# TDP 2 - PRCD
# Problème à N corps avec MPI

Etienne THIERY - Youssef SEDRATI

3 novembre 2016

Dans ce projet nous implémentons un simulateur simple du problème à N-corps, puis le parallélisons en utilisant MPI

## 1 Utilisation

### 1.1 Compilation

La compilation du projet nécessite uniquement la présence d'une implémentation d'openMPI sur la machine (module mpi/openmpi/gcc/1.10.1-tm sur PlaFRIM). L'ensemble du projet peut être compilé via la commande :

```
make
```

### 1.2 Exécution

Une fois compilé, l'exécutable principal (`simulator`) peut être appelé avec différentes options.

```
mpiexec ./simulator -i inputFilePath [-s] [-o outputDir] [-p P] [-n N] [-r R]
```

Le seul argument obligatoire, `inputFilePath` est un chemin vers un fichier d'initialisation (voir 2.4). Quelques configurations initiales sont fournies dans le répertoire `inputs` :

- `inputs/earthMoon.dat` est un système de 2 corps représentant la Terre et la Lune. Il permet notamment de vérifier que les résultats de la version séquentielle sont cohérents, et que la version parallèle donne les mêmes en utilisant 2 processeurs.
- `inputs/solarSystem.dat` est un système à 8 corps représentant le Soleil et les 7 premières planètes du système solaire. Il permet notamment de vérifier que la version parallèle donne les mêmes résultats que la version séquentielle sur un nombre plus important de processeurs.
- `inputs/collision.dat` est un système à 16 corps, obtenu par duplication de la configuration précédente, et qui montre une sorte de collision entre deux systèmes. Il joue essentiellement un rôle de démonstration visuelle intéressante.
- `inputs/512.dat` est un système à 512 corps utilisé pour les études de scalabilité. Il représente une sorte de grille, absolument pas réaliste sur un plan physique à notre connaissance, et est généré par un script (`python3 configGenerator.py 512`).
- `inputs/1024.dat` est un système similaire à 1024 corps.
- `inputs/8192.dat` est un système similaire à 8192 corps.
- `raw_dubinski.dat` est un fichier récupéré sur internet contenant les coordonnées 3D de 80000 étoiles, représentant la collision de la voie lactée avec Andromède. Nous n'avons pas eu le temps de l'exploiter.

Les significations des autres options sont les suivantes :

- `-s` est à spécifier pour utiliser la version séquentielle, la version parallèle étant utilisée par défaut.
- `-o` est à spécifier pour activer la génération de fichiers représentant les états successifs du système pendant la simulation, et permettant de générer une animation de son évolution. Il doit être succédé par le chemin du répertoire (existant ou non) dans lequel doivent être générés les fichiers. Attention, si le répertoire contient déjà des fichiers, ceux-ci seront remplacés. Aussi, ne pas utiliser cette option pour mesurer le temps d'exécution du simulateur.

— Par défaut, un fichier est généré à chaque pas de temps, ce qui ralenti l'exécution, et rend la création d'une animation très lente. L'option `-p` permet de spécifier une période plus grande : `-p 10` génère un fichier tous les 10 pas de temps par exemple.
— Par défaut, le pas de temps à chaque itération est choisi de façon à ce que toute particule ne puisse parcourir qu'au plus 0.1 fois la distance la séparant de la particule la plus proche (voir 2.3). Ce ratio peut être modifié avec l'option `-r`.
— Par défaut, le nombre d'itérations effectuées par la simulation est 1000. Il peut être modifié avec l'option `-n`

Une fois un ensemble de fichiers représentant les états successifs du système généré, le script Python `videoGenerator.py` permet de générer une animation de son évolution. Python3 et ffmpeg doivent être installés sur la machine pour cela (pas le cas sur PlaFRIM).

```
python3 videoGenerator inputFilePath outputDir durationInMs videoFileName
```

Note : la vidéo est générée en 20 frames par seconde. Si le nombre de fichiers est trop élevé pour la durée spécifiée, alors plusieurs états sont affichés sur une même frame, ce qui produit un effet de trainée derrière les particules (utile pour en visualiser la vitesse bien que celle-ci soit également affectée par le pas de temps variable). En jouant avec le nombre de fichiers générés par la simulation avec l'option `-p`, cet effet peut être accentué ou supprimé.

