Parallélisation de l'Algorithme du Gradient Conjugué pour des matrices creuses

Rapport final

Projet HPC

Achille Baucher &Najwa Moursli
Polytech Sorbonne
Spécialité
Mathématiques Appliquées et Informatique Numérique
Année 4
2019 – 2020

Parallélisation de l'Algorithme du Gradient Conjugué pour des matrices creuses
Rapport final
Projet HPC
Version n°1


POLYTECH
SORBONNE
SCIENCES
SORBONNE
UNIVERSITÉ

Table des matières

Table des figures

1. Introduction

ABSTRACT

La méthode du gradient conjugué est généralement mise en oeuvre sous la forme d'un algorithme itératif, applicable aux systèmes trop larges pour être traités par une implémentation directe. Il est donc très judicieux d'appliquer les techniques de parallélisation de mémoire partagée et distribuée. Pour ce faire, nous utilisons **MPI** (mémoire distribuée) et **OpenMP** (mémoire partagée).
L'objectif de ce projet serait de trouver une stratégie de calcul haute performance permettant une accélération significative du temps de calcul par rapport au code séquentiel à la fois en utilisant seulement **MPI** puis un code hybride **MPI+OpenMP**.

1.1 Input/Output

La matrice A et le vecteur b correspondant au système Ax=b sont definis et sauvegardés dans un document grâce au script python *Runner.py*. Ensuite, ce document est lu dans notre base de données grâce à ifstream flow. Ce processus de lecture se fait grâce au rang 0 et se stocke comme une variable dans notre code. L'écriture du vecteur solution a également été faite par *my_rank==0* après avoir effectué une verification avec le calcul d'un vecteur d'erreur.
Le vecteur solution est relativement petit par rapport à la matrice, de sorte qu'une sortie parallèle n'est pas nécessaire. En outre, chaque fois que nous utilisons notre implémentation, nous vérifions si les résultats de notre algorithme CG sont corrects. Pour ce faire, nous calculons le résultat de A*x où A est la matrice donnée en entrée et x est la solution que nous avons trouvée. Ce résultat doit être égal au vecteur b donné en entrée dans l'algorithme CG.
Ainsi, pour chaque élément du vecteur b, on soustrait la valeur réelle à celle trouvée et on vérifie si la valeur absolue de différence, qui correspond à l'erreur, est inférieure à une tolérance choisie. Dans notre cas, la tolérance a été fixée à 1e-8. En outre, les solutions trouvées lors du changement du numéro de rang et de threads différents ont également été vérifiées manuellement pour s'assurer qu'elles sont presque identiques à celles trouvées avec le cas séquentiel de départ.

2. Implémentation du Parallélisme

2.1 MPI

2.1.1 Distribition de la mémoire

Après étude du code séquentiel, nous avons fait un tri entre les informations nécessitant d'être partagées et les autres.

2.1.1.1 Distribuables

Nous avons commencé par identifier les vecteurs qui pouvaient ne s'utiliser qu'en local :
- x : On ne s'occupe que de le remplir et il n'est pas utilisé dans le calcul.
- r : Utilisé pour le calcul de z, mais le calcul de z_i ne demande que r_i. Utilisé pour le calcul de β mais seulement pour le produit scalaire. Utilisé pour le calcul de l'erreur, mais seulement pour une somme.
- z : Utilisé pour le calcul de p, mais le calcul de p_i ne demande que z_i. Utilisé pour le calcul de β , mais seulement pour le produit scalaire.
- q : Utilisé pour le calcul de pr, mais le calcul de p_i ne demande que q_i. Utilisé pour le calcul de α , mais seulement pour le produit scalaire. Utilisé dans *sp_gemv* mais seulement pour être remplie.
- d : N'est jamais modifié.

Pour que ces éléments soient effectivement distribuables, il faut aussi distribuer les produits scalaires (et donc les normes) selon la même mémoire locale, ce qui est possible : si chaque noeud p contient une partie x_{I_p} et y_{I_p} des vecteurs x et y, alors on peut effecter le produit scalaire sur chacune de ces parties pour les sommer ensuite pour obtenir le produit scalaire de x et y. Il faut de plus que la fonction *sp_gemv* puisse être éxécutée localement, ce qui est possible car on peut choisir de remplir indépendament les lignes de q, à condition que p soit partagé.

On en déduit que les vecteurs x,r,z et q peuvent être locaux après leur initialisation, avec x à rassembler à la fin, et que d n'en a même pas besoin.

2.1.1.2 À rassembler

Il restait donc seulement quelques informations nécessitant d'être partagées régulièrement :

- p : Utilisé pour le calculs de x, mais le calcul de x_i ne demande que x_i. Utilisé pour le calcul de β mais seulement pour le produit scalaire. En revanche, la fonction *sp_gemv* demande l'intégralité du vecteur p.
- L'erreur, rz, α et β doivent être identiques et nécessitent l'intégralité de certains vecteurs pour être calculées.

On en déduit que seul le vecteur p est à rassembler à chaque itération, avant chaque utilisation de *sp_gemv*, comme expliqué dans la partie 2.1.3.2. Les autres valeurs scalaires seront aussi à mettre régulièrement en commun, comme expliqué dans la partie 2.1.3.3.

