Basile Vu

RES – Laboratoire BufferedIOBenchmark

Introduction

Dans ce laboratoire, nous allons nous concentrer sur l'analyse des temps d'écriture et de lecture de données dans des fichiers, en fonction de différentes stratégies (Stream buffered ou non, écriture byte par byte ou bloc par bloc, en variant les tailles, etc.). Pour ce faire, nous utiliserons des graphiques dont les valeurs seront les sorties d'un programme de test fourni.

Environnement utilisé

Avant toute chose, il peut être utile de mentionner l'environnement utilisé pour lancer les tests.

- Processeur : Inter®Core™ i7-2760QM @ 2.4 GHz
- OS : Windows 8.1, 64 bits
- Disque dur : 5400 tours/minute (Toshiba MQ01ABD100)

Améliorations apportées au programme fourni

La première amélioration apportée au programme est **l'ajout de tests supplémentaires** (plus de tailles de blocs testées) afin d'avoir une idée plus précise du comportement du temps d'exécution en fonction de la taille des blocs. Ce n'est donc plus des tailles de bloc de 500 à 5 que l'on testera mais de 50 000 000 à 5 (toujours en divisant la taille par 10 à chaque fois). Pour ce faire, on utilisera simplement une boucle.

La deuxième amélioration, la plus importante dans ce laboratoire, consiste à **exporter les résultats dans un fichier**. En effet, il esst fastidieux de devoir relever toutes les valeurs depuis la console (particulièrement si les tests sont nombreux). Le fichier étant au format CSV, son traitement en est grandement facilité et on peut facilement l'importer dans une feuille de calcul afin de tracer des graphiques. Voyons donc comment nous allons implémenter cette amélioration.

Le **fichier CSV** devrait avoir idéalement, sur chaque ligne, le temps d'exécution en fonction de l'opération, de la stratégie, de la taille des blocs et de la taille du fichier. Pour ce faire, l'idée est de **grouper les données** ensemble au moyen d'un objet. Dans notre cas, il s'agit de la classe **BufferedIOBenchmarkData**. Afin de rendre le code plus générique, on peut la faire implémenter un interface **IData**, qui demande de mettre à disposition les méthodes *get* et *getValues*, respectivement pour récupérer une valeur donnée (avec une clé) ou toutes d'un coup. De cette manière, on pourrait réutiliser cet interface plus tard si l'on voulait créer une autre classe qui stockerait des données.

Maintenant que l'on a nos données regroupées, il nous faudrait un moyen de « formatter » ces données pour que l'on puisse les écrire séparées par une virgule et avec un retour à la ligne lorsque l'on a fini. Pour ce faire, on créera donc une classe **CsvSerializer** qui nous permettra d'écrire ces données au format voulu dans un stream. Elle aussi implémentera un interface, **ISerializer**, réutilisable. Ce dernier demande d'implémenter une méthode *serialize*, qui envoie dans un PrintStream un IData formatté comme on le souhaite.

Ensuite, il ne nous reste plus qu'à créer une classe s'occupant de gérer le stream dans lequel on va écrire nos données sérialisées. Du nom de **FileRecorder**, cette classe implémente un interface IRecorder, qui demande d'implémenter la méthode *record*. Dans l'idée, cette dernière enregistre une IData. Dans notre cas, notre FileRecorder l'enregistre, formattée par notre CsvSerializer, dans un fichier auquel on peut accéder grâce au PrintStream qu'on lui passe. Ce dernier est créé à l'extérieur de notre FileRecorder, ce qui laisse à l'utilisateur le soin de gérer la création et la fermeture du PrintStream.

Note : Un autre choix d'implémentation consisterait à créer le stream dans le FileRecorder et mettre à disposition une méthode *close*. Il faudrait dans ce cas aussi mettre à disposition une méthode close dans la

classe BufferedIOBenchmark. Le choix est libre; adopter une de ces implémentations revient à choisir entre passer un nom de fichier au constructeur puis appeler la méthode close ou passer un PrintStream.

