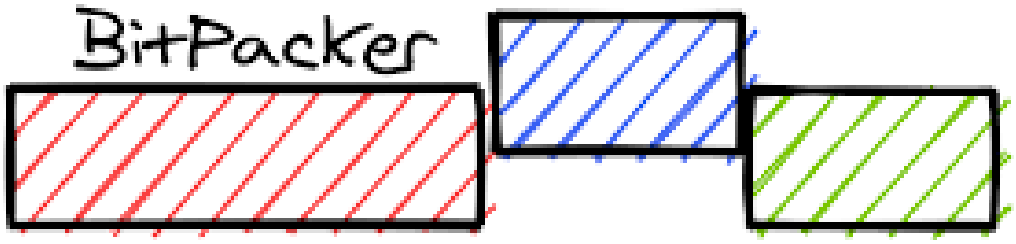
Rapport de Projet
Compression d'entiers Bit Packing



Par Baptiste Giacchero

Enseignant : Jean-Charles Régin le 02/11/2025

Sommaire

1. Introduction

Ce projet s'inscrit dans le cadre du module de génie logiciel 2025. L'objectif était d'étudier différents modes de transmission d'entiers à travers une approche de compression binaire, appelée Bit Packing. Le principe consiste à réduire la taille mémoire nécessaire à la transmission d'un tableau d'entiers, tout en conservant un accès direct à chaque élément après compression.

Trois versions distinctes de l'algorithme ont été implémentées en Java afin d'évaluer leur efficacité et leurs performances.

2. Présentation de la solution

La solution repose sur une architecture modulaire composée de trois classes principales en Java implémentant l'interface BitPacker :

- CrossBitPacker : compression autorisant le chevauchement des entiers sur deux mots consécutifs de 32 bits.
- NoCrossBitPacker : compression stricte sans chevauchement, garantissant qu'un entier compressé reste contenu dans un seul mot de 32 bits.
- OverflowBitPacker : version avancée intégrant une zone de débordement (overflow area) pour traiter efficacement les valeurs nécessitant plus de bits que la moyenne.

Une factory (BitPackerFactory) permet d'instancier dynamiquement le type de compresseur souhaité selon un paramètre unique, améliorant la flexibilité et la maintenabilité du code.

Une classe de benchmark est utilisée afin de mesurer équitablement tout les algorithmes de chaque classe afin d'en déterminer le meilleur.

3. Difficultés d'implémentation

Au départ, j'ai tenté de gérer la compression sans recourir aux opérateurs de décalage, car leur fonctionnement me paraissait complexe. Cependant, je me suis rapidement rendu compte qu'ils simplifiaient considérablement le code et rendaient la gestion des bits beaucoup plus claire.

L'un des principaux défis a été la gestion du chevauchement binaire, nécessitant un calcul précis des positions de bits pour éviter les erreurs de décalage.

Il a également fallu déterminer le nombre optimal de bits afin de trouver un équilibre entre la précision et la compacité du stockage. La mise en place de la zone de débordement a représenté une difficulté supplémentaire, car elle impliquait une logique complexe pour indexer les valeurs hors plage sans perdre l'accès direct aux données.

4. Méthodes utilitaires de manipulation binaire

Ces deux méthodes statiques constituent les fondations de toutes les opérations de compression dans les différentes classes du projet.
Elles sont regroupées dans la classe principale afin d'être réutilisées par les implémentations CrossBitPacker, NoCrossBitPacker et OverflowBitPacker.

4.0.1 Méthode bitsNeeded(int v)

Cette fonction calcule le nombre minimal de bits nécessaires pour représenter un entier positif v en binaire.
Elle repose sur la méthode native Integer.numberOfLeadingZeros(v) qui renvoie le nombre de zéros précédant le premier bit à 1 dans la représentation binaire de l'entier.

```
static int bitsNeeded(int v) {
    if (v <= 0) throw new IllegalArgumentException("Seulement les entiers positifs sont supportés");
    return 32 - Integer.numberOfLeadingZeros(v);
}
```

Ainsi, pour un entier donné, bitsNeeded() renvoie la longueur binaire exacte (par exemple, bitsNeeded(8) renvoie 4).
Cette méthode garantit que la compression alloue uniquement le nombre de bits réellement nécessaires, optimisant la taille du flux binaire.

