UVSQ université PARIS-SACLAY | ISTY Institut des Sciences et Techniques des Yvelines CAMPUS DE MANTES EN YVELINES CAMPUS DE SAINT-QUENTIN-EN-YVELINES | école normale supérieure paris-saclay | université PARIS-SACLAY

M2 Calcul Haute Performance, Simulation

Rapport de projet d'AOC:

Algebraic Multigrid solver (AMG)

Réalisé par :

Naim BESSAHA
Lotfi Rafik BOUCHAFA

Encadré par :

Mr.Emmanuel Oseret

Année universitaire : 2022 - 2023

Table des matières

Table des figures

Introduction générale

Le présent document est un rapport sur l'analyse et l'optimisation de code d'une application dans le cadre du module AOC du Master 2 Calcul Haute Performance. L'application étudiée est une librairie de l'algèbre linéaire appelée AMG, cette dernière permet de résoudre des systèmes linéaires tridimensionnels d'une manière générale, et plus précisément des systèmes linéaires issus de problèmes sur des grilles non structurées.

La librairie AMG (Algebric MultiGrid), comme son nom l'indique utilise la méthode itérative multigrille qui implémente un algorithme très optimal permettant de résoudre un système linéaire à N inconnues avec seulement O(N), ce qui en fait un solveur idéal pour la résolution des systèmes d'équations complexes et de très grandes tailles.

Le code de la librairie AMG est écrit en langage ISO-C. Le modèle de programmation utilisé pour permettre à l'application de s'exécuter sur des architectures parallèles est le modèle SPMD (Single Pro- gram Multiple Data). Ce parallélisme de données est réalisé par la décomposition de la matrice en blocs logiques de dimension P*Q*R (en 3D) de taille égale, ainsi que par l'utilisation des directives d'OpenMP et l'utilisation de MPI pour l'expression du parallélisme sur des architectures distribuées.

Dans un premier temps, on va présenter l'architecture de la machine test et l'environnement de travail, puis on va faire le bilan de performance de l'application dans les deux modes séquentielle et parallèle, ceci sera établi en exécutant le programme plusieurs fois sur un dataset préalablement choisi en prenant les options de compilation par défaut et en variant le nombre de threads a chaque exécution. Ensuite, en se servant du logiciel Maqao, on va optimiser la compilation, ceci en changeant les options de cette dernière. Par la suite on va analyser les performances et proposer des optimisations qui nous permettent d'aboutir à des meilleures performances. Enfin, on va conclure par les différents savoirs qu'on a pu acquérir à travers ce projet.

Chapitre 1

Environnement D'éxecution

L'ensemble de mesures et tests ont été effectuées dans l'environnement de travail suivant :

Architecture	x86_64
Processeur	Intel(R) Genuine Intel(R) CPU 0000 @1.30GHz
Cache Size	L1d=2 MiB, L1i=2 MiB, L2=32 MiB
Coeurs	64
Threads par coeur	4
Micro Architecture	KNIGHTS_LANDING

Le programme à été compilé suivant les caractéristiques suivantes :

Systeme d'exploitation	Ubuntu 20.04.5 LTS
Compilateur	GCC 9.4.0
MPI	MPICC 9.4.9
Autres outils	MAQAO 12.5.0

Chapitre 2

Cas séquentielle

2.1 Compilation et éxécution en séquentielle

Dans ce cas nous avons exécuté le programme avec un seul thread openMp, un seul processus MPI et sans ajouter d'optimisation pour le code d'origine, et cela pour avoir l'exécution séquentielle parfaite du programme.

Pour exécuter le code avec un seul processus MPI on utilise la commande suivante :

mpirun -n 1

Pour ce qui est des options de compilation initiales :

-g -O2 -DHYPRE_SEQUENTIAL=1

Le flague **-DHYPRE_SEQUENTIAL=1** permet d'activer l'exécution séquentielle du programme et de lui spécifier qu'il doit s'exécuter avec un seul thread OpenMP et un seul processus MPI.
Ce programme nous donne aussi la possibilité de configurer la taille du problème à résoudre dans la ligne de commande lors de l'exécution. Ainsi, pour obtenir un temps d'exécution compris entre 1 minute et 1 minute 30 secondes, nous avons choisi la taille de 150 x 150 x 100, et nous avons obtenu un temps de 88,35 secondes.

