Optimisation des corrélations PAON-4

Hadrien Grasland

2020-12-14

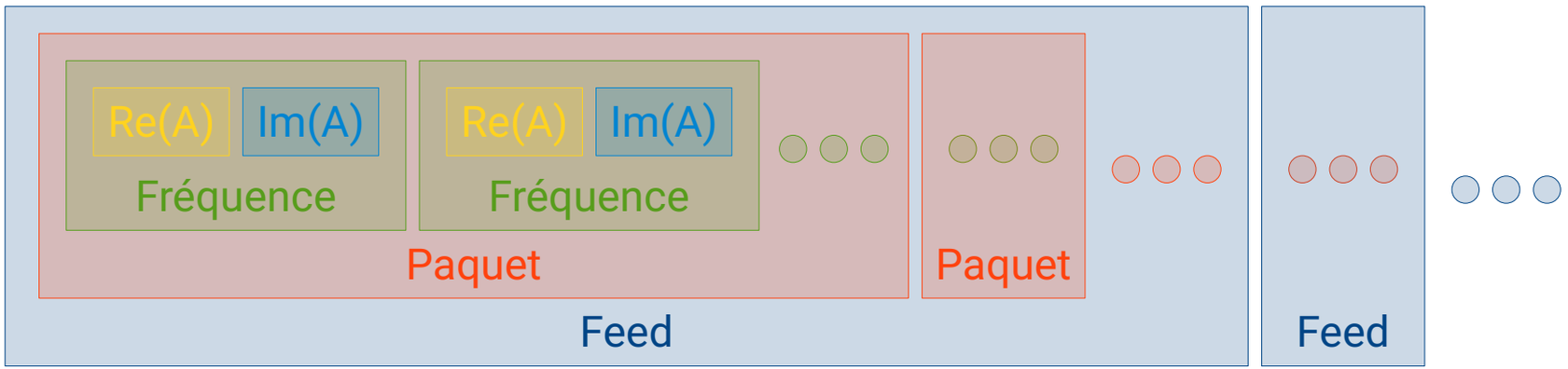
Introduction

- Il est plus difficile d'optimiser du code GPU que du code CPU
  - Moins d'outils d'analyse, plus de « pièges » matériels à gérer
- Les optimisations CPU bénéficient généralement au GPU
  - Architectures matérielles plus similaires qu'il n'y paraît
- Le code CPU était clairement sous-optimal
  - 500 MFLOPS c'est très peu pour un processeur à >2 GHz…
- J'ai donc commencé par travailler sur le code CPU
  - En prenant le tstthrust.cc d'Olivier comme point de départ

Situation initiale

Format données d'entrée (original)

- On part des amplitudes complexes des FFT des signaux
  - Le format actuel est une liste de n_i feeds i…
  - …pour chaque feed, on a une liste de n_p paquets p
  - …pour chaque paquet, on a une liste de n_f fréquences f
  - …et pour chaque fréquence, on a l'amplitude complexe en coordonnées cartésiennes (Re, Im), notons-la A(i, p, f)

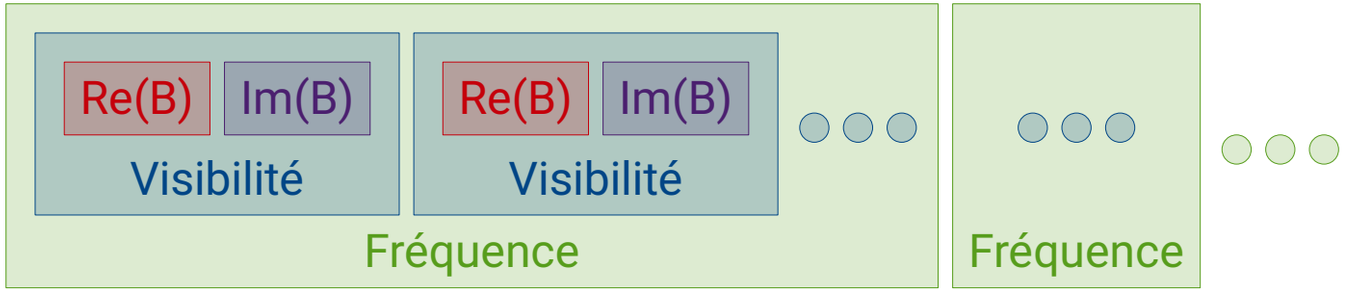


Calcul effectué (original)

- Pour chaque paquet p
  - Pour chaque fréquence f
    - Pour chaque visibilité (paire de feeds) i, j
      - On accumule B(f, (i, j)) += A(i, p, f) * conj(A(j, p, f))
      - Somme des TFs de la fonction de cross-correlation
- Notez qu'on intègre sur les paquets
  - Cela tend à réduire les volumes de sortie (fait pour!)

Format données de sortie (original)

- Le format actuel dans le code est une liste de fréquences f…
  - …pour chaque fréquence f, on a une liste de visibilités (i, j)
  - …pour chaque visibilité, on a l'amplitude complexe sommée B(f, (i, j)), toujours en coordonnées cartésiennes



- Problème : D'après Olivier, on veut B((i, j), f). Qui a raison ?

Facteur matériel limitant ?

- A chaque itération, on lit 6 flottants* :
  - Valeurs d'entrée de x = A(i, p, f) et y = A(j, p, f)
  - Valeur originale de z = B(f, (i, j))
- On écrit 2 flottants : nouvelle valeur de z
- On effectue 8 calculs* :
  - $Re(z) \leftarrow Re(z) + Re(x) * Re(y) - Im(x) * Im(y)$
  - $Im(z) \leftarrow Im(z) - Re(x) * Im(y) + Re(y) * Im(x)$
- Intensité arithmétique 8/8 = 1 $\longrightarrow$ **Limité par les accès mémoire**

* J'ignore registres et FMA pour simplifier, mais même avec eux, le résultat est le même.

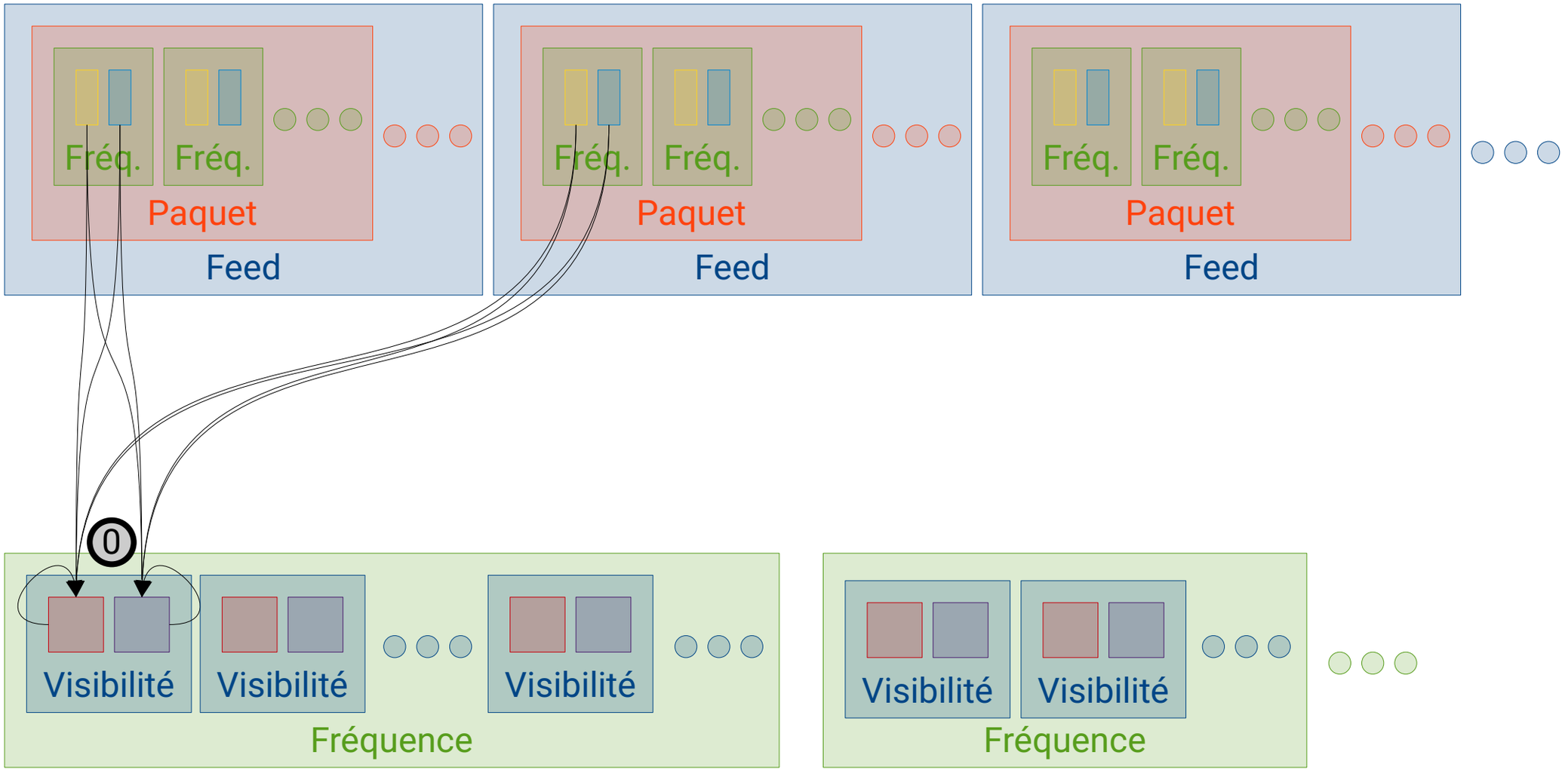
Un mot sur les caches (1)

- La RAM est très lente par rapport aux capacités de calcul
  - Core i3-3220 : 6,4 Gf32/s vs 106 GFLOPS f32
  - Xeon 4210 : 29 Gf32/s vs 352 GFLOPS f32
  - Quadro P4000 : 72 Gf32/s vs 5,3 TFLOPS f32
- Pour échapper à ça, les processeurs utilisent des caches
  - Mémoires plus rapides (SRAM) et plus basse latence
  - **Capacités très faibles** (32 ko de cache L1d sur CPUs x86)