2.1.1.3 Distribution des indices

La première étape consiste donc à segmenter la mémoire en plusieurs parties locales à peu près de même taille. Les vecteurs de la version séquentielle étant tous de taille n, on peut choisir des bornes d'indices $start\_pos_p$ et end_pos_p compris entre 0 et n pour chaque noeud p pour une décomposition par bloc de lignes. Nous avons utilisé la segmentation suivante, la plus intuitive :

$$start\_pos_p = \frac{rank*n}{np} \tag{2.1}$$

$$end\_pos_p = \frac{(rank+1)*n}{np} \tag{2.2}$$

$$n\_local_p = end\_pos_p - start\_pos_p \tag{2.3}$$

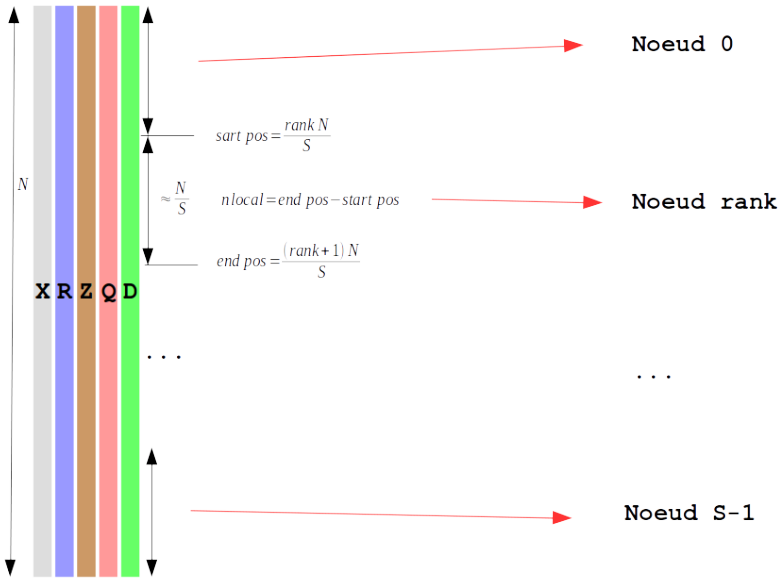

Figure 2.1 – Distribution des N données dans les différents noeuds

Avec *rank* le rang du noeud p et np le nombre de processeurs utilisés. Ce sont bien sûr des divisions euclidiennes. Nous avons stocké ces informations dans deux tableau, *recvcounts* pour les tailles et

displs pour les positions de départ, afin d'éviter d'avoir à les recalculer et de les mettre directement en argument des fonctions qui les utilisent.

2.1.2 Équilibrage des charges

Avec cette approche parallèle du calcul, chaque rang et chaque thread a une quantité approximativement égale de données et d'opérations à effectuer et l'équilibrage de la charge est réalisée naturellement. En effet, la quantité de tâches à effectuer pour un noeud dans ce programme ne dépend que du nombre de données attribuées, aucun déséquilibre ne peut donc être créé. Pour *dot*, on a choisi d'utiliser *MPI_Allreduce* qui permet d'assurer l'équilibrage des charges en faisant en sorte que tous les rangs effectuent la réduction. L'autre option consisterait à utiliser le *MPI_Reduce* pour réduire au rang 0, puis faire en sorte que le rang 0 diffuse le résultat scalaire à tous les rangs. Cela augmenterait de manière disproportionnée la charge du Rang 0 et entraînerait également un code inutilement plus compliqué. La seule exception à cette équilibre est que le rang 0 effectue la distribution dans le *MPI_Allgatherv*du vecteur p, mais c'est une opération nécessaire et il serait tout aussi coûteux de paralléliser cette distribution. .

2.1.3 Calcul

2.1.3.1 Inititalisation

Par rapport au code séquentiel, on retire les vecteurs z, q et r pour les remplacer par leur version locale. On ajoute la version locale de p, qui doit cependant aussi garder une version entière comme chaque noeud en a besoin dans son intégralité dans la fonction *sp_gemv*. On laisse d comme avant car il n'est pas modifié. On initialise les vecteurs de la même manière, et on calcule l'erreur et rz comme expliqué dans la partie 2.1.3.3.

2.1.3.2 Fonction principale

On distribue dans la fonction principale cg_solve le calcul des vecteurs x,r,z et p qui ne nécessite que leur version locale. À chaque fois, on fait varier i de *start_pos* jusqu'à *end_pos*, et on complète chaque vecteur local à la position [$i-$*start_pos*].

Il faut aussi penser à rassembler le vecteur p à partir de ses versions locales chaque fin de boucle, afin qu'il soit à jour lors de son entrée dans la fonction *sp_gemv*. On utilise pour cela la fonction *MPI_Allgatherv* au lieu de *MPI_Allgather* car les tailles des vecteurs sont inégales, à cause de n qui a la fâcheuse habitude de ne pas être divisible par le nombre de processeurs np. On passe en argument les tailles *recvcounts* et les positions *displs* définies dans la partie 2.1.1.3.

```
for (int i = start_pos; i < end_pos; i++)        // r <-- r - alpha*q
    r_local[i-start_pos] = r_local[i-start_pos] - alpha * q_local[i-start_pos];
```

Listing 1 – Parallélisation d'une des boucles for de la fonction *cg_solve*

À la fin du calcul, on rassemble de la même manière que p le vecteur x à partir des versions locales.

