Bastien Clément

Labo 02 : Mesure des performances des entrées-sorties Java

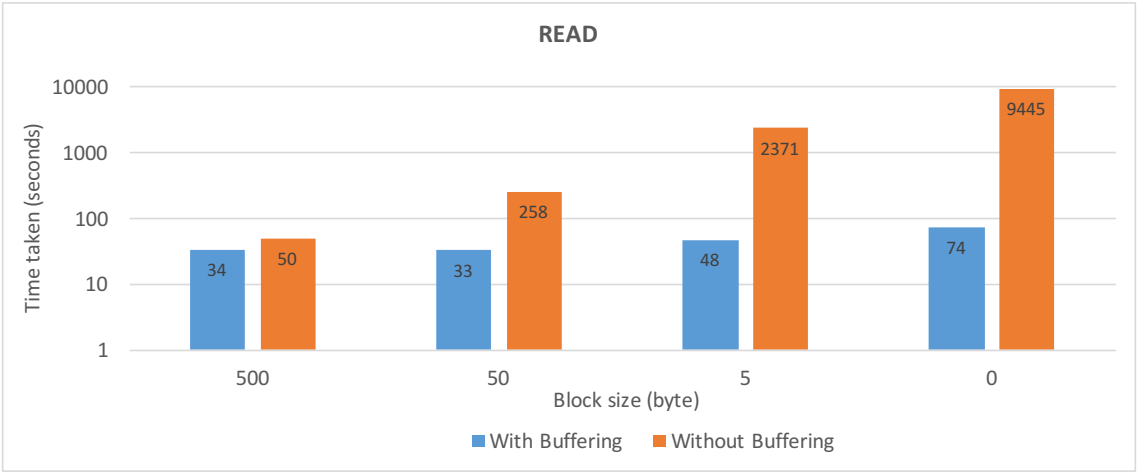
1 Conditions de l'expérience

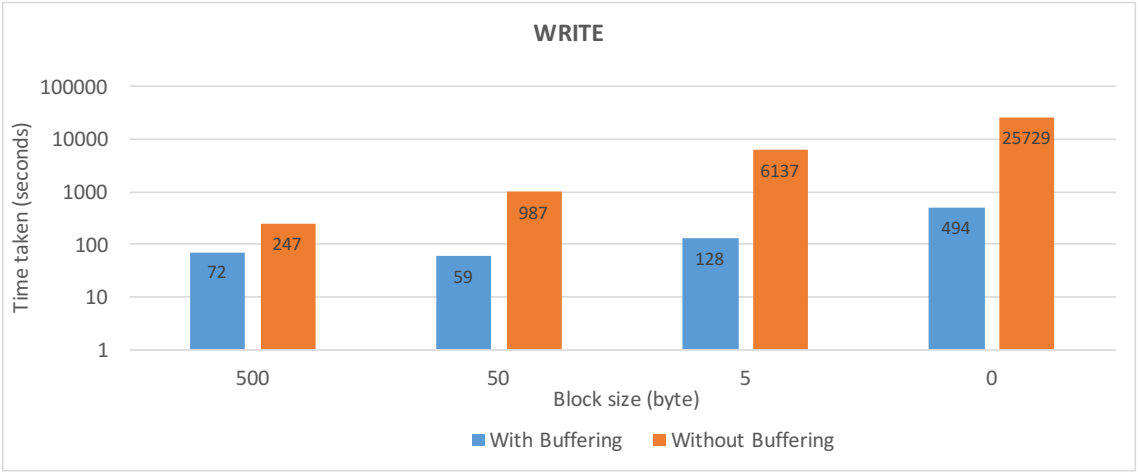
L'objectif de ce laboratoire est de mesurer et comparer les performances d'entrées-sorties en Java. Plus spécifiquement, nous observerons les différences de performances entre la lecture-écrite byte par byte ou par bloc, ainsi que l'impact de l'utilisation des versions *buffered* des flux d'entrées-sorties.

