GLCS - Rendu du TD3

Yingqin SU 22006964
yingqin.su@ens.uvsq.fr
ISTY - M2 CHPS parcours informatique

29 Février 2025

Abstract

XcalableMP, abrégé XMP, est une extension du langage reposant sur des directives. Elle permet aux utilisateurs de concevoir aisément des programmes parallèles pour des systèmes à mémoire distribuée, tout en offrant la possibilité d'optimiser les performances grâce à une syntaxe à la fois minimale et simple.

Keywords XMP · OpenMP · MPI

1 Introduction

Le TD3 a pour objectif d'évaluer les effets dissipatifs de l'utilisation de Docker combinée à XcalableMP, en la comparant aux optimisations classiques réalisées avec gcc et OpenMP, avec et sans Docker. Pour ce faire, nous utilisons un cas pratique portant sur le calcul d'un histogramme. Dans l'environnement Docker, nous déployons quatre nœuds distincts ainsi qu'un environnement de compilation dédié à XcalableMP, sur lequel s'exécutent nos deux versions d'exécutables (XMP MPI versus OpenMP). Dans un premier temps, nous comparerons les performances obtenues avec différentes configurations (1, 2 et 4 nœuds), en prenant pour référence une configuration « gold standard » composée de 2 nœuds compilés avec le flag O3 et exploitant un fichier de 10 millions de données, afin d'estimer les gains de performance en pourcentage. Par ailleurs, nous analyserons également l'impact de la taille des fichiers sur ces performances.

2 Préparation

Pour mettre en place l'expérience, il faut avoir installer gcc, docker, docker compose pour lancer les images yml.

2.1 Environnement docker

Nous disposons d'un fichier Docker, fourni lors du CM5, qui permet de déployer, dans un environnement virtuel composé de 4 nœuds, une installation de XMP déjà configurée et préinstallée. Ce fichier génère notamment un conteneur « mpihead » ainsi que trois conteneurs « mpinode », chacun correspondant à un nœud, en plus de plusieurs services que nous n'utilisons pas dans le cadre de cette expérience. C'est dans ce contexte que nous réalisons nos tests.

Pour mettre en place l'environnement:

```
git clone https://gogs.eldarsoft.com/M2_IHPS/GLCS-CM5-2024
docker compose up
docker exec -it -u mpiuser nomdurepertoire-mpihead-1 /bin/bash
```

Nous utilisons l'option -u afin d'exécuter MPI en tant qu'utilisateur mpiuser, évitant ainsi de lancer MPI avec les privilèges du superutilisateur.

Tous les tests suivants seront exécutés dans l'environnement Docker, au sein du répertoire /usr/local/var/mpishare, qui sert de canal de communication pour les nœuds MPI.

2.2 Génération des données en entrée

Nous possédons un binaire nommé scramblegenerator qui, comme son nom l'indique, génère des nombres selon une distribution avec une dérivation. Cet outil offre les fonctionnalités suivantes :

```
Usage of ./scramblegenerator:
  -digitcountskew int
        The precision of the skew is determined by the number of digits (>10 and
<300). (default 20)
  -outputlinecount int
        Number of lines in the output data set (default 100)
  -randomhighint int
        Highest integer value for random data set. (default 20)
  -skewseed uint
        An 8-digit number seed to skew the distribution of integer values. (default
22111100)
  -valueperline int
        Number of values per line. (default 10)
```

Pour générer le fichier d'entrée de référence nécessaire aux tests, nous utilisons la commande suivante :

```
./scramblegenerator --skewseed 22006964 --outputlinecount 10000000 --valueperline 1 >
input.txt
```

Nous avons toutefois constaté que l'argument **valueperline** n'influence pas réellement la répartition des nombres par ligne dans le fichier **input.txt**, ce qui ne pose pas de problème pour nos calculs. Par conséquent, nous avons décidé d'utiliser la commande suivante :

```
./scramblegenerator --skewseed 22006964 --outputlinecount 1000000 > input.txt
```

Cette commande génère ainsi un fichier contenant 1 million de lignes, chacune comportant 10 valeurs comprises entre 0 et 19.

