Bureau d'Étude de Calcul Parallèle

Durée : 3 Heures

Documents autorisés : planches du Cours + Corrections du TP1

Vous devez travailler sur les machines des salles de TP.

Vous êtes autorisés à consulter le site `https://rookiehpc.org/index.html` pour voir comment utiliser les routines MPI ainsi que la page `Moodle` du cours (accès aux slides et aux solutions des TP)

Les 3 exercices sont indépendants et peuvent être traiter dans n'importe quel ordre.

Bien lire le sujet en entier pour déterminer par quel exercice commencer.

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 dans le terminal où vous allez compiler et exécuter les programmes :

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/opt2023/simgrid-3.32/bin
```

En exécutant la commande `alias`, assurez-vous aussi que la commande pour lancer les processus MPI `Smpirun` a bien comme paramètres la plateforme et l'architecture. 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.

Exercices 1 et 2 : Instructions

1. Aux deux premiers exercices correspondent deux répertoires différents.
2. Pour ces deux exercices qui demandent un codage, un fichier `Makefile` est présent dans le répertoire correspondant pour compiler l'exécutable (2 exécutables pour l'exercice 2).
3. Le fichier à compléter est indiqué dans la description de chaque exercice.
4. **Pour l'exercice 2, vous devez répondre à des questions sur la "feuille à petits carreaux" fournie.**

Exercice 3 : vous devez répondre aux questions de cet exercice sur la "feuille à petits carreaux"

Instruction pour le Rendu

Vous avez dans le répertoire `Fournitures`, un fichier `Makefile` ; en exécutant la commande (**au moment de rendre votre travail**)

make

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

`Calcul_username_machine.tar`

Cette archive est à déposer à la fin de la scéance sous `Moodle` dans le dépôt de votre groupe (salle).

Vous devez rendre la ou les copie(s) à l'enseignant qui surveille votre salle.

1 Implantation du `MPI_Scan` avec des communications point à point (5 points)

`Répertoire de développement :` **01_MPI_Scan**

`Fichier à modifier :` **mpi_scan.c**

Nous disposons de N processus MPI, et on souhaite simuler la communication collective `MPI_Scan` en l'implantant avec des communications point à point.

L'interface de cette fonction est la suivante :

```
int MPI_Scan(void* sendbuf, void* recvbuf,
             int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm);
```

Le `MPI_Scan` effectue une réduction partielle de données distribuée sur un groupe de processus (voir Figure 1).

Après exécution de la commande, le buffer de réception du processus i contiendra la réduction des données des buffers d'envoi des processus de rang inférieurs ou égaux à i. Les opérations de réductions sont les opérations habituelles (somme, produit, max, min, ...).

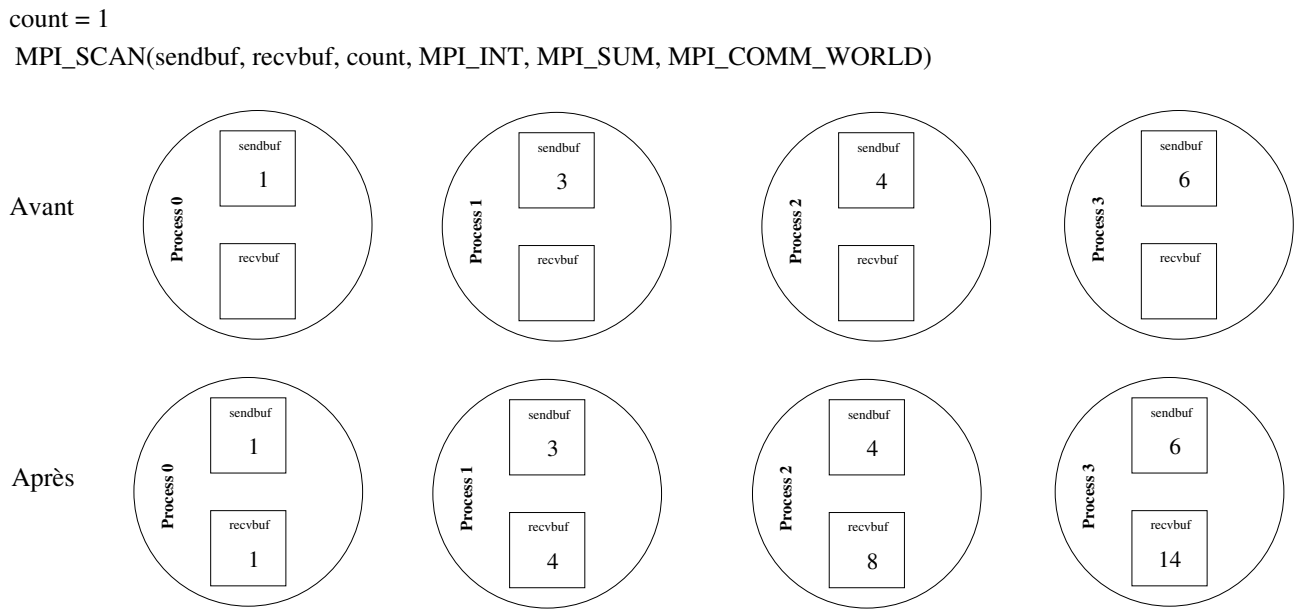

Figure 1 – Example de MPI_Scan avec 4 processus

Le processus de rang 0 aura comme résultat ses données et le processus de rang le plus élevé la réduction complète telle que celle obtenue avec un `MPI_Reduce`.

— complétez le code du fichier `mpi_scan.c` en suivant les instructions données en commentaires. Le code fourni appelle le `MPI_Scan` classique pour que vous ayez une base de comparaison.

2 Étirement de contraste d'une image (8 points)

`Répertoire de développement:` **02_Contraste**

`Fichier à modifier :` **stretching_par.c**

Cet exercice consiste à paralléliser un code séquentiel qui effectue un traitement d'image (étirement de contraste – *contrast stretching*).

(a) Image originale

(b) Résultat du stretching

Figure 2 – Exemple d'exécution avec 10 processus

Vous avez la possibilité de décrire sur papier votre parallélisation (voir fin d'exercice).

Le code séquentiel est donné par le fichier `stretching.c`.

Les étapes sont :

1. lecture de la taille de l'image
2. allocation de la mémoire pour l'image
3. lecture de l'image et transformation en niveau de grix
4. recherche des valeurs `min` et `max` de l'image
5. calcul à partir des deux valeurs précédentes d'un scalaire alpha utilisé pour le traitement
6. traitement de l'image
7. affichage et sauvegarde de l'image résultat

Le fichier **stretching_par.c** initial, seul fichier à modifier, est une recopie du fichier `stretching.c`.

Vous devez le transformer pour en faire un code parallèle MPI en
- identifiant les calculs parallélisables,
- les allocations mémoire à rajouter,
- les communications à mettre en place ; ce sont exclusivement des **communications collectives**,

tout en respectant les consignes en commentaire, en particulier les actions dévolues au processus 0.

Le fichier `Makefile` permet de générer deux exécutables : **stretching** et **stretching_par**.

Pour appeler le code séquentiel

