Rapport sur le cours de programmation avancée et qualité de développement

PAVLETIC Thomas - INFO 3 FA

Réalisé à l'aide de ChatGPT pour les définitions et Microsoft Copilot pour les analyses de code

Introduction

Ce rapport a pour objectif de détailler les notions abordées dans le cours de programmation avancée et de qualité de développement. Ce cours a couvert divers aspects de la programmation parallèle, la mémoire partagée et distribuée, ainsi que les normes de qualité logicielle. Nous avons mis en pratique ces concepts à travers des exercices et des tests de scalabilité, tout en se rapprochant des normes ISO 25010 et 25022.

Sommaire

Programmation parallèle

La programmation parallèle est une approche de développement qui permet d'exécuter plusieurs instructions simultanément en exploitant les ressources matérielles disponibles, comme les processeurs multicœurs ou les architectures distribuées. Elle est utilisée pour accélérer les traitements, améliorer la performance des applications et optimiser l'utilisation des ressources.

La classe Thread en Java

Nous avons étudié la classe Thread en Java, qui permet de créer et gérer des threads pour l'exécution parallèle. Cette classe est essentielle pour développer des applications multithreadées en Java.

API Concurrent de Java

L'API Concurrent de Java fournit des outils et des structures de données pour faciliter la programmation parallèle. Elle inclut des classes comme ExecutorService, qui permet de gérer des pools de threads, et des collections concurrentes pour une manipulation sécurisée des données partagées.

Concepts et paradigmes

La programmation parallèle vise à exécuter plusieurs processus simultanément pour améliorer l'efficacité et la performance des applications. Nous avons exploré différents paradigmes, dont le modèle "master-worker", où un processus maître distribue les tâches à plusieurs processus travailleurs et collecte les résultats.

Mémoire partagée et distribuée

Il existe plusieurs façons d'organiser l'exécution parallèle d'un programme en fonction de l'architecture utilisée. On distingue principalement trois modèles : la mémoire partagée, la mémoire distribuée et le modèle hybride.

Mémoire partagée

Dans ce modèle, plusieurs threads ou processus travaillent sur un même espace mémoire. Cela signifie qu'ils peuvent accéder et modifier les mêmes données en parallèle. Ce modèle est couramment utilisé dans les systèmes multicœurs, où plusieurs threads s'exécutent sur un même processeur tout en partageant la mémoire principale.

Cependant, cette approche présente certains défis, notamment le risque de conditions de course, où plusieurs threads tentent de modifier une même donnée simultanément, entraînant des incohérences. Pour éviter ces problèmes, des mécanismes de synchronisation comme les verrous (locks) ou les sémaphores sont utilisés.

Par exemple, en Java, il est possible de programmer selon ce modèle en utilisant la classe Thread ou l'API java.util.concurrent, qui fournit des outils pour gérer l'exécution parallèle et la synchronisation des accès à la mémoire.

Mémoire distribuée

Contrairement au modèle précédent, le modèle de mémoire distribuée ne repose pas sur un espace mémoire partagé. Chaque processus possède sa propre mémoire et doit communiquer avec les autres via un réseau. Cette approche est utilisée lorsqu'un programme est exécuté sur plusieurs machines, comme dans les clusters de calcul ou les systèmes de calcul haute performance (HPC).

La communication entre les processus s'effectue grâce à un échange de messages, un principe connu sous le nom de message passing. Chaque processus envoie et reçoit des données via des canaux de communication réseau.

Un exemple concret d'implémentation de ce modèle est l'utilisation de MPI (Message Passing Interface), une bibliothèque permettant d'échanger des messages entre processus sur différentes machines. En Java, un mécanisme similaire peut être mis en place à l'aide des sockets réseau (Java Sockets), qui permettent aux programmes de communiquer entre eux en établissant des connexions via TCP/IP.

Modèle hybride

Enfin, le modèle hybride combine les deux approches précédentes pour exploiter au mieux les ressources disponibles. Il est souvent utilisé dans les supercalculateurs ou les architectures de clusters modernes.

Dans ce modèle, plusieurs processus s'exécutent sur des machines différentes en utilisant une mémoire distribuée, mais chaque machine peut également exécuter plusieurs threads en mémoire partagée sur ses propres cœurs.

Ce modèle offre une grande flexibilité et permet de tirer parti à la fois de la rapidité du parallélisme local et de la puissance du calcul distribué.

Tests de scalabilité

Les tests de scalabilité permettent d'évaluer comment un système ou une application réagit lorsqu'on augmente la charge de travail ou les ressources disponibles. L'objectif est de vérifier si les performances restent acceptables et d'identifier les éventuels points de contention qui pourraient limiter l'efficacité du système.

En programmation parallèle, il existe deux principaux types de scalabilité : la scalabilité forte et la scalabilité faible.

Afin d'évaluer la scalabilité d'un système, plusieurs métriques sont couramment utilisées. La première est le temps d'exécution total, qui permet d'analyser directement la rapidité du programme en fonction des ressources disponibles. Une autre mesure essentielle est le speedup, ou accélération, qui correspond au rapport entre le temps d'exécution sur un seul cœur et le temps d'exécution avec plusieurs cœurs. Enfin, l'efficacité est un indicateur qui permet de déterminer si l'ajout de ressources entraîne un gain réel de performances ou s'il engendre un gaspillage de puissance de calcul. Pour réaliser ces tests, des mesures manuelles peuvent être effectuées en instrumentant le code à l'aide de timestamps, par exemple avec la méthode System.nanoTime().

Les tests de scalabilité sont essentiels pour s'assurer que les applications parallèles et distribuées exploitent pleinement les ressources matérielles mises à leur disposition. Ils permettent d'optimiser les performances des logiciels tout en identifiant les limites liées à la gestion de la concurrence et à la communication entre les unités de calcul.

Scalabilité forte

La scalabilité forte mesure l'évolution des performances lorsqu'on augmente les ressources sans modifier la charge de travail. Concrètement, cela signifie que le volume de données traité reste constant tandis que le nombre de cœurs de calcul ou de machines est progressivement augmenté. Si la scalabilité est idéale, alors le temps d'exécution doit diminuer proportionnellement au nombre de ressources supplémentaires. Par exemple, si une tâche prend dix secondes avec un seul cœur, elle devrait prendre cinq secondes avec deux cœurs et deux secondes et demie avec quatre cœurs. Cependant, en pratique, cette amélioration est limitée par plusieurs facteurs, notamment les coûts liés à la synchronisation et à la communication entre les unités de calcul. À mesure que le nombre de ressources augmente, ces coûts peuvent devenir significatifs et empêcher le programme de continuer à gagner en rapidité. Ce phénomène est expliqué par le fait que l'accélération maximale d'un programme parallèle est contrainte par la fraction de code qui doit s'exécuter de manière séquentielle. On appelle aussi cela un problème de granularité des tâches. Un exemple typique de test de scalabilité forte serait un algorithme de tri parallèle, dans lequel un tableau est divisé en plusieurs sous-parties traitées indépendamment avant d'être fusionnées.

Scalabilité faible

La scalabilité faible, quant à elle, consiste à observer comment évoluent les performances lorsqu'on augmente simultanément la taille du problème et les ressources allouées. Dans ce cas, l'objectif est de vérifier si le temps d'exécution reste stable malgré l'augmentation de la charge de travail. Si la scalabilité est optimale, alors le temps d'exécution ne devrait pas varier lorsque le

nombre de ressources et la taille du problème augmentent proportionnellement. Par exemple, si un programme de rendu 3D met dix minutes à générer une image en utilisant un certain nombre de serveurs, il devrait mettre également dix minutes à générer une image deux fois plus complexe en doublant le nombre de serveurs disponibles. Toutefois, dans certains cas, les coûts de communication entre les machines peuvent dépasser les gains apportés par l'ajout de ressources, rendant ainsi l'augmentation de la capacité inefficace.

Normes de qualité logicielle

Nous avons suivi les normes de qualité logicielle ISO 25010 et 25022 pour garantir la robustesse, la maintenabilité et l'efficacité de nos applications. Ces normes fournissent des critères et des indicateurs pour évaluer la qualité des logiciels.

Les normes ISO 25010 et ISO 25022 font partie de la famille des normes ISO/IEC 25000, qui définit un cadre pour l'évaluation de la qualité des logiciels et des systèmes informatiques. Ces normes sont utilisées pour garantir que les logiciels répondent aux exigences de performance, de fiabilité et d'expérience utilisateur.

Les normes ISO 25010 et ISO 25022 sont complémentaires. ISO 25010 fournit un cadre général pour définir les critères de qualité d'un logiciel, tandis qu'ISO 25022 propose des indicateurs et des méthodes de mesure pour évaluer cette qualité en situation réelle. En les appliquant, les on peut s'assurer que les logiciels sont performants, robustes et adaptés aux besoins des utilisateurs, tout en respectant les standards internationaux de qualité logicielle.

ISO 25010

La norme ISO 25010 définit un modèle de qualité permettant d'évaluer un logiciel en fonction de plusieurs critères. Ce modèle se divise en deux parties principales : les caractéristiques de qualité du produit logiciel et celles de qualité en usage.

