PPAR
Travaux Dirigés

Chargés de TD : Chirstoph Lauter et Pierre Fortin

Contact : Christoph.Lauter@lip6.fr, Pierre.Fortin@lip6.fr

26 juillet 2013

Ce document est le résultat de plusieurs années d'enseignement sur le parallélisme. Le principal contributeur a été J.-L. Lamotte. Un certain nombre de corrections et d'améliorations ont été apportées par D. Béréziat et P. Fortin.

Chapitre 1

Algorithmique parallèle

1.1 Tri pair-impair

Nous allons étudier une adaptation parallèle de l'algorithme séquentiel du tri par bulle, dont le principe est donné par l'algorithme 1. La complexité en nombre de comparaisons de l'algorithme séquentiel du tri par bulle est :

$$\sum_{i=0}^{N-2} N-1-i = \sum_{j=1}^{N-1} j = N(N-1)/2 = \mathcal{O}(N^2)$$

Algorithme 1 Tri par bulle

Pré-condition : T : tableau de N données à trier
1: **Pour** $i = 0$ à $N-2$ **faire**
2: **Pour** $j = 1$ à $N-1-i$ **faire**
3: **Si** $T[j] < T[j-1]$ **Alors**
4: échanger $T[j]$ et $T[j-1]$
5: **Fin si**
6: **Fin pour**
7: **Fin pour**

Pour la version parallèle, on dispose d'un réseau linéaire de P processeurs (architecture MIMD-DM), sur lequel les processeurs communiquent par échange de message.

1. On suppose tout d'abord qu'on dispose de $P = N$ processeurs. Chaque processeur (P_i) possède donc un élément du tableau $(T[i])$. Ecrire un algorithme SPMD de tri parallèle basé sur les procédures `compare_echange_min` et `compare_echange_max`. A la première étape, les processeurs de numéro pair comparent leur élément avec leur voisin de droite (et réciproquement). A l'étape suivante les processeurs de numéro pair comparent leur élément avec leur voisin de gauche (et réciproquement), et ainsi de suite...
   On pourra tester l'algorithme sur le tableau suivant :

34	87	65	15	71	45	32	10

2. On suppose maintenant qu'on dispose de $P = N/k$ processeurs. Ecrire l'algorithme SPMD de tri parallèle correspondant en se basant sur les procédures `compare_echange_mins` et `compare_echange_maxs`.

Procédure 2 compare_echange_{min,max}(id_voisin)

Pré-condition : id_voisin : paramètre d'entrée (compris entre 0 et $P-1$)
Pré-condition : id : numéro du processeur courant (de 0 a $P-1$)
1: **Si** $id\_voisin \geq 0$ et $id\_voisin \leq P-1$ **Alors**
2: envoyer $T[id]$ à id_voisin et recevoir $T[id\_voisin]$ de la part de id_voisin
3: $T[id] = \{\text{MIN,MAX}\}(T[id], T[id\_voisin])$
4: **Fin si**

Procédure 3 compare_echange_{mins,maxs}(id_voisin, k)

Pré-condition : id_voisin : paramètre d'entrée (compris entre 0 et $P-1$),
k : paramètre d'entrée
Pré-condition : id : numéro du processeur courant (de 0 a $P-1$)
1: **Si** $id\_voisin \geq 0$ et $id\_voisin \leq P-1$ **Alors**
2: envoyer les k éléments de id à id_voisin, et recevoir de la part de id_voisin les k éléments de id_voisin
3: conserver les k plus {**petits**, **grands**} éléments de l'ensemble des k éléments de id et des k éléments de id_voisin
4: **Fin si**

3. En supposant qu'on utilise un algorithme de tri (par comparaison) optimal en $\mathcal{O}(N \ln N)$ (par exemple le « tri rapide »- *quicksort*) pour le tri initial local à chaque processeur, quelle est la complexité théorique de cet algorithme en parallèle ? Quelle est l'accélération correspondante ?

1.2 Produit matriciel

On dispose de P processeurs et on souhaite effectuer le produit matriciel : $C = A \times B$, où A, B et C sont des mactrices carrées de $N \times N$ éléments.

