Rapport HPC Lab 7 - Parallélisme des tâches - Samuel Roland

Première partie — Analyse des k-mers

Note: j'ai pris des fichiers de décimales de PI (`1k.txt` , `10k.txt` , `100k.txt` , `1m.txt`).

Baseline

Le code tel quel nous un résultat assez différent selon le nombre de `k` mais voici sur 100'000 décimales de PI le résultat. Plus `k` est élevé plus le temps est long. Nous sommes à **16.210s**.

```
> hyperfine './build/k-mer data/100k.txt 10'
  Time (mean ± σ):     16.210 s ±  1.538 s    [User: 15.869 s, System: 0.246 s]
```

Première lecture des éléments inefficaces

Ouverture des fichiers

On voit que l'ouverture et fermetures des fichiers n'a pas du tout été optimisé. Dans l'ensemble, on devrait pouvoir ouvrir le fichier, charger tout son contenu en mémoire, le fermer et ne plus avoir besoin de le réouvrir du tout.

Pour calculer la taille, il est ouvert puis refermé.

```
    FILE *file = fopen(input_file, "r");
    fseek(file, 0, SEEK_END);
    long file_size = ftell(file);
    fclose(file);
```

Puis cette boucle va appeler `read_kmer` presque pour chaque caractères et `read_kmer` va également ouvrir et fermer le fichier à chaque fois !! On a donc autant d'ouverture dque de nombre de caractères.

```
    for (long i = 0; i <= file_size - k; i++) {
        read_kmer(input_file, i, k, kmer);
        add_kmer(&table, kmer);
    }
```

Si on passe le `FILE*` dans la fonction `read_kmer` et on l'ouvre et ferme une seule fois, on passe à **14.314s**, déjà un peu mieux.

```
  Time (mean ± σ):     14.314 s ±  0.991 s    [User: 14.234 s, System: 0.020 s]
```

Flags de compilation

Un autre "low hanging fruit" est d'activer `-O3` dans les flags GCC. On arrive à **11.859s**.

```
  Time (mean ± σ):     11.859 s ±  1.608 s    [User: 11.758 s, System: 0.022 s]
```

Refactoring de read_kmer

La fonction `read_kmer` a pour but uniquement de lire `k` bytes et les stocker dans un buffer.

Sur l'implémentation actuelle on trouve plusieurs problèmes. Déjà le `kmer[k] = '\0';` pourrait se faire une seule fois en dehors de la boucle qui appele `read_kmer` , puisque notre `k` ne change pas sa longueur de la chaine extraite ne change pas. Ensuite, appeler `fgetc` pour chaque charactère n'est pas le plus efficace, c'est bufferisé mais on fera mieux d'utiliser `fread` pour lire directement le bon nombre de bytes.

```
void read_kmer(FILE *f, long position, int k, char *kmer) {
    fseek(f, position, SEEK_SET);
    for (int i = 0; i < k; i++) {
        int c = fgetc(f);
```

```
        if (c == EOF) {
            fprintf(stderr, "Error: Reached end of file before reading k-mer.\n");
            fclose(f);
            exit(1);
        }
        kmer[i] = (char) c;
    }
    kmer[k] = '\0';
}
```

Ainsi après avoir déplacé `kmer[k] = '\0';` juste après la définition de `kmer`, cette instruction est lancée une seule fois, et après avoir utilisé `fread`, on obtient ce code

```
void read_kmer(FILE *f, long position, int k, char *kmer) {
    fseek(f, position, SEEK_SET);
    int n = fread(kmer, sizeof(char), k, f);
    if (n != k) {
        fprintf(stderr, "Error: Reached end of file before reading k-mer.\n");
        fprintf(stderr, "position = %li, kmer = %s, n = %d\n", position, kmer, n);
        fclose(f);
        exit(1);
    }
}
```

Et le résultat du benchmark pour ce cas est légèrement plus difficile à analyser, selon les exécutions il y a plus ou moins 1s de plus ou moins. Mais la moyenne reste quand même en dessous et le range est plus petit (0.367s au lieu de 1.608s). Nous retiendrons le **10.399s** comme valeur “au milieu” des trois résultats.

```
  Time (mean ± σ):     10.000 s ±  0.367 s    [User: 9.942 s, System: 0.024 s]
  Time (mean ± σ):     10.399 s ±  0.322 s    [User: 10.346 s, System: 0.016 s]
  Time (mean ± σ):     11.073 s ±  0.656 s    [User: 11.006 s, System: 0.028 s]
```

Il reste encore une optimisation avec cette approche: Si il nous retourne pas la taille prévue `if (n != k)`, c’est qu’on est arrivé au bout du fichier, ce qui n’arrivera pas si le paramètre `position` est correctement géré à l’extérieur de l’appel. Ainsi, on peut retirer la condition et ajouter une hypothèse en commentaire.

Notre code devient ainsi très court.

```
void read_kmer(FILE *f, long position, int k, char *kmer) {
    fseek(f, position, SEEK_SET);
    fread(kmer, sizeof(char), k, f);
}
```

Etonnement retirer le branchement, s’avère pire en moyenne et on a nouveau cette large plage de plus ou moins… Je n’arrive pas à comprendre le sens de cette histoire, c’est étrange. J’ai fixé sur le coeur 3 pour limiter les instabilités.

```
Time (mean ± σ):     11.157 s ±  1.338 s    [User: 11.100 s, System: 0.016 s]
```

J’ai aussi inliné la fonction comme elle était trop courte et probablement déjà inliné par `-O2` et plus simple à gérer sur place.

fseek+fread à mmap

Je n’étais pas sûr du fonctionnement de fseek, s’il était possible de “déplacer le curseur” sans faire de syscall, j’ai fait un programme séparé pour pouvoir compter les syscalls séparement.

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    char kmer[100];
    FILE *file = fopen("100k.txt", "r");
    fseek(file, 0, SEEK_END);
    int size = ftell(file);
    fseek(file, 0, SEEK_SET);
    for (long i = 0; i <= size - 3; i++) {
        fseek(file, i, SEEK_SET);
        fread(kmer, sizeof(char), 3, file);
    }
    fclose(file);
```

```
    return 0;
}
```

On voit clairement que le syscall `lseek` est appelé à chaque appel de `fseek` puisqu'il y a le même nombre que de caractères.

```
> strace ./build/main &| grep lseek | wc -l
100000
```

Conclusion: se déplacer via le curseur de l'OS est probablement une très mauvaise idée comme il serait possible de déplacer un pointeur coté user space pour faire pareil. On pourrait lire tout le fichier une fois avec `fread` mais je vais partir sur `mmap` qui fait un chargement au fur et à mesure de l'usage.

Ainsi après mappage dans le pointeur `content`, on peut changer ce morceau

```
    for (long i = 0; i <= file_size - k; i++) {
        fseek(file, i, SEEK_SET);
        fread(kmer, sizeof(char), k, file);

        add_kmer(&table, kmer);
    }
```

avec celui-ci, même plus besoin de copier les k caractères, on peut juste les référencer et ajouter la taille `k` en paramètre de `add_kmer`

```
    for (long i = 0; i <= file_size - k; i++) {
        add_kmer(&table, content + i, k);
    }
```

A ce point, les temps des 2 versions avaient augmentés sur machine (sans explication particulière) j'ai relancé pour les 2 versions.

Avant changement, **13.156s**

```
Time (mean ± σ):     13.156 s ±  0.350 s    [User: 13.092 s, System: 0.021 s]
```

Après changement, on passe à **11.492s**

```
Time (mean ± σ):     11.492 s ±  0.131 s    [User: 11.445 s, System: 0.007 s]
```

On observe maintenant que les `lseek` ont bien été remplacé par quelques `mremap` bien moins nombreux.

