Rapport Master M1

Master Calcul Haute Performance Simulation (CHPS)

Optimisation du Code CNN pour la Reconnaissance des Plantes

Réalisé par: Aicha Maaoui

Encadré par: Prof. Hugo Bolloré

Mai 2022

Institut des Sciences et Techniques des Yvelines (ISTY)

Abstract

Contents

List of Figures

List of Tables

Chapter 1

Optimisation du code en séquentiel

1.1 Profilage du code

Pour faire un profilage du code, on utilise *Perf*. Pour se faire, on ajoute le flage *-g* dans le *Makefile*.

1.2 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 (1.1) 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 1.1: 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 (1.1).

Ainsi, on en deduit 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*.

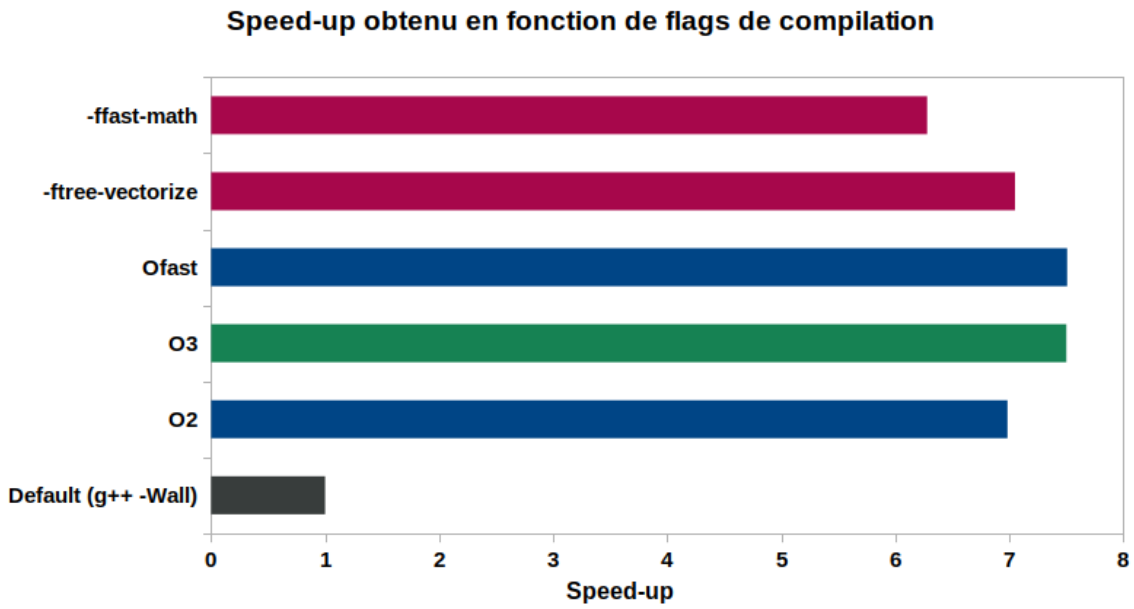


Figure 1.1: Speed-up obtenu en fonction de flags de compilation.

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

Chapter 2

Parallélisation du code avec *MPI*

2.1 Avant propos

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

2.2 Strucutre du code original

Le code de la partie training est décrit par la figure (2.1).

