

Rapport Qualité de Développement

Sylvain COUTURIER



université PARIS-SACLAY

IUT de Vélizy-Rambouillet
CAMPUS DE VÉLIZY-VILLACOUBLAY
CAMPUS DE RAMBOUILLET

I. Introduction	3
II. Scalabilité Forte	3
III. Scalabilité Faible	5
IV. Conclusion	7
Annexe - Évaluation de Pi.java	8
Introduction	8
Scalabilité forte	8
Scalabilité faible	9
Précision	10

I. Introduction

L'objectif de cette étude est d'évaluer les performances d'une implémentation parallèle de l'algorithme Monte Carlo pour le calcul de π , basée sur une architecture Master-Worker utilisant des sockets TCP/IP en Java. Pour analyser la qualité de cette parallélisation, nous nous appuyons sur plusieurs métriques standardisées issues de la théorie du calcul parallèle.

Le critère principal de qualité est le speedup, défini par la formule $S(p) = T_1/T_p$ où T_1 représente le temps d'exécution avec un seul worker et T_p le temps d'exécution avec p workers. Un speedup linéaire idéal correspond à $S(p) = p$, ce qui signifie que doubler le nombre de workers divise le temps d'exécution par deux.

Cette métrique est évaluée en référence aux normes théoriques du parallélisme. La **loi d'Amdahl** constitue notre référence principale, car elle établit la limite supérieure du speedup en fonction de la proportion de code séquentiel dans l'application. Dans notre implémentation Master-Worker, la partie séquentielle inclut la distribution initiale des tâches, la collecte des résultats par le Master (boucles séquentielles dans MasterSocket.java), et le calcul final de π .

Lien du git : <https://github.com/sylvain922/MC-Socket-Performance>

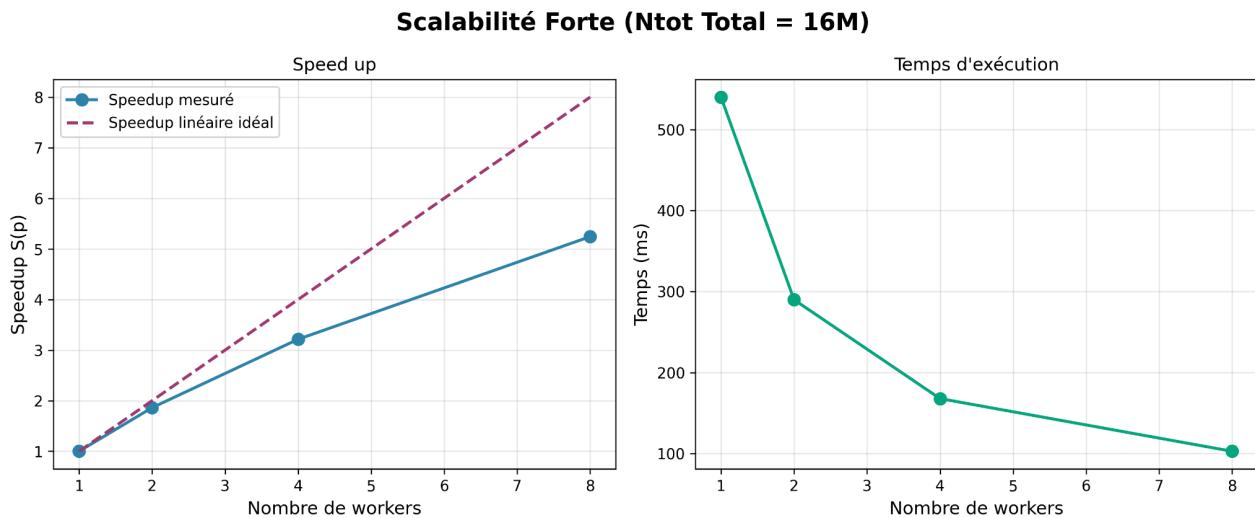
II. Scalabilité Forte

La scalabilité forte mesure la capacité du système à réduire le temps d'exécution lorsque le nombre de processeurs augmente, tout en maintenant la taille totale du problème constante. Concrètement, nous fixons le nombre total de points Monte Carlo à simuler ($N_{\text{tot total}}$) à 16 000 000 points et nous augmentons progressivement le nombre de workers de 1 à 8. Dans notre implémentation, la variable `totalCount` représente le nombre de points traités par chaque worker individuellement. Par conséquent, pour maintenir un $N_{\text{tot total}}$ constant, nous devons ajuster `totalCount` selon la formule : $\text{totalCount} = 16\ 000\ 000 / \text{numWorkers}$. Ainsi, avec 1 worker, `totalCount` vaut 16 millions ; avec 2 workers,

chaque worker traite 8 millions de points ; avec 4 workers, chaque worker traite 4 millions ; et avec 8 workers, chaque worker ne traite que 2 millions de points.