```
read(3, "432472338884521534372725012858 97"..., 1696) = 1696
mmap(NULL, 100000, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f3e00c65000
mmap(NULL, 217088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3e00a26000
mremap(0x7f3e00a26000, 217088, 430080, MREMAP_MAYMOVE) = 0x7f3e009bd000
mremap(0x7f3e009bd000, 430080, 856064, MREMAP_MAYMOVE) = 0x7f3e008ec000
...
```

Optimisation de add_kmer par division en sous listes et propre memcmp

Je vais passer un coup de profiling pour voir où est passé le plus de temps dans `add_kmer`, pour voir dans quelle direction améliorer pour commencer. Je profile avec callgrind et seulement 3 et pas 10, parce que le temps de génération est trop lent pour `10`.

```
valgrind --tool=callgrind ./build/k-mer data/100k.txt 3
```

J'ai profilé le code du début du labo sans modification et j'ai à nouveau fait l'erreur de ne pas profiler au tout début, `add_kmer` prend **88.5%** du temps, par rapport à `read_kmer` seulement **11.3%*.

```
    299,998 ( 0.02%)      for (long i = 0; i <= file_size - k; i++) {
    499,990 ( 0.03%)          read_kmer(input_file, i, k, kmer);
194,420,601 (11.30%)  => src/main.c:read_kmer (99,998x)
    699,986 ( 0.04%)          add_kmer(&table, kmer);
1,523,940,811 (88.54%)  => src/main.c:add_kmer (99,998x)
          .               }
```

Avec le code modifié, on voit sans surprise que tout le travail est fait dans cette fonction (99.86% du temps passé dedans)

```
    299,999 ( 0.02%)      for (long i = 0; i <= file_size - k; i++) {
    799,984 ( 0.05%)          add_kmer(&table, content + i, k);
1,633,147,392 (99.86%)  => src/main.c:add_kmer (99,998x)
          .               }
```

Analysons maintenant qu'est-ce qui prend autant de temps dans `read_kmer`, c'est clairement la recherche de l'entrée à insérer qui prend le plus de temps.

```
149,261,228 ( 9.13%)      for (int i = 0; i < table->count; i++) {
348,035,874 (21.28%)          if (memcmp(table->entries[i].kmer, kmer, k) == 0) {
1,133,208,368 (69.29%)  => /usr/src/debug/glibc-2.40-25.fc41.x86_64/string/../sysdeps/x86_64/multiarch/memcmp-avx2-movbe.S:__m
        673 ( 0.00%)  => /usr/src/debug/glibc-2.40-25.fc41.x86_64/elf/../sysdeps/x86_64/dl-trampoline.h:_dl_runtime_resolve_xs
     98,998 ( 0.01%)              table->entries[i].count++;
          .                       return;
```

J'avais fait l'hypothèse à la première lecture que le `realloc` allait valoir la peine d'être optimisé pour éviter des réallocations, hors cette mesure montre qu'il n'a été appelé que 11fois et prend moins du 1% du temps.

```
  63    |       1,669 ( 0.00%)  => /usr/src/debug/glibc-2.40-25.fc41.x86_64/malloc/malloc.c:realloc (11x)
```

Benchmark actuel

```
Benchmark 1: taskset -c 3 ./build/k-mer data/100k.txt 10 > gen/100k.txt
  Time (mean ± σ):     12.122 s ±  0.349 s    [User: 12.073 s, System: 0.008 s]
  Range (min … max):   11.819 s … 12.503 s    3 runs
```

A cause de cette algorithme en $O(N)$, on doit constamment parcourir une grande partie de la liste avant de trouver l'entrée en question ou arriver tout au bout pour se rendre compte qu'elle n'existe pas. Plus `k` est grand, plus la probabilité est élevée que le mot ne se trouve pas dans la liste et que cette liste soit grande, et que le parcours se fasse en entier. Ce qui explique le temps augmenté plus `k` est grand.

On pourrait tenter d'implémenter ou d'importer une implémenation d'un arbre binaire, qui nous permettrait de faire de rechercher en $O(log(N))$, ou d'une hashmap pour avoir du $O(1)$ amorti. Nous allons essayer de découper en plusieurs morceaux la liste en fonction du premier charactère. J'ai séparé les 10 numéros, des lettres et du reste:

```c
#define LETTERS_COUNT 26
#define NUMBERS_COUNT 10
typedef struct {
    KmerTable letters[LETTERS_COUNT];// 26 KmerTable for words starting with the 26 alphabet letters (case insensitive)
    KmerTable numbers[NUMBERS_COUNT];// 10 KmerTable words starting with for numbers
    KmerTable rest;                  // all other words (starting with special chars)
} KmerTables;
```

Comme le fichier ici ne contient que des nombres, on peut s'imaginer une réduction d'environ 10 fois le temps total, puisque les listes sont remplis assez équitablement. La preuve avec ce dataset, le nombre d'entrées par premier caractère est très proche (exemple: il y a `10137` k-mers de avec `k=10` qui commencent par `1`, 9999 qui commencent par 0, 9907 qui commencent par 2, etc).

Le code nécessite plus de boucles d'initialisations et d'affichage des résultats, en plus d'une partie de "routing" vers la bonne sous table

```c
// Pick among one of the 37 available subtable
char firstChar = tolower(content[i]);
KmerTable *subtable = NULL;
if (firstChar >= '0' && firstChar <= '9') {
    subtable = tables.numbers + (firstChar - '0');// get the table of the first digit
} else if (firstChar >= 'a' && firstChar <= 'z') {
    subtable = tables.letters + (firstChar - 'a');// get the table of the first letter
} else {
    subtable = &tables.rest;
}
```

Et c'est effectivement de l'ordre 10x qu'on obtient maintenant **1.143s** !!

```
Time (mean ± σ):      1.143 s ±  0.011 s    [User: 1.133 s, System: 0.006 s]
```

Repassons un coup de profiling On voit que le code en plus autour de `add_kmer` n'est pas devenu significatif, on passe de 99.8% à 98.3% seulement.

```
166,969,262 (98.31%)  => src/main.c:add_kmer (99,998x)
```

C'est toujours la partie recherche et comparaison de strings qui prend la plus grande proportion.

```
35,107,139 (20.67%)          if (memcmp(table->entries[i].kmer, kmer, k) == 0) {
114,031,336 (67.14%)  => /usr/src/debug/glibc-2.40-25.fc41.x86_64/string/../sysdeps/x86_64/multiarch/memcmp-avx2-movbe.S:__mem
```

Je me demande si en évitant de comparer un caractère (le tout premier qui est déjà égal) dans le cas des lettres et nombres (pas du reste par contre), si on peut gagner du temps ou pas. Eh bien, il se trouve que non, on perd +30ms au lieu d'en gagner.