La qualité du produit logiciel est évaluée selon huit caractéristiques essentielles. La première est la fonctionnalité, qui mesure si le logiciel remplit correctement les tâches pour lesquelles il a été conçu. La deuxième est la performance et l'efficacité, qui évaluent la rapidité d'exécution et l'utilisation optimale des ressources. La compatibilité est un autre critère clé et concerne la capacité du logiciel à fonctionner avec d'autres systèmes ou plateformes sans conflits. Ensuite, la fiabilité mesure la capacité du logiciel à fonctionner sans erreurs et à récupérer rapidement en cas de panne.

D'autres aspects sont également pris en compte. La sécurité est un critère fondamental qui concerne la protection des données et la résistance aux attaques. La maintenabilité évalue la facilité avec laquelle le logiciel peut être modifié ou corrigé sans engendrer de nouvelles erreurs. L'utilisabilité concerne l'ergonomie et l'accessibilité du logiciel pour les utilisateurs finaux. Enfin, la portabilité désigne la capacité du logiciel à être exécuté sur différentes plateformes sans nécessiter de modifications majeures.

La qualité en usage, quant à elle, se focalise sur l'expérience utilisateur et l'efficacité du logiciel dans un contexte réel. Elle inclut la pertinence, qui évalue si le logiciel répond bien aux besoins des utilisateurs, ainsi que l'efficience, qui mesure la rapidité et la facilité avec lesquelles les utilisateurs atteignent leurs objectifs. L'apprentissage et l'accessibilité font également partie de cette catégorie, garantissant que le logiciel est facile à comprendre et utilisable par un large éventail de personnes, y compris celles ayant des besoins spécifiques.

La norme ISO 25010, propose donc un cadre structuré pour concevoir et évaluer des logiciels de haute qualité, en assurant leur robustesse, leur sécurité et leur facilité d'utilisation.

ISO 25022

La norme ISO 25022 complète ISO 25010 en définissant des méthodes de mesure permettant d'évaluer objectivement la qualité d'un logiciel en usage. Alors que la norme ISO 25010 décrit les caractéristiques de qualité, ISO 25022 propose des indicateurs concrets pour mesurer ces caractéristiques dans des conditions réelles d'utilisation.

Par exemple, pour mesurer l'efficacité d'un logiciel, ISO 25022 recommande de collecter des données sur le temps nécessaire aux utilisateurs pour accomplir une tâche donnée et sur le taux de succès des opérations effectuées. L'efficience peut être évaluée en analysant le nombre d'actions ou d'étapes nécessaires pour réaliser une tâche avec succès, ce qui permet d'identifier d'éventuelles complexités inutiles dans l'interface ou le processus.

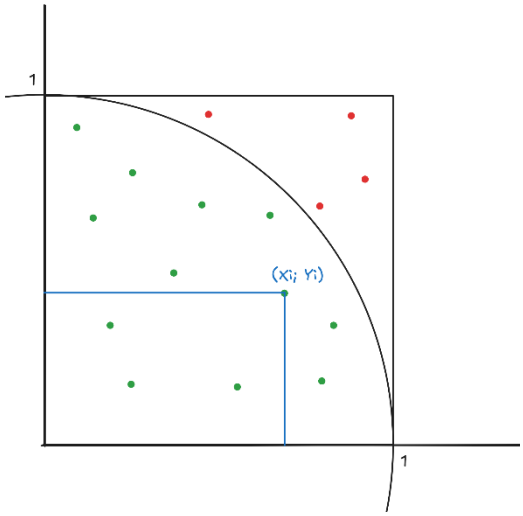
Les principes de la norme ISO 25022, permettent d'identifier de manière précise les points forts et les faiblesses d'un logiciel et mettre en place des améliorations ciblées pour optimiser l'expérience utilisateur et l'efficacité globale du produit.

Mise en application

Pour mettre en application tous ces principes, nous avons utilisé plusieurs codes qui utilisent des techniques de programmations différentes.

Nous avons utilisé comme exemple une méthode de calcul de pi, qui se base sur la méthode de Monte Carlo. Chaque code suit ce même exemple, ce qui nous permet d'évaluer les performances entre ces différents codes.

La méthode de Monte-Carlo consiste à estimer la valeur de π en générant aléatoirement des points dans un carré de côté 1. On compte ensuite combien de ces points tombent à l'intérieur d'un quart de cercle de rayon 1 inscrit dans ce carré. La proportion des points à l'intérieur du cercle par rapport au total des points générés permet d'estimer π. Pour ce faire, on multiplie le rapport par 4, car le quart de cercle représente un quart de l'aire totale du carré. Ainsi, la relation utilisée est $\pi \approx 4 \times$ (nombre de points dans le cercle / nombre total de points).



Dans notre cours, nous avons appliqué cette méthode en utilisant un quart de disque pour simplifier les calculs. Le principe reste le même : des points sont placés aléatoirement sur la surface du carré contenant le quart de disque, et nous vérifions si ces points se situent à l'intérieur du disque ou en dehors. Cette vérification se fait en calculant la distance de chaque point par rapport à l'origine ; si cette distance est inférieure ou égale au rayon du cercle, le point est considéré comme étant à l'intérieur du quart de disque. Cette distance se calcule simplement par la formule suivante : $\sqrt{(Xi^2+Yi^2)}$; Que l'on a simplifié en Xi^2+Yi^2, car notre cercle est de rayon 1. Cette simplification permet une réduction du temps de calcul, car on retire complètement l'opération de racine carrée.

En répétant ce processus avec un grand nombre de points, nous obtenons une estimation plus précise de la valeur de π. En divisant le nombre de points à l'intérieur du quart de disque par le nombre total de points générés, puis en multipliant le résultat par 4, nous pouvons approcher la valeur de π. Cette méthode est particulièrement intéressante en raison de sa simplicité et de son efficacité, surtout lorsqu'elle est exécutée en parallèle pour accélérer les calculs.

Matériel utilisé pour les tests

Suite à un manque de données récoltées durant les cours, tous les tests présentés dans ce rapport ont été exécutés sur u PC personnel. Celui-ci est composé d'un processeur Intel Core i7-1255U. Ce processeur compte 10 cœurs physiques, dont seulement 2 sont hyperthreadés. Ce processeur affiche une vitesse d'horloge de 2,6 GHz. Il possède 3 niveaux de cache, L1, L2 et L3 qui peuvent stocker respectivement 928ko, 6.5Mo et 12Mo de données.

Cette architecture diffère donc complètement des architectures présentes en salle de classe, où nous trouvions des processeurs à 8 cœurs non hypertrheadés. (places au fond à droite de la G26).

Assignement 102

Ce programme illustre la programmation parallèle en Java à travers l'estimation de la valeur de π en utilisant la méthode de Monte-Carlo. Il exploite l'API concurrentielle de Java pour paralléliser les calculs à l'aide d'un pool de threads, permettant d'accélérer les calculs en fonction du nombre de cœurs disponibles.

Décomposition du code

a) La classe PiMonteCarlo

Cette classe encapsule la logique d'estimation de π.

Attributs :

- AtomicLong nAtomSuccess : compteur thread-safe pour stocker le nombre de points qui tombent dans le cercle.
- long nThrows : nombre total de points à générer pour l'expérience.
- double value : valeur estimée de π.

Sous-classe MonteCarlo :

- Cette classe implémente Runnable et représente une tâche exécutée par un thread.
- Lorsqu'un thread exécute la tâche, il génère un point (x, y) aléatoire dans le carré et vérifie s'il est dans le cercle.
- Si c'est le cas, il incrémente nAtomSuccess de manière thread-safe grâce à AtomicLong.

Méthode getPi(int nProcessors) :

- Exécution parallèle :
  - Un pool de threads est créé avec Executors.newWorkStealingPool(nProcessors).
  - Un grand nombre de tâches Monte-Carlo sont soumises au pool de threads.
  - Chaque tâche représente un lancer de point dans le carré.
- Attente de la fin de l'exécution :
  - executor.shutdown() indique que plus aucune tâche ne sera soumise.
  - Une boucle while (!executor.isTerminated()) {} attend que toutes les tâches soient terminées.
- Calcul final :
  - La valeur de π est estimée grâce à la formule :

$$\pi = 4.0 \times nAtomSuccessnThrows\pi = 4.0 \times \frac{\text{nAtomSuccess}}{\text{nThrows}}\pi = 4.0 \times nThrowsnAtomSuccess$$

b) La classe Assignment102

Cette classe contient le main et exécute les simulations avec différentes configurations.

1. Lancement de l'expérience :
   - Un objet PiMonteCarlo est instancié avec nThrow * nProc, pour s'assurer que le nombre total de simulations reste proportionnel au nombre de processeurs.
   - Le temps d'exécution est mesuré à l'aide de System.currentTimeMillis().
2. Affichage et enregistrement des résultats :
   - Les résultats sont affichés sur la console.
   - Ils sont également enregistrés dans un fichier CSV (CsvOutput.write(...)) pour analyse et traçage.

