RES : exemple 01 "BufferedIOBenchmark"

1 Présentation

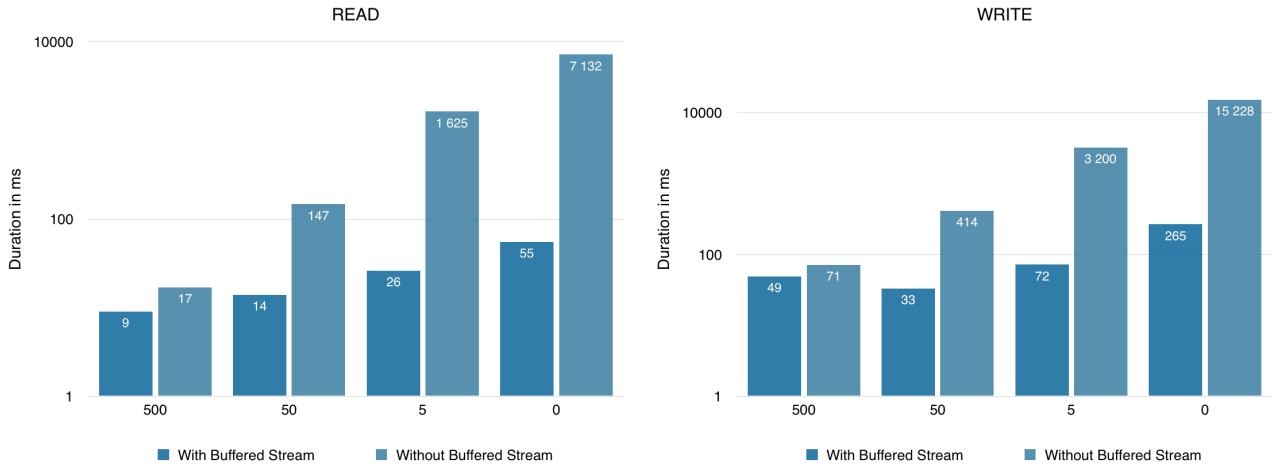
Ce programme à pour objectif de permettre de visualiser les différences en terme de durée d'écriture et de lecture lors de l'utilisation ou non des buffers d'entrées/sorties.

Les tests ont été effectués sur un ordinateur Macbook Pro 2013, Intel Core i7 2.3 GHz.

2 Résultats

operation	strategy	blockSize	fileSizeInBytes	durationInMs
WRITE	BlockByBlockWithBufferedStream	500	10485760	49
WRITE	BlockByBlockWithBufferedStream	50	10485760	33
WRITE	BlockByBlockWithBufferedStream	5	10485760	72
WRITE	ByteByByteWithBufferedStream	0	10485760	265
WRITE	BlockByBlockWithoutBufferedStream	500	10485760	71
WRITE	BlockByBlockWithoutBufferedStream	50	10485760	414
WRITE	BlockByBlockWithoutBufferedStream	5	10485760	3200
WRITE	ByteByByteWithoutBufferedStream	0	10485760	15228
READ	BlockByBlockWithBufferedStream	500	10485760	9
READ	BlockByBlockWithBufferedStream	50	10485760	14
READ	BlockByBlockWithBufferedStream	5	10485760	26
READ	ByteByByteWithBufferedStream	0	10485760	55
READ	BlockByBlockWithoutBufferedStream	500	10485760	17
READ	BlockByBlockWithoutBufferedStream	50	10485760	147
READ	BlockByBlockWithoutBufferedStream	5	10485760	1625
READ	ByteByByteWithoutBufferedStream	0	10485760	7132

2.1 Comparaison des vitesses de lecture et d'écriture



L'utilisation du buffer apporte un gain de vitesse conséquent lors de la lecture, notamment lors de la lecture de petits blocs voir même de simples Bytes. Ceci, car comme vu durant le cours, les buffers permettent de "pré-charger" plusieurs Bytes (en fonction de la taille du buffer) même lors de la lecture d'un seul Byte. Ainsi, si une nouvelle lecture doit être faite, celle-ci sera effectuée en mémoire et non dans le fichier directement (ou un autre support), ce qui est beaucoup plus rapide. C'est pour ça que plus les blocs lus sont petits, plus il est nécessaire de retourner lire le fichier.

Il en est de même pour l'écriture dans un fichier. Plus les blocs sont petits, plus le temps de lecture total est grand.

3 Modification du programme

Afin d'enregistrer les résultats dans un fichier CSV, les classes ont été implémentées comme expliqué dans l'exemple mis à notre disposition.

Un point particulier à relever concerne la classe **BenchmarkExperimentData** destinée à contenir toutes les informations d'un test. Ces données sont stockée sous forme d'attributs (String et Long) qui sont ensuite transmis à l'aide d'une liste à l'objet CsvSerializer. Celui-ci n'a ensuite plus qu'à traiter chaque élément à la suite afin de créer une chaîne une chaine de caractères dans l'ordre souhaité à destination du fichier CSV.

Dans le programme principal, le **recorder** est un attribut static qui va récolter les données des tests au fûr et à mesure qu'ils sont effectués. L'initialisation de cet objet se fait directement dans la méthode main. Il lui est transmis le chemin vers le fichier de sortie, ainsi que le serializer à utiliser (ici CsvSerializer). Un premier record est ensuite enregistré sous forme d'un objet IData anonyme qui va simplement redéfinir la méthode getValues() pour retourner les en-têtes du fichier CSV. En fin de programme, le recorder est fermé ce qui entraine, dans le cas d'un FileRecorder, la fermeture du fichier.

Le resultat des tests est transmis en deux endroits au recorder : à la fin des méthodes produceTestData() et consumeTestData(). Ceci est fait en créer une nouvelle instance de BenchmarkExperimentData contenant les données souhaitées et en transmettant cet objet au recorder au travers de la méthode record().
La méthode consumeDataFromStream() a également été modifiée afin de retourner le nombre de Bytes lus, pour que ce dernier soit inclus dans le fichier CSV.