Mazzucotelli Timothée
Ersfeld Thomas
Pallamidessi Joseph
Preda Madalina

Rapport Projet d'algorithme de recherche

Étude de la croissance de codes circulaires de tétranucleotides autocomplémentaires

Algorithme

Combinaisons

Nous avons décidé de mettre en place un parcours en largeur plutôt qu'un parcours en profondeur utilisant une pile. Ce choix est justifié du fait de la facilité de multi-threader beaucoup plus simplement, de manière massive (super calculateur) et distribuée (architecture cloud et grid). Notre programme procède donc de manière itérative, en traitant d'abord les codes de longueur N puis les codes de longueurs N+1, et ainsi de suite.

Comme notre algorithme n'utilise pas de pile, aucun élagage n'est effectué sur l'arbre des combinaisons. Cependant, afin de simuler cet élagage et de ne pas vérifier la validité de codes dont le ou les sous-ensembles sont invalides, nous reprenons les résultats valides précédemment calculés et les complétons avec les mots qui ne les composent pas déjà. C'est de cette manière que nous construisons les différentes combinaisons possibles afin de les vérifier.

Pour construire un code, qui pour le rappeler est une combinaison non ordonnée et sans répétition de tétranucléotides, nous piochons dans l'ensemble appelé S12 (défini dans le sujet du projet), et dans l'ensemble appelé S108. S108 est un sous-ensemble de S114 (défini dans le sujet du projet). En effet, après avoir lancé quelques tests préalables, nous avons remarqué sur L2 que les codes valides n'étaient composés d'aucun des 6 tétranucléotides suivants : ATCG, ATGC, ATTA, CGGC, CGTA, GCTA. Cela s'explique par le fait que combiné à leur complémentaire, chacun de ces tétranucléotides forment un couple qui, une fois transformé en graphe, crée un cycle. Nous avons donc soustrait ces six tétranucléotides à S114 pour former S108.

Nous ne piochons jamais plus de six tétranucléotides de S12, car comme vu dans le sujet et confirmé par quelques tests sur S12 uniquement, on ne peut jamais obtenir de code valide avec plus de six tétranucléotides de S12.

Longueur de code	**Nombre de codes valides**	**Nombre de combinaisons**
1	12	12
2	60	66
3	160	220
4	240	495
5	192	792

6	64	924
7	0	792
8	0	495
9	0	220
10	0	66
11	0	12
12	0	1

Lorsqu'un tétranucléotide est pioché dans S108 pour construire un code, nous ajoutons automatiquement son complémentaire au code, car les codes doivent être autocomplémentaires (voir sujet) !

Ci-après sont annotées les équations du nombre de combinaisons possibles selon la longueur de code :

Longueur	**Équation**	**Résultat**
1	$\binom{12}{1}$	12
2	$\binom{12}{2} + \binom{108}{1}$	174
3	$\binom{12}{3} + \binom{12}{1} * \binom{108}{1}$	1 516
4	$\binom{12}{4} + \binom{12}{2} * \binom{108}{1} + \binom{108}{2}$	13 401
5	$\binom{12}{5} + \binom{12}{3} * \binom{108}{1} + \binom{12}{1} * \binom{108}{2}$	93 888
6	$\binom{12}{6} + \binom{12}{4} * \binom{108}{1} + \binom{12}{2} * \binom{108}{2} + \binom{108}{3}$	639 888
7	$\binom{12}{5} * \binom{108}{1} + \binom{12}{3} * \binom{108}{2} + \binom{12}{1} * \binom{108}{3}$	3 806 568
8	$\binom{12}{6} * \binom{108}{1} + \binom{12}{4} * \binom{108}{2} + \binom{12}{2} * \binom{108}{3} + \binom{108}{4}$	21 793 293
9	$\binom{12}{5} * \binom{108}{2} + \binom{12}{3} * \binom{108}{3} + \binom{12}{1} * \binom{108}{4}$	113 799 636
10	$\binom{12}{6} * \binom{108}{2} + \binom{12}{4} * \binom{108}{3} + \binom{12}{2} * \binom{108}{4} + \binom{108}{5}$	571 565 538
…	…	