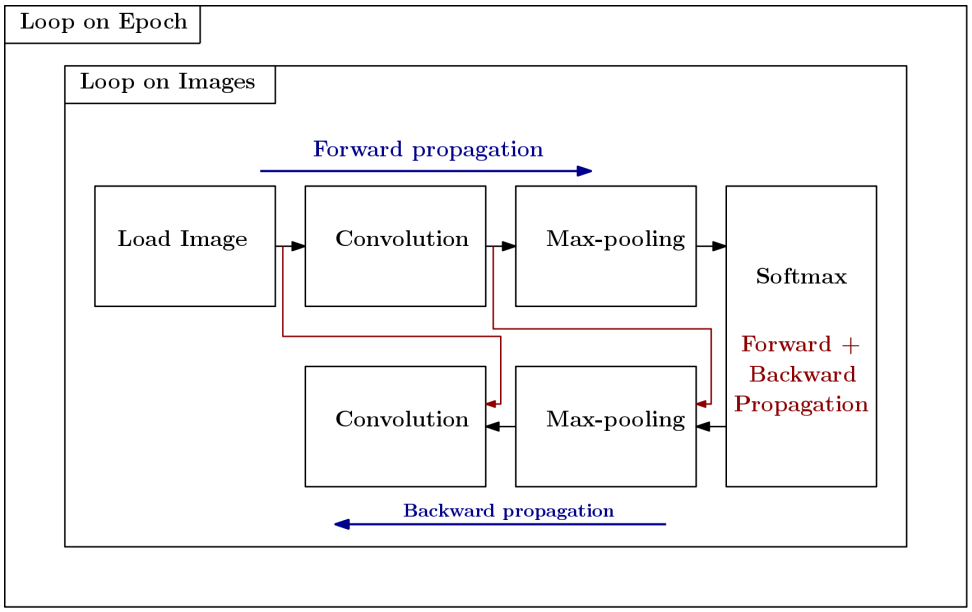


Figure 2.1: Structure du code original.

On a d'abord une première boucle sur le nombre d'epochs. 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.

2.3 1ere tentative de parallélisation: Approche 1

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

D'après les résultats de *Gprof* (illustrées dans la figure (2.2)), la partie convolution consomme environ 16% du temps d' exécution.

analysis_convolution_code_structuring.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 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
 15.88      3.04     0.91     6038     0.00     0.00  Convolution_layer::convolution_parameters(std::vector<double, std::allocator<double> > const&, int, i
 13.79      3.83     0.79     3019     0.00     0.00  Softmax_layer::BackPropagation(std::vector<double, std::allocator<double> > const&, double)
 10.65      4.44     0.61     6038     0.00     0.00  Softmax_layer::Softmax_start(std::vector<std::vector<double, std::allocator<double> >, std::allocator
  7.50      4.87     0.43     3019     0.00     0.00  Pooling_layer::BackPropagation(std::vector<std::vector<double, std::allocator<double> >, std::allocat
  5.93      5.21     0.34     6038     0.00     0.00  Pooling_layer::Pooling_parameters(std::vector<std::vector<double, std::allocator<double> >, std::allo
  3.32      5.40     0.19     3019     0.00     0.00  Convolution_layer::BackPropagation(std::vector<std::vector<double, std::allocator<double> >, std::all
  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
  1.75      5.66     0.10     6038     0.00     0.00  Data::create_canal(cv::Mat*)
  1.22      5.73     0.07                             _init
  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
  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
  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
  0.00      5.73     0.00    18114     0.00     0.00  std::vector<double, std::allocator<double> >::_M_default_append(unsigned long)
  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
  0.00      5.73     0.00     6038     0.00     0.00  Softmax_layer::Hidden()
  0.00      5.73     0.00     6038     0.00     0.00  Convolution_layer::Hidden(std::vector<double, std::allocator<double> > const&)
  0.00      5.73     0.00     6038     0.00     0.00  output::prediction(int, int&, int&)
  0.00      5.73     0.00     6038     0.00     0.00  Data::get_fusion_canal() const
  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
  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 2.2: 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 (2.3).

```
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.isra.0
   1,16%  program  program                      [.] Data::create_canal
```

Figure 2.3: Résultat de profilage du code original avec *Perf*.

D'après la figure (2.3), on remarque que la couche de convolution consomme 8.14% du temps d'exécution.

2.3.2 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 (2.4).