2.1.3.3 Les fonction dot et norm

Pour paralléliser le produit scalaire de 2 vecteurs dans la fonction *dot*, on peut simplement laisser chaque noeud s'occuper indépendament d'une section choisie des vecteurs et sommer ensuite les résultats obtenus. On comence donc par calculer le produit *dot* de manière locale c'est à dire avec la taille *n_local* sur un vecteur local et dans une sous-somme locale. Puis puis on rassemble les produits obteus en sommant les sommant avec la fonction *MPI_Allreduce* et l'option *MPI_SUM*. Nous avons utilisé cette méthode pour le calcul de du produit de *rz* et celui *pq*. C'est exactement la même chose pour la fonction *norm*, qui ne fait qu'utiliser *dot* sur le même vecteur pour ensuite calculer la racine du résultat obtenu. Nous l'utilisons une seule fois dans la boucle principale, pour le calcul de l'erreur *err_actuelle*.

```
double pq_local = dot(n_local, p_local,q_local);
double pq = 0.0;
MPI_Allreduce( &pq_local,&pq,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
```

Listing 2 – Réduction sur le produit de p et de q

Cout de communication : De l'ordre du nombre de processus **np**, temps de calcul en log *np*

2.1.3.4 La fonction sp_gemv

Dans cette fonction, on choisit de paralléliser la première boucle puisque la deuxième nécessiterait d'éffectuer une dispersion des données à chaque itération par *Broadcast*. Nous avons décidé d'implémenter une version locale de cette fonction qui complète le vecteur y avec des indices désirés de taille et de position initiale passés en argument. Cela permet de ne pas s'encombrer d'un coût de communication inutile.

```
void sp_gemv_local(const struct csr_matrix_t *A, const double *x,
    double *y_local, int n_local, int pos_local){
    int *Ap = A->Ap; int *Aj = A->Aj; double *Ax = A->Ax;

// !# On ne s'occupe que des indices locaux
    for (int i = pos_local; i < pos_local + n_local; i++) {
            y_local[i-pos_local] = 0;
        for (int u = Ap[i]; u < Ap[i + 1]; u++) {
                int j = Aj[u];
                double A_ij = Ax[u];
                y_local[i-pos_local] += A_ij * x[j];
        }
    }
}
```

Listing 3 – Fonction locale de *sp_gemv*

2.1.3.5 Schéma résumé

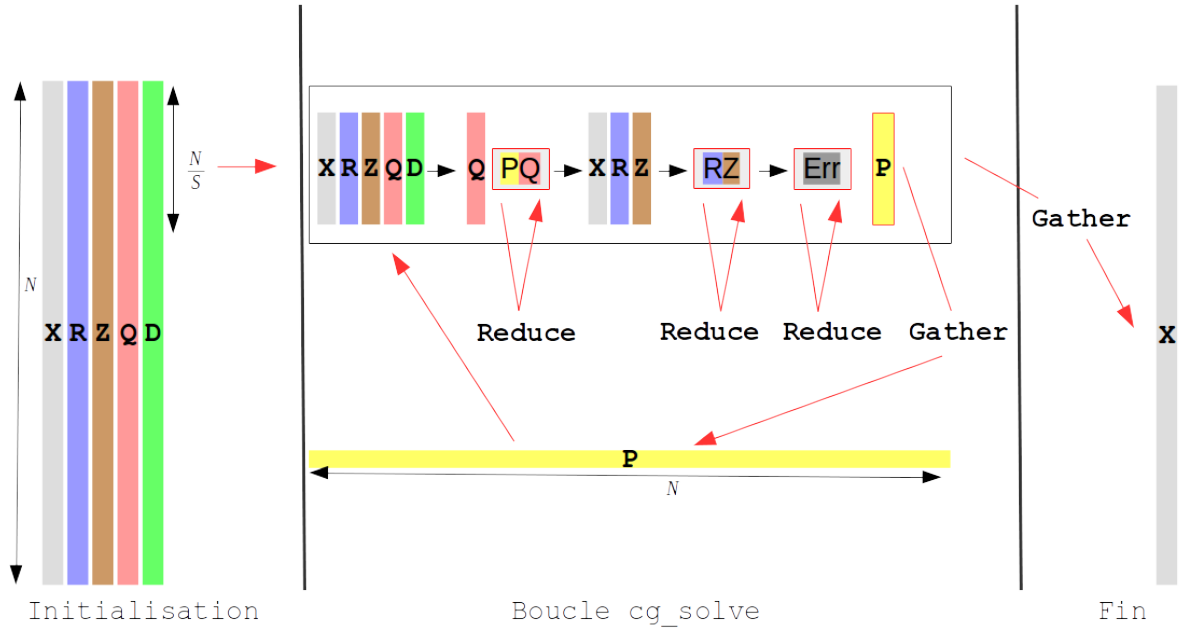

FIGURE 2.2 – Shéma résumé des échanges pour un noeud

2.2 OpenMP

Comme nous le verrons plus en détail dans la partie 3.1.4, la parallélisation MPI ne s'est pas révélée très fructueuse à cause des coûts de communication élevé, et nous avons donc choisi d'effectuer en priorité la parallélisaiton OpenMP sur le code séquentiel, pour obtenir des résultats avantageux.

2.2.1 Implémentation

Le code séquentiel a été assez simple à paralléliser car le gros du calcul s'effectue sur des boucles for avec un nombre d'itération fixe. Nous avons donc simplement ajouté des *pragma omp parallel for* à toutes les boucles de la fonction *cg_solve* et *sp_gemv*, qui permet de répartir équitablement les charges entre tous les threads. En effet, dans la boucle *cg_solve*,peu importe l'indice, le calcul consiste en le même nombre d'opérations flottantes, nous n'avons donc pas à nous soucier d'un équilibrage de charges plus complexe ou dynamique. Dans *sp_gemv* il est possible que les charges soient déséquilbrées si certaines zones sont plus creuses que d'autres, mais ce n'est pas ce que nous avons remarqué sur les matrices utilisées. Nous avons aussi ajouté un *pragma omp parallel for reduction* à la boucle du produit matriciel.

