Valérian Lambert
Julie Leroy
Arsene Jerome
Thomas Madrange-Maire
Clément Berard

Compte rendu - Scalabilité des programme Prime et MonteCarlo

sae 3e année 2025-2026

Sommaire

Prime

Dans cette partie,nous allons évaluer les performances des codes prime prime_improve, prime_DLB et prime_DLB_improve
Le code prime cherche tous les nombre premier de 2 à N. Dans une expérience On appelle plusieurs processus dont un est désigné comme étant le master.
Les paramètres du RPI:

- 4 rpi 0,
- 4 processeurs logiques

Les critères de qualité :

- Performance Efficiency de Product Quality de la norme ISO/IEC 25010 qui décrit la performance par rapport à la quantité de ressources utilisée dans les conditions énoncées.
- Efficiency de Quality in use de la norme ISO/IEC 25022 qui décrit la qualité à l'utilisation des ressources du logiciel ou du système

Métrique:
Pour la métrique ce qui est utilisé c'est Time efficiency définie dans ISO/IEC 25022.
Elle est calculé par Tt (time target)/ Ta(actual time)
Le speedup (calculé par la formule Sp = T1/TP où p est un nombre de processeur) permet de mesurer la scalabilité forte.
L'efficacité faible (calculé par la formule Ew(p) = T1/Tp avec :

T1 : le temps de calcul avec un processeur et taille n
Tp : le temps de calcul avec p processeur et taille n*p)

L'efficacité faible idéal est Ew(p) = 1

La scalabilité forte décrit la capacité d'un programme ou d'un système à effectuer une tâche de plus en plus rapidement avec un problème fixé. Le speedup idéal dans ce cas-ci est Sp = p. La scalabilité faible quant à elle décrit la capacité d'un programme ou d'un système à effectuer une tâche et à maintenir un temps d'exécution stable lorsque la taille du problème augmente proportionnellement au nombre de processus. Le speedup idéal dans ce cas-ci est Ew(p) = 1.

Scalabilité forte

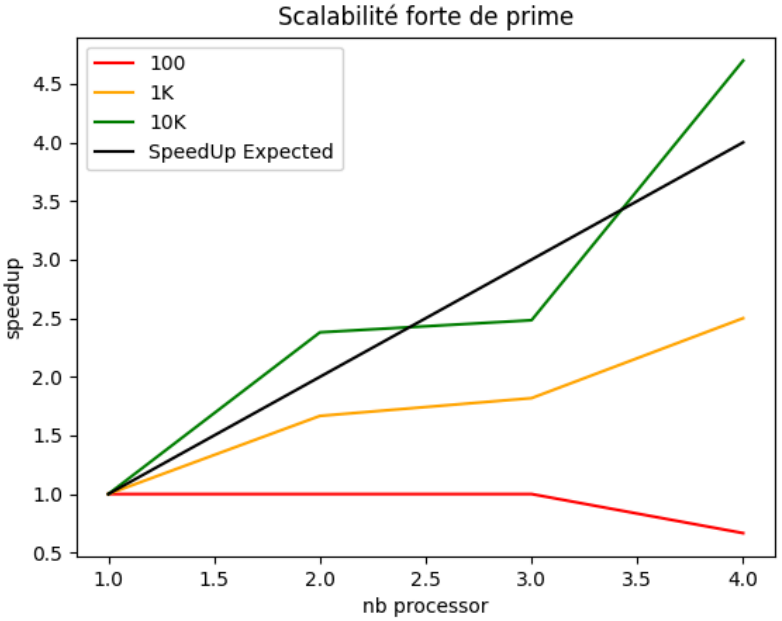
Ce chapitre,pour chaque fichier traité, affiche les données correspondant au résultat de 3 ou 4 expériences de scalabilité forte d'un programme sous le format suivant :

[nombre final, nombre de processus, temps de calcul en seconde]

Ensuite, il affiche le graphique des résultats pour ce programme.
Enfin, les moyennes de l'ensemble de speedup sont mises en commun dans un seul et même graphique pour pouvoir les comparer.

prime.py

```
"t1": [
        [100, 1, 0.02],
        [100, 2, 0.02],
        [100, 3, 0.02],
        [100, 4, 0.03],],
"t2": [
        [1000, 1, 0.2],
        [1000, 2, 0.12],
        [1000, 3, 0.11],
        [1000, 4, 0.08]],
"t3": [
        [10000, 1, 21.75],
        [10000, 2, 9.14],
        [10000, 3, 8.76],
        [10000, 4, 4.63]]
```



On remarque que pour la courbe rouge, au lieu d'augmenter, le SpeedUp baisse.
Pour la courbe verte, le speedUp est plus haut que la courbe noire lorsque le nombre de processus est à 2 et 4.

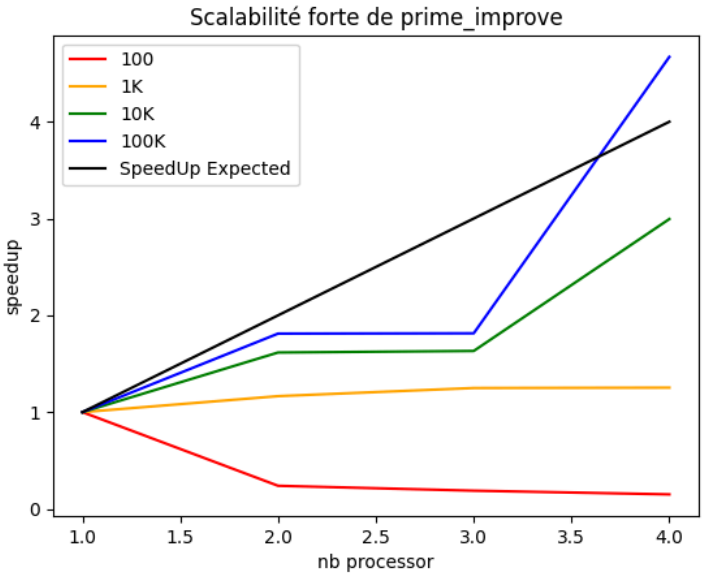
Pour les petites tailles de problème (N = 100 et 1000), le speedup reste proche ou inférieur à 1. Cela s'explique par le fait que le temps de calcul est trop faible pour amortir les coûts de création des processus et de communication.

Pour N = 10000, on observe une amélioration progressive du speedup lorsque le nombre de processus augmente, bien que la courbe reste éloignée du speedup idéal. Cette version naïve souffre d'un coût algorithmique élevé et d'une mauvaise exploitation du parallélisme.