On suppose que N est divisible par P. Chaque processeur est identifié par un numéro $0 \leq r \leq P-1$ (son « rang »).

1. On attribue un bloc continu de $h = N/P$ lignes complètes de C à chaque processeur. Donner un algorithme simple du produit matriciel parallèle correspondant.
2. Quel est l'inconvénient de cet algorithme ?
3. En considérant que les processeurs forment un anneau, donner un autre algorithme du produit matriciel parallèle qui ne présente pas cette contrainte.

1.3 Réduction

On dispose de P processus et on souhaite implémenter un algorithme de réduction effectuant la sommation des valeurs entières possédées par chaque processus. La racine de l'algorithme de réduction sera le processus de numéro 0 (et c'est donc lui qui affichera le résultat).

Définissez un algorithme efficace approprié *sans routine de communication collective.*

Chapitre 2

Introduction à MPI

Nous allons utiliser **OpenMPI** qui implémente le standard **MPI** sur le réseau de stations de la salle de TP. Le mode de programmation utilisé sera le mode **SPMD** (le même exécutable pour tous les processus, des branchements en fonction du **numéro du processus** (son **rang**) permettant de leur faire exécuter des tâches différentes)

2.1 OpenMPI

OpenMPI est un ensemble de programmes permettant l'utilisation de **MPI** sur un réseau hétérogène de machines (c'est le successeur d'une précédente implémentation de MPI nommée **LAM** - **L**ocal **A**rea **M**ulticomputer). Pour qu'un utilisateur puisse utiliser **OpenMPI** sur une machine parallèle constituée par exemple d'un PC sous linux (nous lui donnerons le nom pc_linux) et d'une station SUN (que nous appellerons station_sun), il faut que :

- cet utilisateur ait un compte sur les 2 machines
- **OpenMPI** soit installé sur les 2 machines et que la variable d'environnement `PATH` de l'utilisateur contienne sur les 2 machines le répertoire contenant l'application **MPI** à exécuter.

De plus, pour avoir accès aux commandes **OpenMPI** le répertoire contenant ces commandes doit être contenu dans la variable d'environnement `PATH`. A l'ARI, c'est le répertoire :

```
/usr/local/bin
```

De même, pour que le programme s'exécute correctement sur chaque machine, le répertoire contenant les bibliothèques dynamiques d'**OpenMPI** doit être contenu dans la variable d'environnement `LD_LIBRARY_PATH`. A l'ARI, c'est le répertoire

```
/usr/local/lib
```

Vous devez donc modifier votre fichier de configuration (`.bashrc` par exemple) en veillant à ce que les modifications soient bien prises en compte pour les shells non interactifs.

Concrètement, il faut rajouter les lignes suivantes dans votre `~/.bashrc` :

```
export PATH=/usr/local/bin:${PATH}
export LD_LIBRARY_PATH=/usr/local/lib:${LD_LIBRARY_PATH}
```

Attention, il faut les mettre au début du fichier `.bashrc`, avant les éventuelles lignes suivantes :


```
# If not running interactively, don't do anything
[ -z "$PS1" ] && return
```

Par ailleurs, il peut être nécessaire de vérifier que les lignes suivantes sont présentes dans le fichier `.bash_profile` :

```
# include .bashrc if it exists
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

Par défaut `ssh` demande le mot de passe à chaque connexion, mais il est possible d'utiliser un agent `ssh` avec un système de clefs privées et publiques pour éviter cela. .

Pour cela :

- vous devez créer des clefs `ssh` grâce à la commande [1] :
  `$ ssh-keygen -t dsa`
  Ces clefs **doivent avoir un mot de passe**. Vous les générez **une fois pour toutes**, vous n'aurez **jamais** besoin de les recréer : vous les conservez pour les prochains TPs.
- Vous devez ensuite permettre l'accès à votre compte grâce à la clef :
  `$ cd ~/.ssh/`
  `$ cat id_dsa.pub >> authorized_keys`

Ensuite il est possible de se connecter à une machine sans utiliser le mot de passe de votre compte mais celui de la clef. L'agent `ssh` peut alors donner ce mot de passe automatiquement.

Lorsque vous autorisez la connexion par clefs, l'agent s'occupe de vous authentifier automatiquement, pour cela, il suffit de le lancer :

