Projet HPC

Fangzhou YE

03/06/2020

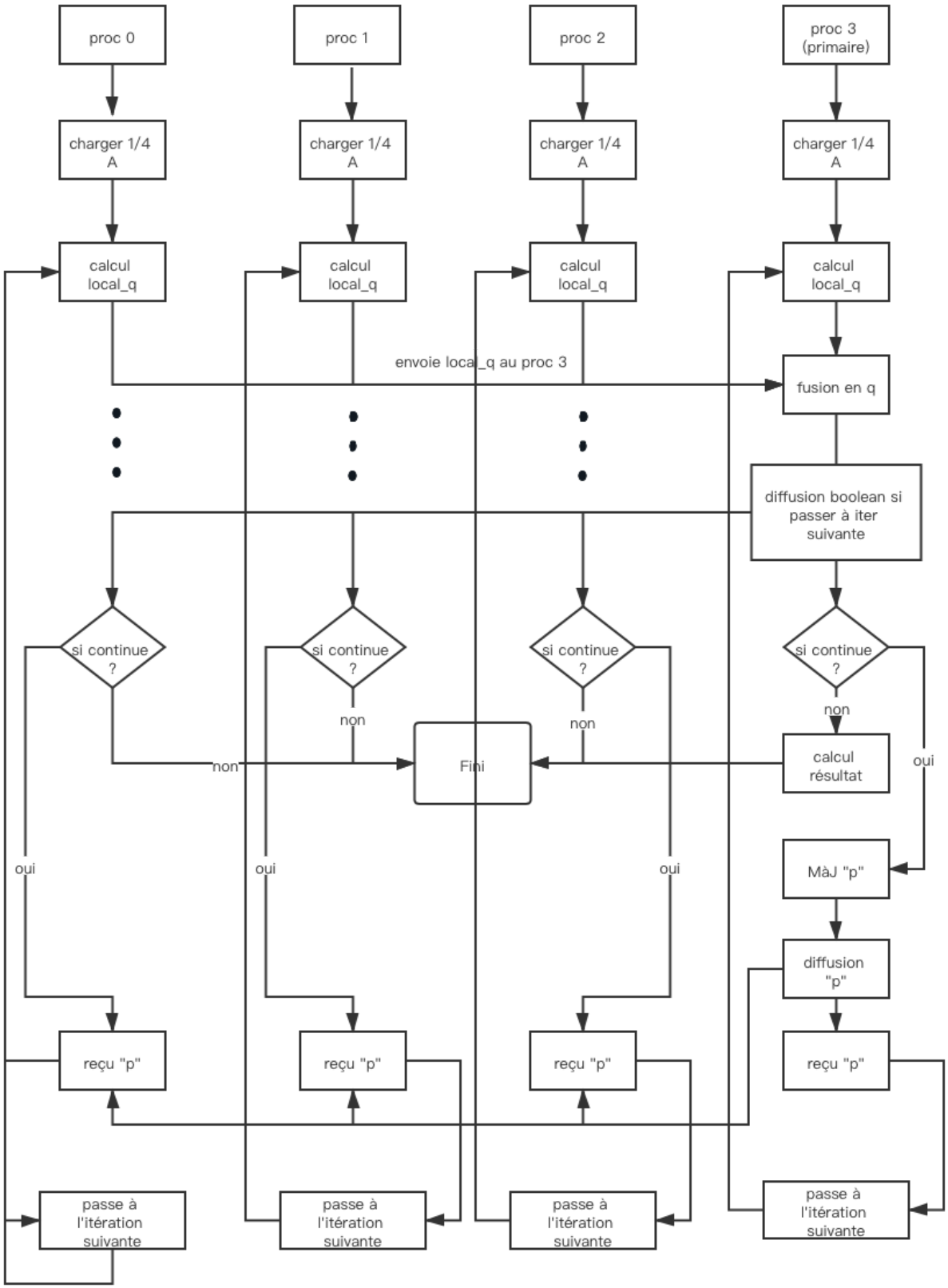
Ordinateur personnel utilisé pour le test:
MacBook Pro 2018, 2.6 GHz 6-Core Intel Core i7, 16G memory

I. Introduction

Dans ce projet, on envisage de paralléliser un programme séquentiel de méthode du gradient conjugué. C'est un algorithme itératif permettant de résoudre le système linéaire creux de type Ax = b, où A est une matrice creuse de taille N*N avec N très grande, b est un vecteur de taille N. x est le vecteur du résultat. La matrice creuse est représentée sous le format CSR dans les programmes. Au seine de l'algorithme, la multiplication de A et un vecteur consomme énormément du temps. C'est ici qu'on va appliquer la notion de parallélisme afin d'accélérer le calcul. On a proposé une optimization avec MPI et une optimization avec MPI+OpenMP.

