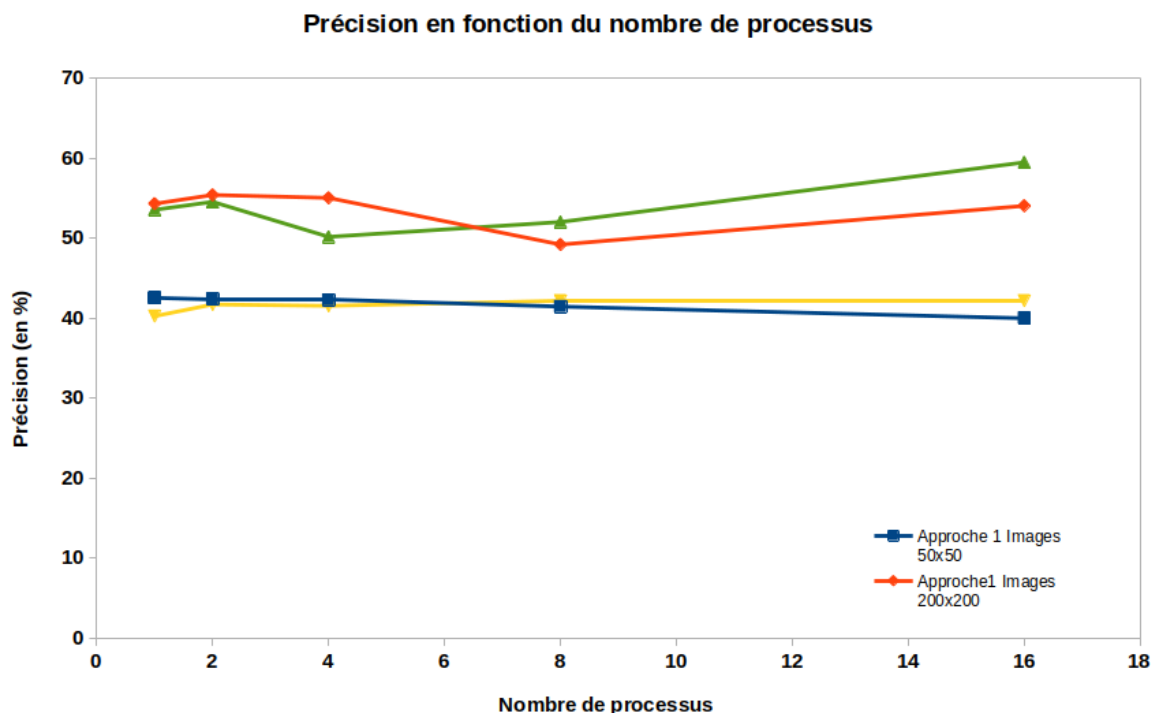


Figure 4.8: Précision obtenue en fonction du nombre de processus, pour les approches 1 et 1bis, tailles d'images 50×50 et 200×200 .

4.4 Approche 2: Décomposition de l'image

Jusqu'à maintenant, le programme est lancé sur plusieurs processus qui exécutent la même chose et seulement l'application du filtre dans la couche de convolution est parallélisée.

Il serait intéressant de décomposer l'image dès sa lecture. Ainsi, chaque processus reçoit uniquement un sous-domaine de l'image. L'application de cette approche est facile pour la couche de convolution et de max-pooling. Cependant, il n'est pas évident pour la partie softmax et calcul de backpropagation. On a essayé d'implémenter cette approche dans la branche *Aicha_Optimisation_MPI_Approach2*, mais malheureusement sans succès. La figure (4.9) montre la partie du code dans laquelle circule l'image décomposée.

L'image est lue (en utilisant *imread* de OpenCv) sur un seul processus, ensuite elle est décomposée et les sous-domaines sont partagés en utilisant *MPI_Scatterv*. Après la couche de max-pooling, les différents sous-domaines doivent être assemblés pour continuer la couche de softmax avec une entrée complète. La difficulté d'implémentation réside dans le rassemblement des entrées des parties max-pooling et convolution dans la back-propagation.