Concepts illustrés

a) Programmation parallèle avec ExecutorService

Le programme montre comment exécuter un grand nombre de tâches simultanément grâce à un pool de threads. Cela permet d'exploiter les ressources des processeurs disponibles.

b) Synchronisation des accès partagés avec AtomicLong

L'attribut nAtomSuccess est modifié par plusieurs threads en parallèle. L'utilisation de AtomicLong garantit que l'incrémentation est atomique, évitant ainsi les conditions de course et garantissant la cohérence des données.

Limitations et améliorations possibles

Impact de la latence et du cache processeur :

Avec un grand nombre de threads, des problèmes de mémoire et de cache peuvent apparaître, réduisant l'efficacité du parallélisme.

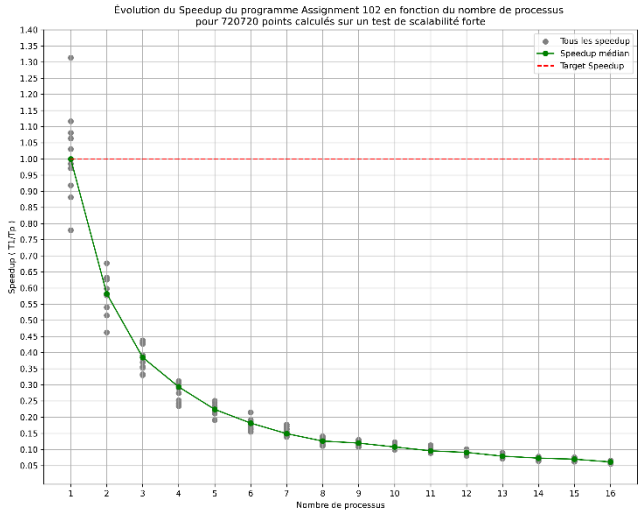
En résumé

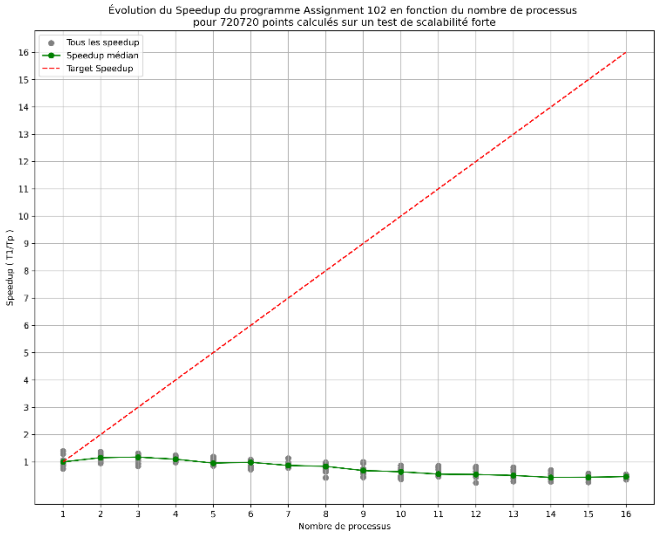
Ce programme met en avant plusieurs concepts clés de la programmation parallèle :

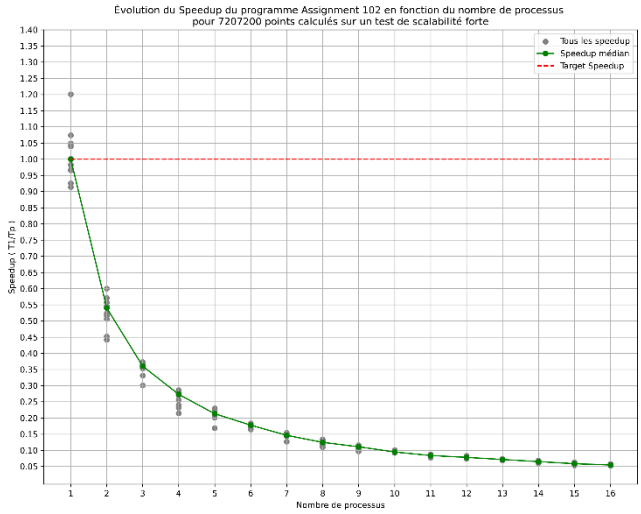
1. Utilisation d'un pool de threads pour paralléliser un calcul probabiliste.
2. Gestion des accès concurrents avec AtomicLong pour éviter les conditions de course.

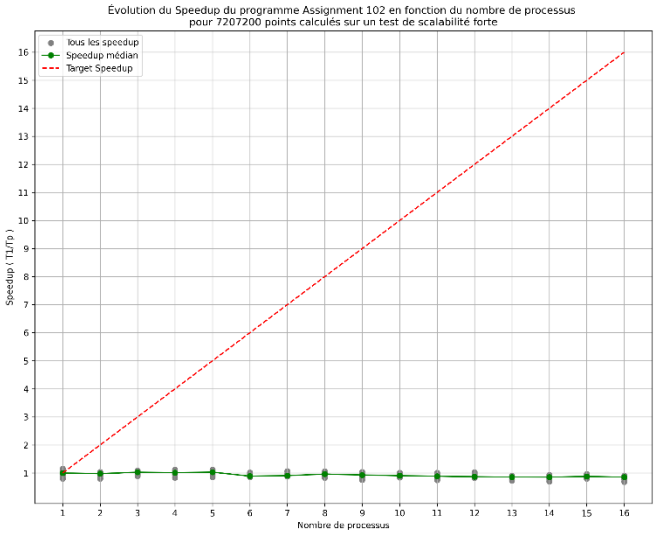
3. Méthode de Monte-Carlo comme exemple d'algorithme massivement parallèle nécessitant peu de synchronisation.

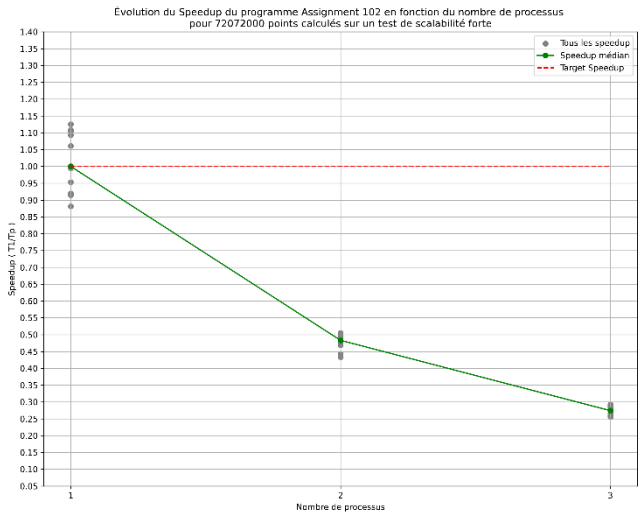
Cette implémentation montre comment tirer parti du parallélisme pour accélérer des calculs intensifs, tout en tenant compte des limites pratiques du matériel et des algorithmes.