4.0.2 Méthode maskLow(int k)

Cette fonction génère un masque binaire dont les k bits de poids faible sont égaux à 1. Elle est utilisée pour isoler ou manipuler une portion précise de bits lors des opérations de décalage ou d'écriture dans un flux compressé.

```
static int maskLow(int k) {
    if (k >= 32) return ~0;
    return (1 << k) - 1;
}
```

-Si k est inférieur à 32, la fonction renvoie (1 << k) - 1, soit un masque de k bits à 1.

-Si k vaut 32 ou plus, elle renvoie ~0, c'est-à-dire tous les bits à 1 sur 32 positions.

Ce masque est indispensable pour assurer la cohérence du découpage binaire dans les opérations de compression et de décompression.

4.1 La classe CrossBitPacker

La classe CrossBitPacker implémente une version de la compression Bit Packing dans laquelle les entiers compressés peuvent être répartis sur deux mots de 32 bits consécutifs.
Cette approche maximise l'utilisation de l'espace mémoire disponible et permet d'obtenir une compaction binaire optimale, au prix d'une légère complexité dans la gestion des bits.

4.1.1 Objectif

L'objectif de CrossBitPacker est de réduire la taille d'un tableau d'entiers en regroupant leurs représentations binaires dans un flux compact de bits.
Contrairement à la version sans chevauchement, cette implémentation autorise une valeur à déborder sur le mot suivant lorsque les bits restants du mot courant ne suffisent plus.
Ainsi, aucune perte d'espace n'est observée entre les entiers successifs.

4.1.2 Structure et variables principales

Variable	Description
n	nombre total d'entiers à compresser
k	nombre de bits utilisés par valeur
compresse[]	tableau d'entiers contenant les bits compressés

Le paramètre k est déterminé dynamiquement à partir du plus grand entier du tableau d'entrée, grâce à la fonction bitsNeeded().
Si k dépasse 32, il est borné à cette valeur, garantissant que chaque valeur compressée tienne au maximum sur deux entiers de 32 bits.

4.1.3 Méthode compress(int[] tab)

Cette méthode constitue le cœur de la classe.
Elle commence par calculer la taille maximale (max) et en déduit le nombre de bits nécessaires (k).
La taille totale en bits est ensuite donnée par :

totalBits = n×k

Le tableau compresse[] est dimensionné pour contenir cette séquence binaire complète :

tailleSortie = (totalBits+31)/32

Pour chaque entier du tableau :

1. On calcule sa position bit à bit dans le flux compressé (bitpos = i * k), puis le mot (word) et le décalage (offset) correspondants.

2. Si le champ compressé tient dans un seul mot de 32 bits, il est inséré directement par décalage gauche (val << offset).
3. Si le champ déborde sur le mot suivant, on le divise en deux parties :
    - poidsFaible (bits stockés dans le mot courant)
    - poidsFort (bits stockés dans le mot suivant)

Le code suivant illustre cette logique :

```
if (offset + k <= 32) {
   compresse[word] |= (val << offset);
} else {
   int nbrbitsfaible = 32 - offset;
   int lowMask = maskLow(nbrbitsfaible);
   int poidsFaible = val & lowMask;
   int poidsFort = val >>> nbrbitsfaible;
   compresse[word] |= (poidsFaible << offset);
   compresse[word + 1] |= poidsFort;
}
```

Grâce à ce procédé, la compression exploite au maximum chaque bit disponible sans espace perdu.

4.1.4 Méthode decompress(int[] tab)

La décompression repose sur la même logique de positionnement bit à bit.
Pour chaque élément à reconstruire, la méthode appelle get(i), qui lit les bits correspondants depuis le flux compresse[].
Le résultat est ensuite replacé dans le tableau de sortie.

