Optimisation de code

Fabrice Rossi

Télécom ParisTech

Novembre 2009

Plan

Plan

Motivations

- Motivation principale : écrire du code rapide

Motivations

- Motivation principale : écrire du code rapide
  - c'est-à-dire exploitant la puissance disponible
  - sans pour autant passer par de l'assembleur
  - et même dans les langages éloignées du « métal » (Java ou C#)

  $\Rightarrow$ modifications algorithmiques

Motivations

- Motivation principale : écrire du code rapide
  - c'est-à-dire exploitant la puissance disponible
  - sans pour autant passer par de l'assembleur
  - et même dans les langages éloignées du « métal » (Java ou C#)

  $\Rightarrow$ modifications algorithmiques
- Contexte : données volumineuse

Motivations

- Motivation principale : écrire du code rapide
  - c'est-à-dire exploitant la puissance disponible
  - sans pour autant passer par de l'assembleur
  - et même dans les langages éloignées du « métal » (Java ou C#)

  $\Rightarrow$ modifications algorithmiques
- Contexte : données volumineuse
  - multimédia : par ex. traitement d'images et de vidéo
    - image 24 $\times$ 36 reflex haut de gamme : 6048 $\times$ 4032 pixels $\simeq$ 73 Mo
    - vidéo HD 1920 $\times$ 1080 $\simeq 4,8$ Mo/s en H.264
  - flux de données : par ex. requêtes sur un site web (wikipedia : pointes à 70 000 requête/s)

Motivations

- Motivation principale : écrire du code rapide
  - c'est-à-dire exploitant la puissance disponible
  - sans pour autant passer par de l'assembleur
  - et même dans les langages éloignées du « métal » (Java ou C#)

  $\Rightarrow$ modifications algorithmiques
- Contexte : données volumineuse
  - multimédia : par ex. traitement d'images et de vidéo
    - image 24 $\times$ 36 reflex haut de gamme : 6048 $\times$ 4032 pixels $\simeq$ 73 Mo
    - vidéo HD 1920 $\times$ 1080 $\simeq 4,8$ Mo/s en H.264
  - flux de données : par ex. requêtes sur un site web (wikipedia : pointes à 70 000 requête/s)
- goulet d'étranglement : le transfert des données vers le CPU

CPU

- Les CPU (x86) modernes sont très rapides :
  - fréquence d'horloge 3/3.2 Ghz (Core 2 Duo E8400 / Core 2 Extreme QX9770)
  - performances théoriques de l'ordre de 10 Gflop/s (en exploitant les 2 coeurs)

CPU

- Les CPU (x86) modernes sont très rapides :
  - fréquence d'horloge 3/3.2 Ghz (Core 2 Duo E8400 / Core 2 Extreme QX9770)
  - performances théoriques de l'ordre de 10 Gflop/s (en exploitant les 2 coeurs)
- Exemples théoriques :
  - chacun des 6048×4032 pixels d'une photo peut être traité 400 fois par seconde
  - TV HD 50Hz : environ 100 opérations par seconde par pixel

CPU

- Les CPU (x86) modernes sont très rapides :
  - fréquence d'horloge 3/3.2 Ghz (Core 2 Duo E8400 / Core 2 Extreme QX9770)
  - performances théoriques de l'ordre de 10 Gflop/s (en exploitant les 2 coeurs)
- Exemples théoriques :
  - chacun des 6048×4032 pixels d'une photo peut être traité 400 fois par seconde
  - TV HD 50Hz : environ 100 opérations par seconde par pixel
- Exemples réels :
  - Conversion Wav -> MP3 (lame) sur E8400 $\simeq$ 6.54 Mo/s :
    - Wav : 176 400 octets par seconde (39 fois le temps réel)
    - 56700 flop par octets ! ! !
  - Conversion VOB (MPEG-2) -> Divx sur E8400 $\simeq$ 1.20 Mo/s (temps réel)

CPU

- Les CPU (x86) modernes sont très rapides :
  - fréquence d'horloge 3/3.2 Ghz (Core 2 Duo E8400 / Core 2 Extreme QX9770)
  - performances théoriques de l'ordre de 10 Gflop/s (en exploitant les 2 coeurs)
- Exemples théoriques :
  - chacun des 6048×4032 pixels d'une photo peut être traité 400 fois par seconde
  - TV HD 50Hz : environ 100 opérations par seconde par pixel
- Exemples réels :
  - Conversion Wav -> MP3 (lame) sur E8400 $\simeq$ 6.54 Mo/s :
    - Wav : 176 400 octets par seconde (39 fois le temps réel)
    - 56700 flop par octets ! ! !
  - Conversion VOB (MPEG-2) -> Divx sur E8400 $\simeq$ 1.20 Mo/s (temps réel)
- Où passe la puissance brute ?

Mémoire principale

- La mémoire centrale est relativement lente :
  - fréquence du bus 667/800 MHz (E8400/QX9770)
  - horloge de la mémoire 166/200 MHz : 15 fois plus lente que celle du CPU
  - mais la bande passante est élevée : 10,7/12,8 Go/s

Mémoire principale

- La mémoire centrale est relativement lente :
  - fréquence du bus 667/800 MHz (E8400/QX9770)
  - horloge de la mémoire 166/200 MHz : 15 fois plus lente que celle du CPU
  - mais la bande passante est élevée : 10,7/12,8 Go/s
- Le ratio flops / bande passante est correct : 1 octet par flop

Mémoire principale

- La mémoire centrale est relativement lente :
  - fréquence du bus 667/800 MHz (E8400/QX9770)
  - horloge de la mémoire 166/200 MHz : 15 fois plus lente que celle du CPU
  - mais la bande passante est élevée : 10,7/12,8 Go/s
- Le ratio flops / bande passante est correct : 1 octet par flop
- Exemples théoriques :
  - 5,4 centièmes de seconde pour lire une image 6048×4032
  - 14 centièmes de seconde pour lire une minute de vidéo HD 1920×1080 (30 images)

Mémoire principale

- La mémoire centrale est relativement lente :
  - fréquence du bus 667/800 MHz (E8400/QX9770)
  - horloge de la mémoire 166/200 MHz : 15 fois plus lente que celle du CPU
  - mais la bande passante est élevée : 10,7/12,8 Go/s
- Le ratio flops / bande passante est correct : 1 octet par flop
- Exemples théoriques :
  - 5,4 centièmes de seconde pour lire une image 6048×4032
  - 14 centièmes de seconde pour lire une minute de vidéo HD 1920×1080 (30 images)
- Conséquence : on devrait pouvoir atteindre le pic de puissance même avec un algorithme en $O(N)$

Mémoire principale

- La mémoire centrale est relativement lente :
  - fréquence du bus 667/800 MHz (E8400/QX9770)
  - horloge de la mémoire 166/200 MHz : 15 fois plus lente que celle du CPU
  - mais la bande passante est élevée : 10,7/12,8 Go/s
- Le ratio flops / bande passante est correct : 1 octet par flop
- Exemples théoriques :
  - 5,4 centièmes de seconde pour lire une image 6048×4032
  - 14 centièmes de seconde pour lire une minute de vidéo HD 1920×1080 (30 images)
- Conséquence : on devrait pouvoir atteindre le pic de puissance même avec un algorithme en $O(N)$
- Mais il y a la latence

Latence

- Latence : temps d'attente entre la début d'une opération et sa complétion
- pour la mémoire : on demande l'octet d'adresse x, combien de temps faut-il l'attendre ?

Latence

- Latence : temps d'attente entre la début d'une opération et sa complétion
- pour la mémoire : on demande l'octet d'adresse x, combien de temps faut-il l'attendre ?
- latence mémoire perçue $=$ latence CPU + latence BUS + latence mémoire
- sur un Core 2 Duo : 14 cycles CPU + 5,5 cycles bus + mémoire :
  - 5,5 cycles bus $\simeq$ 22 cycles CPU
  - latence mémoire classique : 5 à 10 cycles mémoire (80 à 160 cycles CPU)
  - latence totale : de l'ordre de 200 cycles CPU
  - puis il faut attendre pour pouvoir demander un nouveau transfert (de l'ordre de 5 cycles mémoire)

Latence

- Latence : temps d'attente entre la début d'une opération et sa complétion
- pour la mémoire : on demande l'octet d'adresse x, combien de temps faut-il l'attendre ?
- latence mémoire perçue = latence CPU + latence BUS + latence mémoire
- sur un Core 2 Duo : 14 cycles CPU + 5,5 cycles bus + mémoire :
  - 5,5 cycles bus $\simeq$ 22 cycles CPU
  - latence mémoire classique : 5 à 10 cycles mémoire (80 à 160 cycles CPU)
  - latence totale : de l'ordre de 200 cycles CPU
  - puis il faut attendre pour pouvoir demander un nouveau transfert (de l'ordre de 5 cycles mémoire)
- 1 octet tout les 200 cycles : 15 Mo/s !
- taux de transfert d'un bon disque dur : 50 Mo/s !

Largeur de bus

- Mais le bus fait 64 bits de large : 8 octets par transfert
- 8 octets tout les 200 cycles : 120 Mo/s

Largeur de bus

- Mais le bus fait 64 bits de large : 8 octets par transfert
- 8 octets tout les 200 cycles : 120 Mo/s
- Mémoire double canal : deux mémoires accédées en parallèle
- 128 bits par lecture : 240 Mo/s (contre 25,6 Go/s en théorie !)

Largeur de bus

- Mais le bus fait 64 bits de large : 8 octets par transfert
- 8 octets tout les 200 cycles : 120 Mo/s
- Mémoire double canal : deux mémoires accédées en parallèle
- 128 bits par lecture : 240 Mo/s (contre 25,6 Go/s en théorie !)
- Ratio calcul/accès :
  - au mieux 42 flops par octet
  - en supposant qu'on peut traiter les données par blocs de 16 octets
  - on ne peut plus atteindre le pic de puissance pour un algorithme en cN, sauf si c est grand
  - on ne fait que marginalement mieux que le disque dur !

Solutions

- Transferts par blocs :
  - la mémoire peut atteindre sa bande passante théorique
  - à condition de transférer des blocs entiers
- Caches :
  - le CPU est aidé par des caches
  - mémoire rapide (en transfert et en latence) mais petite
- Préchargement :
  - le contrôleur de mémoire précharge des blocs dans le cache
  - principe : réduire la latence effective
- Ça ne suffit pas :
  - il faut que les accès mémoire soient localisés
  - il faut aider le CPU et la mémoire

Objectifs du cours

- Comprendre la source des problèmes :
  - Fonctionnement de la SDRAM
  - Impact sur les performances
- Limiter les problèmes :
  - limitation de l'algorithmique classique
  - favoriser la localité en mémoire
  - analyse d'un algorithme :
    - complexité en temps
    - complexité en espace
    - complexité en accès
  - techniques d'optimisation :
    - transformer de boucles
    - positionner les données correctement

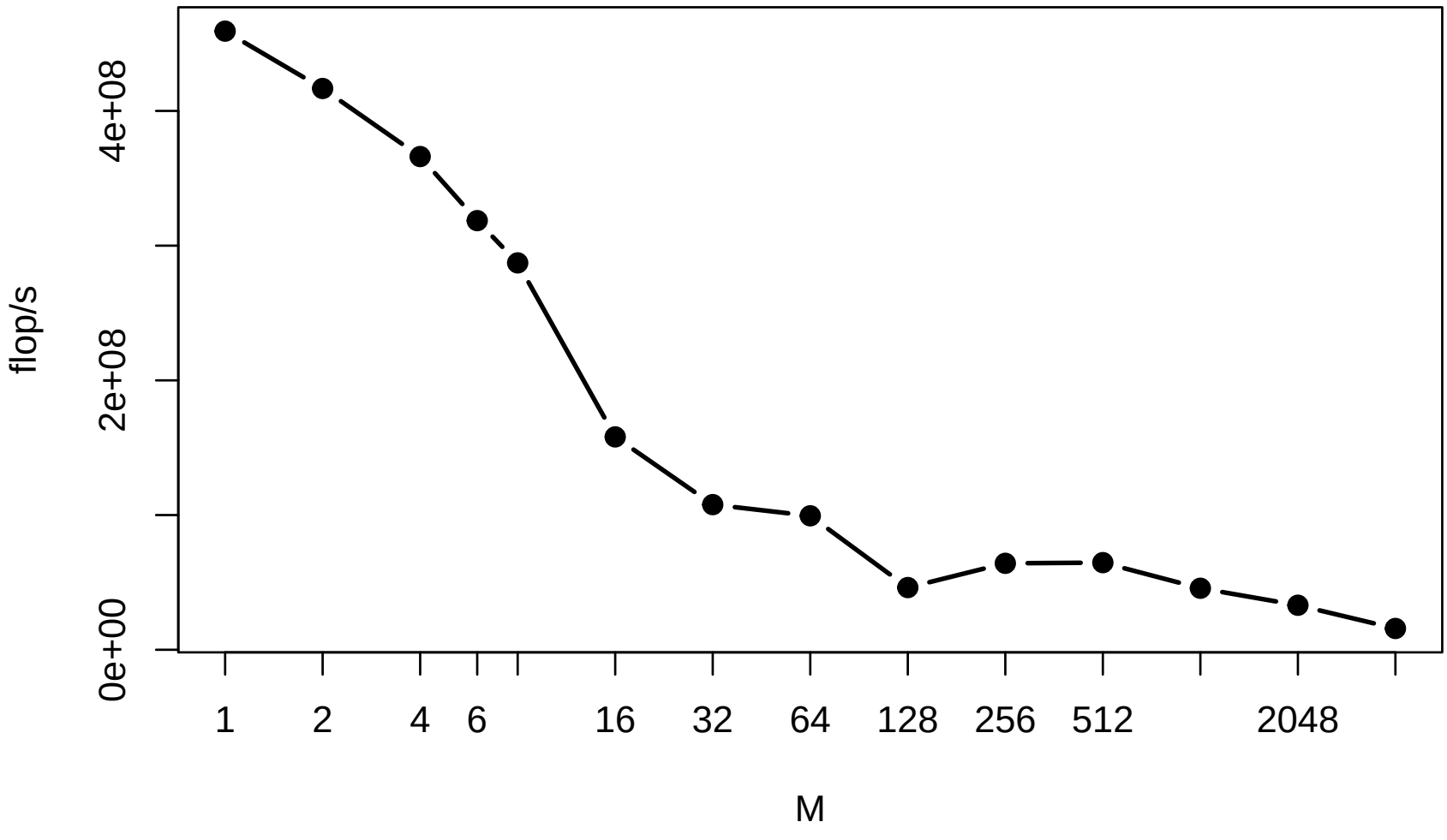
Un exemple élémentaire

- un tableau de données : N lignes et M colonnes
- un `double` par case : $8NM$ octets
- but : calculer la moyenne de chaque colonne $O(NM)$
- code (Java) :

```
double[][] data=new double[N][M];
...
double[] mean=new double[M];
for(int j=0;j<M;j++) {
  double tmp=0;
  for(int i=0;i<N;i++) {
    tmp += data[i][j];
  }
  mean[j]=tmp/N;
}
```

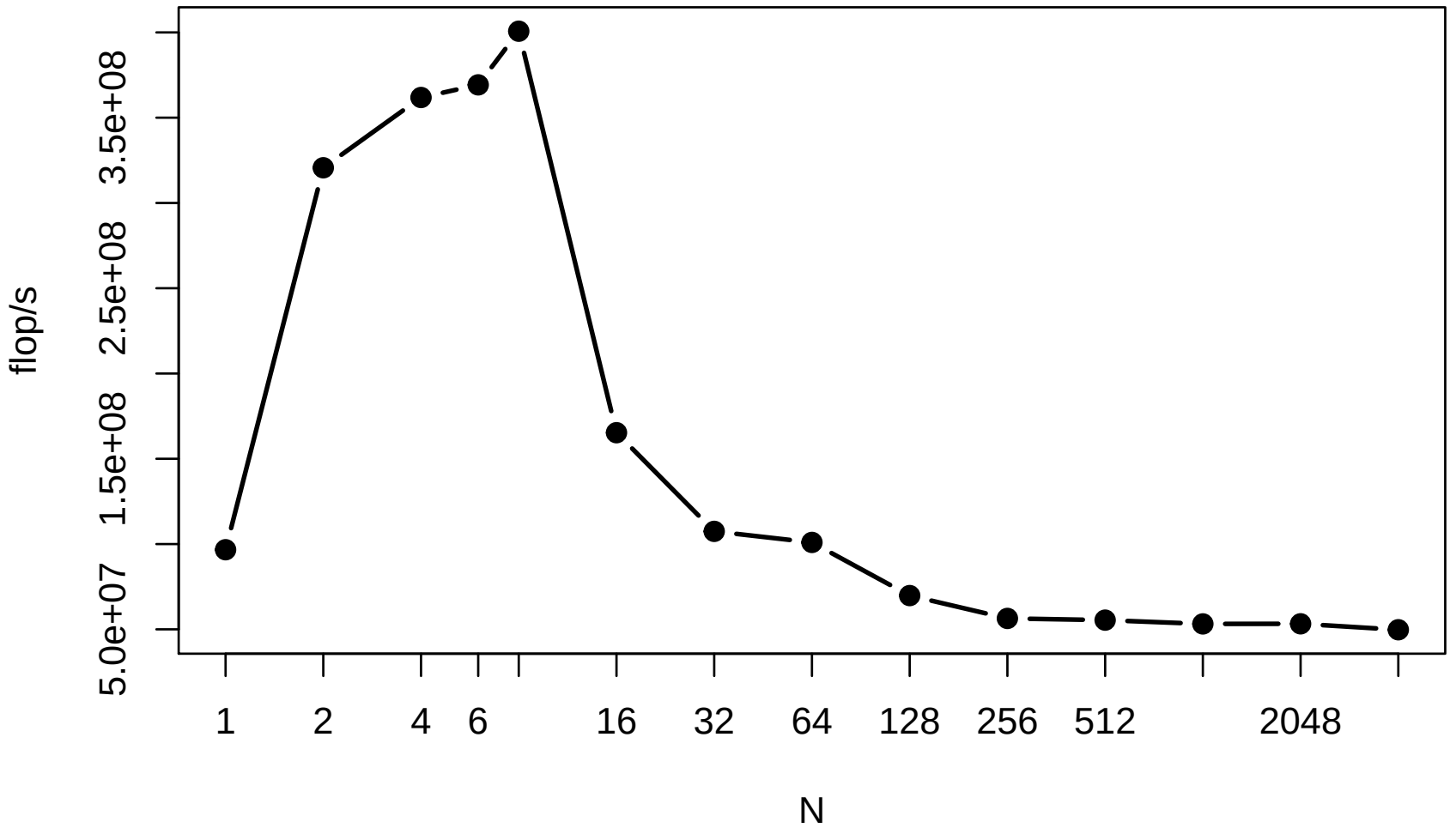
- calcul colonne par colonne

Résultats pour $N = 16384$

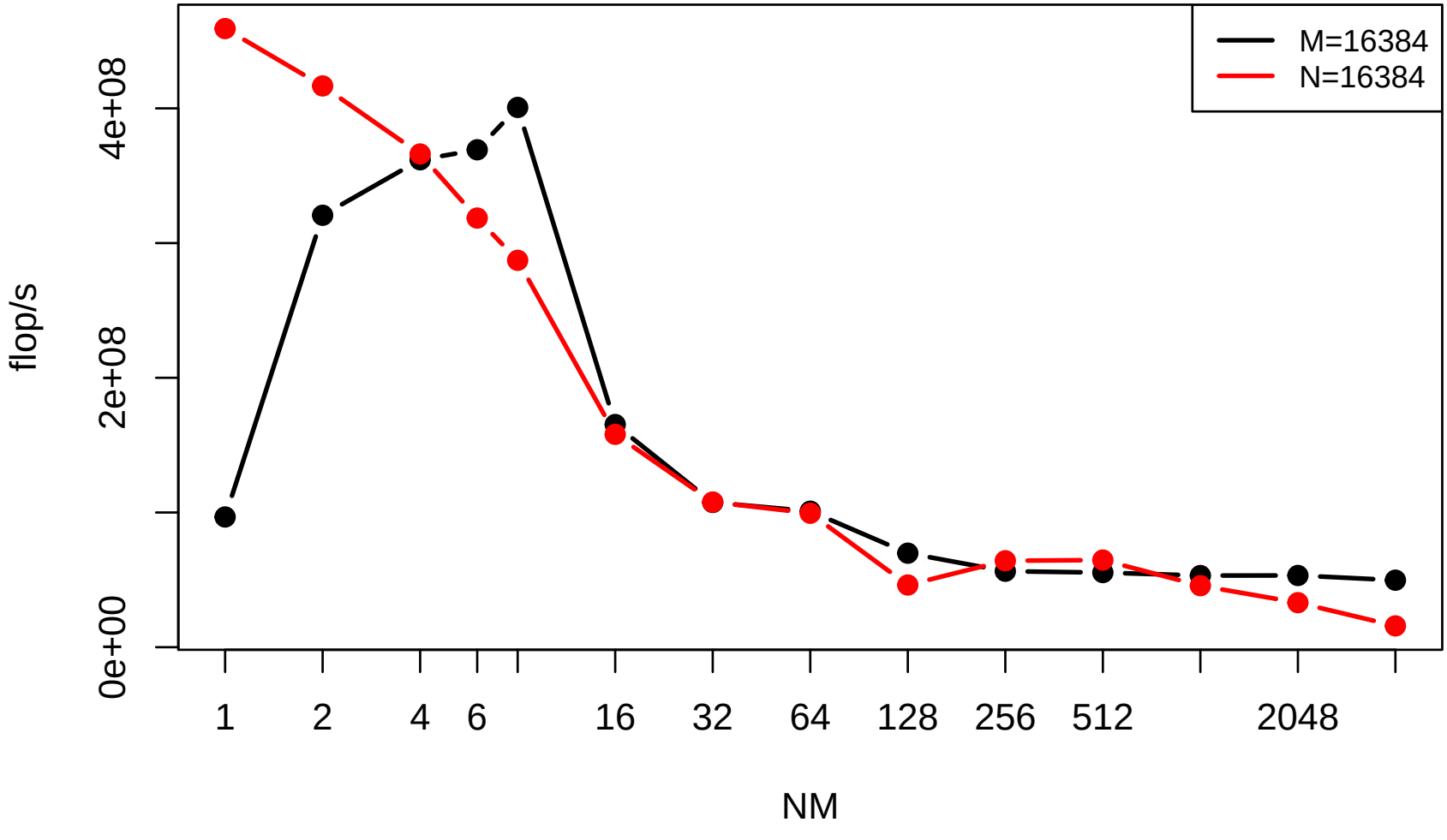


Sur un Core 2 Duo P8600 (2.4 GHz)

Résultats pour $M = 16384$



Sur un Core 2 Duo P8600 (2.4 GHz)



Sur un Core 2 Duo P8600 (2.4 GHz)

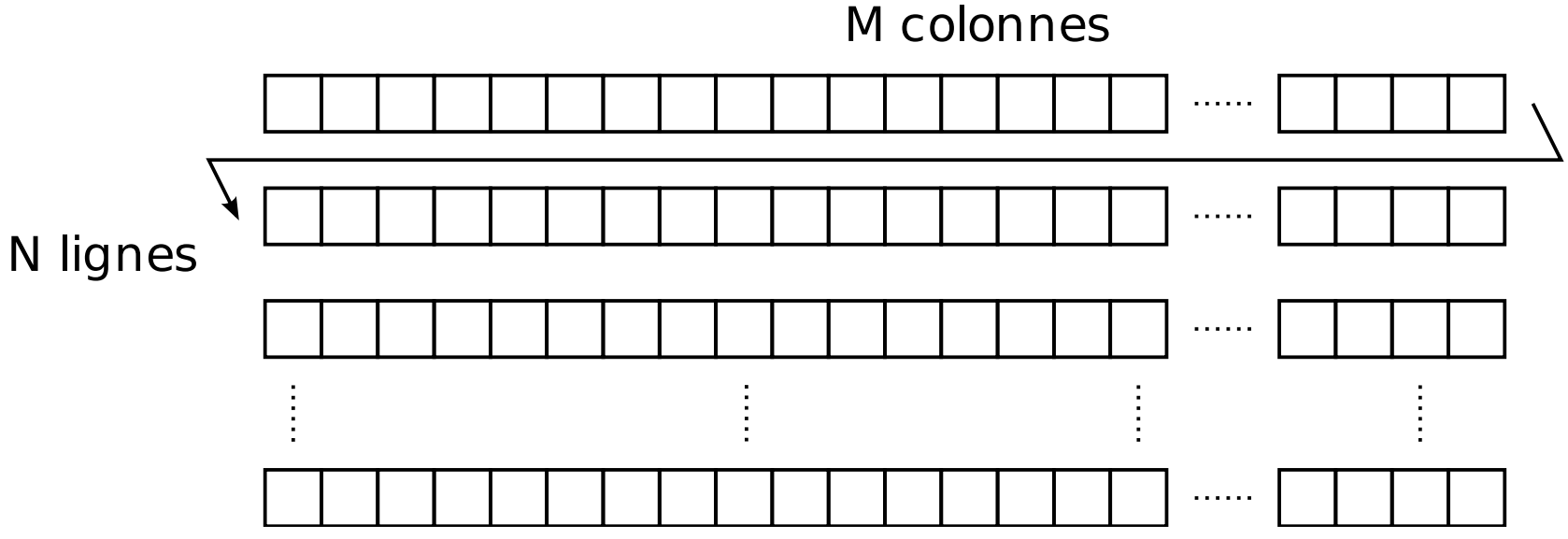
Analyse

- la puissance effective chute avec la taille des données
- l'accès aux données n'est pas local :

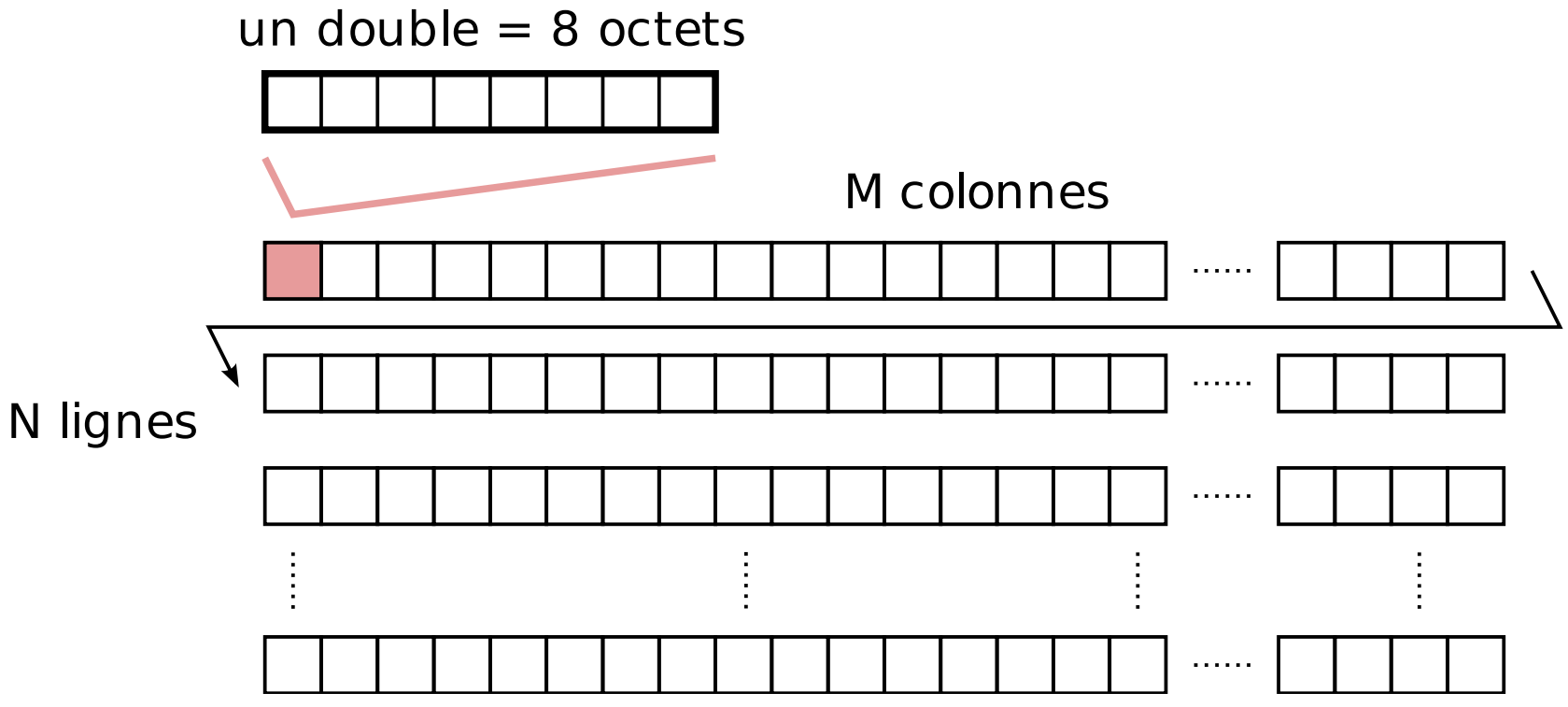
```
for(int j=0;j<M;j++) {
  double tmp=0; // registre
  for(int i=0;i<N;i++) {
    tmp += data[i][j]; // saut de 8M octets
  }
  mean[j]=tmp/N;
}
```

- la latence de la mémoire peut rendre l'accès à `data[i][j]` très lent
- pour $N = 16384$ et $M = 4096$ (512 Mo) on tombe à 15 Mflop/s soit 157 cycles d'horloge par calcul !