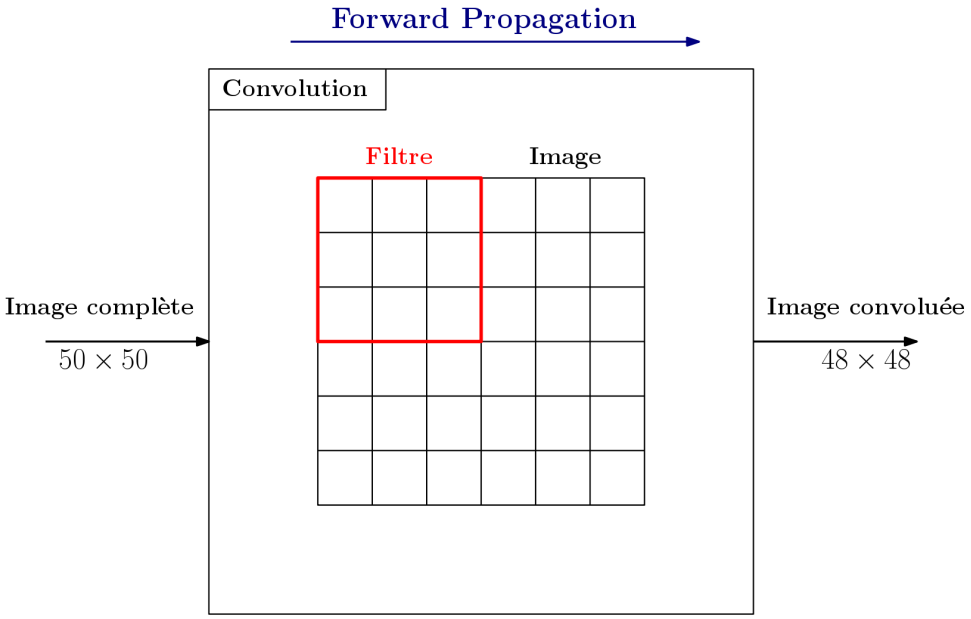
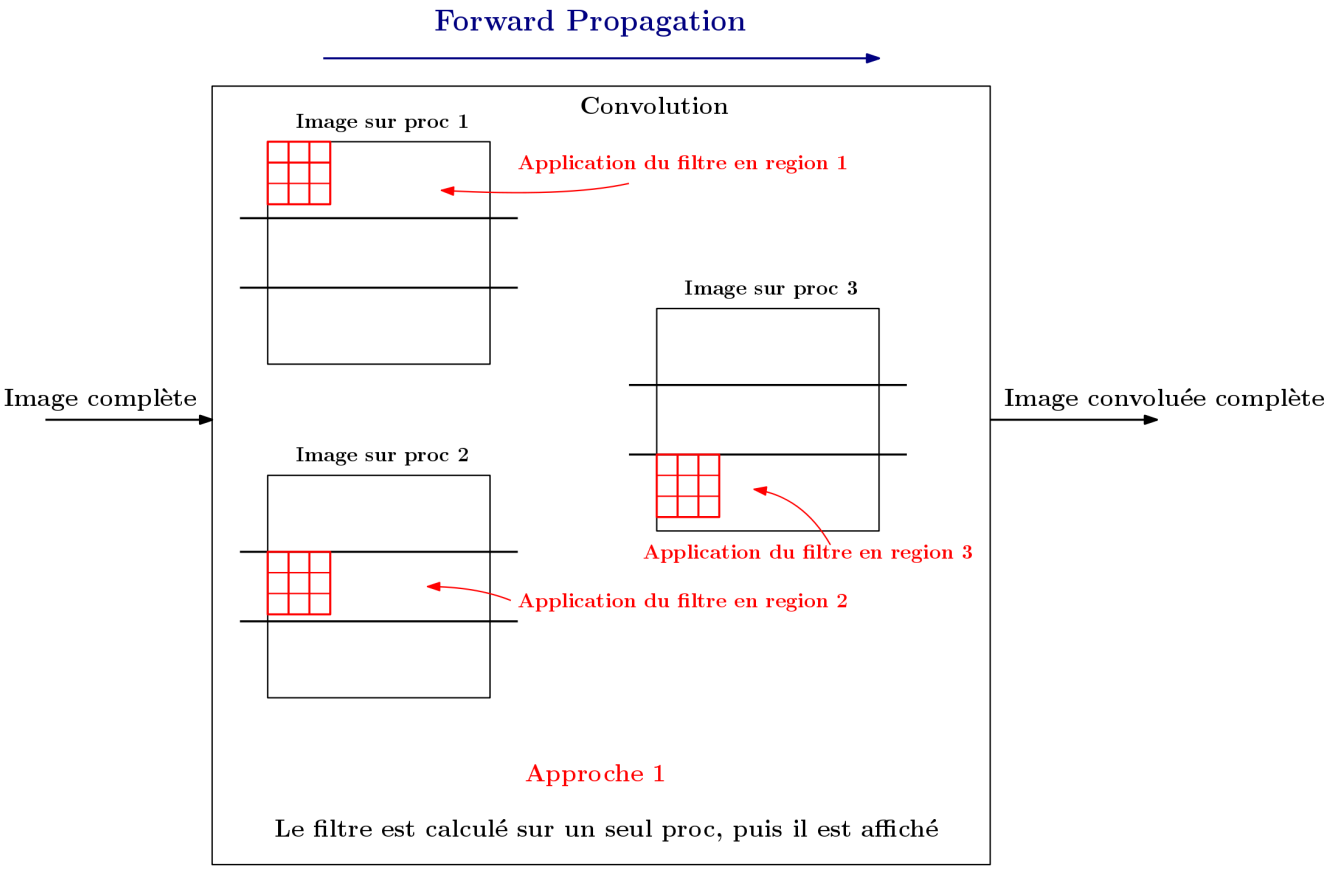


Figure 2.4: Illustration de la procédure de convolution.