Le protocole expérimental consiste à lancer successivement des calculs avec 1, 2, 4 et 8 workers en ajustant totalCount proportionnellement. Pour chaque configuration, nous mesurons le temps d'exécution $T(p)$ affiché par le Master après synchronisation de tous les workers. Le speedup est ensuite calculé en divisant le temps de référence $T(1) = 540$ ms par le temps mesuré $T(p)$.

p	1	2	4	8
T(p) en ms	540	290	168	103
S(p)	1.0	1.86206896552	3.21428571429	5.2427184466



Les résultats permettent de tracer deux courbes essentielles. La première courbe représente le speedup en fonction du nombre de workers, avec la droite $y = x$ comme référence du speedup linéaire idéal. Notre implémentation montre un speedup de 5.24 avec 8 workers, ce qui reste inférieur au speedup idéal de 8.0. La seconde courbe présente le temps d'exécution absolu, permettant de visualiser directement le gain de performance : le temps passe de 540 ms avec 1 worker à 103 ms avec 8 workers.

L'analyse des résultats révèle que l'écart par rapport au speedup linéaire s'explique par plusieurs facteurs. Dans notre architecture, la synchronisation séquentielle des workers par le Master constitue un goulot d'étranglement : le Master envoie les tâches dans une première boucle, puis attend séquentiellement la réponse de chaque worker dans une seconde boucle. Cette attente séquentielle dégrade les performances car le Master ne peut pas traiter en parallèle les réponses des différents workers. De plus, chaque connexion socket introduit des latences de communication TCP/IP, incluant la sérialisation et déserialisation des messages.

III. Scalabilité Faible

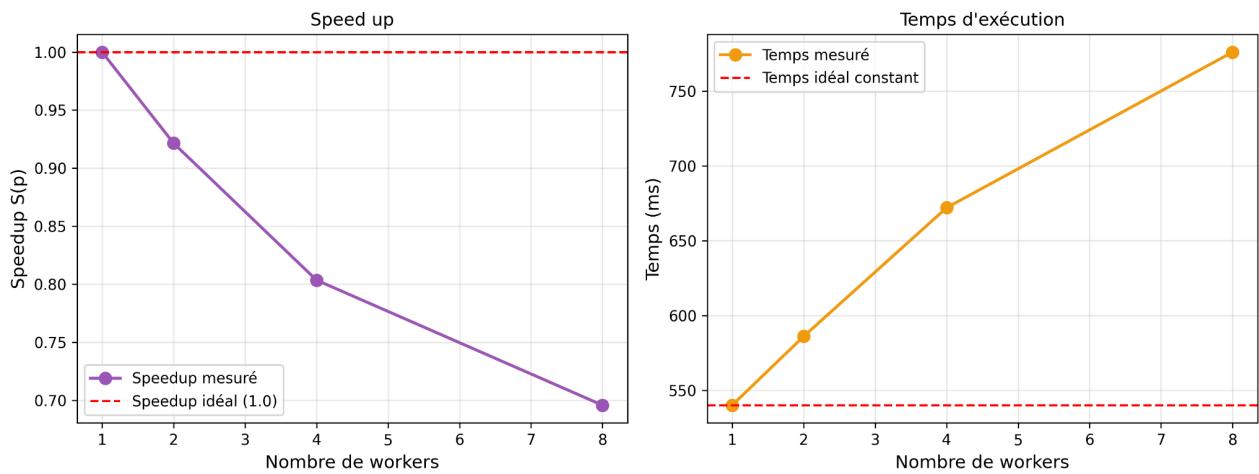
La scalabilité faible évalue la capacité du système à maintenir un temps d'exécution constant lorsque le nombre de workers et la taille du problème augmentent proportionnellement. Dans ce scénario, chaque worker traite toujours la même charge de travail individuelle, indépendamment du nombre total de workers. Ce test révèle l'impact de l'infrastructure parallèle elle-même, car idéalement, si chaque worker calcule toujours le même nombre de points, le temps d'exécution devrait rester stable.

