Analyse des performances d'un simulateur de systèmes solaires parallèle

Alexis Chrétien

INF5171
Programmation concurrente et parallèle

18 décembre 2017

UQÀM

1 Description du problème

Le problème principal découlant d'une application simulant l'évolution d'un système planétaire vient du fait que les équations mathématiques définissant cette évolution s'applique plutôt mal lorsqu'on désir simuler un système composé d'un très grand nombre d'éléments.

Considérons un système planétaire S composé de n objets célestes dénotés C_i, $1 \leq i \leq n$, et soi :

- m_i, la masse de l'objet céleste C_i.
- $\vec{p_i}$, le vecteur de position de l'objet céleste C_i.
- $\vec{v_i}$, le vecteur de vélocité de l'objet céleste C_i.
- $\vec{a_i}$, le vecteur d'accélération de l'objet céleste C_i.

Selon la loi de la gravitation universelle de Newton, la force gravitationnelle $\vec{F_{ij}}$ exercé sur l'objet céleste C_i par l'objet céleste C_j est donné par :

$$\vec{F_{ij}} = \frac{G m_i m_j}{{d_{ij}}^2} \ \hat{\mathbf{d}}_{\mathbf{ij}}$$

où G représente la constante gravitationnelle ($G \approx 6.67 \times 10^{-11} \frac{m^3}{kg \cdot s^2}$) et d_{ij} représente la distance entre les vecteurs p_i et p_j ($\vec{d_{ij}} = \vec{p_j} - \vec{p_i}, \quad \vec{d_{ij}} = d_{ij}\hat{\mathbf{d}}_{\mathbf{ij}}$).

Nous savons aussi que, selon la seconde loi de Newton, une masse m sur laquelle est exercée une force $\vec{F}$ subit une accélération $\vec{a}$ définie par :

$$\vec{F} = m\vec{a}$$

En combinant les deux équations, nous obtenons :

$$\vec{F_{ij}} = \frac{G m_i m_j}{{d_{ij}}^2} \ \hat{\mathbf{d}}_{\mathbf{ij}} = m_i \vec{a_i}$$
$$\vec{a_i} = \frac{G m_j}{{d_{ij}}^2} \ \hat{\mathbf{d}}_{\mathbf{ij}}$$

Enfin, il en découle que l'accélération qui est exercée sur l'objet céleste C_i dans un système composé de n objets célestes est donné par :

$$\vec{a_i} = G \sum_{i=1}^{n} |_{i \neq j} \left(\frac{m_j}{{d_{ij}}^2} \hat{\mathbf{d}}_{\mathbf{ij}} \right)$$

En connaissant cette valeur d'accélération, il est possible d'estimer les nouvelles valeurs de vélocité et de position de l'objet céleste C_i en utilisant une intervalle de temps Δt suffisamment petite :

$$\vec{v_{i(t+\Delta t)}} \approx \vec{v_{i(t)}} + \vec{a_i} \Delta t$$
$$\vec{p_{i(t+\Delta t)}} \approx \vec{p_{i(t)}} + \vec{v_i} \Delta t$$

La conséquence de ces équations est que, pour chaque intervalle de temps Δt, le programme doit calculer la somme des $n - 1$ accélérations exercées sur chacun des n objets célestes du système. Un programme qui simule l'évolution d'un système planétaire en utilisant les lois Newtoniennes est donc caractérisé par une complexité temporelle de l'ordre de $\Theta(n^2)$, et verra ses temps d'exécution grandir exponentiellement en fonction du nombre d'objets célestes. Bien qu'une implémentation parallèle de l'algorithme de la simulation ne pourra pas diminuer la complexité temporelle, elle pourra tout de moins diminuer les temps d'exécutions. La magnitude de cette amélioration est le sujet de ce travail.

2 Description de l'approche de la solution parallèle

Soit le pseudo-code de l'implémentation de l'algorithme séquentiel de l'exécution d'une simulation:

procédure executerSequentielle

---

entrée :
ObjCel : tableau d'instances de classe *ObjetCeleste* indicé de 1 à n.
Δt : Le temps qui doit s'écouler entre chaque mise à jour des valeurs de positions, de vélocités et d'accélérations des éléments de *ObjCel*.
duree : La durée de la simulation.

début

```
1   pour t ← 1 à duree pas Δt faire
2     pour i ← 1 à n pas 1 faire
3       ObjCel[i].acceleration ← [0, 0, 0]
4       pour j ← 1 à n pas 1 faire
5         si i ≠ j alors
6           ObjCel[i].acceleration ← ObjCel[i].acceleration + accGrav(ObjCel[i], ObjCel[j])
7         fin si
8       fin pour
9       ObjCel[i].velocite ← ObjCel[i].velocite + ObjCel[i].acceleration × Δt
10      ObjCel[i].position ← ObjCel[i].position + ObjCel[i].velocite × Δt
11    fin pour
12  fin pour
```

fin executerSequentielle

La spécification de la méthode *accGrav* a été omise afin d'alléger l'algorithme, mais il s'agit d'une méthode procédant à un nombre constant d'opérations arithmétiques pour retourner l'accélération gravitationnelle exercée par le deuxième objet céleste en argument sur le premier.

Nous observons que les opérations effectuées sur les éléments du tableau *ObjCel* sont parallélisables, ou presque. Comme la valeur d'accélération à un temps T d'un objet céleste donnée dépend des positions des autres objets célestes du système au temps $T - \Delta t$, il est nécessaire que les valeurs d'accélérations de tous les objets célestes aient été calculées avant de mettre à jour la moindre valeur de position, sans quoi une implémentation parallèle de cet algorithme produirait des valeurs de positions plus ou moins différentes que la version séquentielle, particulièrement si Δt est grand. Néanmoins, il serait possible de contrer ce problème en découpant la boucle **pour** en deux : Une boucle effectuée de façon parallèle pour calculer la nouvelle accélération et mettre a jour la vélocité de chaque objet céleste, suivit d'une seconde boucle parallèle pour mettre à jour les valeurs de positions.

procédure executerParallele

---

entrée :
ObjCel : tableau d'instances de classe *ObjetCeleste* indicé de 1 à n.
Δt : Le temps qui doit s'écouler entre chaque mise à jour des valeurs de positions, de vélocités et d'accélérations des éléments de *ObjCel*.
duree : La durée de la simulation.

