5A ModIA

Rapport de Projet

Using MPI to efficiently distribute GEMM computations

Elèves :
Karima GHAMNIA
Cassandra MUSSARD

Enseignant :
Ronan GUIVARCH

12 janvier 2025

Table des matières

1 Introduction

Dans ce projet nous allons analyser les performances de la routine GEMM qui contient divers algorithmes de multiplication générale de matrices, en utilisant différents types de communications (communications bloquantes paires à paires, communications collectives bloquantes, communications paires à paires non bloquantes) à l'aide de MPI. Nous analyserons la scalabilité de cet algorithme en faisant varier certains paramètres pour conclure sur l'impact de chacune des communications sur la performance de cette routine.
Pour réaliser cela nous utiliserons l'émulateur Simgrid qui permet de simuler des applications utilisant le standard MPI (Message Passing Interface) et de tester cet algorithme sur un environnement de systèmes distribués.

2 Algorithme étudié

Comme expliqué dans l'introduction nous allons analyser les performances de l'algorithme GEMM (General Matrix Multiplication).
L'objectif de cet algorithme est de faire le calcul suivant :

$$C = \alpha * opA(A)opB(B) + \beta * C$$

Ici nous fixons $\alpha = 1$, $\beta = 0$ et $op(.) = Id$.
Nous supposons que la matrice C est de taille $m * n$, A de taille $i * k$ et B de dimension $k * j$.
Nous utilisons une grille de processus pour distribuer les blocs de matrices. Cette grille est de taille $p * q$ avec p le nombre de processus sur chaque ligne de la grille et q le nombre de processus sur chaque colonne de la grille.
Les matrices sont divisées en blocs de taille $b * b$, et chaque bloc est attribué à un processus particulier dans la grille.
Pour étudier la scalabilité et les performances de ces algorithmes selon le type de communication utilisé nous ferons varier les hyperparamètres cités au-dessus.

3 Différentes implémentations

3.1 Communications bloquantes paires à paires

Dans cette section, nous avons implémenté les fonctions `p2p-transmit-A` et `p2p-transmit-B` dans le fichier `exo1.c` en utilisant MPI.
Ces deux fonctions gèrent la transmission des blocs des matrices A et B en fonction de leur possession et de leur distribution sur une grille de nœuds de dimensions $p \times q$

L'utilisation de `MPI-Ssend` effectue un envoi synchrone suivi d'une réception classique avec `MPI_Recv`.
L'envoi des blocs A et B se fait à l'aide d'une communication bloquante paire à paire. Un processus va donc envoyer un message directement à un autre processus, et cette opération est bloquante. Cela signifie que l'opération d'envoi ou de réception de message ne se termine pas tant que le message (les blocs de matrices) n'a pas été complètement transféré.

3.1.1 Trace des communications

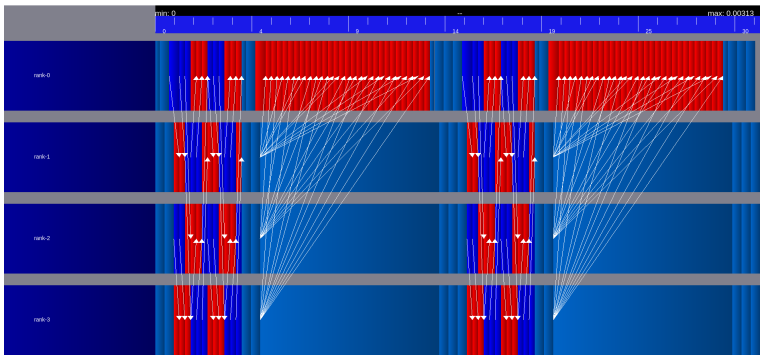


FIGURE 1 – Traces des communications bloquantes paires à paires

La figure 1 a été obtenue en prenant $m = n = k = 20$, $b = 5$, $p = q = 2$ et 4 processus.
Sur l'axe vertical nous avons les rangs 0 à 3 qui représentent les processus.
Dans le cadre d'une communication paire à paire les processus communiquent directement entre eux, en envoyant et recevant des données de manière séquentielle. Chaque processus envoie des données à plusieurs autres processus, comme indiqué par les flèches partant d'un rang et se dirigeant vers d'autres. Cela correspond à l'appel de MPI_Ssend qui va permettre de transmettre les blocs A et B.
Nous pouvons voir que de 0 à 5 unités de temps, les processus rang-0 à rang-3 sont engagés dans des envois de données (bleu) suivis par des calculs (rouge).
À partir de la 5ème unité, les processus du rang 1 à 3 envoient des données au rang-0 qui va se charger de faire les calculs.
La zone en rouge correspond au calcul suivant $C = A * B$.
Un comportement similaire est ensuite observé entre les unités 15 et 30. Nous avons deux fois le même comportement car nous avons 2 itérations.
En conclusion, nous voyons que les processus du rang 1 à 3 sont surtout engagés dans des envois de données (avec quelques calculs). Le processus de rang 0 (master) est surtout impliqué dans des calculs (avec quelques échanges de données au début).

3.2 Communications collectives bloquantes

Dans cette section, nous avons implémenté les fonctions `bcast-A` et `p2p-bcast-B` dans le fichier `exo2.c` en utilisant MPI. Les deux fonctions facilitent la communication collective pour diffuser des blocs de matrices entre les nœuds tel que `bcast-A` diffuse le bloc $A[i, l]$ entre les nœuds de la même ligne en utilisant `MPI-bcast` et vérifie si le bloc appartient à la ligne du nœud actuel et effectue la diffusion en conséquence. Similairement, `bcast-B` réalise la même chose pour les colonnes.