Voici la commande d'éxécution séquentielle final du programme :

mpirun -n 2 ./amg -n 150 150 100 -P 1 1 1

Afin d'optimiser le programme, nous nous sommes référés à l'outil d'analyse et d'optimisation de performances MAQAO qui permet de générer des rapports synthétiques sur le binaire exécuté et fournit des conseils et des indications pour améliorer les performances du programme. Afin de générer le rapport de MAQAO nous utilisons la commande suivante :

maqao oneview -R1 –executable=./amg –run_command="executable -n 150 200 200 -P 1 1 1" –mpi_command="mpirun -n 1"

Le rapport produit par Maqao est enregistré dans un dossier nommé "oneview_baseline".

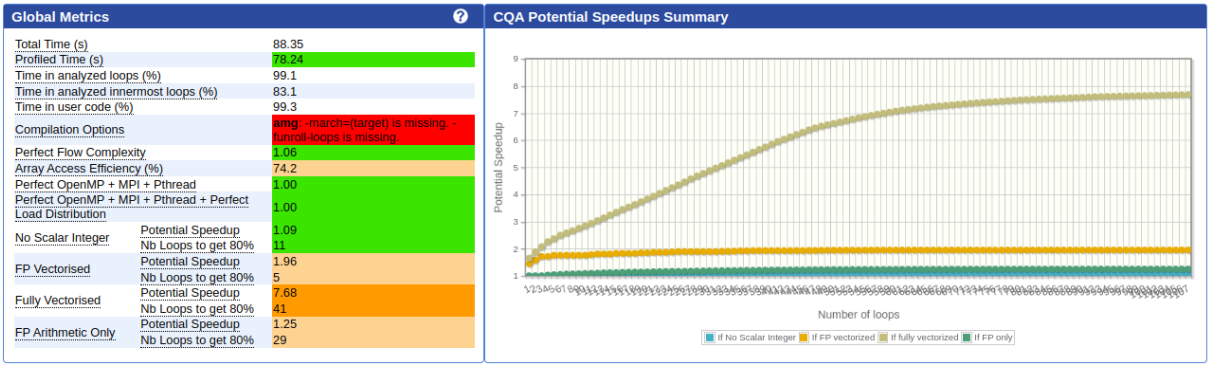
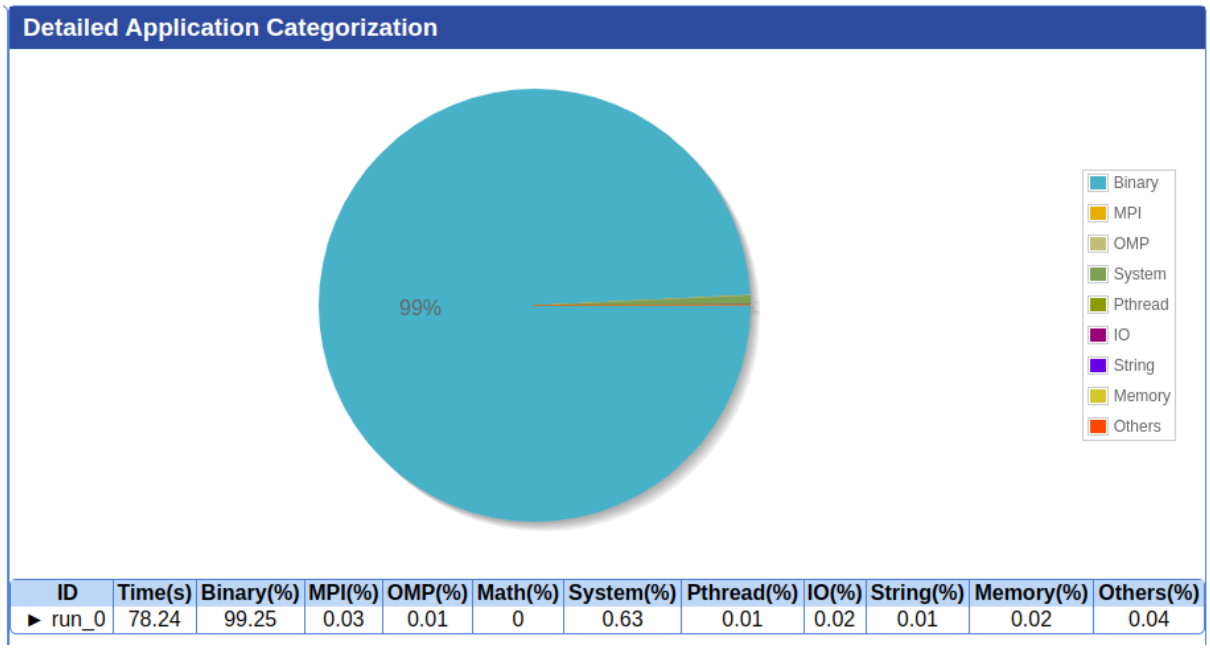


