Bureau d'Étude de Calcul Parallèle

Durée : 4 Heures

Documents autorisés : planches du Cours

Vous devez travailler sur les machines des salles de TP.

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.

Sujet sur 8 pages

Les exercices sont indépendants.

L'exercice 3 est un exercice de réflexion.

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

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/opt2023/simgrid-3.32/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 2 premiers exercices qui demandent un codage, un fichier `Makefile` est présent dans le répertoire correspondant pour compiler l'exécutable de l'exercice.
3. Le fichier à compléter est indiqué dans la description de chaque exercice.
4. **Pour tous les exercices, vous devez répondre à des questions sur une feuille "petits carreaux" distribuée par votre enseignant.**
5. **Pour le premier exercice**, les endroits du code où insérer des appels à MPI (ou d'autres instructions) sont indiqués ainsi :

```
// ...
// TODO
// ...
```

6. **Pour le premier exercice**, vous n'avez, a priori, pas besoin de déclarer de nouvelles variables. Regardez donc bien les variables déclarées.
7. Les commentaires du code sont là pour vous guider.

Instruction pour le Rendu

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 à la fin de la scéance sous `Moodle` dans le dépôt de cette session 1

Conseils pour le debuggage

1. Bien lire les *warnings* de la compilation, ils indiquent souvent un mauvais usage du C
2. les exercices 1 et 2 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.

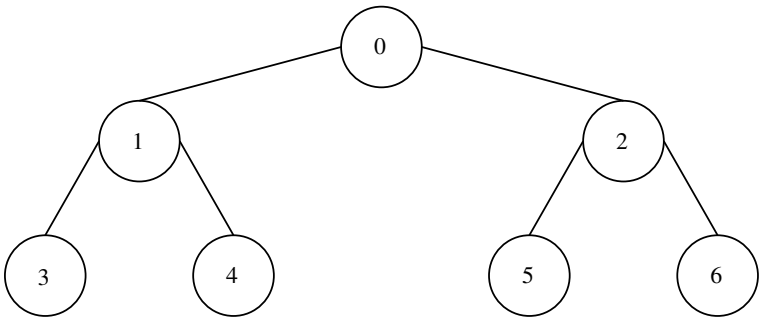

Figure 1 – Arbre binaire de profondeur 3

1 Communication le long d'un arbre binaire complet

`Répertoire de développement:` **01_BinTree_leaves**, fichier `bintree_leaves.c`.

Nous considérons une configuration logique où les noeuds MPI sont connectés par un réseau représenté par un arbre binaire complet de profondeur l.

Cet arbre dispose donc de $2^n - 1$ noeuds (voir Figure 1 pour un arbre de profondeur 3).

On souhaite faire transiter des messages des feuilles de l'arbre vers sa racine en suivant les liens entre les noeuds de l'arbre.

La difficulté pour généraliser les tranferts de message le long d'un arbre binaire (depuis les feuilles vers la racine, mais aussi de la racine vers les feuilles) est que chaque noeud sache qui est son parent et qui sont ses enfants.

Une façon de s'y retrouver est de considérer la numérotation des noeuds sous une forme binaire (**attention, pour que cela fonctionne, il faut numéroter les noeuds à partir de 1** et pas de 0 comme le sont les processus MPI).

Dans notre exemple la racine est donc le noeud 1 = "1" en binaire.

- Comment trouver le numéro de ses enfants ? À partir d'un noeud numéroté "X" en binaire, pour obtenir son enfant gauche (resp. droit), il suffit de rajouter un '0' (resp. '1') à droite (en poids faible) de son numéro : l'enfant gauche de "X" (resp. droit) a donc comme numéro "X0" (resp. "X1").
- Comment trouver le numéro de son parent : il suffit d'enlever le bit de poids faible : les noeuds "Y0" et "Y1" ont pour parent "Y".

Bien entendu, le noeud racine n'a pas de parent et les noeuds feuilles n'ont pas d'enfants.

La Figure 2 illustre cette numérotation.

Attention, il faut tout décaler de 1 pour retrouver la numérotation des processus MPI.