```
$ ssh-agent bash
```

puis de lui donner le mot de passe :

```
$ ssh-add
```

Cette authentification unique permet de se connecter directement sur toutes les machines de l'ARI, sans mot de passe. Vérifiez dès maintenant que vous pouvez effectivement vous connecter sur n'importe quelle machine distante de la salle sans mot de passe (et sans message d'erreur).

Lorsque vous aurez fini à la fin du TP, vous détruirez l'agent `ssh` par

```
$ ssh-agent -k
```

2.1.1 Le schéma de boot

Le schéma de boot est défini dans un fichier du nom de votre choix (par exemple **hostfile**) que vous placez dans votre répertoire de travail (répertoire `PPAR`, par exemple). Celui-ci contient le nom de toutes les machines qui vont former votre machine parallèle, avec comme syntaxe un nom de machine par ligne. Vous devez inclure le nom de la machine sur laquelle vous êtes actuellement connectés. Soit, dans notre cas :

1. Le signe `$` indique l'invite de l'interpréteur de commande

```
# Toute ligne commencant pas un dièse
# est considérée comme un commentaire
#
# Notre machine est composée de 2 noeuds
pc_linux
station_sun
```

Le PC sous linux est alors considéré comme le premier nœud de la machine, et la station SUN comme le second. Nous aurions pu ajouter toutes les machines disponibles sur notre réseau vérifiant les conditions ci-dessus. Cela ne veut pas dire qu'à chaque fois que nous exécuterons un programme, tous les nœuds seront utilisés.

2.1.2 Compilation et exécution de programme MPI

Pour compiler le programme C `toto.c`, il suffit de taper la commande suivante :

```
$ mpicc -o toto toto.c
```

L'exécutable obtenu s'appelle donc `toto`. Celui-ci est lancé à l'aide de la commande `mpirun`. Pour que le programme s'exécute correctement sur chaque machine, le répertoire contenant les bibliothèques dynamiques d'OpenMPI (à l'ARI, c'est le répertoire `/usr/local/lib/`) doit être contenu dans la variable d'environnement `LD_LIBRARY_PATH`. Vous devez donc modifier votre fichier de configuration (`.bashrc` par exemple) en veillant à ce que la modification soit bien prise en compte pour les shells non interactifs.

L'option `-n #` stipule le nombre de processus à créer sur les nœuds de votre machine parallèle.

```
$  mpirun -n 2 -hostfile hostfile ./toto
```

Dans ce cas deux processus exécutant le programme toto sont lancés, le premier sur le PC Linux, le second sur la station SUN. La commande

```
$ mpirun  -n 3 -hostfile hostfile ./toto
```

aurait créé trois processus, le premier sur le PC, le second sur la station SUN et le troisième sur le PC. L'attribution des processus est cyclique par rapport au numéro des nœuds. Dans le cas de mesures de performance, vous ne devrez bien sûr pas lancer plusieurs processus sur un même nœud (sauf éventuellement sur un nœud multicœur).

2.1.3 Déboguer votre programme

Nous déboguerons nos programmes MPI avec des `printf` uniquement (attention à l'ordre des affichages entre les différents processus, voir section 2.2).

Il est aussi possible d'utiliser `gdb` : le plus simple est alors de lancer tous les processus MPI en local sur votre machine, mais ceci change l'exécution parallèle et ne permet donc pas de détecter tous les bugs qui apparaissent en parallèle.

Pour plus d'informations sur le déboguage avec OpenMPI, voir :
`http://www.open-mpi.org/faq/?category=debugging#serial-debuggers`

2.2 TP

Pour commencer, vous pouvez récupérer la documentation sur MPI (version 1.1) au format `pdf` à cette adresse : `http://www.mpi-forum.org/`

Pour le premier TP, nous allons manipuler les fonctions Send et Recv. La première difficulté consiste à faire marcher le programme suivant sur la machine parallèle de votre choix.

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
#include <unistd.h>

#define SIZE_H_N 50

int main(int argc, char* argv[])
{
    int          my_rank;       /* rang du processus      */
    int          p;             /* nombre de  processus   */
    int          source;        /* rang de l'emetteur */
    int          dest;          /* rang du recepteur      */
    int          tag = 0;       /* etiquette du  message      */
    char         message[100];
    MPI_Status   status;
    char hostname[SIZE_H_N] ;

    gethostname(hostname,SIZE_H_N);

    /* Initialisation  */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);


    if (my_rank != 0)
    {
        /* Creation du message */
        sprintf(message, "Coucou du processus #%d depuis %s!",
                my_rank, hostname);
        dest = 0;

        MPI_Send(message, strlen(message)+1, MPI_CHAR,
            dest, tag, MPI_COMM_WORLD);
    } else
    {
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("Sur %s, le processus #%d a recu le message : %s\n",
                    hostname, my_rank, message);
        }
    }

    /* Desactivation */
    MPI_Finalize();
}
```