```
Time (mean ± σ):      1.179 s ±  0.006 s    [User: 1.170 s, System: 0.005 s]
```

Je ne garde donc pas ce changement.

J'ai essayé ensuite d'implémenter moi-même le `memcmp` pour éviter l'appel de la fonction et on obtient encore un léger gain de **168ms** avec **1.011s**

```
Time (mean ± σ):      1.011 s ±  0.008 s    [User: 1.002 s, System: 0.006 s]
```

Cette fois-ci, la stratégie de skipper le premier caractère évoquée plus haut fonctionne et nous donne encore un gain important de +100ms, nous sommes maintenant à **884.0ms**.

```
Time (mean ± σ):     834.5 ms ±   3.7 ms    [User: 827.2 ms, System: 5.0 ms]
```

Je me suis rendu compte une heure plus tard que la stratégie en elle-même est incorrecte, car cela considère `Za` et `za` pareil, puisque je mets en minuscule le première caractère pour choisir la `KmerTable` pour `z` et `Z`, je ne peux pas ignorer le premier caractère au final. J'ai rétabli la version précédente.

Optimisation des comparaisons de groupe de 8 et 4 caractères

En fait, la majorité du temps étant passée à comparer des strings, il est encore possible d'améliorer la performance en comparant plus d'un caractère à la fois ! Dans la même idée que le pattern du code SIMD, si `k=21`, on peut comparer 2 fois 8 bytes en les considérant comme deux `u_int64_t`, puis les bytes 17 à 21 c'est 4 bytes à considérer en `u_int32_t` puis finalement le dernier byte à gérer tout seul.

J'ai séparé le code en 2 cas, si `k<4`, rien ne sert d'ajouter de l'overhead de branches qui seront fausses dans tous les cas, je garde le code de départ. Pour tous les autres cas, on essaie de comparer à un multiple de 8 et 4 pour faire le plus possible de comparaisons en `u_int64_t` puis gérer le reste.

En théorie, si `k>=8`, on devrait avoir un peu moins de 8 fois d'accéleration, un peu moins parce qu'il y a un certain overhead non négligeabl de mise en place de ces boucles. Si `4 <= k < 8`, on devrait avoir également un facteur 4.

L'implémentation a eu plusieurs bug de logiques, qui ont heureusement pu être attrapée par mon système de tests anti-regression, qui vérifie que la sortie triée est la même qu'au départ sur tous les fichiers (`1k 10k 100k en.big`) et plusieurs `k` (`1 2 3 5 10 20 50 98`).

Etonnemment, on est loin d'avoir les 4x et 8x de gain…Pour `k=4`, on a 3.3x ce qui est le facteur le plus élevé, mais cela ne reste pas pour k=5, 6 et 7. Pour k=8 on est très très loin du facteur 8 (1.91x). On perd très légèrement pour `k=1` et `k=3` mais cela est largement insignifiant par le reste des gains.

k	File	Time before	Time after	Improvement factor
1	`1m.txt`	0.0030s	0.0032s	0.93x
2	`1m.txt`	0.0160s	0.0152s	1.04x
3	`1m.txt`	0.0827s	0.0864s	0.95x
4	`1m.txt`	0.8465s	0.2558s	3.30x
5	`100k.txt`	0.5429s	0.2583s	2.10x
6	`100k.txt`	0.9499s	0.4868s	1.95x
7	`100k.txt`	1.0180s	0.5329s	1.91x

k	File	Time before	Time after	Improvement factor
8	`100k.txt`	1.0192s	0.5322s	1.91x
10	`100k.txt`	1.0244s	0.5235s	1.95x
20	`100k.txt`	1.0113s	0.5211s	1.94x
55	`100k.txt`	1.0239s	0.5256s	1.94x
64	`100k.txt`	1.0177s	0.5190s	1.96x

J'ai testé aussi une suggestion de Copilot de forcer l'alignement sur 8 du champ `kmer` pour que les comparaisons sur 8 et 4 bytes puissent aussi être alignés. -> `char kmer[MAX_KMER] __attribute__((aligned(8)));`, mais cela ne change rien voir empire légèrement le gain.

Optimisation de l'usage du cache

Ma supposition pour le problème précédent est qu'on est limité quelque part par la bande passante mémoire et que notre usage du cache est peu efficace. En effet, de par la structure de `KmerEntry`, on a besoin de 104 bytes pour stocker une entrée, qui parfois ne fait que 9 bytes (si `k=4`, car 4 bytes pour le int + 4 pour les caractères + 1 pour le null byte).

```
#define MAX_KMER 100
typedef struct {
    unsigned count;
    char kmer[MAX_KMER];
} KmerEntry;
```

Cette taille large, très souvent beaucoup plus grande que nécessaire, fait que nos listes `entries` prennent beaucoup plus de place que nécessaire, ce qui demande plus de ligne de caches à charger au total. Si on arrive à compacter l'espace, cela nous fera encore gagner du temps.

Pour valider cette hypothèse, si on mesure les cache-miss, en L1 on est actuellement à 31% et L3 à 0.5%, pour `k=32` (pas d'événement supporté pour L2).