La version Prime de base n'est pas adaptée à une montée en charge efficace, car chaque test de primalité est coûteux et peu optimisé.

prime_improve.py

```
"t1": [
        [100, 1, 0.005],
        [100, 2, 0.0209],
        [100, 3, 0.0265],
        [100, 4, 0.0332]],
"t2": [
        [1000, 1, 0.0416],
        [1000, 2, 0.0357],
        [1000, 3, 0.0333],
        [1000, 4, 0.0332]],
"t3": [
        [10000, 1, 0.5849],
        [10000, 2, 0.3619],
        [10000, 3, 0.3584],
        [10000, 4, 0.1953]],
"t4": [
        [100000, 1, 15.505],
        [100000, 2, 8.5589],
        [100000, 3, 8.5453],
        [100000, 4, 3.3201], ]
```



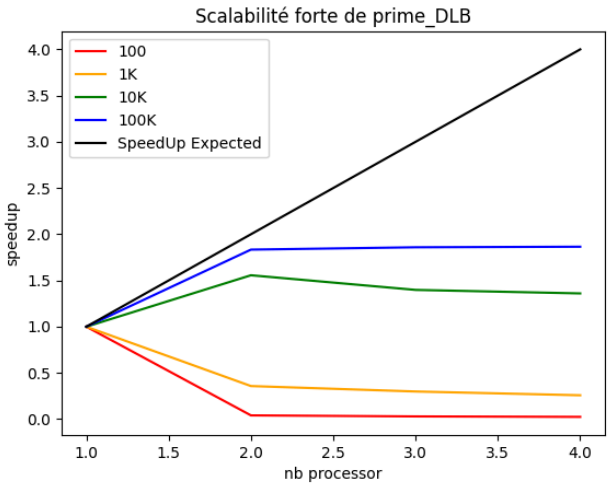
On remarque que le speedUp de la courbe rouge baisse tandis que le speedUp de la courbe bleu augmente plus haut que la courbe noire avec 4 processus.

La version améliorée présente de meilleurs temps séquentiels et une montée en charge plus cohérente pour les grandes tailles de N. Le speedup devient significatif à partir de N = 10 000 et surtout 100 000, ce qui montre que la réduction de la complexité algorithmique (test jusqu'à √N) améliore directement l'efficacité du parallélisme.

Prime_improve est clairement plus performant que Prime, mais reste limité par une répartition statique du travail, ce qui empêche d'atteindre une scalabilité idéale.

prime_DLB.py

"t1": [
[100, 1, 0.0014400482177734375],
[100, 2, 0.03546476364135742],
[100, 3, 0.048842668533325195],
[100, 4, 0.058873891830444336],],
"t2": [
[1000, 1, 0.015547990798950195],
[1000, 2, 0.04344987869262695],
[1000, 3, 0.052011728286743164],
[1000, 4, 0.0602116584777832],],
"t3": [
[10000, 1, 0.21133923530578613],
[10000, 2, 0.13579154014587402],
[10000, 3, 0.15116572380065918],
[10000, 4, 0.1553936004638672],],
"t4": [
[100000, 1, 3.337921380996704],
[100000, 2, 1.8201813697814941],
[100000, 3, 1.7954185009002686],
[100000, 4, 1.7897183895111084],]



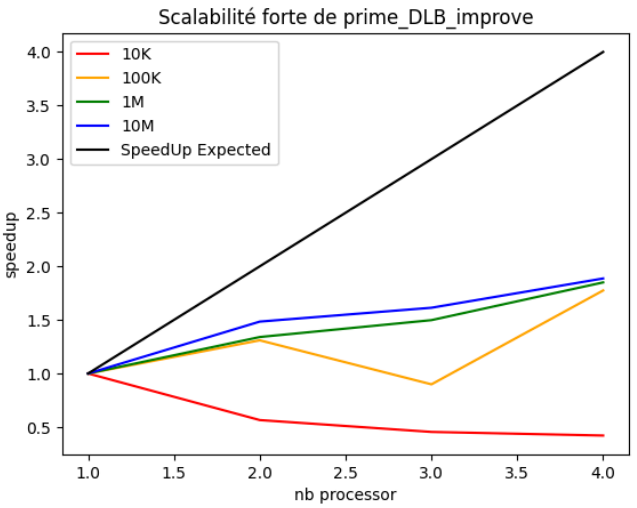
On remarque que le speedUp pour la courbe rouge atteint presque le 0 à partir de 2 processus tandis que la courbe bleu n'atteint pas 2. En résumé, aucune courbe ne se rapproche de la courbe idéale dans le long terme

Les résultats montrent un speedup faible, voire inférieur à celui de Prime_improve. Cela peut sembler contre-intuitif, mais s'explique par le fait que le Dynamic Load Balancing introduit un surcoût non négligeable (en communication des données) dans un cluster homogène, où tous les workers ont des performances similaires.

Le DLB n'apporte pas de bénéfice significatif dans ce contexte précis. En revanche, cette implémentation serait plus efficace sur un cluster hétérogène (par exemple en intégrant le RPI4 dans les calculs) , où les différences de puissance entre nœuds justifieraient un équilibrage dynamique.

prime_DLB_improve.py

"t1": [
[10000, 1, 0.028735637664794922],
[10000, 2, 0.050812721252441406],
[10000, 3, 0.06317591667175293],
[10000, 4, 0.06822466850280762],],
"t2": [
[100000, 1, 0.2630581855773926],
[100000, 2, 0.20087838172912598],
[100000, 3, 0.2926511764526367],
[100000, 4, 0.14823150634765625],],
"t3": [
[1000000, 1, 2.523214101791382],
[1000000, 2, 1.8826513290405273],
[1000000, 3, 1.6845901012420654],
[1000000, 4, 1.3639791011810303],],
"t4": [
[10000000, 1, 29.687586545944214],
[10000000, 2, 20.0024516582489],
[10000000, 3, 18.40053939819336],
[10000000, 4, 15.735610008239746],]



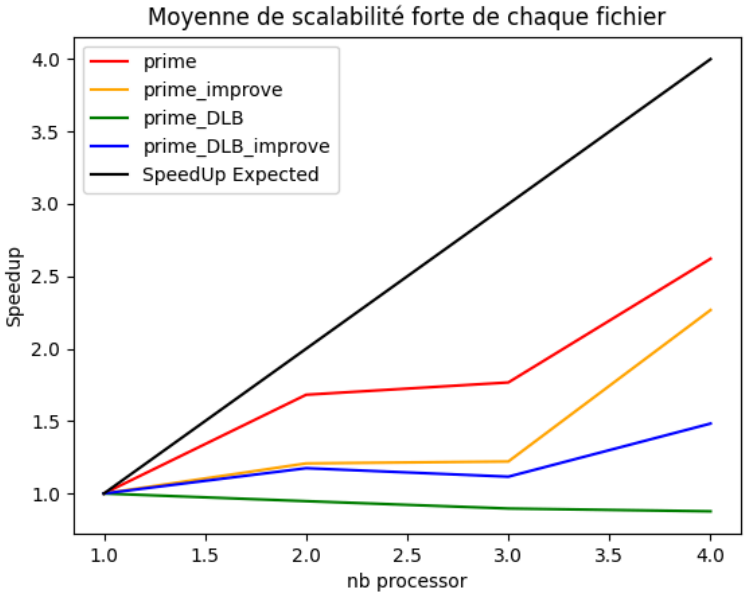
On remarque que les courbes n'atteignent pas 2 en speedUp et que la courbe rouge se rapproche de 0.5 en SpeedUp. En résumé, aucun ne se rapproche de la courbe idéale dans le long terme.