début
1 **pour** $t \leftarrow 1$ **à** *duree* **pas** Δt **faire**
2 **parallel_for** $i \leftarrow 1$ **à** n **faire**
3 $ObjCel[i].acceleration \leftarrow [0,0,0]$
4 **pour** $j \leftarrow 1$ **à** n **pas** 1 **faire**
5 **si** $i \neq j$ **alors**
6 $ObjCel[i].acceleration \leftarrow ObjCel[i].acceleration + accGrav(ObjCel[i], ObjCel[j])$
7 **fin si**
8 **fin pour**
9 $ObjCel[i].velocite \leftarrow ObjCel[i].velocite + ObjCel[i].acceleration \times \Delta t$
10 **fin parallel_for**
11 **parallel_for** $i \leftarrow 1$ **à** n **faire**
12 $ObjCel[i].position \leftarrow ObjCel[i].position + ObjCel[i].velocite \times \Delta t$
13 **fin parallel_for**
14 **fin pour**
fin executerParallele

Comme ces deux boucles parallèles peuvent être effectuées sur une collection, et que chaque tâche contient toujours le même nombre d'opérations de mêmes natures (lourdeur des tâches constante), le parallélisme de données tel qu'offert par la librairie *TBB* via la méthode *parallel_for* serait un choix intéressant, particulièrement si on suppose que l'on veuille surtout procéder à des simulations contenant un grand nombre d'objets célestes n (n au moins plus grand que le nombre de processeurs). Même si l'implémentation de la méthode *parallel_for* de *TBB* suit en réalité de patron de parallélisme de tâches, la forme concrète que celle-ci prend est très similaires à un *pmap* en Ruby (parallélisme de donnée). Dans les deux cas, il s'agit d'effectuer une même série d'opérations sur chaque élément d'une collection. De plus, supposant que l'implémentation de la librairie associe initialement un même nombre d'objets célestes à chaque tâche, nous savons que chacune de ces tâches représentera environ la même quantité de travail. Cette constance devrait faire en sorte que, pour un grand nombre d'objets célestes, peu d'occurrences de "vols de tâches" devrait se produire, et leurs impacts négatifs sur les performances devrait être ainsi négligeables.

3 Résultats expérimentaux

Les résultats expérimentaux ont été produits via l'exécutable **SimMesures**, dont le code source se trouve à l'emplacement **src/SimMesures.cpp**. Cet exécutable produit le fichier **data/mesures.dat**, qui peut être utilisé par *Gnuplot* pour produire des graphes des temps d'exécutions et des accélérations aux emplacements **data/temps.png** et **data/accelerations.png** (voir le script **src/gnuplotMesures.gpi**). La compilation, l'exécution et la production des graphes peut être fait d'un seul coup via la commande **make mesures**.

Le programme **SimMesures** effectue les mesures sur les temps et les accélérations absolues des méthodes séquentielles et parallèles en utilisant 4, 16, 32 et 64 threads. pour chaque version de la méthode et pour chaque valeur de nombre N d'objets célestes, le programme effectue trois mesures. La moyenne de ces trois mesures est utilisé pour définir le temps d'exécution pour cette version de la méthode et cette valeur de N, et son accélération absolue est obtenue en divisant cette moyenne par la moyenne du temps d'exécution de la méthode séquentielle pour le même nombre N.

Il faut noter, bien qu'il ne soit absolument pas recommandé de spécifier à la librairie *TBB* quel nombre de threads utiliser, il s'avère quand même intéressant de prendre ces données en compte pour déduire le niveau de dimensionabilité de l'algorithme parallèle.

Veuillez aussi noter qu'il est possible de spécifier le nombre d'objets minimal, maximal, les intervalles de nombre d'objets et la durée des simulations via les constantes du fichier **src/SimMesures.cpp**. Pour les fins de cette analyse, trois suites de prises de mesures ont été effectuées (toutes en utilisant une durée de 100 jours) :

- Mesures 1 : N allant de 1 à 64 à pas de 1
- Mesures 2 : N allant de 50 à 500 à pas de 50
- Mesures 3 : N allant de 500 à 2500 à pas de 500

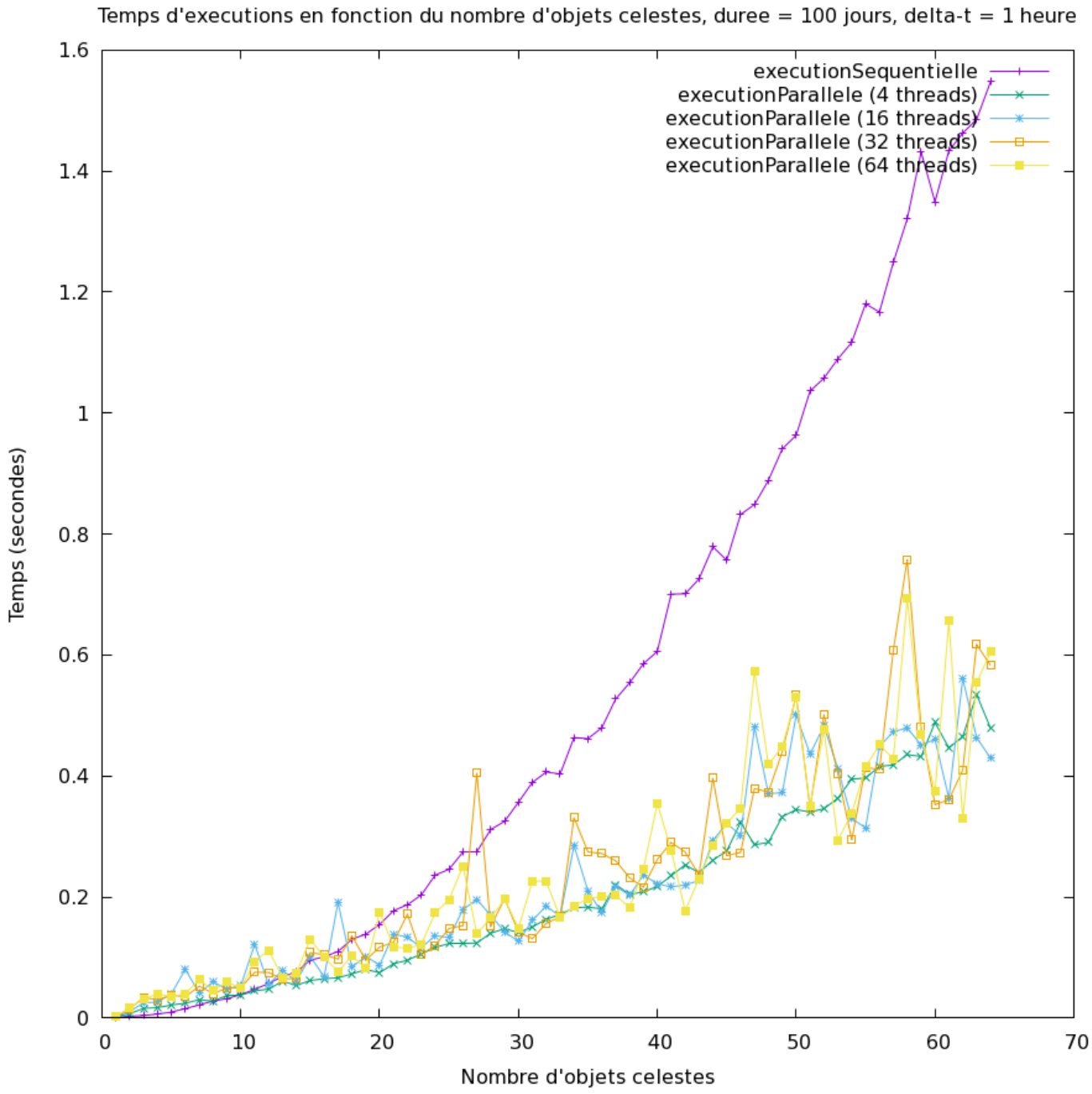
En étudiant les graphes produits par ces trois suites de mesures, nous pouvons remarquer certains points:

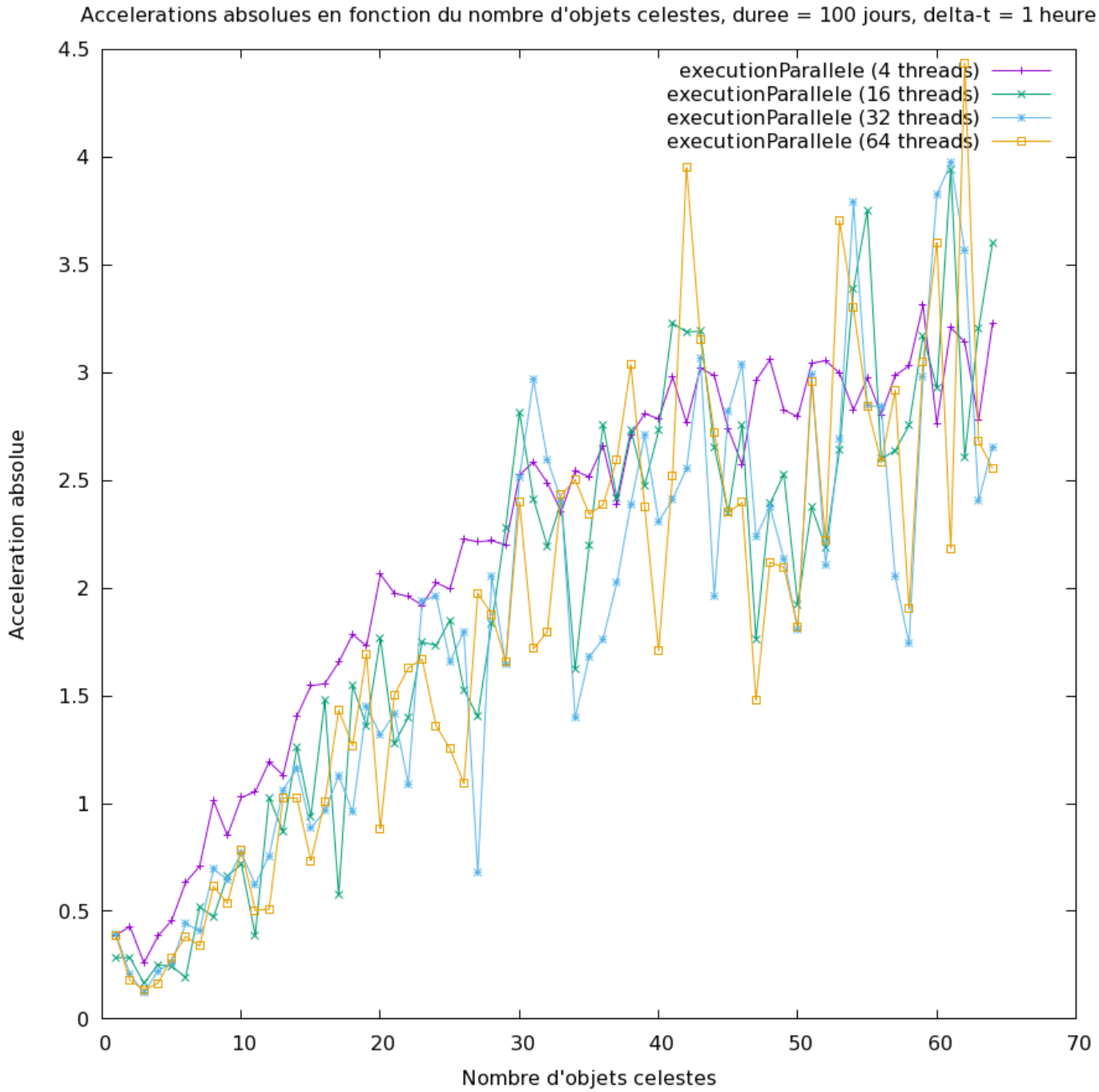
- Dans la première suite de mesures, nous remarquons qu'il faut assez peu d'objets avant que la version séquentielle produise des temps d'exécutions inférieurs aux versions parallèles. Avec $N > 20$, les versions parallèles produisent presque toujours des accélérations absolues supérieures à 1 pour tout nombre de threads.
- On remarque également que, pour les petites valeurs que prend N dans cette suite de mesure, l'amplitude des écarts/l'instabilité des données successives pour les versions parallèles semble proportionnel au nombre de threads.
- De plus, on note que, pour de très petites valeurs N ($N < 30$), la version parallèle utilisant 4 threads se démarque légèrement des autres, en produisant des valeurs d'accélérations légèrement plus élevées.
- Remarquons que pour des valeur de N supérieur à 200, l'ordre des valeurs d'accélération est le même que l'ordre des nombre de threads (plus de threads implique de meilleurs accélérations).
- Tel qu'on s'y attendait à la section 1, la parallélisation de l'algorithme ne peut modifier sa complexité temporelle, mais nous observons tout de même une bonne diminution des temps d'exécution et de bonnes accélérations, allant jusqu'à 43 pour la version avec 64 threads et avec $N = 2500$.
- Enfin, on remarque que les accélérations des versions parallèles tendent lentement vers une valeur limite, toujours inférieur à la limite théorique du nombre de threads disponibles. Il semble également que les versions parallèles utilisant le moins de threads soient les plus efficaces pour un très grand nombre N : leur quotient de la valeur d'accélération absolue sur le nombre de threads est plus élevé. Soit $E_{nbThread}$ l'efficacité de l'algorithme parallèle pour $N = 2500$. Nous avons

