



# Projet parallélisme I2

# Sommaire

Stratégie de parallélisme et optimisations .....	2
Enregistrement binaire des nucléotides .....	2
Occupation mémoire et utilisation réseau .....	3
Processus de lecture .....	4
Communications .....	5
Algorithme de la partie communicante.....	6
Tests.....	7
1 processus .....	7
2 processus .....	7
3 processus .....	7
Code source.....	8
look4genes.h.....	8
look4genes.c.....	10

## Stratégie de parallélisme et optimisations

La première opération qui est faite en parallèle dans le programme est la lecture du fichier d'ADN, puisque chaque processus lit une partie qui lui est propre. On ajoute 2 nucléotides redondants au début et à la fin de cette partie (càd qui se trouvent aussi sur le processus précédent et suivant) afin que les processus puissent lire leur partie dans les 6 sens de lecture.

Les nucléotides lus ayant 4 valeurs possibles, il nous a semblé judicieux d'optimiser l'occupation mémoire de notre programme en utilisant 2 bits pour représenter un nucléotide au lieu d'un char (8 bits) comme dans le fichier. Le C ne proposant pas ce type de donnée, on utilise une structure de donnée de type int pour stocker 16 nucléotides. Cela permet aussi une accélération I/O puisque qu'une structure int est traitée en une passe par les processeurs 32bits.

Afin de réaliser cela on utilise les fonctions « bitwise » du C (décalage binaire et opération logiques binaires).

### ***Enregistrement binaire des nucléotides***

```
/*
 * 00 : A (0x0)
 * 01 : C (0x1)
 * 10 : G (0x2)
 * 11 : T (0x3)
 */
```

ADN : TA-AC-CC-TA-AC-CC-TA-AC  
Binaire : 1100-0001-0101-1100-0001-0101-1100-0001  
C (hexa) : `int brin_adn = 0xC15C15C1;`

Pour lire un le nucléotide à la position 4 (C) dans `brin_adn` il faut faire un ET binaire avec 0000-0011-0000-0000-0000-0000-0000-0000 (0x03000000), puis décaler le resultat obtenu de 24 bits vers la droite :

```
int nucleotide = brin_adn & 0x03000000;  
nucleotide = nucleotide << 24;  
if (nucleotide == 0x1) printf("nucléotide C trouvé en position 4 dans  
brin_adn");
```

La plupart des opérations binaires que l'on effectue dans notre programme sont en fait déportées dans des fonctions ce qui permet d'en faire abstraction dans une bonne partie du traitement et de bien comprendre la partie principale du programme qui est la recherche d'exons.

### **Occupation mémoire et utilisation réseau**

En négligeant les variables en nombre constant dans notre programme et la liste chaînée des codons start (puisqu'elle est vidée à chaque codon stop rencontré), on peut évaluer l'occupation mémoire totale de notre programme de la façon suivante :

$$\frac{\text{tailledufichier}}{4} + (nb\_procs - 1) \times 1$$

Ceci nous donne donc l'occupation mémoire en octet,  $(nb\_procs - 1) \times 1$  étant les octets pour les nucléotides redondants. Cette évaluation est donc valable pour au moins 3 processeur et une taille de fichier à traiter importante, puisque qu'on néglige l'occupation mémoire du code.