Ce choix d'implémentation assure la symétrie parfaite entre compression et décompression, et facilite la vérification de correction (Arrays.equals()).

4.1.5 Méthode get(int i)

La fonction get() permet d'accéder directement à la *i*-ème valeur compressée sans décompresser le tableau entier.
Elle identifie la position du champ correspondant (bitpos, word, offset), puis :

- extrait la valeur directement si elle est contenue dans un seul mot,
- ou recompose la valeur à partir de deux mots consécutifs si elle est divisée.

```
if (offset + k <= 32) {
```

```
    int w = compresse[word];
    return (w >>> offset) & maskLow(k);
} else {
    int nbrbitsfaible = 32 - offset;
    int poidsFaible = (compresse[word] >>> offset) & maskLow(nbrbitsfaible);
    int poidsFort = compresse[word + 1] & maskLow(k - nbrbitsfaible);
    return (poidsFort << nbrbitsfaible) | poidsFaible;
}
```

Cette méthode conserve un temps d'accès constant O(1), ce qui est crucial pour des applications de transmission ou de traitement en flux.

4.2 La classe NoCrossBitPacker

La classe NoCrossBitPacker implémente une version simplifiée du Bit Packing. Contrairement à CrossBitPacker, elle ne permet pas le chevauchement des bits entre deux mots de 32 bits.
Chaque mot binaire contient un nombre fixe de valeurs compressées, ce qui facilite la lecture et la maintenance du code, au prix d'un léger gaspillage d'espace dans certains cas.

4.2.1 Objectif

L'objectif principal de cette implémentation est de **simplifier la logique de compression** tout en conservant un bon taux de compaction.
En divisant l'espace de 32 bits en plusieurs segments de taille fixe k, on évite d'avoir à gérer les cas complexes où une valeur déborde sur deux mots consécutifs.

Cette approche garantit une **structure régulière** du flux compressé, plus simple à manipuler lors de la décompression et des accès directs.

4.2.2 Structure et variables principales

Variable	Description
n	nombre total d'entiers à compresser
k	nombre de bits utilisés par valeur
compresse[]	tableau d'entiers représentant les données compressées
taillecompressée	nombre de valeurs stockées dans un mot de 32 bits

La variable taillecompressée joue un rôle clé : elle indique combien de valeurs peuvent être contenues dans un seul entier de 32 bits selon la taille k.
Ainsi, la mémoire est découpée en blocs homogènes de même taille.

4.2.3 Méthode compress(int[] tab)

La méthode compress() effectue les étapes suivantes :

1. **Calcul du nombre de bits nécessaires**
   On parcourt le tableau d'entrée pour trouver la valeur maximale (max) et déterminer k = bitsNeeded(max).
   Si k > 32, il est borné à 32.

2. **Détermination du nombre de valeurs par mot**
   Le nombre de valeurs stockables par mot est calculé comme :

   taillecompressée = 32/k

   Si ce résultat vaut zéro, il est corrigé à 1.

3. **Allocation du tableau compressé**
   La longueur de sortie est donnée par :

   outLen = (n+taillecompressée−1)/taillecompressée

   ce qui garantit que toutes les valeurs trouveront leur place.

4. **Insertion des valeurs compressées**
   Chaque valeur est placée dans le mot correspondant, à la position déterminée par son indice :

   - word = i / taillecompressée
   - slot = i % taillecompressée
   - offset = slot * k

   Le code d'écriture est le suivant :

   int val = tab[i] & maskLow(k);

   compresse[word] |= (val << offset);

Chaque mot contient ainsi un nombre fixe de valeurs successives, toutes alignées sans chevauchement.

4.2.4 Méthode decompress(int[] tab)

La méthode decompress() reconstruit le tableau d'origine en appelant get(i) pour chaque indice i.
La lecture est très simple puisque les valeurs sont stockées de manière alignée et régulière dans chaque mot.
Il n'y a aucune gestion de débordement à effectuer, ce qui rend la décompression extrêmement rapide.