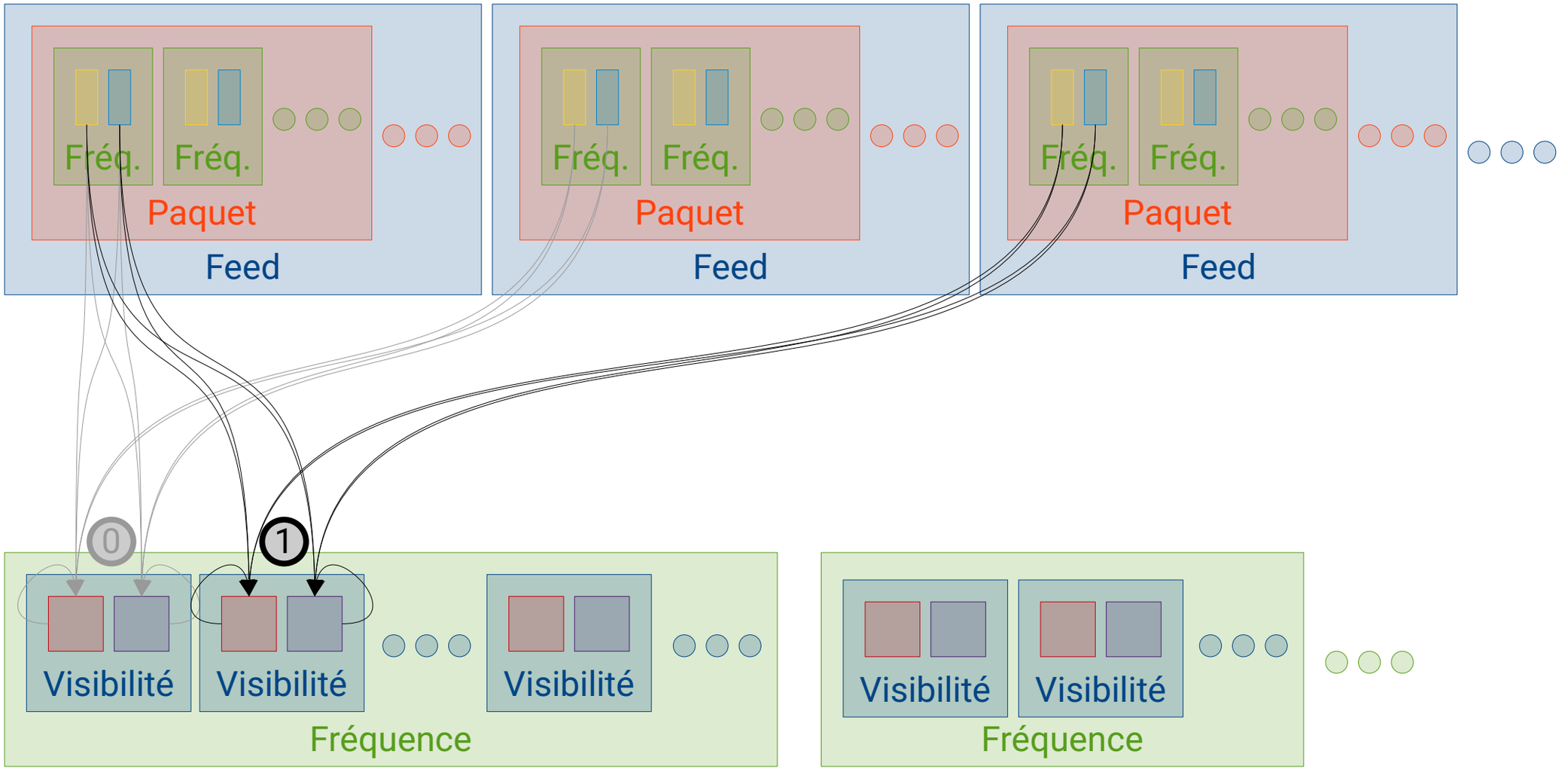
Un mot sur les caches (2)

- Les caches fonctionnent selon 2 principes :
  - Chargement par lot (64 octets = 16 f32 sur CPUs x86)
  - Logique d'élimination « least recently used » (LRU)
- Un code limité par la mémoire doit donc suivre 2 principes :
  - Localité spatiale : Utiliser des données voisines en RAM
  - Localité temporelle : Réutiliser rapidement chaque donnée
  - On parle de **localité de cache**

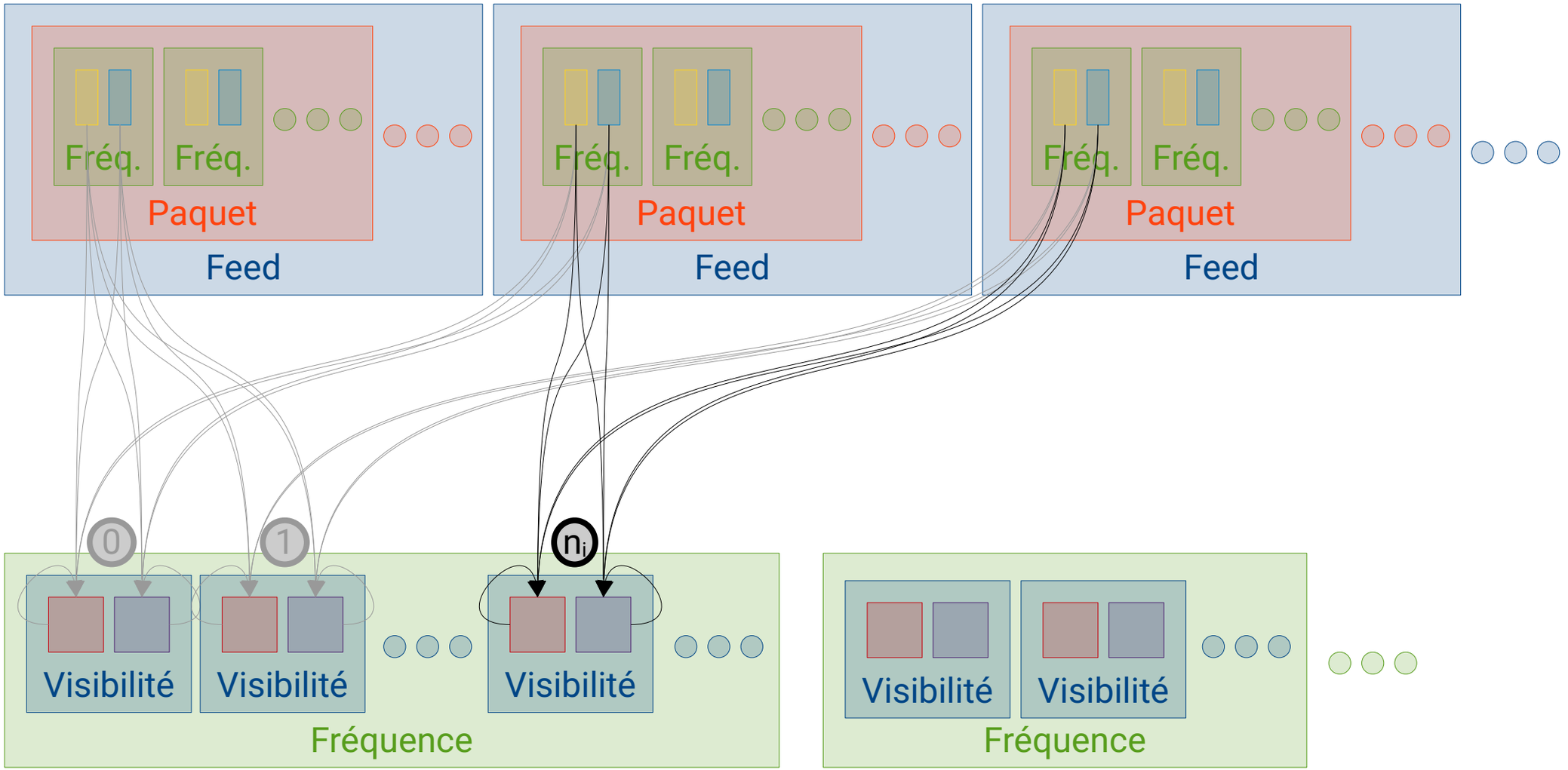
Flux de données (original)



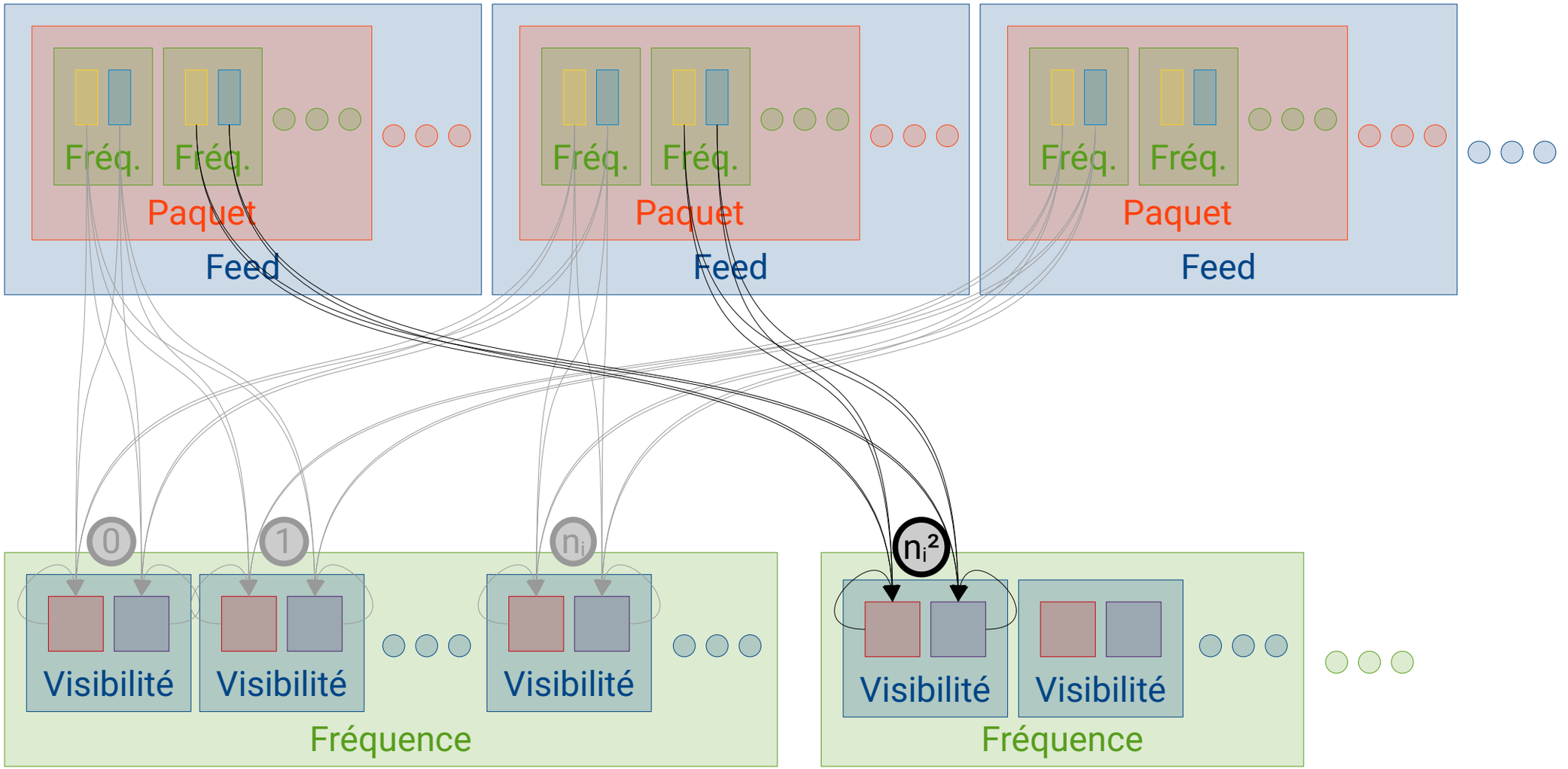
Flux de données (original)



