
TP5: Diffusion sur réseau carré

Rapport

BEN MBARKA MOEZ GHRISS HICHAM

1 Introduction

2 Structures des données

La grille est modélisée par une matrice. Celle ci est définie dans le fichier *matrix.h* par la structure suivante :

```
typedef struct matrix_{
    int size ; /* La taille de la matrice */
    double* content; /* Pointeur vers le début de son contenu */
}matrix;
```

Une matrice est stockée en colonne sous forme d'un bloc contigu de n éléments. Le fichier *matrix.c* implémente les différents méthodes et accesseurs permettant de manipuler cette structure.

Dans la suite la valeur d'une cellule dans la position i, j dans une grille u sera noté $u_{i,j}$. La valeur de cette cellule à l'itération t sera noté $u_{i,j}^t$.

La taille de la grille est n . La taille d'une portion carré est M .

3 Conception

Le but de l'application est de réaliser une diffusion sur une grille rectangulaire et non torique. La diffusion consiste à appliquer la relation suivante sur toute cellule de la grille u de taille n :

$$u_{i,j}^{t+1} = u_{i+1,j}^t + u_{i-1,j}^t + u_{i,j+1}^t + u_{i,j-1}^t \text{ avec les conditions aux bords :}$$
$$u_{i+1,j}^t = u_{i,j}^t \text{ si } i + 1 > n$$
$$u_{i-1,j}^t = u_{i,j}^t \text{ si } i - 1 < 0$$
$$u_{i,j+1}^t = u_{i,j}^t \text{ si } j + 1 > n$$
$$u_{i,j-1}^t = u_{i,j}^t \text{ si } j - 1 < 0$$

Cette diffusion doit être parallélisé en distribuant des portions carrés de la grille sur un certain nombre de processus.

Les calculs à effectuer par chaque processus peuvent être divisés en deux types :

- Diffusion totalement locale : Elle concerne la partie intérieure de la portion locale pour laquelle le processus possède tous les opérants de la formule de diffusion.
- Diffusion sur les bords : Elle concerne les premières et les dernières lignes et colonnes où le processus ne possède pas au moins un des opérants. Pour réaliser cette étape, le processus a donc besoin de communiquer avec les processus voisins.

Pour effectuer ces communications, un processus donné a besoin de savoir les rangs de ses voisins pour pouvoir échanger les valeurs sur les bords. Ceci est assuré par l'utilisation d'une même numérotation de la grille visible pour tous les processus.(voir 3)

Pour recevoir les bords des portions voisines, chaque processus utilise des zones “fantômes” autour de sa portion locale, ce qui revient à utiliser des matrices de taille $M+2$ à la place de M .

D’autre part, la valeur d’une cellule étant utilisée par les 4 cellules voisines, la diffusion ne peut pas se faire sur place. A chaque itération, le résultat de la diffusion dans une matrice u est écrit dans une autre matrice w . A l’itération suivante, w joue le rôle de u et inversement.

L’algorithme utilisé est présenté dans le paragraphe suivant. Les sections qui suivent détaillent les choix pris ainsi que les implémentations utilisés.

3.1 Algorithme

```

Algorithme Diffusion dans une grille rectangulaire
Début
  fini=0
  TantQue fini=0 Faire
    Envoyer les bords aux voisins
    { L’envoi n’est pas bloquant }
    Recevoir les bords des voisins
    { Cette réception n’est pas bloquante }
    Diffusion locale
    { On fait les calculs ne nécessitant que des valeurs locales }
    Attente fin de la réception
    Terminer la diffusion locale
    Si mon_rang=0 Alors
      { Un seul processus fait le test de la terminaison }
      { à chaque itération }
      min:= calculer_min_global
      max:= calculer_max_global
      Si max-min< $\epsilon$  Alors
        fini =1
        { Terminaison détectée }
        Propager la terminaison
      FinSi
    FinTantQue
  Fin

```

Cet algorithme est exécuté par chacun des processus. Les détails sur les communications mises en place sont détaillés dans la section suivante(4.1).

3.2 terminaison

Dans le contexte des applications parallèles, on est amené à prouver deux types de terminaisons :

3.2.1 Terminaison locale

Il s'agit ici de montrer que la terminaison est détecté par au moins un processus (dans l'algorithme par exactement le processus de rang 0). Ce processus récupère avant la fin de chaque itération les valeurs *min* et *max* qui sont respectivement le minimum global et le maximum global dans la totalité de la grille (voir section 4.1).

On va s'intéresser ici à prouver que la condition de la terminaison ($max - min < \epsilon$) est vérifiée à partir d'une certaine itération.

Si on a la même valeur sur toute la grille, l'algorithme termine dès la première itération. On va supposer donc qu'il y a au moins deux valeurs distinctes.

