Projet de Programmation Numérique sur Machine Parallèle "Analyseur génétique"

Dépôt Git

Encadré par : **Mohammed-Salah Ibnamar**

Réalisé par :

Thiziri SALEM
Fatma HASSANI
Kevin PEYEN
Faouzi LOUGANI
Lounas KHERIS

M1CHPS

9 mai 2021

Table des matières

Introduction

L'information génétique est conservée par la cellule au niveau de son ADN qui est localisé à l'intérieur du noyau. Il comporte une multitude de gènes qui ne s'expriment pas tous simultanément dans les cellules de l'organisme. Une séquence ADN est composée de bases appelées nucléotides ; A pour Adénine, T pour la Thymine, C pour la Cytosine et G pour Guanine. Une autre base existe le U pour l'Uracil. Cette information est transcrite en ARNm qui est une copie d'une portion d'ADN, synthétisé dans le noyau. ARN et ADN sont deux molécules constituées d'un enchaînement de nucléotides, mais dans l'ARN, le sucre des nucléotides est du ribose, et non pas le désoxyribose de l'ADN et la base azotée uracile (U) remplace la thymine (T). De plus, l'ARN est une molécule formée d'une seule chaîne de nucléotides, tandis que l'ADN a deux chaînes complémentaires disposées en double hélice.

Ces nucléotides sont assemblés par groupe de 3 pour former un codon. Chacun de ces codons représente un acide aminé, présent dans une protéine. Il y a donc 64 combinaisons possibles pour former un codon mais seulement 21 acides aminés peuvent être formés car plusieurs combinaisons différentes peuvent former une même acide aminé. De plus, comme nous le verrons, 3 codons permettent de représenter la fin d'un gène. La copie d'un gène s'effectue d'abord, grâce à une enzyme nommée ARN-polymérase, dans le noyau sous forme d'ARN messager : cette première étape s'appelle la transcription. [2] Les ARN produits quittent le noyau par les pores de l'enveloppe nucléaire, ils jouent le rôle de messagers. Dans le cytoplasme, les acides aminés sont assemblés, par de petits organites nommés ribosomes, en chaînes polypeptidiques, dans l'ordre imposé par le message contenu dans l'ARN : cette seconde étape s'appelle la traduction.

1 Analyseur génétique

Dans ce projet, on s'est focalisé sur l'aspect Informatique du domaine. Il consiste à développer un ensemble d'analyses pouvant fournir un aperçu approfondi sur de larges codes génétiques, séquences d'ADN de plusieurs mégaoctets. Pour cela deux approches ont été envisageables. On a opté pour la deuxième approche qui consistait à ajouter une phase de pré-traitement qui va préparer les données en générant un nouveau format, puis effectuer les traitements et générer la sortie. Avec cette méthode il y a une perte de temps dû à la surcharge du pré-traitement mais le gain par la suite est bien supérieur. Elle nous sera ensuite bénéfique pour la seconde partie pendant la parallélisation car le data-set sera partagé en plusieurs sous fichier sur lesquels on établira un ensemble d'analyses pouvant fournir un aperçu approfondi de ce dernier.

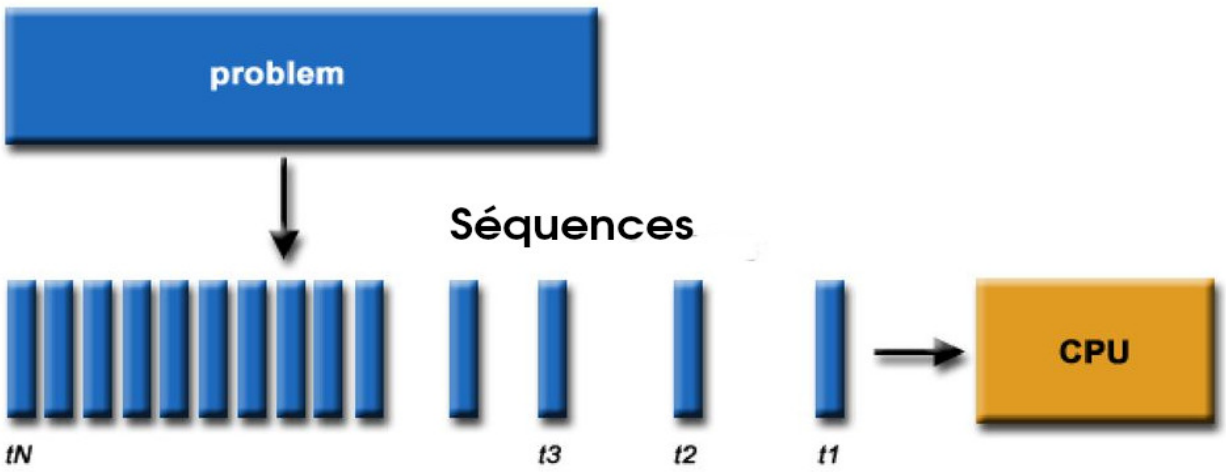
1.1 Objectifs :