3.2.1 Trace des communications

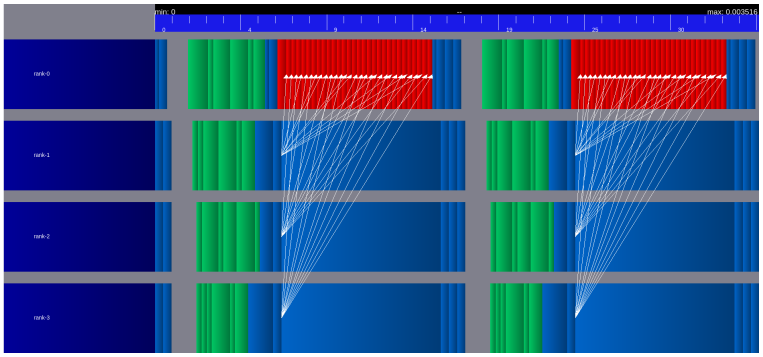


FIGURE 2 – Traces des communications collectives bloquantes

La figure 2 a aussi été obtenue en prenant $m = n = k = 20$, $b = 5$, $p = q = 2$ et 4 processus.
Nous voyons sur cette figure que l'algorithme commence par effectuer la diffusion des blocs des matrices A et B à l'aide de la fonction MPI_Bcast (blocs verts). Cette opération est effectuée par tous les processus de rang 0 à 3.
Après les diffusions initiales, les processus peuvent envoyer des données supplémentaires si nécessaire. Les segments bleus montrent ces envois de données. Cette opération est aussi effectuée par tous les processus.
Ensuite, vers l'unité de temps 5, les processus de rangs 1 à 3 réalisent des envois de données pair à pair au processus de rang 0 pour par exemple faire des synchronisations entre les phases de calcul ou des envois de résultats intermédiaires entre les processus.
Enfin, à partir de l'unité 6, le processus de rang 0 se charge de faire le calcul $C = A * B$ (blocs rouges).

3.3 Communications paires à paires non bloquantes

Dans cette partie nous avons implémenté les fonctions `MPI-p2p-i-transmit-A`, `MPI-p2p-i-transmit-B` et `p2p-i-wait-AB` dans le fichier `exo3.c`. La fonction `MPI-p2p-i-transmit-A`

et `MPI-p2p-i-transmit-B` postent les envois et réceptions non bloquants, tandis que `p2p-i-wait-AB` attend la fin de ces communications avant d'effectuer les calculs nécessaires. Dans cette implémentation nous avons un paramètre `lookahead` qui correspond au nombre d'étapes futures de communication que l'algorithme prépare à l'avance.

3.3.1 Trace des communications

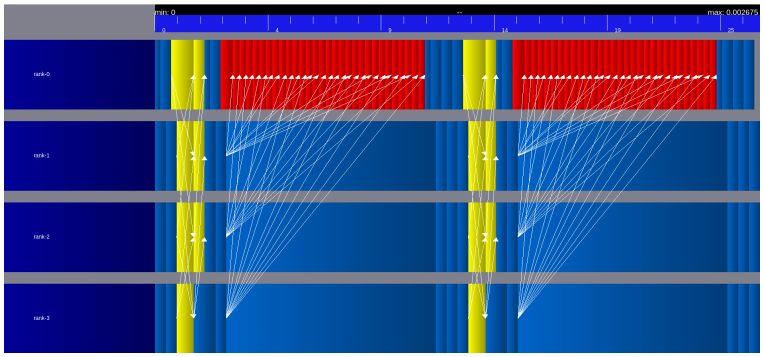


FIGURE 3 – Traces des communications paires à paires non bloquantes

La figure 3 a été obtenue en prenant $m = n = k = 20$, $b = 5$, $p = q = 2$ et 4 processus.
Au début les segments jaunes montrent les périodes où tous processus préparent les transmissions non bloquantes (utilisation de MPI_Issend et MPI_Irecv), les flèches blanches montrent les échanges effectués pour pouvoir préparer cette transmission.
Il y a ensuite une phase de transmission réelle des données entre tous les processus (en bleu). Les flèches blanches montrent bien cet échange de données. On remarque d'ailleurs que l'échange se fait des processus 1 à 3 vers le processus 0.
Enfin, une fois que le processus de rang 0 a récupéré toutes les données il peut effectuer le calcul $C = A * B$.

4 Analyse des performances

Pour analyser la scalabilité et la vitesse d'exécution de l'algorithme à l'aide de plusieurs types de communications nous devons étudier plusieurs hyperparamètres.

— p et q qui correspondent au nombre de processus sur chaque ligne et colonne (initialement à 2).

— m,n et k les tailles des matrices (initialement à 20).

— lookahead le nombre d'étapes futures de communication (initialement à 1).

— b, la taille des blocs (initialement à 5).

Nous allons faire varier chacun des paramètres (à l'aide du fichier `bench.sh` qui permet de créer un benchmark) l'un après l'autre pour observer leur impact sur la vitesse d'exécution de l'algorithme.

4.1 Variation du paramètre lookahead

Dans cette partie nous cherchons à comprendre l'impact du paramètre lookahead sur le dernier alogorithme.
Ce paramètre correspond au nombre de blocs qu'un processus doit commencer à communiquer avant de commencer ses calculs.
Nous avons fixé les paramètres p et q à 2, b à 256.
Nous affichons alors l'évolution des Gflops/s (vitesse d'exécution de l'algorithme) en fonction de la taille du problème n pour différentes valeurs de lookahead.

