TP 2 : Aller plus loin avec MPI

1 Implantation du `MPI_Scan` avec des communications point à point

`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, ...).

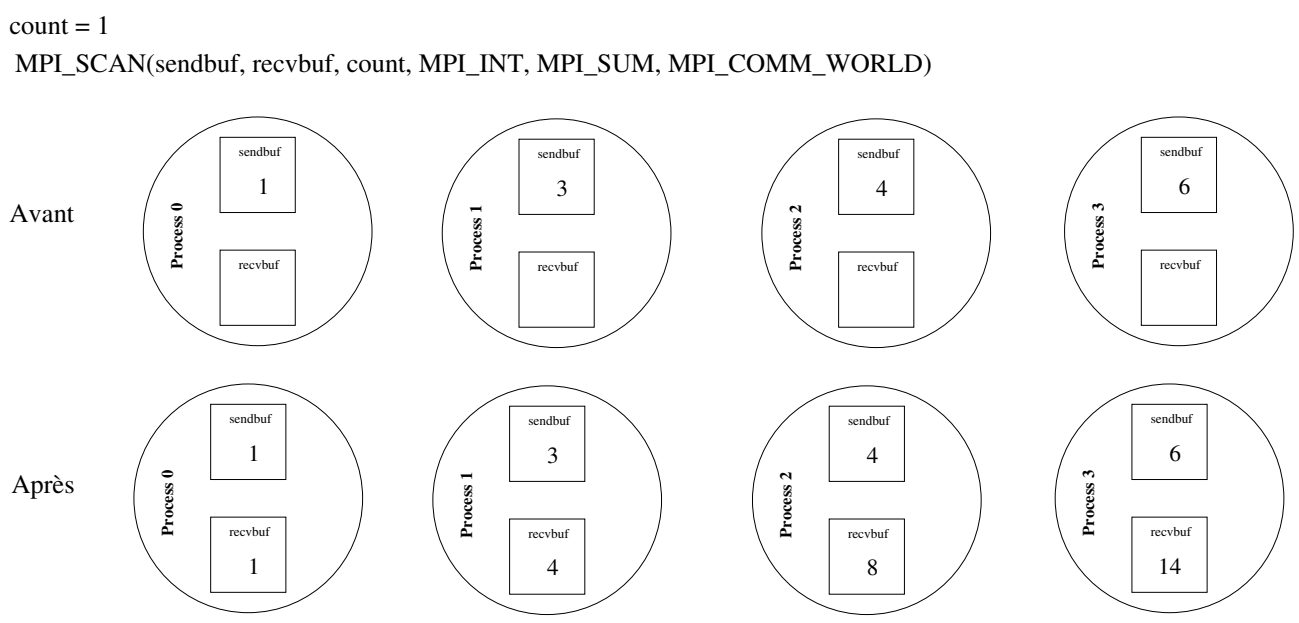


Figure 1 – Example de MPI_Scan avec 4 processus

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

`Répertoire de développement :` **02_Overmean**

`Fichier à modifier :` **overmean.c**

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).

3 Produit scalaire avec distribution de données non-contigues

`Répertoire de développement:` **03_Scatter_stride**

`Fichier à compléter :` **scatter_stride.c**

On souhaite effectuer en parallèle le produit scalaire entre deux vecteurs X et Y qui se trouvent sur le processus 0.

Mais contrairement aux exercices des TP, les données ne vont pas être distribuées de manière contiguë aux autres processus MPI.

On veut en effet effectuer une distribution *Round-Robin*, ie la première donnée est confiée au premier processus, la deuxième au deuxième processus, etc et ceci de manière circulaire.

Cette distribution est illustrée par la Figure 2 où 12 données sont distribuées entre 4 processus, chaque couleur correspond à un processus (bleu processus 0, vert processus 1, ...).

Figure 2 – Distribution Round-Robin des données

On se propose de distribuer les deux vecteurs de taille n de deux façons différentes sur les N processus :

1. le vecteur X en envoyant les données une à une (n communications point à point)
2. le vecteur Y en effectuant $\frac{n}{N}$ communications collective de *scatter*

Voici quelques questions (à répondre sur le sujet) pouvant vous guider vers les deux solutions.

Travail à effectuer :

1. pour la distribution du vecteur X : étant la donnée i vers quel processus doit-elle être envoyée ? Est-ce que les données du processus 0 doivent être traitées différemment ?
2. pour la distribution du vecteur Y : quels sont les données distribuées par chaque *scatter*, où commencent ces données ?
3. question subsidiaire : voyez-vous d'autres façons d'organiser cette distribution non contiguë ?

Travail à effectuer :

— complétez le code du fichier `scatter_stride.c` en suivant les instructions données en commentaires.
On n'oubliera pas de faire la dernière communication collective permettant d'obtenir le produit scalaire global sur le processus 0.

4 Calcul de l'entier inférieur à n possédant le plus de diviseurs

`Répertoire de développement:` **04_Diviseurs**, fichiers : `diviseurs_par.c` puis `diviseurs_par2.c` à créer.