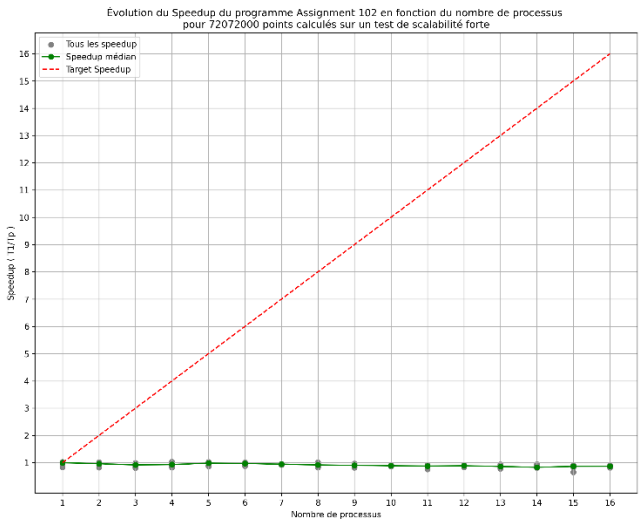




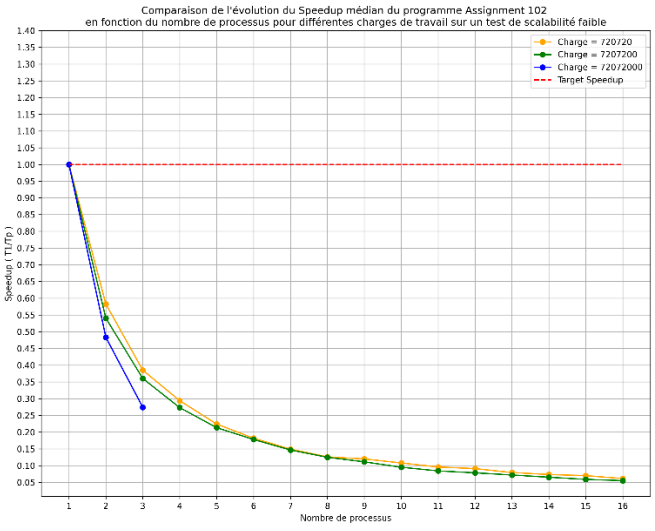


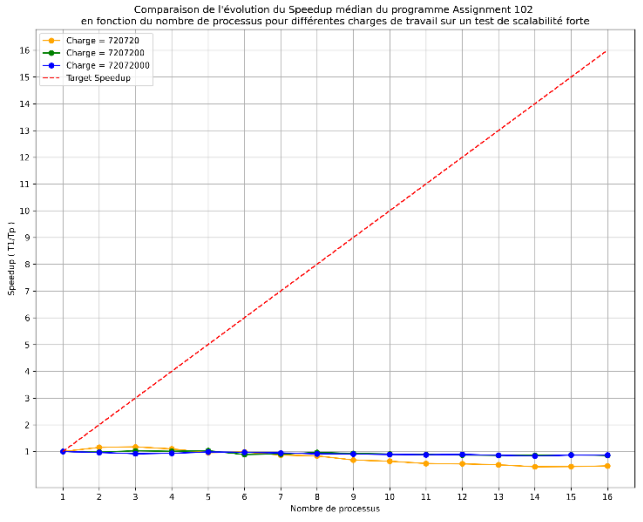






On remarque sur ces courbes de scalabilité que le programme Assignement 102 est très mauvais à paralléliser. En effet, que ce soit en scalabilité forte ou faible, le programme perd en performance au fur et à mesure qu'on augmente le nombre de ressources. En scalabilité forte, on obtient des speedup inférieurs à 1, ce qui est désastreux. On remarque aussi que ce programme ne tient pas une charge trop importante, car il atteint les limites de java dans le dernier graphique de scalabilité faible. Le programme n'a pas pu terminer le test.





Malgré des tests effectués sur plusieurs charges différentes, le programme Assignement 102 reste extrêmement mal optimisé pour de l'exécution parallèle.

Pi.java

Ce programme réalise une approximation de la valeur de π en appliquant la méthode de Monte-Carlo, mais avec une gestion avancée du parallélisme à l'aide de Callable, Future et un pool de threads. Il met en avant une approche différente de celle utilisée dans le code précédent (Assignment102.java), en permettant aux threads de retourner un résultat et de collecter ces résultats efficacement.

Explication du fonctionnement

Le programme divise la tâche de simulation en plusieurs sous-tâches distribuées entre différents threads. Chaque thread exécute un certain nombre d'itérations du processus Monte-Carlo, puis retourne le nombre de points tombant dans le quart de cercle. Un Master récupère ces résultats et les agrège pour calculer une estimation de π.

L'approche utilise :

- Callable au lieu de Runnable : permet à chaque thread de retourner un résultat.
- Future pour récupérer les résultats des threads de manière asynchrone.
- ExecutorService avec un pool de threads fixe (FixedThreadPool) pour gérer l'exécution des tâches de façon optimisée.

Décomposition du code

a) Classe Worker (Exécution parallèle des tâches Monte-Carlo)

La classe Worker représente une tâche exécutée par un thread. Elle implémente Callable<Long>, ce qui permet à chaque instance de retourner un résultat.

- L'attribut numIterations détermine combien de points seront générés par ce worker.
- call() génère des points aléatoires (x, y) et vérifie s'ils tombent dans le quart de cercle.
- Le nombre total de points dans le cercle est retourné sous forme de Long.

Ainsi, chaque thread exécute une fraction du calcul total et retourne son propre nombre de succès, plutôt que de modifier une variable partagée comme dans Assignment102.java.

b) Classe Master (Coordination et agrégation des résultats)

Le rôle de cette classe est de :

1. Créer et distribuer les tâches aux threads
2. Récupérer et agréger les résultats
3. Calculer et afficher la valeur estimée de π
4. Enregistrer les résultats dans un fichier CSV

Méthode doRun(long totalCount, int numWorkers)

1. Création des tâches
    - Un ensemble de tâches Callable<Long> est créé, chacune recevant une fraction de totalCount.
    - On répartit uniformément les simulations entre numWorkers threads.
2. Exécution en parallèle avec ExecutorService
    - On utilise un FixedThreadPool(numWorkers), ce qui crée un nombre fixe de threads.

- Les tâches sont soumises via invokeAll(tasks), qui retourne une liste de Future<Long>.

3. Récupération des résultats
   - Les valeurs retournées par chaque worker sont récupérées avec future.get().
   - La somme des valeurs donne le nombre total de points dans le cercle.
4. Calcul de π et affichage des résultats
   - La valeur estimée de π est obtenue par la même formule que précédemment présentée.
   - L'erreur relative par rapport à la vraie valeur de π est également calculée.
   - La durée d'exécution est mesurée.
5. Sauvegarde des résultats dans un fichier CSV
   - logToCSV(...) enregistre les résultats expérimentaux pour une analyse ultérieure.

c) Classe Pi (Programme principal - main)

La classe Pi est responsable du lancement des expériences avec différentes valeurs de totalCount (nombre total de simulations).

1. Définition des paramètres expérimentaux
   - tabTC contient différentes valeurs pour totalCount, augmentant progressivement la taille de l'échantillon.
   - Le programme exécute chaque configuration 10 fois pour réduire les biais.
2. Lancement des expériences avec Master
   - doRun(count, nwork) est appelé avec count (nombre total de points simulés) et nwork = 12 (nombre de workers).
   - Le nombre total de points dans le cercle est affiché à la console.

Concepts illustrés

a) Utilisation de Callable et Future pour gérer le parallélisme

Contrairement à Runnable, qui ne permet pas de retourner une valeur, Callable permet à chaque thread de renvoyer son propre résultat. Cela facilite l'agrégation des résultats et élimine le besoin d'une variable partagée et synchronisée (AtomicLong dans le code Assignment102.java).

b) Gestion efficace des threads avec un pool fixe

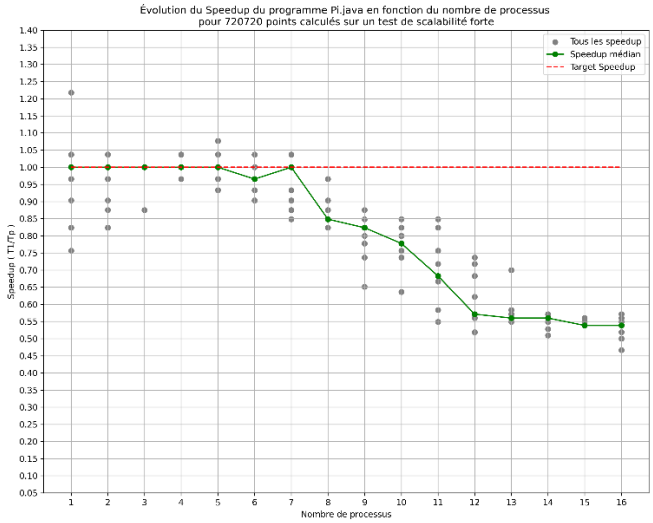
L'utilisation d'un pool de threads fixe (FixedThreadPool) est plus efficace que de créer un grand nombre de threads dynamiques. Elle permet de limiter l'utilisation des ressources CPU/mémoire.

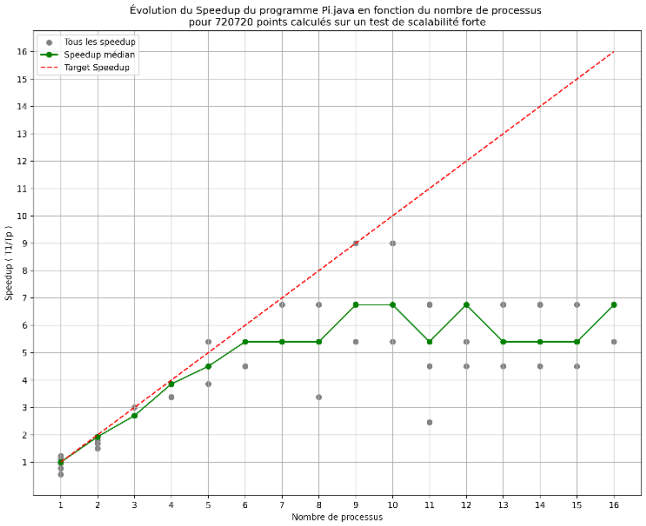
Différences avec Assignment102.java

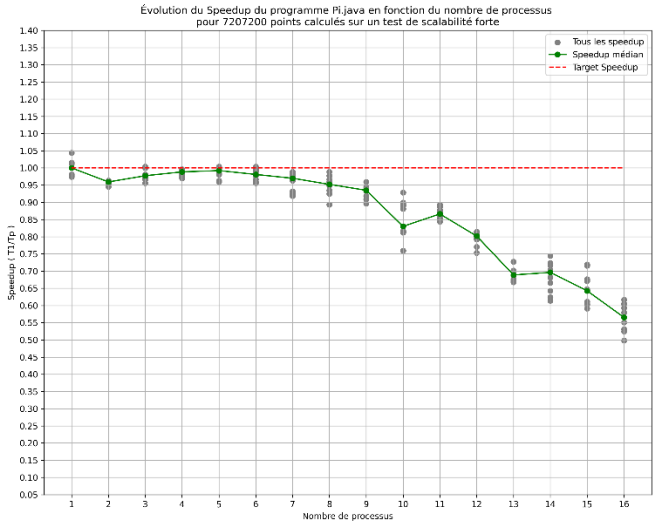
Aspect	Pi.java	Assignment102.java
Parallélisme	Callable, Future, FixedThreadPool	Runnable, WorkStealingPool
Retour de valeurs	Chaque thread retourne un résultat avec Future	AtomicLong modifié directement par plusieurs threads
Gestion des threads	FixedThreadPool(numWorkers) avec invokeAll()	WorkStealingPool(nProcessors) avec execute()
Attente des threads	Future.get() récupère les résultats	Boucle while (!executor.isTerminated())

