Université Saint Quentin en Yvelines
Université Paris Saclay

Rapport PPN

M1 Calcul Haute Haute Performance, Simulation

Utilisation et analyse critique des rapports MAQAO pour optimiser des mini-apps

Étudiants:
Anis Mehidi
Arezki Takfarines Hamidani
Katia Mouali
Madjid bouzourene
Sylia Benbachir

Responsables:
Cédric Valensi
Emmanuel Oseret

MAI 2022

Table des matières

Table des figures

1 Introduction

Dans le monde de la programmation informatique, l'optimisation est le processus qui consiste à réduire le temps d'exécution d'une fonction, l'espace occupé par les données et le programme, ou la consommation d'énergie.

En règle générale, l'optimisation doit s'effectuer une fois que le programme est fonctionnel et qu'il réponde aux spécificités attendues avant de commencer l' optimisation, pour cela, il existe plusieurs approches d'optimisation l'une plus complexe que l'autre, on peut citer quelques unes :

— Au niveau algorithmique, en choisissant un algorithme de complexité inférieure (au sens mathématique) et des structures de données adaptées.

— Au niveau du langage de développement, en ordonnant au mieux les instructions et en utilisant les bibliothèques disponibles.

— En utilisant localement un langage de bas niveau, qui peut être le langage C ou, pour les besoins les plus critiques, le langage assembleur.

La parallélisation est l'une des principales approches de l'optimisation de programme et la plus efficace, si correctement mise en place. Cependant il est très délicat de paralléliser un programme sans pour autant rendre les résultats attendus de ce dernier obsolètes, c'est d'ailleurs l'un des plus grands défis auxquels font face les ingénieurs développement calcul scientifique.

Nous allons donc, dans cette deuxième partie de notre projet utiliser l'outil MAQAO sur des codes qui s'exécutent en parallèle en utilisant plusieurs API (runtime) incontournables du parallélisme (Threading, OpenMP, MPI) et analyser la façon dont cet outil aide à l'optimisation dans ce cas afin d'en faire une synthèse.

2 Parallélisation et performance

Le parallélisme est la mise en œuvre d'architecture permettant de traiter des informations de manière simultanée, ceci afin de réaliser le plus d'opérations en un minimum de temps possible. Ce paradigme répond à la difficulté d'augmenter la vitesse de traitement des unités de calcul. En effet, l'augmentation de la fréquence connaît des limites car entraînant une plus forte consommation d'énergie, une grande dissipation de chaleur et des contraintes de gravure des microprocesseurs. Le gain de performance passe donc par la mise en œuvre du parallélisme et de ses technologies sous-jacentes comme les FMA, la vectorisation ou l'exécution de tâches par plusieurs processus ou threads.

2.1 Modèles de programmation parallèle

1. **PTHREAD (POSIX Thread libraries) :**
   Sont une API de thread basée sur des normes pour C/C++. PTHREAD permet de générer un nouveau flux de processus simultané. Il est plus efficace sur les systèmes multiprocesseurs ou multicœurs où le flux de processus peut être programmé pour s'exécuter sur un autre processeur, gagnant ainsi en vitesse grâce à un traitement parallèle ou distribué.
2. **OpenMP :**
   OpenMP (Open Multi-Processing) est une interface de programmation d'applications (API) pour le calcul parallèle sur architecture à mémoire partagée. en C/C++ et Fortran sur de nombreuses architectures, y compris les plateformes Unix et Microsoft Windows. Elle consiste en un ensemble de directives de compilateur, de routines de bibliothèque et de variables d'environnement qui influencent le comportement d'exécution.
3. **MPI (Message Passing Interface) :**
   Est une norme de passage de messages entre ordinateurs distants ou dans un ordinateur multiprocesseur. Autrement dit, la norme MPI est un protocole de communication entre différents processus. Elle définit une bibliothèque de fonctions, utilisable avec les langages C, C++ et Fortran. MPI a pour but d'obtenir de bonnes performances aussi bien sur des machines massivement parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée.

3 Outils d'analyse MAQAO

3.1 Définition

MAQAO (Modular Assembly Quality Analyzer and Optimizer) est un framework qui permet d'analyser et d'optimiser les performances d'un programme grâce à un ensemble de modules (CQA, LPROF, ONEVIEW). MAQAO effectue une analyse dynamique et statique du code et ainsi détermine les éléments limitant la performance d'une zone de l'application, il travaille au niveau binaire il n'a pas de restriction par rapport au langage utilisé dans le code source. l'objectif principal de MAQAO est de guider les développeurs d'applications tout au long du processus d'optimisation grâce à des rapports synthétiques et des astuces [w1].

3.2 Fonctionnement

Le module LPROF lance le programme à analyser et puis vérifie son état à l'aide d'un thread watcher. Cette vérification consiste à prendre des échantillons, par défaut 200 par seconde, puis les analyser pour déterminer quelle fonction et quelle boucle s'exécutent et calculer leur couverture par rapport au temps d'exécution total. Le module CQA analyse le code généré statiquement et évalue son efficacité sur la base de l'architecture de la machine cible et des registres et instructions utilisées par le programme. Ces deux modules sont lancés par ONEVIEW afin que ce dernier puisse mettre en place les résultats d'une analyse dans un format comme l'HTML.

3.3 Utilisation de l'outil MAQAO

On va donc lancer MAQAO sur le binaire de nos codes et celui-ci va procéder à l'analyse de ce dernier et nous fournir un rapport détaillé sous format HTML contenant entre autres des astuces "Hints" à effectuer sur notre code source afin d'améliorer les performances de celui-ci. Au terme de chaque analyse, les speedups des runs précédents sont soit confirmés, soit infirmés, et de nouveau goulots d'étranglement peuvent être identifiés, entraînant une nouvelle modification du code.

4 Environnement de travail

L'environnement de travail dans lequel nous allons principalement effectuer nos analyses et optimisations d'applications avec maqao se feront sur des clusters mis à notre disposition dans le cadre du projet de programmation numérique.

4.1 Machine FOB1 du laboratoire LI-PaRaD

Le Cluster, encore appelé couramment Grappe de calcul, est une approche d'architecture distribuée dans laquelle un ensemble d'ordinateurs étroitement connectés travaillent ensemble et sont vus comme un seul ordinateur par l'utilisateur.

```
1  Architecture:                    x86_64
2  CPU(s):                          256
3  On-line CPU(s) list:             0-255
4  Thread(s) per core:              4
5  Core(s) per socket:              64
6  Socket(s):                       1
7  NUMA node(s):                    4
8  Vendor ID:                       GenuineIntel
9  CPU family:                      6
10 Model:                           87
11 Model name:          Intel(R) Genuine Intel(R) CPU 0000 @ 1.30GHz
12 CPU max MHz:                     1500.0000
13 CPU min MHz:                     1000.0000
```

4.2 Les implémentations utilisées

4.2.1 Cas MPI

MPICC développé par Argonne National Laboratory, MPICH est une implémentation haute performance, open source, gratuite et multi-plateforme (linux-MacOS- Solaris-windows) portable de MPI, dont l'objectif est de fournir une implémentation MPI qui peut prendre en charge les plateformes de calcul et de communication ; tel que les différents clusters (architecture multi-coeurs, systèmes distribués, systèmes à mémoire partagée.