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 (2.5) illustre un exemple d'utilisation de 3 processeurs.



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

Figure 2.5: 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:

$$\text{Nombre de lignes dans chaque région} = \frac{\text{Nombre de lignes de l'image convoluée}}{\text{Nombre de processeurs}} + 2 \qquad (2.1)$$

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

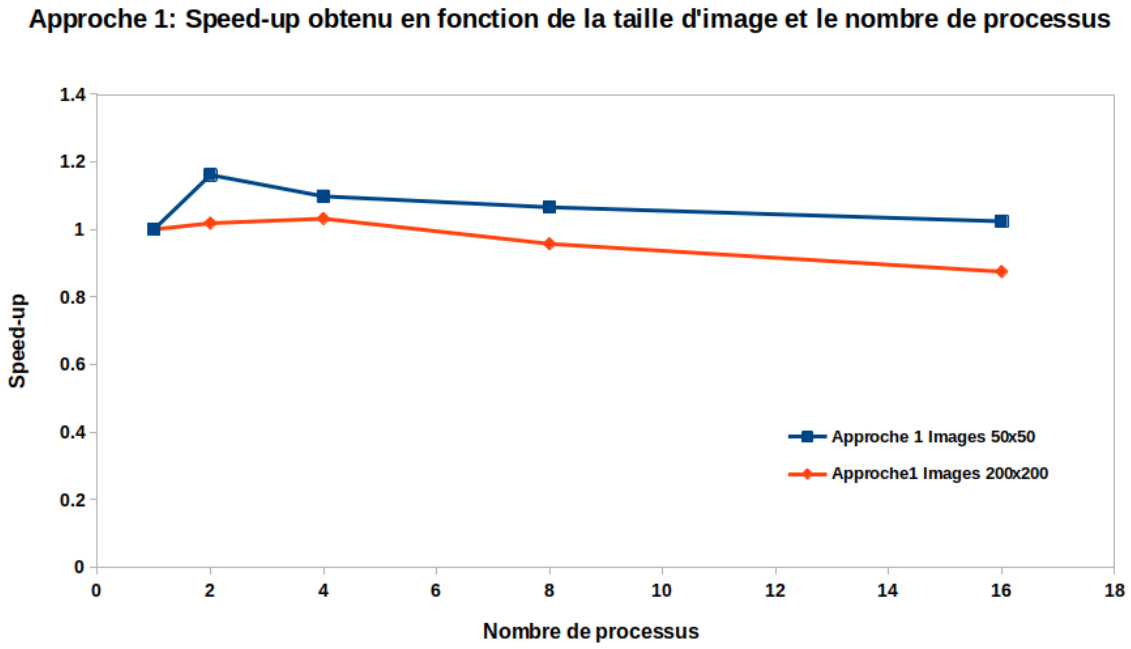


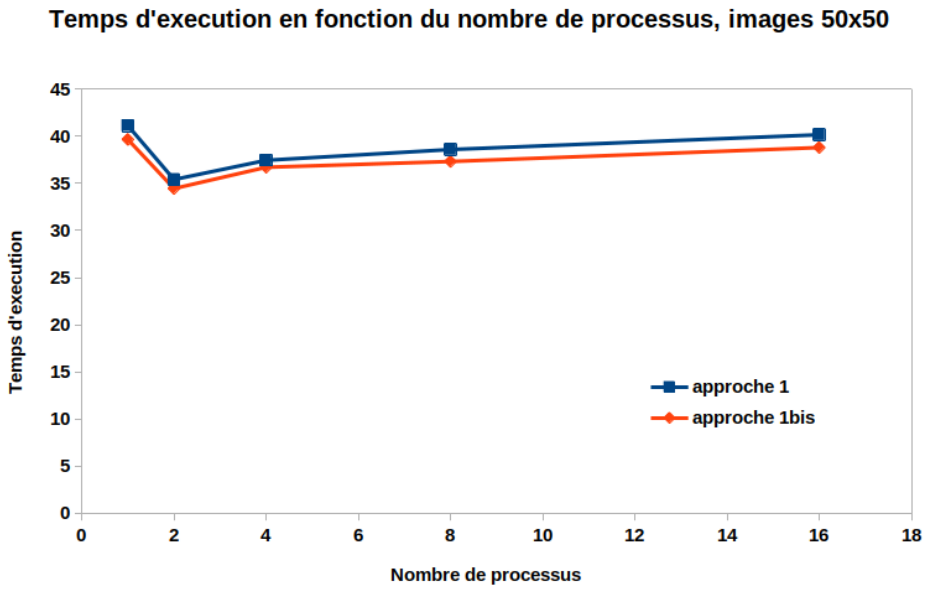
Figure 2.6: Speed-up obtenu en fonction du nombre de processus et taille des images d'entrée