2.2.2 Prévisions

Comme les charges sont à priori équilibrées, on peut prévoir de réduire le temps de calcul proportionnelement au nombre de thread utilisés.

3. Résultats et Interprétations

3.1 MPI

3.1.1 Résultats théoriques attendus

Comme indiqué précédemment, les opérations dominantes lors d'une itération de l'algorithme CG sont les produits matrice-vecteur. En général, la multiplication matrice-vecteur nécessite *O(nnz)* opérations, où nnz est le nombre d'entrées non nulles dans la matrice. Supposons que nous souhaitons effectuer suffisamment d'itérations pour réduire la norme de l'erreur d'un facteur de $\epsilon = 10^{-8}$. Grâce aux formules mathématiques, nous pouvons montrer que la méthode du gradient conjugué a une complexité temporelle de $O(nnz^*\sqrt{k})$; tout en ayant une complexité spatiale de *O(m)*, où k est le nombre de conditionnement de A.
Comme nous divisons notre système d'entrée en plusieurs sous-systèmes pour travailler en parallèle, la complexité de notre algorithme devrait diminuer avec le nombre de processus car nous diminuons le nombre nnz. Ainsi, l'accélération due à ce niveau principal de parallélisme devrait être élevée, voire linéaire pour un petit nombre de processeurs. Cependant, le temps nécessaire pour effectuer la collecte de tout ce sous-système est de plus en plus long avec l'augmentation du nombre de processeurs.
Ainsi, il existe un seuil entre ces valeurs qui deinissent l'ampleur de l'accélération. Cependant, cette mise en oeuvre permettrait une accélération si et seulement si la taille du problème est suffisamment importante pour souligner l'intérêt d'un tel algorithme. L'accélération sera également obtenue grâce au parallélisme efectué au niveau des opérations, la multiplication des vecteurs matriciels étant le processus le plus important. Toutefois, cette accélération devrait avoir un impact moindre sur le résultat de final.

Nous avons éxécuté le programme avec les modifications MPI, qui donne la bonne solution mais avec des performances peu désirables.

3.1.2 Remarques

Lors de nos tests sur la parrallélisation de la fonction *dot*, nous avons remarqué que les résultats étaient légèrement différents en séquentiel qu'avec *MPI_reduce*, un écart de l'ordre de 10^{-9}. Celui-ci est du à la non-associativité de l'addition des flottants, qui entraîne une légère différence de résultats selon l'ordre d'addition. Cet ordre est modifié lors de la parallélisation, mais les différences obtenues sont trop faibles pour influencer les résultats des matrices.

3.1.3 Temps de calcul

Nous avons pris différentes mesures pour rendre compte du temps de résolution du problème. Pour une matrice (ici bkcs13) et une graine fixée, nous avons fait varier le nombre de noeuds et prélevé certaines données.

Il s'agit d'abord de vérifier si le nombre de tours dans la boucle principale de *cg_solve* est le même. Les légères différences observées dans la figure 3.1 s'expliquent par les variations de calculs de flottants mentionnés dans la section 3.1.2.

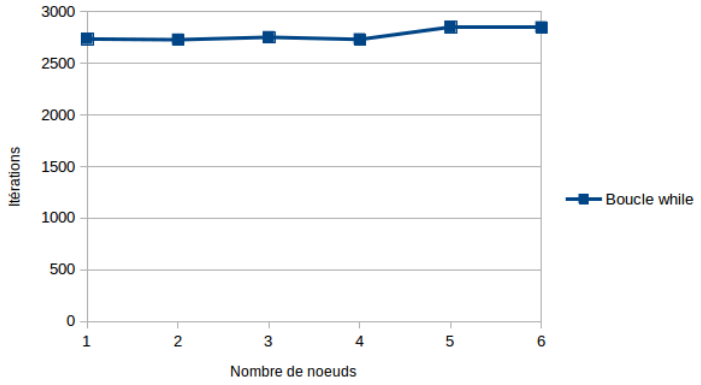

FIGURE 3.1 – Nombre de tours de boucles dans la fonction cg_solve

Nous avons ensuite vérifié si le nombre d'itérations dans les boucles for et les produits matriciels étaient bien inversement proportionnels au nombre de noeuds, ce qui est bien le cas dans la Figure 3.2.

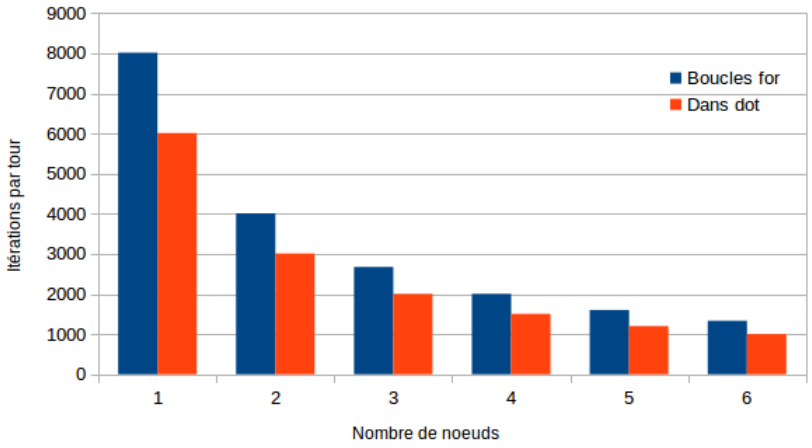

FIGURE 3.2 – Itérations par tour de boucle

Pour la fonction *sp_gemv*, il s'agissait aussi de vérifier s'il n'y avait pas de déséquilibrage de charges, nous avons donc choisi d'afficher sur la Figure 3.3 le nombre d'itérations dans la fonction pour chaque noeud, qui semble bien réduire et être assez équilibré à chaque fois.