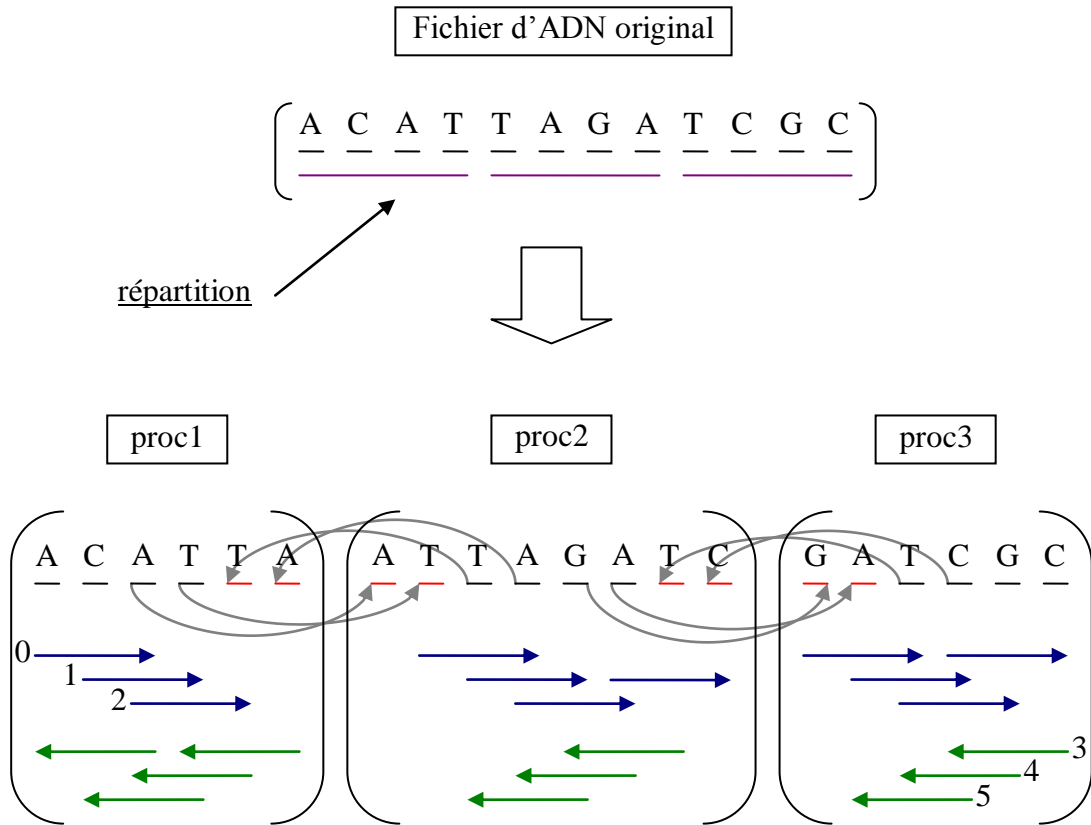
Pour le traitement d'un fichier d'ADN de 250 Mo sur 5 processeurs, on a donc une occupation mémoire par processeur pendant le traitement d'environ 12 Mo.

On a choisit de ne pas faire une lecture séquentielle du fichier (au fur et à mesure que l'on trouve des exons), car le nombre des appels système serait trop important, d'autant plus que le fichier se trouve à la base sur le processeur 0 et donc tout ce qui est lu doit passer par le réseau entre les processeurs.

Il serait possible d'optimiser l'occupation de la bande passante réseau en lisant un fichier d'ADN binaire et non ASCII (réduction de l'utilisation de la BP par 4 lors de la lecture des partitions du fichier).

# Processus de lecture

- nucléotide    — nucléotide redondant
- lectures 0 à 2 (gauche-droite)   ← lectures 3 à 5 (gauche-droite)