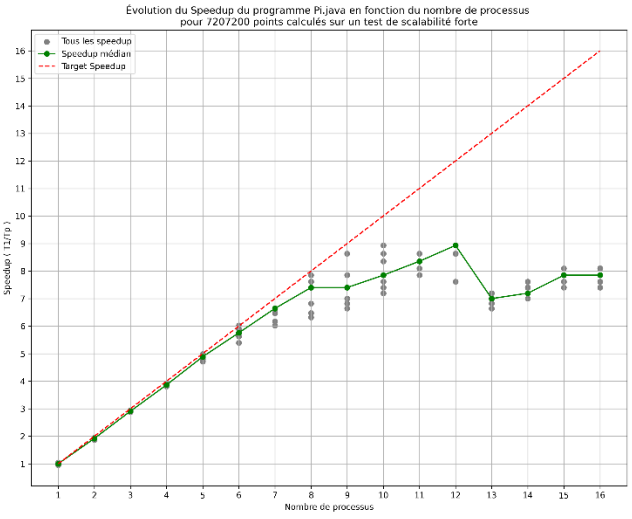
En résumé

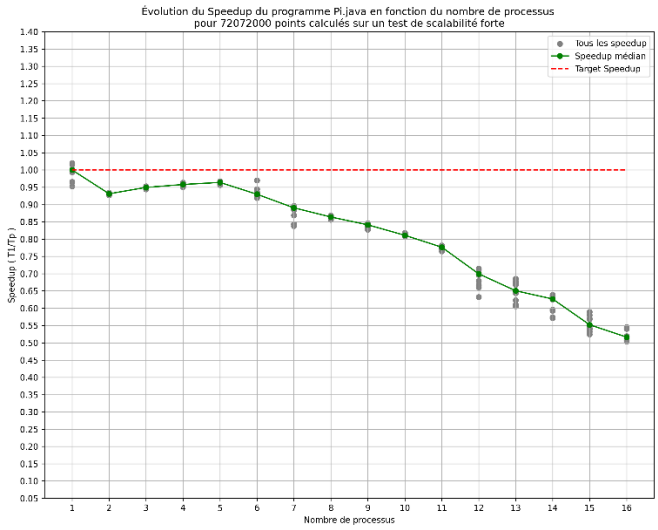
Ce programme illustre une approche avancée de la programmation parallèle avec Callable et Future, permettant une gestion plus efficace des résultats des threads. L'utilisation d'un pool de threads fixe améliore la gestion des ressources, et l'enregistrement des résultats permet une analyse expérimentale de la précision et des performances de la méthode de Monte-Carlo.

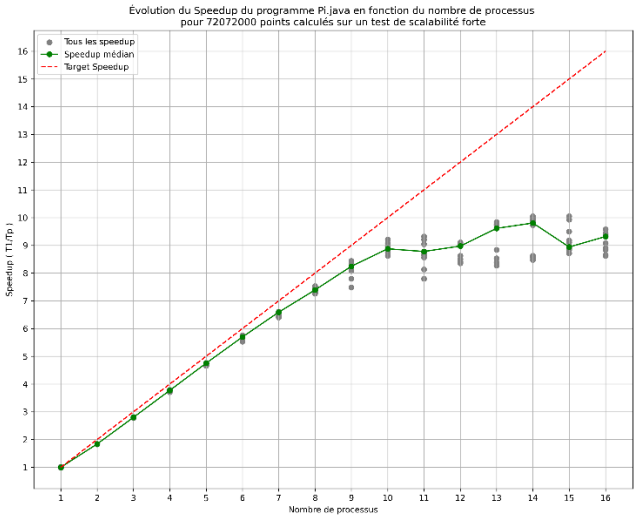


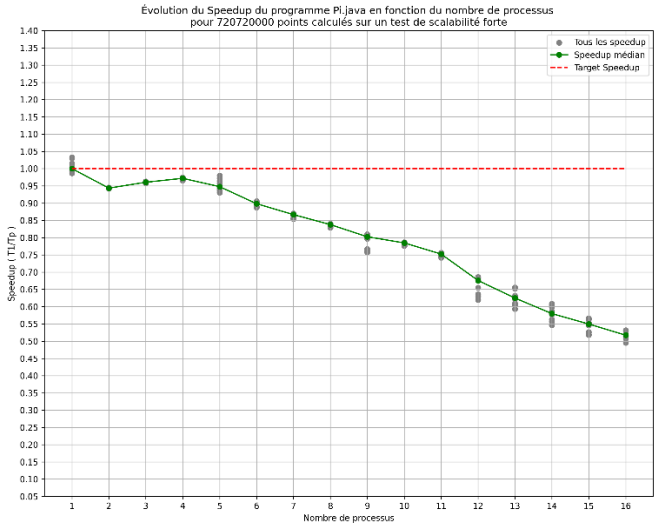


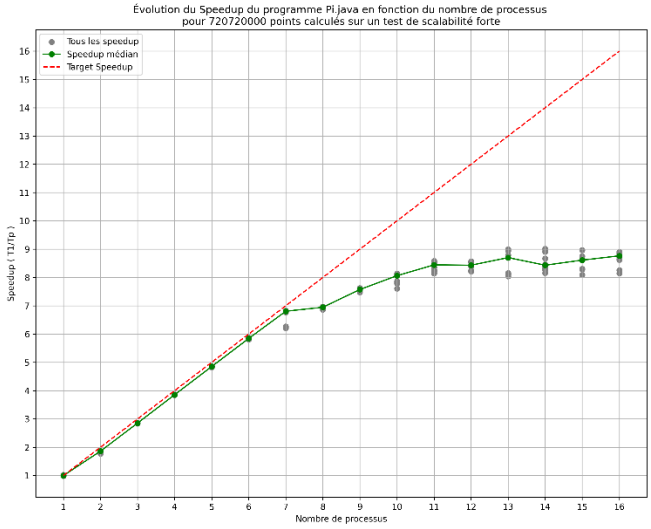




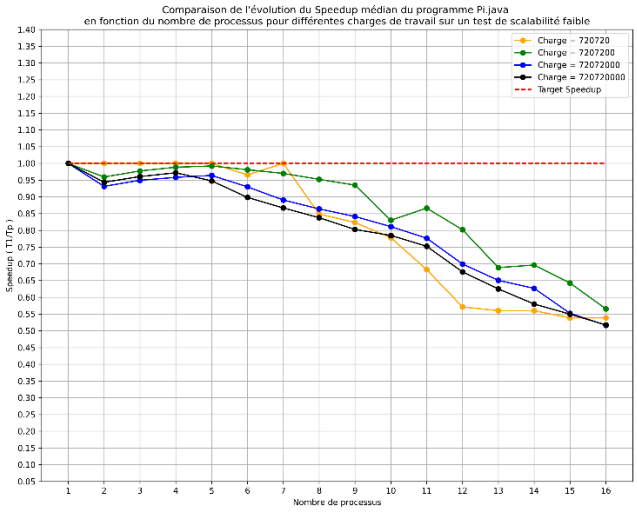


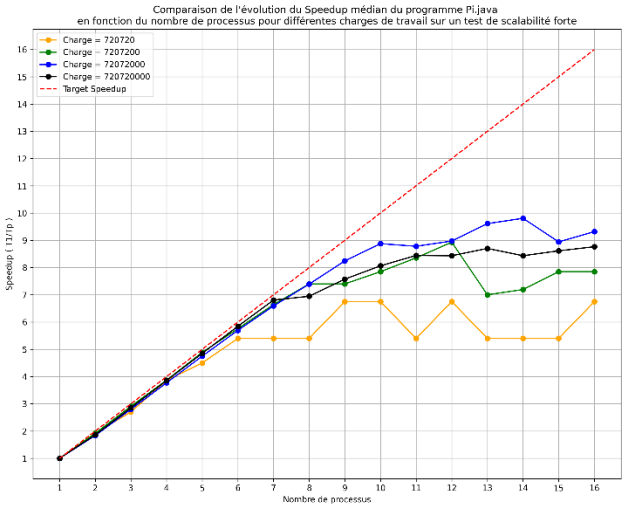






Ces courbes de scalabilité faible et fortes montre que le programme Pi.java s'est bien parallélisé, et les résultats sont satisfaisants.





