Communicateurs et produit matriciel en MPI

Résumé: Dans ce TD, nous allons mettre en œuvre un produit de matrice en MPI basé sur l'algorithme de double diffusion. Les diffusions se faisant sur certains processeurs uniquement, nous allons créer de nouveaux groupes de processus pour pouvoir utiliser la fonction `MPI_Bcast`.

1 Un mot d'histoire

Tout d'abord, il convient de rappeler quelques généralités sur ce qu'est MPI et sur ce que ce n'est pas. MPI n'est pas une bibliothèque de communication développée par une université ou une entreprise : c'est un standard qui est né au début des années 90 pour répondre à un besoin de clarification. En effet, à cette époque, la seule façon de faire du calcul parallèle efficace consistait à acheter une grosse machine parallèle propriétaire. Ces dernières étaient généralement livrées avec leur propre bibliothèque de communication qui n'était que rarement compatible avec celle de la machine précédente et jamais avec celle des concurrents. Il était donc très difficile de maintenir un programme à jour et un gros travail était nécessaire à chaque fois que l'on souhaitait changer de machine. Le standard MPI est donc né de la collaboration entre des universitaires et des industriels de tous domaines scientifiques. Cependant, si ce standard a permis de résoudre la majorité des problèmes que l'on pouvait avoir au moment de sa création, il souffre désormais d'un bon nombre de limitations et d'une inadéquation aux plateformes de calcul actuelles.

En effet, la mode actuelle en matière de calcul distribué et parallèle est à l'interconnection, via des liens à très haut débit, de grands centres de calculs disséminés à l'échelle d'un pays, d'un continent voire du monde. C'est ce que l'on appelle le *meta-computing*. Personne ne sait si on pourra un jour exploiter correctement de telle plateformes mais le jeu en vaut la chandelle.

Mais revenons à MPI. Un des gros problèmes auquel on se heurte dès que l'on essaie d'utiliser un programme MPI sur une plateforme de *meta-computing* est que les bibliothèques MPI d'un centre de calcul à l'autre ne sont pas forcément identiques et rien dans le standard n'oblige une bibliothèque à perdre en performance pour être compatible avec une bibliothèque concurrente. À cela s'ajoutent la difficulté de déploiement d'un tel programme, la nécessité de prendre en compte (autant au niveau algorithmique qu'au niveau système) l'hétérogénéité des processeurs et des réseaux, le coût énorme des synchronisations et donc des opérations bloquantes, la non-tolérance aux pannes,... Même s'il serait plaisant de pouvoir programmer et utiliser ces plateformes comme une simple station de travail, nous en somme très loin actuellement et leur complexité semble rendre cette entreprise un peu utopique.

Ce portrait de MPI et du calcul parallèle actuel peut sembler un peu pessimiste mais tous ces projets ont aussi donné naissance à de très bonnes choses. Même si MPI n'est plus adapté aux plateformes de calcul telle qu'elles sont envisagées actuellement, l'aventure MPI est quand même un succès étant donné qu'elle a permis à bon nombre de personnes non informaticiennes de développer des programmes efficaces pour les plateformes de calcul parallèle classiques et de collaborer grâce à des codes portables. L'émulation créée par ces projets ambitieux de *global-computing* permet à bon nombre de scientifiques non informaticiens de résoudre des problèmes qu'ils n'auraient jamais pu espérer résoudre il y a de cela quelques années. Enfin, même si tous ces projets semblent un peu fous et extrêmement ambitieux, il est indéniable que la communauté scientifique a un besoin toujours croissant de puissance de calcul et de communication et que nous ne sommes actuellement pas en mesure de répondre à leurs besoins.

Un nouveau standard MPI-2 a été mis en place il y a quelques années et résout certains des problèmes précédemment évoqués. Il existe cependant encore très peu de bibliothèques mettant en œuvre l'intégralité du standard MPI-2.