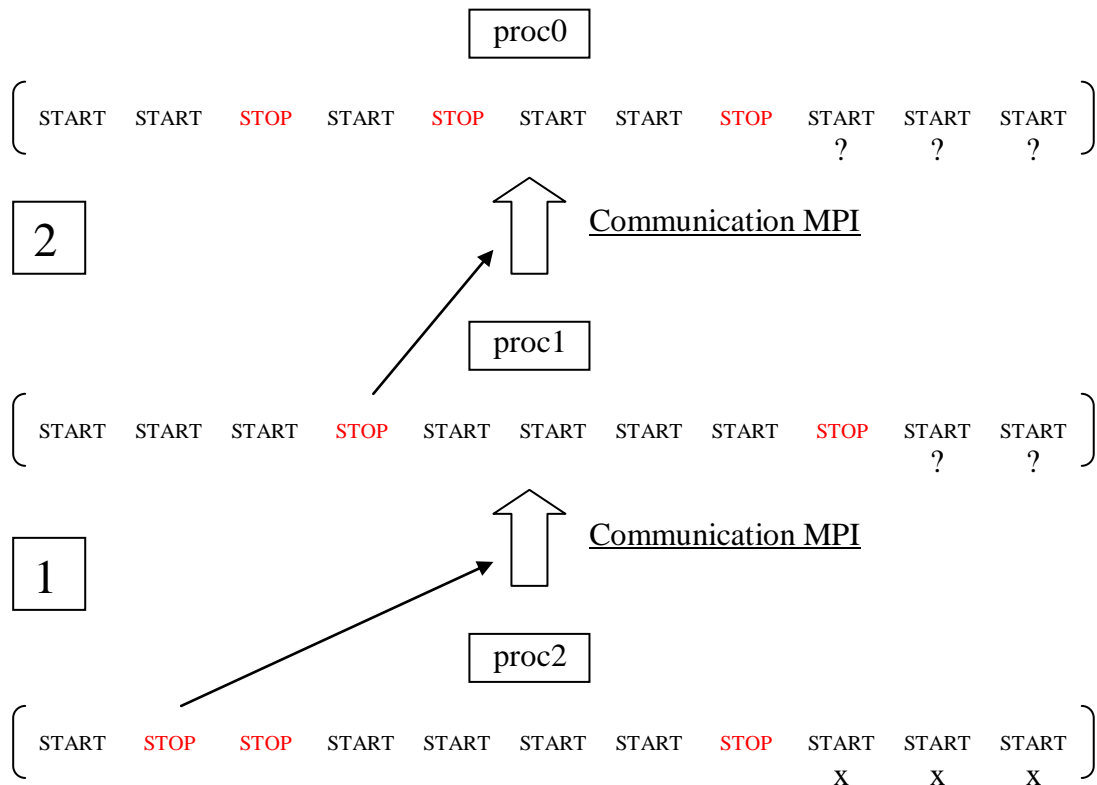
Note : Les nucléotides redondant sont lus lors de la lecture du fichier d'ADN et ne sont pas obtenu en échangeant des messages entre les processus.

Utilisation des triplets redondants : Lors de la première lecture, les processus n'utilisent pas les nucléotides redondants se trouvant à la fin de leur lecture, lors de la deuxième, ils peuvent en utiliser un si cela leur permet de faire un triplet supplémentaire, et lors de la troisième, deux. Les nucléotides redondants se trouvant en début de lecture sont utilisés dans tout les cas pour lire les nucléotides qui n'ont pas été lus par le processus précédent dans la lecture (on sait jusqu'où à lu le processus d'avant a lu des triplets avec simple modulo 3 sur l'emplacement où commence le processus en cours).

# Communications

On utilise les communications MPI afin de savoir quels sont les prochains codons STOP qui se trouvent sur le processus de rang supérieur (en lecture de gauche à droite) ou inférieur (en lecture de droite à gauche). C'est la seule communication nécessaire. Il faut tout de même connaître cette information pour les 6 sens de lecture. Afin que les processeurs ne cherchent pas cette information avant de la transmettre, on effectue cette transmission une fois que les processeurs ont déjà analysé toute leur portion d'ADN. Au premier codon stop trouvé dans le sens de lecture s lors de la recherche contigüe, on enregistre sa position dans premier\_stop[s] afin de le transmettre au processeur qui le précède dans le sens de lecture.

Exemple : en lecture 0 on trouve les codons actifs suivants



Note : Les communications se font dans le sens inverse de la lecture : cela permet au processus qui n'a aucun codon STOP (donc pas de premier codon STOP) de communiquer la position du codon STOP du processus qui est après lui dans la lecture.