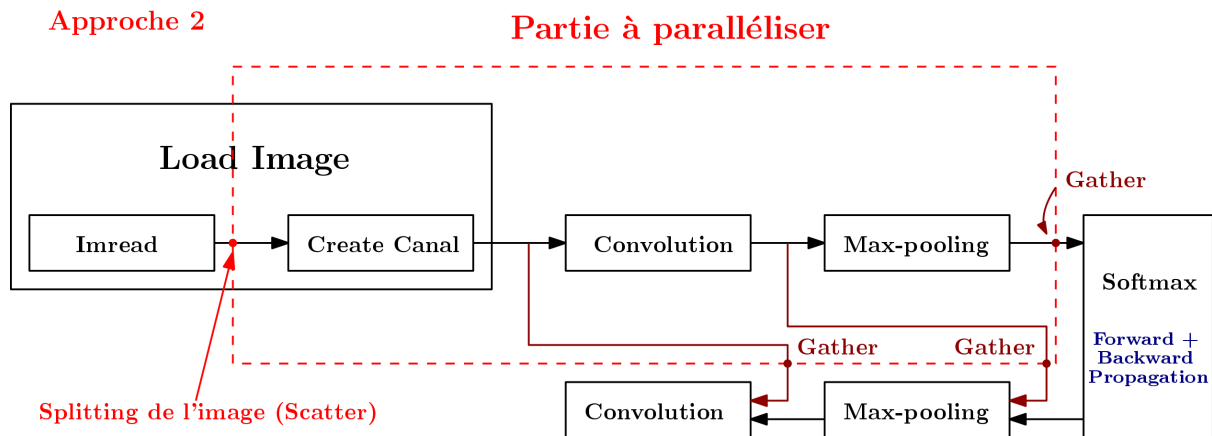


Figure 4.9: Partie a paralléliser dans l'approche 2.

4.5 Approche 3: Décomposition de l'ensemble des images

Puisque la décomposition de l'image n'a pas permis d'obtenir un speed-up aussi important, on a essayé une approche complètement différente. Au lieu de considérer l'image et sa taille comme un domaine à paralléliser, on considère ici l'ensemble des images à traiter. Les parties lecture d'image, convolution et max-pooling peuvent en réalité être exécutés une seule fois pour chaque image à la première itération.

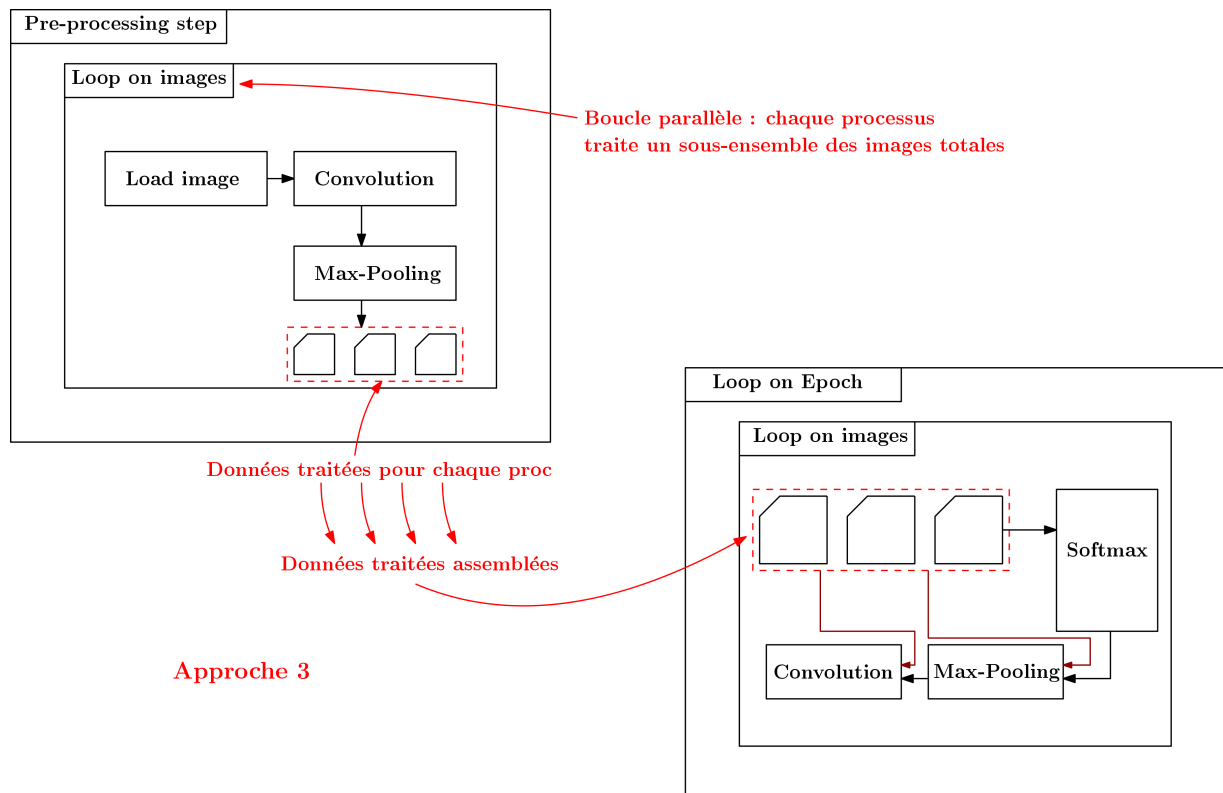


Figure 4.10: Partie a paralléliser dans l'approche 3.