Dans ce qui suivra, en premier temps, nous discuterons sur la partie séquentielle du projet. On verra essentiellement les algorithmes implémentés et on calculera leurs complexités. On comparera les différents compilateurs (gcc, icc, clang) sur ce projet avec les différents flags d'optimisation (O0,...,Ofast, ...). Ensuite on introduira la partie parallèle du projet, on expliquera la démarche effectuée afin d'aboutir à une solution parallèle où on verra le schéma de découpage choisi, différents processus, leurs synchronisation, et les différents communications. Ensuite on passera aux tests de performance, on évaluera la scalabilité du code et son comportement avec les différents compilateurs et flags d'optimisations.

Enfin, nous comparerons les deux version séquentielle et parallèle, et on tirera des conclusions.

2 Version séquentielle

Le calcul séquentiel consiste en l'exécution d'un traitement étape par étape, où chaque opération se déclenche que lorsque l'opération précédente est terminée, y compris lorsque les deux opérations sont indépendantes. Le calcul séquentiel s'oppose au calcul parallèle, où plusieurs opérations peuvent être simultanées.
Il s'agit du mode de traitement le plus fréquemment utilisé, car compatible avec n'importe quel type d'opération. Il est cependant limité par la puissance des calculateurs, là où le calcul parallèle n'est limité que par le nombre de calculateurs.
Dans notre projet, l'ensemble des traitement qu'on doit effectuer sur chaque séquence est séquentiel, c'est à dire, on effectue les traitement que sur une seule séquence à la fois. Si on suppose que toutes les séquences sont similaires (de même taille), le temps nécessaire de traiter toutes les séquences est de t*n (t : le temps de traitement d'une seule séquence).



2.1 Complexités

Dans cette partie on s'intéresse a estimer et comparer les temps d'exécutions requis par les différents fonctions, avant d'entamer une phase de parallélisation du code, on cherche d'abord a évaluer de manière théorique la quantité d'appels aux opérations les plus coûteuses en temps, nous parlons ici de la complexité temporelle, Nous pouvons alors estimer le temps d'exécution suivant l'optique dans le pire des cas.

2.1.1 Algorithme Détection de gène

Le temps pour trouver une occurrence d'un codon est en O(3 * N), 3 étant la taille d'un codon et N la taille de la séquence. On recherche au minimum un codon de départ et pour chaque codon de départ trouvé, on exécute trois recherches de codon d'arrêt. Par contre N ne fait que diminuer car on ne recherche qu'à partir des derniers éléments trouvés, donc cette algorithme est de complexité linéaire.

2.1.2 Algorithme génération de l'ARN messager

Le temps pour générer tous les ARN messager dépend du nombre de gènes trouvés dans la séquence, N, et de leur taille, M. La complexité de notre algorithme est O(N * M).

2.1.3 Algorithme génération des protéines

L'algorithme génération des protéines dépend de nombre de protéines P qui est la taille de l'ARN messsager divisé sur trois, donc la complexité de cet algorithme sera de O(P) .

2.1.4 Algorithme détection des zones à risque de mutations

Vu que nous parcourons tous les gènes caractère par caractère la complexité de notre algorithme dépend de N qui représente le nombre de gènes, M la taille du gène donc la complexité sera de O(N * M).

2.1.5 Algorithme taux de correspondance entre 2 séquences

Cet algorithme dépend de la taille des deux séquences, N étant la taille de la grande, M la taille de la petite et I étant le nombre de taux à calculer. I est égal à N divisé par M + N modulo M. Notre complexité est O(I * M) pour calculer tous les taux de correspondance.

3 Version parallèle

3.1 Parallélisme

Dans le sens le plus simple, le calcul parallèle est l'utilisation simultanée de plusieurs ressources de calcul pour résoudre un problème de calcul :

- Un problème est divisé en parties distinctes qui peuvent être résolues simultanément
- Chaque partie est ensuite décomposée en une série d'instructions
- Les instructions de chaque partie s'exécutent simultanément sur différents processeurs
- Un mécanisme global de contrôle / coordination est utilisé

La programmation parallèle fournit

- Plus de ressources CPU
- Plus de ressources mémoire
- La capacité de résoudre des problèmes qui n'étaient pas possibles avec le programme série
- La capacité de résoudre les problèmes plus rapidement

Deux approches sont envisageables :

3.1.1 Mémoire partagée :

- Utilisé par la plupart des machines
- Plusieurs cœurs (processeurs)
- Partager un espace mémoire global
- Les cœurs peuvent échanger / partager efficacement des données

3.1.2 Mémoire distribuée

- Collection de nœuds qui ont plusieurs cœurs
- Chaque nœud utilise sa propre mémoire locale
- Travaillez ensemble pour résoudre un problème
- Communiquez entre les nœuds et les cœurs via des messages
- Les nœuds sont mis en réseau ensemble