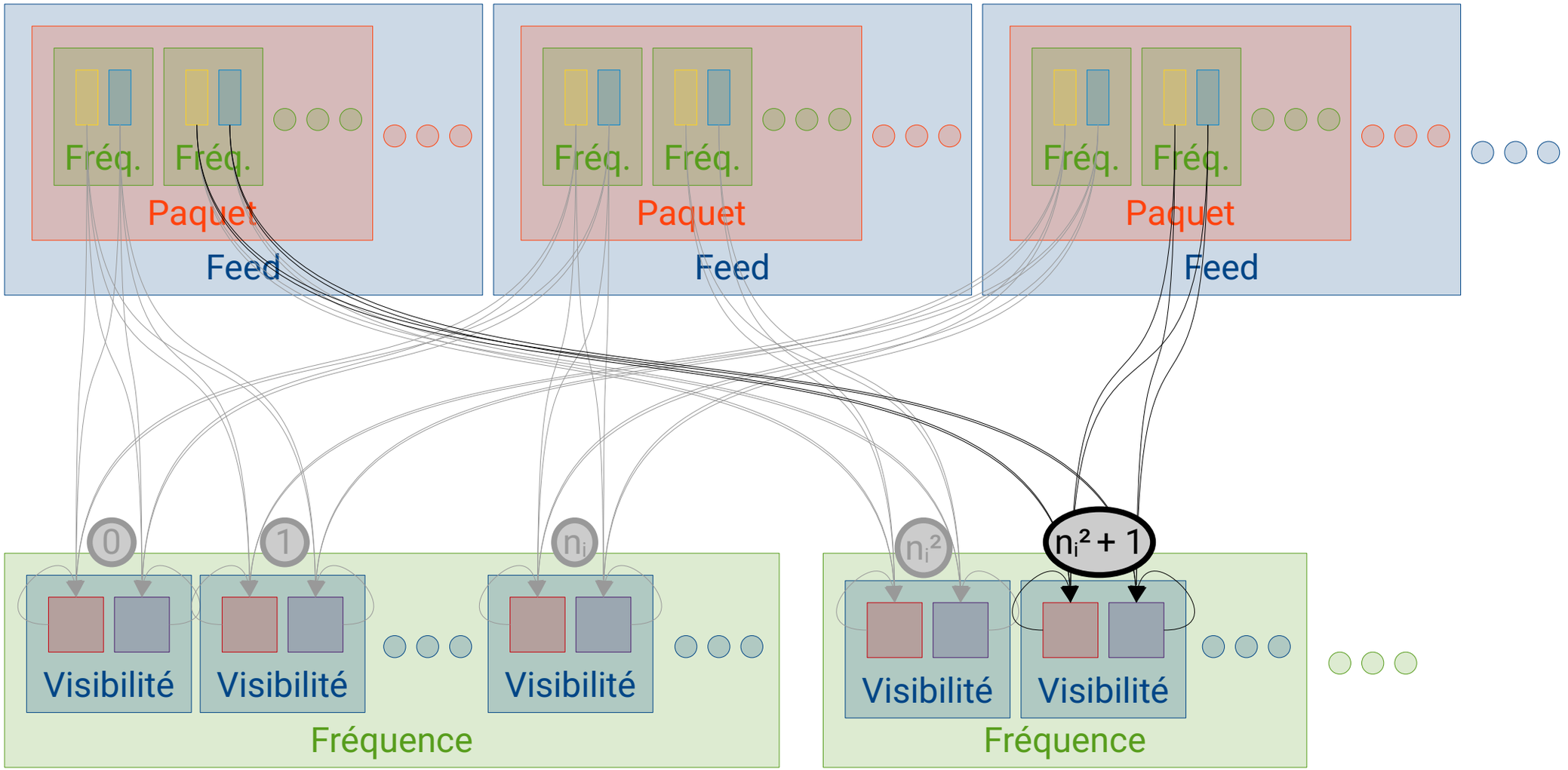
Flux de données (original)



Flux de données (original)



Flux de données (original)



Bilan sur localité de cache

- Spatiale (accès à des données contigües en mémoire)
  - Mauvaise pour flux d'entrée (accès par fréquence espacés)
  - Parfaite pour l'accumulateur de sortie
- Temporelle (réutilisation rapide des données en cache)
  - 1er flux d'entrée : OK (« oublié » entre itérations fréquence)
  - 2e flux d'entrée : Nulle (« oublié » avant accès suivant)
  - Accumulateur : Nulle (« oublié » avant paquet suivant)
- Ces problèmes doivent être résolus avant d'optimiser le calcul !

Autres soucis du code original

- Différences thrust vs tableau C ?
  - Test d'une variable « verbose » en fond de boucle « C »
    - Paramètre CLI ⟶ Non connue à la compilation
    - Si vraie, alors appel std::cout ⟶ Effets de bord
    - Conséquence : Le compilateur optimise moins bien
  - Accumulateur « nvisi » non initialisé dans la version « C »
    - Conséquence : Résultats faux + optimisations injustes
- Mauvaise vectorisation
  - Nous verrons plus loin pourquoi et comment faire mieux

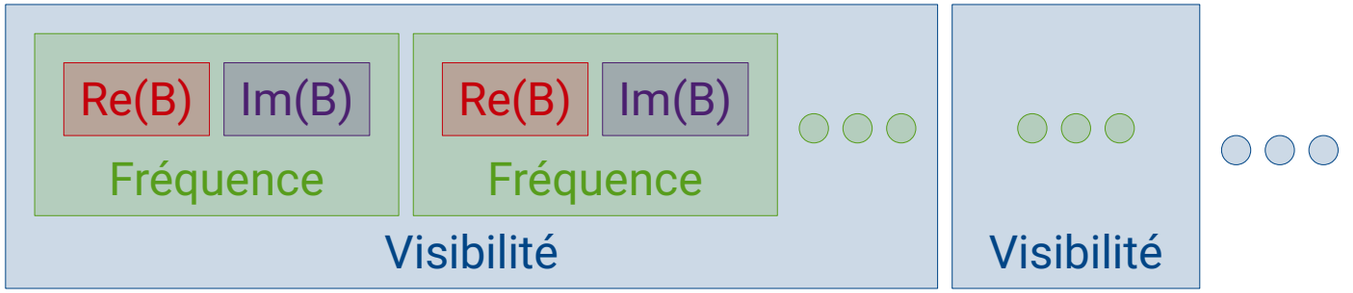
Optimisations mémoire

Quel point de départ ?

- D'abord, rétrécir l'accumulateur
  - B(f, (j, i)) = conj(B(f, (i, j))) → Information redondante !
  - On ne calculera donc que B(f, (i, j)) pour $i \leq j$
  - Si besoin, on calculera les conjugués des sommes à la fin
- Ensuite, pour PAON-4, optimiser en priorité l'accès aux entrées
  - Entrées plus nombreuses si $n_p > n_v = n_i * (n_i + 1) / 2$
  - Pour $n_p = 1024$ et $n_i = 8$, elle sont donc **28x** plus nombreuses