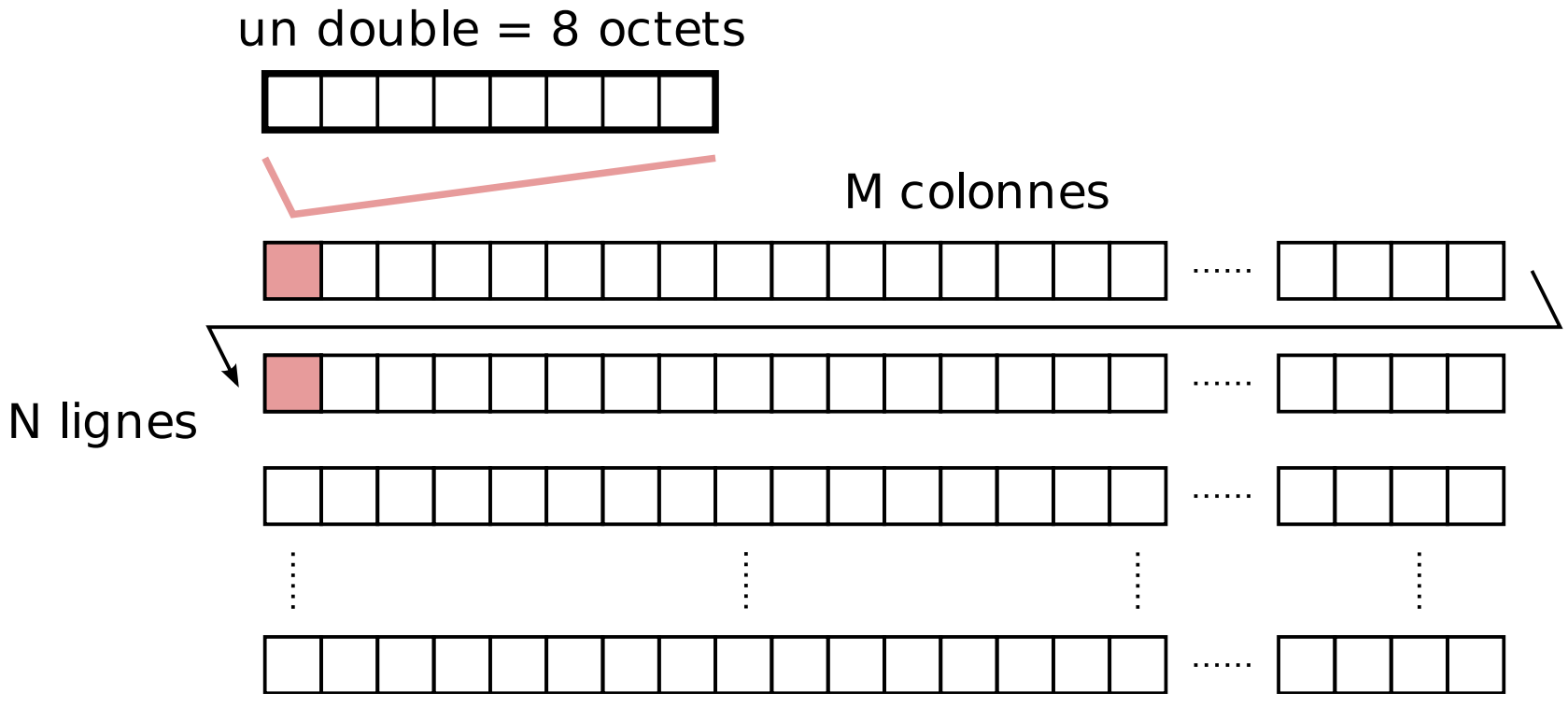
Disposition en mémoire



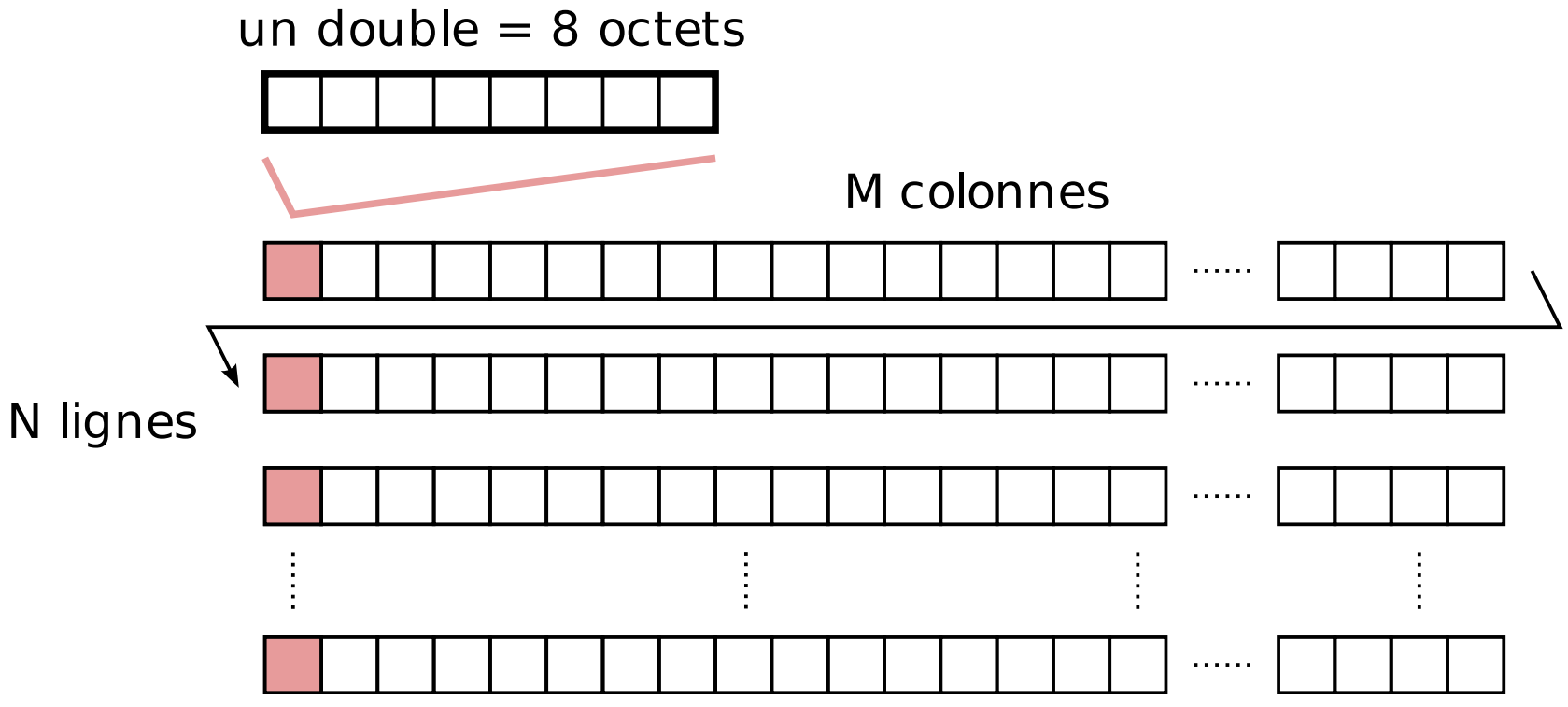
Disposition en mémoire



Disposition en mémoire



Disposition en mémoire

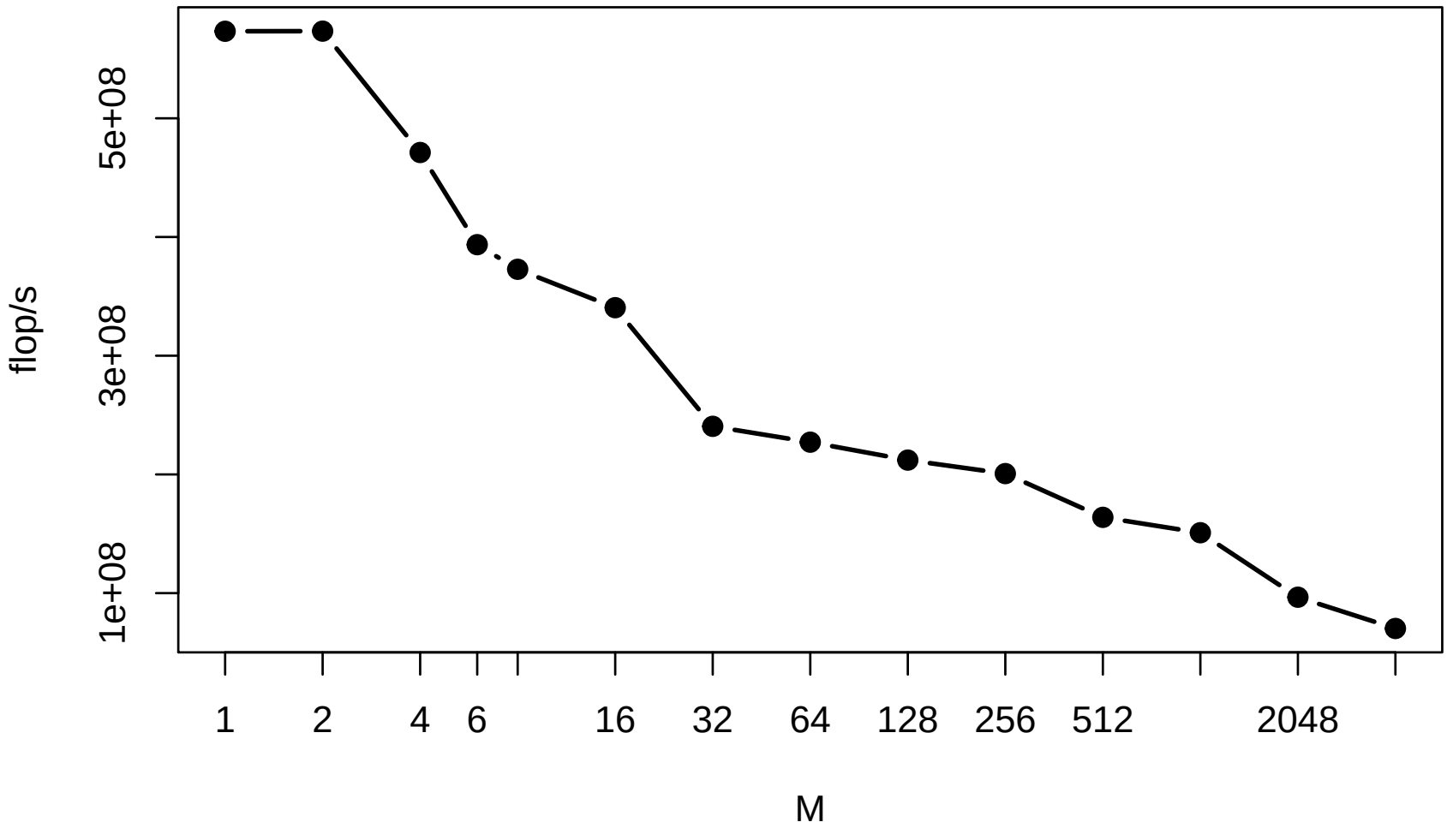


Et en C ?

- implémentation identique (ou presque) :

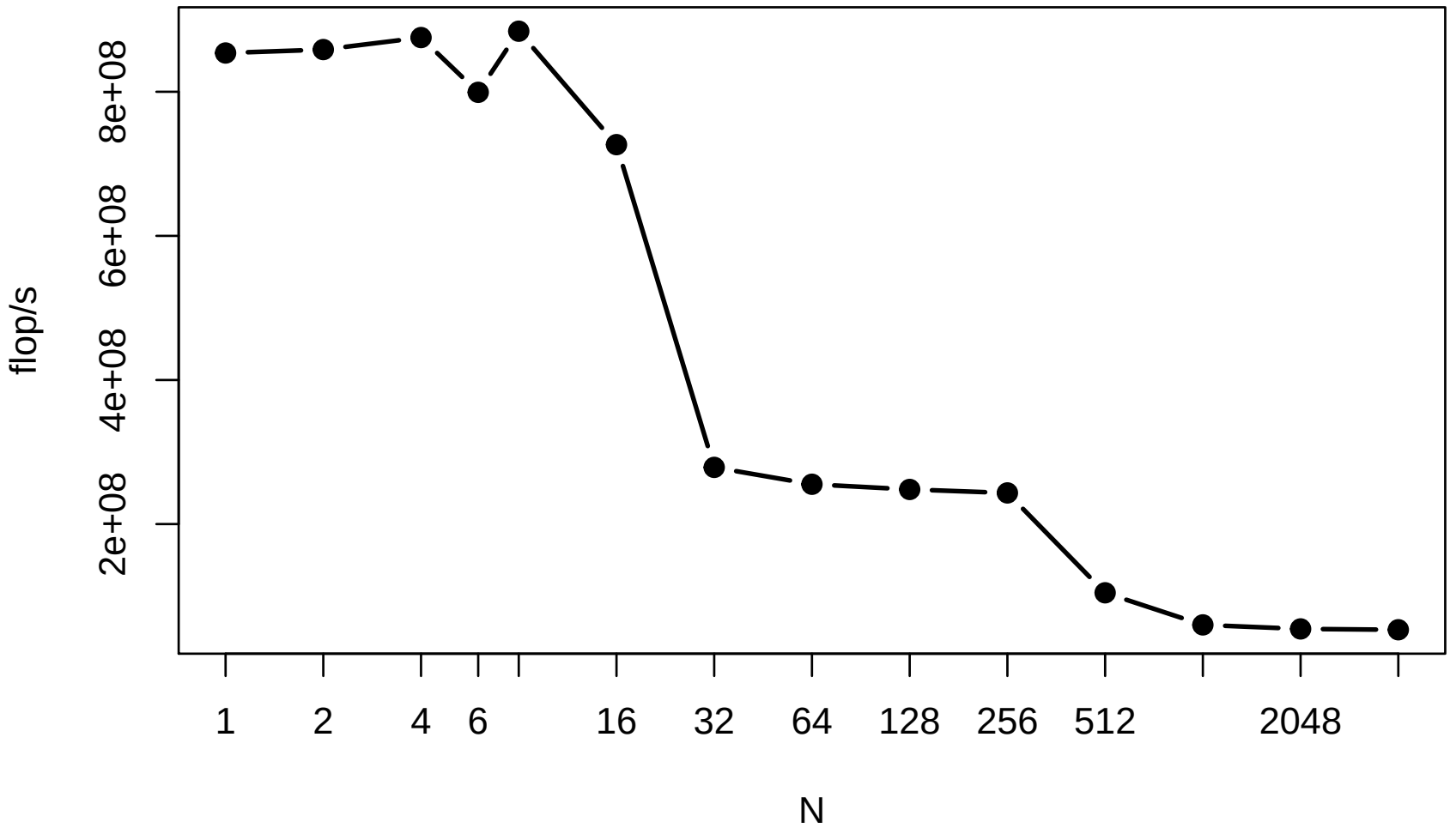
```
double **data;
double *mean;
double tmp;
...
data = (double **)malloc(sizeof(double *)*N);
..
mean = (double *)malloc(sizeof(double)*M);
for(j=0;j<M;j++) {
  tmp = 0;
  for(i=0;i<N;i++) {
    tmp += data[i][j];
  }
  mean[j] = tmp/N;
}
free(mean);
```

Résultats pour $N = 16384$

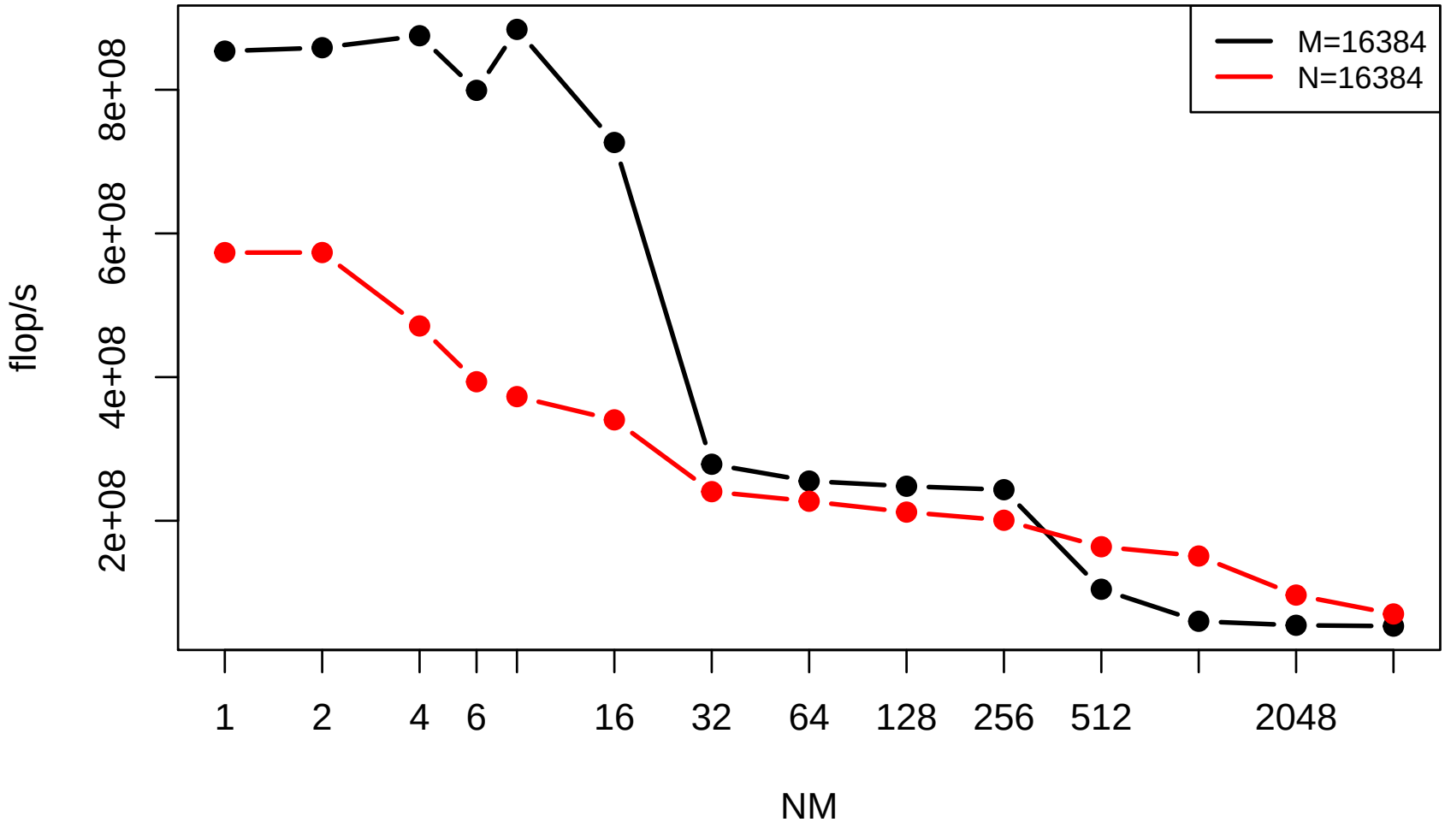


Sur un Core 2 Duo P8600 (2.4 GHz)

Résultats pour $M = 16384$

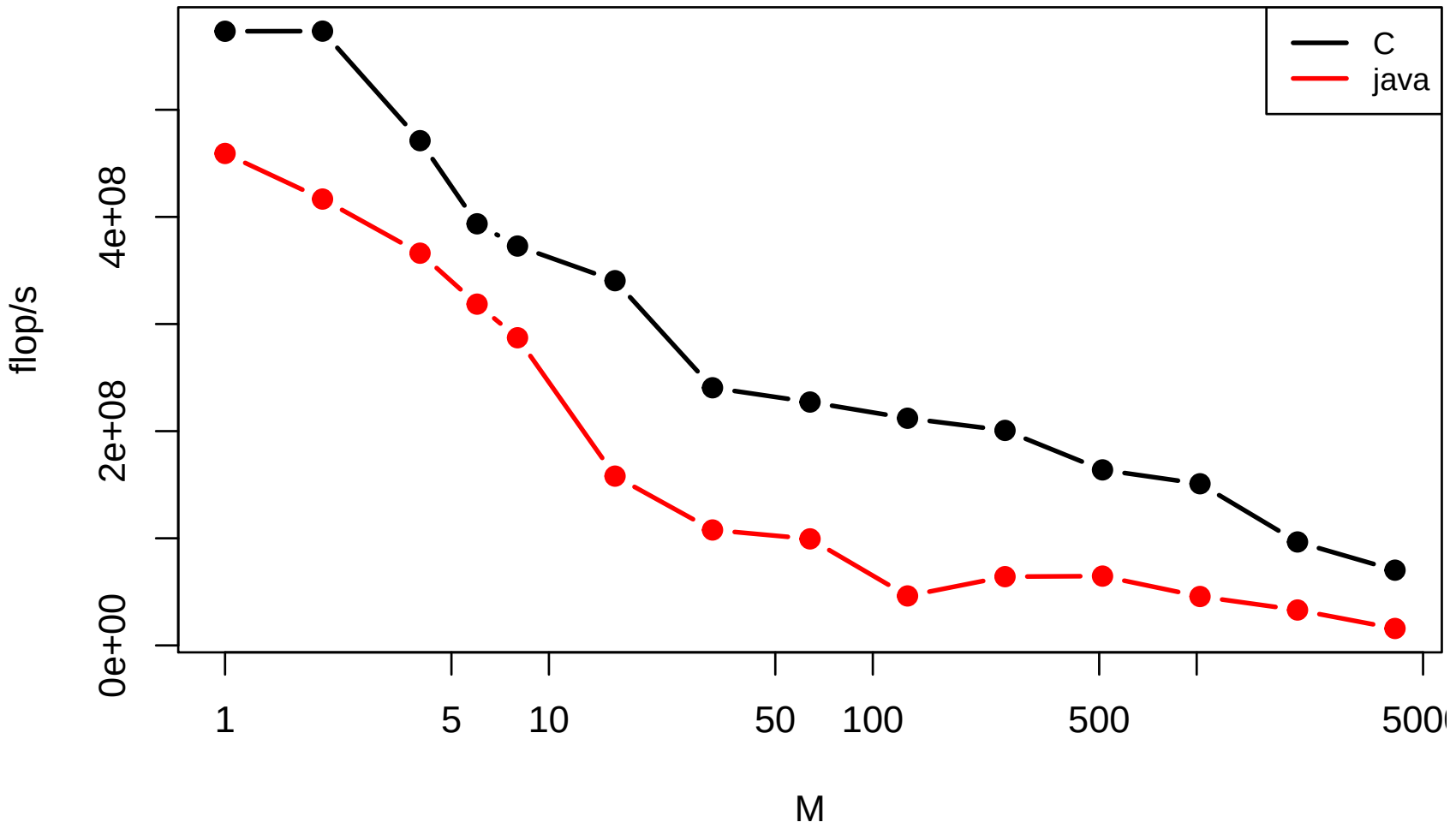


Sur un Core 2 Duo P8600 (2.4 GHz)



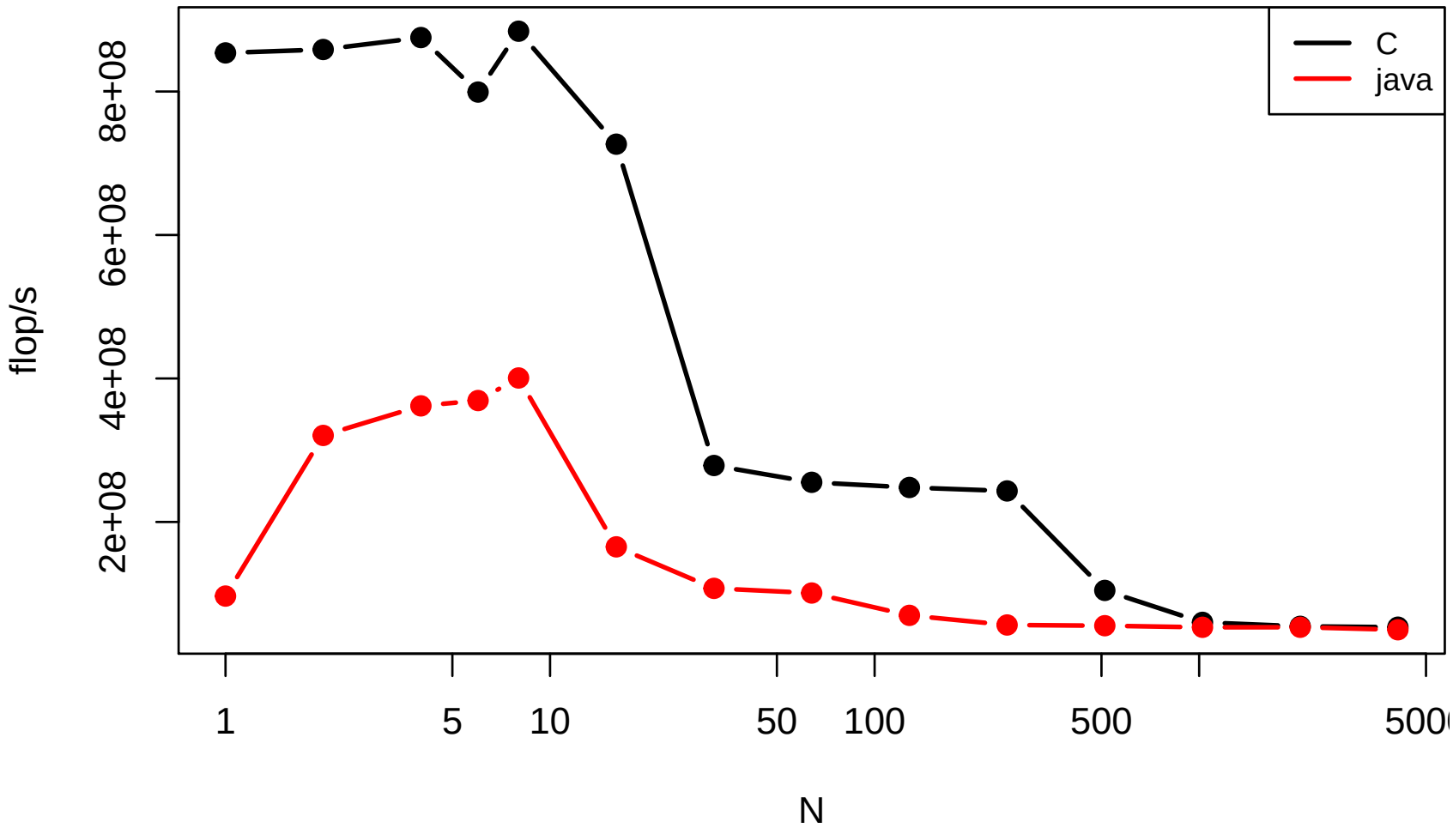
Sur un Core 2 Duo P8600 (2.4 GHz)

C vs Java pour $N = 16384$



Sur un Core 2 Duo P8600 (2.4 GHz)

C vs Java pour $M = 16384$



Sur un Core 2 Duo P8600 (2.4 GHz)

Analyse

- comportement moins erratique
- meilleures performances :
  - pic Java à 449 Mflop/s, pic C à 948 Mflop/s
  - plancher Java à 15 Mflops/s, plancher C à 52 Mflop/s
- mais ça reste une catastrophe : 46 cycles d'horloge par calcul !
- source de la différence : processus de compilation ou hardware ?

Implémentation alternative

- solution : rendre l'accès plus local
- travailler sur `data[i][j]` pour `i` fixé

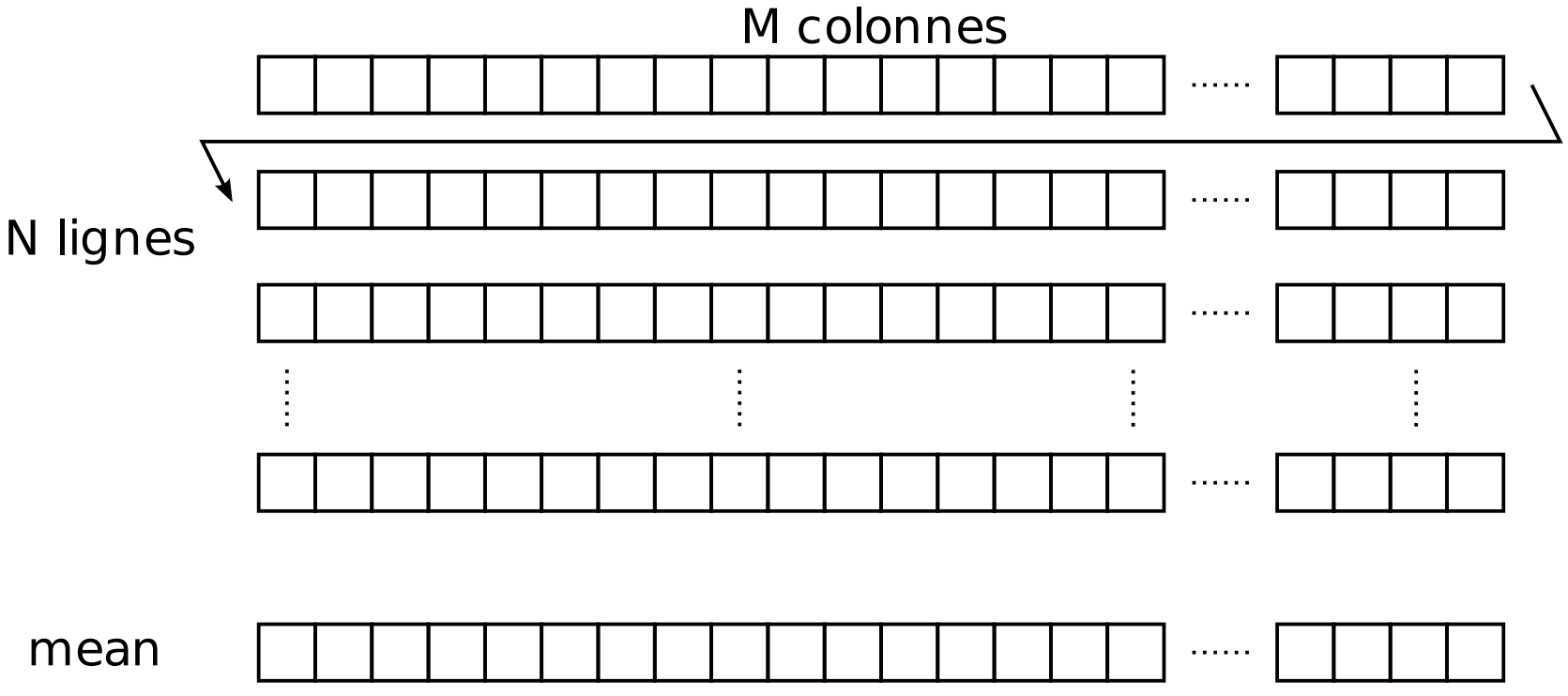
Implémentation alternative

- solution : rendre l'accès plus local
- travailler sur `data[i][j]` pour `i` fixé
- par exemple (en Java) :

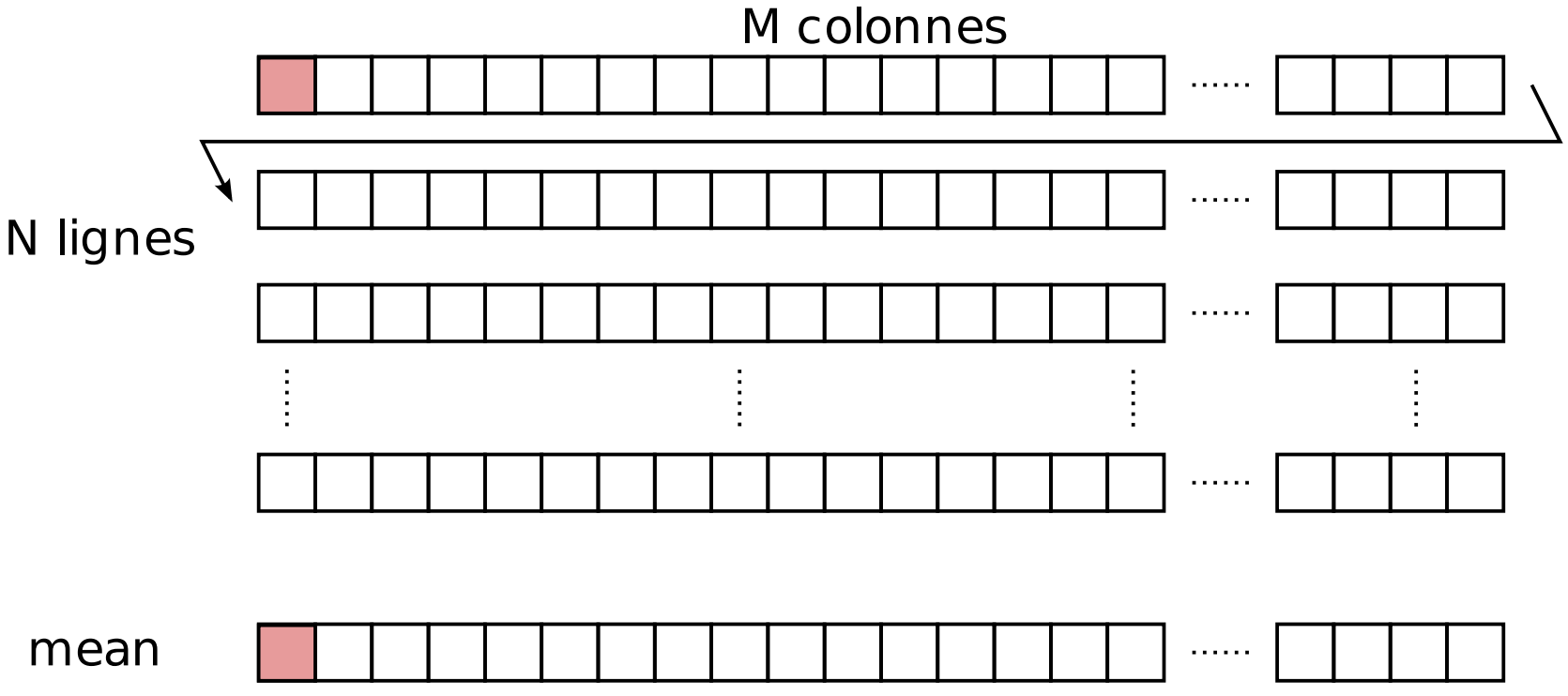
```
double[][] data=new double[N][M];
...
double[] mean=new double[M];
for(int i=0;i<N;i++) {
  for(int j=0;j<M;j++) {
    mean[j]+= data[i][j]; // saut de 8 octets
  }
}
for(int j=0;j<M;j++) {
  mean[j]/=N;
}
```