Prouver la terminaison revient à montrer que la suite : $f(t) = max^t - min^t$ tends vers 0 quand t (le nombre d'itérations) tends vers l'infini.

On a $max^{t+1} = 1/5(a^t + b^t + c^t + d^t + e^t)$ avec a, b, c, d et e sont les voisins de la cellule qui contient le max à l'itération $t + 1$. D'où : $max^{t+1} \leq 1/5(5 * max^t) = max^t$. Et donc la suite max^t est décroissante. De même on prouve que min^t est croissante. Donc la suite $f(t)$ est décroissante. Et comme elle est réelle positive, elle est convergente. Supposant qu'elle converge vers r . Si $r > 0$ alors, $\forall \epsilon > 0 \exists$ un t_0 assez grand tels que $\forall t > t_0$ on a $f(t) - r < \epsilon$. Or à t_0 il existe au moins une case voisine à une case qui contient le max et dont la valeur est inférieure au max . D'où après un certains nombre d'itérations s (s dépend du nombre de cellules contenant la même valeur max à t_0) on a $max^{t_0+s} < max^{t_0}$ et donc $f(t_0 + s) < r$ et donc $f(t_0 + s) - r < 0$ ce qui contredit la convergence vers r .

En conclusion **f(t) converge vers 0**. D'où la terminaison locale de l'algorithme.

3.2.2 Terminaison globale

La terminaison globale est assurée par la propagation de la détection du processus 0 aux autres processus. (voir 4.1.3)

4 Réalisation

4.1 Communications

L'application implémente trois phases de communication.

4.1.1 Echanges

À chaque itération, un processus doit échanger les lignes et colonnes du bord avec les processus voisins. Pour optimiser ces communications on a utilisé des types dérivés spéciales pour les lignes et les colonnes :

- **MPL_line** : Un bloc contigu de M double.

```
MPI_Type_vector(M, 1, M, MPI_DOUBLE, &MPI_line);
```

- **MPI_column** : Un vecteur de n double avec un décalage de M .

```
MPI_Type_contiguous(M, MPI_DOUBLE, &MPI_column);
```

D'autre part, les différents échanges effectués impliquent toujours les mêmes buffers :

- Envois : Les premiers lignes et colonnes de la portion locale.
- Réception : La zone fantôme ajoutée autour de la portion.

Ceci suggère l'utilisation des communications persistantes. Ce type de communications permet de gagner sur le coût de la latence et les appels aux routines de communications dans une boucle.

Le schémas des communications persistantes est réalisé en deux étapes :

- **Déclaration des communication** : Elle est faite en dehors de la boucle avec les routines *MPI_Recv_init* et *MPI_Send_init* avec les buffers concernés. Ces routines construisent des structures de type *request*.

A noter que, vue les buffers concernés sont initialement dans u puis w puis u et ainsi de suite, on initialise les communications pour chacune des matrices.

- **Lancement effective des communication** : Dans la boucle, avec *MPI_Startall* et en utilisant les *request*. Selon la parité du numéro de l'itération, les *request* activés sont ceux correspondant à u ou à w .

4.1.2 Recouvrement calculs/communications

Pour pouvoir espérer faire du recouvrement des communications par les calculs, les communications mises en places pour les échanges sont non bloquantes. En effet, l'activation par *MPI_Start* d'une requête créée par un *MPI_xxxx_Init* est équivalente à une communication non bloquante faite par *MPI_Ixxxx*.

A chaque itération :

- Un processus donné doit échanger 2 lignes et 2 colonnes. Ce qui donne des communications en $O(M)$ où M est la dimension de la portion locale.
- Il doit effectuer un calcul local sur sa portion. A l'exception des cellules sur la frontière, ce calcul ne dépend que des valeurs locales. D'où un calcul totalement local en $O(M^2) > O(M)$.

Les communications asynchrones effectués par les communications persistantes non bloquantes permettent effectivement de recouvrir les communications par les calculs.

4.1.3 Vérification de la condition de terminaison et sa propagation

A chaque itération, tout processus calcul le minimum et le maximum locaux. Ce calcul est effectué, en temps réel, en même temps que la diffusion puisque celle-ci va impliquer la lecture de toutes les valeurs de la portion locale.

Pour vérifier la terminaison, le processus 0 récupère à chaque itération le minimum et maximum globaux. Ceci est fait par l'opération de réduction *MPI_MIN*

en utilisant `MPI_Reduce`.

Si la condition de terminaison est vérifiée, le processus 0 met localement la variable *fini* à 1.

Pour propager cette information, on a implémenté deux méthodes :

- Une diffusion standard : Le processus 0 diffuse la valeur de *fini* à la fin de l’itération.
- Communications unilatérales : le processus 0 ouvre une fenêtre sur cette variable avec `MPI_Win_create` et débute une zone d’accès avec `MPI_Win_fence` pour permettre aux autres processus de lire la valeur de *fini*. Cette méthode requiert l’utilisation de MPI2. Pour l’utiliser il faut compiler avec une directive spéciale (voir le mini-manuel d’utilisation).