```
> sudo taskset -c 3 perf stat -e L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses ./build/k-mer data/100k.txt 3
     1,117,894,566      cpu_core/L1-dcache-loads/
       337,330,960      cpu_core/L1-dcache-load-misses/  #   30.18% of all L1-dcache accesses
       297,271,724      cpu_core/LLC-loads/
         1,451,345      cpu_core/LLC-load-misses/        #    0.49% of all LL-cache accesses
```

Si on teste le minimum nécessaire pour `k=32`, avec `#define MAX_KMER 33` au lieu de `100`, on passe à **12.26%** en L1 et **0.02%** en L3, et cela se traduit aussi par un gain de temps: d'un coup les facteurs augmentent de 1.5-2 fois dès qu'on dépasse `k=4` !

Attention: Dans tous les tableaux de cette section "Time before" se base sur le temps de la version avant la section précédente, pour voir si les facteurs augmentent vers les valeurs attendues.

k	File	Time before	Time after	Improvement factor
1	`1m.txt`	0.0029s	0.0031s	0.94x
2	`1m.txt`	0.0161s	0.0153s	1.05x
3	`1m.txt`	0.0816s	0.0801s	1.01x
4	`1m.txt`	0.8511s	0.2321s	3.66x
5	`100k.txt`	0.5577s	0.1379s	4.04x
6	`100k.txt`	0.9547s	0.2424s	3.93x
7	`100k.txt`	1.0197s	0.2627s	3.88x
8	`100k.txt`	1.0259s	0.2423s	4.23x
10	`100k.txt`	1.0290s	0.2441s	4.21x

Cet essai temporaire n'est pas viable puisque on ne veut pas être limité à 32 caractères pour `k`, mais bien rester à 100. Mais en fait, au lieu de copier constamment ces `k` bytes dans des buffers séparés trop grand, pourquoi ne pas juste stocker un pointeur vers le premier caractère et ne pas gérer le null byte ? Cela est possible comme la taille est toujours pareil (toujours `k`) et il suffit de garder le fichier ouvert jusqu'au bout pour pointer sur son contenu durant les analyses. Cela nous permet de passer de 104 bytes à 12 bytes.

Le gain est très net, les facteurs 4 restent stables et augmentent même vers en facteur 5. Au final, on est passé avec ce changement pour `k=64` de **519ms à 200ms**. Le facteur pour `k=10` est passé de **1.95x à 5.17x** !

k	File	Time before	Time after	Improvement factor
1	`1m.txt`	0.0030s	0.0033s	0.92x
2	`1m.txt`	0.0164s	0.0163s	1.00x
3	`1m.txt`	0.0820s	0.0875s	0.93x
4	`1m.txt`	0.8532s	0.1930s	4.42x
5	`100k.txt`	0.5515s	0.1180s	4.67x
6	`100k.txt`	0.9590s	0.1881s	5.09x
7	`100k.txt`	1.0261s	0.2038s	5.03x
8	`100k.txt`	1.0371s	0.2016s	5.14x
10	`100k.txt`	1.0405s	0.2009s	5.17x
20	`100k.txt`	1.0326s	0.2011s	5.13x
55	`100k.txt`	1.0408s	0.1998s	**5.20x**
64	`100k.txt`	1.0348s	0.2052s	5.04x

Si on pousse encore plus loin les tests avec le fichier `1m.txt`, l'overhead des boucles et branchements rajoutés prend de moins de place et le facteur grandit encore, jusqu'à **9.59x** au maximum.

k	File	Time before	Time after	Improvement factor
5	`1m.txt`	9.9946s	1.9949s	5.01x
6	`1m.txt`	127.8112s	13.3154s	**9.59x**
10	`1m.txt`	241.4931s	28.8358s	**8.37x**
20	`1m.txt`	243.9942s	29.0963s	8.38x
55	`1m.txt`	241.1902s	29.2314s	8.25x
64	`1m.txt`	238.9192s	28.8191s	8.29x

Problème de grosse latence sur ASCII random

Tous mes benchmark jusqu'à maintenant ont été réalisés sur des fichiers avec des chiffres, j'ai donc ignoré la possibilité d'avoir des fichiers avec des caractères ASCII autre qu'alphabétique et numéros. J'ai généré un fichier `ascii-1m.txt` (1million de caractères ASCII visibles, entre code le 32 et 126). Sans surprise au vu de la stratégie actuel, le fichier ascii est super long à traiter alors qu'il a la même taille.

```
Benchmark 1: taskset -c 3 ./build/k-mer data/1m.txt 2
  Time (mean ± σ):      16.2 ms ±   0.1 ms    [User: 15.2 ms, System: 0.8 ms]
Benchmark 1: taskset -c 3 ./build/k-mer data/ascii-1m.txt 2
  Time (mean ± σ):     566.5 ms ±   1.6 ms    [User: 564.6 ms, System: 0.8 ms]
```

J'ai donc simplifié ma structure de donnée qui était inutilement compliqué pour gagner un poil de mémoire, en la remplaçant par un tableau `KmerTable tables[ASCII_COUNT];`. Cela a également eu le bénéfice de simplifier une partie du code. On revient dans des grandeurs beaucoup plus proches. La différence restante est normale, puisqu'il y a 95 caractères différents au lieu de 10, ce qui fait des listes plus longues.

```
Benchmark 1: ./build/k-mer data/1m.txt 2
  Time (mean ± σ):      16.7 ms ±   0.6 ms    [User: 15.7 ms, System: 0.9 ms]
Benchmark 1: ./build/k-mer data/ascii-1m.txt 2
  Time (mean ± σ):      46.9 ms ±   1.5 ms    [User: 45.8 ms, System: 0.8 ms]
```

Conclusion de la préparation

J'ai travaillé sur de nombreux aspects afin d'améliorer le code existant mono-threadé, la majorité du temps était passée à chercher une entrée existante dans la liste et en comparant les caractères, cette partie a été optimisée un bon morceau. Je suis allé plus loin que demandé dans cette

partie parce que c'était vraiment fun de pousser le challenge si loin.

Voici un résumé des améliorations sur le fichier `100k.txt` avec `k=10`.

Section	Temps	Facteur relatif (à l'étape précédente)	Facteur absolu
Départ	**16.210s**	-	1
Ouverture des fichiers	14.314s	1.133x	1.133x
Flags de compilation	11.859s	1.207x	1.367x
Refactoring de read_kmer	11.492s	1.032x	1.411x
Optimisation de add_kmer par division en sous listes et propre memcmp	834.5ms	**13.77x**	19.43x
Optimisation des comparaisons de groupe de 8 et 4 caractères	523.5ms	1.595x	30.97x
Optimisation de l'usage du cache	**200.9ms**	2.605x	**80.70x**

On voit clairement que la section la plus efficace est la réduction de la la taille des listes, puisqu'elle a largement réduit l'espace de recherche. Si on voulait continuer encore, il faudrait passer à une hashmap pour diminuer encore largement le nombre d'éléments comparés. A noter aussi que les performances ne sont pas aussi élevées pour les fichiers avec des caractères ASCII aléatoire, puisque tous les caractères spéciaux sont placés dans une seule liste (sous `KmerTable rest`).

Benchmark général pour développement du multithreading

J'ai choisi de prendre les fichiers `1m.txt` et `ascii-1m.txt` et k = `2 5 50`. J'ai aussi généré un fichier `10m.en.txt` (10MB de notes prises en anglais sur des outils ou cours).

La répartition des caractères n'est pas équitable pour `10m.en.txt` comme le montre cette extrait de `k=1`. Cela va demander de gérer différement que les 2 fichiers précédents la répartition de la charge entre les threads.

