# Étude des algorithmes GEMM distribués avec MPI

HPC

**Ines BESBES, Sara ROOL**

5 ModIA

Lien Github du TP

July 6, 2025

# Contents

# 1 Introduction

La multiplication de matrices, aussi appelée GEMM (General Matrix-Matrix Multiplication), est une opération essentielle dans divers domaines scientifiques et industriels, tels que l'algèbre linéaire, l'apprentissage automatique et la simulation numérique. Pour exploiter efficacement la puissance de calcul des clusters distribués, il est central de paralléliser cette opération.

Dans ce TP, on va se concentrer sur la parallélisation de l'algorithme GEMM sur une grille de processus en mémoire distribuée, en utilisant l'interface MPI (Message Passing Interface). Pour ce faire, on va évaluer trois différentes approches d'implémentation :

- une version avec communications **point-à-point bloquantes** (`p2p`) ;
- une version avec communications **collectives bloquantes** (`bcast`) ;
- une version avec communications **point-à-point non bloquantes** et **lookahead** (`p2p-i-la`).

Les matrices sont réparties selon une distribution *2D block-cyclic* sur une grille de processus de dimensions $p \times q$. Chaque processus calcule les blocs locaux de la matrice $C = A \times B$ à partir des blocs distribués de $A$ et $B$, en suivant l'algorithme SUMMA.

L'objectif est d'analyser les performances de ces trois approches, en fonction de la taille des matrices et de la taille de la grille de processus, afin de mieux comprendre les compromis entre communications et calculs, et d'évaluer la scalabilité des solutions proposées.

# 2 Répartition des données (2D block-cyclic)

Le schéma *2D block-cyclic* permet de paralléliser la multiplication de matrices $C = A \times B$ dans un environnement distribué. Les matrices $A$, $B$ et $C$ sont découpées en blocs de taille fixe $b \times b$. Les matrices $A$ de taille $m \times k$, $B$ de taille $k \times n$ et $C$ de taille $m \times n$ sont donc représentées par des grilles de blocs de tailles respectives $(m/b) \times (k/b)$, $(k/b) \times (n/b)$, et $(m/b) \times (n/b)$.

Les blocs sont répartis sur une grille logique de processus de dimension $p \times q$. Un bloc $A_{i,j}$ est stocké sur le processus de rang :

$$\text{owner}(A_{i,j}) = q \cdot (i \mod p) + (j \mod q)$$

Cette distribution permet de bien équilibrer les blocs et de les répartir de manière régulière sur les processus.

En effet, chaque processus est responsable de stocker localement certains blocs des matrices $A$, $B$ et $C$, et est chargé de calculer les blocs locaux $C_{i,j}$ qu'il possède selon l'équation :

$$C_{i,j} = \sum_{l} A_{i,l} \cdot B_{l,j}$$

L'exécution parallèle consiste alors à distribuer efficacement les blocs nécessaires à chaque processus à chaque étape de la somme sur $l$.

# 3 Présentation des algorithmes

## 3.1 Algorithme `p2p` : communications point-à-point bloquantes

Cet algorithme correspond à une version distribuée de GEMM, où les communications sont effectuées à l'aide de primitives bloquantes : `MPI_Ssend` et `MPI_Recv`.

À chaque étape $l$, chaque bloc $A_{i,l}$ est transmis à tous les processus de la même ligne de la grille (broadcast horizontal simulé par des envois point-à-point), et chaque bloc $B_{l,j}$ à tous les processus de la même colonne (broadcast vertical).

Chaque processus :

- envoie les blocs qu'il possède sur sa ligne ou sa colonne
- attend de recevoir les blocs nécessaires de $A$ et $B$
- effectue le calcul local $C_{i,j} += A_{i,l} \cdot B_{l,j}$ pour tous ses blocs.

De fait, utiliser les communications bloquantes signifie que les transmissions doivent être terminées avant que le calcul ne puisse commencer. Cela limite le recouvrement communication-calcul et peut entraîner des temps d'attente significatifs.