3.2 Problèmes de programmation parallèle

L'objectif est de réduire le temps d'exécution
- Temps de calcul
- Temps d'inactivité - attente des données d'autres processeurs
- Temps de communication - temps nécessaire aux processeurs pour envoyer et recevoir des messages

L'équilibrage de charge
- Répartir le travail de manière égale entre les processeurs disponibles

Minimiser la communication
- Réduire le nombre de messages passés
- Réduire la quantité de données transmises dans les messages

Dans la mesure du possible - chevauchement de la communication et du calcul

De nombreux problèmes s'adaptent bien à un nombre limité de processeurs

3.3 La loi d'amdhal

La loi d'Amdahl donne l'accélération théorique en latence de l'exécution d'une tâche à charge d'exécution constante que l'on peut attendre d'un système dont on améliore les ressources.
Une tâche exécutée par un système dont les ressources sont améliorées par rapport à un système similaire initial peut être séparée en deux parties :

une partie ne bénéficiant pas de l'amélioration des ressources du système ; une partie bénéficiant de l'amélioration des ressources du système.

Exemple :
Un programme qui traite les fichiers d'un disque. Une partie du programme commence par lire le répertoire du disque et crée une liste de fichiers en mémoire. Puis une autre partie du programme passe chaque fichier à un fil d'exécution pour traitement. La partie qui lit le répertoire et crée la liste de fichiers ne peut pas être accélérée sur un ordinateur parallèle, mais la partie qui traite les fichiers peut l'être.

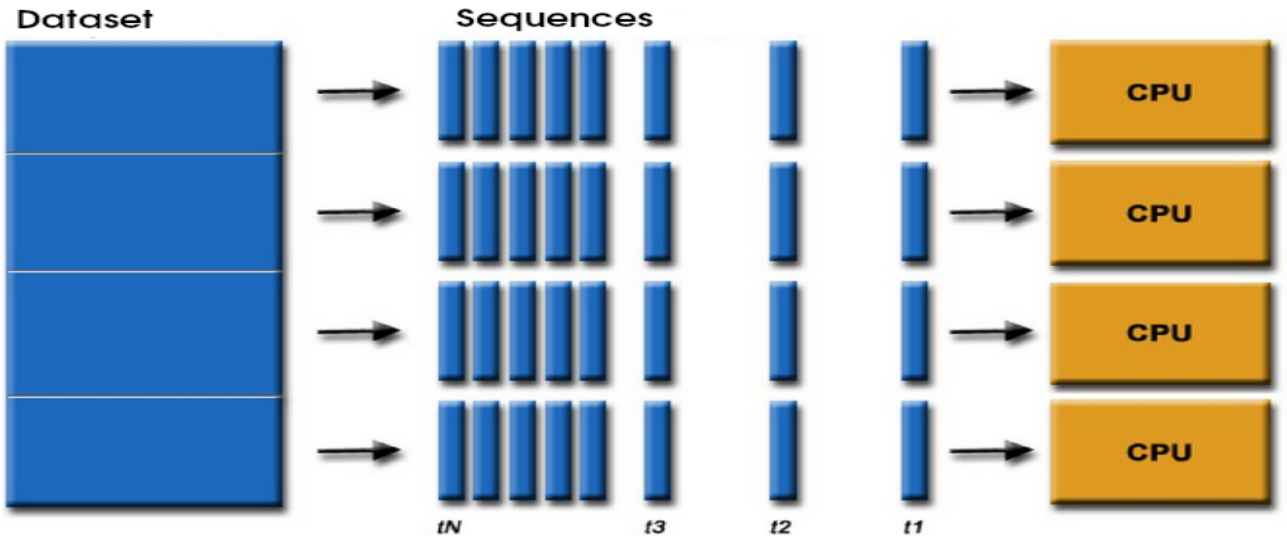
La loi d'Amdahl peut être formulée de la façon suivante :

$$speedup = \frac{1}{(1 - P) + \frac{P}{n}}$$

Où : p est le pourcentage du temps d'exécution de toute la tâche concernant la partie bénéficiant de l'amélioration des ressources du système avant l'amélioration.
et n est le nombre de fils d'exécutions (threads) utilisés pour exécuter la tâche.