59	$\binom{12}{5} * \binom{108}{27} + \binom{12}{3} * \binom{108}{28} + \binom{12}{1} * \binom{108}{29}$	3.1984726e+28
60	$\binom{12}{6} * \binom{108}{27} + \binom{12}{4} * \binom{108}{28} + \binom{12}{2} * \binom{108}{29} + \binom{108}{30}$	6.0932847e+28

Les codes valides composés uniquement de tétranucléotides de S12 étant peu nombreux, nous les avons sauvegardés dans des fichiers et nous les chargeons en mémoire au démarrage du programme afin de ne pas avoir à les regénérer de multiples fois. On dispose ainsi de combinaisons valides sur S12 de longueur 1, 2, 3, 4, 5 et 6 en mémoire pour un accès rapide.

En reprenant le tableau précédent dans lequel on remplace les combinaisons sur S12 par le nombre de codes valides sur S12 correspondant, on observe une diminution des combinaisons totales à tester :

Longueur	**Équation**	**Résultat**
1	12	12
2	$60 + \binom{108}{1}$	168
3	$160 + 12 * \binom{108}{1}$	1 456
4	$240 + 60 * \binom{108}{1} + \binom{108}{2}$	12 498
5	$192 + 160 * \binom{108}{1} + 12 * \binom{108}{2}$	86 808
6	$64 + 240 * \binom{108}{1} + 60 * \binom{108}{2} + \binom{108}{3}$	576 820
7	$192 * \binom{108}{1} + 160 * \binom{108}{2} + 12 * \binom{108}{3}$	3 395 088
8	$64 * \binom{108}{1} + 240 * \binom{108}{2} + 60 * \binom{108}{3} + \binom{108}{4}$	19 002 087
9	$192 * \binom{108}{2} + 160 * \binom{108}{3} + 12 * \binom{108}{4}$	98 083 476
10	$64 * \binom{108}{2} + 240 * \binom{108}{3} + 60 * \binom{108}{4} + \binom{108}{5}$	482 382 108
…	…	
59	$192 * \binom{108}{27} + 160 * \binom{108}{28} + 12 * \binom{108}{29}$	1.5751556e+28
60	$64 * \binom{108}{27} + 240 * \binom{108}{28} + 60 * \binom{108}{29} + \binom{108}{30}$	2.6401244e+28

Enfin, en réutilisant les résultats précédents sur S108 de la même manière que sur S12, on peut encore drastiquement diminuer le nombre de combinaisons à tester et parvenir à

l'équivalent d'un élagage avec pile de l'arbre des combinaisons (dès qu'un code est invalide, on ne teste plus les codes qui le contiennent) !

Pseudo-code

Nous décrivons ci-dessous l'algorithme de manière plus formelle, en pseudo-code.