$$E_4 \approx \frac{3.94}{4} \approx 0.99$$
$$E_{16} \approx \frac{13.40}{16} \approx 0.84$$

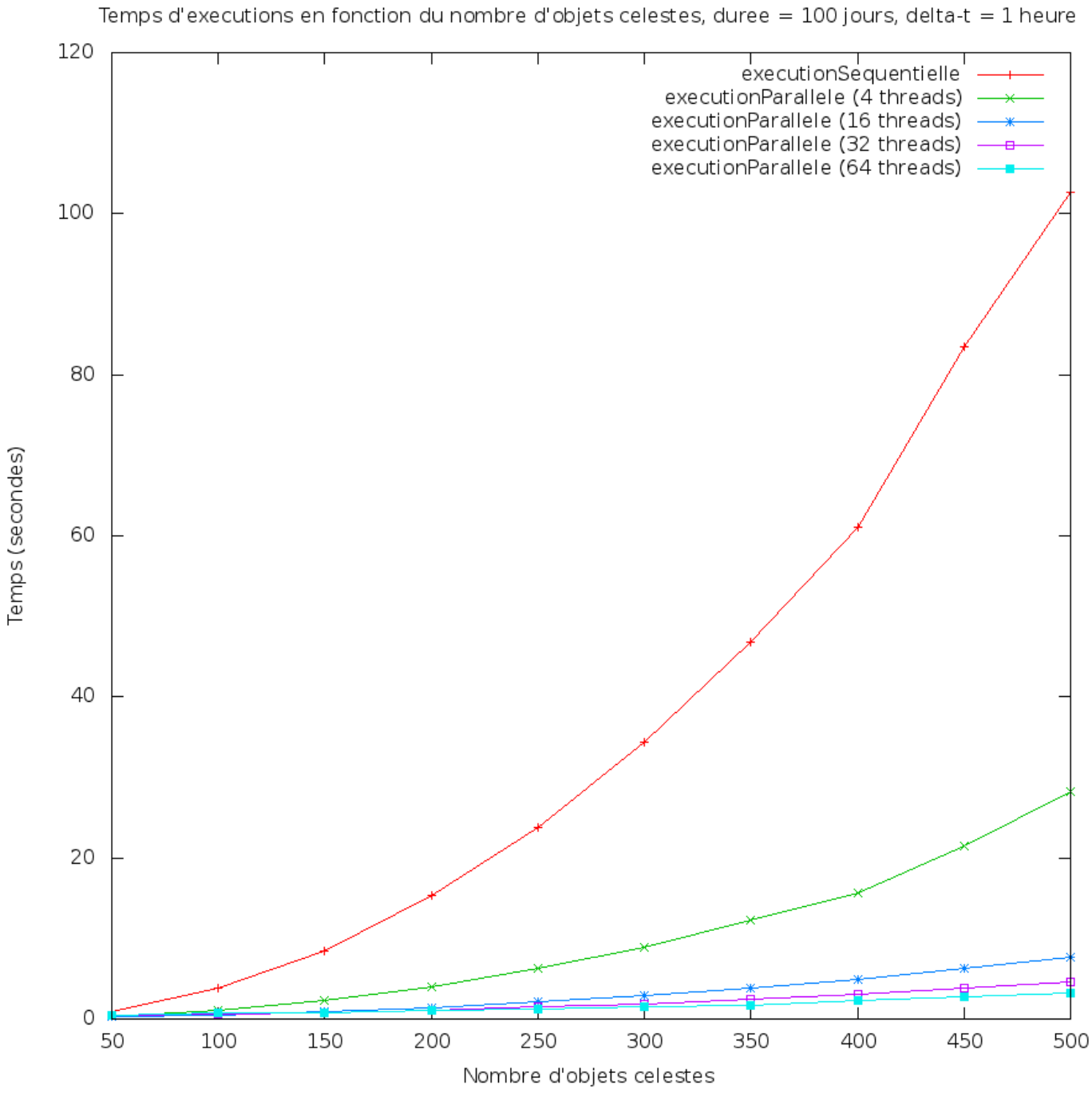
$$E_{32} \approx \frac{24.79}{32} \approx 0.77$$
$$E_{64} \approx \frac{43.04}{64} \approx 0.67$$