Pour notre expérimentation, nous fixons la charge de travail par worker à une valeur constante de 16 000 000 de points en maintenant la variable `totalCount` constante dans le code. Ainsi, chaque worker traite systématiquement 16 millions de points, quelle que soit la configuration. Avec 1 worker, le `Ntot` total vaut 16 millions de points ; avec 2 workers, le `Ntot` total passe à 32 millions ; avec 4 workers, 64 millions ; et avec 8 workers, 128 millions de points au total. Contrairement à la scalabilité forte où nous ajustons `totalCount` pour maintenir le `Ntot` total constant, ici nous gardons `totalCount = 16 000 000` fixe dans le code.

Les mesures consistent à enregistrer le temps d'exécution pour chaque configuration. Dans un scénario idéal de scalabilité faible parfaite, le temps d'exécution devrait rester constant quelle que soit la configuration. En pratique, on observe généralement une augmentation du temps, révélant les coûts croissants de gestion des communications et de synchronisation.

p	1	2	4	8
T(p) en ms	540	586	672	776
S(p)	1.0	0.92150170648	0.80357142857	0.69587628866

Scalabilité Faible (16M points/worker)



La courbe représente le temps d'exécution en fonction du nombre de workers. Une ligne horizontale au niveau du temps de référence (540 ms) matérialise le comportement idéal. L'écart vertical entre la courbe mesurée et cette référence révèle la dégradation des performances : le temps augmente de 540 ms à 776 ms, soit une augmentation de 236 ms (+43.7%).

Dans notre implémentation Master-Worker par sockets, plusieurs facteurs expliquent cette dégradation. Premièrement, le Master doit gérer un nombre croissant de connexions socket, chacune nécessitant des ressources système. Deuxièmement, les boucles de synchronisation dans le Master deviennent proportionnellement plus longues : avec 8 workers, le Master doit itérer 8 fois pour envoyer les tâches, puis 8 fois pour collecter les résultats. Ce temps de synchronisation s'ajoute au temps de calcul pur. Troisièmement, la lecture bloquante séquentielle des résultats (`reader[i].readLine()`) impose que le Master attende chaque worker tour à tour, même si certains workers ont terminé avant d'autres.

IV. Conclusion

Cette étude de performance a permis d'évaluer quantitativement la scalabilité de notre implémentation Monte Carlo Master-Worker par sockets Java. Les résultats de scalabilité forte montrent un speedup maximal de 5.24 avec 8 workers, soit 65% du speedup linéaire théorique. En scalabilité faible, le temps d'exécution augmente de 43.7% entre 1 et 8 workers, révélant l'impact croissant des coûts de communication et de synchronisation.

L'architecture Master-Worker par sockets présente de bonnes performances jusqu'à 4 workers, mais montre rapidement ses limites au-delà. Les principaux goulets d'étranglement identifiés sont la synchronisation séquentielle des workers (le Master attend chaque réponse tour à tour) et les latences de communication TCP/IP. L'utilisation de sockets en mode texte ajoute également des conversions String-Integer coûteuses à chaque échange.

Méthodologie et Outils d'Automatisation

Pour garantir la reproductibilité et la rigueur des tests de performance, nous avons développé une infrastructure d'automatisation basée sur deux scripts shell, conçus avec l'assistance de l'assistant IA Claude.

Le premier script, **launch_workers.sh**, résout le problème pratique du lancement manuel de multiples workers. Sans automatisation, tester une configuration avec 8 workers nécessiterait d'ouvrir 8 terminaux distincts et de lancer manuellement chaque worker avec son port spécifique, processus fastidieux et source d'erreurs. Le script prend en paramètre le nombre de workers souhaité et les lance automatiquement en arrière-plan sur des ports consécutifs (25545, 25546, etc.). Chaque worker est exécuté avec redirection de sortie pour éviter la pollution de la console, et le script affiche le PID de chaque processus pour faciliter le débogage.

Le second script, **run_benchmarks.sh**, automatise l'ensemble du protocole expérimental. Il accepte un paramètre définissant le type de test (scalabilité forte ou faible) et exécute séquentiellement toutes les configurations nécessaires (1, 2, 4, 8 workers). Pour chaque configuration, le script lance les workers appropriés, attend leur initialisation complète, déclenche les calculs Monte Carlo, collecte les temps d'exécution, et termine proprement

les processus workers avant de passer à la configuration suivante. Les résultats sont automatiquement enregistrés dans des fichiers CSV structurés, prêts pour l'analyse avec le script Python.