4.2.2 Cas OpenMP

OpenMP regroupe des directives de compilation et des fonctions. Le compilateur **gcc** supporte OpenMP depuis la **version 4.2**, en ajoutant simplement une option sur la ligne de commande et en incluant le fichier d'en-têtes **omp.h**. La gestion des différentes fonctions est assurée par la libraire libgomp.

4.2.3 Cas PThread

En incluant le fichier d'en-tête **pthread.h**, on peut utiliser différentes opérations de thread incluent la création, la terminaison, la synchronisation (jointures, blocages), la planification, la gestion des données et l'interaction des processus.

Un thread ne gère pas une liste des threads créés et ne connaît pas non plus le thread qui l'a créé et tous les threads d'un processus partagent le même espace d'adressage.

5 Applications de travail

5.1 NPM : lu-mz

Les NAS Parallel Benchmarks (NPB) sont un petit ensemble de programmes conçus pour aider à évaluer les performances des supercalculateurs parallèles.

LU-MZ est une version multi-zones du NAS Parallel Benchmarks (NPB), conçue pour exploiter plusieurs niveaux de parallélisme dans les applications et pour tester l'efficacité des paradigmes et outils de parallélisation multi-niveaux et hybrides.

Cette version est de zones de taille égale dans une classe de problèmes, un nombre fixe de zones pour toutes les classes de problèmes.

5.2 Simulation Nbody3D

L' algorithme à N corps est assez simple, il consiste simplement à calculer toutes les interactions particule-particule pour chaque étape de la simulation en utilisant les lois de Newton de la mécanique classique. Étant donné un système à N objets, cela se traduit par une complexité arithmétique $O(n^2)$.

6 Travail effectué

Cette section représente le travail que nous avons effectuer tout au long de ce semestre. Dans un premier temps, dans le but de se familiariser avec l'outil MAQAO dans un contexte parallèle, nous avons effectuer le profilage de l'application **lu-mz** afin de pouvoir observer et d'analyser son exécution.
Nous avons, par la suite, procéder à la parallélisation de l'application **Nbody3D**, dans un environnement parallèle en implémentant les API citées précédemment, tout en s'appuyant sur les analyses faites avec l'outil MAQAO.

6.1 Cas lu-mz

6.1.1 Profilage d'une application avec MAQAO : Cas lu-mz

Pour bien comprendre le fonctionnement de MAQAO dans un environnement parallèle, nous avons analyser l'application **lu-mz**

La prochaine étape consiste à créer un fichier de configuration avec l'extension **.lua**, le fichier contient l'ensemble des indications nécessaires pour configurer notre expérience et édité ce dernier selon nos besoins.

Commande pour créer le fichier de configuration est la suivants :

```
maqao oneview --create-config=<nom de notre fichier>.lua
```

Les différents champs d'un fichier de configuration sont les arguments que l'on passe à **Oneview** que l'on peut facilement consulter dans la section help du manuel avec la commande suivante :

```
maqao oneview --help
```

L'ensemble des champs d'une expérience parallèle les plus important sont les suivants :

- experiment_name : Le nom du rapport générer
- run_command : Passage de l'exécutable <executable>
- number_processes : Nombre de processus MPI lancer.
- number_processes_per_node : Nombre de processus MPI par noeuds
- mpi_command : La commande a exécuter le programme MPI
- envv_OMP_NUM_THREADS : Spécifie le nombre de threads OpenMP pour chaque run
- multiruns_params : Spécifier les paramètres des différentes runs à effectuer.

Pour effectuer l'analyse d'une application donnée, il suffit de lancer MAQAO grâce aux commandes suivantes :

- Default :
  `maqao oneview -R1 -xp=dir_name -c=config.lua`
- Scalability :
  `maqao oneview -R1 -WS -xp=dir_name -c=config.lua`
  On ajoute l'option **-WS** pour être en mode Scalabilité .

Configuration du fichier .lua : Pour configurer notre fichier .lua il faudra tout d'abord spécifier le run de référence qui est généralement avec un process et un thread comme suit :
Le run de référence :

experiment_name = "Analyse de code lu-mz CLASS=A en mode scalabilité sur le cluster FOB1"
executable = "./lu-mz.A.x"
number_processes = 1
number_processes_nodes = 1
mpi_command = "mpirun -n <number_processes>"
envv_OMP_NUM_THREADS= 1

Par la suite on spécifie l'ensemble des runs avec des configuration différentes qui seront lancé à la fois dans un seul lancement de l'analyse **Maqao**.

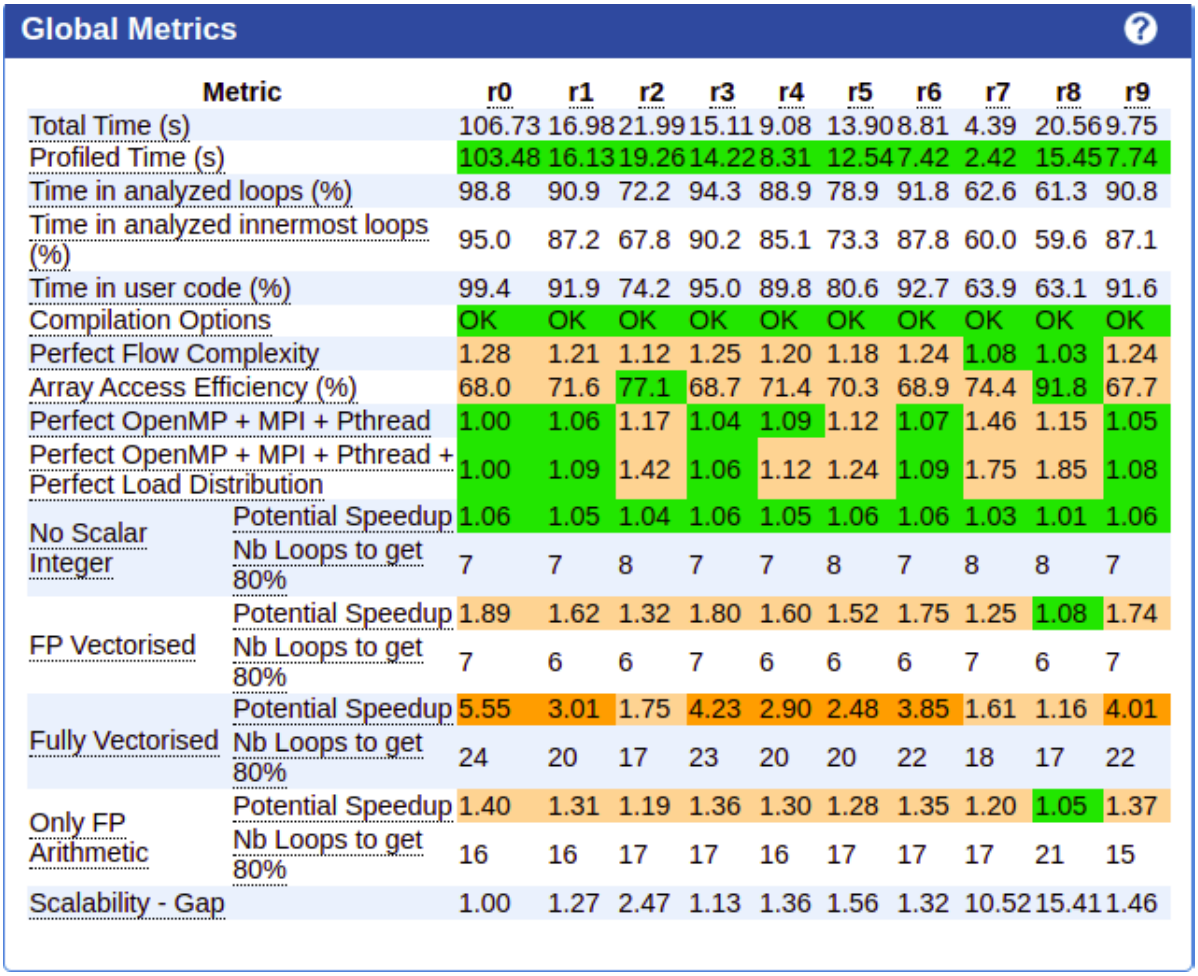
Ensemble des configuration a testé (multiruns_params) :

```
multiruns_params ={
    {name="R2x4",envv_OMP_NUM_THREADS="4", number_processes=2,
    number_processes_per_node = 2},
    {name="R3x4",envv_OMP_NUM_THREADS="4", number_processes=3,
    number_processes_per_node = 3},
    {name="R4x2",envv_OMP_NUM_THREADS="2", number_processes=4,
    number_processes_per_node = 4},
    {name="R4x4",envv_OMP_NUM_THREADS="4", number_processes=4,
    number_processes_per_node = 4},
    {name="R6x2",envv_OMP_NUM_THREADS="2", number_processes=6,
    number_processes_per_node = 6},
    {name="R8x2",envv_OMP_NUM_THREADS="2", number_processes=8,
    number_processes_per_node = 8},
    {name="R16x16",envv_OMP_NUM_THREADS="16", number_processes=16,
    number_processes_per_node = 16},
    {name="R10x8",envv_OMP_NUM_THREADS="8", number_processes=10,
    number_processes_per_node = 16},
    {name="R16x1",envv_OMP_NUM_THREADS="1", number_processes=16,
    number_processes_per_node = 16},
}
```