## ***Algorithme de la partie communicante***

### lecture lect de 0 à 5

- \* on raisonne ici entièrement dans un sens de lecture lect et par triplets
- \* on est sur proc n/N
  
- début de la lecture first\_stop est à NOT\_SET sur chaque proc, au premier stop rencontré lors de la lecture, l'initialiser avec la position du stop
  
- \* lect sur tout les procs : tout les codons start sont enregistrés (leur position) dans une liste chaînée, dès qu'on tombe sur un codon stop, on écrit les exons correspondants trouvés sur la sortie (ou sur un fichier) et on vide la liste (c'est une optimisation qui permet d'économiser l'espace mémoire utilisé)
  
- \* (optimisation : au fur et a mesure de la lecture, a chaque stop rencontré, vider la mémoire dans le fichier)
  
- \* ex : condons\_start[0 (lecture)][9] = 100 (position du premier nucléotide du codon);
  
- \* les processus ont tous terminé leur lecture, s'il leur reste des codons start il faut maintenant qu'il connaissent la position du premier codon stop qui se trouve après ces codons start
  
- \*\*\*\*\*
- \* transmissions des prochains codons de stop :
  
- tout les proc en ecoute sauf n
  
- (n) fait savoir la position de son premier stop ou NOT\_SET à (n-1) -> next\_stop  
  (n-1) recoit et ne se met plus en ecoute
  
- (n-1) fait savoir la position de son premier stop ou next\_stop à (n-2) -> next\_stop  
  (n-2) recoit et ne se met plus en ecoute
  
- ...
- ...
  
- (1) fait savoir la position de son premier stop ou next\_stop à (0) -> next\_stop  
  0 recoit et ne se met plus en ecoute
  
- \* fin
- \*\*\*\*\*
  
- \* pour chaque proc prendre la liste des codons start restants pour chaque start :
  - ecrire l'intervalle du gene du start à next\_stop dans genes\_l\_n\_inter



# Code source

## look4genes.h

```
/*
 * Projet parallélisme I2
 * -----
 * Tarik ANSARI, Florent GRANGENOIS
 * 14 avril 2006
 *
 * look4genes.h
 * Recherche d'exons dans l'ADN
 */

#include "mpi.h"
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0
#define NOT_SET -1

int nb_procs, rang;
int tag = 100;
MPI_Status status;

// root est un "boolean" pour savoir si on est sur le proc 0, tail pour savoir si on est sur
le dernier proc
int root = FALSE, tail = FALSE;
int t; // taille totale du fichier (nucléotides)

// -- buffer des nucléotides

// type de donnée optimisé on stocke 1 AA sur 2 bit au lieu d'1 char (8bits)
typedef struct
{
    int d; // début de la portion d'ADN sur ce proc par rapport au fichier
d'origine
    int n; // nombre de nucléotides
    int t; // taille de la structure
    unsigned int *chaine; // le type int est traité en une passe par les proc 32 bits, on
stocke 16 AA sur un maillon
} buff_adn;
buff_adn proc_adn;

/*
 * 00 : A
 * 01 : C
 * 10 : G
 * 11 : T
 */

void lecture_fichier(char *fichier);
void adn_write(char buff[16], int index);

// offset que doit utiliser le proc dans la lecture de sa partie (sens gauche et droit)
int lg_offset, ld_offset;

// -- lecture des AA

typedef enum {CODON_START, CODON_STOP, CODON_AUTRE} codon_borne;
typedef enum {GAUCHE, DROITE} sens_lecture;

codon_borne type_codon(int position, sens_lecture sens);

int *buff_starts[6]; // positions des codons start
int size_starts[6] = {0, 0, 0, 0, 0, 0};
void ajout_start (int lect, int pos);
```

```

/*
* 0 : lecture gauche-droite en partant de 0
* 1 : lecture gauche-droite en partant de +1
* 2 : lecture gauche-droite en partant de +2
* 3 : lecture droite-gauche en partant de 0
* 4 : lecture droite-gauche en partant de +1
* 5 : lecture droite-gauche en partant de +2
*/

// position générale du premier codon STOP trouvé par ce proc (pour chaque sens de lecture)
int premier_stop[6] = {NOT_SET, NOT_SET, NOT_SET, NOT_SET, NOT_SET, NOT_SET};
// position générale des premiers codon STOP trouvé par le proc suivant (par rapport au sens
de lecture)
int next_stop[6] = {NOT_SET, NOT_SET, NOT_SET, NOT_SET, NOT_SET, NOT_SET};

/*
* Recherche de protéines fonctionnelles :
* -----
* tous les exons inférieurs à taille_mini_exon seront exclus de la recherche, si
* taille_mini_exon = 0, il n'y a pas de taille minimum
*
* ! dans notre algorithme, un codon STOP interrompt la recherche avec les codons
*   START qui le précédaient, même si la distance est inférieure à taille_mini_exon
*/
int taille_mini_exon = 0; // essayer avec 300