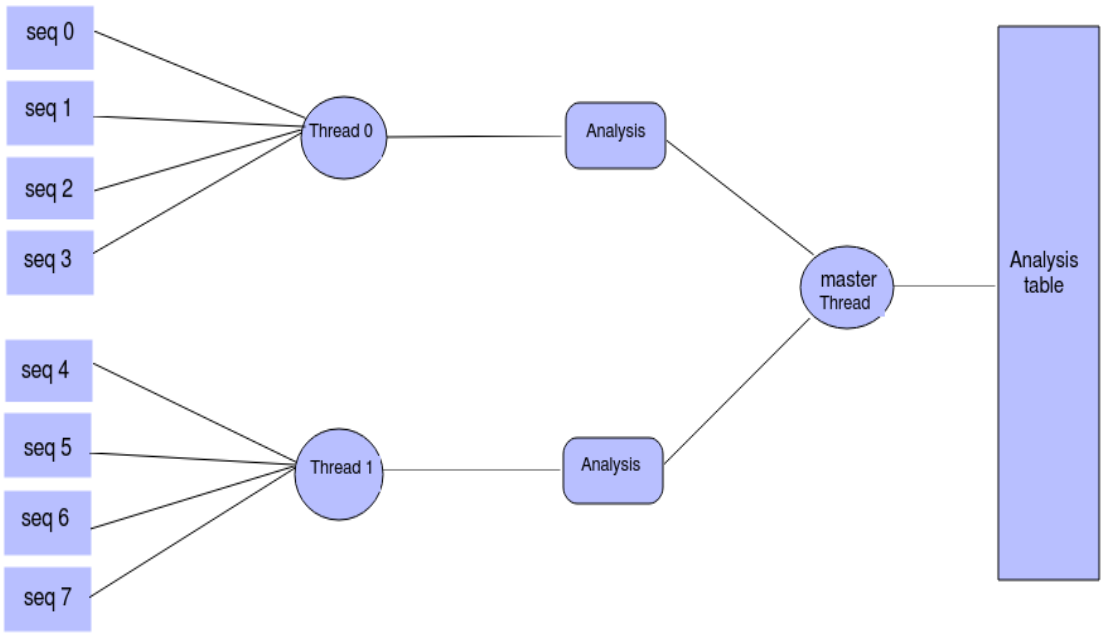
3.4 Schéma de découpage

L'ensemble des séquence est représentable par une suite de traitements totalement ordonnée, alors l'algorithme est dit séquentiel. Sauf que dans notre cas, les traitements sont dépendants, on ne peut pas effectuer le prochain sans terminer celui en cours.



Le principe consiste a charger les données dans le processus maître, ensuite établir le découpage du fichier "sequences.fasta", ensuite envoyer séquence par séquence aux processus différent du maître (esclaves) en fonction du nombre de thread, et effectuer l'ensemble des traitements prévus sur chaque séquence.

3.5 Parallélisation MPI



Les séquences de l'échantillon doivent être regroupées en blocs, chaque bloc sera associé a un thread qui va faire son analyse et générer un rapport d'informations ,une fois les rapports générés le thread père va les regrouper dans une table finale, chaque entrée de cette table sera l'analyse d'une séquence.

3.6 Communication

MPI_Scatter prend un tableau d'identifiants des séquences et distribue les éléments dans l'ordre de classement du processus. Le premier élément va au processus zéro, le deuxième élément au processus un, et ainsi de suite. Bien que le processus racine (processus zéro) contienne tout le tableau de données, MPI_Scatter copiera l'élément approprié dans le tampon de réception du processus.
Ensuite une fois les traitements effectués, ils renvoient les résultats au processus 0 et les stocke dans la table d'analyse.

4 Comparaison test de performance séquentiel vs parallèle

4.1 Test sur plusieurs compilateurs :

Dans cette section, nous allons tester notre programme séquentiel sur différents compilateurs et comparer les performances, en variant les flags d'optimisation. Pour réduire les fluctuations de performances causées par la migration du processeur ou le changement de contexte, on exécute sur un cœur de processeur à l'aide "taskset".

4.1.1 Compilateurs

GCC Désigne le compilateur pour langage C créé dans le cadre du projet GNU. mais peut signifier aussi GNU Compiler Collection (en anglais) dont il est inclus qui représente la collection de logiciels libres intégrés capables de compiler les codes écrits en langage de programmation C, C++, Objective-C, Java, Ada et Fortran. Ce dernier est par défaut installé en tant que composant standard des principales distributions Linux et autres systèmes d'exploitation de type Unix mais également disponible en ligne,avec des binaires pré-compilés pour plusieurs plates-formes.[2]

Clang est un compilateur multi-langage (C, C++, Objective C/C++, OpenCL, CUDA, RenderScript) et multi-platforme (compatible avec GCC), utilisant les bibliothèques LLVM pour l'optimisation et la génération du binaire final. Ce compilateur fournit des outils comme clang-tidy permettant de corriger les erreurs de programmation typiques, telles que des violations de style, une mauvaise utilisation de l'interface ou des bugs pouvant être déduits via une analyse statique.[1]

Intel C++ compiler est un compilateur pour les langages C et C++ de la chaîne d'outils élaboré par intel offrant 3 alternatives [3] :
- Compilateur Intel DPC++ inclus dans le kit d'outils de base intel oneAPI.
- Compilateur Intel C++ inclus dans le kit d'outils de base intel oneAPI.
- Compilateur Intel C++ classique inclus dans le kit d'outils de base intel HPC oneAPI.