Ensuite, nous exécutons le binaire histo pour obtenir une sortie ayant deux ligne avec la première ligne les nombres allant de 0 à 19 et la deuxième ligne correspond au nombre d'occurrences de chaque valeur dans le fichier input.txt.

2.3 Lancement du code

Pour exécuter l'expérience, nous allons lancer le code suivant dans le répertoire /usr/local/var/mpishare:

```
make
mkdir -p output_prog
```

```
# Exécuter dans l'environnement docker
./histo 2> output_prog/output_xmp_n1
mpirun -np 2 -host mpihead,mpinode1 ./histo 2> output_prog/output_xmp_n2
mpirun -np 3 -host mpihead,mpinode1,mpinode2 ./histo 2> output_prog/output_xmp_n3
mpirun -np 4 -host mpihead,mpinode1,mpinode2,mpinode3 ./histo 2> output_prog/
output_xmp_n4

export OMP_NUM_THREADS=1
./histo_omp input.txt 2> output_prog/output_omp_n1
export OMP_NUM_THREADS=2
./histo_omp input.txt 2> output_prog/output_omp_n2
export OMP_NUM_THREADS=3
./histo_omp input.txt 2> output_prog/output_omp_n3
export OMP_NUM_THREADS=4
./histo_omp input.txt 2> output_prog/output_omp_n4

# Exécuter sans l'environnement docker
export OMP_NUM_THREADS=1
./histo_omp input.txt 2> output_prog/output_omp_no_docker_n1
export OMP_NUM_THREADS=2
./histo_omp input.txt 2> output_prog/output_omp_no_docker_n2
export OMP_NUM_THREADS=3
./histo_omp input.txt 2> output_prog/output_omp_no_docker_n3
export OMP_NUM_THREADS=4
./histo_omp input.txt 2> output_prog/output_omp_no_docker_n4
make clean
```

Pour avoir les informations des erreurs fusionnés, nous pouvons également utilisé l'argument: `--mca orte_base_help_agregate 0`