Cette version obtient les meilleurs temps de calcul globaux. Le speedup reste toutefois inférieur à l'idéal, principalement en raison :

- des coûts de synchronisation,
- de l'accès partagé aux structures de données,
- et des limites du parallélisme sur le crible.

Prime_DLB_improve est la version la plus rapide de loin en termes de temps, mais la nature même du crible d'Ératosthène limite le gain de performance lorsque le nombre de processus augmente.

Moyenne de la scalabilité forte



Conclusion - Scalabilité forte Prime

Aucune implémentation n'atteint le speedup idéal, ce qui est attendu compte tenu :

- des surcoûts du parallélisme,
- de la granularité du problème,
- et de l'architecture homogène utilisée.

Cependant, une hiérarchie claire se dégage (en temps et non en speedup) :

Prime < Prime_improve ≈ Prime_DLB < Prime_DLB_improve

Les améliorations algorithmiques ont un impact bien plus important que les mécanismes d'équilibrage dynamique dans ce contexte.

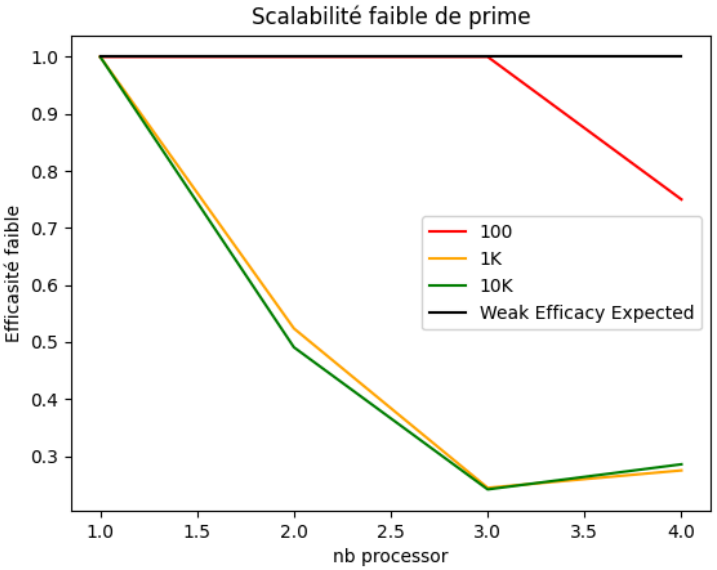
Scalabilité faible

Dans ce chapitre,pour chaque fichier, on affiche les données correspondent au résultat de 3 ou 4 expérience d'un fichier sur la scalabilité faible sous le format suivant :
[nombre final, nombre de processus, temps de calcul en seconde]
Ensuite, on affiche le graphique des résultats du fichier.

Enfin, en fin de chapitre on affiche la moyenne de l'efficacité faible de chaque fichier dans un seul graphique

prime.py

"t1": [
[100, 1, 0.03],
[200, 2, 0.03],
[300, 3, 0.03],
[400, 4, 0.04],],
"t2": [
[1000, 1, 0.22],
[2000, 2, 0.42],
[3000, 3, 0.9],
[4000, 4, 0.8],],
"t3": [
[10000, 1, 17.58],
[20000, 2, 35.82],
[30000, 3, 72.73],
[40000, 4, 61.53],]



On remarque que toutes les lignes baisse très vite et qu'aucun ne reste près de 1 avec 4 processus

Les résultats montrent une dégradation rapide de l'efficacité faible lorsque le nombre de processus augmente. Pour toutes les tailles de problème testées, l'efficacité s'éloigne fortement de la valeur idéale dès 2 processus et chute encore davantage à 3 et 4 processus.

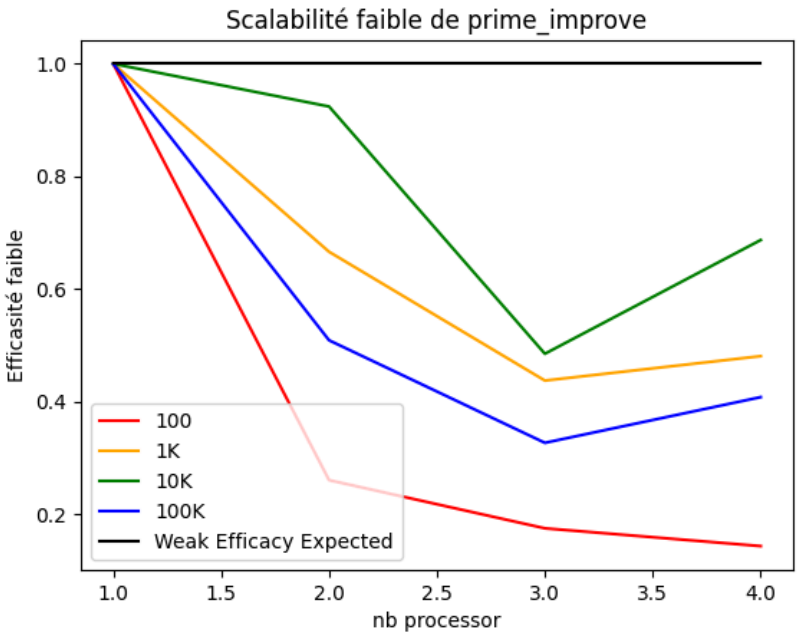
Cette baisse est principalement due :

- à la complexité algorithmique élevée de la version naïve,
- à une répartition statique et déséquilibrée de la charge de travail,
- aux surcoûts de communication, qui deviennent dominants lorsque la taille du problème augmente.

La version Prime de base ne parvient pas à maintenir un temps d'exécution stable lorsque la charge augmente proportionnellement au nombre de processus. Elle constitue donc un mauvais candidat à la scalabilité faible.