- remarque : plus d'écritures dans cette solution

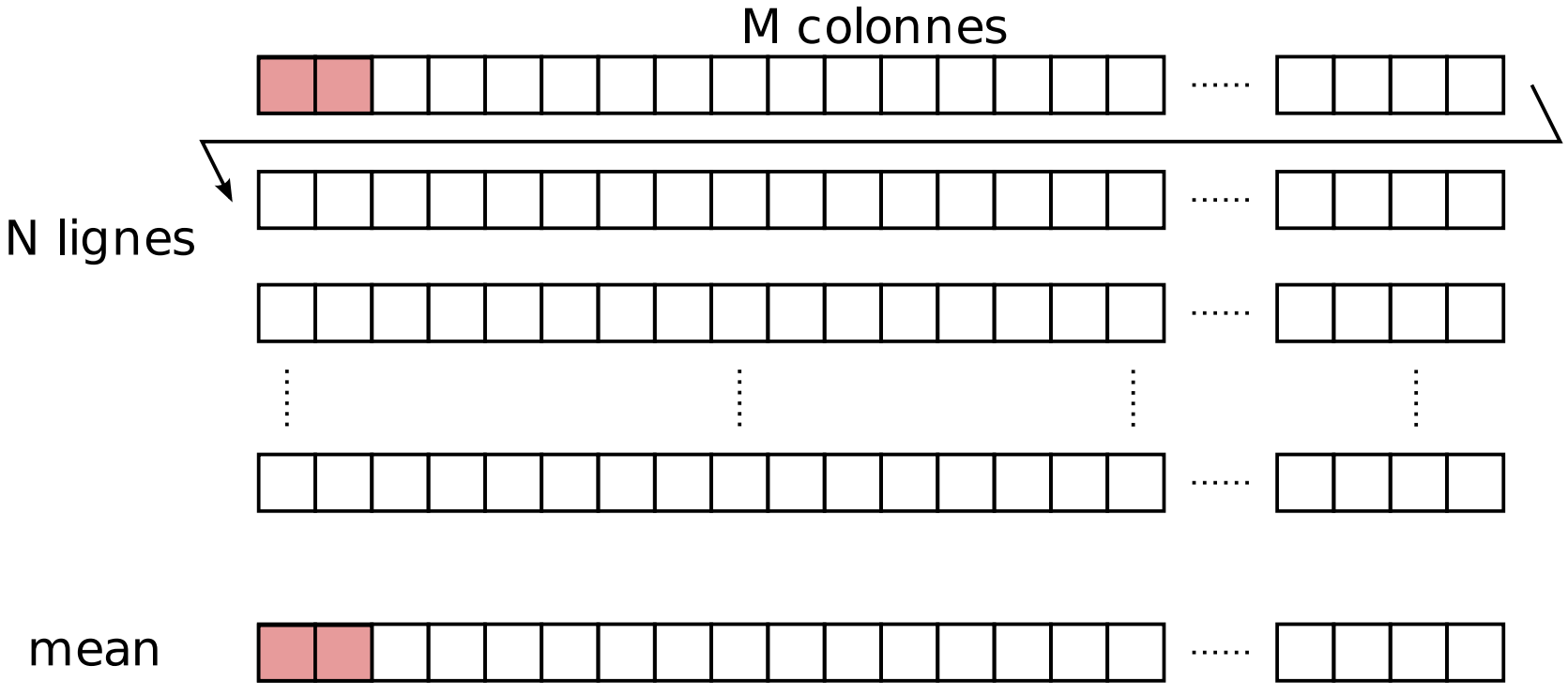
Disposition en mémoire



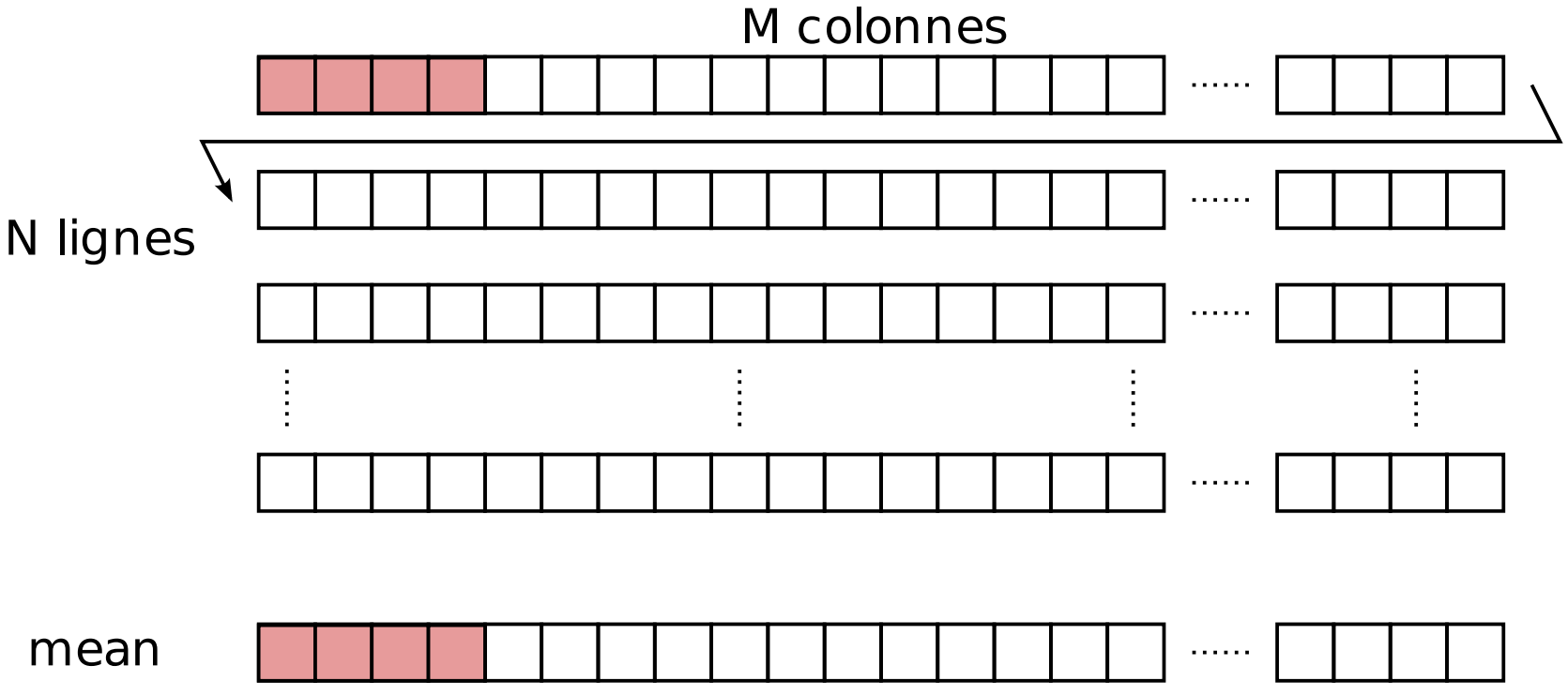
Disposition en mémoire



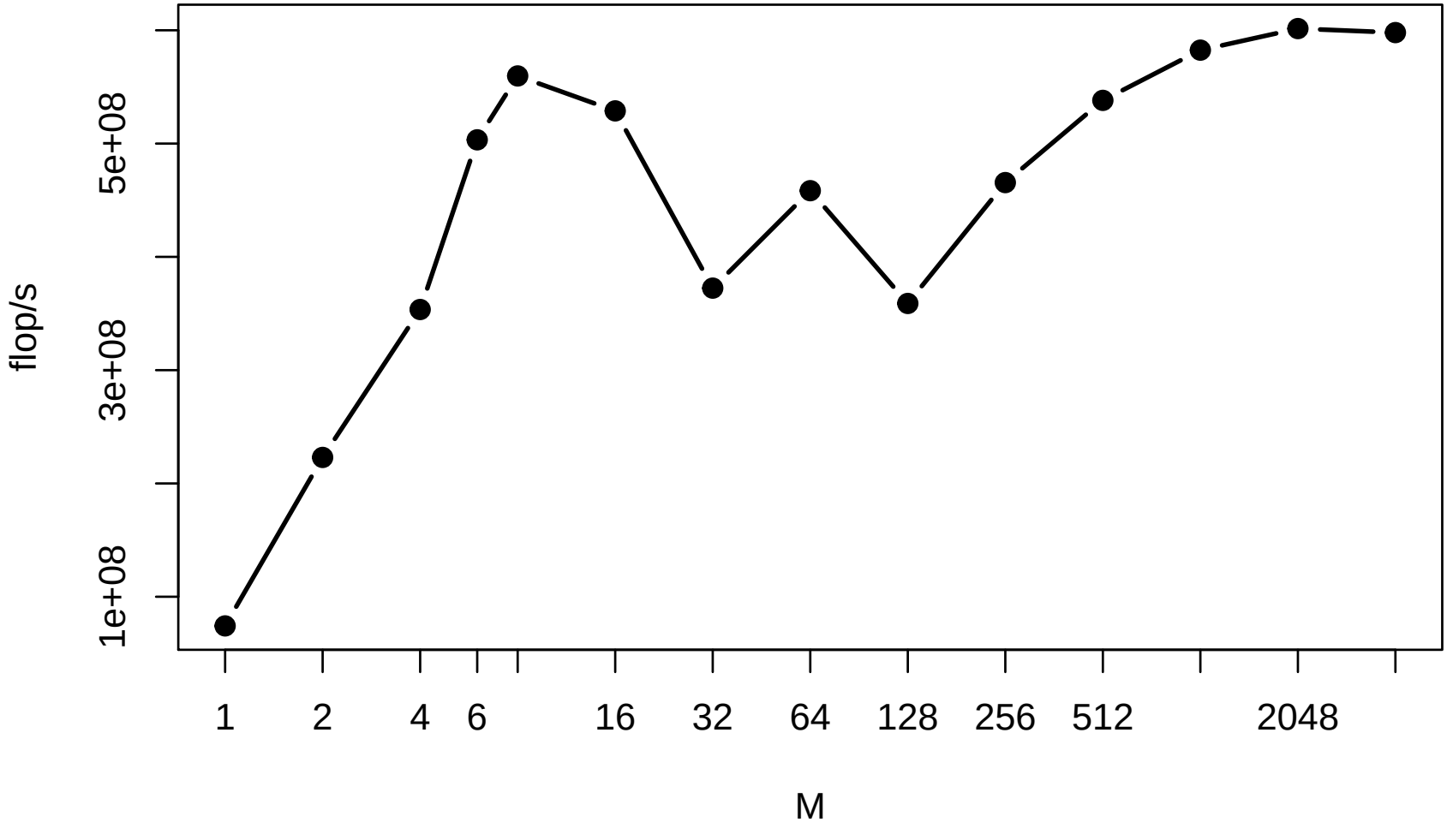
Disposition en mémoire



Disposition en mémoire

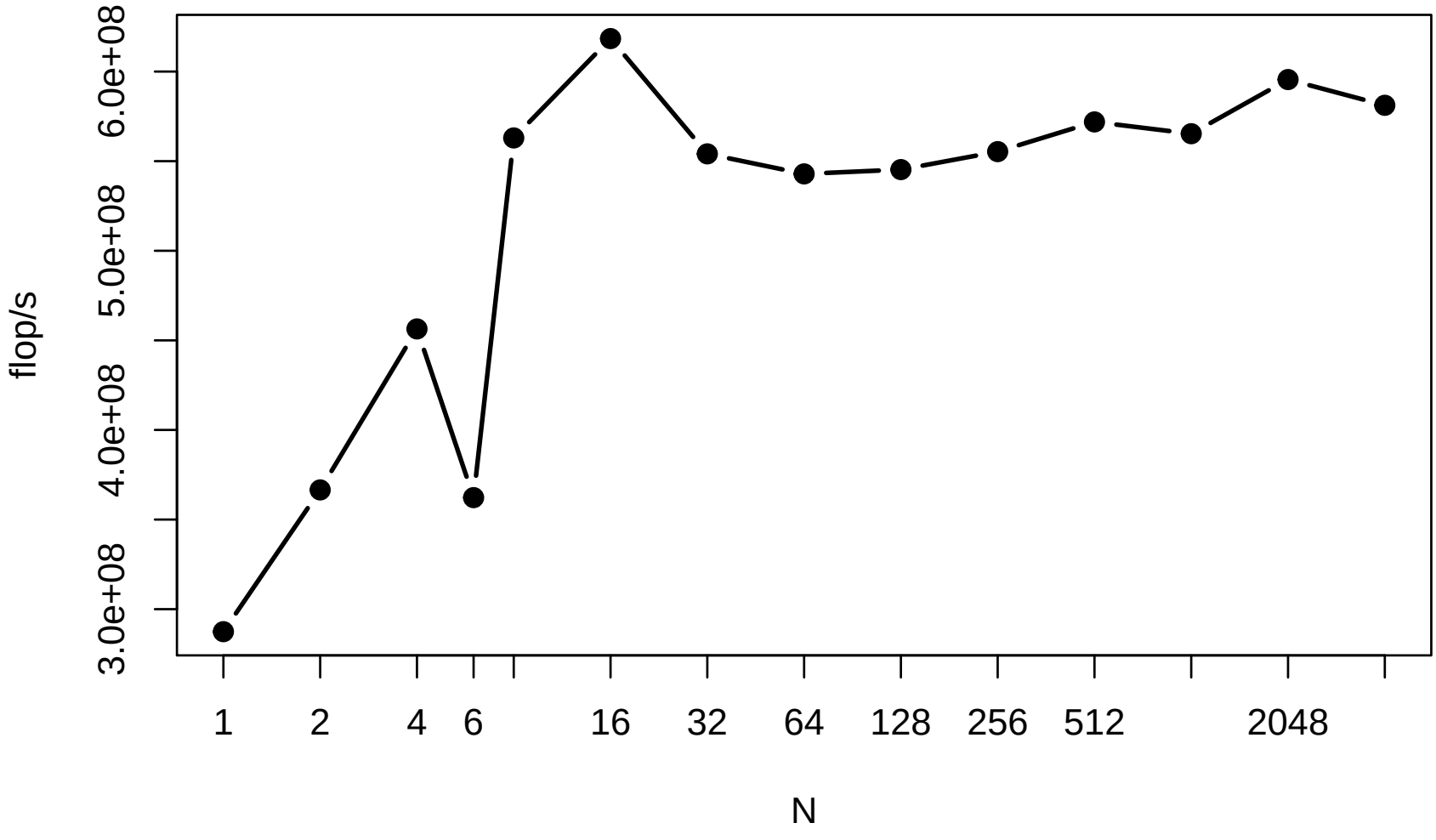


Résultats en Java pour $N = 16384$



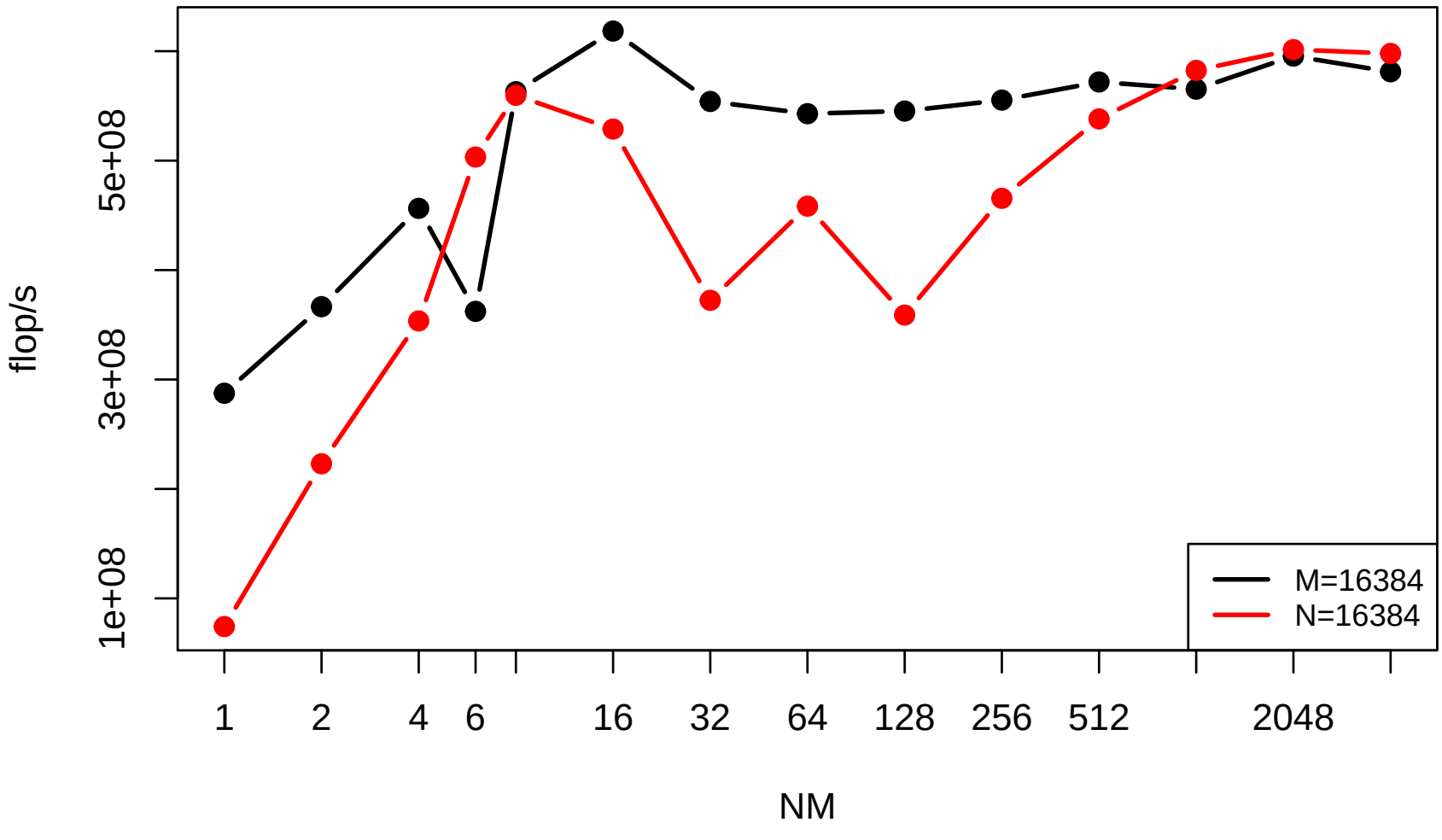
Sur un Core 2 Duo P8600 (2.4 GHz)

Résultats en Java pour $M = 16384$



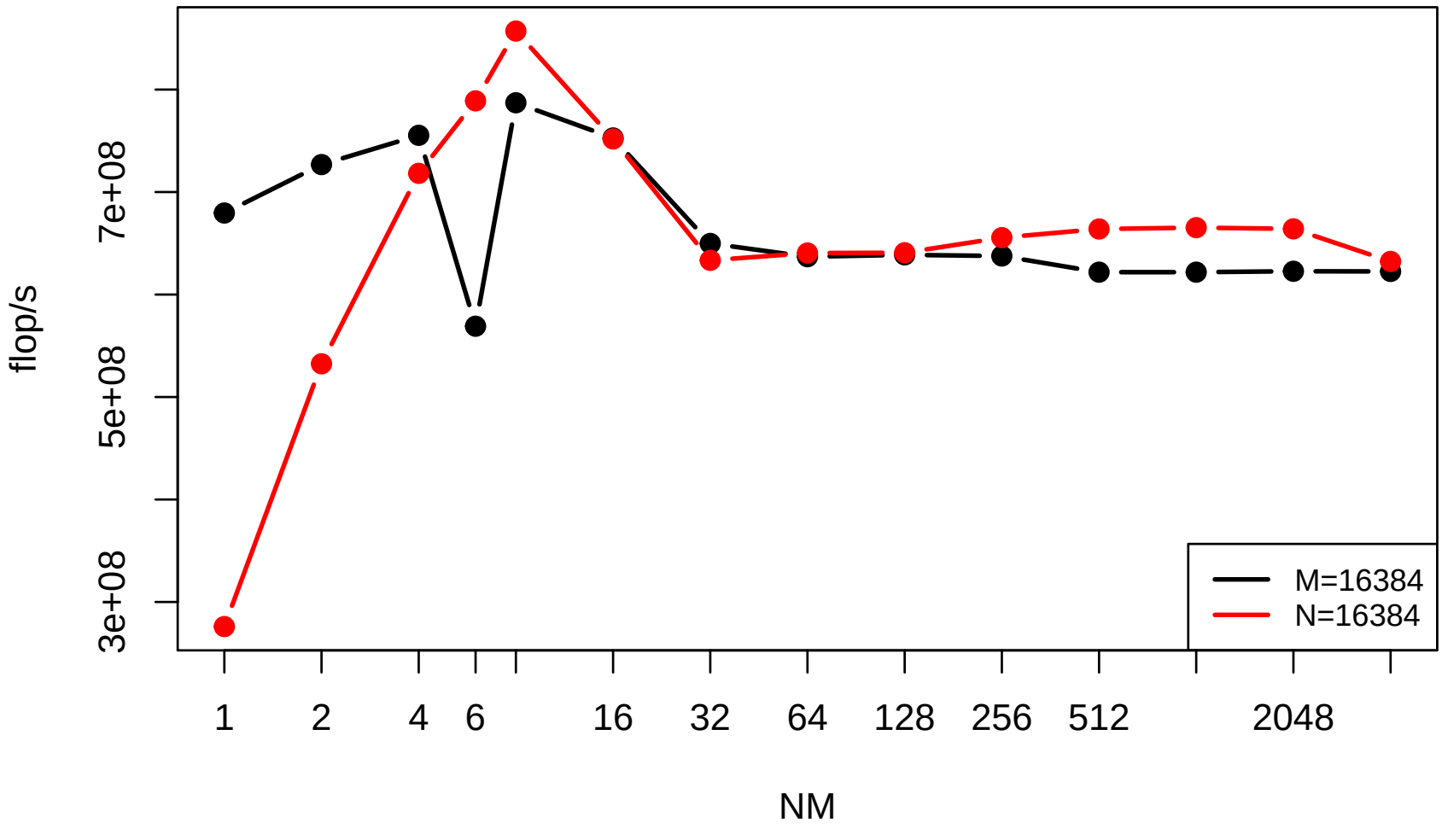
Sur un Core 2 Duo P8600 (2.4 GHz)

Java : *N* et *M*



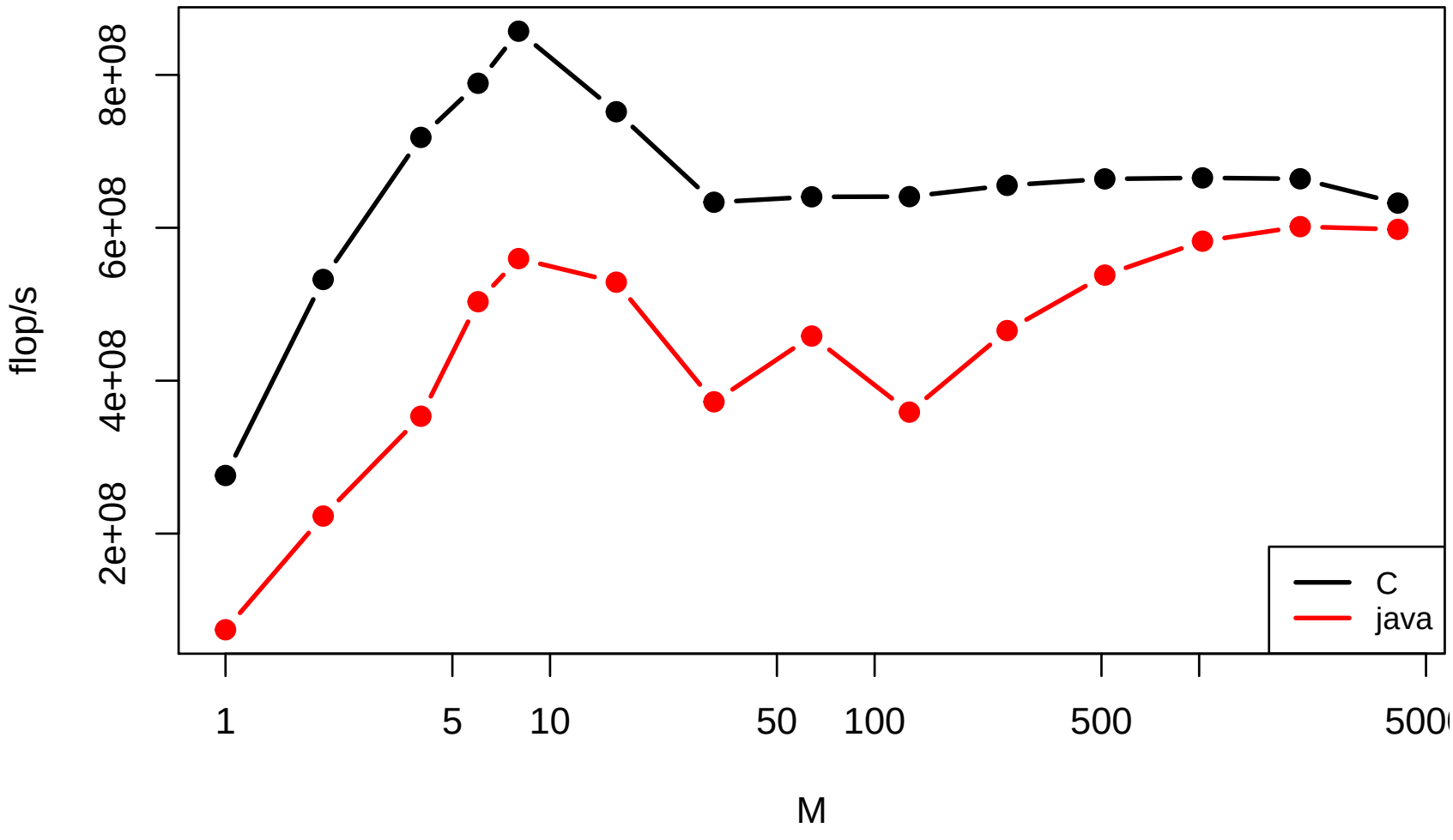
Sur un Core 2 Duo P8600 (2.4 GHz)

Résultats en C



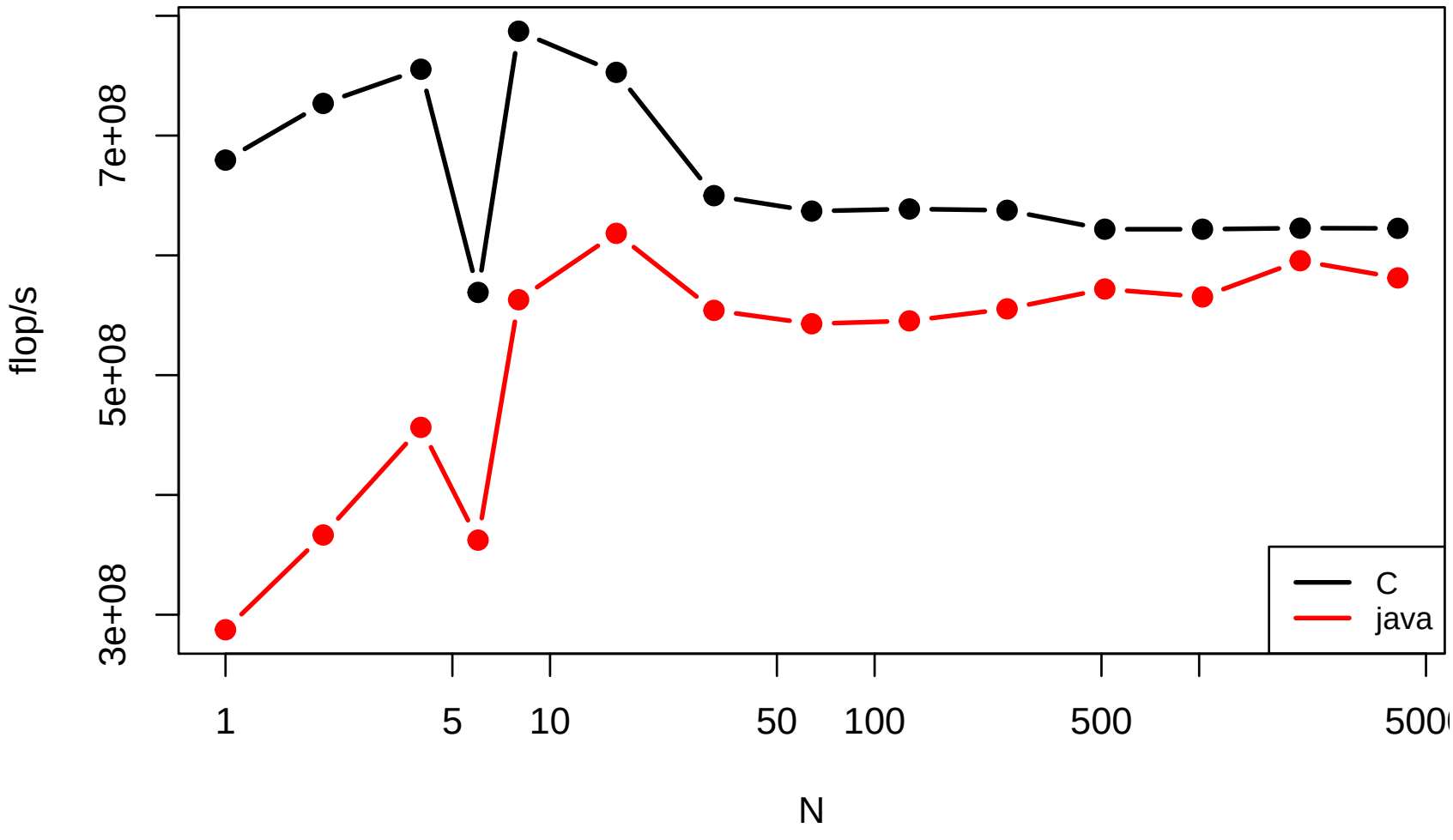
Sur un Core 2 Duo P8600 (2.4 GHz)

C vs Java pour $N = 16384$



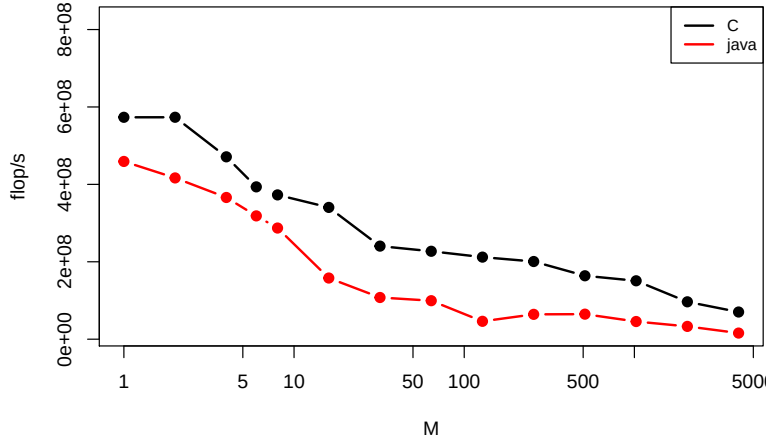
Sur un Core 2 Duo P8600 (2.4 GHz)

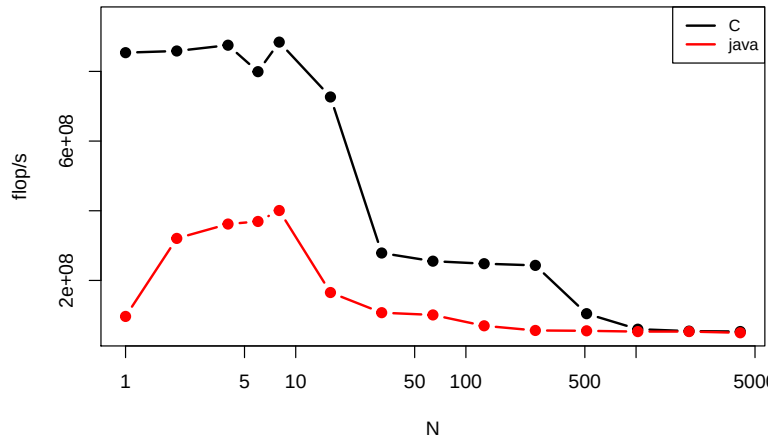
C vs Java pour $M = 16384$

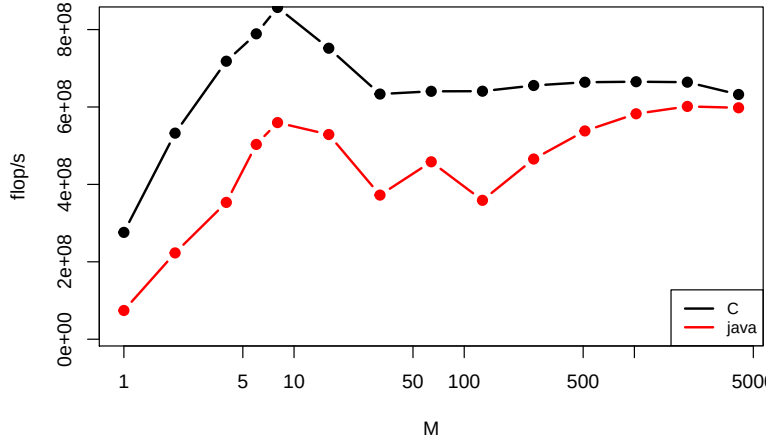


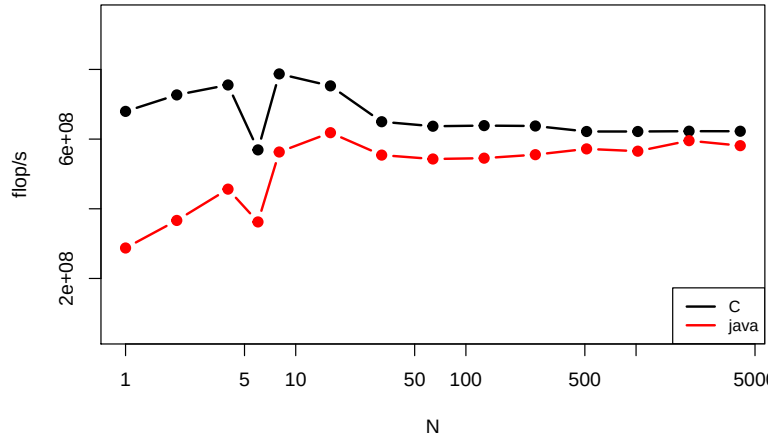
Sur un Core 2 Duo P8600 (2.4 GHz)

Résumé


flop/s
0e+00
2e+08
4e+08
6e+08
8e+08
1
5
10
50
100
500
M
C
java


flop/s
2e+08
6e+08
1
5
10
50
100
500
N
C
java


flop/s
0e+00
2e+08
4e+08
6e+08
8e+08
1
5
10
50
100
500
M
C
java


flop/s
2e+08
6e+08
1
5
10
50
100
500
N
C
java

Résumé

- Solution classique : 4,4 secondes un tableau 16384×4096
- Solution « locale » : 0,11 seconde pour le même tableau ; 38 fois plus rapide
- puissance de crête autour de 600 Mflop/s soit 4 cycles d'horloge par calcul : difficile de faire mieux pour un algorithme qui utilise une seule fois chaque donnée (cf la suite...)
- la première solution est meilleure que la seconde sur les petits jeux de données
- C vs Java :
  - même comportement global
  - plus d'impact sur Java que sur C (*working set* plus grand à cause du JIT et du GC)
  - Java rattrape le C sur du code optimisé

Remarque méthodologique

- mesure du temps :

```
initialisation
démarrage du chronomètre
boucle de répétition
  code à évaluer
fin de la boucle
arrêt du chronomètre
```

- le chronomètre est fiable :
  - appel système POSIX `clock_gettime`
  - horloge `CLOCK_PROCESS_CPUTIME_ID`
  - temps CPU du processus en nanosecondes
- mais évaluation « à chaud » :
  - l'initialisation charge les données dans les caches
  - puis on boucle sur ces données
  - temps de calculs sous estimés dans les cas favorables

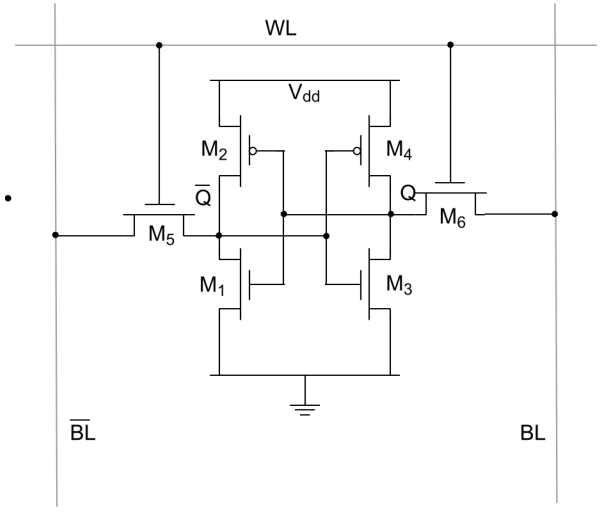
Enseignements

- l'analyse algorithmique classique est insuffisante
- il ne faut pas accéder n'importe comment à la mémoire
- les effets « mémoire » apparaissent même pour une faible quantité de données
- un simple changement d'ordre peut modifier radicalement le temps calcul
- l'algorithme optimal dépend de la taille des données

Plan

Types de mémoire

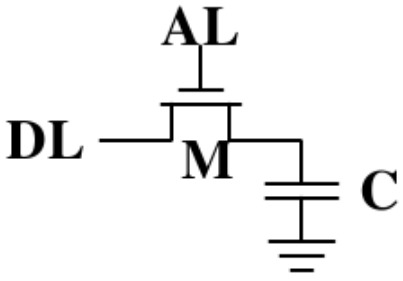
- mémoire statique et mémoire dynamique
- mémoire statique :
  - SRAM
  - 6 transistors par bit
  - très rapide
  - latence très faible
  - chère (occupe beaucoup de place)
  - utilisée dans les caches



http://commons.wikimedia.org/wiki/Image:6t-SRAM-cell.png

Mémoire dynamique

- DRAM
- un transistor et un condensateur par bit
- principe : on ouvre M grâce à AL et on attend la décharge du condensateur en regardant DL
- bon marché (très dense)
- lente (latence importante)
- mécanisme de rafraîchissement (entretien de la charge du condensateur)
- amplification du signal
- utilisée pour la mémoire centrale



Accès

- deux lignes par bit :
  - une ligne (AL) pour ouvrir M
  - une ligne (DL) pour lire C
- impraticable sans partage !

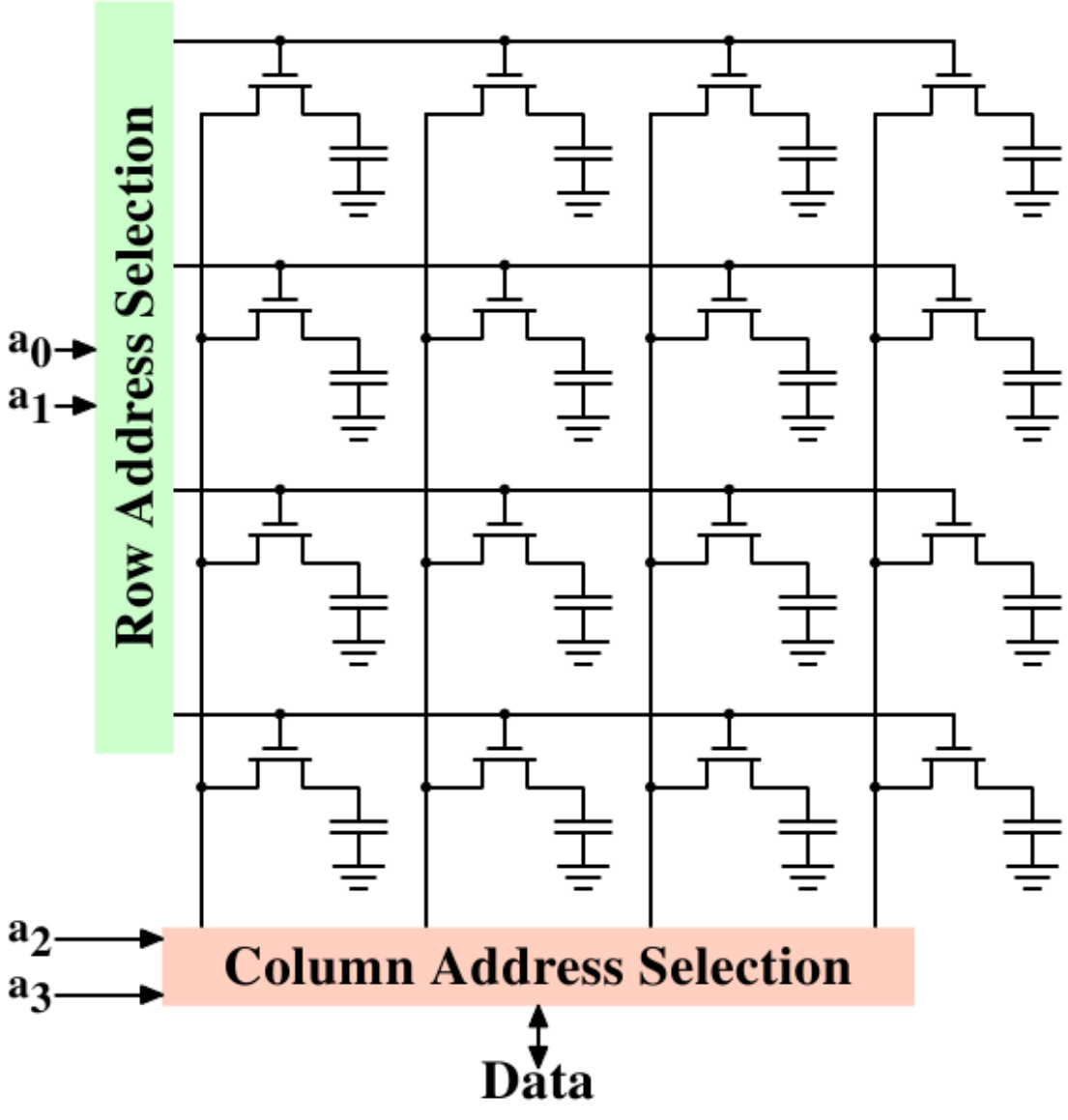
Accès

- deux lignes par bit :
  - une ligne (AL) pour ouvrir M
  - une ligne (DL) pour lire C
- impraticable sans partage !
- organisation en grille :
  - chaque couple transistor–condensateur est à l'intersection d'une ligne et d'une colonne
  - 2^k couples $\longrightarrow 2 \times 2^{\frac{k}{2}}$ lignes
  - $k = 30 \longrightarrow 65536$ lignes au lieu d'un milliard !

