Département de génie logiciel et des TI

LOG645-Laboratoire #01

Introduction au traitement parallèle avec MPI

Étudiants	Arker, Filipe Phieu, Howard
Cours	LOG645
Session	Été 2020
Groupe	01
Sous-groupe	Groupe F
Laboratoire	Introduction au traitement parallèle avec MPI
Chargé de laboratoire	Jordy Ajanohoun
Date	7 juin 2020

Introduction

Ce laboratoire à le but d'introduire au traitement parallèle et la librairie MPI. Pour ce faire, le laboratoire présente deux problèmes qui utilisent le traitement séquentiel et parallèle pour les compléter. Pour une matrice 8x8, chaque champ de celle-ci est initialisée avec une valeur initiale passée en paramètre.

Pour le premier problème, la matrice finale est calculée en fonction de la matrice précédente, la ligne et la colonne multipliés par l'itération k.
Le deuxième problème doit prendre en compte de la valeur de la colonne précédente, en plus de la matrice précédente, le numéro de la ligne et l'itération k.

Pour résoudre ces deux problèmes, on utilisera les traitements séquentiels et parallèles. Le traitement séquentiel utilisé les calculs uns à la suite des autres. Le parallèle, tant qu'à lui, vise à traiter les données en concurrence.

Analyse

Q1: Pour une exécution du programme donnée ("prog 1 5 8"), combien de fois la fonction de calcul est-elle exécutée? Donner la moyenne par processeurs.

Les paramètres sont les suivants: Problème 1, valeur initiale 5 et 8 itérations. Celle qui affecte le nombre d'opérations est le nombre d'itération, qui est 8.

Puisque le calcul est exécuté dans une matrice de 8x8, il passe par tous les champs de celle-ci. Il y aura alors au total 8x8x8=512 opérations. Si on considère que tous les processus ont un nombre égal de calculs, et qu'on a 64 processeurs, alors chaque processeur aura: 512 / 64 = 8 calculs par processeur.

On peut alors conclure que la fonction de calcul sera alors fait 512 fois, dont 8 par processeur en moyenne.

Q2: Quels sont les vecteurs de dépendance des cellules (excluant celles aux bordures de la matrice qui ont des dépendances différentes)?

Problème 1
Les vecteurs de dépendance des cellules sont celles que le calcul fait appel pour obtenir le résultat recherché. Pour ce problème, on à la dépendance au niveau du champ précédent et la dépendance envers l'itération précédente. Pour le premier problème, le calcul se fait entièrement sur le champ *i* et *j* de la matrice, en utilisant la valeur calculée dans l'itération précédente de ce même champ. On a alors M[i, j] = M[i,j] + (i + j) * iteration.
Le vecteur de dépendance des cellules dans le premier problème est alors [0,0].

Problème 2

Dans ce problème, on calcule encore avec les mêmes variables que dans le problème un mais on ajoute de la complexité: la première colonne de la matrice à un calcul différent des autres puis pour les autres on ajoute la valeur de la colonne précédente qu'on a calculé, ce qui est une dépendance de plus. Le calcul est le suivant:

matrice[i][j] = matrice[i][j] + matrice[i][j - 1] * k

Ainsi, le vecteur de dépendance est la suivante [0, -1] où 0 est la rangée et le -1 est la colonne.

Q3: Quels sont les impacts de ces dépendances sur la communication et comment cela affecte-t-il votre agglomération? Utiliser un schéma si nécessaire.

Problème 1

Pour le premier problème, puisque la lecture se fait sur la donnée elle-même seulement, il y a pas de vecteur de dépendance. Ainsi, il y a pas d'impact sur la communication ni l'agglomération.

Problème 2

Pour le deuxième problème, la dépendance est liée vers la colonne précédente [0, -1]. Ainsi, si cette donnée n'est pas à jour, le calcul sera erroné.

Q4: Combien de messages les processeurs vont-ils échanger au total (pour chaque problème)?

Problème 1

Pour le premier problème, le premier message qui est échangé est MPI_Scatterv, envoyé par le processeur qui possède le rang 0, à tous les autres processeurs, dans le but de distribuer les données de la matrice initiale.

Après les calculs de la matrice, les processeurs s'échangent les messages une dernière fois avec le MPI_Gatherv par le processeur qui possède le rang 0 vers la fin. Cette opération combine les matrices travaillés par chacun des processeurs et les combinent pour donner le résultat final.

Ainsi, on a deux messages échangés par les processeurs: MPI_Scatterv et MPI_Gatherv.

Problème 2

Tout comme le premier problème, il utilise une structure de communication similaire avec MPI_Scatterv et MPI_Gatherv au début et à la fin.