Prime_improve.py

```
"t1": [
        [100, 1, 0.005],
        [200, 2, 0.0192],
        [300, 3, 0.0286],
        [400, 4, 0.0349]],
"t2": [
        [1000, 1, 0.0373],
        [2000, 2, 0.056],
        [3000, 3, 0.0853],
        [4000, 4, 0.0776]],
"t3": [
        [10000, 1, 0.6943],
        [20000, 2, 0.7513],
        [30000, 3, 1.4318],
        [40000, 4, 1.0107]],
"t4": [
        [100000, 1, 10.7814],
        [200000, 2, 21.1875],
        [300000, 3, 33.0022],
        [400000, 4, 26.4335]]
```



On remarque que 3 des 4 courbes remontent à partir de 3 processus après avoir chuté à partir de 1 processus

(test jusqu'à √N)

Cette version présente une efficacité faible légèrement supérieure à celle de prime.py. Pour certaines expériences, on observe une stabilisation, voire une légère remontée de l'efficacité à partir de 3 processus, notamment pour les tailles intermédiaires.

Cependant, malgré l'optimisation algorithmique, l'efficacité reste globalement inférieure à 1 et décroît lorsque le nombre de processus augmente.

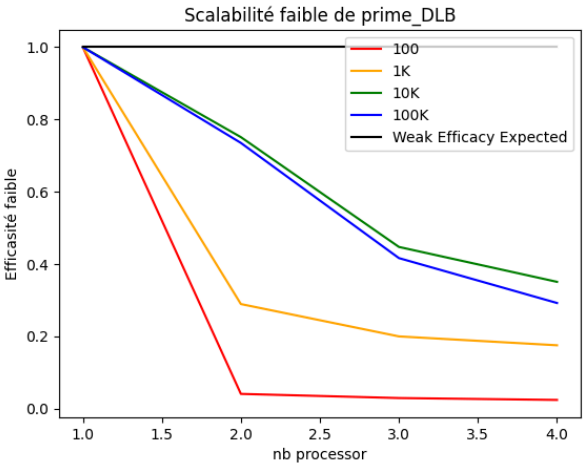
La réduction de la complexité algorithmique améliore la situation, mais ne suffit pas à compenser :

- la croissance du volume de données,
- le coût de la communication inter-processus,
- et le déséquilibre de la charge.

Prime_improve reste donc limité en scalabilité faible, bien qu'il soit plus performant que la version naïve.

Prime_DLB.py

"t1": [
[100, 1, 0.0014691352844238281],
[200, 2, 0.035928964614868164],
[300, 3, 0.049948930740356445],
[400, 4, 0.060632944107055664],],
"t2": [
[1000, 1, 0.015206098556518555],
[2000, 2, 0.05255389213562012],
[3000, 3, 0.07607054710388184],
[4000, 4, 0.0867156982421875],],
"t3": [
[10000, 1, 0.20366525650024414],
[20000, 2, 0.27117013931274414],
[30000, 3, 0.4552922248840332],
[40000, 4, 0.580535888671875],],
"t4": [
[100000, 1, 3.4519975185394287],
[200000, 2, 4.696336030960083],
[300000, 3, 8.28937292098999],
[400000, 4, 11.808725118637085],]



Dans ce graphique, on remarque que toutes les courbes baissent fortement et la courbe rouge se rapproche dangereusement de 0.

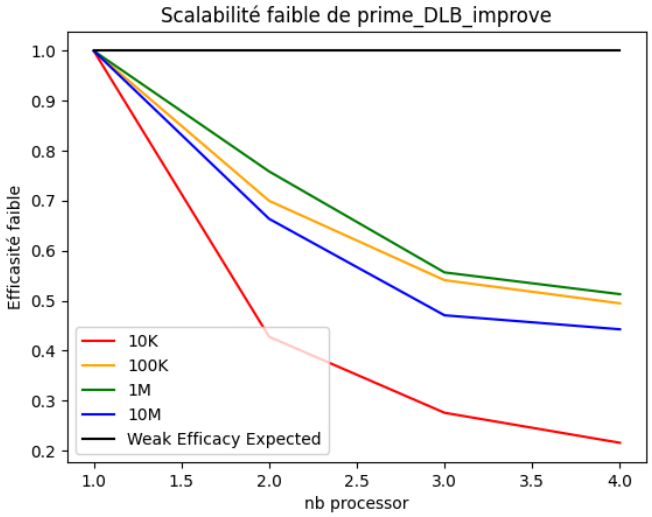
Les résultats montrent une baisse marquée de l'efficacité faible, parfois plus importante que pour prime_improve. Cette dégradation s'accentue lorsque le nombre de processus augmente.

Le mécanisme de DLB introduit un surcoût supplémentaire lié à la gestion dynamique des tâches, qui n'est pas compensé dans un environnement homogène où les processus ont des performances similaires.

Dans ce contexte expérimental, le DLB n'apporte aucun bénéfice pour la scalabilité faible. Il aggrave même la situation en augmentant les coûts de synchronisation et de communication.

Prime_DLB_improve.py

"t1": [
[10000, 1, 0.02992105484008789],
[20000, 2, 0.06991958618164062],
[30000, 3, 0.10847663879394531],
[40000, 4, 0.1386864185333252],],
"t2": [
[100000, 1, 0.26768994331359863],
[200000, 2, 0.3824474811553955],
[300000, 3, 0.4947214126586914],
[400000, 4, 0.5410463809967041],],
"t3": [
[1000000, 1, 2.804786205291748],
[2000000, 2, 3.6971659660339355],
[3000000, 3, 5.039065599441528],
[4000000, 4, 5.466469764709473],],
"t4": [
[10000000, 1, 28.82850670814514],
[20000000, 2, 43.42612099647522],
[30000000, 3, 61.231879472732544],
[40000000, 4, 65.10547947883606],]



On remarque que toutes les courbes baissent fortement.