### 3.2 Algorithme `bcast` : communications collectives bloquantes

Dans cette variante, les blocs $A_{i,l}$ et $B_{l,j}$ sont transmis à l'aide de l'appel collectif `MPI_Bcast`, respectivement dans des communicateurs ligne et colonne. Cela permet de remplacer les multiples appels `MPI_Ssend`/`MPI_Recv` par un seul appel collectif par ligne (pour $A$) ou colonne (pour $B$).

Les communicateurs sont construits une fois pour toutes au démarrage (via `MPI_Comm_split`) pour chaque ligne et chaque colonne de la grille.

Cette approche permet de :

- simplifier la gestion des communications
- exploiter les optimisations internes des implémentations MPI pour les communications collectives
- réduire le volume de code et potentiellement le coût de synchronisation.

Cependant, comme les appels `MPI_Bcast` sont aussi bloquants, le recouvrement entre communications et calculs reste limité.

### 3.3 Algorithme `p2p-i-la` : communications non bloquantes avec lookahead

Cette version utilise des communications point-à-point non bloquantes via `MPI_Isend` et `MPI_Irecv`, couplées à une anticipation (*lookahead*) du transfert des blocs. L'objectif est de permettre un recouvrement entre les communications et le calcul, en lançant à l'avance l'envoi et la réception des blocs nécessaires aux prochaines itérations.

L'algorithme fonctionne comme suit :

- au début, les communications des $l_{\text{init}}$ premières étapes (selon le lookahead) sont postées ;
- à chaque itération $l$, les blocs $A_{i,l}$ et $B_{l,j}$ sont utilisés dès qu'ils sont disponibles (via `MPI_Wait`) pour effectuer le calcul ;
- en parallèle, les communications pour l'étape $l +$ lookahead sont lancées.

Ce modèle pipeline les communications et le calcul, en réduisant les périodes d'inactivité. Il nécessite cependant :

- une gestion fine des requêtes MPI (via `MPI_Request`)
- une synchronisation explicite avec `MPI_Wait` pour assurer que les données sont disponibles avant utilisation
- une mémoire tampon suffisante pour stocker les blocs en avance.

Cette approche est potentiellement plus performante que les précédentes, surtout sur des architectures avec latence de communication non négligeable.

### 3.4 Visualisation et analyse des traces

Afin d'analyser le comportement temporel des différents algorithmes de GEMM distribués, nous avons utilisé l'outil `Vite` pour visualiser les traces générées par `SimGrid`. Ces traces montrent pour chaque processus les phases de communication, d'attente et de calcul.

Les trois algorithmes ont été lancés avec les mêmes paramètres pour permettre une comparaison équitable :

- **Grille de processus** : $p = 2$, $q = 2$ (soit 4 processus MPI)
- **Taille des matrices** : $n = m = k = 10$
- **Taille de bloc** : $b = 5$
- **Nombre d'itérations GEMM** : 2

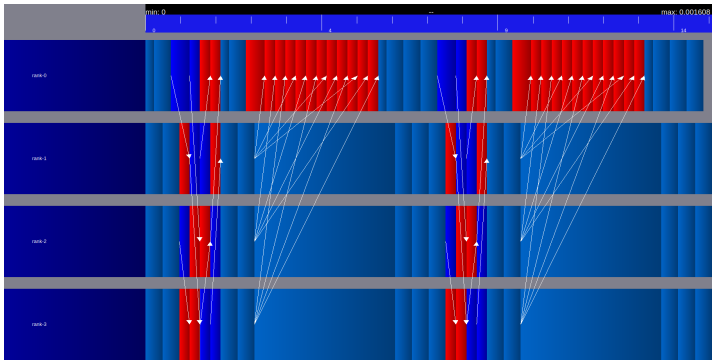

Figure 1: Trace de l'algorithme `p2p` : communications point-à-point bloquantes.