Cependant, comme il y a 1 dépendance de flot dans les calculs, nous devrons utiliser MPI_Recv et MPI_Send. Pour chaque processeur qui calcule une sous-matrice, il envoie au processeur suivant. Si nous considérons notre conception où nous utilisons 16 processeurs, et que chaque processeur appellerait MPI_Send et MPI_Recv (2 échanges) à l'exception du processeur final qui n'appellera pas MPI_Send, alors le nombre total d'échanges dans la phase de calcul serait de 2 * (16) - 1 = 31 échanges.

Au total, le deuxième problème a 33 messages échangés.

Q5: Quelle est l'efficacité de votre programme parallèle et expliquez pourquoi elle n'est pas de 100%?

En premier lieu, on défini l'efficacité du programme parallèle avec le calcul suivant:

$$Efficacité = \frac{Temps\ Séquentiel / Temps\ Parallèle}{Nombre\ de\ processeurs}$$

En itérant les deux problèmes de 1 à 99 itérations, nous avons mesuré le temps moyen suivant.

Problème 1

Séquentiel:	3.379577s
Parallèle:	0.651164s
Efficacité parallèle calculée:	32,44%

Problème 2

Séquentiel:	3.376074s
Parallèle:	0.870514s
Efficacité parallèle calculée:	24.24%

La raison pour laquelle notre efficacité n'est jamais à 100% est qu'une grande partie de notre programme est consacrée à la communication. Par contre, on voit que le problème 1 est plus efficace que 2 simplement parce qu'il y a moins d'échanges de messages.

Conception de la version Séquentielle

La version séquentielle de cette solution boucle à travers chacun des champs de la matrice pour faire le calcul. Voici le déroulement normal de l'application. Le programme doit débuter en passant trois paramètres en entrée: le numéro du problème, la valeur initiale pour les champs de la matrice et le nombre d'itérations. On initialise les variables utilisées pour les calculs et on instancie la matrice avec ses données initiales. Ensuite, on récupère le timestamp initial avant de démarrer le calcul. Rendu à cette étape, soit le problème 1 ou 2 est exécuté dépendamment de la valeur passée dans les paramètres.

Pour le problème 1, le programme passe par tous les champs de la matrice X nombres de fois chacune, X étant l'itération passée en paramètre à l'exécution du programme. Pour chaque champ, on fait le calcul

$$matrice[i][j] = matrice[i][j] + (i + j) * k.$$

Pour le problème 2, le programme commence par boucler encore à travers l'itération pour chacun des champs de la matrice mais il commence par faire un calcul pour tous les champs dans la première colonne, en utilisant la formule:

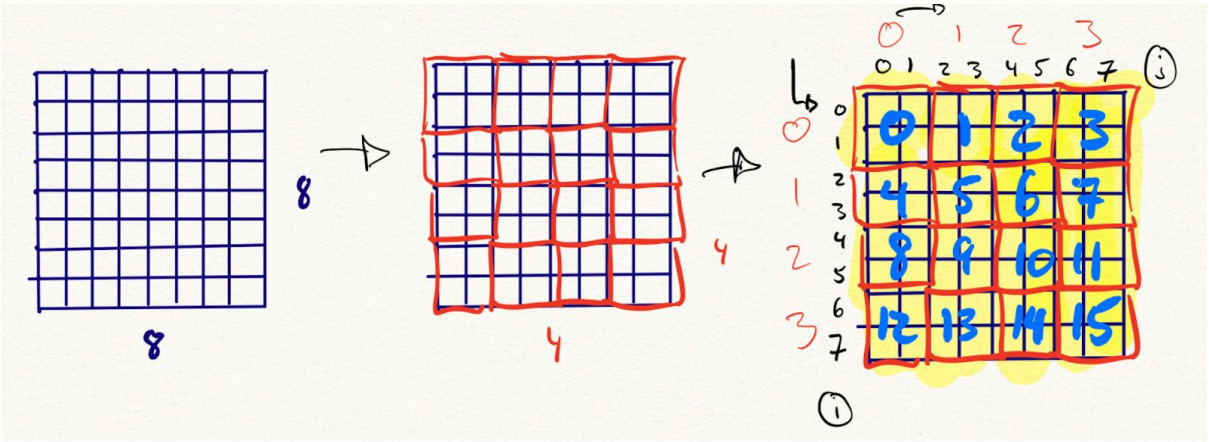
$$matrice[i][0] = matrice[i][0] + (i * k).$$

Ensuite, le programme couvre le reste de la matrice avec le calcul suivant:

$$matrice[i][j] = matrice[i][j] + matrice[i][j - 1] * k.$$

Après avoir résolu le problème 1 ou 2, on obtient le nouveau temps puis ensuite on affiche le résultat de la matrice, le qui a pris pour calculer la matrice et finalement l'application libère la matrice de la mémoire.