FIGURE 2.1 – Métriques globales, accélération potentielle

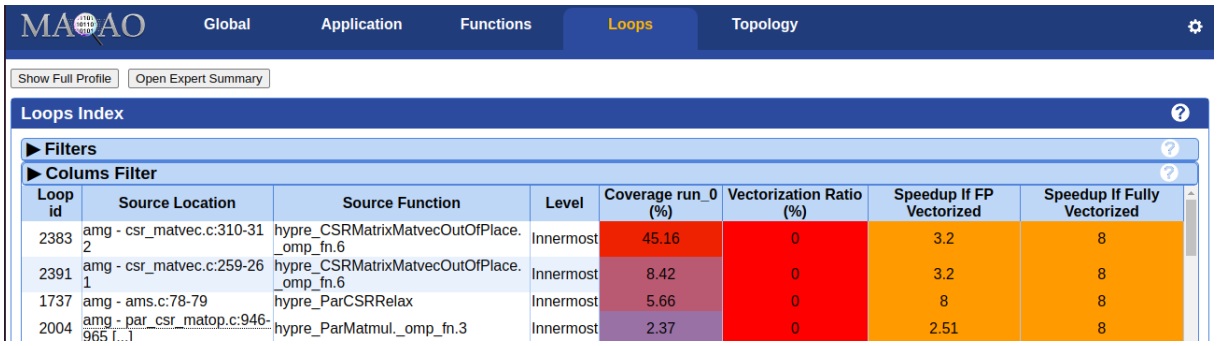
Premièrement, Maqao indique que les options de compilation par défaut ne sont pas très efficaces (score de 50/100), et nous conseille d'ajouter d'autres options.
D'après les estimations de maqao, le programme à un potentiel d'accélération (speedup) de 4.38 dans le cas ou on vectorise toutes les instructions en virgule flottante dans toutes les boucles et un potentiel de 4.38 si on applique une vectorisation de toutes les instructions, aussi bien les instructions de calcul en virgule flottante que les instructions d'accès à la mémoire dans toutes les boucles.



ID	Time(s)	Binary(%)	MPI(%)	OMP(%)	Math(%)	System(%)	Pthread(%)	IO(%)	String(%)	Memory(%)	Others(%)
► run_0	78.24	99.25	0.03	0.01	0	0.63	0.01	0.02	0.01	0.02	0.04

FIGURE 2.2 – Catégorisation de l'application

D'après la figure 2.2, on remarque que le programme AMG passe la majeur partie du temps a exécuter des instructions sur le CPU (faire "des calculs") et non pas a faire des accès mémoire ou disque (I/O) ou autres choses tel que les appels systèmes. ceci nous indique que la vitesse d'exécution des instructions sur le CPU aura donc un impact important sur les performances.



Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Vectorization Ratio (%)	Speedup If FP Vectorized	Speedup If Fully Vectorized
2383	amg - csr_matvec.c:310-312	hypre_CSRMatrixMatvecOutOfPlace._omp_fn.6	Innermost	45.16	0	3.2	8
2391	amg - csr_matvec.c:259-261	hypre_CSRMatrixMatvecOutOfPlace._omp_fn.6	Innermost	8.42	0	3.2	8
1737	amg - ams.c:78-79	hypre_ParCSRRelax	Innermost	5.66	0	8	8
2004	amg - par_csr_matop.c:946-965 [...]	hypre_ParMatmul._omp_fn.3	Innermost	2.37	0	2.51	8