4.2.5 Méthode get(int i)

Cette méthode permet d'accéder directement à la i-ème valeur sans décompresser l'ensemble du tableau.
L'algorithme localise le mot correspondant et extrait la valeur en appliquant un décalage à droite (>>> offset) et un masque binaire (maskLow(k)).

```
int word = i / valuesPerWord;

int slot = i % valuesPerWord;

int offset = slot * k;

int w = compresse[word];

return (w >>> offset) & maskLow(k);
```

Cette approche garantit un accès constant (O(1)), idéal pour des opérations répétées ou aléatoires sur des données compressées.

4.2.6 Synthèse rapide

NoCrossBitPacker est une version plus simple et plus régulière du Bit Packing.
Elle offre un bon compromis entre efficacité et lisibilité du code, tout en maintenant un accès direct rapide aux données compressées.
Son absence de chevauchement facilite grandement la maintenance et la vérification de la justesse des opérations.

4.3 La classe OverflowBitPacker

La classe OverflowBitPacker représente la version la plus évoluée du mécanisme de compression d'entiers mis en œuvre dans ce projet.
Elle a pour objectif d'optimiser le taux de compression en s'adaptant automatiquement à la répartition des valeurs dans le tableau d'entrée, tout en garantissant un accès direct à chaque élément après compression.

Contrairement aux approches classiques (CrossBitPacker et NoCrossBitPacker), qui utilisent un même nombre de bits k pour tous les éléments, cette version introduit une zone de débordement destinée à stocker les valeurs exceptionnelles nécessitant davantage de bits que la majorité.

4.3.1 Objectif et principe général

L'idée de base repose sur un constat simple : dans un grand tableau d'entiers, la majorité des valeurs peut être représentée sur un petit nombre de bits, tandis que quelques valeurs isolées exigent davantage de bits.
Les méthodes classiques imposent de réserver k bits pour tous les éléments, où k correspond à la taille nécessaire pour représenter la valeur la plus grande.
Cela provoque un gaspillage d'espace binaire, puisque la majorité des entiers n'utilise pas toute cette capacité.

OverflowBitPacker remédie à ce problème en séparant les données en deux

ensembles :

- une zone principale contenant la majorité des valeurs, encodées sur un petit nombre de bits petitK ;
- une zone de débordement, qui stocke uniquement les valeurs trop grandes pour être représentées sur petitK bits.

Chaque champ compressé contient alors un bit de drapeau (flag) indiquant si la valeur est stockée directement ou bien dans la zone de débordement :

- flag = 0 → la valeur est stockée directement sur petitK bits ;
- flag = 1 → le champ contient un index vers la position de la valeur réelle dans la zone de débordement.

Ce mécanisme assure une compression fine et non destructive, tout en conservant la possibilité d'un accès direct à n'importe quel élément du tableau compressé.

4.3.2 Structure interne et variables principales

La classe repose sur plusieurs attributs essentiels :

Variable	Description
taille	Nombre total d'entiers à compresser
petitK	Nombre de bits utilisés pour la majorité des valeurs
bitsChamp	Taille totale d'un champ compressé (1 bit de drapeau + contenu)
nbDebordement	Nombre de valeurs déportées dans la zone de débordement
valeursDebordement[]	Tableau contenant les valeurs réelles des débordements
champsPackes[]	Tableau d'entiers représentant la séquence binaire compressée

4.3.3 Calcul automatique du paramètre optimal petitK

Le choix du nombre de bits petitK constitue l'élément central de l'algorithme. Celui-ci est déterminé automatiquement à partir d'une minimisation du coût total en bits :

$$totalBits = n \times bitsChamp + nbDebordement \times 32$$

où :

$$bitsChamp = 1 + \max(petitK, bitsNeeded(nbDebordement - 1))$$

Le premier terme représente le coût de la zone principale, et le second celui de la zone de débordement (chaque valeur de débordement occupant 32 bits).

L'algorithme parcourt toutes les valeurs candidates de petitK comprises entre 0 et le nombre de bits nécessaires à la plus grande valeur du tableau (bitsMax).
Pour chaque candidat, il calcule :

- le nombre de valeurs en débordement (nbDebordement),
- la taille totale du flux compressé (totalBits).