Soit un entier n, quel est le plus petit entier inférieur ou égal à n qui possède le plus de diviseurs (1 exclu [1], n lui-même compté) ?

Par exemple si $n = 20$, les entiers 12, 18 et 20 sont les entiers ayant le plus de diviseurs (5) et le résultat doit être 12 (le plus petit).

Le fichier `diviseurs.c` contient le code séquentiel qui effectue ce calcul.

La première parallélisation que vous allez effectuer consiste à répartir l'espace de travail en sous-espaces contigus. Dans un souci de simplification, on supposera que n est divisible par le nombre de processus. Ainsi si nous avons 4 processus et que n vaut 100, le processus 0 s'occupera de faire le calcul sur les entiers de 1 à 25, le processus 1 sur les entiers de 26 à 50, le processus 2 sur les entiers de 51 à 75 et enfin le processus 3 sur les entiers de 76 à 100.

Pour cela on vous fournit un fichier `diviseurs_par.c`, où la mécanique MPI (initialisation, récupération du rang et du nombre de processus, finalisation) a été insérée et où tous les processus font le même

1. sauf pour $n = 1$

calcul (de 1 à n) sans communication.

À vous de répartir le travail et de collecter les résultats pour fournir la solution.

Commencez par répondre à cette question pour vous aider à réfléchir à la parallélisation :

Travail à effectuer :

1. Expliquer pourquoi un `reduce` "simple" ne permettra pas une collecte efficace des résultats locaux et qu'il vaut mieux implémenter cette collecte sans utiliser de `reduce` ?

Travail à effectuer :

— complétez le code du fichier `diviseurs_par.c` pour paralléliser le calcul en découpant le travail comme indiqué.

Cette version parallèle affiche le temps de chaque processus, ainsi que le temps du processus 0 qui, normalement, va collecter les résultats locaux et calculer le résultat global.

Travail à effectuer :

2. que constatez-vous au niveau du temps de travail de chaque processus ? (n'hésitez pas à considérer un n grand)
3. expliquez ces temps ?
4. proposez une autre répartition du travail pour améliorer l'efficacité de la parallélisation.

Travail à effectuer :

— implémenter l'algorithme proposant une autre répartition dans un fichier `diviseurs_par2.c` en copiant votre fichier `diviseurs_par.c` (**make diviseurs_par2** pour compiler).
— vous devriez constater un équilibrage du temps de calcul de chaque processeur.
— vous pouvez maintenant réfléchir à un moyen d'utiliser un `reduce` (pas simple).

5 Étirement de contraste d'une image

`Répertoire de développement:` **05_Contraste**

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

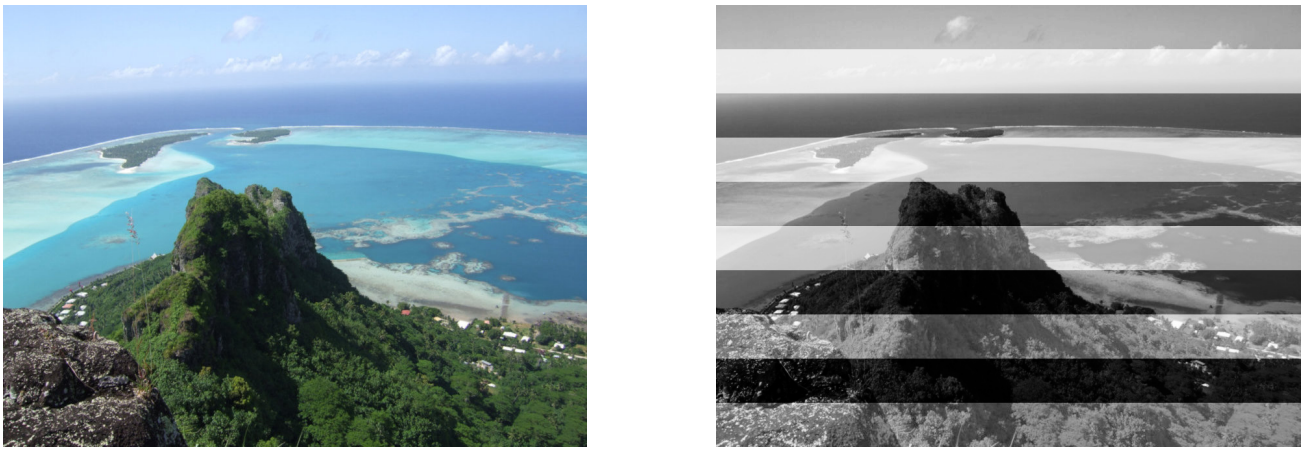
Cet exercice consiste à paralléliser un code séquentiel qui effectue un traitement d'image (étirement de contraste – *contrast stretching*).

Il vous faudra peut-être installer le package `libopencv-dev` pour pouvoir compiler le code.

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



(a) Image originale (b) Résultat du stretching

Figure 3 – Exemple d'exécution avec 10 processus

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).