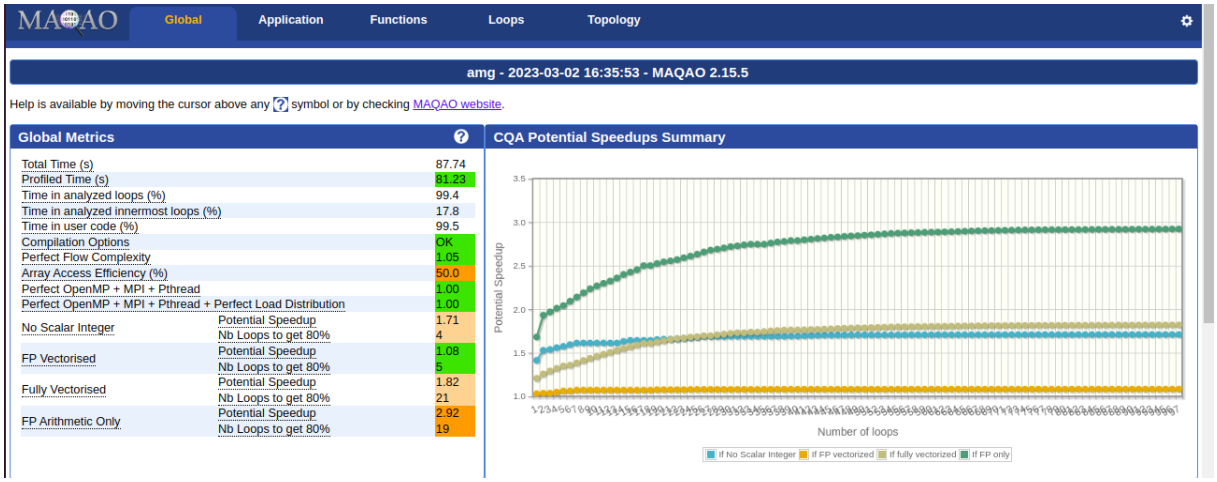
FIGURE 2.3 – Les boucles chaudes

On constate que la boucle présente dans la fonction "hypre_CSRMatrixMatvecOutOfPlace" occupe 45.16% du temps d'exécution, donc l'optimisation de celle-ci sera très bénéfique pour l'obtention de meilleur performance.

2.2 Optimisation

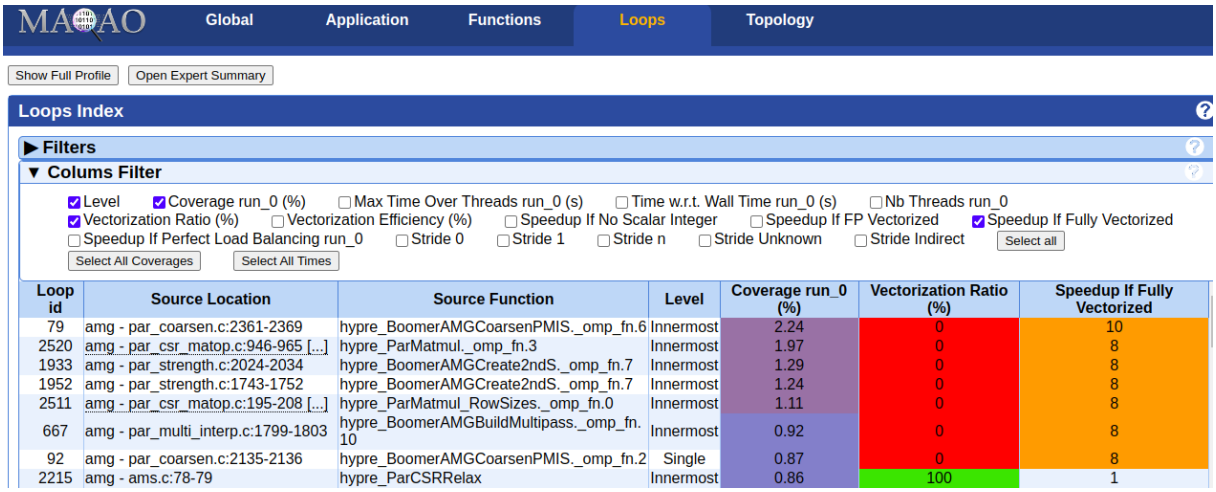
Nous avons changé l'option de compilation de -O2 a -Ofast afin de pousser le compilateur à mieux exploiter les optimisations tel que la vectorisation des boucles quand ceci est possible et l'optimisation des opérations mathématiques. Ensuite, on ajoute l'option -march=native tel qu'indiqué par maqao pour activer les derniers jeux d'instructions SIMD pris en charge par le processeur (AVX512 au lien de SSE2) et l'option -funroll-loops pour appliquer le déroulage automatique des boucles.