En comparant ces graphiques, en scalabilité faible, on remarque que la charge idéale est de l'ordre de 7 millions de points calculés. Cependant, avec d'autres charges, le programme reste très performant lorsqu'il est parallélisé. En scalabilité forte, nous remarquons que la charge idéale est de l'ordre de 70 millions de points calculés. Les autres charges sont aussi satisfaisantes, sauf pour celle de 720720 points qui stagne lorsqu'il y a trop de ressources. Cela s'explique par la granularité des tâches. Les temps d'accès en mémoire et d'envoi de message ont un impact comparable à la charge mise pour chaque ressource.

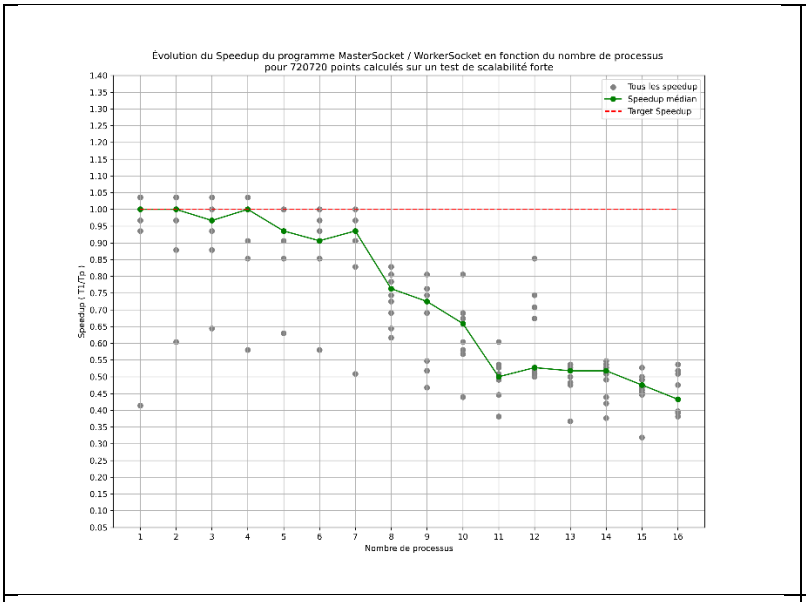
Master Socket / Worker Socket

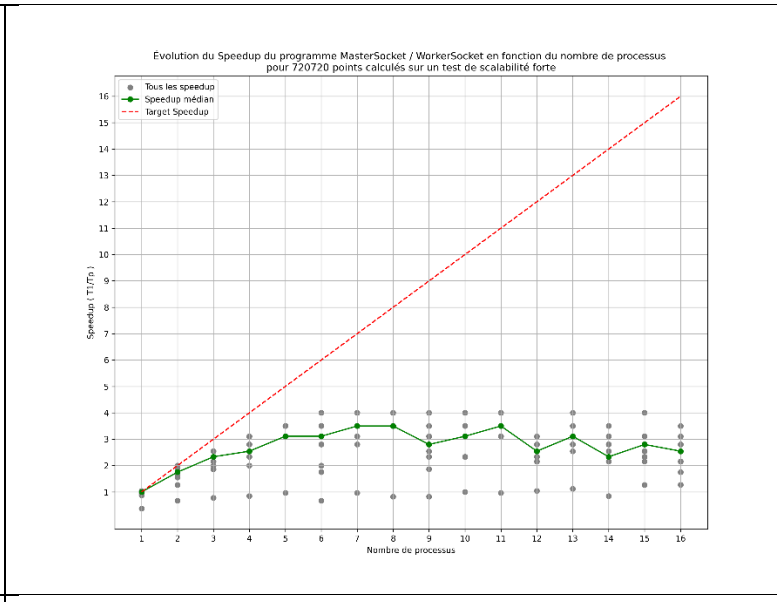
Ce code implémente deux approches pour approximer la valeur de π (Pi) en utilisant la méthode de Monte Carlo :

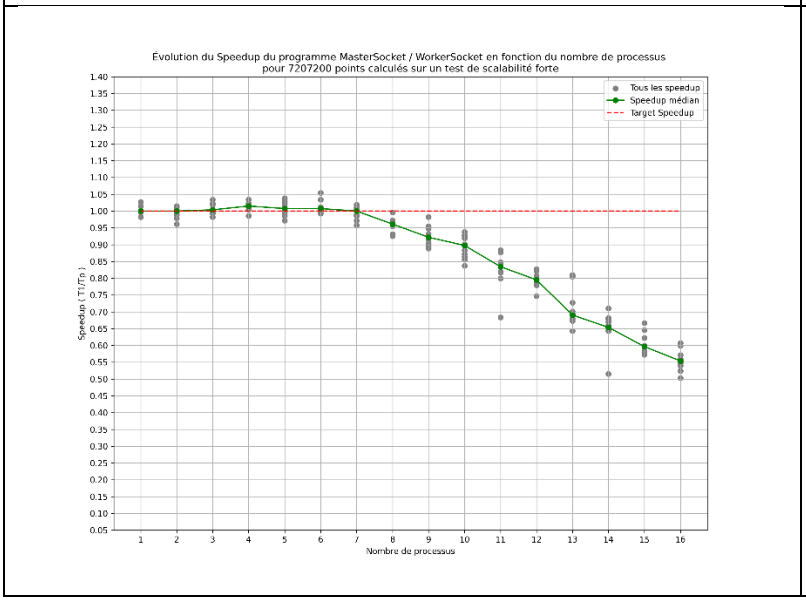
1. Code 2 (Pi.java)
    - Utilise un pool de threads (ExecutorService) pour répartir les calculs entre plusieurs threads.
    - Un Master crée des Worker qui effectuent les simulations et retournent leurs résultats.
    - La valeur de π est estimée à partir des résultats agrégés de tous les threads.
2. Code 3 (MasterSocket.java & WorkerSocket.java)
    - Implémente une architecture client-serveur où MasterSocket (le client) envoie des requêtes aux WorkerSocket (les serveurs).
    - Chaque WorkerSocket effectue des simulations et renvoie le nombre de points à l'intérieur du quart de disque.
    - Le MasterSocket regroupe les résultats et calcule une approximation de π.
    - Les résultats sont stockés dans des fichiers CSV avec des noms dynamiques (dataMW_SFa*.csv).

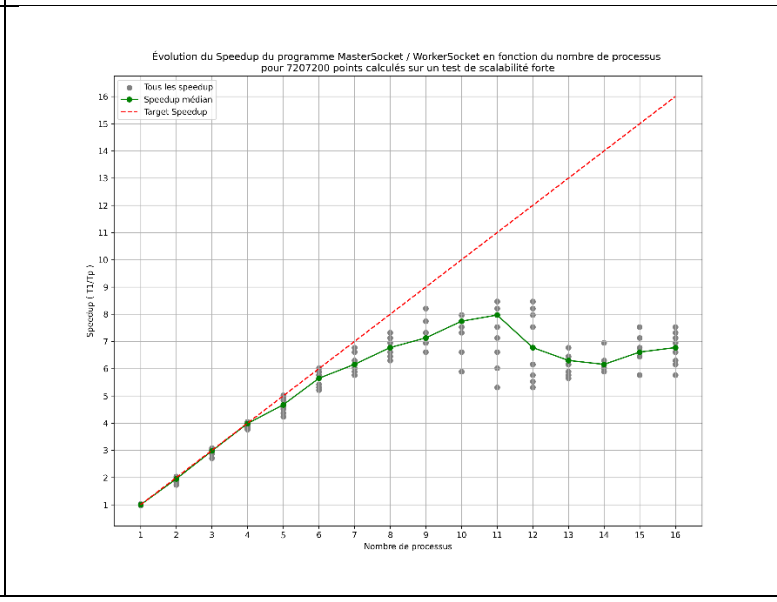
Comparaison des deux implémentations :

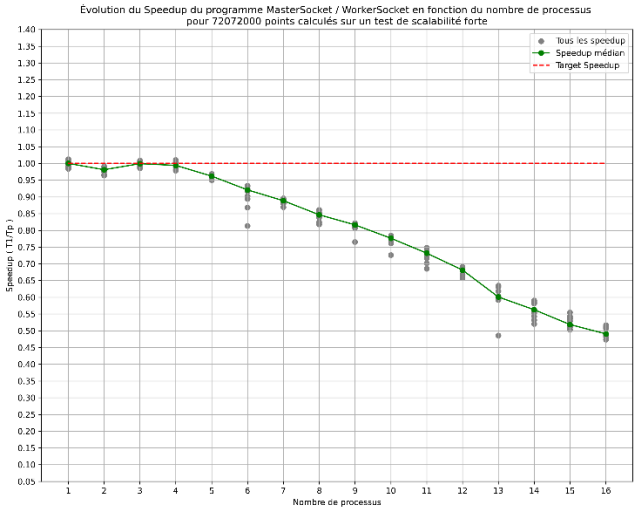
Critère	Pi.java (Threads)	MasterSocket/WorkerSocket.java (Sockets)
Communication	Interne (même processus)	Réseau (clients-serveurs)
Parallélisme	Multithreading (exécuteur)	Multiprocessus réparti sur plusieurs machines
Facilité de mise en œuvre	Plus simple (gère la concurrence avec ExecutorService)	Plus complexe (gestion des sockets et de la communication réseau)
Performances	Dépend du nombre de cœurs	Dépend de la latence réseau et du nombre de serveurs disponibles
Scalabilité	Limité à une seule machine	Peut être distribué sur plusieurs machines