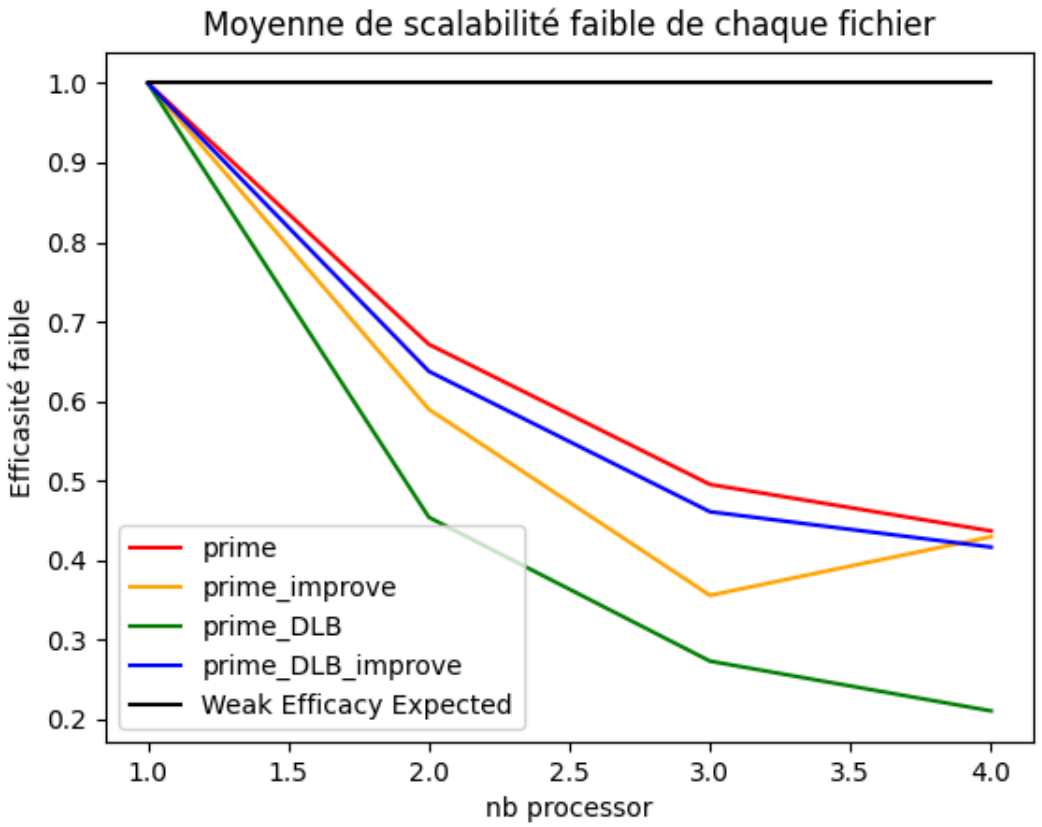
Cette version obtient les meilleures performances globales parmi les quatre implémentations. L'efficacité faible reste cependant inférieure à la valeur idéale et décroît progressivement avec l'augmentation du nombre de processus.

La baisse observée s'explique par :

- la nécessité de partager ou synchroniser des structures de données liées au crible,
- l'augmentation des communications lorsque la taille du problème croît,
- les limites intrinsèques du parallélisme du crible d'Ératosthène.

Bien que Prime_DLB_improve soit la version la plus rapide, elle ne parvient pas à maintenir une efficacité faible proche de 1. Cela montre que l'optimisation algorithmique ne garantit pas une bonne scalabilité faible.

Moyenne de la scalabilité faible



Ainsi, toutes les courbes baissent fortement car la répartition de la charge de travail pour chaque worker est strictement inégale.

Conclusion - Scalabilité faible Prime

L'analyse moyenne de l'efficacité faible met en évidence une tendance commune :
Toutes les implémentations voient leur efficacité décroître lorsque le nombre de processus augmente.

La hiérarchie suivante se confirme (en temps pas en speedup) :

Prime < Prime_improve ≈ Prime_DLB < Prime_DLB_improve

Cependant, même la meilleure version reste éloignée de l'efficacité idéale, ce qui souligne que le problème du calcul des nombres premiers, dans ce cadre expérimental, ne se prête pas à la scalabilité faible.

MonteCarlo

Dans cette partie, nous allons évaluer les performances des code Monte Carlo avec l'implémentation MPI, JavaSocket et PythonSocket.
Dans les expériences de JavaSocket et PythonSocket, le Master aura 4 Worker et
Les paramètres de la machine sont toujours :

- 4 rpi 0
- 4 processeurs logiques

Les critères de qualité sont toujours les mêmes utilisés dans la partie Prime.

Scalabilité forte

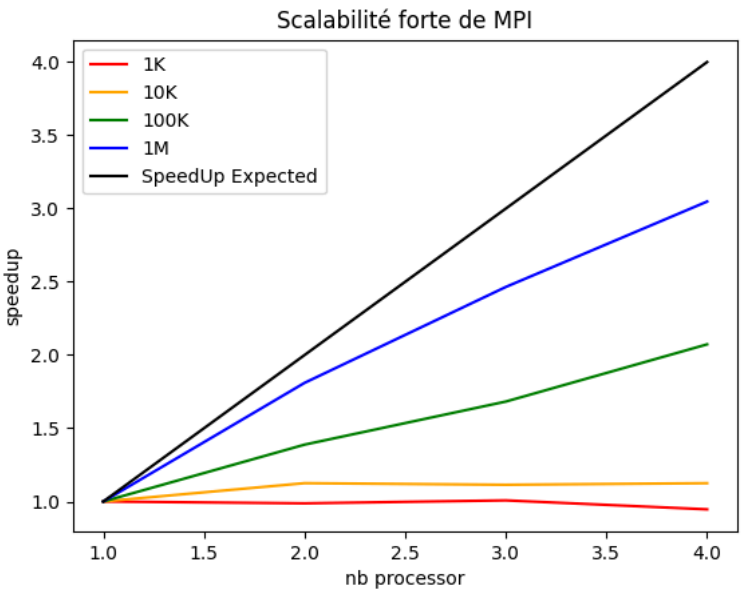
Ce chapitre contient la même structure que le chapitre Prime Scalabilité forte. Il contient des resultats des experiences avec :

- le nombre de boucle
- le nombre de processus
- le temps de calcul

Il contient aussi les graphique de chaque fichier puis un graphique général

MonteCarlo avec MPI

"t1": [
[1000, 1, 4.4448],
[1000, 2, 4.5005],
[1000, 3, 4.4110],
[1000, 4, 4.6959],],
"t2": [
[10000, 1, 5.2903],
[10000, 2, 4.7013],
[10000, 3, 4.7477],
[10000, 4, 4.7019],],
"t3": [
[100000, 1, 13.7532],
[100000, 2, 9.9072],
[100000, 3, 8.1752],
[100000, 4, 6.6350],],
"t4": [
[1000000, 1, 93.2222],
[1000000, 2, 51.5049],
[1000000, 3, 37.8232],
[1000000, 4, 30.5890],]



On remarque que les courbes rouge et jaune reste près du SpeedUp 1 tandis que les autres augmente légèrement. Aucun ne suivent la trajectoire prise par la courbe noire