Cette infrastructure d'automatisation présente plusieurs avantages. Premièrement, elle élimine les erreurs humaines lors des manipulations répétitives (oubli de workers, confusion de ports, mesures incomplètes). Deuxièmement, elle garantit la reproductibilité : n'importe quel utilisateur peut relancer les tests exactement dans les mêmes conditions. Troisièmement, elle réduit drastiquement le temps nécessaire pour exécuter l'ensemble du protocole expérimental, permettant des tests itératifs rapides lors du développement et du débogage.

Annexe - Évaluation de Pi.java

Introduction

Dans ce rapport, j'analyse l'implémentation parallèle du calcul de π par Monte Carlo en Java : Pi.java (basé sur Callable/Future)

On génère 8 millions de points aléatoires dans un carré unité. Le ratio de points tombant dans le quart de cercle ($rayon=1$) multiplié par 4 donne π . Chaque point est indépendant → parallélisme embarrassingly parallel idéal pour multicœurs.

J'évalue selon performance (T_p , $S_p = T_1/T_p$, scalabilité forte/faible) et qualité logicielle (lisibilité, testabilité, gestion erreurs, isolation des tâches).

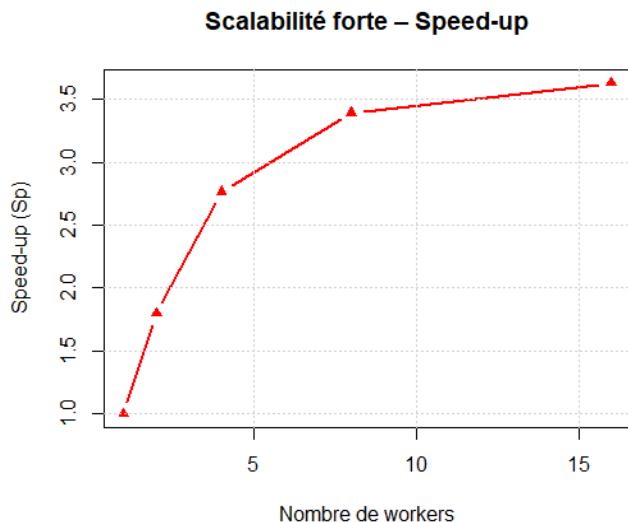
Scalabilité forte

Chaque Callable génère ses points et renvoie le nombre de succès (points dans cercle). Le main thread somme tous les Future.get().

Workers	Tp (ms) moyen	Sp (T1/Tp, T1≈201.7ms)
1	201.7	1.00
2	111.7	1.81

4	72.8	2.77
8	57.5	3.51
16	55.0	3.67

Analyse : Quasi-linéaire jusqu'à 8 workers. Au-delà, overhead Future (attente get () + fermeture pool) + risque de déséquilibre dans la file d'attente unique.

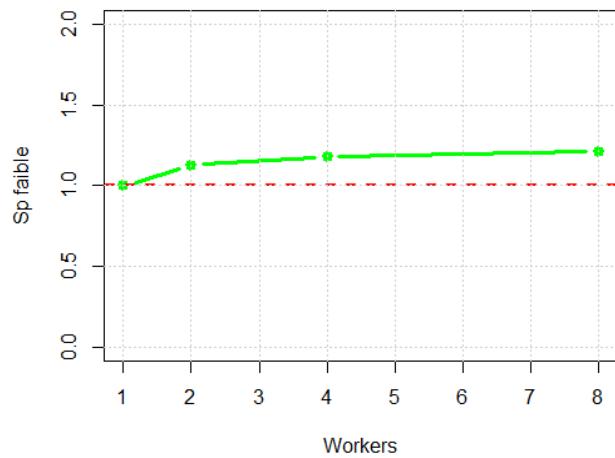


Scalabilité faible

En scalabilité faible, j'augmente N et le nombre de workers proportionnellement : chaque tâche garde une charge constante.

Workers	Tp (ms) moyen	Sp (T1/Tp, T1≈31ms)
1	31	1.0
2	55	1.13
4	105	1.18
8	205	1.21

Scalabilité faible – Speed-up



Précision

La précision suit la loi statistique : erreur $\approx 1/\sqrt{N}$ total, identique aux deux implémentations pour même N. Les Callable facilitent les tests : vérifier somme résultats, propager exceptions d'une tâche défaillante, sans risque de course sur état partagé.