On a testé un ensemble de configuration pour découvrir le run qui contient les meilleurs paramètres pour une meilleure exécution avec une parallélisation OpenMp et/ou MPI.

6.1.2 Résultats de l'expérience

1. **Global Metrics**



Global Metrics

Metric		r0	r1	r2	r3	r4	r5	r6	r7	r8	r9
Total Time (s)		106.73	16.98	21.99	15.11	9.08	13.90	8.81	4.39	20.56	9.75
Profiled Time (s)		103.48	16.13	19.26	14.22	8.31	12.54	7.42	2.42	15.45	7.74
Time in analyzed loops (%)		98.8	90.9	72.2	94.3	88.9	78.9	91.8	62.6	61.3	90.8
Time in analyzed innermost loops (%)		95.0	87.2	67.8	90.2	85.1	73.3	87.8	60.0	59.6	87.1
Time in user code (%)		99.4	91.9	74.2	95.0	89.8	80.6	92.7	63.9	63.1	91.6
Compilation Options		OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
Perfect Flow Complexity		1.28	1.21	1.12	1.25	1.20	1.18	1.24	1.08	1.03	1.24
Array Access Efficiency (%)		68.0	71.6	77.1	68.7	71.4	70.3	68.9	74.4	91.8	67.7
Perfect OpenMP + MPI + Pthread		1.00	1.06	1.17	1.04	1.09	1.12	1.07	1.46	1.15	1.05
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.09	1.42	1.06	1.12	1.24	1.09	1.75	1.85	1.08
No Scalar Integer	Potential Speedup	1.06	1.05	1.04	1.06	1.05	1.06	1.06	1.03	1.01	1.06
	Nb Loops to get 80%	7	7	8	7	7	8	7	8	8	7
FP Vectorised	Potential Speedup	1.89	1.62	1.32	1.80	1.60	1.52	1.75	1.25	1.08	1.74
	Nb Loops to get 80%	7	6	6	7	6	6	6	7	6	7
Fully Vectorised	Potential Speedup	5.55	3.01	1.75	4.23	2.90	2.48	3.85	1.61	1.16	4.01
	Nb Loops to get 80%	24	20	17	23	20	20	22	18	17	22
Only FP Arithmetic	Potential Speedup	1.40	1.31	1.19	1.36	1.30	1.28	1.35	1.20	1.05	1.37
	Nb Loops to get 80%	16	16	17	17	16	17	17	17	21	15
Scalability - Gap		1.00	1.27	2.47	1.13	1.36	1.56	1.32	10.52	15.41	1.46

FIGURE 1 – Global Metrics

Le run R7 est le meilleur run en temps d'exécution (16 process/16 Threads) par contre l'ordonnancement des entités parallèles n'est pas optimal. On remarque que les meilleures exécutions avec un meilleur ordonnancement sont les runs qui possède quatre ou huit entités parallèles **(mpi/openmp)** aussi les runs qui exécute uniquement avec processus **mpi**.