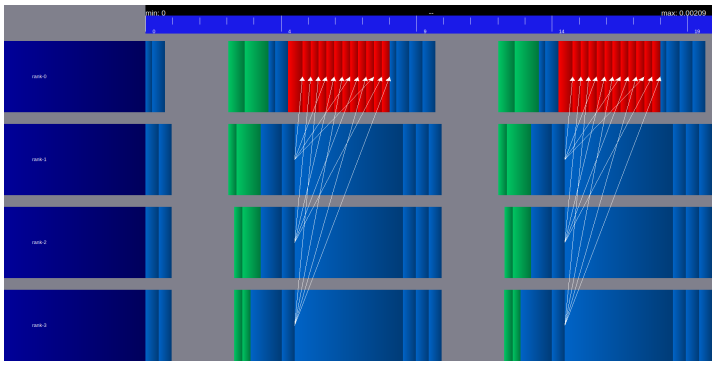

Figure 2: Trace de l'algorithme `bcast` : communications collectives bloquantes.

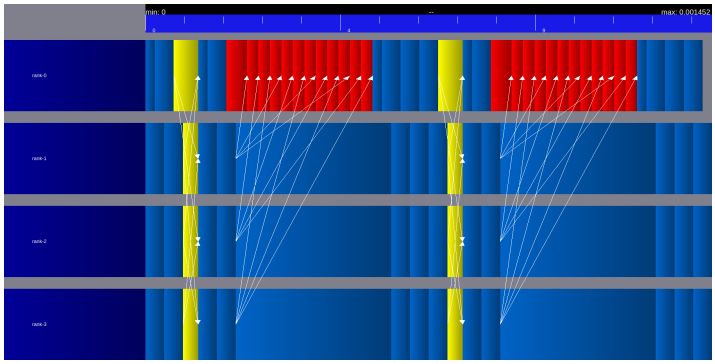

Figure 3: Trace de l'algorithme `p2p-i-la` : communications non bloquantes avec lookahead = 2.

On peut observer que :

- Dans `p2p`, les communications sont bien visibles et précèdent les calculs, sans recouvrement. Les processus communiquent séquentiellement.

- Dans `bcast`, les transmissions sont regroupées dans les appels collectifs au début. Puis le calcul reste séquentiel.
- Dans `p2p-i-la`, les blocs de communication et de calcul sont davantage entrelacés : cela confirme la présence d'un overlap entre communication et calcul, grâce au mécanisme de lookahead.

# 4 Analyse des performances

On va à présent évaluer les performances des trois algorithmes distribués, en faisant varier plusieurs paramètres importants liés à la configuration du système et à la taille du problème :

| **Paramètre** | **Description** | **Valeurs testées** |
|---|---|---|
| $p$, $q$ | Dimensions de la grille de processus MPI | $p, q \in \{1, 2, 4\}$ |
| $m$, $n$, $k$ | Dimensions des matrices d'entrée/sortie | $m = n = k \in \{512, 1024, 3072,$ $2048,\ 1536,\ 4096,\ 6144\ \}$ |
| $b$ | Taille des blocs | $b \in \{128, 256, 512\}$ |
| `lookahead` | Anticipation des communications dans `p2p-i-la` | $\in \{1, 2, ..., 12\}$ |