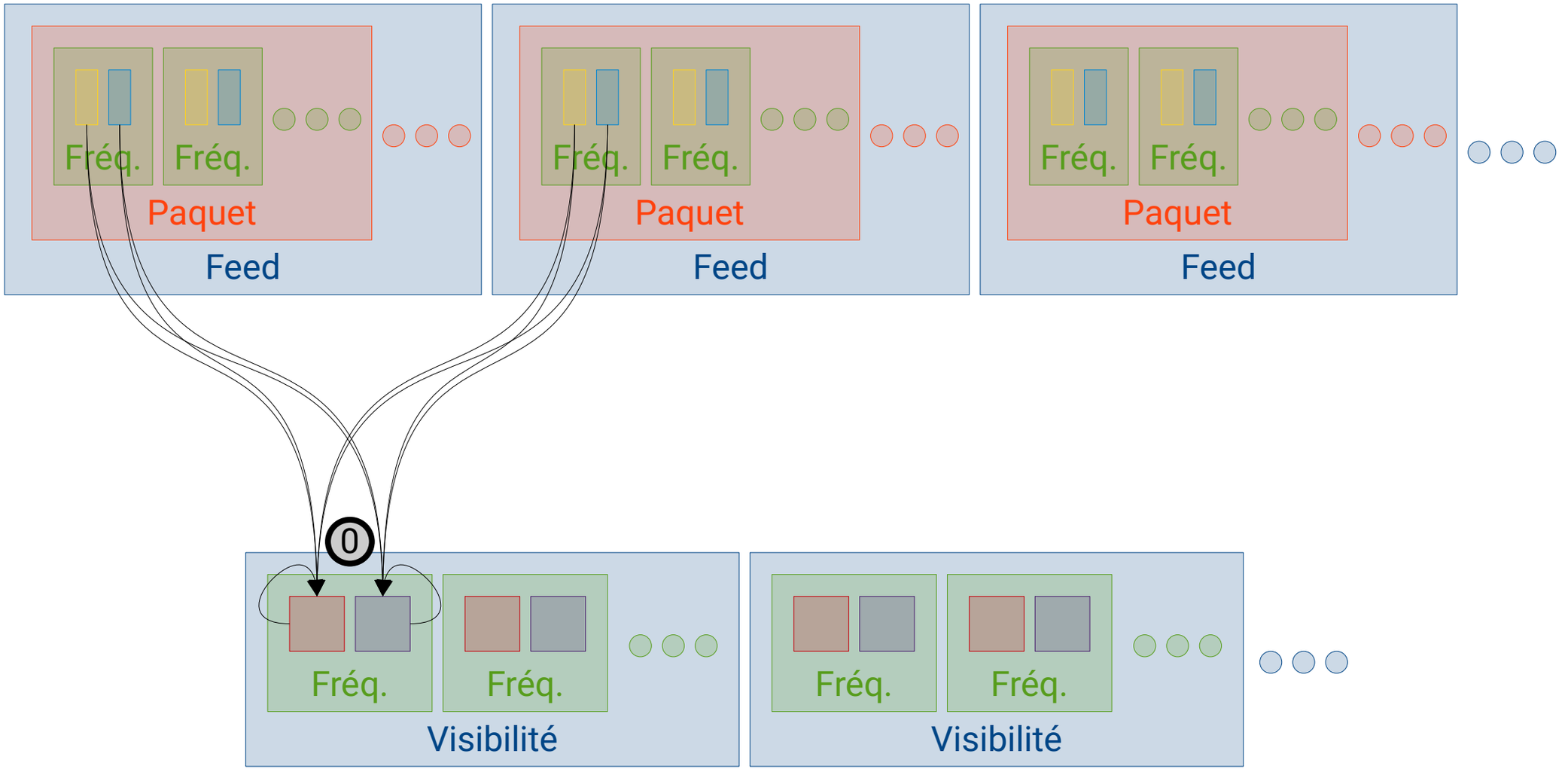
Optimiser la localité spatiale

- En entrée, localité spatiale pour l'accès à **fréqs. consécutives**
  - La boucle la plus interne doit donc être sur les fréquences
- Mais cet ordre n'est pas favorable pour l'accumulateur actuel !
  - On utilisera donc un accumulateur transposé…
  - …quitte à re-transposer à la fin si le format de sortie l'exige*

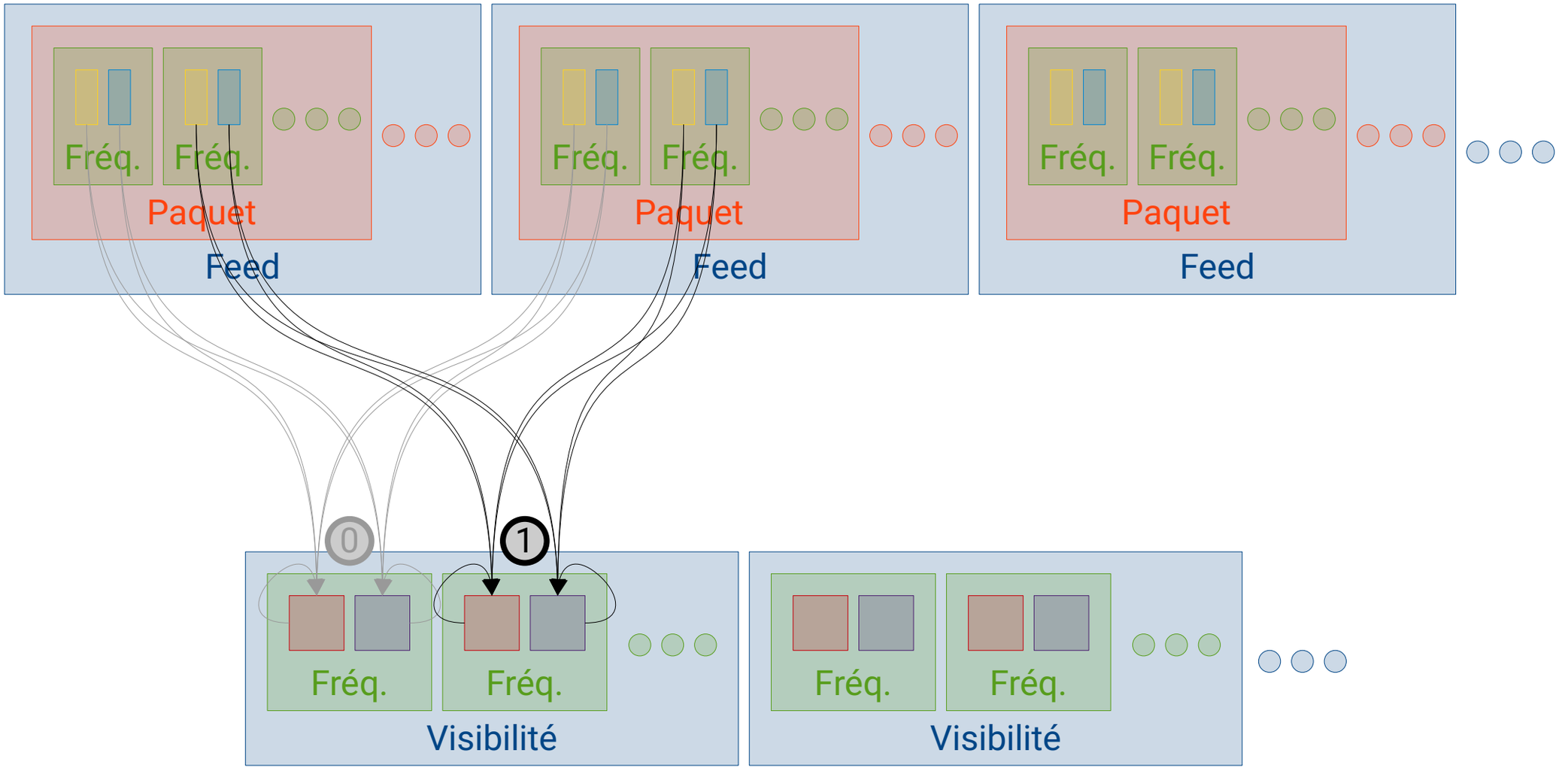


* Coût négligeable si l'accumulateur est beaucoup plus petit que les flux d'entrée, comme ici.

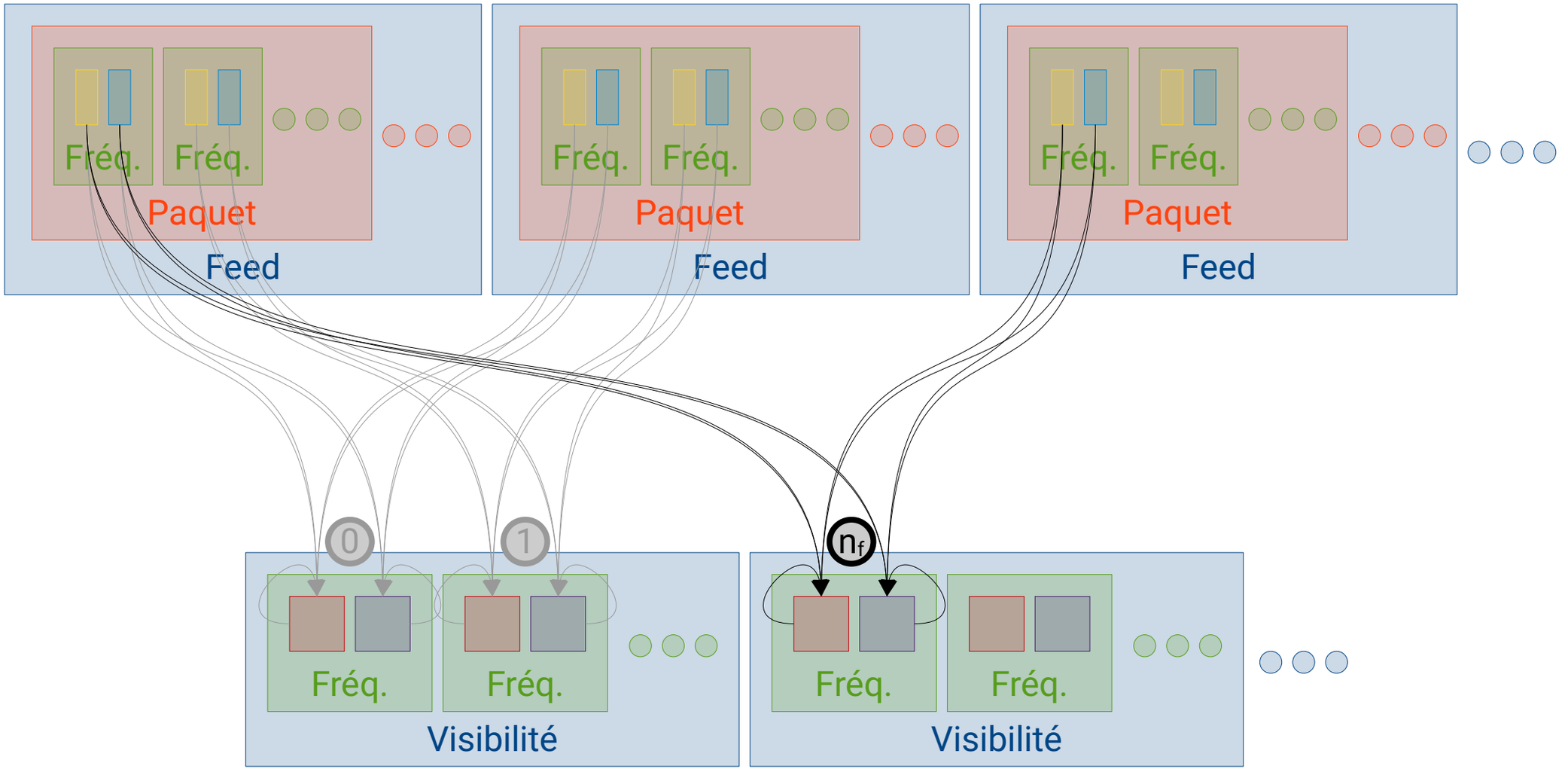
Flux de données (avec localité spatiale)