2. **Scalabilité :**

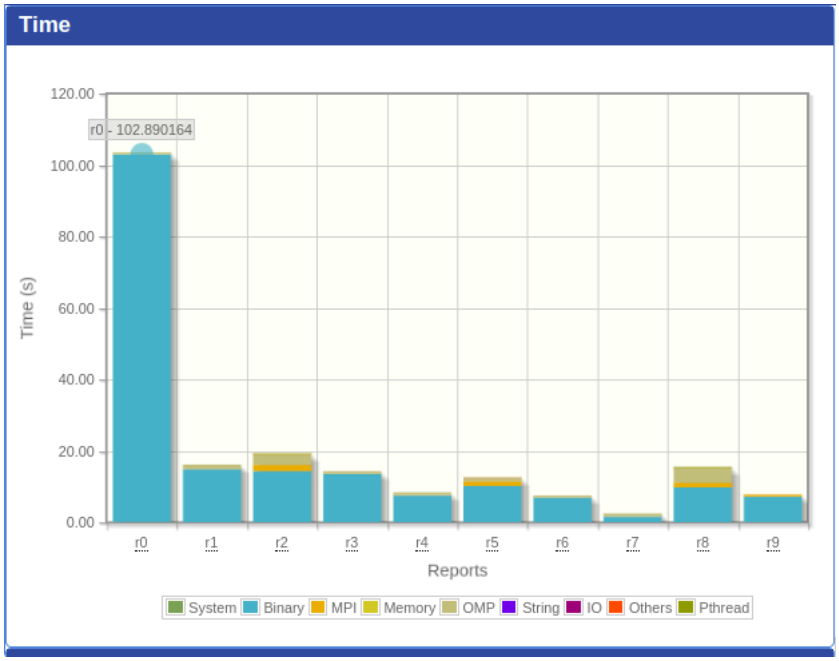


FIGURE 2 – Temps d'exécutions des différents runs

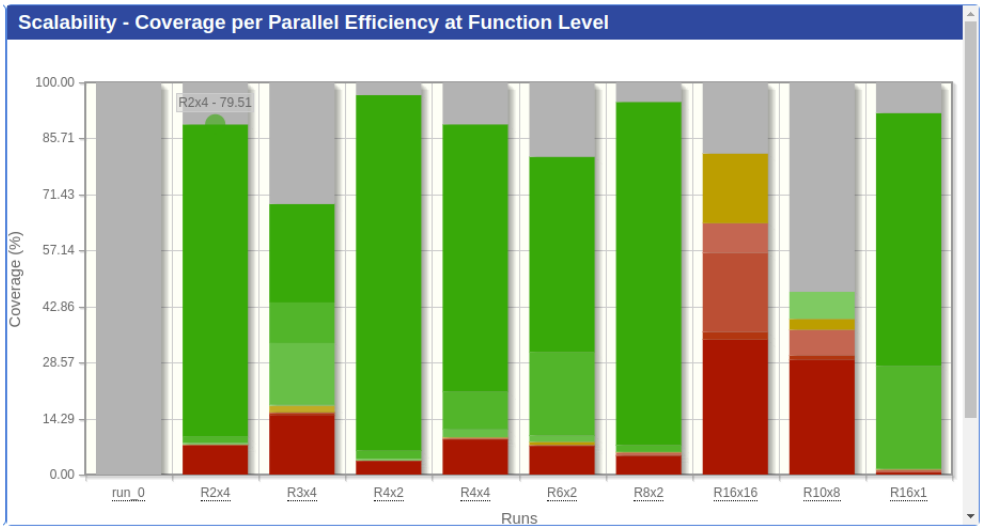


FIGURE 3 – Scalabilité - Couverture par efficacité parallèle

Au niveau de l'efficacité de la parallélisation, on remarque que même si le run 7 est le plus rapide mais il reste le plus mauvais au niveau de l'ordonnancement parallèle car il passe beaucoup plus de temps dans les runtimes MPI et/ou OpenMP.

3. **MAQAO thread rank**
   On remarque que la distribution des threads n'est pas uniforme au niveau de run 7 par contre le run 6 il presque identique.

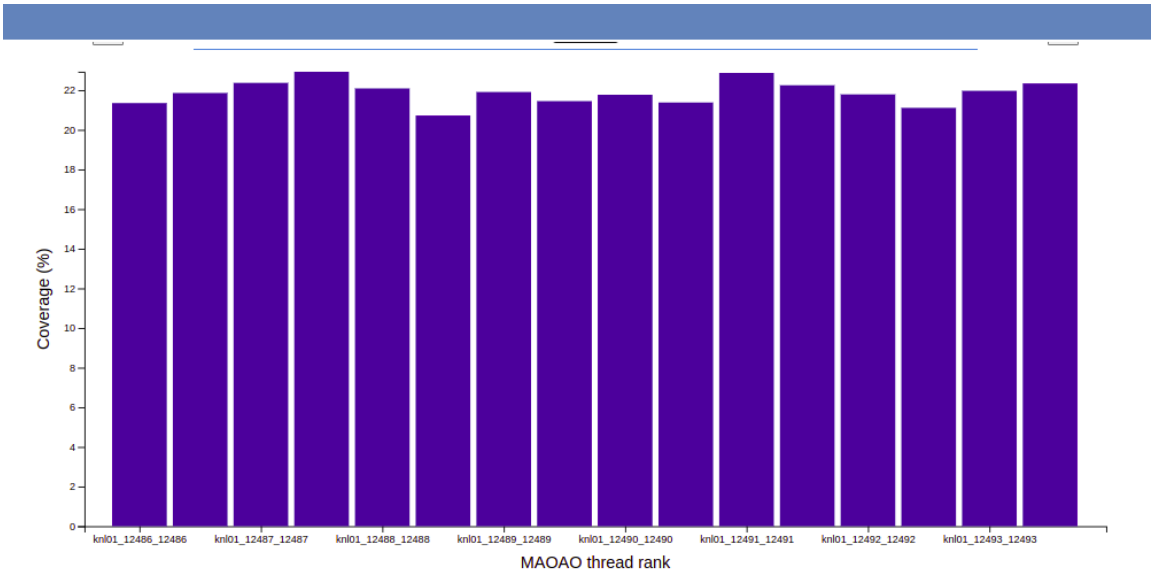


FIGURE 4 – Rang de thread de run 06

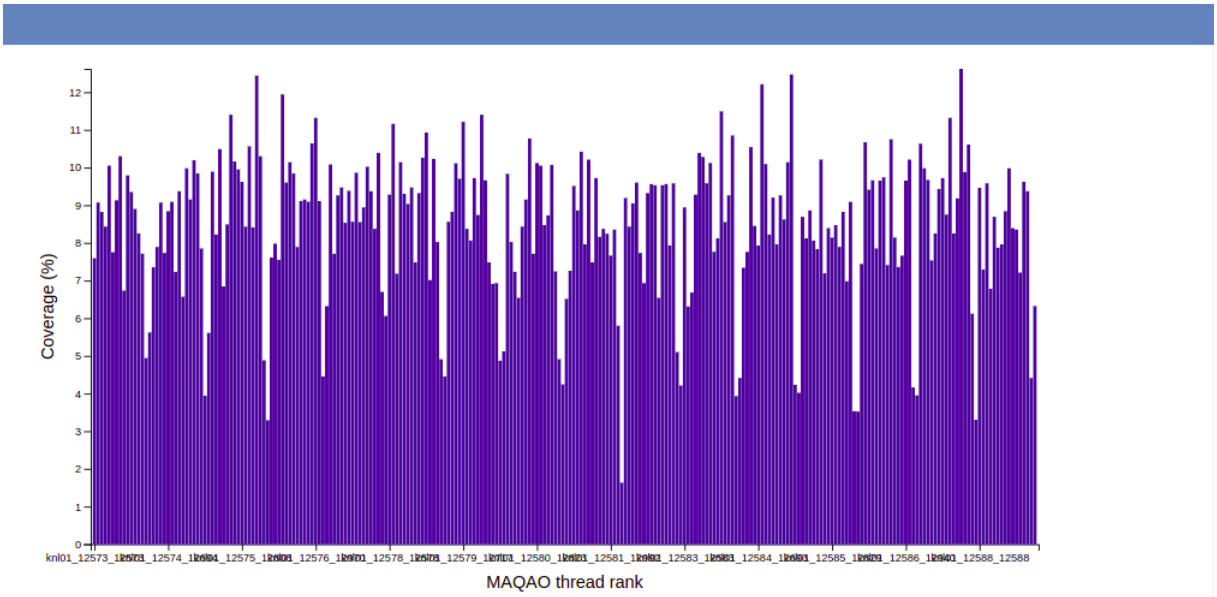


FIGURE 5 – Rang de thread de run 07

4. **Topologie**

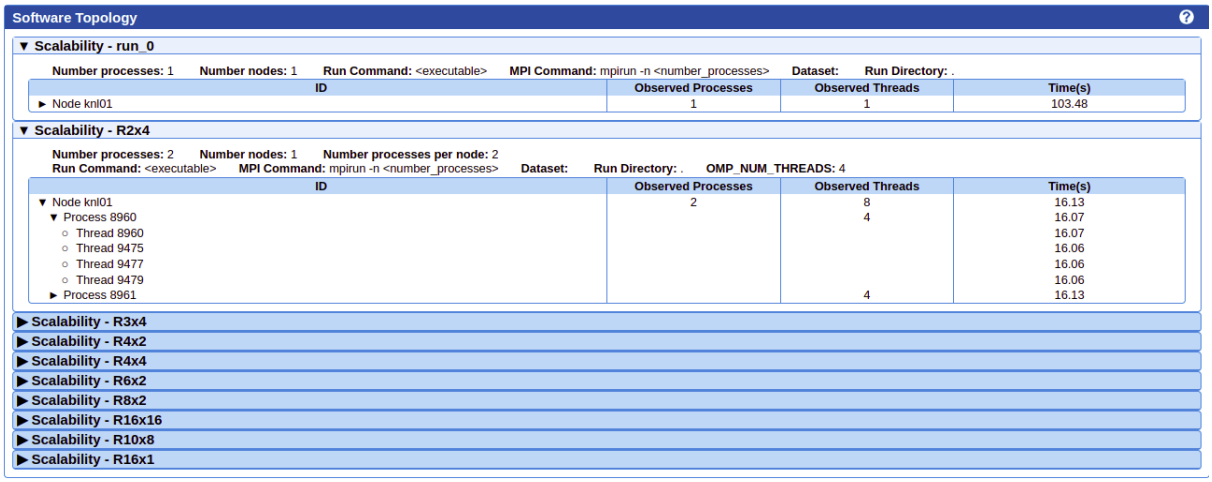


FIGURE 6 – la topologie et vue générale de notre environnement

6.2 Cas Nbody3D

Déjà fait une première version au premier semestre pour son optimisation, on a décide de continue avec ce programme afin de le paralléliser en utilisant MPI, OpenMP et Pthread. Avant de commencer la parallélisation, on va faire un point sur le dernier résultat obtenue au premier semestre :