Table 1: Paramètres étudiés pour l'analyse expérimentale

## 4.1 Influence du *lookahead*

### 4.1.1 Analyse de l'impact du paramètre *lookahead* sur les performances pour différentes tailles de blocs $b$

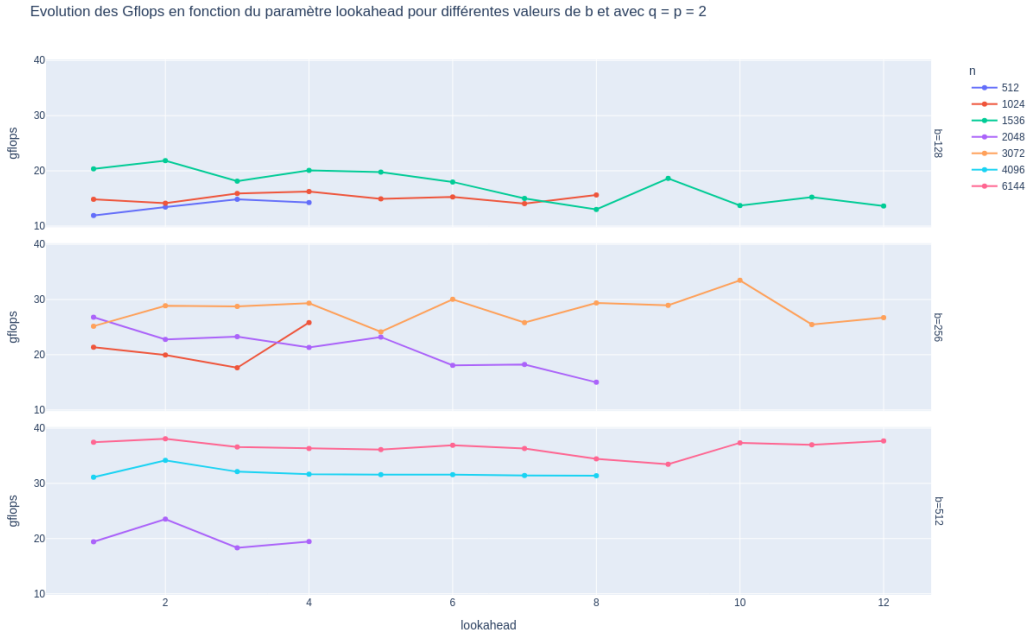


Figure 4: Evolution des Gflops en fonction du paramètre lookahead pour différentes valeurs de b (128, 256 et 512), avec q = p = 2

Pour $b = 128$, on observe que les performances (Gflops) restent relativement stables avec de petites variations. Cela suggère que le *lookahead* a peu d'impact pour cette taille de bloc.

D'autre part, quand $b = 256$, on remarque une légère augmentation des performances quand on augmente le *lookahead*, surtout pour les grandes valeurs de $n$. De fait, anticiper les communications commence à avoir un effet positif.

Pour $b = 512$, les performances sont non seulement plus élevées, mais mais elles tendent également à s'améliorer avec l'augmentation *lookahead*, surtout pour les grandes valeurs de $n$. Ainsi, l'anticipation des communications est plus bénéfique pour des blocs plus grands.

En outre, on observe que, pour toutes les tailles de blocs, les performances augmentent généralement avec la taille de la matrice $n$. En effet, les opérations sur des matrices plus grandes peuvent mieux exploiter le parallélisme et les optimisations de calcul. En particulier, on observe pour $n = 6144$ une meilleure amélioration des performances avec l'augmentation du *lookahead*. Cela indique que l'anticipation des communications est plus efficace pour les grands problèmes.

#### 4.1.2 Recherche du meilleur *lookahead* pour différentes combinaisons de $p$ et $q$

Comme on a vu précédemment, on obtient de meilleures performances avec $b = 512$ et $n = 6144$. On a fixé ces valeurs pour trouver le meilleur *lookahead* selon $p$ et $q$.