Question 1 Faîtes tourner l'application plusieurs fois avec un nombre égal de processus, puis

en les faisant varier.

Question 2 Mettez des `printf` un peu partout dans le programme. Que se passe-t-il ?

Question 3 Remplacez la variable `source` dans le `MPI_Recv` par l'identificateur `MPI_ANY_SOURCE`. Faîtes les tests plusieurs fois de suite. Que se passe-t-il ? Expliquez.

Question 4 Écrivez un programme tel que chaque processus envoie une chaîne de caractères à son successeur (le processus $rang+1$ si $rang < p-1$, le processus 0 sinon), et qu'il reçoit un message du processus précédent. Une fois que votre programme fonctionne, remplacez `MPI_Send` par `MPI_Ssend`. Que se passe-t-il ? On appelera ce programme `ex_ssend.c`

Question 5 Recopiez `ex_ssend.c` dans `ex_ssend_correcte.c` Tout en gardant `MPI_Ssend`, changez l'algorithme pour que le processus 0 envoie en premier son message au processus 1, qui n'enverra son message au processus 2 qu'après avoir reçu son message de 0. De la même façon, le processus 2 n'enverra son message au processus 3 qu'après avoir reçu de 1, et ainsi de suite...

Question 6 Recopiez `ex_ssend.c` dans `ex_send.c` Remplacez le `MPI_Ssend` par `MPI_Send`. Faites varier la taille des données envoyées par le `MPI_Send` jusqu'à 100 ko. Que se passe-t-il ?

Exercice supplémentaire On reprend l'exercice du TD1 sur l'algorithme de réduction effectuant la sommation des valeurs entières possédées P processus.

1. Réalisez un programme parallèle MPI implémentant cet algorithme. Le programme calculera la somme globale des valeurs aléatoires entières générées par chaque processus, de façon à ce que le processus 0 reçoive et affiche la valeur somme.
2. Réécrivez votre programme à l'aide d'une routine de communication collective, puis modifiez le de façon à ce que la somme globale soit disponible simultanément sur tous les processus.

Nettoyage A la fin du TP, ne pas oublier de tuer son agent `ssh` : `$ ssh-agent -k`
Vérifier ses processus sur la machine locale : `ps uxww`

2.3 Références

Quelques implémentations du domaine public :
- MPICH (Argonne NL Missisipi State U.)
  `http://www.mcs.anl.gov/mpi/mpich`
- LAM (Ohio supercomputer Center) :
  `http://www.mpi.nd.edu/lam/`
- Open MPI :
  `http://www.open-mpi.org/`

Livres :
- `http://www-unix.mcs.anl.gov/mpi/usingmpi/`
- `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`
- `http://fawlty.cs.usfca.edu/mpi/`
- *Parallel Programming in C with MPI and OpenMP*, M.J. Quinn, McGraw-Hill

Doccumentations diverses :
- documents officiels : `http://www.mpi-forum.org/`

- `http://www.idris.fr/data/cours/parallel/mpi/mpi\_cours.html`
- `http://www-unix.mcs.anl.gov/mpi/`
- `http://www.mpi.nd.edu/lam/`
- Newsgroup : `comp.parallel.mpi`
- moteurs de recherche : mots clefs : mpi, mpich, lam, message passing, ...