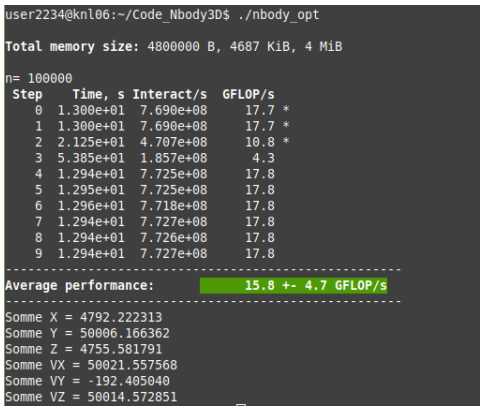


FIGURE 7 – Dernier résultat obtenue au premier semestre

6.2.1 Version parallélisé avec OpenMP

La dernière version du premier semestre à été mise à jour pour être parallélisées à l'aide de OpenMP. Le fichier **nbody_omp.c** est simplement parallélisé en ajoutant des directives pragma sur les boucles externes dans la fonction **move_particles** :

```
void move_particles(particle_t p, const f32 dt, u64 n)
{ //code...
#pragma omp parallel for
  for (u64 i = 0; i < n; i++)
  { //code...
#pragma omp simd
    for (u64 j = 0; j < n; j++)
    {//code...}
  // 3 floating-point operations
#pragma omp parallel for
  for (u64 i = 0; i < n; i ++)
  {//code...}
}
```

Voici le résultat obtenue lors de l'exécution de notre programme :

```
user2234@knl06:~/Code_Nbody3D$ OMP_NUM_THREADS=8 ./nbody_omp

Total memory size: 4800000 B, 4687 KiB, 4 MiB

n= 100000
 Step    Time, s Interact/s  GFLOP/s
    0  1.865e+00  5.363e+09    123.4 *
    1  1.864e+00  5.365e+09    123.4 *
    2  1.863e+00  5.367e+09    123.4 *
    3  1.870e+00  5.347e+09    123.0
    4  1.864e+00  5.365e+09    123.4
    5  1.863e+00  5.369e+09    123.5
    6  1.864e+00  5.365e+09    123.4
    7  1.864e+00  5.366e+09    123.4
    8  1.863e+00  5.368e+09    123.5
    9  1.864e+00  5.365e+09    123.4
-----------------------------------------------------
Average performance:          123.4 +- 0.2 GFLOP/s
-----------------------------------------------------
Somme X = 4792.222313
Somme Y = 50006.166362
Somme Z = 4755.581791
Somme VX = 50021.557568
Somme VY = -192.405040
Somme VZ = 50014.572851
```

FIGURE 8 – Résultat obtenue pour la parallélisation avec OpenMP

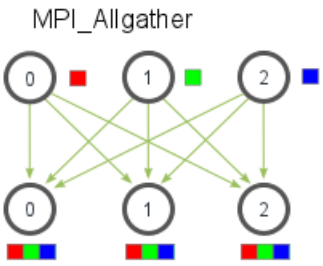
6.2.2 Version parallélisé avec MPI

Une implémentation en utilisant MPI à été conçue pour essayer de tirer parti des performances sur plusieurs nœuds de cluster. Pour cela, la parallélisation MPI sur la fonction principale du programme qui consomme le plus (**move_particles**) est nécessaire.

Voici l'ensemble des principaux changements effectué au niveau de notre code :

```
1  void move_particles(particle_t p, const u64 nb_particles, const f32 dt,
       const int mpi_rank, const int mpi_world_size, u64 particles)
2  {
3    //
4    const int start_particle = mpi_rank * particles;
5    const int end_particle = (mpi_rank + 1) * particles;
6    //code
7    for (u64 i = start_particle; i < end_particle; i++)
8    {//code...}
9    MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, p.x, particles,
       MPI_FLOAT, MPI_COMM_WORLD);
10   MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, p.y, particles,
       MPI_FLOAT, MPI_COMM_WORLD);
11   MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, p.z, particles,
       MPI_FLOAT, MPI_COMM_WORLD);
12   MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, p.vx, particles,
       MPI_FLOAT, MPI_COMM_WORLD);
13   MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, p.vy, particles,
       MPI_FLOAT, MPI_COMM_WORLD);
14   MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, p.vz, particles,
       MPI_FLOAT, MPI_COMM_WORLD);
15 }
16 int main(int argc, char **argv){
17     //code...
18      const int particles = n / mpi_world_size;
19      //code...
20 }
```

Étant donné un ensemble d'éléments répartis sur tous les processus, MPI_Allgather rassemblera tous les éléments de tous les processus. Dans le sens le plus élémentaire, MPI_Allgather est un MPI_Gather suivi d'un MPI_Bcast. L'illustration ci-dessous montre comment les données sont distribuées après un appel à MPI_Allgather.



Voici le résultat obtenue lors de l'exécution de notre programme :

```
user2234@knl06:~/Code_Nbody3D$ mpirun -np 8 ./nbody_mpi
Unexpected KNL MCDRAM cache size 271278080
Unexpected KNL MCDRAM cache size 271278080
Unexpected KNL MCDRAM cache size 271278080
Unexpected KNL MCDRAM cache size 271278080
Unexpected KNL MCDRAM cache size 271278080
Unexpected KNL MCDRAM cache size 271278080
Unexpected KNL MCDRAM cache size 271278080
Unexpected KNL MCDRAM cache size 271278080

Total memory size: 4800000 B, 4687 KiB, 4 MiB

n= 100000
 Step    Time, s Interact/s  GFLOP/s
    0  5.219e-01  1.916e+10    440.7 *
    1  1.667e-01  6.000e+10   1380.1 *
    2  1.606e-01  6.226e+10   1431.9 *
    3  1.600e-01  6.250e+10   1437.6
    4  1.599e-01  6.254e+10   1438.4
    5  1.596e-01  6.266e+10   1441.3
    6  1.597e-01  6.263e+10   1440.4
    7  1.598e-01  6.256e+10   1439.0
    8  1.602e-01  6.243e+10   1435.9
    9  1.597e-01  6.262e+10   1440.3
-----------------------------------------------------
Average performance:       1439.0 +- 1.7 GFLOP/s
-----------------------------------------------------
Somme X = 4792.273684
Somme Y = 50006.168425
Somme Z = 4755.621935
Somme VX = 50022.048241
Somme VY = -192.360922
Somme VZ = 50015.068088
```

FIGURE 9 – Résultat obtenue pour la parallélisation avec MPI

6.2.3 Version parallélisé avec Pthreads

On a fait une implémentation en utilisant openMP et MPI, pour cela on s'est dit qu'une implémentation Pthread serait nécessaire afin de comparer les résultats des trois différentes versions et techniques de parallélisation qu'on a vu en cours.