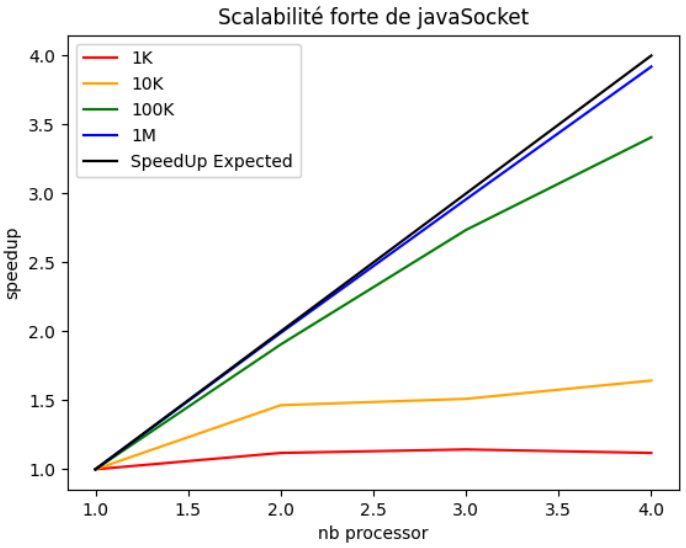
Pour les petites tailles de problème (1000 et 10 000 itérations), le speedup reste proche de 1. Cela indique que le coût de communication et de synchronisation domine le temps de calcul, rendant l'augmentation du nombre de processus peu bénéfique.

Pour des tailles plus importantes (100 000 et 1 000 000 d'itérations), on observe une amélioration progressive du speedup lorsque le nombre de processus augmente. Toutefois, la courbe reste nettement en dessous de la courbe idéale.

L'implémentation MPI bénéficie du parallélisme lorsque la charge de calcul devient suffisante, mais les coûts de communication inhérents à MPI limitent la scalabilité forte globale.

MonteCarlo avec Java Socket

"t1": [
[1000, 1, 1.2273716],
[1000, 2, 1.0970572],
[1000, 3, 1.072838],
[1000, 4, 1.0970625],],
"t2": [
[10000, 1, 0.79533714],
[10000, 2, 0.54292],
[10000, 3, 0.5263509],
[10000, 4, 0.483958]],
"t3": [
[100000, 1, 5.7615037],
[100000, 2, 3.0240145],
[100000, 3, 2.106051],
[100000, 4, 1.6907399],],
"t4": [
[1000000, 1, 55.733814],
[1000000, 2, 28.019552],
[1000000, 3, 18.843683],
[1000000, 4, 14.2178545],],



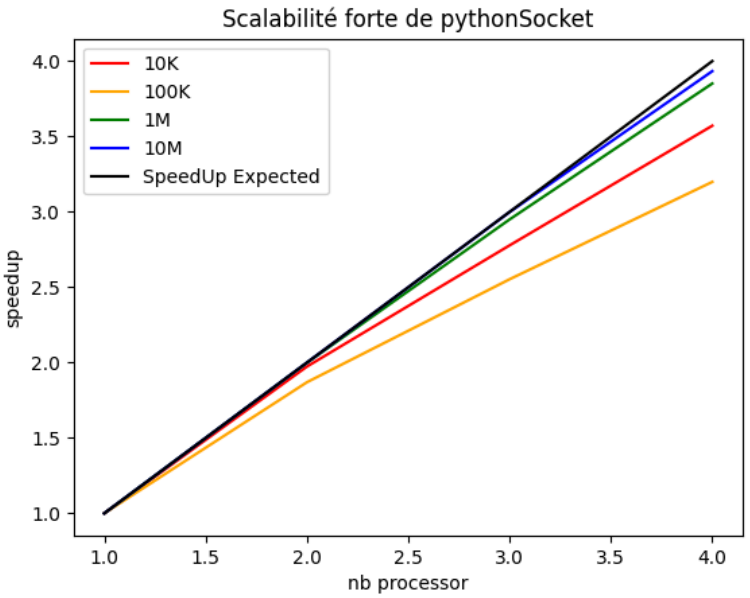
Sur ce graphique on remarque la courbe rouge reste près du speed Up 1 tandis que la courbe bleu se superpose presque avec la courbe noire.

Pour les petites tailles, le speedup reste proche de 1, ce qui est attendu compte tenu du faible volume de calcul.
En revanche, pour les tailles de problème plus importantes, le speedup augmente de manière significative et se rapproche de la courbe idéale, en particulier à partir de 3 et 4 processus.

L’implémentation Java Socket présente une meilleure exploitation du parallélisme que MPI, grâce à une gestion plus légère des communications et à une répartition plus efficace du travail entre les workers.

MonteCarlo avec Python Socket

"t1": [
[10000, 1, 0.075],
[10000, 2, 0.038],
[10000, 3, 0.027],
[10000, 4, 0.021],],
"t2": [
[100000, 1, 0.531],
[100000, 2, 0.284],
[100000, 3, 0.208],
[100000, 4, 0.166],],
"t3": [
[1000000, 1, 5.292],
[1000000, 2, 2.648],
[1000000, 3, 1.794],
[1000000, 4, 1.374],],
"t4": [
[10000000, 1, 52.893],
[10000000, 2, 26.448],
[10000000, 3, 17.638],
[10000000, 4, 13.447],]