3 Résultat

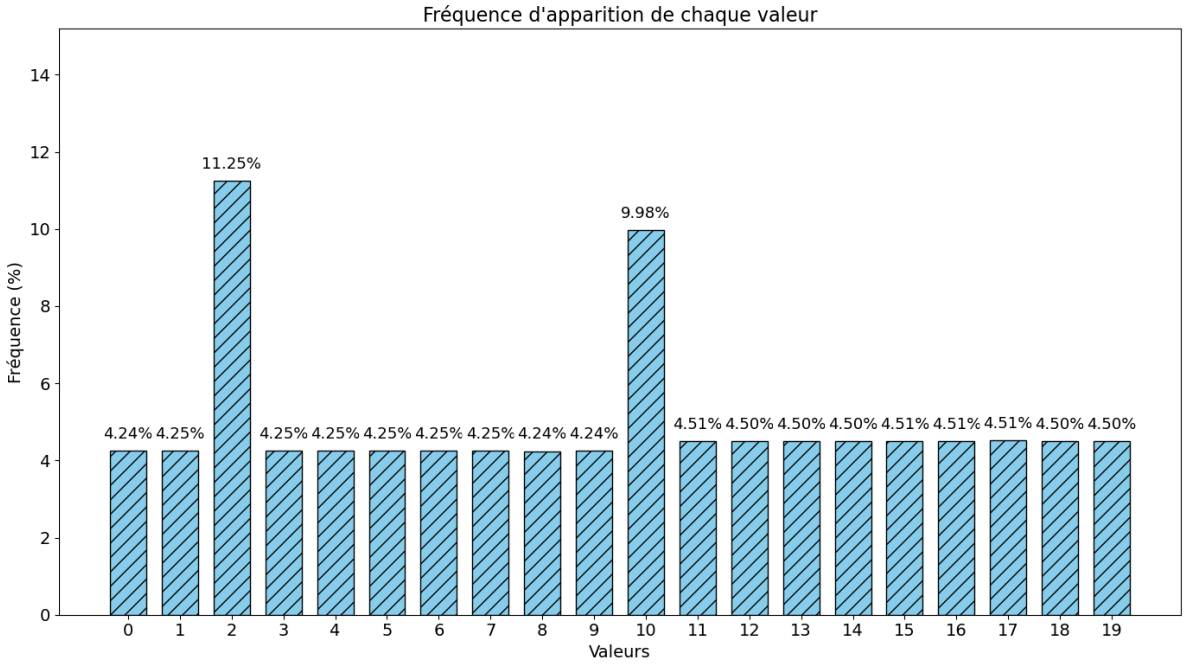
3.1 Distribution des valeurs

Nous allons tout d'abord créer une image illustrant la distribution. Pour simplifier cette étape, nous avons fait appel à ChatGPT o3-mini-high pour nous apporter son aide.

Prompt :

```
Peux-tu générer un script Python qui accepte deux arguments en entrée ? Le premier
argument correspond au nom d'un fichier et le deuxième à un entier n. Ce fichier
contient exactement deux lignes : la première ligne liste des valeurs, et la deuxième
ligne indique le nombre d'occurrences pour chaque valeur. Ton script devra d'abord
vérifier que la somme totale des occurrences correspond bien à n, puis créer une
image où l'axe des abscisses représente les valeurs et l'axe des ordonnées affiche,
en pourcentage (avec quatre décimales), la fréquence d'apparition de chaque valeur.
```

Nous avons obtenu l'image suivante après avoir envoyé quelques prompts pour améliorer son rendu visuel :



Ce résultat est généré par le script Python generation_image.py. Pour obtenir le résultat, utilisez la commande suivante :

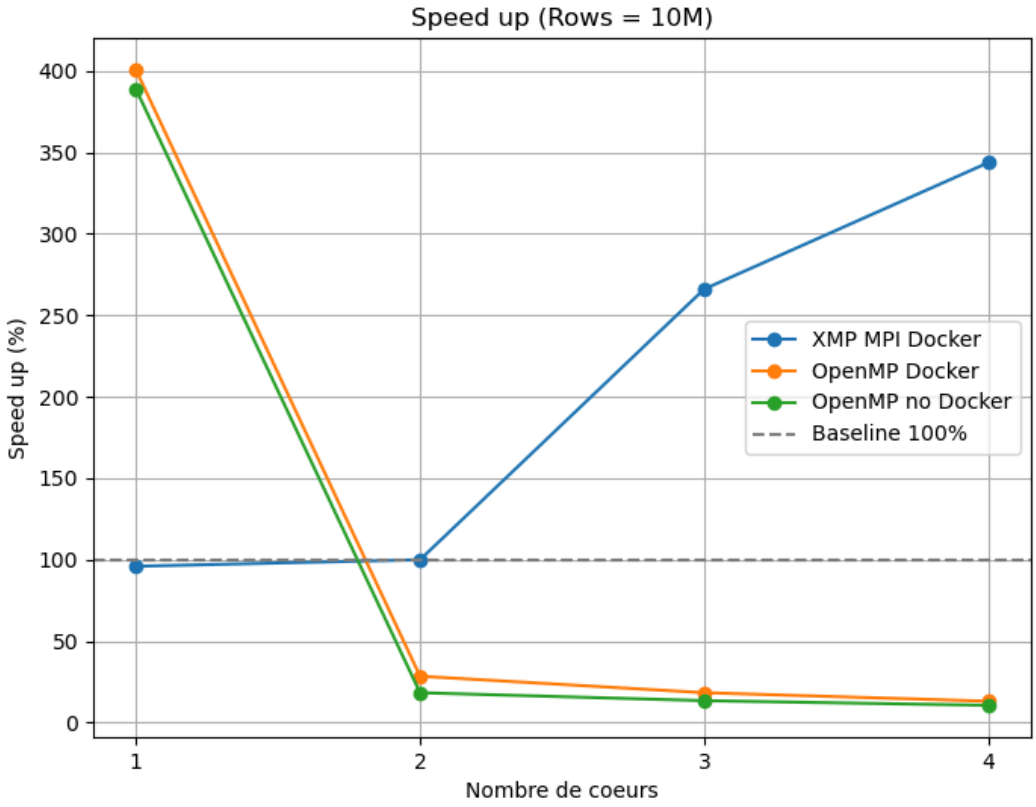
python3 generation_image.py output.txt.0 10000000