Et voici les principaux changements effectués au niveau de notre code

```
1  typedef struct srtuct_1
2  {
3    particle_t *p;
4    u64 old;
5    u64 new;
6  } srtuct_1;
7  void *init(void *arg)
8  {
9    srtuct_1 *particule = (srtuct_1 *)arg;
10   for (u64 i = particule->old; i < particule->new; i ++)
11   {//code}
12   pthread_exit(NULL);
13 }
14 void *move_particles(void *arg)
15 {
16   srtuct_1 *particule = (srtuct_1 *)arg;
17     //code
18   for (u64 i = particule->old; i < particule->new; i ++)
19   {
20     particule->p->x[i] += 0.01 * particule->p->vx[i];
21     particule->p->y[i] += 0.01 * particule->p->vy[i];
22     particule->p->z[i] += 0.01 * particule->p->vz[i];
```

```
23    }
24    pthread_exit(NULL);
25  }
26  int main(int argc, char **argv)
27  {
28    pthread_t t1[NB_PTHREADS];
29    pthread_t t2[NB_PTHREADS];
30    srtuct_1 arg1[NB_PTHREADS];
31  //code
32    for (u64 i = 0; i < NB_PTHREADS; i++)
33    {
34      arg1[i].p = p;
35      arg1[i].old = (i * n)+1 / NB_PTHREADS;
36      arg1[i].new = n+1 / NB_PTHREADS;
37    }
38    for (u64 i = 0; i < NB_PTHREADS; i++)
39    {
40      pthread_create(&t1[i], NULL, init, (void *)&arg1[i]);
41    }
42    for (u64 i = 0; i < NB_PTHREADS; i++)
43    {
44      pthread_join(t1[i], NULL);
45    }
46    //code...
47    for (u64 i = 0; i < steps; i++)
48    {
49      const f64 start = omp_get_wtime();
50      for (u64 m = 0; m < NB_PTHREADS; m++)
51      {
52        pthread_create(&t2[m], NULL, move_particles, (void *)&arg1[m]);
53      }
54      for (u64 m = 0; m < NB_PTHREADS; m++)
55      {
56        pthread_join(t2[m], NULL);
57      }
58  //code
59    }
60  }
```

Voici le résultat obtenue lors de l'exécution de notre programme :

```
user2234@knl06:~/Code_Nbody3D$ ./nbody_pthread

Total memory size: 4800000 B, 4687 KiB, 4 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  1.348e+01  7.418e+08     17.1 *
    1  1.356e+01  7.374e+08     17.0 *
    2  1.406e+01  7.114e+08     16.4 *
    3  1.356e+01  7.375e+08     17.0
    4  1.348e+01  7.419e+08     17.1
    5  1.357e+01  7.371e+08     17.0
    6  1.347e+01  7.422e+08     17.1
    7  1.356e+01  7.373e+08     17.0
    8  1.349e+01  7.414e+08     17.1
    9  1.357e+01  7.369e+08     16.9
-----------------------------------------------------
Average performance:          17.0 +- 0.1 GFLOP/s
-----------------------------------------------------
Somme X = 4792.223751
Somme Y = 50006.168259
Somme Z = 4755.584428
Somme VX = 50021.565381
Somme VY = -192.404857
Somme VZ = 50014.574854
```

FIGURE 10 – Résultat obtenue pour la parallélisation avec Pthread

On remarque que les résultats avec Pthread sont pas performants comme on le croyait, du coup on va juste s'intéresser aux résultats de OpenMP et MPI en utilisant MAQAO comme lancement d'analyse et sur un ensemble des runs avec des configurations différentes.

6.2.4 Résultats de l'expérience

Résultat MPI

L'ensemble de configuration de cette expérience :

```
experiment_name = "Analyse de code Nbody3D MPI  en mode scalabilit
    sur le cluster FOB1"
executable = "./nbody_mpi"
number_processes = 1
number_processes_nodes = 1
mpi_command = "mpirun -np <number_processes>"
envv_OMP_NUM_THREADS= 1

multiruns_params = {
{name="R2", number_processes=2,envv_OMP_NUM_THREADS= 1},
{name="R3", number_processes=3,envv_OMP_NUM_THREADS= 1},
{name="R4", number_processes=4,envv_OMP_NUM_THREADS= 1},
{name="R8", number_processes=8,envv_OMP_NUM_THREADS= 1},
{name="R12", number_processes=12,envv_OMP_NUM_THREADS= 1},
{name="R16", number_processes=16,envv_OMP_NUM_THREADS= 1},
{name="R20", number_processes=20,envv_OMP_NUM_THREADS= 1},
{name="R32", number_processes=32,envv_OMP_NUM_THREADS= 1},
}
```

On a lancé une analyse maqao sur notre programme MPI en mode multirun avec des configurations différentes afin de visualiser les différents résultats obtenues de ce dernier et déterminer le meilleure paramètre qui nous donne un résultat plus performant que les autres.

Global Metrics

Metric		r0	r1	r2	r3	r4	r5	r6	r7	r8
Total Time (s)		136.64	24.52	12.16	7.39	3.46	2.95	3.26	3.41	4.84
Profiled Time (s)		134.17	23.98	11.63	6.76	2.26	1.19	0.93	0.61	0.50
Time in analyzed loops (%)		99.9	99.1	96.6	95.7	86.0	75.1	60.0	52.9	30.9
Time in analyzed innermost loops (%)		99.1	99.0	96.2	94.6	84.5	74.0	58.5	50.5	29.1
Time in user code (%)		99.9	99.2	96.6	95.8	86.1	75.2	60.3	53.2	31.2
Compilation Options		OK	OK	OK	OK	OK	OK	OK	OK	OK
Perfect Flow Complexity		1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Array Access Efficiency (%)		100.0	100.0	99.9	99.9	100.0	99.2	99.2	99.5	99.0
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00	1.01	1.03	1.06	1.21	1.18	1.35
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.03	1.03	1.07	1.16	1.32	1.38	1.86
No Scalar Integer	Potential Speedup	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.01	1.01
	Nb Loops to get 80%	2	1	1	1	1	1	1	1	1
FP Vectorised	Potential Speedup	1.00	1.00	1.00	1.01	1.01	1.04	1.04	1.04	1.04
	Nb Loops to get 80%	1	2	3	2	2	2	2	3	3
Fully Vectorised	Potential Speedup	1.00	1.00	1.01	1.01	1.02	1.05	1.05	1.06	1.05
	Nb Loops to get 80%	1	2	3	3	2	3	3	2	3
Only FP Arithmetic	Potential Speedup	1.26	1.26	1.26	1.25	1.23	1.20	1.15	1.14	1.09
	Nb Loops to get 80%	1	1	1	1	1	1	1	2	2
Scalability - Gap		1.00	0.36	0.27	0.22	0.20	0.26	0.38	0.50	1.13

FIGURE 11 – Global Metrics Nbody3D avec multiruns MPI

On remarque que le run R5 est le meilleur run en temps d'exécution avec 12 process, malgré ça, on a l'ordonnancement des entités parallèles qui n'est pas optimal par rapport aux autres runs. On remarque que les meilleures exécutions avec meilleur ordonnancement sont les runs qui possèdent 4 ou 8 entités parallèles (MPI).