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:

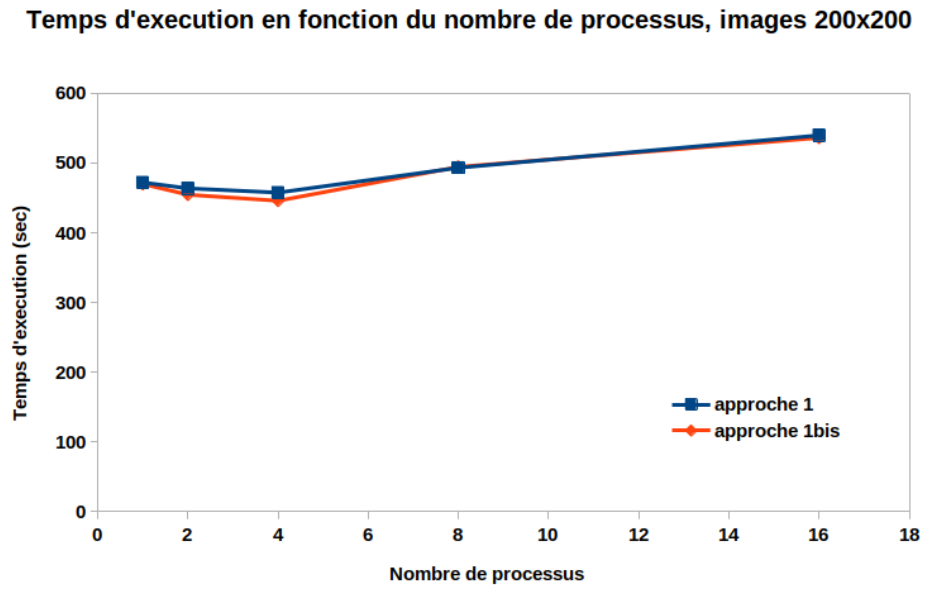
- le coût d'échange de données volumineux masque le coût gagné pendant le calcul,
- Il aussi probable que l'utilisation d'images plus grandes est accompagné par un surcoût dans d'autres parties du code qui masque le gain dans la couche de convolution

2.3.3 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, comme le montre le listing (2.1). La figure (2.7) 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 2.7: Comparison du temps d'exécution avec les approches 1 et $1bis$.

On peut constater d'après la figure (2.7) 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 peut, comme l'indique la figure (2.7). 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.