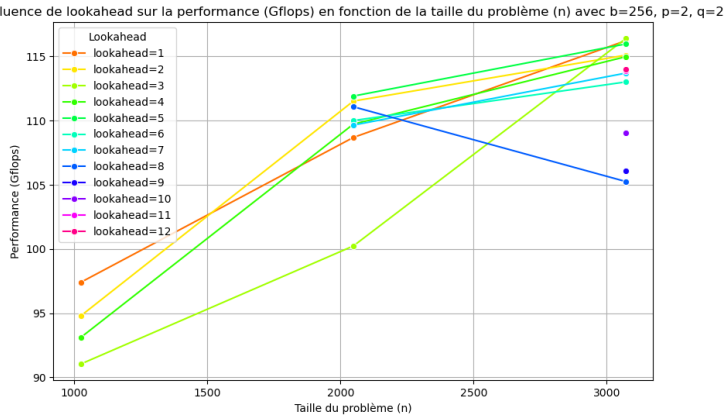


FIGURE 4 – Impact de lookahead sur la performance de l'algorithme p2p-i-la

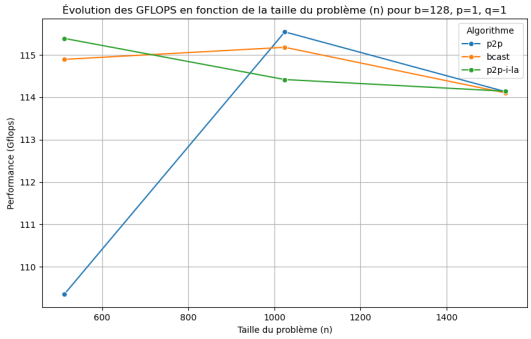
Nous remarquons que si la taille du problème est inférieure à 1500 alors il faut prendre un lookahead très petit (=1).
Si la taille du problème est comprise entre 1500 et 2048, il faut prendre lookahead à 2.
Enfin, si la taille du problème est supérieure à 2048 alors il faut prendre lookahead à 5.
Ceci paraît logique car si la taille du problème est petite le processus n'a pas besoin de communiquer beaucoup de blocs avant de commencer les calculs et inversemment si la

taille du problème est grande.

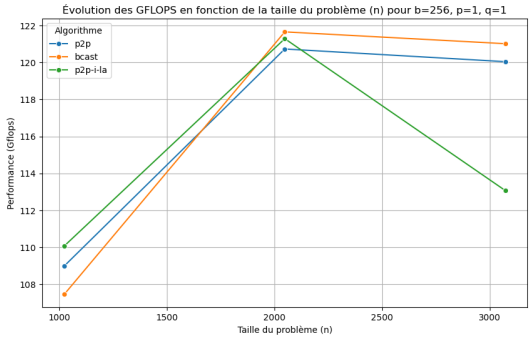
4.2 Variation du paramètre b

Dans cette partie nous faisons varier le paramètre b qui correspond à la taille des blocs des matrices A et B.
Dans un premier temps nous allons afficher l'évolution du nombre de Gflops/s en fonction de la taille du problème (n) pour différentes valeurs de b (128, 256, 512).
Nous étudierons cette évolution pour trois valeurs de p et q (1, 2 et 4).
Comme nous avons vu l'impact du paramètre lookahead dans la partie précédente nous prenons $lookahead = 1$ si la taille du problème est inférieure à 1000, $lookahead = 2$ si la taille du problème est entre 1024 et 2048 et $lookahead = 5$ si la taille du problème est supérieure à 2048.

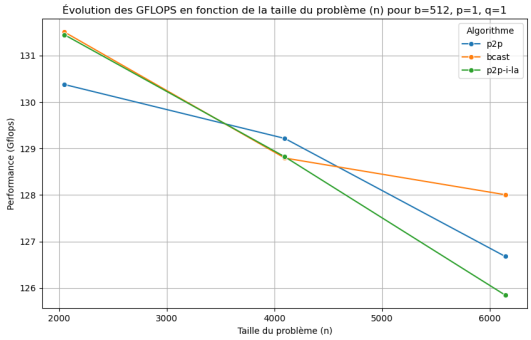
4.2.1 Variation de b avec $p = q = 1$



(a) b=128, et p=q=1



(b) b=256, et p=q=1



(c) b=512, et p=q=1

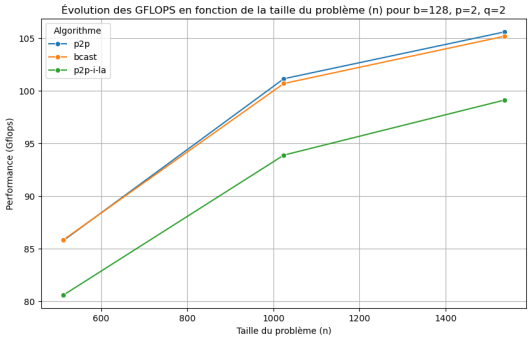
FIGURE 5 – Évolution des Gflops/s en fonction de la taille du problème pour différentes valeurs de b et p,q

— $\underline{p = q = 1, b = 128 :}$ Nous remarquons que pour l'algorithme p2p la performance en Gflops/s augmente de manière significative jusqu'à environ $n = 1024$, puis diminue rapidement. Pour l'algorithme bcast nous voyons une légère augmentation de performance jusqu'à $n = 1024$, suivie d'une légère diminution. Enfin, la performance de l'algorithme p2p-i-la diminue progressivement à mesure que n augmente. L'algorithme p2p présente une augmentation notable jusqu'à un certain point, ce qui indique une bonne efficacité de communication et de calcul pour des tailles de problème intermédiaires. Les algorithmes bcast et p2p-i-la sont plus stables mais moins performants que p2p à des tailles plus petites et intermédiaires.