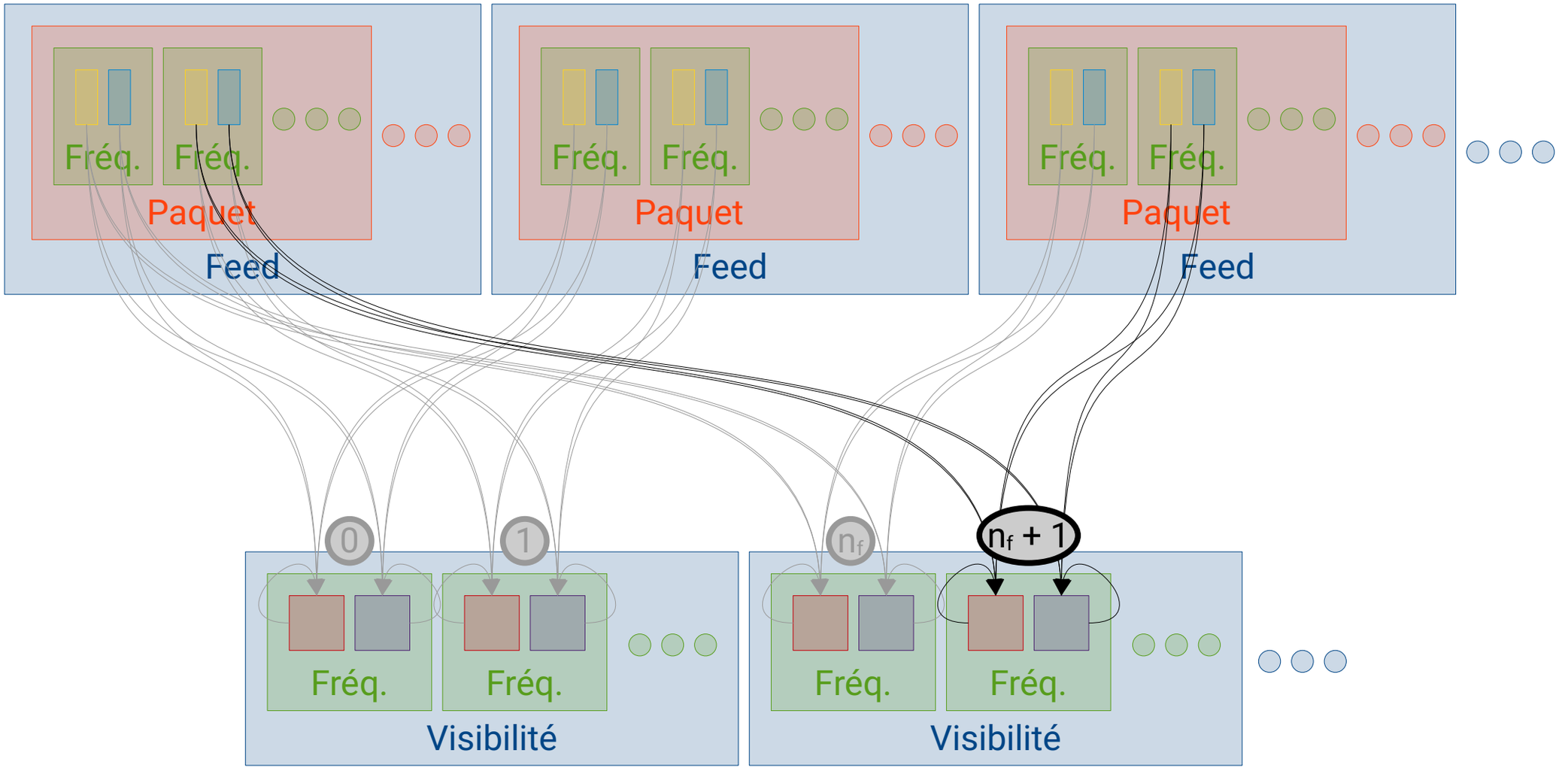
Flux de données (avec localité spatiale)



Flux de données (avec localité spatiale)



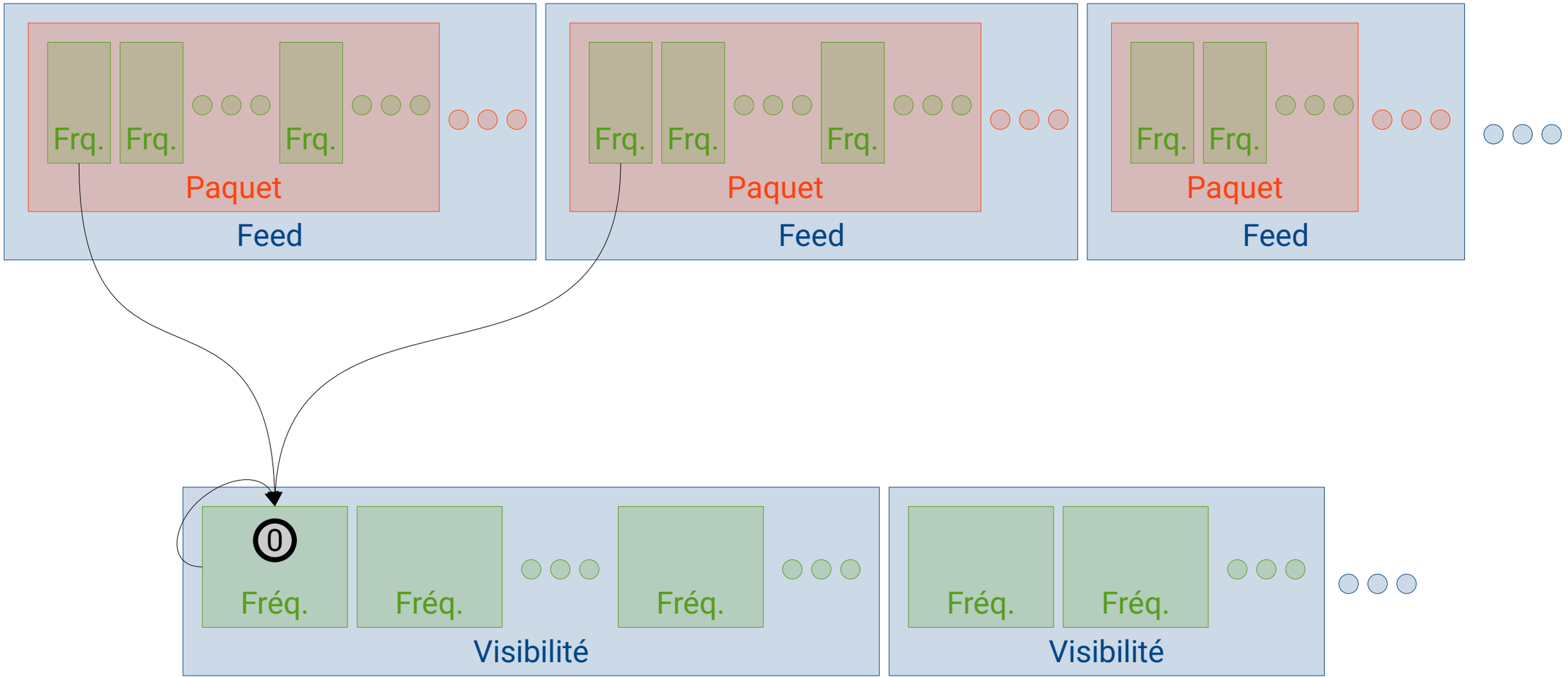
Flux de données (avec localité spatiale)



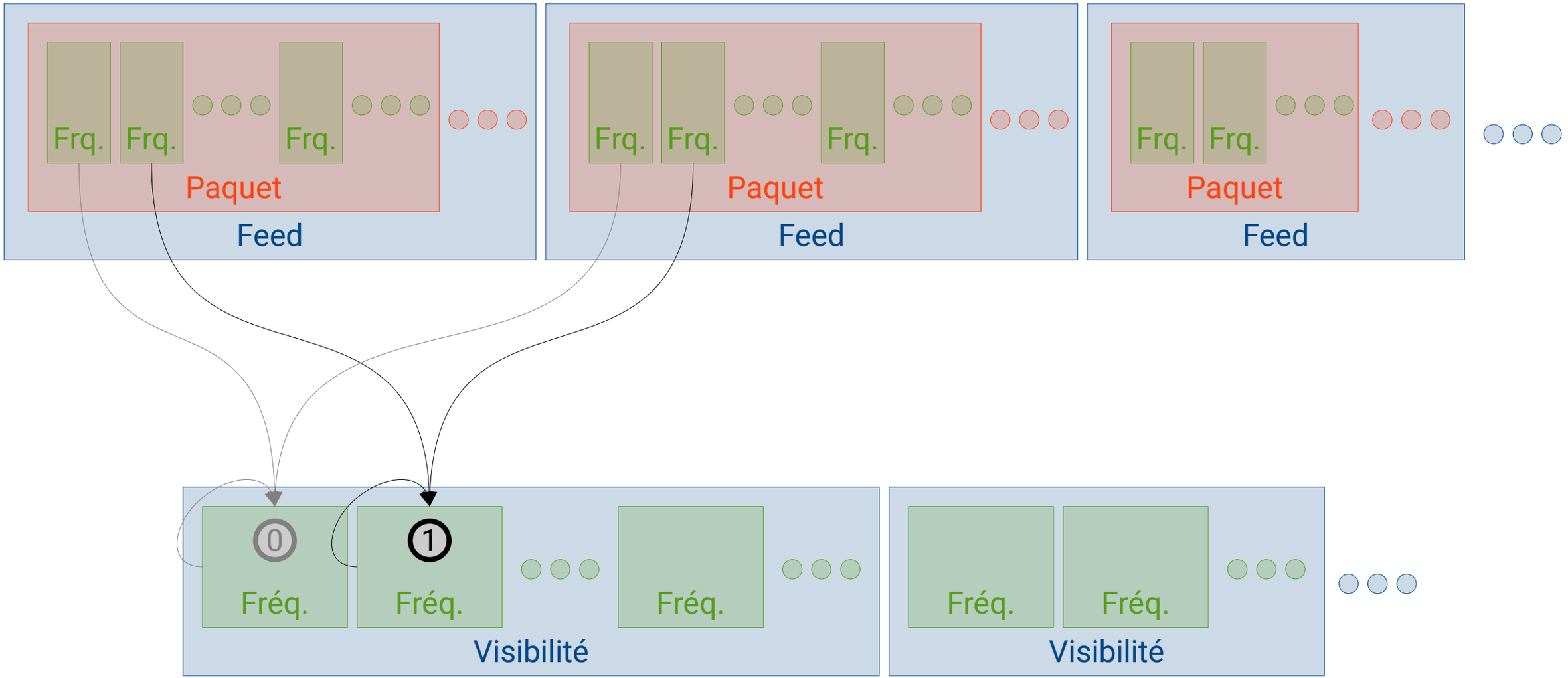
Optimiser la localité temporelle

- La localité spatiale est maintenant convenable
  - Accès à des données contiguës, en entrée et en sortie
- Mais la localité temporelle est encore plus mauvaise qu'avant…
  - 1ère fréquence du 1er feed « oubliée » entre 2 visibilités
- Pour éviter ça, on travaille par blocs de n_{bf} fréquences
  - Boucles imbriquées : blocFreq{ visibilités{ freq@blocFreq }}}
  - Taille de bloc choisie pour tenir dans le cache L1 (32 ko)