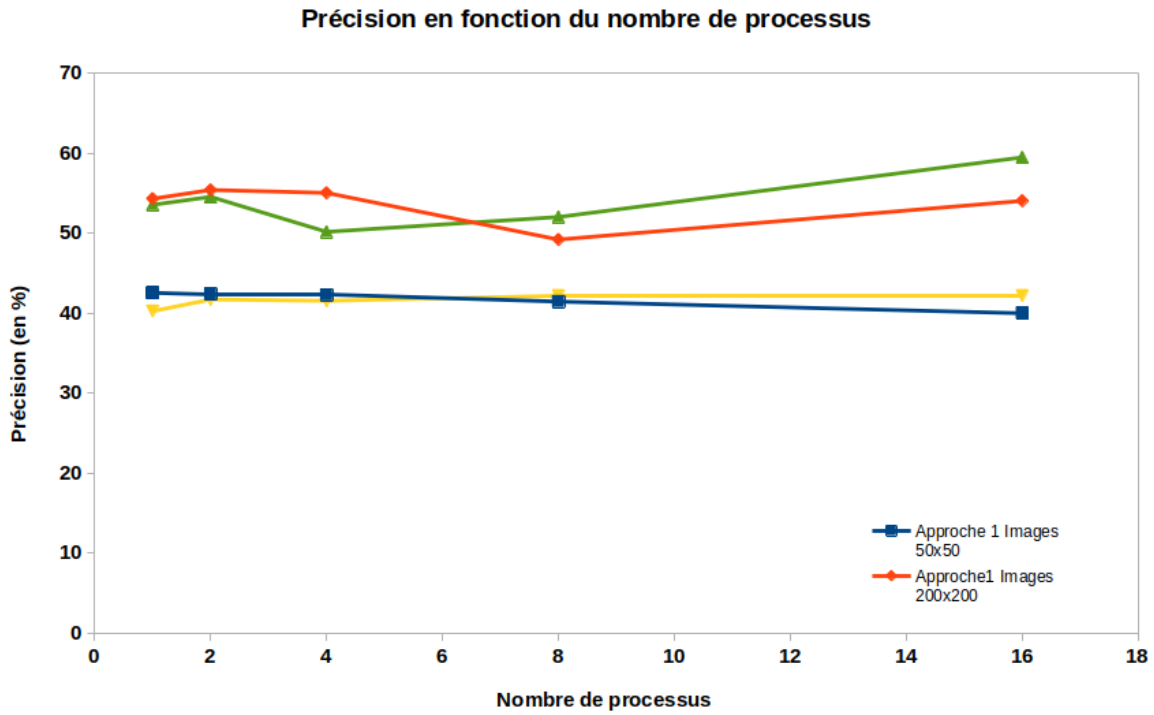


Figure 2.8: Précision obtenue en fonction du nombre de processus, pour les approches 1 et 1bis, tailles d'images 50×50 et 200×200.

2.4 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_Approach2***, mais malheureusement sans succès. La figure (2.9) 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 push 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.

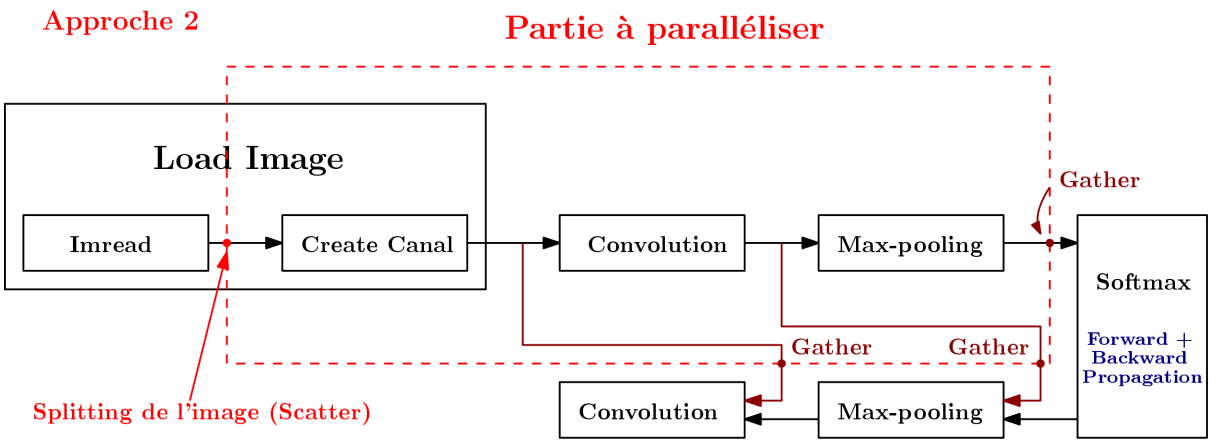


Figure 2.9: Partie a paralléliser dans l'approche 2.

2.5 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.