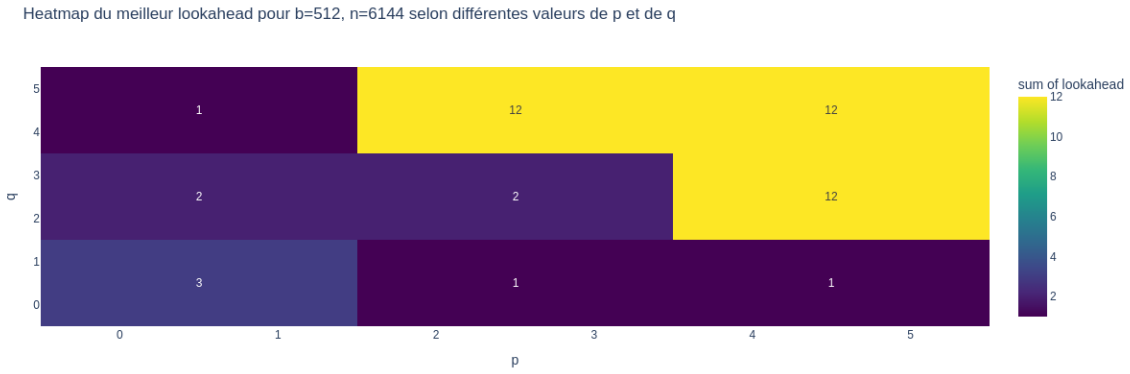


Figure 5: Meilleur lookahead selon p et q

Pour des configurations simples comme $p = 1$ et $q = 1$, le *lookahead* optimal est faible, car il n'y a pas de communication inter-nœuds à anticiper. Cependant, pour des grilles plus grandes, comme $p = 2$ et $q = 2$, ou $p = 3$ et $q = 3$, le *lookahead* optimal augmente légèrement. Ceci indique que l'anticipation des communications commence à jouer un rôle dans l'optimisation des performances.

Pour des grilles encore plus grandes, comme $p = 4$ et $q = 4$, ou $p = 5$ et $q = 5$, le *lookahead* optimal devient plus élevé.

Par la suite, on fixe *lookahead* de façon à garder la meilleure performance pour chaque combinaison de paramètre.

### 4.2 Influence de la grille de processus $p$ et $q$

#### 4.2.1 Cas $b = 128$

On observe que lorsque $p = 1$, les performances sont relativement faibles, car la parallélisation est limitée à une seule ligne de processus, ce qui augmente la charge de communication verticale.

À $p = 1$ et $q = 1$, les Gflops atteignent environ 18 pour l'algorithme `p2p-i-la`, contre seulement 10 à 12 pour `p2p` et `bcast`. En augmentant $p$ et $q$, les performances augmentent. En effet, à $p = 2$, $q = 2$, et $n = 1536$, `p2p-i-la` dépasse les 22 Gflops, tandis que `p2p` et `bcast` plafonnent autour de 18/19 Gflops. À $p = 4$, $q = 4$, on atteint près de 24 Gflops pour `p2p-i-la`.

Cela montre que même avec de petits blocs ($b = 128$), l'anticipation des communications dans `p2p-i-la` permet d'obtenir un meilleur recouvrement communication/calcul.