En revanche,en ce qui concerne le temps de calcul, nous obtenons des résultats beaucoup moins performants, comme le montre les Figures 3.4 et 3.5 .

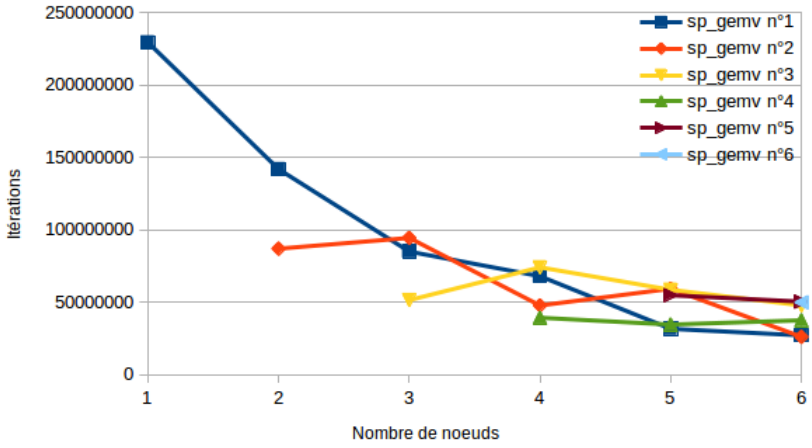

FIGURE 3.3 – Itérations par noeud

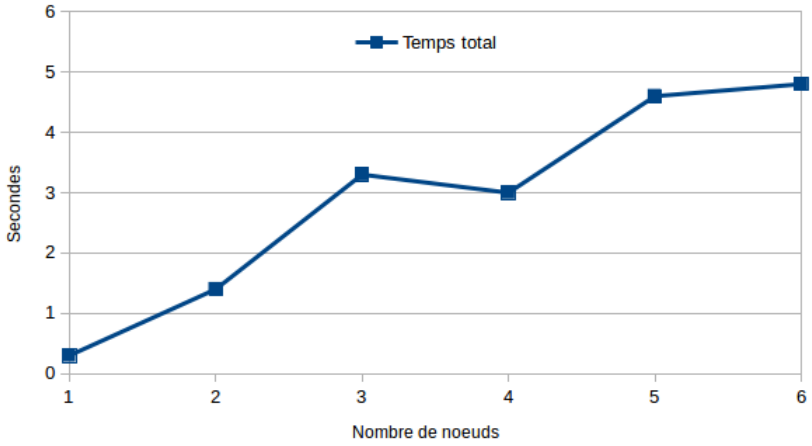

FIGURE 3.4 – Temps de calcul en fonction du nombre de noeuds pour bkcs

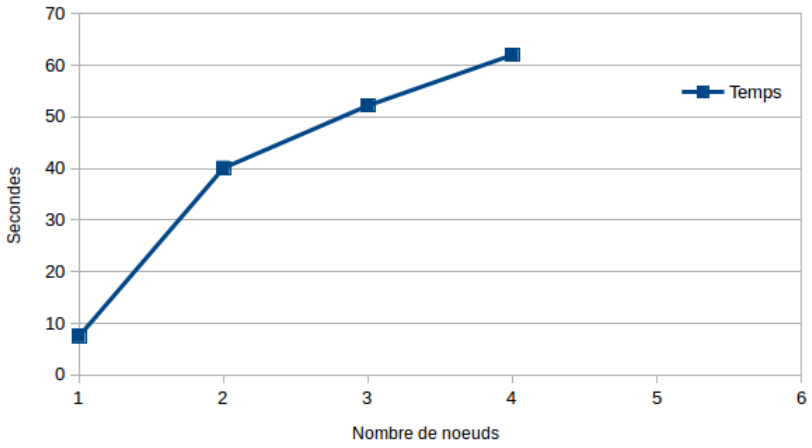

FIGURE 3.5 – Temps de calcul en fonction du nombre de noeuds pour cfd1

Nous avons essayé de comprendre le paradoxe de cette augmentation du temps total de calcul devant la réduction du nombre d'itérations à effectuer par noeuds dans la partie 3.1.4.

3.1.4 Estimer le temps de calcul

En observant de plus près l'éxécution du code (pour 4 noeuds et la matrice bkcs), nous avons remarqué qu'un tour de boucle de *cg_solve* prenait environ 20 fois plus de temps avec la commande *MPI_Allgatherv* que sans (nous l'avons enlevé la ligne correspondante lors d'un test, qui nous donne bien sur une solution abérante). Cela signifie que la communication du vecteur p à tous les noeuds prend environ 20 fois plus de temps que les opérations effectuées par les noeuds. Ce qui pose un énorme problème puisque ce vecteur doit forcément être communiqué à tous les autres à chaque tour de boucle. Nous avons donc procédé à quelques calculs afin de savoir dans quelle mesure la parallélisation MPI était envisageable.

- N : taille de la matrice et donc du vecteur p
- S : nombre de noeuds utilisés
- α : coût en communication d'un flottant entre 2 noeuds
- β : coût de l'établissement de la communication entre 2 noeuds distincts
- δ : coût d'un FLOP

Un noeud s s'occupe d'environ $n_s = \frac{N}{S}$ données. En prenant en compte les 4 boucles for, les 3 produits matriciels (2 + celui du calcul de la norme), et la fonction *sp_gemv*, on arrive à un total de $8n_s$ FLOPs par tour de boucle While.