Avec output.txt.0 le nom du fichier de sortie et 10000000 le nombre total d'occurrences.

3.2 Accélération multicoeurs

En prenant comme référence l'utilisation de 2 cœurs avec 10 millions de données, nous obtenons le tableau ainsi l'image suivant :

Rows	Expérience	N	Temps moyen (sec)	Ecart type (%)	Speed up
10M	XMP MPI Docker	1	2.656e-02	1.23	96.00%
10M		2	2.551e-02	0.95	100.00%
10M		3	9.589e-03	2.74	266.09%
10M		4	7.421e-03	2.33	343.73%
10M	OpenMP Docker	1	6.365e-03	2.25	400.70%
10M		2	8.936e-02	1.44	28.54%
10M		3	1.385e-01	0.66	18.42%
10M		4	1.932e-01	0.80	13.19%
10M	OpenMP no Docker	1	6.558e-03	3.40	388.90%
10M		2	1.386e-01	3.16	18.40%
10M		3	1.885e-01	0.72	13.53%
10M		4	2.392e-01	4.84	10.66%



Nous pouvons constater que la version OpenMP offre des performances nettement supérieures à celle de la version XMP MPI en conteneur Docker. Toutefois, l'efficacité diminue de manière significative avec l'augmentation du nombre de threads, probablement en raison d'une zone sujette aux conditions de course dans notre code.

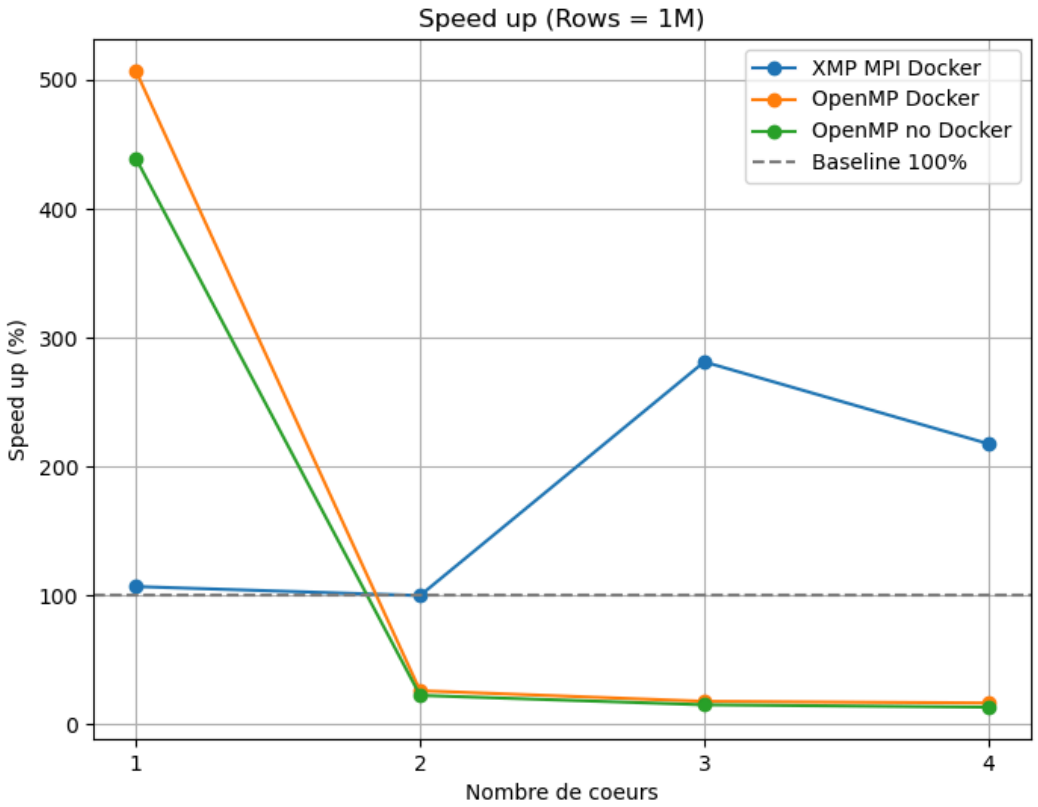
Par ailleurs, l'utilisation d'un nombre plus élevé de nœuds XMP MPI améliore les performances, bien que les résultats obtenus avec un ou deux nœuds restent relativement similaires. Ce constat s'explique par le fait que le temps de communication entre deux nœuds pénalise fortement la performance de base.