```
X: 1650
Y: 6150
Z: 1250
[: 16450
\: 8850
]: 16450
^: 1450
_: 13850
`: 186950
a: 509200
b: 111350
c: 235200
d: 258550
e: 778500
```

k	File	Temps mono-threadé
2	`1m.txt`	0.0162s
5	`1m.txt`	2.2940s
50	`1m.txt`	27.9965s
2	`ascii-1m.txt`	0.0449s
5	`ascii-1m.txt`	4.1614s
50	`ascii-1m.txt`	4.2618s
2	`10m.en.txt`	0.1955s
5	`10m.en.txt`	4.6121s
50	`10m.en.txt`	23.3860s

Stratégie de parallélisation

L'analyse du fichier est parallélisable, puisque chaque kmer peut être recherché et inséré en parallèle, à condition qu'on fasse attention aux incréments du compteurs une fois l'entrée trouvée et durant les réallocations de listes. En effet, plusieurs race conditions peuvent arriver:

1. si 2 kmer identiques ont trouvés leur `KmerEntry` en même temps, le compteur pourrait n'avoir qu'un seul incrément au lieu de deux
2. si 2 kmer dans la même sous liste sont trouvées en même temps et insérés en bout de liste sur le dernier élément disponible, selon l'ordonnancement effectué, on pourrait avoir 2 insertions au même endroit (une serait perdue)
3. 1 kmer avec deux entrées successives, parce que la première entrée a été inséré juste après que l'autre chercher ait fini de parcourir la liste
4. durant une réallocation la liste n'est pas utilisable, sinon on court le risque d'utiliser la zone précédente ou de réallouer 2 fois et fuiter une des allocations ou d'avoir les paramètres de capacités et taille pas encore tout à fait mis à jour sur le nouveau bloc alloué…

Le parcours du fichier et des entrées en lecture peuvent se faire en parallèle, ça tombe bien puisque c'est justement la recherche qui prend le plus de temps. Par contre, l'écriture du compteur, insertions ou les réallocs doivent se faire sans que personne ne lise la liste associée.

Je considère que mon code tourne sur une machine de 10 coeurs (la mienne en a 12) avec 4 coeurs performance sur les numéros 0-3. Je vais donc démarrer 10 threads au maximum ou moins si cela ne fait pas sens.

Pour respecter ces contraintes, je vois deux approches possibles, en considérant 10 threads disponibles:

1. Découper le fichier en morceaux équivalents pour 9 threads. Avoir un thread qui gère tout ce qui écriture, et 9 autres travaillent à gérer les kmers sur leur section du fichier. Pour protéger la sous liste spécifique dans laquelle il y aura une modification, on retrouve ainsi un pattern vu en cours de PCO de lecteurs/rédacteurs, une sorte de mutex amélioré qui permet d'accéder à la liste en lecture à plusieurs thread lecteurs ou tout seul pour un thread rédacteur. On aurait besoin de ce système pour chaque sous liste. L'intérêt d'avoir géré les listes séparemment pour chaque premier caractère possible, permet de bloquer seulement une liste à la fois en écriture et bloquer temporairement uniquement une partie des threads lecteurs.
2. Découper l'espace des caractères ASCII pour grouper certaines sous listes ensembles. Chaque thread cherche ainsi les caractères dans une plage donnée (par ex. de `A` à `R`) en parcourant tout le fichier et ignorant tous les caractères hors de la plage. Une fois un kmer trouvé, il le gère lui-même. Chaque thread lit tout le fichier mais il n'y aucune opération d'écriture sur le fichier donc cela ne pose pas soucis. Comme chaque thread travaille sur un sous ensemble de liste bien différentes des autres, il n'y a aucun accès écriture et lecture à une zone partagé. Pas de section critique à gérer donc.

De par son indépendance et son absence complet de section critiques, je vais prendre l'option deux, cela jouera probablement en faveur de simplifier l'implémentation et j'imagine améliorer la performance.

J'utilise la commande suivante pour benchmarker, j'utilise (à nouveau 😃) `time -v` pour récupérer un pourcentage d'usage des CPUs globalement, pour avoir une idée d'à quel point on utilise bien nos 10 coeurs. Je pin les threads sur les coeurs 1 à 11 (comme il y a 11 threads qui vont tourner (thread principal + 10 démarrés).

```
hyperfine --max-runs $runs "taskset -c 1-11 /usr/bin/time -v ./build/k-mer data/$file $count |& grep 'Percent of CPU' > percen
```

Le pourcentage espéré est de 1000% (100% sur 10 coeurs). Le code de stratégie a été implémentée dans `calculateThreadRepartitionStrategy()` afin de simplifier la relecture.

Parallélisation avec répartition statique

On voit que le temps n'a fait qu'augmenter pour `1m.txt`, c'est normal, aucun thread n'a fait du travail utile sauf le seul qui avait la plage des numéros. On a plus de temps à cause de l'overhead de création des threads. `ascii-1m.txt` a été géré plus rapidement, entre 2 et 2.5 fois plus rapidement, c'est déjà un gain mais cela me parait bizarre que cela soit si petit au vu des 10 threads. Finalement le facteur de 1.6-1.7 pour `10m.en.txt` s'explique par le fait que les caractères sont inéquitablement répartis, une partie des threads travaillent probablement beaucoup plus que les autres.

k	File	Time before	Time after	CPU usage	Improvement factor
2	`1m.txt`	0.0162s	0.0183s	146%	0.88x
5	`1m.txt`	2.2940s	2.3165s	100%	0.99x
50	`1m.txt`	27.9965s	28.3732s	99%	0.98x
2	`ascii-1m.txt`	0.0449s	0.0213s	531%	2.10x
5	`ascii-1m.txt`	4.1614s	1.4788s	524%	2.81x
50	`ascii-1m.txt`	4.2618s	1.5908s	603%	2.67x
2	`10m.en.txt`	0.1955s	0.1149s	450%	1.70x
5	`10m.en.txt`	4.6121s	2.5912s	292%	1.77x

k	File	Time before	Time after	CPU usage	Improvement factor
50	`10m.en.txt`	23.3860s	14.5286s	258%	1.60x

Parallélisation avec répartition dynamique

Pour s'adapter à ces différents types de données il est nécessaire de découper l'espace des caractères ASCII afin de minimiser la différence du nombre de kmers à gérer par liste entre chaque thread.

Un élément très intéressant que je n'avais pas remarqué et que `k=1` est toujours très rapide, faire un premier tour avec `k=1` peut tout à fait valoir la peine.

```
Benchmark 1: ./build/k-mer ./data/ascii-1m.txt 1
  Time (mean ± σ):       3.7 ms ±   0.4 ms    [User: 3.1 ms, System: 0.4 ms]
```

Cela nous donne des compteurs exactes du nombre de fois que se trouve chaque caractère, permettant ainsi de faire des groupes de caractères contigus. Pour choisir combien de caractère mettre dans ces groupes, on se base sur la longueur des listes, en les sommant on s'arrête juste avant ou après dépasser le nombre `taille du fichier / nombre de threads`. Il faut évidemment que cette préparation soit négligeable pour ne pas créer d'overhead significatif sur le programme.

Pour le fichier `1m.txt`, on va pouvoir ainsi compléter ignorer les caractères non numérique comme on sait qu'il n'y a aucun et dédier un thread par chiffre!

J'ai ajouté un flag `--strategy` à la fin (ce qui reste rétrocompatible en terme d'interface), pour afficher la stratégie de répartition des caractères ASCII entre threads au lieu de lancer l'exécution.

```
./build/k-mer data/ascii-gen/ascii-1m.clean.txt 5 --strategy
```

Cette stratégie ne prend comme imaginé pas vraiment plus de temps que le premier tour avec `k=1`.

```
> hyperfine './build/k-mer data/ascii-gen/ascii-1m.clean.txt 5 --strategy' -i
Time (mean ± σ):       3.4 ms ±   0.5 ms    [User: 2.8 ms, System: 0.6 ms]
```

Si on inspecte la stratégie pour `ascii-1m.txt`, les caractères sont bien répartis, un par thread, même si malheureusement le double du travail devra être fait par le dernier thread le numéro 9 étant doublement présent.