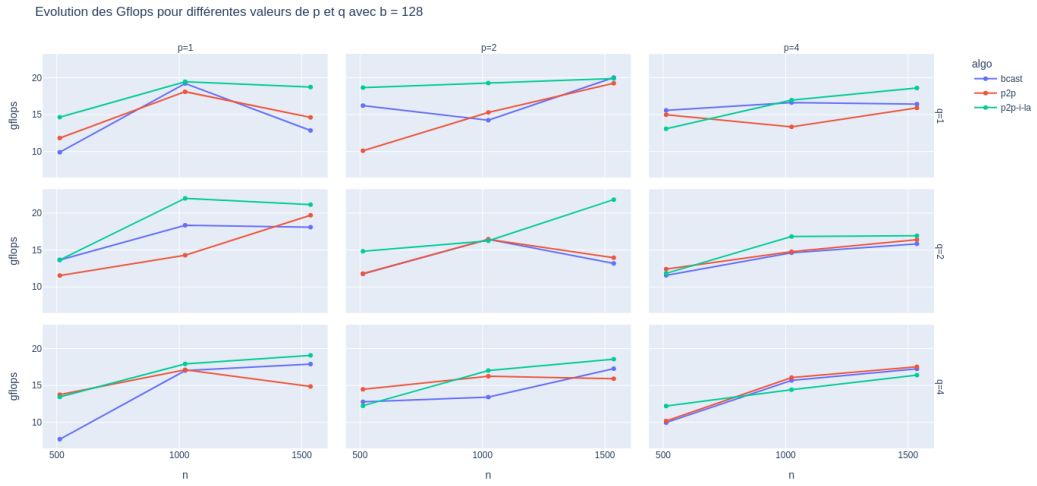


Figure 6: Evolution des Gflops en fonction de $p$ et $q$, avec $b = 128$

### 4.2.2 Cas $b = 256$


Figure 7: Evolution des Gflops en fonction de $p$ et $q$, avec $b = 256$

Dans le cas $b = 256$, on observe une amélioration globale des performances par rapport à $b = 128$. En effet, augmenter la taille des blocs améliore la localité des calculs, ce qui se traduit par un plus grand nombre d'opérations en virgule flottante (flops) par communication.

À mesure que $p$ augmente (de 1 à 4), on observe une montée régulière des Gflops, ce qui reflète une meilleure scalabilité. En effet, à $p = 1$, $q = 1$, et $n = 3072$, `p2p-i-la` atteint environ 30 Gflops, tandis que `p2p` se stabilise à 27 Gflops, et `bcast` autour de 29. Pour des configurations plus parallèles ($p = 2$, $q = 2$), `p2p-i-la` monte à plus de 33 GFLOPS pour $n = 3072$, alors que `p2p` et `bcast` varient entre 25 et 30.

Ainsi, le gain apporté par le lookahead est ici plus visible, car la taille de bloc permet un meilleur ratio calcul/communication.

### 4.2.3 Cas $b = 512$

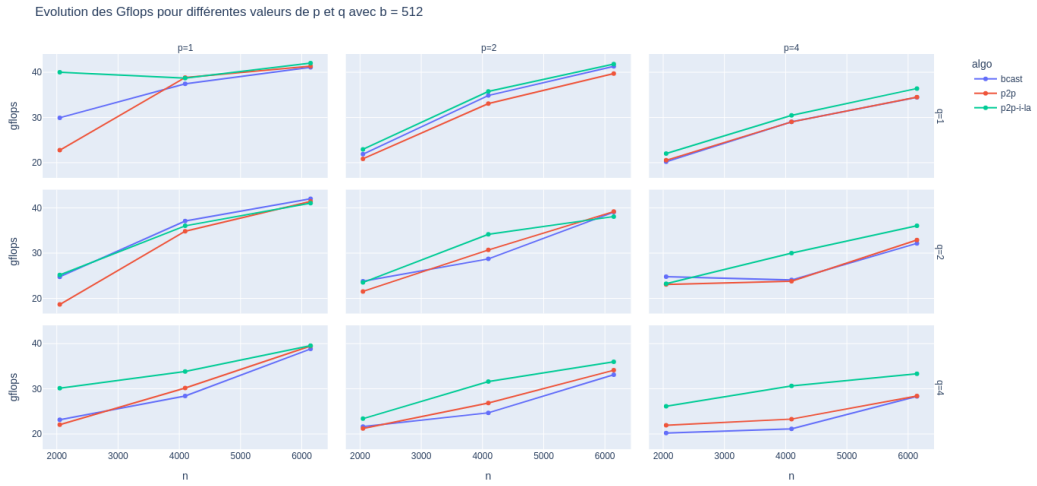


Figure 8: Evolution des Gflops en fonction de $p$ et $q$, avec $b = 512$

Avec $b = 512$ , la taille des blocs plus importante réduit l'impact de la communication au profit du calcul. Cela permet d'atteindre les meilleures performances parmi les trois cas testés.

Encore une fois, l'algorithme `p2p-i-la` est le meilleur : il tire pleinement parti de l'overlap communication/calcul, surtout pour les grilles de plus grande taille ($p = 4$, $q = 4$). `bcast` reste légèrement en retrait dans la majorité des cas, probablement en raison des temps de synchronisation globaux inhérents à l'appel `MPI_Bcast`. Enfin, `p2p` reste compétitif mais montre parfois des limites en scalabilité.