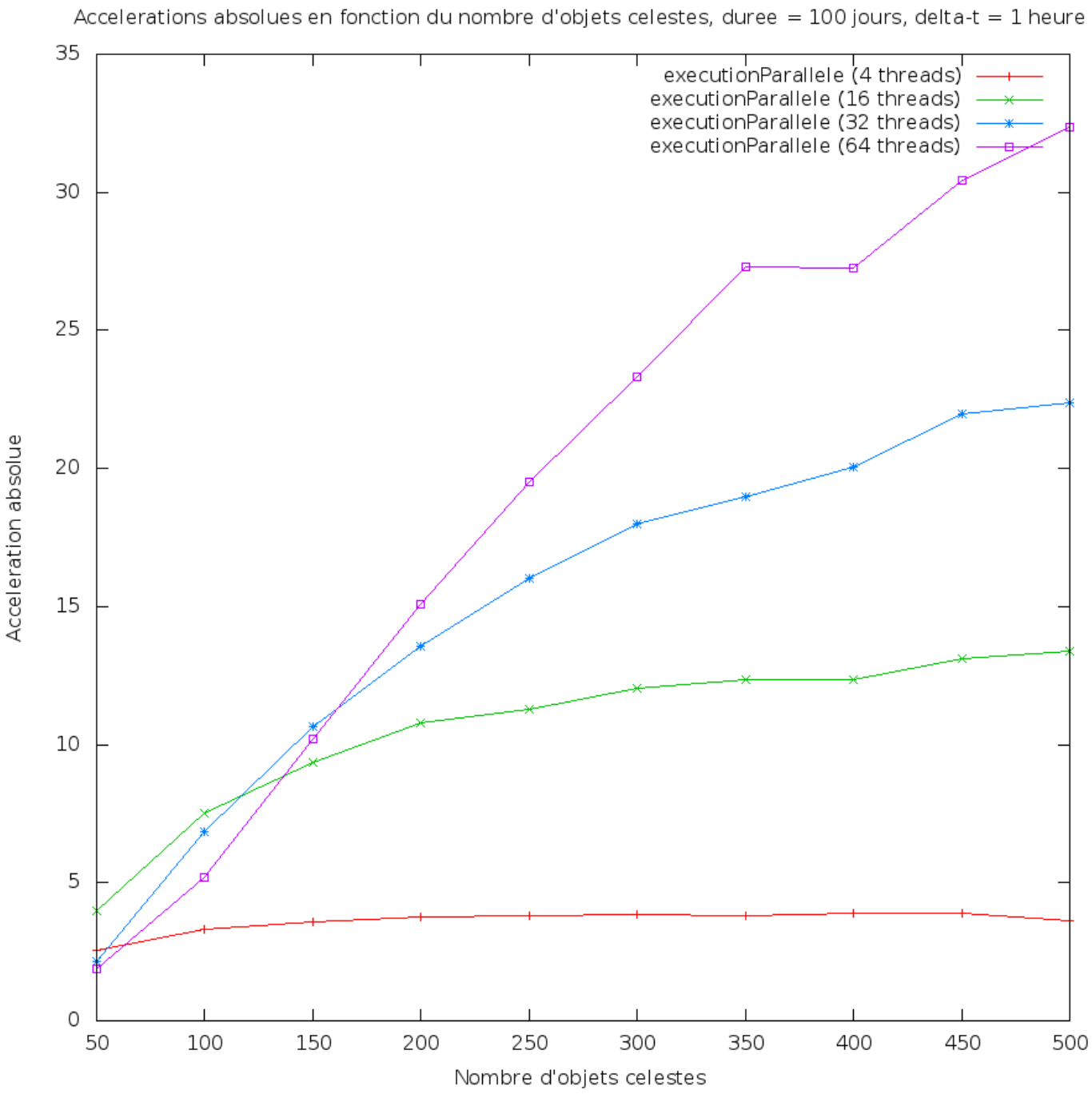
3.1 Mesures 1



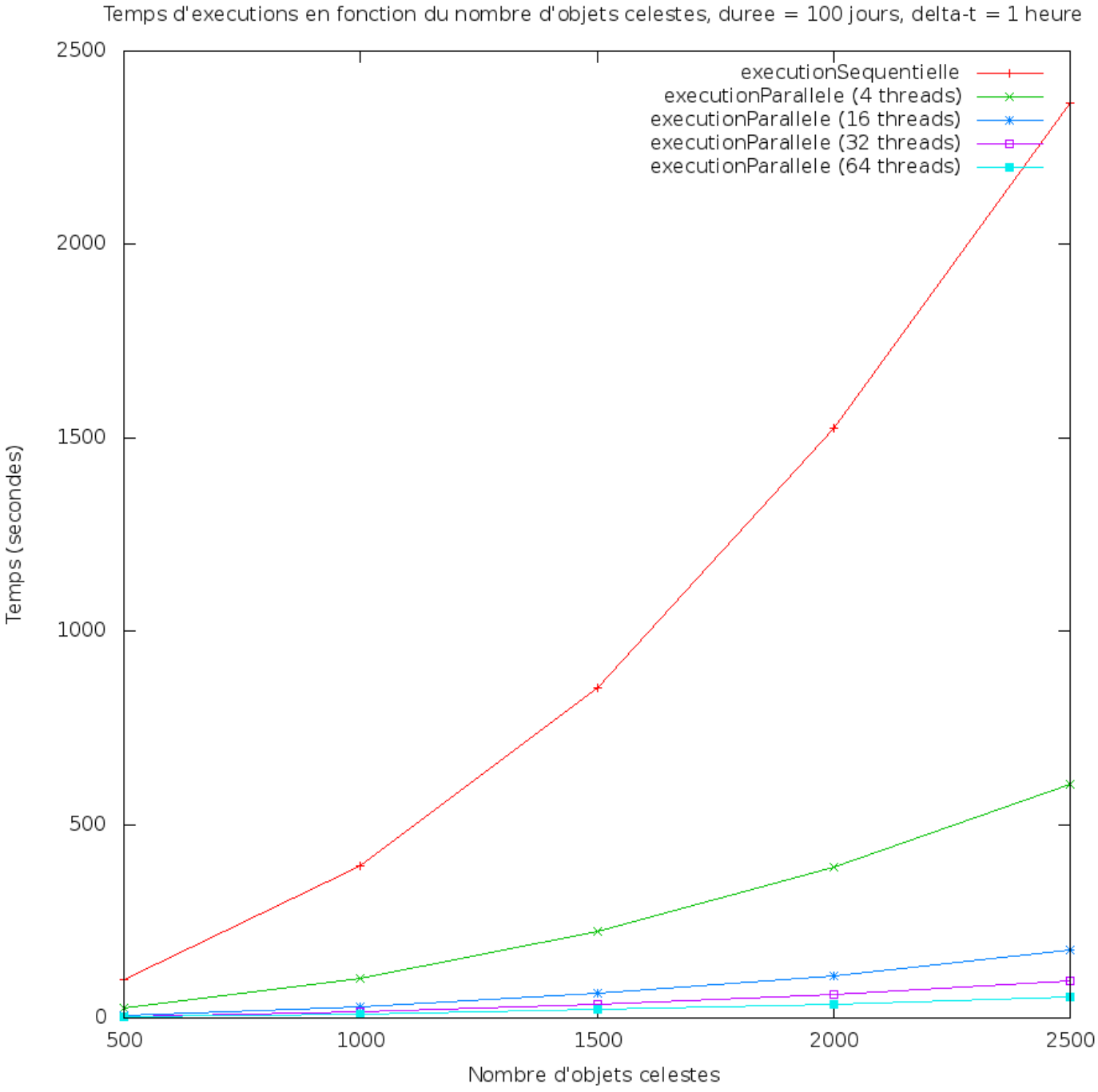

Accelerations absolues en fonction du nombre d'objets celestes, duree = 100 jours, delta-t = 1 heure
executionParallele (4 threads)
executionParallele (16 threads)
executionParallele (32 threads)
executionParallele (64 threads)
Acceleration absolue
Nombre d'objets celestes
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
0
10
20
30
40
50
60
70