- *maxS12* correspond au nombre maximum de tétranucléotides de S12 qui composent les codes en construction.
- $S12_n$ correspond au tableau contenant les codes valides sur S12 uniquement, de longueur n.
- $S108_n$ correspond aux codes valides sur S108 uniquement, de longueur n.
- La procédure *compl(T)* renvoie le complémentaire du tétranucléotide T.
- La procédure *absent(C)* renvoie les tétranucléotides de S108 absents du code C.
- La procédure *vérifier(C)* valide ou non le code C en construisant son graphe et en vérifiant l'absence de cycle. C'est dans cette procédure que se passe tout le multi-threading (volontairement caché dans le pseudo-code).
- Les procédures *récupérerRésultat()* et *renvoyer(R)* récupèrent la somme des codes valides et la renvoient au thread principal.

```
codes_valides(L) :

	// Initialisation
	si L ≤ 6:
		maxS12 ⟵  L
	sinon si L est paire:
		maxS12 ⟵ 6
	sinon:
		maxS12 ⟵ 5

	// 1ère étape : construction des codes avec S12, S108 et les résultats précédents.
	pour nbS12 de maxS12 à 0 (exclu), de 2 en 2:

		// un tétra de S108, son complémentaire, et le reste de S12
		si L - nbS12 == 2:
			pour chaque tetra T de S108:
				pour chaque code C de S12_nbS12:
					code ⟵ C ∪ T ∪ compl(T)
					vérifier(code)

		// S12 uniquement. Calculs redondant mais temps d'exécution négligeable.
		// Exécuté uniquement jusqu'à L6.
		sinon si L == nbS12:
			pour chaque code C de S12_nbS12:
				code ⟵ C
```

```
                        vérifier(code)

        // un code valide sur S108 combiné à un code valide sur S12
        sinon:
                pour chaque code C1 de S108_{L-nbS12}:
                        pour chaque code C2 de S12_{nbS12}:
                                code ← C1 ∪ C2
                                vérifier(code)

// 2ème étape : construction des codes avec S108 et les résultats précédents.
// Seulement dans le cas où L est paire (un code impair a forcément
// au moins un tétranucléotide de S12).
si L est paire:

        // Exécuté une seule fois, lorsque L vaut 2
        si L == 2:
                pour chaque tetra T de S108:
                        code ← T ∪ compl(T)
                        vérifier(code)

        sinon:
                pour chaque code C de S108_{L-2}:
                        pour chaque tetra T de absent(C):
                                code ← C ∪ T ∪ compl(T)
                                vérifier(code)

total ← récupérerRésultat()
renvoyer(total)
```

En pratique

On voit le pseudo-code et dans les équations ci-dessus que chaque longueur dépend des résultats des trois longueurs paires précédentes (avec $\binom{108}{n}$ correspondant aux codes valides composés uniquement de tétranucléotides de S108 et de longueur n*2).

Le chargement en mémoire n'étant pas concevable pour de tels volumes de données (les codes valides des trois longueurs paires précédentes), nous avons opté pour l'écriture des résultats et la lecture des résultats précédents sur le disque, dans des fichiers texte. L'algorithme a donc besoin d'un accès aux résultats des trois longueurs paires précédentes pour calculer la longueur courante.

Grâce à cette manière de procéder, il est possible de découper les jeux de données pour effectuer chaque partie du calcul sur des machines différentes, en parallèle. Avec dix machines, on divise le temps de calcul par dix. Avec vingt machines, par vingt, etc. Évidemment, certaines combinaisons à tester ne dépendent pas des résultats précédents, mais seulement des codes

valides sur S12, il faut donc indiquer à chaque programme s'il doit ou non effectuer tous les calculs ou seulement ceux qui dépendent des résultats précédents afin de ne pas avoir de redondance dans les calculs.

Nous avons implémenté ce comportement avec une option à fournir en ligne de commande : -m ou --master pour la machine qui effectuera tous les calculs, et -w ou --worker pour les machines qui effectueront seulement les calculs dépendant des résultats précédents.

Il est également possible, toujours grâce à la relecture des résultats précédents, de ne calculer que les longueurs paires ! En effet les résultats des longueurs impaires ne sont pas réutilisables pour les calculs des longueurs suivantes, il est donc possible de les éviter et d'itérer sur les longueurs paires uniquement, ce qui permet de gagner un temps considérable si le but est simplement d'aller le plus loin possible dans les longueurs de codes.

Multi-threading et Performances

Le programme est écrit en Java version 7. Nous avons utilisé la bibliothèque libre et gratuite JGraphT pour les opérations de manipulation des graphes par exemple leur algorithme de détection de cycles.

Le multi-threading est effectué grâce au très bon package java **java.util.concurrent**, qui permet de manipuler de manière optimisée des pools de threads, mais aussi de récupérer les résultats renvoyés par chaque thread.