Les mesures ont été effectuées sur un Macbook Air (13-inch, Mid 2012) avec un processeur 1.8 GHz Intel Core i5, 4 GB de RAM, d'un disque SSD (aux caractéristiques inconnues) et Mac OS 10.11.

2 Mesures

Les graphiques ci-dessous représentent les résultats obtenus, une fois pour la lecture, une fois pour l'écriture. L'axe vertical représente le temps total nécessaire pour exécuter une mesure particulière, ainsi les valeurs les plus faibles sont les meilleures. Une taille de bloc de 0 correspond à une lecture byte par byte.





3 Analyse

Nous observons immédiatement que l'utilisation du buffering est toujours avantageuse, que ce soit en lecture ou écriture, et indépendamment de la taille des blocs utilisée. La taille des blocs ne semble n'avoir une influence sur les performances d'entrées-sorties que lorsque nous utilisons des flux sans buffering. Avec buffering, la taille des blocs influe considérablement moins sur les performances. La raison est probablement que le buffering est en pratique équivalent à la lecture par plus gros blocs.

Malgré cela, et particulièrement pour l'écriture, l'utilisation d'entrées-sorties byte par byte semble sensiblement plus coûteuse que les entrées-sorties par blocs, et ceci même avec utilisation du buffering. J'imagine que le temps supplémentaire est probablement dépensé dans la JVM. En effet il n'y a pas de raison que la fréquence des appels système de lecture ou d'écriture soient différents puisque le flux utilise de toute façon des lectures par blocs, de même pour les performances d'entrées-sorties du disque. À l'inverse, la lecture byte par byte implique un nombre beaucoup plus important d'itérations de la boucle de lecture/écriture et d'appels de fonction dans le programme Java.

Il me parait aussi important de noter que la méthodologie utilisée lors des mesures de performances ne me semble tout à fait rigoureuse. Premièrement, seul 10 MB sont lus ou écrits. Les temps mesurés sont de quelques millisecondes. À cette échelle, une petite perturbation due aux processus parallèles sur le système peuvent représenter une variation relativement importante sur la mesure. Effectuer les test avec des fichiers plus gros serait probablement une bonne idée. De même, la mesure n'est effectuée qu'une seule fois, pour plus de précision, réitérer l'expérience conduirait à des résultats plus fiables.

Finalement, les mécanismes de cache du système de fichier et du noyau ne sont pas du tout considérés. En pratique, la lecture d'un fichier qui vient d'être écrit (tel que c'est le cas dans ce laboratoire) peut probablement être effectuée directement depuis les caches du système de fichier, sans accès réel au disque. Cela ne semble cependant pas être le cas ici.

4 Explications

Afin de générer le fichier `.csv`, j'ai choisi une implémentation simple et directe, sans particulièrement me soucier de la réutilisabilité du code écrit.

Ainsi j'ai implémenté deux classes `TestResult` et `TestOutputBuilder`.

`TestResult` représente les résultats d'un test spécifique. C'est un bête conteneur de valeurs sans aucune méthode, c'est pourquoi j'ai choisi de déclarer tous les attributs `public static final` au lieu d'utiliser des getters. Ce modèle est inspiré des *case class* du langage Scala.

`TestOutputBuilder` est un objet construit avec comme paramètre un `OutputStream` dans lequel les données CSV textuelles doivent être écrites. Sa méthode `headers(String ...headers)` permet de spécifier une ligne d'en-têtes à écrire sur le flux de sortie avant les premier résultats. La méthode `add(TestResult r)` permet d'écrire une ligne sur le flux de sortie, correspondant au résultat d'un test en particulier. Finalement la méthode `close()` ferme le flux sous-jacent et vide ainsi tous les tampons entre le flux haut-niveau et le fichier sur le disque.

Lors de la construction de l'objet `TestOutputBuilder` dans la fonction `main`, j'ai également décoré mon `FileOutputStream` avec un `BufferedOutputStream`, puisque comme le test a pu le montré, les performances sont supérieures à celles d'un flux non décoré.

J'ai également choisi de modifier le type de retour de certaines fonctions afin de faire remonter les résultats des tests jusque dans la fonction principale.