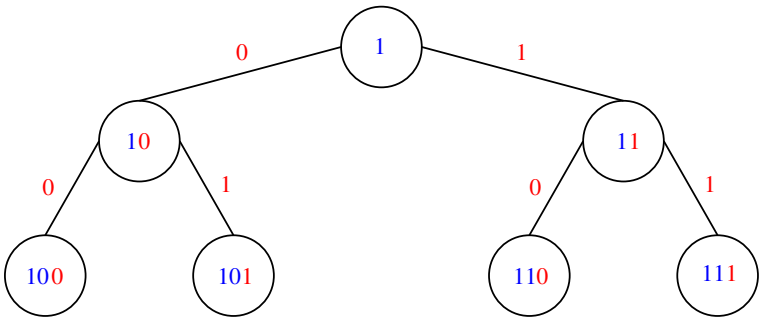

Figure 2 – Arbre binaire de profondeur 3 avec numérotation binaire

Pour vous aider à mener à bien l'implémentation des transferts, répondez aux questions suivantes sur la feuille "petits carreaux" distribuée :

Travail à effectuer :

1. combien d'ensembles de noeuds avez-vous à considérer ? (les noeuds d'un même ensemble effectuent le même ensemble d'opérations de communication)
2. comment les distinguer ?
3. quels sont les communications effectuées par chaque ensemble ?

Travail à effectuer :

— complétez le code du fichier `bintree_leaves.c` en suivant les instructions données en commentaires.

Point technique C :

En C les décalages d'un bit vers la gauche (resp. vers la droite) se sont avec l'opérateur `<<` (resp. `>>`)

exemples :

```
v = 3 << 1;       // v recoit 6
v = (3 << 1) + 1; // v recoit 7
v = 7 >> 1;       // v recoit 3
```

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

`Répertoire de développement:` **02_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 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 sur la feuille "petits carreaux" distribuée.

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.

Répondez aux questions suivantes sur la feuille "petits carreaux" distribuée.

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.

1. sauf pour $n = 1$

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.

3 Équilibrage de la luminance d'une image par la méthode des histogrammes.

Ce sujet s'inspire d'un exercice de Georges Da Costa (IRIT).

`Répertoire de réflexion:` **03_Equilibrage**, fichier `equilibrage_par.c`

Soit le code séquentiel (voir dernière page, aussi donné dans le fichier `equilibrage.c` du répertoire de réflexion) qui effectue un travail d'équilibrage (equalizer) de la luminance d'une image par la méthode des histogrammes.

On désire paralléliser ce traitement sur un nombre quelconque de processus [2]On continue à faire la simplification que la taille des objets manipulés sera divisible par ce nombre de processus.)

1. Faire le schéma d'une exécution parallèle avec 4 processus (calculs, communications des données) (**sur la feuille "petits carreaux"**),
2. Écrire la parallélisation s'appuyant sur MPI pour un nombre quelconque de processus (**dans le fichier** `equilibrage_par.c` (recopie de `equilibrage.c`)),
3. Exprimer, pour chaque étape, la complexité nécessaire (en nombre d'opérations ou en quantité de communications) en fonction de la taille de l'image (en nombre de pixels) et du nombre de processus (hors fonctions read_image et save_image).

Remarques

Ce qui est évalué dans cet exercice est votre aptitude à paralléliser un code séquentiel
- quels sont les calculs parallélisables,
- comment se répartit le travail,
- quelles sont les communications, de quelle nature et quand doivent-elles s'effectuer,
- quels sont les nouveaux objets nécessaires ?

On ne s'attachera pas, au niveau de l'évaluation, à vérifier que c'est syntaxiquement correct (on ne vous fournit d'ailleurs pas la possibilité de compiler cet exercice).

2. (

```
// N nombre de pixels de l'image
int N = get_image_size(filename);

// allocation de la mémoire pour contenir l'image
// variable pixels : tableau de 3*N octets (3 octets par pixels, RVB)
char *pixels = allocate_image(N);

// lecture de l'image
read_image(pixels, N, filename);

// tableau lumi contenant la luminance de chaque pixel
// (valeur entre 0 et 255 => tableau d'entiers de taille N)
int *lumi = allocate_lumi(N);

// calcul de la luminance -> complexité : 1 calcul par pixel
compute_luminance(lumi, pixels, N);

// tableau histo de taille 256 qui compte le nombre de pixels
// ayant la luminance correspondante
int *histo = allocate_histo(256);

// calcul de l'histogramme -> complexité : 1 calcul par pixel
compute_histo(histo, lumi, N);

// Équilibrage de l'image d'origine en utilisant l'histogramme :
// on crée une nouvelle image -> complexité : 1 calcul par pixel
// (calcul faisant intervenir la valeur du pixel et l'histogramme
// de l'image complète) ;
// la saturation (nombre de pixels ayant la valeur 0 ou la valeur 255)
// est aussi calculée
char *new_pixels = allocate_image(N);
int saturation;

equalize(new_pixels, &saturation, pixels, histo, N);

// sauvegarde de l'image égalisée
save_image(new_pixels, N, filename2);

// affichage de la saturation
printf("la saturation est de %d\n", saturation)
```