Chapitre 3

Ensemble de Mandelbrot

3.1 Introduction

L'ensemble de Mandelbrot est constitué des points c du plan complexe C pour lesquels le schéma itératif suivant :

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases} \tag{3.1}$$

ne diverge pas. En posant $z = x + iy$ et $c = a + ib$, l'équation (3.1) se réécrit :

$$\begin{cases} x_{n+1} = x_n^2 - y_n^2 + a \\ y_{n+1} = 2x_n y_n + b \end{cases}$$

avec les conditions initiales $x_0 = y_0 = 0$.

On peut montrer que s'il existe un entier n tel que $|z_n| \geq 2$ (soit $|z_n|^2 = (x_n^2 + y_n^2) \geq 4$), alors le schéma (3.1) diverge. Pour en savoir plus, on pourra consulter [1, 2].

3.2 Structure des images en mémoire

Une image est un tableau à deux dimensions. Chaque élément de ce tableau s'appelle un *pixel*, abbréviation de *picture element*. La valeur de ce point est, selon le type d'image, une valeur de niveau de gris, une couleur ou une valeur de radiance. Ce tableau est organisé en mémoire ligne par ligne : on trouve la première ligne, puis la seconde, *etc.* En particulier, nous manipulerons des images codées sur un octet (taille d'un pixel), et la valeur de chaque pixel (donc un entier compris entre 0 et 255) représente un index dans une table de couleur.

En C, cela se traduit par :

```
{
  unsigned char *Image;
  int dimx, dimy;        // largeur et hauteur de l'image
  int i,j;              // indices pour parcourir les pixels de l'image

  Image = (unsigned char *) malloc (sizeof(unsigned char)*dimx*dimy);

  for(i=0; i<dimy; i++)
    for(j=0; j<dimx; j++)
      Image[j+i*dimx] = 0;  // acces au pixel i,j
}
```


Par convention, le pixel de coordonnées $(0, 0)$ est le point supérieur gauche d'une image affiché à l'écran, et c'est donc aussi le premier élément du tableau `Image` en mémoire.

3.3 Format d'image

Il existe une très grande multitude de format d'image, c'est-à-dire de façon de stocker dans un fichier une image. Nous utilisons le format Sun Rasterfile qui a le grand mérite d'être très simple à mettre en œuvre et qui est visualisable avec la plupart des programmes d'affichage d'image [1].

Un fichier rasterfile est constitué d'une entête qui décrit les caractéristiques de l'image (taille, codage, ...), suivie d'une suite de mot de 1 octet décrivant la table des couleurs (si besoin). Ensuite vient l'image proprement dite, stockée sous la forme d'un tableau de données brutes.

3.4 Algorithme séquentiel

Synopsis de l'algorithme :

1. Pour chaque point du domaine :
   (a) calculer le nombre d'itération pour lequel le schéma itératif diverge ;
   (b) mettre à jour la valeur du pixel correspondant ;
2. Sauver l'image.

3.5 Questions

1. Discuter de la manière dont l'algorithme présenté en section 3.4 pourrait être parallélisé.
2. Proposer une version parallèle de cet algorithme qui répartisse de façon équilibrée les données à calculer sur chaque processeur. On utilisera les transmissions basiques de MPI (`MPI_Send` et `MPI_Recv`). En écrire le code C.
3. Comparer les temps d'exécution pour chaque processeur avec différents nombres de processeurs, ainsi que les efficacités parallèles obtenues. On étudiera en particulier le cas à 8 processeurs avec les paramètres par défaut.
4. L'équilibrage de charge précédent possède un gros défaut. Lequel ? Proposer un autre équilibrage de charge pour résoudre ce problème.
5. Implémenter en C l'algorithme correspondant à ce nouvel équilibrage de charge.