Dans la figure (4.10), on a séparé ces calculs dans une étape de pré-processing. Les output de max-pooling ainsi que les résultats intermédiaires pour chaque image sont enregistrés dans des tableaux et sont ensuite utilisés par les autres étapes.

Ici c'est la boucle sur les images de pré-processing qui est parallélisée. Chaque processeur prend en charge un sur-ensemble des images, et communique après ces résultats au processus rank 0 qui continue le calcul de manière normale. Comme uniquement la partie de pré-processing est optimisée, on a mesuré les performances en prenant compte le temps d'exécution de cette dernière. Ainsi, le temps du pre-processing ainsi que le speed-up obtenu en fonction du nombre de processus sont illustres dans les figures (4.11) et (4.12), respectivement, pour des tailles des images 50×50 .

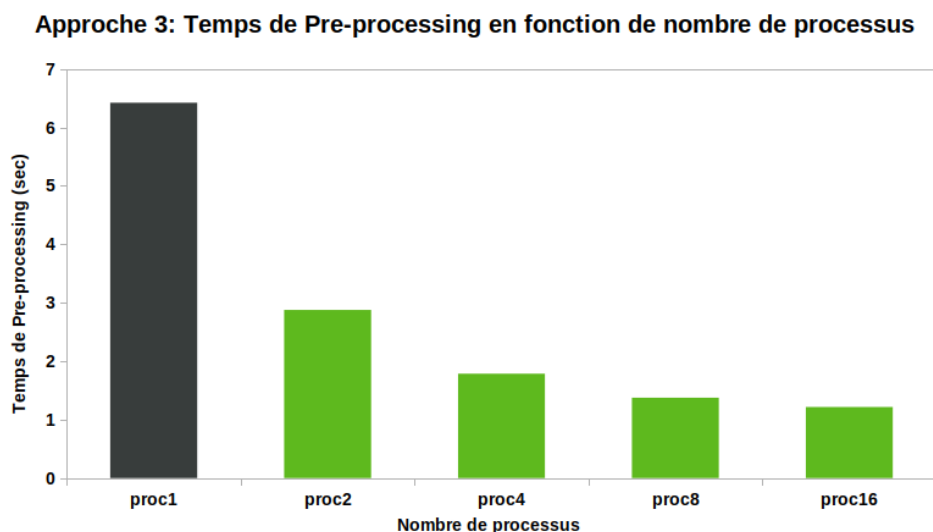


Figure 4.11: Temps de Pré-processing en fonction du nombre de processus.

On remarque d'après la figure (4.12) que la partie de pre-processing est fortement scalable jusqu'à 4 processus. Ensuite, le speed-up commence à se dégrader en le comparant au speed-up idéal. Le plateau est ainsi atteint pour 16 processus. Il est à noter également que la précision reste peu changeable lors de la parallélisation du code avec *MPI* lors de l'approche 3 comme montré dans la figure (4.13).

4.6 Conclusions

Toutes les optimisations du code original (partie convolution et Data) en utilisant *MPI* n'ont pas aboutit à une grande amélioration au niveau de speed-up. Il s'avertit ainsi en contemplant