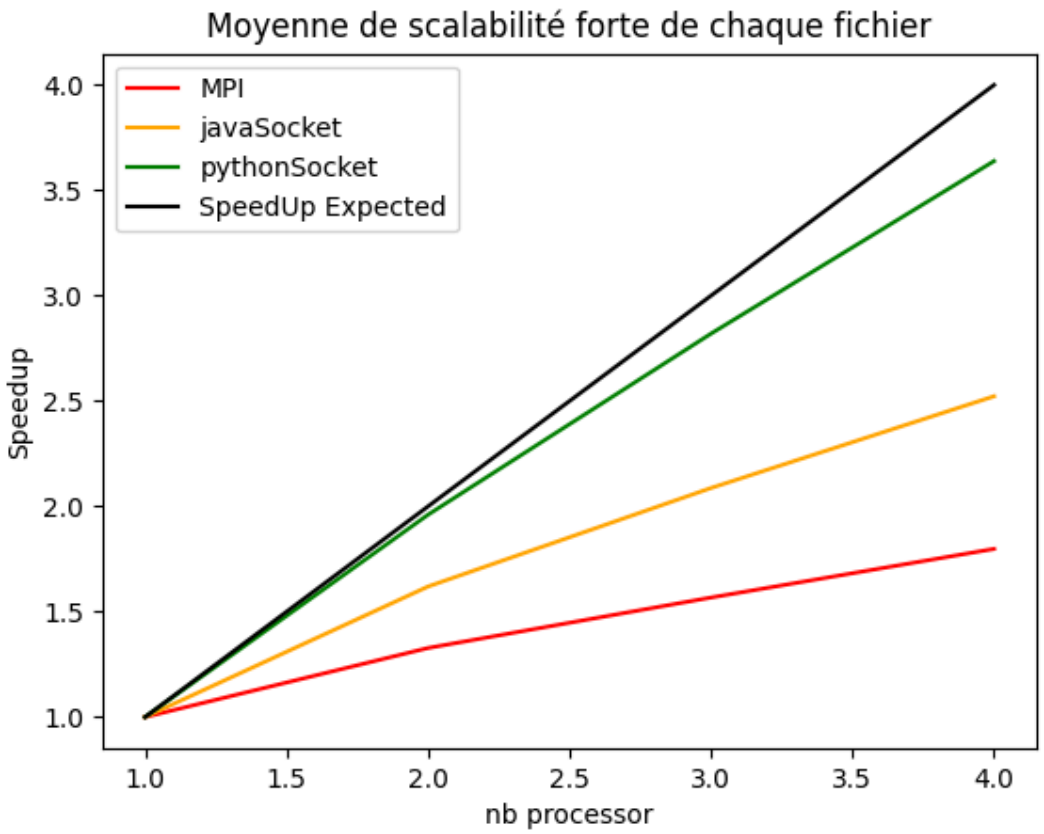


Sur ce graphique, on remarque que toute les courbes suivent plus ou moins la trajectoire de la courbe noire.

Pour toutes les tailles de problème testées, le speedup augmente de manière régulière avec le nombre de processus. Les courbes suivent de près la trajectoire de la courbe idéale, en particulier pour les tailles élevées.

L’implémentation Python Socket est la plus efficace en scalabilité forte. Le caractère indépendant des simulations Monte Carlo, combiné à une gestion simple et efficace des communications, permet une très bonne exploitation du parallélisme.

Moyenne de la scalabilité forte



Ainsi, on remarque que python Socket est le plus performant des trois programmes de MonteCarlos et est suivi par javaSocket et enfin MPI. La courbe verte qui est celle de python Socket reste proche de la courbe noire.

Conclusion - Scalabilité forte MonteCarlo

La comparaison des speedups moyens met en évidence une hiérarchie claire entre les implémentations :

Python Socket > Java Socket > MPI

L'implémentation Python Socket reste la plus proche de la courbe idéale, tandis que MPI est davantage pénalisé par les coûts de communication. Ces résultats confirment que le calcul Monte Carlo est particulièrement bien adapté à la scalabilité forte.

Scalabilité faible

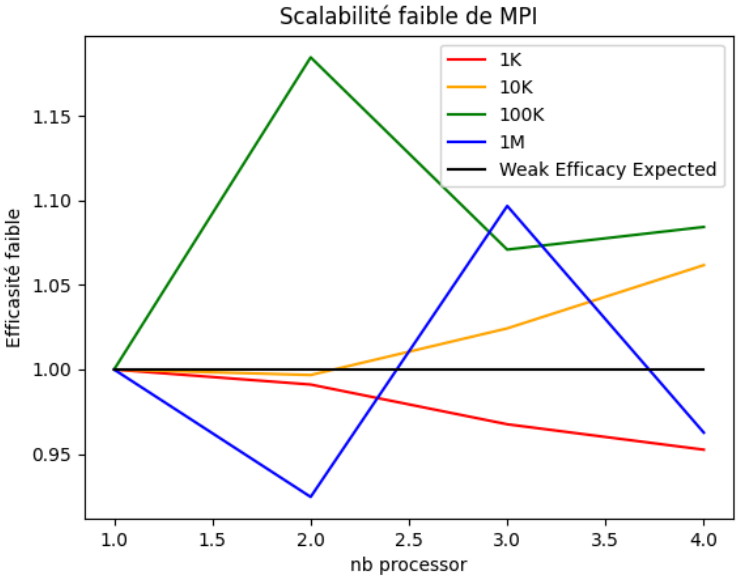
Ce chapitre contient la même structure que le chapitre Prime Scalabilité faible. Il contient des resultats des experiences avec :

- le nombre de boucle
- le nombre de processus
- le temps de calcul

Il contient aussi les graphique de chaque fichier puis un graphique général

MonteCarlo avec MPI

"t1": [
[1000, 1, 4.4221],
[2000, 2, 4.4618],
[3000, 3, 4.5703],
[4000, 4, 4.6425],],
"t2": [
[10000, 1, 5.5215],
[20000, 2, 5.5396],
[30000, 3, 5.3902],
[40000, 4, 5.2002],],
"t3": [
[100000, 1, 15.6544],
[200000, 2, 13.2122],
[300000, 3, 14.6168],
[400000, 4, 14.4359],],
"t4": [
[1000000, 1, 103.2830],
[2000000, 2, 111.7124],
[3000000, 3, 94.1620],
[4000000, 4, 107.2939],]



Dans ce graphique, on remarque que les courbes restent dans les alentours de 1.20 et 0.90, et donc près de la courbe noire.

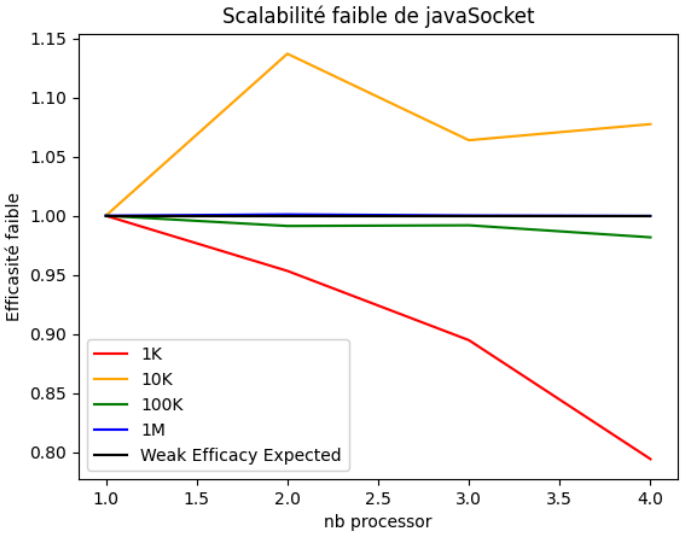
Les résultats montrent que l'efficacité faible reste globalement proche de 1 pour l'ensemble des tailles testées, avec de légères variations autour de cette valeur.