Conception de la version Parallèle



La version parallèle à le but de diviser les tâches de calcul entre processeurs dans le but d'accélérer le processus de calcul de la matrice. Dans le contexte de cet exercice, jusqu'à 64 processeurs peuvent être utilisés. En appliquant la méthode PCAR, on explique comment cette version a été conceptualisée.

Partitionnement:

Le partitionnement est le même pour les problèmes 1 et 2.

Étant donné que nous avons une matrice de 8x8 (que nous désignerons comme notre matrice globale), cela signifie qu'il y a 64 cellules. Nous avons décidé de concevoir notre programme pour utiliser 16 processeurs et avons donc décidé de créer une partition processeur de 4x4 (appelée "CPUGrid" dans notre programme). Notre partition de processeur aura également son propre système de coordonnées (défini comme une structure dans notre programme, où les coordonnées i, j sont de 0 à 3).

Ce partitionnement permettrait une distribution égale de 64 cellules sur 16 processeurs.

Communication

Pour la communication, nous utiliserons MPI_Scatterv et MPI_Gatherv. Pour utiliser ces appels, il est nécessaire de créer un MPI_Datatype personnalisé car nous voulons conserver notre matrice 2D. Dans notre programme, notre fonction createMatrixDataType () utilise MPI_Type_create_subarray pour créer un sous-tableau à partir de notre matrice globale et se redimensionne avec MPI_Type_create_resized, puis valide le sous-type à l'aide de MPI_Type_commit.

Dans le premier problème, MPI_Scatterv prend notre matrice globale et la diffuse (sous forme de sous-matrice) dans 16 processeurs de rang de processeur 0. Lorsque tous les calculs sont

effectués, toutes les sous-matrices sont rassemblées à l'aide de MPI_Gatherv à nouveau de rang 0.

Le deuxième problème utilise le même modèle de communication global que le premier lors du démarrage et de la finalisation, mais en raison de sa dépendance de flot, il y aura une intercommunication entre les processeurs localement en utilisant MPI Send / Recv. La communication elle-même est structurée comme un processeur transmettrait ses valeurs calculées au processeur suivant. Il est statique du fait que ses partenaires de communication ne changent pas dans le temps et sont synchrones car le transfert de données est transmis de manière coordonnée d'une sous-matrice à l'autre.

Agglomération

Pour l'agglomération, considérons que nous utilisons 16 processeurs sur une matrice de 64 cellules. Cela signifie que chaque processeur lui-même contiendrait 4 cellules où les calculs seraient effectués.

Pour notre premier problème, car il n'y a pas de dépendance, chaque processeur peut traiter ses propres calculs, puis les résultats sont ensuite mappés sur la matrice globale.

Dans notre deuxième problème, parce qu'il existe une dépendance, chaque fois qu'un processeur termine son calcul, une communication entre les processeurs se produit. Cela signifie que l'augmentation du nombre de processeurs pourrait en fait augmenter les frais de communication, nous avons donc décidé de le maintenir à 16 processeurs.

Répartition

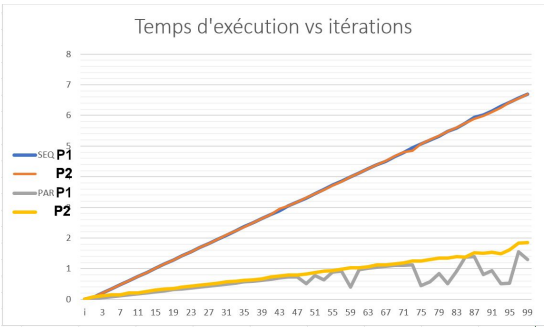
Pour la répartition, nous nous sommes assurés que pour tirer le meilleur parti des calculs dans un seul processeur, nous avons décidé que chaque processeur aurait une sous-matrice 2x2.

Pour le premier problème, les 16 processeurs pourraient tous effectuer leur tâche de calcul en même temps, puis compiler dans la matrice globale.

Pour le deuxième problème, cela lui permettrait de recevoir des données sur la première colonne, d'exécuter les calculs et de les enregistrer sur la deuxième colonne, puis de les envoyer au processeur suivant.