```

## look4genes.c

```
/*
 * Projet parallélisme I2
 * -----
 * Tarik ANSARI, Florent GRANGENOIS
 * 14 avril 2006
 *
 * look4genes.c
 * Recherche d'exons dans l'ADN
 */

#include "look4genes.h"

main(int argc, char **argv)
{
int i, j, s, c, d, stop, count exons = 0, total exons;
sens_lecture sens = GAUCHE;
codon_borne tcodon;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rang);
    if (rang == 0) root = TRUE;
    if (rang == nb_procs - 1) tail = TRUE;

    if (argc < 1)
    {
        if (root) fprintf(stderr, "Veuillez entrer le nom du fichier à traiter en
argument\n");
        return 0;
    }

    // -- étape 1 : lecture d'une partie du fichier pour chaque proc
    lecture_fichier(argv[1]);

    // -- étape 2 : réalisation des 6 sens de lecture
    printf("taille p%d : %d\n", rang, proc_adn.n);
    for (i = 0; i < 6; i++)
    {
        d = i;
        if (i > 2)
        {
            sens = DROITE;
            d -= 3;
        }

        for (c = (sens == GAUCHE && root) ? d : ((sens == DROITE && tail) ? d :
proc_adn.n - d) % 3));
            c < proc_adn.n - 2; c += 3)
        {
            tcodon = type_codon(c, sens);

            if (tcodon != CODON AUTRE)
            {
                if (tcodon == CODON_STOP)
                {
                    // on a trouvé un codon STOP
                    stop = c + ((sens == GAUCHE) ? proc_adn.d : (t -
proc_adn.d - proc_adn.n));

                    //printf("p%d : codon stop en %d (lect %d)\n", rang, stop,
i);

                    if (premier_stop[i] == NOT_SET)
                        premier_stop[i] = stop;

                    // parcours des START trouvés avant (on est dans le même
sens de lecture)
                    for (s = 0; s < size starts[i]; s++)
                    {
                        // on ne prend pas en compte les exons plus petits
que taille_mini_exon
```

```

        if (stop - buff_starts[i][s] >= taille_mini_exon
|| !taille_mini_exon)
        {
            // ajout des coordonnées des exons trouvés
            dans un fichier intermédiaire
            printf("protéine trouvée sur p%d : %d - %d
(sens %s)\n", rang, buff_starts[i][s], stop, (sens == GAUCHE) ? "normal" : "inverse");
            count exons++;
        }
    }
    // le STOP est interrompant dans notre algorithme : on
oublit tous les codons START trouvés avant
    if (size_starts[i]) free(buff_starts[i]);
    size_starts[i] = 0;
}
else
{
    // on ajoute le codon START à la liste des codons START
trouvés
    ajout start(i, c + ((sens == GAUCHE) ? proc_adn.d : (t -
proc_adn.d - proc_adn.n)));
}
}
}

// -- étape 3 : récupération des codons STOP du proc de rang supérieur
// cette étape nous permet de trouver les exons qui ont été coupés par
// le découpage du fichier, il suffit d'avoir le premier codon STOP du
// proc de rang supérieur au proc en cours

// tous les procs se mettent en écoute sauf le dernier (transmission des
// codons stop en cascade inversée)

for (i = 0; i < 2 && nb_procs > 1; i++)
{
    if (i == 0)
        sens = GAUCHE;
    else
        sens = DROITE;

    // réception des 3 condon STOP suivant (dans une direction de lecture)

    if (!(sens == GAUCHE) ? tail : root)
    {
        // on écoute le proc de rang supérieur (par rapport au sens de lecture)
        MPI_Recv(&next_stop[3 * i], 3, MPI_INT, rang + ((sens == GAUCHE) ? 1 :
(-1)), tag, MPI_COMM_WORLD, &status);
    }

    // on a reçu la donnée, on transmet maintenant au proc de dessous (par rapport
au sens de lecture)
    if (!(sens == GAUCHE) ? root : tail)
    {
        for (j = 0; j < 3; j++)
            if (premier_stop[3 * i + j] == NOT_SET)
                premier_stop[3 * i + j] = next_stop[3 * i + j];

        MPI_Send(&premier_stop[3 * i], 3, MPI_INT, rang - ((sens == GAUCHE) ? 1 :
(-1)), tag, MPI_COMM_WORLD);
    }
}

// -- étape 4 : recherche des exons intermédiaires
sens = GAUCHE;
for (i = 0; i < 6; i++)
{
    d = i;
    if (i > 2)
    {
        sens = DROITE;
        d -= 3;
    }

    if (next_stop[i] != NOT_SET)
    {