Voir `Makefile` pour des exemples complets.

## 1.3 Tests

Le projet inclut quelques tests unitaires, qui peuvent être exécutés via la commande :

```
./tests
```

Une inspection visuelle sur les petits exemples permet de vérifier que les résultats générés par la version séquentielle semble cohérents. Nous avons facilité la génération des vidéos via le Makefile, avec les commandes suivantes :
— `make earthMoonSeq.mp4` (2 secondes de vidéo, en quelques secondes sur un laptop récent)
— `make solarSystemSeq.mp4` (10 secondes de vidéo, en une trentaine de secondes sur un laptop récent)
— `make collisionSeq.mp4` (10 secondes de vidéo, en une trentaine de secondes sur un laptop récent)

Les vidéos résultantes sont également téléchargeables aux liens suivants :
— ethiery.vvv.enseirb-matmeca.fr/files/earthMoonSeq.mp4
— ethiery.vvv.enseirb-matmeca.fr/files/solarSystemSeq.mp4
— ethiery.vvv.enseirb-matmeca.fr/files/collisionSeq.mp4

Pour vérifier que la version parallèle génère bien le même résultat, il suffit de faire un `diff` sur les 2 ensembles de fichiers obtenus. Si aucune ligne ne s'affiche après le `diff` c'est que les fichiers générés sont les mêmes. Nous avons également facilité cette opération avec les commandes suivantes :
— `make compEarthMoon` (2 threads)
— `make compSolarSystem` (8 threads)
— `make compCollision` (16 threads)

## 1.4 Reproduction de l'étude de scalabilité

Les études de scalabilité dont les résultats sont présentés plus bas (voir 3) sont reproducibles une fois connecté sur PlaFRIM en programmant les jobs du répertoire `jobs` :

```
sbatch jobs/job512.sh
sbatch jobs/job512bis.sh
sbatch jobs/job1024.sh
```

```
sbatch jobs/job8192.sh
```

Puis une fois les jobs terminés, les courbes sont tracables avec gnuplot et les scripts du répertoire `generated` :

```
cd generated
gnuplot/fig1.p
gnuplot/fig2.p
gnuplot/fig3.p
gnuplot/fig4.p
gnuplot/fig5.p
gnuplot/fig6.p
gnuplot/fig7.p
```

# 2 Architecture du projet

## 2.1 Modélisation des particules

Les fichiers `Particle.h|c` définissent deux structures :

- `Particle_Private` contient l'ensemble des données d'une particule : masse, position, vitesse, force s'exerçant sur elle et distance minimale de la particule la plus proche.
- `Particle_Shared` contient seulement les données d'une particule nécessaire au calcul de la force gravitationnelle exercée par celle-ci : masse et position. Cela permet d'effectuer des échanges plus petits (24 octets par particule contre 64) et donc plus rapides entre les différentes processus dans la version MPI du solveur.

Ils définissent aussi le type MPI correspondant `MPI_PARTICLE`.

## 2.2 Calcul des intéractions entre particules

Ces mêmes fichiers contiennent également le code permettant de calculer les intéractions entre particules :

- `addInteractions` calcule les forces appliquées par deux `Particle_Private` l'une sur l'autre et les ajoute aux totaux des forces s'exerçant sur elles. Grâce à la troisième loi de Newton ces forces sont égales (de direction opposées), ce qui permet de ne calculer que $\frac{N*(N+1)}{2}$ au lieu de $N*(N+1)$ forces pour un ensemble de $N$ particules à chaque pas de temps dans la version séquentielle.
- `addAction` calcule la force appliquée par une `Particle_Shared` sur une `Particle_Private` mais pas la force réciproque, car l'algorithme adopté pour la version parallèle MPI (topologie en anneau et recouvrement communications/calculs) est moins adapaté à l'astuce précédente consistant à ne calculer que la moitié des intéractions.
- `applyForces` applique la force totale calculée pour une `Particle_Private`, durant un pas de temps donné.

## 2.3 Calcul du pas de temps de la simulation

Le pas de temps séparant deux itérations de la simulation est variable, de façon à simuler de façon plus fine lorsque les particules se rapprochent, et sont donc soumises à des forces plus importantes.

On souhaitait pouvoir fixer un $r \in ]0, 1[$ tel qu'à chaque itération, toute particule ne puisse parcourir qu'une portion $r$ de la distance la séparant de la particule la plus proche. Calculer le pas de temps maximum exact étant relativement compliqué, nous nous sommes contenté d'une majoration.