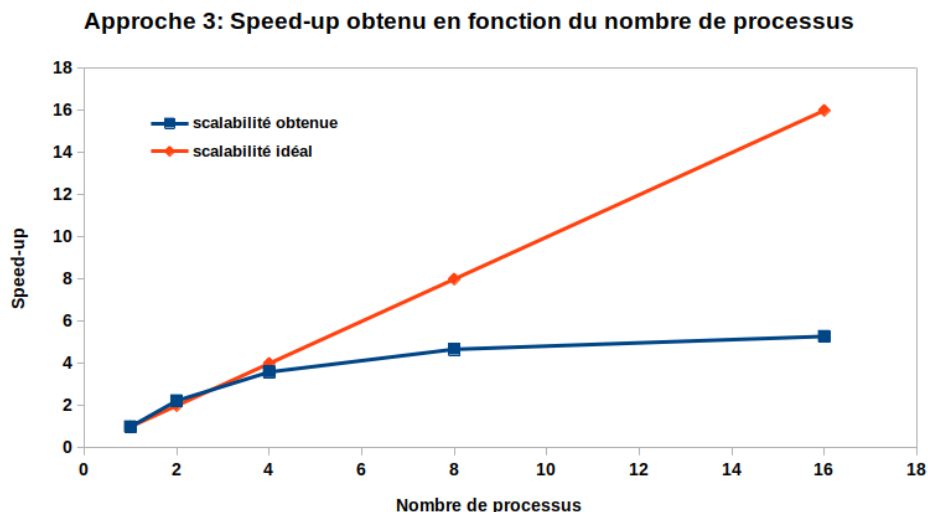


Figure 4.12: Analyse de scalabilité forte: Speed-up de la partie pre-processing obtenu en fonction de nombre de processus.

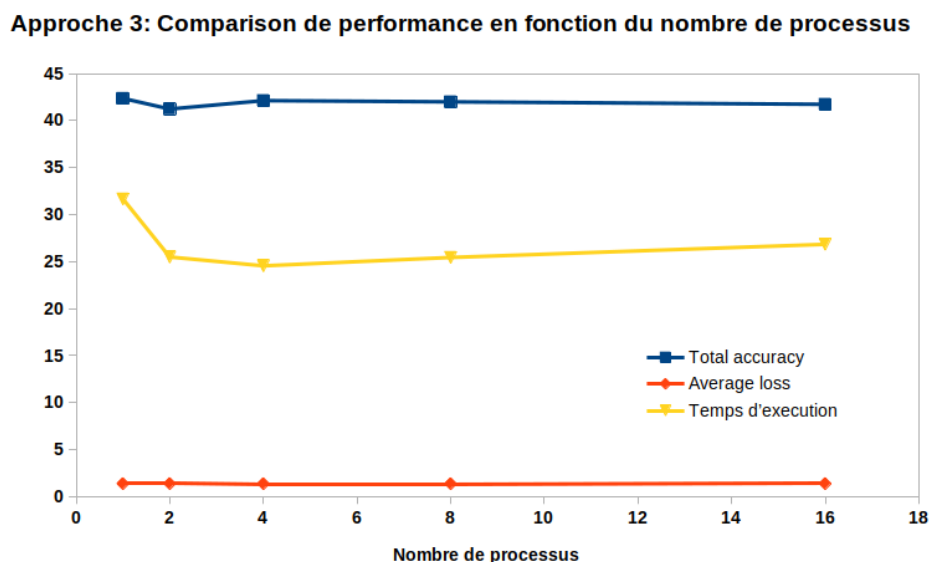


Figure 4.13: Evolution de la précision, perte et temps d'exécution en fonction de nombre de processus.

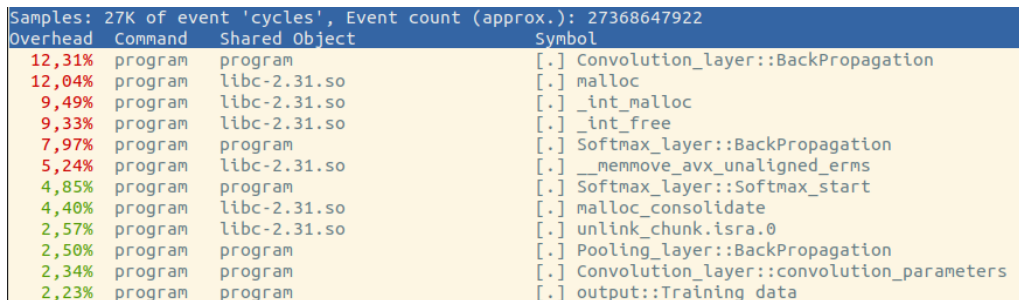
le code ainsi que les résultats de profilage avec *Perf* et *Gprof* (figures (4.2) et (4.3)) que la partie la plus consommande du temps d'exécution est dédiée au *Realloc*. Ceci est dû au fait d'utiliser le *push_back* dans les vecteurs du code. Il est à noter que lors de l'implémentation du code parallélisé avec *MPI*, tous les *push_back* de la partie convolution ont été remplacés: On fait une allocation a priori (on peut connaître à l'avance la taille nécessaire et l'utilisation de *push_back* se fait dans le cas ou on ne peut pas prédire la taille du vecteur), ensuite on fait une modification de valeurs. Le prochain chapitre sera donc dédié à la comparaison de speed-up suite à la suppression des différents *push_back*.