```
./stretching
```

et pour le code parallèle

```
Smpirun -np 4 ./stretching_par
```

(cela fonctionne, pour l'image fournie, avec un nombre pair de processus inférieur ou égal à 10).

Question (à repondre sur la copie "petits carreaux")

Est-ce que les deux calculs parallélisables pourraient efficacement être déportés sur un GPU ? Justifier votre réponse.

Alternative au développement C

Si vous avez des difficultés à mener à bien la parallélisation du code, vous pouvez, sur la copie, en pseudo-code, expliciter votre parallélisation.

3 Matrices stochastiques et doublement stochastiques (7 points)

Une matrice A carrée $n \times n$ est **stochastique** (aussi appelée matrice de **Markov**) si elle vérifie les deux propriétés suivantes :

1. chacune de ses composantes est réelle positive $a_{ij} >= 0$.
2. pour chaque ligne i, la somme de ses éléments est égale à 1 :

$$\forall i \in \{1, \ldots, n\}, \sum_{j=1}^{n} a_{ij} = 1.$$

Cela correspond, en théorie des probabilités, à la matrice de transition d'une chaîne de Markov.

Une matrice A carrée $n \times n$ est **bistochastique ou doublement stochastique** si elle vérifie les deux propriétés précédentes ainsi qu'une troisième :

3. pour chaque colonne j, la somme de ses éléments est égale à 1 :

$$\forall j \in \{1, \ldots, n\}, \sum_{i=1}^{n} a_{ij} = 1.$$

Vous trouverez sur la page suivante un pseudo-code séquentiel permettant de vérifier qu'une matrice A est **bistochastique** en supposant que

1. la vérification de la propriété 1 (toutes les composantes de A sont positives) est faite au niveau de la lecture de A,
2. la matrice A est stockée ligne par ligne

En supposant que l'on a un nombre de processus qui divise n, et en sachant que la matrice A va être distribuée par blocs de lignes sur les processus, proposez une parallélisation, utilisant des **communications collectives**, de la vérification des propriétés 2 et 3 (on laisse le processeur 0 lire la matrice et vérifier la propriété 1).

1. Dessinez le schéma de parallélisation (gestion des données, communications et calculs) de la vérification de la propriété 2 (sur les lignes),
2. Dessinez le schéma de parallélisation (gestion des données, communications et calculs) de la vérification de la propriété 3 (sur les colonnes),
3. Écrivez le pseudo-code correspondant à votre proposition de parallélisation.
4. Est-ce que l'algorithme parallèle sera toujours plus rapide que l'algorithme séquentiel ? Donnez des exemples de matrices illustrant l'une et l'autre des situations.

```
// n taille de la matrice
int n = get_matrix_size(filename);

// allocation de la mémoire pour contenir la matrice
float *A = allocate_matrix(n, n);

// lecture de la matrice et vérification de la propriété 1
bool prop1_ok = read_matrix(filename, A);
// on arrête si la propriété 1 n'est pas vérifiée
if(!prop1_ok){
  printf("la matrice ne contient pas que des composantes positives\n");
  return 1;
}

// vérification de la propriété 2
bool prop2_ok = true;
int i = 0;
// verification_somme_ligne(B, m, n, l) renvoie la somme des éléments
// de la ligne l de la matrice B de taille m * n)
while( prop2_ok && i < n ){
   prop2_ok = (verification_somme_ligne(A, n, n, i) == 1.0);
   i++;
}
// on arrête si la propriété 2 n'est pas vérifiée
if(!prop2_ok){
  printf("au moins une des lignes de la matrice n'a pas une somme de 1\n");
  return 2;
}

// vérification de la propriété 3
bool prop3_ok = true;
int j = 0;
// verification_somme_colonne(B, m, n, l) renvoie la somme des éléments
// de la colonne l de la matrice B de taille m * n)
while( prop3_ok && j < n ){
   prop3_ok = (verification_somme_colonne(A, n, n, j) == 1.0);
   j++;
}
// on arrête si la propriété 3 n'est pas vérifiée
if(!prop3_ok){
  printf("au moins une des colonnes de la matrice n'a pas une somme de 1\n");
  return 3;
}

printf("la matrice est bistochastique\n");
return 0;
```