II. MPI

Dans le programme séquentiel, l'étape *q* ← *A*p* s'agit d'une multiplication de matrice creuse et un vecteur. Étant donné *nb_proc* processeurs disponibles au calcul. D'abord, on sépare la matrice A en *nb_proc* parties. Chacun est de taille (N/nb_proc)*N, le dernier segment peut avoir un hauteur inférieur aux autres si N n'est pas divisible par *nb_proc*. Puis, chaque processus va charger la matrice local_A qui est un segment de la matrice A correspondant à son rang. Pour ce faire, On a implémenté une méthode **split_csr(A, nb_proc, i)** qui permet d'obtenir le i ème segment de matrice A parmis *nb_proc* segments en passant son rang et le nombre de processeur. Ensuites, chaque processus calcule local_q ← local_A * p qui est un résultat partiel de q. Chaque processus va envoyer son local_q au processus primaire (ici, celui de rang *nb_proc*-1). Les résultats partiels de local_q vont en suites être fusionnés en résultat complet de q. Cette étape a été réalisé avec la primitive MPI_Gatherv car le longueur du buffers envoyé au processus primaire peut être différent. Comme c'est un algorithme itératif. Chaque processus doit être synchronisé dans la même itération. Pour ce faire, on a imbriqué les processus non-primaire dans une boucle infinie. Ainsi, on a introduit une variable booléen qui décide s'ils vont entrer dans la prochaine itération. Cette valeur est décidée par le processus primaire en vérifiant si la précision de calcul soit atteinte. Cette variable est diffusée par le processus primaire en utilisant la primitive MPI_Bcast. Si la précision n'est pas encore atteinte, la valeur p est mise à jour et diffusée par le processus primaire pour que les autres puissent entrer dans la prochaine itération. Voici un diagramme qui illustre le fonctionnement du programme optimisé par MPI pur avec 4 processeurs disponibles.


proc 0
proc 1
proc 2
proc 3 (primaire)
charger 1/4 A
charger 1/4 A
charger 1/4 A
charger 1/4 A
calcul local_q
calcul local_q
calcul local_q
calcul local_q
envoie local_q au proc 3
fusion en q
diffusion boolean si passer à iter suivante
si continue ?
si continue ?
si continue ?
si continue ?
non
non
non
Fini
calcul résultat
oui
oui
oui
oui
MàJ "p"
diffusion "p"
reçu "p"
reçu "p"
reçu "p"
reçu "p"
passe à l'itération suivante
passe à l'itération suivante
passe à l'itération suivante
passe à l'itération suivante

III. MPI+OMP

Basé sur le méthode parallélisé avec MPI pur, on a introduit une optimisation multi-thread dans la fonction de multiplication de matrice-vecteur (sp_gemv). Cependant, on doit vérifier la taille de la matrice. L'optimisation de multi-thread n'effectue qu'en matrice de taille grande et/ou dont le nombre de valeur non-nulle est grand. Pour cela, on a fixé un threshold de taille 15000 et le nombre de valeur non-nulle de 2000000.

IV. Test

Pour comparer les comportement de deux optimisations, on a d'abord lancé le test sur le méthode MPI avec différent nombre de processeurs disponibles et le méthode séquentiel. Les matrices testées sont de taille différentes. Voici une comparaison du temps d'exécution. En comparant la performance de hood et nd24k, nd24k a une taille plus petite et admet le nombre de valeur non-nul plus grand que hood. On a observé une tendance qu'un plus grand nombre de valeur non-nulle admet un taux d'optimisation plus grand.

Matrice	Nb_proc = 2	Nb_proc = 3	Nb_proc = 4	seq
cfd1	4.7	4.4	4.7	6.8
cfd2	35	34.7	34.9	45.6
hood	73.8	66.4	64.9	106.6
nd24k	316.9	252.9	230.7	509.1

Voici une comparaison du méthode optimisé purement par MPI et celui de MPI+OpenMP (tout avec 4 coeurs, nb_thread = 4 pour OpenMP). On a observé une tendance que le taux d'optimisation est plus grand pour la matrice dont le nombre de valeur non-nulle est plus grand.

Matrice	MPI	MPI+OpenMP
cfd1	4.7	4.7
cfd2	34.9	35.3
hood	64.9	65
nd24k	230.7	217.3

V. Conclusion

D’après nos tests, on a observé qu’en général, la taille de matrice et le nombre de valeur non-nulle sont deux facteurs principaux de la performance d’optimisation. Plus de coeurs participe au calcul, plus le méthode est accéléré sur le calcul en taille plus grande et/ou de nombre de valeur non-nulle plus grand. Cela vérifie notre intuition. Cependant, sur le calcul de taille plus petite et/ou de nombre de valeur non-nulle plus petite, l'optimisation en multi-coeurs n’a aucun sens même plus pire. Car la communication entre les processus prend du temps. Pour l'optimisation multi-thread avec OpenMP. La génération de thread aussi prend du temps. Donc, il est inutile de faire optimisation de thread pour la matrice segmentée dont le nombre de valeur non-nulle est plus petite et/ou dont la taille est plus petite.