1. par exemple `display` (de la suite logicielle `ImageMagick`) sur Linux

Bibliographie

[1] The Fractal Geometry of the Mandelbrot Set, *Robert L. Devaney* `http://math.bu.edu/DYSYS/FRACGEOM/`

[2] The Spanky Fractal Database, *Noël Giffin*, `http://spanky.triumf.ca/www/welcome1.html`

Chapitre 4

Convolution

4.1 Introduction

Il existe en traitement d'image une opération très importante : le filtrage par convolution. De quoi s'agit-il ? Une convolution est l'opérateur noté $\star$ et définie de la façon suivante :

$$f \star g(x) = \int f(x-y)g(y)dy \tag{4.1}$$

où f et g sont deux fonctions réelles intégrables. Pour les fonctions de $\mathbb{R}^2$, la définition devient :

$$f \star g(x', y') = \int\int f(x'-x, y'-y)g(x,y)dxdy \tag{4.2}$$

Dans l'espace discret où vivent les images, la formule s'écrit :

$$f \star g(k,l) = \sum_i \sum_j f(k-i, l-j)g(i,j) \tag{4.3}$$

On supposera que nos fonctions ont un support (i.e. un domaine de définition) borné. Ainsi, les indices i et j varient dans un intervalle fini.

En particulier, on suppose f définie sur $[1, N] \times [1, P]$ et g définie sur $[-\frac{n}{2}, \frac{n}{2}] \times [-\frac{p}{2}, \frac{p}{2}]$, la formule (4.3) devient :

$$f \star g(k,l) = \sum_{\substack{1 \leq k-i \leq N \\ -\frac{n}{2} \leq i \leq \frac{n}{2}}} \sum_{\substack{1 \leq l-j \leq P \\ -\frac{p}{2} \leq j \leq \frac{p}{2}}} f(k-i, l-j)g(i,j) \tag{4.4}$$

On remarque immédiatement que cette définition est plus compliquée que la précédente. En effet, la convolution revient, entre autre, à translater une des deux fonctions. Sur un support infini, la translation n'a pas d'effet sur le support. En revanche, avec un support fini, le domaine est également translaté. La conséquence est donc que le support de la convoluée est différent des supports des fonctions convoluées. Dans l'exemple de l'équation 4.4, le support de $f \star g$ est $[1 - \frac{n}{2}, N + \frac{n}{2}] \times [1 - \frac{p}{2}, P + \frac{p}{2}]$. La figure 4.1 illustre cela : le calcul du pixel en bas à droite est possible ; sa valeur est a priori non nulle parce qu'elle dépend de la valeur d'un pixel de I (celui en gris), les autres étant nuls.

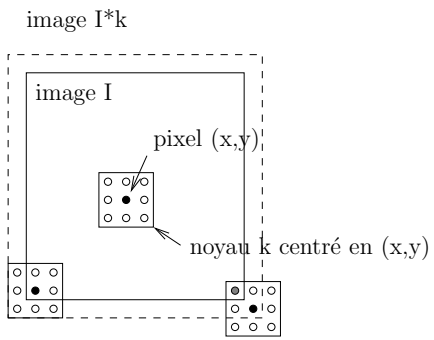

FIGURE 4.1 – Convolution de I par k : $I \star k(x, y)$ est la somme pondérée (par k) des valeurs de I en (x,y) et de leurs voisins. On illustre aussi le fait que le domaine de $I \star k$ (en pointillé) est le domaine de I augmenté de celui de k. Au-dela, les valeurs sont nulles.

4.2 Applications aux images

En pratique, nous l'avons déjà dit, la convolution d'image est une opération très importante. Soit I une image de taille $N \times P$, $I(x, y)$ est donc la valeur de luminescence du pixel (x, y). Cette image peut être *convoluée* avec une autre image k que l'on appelle *noyau (ou kernel en anglais) de convolution*. Le noyau k est généralement de petite taille (typiquement 3×3, 5×5 ou 7×7) et représente souvent un opérateur différentiel (mais pas nécessairement), il est représenté centré en 0.