Le multi-threading est géré par des pools de threads et plus particulièrement avec un système producteur-consommateur. La construction des combinaisons est le travail du thread producteur, qui va créer des paquets de codes à tester et les envoyer aux threads consommateurs qui vont, pour chacun des codes, créer un graphe et vérifier qu'il n'y a pas de cycle. Par défaut, chaque thread consommateur reçoit un paquet de 8192 codes.

Ce modèle permet une gestion dynamique de la charge et du partage des ressources pour un multithreading efficace. Les différents paramètres du programme comme la taille de la file d'attente de threads ou la taille des paquets de codes sont spécifiables par l'utilisateur sur la ligne de commande.

Les paramètres par défaut ont été choisis après quelques tests de performance effectuée avec les variables suivantes : la taille des buffers de threads (le nombre de codes que le producteur leur envoie), la taille de la file d'attente de threads et le nombre de threads actifs dans cette file. Ces paramètres ne seront pas forcément optimaux pour tout type de processeur.

Sur un supercalculateur

Les gains de performance sur un supercalculateur ne sont *a priori* pas très importants, car l'algorithme n'utilise qu'un seul producteur. De plus, l'algorithme se base sur les résultats des calculs précédents et doit donc lire des données sur le disque, la vitesse de production des combinaisons à tester est donc fortement liée à la vitesse de lecture sur le disque. Néanmoins, l'utilisation de plusieurs producteurs est envisageable sans craindre de pertes de performances dues aux lectures multiples sur le disque.

Sur un cluster de machines

Ceci est la meilleure option à envisager, car un cluster de machines offre non seulement plusieurs processeurs mais aussi plusieurs disques sur lesquels lire et écrire les données. Les calculs sont alors véritablement effectués en parallèle et ne risquent plus d'être ralenti par l'accès à un seul disque partagé. De plus l'accès à un cluster de machines nous semble plus réalisable que l'accès à un supercalculateur !

Le programme a bien sûr besoin de modification pour fonctionner automatiquement sur un cluster avec une architecture client-serveur, mais il est déjà possible de découper les jeux de données à la main et de lancer le programme sur plusieurs machines. Comme dit précédemment, avec un cluster de N machines, le temps d'exécution est divisé par N ! Et la seule limite à N est la possibilité de découpage des jeux de données (découpés en 100 ? en 1000 ? en 1 million de parties ?).

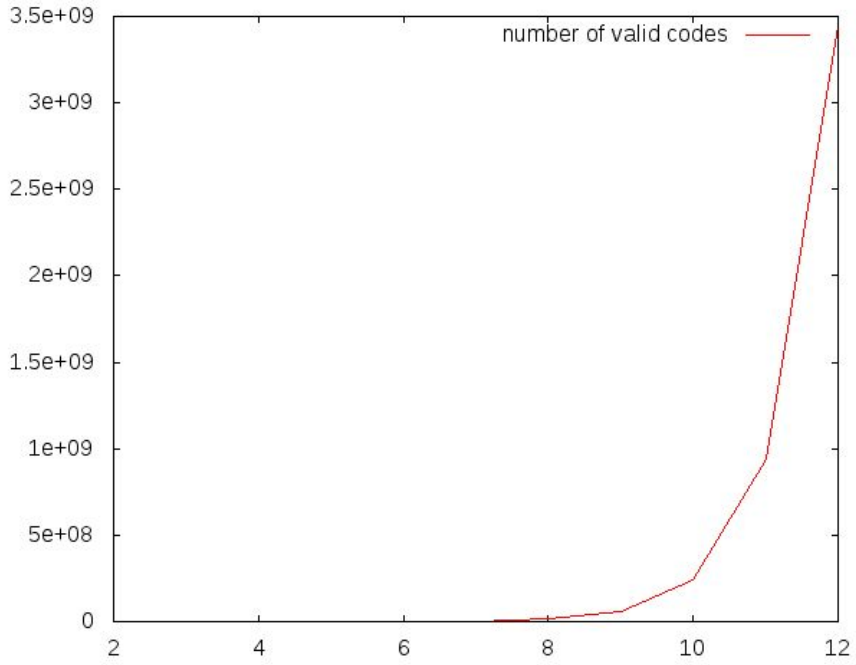
Résultats et Analyses

Voici le tableau récapitulatif des temps et des nombres de codes valides trouvés :