Le petitK minimisant ce total est retenu comme meilleure configuration.
Cette approche adaptative permet d'ajuster dynamiquement la compression à la distribution réelle des données.

4.3.4 Étapes de la compression

Le processus de compression dans OverflowBitPacker se déroule en plusieurs étapes bien définies :

1. **Analyse préliminaire**
   Pour chaque entier du tableau d'entrée, on calcule le nombre minimal de bits nécessaires (bitsnecessaires[]) à sa représentation.
   Cela permet de détecter les valeurs susceptibles de provoquer un débordement.
2. **Recherche du meilleur petitK**
   L'algorithme évalue toutes les valeurs candidates de petitK et conserve celle minimisant la taille totale du flux binaire (voir formule précédente).
3. **Création de la zone de débordement**
   Les valeurs ne pouvant être codées sur petitK bits sont transférées dans une liste valeursDebordement[].
   Chaque valeur de cette zone reçoit un indice unique permettant de la retrouver facilement lors de la décompression.
4. **Construction du flux compressé (champsPackes[])**
   Pour chaque élément du tableau :
   - Si la valeur tient sur petitK bits → on stocke directement la valeur avec flag = 0.
   - Sinon → on stocke l'indice de débordement avec flag = 1.
     L'ensemble est encodé dans des mots de 32 bits, en gérant le cas particulier des valeurs chevauchant deux mots consécutifs.

Ce mécanisme assure un équilibre optimal entre compacité et accessibilité.

4.3.5 Décompression des données

La méthode decompress(int[] tab) reconstitue le tableau original à partir des données compressées.
Pour chaque élément :

- On calcule la position du champ dans le flux (posBits = i * bitsChamp).
- On lit les bits correspondants, puis on extrait le drapeau (flag) et le contenu du champ.
- Si flag == 0, la valeur décompressée correspond directement au contenu.
- Si flag == 1, la valeur est récupérée dans la zone de débordement (valeursDebordement[contenu]).

Cette méthode garantit une décompression sans perte : les données reconstruites sont identiques aux données d'origine, comme confirmé par les tests utilisant Arrays.equals().

4.3.6 Accès direct à un élément compressé

Une des forces de cette approche est la méthode get(int i), qui permet un accès aléatoire direct à un élément du tableau sans décompresser la totalité des données.

Le fonctionnement est identique à celui de la décompression, mais limité à un seul indice i.
Grâce au calcul précis de la position bit à bit (posBits), il suffit de lire le champ correspondant et d'appliquer la même logique de drapeau (flag) pour obtenir instantanément la valeur originale.

Cette propriété est essentielle dans un contexte de transmission ou de traitement de données massives, où l'accès à un élément spécifique doit rester rapide et constant (complexité en O(1)).

4.3.7 Gestion des bits et robustesse

Le traitement bit à bit est un aspect particulièrement technique de cette classe.
Les champs compressés peuvent parfois déborder d'un mot de 32 bits.
Dans ce cas :

- La partie basse du champ est stockée dans le mot courant (poidsFaible),
- La partie haute dans le mot suivant (poidsFort).

Les opérations de masquage (maskLow(k)) et de décalage binaire (<< et >>>) permettent de gérer avec précision ce fractionnement sans perte de données ni chevauchement.
Cela garantit la cohérence du flux binaire quel que soit le nombre de bits utilisé.

5. Objectifs de la classe Bench

L'objectif principal de cette classe est de fournir un cadre de mesure des performances des différents compresseurs.

Les trois métriques évaluées sont :

1. Le temps de compression – mesure la rapidité avec laquelle un tableau d'entiers est transformé en représentation compacte.
2. Le temps d'accès (get) – mesure la capacité du système à restituer rapidement une valeur précise sans décompresser l'ensemble du tableau.
3. Le temps de décompression – mesure la vitesse de reconstruction du tableau d'origine à partir de la version compressée.