Soit $p_n$ et $p_{n+1}$ les positions actuelles et à l'itération suivante d'une particule de masse $m$. Soit $\vec{v_n}$ sa vitesse actuelle et $\vec{F_n}$ la force totale s'exerçant sur elle. Soit $d_{min}$ la distance la séparant de la particule la plus proche.

On veut s'assurer que : $|\overrightarrow{p_n p_{n+1}}| < d_{min}$

Or dans notre modèle de calcul : $\overrightarrow{p_n p_{n+1}} = \vec{v_n} dt + \frac{\vec{F_n}}{m} dt^2$

Par l'inégalité triangulaire, notre condition est remplie si : $|\vec{v_n}|dt + |\frac{\vec{F_n}}{m}|dt^2 < d_{min}$

La borne supérieure correspondante sur $dt$ est obtenue dans la fonction `dtUpperBound` par résolution de cette équation du second degré.

### 2.4 Entrées/Sorties

Le format de fichier utilisé pour les entrées est le suivant :

$N$

$m_1\ x_1\ y_1\ vx_1\ vy_1$

$\dots$

$m_N\ x_N\ y_N\ vx_N\ vy_N$

Les fichiers `Util.h|c` définissent les fonctions suivantes :

— `parseNbParticles` permet de lire le nombre de particules d'un fichier.

— `parseInitialConditions` permet d'allouer et d'initialiser un tableau de `Particle_Private` avec les données d'une portion donnée d'un fichier. Cela permet notamment à chaque processus de charger en mémoire uniquement un sous ensemble des données d'un fichier dans la version parallèle MPI.

— `initSnapshotFile` et `getSnapshotFile` permettent respectivement au premier process d'initialiser un fichier représentant l'état d'un système à une itération donnée et à tous les autres de le trouver.

— `dumpSnapshot` permet à un processus d'écrire l'état des particules dont il est en charge dans un fichier créé/récupéré via les 2 fonctions précédentes. Cette fonction est appelée à tour de rôle (et par ordre de rang grâce à un système de jeton tournant) par les processus, de façon à obtenir la totalité des particules dans un seul fichier (pour chaque itération de la simulation).

### 2.5 Implémentations

L'implémentation séquentielle du simulateur se trouve dans les fichiers `sequential.c|h` et l'implémentation parallèle dans `parallel.c|h`.

Ces implémentations n'ont rien de révolutionnaire. Comme mentionné plus haut, la version séquentielle ne calcule que la moitié des intéractions. La version parallèle utilise une topologie en anneau, avec 2 buffers par processus pour pouvoir effectuer un recouvrement calcul communication.

## 3 Étude de scalabilité

### 3.1 Sur un même hôte

Nous avons d'abord mesuré le temps de simulation de 10000 pas de temps sur 512 particles en utilisant 1, 2, 4, 8 puis 16 processeurs d'une même machine. Les figures 1 et 2 montrent les résultats.

On obtient un speedup correct, mais pas parfait probablement car le recouvrement communication calcul lui même n'est pas parfait, et aussi car la réduction lors calcul en commun du pas de temps ne peut pas être recouverte par du calcul.

FIGURE 1 – Temps de simulation en utilisant plus ou moins de coeurs d'une même machine