Ainsi, il est important de trouver un bon compromis entre coût de communication (lié à $b$) et parallélisme (lié à $p$ et $q$), tout en tenant compte de l'efficacité du lookahead dans les configurations fortement parallèles.

## 4.3 Influence de la taille du bloc $b$

### 4.3.1 Cas $p = q = 2$

Pour une petite grille (4 nœuds), l'augmentation de la taille du bloc $b$ améliore significativement les performances, en particulier pour les plus grandes tailles de problème $n$.

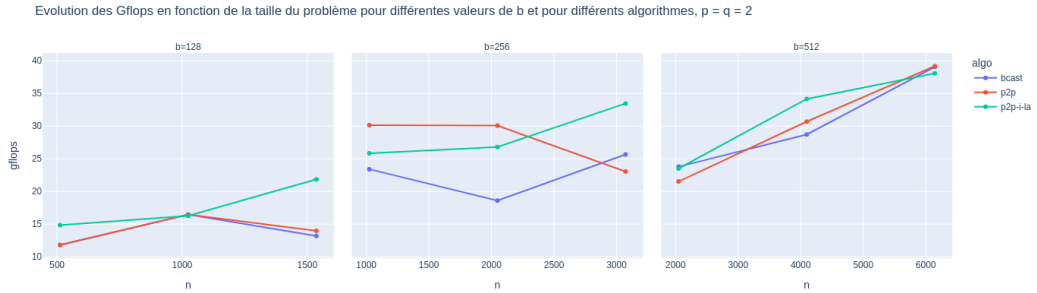


Figure 9: Evolution des Gflops en fonction de $b$ dans le cas où $p = q = 2$

Avec $b = 128$, les Gflops restent relativement modestes. En effet, pour $n = 1500$, les algorithmes atteignent entre 13 et 22 Gflops. Cela s'explique par le fait que les blocs sont petits, ce qui entraine un grand nombre de communications MPI fragmentées. Chaque processus doit échanger de nombreux messages, ce qui augmente la latence et réduit l'efficacité du calcul.

En passant à $b = 256$, les performances augmentent significativement. Quand $n = 3000$, les Gflops sont entre 23 et 34. Cette amélioration montre que des blocs de taille moyenne permettent de mieux amortir les communications sur des volumes de données plus conséquents. La bande passante est donc mieux exploitée.

Lorsque $b$ est augmenté à 512, les performances sont entre 38 et 39 GFLOPS pour $n = 6000$. Cela montre que les grands blocs permettent de maximiser l'utilisation du CPU sur des volumes de données importants, tout en réduisant le coût des échanges de données entre les processus.

### 4.3.2 Cas $p = q = 4$

Avec une grille de processus plus large ($4 \times 4$), l'influence de la taille des blocs devient encore plus marquée.

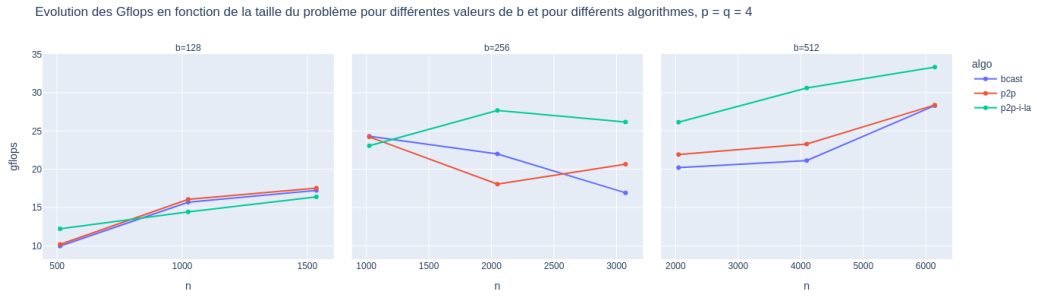


Figure 10: Evolution des Gflops en fonction de $b$ dans le cas où $p = q = 4$

Pour $b = 128$, les performances sont également modestes. Pour $n = 1500$, les meilleurs algorithmes atteignent à peine 17 à 18 Gflops. Cela s'explique par une surcharge de communication accentuée par le nombre plus important de processus. Les petits blocs génèrent de très nombreux messages MPI, ce qui rend la coordination entre processus coûteuse. Cela pénalise les performances globales.