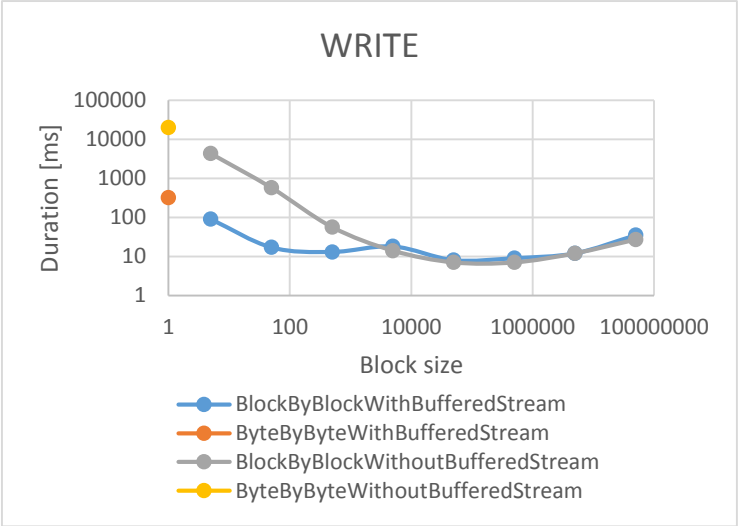
Finalement, dans notre classe **BufferedIOBenchmark**, il ne nous reste plus qu'à créer un FileRecorder et de record une nouvelle IData aussi souvent que nécessaire (donc simplement à la fin d'une opération). Un constructeur pour BufferedIOBenchmark a été rajouté afin de set le PrintStream et d'initialiser la première ligne du fichier csv (noms des colonnes). On notera que cette dernière opération ne se fait pas dans le constructeur du FileRecorder comme on aimerait que ce dernier soit aussi générique que possible.

Analyse des résultats

Pour comparer les résultats obtenus, utilisons notre fichier csv pour tracer des graphiques. On a la taille des blocs (Block size) sur l'axe des abscisses et le temps d'exécution (Duration, en millisecondes) sur l'axe des ordonnées. Des échelles logarithmiques sont utilisées ici pour les deux axes afin de mieux présenter les données.

Note : Une taille de 0 n'est pas acceptée par une échelle logarithmique. Les tailles des blocs des stratégies de type ByteByByte sont 0, mais elle est mise à 1 ici pour pouvoir représenter les temps (c'est le plus important).

Intéressons-nous tout d'abord aux opérations d'écriture (**WRITE**).

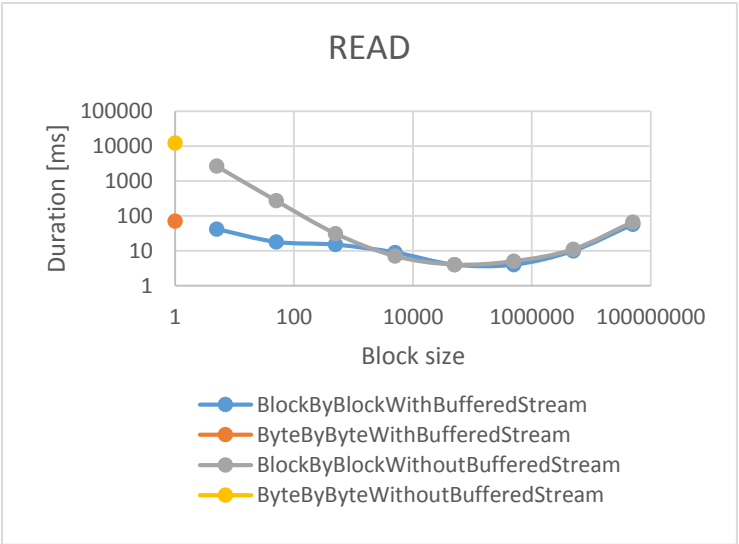


Commençons par observer que pour les stratégies byte par byte, un stream buffered est bien meilleur qu'un non buffered (d'un facteur 60 environ ici).

On remarque que ça a aussi l'air d'être le cas pour les deux autres stratégies par bloc (bien que les deux stratégies semblent tendre vers la même valeur pour des tailles de bloc grandes).

On note aussi que plus la taille des blocs est grande, plus le temps d'exécution semble diminuer. Cependant, cela n'est vrai que jusqu'à une taille de 50000, où on atteint un minimum de 7 ms (pour les deux stratégies) dans notre cas. Passé ce point, le temps augmente à nouveau avec la taille des blocs.

On notera qu'il y a un petit pic des temps avec une taille de blocs de 5000, dans la stratégie avec block bufferisée.



Penchons-nous à présent sur les opérations de lecture (**READ**). Ici aussi, tout ce que nous avions pu constater précédemment avec les opérations d'écriture semble avoir lieu. La différence principale avec l'écriture est que la lecture se fait visiblement plus rapidement.

On peut en tirer en conclusion que les stream bufferisés sont en général meilleurs que les non bufferisés, typiquement lorsqu'on lit byte par byte. En bloc par bloc, nous avons pu constater que les performances changeaient suivant la taille et qu'il peut être intéressant d'effectuer quelques benchmarks pour trouver celle qui convient le mieux en fonction du contexte du programme.