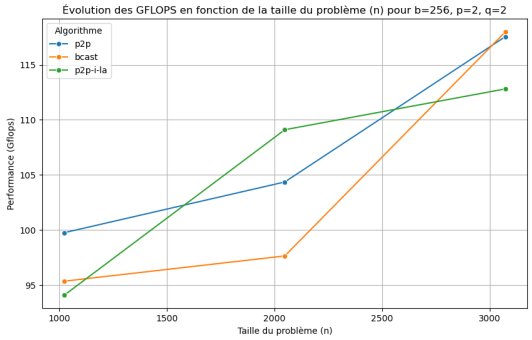
— $\underline{p = q = 1, b = 256 :}$ Nous observons le même comportement pour les 3 algorithmes lorsque $n < 2048$, c'est à dire une augmentation des performances avec une augmentation un peu plus marquée pour l'algorithme bcast.
Une fois que $n > 2048$, nous pouvons noter une très forte diminution des performances pour l'algorithme p2p-i-la et une légère baisse pour les 2 autres algorithmes.

— $\underline{p = q = 1, b = 512}$ Les performances diminuent fortement à mesure que n augemnte (entre 2000 et 6000). La taille des blocs est devenue trop grandes et nous ne bénéficions plus du découpage en blocs.

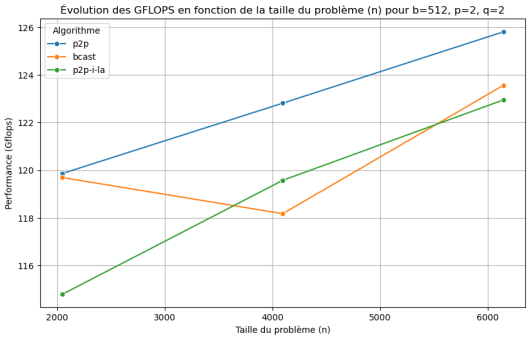
4.2.2 Variation de b avec $p = q = 2$



(a) b=128, et p=q=2



(b) b=256, et p=q=2



(c) b=512, et p=q=2

FIGURE 6 – Évolution des Gflops/s en fonction de la taille du problème pour différentes valeurs de b et p,q

— $p = q = 2$, $b = 128$ Pour $b = 128$, les algorithmes p2p et bcast montrent des performances similaires et constantes, dépassant les 100 Gflops/s. p2p-i-la est légèrement moins performant mais montre une amélioration constante.

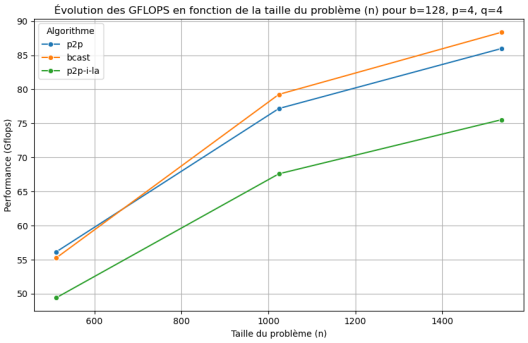
— $p = q = 2$, $b = 256$ Pour $b = 256$, bcast montre une performance supérieure aux autres algorithmes, dépassant les 115 Gflops/s lorsque n est très grand. p2p et p2p-i-la montrent également une amélioration continue et restent meilleurs lorsque $n < 3000$.

— $p = q = 2$, $b = 512$ Pour $b = 512$, p2p montre la meilleure performance avec une augmentation continue, atteignant environ 126 Gflops/s. bcast et p2p-i-la montrent des performances similaires, avec une légère variation mais se stabilisant autour de 125 Gflops/s.

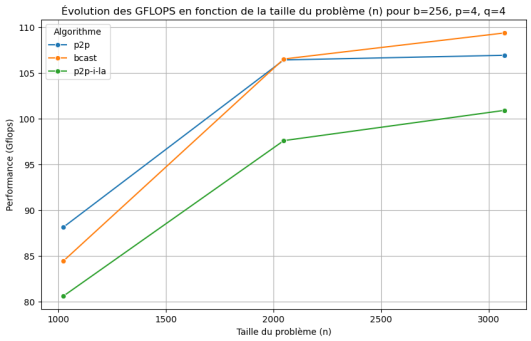
Dans le cas où nous avons 2 processus sur les lignes et colonnes nous ne remarquons pas de baisse de performances pour les 3 algorithmes. Cela nous confirme le fait que les

temps de communication et de transmission de données ne dominent pas.

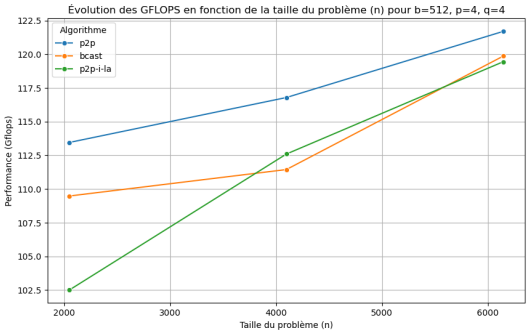
4.2.3 Variation de b avec $p = q = 4$



(a) b=128, et p=q=4



(b) b=256, et p=q=4



(c) b=512, et p=q=4

FIGURE 7 – Évolution des Gflops/s en fonction de la taille du problème pour différentes valeurs de b et p,q

— $\underline{p = q = 4,\ b = 128}$ Même constatations que pour $p = q = 2$ et $b = 128$.