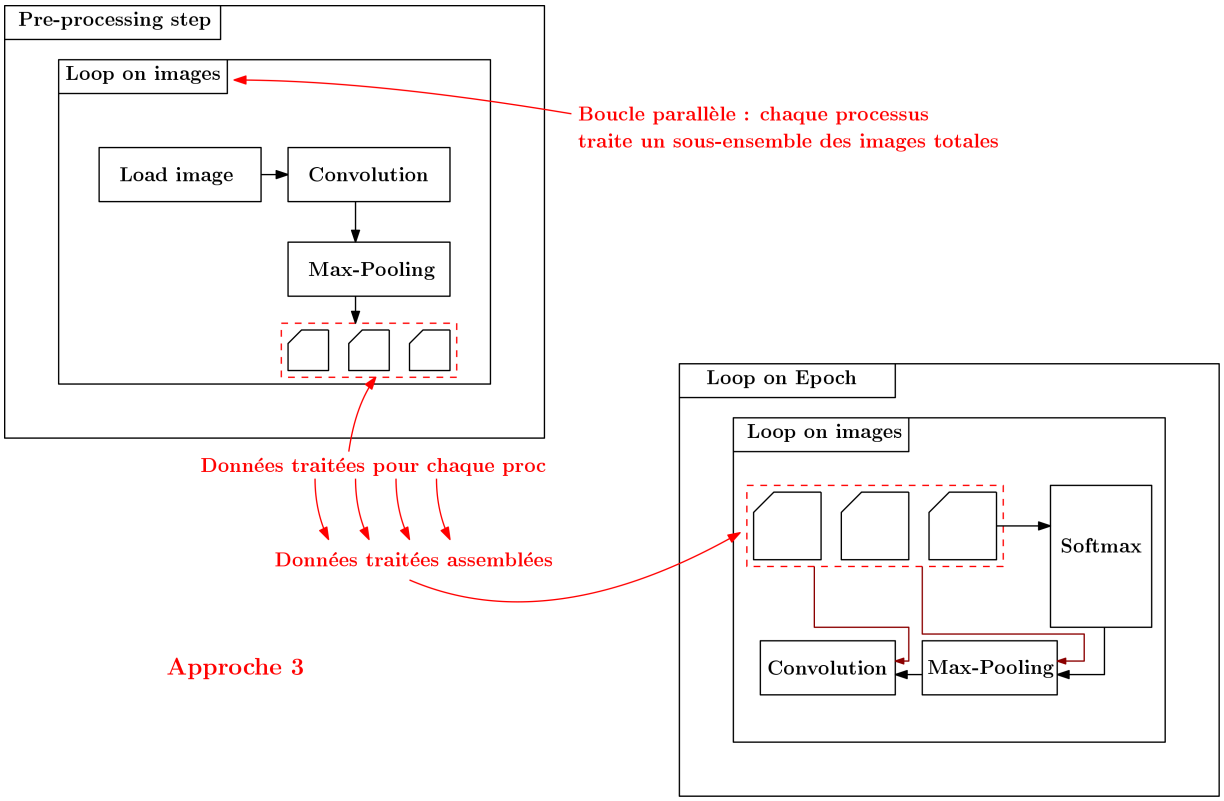


Figure 2.10: Partie a paralléliser dans l'approche 3.

Dans la figure (2.10), 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 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 (2.11) et (2.12), respectivement, pour des tailles des images 50×50.

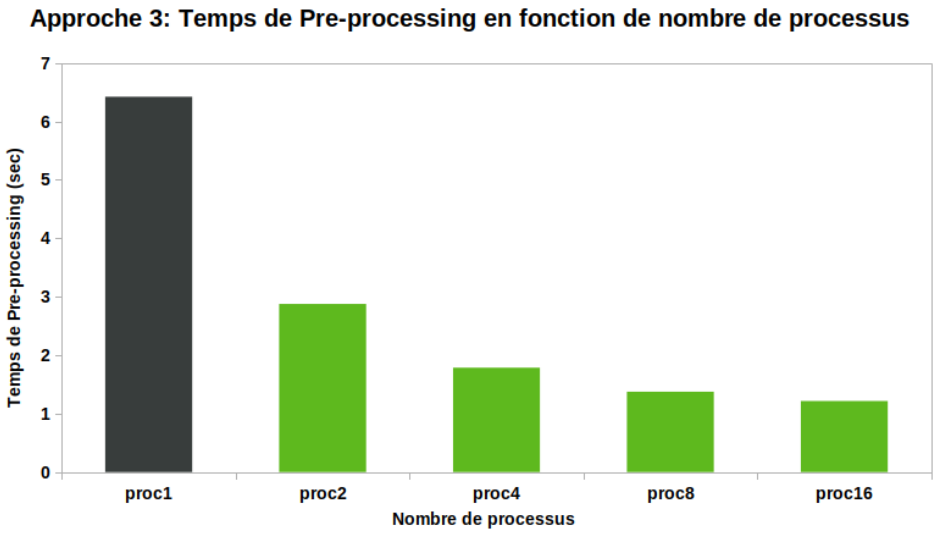


Figure 2.11: Temps de Pré-processing en fonction du nombre de processus.