L'ensemble de ces mesures est effectué à l'aide d'un protocole rigoureux basé sur des exécutions répétées, une phase de *warm-up* et un chronométrage en nanosecondes.

5.1. La phase de warm-up : échauffement de la JVM

L'une des particularités du benchmarking en Java réside dans la présence du JIT (Just-In-Time compiler), composant de la JVM qui optimise le code à l'exécution. Lorsqu'un programme démarre, le code Java est interprété, puis progressivement compilé dynamiquement en instructions machine optimisées à mesure qu'il est exécuté plusieurs fois.
Ainsi, les premières exécutions sont souvent plus lentes que les suivantes, car la JVM n'a pas encore optimisé les portions de code les plus fréquemment appelées.

Pour éviter de fausser les mesures, la méthode runSimpleBench() commence donc par une phase de warm-up :

```
for (int w = 0; w < 20; w++) {
    packer.compress(data);
}
```

Cette étape d'échauffement consiste à exécuter la méthode compress() une vingtaine de fois avant les mesures officielles.
Cela permet au JIT d'optimiser la boucle et au cache processeur de se stabiliser, garantissant ainsi que les mesures suivantes reflètent la vitesse réelle du code optimisé.
Sans cette phase, les résultats seraient plus variables et non représentatifs des performances de production.

5.2. Méthode de mesure : la fonction chronoNs()

La classe Bench repose sur une fonction utilitaire simple mais robuste pour mesurer les durées d'exécution :

```
public static long chronoNs(Runnable r, int repeats) {
    long start = System.nanoTime();
    for (int i = 0; i < repeats; i++) r.run();
    long end = System.nanoTime();
```

```
    return end - start;
}
```

Cette méthode prend en paramètre une fonction (Runnable) et le nombre d'itérations à répéter.
L'usage de System.nanoTime() permet d'obtenir une précision fine, adaptée à des mesures sur des fragments de code de quelques microsecondes.
En répétant plusieurs fois la même action, on lisse les éventuelles variations dues au système d'exploitation ou à l'allocation mémoire, et on obtient une moyenne statistiquement plus fiable.

5.3. Mesure de la compression

La première partie du benchmark évalue la rapidité de la compression :

```
long tCompression = chronoNs(() -> packer.compress(data), iterations);

System.out.printf("compression : total %d ns, moyenne %.2f µs\n",
    tCompression, (tCompression / 1000.0) / iterations);
```

Chaque implémentation (Cross, NoCross, Overflow) est testée sur le même tableau data, avec le même nombre d'itérations.
Le résultat est affiché à la fois en temps total et en moyenne par itération, en microsecondes.
Cette partie mesure la complexité intrinsèque de l'algorithme de compression et permet d'identifier les approches les plus rapides selon le type de données.

5.4 Mesure des accès aléatoires – méthode get()

Une caractéristique clé du Bit Packing est la possibilité d'accéder directement à un élément compressé sans décompresser tout le tableau.
Le benchmark évalue donc la latence moyenne d'un grand nombre d'appels get() aléatoires :

```
long tGet = timeNs(() -> {
    int s = 0;
    for (int q = 0; q < 1000; q++) {
        int idx = rnd.nextInt(n);
        s += packer.get(idx);
    }
}, iterations);
```

Cette boucle réalise 1 000 accès aléatoires par itération.

Les résultats sont exprimés en millisecondes et donne la capacité de l'algorithme à

gérer des accès directs efficaces malgré la compression.

5.5 Mesure de la décompression

Enfin, la troisième phase mesure la vitesse de décompression complète du tableau :

```
long tDecompress = chronoNs(() -> {
    packer.decompress(dest);
}, iterations);
```

La méthode decompress() reconstruit le tableau d'origine à partir de la représentation compressée.
Les durées sont exprimées en microsecondes, ce qui permet de comparer la complexité de décompression entre les trois implémentations.
Dans la majorité des cas, NoCrossBitPacker est légèrement plus rapide grâce à sa structure simple, tandis que OverflowBitPacker présente un léger surcoût dû à la gestion des zones de débordement.