De plus, l'analyse des courbes comparant la version OpenMP sous Docker et hors Docker révèle peu de différences, même si la version Docker se montre légèrement plus performante, ce qui est quelque peu surprenant.

Enfin, nos tests réalisés avec différentes tailles de données (1 million et 100 millions) ont abouti à des résultats comparables.

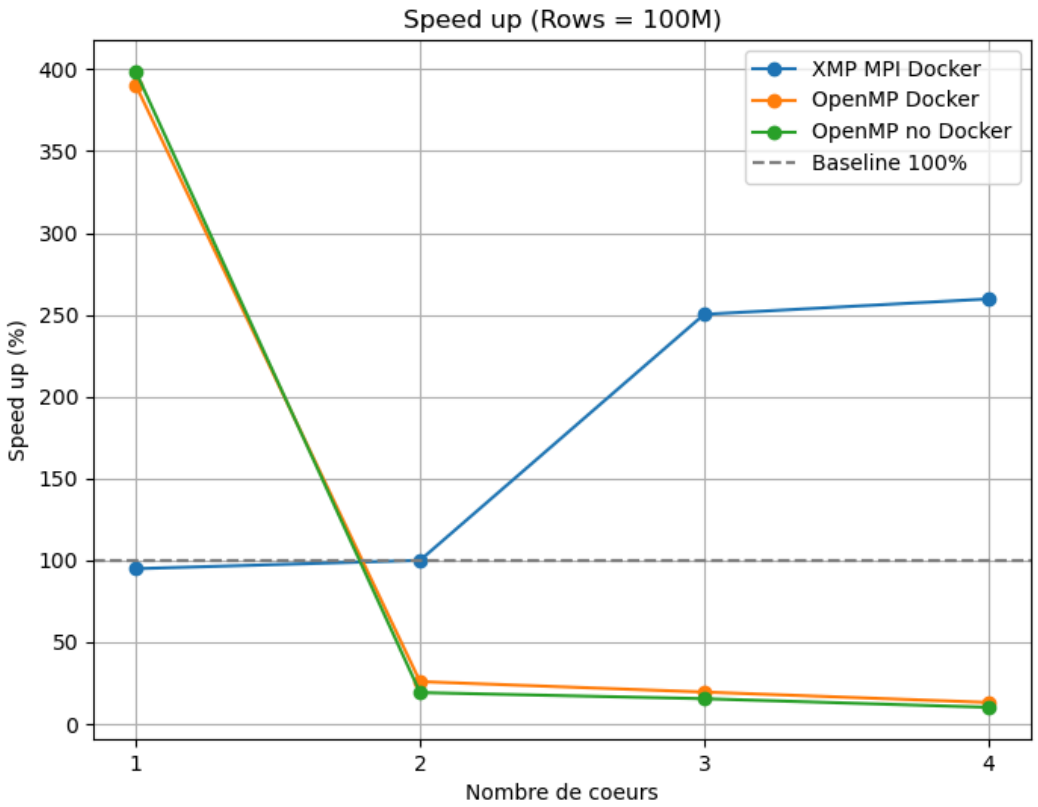
3.3 Expérience avec les tailles de données différentes