```

```

        stop = next_stop[i];

        // parcours des START trouvés avant (on est dans le même sens de lecture)
        for (s = 0; s < size starts[i]; s++)
        {
            // on ne prend pas en compte les exons plus petits que
            taille_mini_exon
            if (stop - buff starts[i][s] >= taille mini_exon
            || !taille_mini_exon)
            {
                // ajout des coordonnées des exons trouvés dans un
                fichier intermédiaire
                printf("protéine trouvée sur p%d (inter) : %d - %d
                (sens %s)\n", rang, buff_starts[i][s], stop, (sens == GAUCHE) ? "normal" : "inverse");
                count_exons++;
            }
        }
    }

    // total de count exon entre les procs
    MPI_Reduce(&count_exons, &total_exons, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (root) printf("Total d'exons trouvés : %d\n", total_exons);

    return 0;
}

void ajout_start (int lect, int pos)
{
    size_starts[lect]++;

    if (size_starts[lect] == 1)
        buff_starts[lect] = (int*) malloc(sizeof(int));
    else
        buff_starts[lect] = (int*) realloc(buff_starts[lect], size_starts[lect] *
sizeof(int));

    buff_starts[lect][size_starts[lect] - 1] = pos;
}

void lecture_fichier(char *fichier)
{
    int i;
    struct stat fstat;
    int fd; // descripteur de fichier
    int e, r;
    int debut, taille, fin;
    char buff[16];

    /*
    * ! le fichier en entrée doit impérativement être UNIQUEMENT composé des
    * caractères ASCII 'A', 'C', 'G', 'T'
    */

    if (stat(fichier, &fstat) < 0)
    {
        if (root) fprintf(stderr, "Erreur infos %s\n", fichier);
        exit(0);
    }

    t = fstat.st_size; // nombre de nucléotides dans l'ADN à lire

    if (t < nb_procs * 3)
    {
        if (root) fprintf(stderr, "Taille du fichier à traiter insuffisante pour le
nombre de processus (%d)\n", nb_procs);
        exit(0);
    }

    if ((fd = open(fichier, O_RDONLY)) < 0)
    {
        if (root) fprintf(stderr, "Erreur ouverture %s\n", fichier);
        exit(0);
    }

    /*
    * on va maintenant calculer la partie que chaque proc doit lire