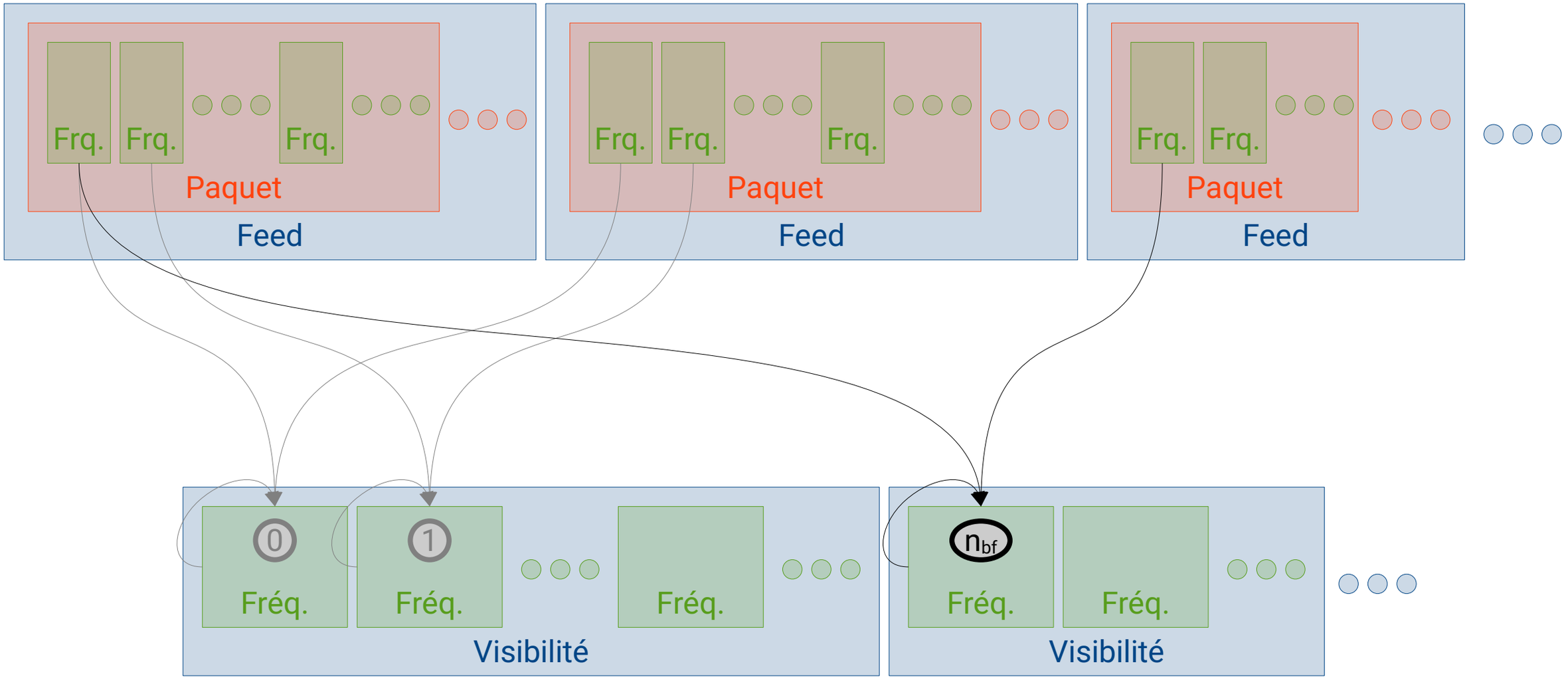
Flux de données (fréquences par blocs)



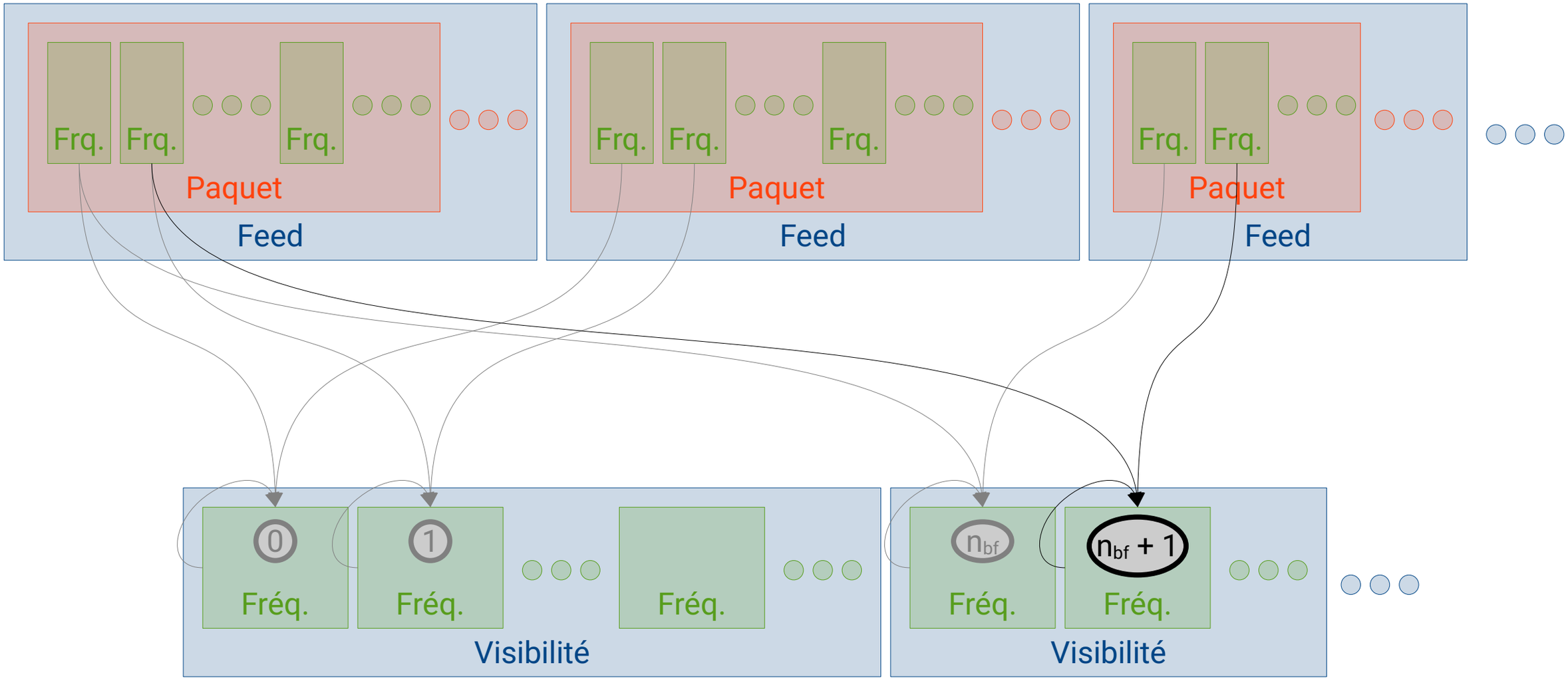
Flux de données (fréquences par blocs)



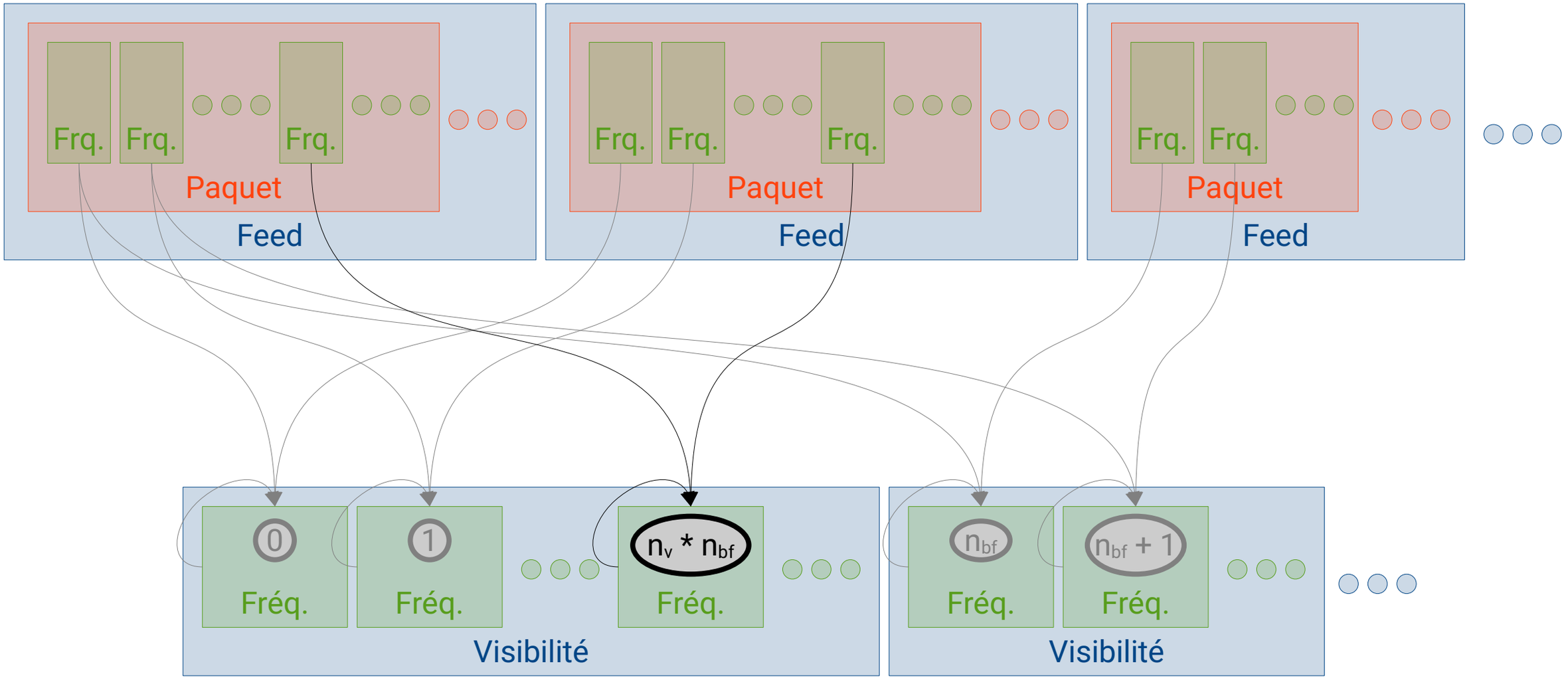
Flux de données (fréquences par blocs)