La commande *MPI_Allgatherv* va nécessiter 2 étapes : l'envoi de n_s flottants au communicateur ce qui coutera $\beta + n_s\alpha$, puis la distribution par le communicateur à tous les processus du vecteur complet obtenu c'est à dire un coût de $(S-1)(\beta + N\alpha)$.[1]

Ainsi le coût d'un tour de boucle est de :

$$\begin{aligned} CT &= 8\frac{N}{S}\delta + \beta n_s + (S-1)(\beta + N\alpha) \\ &\approx 8\frac{N\delta}{S} + S(\beta + N\alpha) \text{ Pour simplifier les calculs} \end{aligned}$$

On remarque malheureusement que le deuxième terme de la somme augmente de manière linéaire avec la taille de la matrice et avec le nombre de processeurs, ce qui rend la parallélisation MPI inutile dès lors que $\alpha >> \delta$ ...

Notre programme parallèle MPI a donc assez peu de chance d'être plus efficace que le séquentiel sur les PC de la ppti, donc le coût de communication réseau est particulièrment élevé.

3.2 OpenMP

Nous avons essayé notre programme OpenMP sur un ordinateur de la ppti pour deux petites matrices, pour obtenir les résultats affichés dans le graphique 3.6. On voit que le temps par itération décroit bien en fonction du nombre de threads. Cependant, ce ne sont pas exactement les résultats optimaux qu'on pourrait attendre en divisant le temps par le nombre de threads, sur la courbe bkcs théorique.

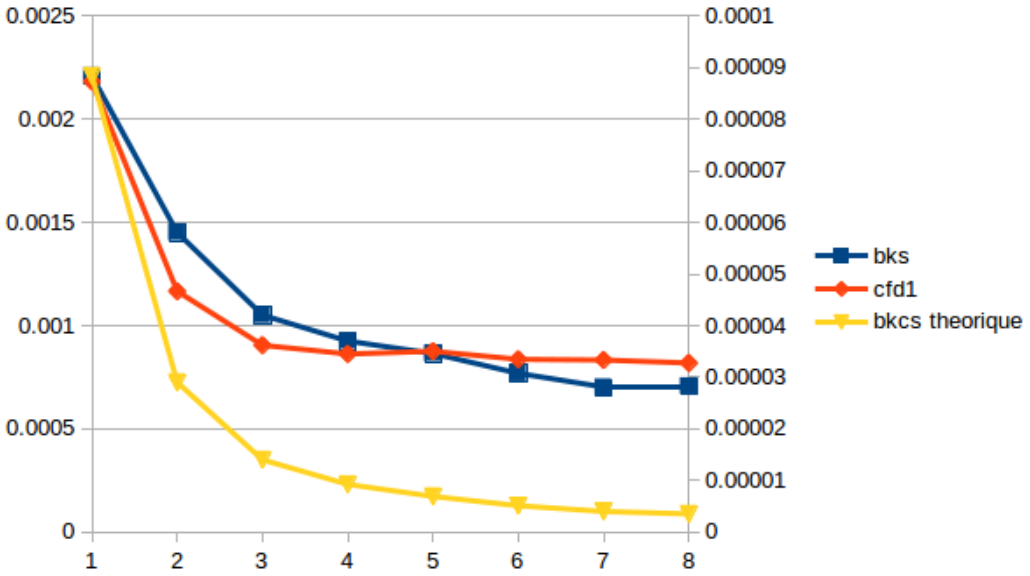

FIGURE 3.6 – Temps par itération obtenu sur deux matrices, en fonction du nombre de threads


```
[
    {
    version: 3,
    users: ["Achille Baucher", "Najwa Moursli"],
    matrix: "cfd1",
    hardware: "PC de la PPTI, 305",
    software: "Version OpenMP complete",
    nodes: 4,
    cores: 8,
    start: 1591086295.9168866,
    seed: 309459981500128185,
    running_time: 2.576429605484009,
    computation_id: 863
    },
    "MEUCIA24UrL/JCeGi5d/yFg9K7rhxPNZRIpN9xPOGBWgB5/
    HAiEAy2IIrxFFIqQ3vBcP7qazrYi+DEHniLQ/bYygGmIV6HU="
]
```

3.3 MPI+OpenMP

3.4 Strong scaling

Dans le cas présent, La loi d'Amdahl[2] peut être formulée comme suit $a=\frac{1}{s+\frac{p}{N}}$ où s est la proportion du temps d'exécution consacré à la partie séquentielle, p est la proportion du temps d'exécution consacré à la partie parallèle, et N est le nombre de processeurs. Généralement par definition, *p = (1-s)*. La loi de l'Amdahls stipule que, pour un problème fxé, la limite supérieure de l'accélération est déterminée par la fraction séquentielle du code. C'est ce que l'on appelle "strong scaling". Dans cette partie, nous allons analyser celle-ci sur notre algorithme. Ainsi, nous étudions la relation entre le nombre de threads/rangs donnés pour une itération avec l'accélération qu'elle permet.