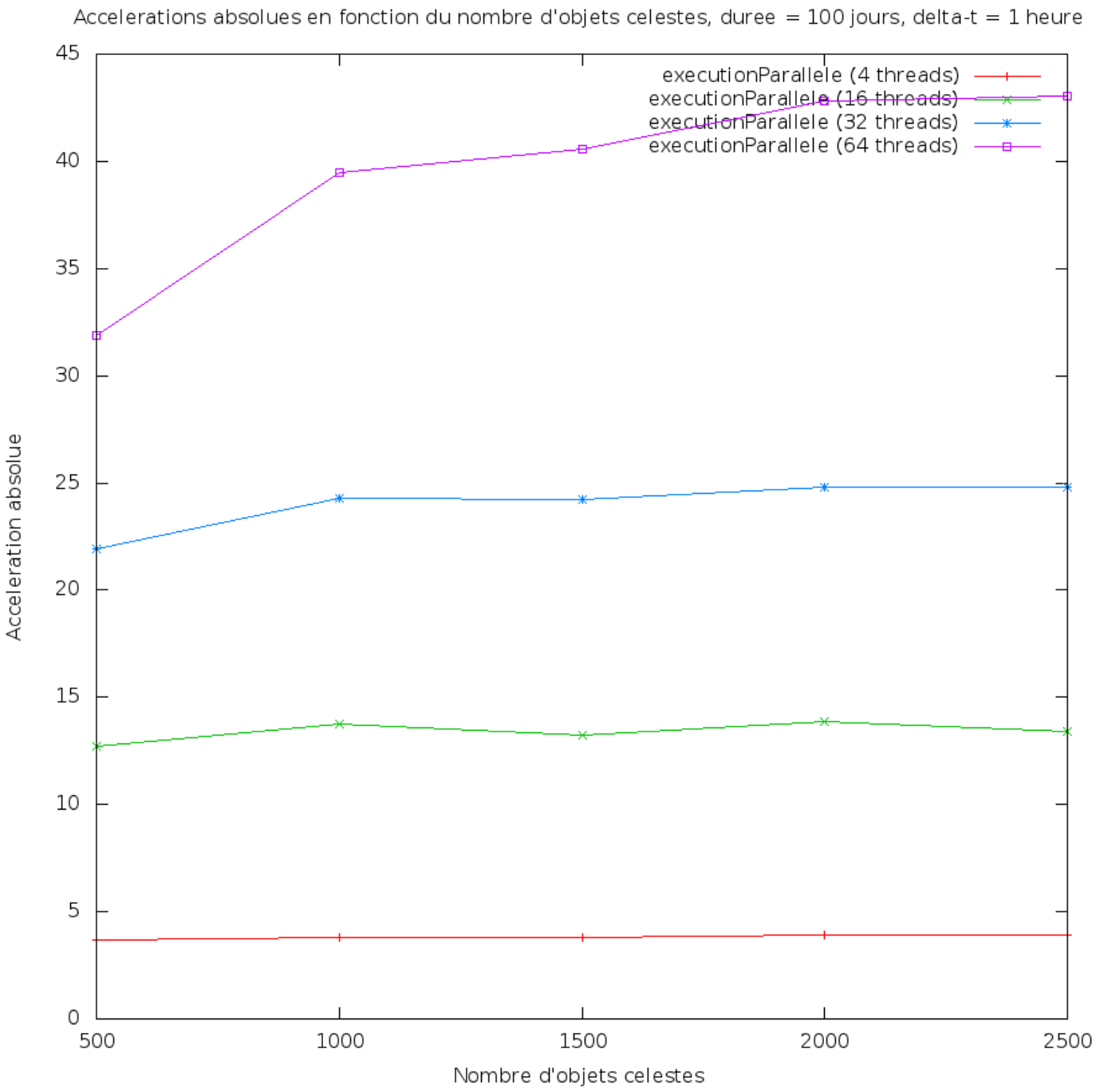
3.2 Mesures 2


Temps d'executions en fonction du nombre d'objets celestes, duree = 100 jours, delta-t = 1 heure
executionSequentielle
executionParallele (4 threads)
executionParallele (16 threads)
executionParallele (32 threads)
executionParallele (64 threads)
Temps (secondes)
Nombre d'objets celestes
0
20
40
60
80
100
120
50
100
150
200
250
300
350
400
450
500


Accelerations absolues en fonction du nombre d'objets celestes, duree = 100 jours, delta-t = 1 heure
executionParallele (4 threads)
executionParallele (16 threads)
executionParallele (32 threads)
executionParallele (64 threads)
Acceleration absolue
Nombre d'objets celestes
0
5
10
15
20
25
30
35
50
100
150
200
250
300
350
400
450
500

3.3 Mesures 3


Temps d'executions en fonction du nombre d'objets celestes, duree = 100 jours, delta-t = 1 heure
executionSequentielle
executionParallele (4 threads)
executionParallele (16 threads)
executionParallele (32 threads)
executionParallele (64 threads)
Temps (secondes)
Nombre d'objets celestes
0
500
1000
1500
2000
2500


Accelerations absolues en fonction du nombre d'objets celestes, duree = 100 jours, delta-t = 1 heure
executionParallele (4 threads)
executionParallele (16 threads)
executionParallele (32 threads)
executionParallele (64 threads)
Acceleration absolue
Nombre d'objets celestes
0
5
10
15
20
25
30
35
40
45
500
1000
1500
2000
2500

4 Stratégie de tests

4.1 Comparaison de tracées d'orbites

Une première stratégie pour comparer les résultats produits par la version sequentielle et la version parallèle revient à produire les tracées des déplacements d'objets célestes. Bien qu'approximatif, cette stratégie de test permet en même temps de valider la bonne implémentation des équations physiques, en démontrant que le programme produit des orbites stables en utilisant des données réelles.