— $\underline{p = q = 4,\ b = 256}$ Pour $b = 256$, bcast montre une performance supérieure lorsque $n > 2048$, atteignant environ 110 Gflops/s. p2p suit de près avec 105 GFLOPS et est meilleur lorsque $n < 2048$. p2p-i-la montre une amélioration continue mais reste en dessous des autres algorithmes.

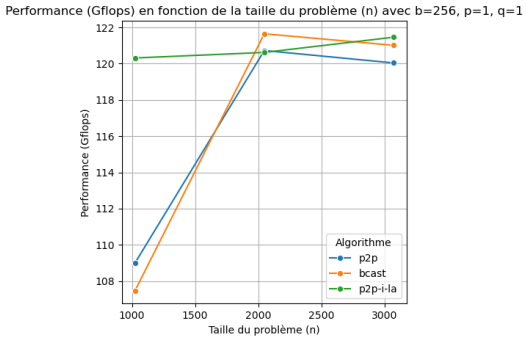
— $\underline{p = q = 4,\ b = 512}$ Pour $b = 512$, p2p montre la meilleure performance avec une augmentation continue, atteignant environ 122.5 Gflops/s. bcast et p2p-i-la montrent des performances similaires, autour de 117.5 Gflops/s. Lorsque $n < 4096$ bcast a de meilleurs performances et les rôles sont inversés lorsque $n > 4096$.

En conclusion, l'algorithme p2p montre des performances optimales pour des tailles de blocs plus grandes ($b = 512$) et pour $p = q = 4$. bcast quant à lui montre une performance supérieure pour $b = 256$, et reste compétitif pour $b = 512$. Enfin l'algorithme p2p-i-la améliore sa performance avec l'augmentation de n et montre une meilleure stabilité avec $p = q = 4$.

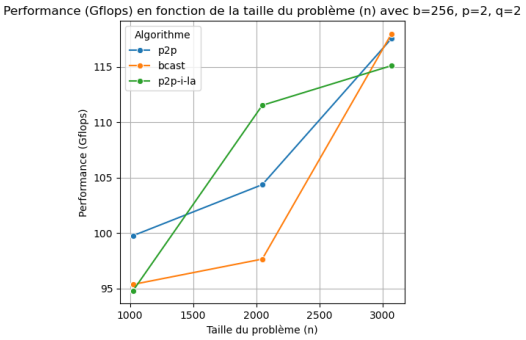
4.3 Variation des paramètres p et q

4.3.1 Avec variation de n,m et k

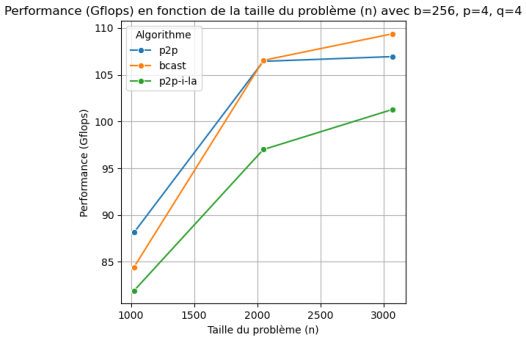
Afin de constater l'impact de ces deux paramètres nous fixons d'abord les paramètres p et q à 1 pour se mettre dans une configuration d'exécution séquentielle.
Nous faisons ensuite varier p et q de 2 à 4 pour observer si cette augmentation permet d'augmenter la vitesse d'exécution ou si le temps de communication (overhead) va dominer et ralentir l'exécution.
Nous fixons b à 256 et pour le paramètre *lookahead* nous prenons $lookahead = 1$ si la taille du problème est inférieure à 1000, $lookahead = 2$ si la taille du problème est entre 1024 et 2048 et $lookahead = 5$ si la taille du problème est supérieure à 2048.
Nous faisons varier n,m et k entre 1024 et 3000.



(a) p=q=1



(b) p=q=2



(c) p=q=4

FIGURE 8 – Évolution des Gflops/s en fonction de la taille du problème pour différentes valeurs de b et p,q

— $\underline{p = q = 1}$ Dans ce cas, nous remarquons une augmentation des performances pour les trois algorithmes avec un pic à 122 Gflops/s pour l'algorithme bcast avec $n = 2048$. Ces performances diminuent ensuite pour p2p et bcast lorsque $n > 2048$.

— $\underline{p = q = 2}$ Les performances augmentent au fur et à mesure que n augmente pour tous les algorithmes.
L'algorithme p2p-i-la est meilleur lorsque n est entre 1500 et 2800. Ce sont finalement les algorithmes bcast et p2p qui obtiennent les meilleurs résultats avec un pic à 115.5 Gflops/s pour n=3000.
Il semblerait que l'augmentation de p et q n'a pas permis d'améliorer les performances.