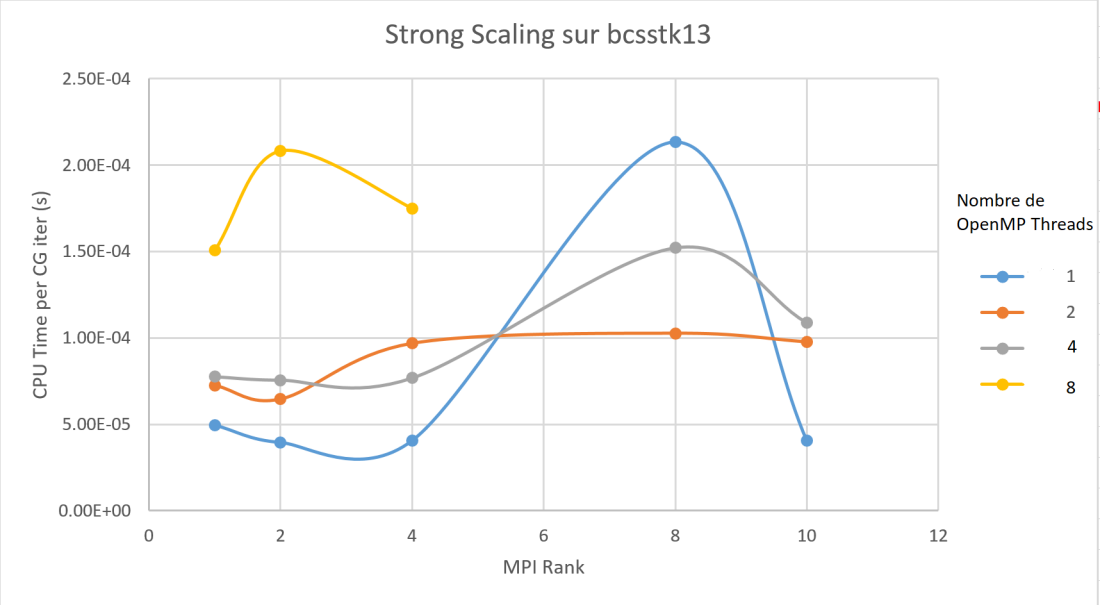

FIGURE 3.7 – Strong scaling sur bcsstk13

L'accélération la plus importante a été obtenue pour 8 et 2 rangs MPI. L'accélération diminue également après un certain seuil de threads et ces seuils sont aussi bas que deux threads. Ce n'est que dans le cas d'un seul rang que l'accélération augmente avec le nombre de threads en continu. Cela signifie que pour une matrice 1000x1000, au lieu de devoir opérer sur 1 million d'éléments, avec le format CRS, nous n'avons besoin d'opérer que sur 7000. Il n'y a pas beaucoup d'accélération gagnée en parallélisant des matrices creuses de petite à moyenne taille en raison du surcroît de travail que représente la communication des messages MPI et la création de threads OpenMP, qui est énorme par rapport au nombre limité d'opérations à effectuer. Pour les grandes matrices creuses, le strong scaling devient plus similaire au cas de la matrice dense. Dans ce projet, nous nous concentrons sur les matrices de petite à moyenne tailles car elles sont plus intéressantes en termes de compréhension du comportement de parallélisation et ce sont les tailles de matrice que nous rencontrons fréquemment dans les travaux de recherche.

3.4.1 Weak scaling

La loi Amdahls[2] donne la limite supérieure d'accélération pour un problème de taille fixée. Cela semble être assez réducteur pour le calcul parallèle. Cependant, comme l'a souligné Gustafson[3], dans la pratique, l'ampleur des problèmes augmente avec la quantité de ressources disponibles. Si un problème ne nécessite qu'un petit nombre de ressources, il n'est pas judicieux d'utiliser une grande quantité de ressources pour effectuer le calcul. Un choix plus raisonnable consiste à utiliser de petites quantités de ressources pour les petits problèmes et de plus grandes quantités de ressources pour les grands problèmes. C'est pourquoi la loi de Gustafsons est basée sur l'approximation que la partie parallèle s'échelonne linéairement avec le nombre de ressources, et que la partie séquentielle n'augmente pas en fonction de l'ampleur du problème. Il fournit la formule pour l'accélération graduelle : accélération échelonnée, $a=s+p*N$ où s, p et N ont la même signification que dans la loi Amdahls.
Dans le cours nous l'avons définis de la manière suivante : étant donné p le nombre de processeurs, β la fraction de code non-parallélisable et n la taille du problème, l'accélération est $a=p-\beta(n)(p-1)$.

Avec la loi de Gustafson, l'accélération échelonnée augmente de façon linéaire en fonction du nombre de processeurs (avec une pente inférieure à u1), et il n'y a pas de limite supérieure pour celle-ci. C'est ce qu'on appelle le "weak scaling" ; où l'accélération de l'échelle est calculée en fonction de la quantité de travail effectuée pour une taille de problème à l'échelle (contrairement à la loi Amdahls qui se concentre sur la taille du problème fixée). Ainsi, nous nous intéressons maintenant à l'étude du temps pris pour différents nombre de thread OpenMP en fonction des rangs MPI et de la taille du problème de telle sorte que la taille du problème par rang MPI reste constante. En effet, pour une faible mise à l'échelle, la taille de la matrice a été augmentée proportionnellement au nombre de rangs MPI utilisés.