Par exemple, si l'on souhaite calculer la dérivée en x de I ($\frac{\partial I}{\partial x}$), on peut approcher par différence finie (développements de Taylor à l'ordre 1) l'opérateur :

$$\frac{\partial I(x,y)}{\partial x} \sim \frac{1}{2}(I(x+1,y) - I(x-1,y)) \tag{4.5}$$

ce qui revient à calculer $I \star k$ avec :

$$k = \begin{pmatrix} 0 & 0 & 0 \\ -1/2 & 0 & 1/2 \\ 0 & 0 & 0 \end{pmatrix} \tag{4.6}$$

Avec d'autres opérateurs, on peut lisser une image, extraire des contours, *etc.* On peut voir la convolution comme une opération de moyennage pondérée sur un voisinage fixé par le noyau de convolution (voir figure 4.1).

4.3 Algorithme séquentiel

Avec la définition donnée de la convolution discrète (équation (4.4)), on ne peut pas calculer la convolution sur les pixels au bord de l'image car elle nécessite l'accès à des pixels

non définis, pourtant le calcul sur les bords est possible (voir la figure 4.1, en bas à gauche et en bas à droite).

Pour résoudre ce problème, il y a deux méthodes :

- soit on agrandit artificiellement la taille de l'image, les nouveaux pixels ont alors une valeur nulle,
- soit on exclut du calcul de convolution les pixels sur les bords de l'image.

Notre algorithme choisit la deuxième méthode car notre propos n'est pas d'implémenter rigoureusement la convolution mais de la paralléliser.

Pour obtenir de longs temps de calcul et illustrer les envois de données de MPI, l'opération de convolution est itérée un grand nombre de fois. On veut donc implémenter l'opération :

$$I \star^n k = I \star \underbrace{k \star k \cdots \star k}_{n \text{ fois}} \tag{4.7}$$

Comme nombre d'applications numériques (comme la résolution d'équations aux dérivées partielles) requièrent la répétition d'un grand nombre de convolutions, ce choix est pertinent.

Finalement, l'algorithme est simple :

1. Lecture de la ligne commande (protocole Argv).
2. Chargement de l'image (fonction lire_rasterfile).
3. Répéter `nbiter` fois la convolution de l'image (≪ pointée ≫ par le champ `r.data` de la variable `r` de type `Raster`) par un des 5 filtres :
   (a) allocation mémoire d'un tampon image intermédiaire,
   (b) pour chaque pixel (i, j) de l'image (sauf les bords) faire :
   – le pixel (i, j) de l'image tampon reçoit une combinaision linéaire (c'est l'opération de convolution) du pixel (i, j) et de ses 8 plus proches voisins dans l'image (`r.data`).
   (c) recopie du tampon intermédiaire dans l'image (`r.data`) et libération mémoire du tampon image intermédiaire.
4. Sauvegarde de l'image (fonction sauve_rasterfile).

Les 5 différentes combinaisons linéaires représente les cinq filtres possibles (voir l'instruction switch dans le listing) : ces opérations sont dans l'ordre : un filtre moyenneur, un autre filtre moyenneur (avec plus de poids au centre), un détecteur de contour (c'est l'opérateur Laplacien discrétisé par différences finies : un tel opérateur donne une forte réponse sur les contours de type "pic", voir figure 4.2), un autre détecteur de contour qui détecte les contours de type "marche" (voir figure 4.2). et finalement un filtre médian (non linéaire : on trie les pixels selon leur luminosité et on retient la valeur médiane).

4.4 Questions

1. Dans la fonction `convolution()`, pourquoi doit-on préparer un tampon intermédiaire au lieu de faire le calcul directement sur l'image ?
2. Quelles sont les séquences parallélisables de l'algorithme ?
3. Sachant que la taille du noyau k de convolution est de 3×3 pixels, quelle est la complexité théorique du calcul d'un pixel de $I\star k$? Quel type d'équilibrage de charge doit-on prévoir entre les processeurs ?

