Bureau d'Étude de Calcul Parallèle

Durée : 4 Heures

Vous devez travailler sur les machines des salles de TP.

Les 4 exercices sont indépendants.

Bien lire le sujet en entier pour déterminer dans quel ordre vous allez traiter les exercices.

Vous êtes autorisés à consulter la page Moodle du Cours de Calcul Parallèle et les documents qui y sont déposés ainsi que le site `https://rookiehpc.github.io/index.html` pour voir comment utiliser les routines MPI.

À la rigueur vous pouvez imprimer les slides du cours pour les consulter durant l'épreuve.

Sujet sur 8 pages

Mise en Place de l'environnement MPI

Nous fournissons dans le répertoire `Fournitures`, le fichier `init.sh` qui permet de positionner un certain nombre de choses (comme pour le premier TP MPI).

Il faut commencer par exécuter

source `init.sh`

Vérifier (en exécutant **echo** `$PATH`) que votre variable d'environnement `PATH` contient à son début le répertoire où sont installés les exécutables de SimGrid :

`/mnt/n7fs/ens/tp_guivarch/opt2021/simgrid-3.31/bin`

En tapant la commande `alias`, assurez-vous que la commande pour lancer les processus MPI `smpirun` a été complétée avec les paramètres donnant l'architecture cible. Ceci a été réalisé dans un souci de simplifier l'exécution des programmes parallèles.

ATTENTION : vous devez ré-exécuter cette initialisation à chaque ouverture d'un nouveau terminal si vous voulez avoir accès aux exécutables de SimGrid.

Instructions pour compléter les codes fournis

1. Les exercices sont répartis dans des répertoires différents
2. Pour les 3 premiers exercices, un fichier `Makefile` est présent pour compiler l'exécutable de l'exercice
3. Le fichier à compléter est indiqué dans la description de chaque exercice
4. Les endroits du code où insérer des appels à MPI (ou d'autres instructions) sont indiqués ainsi :

   // ...

5. A priori, vous n'avez pas besoin de déclarer de nouvelles variables
6. Les commentaires du code sont là pour vous guider

Instructions pour le Rendu Final

Vous avez dans le répertoire `Fournitures`, un fichier `Makefile` ; en exécutant la commande

make

Cela va "nettoyer" les différents répertoires de développement et créer une archive de nom

`Calcul_username_machine.tar`

Cette archive est à déposer avant 12h sous `Moodle` dans le dépôt correspondant à votre salle

Conseils pour le debuggage

1. Bien lire les *warnings* de la compilation, ils indiquent souvent un mauvais usage du C
2. les exercices 1, 2 et 3 peuvent se valider **étape par étape**,
3. des lignes de code permettant l'affichage des données après certaines étapes sont déjà écrites ; vous pouvez les dé-commenter et vérifier ainsi le bon fonctionnement de votre code, en particulier des communications,
4. rien ne vous empêche de prévoir d'autres affichages au besoin.

1 Communication en Anneau - Le retour

`Répertoire de développement :` **01_RingD**

Nous disposons de $2 * N$ noeuds MPI, que nous avons répartis dans 2 *sous-communicateurs MPI*, celui des noeuds pairs et celui des noeuds impairs. On souhaite faire transiter deux messages en anneau, un dans chaque sous-communicateur.

Pour les noeuds pairs, ce message tournera, à partir du noeud 0 du sous-communicateur dans le sens des aiguilles d'une montre (comme en TP). Pour les noeuds impairs, ce tour se fera dans le sens inverse des aiguilles d'une montre, toujours en partant du noeud 0 du sous-communicateur des noeuds impairs.

Pour les noeuds pairs, ce message est constitué d'une valeur qui sera multipliée par deux par chaque noeud ; pour les noeuds impairs, il est constitué d'une valeur qui, elle, sera divisée par deux par chaque noeud de la boucle.

— complétez le code du fichier `ringd.c` en suivant les instructions données en commentaires. Vous veillerez à faire les affichages nécessaires permettant de montrer le bon fonctionnement de votre code.

2 Calcul distribué d'une norme-A

`Répertoire de développement :` **02_normA**

Lorsque qu'une matrice A est symétrique définie positive, si on a x vecteur alors $\sqrt{x^T.A.x}$ est une norme, appelée norme-A.

On se propose de calculer en parallèle cette norme quand A est répartie sur N noeuds suivant un découpage par blocs de lignes (blocs de taille b). Chaque noeud possède la partie du vecteur x correspondante.