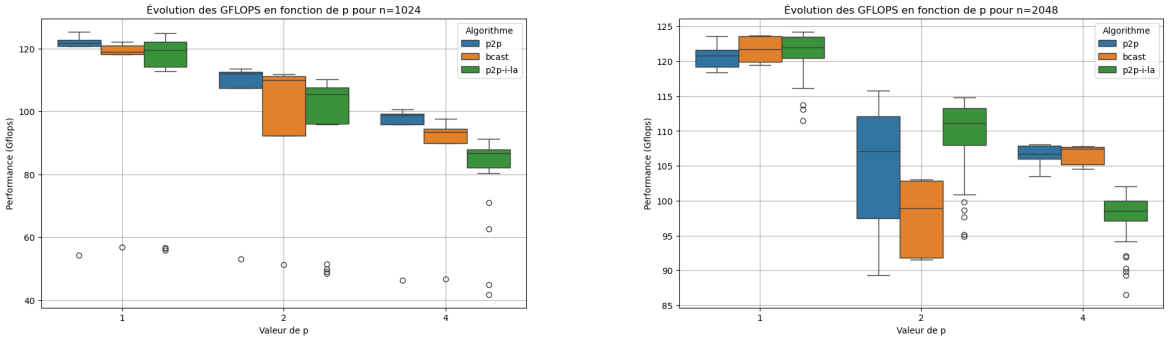
— $\underline{p = q = 4}$ Nous observons aussi une augmentation des performances à mesure que n augmente. L'algorithme p2p est meilleur pour $n < 2048$ et bcast est meilleur lorsque $n > 2048$. Les meilleurs résultats sont obtenus pour bcast à $n = 3000$. Ce résultat est toujours moins bon que lorsque $p = q = 1$.

En conclusion, nous pouvons dire que l'augmentation du nombre de processus sur les lignes et les colonnes n'a pas permis d'améliorer les résultats. Ceci montre alors que

le temps de communication domine sur le temps de calcul et ralentit alors les performances.

4.3.2 Avec n,m et k fixes

Dans cette partie nous allons faire varier les paramètres p et q avec n,m et k fixes. Nous prenons $b = 256$ car il nous a permis d'avoir des résultats corrects pour les 3 algorithmes.
Nous prenons alors m,n et k à 1024 puis 2048.



(a) n=m=k=1024 (b) n=m=k=1024

FIGURE 9 – Évolution des Gflops/s en fonction de p pour n,m et k fixes

Nous remarquons qu'à n, m et k valant 1024 les meilleures performances sont obtenues avec $p = q = 1$ et décroît à mesure que p et q augmentent. L'algorithme p2p semble meilleur quelque soit la valeur de p et q choisit.
Pour n, m et k à 2048 nous observons aussi la même tendance avec une forte diminution de la performance pour l'algorithme bcast lorsque $p = 2$.

5 Conclusion sur la variation des paramètres

5.1 Paramètres

— **Paramètre lookahead :** Nous avons remarqué que la variation de ce paramètre influence grandement les performances de l'algorithme p2p-i-la. En effet, il semblerait que les meilleurs résultats soient obtenus avec *lookahead* = 5 (115 Gflops/s). L'augmentation de ce paramètre ne permet pas d'avoir de meilleurs résultats soulignant alors que le temps de communication domine avec un *lookahead* grand.

— **Paramètre b :** L'augmentation de ce paramètre permet généralement d'améliorer les résultats pour les 3 algorithmes. Nous pourrions utiliser p2p pour des tailles de blocs plus grandes et des configurations de p et q plus élevées. bcast est lui recommandé pour des tailles de blocs intermédiaires ($b = 256$) pour maximiser les performances. Enfin, p2p-i-la peut être optimisé davantage pour améliorer sa performance et stabilité, en particulier pour des tailles de problème plus grandes.

— **Paramètres p et q** Nous avons remarqué que l'augmentation de p et q diminue généralement les performances des 3 algorithmes. Ceci est expliqué par l'augmentation du temps de communication entre les processus.

— **Paramètres n,m et k** Enfin, nous avons noté que sur la plupart des expériences, l'augmentation de m,n et k favorise les algorithmes bcast et p2p.
En effet, l'algorithme p2p-i-la nécessite une trop grande synchronisation alors que les algorithmes p2p et bcast peuvent eux mieux paralléliser les calculs lorsque ces paramètres augmentent.

5.2 Scalabilité

— **Scalabilité faible :** La scalabilité faible mesure comment l'algorithme gère l'augmentation de la taille du problème lorsque l'on augmente le nombre de processeurs. Nous avons bien vu sur la figure 8 que lorsque nous avons augmenté le nombre de processus (p et q) alors que n augmentait nous avions une petite baisse des performances pour chacun des algorithmes. Cette baisse est assez faible (de l'ordre de 5 Gflops/s).

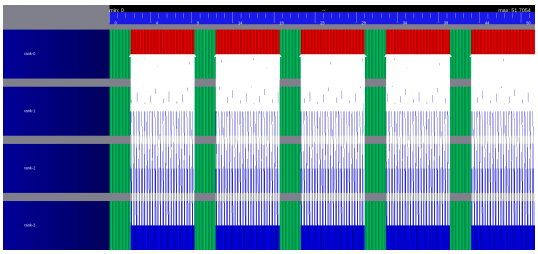
— **Scalabilité forte :** La scalabilité forte évalue l'efficacité d'un système lorsqu'on augmente le nombre de processeurs tout en gardant constant le problème à résoudre. La figure 9 illustre et montre que plus p et q augmentent plus les performances diminuent à n,m et k fixés.