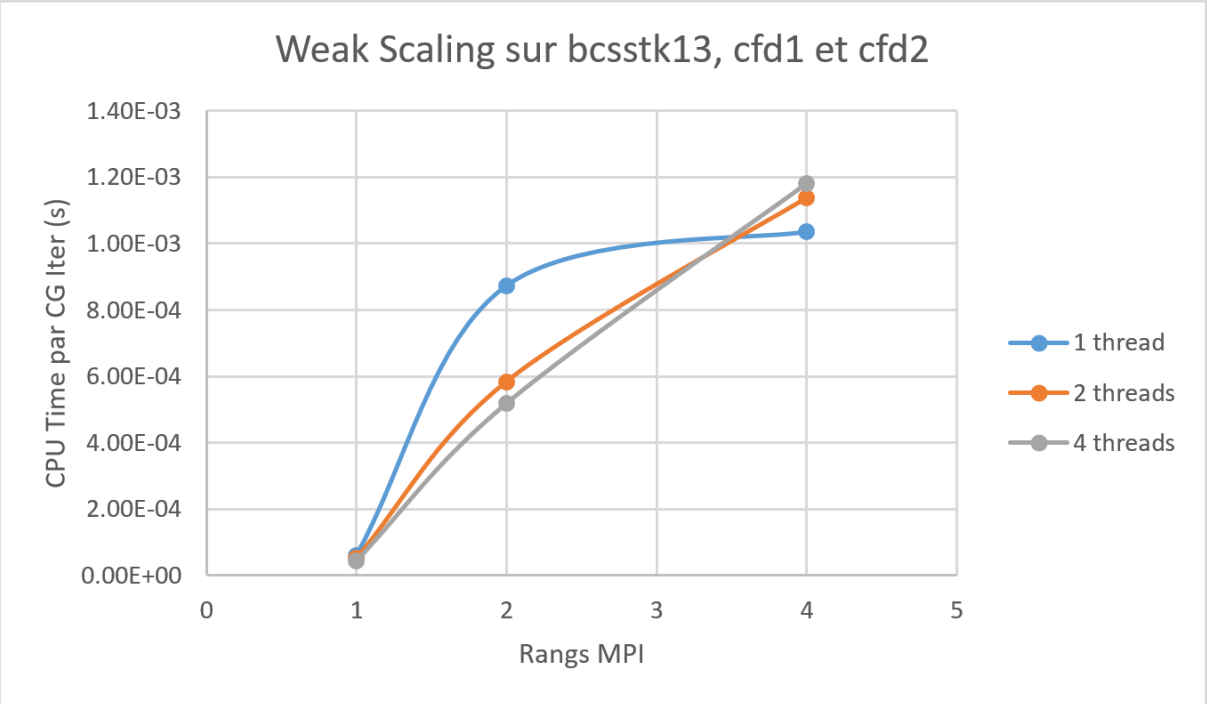

Figure 3.8 – Weak scaling sur les matrice bssckt13, cfd1 et cfd2

Pour le weak scaling, la taille de la matrice a été augmentée proportionnellement au nombre de rangs MPI utilisés en changeant celle-ci entre *bcsstk13, cfd1 et cfd2*. La figure 3.8 montre les propriétés du weak scaling.

Ici, nous avons testé les temps CPU pour nombre de threads OpenMP en fonction des rangs MPI et de la taille du problème (déterminée par le nombre d'éléments dans la matrice) de sorte que la taille du problème par rang MPI reste constante.

```
ENV OMP_NUM_THREADS=(nb thread voulu) mpirun -np (nb processus voulu)
./cg_fusion --matrix matrix.txt
```

Un weak scaling idéal aurait une courbe aplatie, montrant que le temps CPU reste constant comme la taille du problème par rang reste constante.Le temps CPU commence à augmenter lorsque 8 rangs ont été utilisés, probablement en raison de l'important overhead (toute combinaison de temps de calcul, de mémoire, de largeur de bande ou d'autres ressources excédentaires ou indirectes nécessaires à l'exécution d'une tâche spécifique) dans la communication qui n'a pas été suffisamment compensé par le gain de performance lié à l'utilisation de plus de rangs MPI. Rappelons que nous devons utiliser *MPI_Allreduce* et *MPI_Allgatherv*, ce qui permettra d'augmenter le temps de communication du processeur avec une taille de problème plus importante, même si le nombre d'opérations mathématiques par rang reste le même. Nous constatons une augmentation des temps de CPU en raison de la moindre quantité d'opérations nécessaires sur les matrices de format CRS, ce qui fait que l'accélération due à la parallélisation supplémentaire n'est pas suffisante pour compenser l'augmentation de l'overhead.

4. Conclusion

Nous avons essayé dans cette étude une parallélisation avec mémoire ditribuée avec MPI, non-distribuée avec OpenMP, puis hybride de l'algorithme du gradient conjugué pour résoudre les équations à partir de matrices creuses.
La parallélisation avec mémoire distribuée s'est finalement révélée inefficace à cause du trop grand coût de communication nécessaire pour cet algorithme. En partageant la mémoire avec OpenMp, en revanche, celle-ci a montré un assez grand gain de temps. L'utilisation d'OpenMP avec un nombre maximal de threads offre les meilleures performances.
En ce qui concerne le scaling avec des matrices creuses, on s'attend à des accélérations plus importantes lors de la résolution de matrices plus grandes ($>2.10^{-5}$).

Bibliographie

[1] S. D. SANTACREU. (2008). Introduction au calcul parallèle avec la bibliothèque MPI (Message Passing Interface), adresse : https://cel.archives-ouvertes.fr/cel-00395829/document.

[2] G. M. Amdahl, « Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities », in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, New York, NY, USA : Association for Computing Machinery, 1967, p. 483-485, isbn : 9781450378956. doi : 10.1145/1465482.1465560. adresse : https://doi.org/10.1145/1465482.1465560.

[3] J. L. Gustafson, « Reevaluating Amdahl's Law », *Commun. ACM*, t. 31, n° 5, p. 532-533, mai 1988, issn : 0001-0782. doi : 10.1145/42411.42415. adresse : https://doi.org/10.1145/42411.42415.