2.3 Mesure de performances après l'optimisation



Total Time (s)		87.74
Profiled Time (s)		81.23
Time in analyzed loops (%)		99.4
Time in analyzed innermost loops (%)		17.8
Time in user code (%)		99.5
Compilation Options		OK
Perfect Flow Complexity		1.05
Array Access Efficiency (%)		50.0
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.71
	Nb Loops to get 80%	4
FP Vectorised	Potential Speedup	1.08
	Nb Loops to get 80%	5
Fully Vectorised	Potential Speedup	1.82
	Nb Loops to get 80%	21
FP Arithmetic Only	Potential Speedup	2.92
	Nb Loops to get 80%	19

FIGURE 2.4 – Métriques globales, accélération potentielle



Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Vectorization Ratio (%)	Speedup If Fully Vectorized
79	amg - par_coarsen.c:2361-2369	hypre_BoomerAMGCoarsenPMIS._omp_fn.6	Innermost	2.24	0	10
2520	amg - par_csr_matop.c:946-965 [...]	hypre_ParMatmul._omp_fn.3	Innermost	1.97	0	8
1933	amg - par_strength.c:2024-2034	hypre_BoomerAMGCreate2ndS._omp_fn.7	Innermost	1.29	0	8
1952	amg - par_strength.c:1743-1752	hypre_BoomerAMGCreate2ndS._omp_fn.7	Innermost	1.24	0	8
2511	amg - par_csr_matop.c:195-208 [...]	hypre_ParMatmul_RowSizes._omp_fn.0	Innermost	1.11	0	8
667	amg - par_multi_interp.c:1799-1803	hypre_BoomerAMGBuildMultipass._omp_fn.10	Innermost	0.92	0	8
92	amg - par_coarsen.c:2135-2136	hypre_BoomerAMGCoarsenPMIS._omp_fn.2	Single	0.87	0	8
2215	amg - ams.c:78-79	hypre_ParCSRRelax	Innermost	0.86	100	1

FIGURE 2.5 – Boucles

Pour la partie optimisation du code, dans notre cas, il n'y a pas de tableaux de structures ou de structures de tableaux dans le code, il n'est donc pas possible de transformer les AoS en SoA ou bien les SoA en AoS.

Chapitre 3

Cas parallèle

Au cours de cette partie, nous executons le programme AMG en utilisant plusieurs threads pour répartir et subdiviser les données, et nous exploiterons MPI et OpenMP pour exécuter le programme sur plusieurs processeurs, permettant ainsi de résoudre des problèmes plus volumineux.

3.1 Étude de scalabilité

Afin d'évaluer et de visualiser l'extensibilité de la parallélisation, on réalise des mesures de performances en variant le nombre de threads de 1 à 256 threads par puissance de 2. On a utilisé la fonctionnalité "scalabilité" de l'outil maqao qui permet d'exécuter plusieurs fois un programme en variant les paramètres d'exécution de ce dernier tel que le nombre de threads dans notre cas.
Pour cela nous avons préparé trois fichiers de configuration qui décrivent toutes les expériences à exécuter :
config_omp.lua : fichier contenant les configurations d'exécution du programme avec un seul processus et en variant le nombre de threads de 1 à 256 en puissance de 2 (dans ce contexte seul OpenMP est utilisé).
config_mpi.lua : fichier contenant les configurations d'exécution du programme avec un seul thread et en variant le nombre de processus de 1 à 8 en puissance de 2 (dans ce contexte seul MPI est utilisé).
config_hyb.lua : fichier contenant les configurations d'exécution du programme en combinant plusieurs threads et processus en même temps (exécution hybride d'OpenMP et MPI) .

1. **Variation du nombre de threads** :

