Communicateurs et produit matriciel en MPI

1 Communications collectives et communicateurs MPI

L'intérêt principal de MPI réside dans sa grande diversité de fonctions de communications collectives. Une diffusion de données se fait simplement en utilisant la fonction suivante :

```
int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm )
```

- `buffer` est l'adresse du buffer ;
- `count` est le nombre d'éléments dans le buffer ;
- `datatype` est le type MPI des éléments du buffer (`MPI_INT`,`MPI_FLOAT`,...) ;
- `root` l'indice du processeur qui effectue la diffusion ;
- `comm` le groupe de processus (ou *communicateur*) au sein duquel a lieu la diffusion.

Les opérations de communication collective impliquent toujours un communicateur donné et sont bloquantes pour cet ensemble de processus. Tous les processus du groupe effectuant la diffusion doivent donc appeler la fonction `MPI_Bcast` même si, selon leur rang dans le communicateur, cet appel n'a pas le même effet. Pour les processus qui ne sont pas l'émetteur, `buffer` sert à recevoir les données alors que pour l'émetteur, il contient les données à diffuser. Enfin, comme toutes les fonctions de communication collective, cette fonction est bloquante et ne permet donc pas de recouvrement des calculs et des communications.

Nous avons déjà parlé du communicateur `MPI_COMM_WORLD` qui regroupe l'intégralité des processus lancés. Il est possible de créer d'autres communicateurs, c'est à dire d'autres groupes de processus, notamment en utilisant la fonction `MPI_Comm_split`.

```
int MPI_Comm_split ( MPI_Comm comm, int color, int key, MPI_Comm *newcomm )
```

- `comm` est le communicateur que l'on souhaite scinder ;
- `color` est un entier positif qui détermine à quel ensemble on appartiendra, le nouveau communicateur regroupant les processus de même couleur ;
- `key` est un entier qui permet de déterminer le rang du processus au sein du nouveau communicateur. Deux processeurs de même couleur seront donc dans le même communicateur et leur rang sera déterminé en fonction de leurs valeurs respectives de `key` ;
- `newcomm` est le nouveau communicateur.

2 Produit de matrice parallèle

2.1 Rappel sur l'algorithme par double diffusion

On souhaite effectuer le calcul $C_{ij} = (A.B)_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$ pour tout i, j dans $[\![1, n]\!]$. L'algorithme 1 décrit une façon naturelle d'effectuer ce calcul mais pas forcément adaptée à une parallélisation directe.

```
MatMult(A, B, C)
 1 : Pour i = 1 to n
 2 :    Pour j = 1 to n
 3 :       Pour k = 1 to n
              C_ij ← C_ij + A_ik B_kj
```

Algorithme 1: Algorithme général séquentiel du produit de matrice.

Les boucles 1 et 2 de cet algorithme sont parallèles. La boucle de la ligne 3 correspond à une opération de réduction (qui peut s'effectuer par exemple à l'aide d'un arbre) et réduit le parallélisme. On séquentialise généralement une partie de ces boucles afin d'ordonner et de régulariser les calculs (éviter des migrations de données excessives et désordonnées), de simplifier le contrôle et surtout d'adapter la granularité de la machine cible.

Il est donc préférable de permuter les boucles pour arriver à l'algorithme 2 et d'effectuer alors la boucle externe (ligne 1) de manière séquentielle et les boucles internes (lignes 2 et 3) en parallèle. Si chaque C_{ij} est alloué à une unité de calcul fixe, alors les A_{ik} et les B_{kj} doivent circuler entre chaque

MatMult(A, B, C)
Pour $k = 1$ **to** n
  Pour tout i en parallèle
    Pour tout j en parallèle
      $C_{ij} \leftarrow C_{ij} + A_{ik}B_{kj}$

Algorithme 2: Algorithme parallèle du produit de matrice.

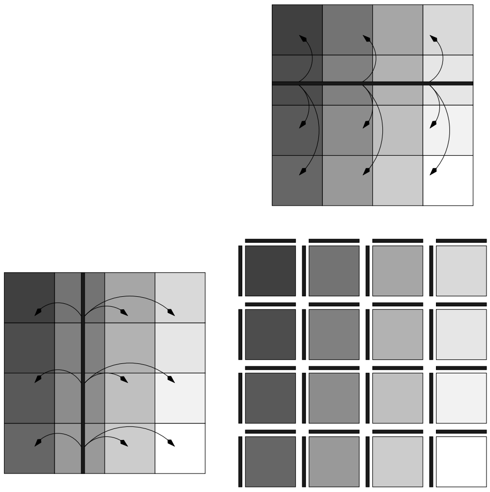

Fig. 1 – Distribution homogène contiguë pour une grille 4×4.

étape de calcul, mais cela ne nuit pas au parallélisme et permet un recouvrement potentiel du calcul et des communications.

Dans le cas où l'on dispose de $p.q$ processeurs identiques inter-connectés en grille comme sur la figure 1, chaque processeur est responsable de sous-matrices de taille $\frac{n}{p} \times \frac{n}{q}$ de A, B et C. L'algorithme 2 se déroule donc en n étapes et à l'étape k, la $k^{\text{ème}}$ colonne de A et la $k^{\text{ème}}$ ligne de B sont diffusées horizontalement (pour la colonne) et verticalement (pour la ligne). Chaque processeur reçoit donc un fragment de colonne de A et un fragment de ligne de B de tailles respectivement n/p et n/q et met à jour la partie de C dont il est responsable (voir Figure 1).

2.2 Mise en œuvre en MPI de l'algorithme par double diffusion

Copiez depuis l'archive`/home/vrehn/teaching/AlgoPar/tp2-src.tar.gz` contenant un canevas `matmult_template.c` pour le programme de produit matriciel, un `Makefile` contenant les commandes `make` et `make run` et les fichiers `hosts` pour la salle Europe et le cluster gdsdmi. Enfin, vous trouverez les scripts permettant de générer vos clefs SSH si nécessaire.

Pour exécuter votre programme, vous devrez utiliser `mpirun` et lui fournir le fichier contenant une liste d'hôtes ainsi que le nombre de processus à utiliser. Celui-ci peut être supérieur au nombre d'hôtes, dans ce cas, `mpirun` effectue une allocation circulaire. Par exemple :

```
mpirun -machinefile hosts -np 4 multmult_template 160
```

Une fois que le programme fonctionne, testez ses performances en calculant le facteur d'accélération. Vous pouvez tester vos programmes soit sur les machines de la salle Europe, soit sur le cluster gdsdmi. Vous pouvez utiliser les premières pour mettre au point votre programme et ensuite tester ses performances sur le cluster gdsdmi.