Pour nos expériences dans ce projet on a choisis la 3ème alternative.

Les fichiers des versions de compilateurs utilisés dans le cadre de ce projet sont accessibles ici : versions des compilateurs.

4.1.2 Les optimisations

Véctorisation La vectorisation, en termes simples, signifie optimiser l'algorithme afin qu'il puisse utiliser les instructions SIMD dans les processeurs. Un grand nombre de compilateurs

optimisant réalisent une vectorisation automatique du code : c'est une fonctionnalité du compilateur qui permet à certaines parties des programmes séquentiels d'être transformés en programmes parallèles équivalents afin de produire du code qui sera bien utilisé par un processeur vectoriel.

Déroulage de boucle le compilateur peut aussi effectuer des transformations sur les boucles. La transformation la plus basique est ce qu'on appelle le déroulage de boucles. Dérouler des boucles consiste à effectuer plusieurs tours de boucle d'un seul coup, en recopiant le corps de la boucle en plusieurs exemplaires. Cette technique est clairement utile pour diminuer le nombre de branchement exécutés : vu qu'on exécute les branchements à chaque itération, diminuer le nombre d'itérations permet d'en diminuer le nombre également. Évidemment, le nombre d'itérations est modifié de manière à obtenir un résultat correct, histoire de ne pas faire des itérations en trop.
Exemple, prenons cette boucle :

```
for(int i=0;i<mini;i++)
  d += popcount( seq[pos+i] ^ seq2[i] );
```

Celle-ci peut être déroulée comme suit :

```
for(int i=0;i<mini;i+=4){
  d += popcount( seq[pos+i] ^ seq2[i] );
  d += popcount( seq[pos+i+1] ^ seq2[i+1] );
  d += popcount( seq[pos+i+2] ^ seq2[i+2] );
  d += popcount( seq[pos+i+3] ^ seq2[i+3] );
}
```

On peut aussi pousser plus loin les optimisations avec loop-blocking et tiling, sauf qu'on n'a pas d'exemples concrets dans notre programme.

On peut ainsi ajouter des options d'optimisations, comme "-march" afin d'indiquer au compilateur quel code il devrait produire pour votre architecture. [4]
"-funroll-loops" : Demande au compilateur d'effectuer un déroulement de boucle de base.
"-ftree-vectorize" : active la vectorisation automatique
La vectorisation automatique, tout comme l'optimisation de boucle ou une quelconque autre optimisation de compilation, doit préserver exactement le comportement du programme.

4.1.3 Préparation de l'environnement

Une préparation de l'environnement de mesure est nécessaire :

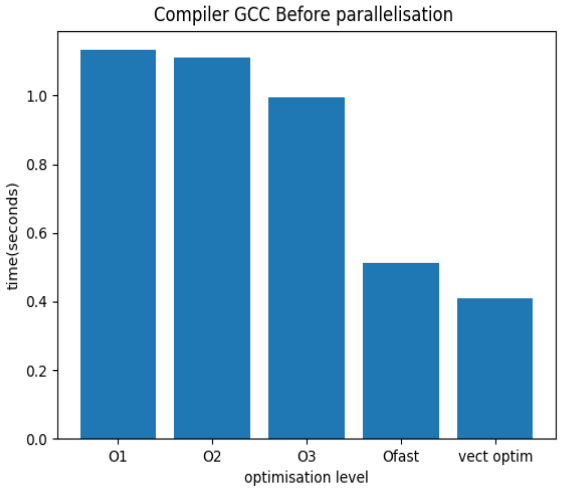
- S'assurer que la machine est sous tension.
- Se limiter à un nombre minimum de services et qu'aucun processus lourd est exécuté en arrière plan.
- Affecter le type **Userspace** à tous les gouverneurs CPU avec la commande :
  sudo cpupower -c all frequency-info
- S'assurer que la fréquence est constante pour tous les coeurs en l' affectant avec la commande : **sudo cpupower -c all frequency-set -f 3.5G**
  tel que (3.5G par exemple) représente la fréquence maximale.
- Désactiver le **boost state support** afin d'éviter une éventuelle augmentation de la fréquence durant les mesures.

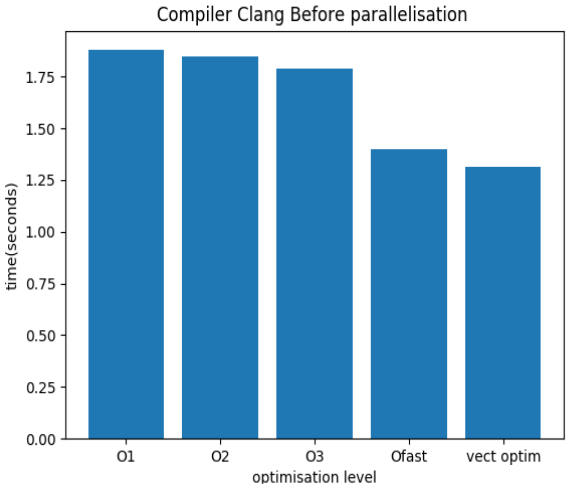
Pour résumer il faut stabiliser le système de façon à avoir une barre de déviation standard sur les mesures en dessous de 0,1% .