5.6 Seuil de rentabilité de la compression

Cette section vise à déterminer, pour chaque méthode de compression implémentée, la latence de transmission réseau minimale à partir de laquelle la compression devient avantageuse en temps total.
Le calcul repose sur la comparaison du temps total compressé et non compressé, en tenant compte à la fois du coût de traitement et de la réduction du volume transmis.

Pour chaque algorithme (CrossBitPacker, NoCrossBitPacker et OverflowBitPacker), les temps suivants sont mesurés à l'aide de la classe Bench :

- **Tcomp** : temps moyen de compression
- **Tdecomp** : temps moyen de décompression
- **r** : taux de compression (rapport entre taille compressée et taille initiale)
- **N** : nombre total de bits à transmettre ($N = n \times 32$)

La compression devient avantageuse lorsque :

$$T_{total\_compressé} = Tcomp + Tdecomp + t \times Bcomp$$
$$\leq$$
$$T_{total\_non\_compressé} = t \times N$$

Ce qui donne le seuil de rentabilité :

$$t \geq (Tcomp + Tdecomp) / ((1 - r) \times N)$$

Données expérimentales

Les mesures suivantes ont été obtenues pour n = 10 000 entiers (voir README.md):

CROSS
n = 10 000, k = 15
Tcomp = 275,92 µs
Tdecomp = 498,84 µs
Tget = 0,39 ms

NOCROSS
n = 10 000, k = 15
Tcomp = 89,24 µs
Tdecomp = 333,42 µs
Tget = 0,53 ms

OVERFLOW
n = 10 000, petitK = 6, bitsChamp = 7, nbDebordement = 20
Tcomp = 1 091,28 µs
Tdecomp = 734,06 µs
Tget = 0,14 ms

Taille non compressée :
N = 10 000 × 32 = 320 000 bits

Tailles compressées :

- CROSS / NOCROSS : Bcomp = 10 000 × 15 = 150 000 bits → r = 0,46875
- OVERFLOW : Bcomp = (10 000 × 7) + (20 × 32) = 70 640 bits → r = 0,22075

Calculs des seuils de rentabilité

CROSS
$t \geq ((275,92 + 498,84) \times 10^{-6}) / ((1 - 0,46875) \times 320\,000)$
$t \approx 4,50 \times 10^{-9}$ s/bit
→ Seuil de rentabilité ≈ 4,50 ns/bit
→ Débit équivalent ≈ 222 Mbit/s

NOCROSS
$t \geq ((89,24 + 333,42) \times 10^{-6}) / ((1 - 0,46875) \times 320\,000)$
$t \approx 2,56 \times 10^{-9}$ s/bit
→ Seuil de rentabilité ≈ 2,56 ns/bit
→ Débit équivalent ≈ 390 Mbit/s

OVERFLOW

$t \geq ((1\,091{,}28 + 734{,}06) \times 10^{-6}) / ((1 - 0{,}22075) \times 320\,000)$

$t \approx 7{,}31 \times 10^{-9}$ s/bit

→ Seuil de rentabilité ≈ 7,31 ns/bit

→ Débit équivalent ≈ 137 Mbit/s

Interprétation

Les résultats montrent que :

- NoCrossBitPacker présente le meilleur compromis : il devient rentable dès que le débit réseau descend sous environ 390 Mbit/s, ce qui couvre la majorité des réseaux courants.
- CrossBitPacker reste intéressant pour des débits inférieurs à 220 Mbit/s, notamment sur des réseaux distants à plus forte latence.
- OverflowBitPacker, plus coûteux en traitement, devient rentable seulement pour des connexions plus lentes (inférieures à 140 Mbit/s), mais offre la meilleure réduction de taille.

Ainsi, selon le contexte d'utilisation (vitesse du lien, volume transmis et fréquence des accès), chaque algorithme peut présenter un intérêt différent.
Dès qu'une certaine latence réseau est présente, la compression apporte un gain net en temps total tout en réduisant la bande passante consommée.