3.3.1 1 Millions de données



Rows	Expérience	N	Temps moyen (sec)	Ecart type (%)	Speed up
1M	XMP MPI Docker	1	2.532e-03	4.36	107.00%
1M		2	2.709e-03	4.59	100.00%
1M		3	9.630e-04	4.19	281.30%
1M		4	1.245e-03	0.99	217.77%
1M	OpenMP Docker	1	5.340e-04	4.81	507.04%
1M		2	1.034e-02	4.71	26.20%
1M		3	1.509e-02	4.37	17.95%
1M		4	1.623e-02	4.59	16.69%
1M	OpenMP no Docker	1	6.170e-04	4.55	439.22%
1M		2	1.201e-02	4.94	22.57%
1M		3	1.776e-02	2.88	15.26%
1M		4	2.021e-02	3.36	13.40%

3.3.2 100 Millions de données



Rows	Expérience	N	Temps moyen (sec)	Ecart type (%)	Speed up
100M	XMP MPI Docker	1	2.682e-01	2.10	94.89%
100M		2	2.545e-01	0.63	100.00%
100M		3	1.017e-01	0.85	250.30%
100M		4	9.797e-02	1.13	259.70%
100M	OpenMP Docker	1	6.519e-02	2.48	390.20%
100M		2	9.818e-01	1.41	25.91%
100M		3	1.304e+00	0.79	19.52%
100M		4	1.932e+00	1.09	13.17%
100M	OpenMP no Docker	1	6.386e-02	4.57	398.51%
100M		2	1.325e+00	4.03	19.21%
100M		3	1.647e+00	4.57	15.45%
100M		4	2.500e+00	3.03	10.18%

Nous remarquons une petite différence de performance entre un fichier de 1 million et un fichier de 10 millions de données, surtout avec XMP MPI sur 4 nœuds. Ce dernier pourrait être dû à la concurrence d'accès aux données, mais il faudrait effectuer un test pour en être certain. Toutefois, la tendance se retrouve clairement sur les trois graphes qui affichent des courbes quasiment identiques.

4 Conclusion

Aujourd'hui, il existe plein de techniques pour faire du calcul en multinoeuds. Pour notre exemple, on a utilisé XcalableMP, et on a montré que ce langage à directives n'était peut-être pas le plus performant comparé à une exécution en monothread avec 4 nœuds de calcul. Cela dit, notre expérience reste assez basique, puisqu'on a juste mesuré la performance du calcul. Dans notre cas, c'est surtout la lecture du fichier qui prend du temps, et du coup, la performance globale est limitée par cette lecture unique. En plus, notre programme OpenMP est écrit en un seul endroit dans le cas du multithreading, ce qui fait chuter la performance à cause d'une race condition.

Par ailleurs, la performance du calcul avec XMP est difficile à mesurer ici, car on utilise un seul disque pour la simulation dans Docker. Dans la vraie vie, on serait sûrement face à des systèmes de fichiers répartis sur plusieurs disques et sur plusieurs nœuds de calcul, et la performance dépendrait beaucoup de la manière dont les données sont stockées (par exemple, avec du RAID).