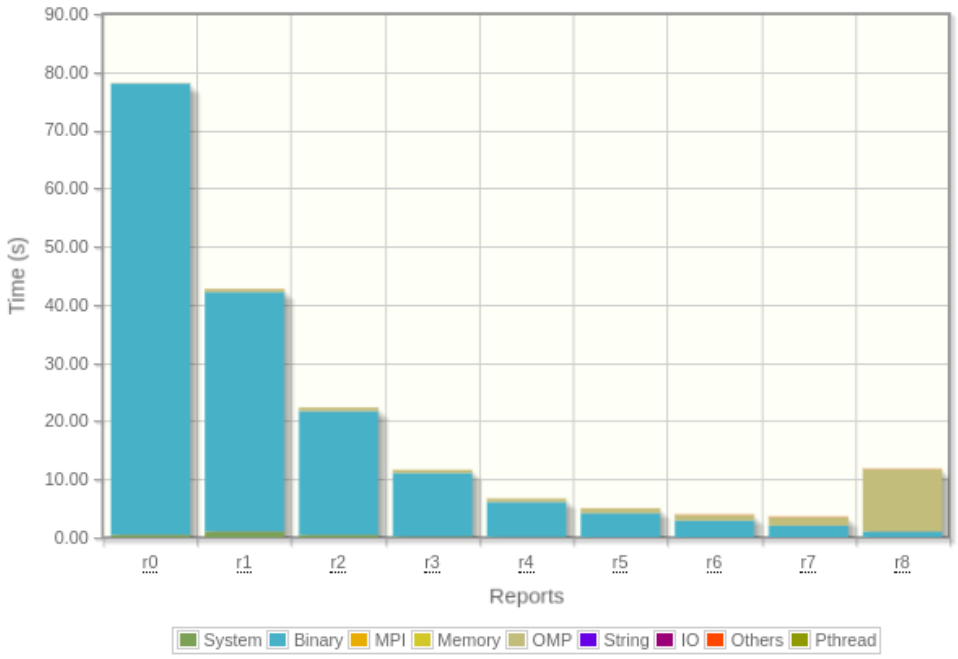
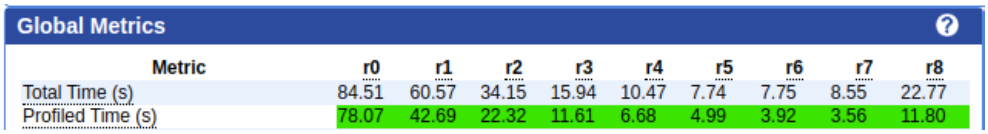


FIGURE 3.1 – Temps d'exécution en fonction du nombre de threads

Pour calculer l'efficacité de la scalabilité, on va comparer les différents temps d'exécution avec le temps d'exécution de la version purement séquentielle (1 seul thread, 1 seul processus) qui est égale à 84.51 secondes.



Global Metrics

Metric	r0	r1	r2	r3	r4	r5	r6	r7	r8
Total Time (s)	84.51	60.57	34.15	15.94	10.47	7.74	7.75	8.55	22.77
Profiled Time (s)	78.07	42.69	22.32	11.61	6.68	4.99	3.92	3.56	11.80

FIGURE 3.2 – Temps d'exécution en fonction du nombre de threads

Nombre de threads	Temps d'exécution	Accélération obtenue	Efficacité
2	60.57	1.39	0.69
4	34.15	2.47	0.61
8	15.94	5.30	0.66
16	10.47	8.07	0.50
32	7.74	10.92	0.34
64	7.75	10.90	0.17
128	8.55	9.88	0.08
256	22.77	3.71	0.01

D'après la figure 3.1, on voit que le temps d'exécution diminue en augmentant le nombre de threads ce qui est logique puisque c'est le même travail (résolution du système linéaire dans notre cas) qui est partagé entre plusieurs threads s'exécutant en parallèle. Cependant on remarque que les performances maximales

sont atteintes lorsqu'on utilise 32 threads OpenMP. Par la suite, la gestion de la parallélisation devient plus coûteuse qu'elle ne rapporte a cause des coûts de communications et de synchronisation entre les différents threads.