Avec la taille donnée en constante ($n = 12$) dans le sujet et le nombre de noeuds imposés ($size = 4$), le nombre de lignes affectées à chaque noeud est $b = 3$.

Cette distribution est illustrée par `Figure 1`.

On vous propose de décomposer ce calcul en deux étapes

1. calcul de $y = A.x$
2. calcul de $n = x^T.y$ suivi d'une racine carrée pour obtenir la norme

Pour la première étape chaque noeud doit disposer du vecteur x complet (Figure 2 pour le noeud 1). Vous allez donc coder une communication collective pour effectuer ce rassemblement.

Une fois que chaque noeud a calculé son y local, il pourra effectuer le produit scalaire de ce y local avec son x local.

Ne restera plus qu'à utiliser une communication collective pour obtenir sur tous les noeuds le produit scalaire global puis la norme-A.

— complétez le code du fichier `MultAv.c` en suivant les instructions données en commentaires [1].

1. à noter que dans le code, les matrices A pouvant être utilisées ne sont pas symétriques définies positives

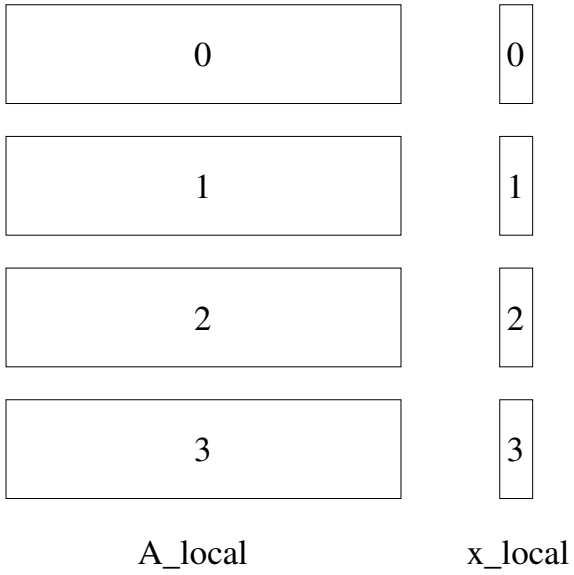

Figure 1 – distribution de A et x

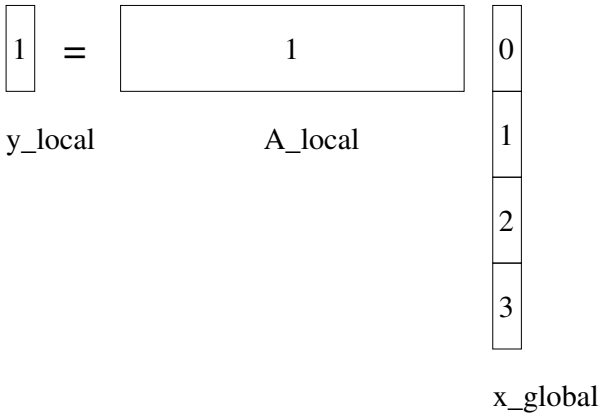

Figure 2 – calcul de la première étape sur le noeud 1

3 Calcul distribué du nombre de valeurs d'un vecteur au dessus de sa moyenne

`Répertoire de développement :` **03_overmean**

On dispose de N processus MPI et d'un vecteur de taille $3 * N$ (taille pour que la répartition du vecteur sur chaque processus soit régulière).

Seul le processus 0 connaît le vecteur complet.

On souhaite calculer le nombre de valeurs du vecteur global qui sont supérieures ou égales à la moyenne des valeurs du vecteur global.

Pour cela il faut (retrouver les étapes dans le code fourni) :

1. distribuer à chaque processus un bout du vecteur global
2. chaque processus calcule la somme de ces élements
3. la somme globale est calculée par réduction sur le processus 0
4. le processus 0 calcule la moyenne du vecteur global
5. le processus 0 partage cette moyenne aux autres processus
6. chaque processus calcule le nombre de valeurs de son bout de vecteur au dessus de la moyenne
7. le processus 0 récupère la somme totale de ces nombres

Complétez le code fourni pour mener à bien ce calcul et vérifiez bien que ce calcul est correct (petit nombre de processus).