Cette conception permettrait d'effectuer 2 tâches de calcul sur chaque processeur à la fois. Avec une grille de traitement 4x4, cela signifie que 8 tâches de calcul sont exécutées, deux fois sur chaque processeur. En raison de la dépendance du problème 2, si nous augmentons le nombre de processeurs, nous réduisons le nombre de tâches pouvant être exécutées dans un seul processeur et augmentons la communication, donc avoir plus de 16 processeurs ne serait pas bénéfique.

Discussion



	SEQ		PAR		Efficacité		Accélération	
i	P1	P2	P1	P2	P1	P2	P1	P2
1	0.067648	0.067648	0.019945	0.09191	21.20%	4.60%	3.39	0.74
3	0.202838	0.202771	0.05193	0.111937	24.41%	11.32%	3.91	1.81
5	0.338091	0.336339	0.079935	0.147947	26.43%	14.21%	4.23	2.27
7	0.473205	0.473378	0.119914	0.151799	24.66%	19.49%	3.95	3.12
9	0.606936	0.608582	0.147924	0.207931	25.64%	18.29%	4.10	2.93
11	0.742982	0.739921	0.178752	0.217006	25.98%	21.31%	4.16	3.41
13	0.877248	0.87413	0.209008	0.255943	26.23%	21.35%	4.20	3.42
15	1.014329	1.008541	0.244951	0.30396	25.88%	20.74%	4.14	3.32
17	1.149371	1.149559	0.27296	0.343941	26.32%	20.89%	4.21	3.34
19	1.283286	1.284854	0.315975	0.358773	25.38%	22.38%	4.06	3.58
21	1.419984	1.420055	0.335912	0.399961	26.42%	22.19%	4.23	3.55
23	1.551946	1.554721	0.371942	0.43576	26.08%	22.30%	4.17	3.57
25	1.686878	1.690585	0.405116	0.463889	26.02%	22.78%	4.16	3.64
27	1.822174	1.825835	0.435902	0.491924	26.13%	23.20%	4.18	3.71
29	1.955477	1.96222	0.46395	0.523925	26.34%	23.41%	4.21	3.75

Amélioration du code

Pour améliorer notre code, nous aurions pu convertir notre matrice 2D en un tableau 1D singulier. Bien que cela puisse augmenter la communication entre les processeurs et nécessiter une manipulation de tableau dans le positionnement, cela pourrait simplifier notre code dans la mesure où nous n'aurions pas besoin de créer un nouveau type de données MPI et de le rendre moins complexe / abstrait à comprendre. Éviter les types de données MPI personnalisés peut également économiser de la mémoire.

Notre code est actuellement conçu pour être exécuté sur 16 processeurs et une matrice 8x8, mais nous pouvons également le concevoir de sorte qu'il puisse s'adapter dynamiquement au nombre de processeurs et à la taille de la matrice. Cela nous permettrait d'exécuter d'autres tests pour observer les changements de performances dans diverses configurations.

Amélioration au laboratoire

Pour améliorer le laboratoire, il pourrait être plus facile de le décomposer en petites sections où nous sommes explicitement chargés d'implémenter des fonctions MPI spécifiques afin que nous puissions mieux comprendre comment fonctionne OpenMPI. Il existe de nombreuses solutions possibles pour résoudre des problèmes parallèles en utilisant MPI_Send / Recv, Scatter / Gather, Scatterv / Gatherv et de nombreuses options qu'il peut être écrasant de comprendre au début.

De plus, étant donné que le langage C était un nouveau langage pour nous, il était un peu difficile d'implémenter certaines fonctions car il nécessitait une compréhension de la manipulation du pointeur. Bien que ce fut une expérience agréable d'apprendre un nouveau langage, il aurait été préférable d'utiliser un langage de programmation plus familier tel que Java afin que nous puissions explorer plus de fonctionnalités d'OpenMPI.

Conclusion

En observant nos résultats mesurés, nous avons pu constater qu'en parallélisant notre programme, nous avons vu une accélération significative de l'accélération. En regardant notre graphique des temps d'exécution vs itérations, le temps augmente linéairement dans notre programme séquentiel dans les deux problèmes 1 et 2. En parallèle, nous avons observé que la pente des problèmes 1 et 2 est nettement moins inclinée. Le problème 1 qui ne nécessite qu'un Scatterv / Gatherv montre des temps de finition plus irréguliers car les 16 processeurs sont indépendants. Dans le problème 2, nous remarquons que les résultats sont plus stables, en raison du fait qu'il existe une dépendance où un processeur doit communiquer avec un autre.

En termes d'efficacité, on peut voir que le problème 1 est meilleur car il y a moins de communication entre les processeurs. L'augmentation du nombre de processeurs entraînerait une baisse de l'efficacité, car cela nécessiterait une gestion plus complexe du programme et des communications entre les processeurs.