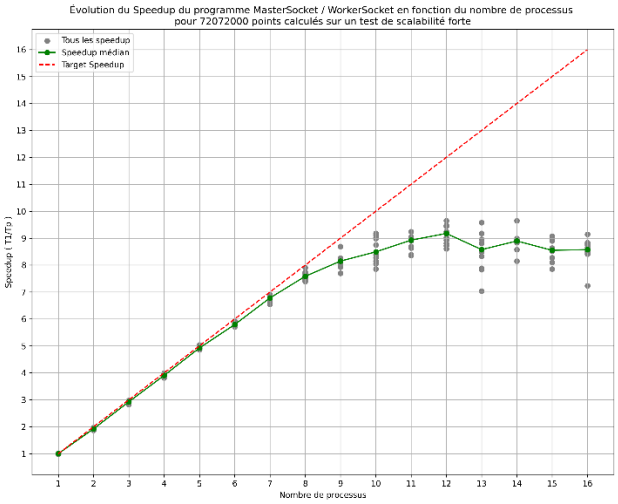


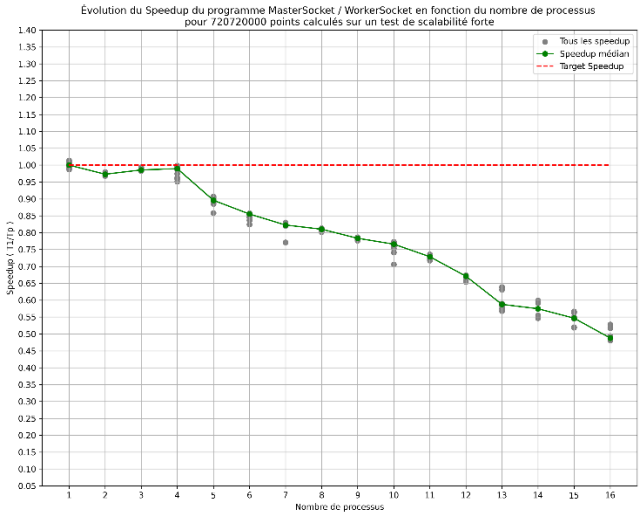


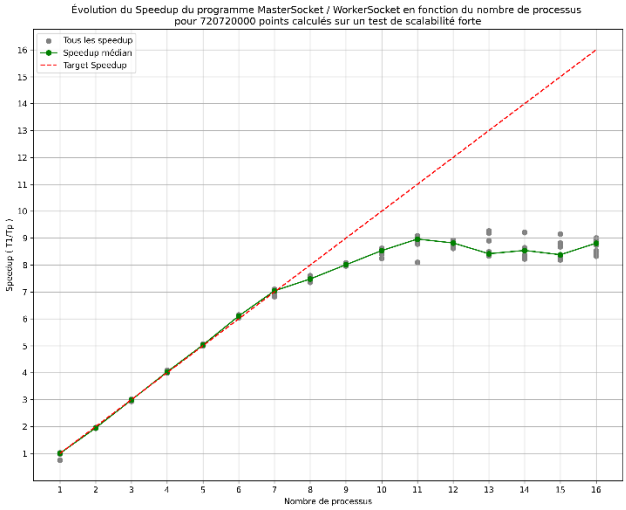




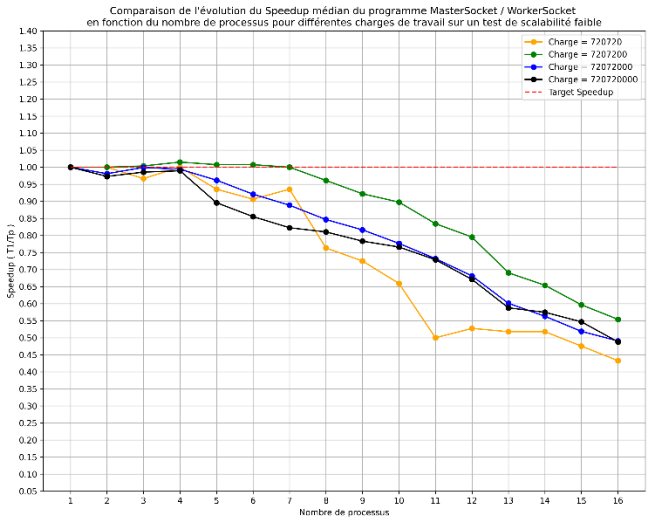


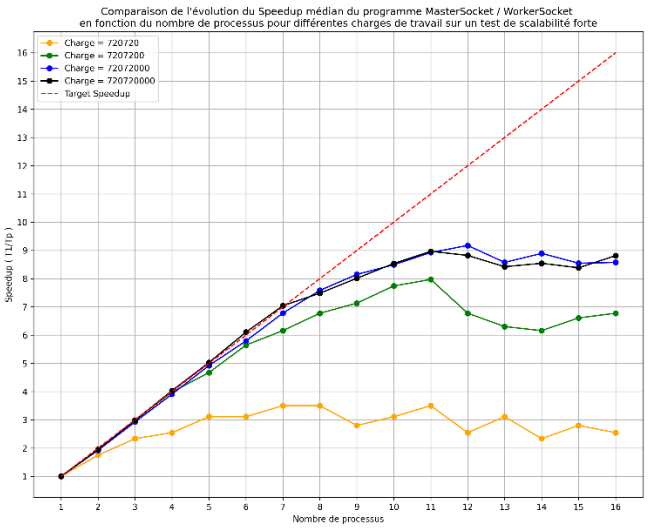






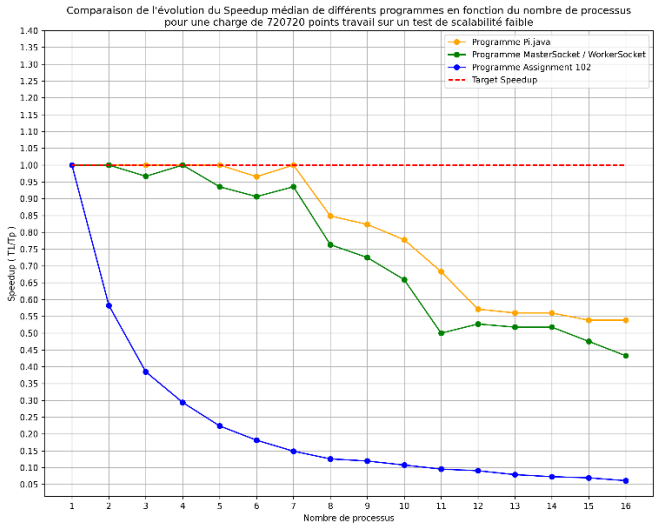
Etant donné que ce programme implémente Pi.java, nous obtenons des résultats de scalabilité forte et faible aussi satisfaisants.