```

```

* on divise le fichier en parts équitables mais chaque proc va
* lire 2 nucléotides supplémentaires avant et après sa partie,
* afin de que les procs puissent effectuer tout les sens de
* lectures sans avoir à faire des communications lourdes en
* ressources
*
* ces 2 nucléotides supplémentaires lus en redondance avant et
* après permettent de plus à tous les procs de pouvoir lire des
* AA dans toutes les situations et non des nucléotides seuls
*
* la taille du fichier nous est donnée par fstat.st_size
* qui nous donne la taille du fichier en octets, dans le fichier
* "épuré" chaque octet est un char ASCII correspondant à 1
* nucléotide (pas de retours charriot)
*/

e = t / nb_procs; // partie entière de la répartition
r = t % nb_procs; // reste a répartir

// que le proc en doit lire
debut = rang * e + ((r > rang) ? rang : r) - ((root) ? 0 : 2);
taille = e + ((rang < r) ? 1 : 0) + ((tail && root) ? 0 : ((tail || root) ? 2 : 4));
fin = debut + taille;

// calcul de l'offset pour lire par codons (triplets)
lg_offset = - debut % 3; // lecture gauche-droite
ld_offset = - (t - fin) % 3; // lecture droite-gauche

// dimensionement de la structure ADN
proc_adn.d = debut;
proc_adn.n = taille;
proc_adn.t = (taille / 16) + ((taille % 16) ? 1 : 0);
proc_adn.chaine = (int*) malloc(proc_adn.t * sizeof(int));

// lecture de la partie du fichier à la position définie
// et transtypage dans la structure optimisée
for (i = 0; i < proc_adn.t; i++)
{
    pread(fd, buff, (i < taille / 16) ? 16 : (taille % 16), debut + i * 16);
    adn_write(buff, i);
}

close(fd);
}

void adn_write(char buff[16], int index)
{
    int i;
    unsigned int tmp;

    proc_adn.chaine[index] = 0;

    for (i = 0; i < 16; i++)
    {
        tmp = 0; // initialisation : case 'A'

        switch (buff[i])
        {
            case 'C':
                tmp = 0x40000000;
                break;
            case 'G':
                tmp = 0x80000000;
                break;
            case 'T':
                tmp = 0xC0000000;
                break;
        }

        proc_adn.chaine[index] |= tmp >> i * 2;
    }
}

codon_borne type_codon(int position, sens_lecture sens)
{
    char codon[3];
    unsigned int tmp;

```

```

int i, pos, dec;
int f = 0;

/*
 * fonction importante :
 * retourne directement le type d'un codon à la position donnée
 * de son premier nucléotide (position relative) dans proc_adn
 */

// prise en compte du décalage induit par la structure int en blocs de 16 nucléotides
// (le dernier block n'est pas forcément complètement rempli donc il faut décaler)
if (sens == DROITE)
    position += (proc_adn.t * 16) % proc_adn.n;

for (i = 0; i < 3; i++)
{
    dec = (position % 16) + i;
    pos = position / 16 + ((int) dec / 16);

    tmp = 0xC0000000;
    tmp &= proc_adn.chaine[(sens == GAUCHE) ? pos : proc_adn.t - 1 - pos] << ((sens
== GAUCHE) ? (dec % 16) * 2 : 32 - (2 * (dec % 16) + 2));

    switch (tmp)
    {
    case 0x00000000:
        codon[i] = 'A';
        break;
    case 0x40000000:
        codon[i] = 'C';
        break;
    case 0x80000000:
        codon[i] = 'G';
        break;
    case 0xC0000000:
        codon[i] = 'T';
        break;
    }
}

if (strcmp(codon, "TAA", 3) == 0 || strcmp(codon, "TAG", 3) == 0 || strcmp(codon,
"TTA", 3) == 0)
    return CODON_STOP;
else if (strcmp(codon, "ATG", 3) == 0)
    return CODON_START;
else
    return CODON_AUTRE;
}

```