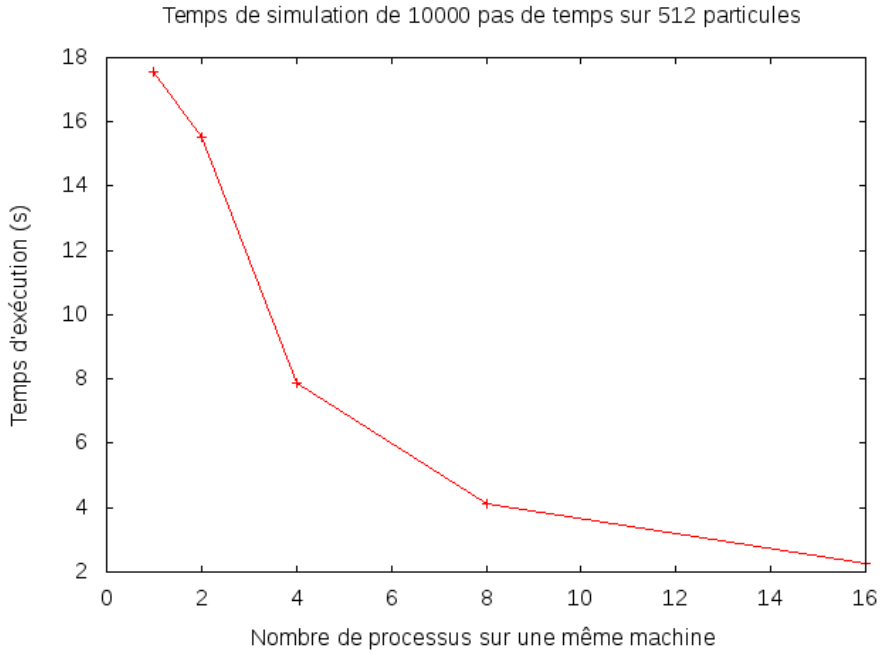
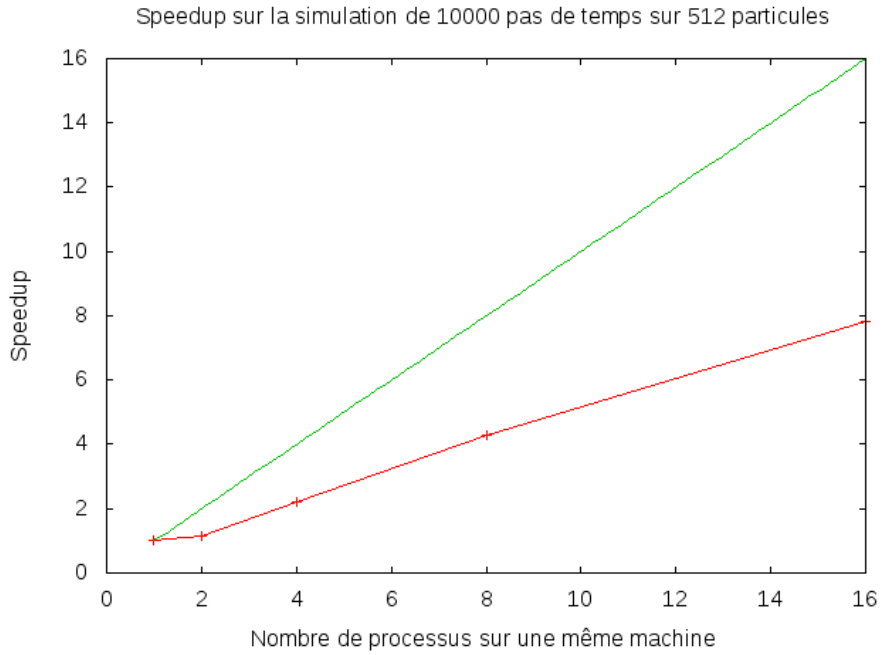



FIGURE 2 – Speed-up en utilisant les coeurs d'une même machine




## 3.2 Sur plusieurs hôtes

Nous avons ensuite essayé d'exécuter la même simulation sur 4 machines, en utilisant 16 coeurs par machine. Les résultats obtenus sont un peu étonnants. Au dela de 16 coeurs, probablement lorsque la charge de travail commence à être répartie sur une seconde machine, les performances s'effondrent, comme le montrent les figures 3 et 4.

Nos hypothèses sont les suivantes :

— Soit notre utilisation de Slurm et MPI est incorrecte, et les mesures avec 32 et 64 processus sont en fait exécutées sur une unique machine. Le nombre trop élevé de processus par rapport au nombre de coeurs disponibles pourrait alors causer un ralentissement.
— Soit ce sont les communications inter machines, souffrant de latences plus importantes et de débits plus faibles, qui sont plus longues que les calculs sensés les recouvrir, et ralentissent donc la simulation.