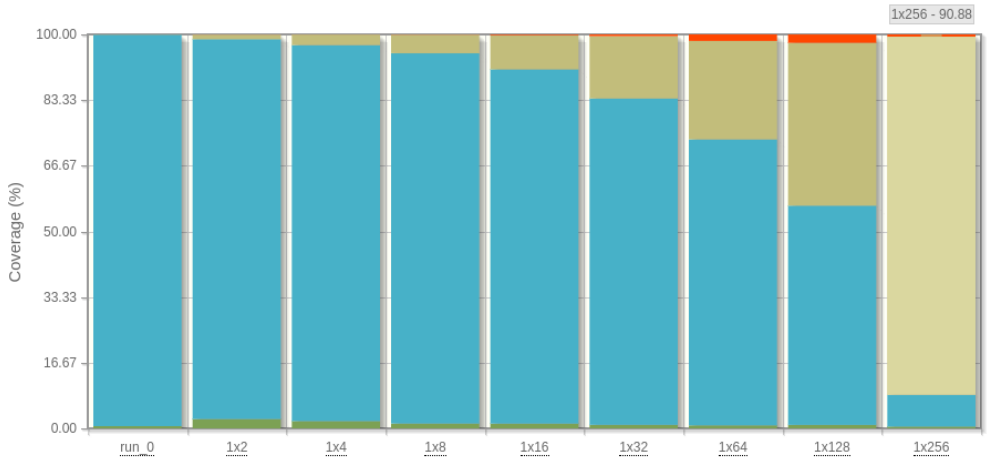


FIGURE 3.3 – Catégorisation de l'application

La figure 3.3 montre clairement qu'en augmentant le nombre de thread le temps passé par le programme dans la bibliothèque OpenMP pour la création, synchronisation ..etc de threads dépasse le temps passé à exécuter réellement les instructions du programme.

2. **Variation du nombre de processus** :

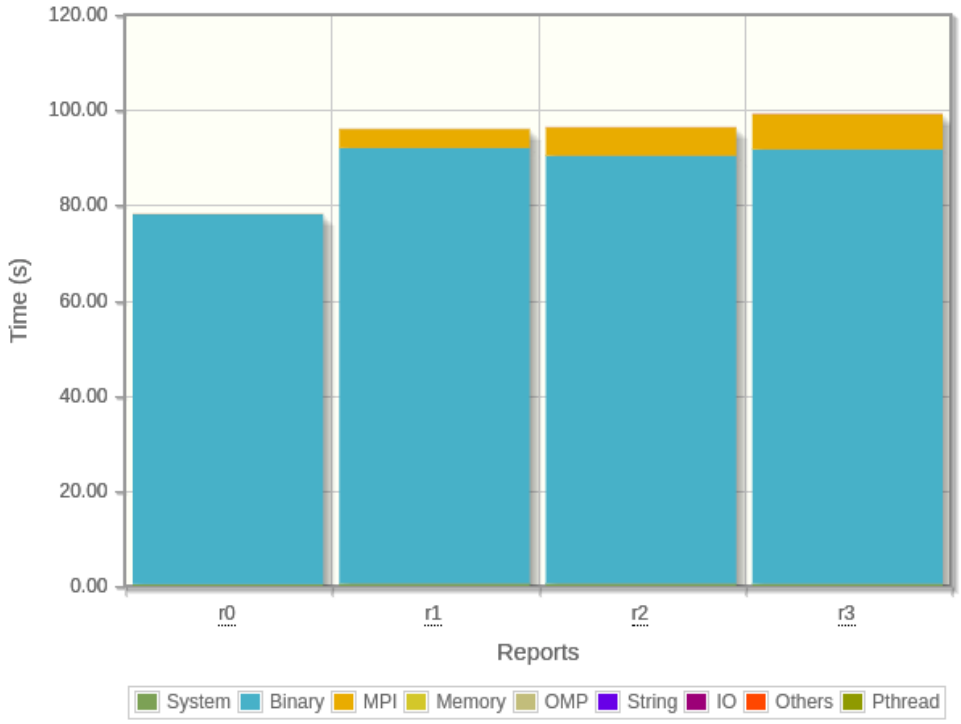


FIGURE 3.4 – Temps d'exécution en fonction du nombre de processus

Contrairement au cas précédent où nous avons varié le nombre de threads en utilisant openMp pour diviser le travail et le partager entre plusieurs threads, dans ce cas là, le travail (résolution du système linéaire de taille N) est dupliqué sur l'ensemble des processus dans chaque exécution ceci est connu comme la scalabilité faible d'un programme.
D'après la figure 3.4, on constate qu'en augmentant le nombre de processus, le temps d'exécution ne change pas trop par rapport au temps d'exécution séquentiel ou on utilise qu'un seul processus.
Par la suite, on va calculer l'efficacité parallèle pour les différentes exécutions effectuées. Dans le cas de faible scalabilité, on dit qu'un programme est totalement efficace (100%) si le temps d'exécution parallèle est identique à celui en séquentiel.