```
Printing multi-threading strategy on 10 threads
Thread 0: [48 '0'; 48 '0'] -> 1 different chars (99959 concrete chars - 10.00%)
Thread 1: [49 '1'; 49 '1'] -> 1 different chars (99758 concrete chars - 9.98%)
Thread 2: [50 '2'; 50 '2'] -> 1 different chars (100026 concrete chars - 10.00%)
Thread 3: [51 '3'; 51 '3'] -> 1 different chars (100229 concrete chars - 10.02%)
Thread 4: [52 '4'; 52 '4'] -> 1 different chars (100230 concrete chars - 10.02%)
Thread 5: [53 '5'; 53 '5'] -> 1 different chars (100359 concrete chars - 10.04%)
Thread 6: [54 '6'; 54 '6'] -> 1 different chars (99548 concrete chars - 9.95%)
Thread 7: [55 '7'; 55 '7'] -> 1 different chars (99800 concrete chars - 9.98%)
Thread 8: [56 '8'; 56 '8'] -> 1 different chars (99985 concrete chars - 10.00%)
Thread 9: [57 '9'; 57 '9'] -> 1 different chars (200212 concrete chars - 20.02%)
```

Si on inspecte la stratégie pour `ascii-1m.txt`, le temps est plutôt bien réparti

```
Printing multi-threading strategy on 10 threads
Thread 0: [32 ' '; 41 ')'] -> 10 different chars (105113 concrete chars - 10.51%)
Thread 1: [42 '*'; 50 '2'] -> 9 different chars (94920 concrete chars - 9.49%)
Thread 2: [51 '3'; 59 ';'] -> 9 different chars (94803 concrete chars - 9.48%)
Thread 3: [60 '<'; 69 'E'] -> 10 different chars (105389 concrete chars - 10.54%)
Thread 4: [70 'F'; 78 'N'] -> 9 different chars (95150 concrete chars - 9.51%)
Thread 5: [79 'O'; 88 'X'] -> 10 different chars (105246 concrete chars - 10.52%)
Thread 6: [89 'Y'; 98 'b'] -> 10 different chars (105150 concrete chars - 10.51%)
Thread 7: [99 'c'; 107 'k'] -> 9 different chars (94875 concrete chars - 9.49%)
Thread 8: [108 'l'; 117 'u'] -> 10 different chars (104724 concrete chars - 10.47%)
Thread 9: [118 'v'; 126 '~'] -> 9 different chars (105052 concrete chars - 10.51%)
```

Note: la colonne Time before fait référence à la dernière version single thread.

k	File	Time before	Time after	CPU usage	Improvement factor

k	File	Time before	Time after	CPU usage	Improvement factor
2	`1m.txt`	0.0162s	0.0128s	520%	1.26x
5	`1m.txt`	2.2940s	0.5791s	778%	3.96x
50	`1m.txt`	27.9965s	9.3944s	830%	2.98x
2	`ascii-1m.txt`	0.0449s	0.0201s	594%	2.23x
5	`ascii-1m.txt`	4.1614s	1.5590s	806%	2.66x
50	`ascii-1m.txt`	4.2618s	1.6562s	741%	2.57x
2	`10m.en.txt`	0.1955s	0.1282s	470%	1.52x
5	`10m.en.txt`	4.6121s	2.1537s	472%	2.14x
50	`10m.en.txt`	23.3860s	13.0733s	313%	1.78x

Les cache misses restent plutôt limités, le fichier de 1MB tient probablement entièrement en cache L3 (qui fait 12MB) et

```
> sudo taskset -c 1-11 perf stat -e L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses ./build/k-mer data/1m.txt
   301,651,971,638      cpu_core/L1-dcache-loads/                                               (24.52%)
     8,911,530,149      cpu_core/L1-dcache-load-misses/  #    2.95% of all L1-dcache accesses   (24.52%)
       852,664,357      cpu_core/LLC-loads/                                                     (24.52%)
         7,461,538      cpu_core/LLC-load-misses/        #    0.88% of all LL-cache accesses    (24.52%)
```

Le dernier cas est étonnement peu performant, en affichant la stratégie on voit que le thread 0 a le double de travail des autres threads et qu'il travaille sur les caractère espace et retour à la ligne, qui sont très présents, ce qui signifie des longues listes respectives.

```
9 '     ': 6550
10 '
': 251250
...
30 '': 0
31 '': 0
32 ' ': 1662450
Printing multi-threading strategy on 10 threads
Thread 0: [9 '  '; 32 ' '] -> 24 different chars (1920250 concrete chars - 18.34%)
Thread 1: [33 '!'; 60 '<'] -> 28 different chars (952500 concrete chars - 9.10%)
Thread 2: [61 '='; 97 'a'] -> 37 different chars (1176000 concrete chars - 11.23%)
Thread 3: [98 'b'; 100 'd'] -> 3 different chars (605100 concrete chars - 5.78%)
Thread 4: [101 'e'; 102 'f'] -> 2 different chars (913100 concrete chars - 8.72%)
Thread 5: [103 'g'; 107 'k'] -> 5 different chars (916250 concrete chars - 8.75%)
Thread 6: [108 'l'; 110 'n'] -> 3 different chars (1012300 concrete chars - 9.67%)
Thread 7: [111 'o'; 114 'r'] -> 4 different chars (1118900 concrete chars - 10.69%)
Thread 8: [115 's'; 116 't'] -> 2 different chars (1091550 concrete chars - 10.42%)
Thread 9: [117 'u'; 126 '~'] -> 10 different chars (938450 concrete chars - 8.96%)
```

En affichant les timestamps de fin, on voit bien que ce thread 0 finit en dernier, bien après les autres.

```
> time ./build/k-mer data/10m.en.txt 50
15:56:09:340 - Thread 140333950166720 is done (from '!' to '<')
15:56:09:599 - Thread 140333916595904 is done (from 'l' to 'n')
15:56:09:609 - Thread 140333820143296 is done (from 'u' to '~')
15:56:09:794 - Thread 140333908203200 is done (from 'o' to 'r')
15:56:09:978 - Thread 140333933381312 is done (from 'b' to 'd')
15:56:10:679 - Thread 140333941774016 is done (from '=' to 'a')
15:56:10:825 - Thread 140333924988608 is done (from 'g' to 'k')
15:56:12:010 - Thread 140333828536000 is done (from 's' to 't')
15:56:13:149 - Thread 140333788690112 is done (from 'e' to 'f')
15:56:20:522 - Thread 140333958559424 is done (from '    ' to ' ')
```

Cette stratégie a cette limite de ne pas être la plus adaptée si un caractère est très présent par rapport à d'autres comme ici 1.6millions d'espaces.

Parallélisation avec priorisation de certain threads sur des performances cores

La plus grande augmentation de facteur de 2.98 à 3.78 pour le `1m.txt` sur `k=50`. Presque tous les pourcentage de CPU sont montés.

Note: la colonne Time before fait encore à la dernière version single thread.

k	File	Time before	Time after	CPU usage	Improvement factor
2	`1m.txt`	0.0162s	0.0146s	430%	1.10x
5	`1m.txt`	2.2940s	0.5462s	845%	4.20x
50	`1m.txt`	27.9965s	7.3970s	825%	**3.78x**
2	`ascii-1m.txt`	0.0449s	0.0227s	566%	1.97x
5	`ascii-1m.txt`	4.1614s	1.2541s	812%	3.31x
50	`ascii-1m.txt`	4.2618s	1.5422s	831%	2.76x
2	`10m.en.txt`	0.1955s	0.1335s	534%	1.46x
5	`10m.en.txt`	4.6121s	1.8788s	562%	2.45x
50	`10m.en.txt`	23.3860s	11.9616s	380%	1.95x

Gestion des petits fichiers ou petits k

Les benchmarks précédents se sont concentrés sur des fichiers de grande taille, le multi-threading a été désactivé uniquement pour `k=1`. Hors les fichiers de petites tailles sont traités beaucoup plus rapidement que les grands, même avec un grand `k=600`, ce qui signifie que l'overhead des threads est parfois contre productive.

Comme on arrive en dessous la milliseconde, hyperfine nous averti de désactiver le shell si possible (car il n'arrive pas estimer le temps de démarrage de celui-ci) ce qu'on fait avec l'option `-N`. Au début de mon analyse j'utilisais les commandes suivantes. Le problème est que la commande `taskset`, `time` et `grep` prennent un temps de l'ordre du dizième de millisecondes, ce qui impactait le résultat !