Le programme **SimSystemeSolaire** produit via **src/SimSystemeSolaire.cpp** crée deux instances de Simulation. La première instance est initialisée en utilisant des données réelles du système solaire obtenues via l'API web HORIZON de la NASA (valeurs de masses, de vecteurs de position et de vecteurs de vélocité du Soleil et des 8 planètes du système solaire dans le système de coordonnées *Solar System Barycenter* (SSB) obtenus en date du 30 novembre 2017). La seconde instance est créée en utilisant le constructeur par copie avec la première simulation comme argument. Les deux simulations sont exécutées, l'une via la méthode séquentielle, l'autre via la méthode parallèle, pour une durée de temps simulée de 365 jours et un Δt de une heure. Ce programme produit les fichiers **data/simSystemeSolaireSeq.dat** et **data/simSystemeSolairePar.dat**, contenant les informations sur les positions successives des objets célestes du système solaire pour chaque itération de temps Δt. Ces fichiers peuvent être lus par le programme *Gnuplot* pour produire les graphes des traces du Soleil et des planètes (la commande **gnuplot src/gnuplotSimSystemeSolaire.gpi** permet de créer automatiquement ces graphiques aux emplacements **data/simSystemeSolaireSeq.png** et **data/simSystemeSolairePar.png** du moment que les fichiers **.dat** existent. À noter que ces graphes ne présentent qu'une projection 2D du système, selon le plan de l'orbite de la Terre (cf. système de coordonnées SSB)). Toutes ces étapes (compilation, exécution, production des graphes en utilisant *Gnuplot* et le script) peuvent être faites d'un seul coup en entrant la commande **make orbites**.

Ci-bas les traces de ces simulations. Dans un premier temps, on remarque que les équations Newtoniennes semblent avoir été respectées : tel qu'on s'y attend, seul Mercure, Vénus et la Terre auront eu le temps de compléter une révolution solaire en 365 jours. Dans un second temps, nous pouvons voir que les orbites des trois premières planètes semblent stables, signifiant que le niveau de granularité choisi ($\Delta t = 1$ heure) était suffisamment fin. Enfin, en comparant les deux graphes, nous voyons que les méthodes parallèle et séquentielle retournent des traces très similaires (peut-être même identique), mais cette observation reste plutôt approximative. La validation quantitative des résultats produits par ces deux méthodes sera le sujet de la prochain sous-section.

4.2 Validation sur les états des simulations à un temps donné

La deuxième stratégie de tests revient à comparer les données courantes d'une simulation exécutée via la méthode séquentielle aux données courantes d'une simulation créée avec les même données initiales et exécutée via la méthode parallèle.

Le programme **SimTests** spécifié par **src/SimTests.cpp** produit une instance de simulation en créant 100 objets célestes ayant des masses et des vecteurs de positions initiales aléatoires, et une seconde instance de simulation est créée via le constructeur par copie. Les simulations sont exécutées, l'une de manière séquentielle, l'autre de manière parallèle, en laissant rouler la simulation pour des durée de temps fictives de manière à ce que, à certains moments, les deux simulations aient roulé pour des valeurs de durées identiques ou différentes.

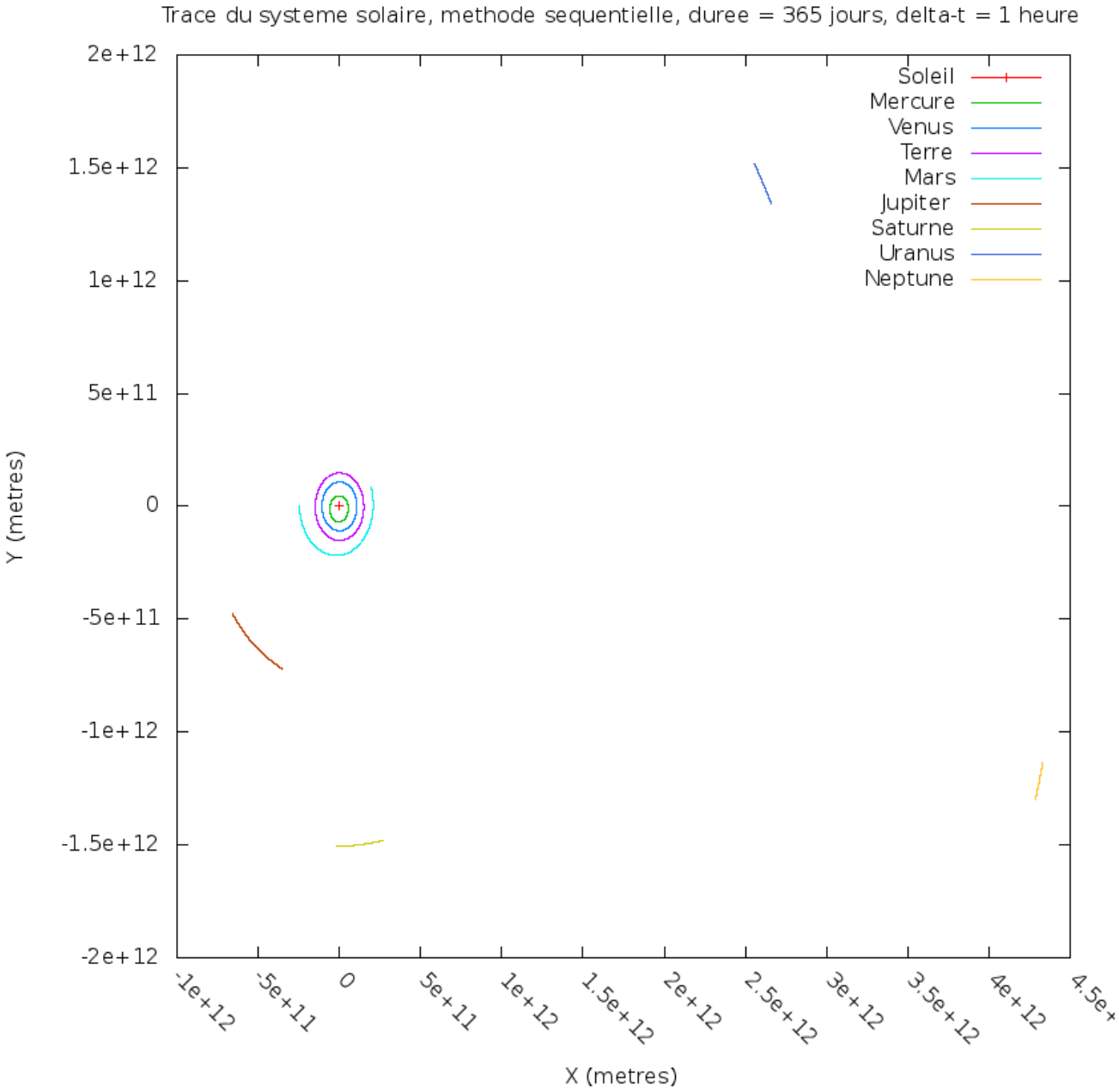
Le programme vérifie que :