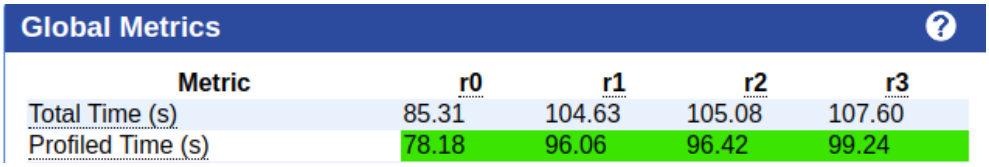


FIGURE 3.5 – Temps d'exécution en fonction du nombre de processus

Nombre de processus	Temps d'exécution	Efficacité
2	104.63	0.81
4	105.08	0.81
8	107.60	0.79

3. **Variation du nombre de processus et threads** :
Afin de visualiser l'effet de variation du nombre de threads et du nombre de processus sur le temps d'exécution et sur les performances du programme d'une manière générale, nous avons étudié 8 combinaisons de threads et de processus à l'aide du mode "scalabilité" du logiciel MAQAO. La figure suivante illustre le temps d'exécution obtenu pour chaque combinaison.

Label expérience	r0	r1	r2	r3	r4	r5	r6	r7	r8
Combinaison	1x1	2x2	2x4	2x8	2x16	4x2	4x4	4x8	4x16

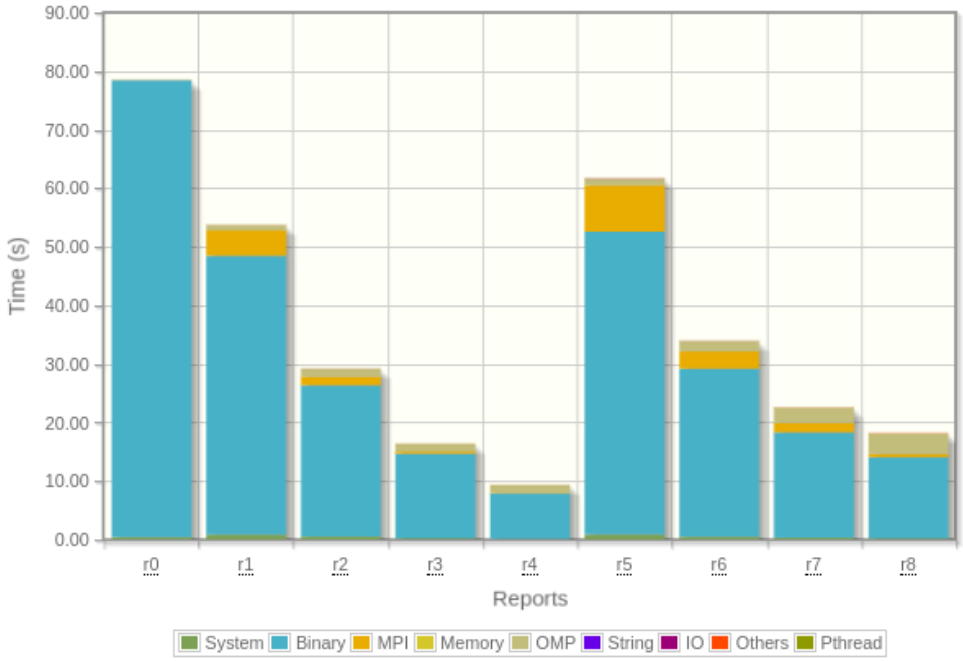


FIGURE 3.6 – Temps d'exécution par combinaison

On constate que la combinaison avec laquelle nous avons obtenu les meilleures performances est la combinaison r4 dans laquelle nous avons utilisé 2 processus avec 16 threads chacun.

Chapitre 4

CONCLUSION

Dans ce projet, nous avons travaillé sur un code qui permet de résoudre des problèmes d'algèbre linéaire de plusieurs dimensions.

Nous avons essayé d'atteindre la meilleure performance lors de l'exécution en exploitant plusieurs threads de la machine sur laquelle le code a été exécuté en utilisant OpenMP et MPI.

Nous avons aussi exploité toutes les optimisations offertes par le compilateur afin d'atteindre la vectorisation maximale des boucles chaudes du code.

Bien que les résultats n'aient pas été à la hauteur de nos attentes, ce projet nous a tout de même permis de mettre en pratique les connaissances théoriques acquises lors du module AOC.

En somme, ce projet a été une opportunité précieuse pour appliquer et renforcer nos compétences en programmation et en optimisation de code.