Quand $b = 256$, l'algorithme `p2p-i-la` atteint 28 Gflops dès $n = 2000$, et maintient des performances élevées (autour de 26–27 Gflops) même pour des tailles plus grandes comme $n = 3000$. En augmentant la taille des blocs, on optimise l'utilisation des capacités de calcul des cœurs tout en diminuant la fréquence des communications, un aspect qui devient crucial avec l'augmentation du nombre de processus.

Enfin, avec $b = 512$, les performances sont meilleures. Pour $n = 6000$, `p2p-i-la` atteint 34 Gflops, surpassant nettement `p2p` et `bcast`, qui sont à environ 28–29 Gflops. Cette progression confirme que les grands blocs améliorent la scalabilité de l'application en diminuant l'impact relatif des communications dans un environnement fortement parallèle.

## 4.4 Influence de la platform

La platform correspond à l'infrastructure matérielle et logicielle sur laquelle le code MPI est exécuté, incluant le cluster, la topologie réseau et les ressources de calcul disponibles. Le graphique ci dessous compare les performances pour deux plateformes distinctes : *cluster_barcross* et *cluster_fat_tree*. Dans les parties précédentes, nous avions utilisé seulement *cluster_barcross*.

Pour les décrire, la topologie crossbar connecte directement chaque noeud à tous les autres à travers une matrice de commutation, ce qui permet des communications directes mais peut rapidement devenir coûteux ou saturé à grande échelle. En revanche, la topologie fat tree est une structure hiérarchique en arbre avec des liaisons plus larges en haut de l'arbre, permettant une meilleure répartition de la bande passante et réduisant les risques de congestion. Elle est particulièrement efficace pour les communications collectives fréquentes.

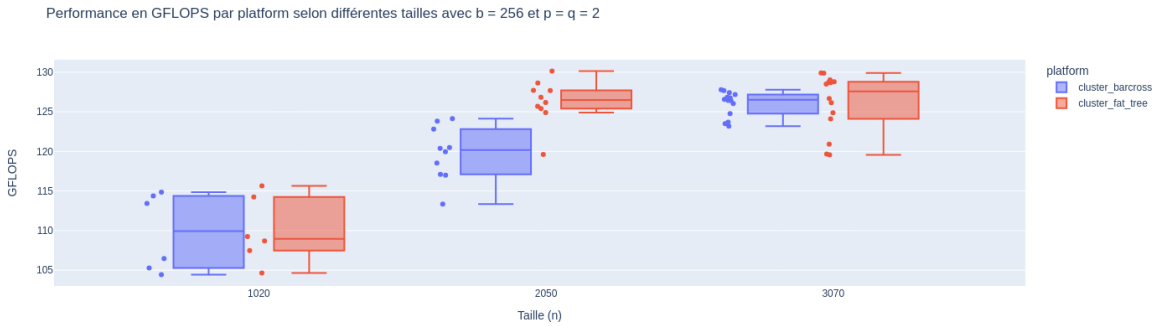


Figure 11: Evolution des Gflops en fonction de $b$ dans le cas où $p = q = 4$

On fixe ici $b = 256$, $p = q = 2$. On observe que, pour de petites tailles, les performances sont comparables, mais dès $n = 2048$, le cluster à topologie fat tree offre de meilleures performances, plus élevées et plus stables. Cette différence s'explique par la topologie réseau : la structure fat tree permet une meilleure bande passante et une communication plus efficace entre les noeuds, ce qui est particulièrement avantageux pour des calculs intensifs en communication comme pour GEMM.

# 5 Analyse de la scalabilité

# 6 Conclusion