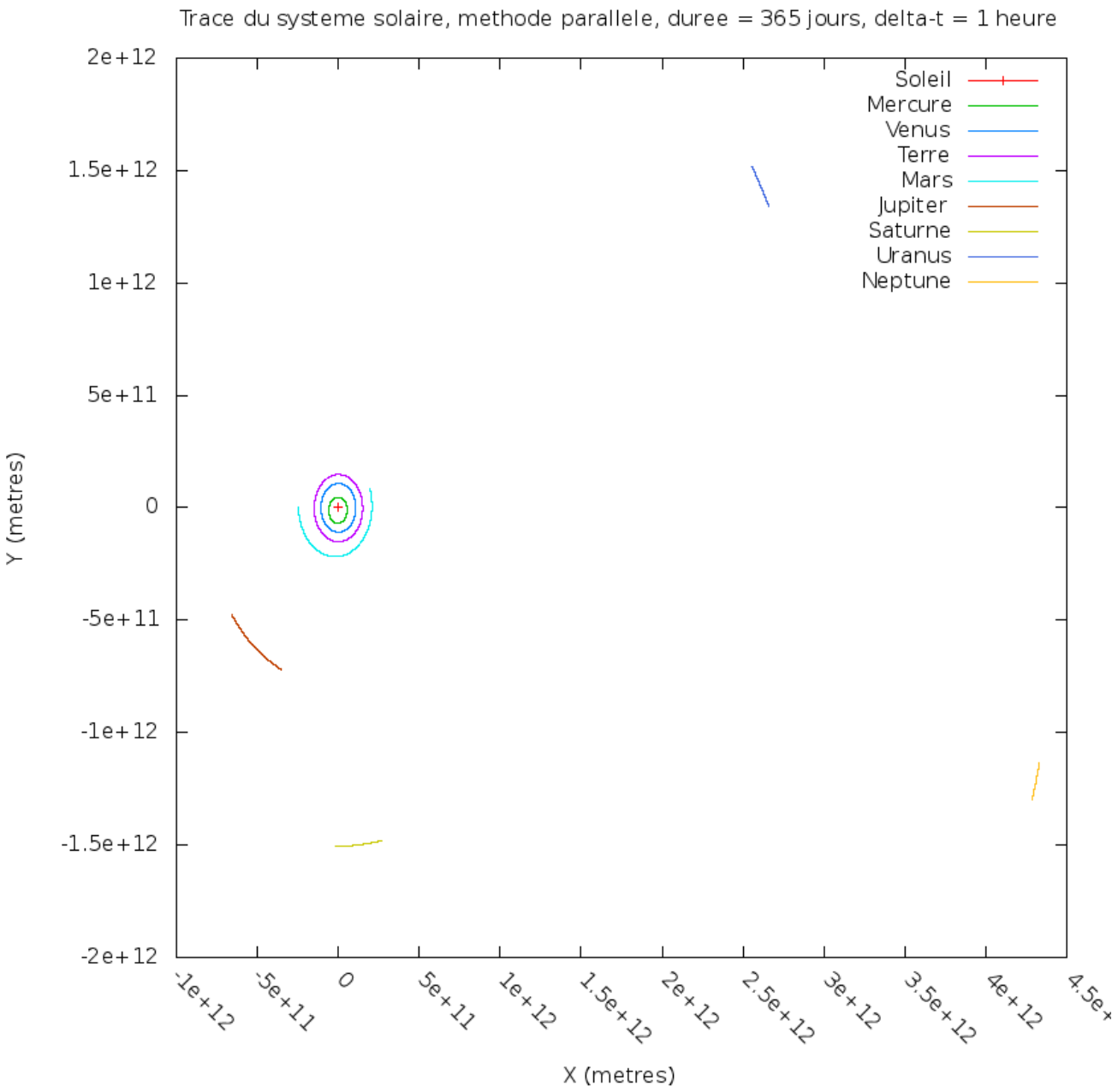
- si les simulations ont roulé pour une même valeur de temps fictive T, alors les objets célestes "frères" des deux simulations possèdent les mêmes valeurs de vecteur de position, de velocité et d'accélération.
- si les simulations ont roulé pour des valeurs de temps fictives T_{seq}, T_{par} différentes, alors les deux simulations possèdent des objets célestes qui sont dans des états différents. Bien que ce ne soit pas garanti à 100% (example, imaginer un système binaire. Ce système au temps T possède les même caractéristiques que ce même système au temps $T + n \cdot tempsRevolution$, où n est un entier), dans le cas d'un système composé de 100 objets, on suppose que cette probabilité soit suffisamment faible pour être ignorée.

Le programme affiche successivement sur la sortie standard les résultats de la série de test, en précisant à la toute fin si au moins un test a échoué ou si tous les tests passent. La compilation et l'exécution de ce plan de tests peut être fait en entrant la commande **make tests**.

À noter que puisque les comparaisons des états des simulations nécessitent de comparer des valeurs flottantes via l'opération d'égalité, le programme prévoit un certain seuil d'inexactitude. Soit n_1, $n_2 (n_1 = 0 \implies n_2 \neq 0)$ deux valeurs flottantes. Dans le contexte du projet, on considère que :

$$\frac{|n_1 - n_2|}{(n_1 + n_2)/2} < 0.0001 \implies n_1 = n_2$$




Trace du systeme solaire, methode parallele, duree = 365 jours, delta-t = 1 heure
Soleil
Mercure
Venus
Terre
Mars
Jupiter
Saturne
Uranus
Neptune
Y (metres)
X (metres)
2e+12
1.5e+12
1e+12
5e+11
0
-5e+11
-1e+12
-1.5e+12
-2e+12
-1e+12
-5e+11
0
5e+11
1e+12
1.5e+12
2e+12
2.5e+12
3e+12
3.5e+12
4e+12
4.5e+