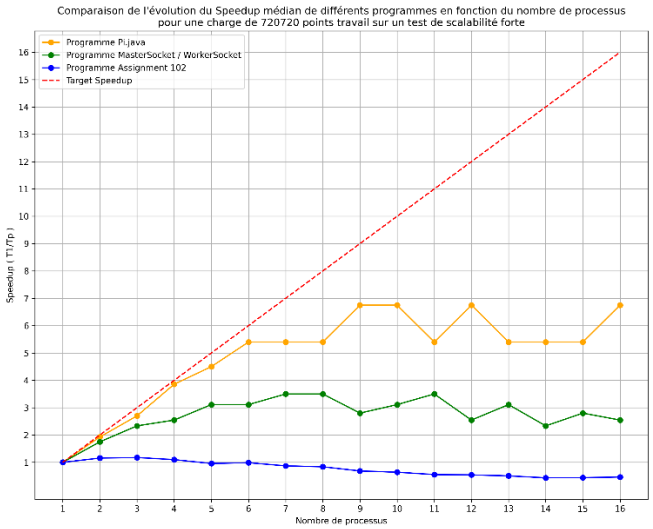


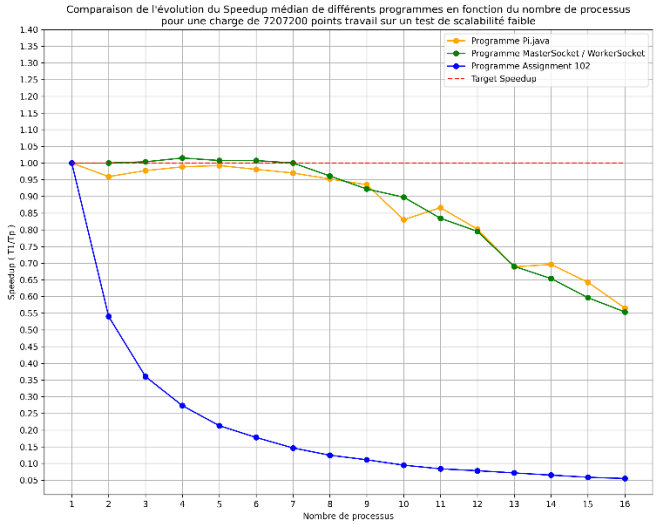


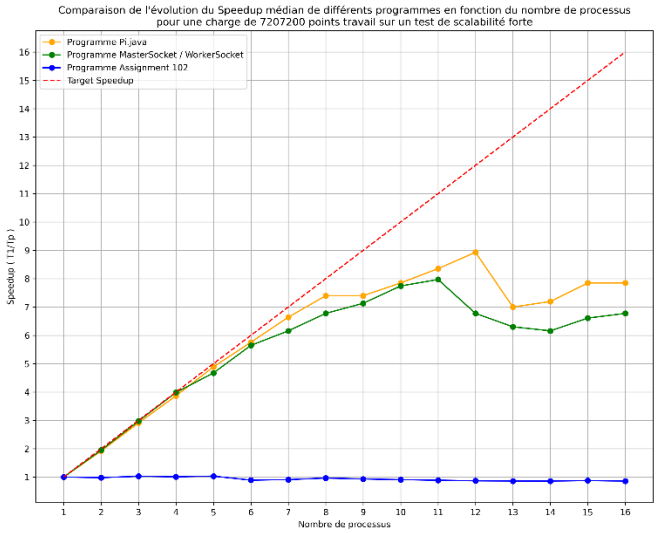
Il en va de même pour la comparaison des exécutions sur différentes charges.

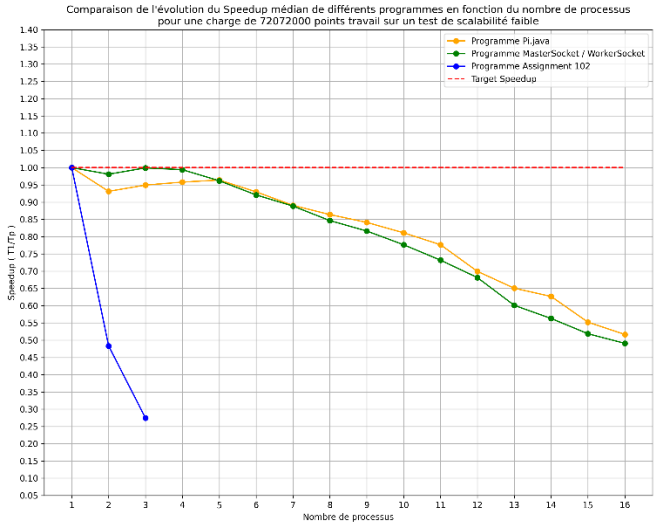
Comparaison des différents programmes :

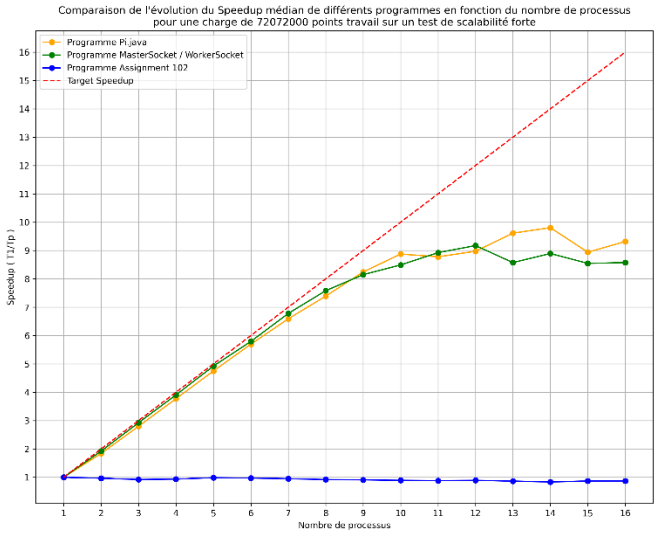


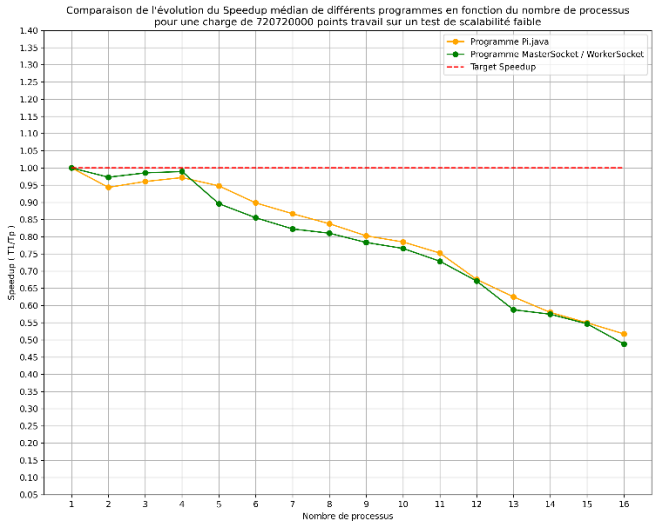


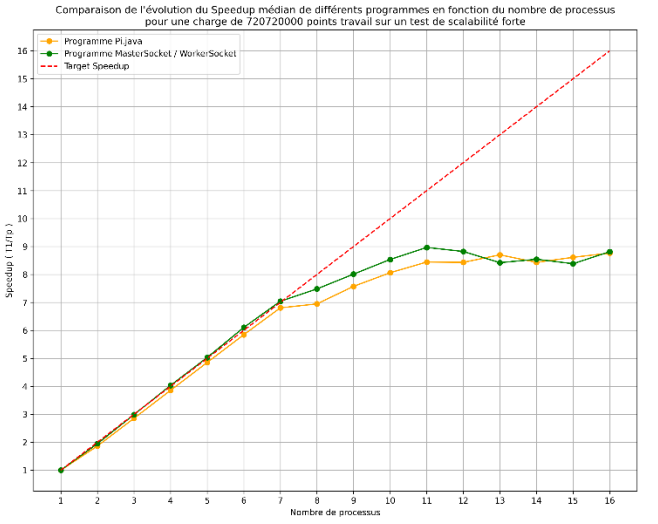

Comparaison de l'évolution du Speedup médian de différents programmes en fonction du nombre de processus
pour une charge de 7207200 points travail sur un test de scalabilité faible
Programme Pi.java
Programme MasterSocket / WorkerSocket
Programme Assignment 102
Target Speedup
Speedup (T1/Tp)
Nombre de processus


Comparaison de l'évolution du Speedup médian de différents programmes en fonction du nombre de processus
pour une charge de 7207200 points travail sur un test de scalabilité forte
Programme Pi.java
Programme MasterSocket / WorkerSocket
Programme Assignment 102
Target Speedup
Speedup (T1/Tp)
Nombre de processus


Comparaison de l'évolution du Speedup médian de différents programmes en fonction du nombre de processus
pour une charge de 72072000 points travail sur un test de scalabilité faible
Programme Pi.java
Programme MasterSocket / WorkerSocket
Programme Assignment 102
Target Speedup
Speedup (T1/Tp)
Nombre de processus


Comparaison de l'évolution du Speedup médian de différents programmes en fonction du nombre de processus
pour une charge de 72072000 points travail sur un test de scalabilité forte
Programme Pi.java
Programme MasterSocket / WorkerSocket
Programme Assignment 102
Target Speedup
Speedup (T1/Tp)
Nombre de processus


Comparaison de l'évolution du Speedup médian de différents programmes en fonction du nombre de processus
pour une charge de 720720000 points travail sur un test de scalabilité faible
Programme Pi.java
Programme MasterSocket / WorkerSocket
Target Speedup
Speedup (T1/Tp)
Nombre de processus


Comparaison de l'évolution du Speedup médian de différents programmes en fonction du nombre de processus
pour une charge de 720720000 points travail sur un test de scalabilité forte
Programme Pi.java
Programme MasterSocket / WorkerSocket
Target Speedup
Speedup (T1/Tp)
Nombre de processus

Ici, nous retrouvons les courbes des différents programmes en scalabilité faible et forte pour différentes charges. Sans surprise, le programme Assignement 102 présente des courbes désastreuses comparé aux deux autres programmes.

Pour ce qui est des comparaisons entre les courbes des programmes Pi.java et Master Worker Socket, les deux courbes sont très proches, car elle implémente toutes les deux la même méthode de calcul. Hors, on remarque que Master Worker Socket est généralement un peu en dessous de Pi.java. Cela est du au fait que ce programme utilise des accès au réseau qui sont beaucoup plus long que des accès en mémoire. Ainsi, la performance de ce programme est fortement impactée par ces envois de messages.

Conclusion

Ce cours de programmation avancée et de qualité de développement nous a permis d'acquérir des compétences essentielles en programmation parallèle, en gestion de mémoire et en respect des normes de qualité logicielle. Les concepts étudiés et les pratiques mises en œuvre nous ont préparés à développer des applications performantes et robustes dans des environnements complexes.