Flux de données (fréquences par blocs)



Flux de données (fréquences par blocs)

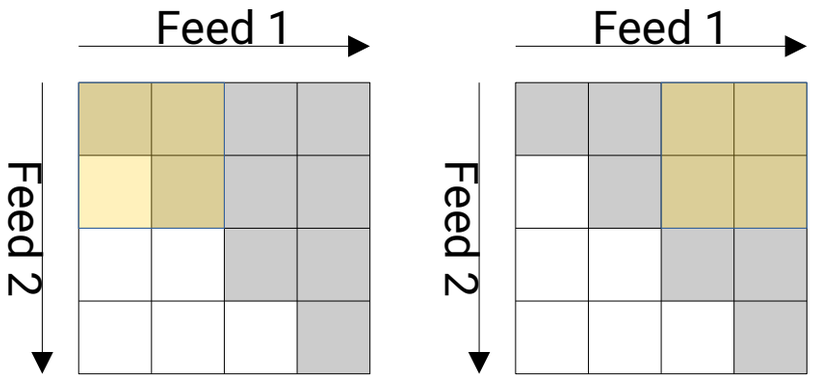


Localité temporelle du 2e feed

- Les blocs suffisent pour une localité temporelle du 1er feed…
  - …mais celle du 2e feed n'est optimisée que si les blocs de **tous** les feeds tiennent dans le cache L1
  - Possible avec de tout petits blocs de fréquences…
    - …mais ça diminue l'efficacité de la boucle interne.
    - ...et ça ne marcherait pas pour davantage de feeds
- Pour faire mieux, il faut optimiser l'itération sur les feeds

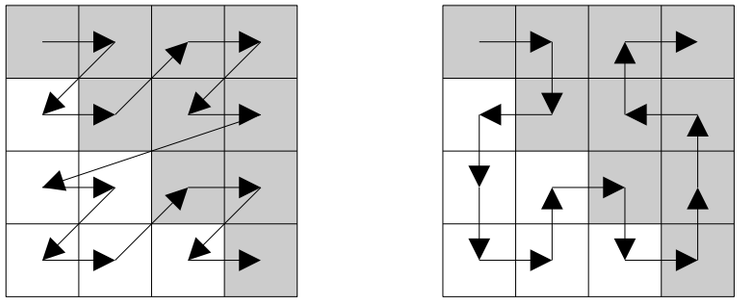
Localité d'itération 2D

- On peut refaire des blocs…



- …mais pénible à « régler »
  - Le bloc optimal dépend de n_i **et** de la taille du cache

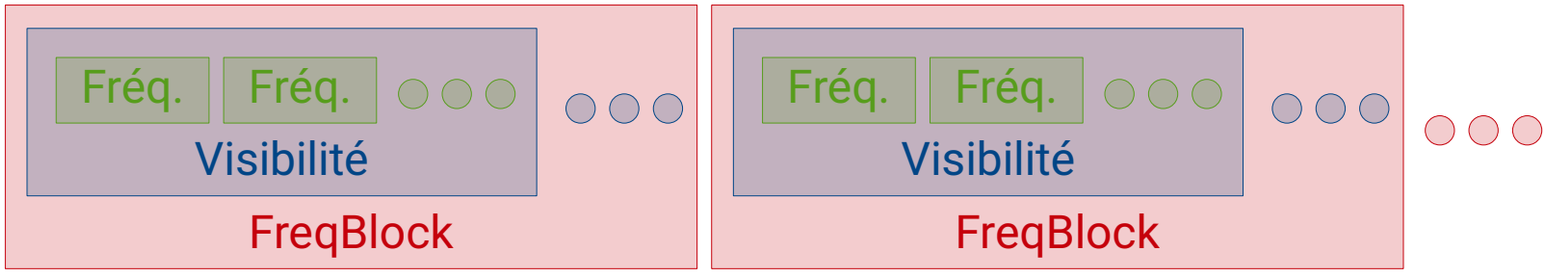
- Les courbes fractales sont une alternative intéressante



- Plutôt étudiées en « carré »
  - J'explore le cas triangulaire avec un simulateur

Localité temporelle de l'accumulateur

- Après les données d'entrée, il faudra optimiser l'accumulateur
  - On ne revient à un de ses points qu'en changeant de paquet
  - La boucle sur les paquets est actuellement la plus externe…
- Possibilités à explorer :
  - Raffiner ordre d'itération : freqBlock{ paquet{ visi{ freq }}} ?
  - Revoir l'ordre de stockage de l'accumulateur, idée actuelle :



Optimisation du calcul

Vectorisation

- Les CPUs et GPUs sont capables de vectorisation (SIMD)
  - 1 instruction pour charger N données **consécutives**
  - 1 instruction pour additionner, multiplier… ces N données
  - 1 instruction pour les sauvegarder de façon **consécutive**
- Pourquoi faire ça quand on a déjà des *threads* ?
  - Globalement plus efficace* que le multi-threading
  - Nécessaire pour utiliser toutes les ALUs (encore + sur GPU !)

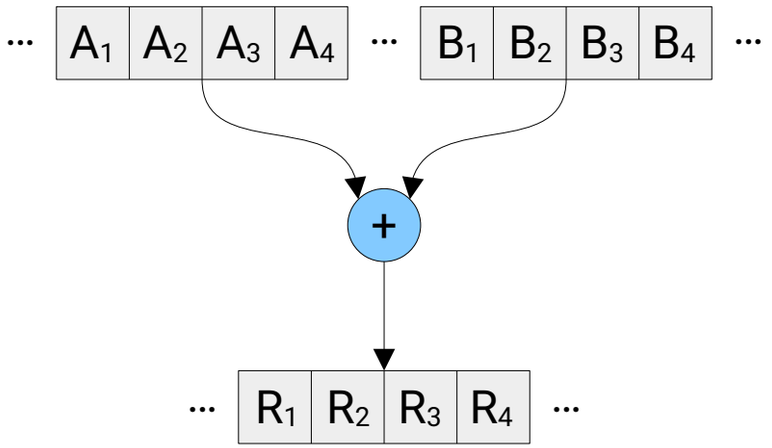
* Que l'on raisonne en termes d'énergie consommée, de synchronisation, d'utilisation et localité de cache, de coût financier, ou de transistors par FLOP au niveau matériel…

Limites de la vectorisation

- Les lectures/écritures doivent être consécutives
  - Sinon, on doit faire N lectures et combiner → très lent
- On doit faire la même opération sur les N données
  - Sinon, il faut les ségréger avant → assez lent
- On doit avoir exactement N données à traiter
  - Si on en a moins, le coût est le même → réductions lentes

Scénarios de vectorisation (1)

Scénario idéal*

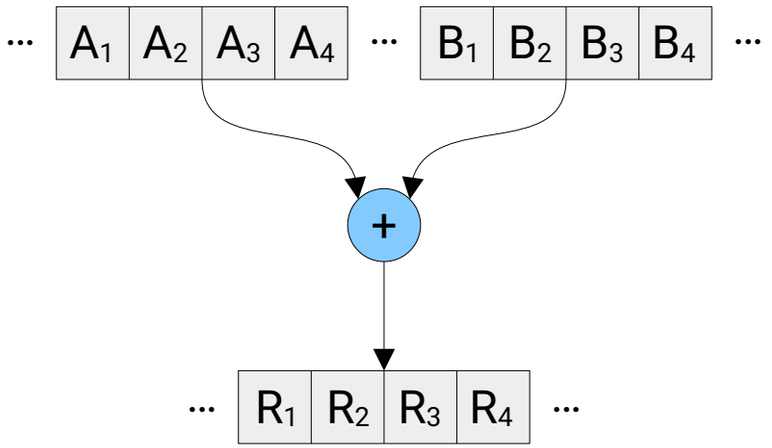


- 2 lectures de données d'entrée
- 1 opération arithmétique désirée
- 1 écriture des données de sortie
- 2 registres vectoriels utilisés
- …et on passe à la suite !

* Si l'on a que des additions à faire

Scénarios de vectorisation (1)