Chapter 5

Etude de l'effet de *Push_back*

Comme était décrit dans la conclusion du chapitre précédent, on a remplacé les *push_back* du code pour vérifier son influence sur le temps de calcul. Dans le code original, les vecteurs utilisés dans une couche de calcul (convolution, max-pooling, softmax) sont détruits par *vector::clear*, ensuite ils sont remplis petit à petit avec des *push_back*. Cette implémentation n'est pas optimale puisque l'espace est initialement réservé pour les vecteurs est dynamiquement adaptés, ce qui est coûteux. On a remplacé dans l'approche 3 les *push_back* dans tout le code dans la branche ***Aicha_Optimization_MPI_Approach3*** par des allocations à priori valables pour toutes les images et les epochs en faisant l'hypothèse que la taille de toutes les images est identique, ce qui est évidemment notre cas.

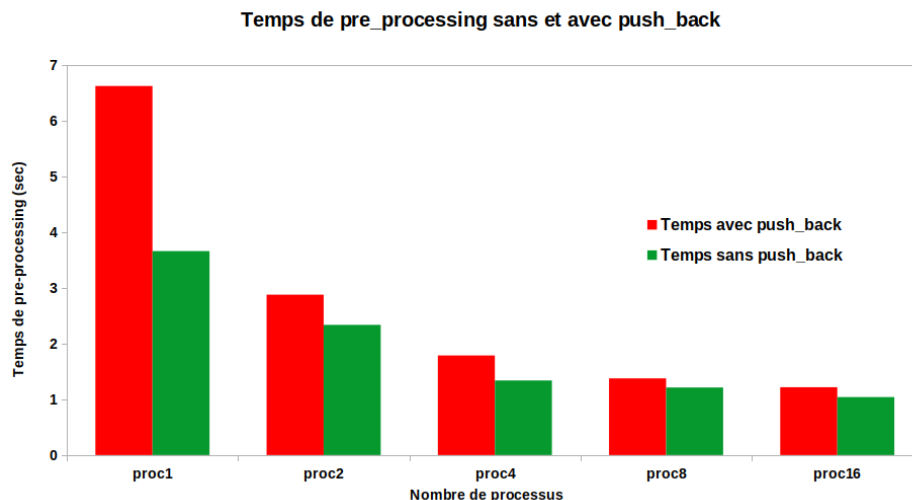
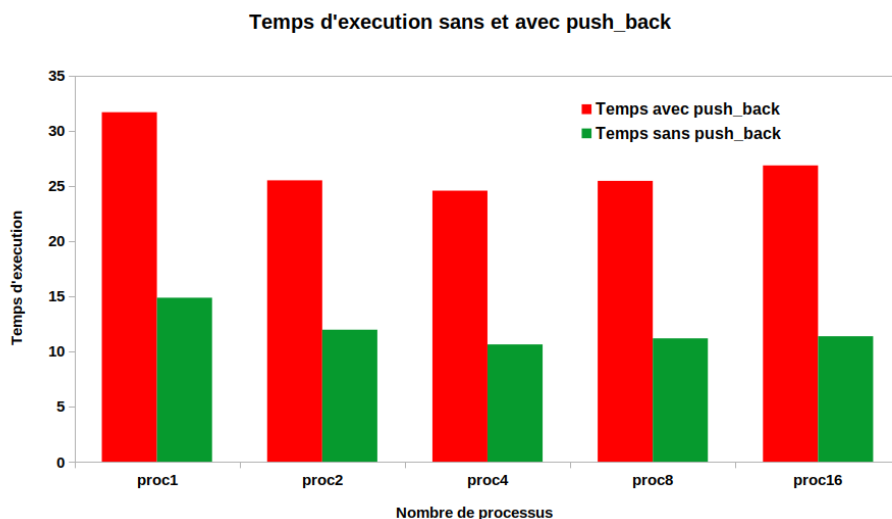
Le profilage après cette modification est illustré dans la figure (5.1). Comparé au profilage initial (voir figure (4.3)), la réallocation de la mémoire n'est plus l'opération la plus coûteuse du programme.