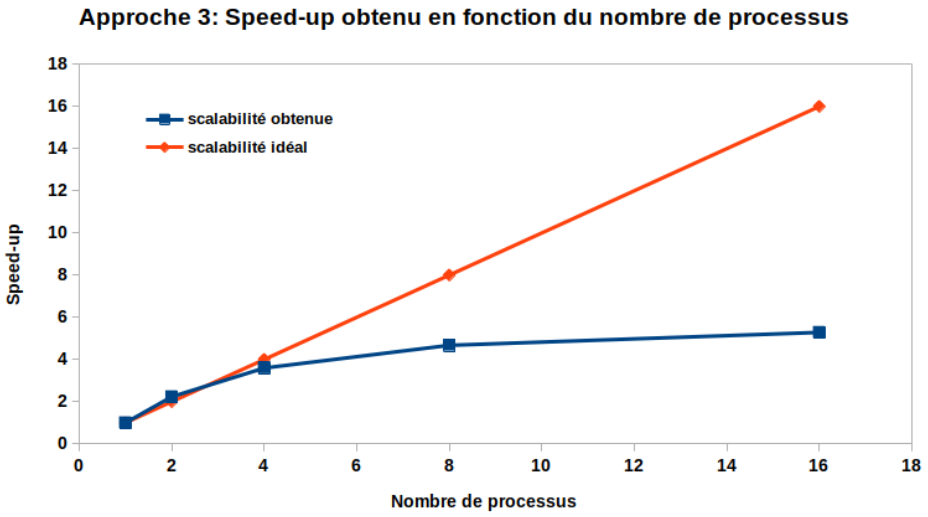


Figure 2.12: Analyse de scalabilité forte: Speed-up obtenu en fonction de nombre de processus.

On remarque d'après la figure (2.12) que le code est fortement scalable jusqu'à 4 processus. Ensuite, le speed-up commence à se dégrader en le comparant au speed-up idéal. Le plafond 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 (2.13).

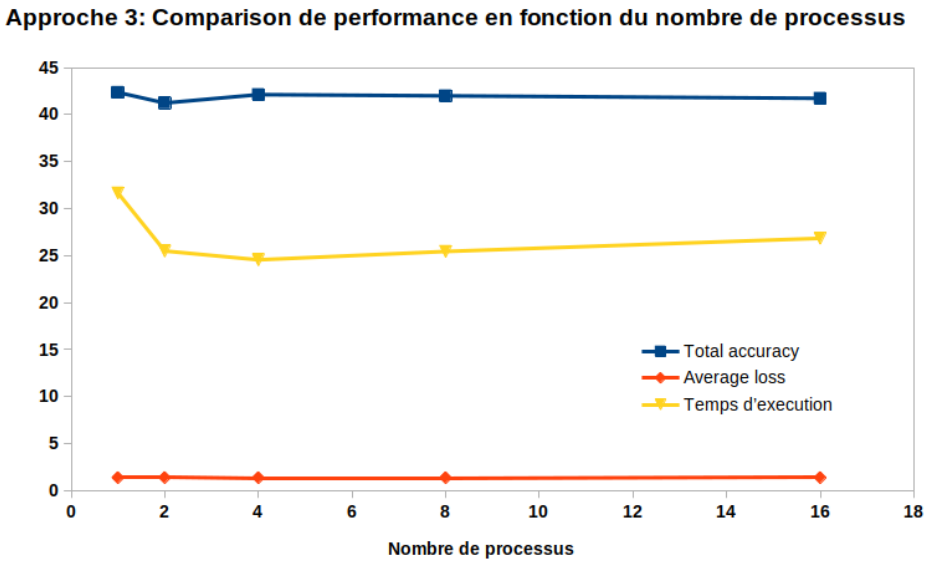


Figure 2.13: Evolution de la précision, perte et temps d'exécution en fonction de nombre de processus.

2.6 Conclusions

Toutes les optimisations du code original (partie convolution et Data) en utilisant *MPI* n'ont pas aboutit a une grande amélioration au niveau de speed-up. Il s'avertit ainsi en contemplant le code ainsi que les résultats de profilage avec *Perf* et *Gprof* (figures (2.2) et (2.3)) que la partie la plus consommante 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* .

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_Approach1***,
- **Approche 2:** branche ***Aicha_Optimization_Approach2***,
- **Approche 3:** branche ***Aicha_Optimization_Approach3***,