2 Introduction à l'utilisation de MPI

2.1 Initialisation et terminaison du programme

Un programme MPI commence en général par un appel de la fonction `MPI_Init` dont le prototypage est le suivant :

```
int MPI_Init(int *argc, char ***argv)
```

Cette fonction initialise les connections MPI en fonction des arguments passés à votre programme. C'est pourquoi avant de lire les arguments de votre programme, il convient de faire un appel à cette fonction.

Un programme MPI se termine généralement par l'appel de la fonction `MPI_Finalize`. Tous les processus doivent appeler ce programme avant leur terminaison. Cette opération est bloquante ne termine que lorsque toutes les opérations de communications en cours ou en attente sont terminées.

Il peut être utile de savoir combien de processus participent au calcul et quel numéro on a. Les fonctions `MPI_Comm_size` et `MPI_Comm_rank` dont le prototype est le suivant. Un `MPI_Comm` est un groupe de processus MPI. Dans un premier temps, on peut utiliser le groupe prédéfini `MPI_COMM_WORLD` qui regroupe l'intégralité des processus MPI participant au calcul.

```
int MPI_Comm_size ( MPI_Comm comm, int *size )
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

Si ces explications ne vous suffisent pas, vous pouvez évidemment consulter la documentation en ligne :

http://www-unix.mcs.anl.gov/mpi/www/

2.2 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.

Le opérations de communications collectivent 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 emetteur, `buffer` sert à recevoir les données alors que pour l'emetteur, 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.

3 Produit de matrice parallèle

3.1 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 à n :
2:   Pour j = 1 à n :
3:     Pour k = 1 à n :
4:       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.

```
MatMult(A, B, C)
1: Pour k = 1 à n :
2:   Pour tout i en parallèle :
3:     Pour tout j en parallèle :
4:       C_ij ← C_ij + A_ik B_kj
```

Algorithme 2: Algorithme parallèle du produit de matrice

Il est donc préférable de permuter les boucles pour arriver à l'Algorithme 2 et d'effectuer alors la boucle externe (ligne 1) séquentiellement 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 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 interconnectés en grille (voir 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 $k^{ème}$ colonne de A et la $k^{ème}$ 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).

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

Vous trouverez dans le répertoire `/homes/alegrand/MIM2/td4/` un canevas pour le programme de produit matriciel (`matmult_template.c`). Comme d'habitude, le programme est commenté et il n'y a qu'à remplir les trous. Il y a un joli makefile qui compile le tout et tire parti du réseau

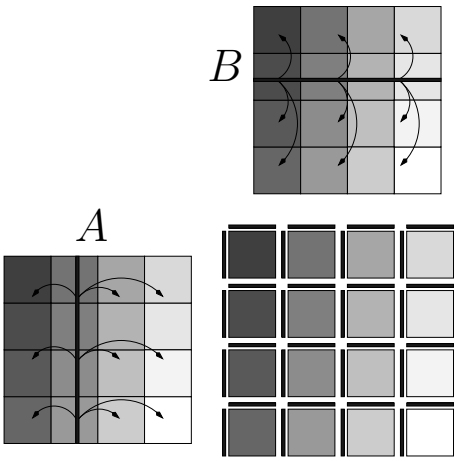

FIG. 1 – Distribution homogène contiguë pour une grille 4×4

MyriNet qui interconnecte les différents nœuds de la grappe. Pour exécuter votre programme sur 4 processeurs, il faut passer par le gestionnaire de batch de la grappe en utilisant la commande suivante :

```
qrsh -pe para 4 mpirun.poi -np 4 $HOME/MIM2/td4/matmult_template 160
```

Vous êtes alors assurés que ces 4 nœuds vous sont réservés pour la durée de votre calcul et que vos communications n'interfèrent pas avec celles des autres.

Une fois que le programme fonctionne, augmentez la granularité des communications, essayez de recouvrir les calculs et les communications. Comment faire pour évitez des copies inutiles lors des envois des fragment de lignes et de colonnes ?