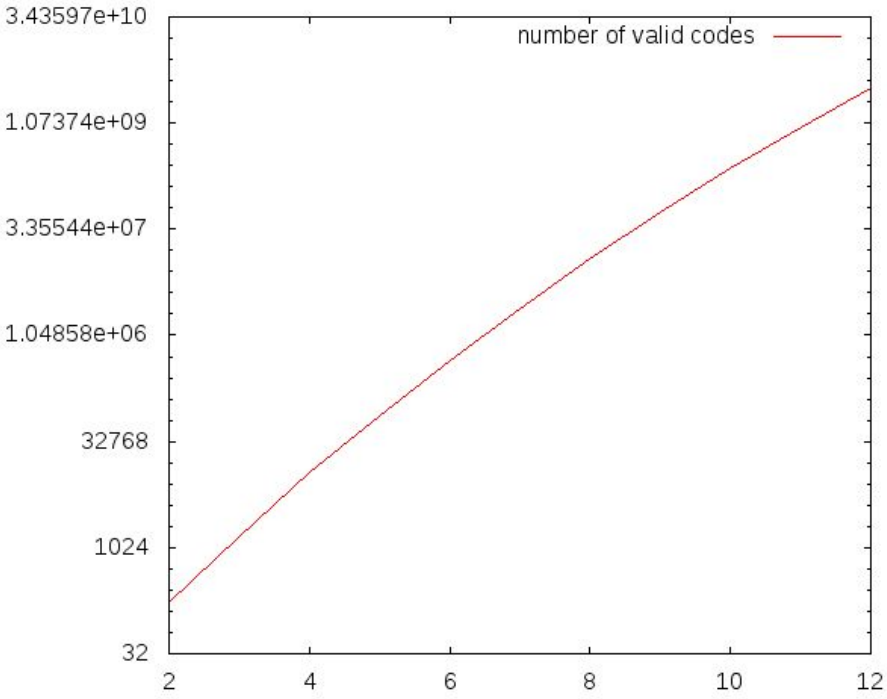
Longueur de code	Nombre de codes valides	Temps d'exécution (cumulé)
2	168	00:00:00:042
3	1 408	00:00:00:107
4	11 728	00:00:00:295
5	76 312	00:00:00:742
6	475 168	00:00:02:889
7	2 530 868	00:00:13:289
8	12 764 634	00:01:14:258
9	57 374 400	00:06:35:048
10	243 658 816	00:34:27:641
11	942 624 972	02:34:16:386
12	3 448 032 652	11:12:49:614

Les calculs ont été effectués sur un processeur Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz.

Nous n'avions malheureusement pas de machine disponible pour faire tourner le calcul sur de longues périodes, nos exécutions se sont donc arrêtées à L12. D'après nos projections, L13 uniquement est calculable en trente-cinq heures sur cette seule et même machine, L14 uniquement en cinq jours, et L15 uniquement en vingt jours.



Graphique des résultats sur une échelle classique



Graphique des résultats sur une échelle logarithmique

Instructions d’utilisation

Nous n’avons pas développé d’interface graphique pour ce projet, tout se fait en ligne de commande. Il suffit de lancer le programme avec java, grâce à la commande suivante :

- Linux: `java -jar tetra.jar [OPTIONS]`
- Windows: `javaw.exe -jar tetra.jar [OPTIONS]`

Sur Windows, il faudra que le programme `javaw.exe` soit dans le path.

Pour obtenir la liste des options disponibles, affichez l’aide avec l’option `-h` ou `--help`.