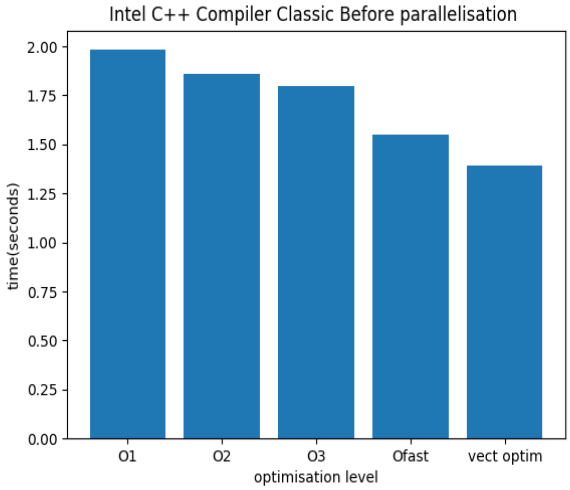
Les informations système de la machine exploité sont disponibles ici : informations système

Afin de mesurer nos performances, on a opté pour l'outil **perf** qui est un profiler permettant de récupérer le temps d'exécution émis par notre programme ainsi que d'autres analyses de performance.
Avant de commencer les mesures, on doit s'assurer que notre programme ne contient pas d'erreurs ou bugs, l'outil **gdb** nous a permis de vérifier cela.

4.1.4 Version séquentielle :

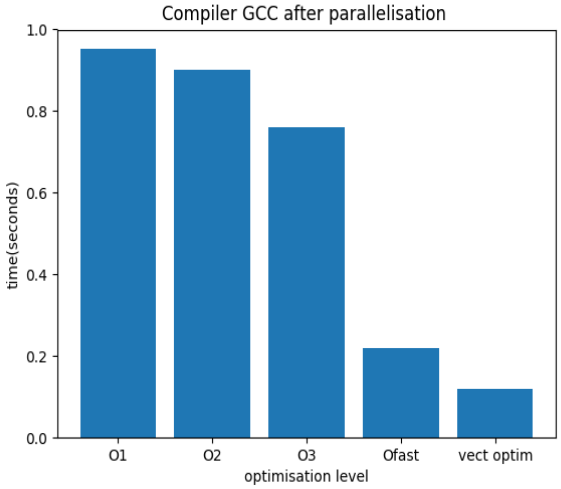


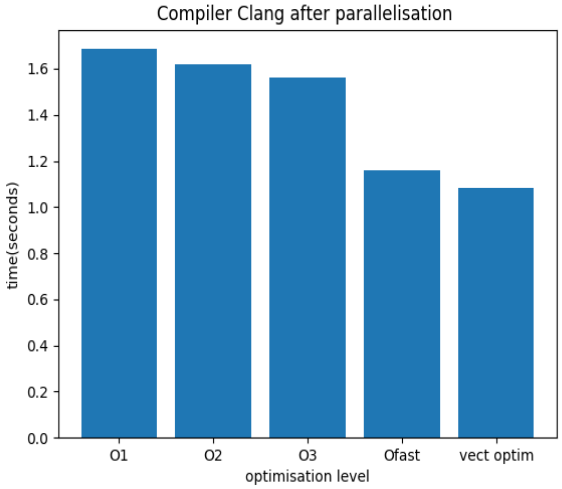


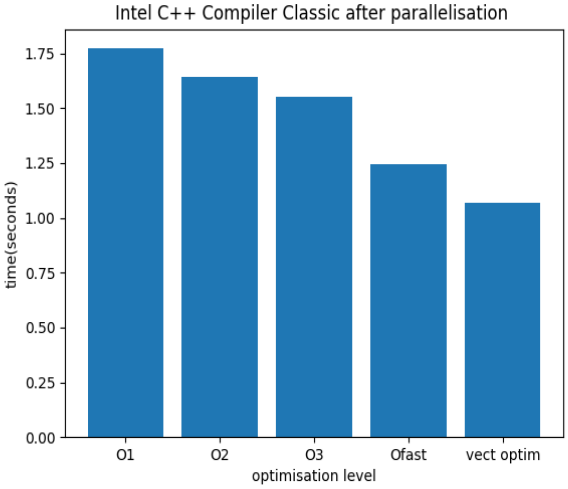


— D'après les résultats obtenus,le compilateur offrant un meilleur temps d'exécution sera **gcc**,ensuite **clang** et **intel C++ Compiler**
— Le changement de **flags** de compilation n'a pas vraiment amélioré le temps d'exécution d'où mettre en place une parallélisation est évidente pour avoir une optimisation meilleure.