Accès

- deux lignes par bit :
  - une ligne (AL) pour ouvrir M
  - une ligne (DL) pour lire C
- impraticable sans partage !
- organisation en grille :
  - chaque couple transistor–condensateur est à l'intersection d'une ligne et d'une colonne
  - 2^k couples $\longrightarrow 2 \times 2^{\frac{k}{2}}$ lignes
  - $k = 30 \longrightarrow 65536$ lignes au lieu d'un milliard !
- nouveaux problèmes :
  - latence : décodage de l'adresse en deux temps, ligne puis colonne
  - consommation : décharge de toute une ligne

Grille mémoire



Principe de fonctionnement

- Algorithme :
  - précharge
  - sélection de la ligne (RAS *Row Address Selection*)
  - sélection de la colonne (CAS *Column Address Selection*)
  - lecture
  - recharge de la ligne

Principe de fonctionnement

- Algorithme :
  - précharge
  - sélection de la ligne (RAS *Row Address Selection*)
  - sélection de la colonne (CAS *Column Address Selection*)
  - lecture
  - recharge de la ligne
- *Timings* t_{CL}-t_{RCD}-t_{RP}-t_{RAS} :
  - t_{CL} : nombre de cycles entre la CAS et la lecture (typiquement 2 à 3)
  - t_{RCD} : nombre de cycles entre la RAS et la CAS (typiquement 2 à 4)
  - t_{RP} : nombre de cycles pour la précharge (typiquement 2 à 4)
  - t_{RAS} : nombre de cycles entre deux précharges (typiquement de 5 à 8)
- Latence la pire : $\max(t_{RP} + t_{RCD} + t_{CL}, t_{RAS})$

Améliorations

- Tampons :
  - on ajoute un tampon en sortie de n tableaux : n bits « à la fois »
  - augmente la bande passante, mais pas la latence
  - DDR ($n = 2$) : un bit sur le front montant, un bit sur le front descendant
  - DDR2 ($n = 4$) : fréquence de bus doublée
  - DDR3 ($n = 8$) : fréquence de bus quadruplée

Améliorations

- Tampons :
  - on ajoute un tampon en sortie de n tableaux : n bits « à la fois »
  - augmente la bande passante, mais pas la latence
  - DDR ($n = 2$) : un bit sur le front montant, un bit sur le front descendant
  - DDR2 ($n = 4$) : fréquence de bus doublée
  - DDR3 ($n = 8$) : fréquence de bus quadruplée
- Transfert d'un bloc :
  - quand le tableau est en lecture, on peut lire plusieurs mots d'un coup
  - la bande passage est ici la limite :
    - le bus fait un mot de large (64 bits)
    - il tourne 4 fois plus vite que le tableau (en DDR3)
    - il envoie un mot en montant et un mot en descendant
    - au total 8 mots par cycle du tableau
  - transfert séquentiel : bloc de 64 octets

Améliorations

- Transfert multiple :
  - généralisation du transfert de bloc
  - ligne fixée, mais colonne changeante
  - nouveau CAS sans RAS
  - une bonne mémoire est T1 : un nouveau CAS par cycle d'horloge
  - ne supprime pas le t_{CL}, mais permet le fonctionnement en pipeline

Améliorations

- Transfert multiple :
  - généralisation du transfert de bloc
  - ligne fixée, mais colonne changeante
  - nouveau CAS sans RAS
  - une bonne mémoire est T1 : un nouveau CAS par cycle d'horloge
  - ne supprime pas le t_{CL}, mais permet le fonctionnement en pipeline
- Superpositions :
  - CAS et lecture : on lit le résultat d'un CAS pendant que le prochain est pris en compte
  - précharge et lecture : on peut préparer un RAS pendant une lecture

Résumé

- SRAM :
  - utilisée par les caches
  - latence faible :
    - induite par la longueur des lignes dans le CPU
    - latence de 3 cycles CPU pour le cache L1 (Core 2 duo)
    - latence de 14 cycles CPU pour le cache L2
  - faible capacité :
    - L1 données : 32 Ko par coeur (Core 2 duo)
    - L2 : 6 Mo sur un E8x00, partagé entre les coeurs
- DRAM :
  - mémoire centrale
  - capacité élevée : jusqu'à 24 Go pour un Core i7
  - latence élevée : 200 cycles pour un accès « aléatoire »
  - bande passante élevée : jusqu'à un octet tout les deux cycles CPU
  - performances très variables selon le mode d'utilisation

Latence CPU

- latence : nombre de cycles d'horloges utilisés pour exécuter une instruction
  - instructions x86 $\longrightarrow$ micro-instructions
  - une μ-instruction par cycle
  - mais plusieurs unités d'exécution : jusqu'à 6 μ-instructions en parallèle par cycle sur un Core 2

Latence CPU

- latence : nombre de cycles d'horloges utilisés pour exécuter une instruction
  - instructions x86 $\longrightarrow$ micro-instructions
  - une μ-instruction par cycle
  - mais plusieurs unités d'exécution : jusqu'à 6 μ-instructions en parallèle par cycle sur un Core 2
- instructions simples : un cycle (par ex. ADD, SUB, AND, OR)
- latence très variable pour les instructions complexes :
  - FMUL (multiplication flottante) : 5 cycles
  - FADD : 3 cycles
  - FCOS (cosinus) : 97 cycles !

Latence CPU

- latence : nombre de cycles d'horloges utilisés pour exécuter une instruction
  - instructions x86 $\longrightarrow$ micro-instructions
  - une μ-instruction par cycle
  - mais plusieurs unités d'exécution : jusqu'à 6 μ-instructions en parallèle par cycle sur un Core 2
- instructions simples : un cycle (par ex. ADD, SUB, AND, OR)
- latence très variable pour les instructions complexes :
  - FMUL (multiplication flottante) : 5 cycles
  - FADD : 3 cycles
  - FCOS (cosinus) : 97 cycles !
- la latence du CPU est partiellement masquée par le modèle d'exécution parallèle (et avec pipeline) : 5 unités d'exécution sur un Core 2 Duo

Caches

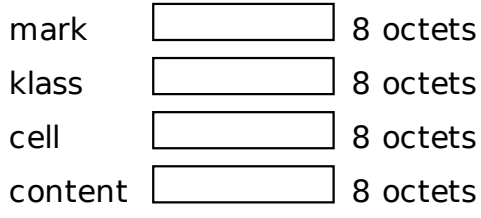
- Masquer la latence de la DRAM grâce à de la SRAM
- Principe sous-jacent :
  - ensemble de travail (*working set*)
  - à un instant donné, le CPU ne travaille que sur une petite portion de la mémoire
  - fortement lié à une notion de localité
- Structure hiérarchique :
  - les registres : latence nulle (en gros)
  - le cache L1 : latence très faible mais très petit
  - le cache L2 : latence faible et plus gros
  - etc.
- Chargement par ligne de cache (par ex. 64 octets par ligne) et donc transfert efficace depuis la mémoire lente

Plan

Latence perçue

- programme simple : parcours d'une liste chaînée
- objet de base (Java) :

```
public class Cell {
  public Cell next;
  public long content;
}
```



- occupe 32 octets sur une machine 64 bits
- allocation (devrait être locale) :

```
Cell[] cells = new Cell[N];
for (int i = 0; i < N; i++) {
  cells[i] = new Cell();
}
```

Parcours

- le programme lui même est un parcours :

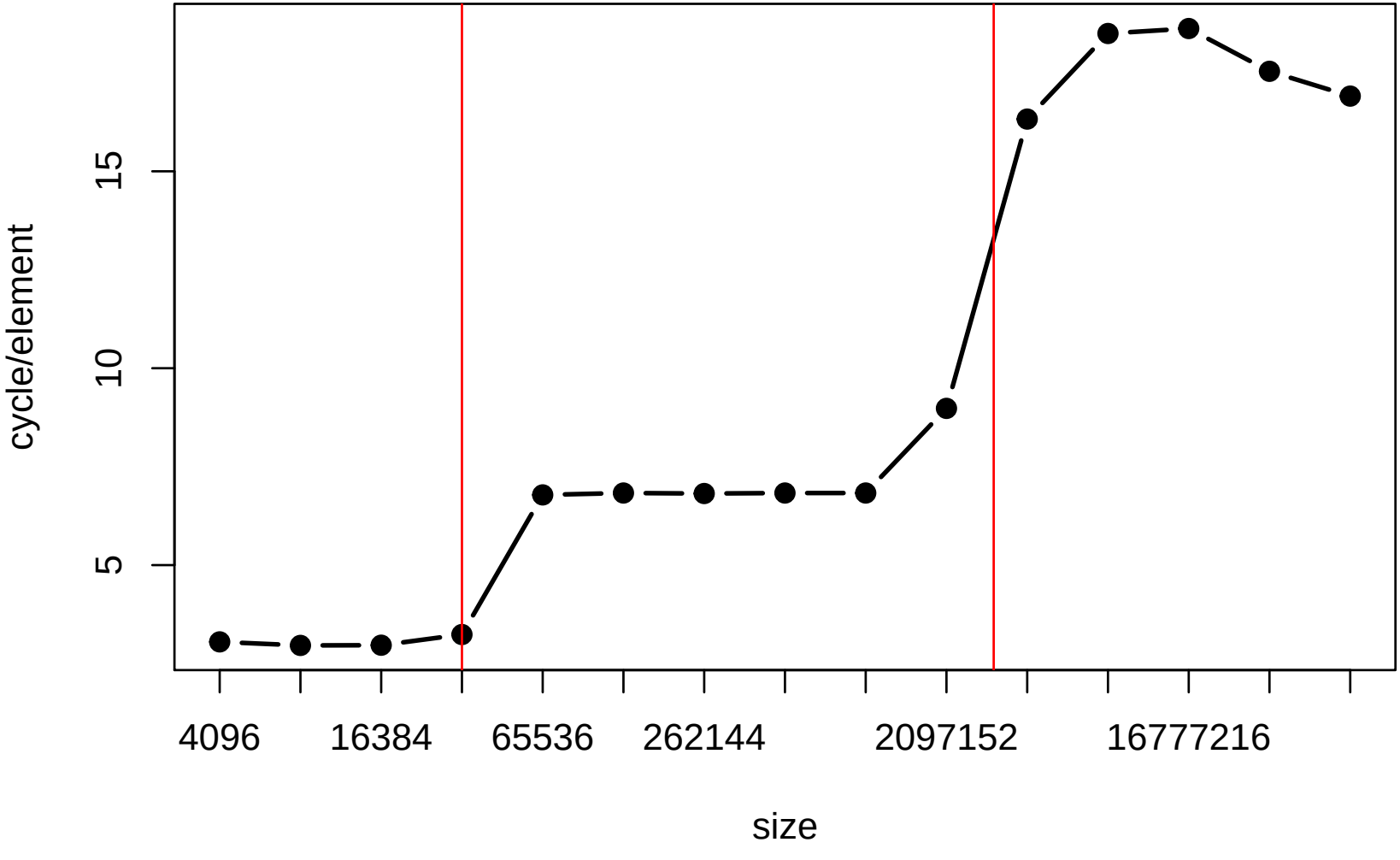
```
Cell current = cells[0];
while (current != null) {
  current = current.next;
}
```

- après établissement des liens ordonnés :

```
for (int i = 1; i < N; i++) {
  cells[i - 1].next = cells[i];
}
```

- attention : on répète de nombreuses fois le parcours

Résultats



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Analyse

- Java n'est pas si nul
- Trois régimes clairs :
  - dans le cache L1 (32 Ko)
  - dans le cache L2 (3 Mo)
  - dans la mémoire principale
- Le cache L2 est partagé (données, code et coeurs) : la transition peut être moins nette

Analyse

- Java n'est pas si nul
- Trois régimes clairs :
  - dans le cache L1 (32 Ko)
  - dans le cache L2 (3 Mo)
  - dans la mémoire principale
- Le cache L2 est partagé (données, code et coeurs) : la transition peut être moins nette
- si le working set (les données) entre dans le cache L1 :
  - une boucle de "préchauffe" : latence importante à chaque accès mémoire
  - puis tout le reste du programme s'exécute "dans le cache" : latence de 3 (valeur annoncée par le constructeur)

Analyse

- si le working set entre dans le cache L2 (mais pas L1) :
  - une boucle de préchauffe pour le cache L2
  - les données ne sont *jamais* dans le cache L1 : on paye la latence de L2
  - on devrait obtenir une latence de 14 et non pas de 7...

Analyse

- si le working set entre dans le cache L2 (mais pas L1) :
  - une boucle de préchauffe pour le cache L2
  - les données ne sont *jamais* dans le cache L1 : on paye la latence de L2
  - on devrait obtenir une latence de 14 et non pas de 7...
- chargement par *ligne de cache*
  - une ligne = 64 octets = 2 cellules
  - transfert en une fois de L2 vers L1 : on paye la latence de 14 une fois sur deux

Analyse

- si le working set entre dans le cache L2 (mais pas L1) :
  - une boucle de préchauffe pour le cache L2
  - les données ne sont *jamais* dans le cache L1 : on paye la latence de L2
  - on devrait obtenir une latence de 14 et non pas de 7...
- chargement par *ligne de cache*
  - une ligne = 64 octets = 2 cellules
  - transfert en une fois de L2 vers L1 : on paye la latence de 14 une fois sur deux
  - mais on paye aussi L1, soit $(14 + 3)/2 = 8.5$ cycles en moyenne...

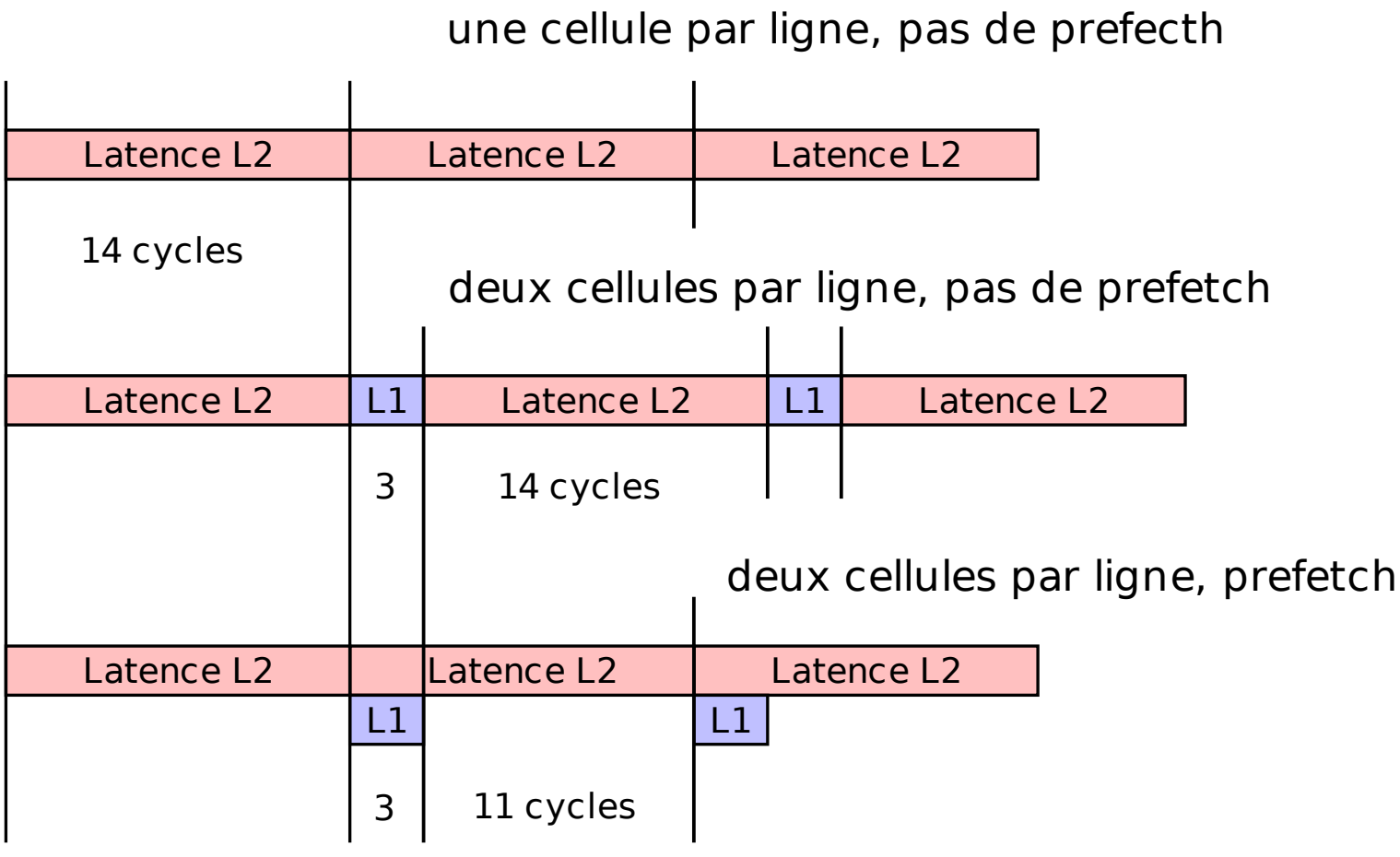
Analyse

- si le working set entre dans le cache L2 (mais pas L1) :
  - une boucle de préchauffe pour le cache L2
  - les données ne sont *jamais* dans le cache L1 : on paye la latence de L2
  - on devrait obtenir une latence de 14 et non pas de 7...
- chargement par *ligne de cache*
  - une ligne = 64 octets = 2 cellules
  - transfert en une fois de L2 vers L1 : on paye la latence de 14 une fois sur deux
  - mais on paye aussi L1, soit $(14 + 3)/2 = 8.5$ cycles en moyenne...
- amélioration par préchargement (*prefetching*)

Préchargement

- chargement anticipé des lignes suivantes : le CPU fait l'hypothèse qu'on travaille de façon séquentielle
- L2 vers L1 : détection après deux accès séquentiels
- gain : recouvrement transfert et calcul
- dans l'exemple :
  - transfert de deux cellules : latence de 14
  - accès au cache L1 pour la deuxième cellule : latence de 3
  - transfert des deux cellules suivante pendant l'accès au cache L1 : latence additionnelle de 11
  - en régime de croisière : $(11 + 3)/2 = 7$ cycles
- remarque : actif même dans la première exécution du parcours des données

Latences


une cellule par ligne, pas de prefecth
Latence L2
Latence L2
Latence L2
14 cycles
deux cellules par ligne, pas de prefetch
Latence L2
L1
Latence L2
L1
Latence L2
3
14 cycles
deux cellules par ligne, prefetch
Latence L2
Latence L2
Latence L2
L1
L1
3
11 cycles

Mémoire principale

- rappel : latence de l'ordre de 200 cycles d'horloge, mais transfert par bloc
- ici : environ 18 cycles par cellule

Mémoire principale

- rappel : latence de l'ordre de 200 cycles d'horloge, mais transfert par bloc
- ici : environ 18 cycles par cellule
- préchargement :
  - de la mémoire vers L2
  - même principe que L2 vers L1, mais avec prise en compte d'un éventuel décalage (*stride*) : jusqu'à 8 lignes de cache d'écart entre deux accès
  - transfert séquentiel : latence de un cycle mémoire ($\simeq$ 15 cycles CPU) entre deux blocs de 8 octets
  - mécanisme assez sophistiqué : fonctionne très bien dans cet exemple

Mémoire principale

- rappel : latence de l'ordre de 200 cycles d'horloge, mais transfert par bloc
- ici : environ 18 cycles par cellule
- préchargement :
  - de la mémoire vers L2
  - même principe que L2 vers L1, mais avec prise en compte d'un éventuel décalage (*stride*) : jusqu'à 8 lignes de cache d'écart entre deux accès
  - transfert séquentiel : latence de un cycle mémoire ($\simeq$ 15 cycles CPU) entre deux blocs de 8 octets
  - mécanisme assez sophistiqué : fonctionne très bien dans cet exemple
- remarque : on obtient 18 cycles/cellule même pour un seul parcours de toutes les données

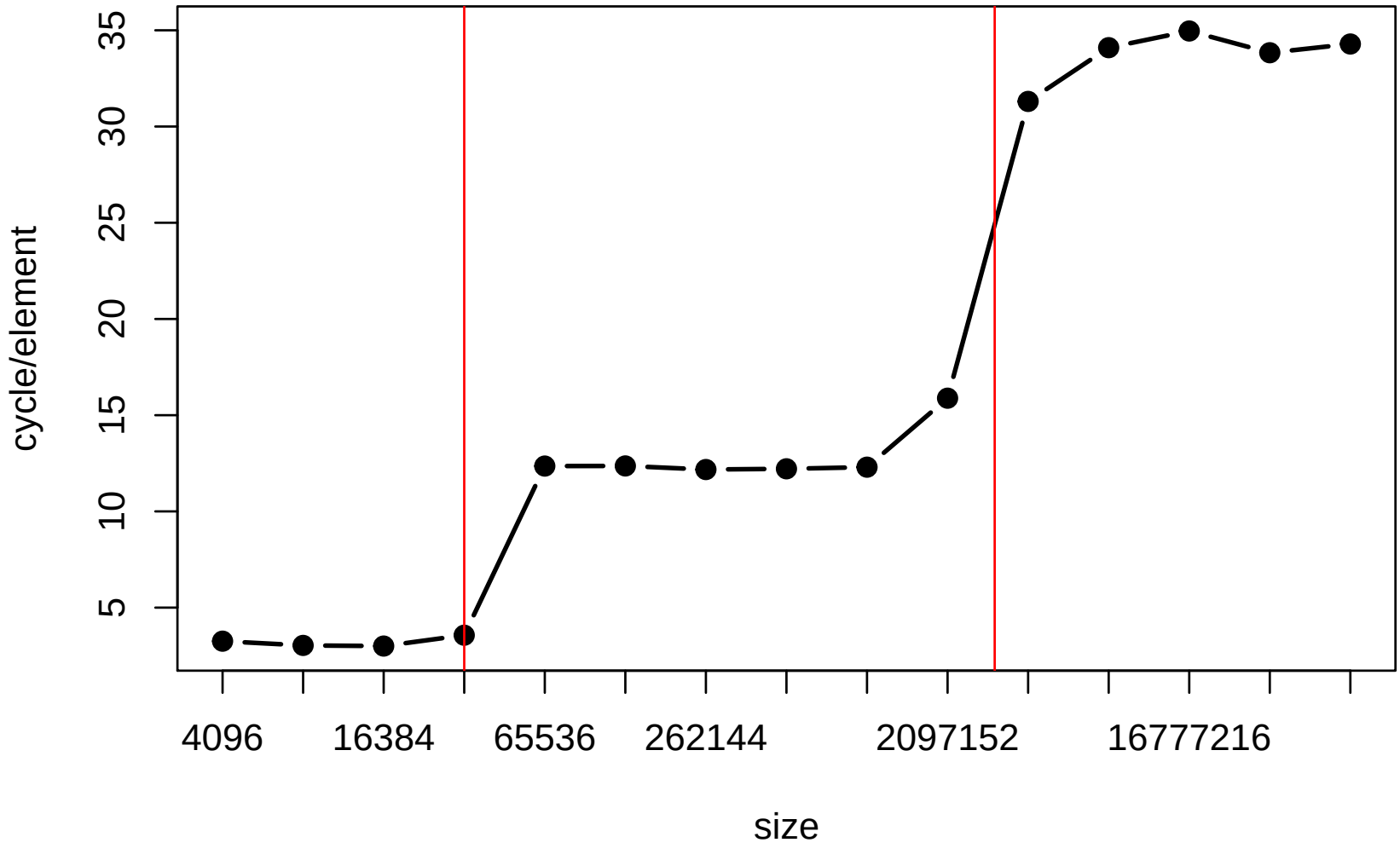
Données espacées

- Que se passe-t-il si on espace les données ?
- Par exemple avec (un peu pénible en Java)

```
public class Cell {
  public Cell next;
  public long content1;
  public long content2;
  public long content3;
  public long content4;
  public long content5;
}
```

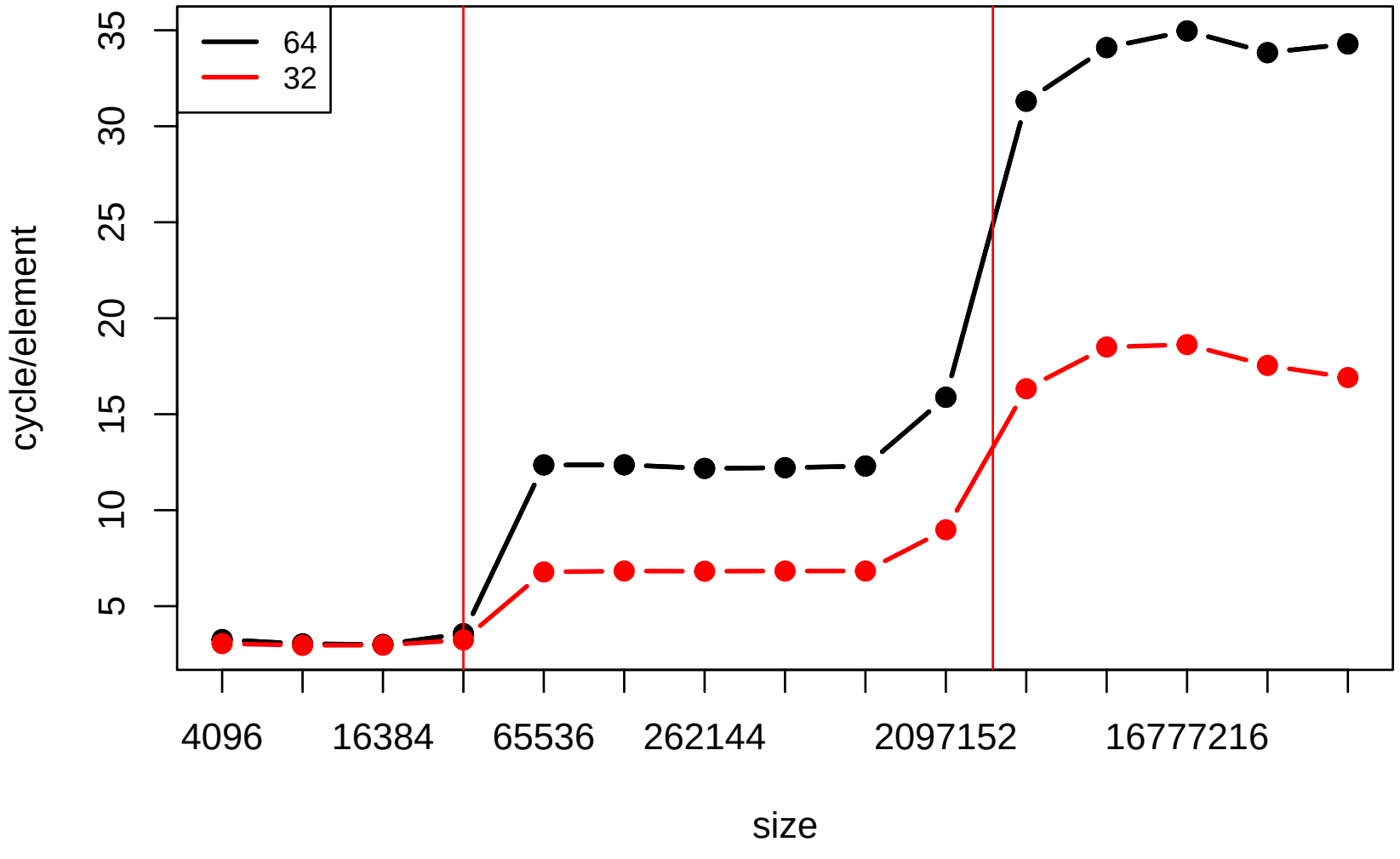
- Chaque cellule pèse 64 octets (une ligne de cache)

Résultats



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Analyse

- Chaque objet utilise maintenant une ligne de cache
- On conserve les trois régimes
- Mais :
  - le *prefetching* L2 dans L1 ne fonctionne plus : le cache L1 ne précharge que la ligne suivante, ici il n'a pas le temps
  - le *prefetching* mémoire dans L2 fonctionne : le CPU détecte accès réguliers avec jusqu'à 8 lignes de cache d'écart (pour les Core)
- On prévoit donc un fonctionnement du *prefetching* L2 jusqu'à 512 octets d'écart entre deux accès

Données plus espacées

- Encore plus d'espace entre les données
- Nouvelle classe

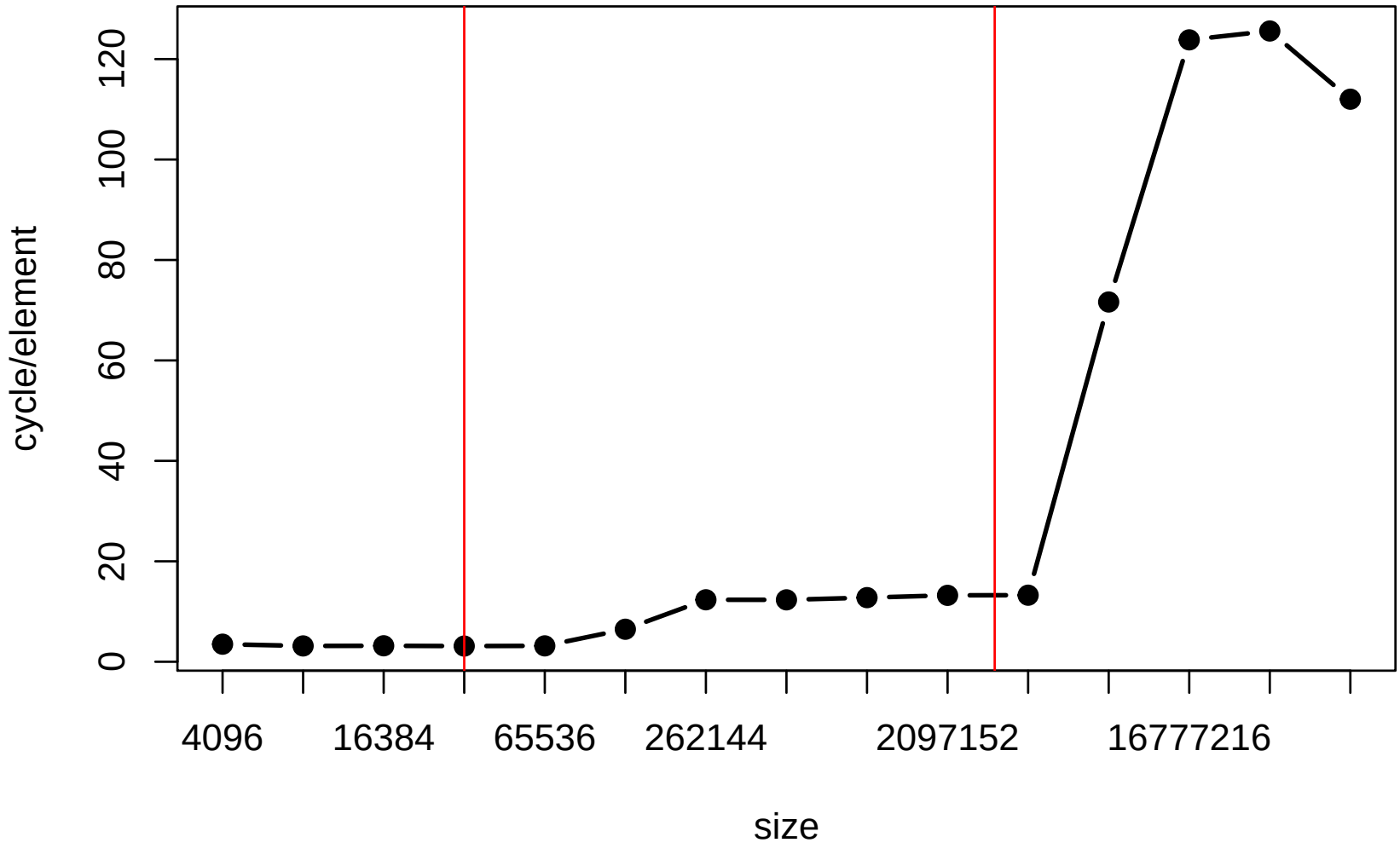
```
public class Cell {
  public Cell next;
  public long[] content;
}
```

- Chaque cellule pèse 32 octets + le tableau
- Allocation « linéaire » :

```
Cell[] cells = new Cell[N];
for (int i = 0; i < N; i++) {
  cells[i] = new Cell(M);
}
```

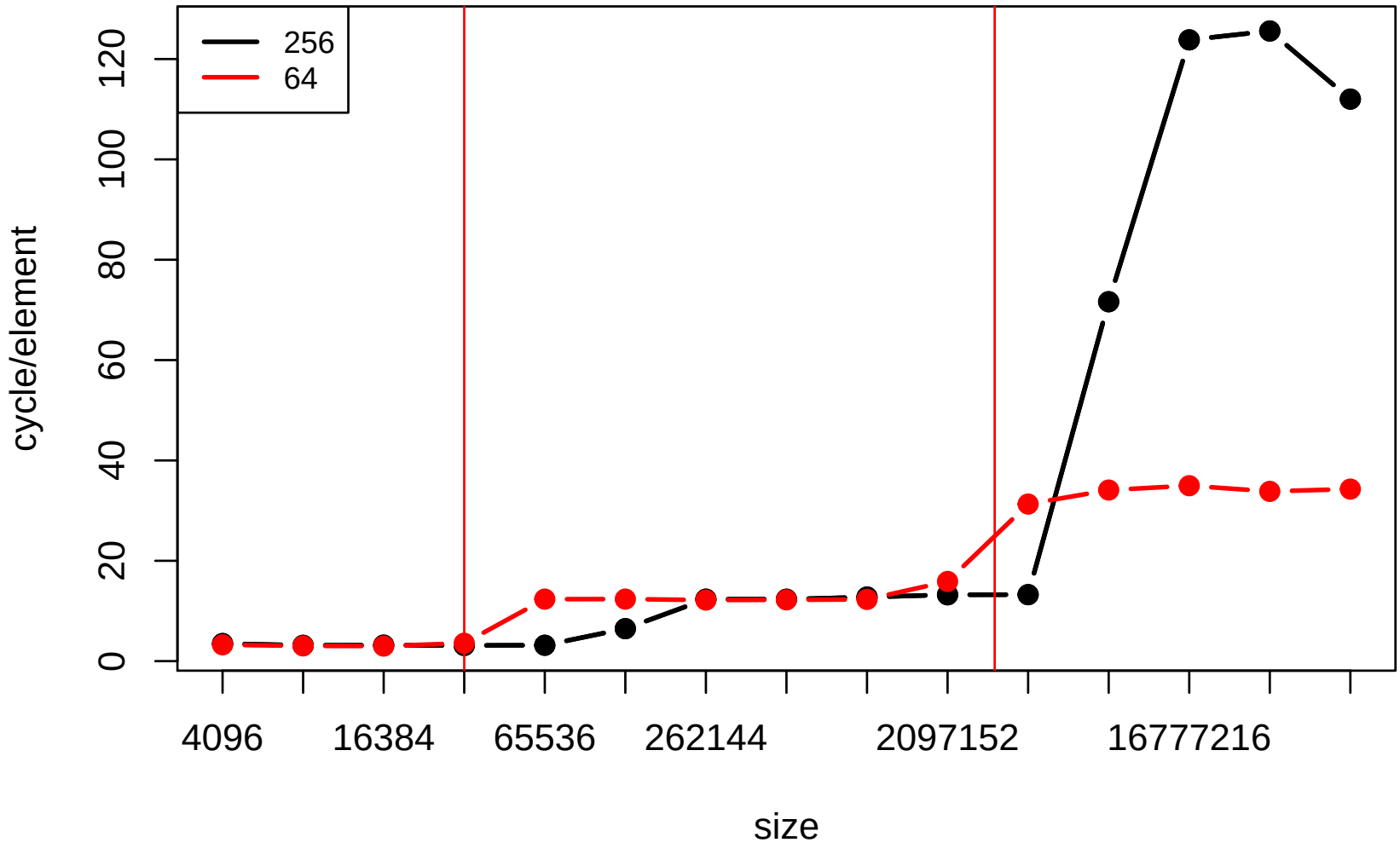
- Attention : en Java, on ne maîtrise pas tout...

