Université de Versailles
Saint-Quentin-en-Yvelines
Département de Informatique

Master 1 Calcul Haute Performance, Simulation

PPN- Profilage

Réalisé par :
- Bouchelga Abdeljalil
- Mohamed Aitmhand

Encadré par :
- Dr. Hugo Bolloré

Année Universitaire :2021-2022

TABLE DES MATIÈRES

TABLE DES FIGURES

PARTIE 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 commencé 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* aussi.

1.1.1 gprof

Gprof est un logiciel GNU Binary Utilities qui permet d'effectuer du profilage de code [3]
.

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<doub
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 [2].

Nous exécutons le programme avec la commande : **perf record ./lbm**

à l'aide de cette commande le programme généré 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 :

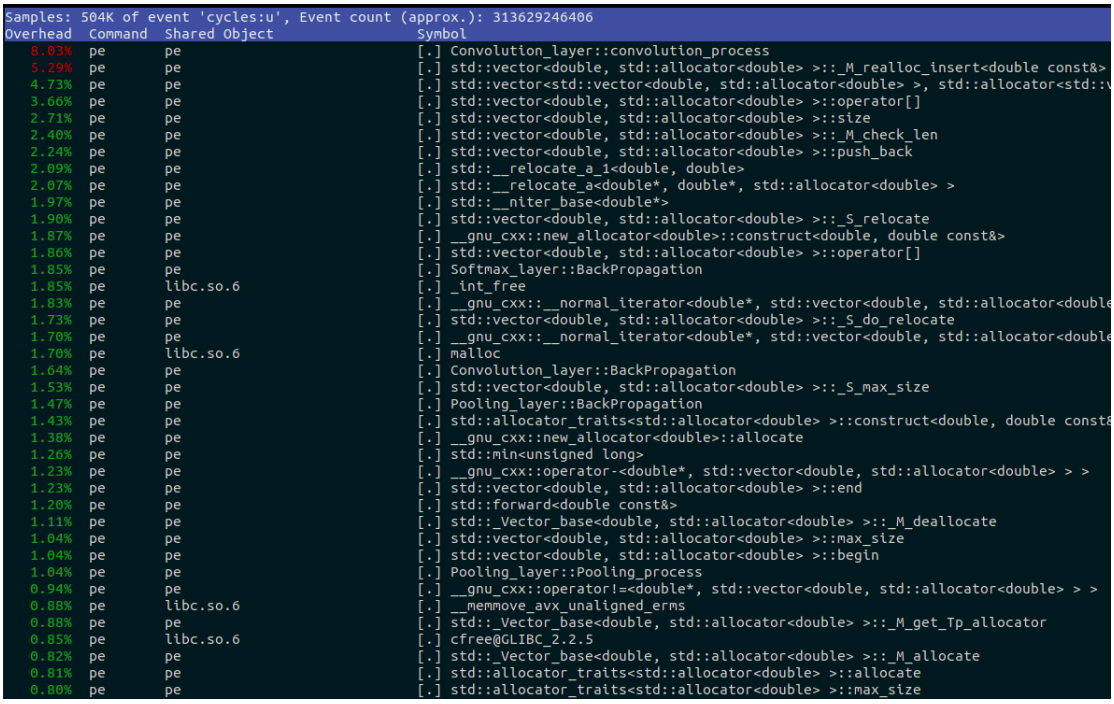
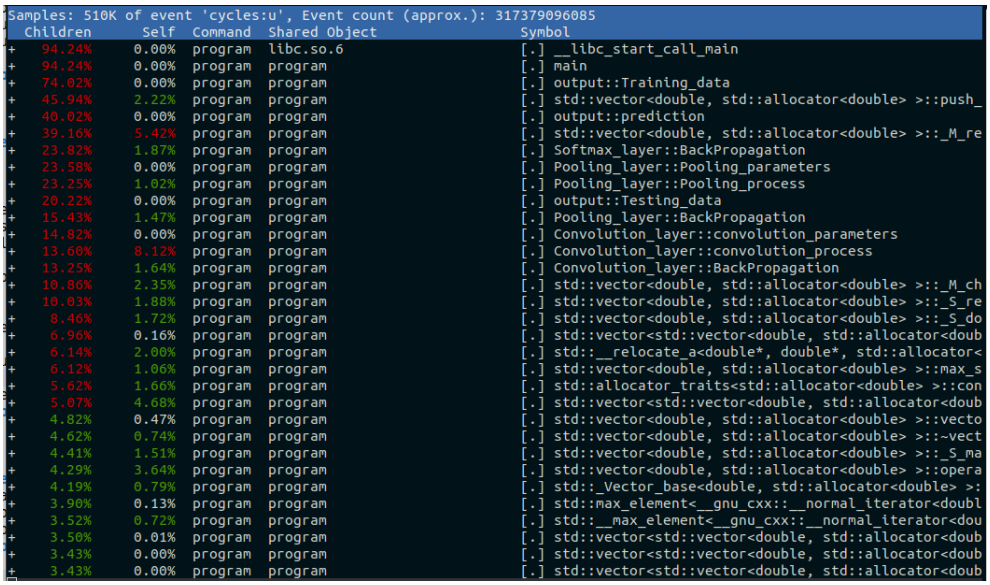


FIGURE 1.2 – Résultats de perf profiler

Par défaut, **perf** ne collecte que les informations temporelles. Pour ce faire, nous exécutons la commande suivante : **perf record -g ./lbm** 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.

BIBLIOGRAPHIE

[1] "Profiler gprof", https://ftp.traduc.org/doc-vf/gazette-linux/html/2004/100/lg100-L.html

[2] "Profiler perf", https://linuxembedded.fr/2019/02/les-traceurs-sous-linux-22

[3] "Profiler gprof", https://fr.wikipedia.org/wiki/Gprof