Avec les deux méthodes, tous les processus sont en courant si la diffusion est terminée avant la fin de l’itération courante.

4.1.4 Rassemblement de la grille

Cette phase est effectuée après la boucle de la diffusion. Elle va permettre au processus 0 de récupérer la totalité de la grille. Cela suppose qu’au moins un processus particulier (ici celui de rang 0) possède suffisamment de mémoire pour sauvegarder la totalité de la grille alors qu’il suffit pour les autres de pouvoir sauvegarder la portion locale de taille plus petite.

Le rassemblement est effectué avec un `MPI_Gather` qui utilise deux nouveaux types dérivés :

- `MPI_bloc` : un vecteur qui permet d’encapsuler le buffer contenant la zone interne de la portion locale (sans les buffers de la frontière qui ont seulement servi pour recevoir les bords des voisins). Ce type sert pour l’envoi.
- `MPI_matrix` : $M * M$ éléments contigus pour encapsuler la totalité d’une portion. Ce type sert pour la réception.

4.2 Calculs locaux

Ces calculs n’utilisent que des valeurs locales au processus. Il sont effectués par la fonction *diffusion_locale* sans attendre la fin des communications. En même temps, en effectue le calcul des valeurs min et max locales.

4.3 Calculs sur les bords

Les calculs sur les bords nécessitent des valeurs sur des processus voisins. Donc, ils ne débutent qu’après la fin de la réception. Ceci est assuré par la routine `MPI_Waitall` sur les requêtes chargées de la réception. Ces calculs sont faits par la fonction *terminer_diffusion* qui met également à jour les valeurs locales de *min* et *max*.

5 Mini-Manuel utilisation

5.1 Configuration

Les paramètres à configurer sur le programme se trouve dans le fichier d'entête *config.h* dans le répertoire *include*. On peut modifier le test de terminaison *EPSILONE*, la taille des matrices locaux M ou le facteur de diminution Q (par défaut égale à 0.2).

Pour l'exécution on peut modifier la liste des noeuds qui participent à l'exécution du programme. Pour cela il faut modifier le fichier *machines_p4*.

5.2 Compilation

Pour compiler le programme il suffit d'exécuter la commande *make* sur la racine.

5.3 Exécution

L'exécution du programme est faite à l'aide du script *go* selon la syntaxe suivante :

```
./go <nombre_de_processus> <nombre_de_lignes> <nombre_de_colonnes>
```

Avec : nombre_de_processus = nombre_de_lignes * nombre_de_colonnes

6 Mesures et exemples

Les mesures présentés dans cette partie suivent la configuration suivante :

- Les machines utilisées sont : node1, node2, node5, node7 et node8.
- Le nombre de processus est fixé à 6 (2x3) puis à 10 (5x2).
- Le test de terminaison est fixé à $EPSILONE = 0.1$
- Le facteur de normalisation est fixé à $Q = 0.2$ (1/5)

Le but est de voir l'effet de la variation de la taille des matrices locaux sur ce système. La figure ?? montre cette variation pour ces deux configurations. La figure ?? montre l'évolution du nombre d'itérations pour les mêmes cas de tests précédentes.

7 Listing des sources

```
|-- bin
| |
| '-- diffCarre
|-- go
|-- include
| |
| |-- config.h
| |-- init_methode.h
| '-- matrix.h
|-- machines_p4
```

```
|-- makefile
|-- src
|  |
|  |-- init_methode.c
|  |-- main.c
|  |-- matrix.c
|  '-- matrix.h
'-- tests
  |-- res_M100_EPS0.1_6_2_3_u.data
  |-- res_M10_EPS0.1_10_5_2_2u.data
  |-- res_M10_EPS0.1_20_5_4_2u.data
  |-- res_M10_EPS0.1_6_2_3_2u.data
  |-- res_M10_EPS0.1_6_2_3_u.data
  |-- res_M150_EPS0.1_6_2_3_u.data
  |-- res_M20_EPS0.1_10_5_2_2u.data
  |-- res_M20_EPS0.1_6_2_3_2u.data
  |-- res_M20_EPS0.1_6_2_3_u.data
  |-- res_M30_EPS0.1_10_5_2_2u.data
  |-- res_M30_EPS0.1_6_2_3_2u.data
  |-- res_M30_EPS0.1_6_2_3_u.data
  |-- res_M40_EPS0.1_10_5_2_2u.data
  |-- res_M40_EPS0.1_6_2_3_u.data
  '-- res_M60_EPS0.1_6_2_3_u.data
```