Résultats $M = 28$ (256 octets)



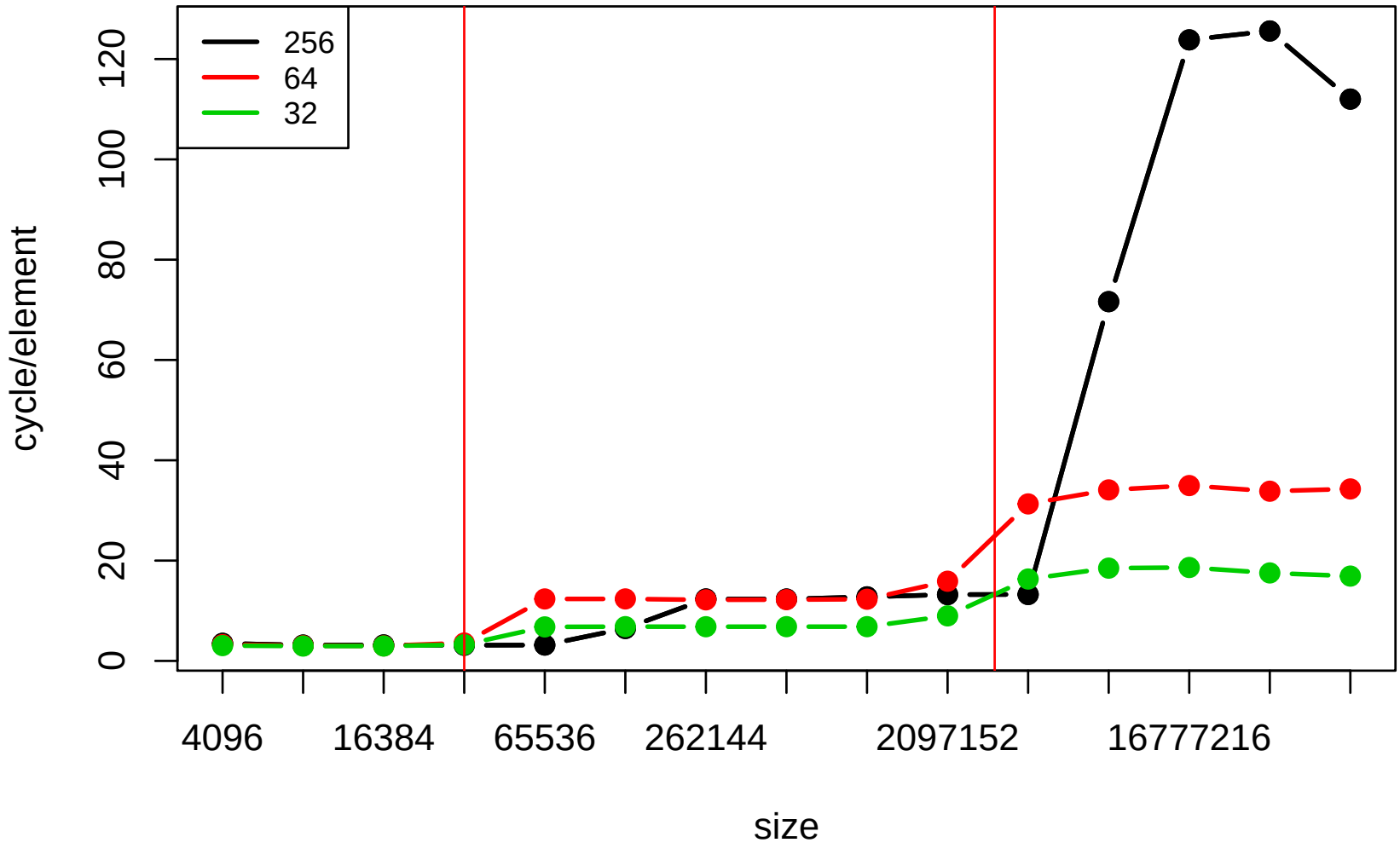
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats $M = 28$ (256 octets)



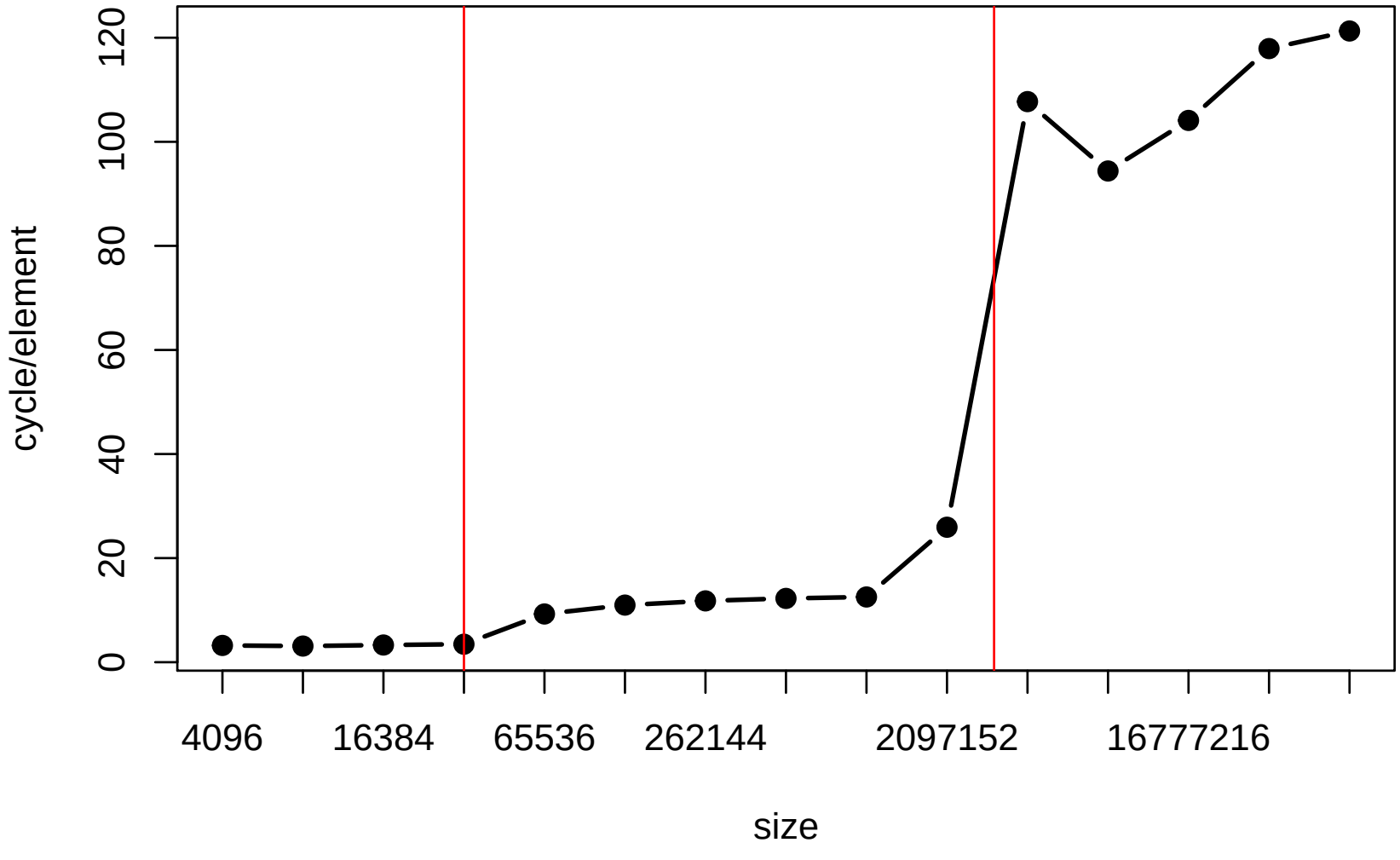
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats $M = 28$ (256 octets)



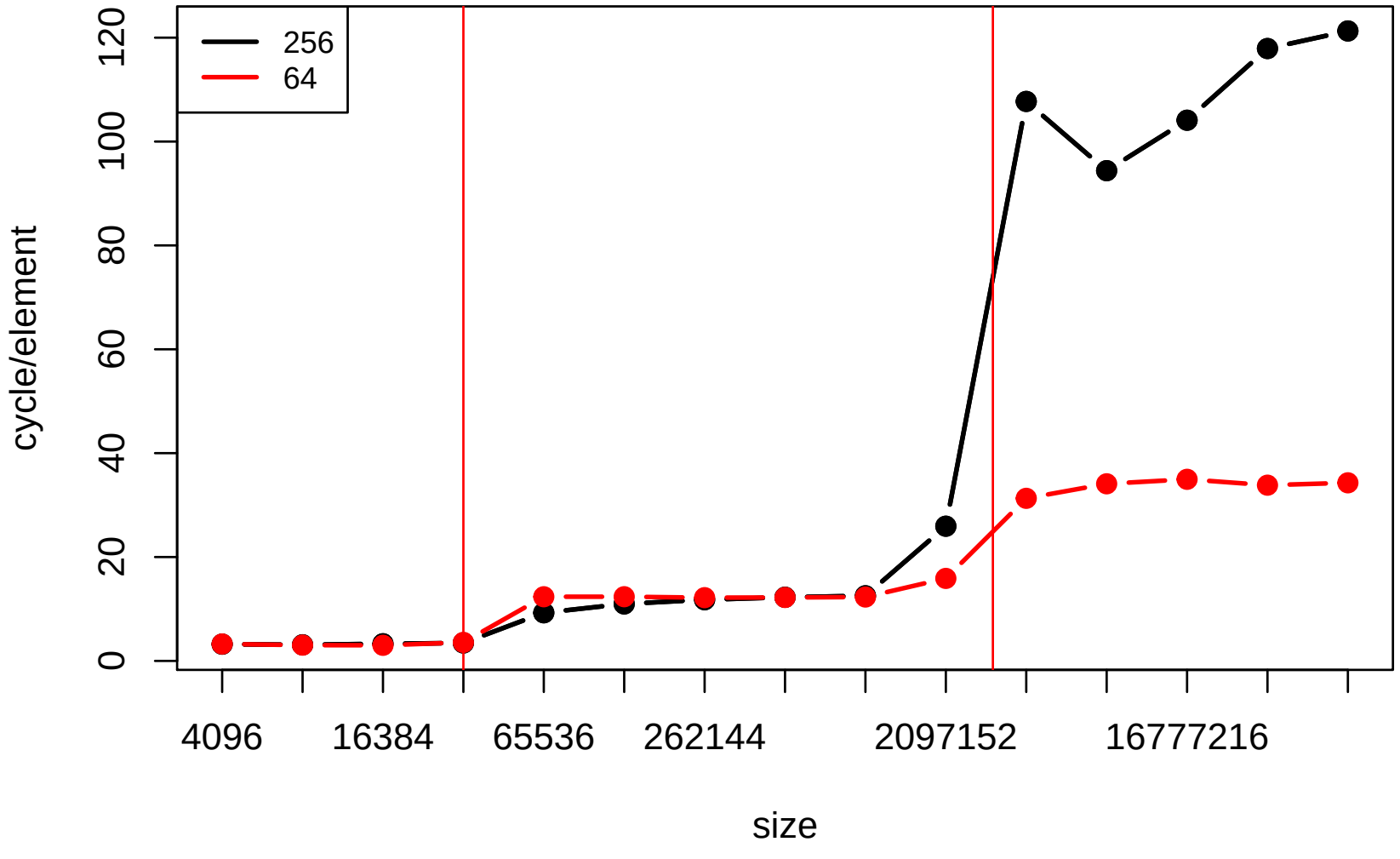
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats 256 octets directs



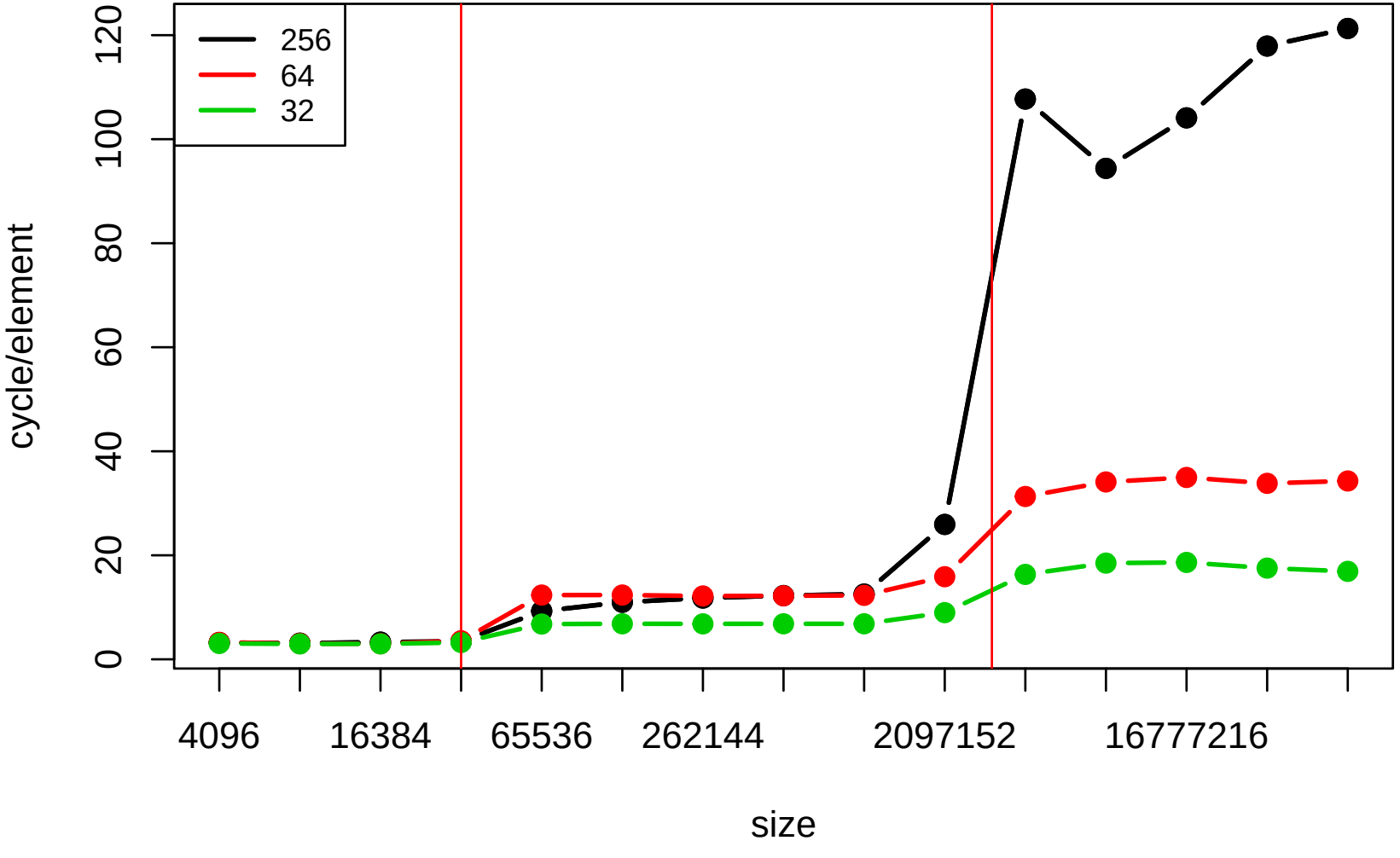
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats 256 octets directs



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats 256 octets directs



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

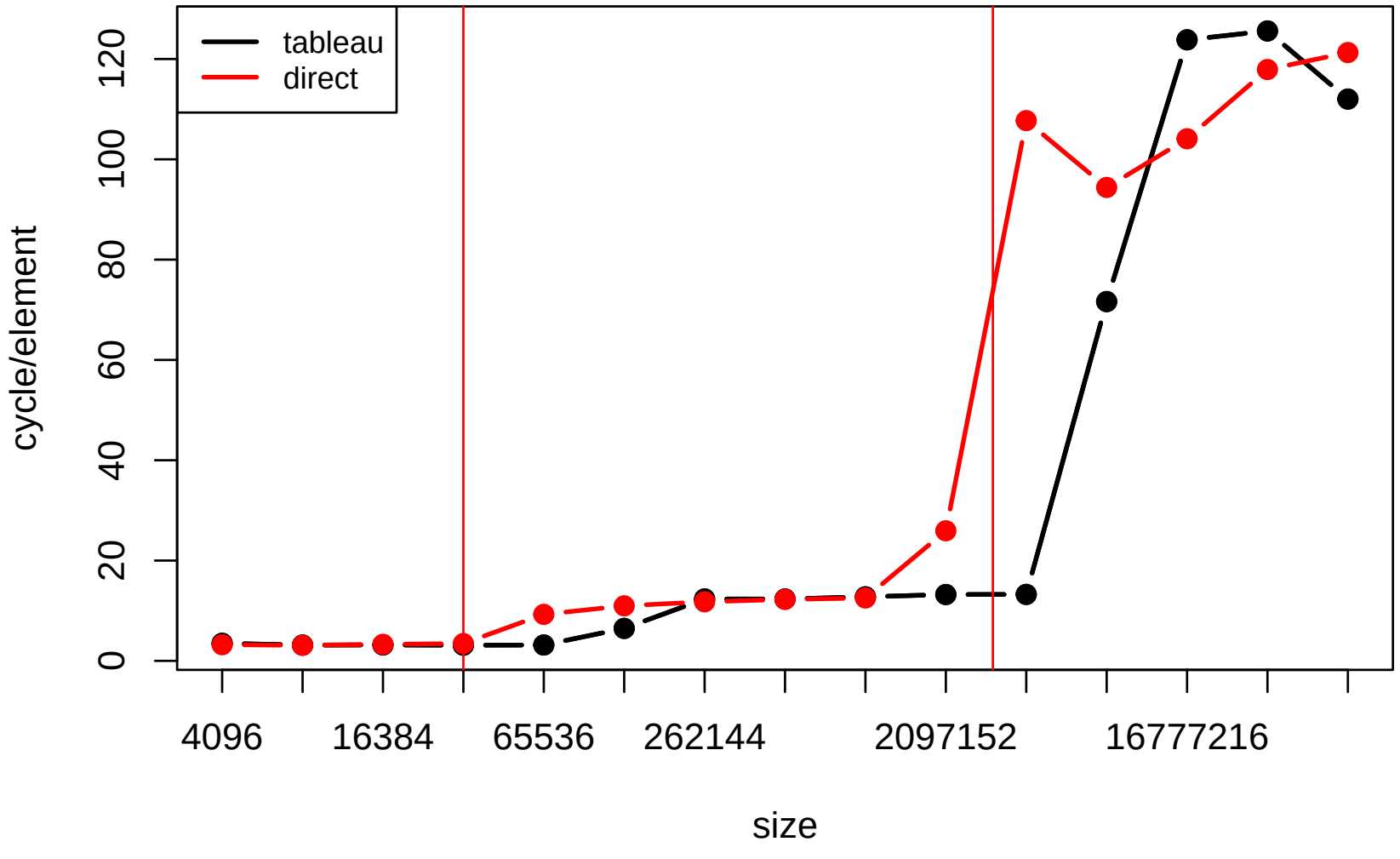
Interprétation

- Quand la cellule occupe 256 octets :
  - comportement classique
  - d'abord L1 (on utilise le début de la cellule seulement)
  - puis L2
  - puis la mémoire avec plus de latence : *prefetching* moins efficace

Interprétation

- Quand la cellule occupe 256 octets :
  - comportement classique
  - d'abord L1 (on utilise le début de la cellule seulement)
  - puis L2
  - puis la mémoire avec plus de latence : *prefetching* moins efficace
- Quand la cellule contient un tableau :
  - comportement plus complexe
  - plus efficace
  - mais on n'accède pas au contenu du tableau !
  - donc tout dépend de l'allocation et des optimisations réalisées au vol par la JVM

Résultats direct et tableau



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

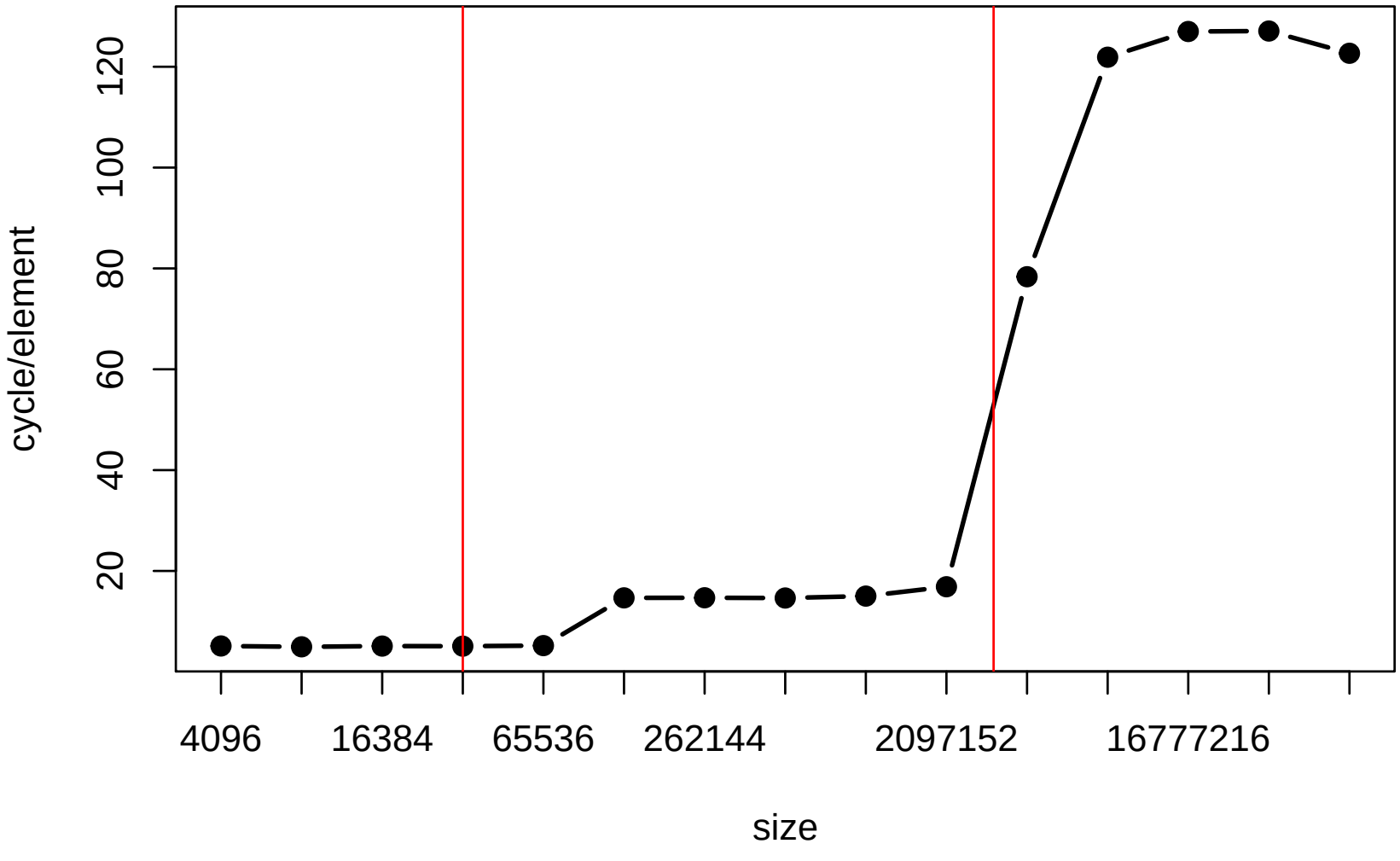
Accès aux données

- Pour forcer l'utilisation du « contenu » de la cellule, on accède au tableau
- Nouveau parcours :

```
Cell current = cells[0];
while (current != null) {
  k = current.content[position];
  current = current.next;
}
```

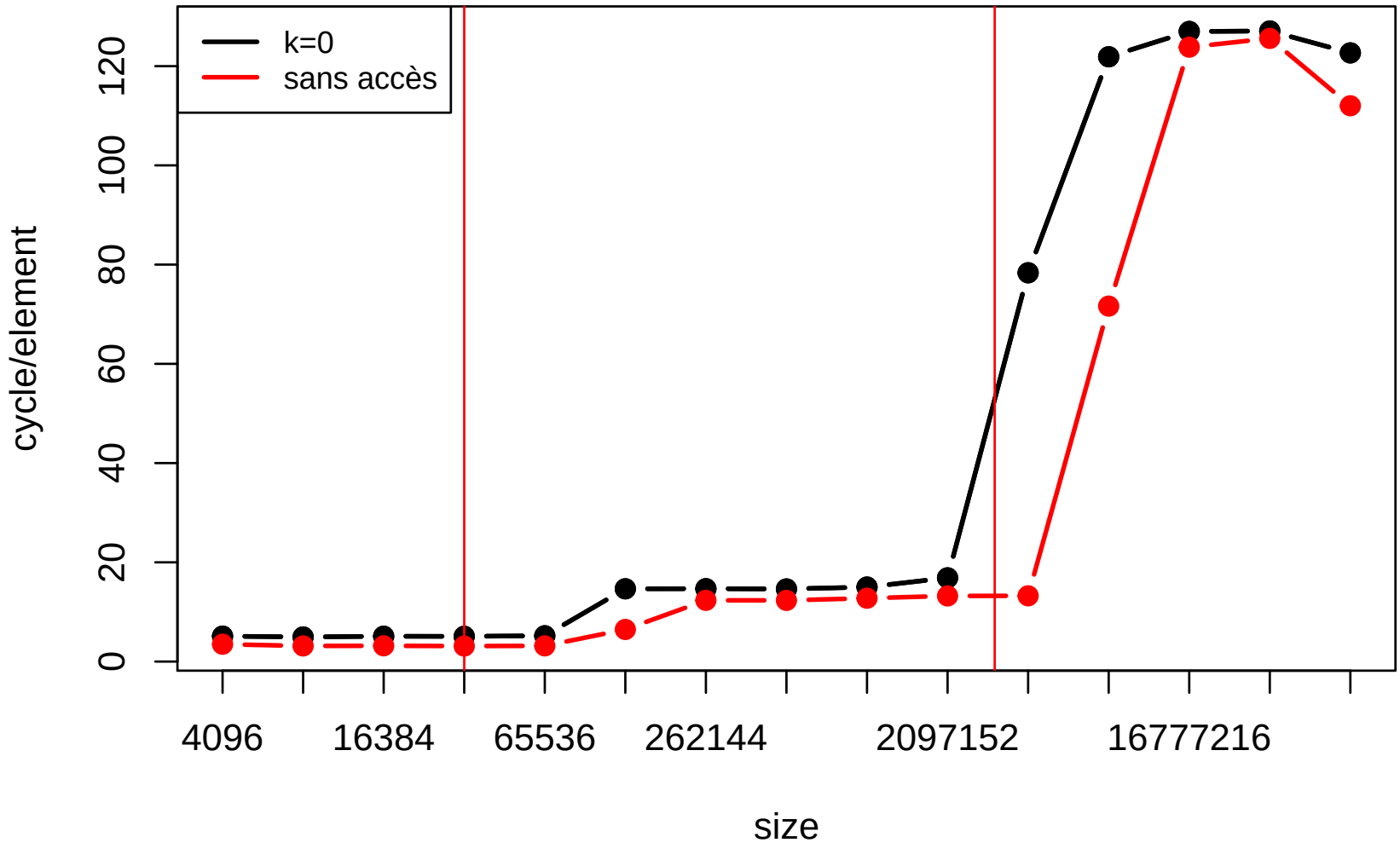
- Potentiellement coûteux :
  - accès à la fin de la cellule (2 mots gestion, 2 mots de référence)
  - indirection
  - vérification des bornes
  - accès « lointain »

Résultats avec position = 0



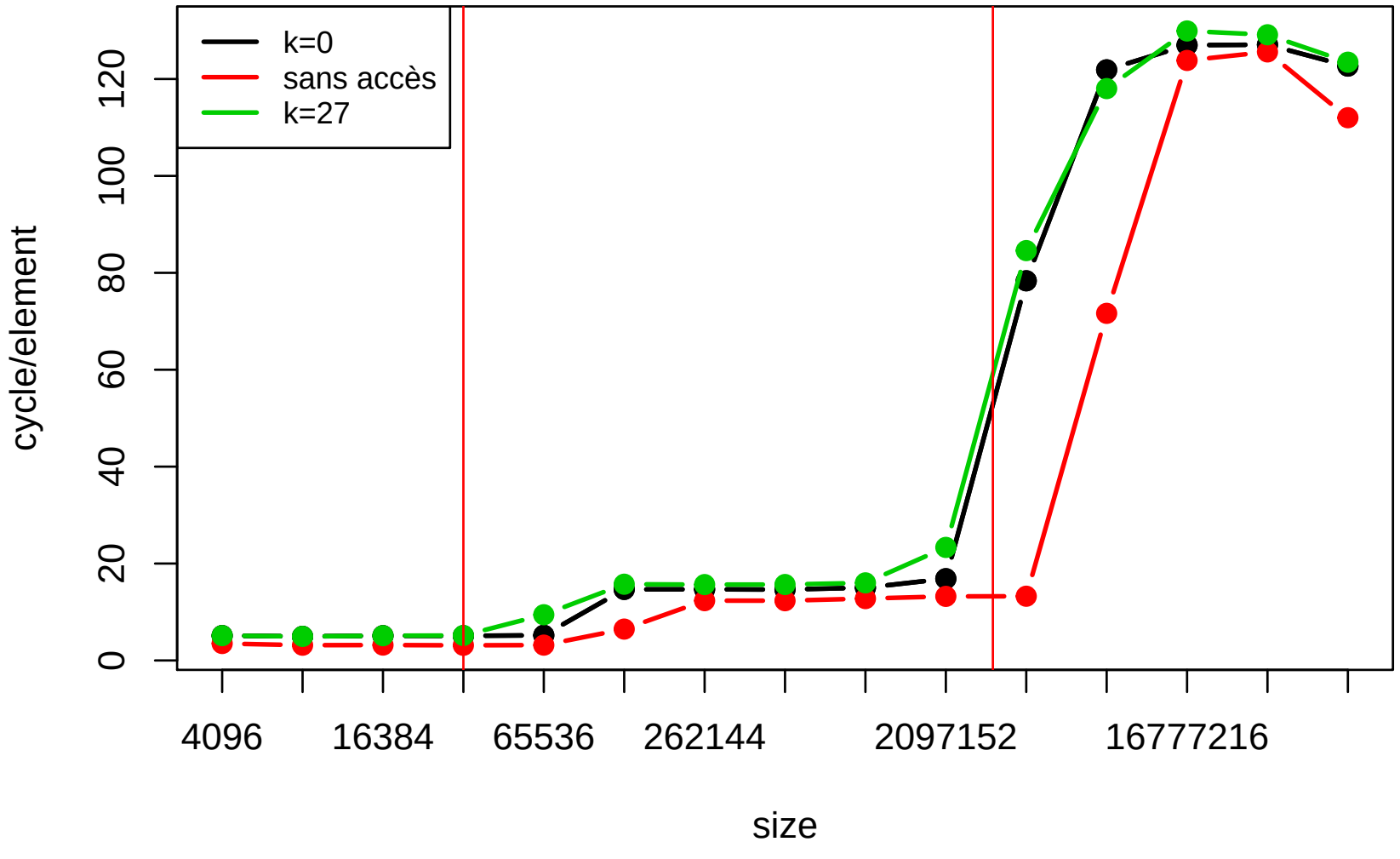
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats avec position = 0



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats avec position = 27



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Conclusions

- Bonne résistance du système de caches :
  - préchargement L1 par anticipation simple : la ligne suivante est préchargée (en parallèle) si on a déjà observé des accès successifs
  - préchargement L2 par anticipation évoluée :
    - suivi des flux d'accès (*stream*)
    - détection des décalages constants (sauts réguliers dans la mémoire)

Conclusions

- Bonne résistance du système de caches :
  - préchargement L1 par anticipation simple : la ligne suivante est préchargée (en parallèle) si on a déjà observé des accès successifs
  - préchargement L2 par anticipation évoluée :
    - suivi des flux d'accès (*stream*)
    - détection des décalages constants (sauts réguliers dans la mémoire)
- Mais il ne faut pas rêver : si on doit travailler sur toute la mémoire, on va observer sa latence tôt ou tard

Conclusions

- Bonne résistance du système de caches :
  - préchargement L1 par anticipation simple : la ligne suivante est préchargée (en parallèle) si on a déjà observé des accès successifs
  - préchargement L2 par anticipation évoluée :
    - suivi des flux d'accès (*stream*)
    - détection des décalages constants (sauts réguliers dans la mémoire)
- Mais il ne faut pas rêver : si on doit travailler sur toute la mémoire, on va observer sa latence tôt ou tard
- On ne peut donc pas espérer atteindre le pic de performance pour un algorithme en cN sauf si :
  - c est très grand : latence masquée par les calculs (200 cycles)
  - N est (très) petit : latence masquée par les caches ($N =$ taille du cache)

Conclusions

- Bonne résistance du système de caches :
  - préchargement L1 par anticipation simple : la ligne suivante est préchargée (en parallèle) si on a déjà observé des accès successifs
  - préchargement L2 par anticipation évoluée :
    - suivi des flux d'accès (*stream*)
    - détection des décalages constants (sauts réguliers dans la mémoire)
- Mais il ne faut pas rêver : si on doit travailler sur toute la mémoire, on va observer sa latence tôt ou tard
- On ne peut donc pas espérer atteindre le pic de performance pour un algorithme en cN sauf si :
  - c est très grand : latence masquée par les calculs (200 cycles)
  - N est (très) petit : latence masquée par les caches ($N =$ taille du cache)
- Accès séquentiel seulement !