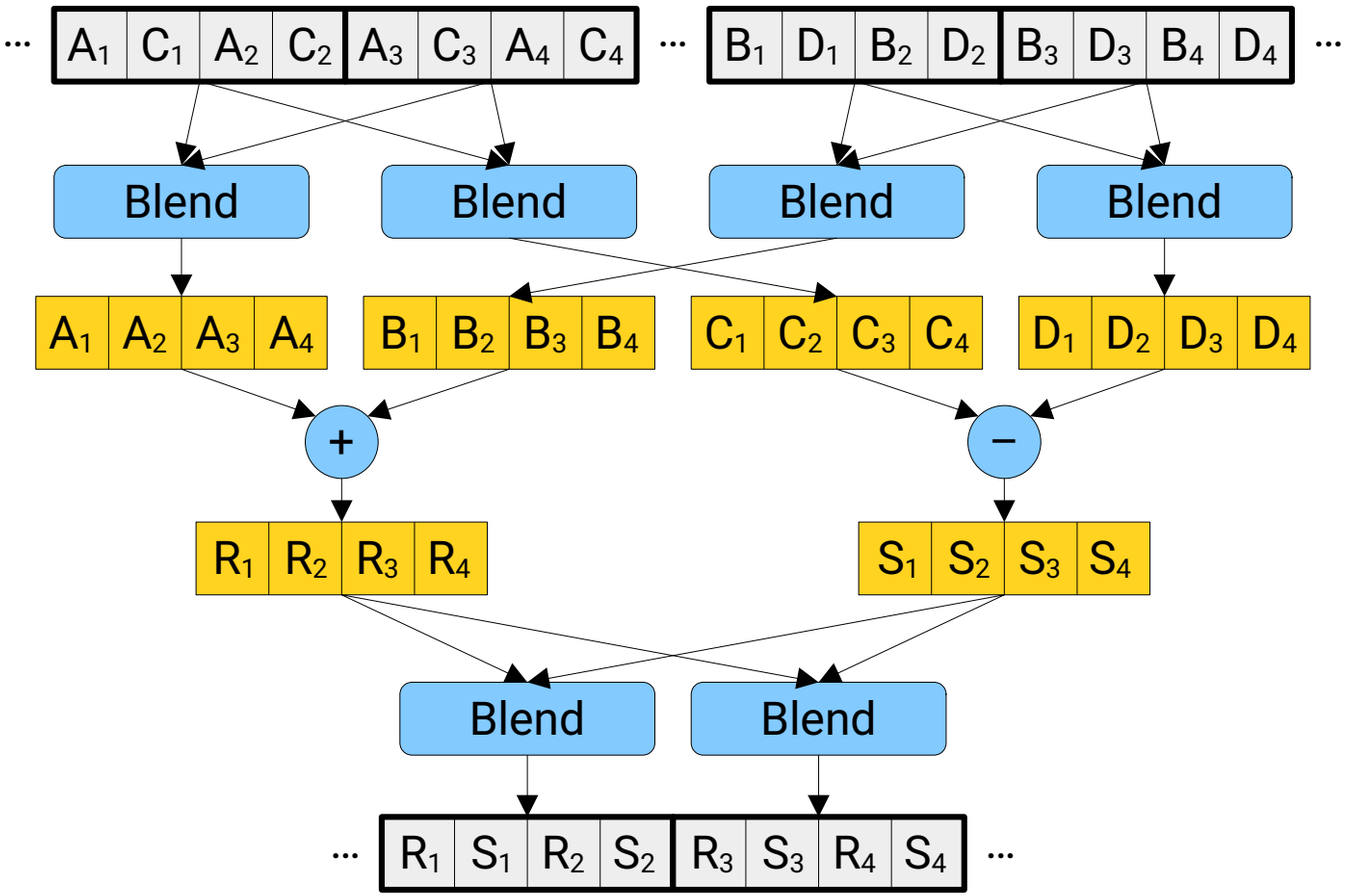
Scénario idéal*



- 2 lectures de données d'entrée
- 1 opération arithmétique désirée
- 1 écriture des données de sortie
- 2 registres vectoriels utilisés
- …et on passe à la suite !

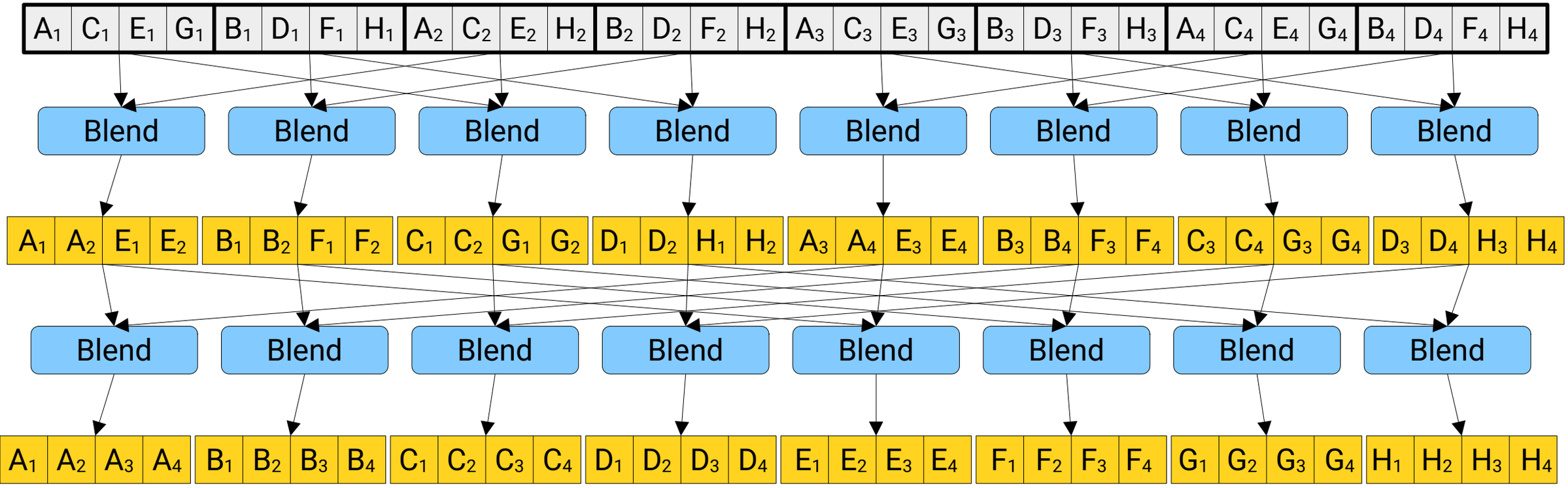
* Si l'on a que des additions à faire

Scénario perfectible



Scénarios de vectorisation (2)

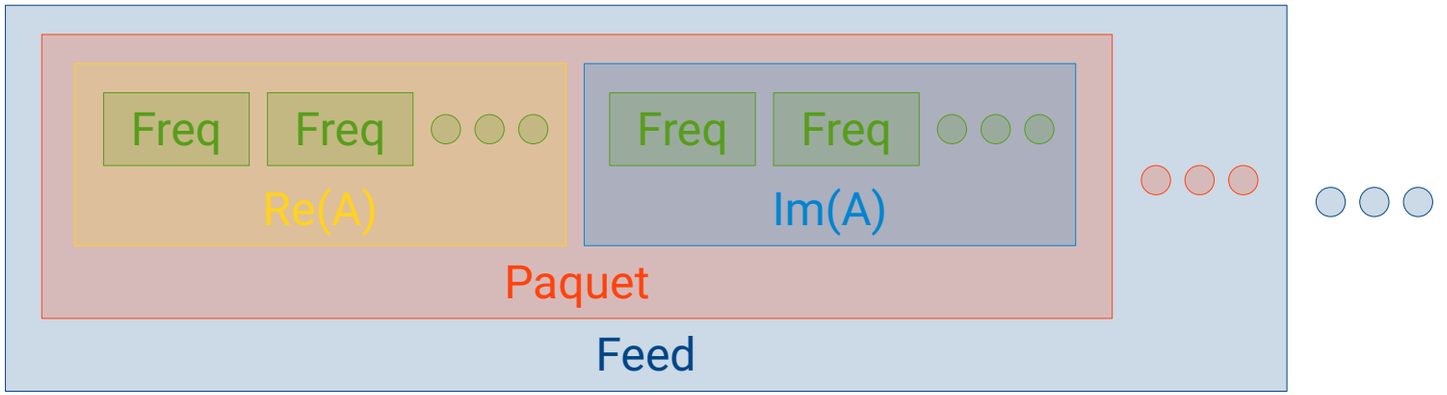
Scénario de l’échec*



* En réalité, on peut faire pire en étalant les données en mémoire, mais le but n’est pas de gagner un concours de médiocrité…

Revenons à nos corrélations

- Comme je disais, inutile d'optimiser le calcul avant la mémoire
  - Avant localité spatiale : pire que le « scénario de l'échec » !
  - Après localité spatiale : proche du « scénario perfectible »
- Pour le « scénario idéal », il faudrait grouper les Re(A) et Im(A)
  - FFTW sait le faire*, possible pour firmware IDROGEN aussi ?



* http://fftw.org/fftw3_doc/Guru-Complex-DFTs.html#Guru-Complex-DFTs

Une fois les données rangées…

- GCC vectorise à partir de `-O3` (ou `-O2 -ftree-vectorize`)…
  - …mais ne change pas l'ordre des opérations flottantes
  - Pour ça, il faut `-Ofast` ou `-O<n> -ffast-math`
  - Nécessaire avec `std::complex`, pas pour un calcul manuel
- Par défaut, GCC n'utilise que la vectorisation SSE2 (4 x f32)
  - Raison : Tous les CPUs ne supportent pas AVX & cie
  - Activable avec `-m<x>` (approche brutale : `-march=native`)

Un mot sur l'alignement

- Les CPUs et GPUs aiment bien les **accès mémoires alignés**
  - Taille de N octets → Adresse mémoire multiple de N
  - Pour un registre vectoriel, concerne le vecteur entier !
- Les CPUs x86 actuels *tolèrent* les accès non alignés…
  - …mais pas la vectorisation GCC, qui en prend un coup !
  - Ni les CPUs x86 plus anciens, autres CPUs, GPUs…*
  - Solution : Utiliser un allocateur aligné et l'indiquer à GCC

* Selon le matériel, l'opération est soit impossible, soit plus lente. Les compilateurs l'évitent donc.

Pour conclure

Statut de ce travail

- Lecture du 1er feed : complètement optimisée
- Lecture du 2e feed : en cours d'optimisation
  - J'étudie les motifs d'itération permettant une localité 2D
- Lecture et écriture de l'accumulateur : pas encore optimisées
- Vectorisation du calcul : brouillon relativement complet
  - Approche actuelle : Re(A)/Im(A) ségrégés en entrée, calcul std::complex, -Ofast -march=native, tableaux alignés.
  - Reste à choisir l'approche finale + dérouler un peu la boucle

A plus long terme

- Evaluer la marge de manœuvre sur les formats de données
  - En entrée : Possibilité de grouper les Re(A) et Im(A) ?
  - En sortie : Ordre d'imbrication ? Stockage triangulaire ?
  - D'autres questions pourraient émerger par la suite
- Porter ces optimisations du calcul aux autres versions
  - D'autant plus de travail qu'il y a de versions, bien sûr
  - Combien de versions compte-t'on garder à terme ?

Performance finale attendue ?

- Je n'ai pas fini le travail pratique, ni même l'analyse théorique
  - Mais au mieux, limité par la lecture des entrées en (V)RAM
  - Sur mon petit Core i3-3220 : 6 Gf32/s
  - Sur le Xeon Silver 4210 de pc-bao2 : 29 Gf32/s
  - Sur les Quadro P4000 : 72 Gf32/s
- Gare aux interconnections, qui peuvent devenir la limite
  - PCI-express entre CPU et GPU : 4 Gf32/s !
  - Données entrantes : 40 Gbit/s (~1,3 Gf32/s) pour 4 PCs ?
  - Pour éviter ça, enchaîner traitements au même endroit !

Merci de votre attention !