```
hyperfine --max-runs $runs -N "taskset -c 3 ./build/$beforebin data/$file $count"
hyperfine --max-runs $runs "set -o pipefail; taskset -c 0-9 /usr/bin/time -v ./build/k-mer data/$file $count |& grep 'Percent
```

J'ai changé par ces 2 commandes simplifiée pour cette partie, `taskset` étant à l'extérieur et ayant retiré le calcul du pourcentage de CPU, on retrouve des statistiques qui font sens.

```
taskset -c 3 hyperfine --max-runs $runs -N "./build/$beforebin data/$file $count" --export-json base.$file.$count.out.json
taskset -c 0-9 hyperfine --max-runs $runs -N "./build/k-mer data/$file $count" --export-json new.$file.$count.out.json
```

Résumé des améliorations avant et après parallélisation. L'overhead des threads et peut-être d'autres choses à ralentis par 2-3 tout une partie des cas. Ce qui est étonnant c'est le **0.59x** pour k=1 sans parallélisation…

k	File	Time before	Time after	Improvement factor
1	`1k.txt`	0.0003s	0.0005s	**0.59x**
2	`1k.txt`	0.0003s	0.0008s	0.36x
3	`1k.txt`	0.0004s	0.0009s	0.44x
10	`1k.txt`	0.0004s	0.0009s	0.45x
50	`1k.txt`	0.0004s	0.0009s	0.47x
100	`1k.txt`	0.0004s	0.0009s	0.46x
600	`1k.txt`	0.0004s	0.0009s	0.40x
1	`10k.txt`	0.0003s	0.0005s	0.67x
2	`10k.txt`	0.0005s	0.0009s	0.51x
3	`10k.txt`	0.0013s	0.0011s	**1.16x**
4	`10k.txt`	0.0019s	0.0018s	1.04x
5	`10k.txt`	0.0029s	0.0022s	1.27x
6	`10k.txt`	0.0030s	0.0024s	1.26x
7	`10k.txt`	0.0028s	0.0023s	1.22x

k	File	Time before	Time after	Improvement factor
8	`10k.txt`	0.0027s	0.0025s	1.08x
9	`10k.txt`	0.0028s	0.0024s	1.13x
10	`10k.txt`	0.0027s	0.0022s	1.18x
100	`10k.txt`	0.0027s	0.0024s	1.09x
600	`10k.txt`	0.0027s	0.0027s	0.99x
1	`100k.txt`	0.0005s	0.0009s	0.57x
2	`100k.txt`	0.0017s	0.0021s	0.84x
3	`100k.txt`	0.0090s	0.0038s	2.37x
4	`100k.txt`	0.0195s	0.0084s	2.32x
5	`100k.txt`	0.1154s	0.0383s	3.01x
6	`100k.txt`	0.1884s	0.0590s	3.19x
7	`100k.txt`	0.1986s	0.0619s	3.21x
8	`100k.txt`	0.1983s	0.0692s	2.86x
9	`100k.txt`	0.1977s	0.0705s	2.80x
10	`100k.txt`	0.1978s	0.0695s	2.84x
11	`100k.txt`	0.1977s	0.0700s	2.82x
15	`100k.txt`	0.1983s	0.0739s	2.68x
20	`100k.txt`	0.1984s	0.0704s	2.82x
30	`100k.txt`	0.1984s	0.0716s	2.77x
40	`100k.txt`	0.1987s	0.0695s	2.85x
50	`100k.txt`	0.1985s	0.0711s	2.79x
80	`100k.txt`	0.1991s	0.0731s	2.72x
90	`100k.txt`	0.1989s	0.0741s	2.68x
100	`100k.txt`	0.2000s	0.0749s	2.67x
600	`100k.txt`	0.2003s	0.0788s	2.54x

Je vais juste désactiver la parallélisation pour les fichiers en dessous de 10k et pour k<3 puisque c'est à partir de 10k et k=3 que l'on repasse à un. Les résultats ne changent que très peu, les facteurs sont remontés de 2x ou 1.5x. Par exemple on passe de 0.44x à 0.72x pour `k=3` et `1k.txt`

k	File	Time before	Time after	Improvement factor
1	`1k.txt`	0.0003s	0.0005s	0.66x
2	`1k.txt`	0.0003s	0.0005s	0.68x
3	`1k.txt`	0.0004s	0.0006s	0.72x
4	`1k.txt`	0.0004s	0.0007s	0.61x
5	`1k.txt`	0.0004s	0.0007s	0.61x
6	`1k.txt`	0.0004s	0.0007s	0.59x
7	`1k.txt`	0.0004s	0.0006s	0.66x
8	`1k.txt`	0.0004s	0.0006s	0.65x
9	`1k.txt`	0.0004s	0.0006s	0.64x
10	`1k.txt`	0.0004s	0.0006s	0.65x
11	`1k.txt`	0.0004s	0.0007s	0.58x

k	File	Time before	Time after	Improvement factor
15	`1k.txt`	0.0004s	0.0006s	0.70x
20	`1k.txt`	0.0004s	0.0006s	0.59x
30	`1k.txt`	0.0004s	0.0006s	0.56x
40	`1k.txt`	0.0004s	0.0007s	0.61x
50	`1k.txt`	0.0004s	0.0007s	0.56x
80	`1k.txt`	0.0004s	0.0006s	0.62x
90	`1k.txt`	0.0004s	0.0007s	0.56x
100	`1k.txt`	0.0004s	0.0006s	0.57x
600	`1k.txt`	0.0003s	0.0007s	0.49x
1	`10k.txt`	0.0003s	0.0006s	0.55x
2	`10k.txt`	0.0004s	0.0007s	0.63x
3	`10k.txt`	0.0012s	0.0011s	1.03x
4	`10k.txt`	0.0018s	0.0018s	0.99x
5	`10k.txt`	0.0026s	0.0023s	1.15x

En fait la différence est super minime, seulement 0.2ms, peut-être que cela est du à la gestion de plus de liste qu'auparavant.

Conclusion parallélisation

Même en exécutant sur un fichier encore plus grand `1g.en.txt` (1GB d'anglais), on obtient `129.13s` avant et `50.24s` après, ce qui ne fait que **2.58** fois de gain.

La parallélisation **a permis de gagner un facteur 1 à 4** selon les cas. Nous avons atteint au maximum **830%** d'usage de CPU (très souvent loin de la parallélisation parfaite des 1000%). Le challenge était surtout lié à la volonté de garder les threads complètement indépendant pour éviter l'usage de mutex, la difficulté a été et reste dans la distribution de la charge de travail tout en gardant cette indépendance. J'ai essayé de jouer un peu avec plus de threads que 10 mais cela ne donnait rien de mieux.

On aurait pu dynamiquement décider de couper en morceaux des listes avec le plus de lettres prévue (refaire ce système de sous listes, mais pour les 2ème caractères ou alors découper en 2 la liste avec pour le 2ème caractère de `0->64` et l'autre de `65->127`). Ce qui aurait permis de mettre plus de threads à la tâche sur la même lettre quand celle-ci est plus présente que le reste. Il aurait fallu adapter un poil le code d'affichage et d'initialisations, cela n'aurait pas été difficile, le plus compliqué aurait été d'arriver à décider du découpage et du nombre de threads pour rester équilibré avec le reste (ne pas mettre 6 threads sur l'espace et plus que 4 pour tout le reste par exemple).

La mention de structure de données permettant un accès en moins que $O(N)$ mentionnée dans la préparation, nous aurait imposé d'autres contraintes, ce découpage en liste nous a sans le voir bien arrangé pour garder une indépendance des threads. Après réflexion avec Aubry, il semble qu'avec un hashmap il aurait été possible d'avoir une hashmap par threads, tout en restant avec un découpage de l'espace des caractères. Il n'y aurait pas eu besoin de fusion, juste d'afficher tous les résultats de toutes les hashmaps l'une après l'autre à la fin.

Deuxième partie — Activité DTMF

Baseline de départ