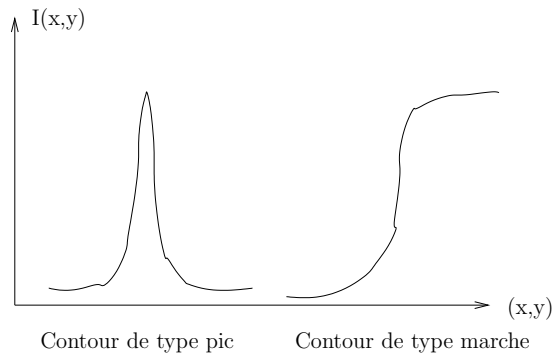

FIGURE 4.2 – Exemple de contours.

4. Quel découpage (répartition des données entre processeur) est naturel dans ce contexte ?
5. Quel problème (aux bords des blocs d'image) survient lors de l'itération de l'opération de convolution ?
6. Implémenter un algorithme parallèle avec des envois bloquants de messages.
7. Question subsidiaire : implémenter l'algorithme avec cette fois des primitives non-bloquantes et de façon à ce que les temps de calcul recouvrent les temps de communication. Analyser les performances obtenues.

Chapitre 5

OpenMP et programmation hybride MPI-OpenMP

Pour cette séance, nous allons utiliser le compilateur `gcc` qui permet de paralléliser des codes contenant des directives OpenMP. Pour la génération du code exécutable, ajoutez l'option `-fopenmp` aussi bien à la compilation qu'à l'édition de lien.

5.1 OpenMP

Nombre de cœurs disponibles. Vérifiez à l'aide de la commande `cat /proc/cpuinfo`, le nombre de cœurs disponibles sur votre machine.

Premier programme. Utilisez un des programmes du cours pour tester votre environnement. N'oubliez pas de de positionner la variable `OMP_NUM_THREADS` !
Rappel pour la compilation : `gcc -fopenmp toto.c -o toto`

Multiplication de deux matrices. Le fichier `matmul.c` contient le code source d'un programme de multiplication de matrices. Deux jeux de tests sont insérés dans les sources.

1. Parallélisez ce code avec OpenMP et ajoutez des affichages qui mettent en évidence l'exécution multi-thread.
2. Après avoir mis en commentaires les affichages dans les boucles de calcul, exécutez le programme en mode statique, et choisissez différentes valeurs pour la taille des blocs.
3. Même chose avec le mode dynamique.
4. Modifiez le code de `matmul.c` pour que la boucle sur `k` soit la plus externe. Quelle est la conséquence de cette modification ? Parallélisez ce nouveau code avec OpenMP, et observez l'efficacité obtenue pour un équilibrage de charge adapté.

Calcul de fractales. Le fichier `mandel.c` contient le code source d'un programme qui calcule les valeurs de l'espace de Mandelbrot.
Parallélisez ce programme avec OpenMP, puis exécutez le avec des équilibrages de charge statiques et dynamiques et avec différentes valeurs de taille de bloc.

Convolution. Le fichier `convol.c` contient le code source d'un programme qui applique des opérateurs de convolution à des images.

Parallélisez ce programme avec OpenMP, puis exécutez le avec des équilibrages de charge statiques et dynamiques et avec différentes valeurs de taille de bloc.

5.2 MPI et OpenMP

On s'intéresse ici à une programmation parallèle hybride MPI-OpenMP.

Support des threads avec MPI. A l'aide de la section 12.4.3 du manuel MPI-2.2 (disponible à l'adresse : `http://www.mpi-forum.org/`), déterminez le niveau du support des threads de l'implémentation MPI dont vous disposez.

Convolution. En reprenant la version parallèle MPI de `convol.c` (**sans** recouvrement des communications par le calcul), écrivez une version parallèle hybride MPI-thread qui respecte le niveau de support des threads offert.

Pour un nombre de processeurs fixé, chaque processeur disposant d'au moins 2 cœurs, quel est l'avantage en théorie de cette version MPI-thread par rapport à la version précédente dite "MPI pur" ?

Vérifiez-le avec des tests de performance en utilisant par exemple deux PC multicœurs.

Aide : pour lancer avec `mpirun` un processus MPI en positionnant certaines variables d'environnement, vous pouvez utiliser par exemple :

```
mpirun -x OMP_NUM_THREADS=2,OMP_SCHEDULE="static\,64" -n 2 ./a.out ...
```