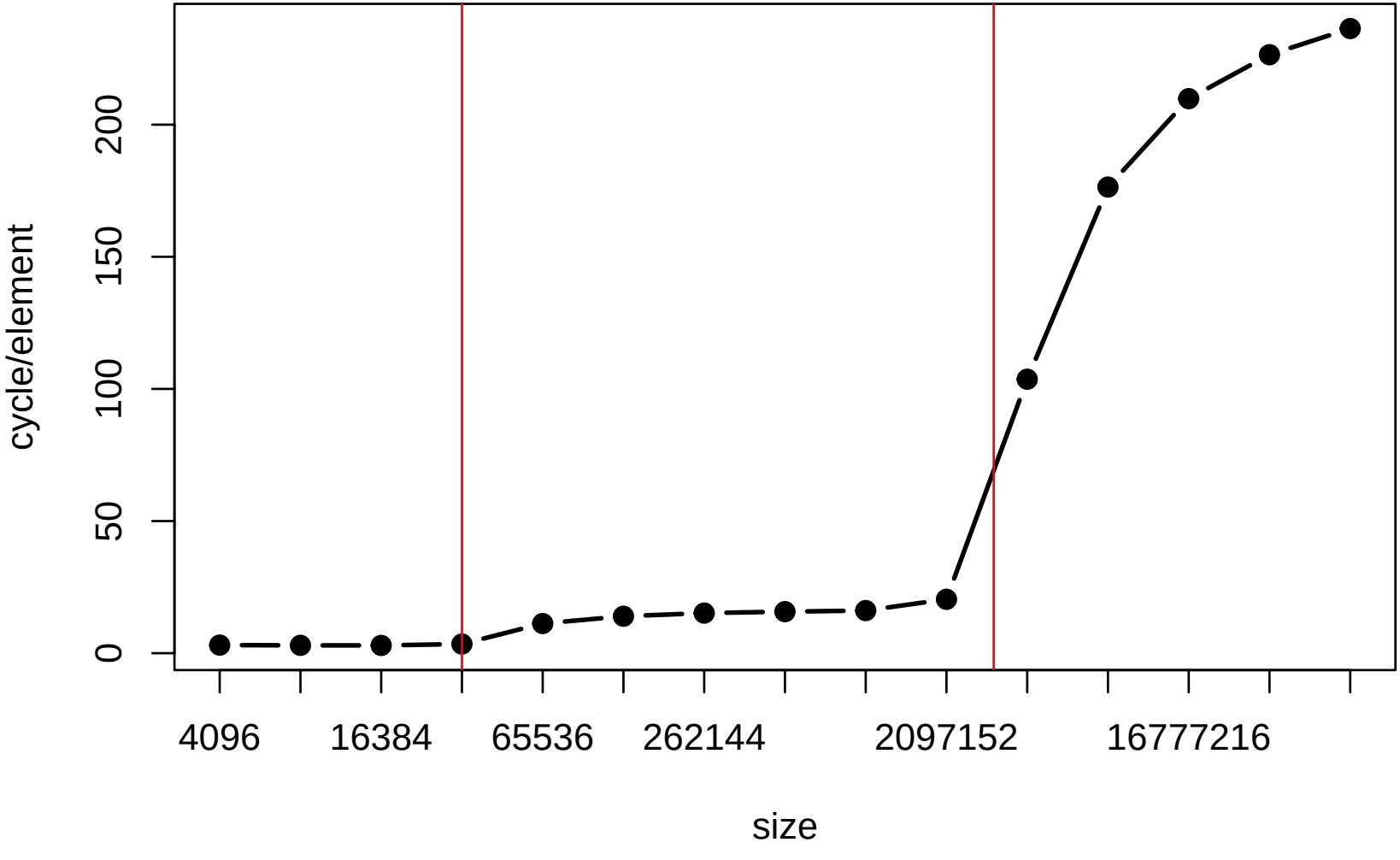
Accès aléatoire

- on modifie le parcours pour qu'il devienne « aléatoire »

```
Random rng = new Random(0);
int[] position = new int[N];
for(int i=0;i<N;i++) {
  position[i]=i;
}
for(int i=0;i<N-1;i++) {
  int pos = i+rng.nextInt(N-i);
  int val = position[pos];
  position[pos]=position[i];
  position[i]=val;
}
for (int i = 1; i < N; i++) {
  cells[position[i - 1]].next = cells[position[i]];
}
```

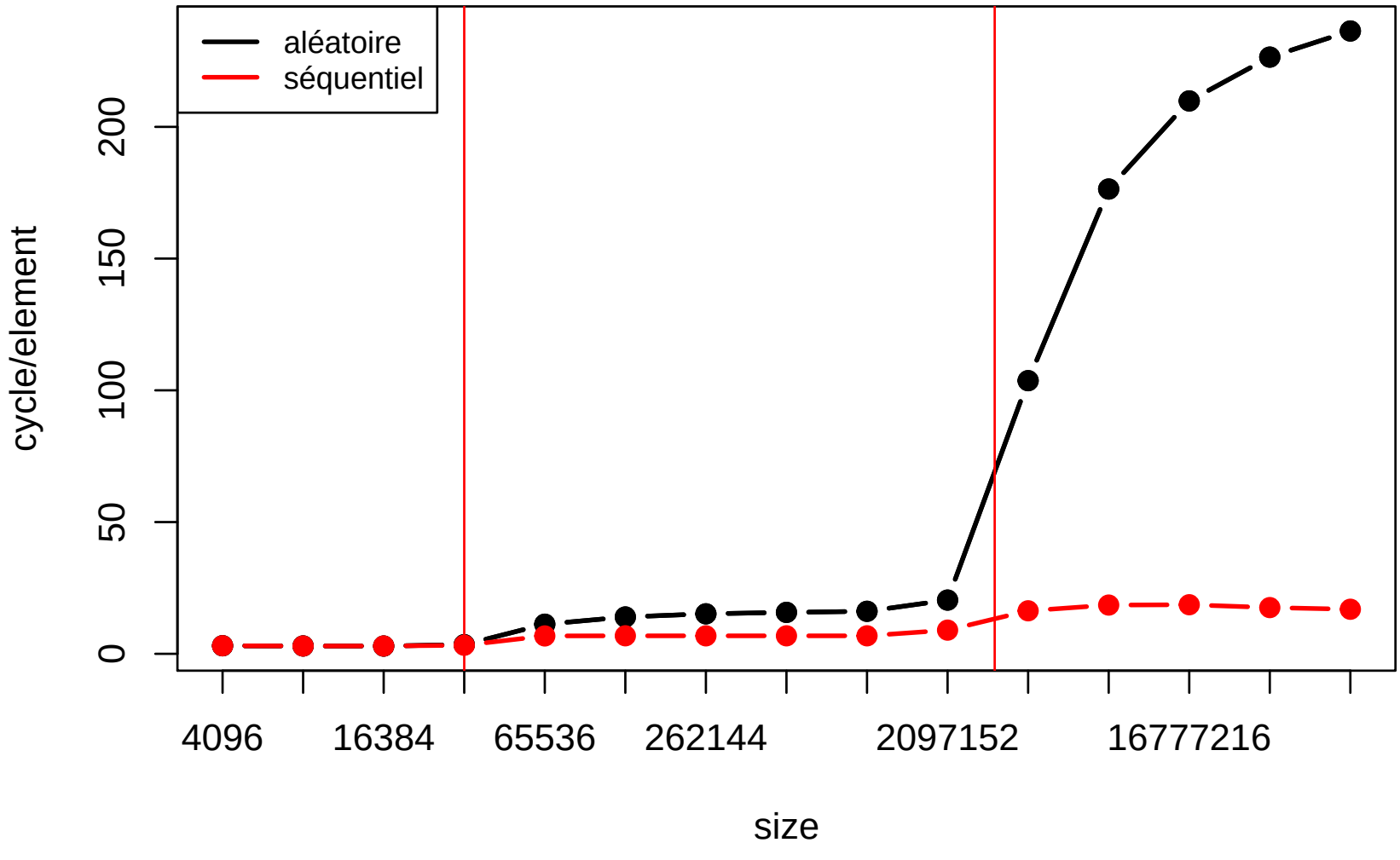
- on suit les références à partir de `cells[position[0]]`

Résultats



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats

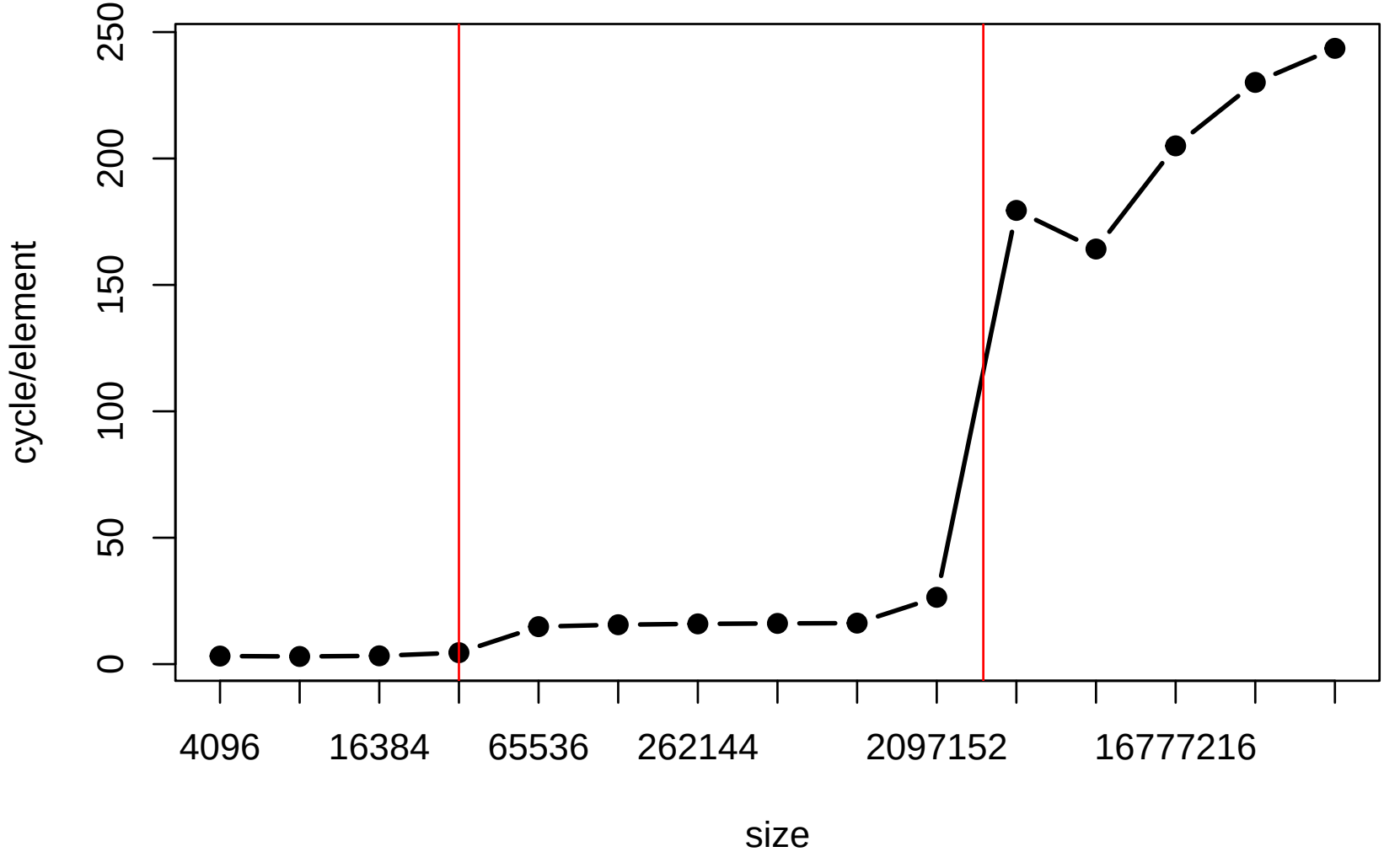


Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Analyse

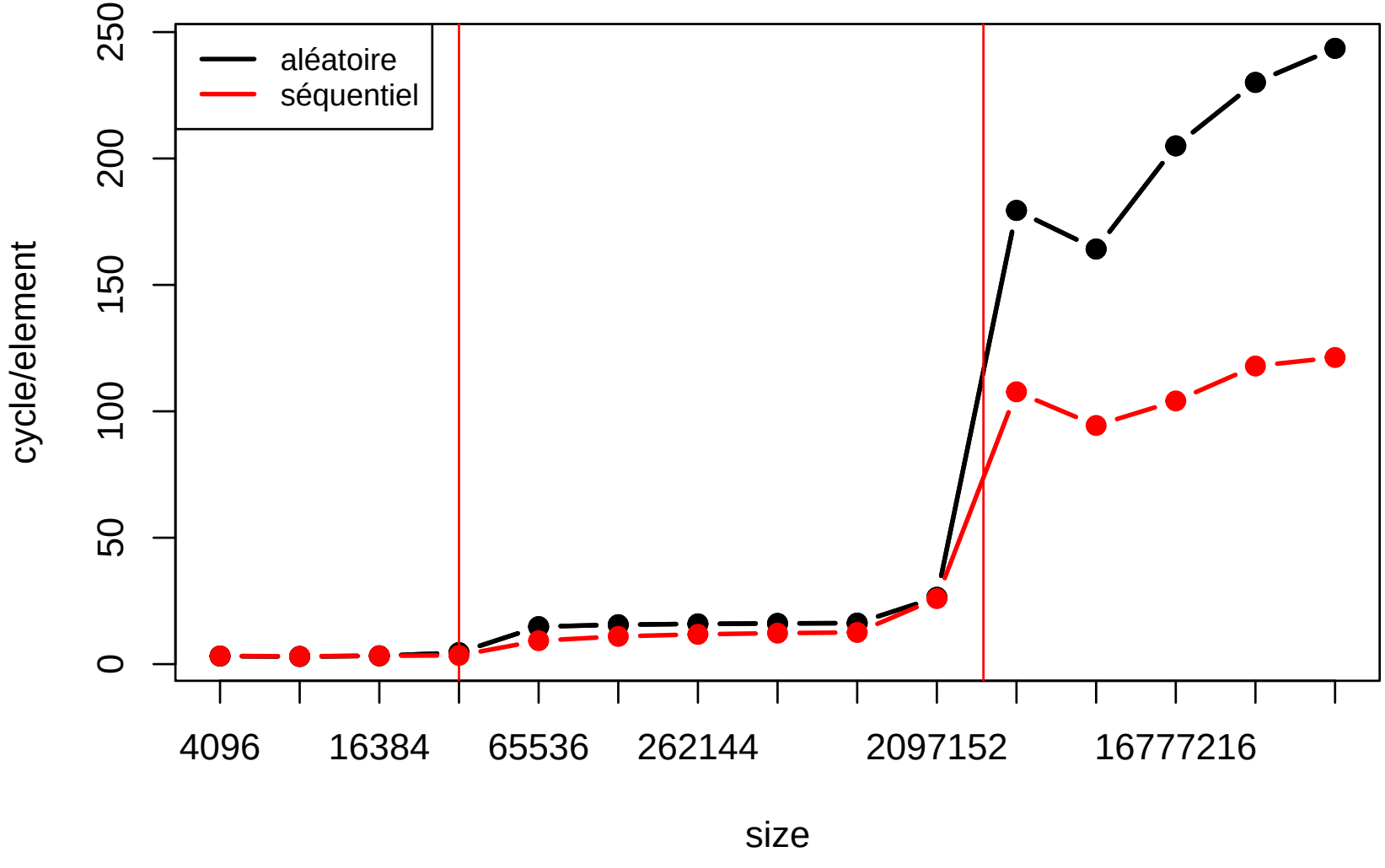
- La cellule pèse toujours 32 octets
- Le cache L1 continue à fonctionner pour une taille faible des données : l'accès est aléatoire même dans le cache, donc latence faible
- Le prefetch L2 vers L1 ne fonctionne pas : latence de 15 cycles quand les données tiennent dans L2 et sont déjà présentes !
- Dès qu'on sort du cache L2, on se heurte à la latence mémoire car le prefetch mémoire vers L2 n'est pas possible : latence de 200 cycles ou plus
- La taille de la cellule ne change pas grand chose

Résultats pour 256 octets



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats pour 256 octets

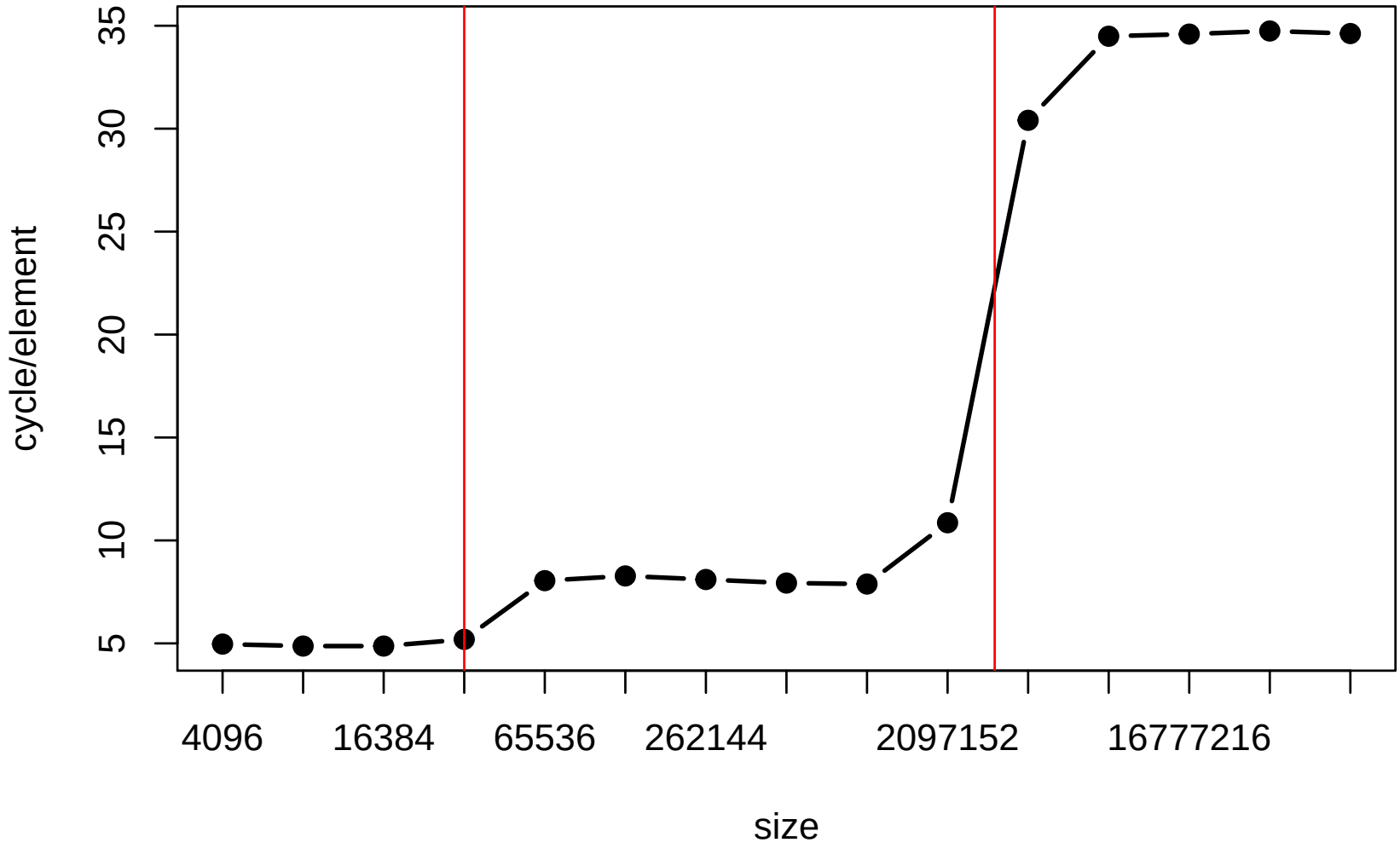


Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Écriture

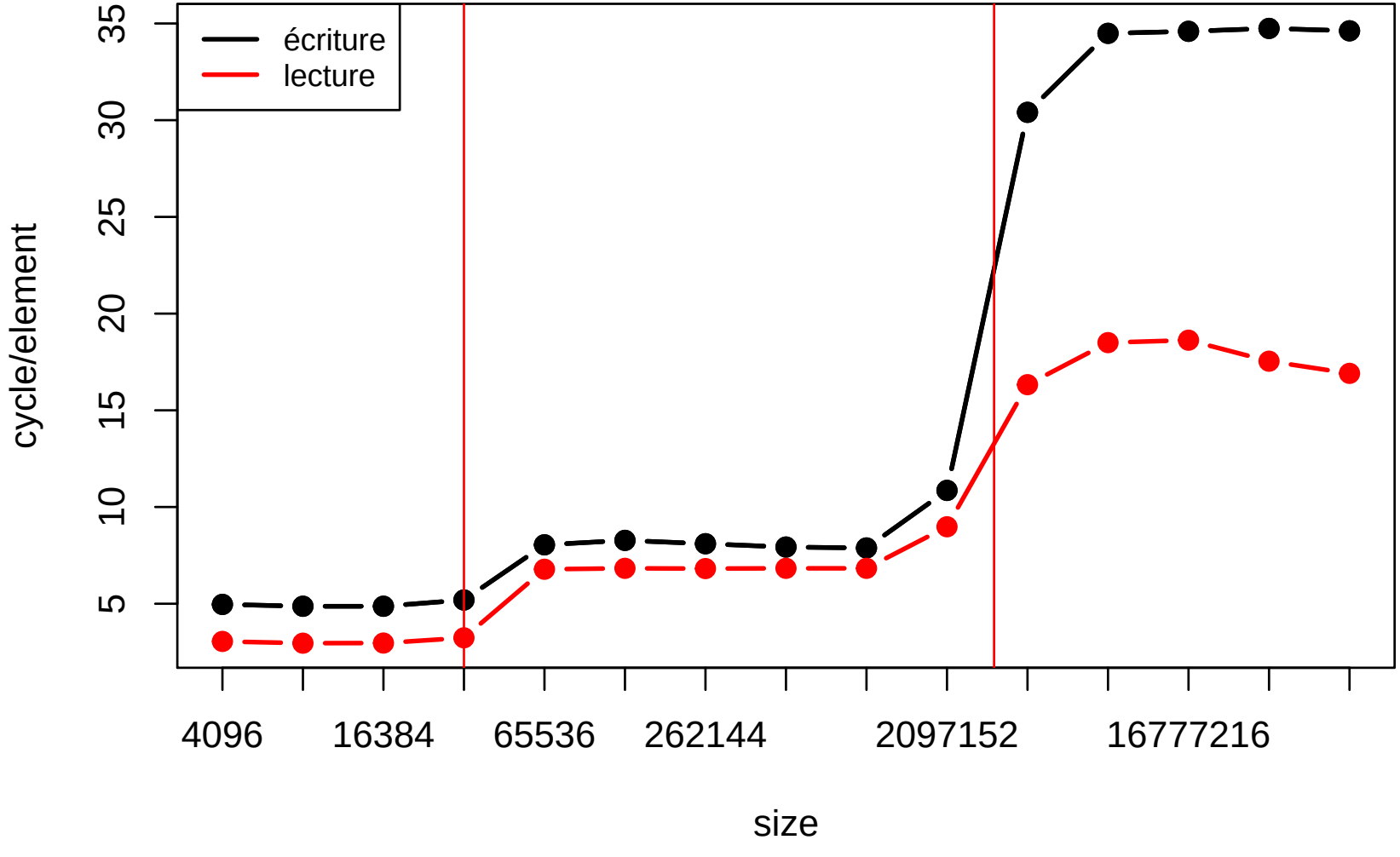
- En théorie, la latence des caches L1 et L2 est la même pour les lectures et les écritures
- En pratique, c'est vrai modulo le temps d'écriture des lignes modifiées :
  - quand on accède à un emplacement qui n'est pas dans le cache, il faut d'abord vider une ligne
  - si la ligne contient des données modifiées, il faut les écrire dans le niveau supérieur
  - la bande passante est divisée par deux en pratique
- Test avec une incrémentation du champ `content` avant le passage à la cellule suivante

Résultats



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Autres effets

- traduction d'adresses :
  - adresses virtuelles $\longrightarrow$ adresses physiques
  - processus coûteux (parcours d'un arbre)
  - cache spécialisé, le *translation lookaside buffer* (TLB)
  - si on travaille sur beaucoup de mémoire, on peut saturer le TLB : grosse dégradation des performances
  - problème supplémentaire : le préchargeur ne peut pas passer les bornes d'une page

Autres effets

- traduction d'adresses :
  - adresses virtuelles $\longrightarrow$ adresses physiques
  - processus coûteux (parcours d'un arbre)
  - cache spécialisé, le *translation lookaside buffer* (TLB)
  - si on travaille sur beaucoup de mémoire, on peut saturer le TLB : grosse dégradation des performances
  - problème supplémentaire : le préchargeur ne peut pas passer les bornes d'une page
- parallélisme :
  - multi-coeurs avec cache L2/L3 partagé
  - *hyper threading*
  - en général, on observe une dégradation des performances en raison du partage des caches
  - mais on peut parfois gagner en utilisant un *thread* de préchargement, par exemple

Plan

Grands principes

- masquer la latence de la mémoire :
  - exploiter les caches : profiter de la localité et du préchargement
  - entrelacer calculs et accès
- deux familles de techniques :
  - optimiser l'accès aux données
  - optimiser la disposition des données en mémoire

Transformation de boucles

- Outil principal pour l'optimisation des accès
- Échange de boucles :
  - permutation de boucles emboîtées
  - but : rendre la boucle interne locale
- Fusion de boucles :
  - remplacer des boucles successives par une seule
  - but : regrouper des traitements portants sur les mêmes données
- Décomposition en blocs :
  - éclater une boucle en plusieurs boucles emboîtées
  - but : rendre locaux les accès à deux structures différentes

Retour sur l'exemple

- calcul des moyennes :

```
double[][] data=new double[N][M];
...
double[] mean=new double[M];
for(int j=0;j<M;j++) {
  double tmp=0;
  for(int i=0;i<N;i++) {
    tmp += data[i][j];
  }
  mean[j]=tmp/N;
}
```

Retour sur l'exemple

- calcul des moyennes :

```
double[][] data=new double[N][M];
...
double[] mean=new double[M];
for(int j=0;j<M;j++) {
  double tmp=0;
  for(int i=0;i<N;i++) {
    tmp += data[i][j];
  }
  mean[j]=tmp/N;
}
```

- analyse :
  - `tmp` : dans un registre, pas de problème
  - `data[i]` : accès à un tableau de taille $8N$ octets
  - `data[i][j]` : deuxième indirection dans un tableau de taille $8M$

Retour sur l'exemple

- calcul des moyennes :

```
double[][] data=new double[N][M];
...
double[] mean=new double[M];
for(int j=0;j<M;j++) {
  double tmp=0;
  for(int i=0;i<N;i++) {
    tmp += data[i][j];
  }
  mean[j]=tmp/N;
}
```

- analyse :
  - `tmp` : dans un registre, pas de problème
  - `data[i]` : accès à un tableau de taille $8N$ octets
  - `data[i][j]` : deuxième indirection dans un tableau de taille $8M$
  - le problème principal vient de la deuxième indirection : si `data` est alloué d'un bloc, on saute à chaque itération de $8M$ octets

Analyse

- si M est très petit, `data[i][j]` et `data[i+k][j]` peuvent être dans la même ligne de cache :
  - $M \geq 8$: impossible
  - $kM < 8$: cela peut arriver, mais pas souvent
  - si $M = 1$, potentiellement 8 valeurs dans la ligne : peut masquer la latence du cache L1

Analyse

- si M est très petit, `data[i][j]` et `data[i+k][j]` peuvent être dans la même ligne de cache :
  - $M \geq 8$: impossible
  - $kM < 8$: cela peut arriver, mais pas souvent
  - si $M = 1$, potentiellement 8 valeurs dans la ligne : peut masquer la latence du cache L1
- Effet du préchargement :
  - si $M < 8$, accès séquentiel dans le cache L1, préchargement de la ligne suivante
  - sinon, détection potentielle des sauts par le cache L2, si $8M \leq 512$ (donc $M \leq 64$)
  - ensuite, c'est la catastrophe

Analyse

- si M est très petit, `data[i][j]` et `data[i+k][j]` peuvent être dans la même ligne de cache :
  - $M \geq 8$: impossible
  - $kM < 8$: cela peut arriver, mais pas souvent
  - si $M = 1$, potentiellement 8 valeurs dans la ligne : peut masquer la latence du cache L1
- Effet du préchargement :
  - si $M < 8$, accès séquentiel dans le cache L1, préchargement de la ligne suivante
  - sinon, détection potentielle des sauts par le cache L2, si $8M \leq 512$ (donc $M \leq 64$)
  - ensuite, c'est la catastrophe
- Amélioration : réduire les écarts

Changer l'ordre des calculs

- Solution simple : modifier l'ordre des calculs
- Principe : faire en sorte que la partie intensive en mémoire travaille séquentiellement et avec des sauts courts

Changer l'ordre des calculs

- Solution simple : modifier l'ordre des calculs
- Principe : faire en sorte que la partie intensive en mémoire travaille séquentiellement et avec des sauts courts
- Dans l'exemple, on échange les boucles :

```
for(int i=0;i<N;i++) {
  for(int j=0;j<M;j++) {
    mean[j]+= data[i][j];
  }
}
for(int j=0;j<M;j++) {
  mean[j]/=N;
}
```

Changer l'ordre des calculs

- Solution simple : modifier l'ordre des calculs
- Principe : faire en sorte que la partie intensive en mémoire travaille séquentiellement et avec des sauts courts
- Dans l'exemple, on échange les boucles :

```
for(int i=0;i<N;i++) {
  for(int j=0;j<M;j++) {
    mean[j]+= data[i][j];
  }
}
for(int j=0;j<M;j++) {
  mean[j]/=N;
}
```

- Ici, on se décale de 8 octets à chaque itération
- On utilise en parallèle deux lignes de cache (`mean` et `data[i]`)
- La précharge fonctionne parfaitement, puisqu'on est purement séquentiel

Fusion de boucles

- exemple de calcul vectoriel : $y \leftarrow \alpha x$ et $z \leftarrow y + z$
- implémentation naïve :

```
for(int i=0;i<N;i++) {
  y[i] = a * x[i];
}
for(int i=0;i<N;i++) {
  z[i] += y[i];
}
```

Fusion de boucles

- exemple de calcul vectoriel : $y \leftarrow \alpha x$ et $z \leftarrow y + z$
- implémentation naïve :

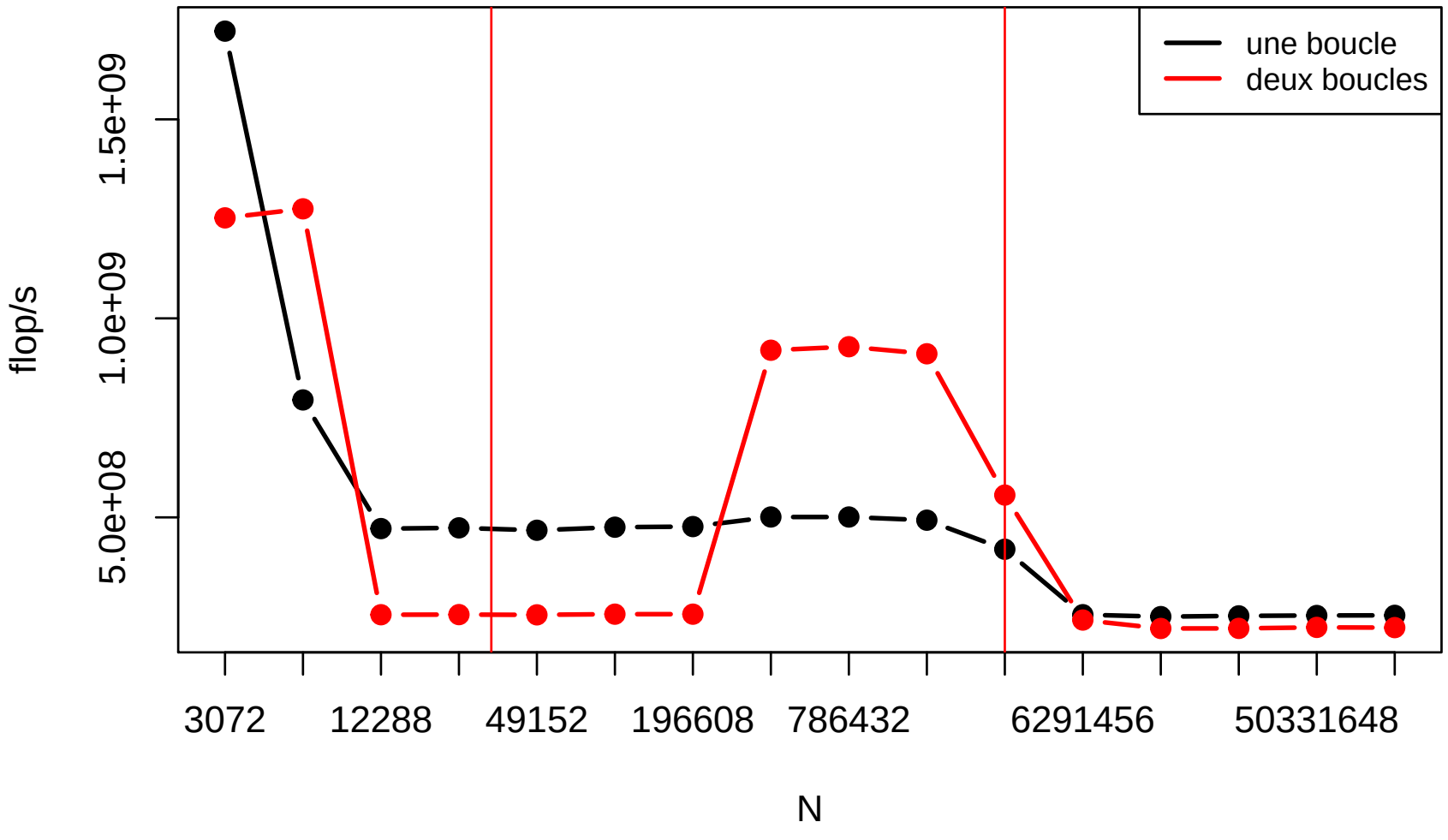
```
for(int i=0;i<N;i++) {
  y[i] = a * x[i];
}
for(int i=0;i<N;i++) {
  z[i] += y[i];
}
```