6 Conclusion

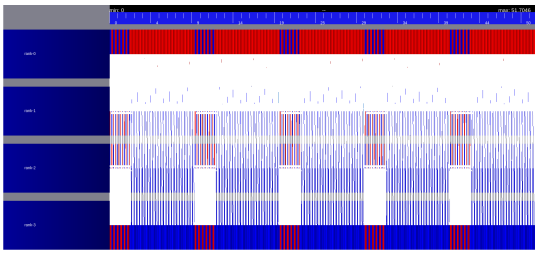
En conclusion, l'implémentation de ces trois algorithmes GEMM et l'analyse de leurs comportements nous ont permis de constater que le choix de l'algorithme et de la stratégie de communication doit dépendre de la forme et de la complexité du problème, ainsi que de l'architecture du système. Nous notons cependant une performance plutôt meilleure avec les algorithmes bcast et p2p. Cette sélection est d'une grande importance pour optimiser les performances et obtenir les meilleurs résultats possibles. La comparaison avec l'exécution séquentielle a mis en évidence l'intérêt de la parallélisation des

calculs, qui doit être employée intelligemment pour optimiser au maximum le temps d'exécution.

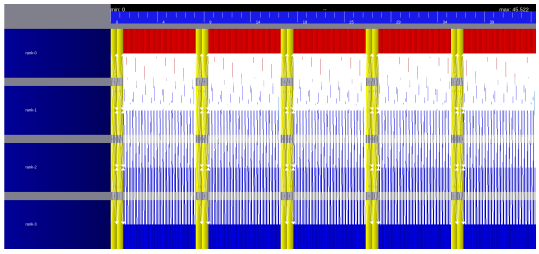
7 Annexes



(a) Algo 1



(b) Algo 2



(c) Algo 3 avec lookahead=5

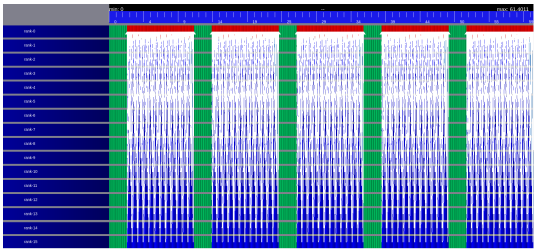
FIGURE 10 – Traces des différents algos avec b = 256 et q=p=2

La figure 10 montre les traces des communications des 3 algorithmes obtenues avec une valeur de b fixé à 256 et $p = q = 2$. Nous pouvons remarquer que l'algorithme 3 (communication paires à paires non bloquantes) prend du temps pour préparer les transmissions non bloquantes (segments jaunes), et que l'algorithme 1 a des phases de broadcast (segments verts) contrairement aux deux autres. Les trois algorithmes prennent à peu près autant de temps pour effectuer les calculs (segments rouges). Nous remarquons également que la parallélisation ici ne permet pas d'optimiser un maximum le temps d'exécution (beaucoup de segments blancs observés). Cependant, la figure 11 pour $p = q = 4$ nous permet d'avoir les même observations mais avec moins de segments blancs, et donc une meilleure optimisation du temps d'exécution.

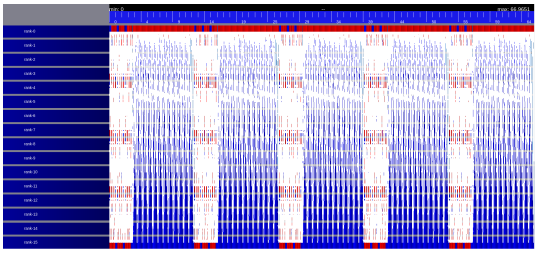
Les figures 12 et 13 montrent également les traces des communications des 3 algorithmes obtenues avec une valeur de b fixé à 128 et $p = q = 2$, $p = q = 4$ respectivement, les observations sur ces deux figures sont similaires aux figures 10 et 11 mais avec encore plus de segments blancs, car la taille du problème est plus petite dans ce cas.

Concernant les figures 14 et 15, elles montrent les traces de communications pour une valeur de b plus grande $b = 518$, où nous retrouvons encore à peu près les mêmes

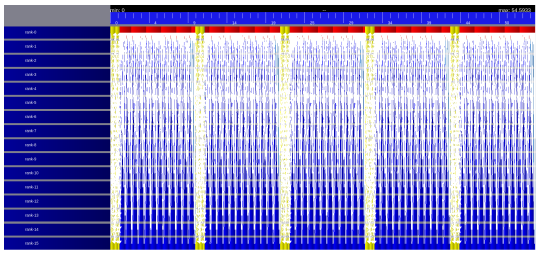
observations que sur 10 et 11 car la taille du problème est beaucoup plus grande, il y'a donc plus de travail pour les différents processus.



(a) Algo 1

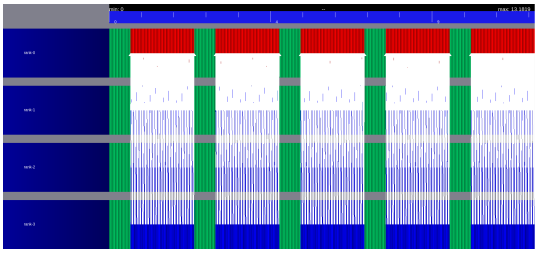


(b) Algo 2

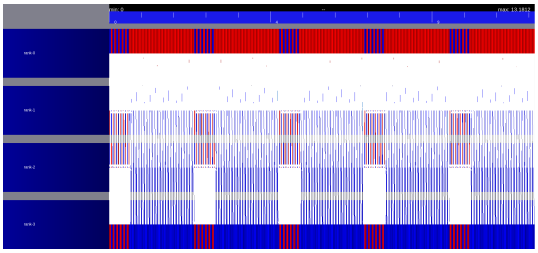


(c) Algo 3 avec lookahead=5

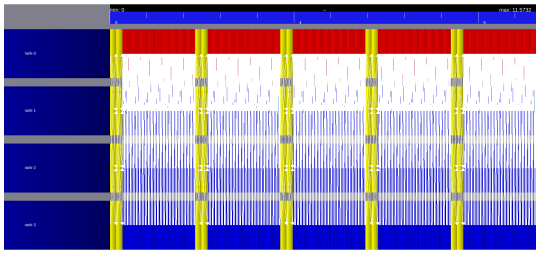
FIGURE 11 – Traces des différents algorithmes avec b = 256 et q=p=4