4.1.5 Version parallèle :







— La parallélisation MPI a apporté de meilleures performances car le temps d'exécution a considérablement diminué pour chacun des compilateurs en gardant le même classement (gcc puis clang et intel C++ Compiler Classic).

	O1	O2	O3	Ofast	vect optim
gcc	0.18	0.21	0.23	0.29	0.30
clang	0.19	0.22	0.23	0.24	0.26
intel C++ compiler	0.21	0.22	0.24	0.31	0.32

— Le tableau précédent explique le gain en temps d'exécution calculé à partir du : **(temps d'exécution pour la version séquentielle − temps d'exécution après parallélisation)** exprimé en secondes.
— le tableau montre que la performance s'améliore mieux si on applique la **parallélisation** avec la **vectorisation**.
— Le seuil pour **gcc** et **intel C++ compiler** est bien plus grand avec les flags **Ofast** et **vect optim** que **clang** car la gestion des multiples cœurs en parallèle diffère d'un compilateur à un autre.

4.2 Étude de scalabilité :

Dans cette étude, on s'intéresse principalement à l'efficacité du programme, en d'autre termes, le problème reste fixe alors que le nombre de cœurs augmente. On s'attendrait idéalement à une scalabilité linéaire, c'est-à-dire que la diminution du temps d'exécution par rapport à la valeur de référence serait réciproque au nombre de cœurs ajoutés. Dans l'exemple suivant, on peut constater le résultat de tests sur la même grappe d'un programme parallèle avec les paramètres d'entrée identiques :

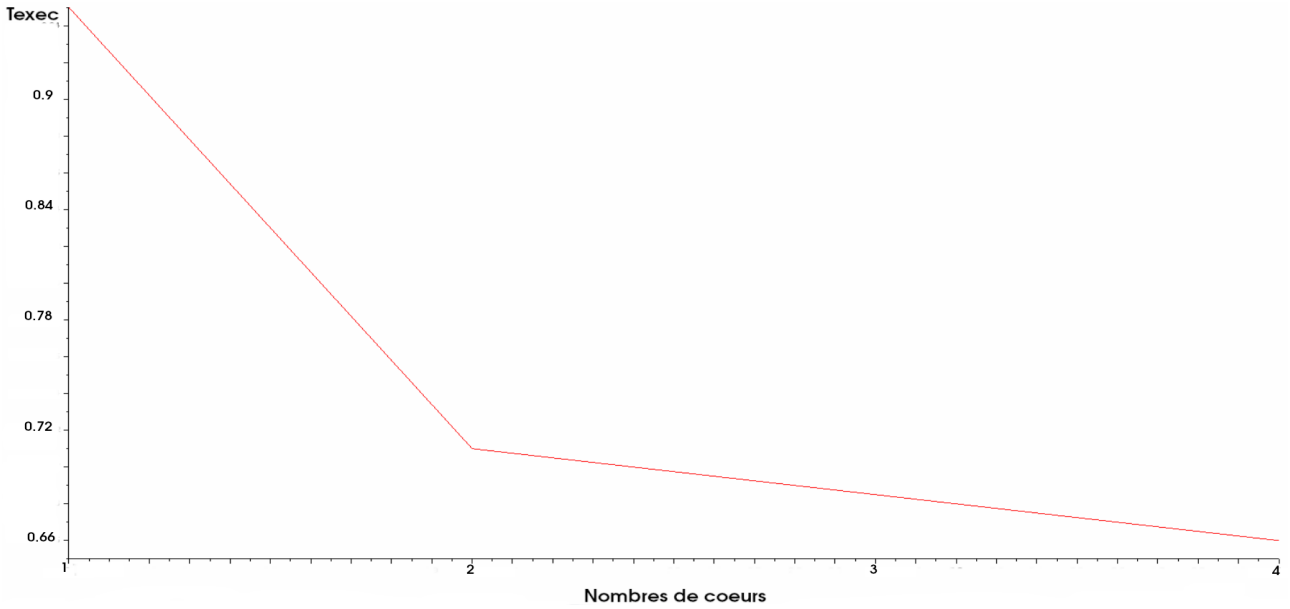


Figure 1 : Scalabilité forte du programme

nombre de cœurs	durée d'exécution (secondes)	efficacité (%)
1	0.95	/
2	0.71	/
4	0.66	53,78 %

Figure 1 : Efficacité du programme

L'efficacité est le rapport de la durée d'exécution avec deux cœurs d'une part et n cœurs d'autre part, divisé par n / 2 puis multiplié par 100.
Une efficacité de 75% ou plus est à privilégier et dans cet exemple, nous somme limité à une machine de 4 coeurs, donc au plus on a une efficacité de 58%. Une perspective est d'essayer d'exécuter ce problème sur un grand nombre de coeurs et constater l'évolution du gain de temps.