Samples: 27K of event 'cycles', Event count (approx.): 27368647922			
Overhead	Command	Shared Object	Symbol
12,31%	program	program	[.] Convolution_layer::BackPropagation
12,04%	program	libc-2.31.so	[.] malloc
9,49%	program	libc-2.31.so	[.] _int_malloc
9,33%	program	libc-2.31.so	[.] _int_free
7,97%	program	program	[.] Softmax_layer::BackPropagation
5,24%	program	libc-2.31.so	[.] __memmove_avx_unaligned_erms
4,85%	program	program	[.] Softmax_layer::Softmax_start
4,40%	program	libc-2.31.so	[.] malloc_consolidate
2,57%	program	libc-2.31.so	[.] unlink_chunk.isra.0
2,50%	program	program	[.] Pooling_layer::BackPropagation
2,34%	program	program	[.] Convolution_layer::convolution_parameters
2,23%	program	program	[.] output::Training_data

Figure 5.1: Structure du code original.

La réévaluation de l'approche 3 avec 3 epochs et des images de taille 50×50 aboutit aux résultats illustrés dans les figures (5.2) et (5.3), respectivement.

Figure 5.2: Comparison du temps de pre-processing avec et sans *push_back*.Figure 5.3: Comparison du temps d'exécution avec et sans *push_back*.

D'après les figures (5.2) et (5.3), on obtient un gain au niveau temps de pré-processing, mais aussi au niveau temps d'exécution, qui est réduit à moins que la moitié en fonction du nombre de processus.

5.1 Conclusion

Cette évaluation montre l'effet considérable de l'allocation dynamique mémoire sur le coût de calcul. C'est un point à considérer en priorité lors de la conception ou l'optimisation d'un code.

Même sans prendre en compte la parallélisation avec MPI, l'ajout des flags de compilation, la séparation de la partie pré-processing et l'enlèvement des *push_back*, permettent d'avoir un

speed-up de 21.2 (comparaison de l'approche 3 optimisée et le code initial, pour 10 epochs, images de taille 50×50 , taux d'apprentissage = 0.003). La parallélisation du preprocessing avec MPI et la parallélisation des boucles avec OpenMP permettent de gagner quelques facteurs en plus. Une réflexion sur la partie softmax et des couches de Back-propagation est nécessaire pour une parallélisation efficace afin d'améliorer davantage les performances.

References

- "*Loop Tiling*" (2021). wikipedia. URL: <https://huyenchip.com/2021/09/07/a-friendly-introduction-to-machine-learning-compilers-and-optimizers.html>.
- "*Optimisation et parallélisme avec OpenMP*" (2007). Article. URL: https://www.martinjucker.com/documents/FlashInformatique_32007.pdf.
- "*Profiler gprof*" (2004). Vinayak Hegde. URL: <https://ftp.traduc.org/doc-vf/gazette-linux/html/2004/100/lg100-L.html>.
- "*Profiler perf*" (2007). Article. URL: <https://linuxembedded.fr/2019/02/les-traceurs-sous-linux-22>.
- "*Scalabilité*" (2022). Enseeiht. URL: <https://hmf.enseeiht.fr/travaux/projnum/2020/programmation-parall%5C%c3%a8le-fortran-avec-mpi/passage-%5C%c3%5C%a0-1%5C%e2%5C%80%5C%99%5C%c3%5C%a9chelle-%5C%e2%5C%80%5C%93-scalabilit%5C%c3%5C%a9>.
- "*Scalability: strong and weak scaling*" (2016). bisqwit. URL: <https://bisqwit.iki.fi/story/howto/openmp/>.
- "*Scaling tutorial*" (2021). hpc-wiki. URL: https://hpc-wiki.info/hpc/Scaling_tutorial.

Appendix A

Relative à l'implémentation *MPI*

A.1 Description de la machine utilisée dans la partie *MPI*

Les analyses de performance relatives à la partie *MPI* ont été faites à l'aide du cluster *OB1-Exascale Computing Research cluster, snb03*, qui ont les caractéristiques montrées dans le tableau (A.1).

Nom du modele	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
Nombre de processuers	12
Nombre de threads	24
Nombre de threads par core	2
Taille du cache	30720 KB

Table A.1: Caractéristiques de la machine utilisée.

A.2 Branches d'implémentation *MPI*

La partie *MPI* a été implémentée dans les branches suivantes:

- **Approche 1:** branche *Aicha_Optimization_MPI_Approach1*,
- **Approche 2:** branche *Aicha_Optimization_MPI_Approach2*,
- **Approche 3:** branche *Aicha_Optimization_MPI_Approach3*,