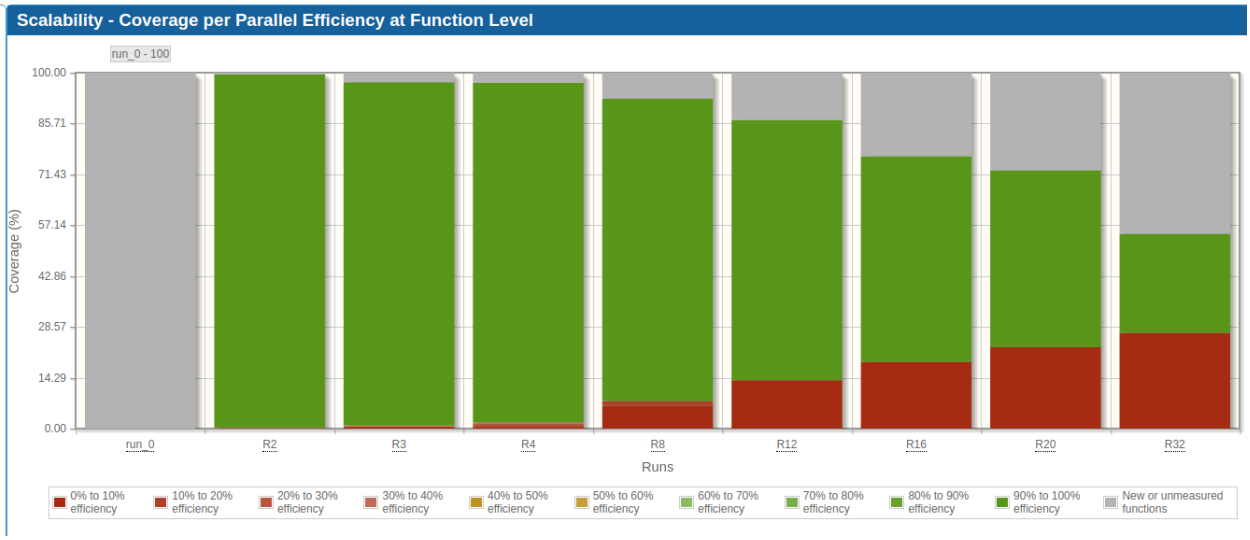


FIGURE 12 – Scalabilité - Couverture par efficacité parallèle pour Nbody3D (MPI)

Au niveau de l'efficacité de la parallélisation, on remarque que même si le run R5 est le plus rapide mais il reste moins performant au niveau de l'ordonnancement parallèle. On remarque que la distribution des threads n'est pas uniforme au niveau de run R5 par contre le run 3 et 4 sont presques identiques.

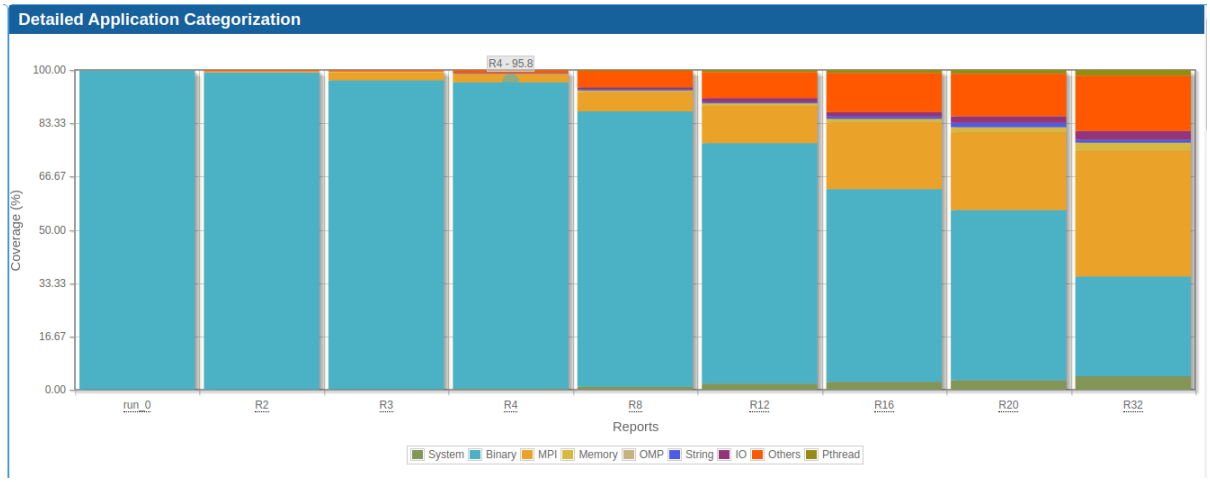


FIGURE 13 – Temps d'exécution des différents runs MPI

Résultat OpenMP

L'ensemble de configuration de cette expérience :

```
1  experiment_name = "Analyse de code Nbody3D OMP  en mode scalabilit
       sur le cluster FOB1"
2  executable = "./nbody_omp"
3  number_processes = 1
4  number_processes_nodes = 1
5  envv_OMP_NUM_THREADS= 1
6
7  multiruns_params = {
8  {name="R2", envv_OMP_NUM_THREADS= 2, number_processes=1},
9  {name="R3", envv_OMP_NUM_THREADS= 3, number_processes=1},
10 {name="R4", envv_OMP_NUM_THREADS= 4, number_processes=1},
11 {name="R8", envv_OMP_NUM_THREADS= 8, number_processes=1},
12 {name="R12",envv_OMP_NUM_THREADS= 12, number_processes=1},
13 {name="R16",envv_OMP_NUM_THREADS= 16, number_processes=1},
14 {name="R20",envv_OMP_NUM_THREADS= 20, number_processes=1},
15 {name="R32",envv_OMP_NUM_THREADS= 32, number_processes=1},
16 }
```

On a lancé une analyse maqao sur notre programme OpenMP en mode multirun avec un nombre différents de OMP_Threads, afin de visualiser les différents résultats et déterminer le meilleure paramètre qui nous donne un résultat plus performant que les autres.

On remarque que le run R8 avec 32 threads est le meilleur run en temps d'exécution ainsi que l'ordonnancement des entités parallèles, et ça même si on augmente le nombre de threads on aura toujours un résultat plus performant que les autres cas pour OpenMP.