(a) Algo 1

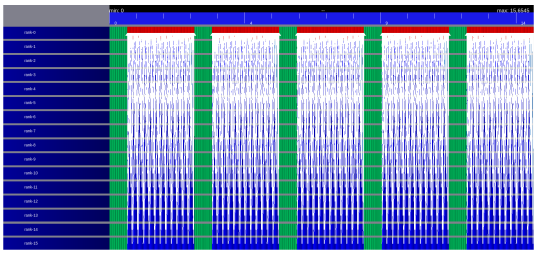


(b) Algo 2

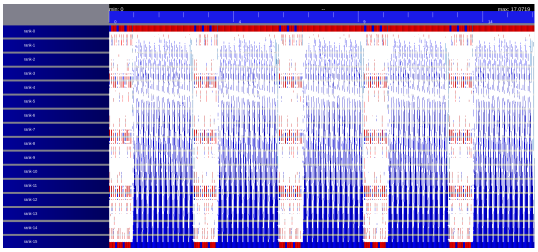


(c) Algo 3 avec lookahead=5

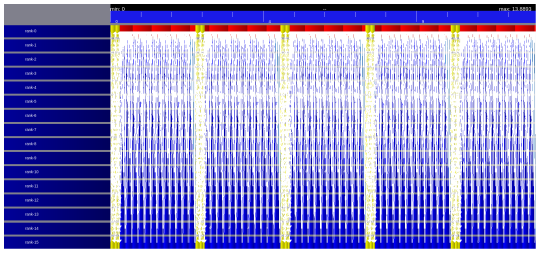
FIGURE 12 – Traces des différents algorithmes avec b = 128 et q=p=2



(a) Algo 1

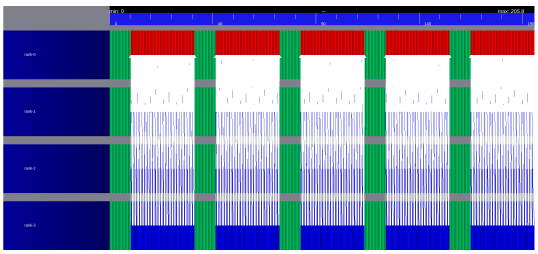


(b) Algo 2

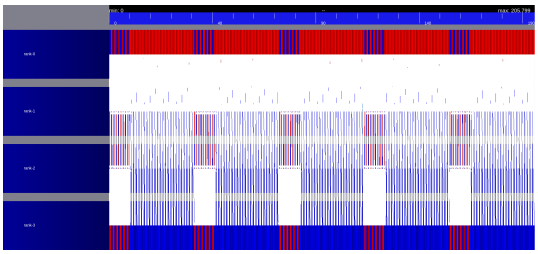


(c) Algo 3 avec lookahead=5

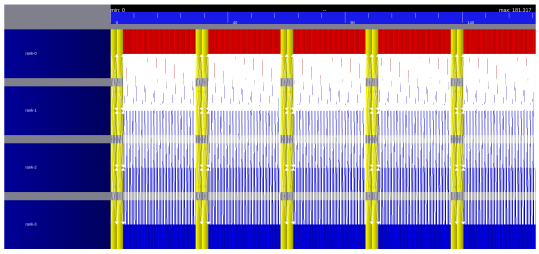
FIGURE 13 – Traces des différents algorithmes avec b = 128 et q=p=4



(a) Algo 1

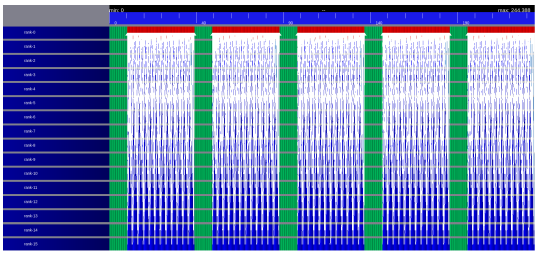


(b) Algo 2

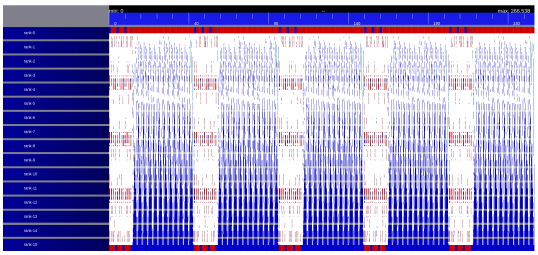


(c) Algo 3 avec lookahead=5

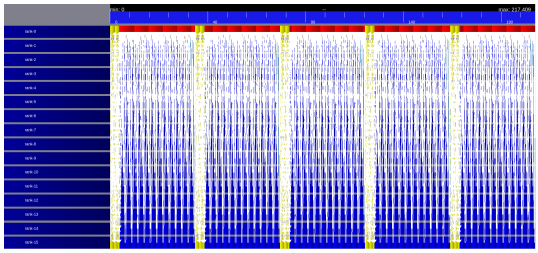
FIGURE 14 – Traces des différents algorithmes avec b = 518 et q=p=2



(a) Algo 1



(b) Algo 2



(c) Algo 3 avec lookahead=5

FIGURE 15 – Traces des différents algorithmes avec b = 518 et q=p=4