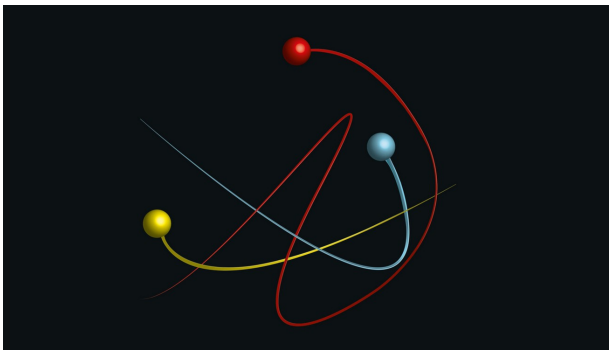

Figure 3 – Illustration (source : Pour la Science)

4 Problème des N-corps : évolution d'un ensemble de corps qui interagissent les uns avec les autres

`Répertoire de réflexion:` **`04_n-corps`**

Pour cet exercice

1. vous écrirez les algorithmes proposés et les réponses aux questions dans le fichier `n-corps.md` du répertoire de réflexion,
2. vous dessinerez les répartitions des données (en prenant par exemple une parallélisation avec 4 processus), leurs transferts à chaque étape des deux algorithmes demandés sur la feuille fournie.

Exemple en astrophysique : le déplacement des planètes d'un système solaire en tenant compte des forces de gravitation.

Pour chaque paire de corps (planète, étoile) (i, j), i subit de la part de j une force $\vec{f_{ij}}$

$$\vec{f_{ij}} = G.\frac{m_i.m_j(\vec{x_i} - \vec{x_j})}{\|\vec{x_i} - \vec{x_j}\|^3}$$

où G est une constante, m_i, m_j sont les masses respectives des deux corps, et $\vec{x_i}$, $\vec{x_j}$ leurs vecteurs position.

La solution de base consiste à chaque itération de temps à

1. pour tous les corps, calculer les forces subies de la part de chacun des autres, et faire la somme de toutes ces forces,
2. pour tous les corps, calculer, à partir des forces subies, les nouvelles positions, vitesse et accélération de ce corps.

Le programme séquentiel est le suivant (en pseudo-code) :

Algorithm 1 Version initiale

```
1:  for t in 1, NB_STEPS do
2:    for i in 1,N do
3:      force[i] = 0
4:      for j in 1, N do
5:        force[i] = force[i] + interaction(data[i], data[j])
6:      end for
7:    end for
8:    for i in 1, N do
9:      data[i] = update(data[i], force[i])
10:   end for
11: end for
```

où N est le nombre de corps, **data[i]** est une structure de données qui comprend la position, la vitesse et l'accélération du corps i, **interaction(data[i],data[j])** est un appel de fonction qui renvoie la force appliquée par le corps j au corps i et **update(data[i],force[i])** est un appel de fonction qui calcule les nouvelles position, vitesse et accélération du corps i en fonction de la force qu'il subit.

On suppose de plus que seul le processus de rang 0 connait initialement data et que le nombre de processus divise le nombre de corps.

4.1 première version

Question 1 : Déterminer quels calculs peuvent être parallélisés, quelles sont les données globales, les données locales, comment les répartir, quelles communications à mettre en place, à quel moment ;

- dessinez les répartitions des données (par exemple sur 4 processus) et leurs transferts à chaque étape sur la **feuille fournie** [2],
- écrire une version parallèle de ce code (dans le fichier `n-corps.md`).

Notez bien : vous écrirez en pseudo-code les instructions nécessaires pour répartir le travail entre les processus ainsi que les communications nécessaires pour effectuer ce travail correctement, sans écrire de "vrais" appels à MPI mais en indiquant quel type de communication utiliser avec quelle données, de qui vers qui.

2 dernières questions sur la page suivante

2. rendre copie vide avec votre nom si vous ne répondez pas à cette question

4.2 seconde version

L'algorithme séquentiel ne tient pas compte du fait que les forces sont opposées :

$$\vec{f}_{ji} = -\vec{f}_{ij}.$$

Il n'est donc pas nécessaire de calculer séparément l'intensité de ces deux forces.

Pour prendre en compte cette observation, on pourrait modifier l'algorithme séquentiel de calcul des forces :

Algorithm 2 Version forces opposées

```
1: for t in 1, NB_STEPS do
2:    for i in 1, N do
3:       force[i] = 0
4:    end for
5:    for i in 1, N do
6:       for j in 1, i-1 do
7:          f = interaction(data[i], data[j])
8:          force[i] = force[i] + f
9:          force[j] = force[j] - f
10:      end for
11:   end for
12:   for i in 1, N do
13:      data[i] = update(data[i], force[i])
14:   end for
15: end for
```

Question 2 : Proposer une version parallèle de ce code (même instructions de rendu que pour la question 1).

Question 3 : Quels sont les inconvénients de cette version ? Proposer une solution pour les atténuer. (à écrire dans le fichier `n-corps.md`).