```
> hyperfine -r 3 -N 'build/dtmf_encdec_buffers_single decode verylong2.wav'
  Time (mean ± σ):      1.649 s ±  0.009 s    [User: 1.226 s, System: 0.418 s]
```

1.649s pour le fichier `verylong2.wav` qui correspond à la sortie de `txt/verylong2.txt` qui contient des répétitions de l'alphabet sur 8000 caractères.

Parallélisation

Un rapide coup de `callgrind` nous rappelle que la zone la plus chaude est la fonction `get_near_score`, nous allons nous concentrer là dessus en premier.

```
              .           inline float get_near_score(const float *audio_chunk, float *reference_tone) {
              .               double sum = 0;
       157,289 ( 0.00%)
              .               for (sf_count_t i = 0; i < TONE_SAMPLES_COUNT; i++) {
 4,162,338,807 (37.42%)           sum += fabs(audio_chunk[i] - reference_tone[i]);
 6,936,444,900 (62.35%)       }
              .               return sum / TONE_SAMPLES_COUNT;
       314,578 ( 0.00%)   }
```

Première implémentation, avec une parallélisation basique,

```
inline float get_near_score(const float *audio_chunk, float *reference_tone) {
    double sum = 0;
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        double local_sum = 0;
        int nbThreads = omp_get_num_threads();
        for (sf_count_t i = id; i < TONE_SAMPLES_COUNT; i += nbThreads) {
            local_sum += fabs(audio_chunk[i] - reference_tone[i]);
        }
#pragma omp critical
        sum += local_sum;
    }
    return sum / TONE_SAMPLES_COUNT;
}
```

```
> hyperfine -r 3 -N 'build/dtmf_encdec_buffers decode verylong2.wav'
Benchmark 1: build/dtmf_encdec_buffers decode verylong2.wav
  Time (mean ± σ):      4.270 s ±  0.070 s    [User: 8.844 s, System: 13.157 s]
  Range (min … max):    4.192 s …  4.325 s    3 runs
```

Au lieu de `1.649s` on est bien pire avec `4.270s`…

2ème implémentation, cette fois-ci avec un tableau de somme intermédiaire, pour éviter de devoir avoir une section critique. Le thread principal fait la somme à la fin une fois que tous les threads ont terminés. Le problème du “false sharing” est minime comme il n’y aura autant d’écriture dans ce tableau partagé `sums` que de threads.

```
inline float get_near_score(const float *audio_chunk, float *reference_tone) {
    double sum = 0;
    double sums[20];        // a bit more place than necessary
    int globalNbThreads = 0;// initialized by thread 0
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        double local_sum = 0;
        int nbThreads = omp_get_num_threads();
        if (id == 0) globalNbThreads = nbThreads;
        for (sf_count_t i = id; i < TONE_SAMPLES_COUNT; i += nbThreads) {
            local_sum += fabs(audio_chunk[i] - reference_tone[i]);
        }
        sums[id] = local_sum;
    }
    for (int i = 0; i < globalNbThreads; i++) {
        sum += sums[i];
    }
    return sum / TONE_SAMPLES_COUNT;
}
```

On gagne un peu de `4.270s` à `4.160s` mais on reste largement en dessus du départ…

```
> hyperfine -r 3 -N 'build/dtmf_encdec_buffers decode verylong2.wav'
  Time (mean ± σ):      4.160 s ±  0.084 s    [User: 8.321 s, System: 13.312 s]
  Range (min … max):    4.075 s …  4.243 s    3 runs
```

Comme 3ème tentative, attaquons nous à un niveau plus, en reprenant l’implémentation de départ de `get_near_score` mais en changeant `detect_button` qui fait les appels de `get_near_score`.

```
inline uint8_t detect_button(const float *audio_chunk, float **freqs_buffers) {

    float min_distance_btn = 200;
    u_int8_t min_btn = BTN_NOT_FOUND;

    for (int i = 0; i < BTN_NUMBER; i++) {
        float score = get_near_score(audio_chunk, freqs_buffers[i]);
        if (score < min_distance_btn) {
            min_distance_btn = score;
            min_btn = i;
        }
    }
```

Trouver le score minimum devrait aussi pouvoir se faire avec openmp.

Avec cette implémentation qui parallélise simplement en scheduling automatique, la recherche pour chaque bouton. J'avais mis la section critique `#pragma omp critical` uniquement dans le `if` au départ mais cela est faux, la condition fait partie de la section critique.

```
#pragma omp parallel for
    for (int i = 0; i < BTN_NUMBER; i++) {
        float score = get_near_score(audio_chunk, freqs_buffers[i]);
#pragma omp critical
        {
            if (score < min_distance_btn) {
                min_distance_btn = score;
                min_btn = i;
            }
        }
    }
```

Au lieu de `1.649s` on est bien mieux avec **940ms** !!

```
  Time (mean ± σ):     940.6 ms ±   9.9 ms    [User: 2932.2 ms, System: 1422.7 ms]
  Range (min … max):   929.3 ms … 947.8 ms    3 runs
```