FIGURE 3 – Temps de simulation en utilisant 4 machines avec 16 coeurs par machine

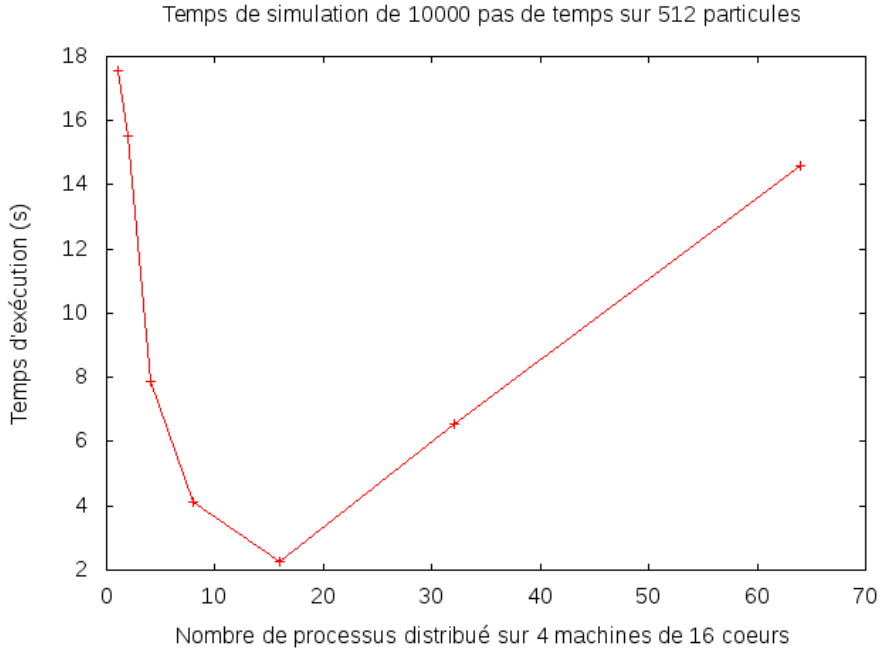
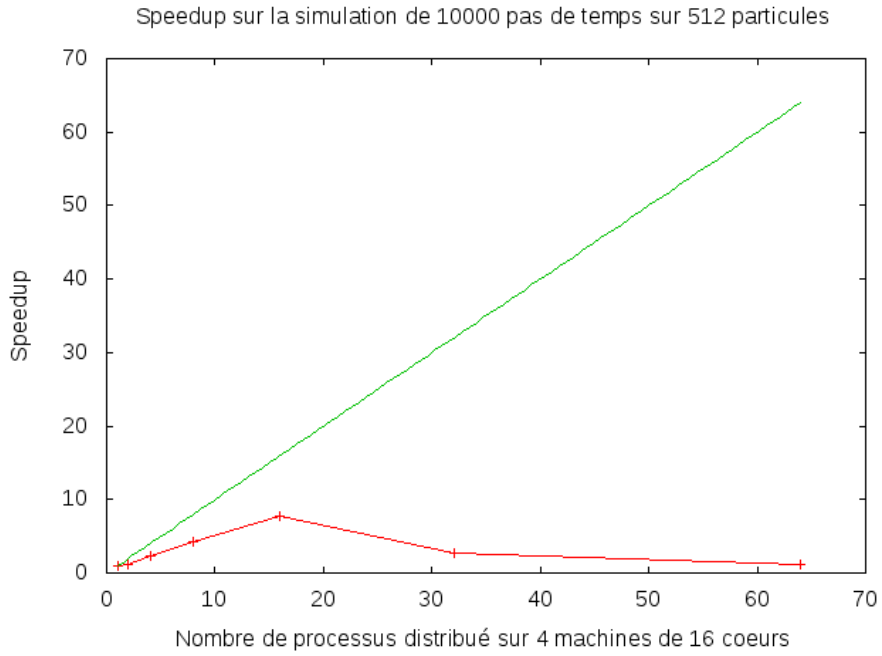



FIGURE 4 – Speed-up en utilisant 4 machines avec 16 coeurs par machine



Pour tester ces hypothèses, nous avons reconduit l'expérience, mais cette fois-ci sur un ensemble de 1024 particules. La quantité de calcul à effectuer par chaque processus augmentant quadratiquement, alors que la taille des données à envoyer augmente linéairement, cela devrait atténuer le ralentissement si notre seconde hypothèse est la bonne.

Effectivement, comme le montre les figures 5 et 6, le speedup sur un nombre de processeurs dépassant 16 est meilleur. Il faudrait utiliser notre code sur des configurations dont le nombre de particules est plus important pour voir apparaître les bénéfices d'une exécution sur plusieurs machines.