- fusion :

```
for(int i=0;i<N;i++) {
  y[i] = a * x[i];
  z[i] += y[i];
}
```

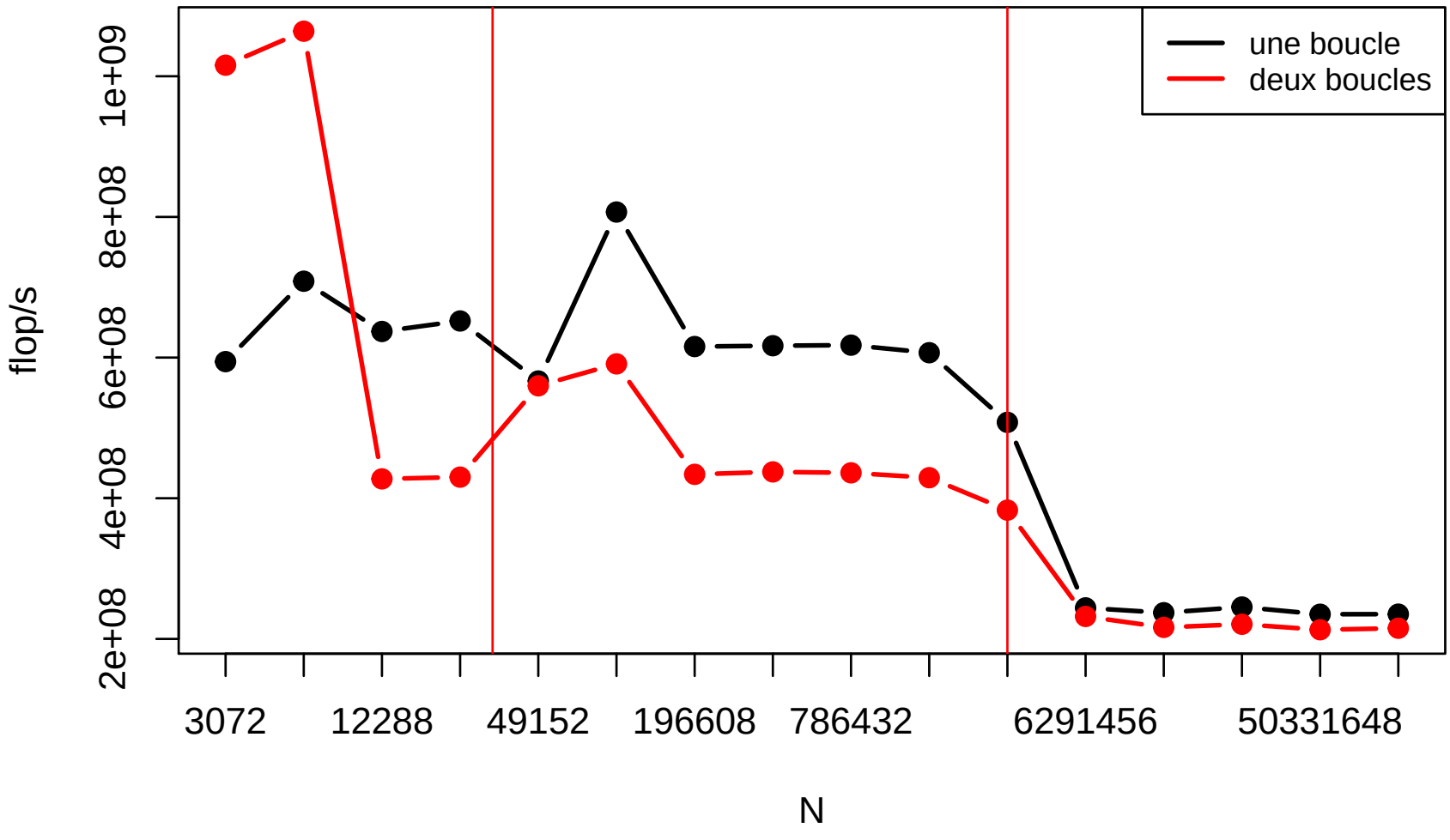
- avantages évidents :
  - moins de surcharge induite par la boucle
  - un seul parcours de la mémoire

Résultats en C



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats en Java



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

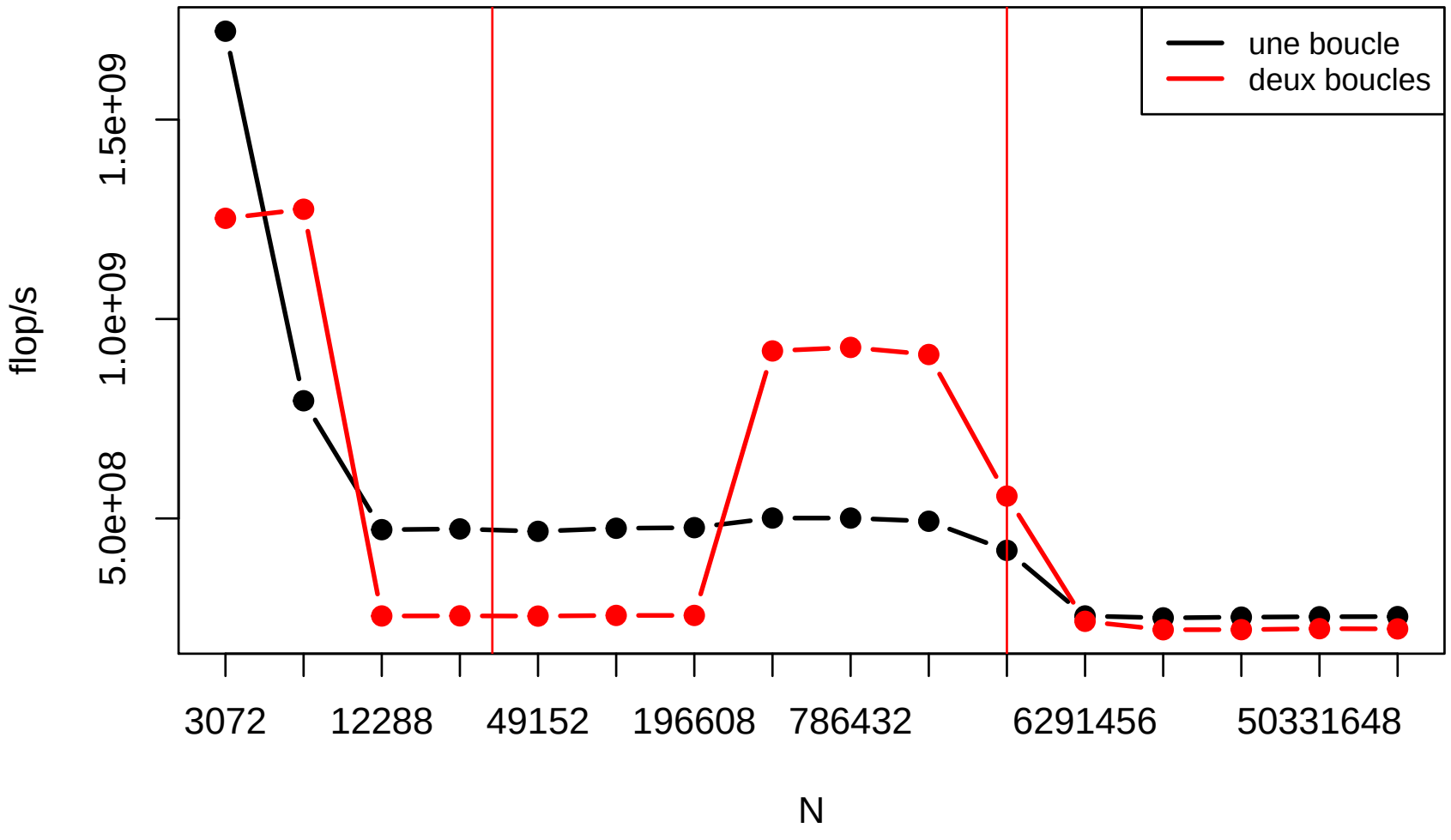
Commentaires

- comportement délicat à analyser :
  - recouvrement calcul/transfert
  - préchargement vs disposition en mémoire
  - optimisation du code (en Java)
- le code optimal dépend des données...

Commentaires

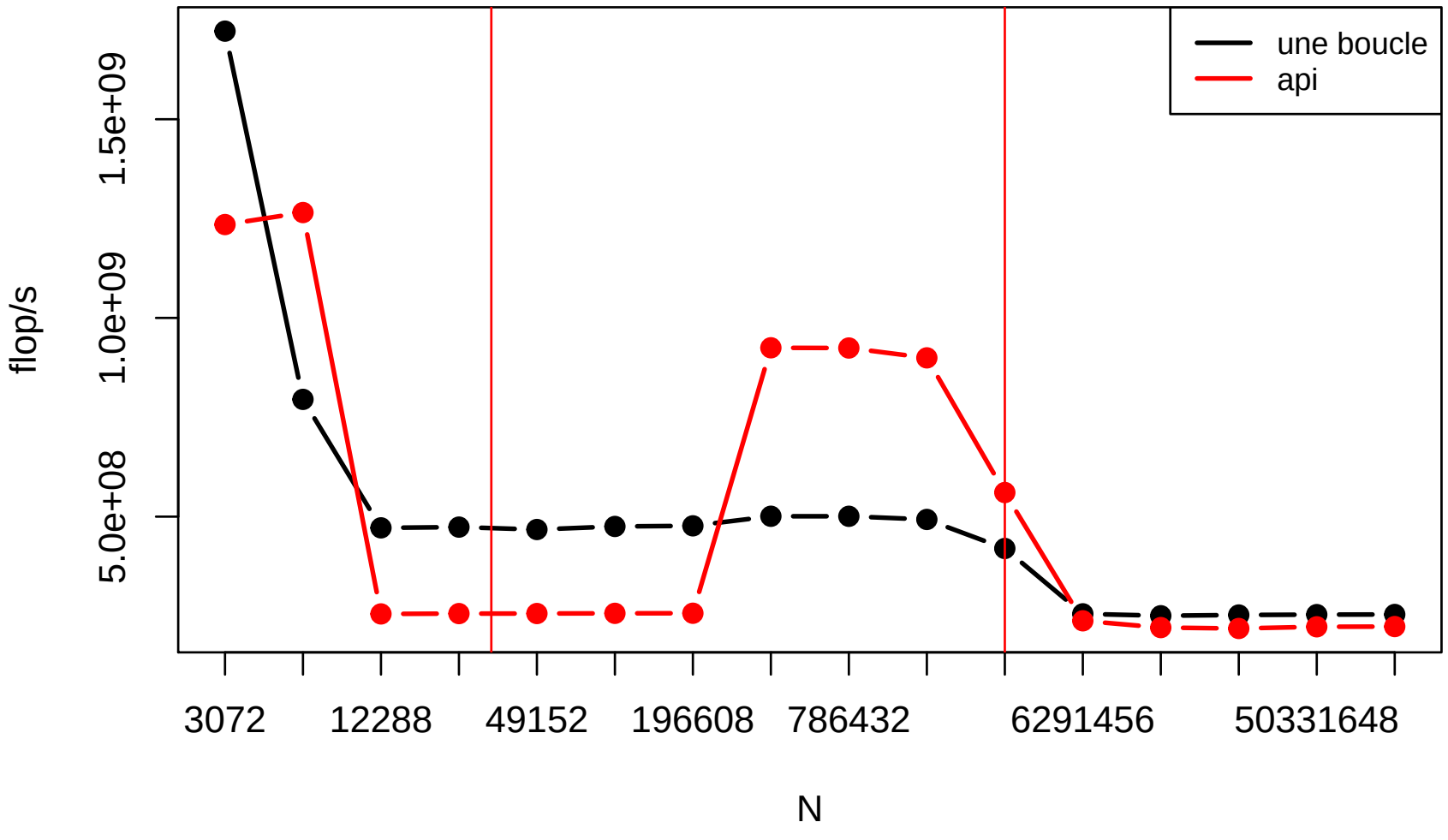
- comportement délicat à analyser :
  - recouvrement calcul/transfert
  - préchargement vs disposition en mémoire
  - optimisation du code (en Java)
- le code optimal dépend des données...
- pourquoi deux boucles en pratique ?
  - abstraction : API de calcul vectoriel
  - richesse fonctionnelle versus complexité
- attention : les problèmes de performances ne viennent pas de l'abstraction elle-même (surcoût de l'appel de fonctions par ex.) mais de ses conséquences en terme de localité mémoire (cf slides suivants)

Résultats en C



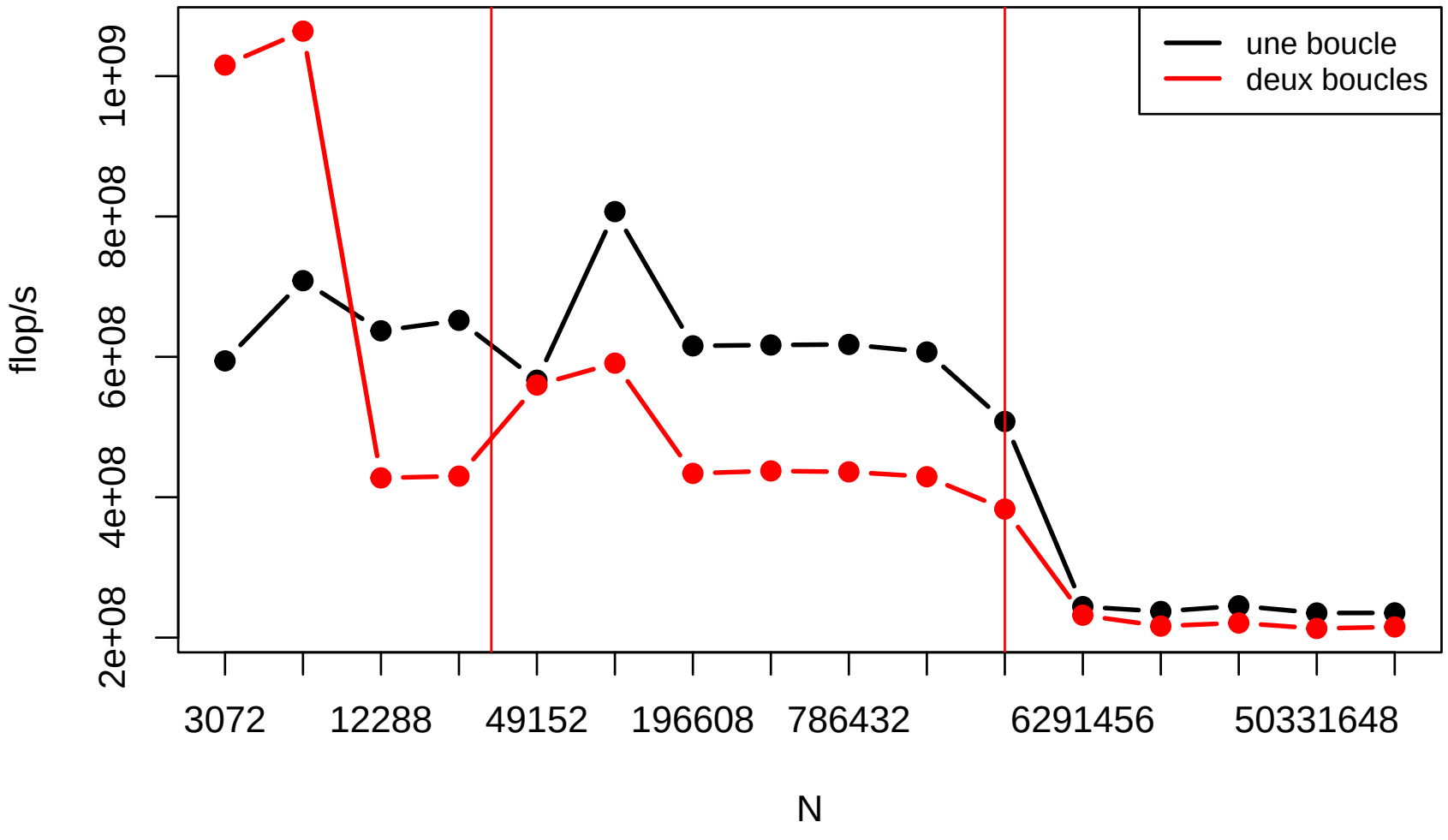
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats en C



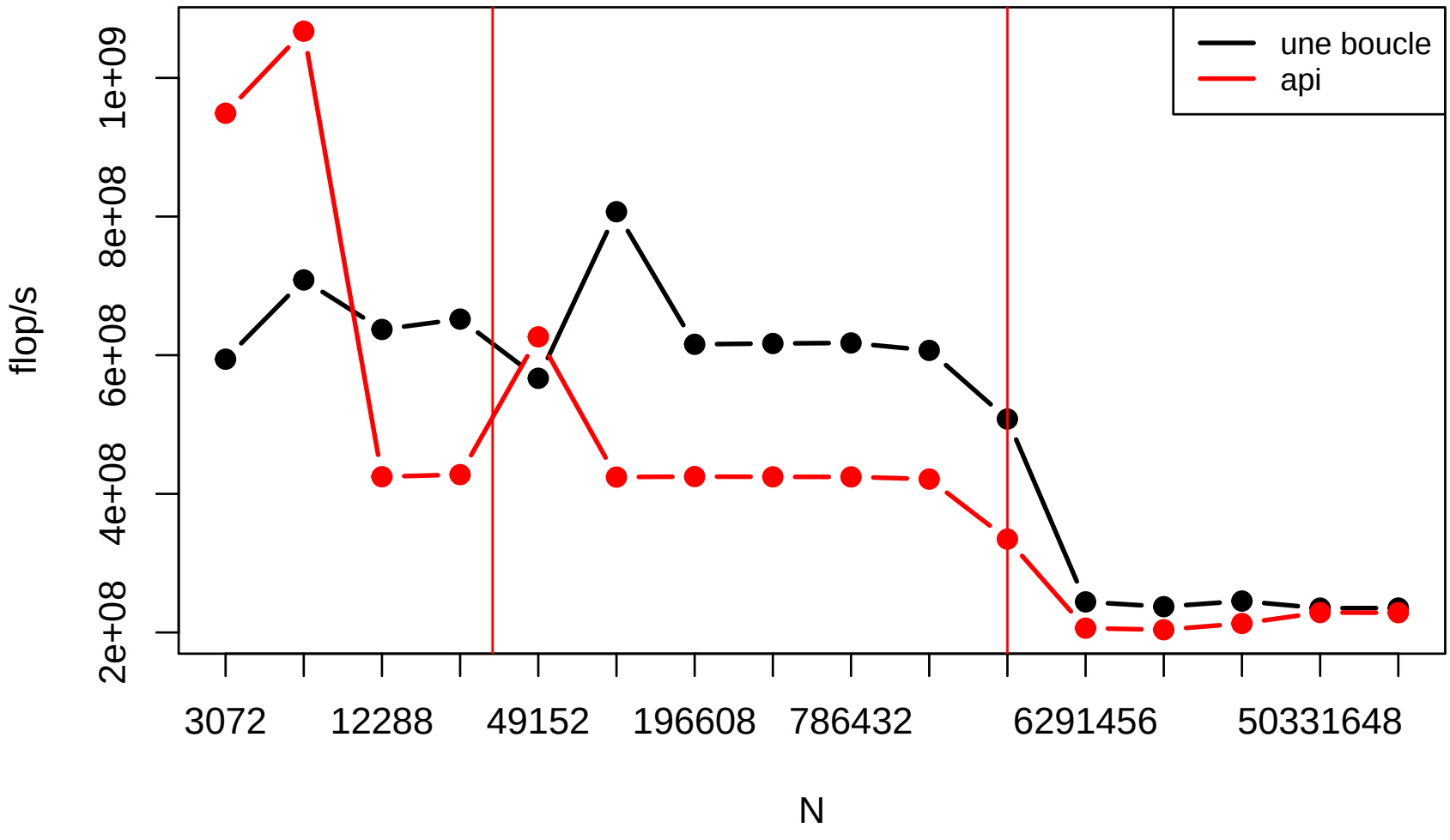
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats en Java



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats en Java



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Décomposition en blocs

- le cas des moyennes est idéal :
  - mauvais ordre des boucles : aucun accès local
  - bon ordre des boucles : tous les accès aux données sont locaux
- dans des problèmes plus complexes, il n'y a pas toujours d'ordre optimal

Décomposition en blocs

- le cas des moyennes est idéal :
  - mauvais ordre des boucles : aucun accès local
  - bon ordre des boucles : tous les accès aux données sont locaux
- dans des problèmes plus complexes, il n'y a pas toujours d'ordre optimal
- exemple : transposition d'une matrice
- deux tableaux $N \times N$, A et B, on veut faire $B \leftarrow A^T$
- approche naïve

```
for(int i=0;i<N;i++) {
  for(int j=0;j<N;j++) {
    B[j][i] = A[i][j];
  }
}
```

- accès non locaux obligatoires (sur A ou sur B)

Transposition par blocs

- pour l'ordre `j` rapide, le problème vient de la distance entre `B[j][i]` et `B[j+1][i]`
- si on limite la boucle sur `j` à quelques itérations, les lignes de `B` concernées peuvent tenir des lignes de cache séparées
- pour éviter la saturation prématurée du cache, il faut aussi limiter l'excursion de `i` : on travaille sur un bloc de `B` (et donc de `A`) de taille `M`×`M`

Transposition par blocs

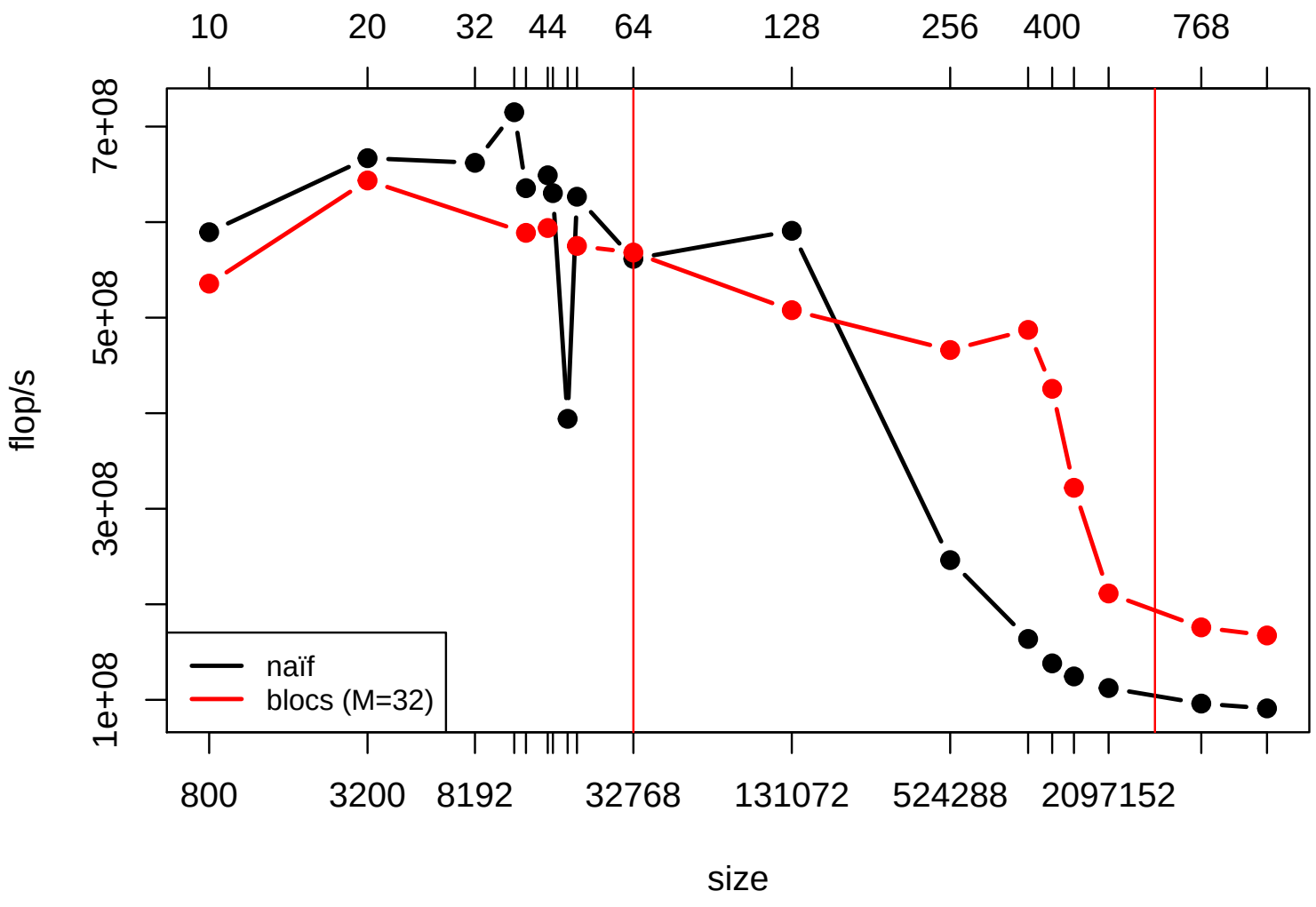
- pour l'ordre `j` rapide, le problème vient de la distance entre `B[j][i]` et `B[j+1][i]`
- si on limite la boucle sur `j` à quelques itérations, les lignes de `B` concernées peuvent tenir des lignes de cache séparées
- pour éviter la saturation prématurée du cache, il faut aussi limiter l'excursion de `i` : on travaille sur un bloc de `B` (et donc de `A`) de taille `M`×`M`

```
for (int oi = 0; oi < N; oi += M) {
  for (int oj = 0; oj < N; oj += M) {
    for (int i = oi; i < N && i < oi + M; i++) {
      for (int j = oj; j < N && j < oj + M; j++) {
        B[j][i] = A[i][j];
      }
    }
  }
}
```

Taille des blocs/matrices

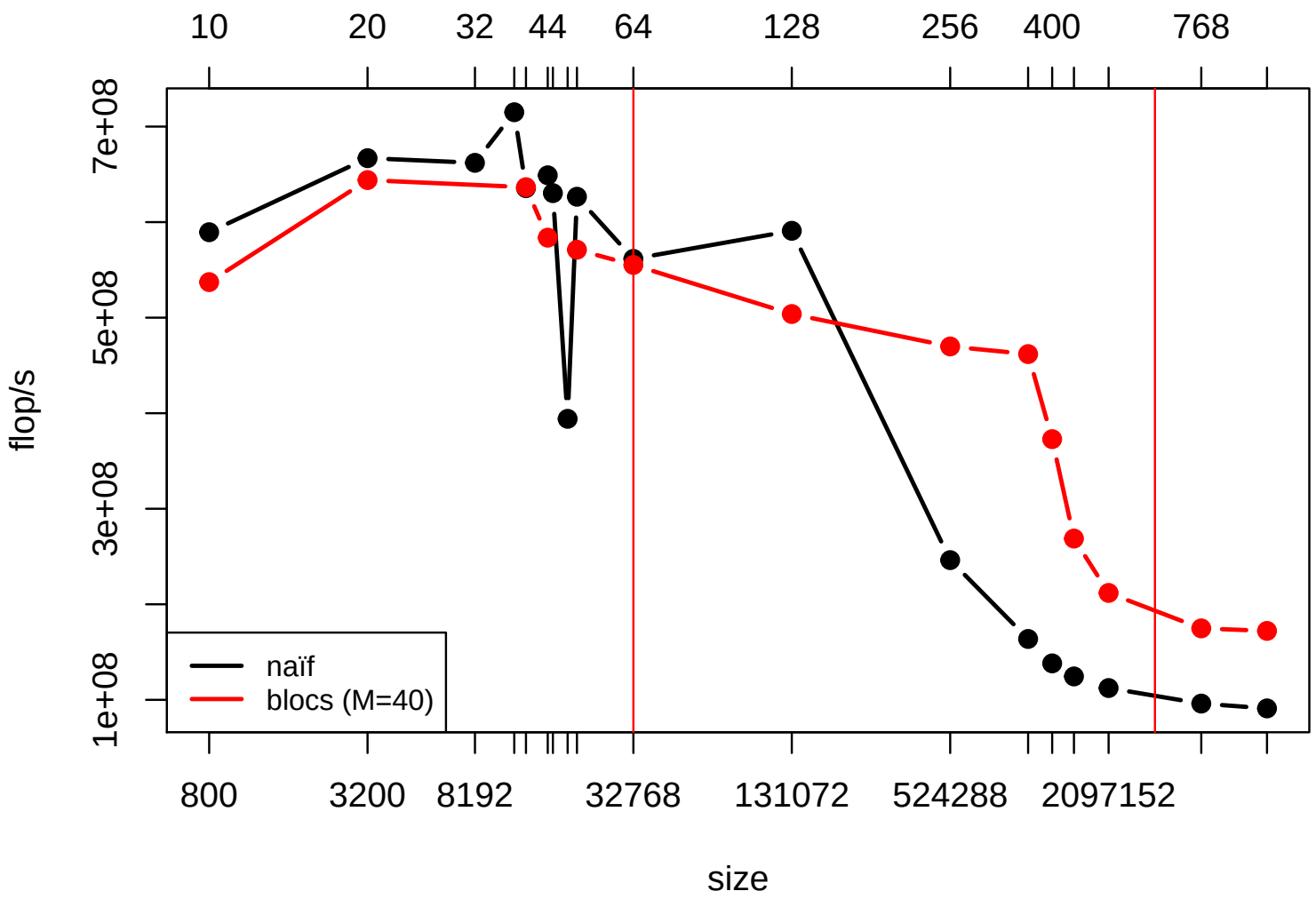
- cache L1 :
  - sur un Core 2, une ligne fait 64 octets soit 8 `double`s, donc au minimum on utilise un bloc 8×8
  - mais un tel bloc n'occupe que 8 lignes, soit 512 octets : très loin des 32 ko disponibles
  - avec deux blocs $M \times M$, on sature le cache L1 pour $M = 45$ (pour une occupation de 32 400 octets)
  - attention à l'alignement : le grain est de 8 `double`s, donc $M = 44$ est un meilleur choix *a priori*
- cache L2 :
  - même raisonnement
  - taille totale 3Mo sur le Core 2 Duo P8600, mais :
    - partagé entre les cores
    - partagé avec le cache d'instruction
  - un bloc 360×360 occupe 1 036 800 octets (environ 1 Mo)

Résultats



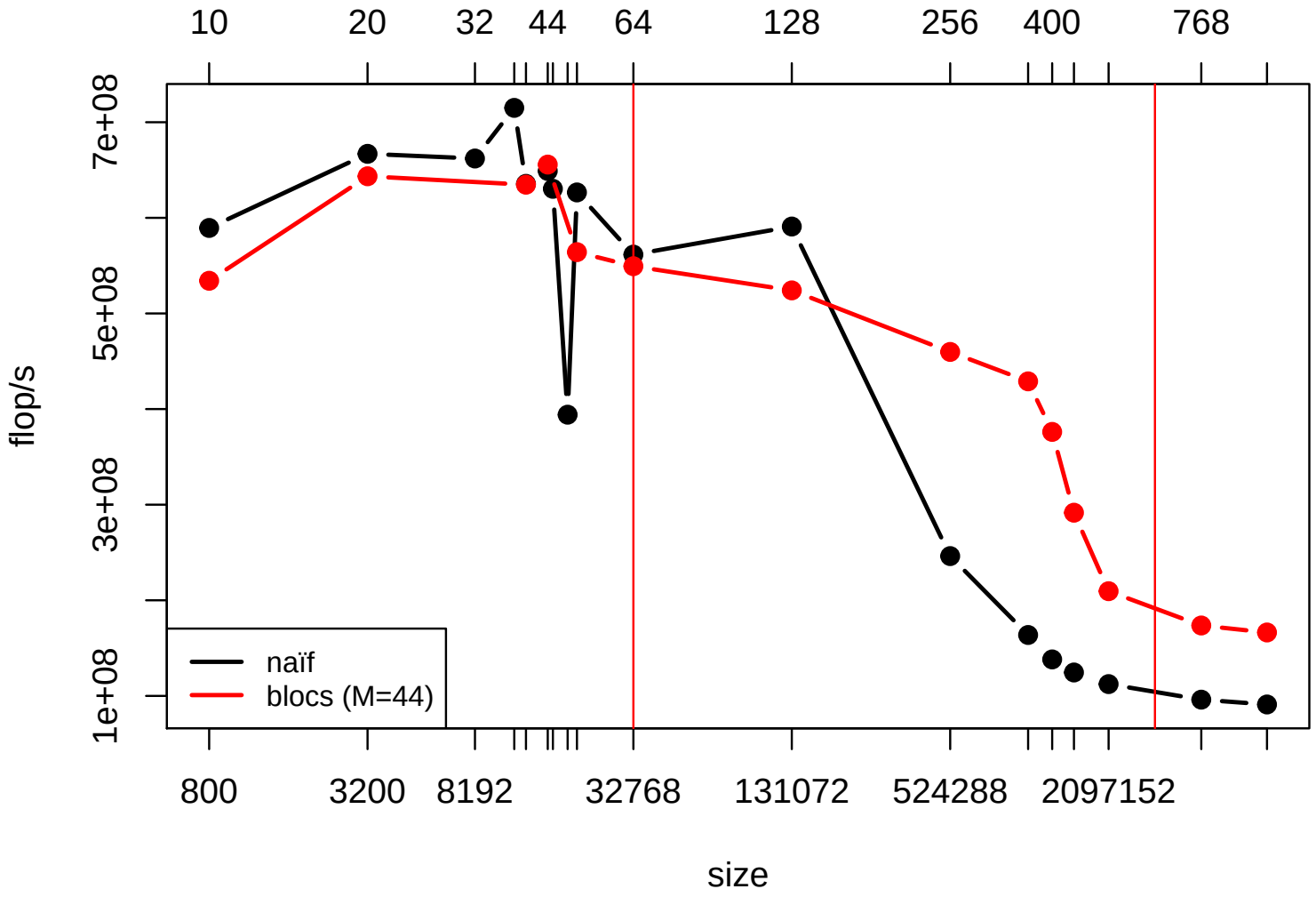
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats



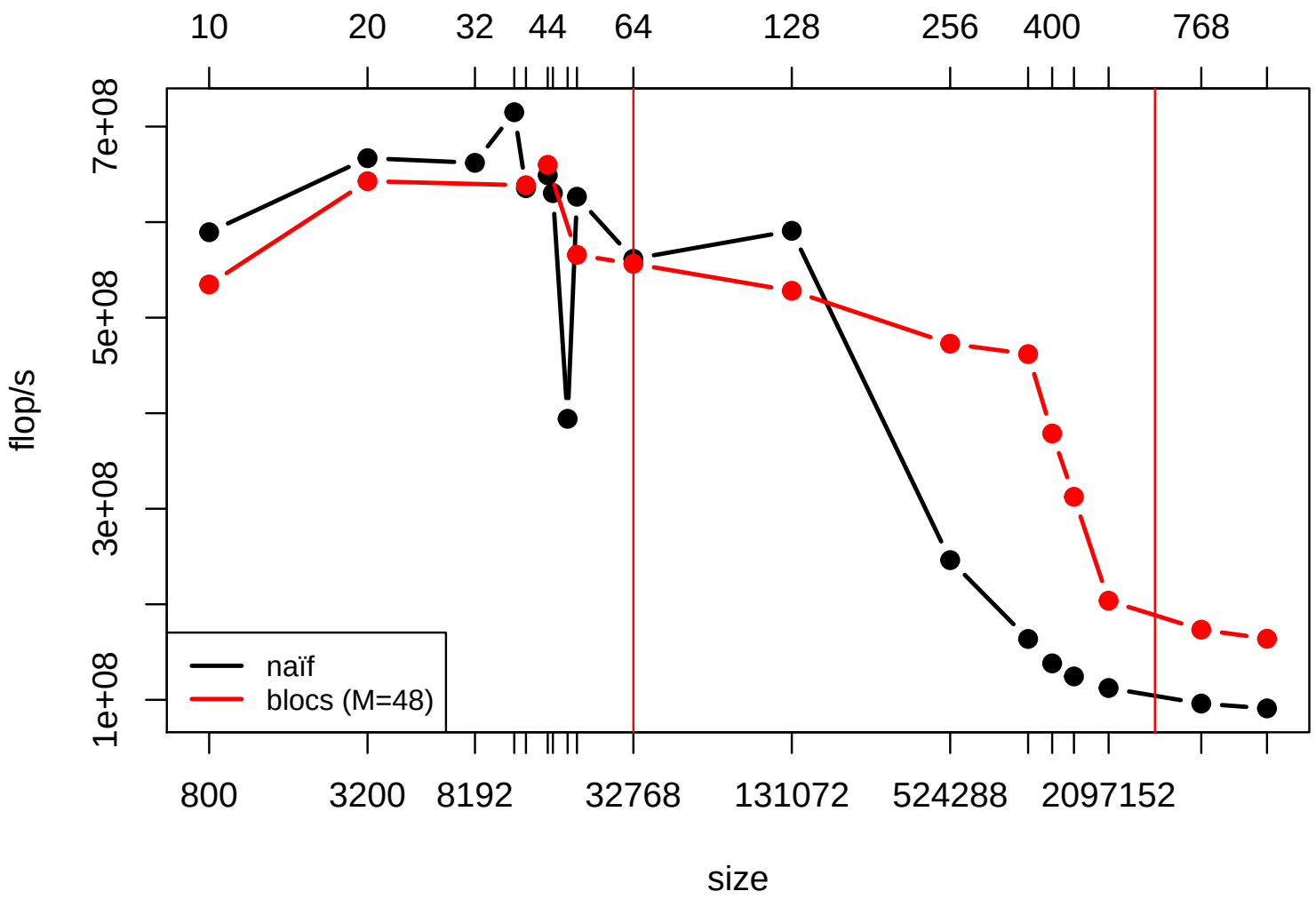
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats



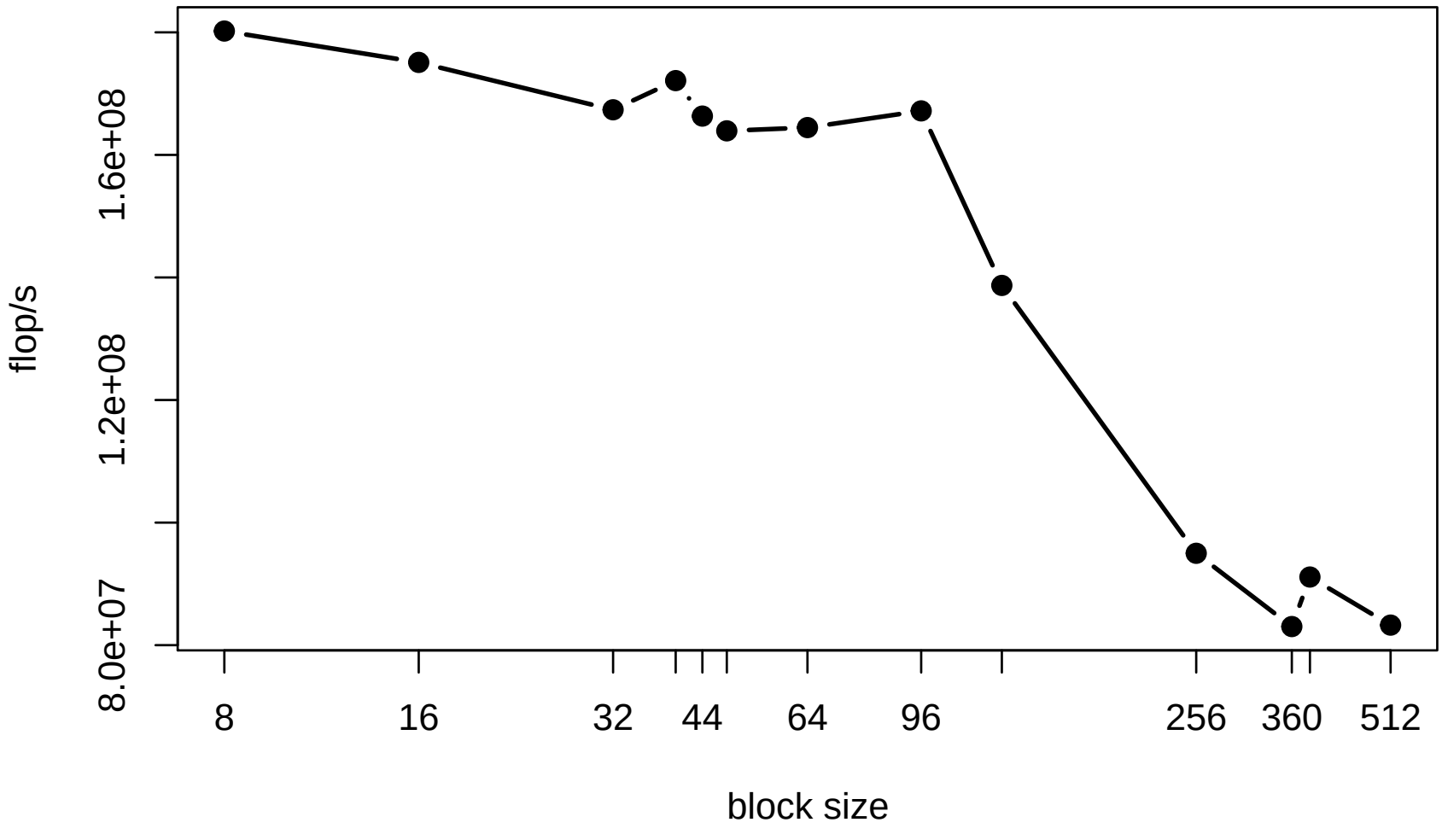
Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Résultats pour $N = 1024$



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo)

Analyse

- léger surcoût induit par les boucles (tant que les matrices tiennent dans le cache L1)
- retarde l'apparition de la latence
- optimisation de la taille des blocs assez délicate :
  - pas de réutilisation des données : seul le préchargement est utilisé
  - effet du préchargement L2 vers L1 peu clair
  - lien complexe entre taille des données et taille du bloc optimal

Disposition des données

- principe : s'arranger pour que les données soient ordonnées en mémoire selon les accès demandés par le programme
- solution alternative/complémentaire à la transformation de boucles
- techniques :
  - *padding* : insérer des données entre deux blocs qui doivent être utilisés conjointement (pour éviter limiter les conflits dans le cache)
  - fusion : rapprocher des données utilisées conjointement (un tableau de structures versus plusieurs tableaux de types fondamentaux)
  - copie avec réarrangement

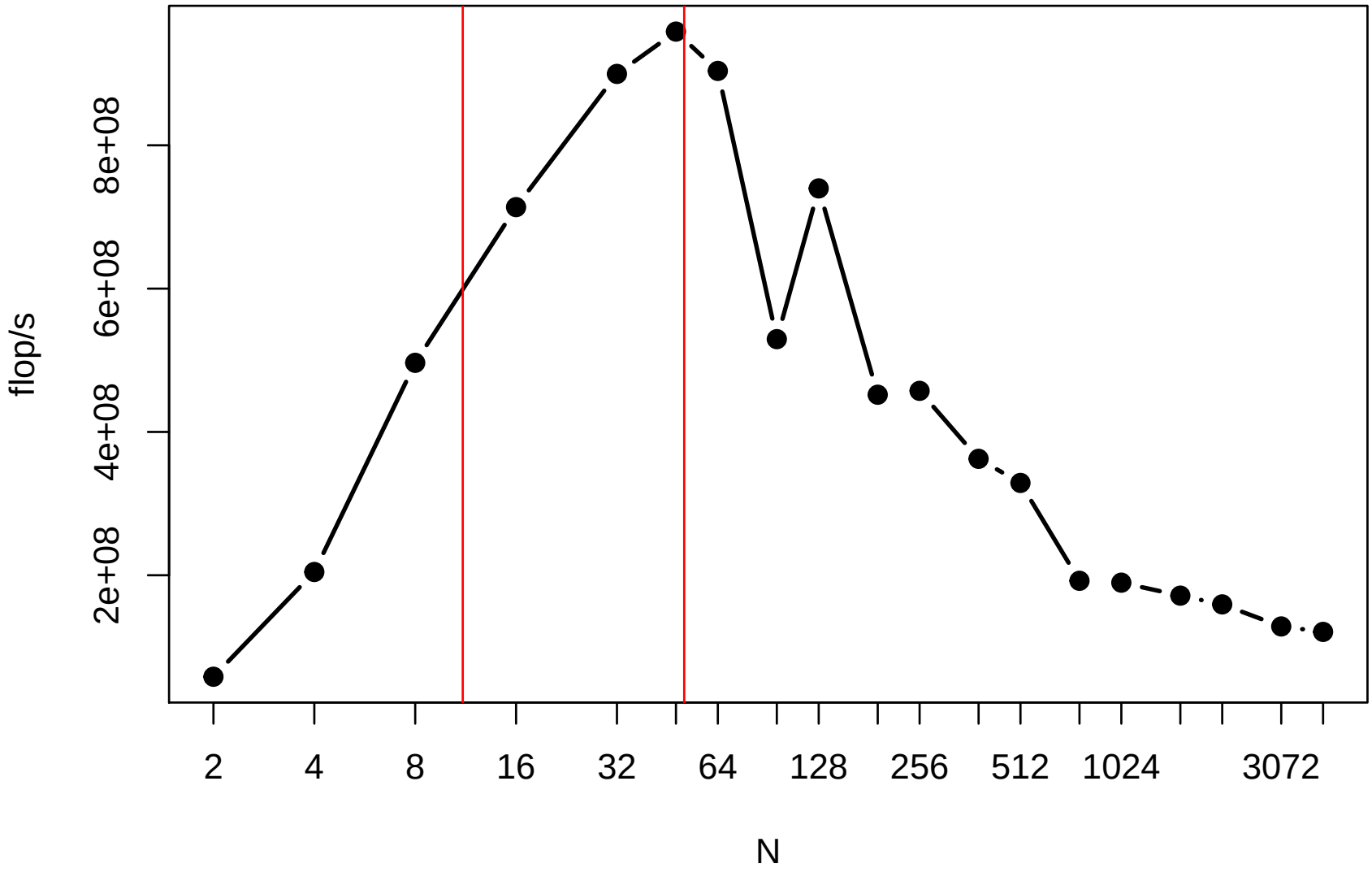
Calcul matriciel

- deux matrices $N \times N$, A et B, on veut calculer $C = AB$
- approche naïve : $C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$

```
for(int i=0;i<N;i++) {
  for(int j=0;j<N;j++) {
    double tmp=0;
    for(int k=0;k<N;k++) {
      tmp+=A[i][k]*B[k][j];
    }
    C[i][j]=tmp;
  }
}
```

- très lent : environ 11 secondes pour $N = 1024$ sur un Core 2 Duo P8600
- très mauvais : 190 Mflop/s

Calcul matriciel



Analyse

- Remarques :
  - dans le cache L1, ça fonctionne bien mieux : pour $N = 32$, 73 μs pour un calcul, soit 900 Mflop/s
  - dans le cache L2, la situation est intermédiaire : pour $N = 256$, 0.07 s pour un calcul, soit 457 Mflop/s
  - si on augmente la taille des données, la situation se maintient : pour $N = 2048$, un calcul prend 108 s, soit 160 Mflop/s
- le problème vient d'accès décalés :
  - `A[i][k]` est ok : décalage d'un `double` (8 octets) à chaque accès
  - `B[k][j]` est catastrophique : décalage de $8N$ octets à chaque accès
  - si $N > 64$ le *prefecth* L2 n'est même pas activable

Une solution

- On ne peut pas simplement échanger des boucles : A et B jouent des rôles symétriques
- Mais on peut transposer B : $C_{ij} = \sum_{k=1}^{N} A_{ik} B^T_{jk}$

```
double[][] D=new double[N][N];
for(int i=0;i<N;i++) {
  for(int j=0;j<N;j++) {
    D[j][i]=B[i][j];
  }
}
for(int i=0;i<N;i++) {
  for(int j=0;j<N;j++) {
    double tmp=0;
    for(int k=0;k<N;k++) {
      tmp+=A[i][k]*D[j][k];
    }
    C[i][j]=tmp;
  }
}
```

Résultats

- Fonctionne très bien : environ 2 secondes pour $N = 1024$ sur un Core 2 Duo P8600, soit 1070 Mflop/s (en Java !)
- Excellentes performances quand les caches sont actifs :
  - L1 : pour $N = 32$, 56 μs pour un calcul, soit 1 140 Mflop/s
  - L2 : pour $N = 360$, 0.06 s pour un calcul, soit 1 540 Mflop/s
- Excellente tenue de charge : pour $N = 2048$, 14 s pour un calcul, soit 1 200 Mflop/s
- Au prix d'une grosse occupation mémoire...
- Analyse :
  - il y a bien des accès non locaux pour fabriquer D, mais « seulement » N^2 (on pourrait utiliser des blocs)
  - ensuite les accès sont séquentiels
  - dans la solution naïve, il y a N^3 accès non locaux
- Moralité : il faut compter les accès non locaux

Utilisation réaliste

- A et B stockées sur disque
- lecture de A et B, calcul de $C = AB$ et sauvegarde du résultat
- deux versions :
  - classique
  - lecture de B avec transposition au vol (pas de stockage en double)
- résultats (Core 2 Duo P8600) :

N	disque (50Mo/s)	classique	transposée
1024	0.48 s	12.5 s	2.4 s
2048	1.92 s	112 s	16 s

temps de démarrage de la JVM inclus...

Java versus C

- on peut faire la même chose en C
- résultats (Core 2 Duo P8600) :

N	disque (50Mo/s)	classique	transposée
1024	0.48 s	8.2 s	1.8 s
2048	1.92 s	73 s	14.2 s

- conclusions :
  - bonnes performances de Java
  - Java souffre plus que C des problèmes d'accès
  - le C non optimisé pour les accès mémoires est beaucoup plus lent que le Java optimisé
  - le C optimisé est marginalement plus rapide que la Java optimisé

Préserver la mémoire

- On peut souvent utiliser l'astuce de la transposition, mais au prix d'une occupation mémoire importante
- Pour travailler sur place, on peut procéder par bloc :
  - comme dans la transposition, le problème vient de la distance entre `B[k][j]` et `B[k+1][j]`
  - on a donc intérêt à travailler sur un bloc de `B`
  - mais on peut aussi travailler sur un bloc de `C` pour favoriser une écriture dans des lignes de caches fixées

Produit matriciel par bloc

- le code

```
for (int oi = 0; oi < N; oi += M) {
  for (int oj = 0; oj < N; oj += M) {
    for (int ok = 0; ok < N; ok += M) {
      for (int i = oi; i < N && i < oi + M; i++) {
        for (int k = ok; k < N && k < ok + M; k++) {
          for (int j = oj; j < N && j < oj + M; j++) {
            C[i][j] += A[i][k] * B[k][j];
          }
        }
      }
    }
  }
}
```

- trois boucles externes : les blocs
- trois boucles internes : à l'intérieur d'un bloc, calcul classique

Code optimisé

- on peut sortir quelques calculs des boucles (fait pour la transposition)
- ce qui donne

```
for (int oi = 0; oi < N; oi += M) {
  int di = Math.min(oi+M, N);
  for (int oj = 0; oj < N; oj += M) {
    int dj = Math.min(oj+M, N);
    for (int ok = 0; ok < N; ok += M) {
      int dk = Math.min(ok+M, N);
      for (int i = oi; i < di; i++) {
        for (int k = ok; k < dk; k++) {
          double tmp = A[i][k];
          for (int j = oj; j < dj; j++) {
            C[i][j] += tmp * B[k][j];
          }
        }
      }
    }
  }
}
```

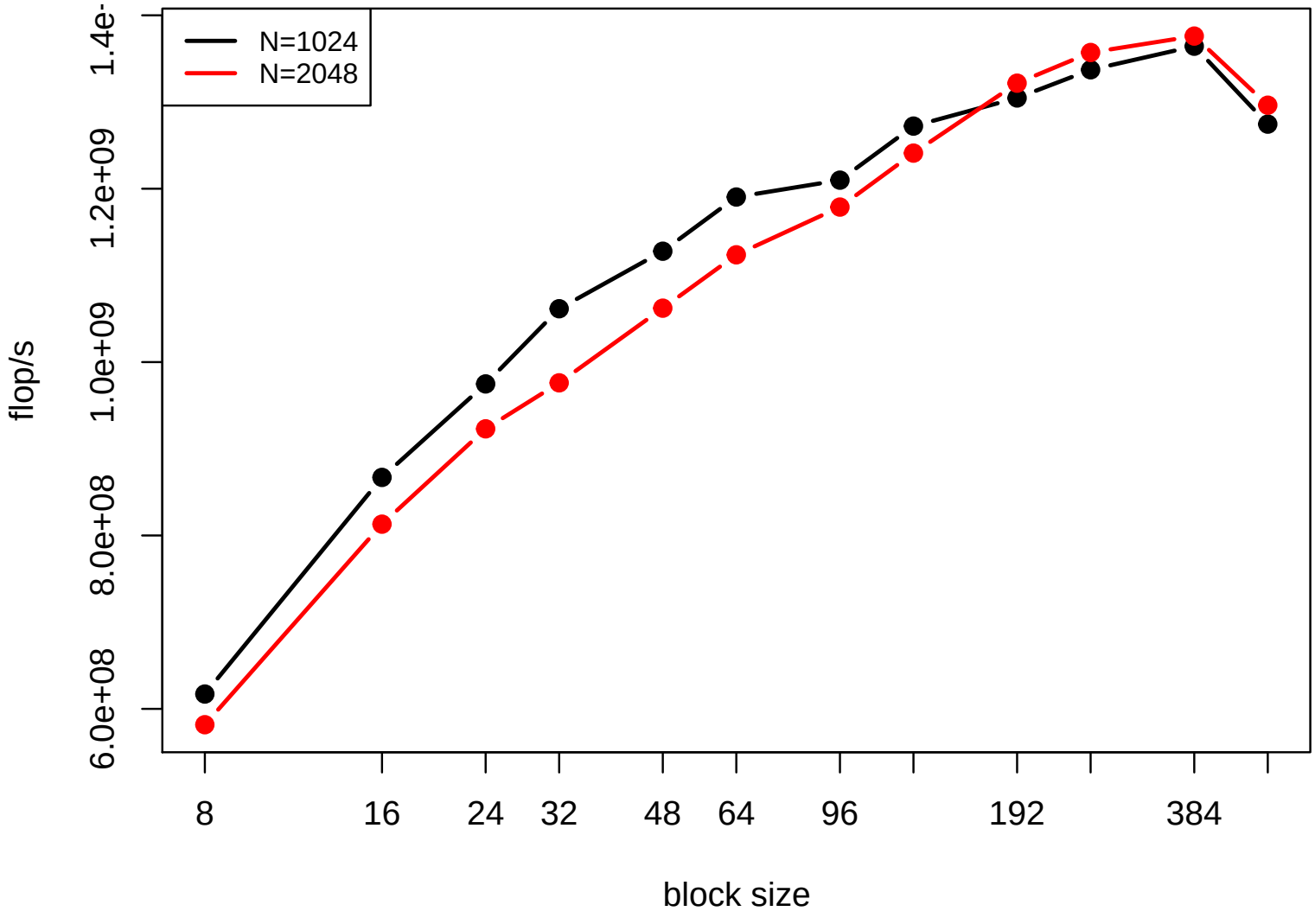
Résultats

- calculs seuls (Core 2 Duo P8600) pour $M = 32$

N	durée	puissance
1024	2.05 s	1050 Mflop/s
2048	17.60s	977 Mflop/s

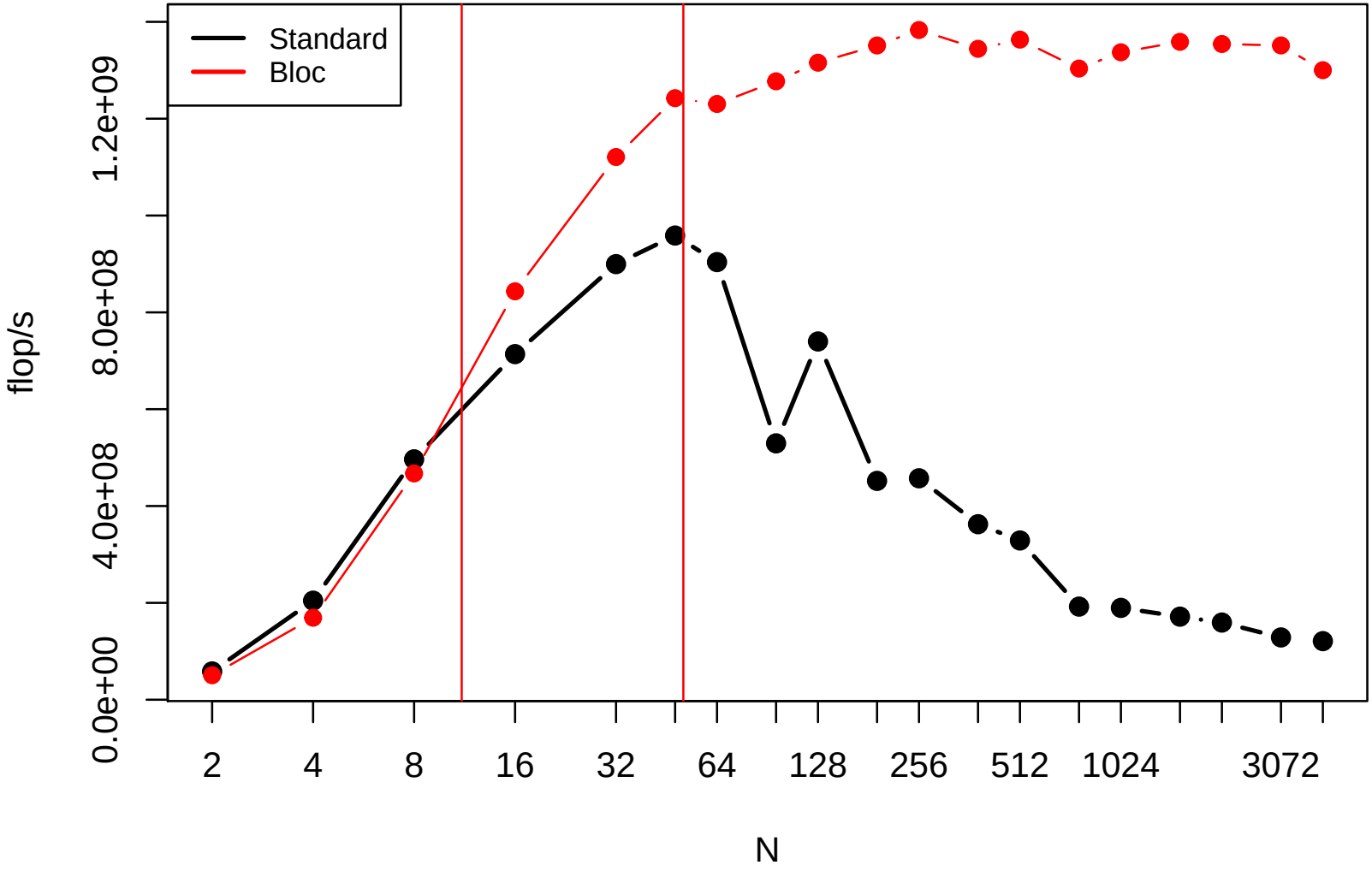
- taille des blocs :
  - 3 blocs de $M \times M$ `double`s : $24M^2$ octets
  - pour un cache L1 de 32 Ko : taille maximale de $M = 36$
  - le respect des alignements plaide pour $M = 32$
- prise en compte du cache L2 :
  - soit avec une hiérarchie de blocs : en trois niveaux de plus dans les boucles
  - soit en s'appuyant sur le préchargement de L2 vers L1 en prenant directement M plus grand que la limite théorique imposée par L1

Résultats



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo), pic à 1 330 Mflop/s

Résultats



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo) $M = 256$

Utilisation réaliste

- même contexte que pour la transposition :
  - A et B sur disque $\Rightarrow$ lecture
  - calcul de $C = AB$ et sauvegarde du résultat
- résultats (Core 2 Duo P8600) ($M = 384$ pour les blocs) :

N	disque (50Mo/s)	classique	B^T	blocs
1024	0.48 s	12.5 s	2.4 s	2 s
2048	1.92 s	112 s	16 s	14.2 s

temps de démarrage de la JVM inclus...

Utilisation réaliste

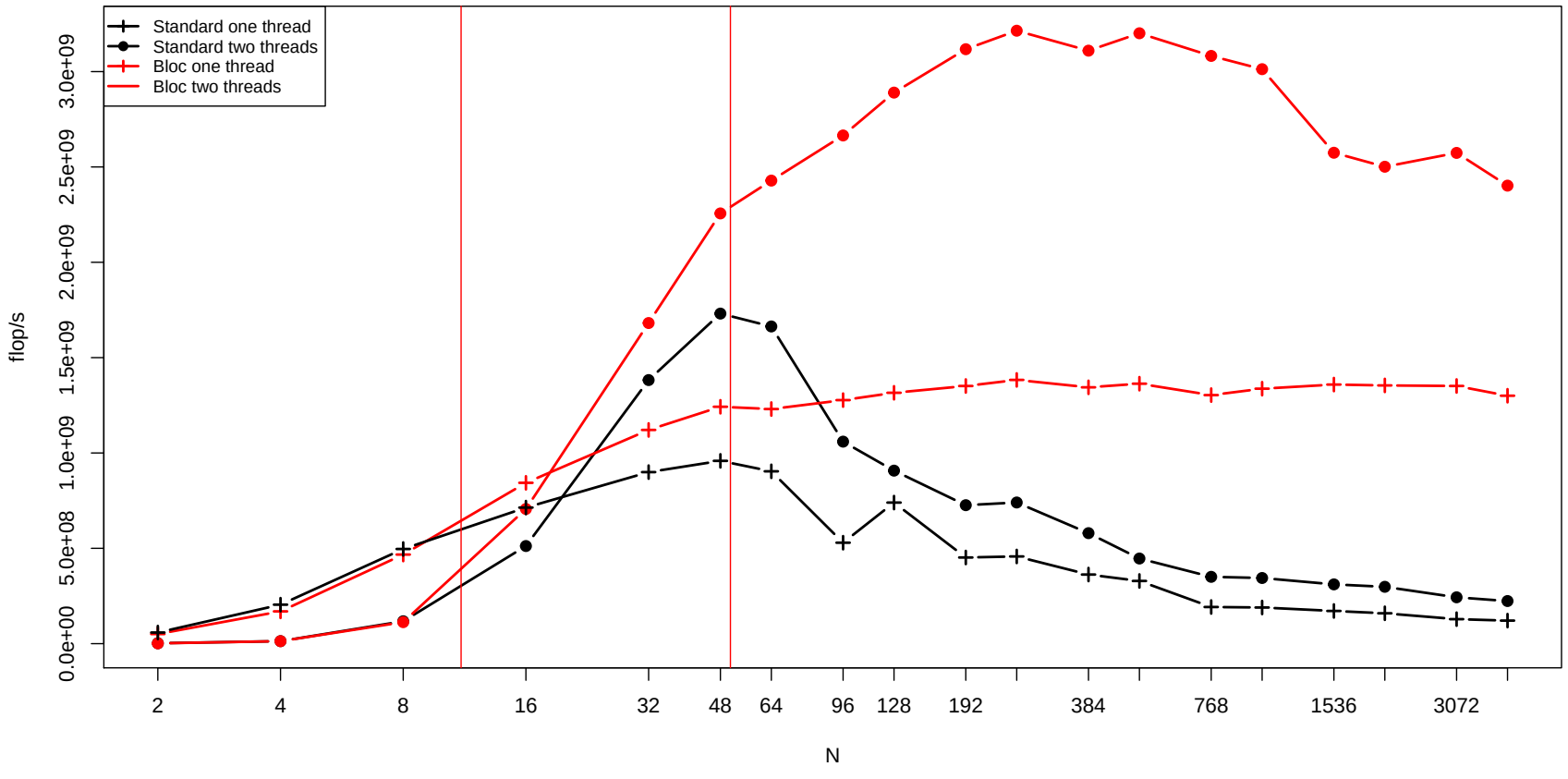
- même contexte que pour la transposition :
  - A et B sur disque $\Rightarrow$ lecture
  - calcul de $C = AB$ et sauvegarde du résultat
- résultats (Core 2 Duo P8600) ($M = 384$ pour les blocs) :

N	disque (50Mo/s)	classique	B^T	blocs
1024	0.48 s	12.5 s	2.4 s	2 s
2048	1.92 s	112 s	16 s	14.2 s

  temps de démarrage de la JVM inclus...
- résultats en C

N	disque (50Mo/s)	classique	B^T	blocs
1024	0.48 s	8.2 s	1.8 s	1.4 s
2048	1.92 s	73 s	14.2 s	13 s

Avec deux threads...



Sur un Core 2 Duo P8600 (2.4 GHz, 32 Ko, 3 Mo) $M = 256$

Cache-oblivious algorithm

- Le principe des blocs est très général
- Conduit à l'idée des *Cache-oblivious algorithm* : algorithmes efficaces pour toute structure hiérarchique de mémoire
- Méthode générale :
  - diviser pour régner
  - on découpe récursivement le problème en sous-problèmes jusqu'à une limite pour laquelle la latence peut être négligée
- Calcul matriciel, mais aussi FFT, tri, etc.
- cf `http://en.wikipedia.org/wiki/Cache-oblivious_algorithm`

Plan

Résumé

- La latence mémoire est très élevée :
  - 200 cyles CPU
  - impact très important sur certains algorithmes (un ou deux ordres de grandeur)
  - gâchis monumental de ressources
- Analyse algorithmique :
  - décomptes d'opérations et de mémoire insuffisants
  - il faut tenir des accès à la mémoire
- La localité est la clé :
  - seuls des accès locaux sont efficaces car aidés par les caches
  - local $\simeq$ dans la fenêtre de préchargement du cache L2 (512 octets)
  - on l'obtient en changeant l'ordre des calculs et/ou en travaillant par blocs et/ou en rangeant correctement les données en mémoire

Et ensuite ?

- En général :
  - utiliser des bibliothèques optimisées !
  - par ex. Atlas pour le calcul matriciel en C
    `http://math-atlas.sourceforge.net/`
  - COLT en java `http://acs.lbl.gov/~hoschek/colt/`

Et ensuite ?

- En général :
  - utiliser des bibliothèques optimisées !
  - par ex. Atlas pour le calcul matriciel en C `http://math-atlas.sourceforge.net/`
  - COLT en java `http://acs.lbl.gov/~hoschek/colt/`
- À haut niveau, on ne peut pas faire grande chose de plus :
  - déplier certaines boucles courtes (par exemple si le bloc choisi est très petit)
  - factoriser certains accès (type `C[i][j]` en sortant `C[i]` de la boucle interne)

Et ensuite ?

- En général :
  - utiliser des bibliothèques optimisées !
  - par ex. Atlas pour le calcul matriciel en C `http://math-atlas.sourceforge.net/`
  - COLT en java `http://acs.lbl.gov/~hoschek/colt/`
- À haut niveau, on ne peut pas faire grande chose de plus :
  - déplier certaines boucles courtes (par exemple si le bloc choisi est très petit)
  - factoriser certains accès (type `C[i][j]` en sortant `C[i]` de la boucle interne)
- En C (par ex. grâce aux *intrinsics* de gcc) :
  - utiliser les extensions vectorielles du CPU (MMX, SSE et AVX)
  - utiliser le *prefetch* explicite : demander au CPU de précharger certaines données
- En C++, magie noire possible (*template metaprogramming*)

Partage de ressources

- Un pan entier du problème n'a pas été traité : le partage des ressources
- Multi-coeurs :
  - caches partagés : contention
  - programmes parallèles : collaboration, synchronisation, protocole de cohérence des caches
- *Hyper threading* : exploitation des unités d'exécution pour ajouter des pseudo-coeurs
- NUMA :
  - *Non Uniform Memory Access*
  - dans les systèmes multi-processeurs
  - mémoire partagée mais avec une latence non uniforme
- Problème général : optimisation du code concurrent et parallèle

Outils

- Les CPU proposent des outils de surveillance des performances : *Performance Monitoring Unit*
- Beaucoup d'informations comme les défauts de cache, de TLB, etc.
- Accès sous linux :
  - Oprofile `http://oprofile.sourceforge.net/`
  - Pfmon `http://perfmon2.sourceforge.net/`
- Pour Java :
  - infrastructure complexe de supervision et d'instrumentation (JVM TI)
  - outil intégré : visualvm `https://visualvm.dev.java.net/`
  - support possible aussi par Oprofile

Pour en savoir plus...

- ce cours est basé sur « *What Every Programmer Should Know About Memory* » de Ulrich Drepper (développeur Red Hat), à lire absolument

  `http://people.redhat.com/drepper/cpumemory.pdf`

- et sur « *Intel 64 and IA-32 Architectures Optimization Reference Manual* »

  `http://www.intel.com/products/processor/manuals/`

- on trouve aussi des choses très intéressantes dans les articles sur ATLAS

  `http://math-atlas.sourceforge.net/`