Pour les plus grandes tailles, on observe une certaine instabilité, mais l'efficacité reste dans une plage acceptable, sans chute brutale.

L'implémentation MPI gère correctement l'augmentation simultanée de la charge et du nombre de processus, malgré un surcoût de communication non négligeable.

MonteCarlo avec Java Socket

"t1": [
[1000, 1, 0.24249129],
[2000, 2, 0.25435197],
[3000, 3, 0.27098712],
[4000, 4, 0.30524653],],
"t2": [
[10000, 1, 0.8521876],
[20000, 2, 0.7495973],
[30000, 3, 0.8010893],
[40000, 4, 0.7909831],],
"t3": [
[100000, 1, 5.7430806],
[200000, 2, 5.793276],
[300000, 3, 5.789906],
[400000, 4, 5.8498306],],
"t4": [
[1000000, 1, 56.052273],
[2000000, 2, 55.992096],
[3000000, 3, 56.04285],
[4000000, 4, 56.063],]



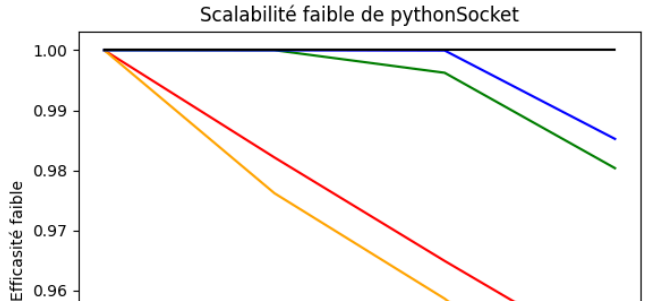
Dans ce graphique, on remarque que la courbe bleu est à peine visible car il est masqué par la courbe noire. La courbe rouge baisse légerement tandis que la courbe jaune augmente légerment.

Les courbes d'efficacité faible sont très proches de la courbe idéale. Certaines variations apparaissent selon le nombre de processus, mais elles restent faibles et ne remettent pas en cause la stabilité globale.

Java Socket maintient une efficacité faible élevée, ce qui indique une bonne répartition du travail et un coût de communication bien maîtrisé.

MonteCarlo avec Python Socket

"t1": [
[10000, 1, 0.055],
[20000, 2, 0.056],
[30000, 3, 0.057],
[40000, 4, 0.058],],
"t2": [



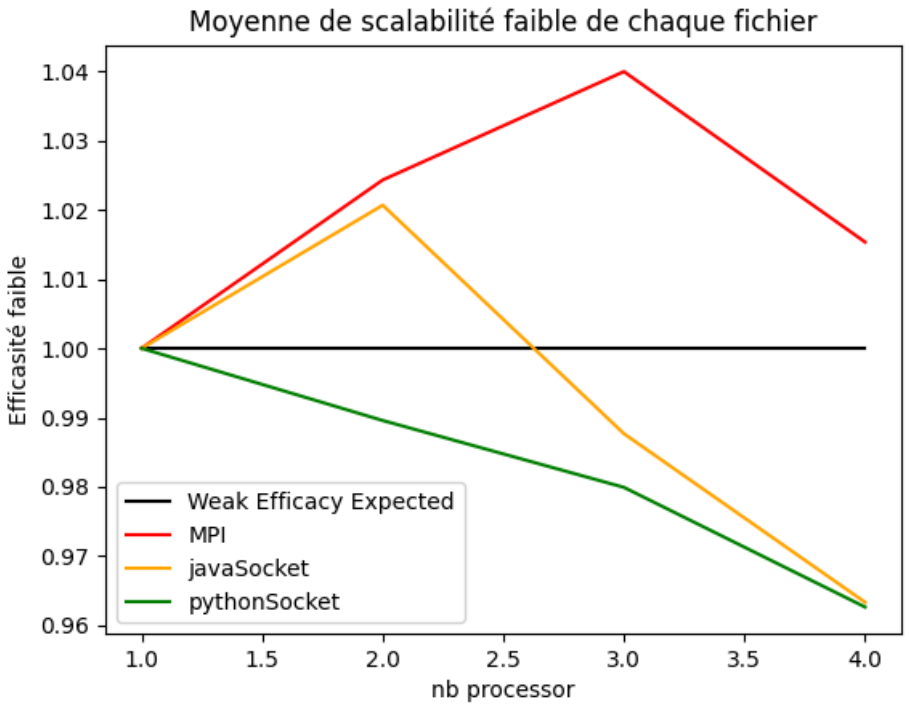
[100000, 1, 0.533],
[200000, 2, 0.546],
[300000, 3, 0.556],
[400000, 4, 0.569],],
"t3": [
[1000000, 1, 5.291],
[2000000, 2, 5.291],
[3000000, 3, 5.311],
[4000000, 4, 5.397],],
"t4": [
[10000000, 1, 52.891],
[20000000, 2, 52.892],
[30000000, 3, 52.894],
[40000000, 4, 53.684],]

Dans ce graphique, on remarque que toutes les courbes baissent très légèrement. La courbe jaune baisse fortement par rapport au autres courbe mais dépace 0.94 en efficacité faible avec 4 processus.

L'efficacité faible reste très proche de 1 pour la majorité des expériences. On observe une très légère baisse lorsque le nombre de processus augmente, mais celle-ci reste limitée, même pour les plus grandes tailles de problème.

Python Socket est l'implémentation la plus stable en scalabilité faible. Elle démontre une excellente capacité à absorber l'augmentation de la charge de calcul sans dégradation significative des performances.

Moyenne de la scalabilité faible



Conclusion - Scalabilité faible Monte Carlo

Ainsi les trois fichiers restent dans près de la courbe noire. La courbe jaune et verte ne passe pas en dessous de 0.96 en efficacité faible.

Les moyennes d'efficacité faible montrent que les trois implémentations restent proches de la valeur idéale. Aucune ne passe significativement en dessous de 0,96, ce qui confirme une très bonne scalabilité faible globale.

Conclusion - Monte Carlo

Les résultats obtenus pour Monte Carlo montrent que ce type de problème est particulièrement bien adapté au calcul parallèle.

- En scalabilité forte, Python Socket se distingue comme l'implémentation la plus performante, suivie de Java Socket, puis MPI.

- En scalabilité faible, les trois implémentations maintiennent une efficacité élevée, proche de l'idéal, avec un léger avantage pour MPI.

Ces résultats confirment que la nature indépendante des simulations Monte Carlo permet une exploitation efficace des ressources parallèles, aussi bien lorsque le nombre de processus augmente que lorsque la charge de travail croît proportionnellement.