5 Conclusion

Nous avons vu en partie des optimisations que tout compilateur effectue automatiquement sur votre code source. Bien sûr, un compilateur peut effectuer d'autres optimisations un peu plus complexes sur votre code source, mais les optimisations vues au-dessus sont un aperçu assez intéressant, suffisant pour avoir une bonne idée de ce qui se passe quand vous compilez un programme. On a aussi vu les différents flags d'optimisation en comparant le temps d'exécution avec les deux versions, et une étude de scalabilité, qui nous montre l'efficacité du parallélisme, certe, on n'a pas eu un gain énorme mais sur un cluster d'exécution on est persuadé qu'on aura des gain meilleurs.

Annexes

Algorithm 1 Détection des gène

Type
Structure genemap
genecounter :entier
genestart[1000] : tableau d'entiers
geneend[1000] :tableau d'entiers
FinStruct
Entrées : seq :^caractère
Sortie :gm
Var :
startpos, stoppos, check : entier
Début
startpos← 0, gm.genecounter ← 0, stoppos←0, check← -1
Répéter
Chercher le motif "ATG" dans la séquence à partir de la position du dernier codon stop que on a trouvé
check← find("ATG",seq+stoppos)
Si on a trouvé le motif 'ATG' on stocke sa position
chercher le premier codon stop à partir de la position du dernier codon start trouvé.
Si (on a trouvé un stop)
stoppos← startpos+check-3
enregistrer leurs positions trouvés dans la structure genemap
Finsi
Finsi
Jusqu'à (trouver un codon start et stop)
Fin

Algorithm 2 Génération d'ARN

Entrées : genmap : gm ; seq ^ caractère
Sortie : ARNm
Var : posarn, pos,i : entier
ARNm[gm.gencounter] : Tableau de caractères
Début
Initialiser les variables locales à 0
Pour(i ← 0 jusqu'à gm.genecounter faire)
initialiser pos au début du gene et réinitialiser posarn à 0
Tantque (on a pas arrivé à la fin de gène)faire
Si (seq[pos]='T')
ARNm[i][posarn]←'U'
Sinon
ARNm[i][posarn]←seq[pos]
pos←pos+1
posarn←posarn+1
Finsi
FinTantque
Finpour
Fin

Algorithm 3 Génération des protéines

Entrées : arm, codons : ^caractère
Var :
i ←0 , nbprot←$longeur(arn)/3$:entier
pos←0 :entier
symbols[nbprot] : Tableau d'entiers
prot[3] : Tableau de caractères
Début
Pour(i←0 jusqu'a nbprot)
récupérer les trois nucléotides (codon) dans le tableau prot
Tantque (on a pas trouver l'acide aminée correspond au codon et que on a pas arrivé a la fin du tableau de codons) faire
i←i+3
symbols[pos]←i+2
FinTantque
Finpour
afficher le tableau de symboles des codons trouvés
Fin

Algorithm 4 Détection des mutations

```
Entrées : gene : ^ caractère
var : pos, occurrence, cptzar : entier
riskrate : réel
SEUIL←10 : constante
Début
    initialiser tous les variables à 0
    zar[1000] : Tableau d'entier initialiser à 0
     Tanque(gene[pos] <> 0)
        Si on croise un G ou un C
            Si (occurance=0)
            occurrence←occurrence+1
            Sinon
                Si (occurence>0)
                /* incrémenter le compteur de zone/*
                cptzar←cptzar+1
                calculer le pourcentage de risque de mutation(risquerate)
                    Si((riskrate>SEUIL)
                    zar[i]←-cptzar
                    /* incrémenter le nombre de zone à risque/*
                     i←i+1
                    Finsi
                 réinitialiser l'occurrence pour terminer la zone
                 Finsi
            Finsi
        afficher le caractère courant
        pos←pos+1
      FinTanque
    Si (zar[0]=0)
    aucune zone n'est à risque de mutation
    Sinon
    afficher les zone qui sont à risque mutation
    Finsi
Fin
```

Algorithm 5 Taux de correspondance

Entrées : seq, seq2 : *caractre*
Var : d : réel
pos, mini, i : entier
Début
d←0, pos←0
mini=min (seq, seq2)
Tanque (seq[pos+mini-1]<> 0)
Afficher les morceaux des deux séquences à comparer
faire un XOR aux deux chaînes caractère par caractère
compter le nombre total des caractères différents entre les deux séquences (seq, seq2)
d ← d * 100
d ← d / (8*mini)
réinitialiser d à 0
pos←pos+1
FinTanque
Fin

Références

[1] *Clang C Language Family Frontend for LLVM*. https ://clang.llvm.org/.

[2] *GCC, the GNU Compiler Collection*. https ://gcc.gnu.org/.

[3] *Intel oneAPI HPC Toolkit*. https ://software.intel.com.

[4] *Optimizing subroutines in assembly language*. https ://agner.org/optimize/optimizing$_a$*ssembly.pdf*.