Global Metrics

Metric		r0	r1	r2	r3	r4	r5	r6	r7	r8
Total Time (s)		133.89	76.16	51.02	38.11	20.04	13.41	10.88	7.92	5.77
Profiled Time (s)		131.97	75.18	50.36	37.55	19.67	13.07	10.51	7.66	5.45
Time in analyzed loops (%)		99.9	99.9	99.9	99.9	99.8	99.7	99.7	99.7	99.4
Time in analyzed innermost loops (%)		99.5	99.4	99.2	99.6	99.0	99.0	99.0	99.3	98.9
Time in user code (%)		99.9	99.9	99.9	99.9	99.8	99.7	99.6	99.7	99.4
Compilation Options		OK	OK	OK	OK	OK	OK	OK	OK	OK
Perfect Flow Complexity		1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Array Access Efficiency (%)		100	100	100	100	100	100	100	100	100
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.01	1.01
No Scalar Integer	Potential Speedup	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Nb Loops to get 80%	1	1	1	1	1	1	1	1	1
FP Vectorised	Potential Speedup	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Nb Loops to get 80%	1	1	1	1	1	1	1	1	1
Fully Vectorised	Potential Speedup	1.00	1.00	1.00	1.00	1.01	1.00	1.00	1.00	1.00
	Nb Loops to get 80%	1	1	1	1	1	1	1	1	1
Only FP Arithmetic	Potential Speedup	1.26	1.26	1.26	1.27	1.26	1.26	1.26	1.26	1.26
	Nb Loops to get 80%	1	1	1	1	1	1	1	1	1
Scalability - Gap		1.00	1.14	1.14	1.14	1.20	1.20	1.30	1.18	1.38

FIGURE 14 – Global Metrics Nbody3D avec multiruns OpenMP

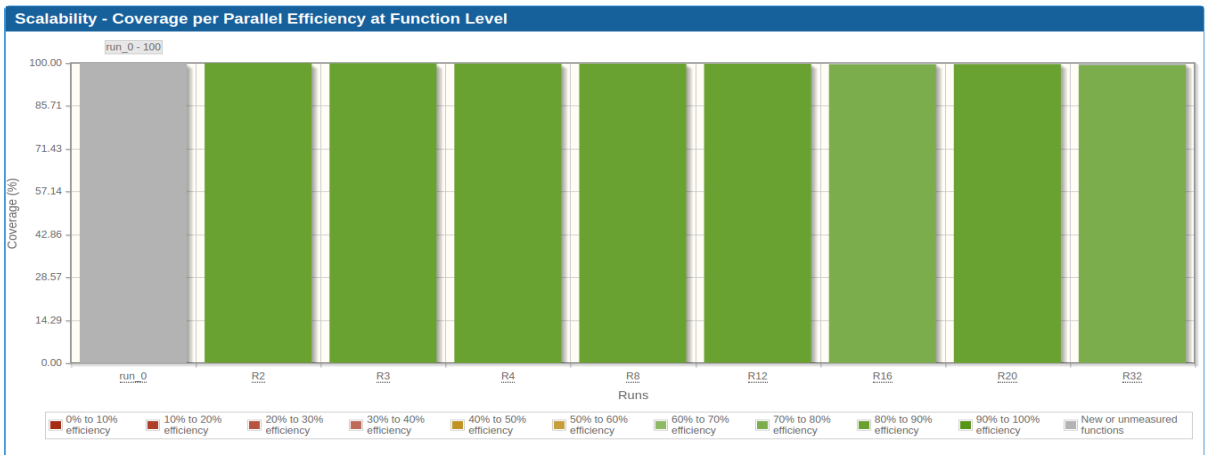


FIGURE 15 – Scalabilité - Couverture par efficacité parallèle pour Nbody3D (OpenMP)

Au niveau de l'efficacité de la parallélisation, on remarque qu'elle est parfaite pour l'ensemble des runs.

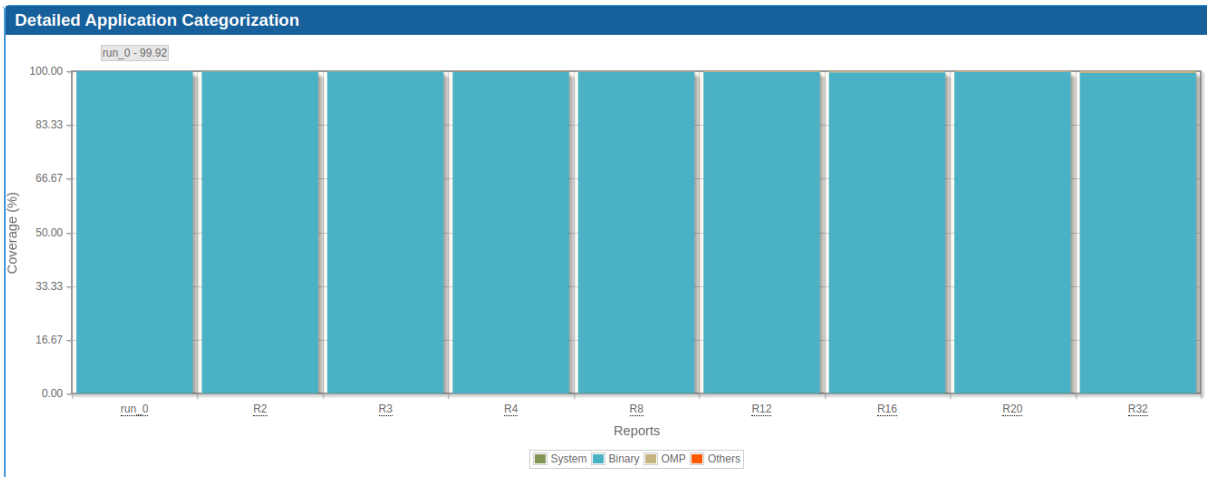


FIGURE 16 – Temps d'exécution des différents runs OpenMP

Malgré que dans la parallélisation MPI on a gangé énormément du temps mais notre programme passe le plus de temps dans les runtimes MPI à chaque augmentation de nombres de process par contre dans la parallélisation OpenMP, on peut voir qu'on passe le même temps dans le code de l'utilisateur sur les différents runs testés et les entités parallèles sont bien gérées.

7 Conclusion

Pour conclure, nous pouvons dire que MAQAO offre des possibilités afin d'optimiser des programmes ainsi que l'identification de la bonne topologie afin de mettre en marche son exécution, en plus de ça , il nous offre la possibilité d'observer l'évolution du temps de calcul et aussi il nous permet d'identifier les parties de codes qui n'utilisent pas toute les performances de notre machine, afin de porter des modifications au programme pour obtenir des meilleurs performances.

Durant notre projet, on a pu améliorer notre programme sur les bases des rapports MAQAO obtenues ensuite on a pu paralléliser notre programme avec divers techniques et ce dernier nous a permis de bien visualiser nos résultats en utilisant plusieurs configurations afin de déduire les meilleurs paramètres qui nous permettent d'avoir des bonnes performances en utilisant le module ONEVIEW qui permet de gérer des expériences et stocker les résultats dans un format simple qui est facile à lire, une interface claire et à tout moment on pourra revenir aux anciens résultats obtenues afin d'observer les résultats rapportés par les changements effectués sur le code.

Enfin, on peut dire qu'on a eu un outil très puissant capable de guide pour l'amélioration de plusieurs critères, en ce qui concerne l'utilisation des registres, les accès mémoires, l'accélération de calculs et observation des phases d'optimisation et de parallélisation, il offre un environnement d'expérimentation simple, un rapport CQA détaillé et un affichage des données lisibles.

Annexe :

`https://github.com/Madjid-Bzr/PPN`

Bibliographie

http://www.maqao.org/
http://www.maqao.org/release/MAQAO_QuickReferenceSheet_V12.pdf
http://www.maqao.org/release/MAQAO.Tutorial.ONEVIEW.pdf
http://www.maqao.org/release/MAQAO.Tutorial.LProf.pdf
http://www.maqao.org/release/MAQAO.Tutorial.CQA.intel64.pdf
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
https://www.nas.nasa.gov/assets/npb/NPB3.4.2.tar.gz
https://man7.org/linux/man-pages/man7/pthreads.7.html
https://www.openmp.org/
https://www.open-mpi.org/doc/v4.0/man3/MPI.3.php