FIGURE 5 – Temps de simulation en utilisant 4 machines avec 16 coeurs par machine

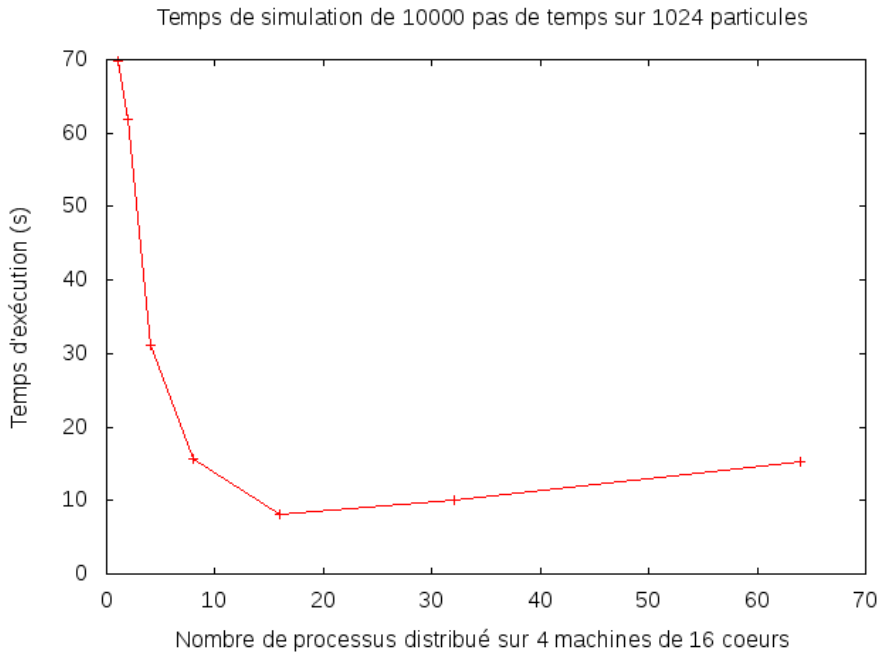
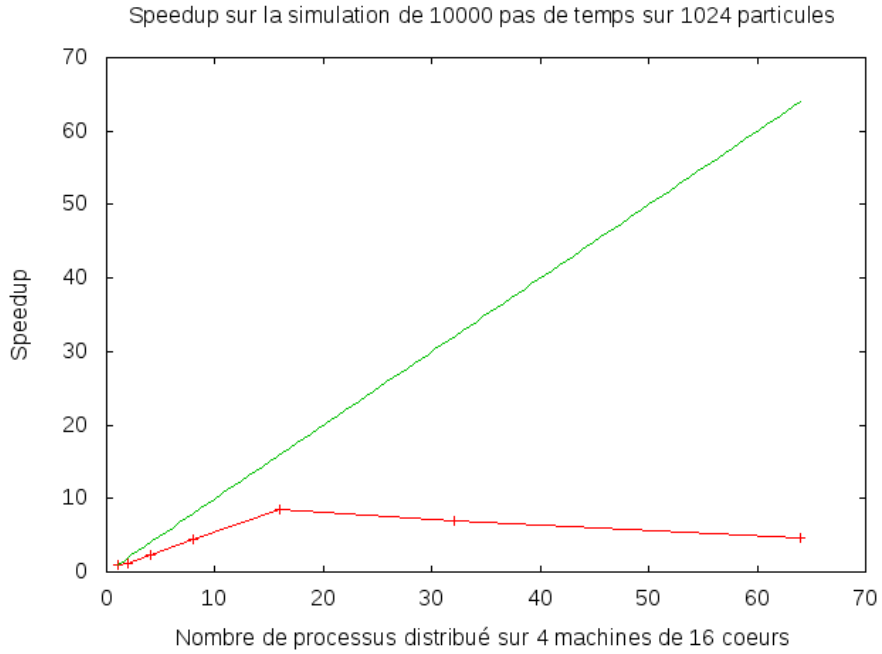


FIGURE 6 – Speed-up en utilisant 4 machines avec 16 coeurs par machine




C'est ce que nous avons fait, avec une simulation d'un ensemble de 8192 particules. Le résultat, en figure 7 confirme notre hypothèse : sur ce problème de plus grosse taille, utiliser plus de 16 coeurs accélère la simulation.

FIGURE 7 – Temps de simulation en utilisant 4 machines avec 16 coeurs par machine