INF7440 : Devoir #3

(À remettre *au plus tard* lundi 19 nov. 2007, à 13h00)

1. Parallélisme récursif (10 pts)

```
procedure maxsum( int x[*], int l, int u ) returns int r
{
  if (l  > u) { r = 0;              return; }
  if (l == u) { r = max( 0, x[l] ); return; }
  int m = (l + u) / 2;
  int lmax = 0;
  int sum = 0;
  for [i = m downto l] {
    sum += x[i];
    lmax = max( lmax, sum );
  }
  int rmax = 0;
  sum = 0;
  for [i = m+1 to u] {
    sum += x[i];
    rmax = max( rmax, sum );
  }
  r = max( lmax + rmax, maxsum(x, l, m), maxsum(x, m+1, u) );
}
```

Algorithme 1: Algorithme diviser-pour-régner pour le calcul de la somme maximum d'éléments contigus

Soit une séquence de valeurs $x = [x_1, \ldots, x_n]$. Parmi toutes les sous-séquences possibles d'éléments adjacents de x, on désire déterminer la somme maximale possible. Par exemple, soit le tableau x suivant :

$$x = [31, -41, 59, 26, -53, 58, 97, -93, -23, 84]$$

L'algorithme doit alors retourner 187, qui correspond à la somme des éléments de la sous-séquence x`[3:7]` — $59 + 26 - 53 + 58 + 97$.

Bentley, dans son livre «*Programming pearls*» [Ben00], présente divers algorithmes pour résoudre ce problème, dont un algorithme récursif basé sur une approche diviser-pour-régner.

L'algorithme 1 présente une version MPD de l'algorithme récursif présenté par Bentley[1]

a. Quelles modifications faudrait-il effectuer à cet algorithme pour obtenir un algorithme *avec parallélisme récursif* ?

b. Donnez alors les caractéristiques de l'algorithme parallèle résultant : temps d'exécution, nombre de processeurs et coût. Indiquez aussi si l'algorithme est optimal *en coût*.

   Pour le temps d'exécution, indiquez *explicitement* les équations de récurrence appropriées avant de donner votre solution Θ.

[1] Cet algorithme, ainsi que d'autres versions séquentielles, sont disponibles sur le site Web de l'auteur: `http://www.cs.bell-labs.com/cm/cs/pearls/sketch08.html`.

2. Alignement global de deux séquences (30 pts)

Description du problème

Le problème qui suit est un problème classique de bio-informatique. Les explications, exemples et équations récursives qui suivent sont basés sur un document de K.-M. Chao intitulé «*Dynamic-Programming Strategies for Analyzing Biomolecular Sequences*» [Cha04].

Soit deux séquences $A = [a_1, \ldots, a_{n_1}]$ et $B = [b_1, \ldots, b_{n_2}]$. Un *alignement (global)* de A et B est obtenu en introduisant des tirets ("–", qu'on appelle aussi des *gaps*) entre certains éléments des séquences de façon à ce que :

- les deux séquences deviennent de même longueur ;
- pour un index donné i, $A[i]$ et $B[i]$ ne sont jamais tous les deux un tiret.

Un score est associé à un alignement, qui dépend de la présence de tirets et de la présence ou non d'éléments identiques aux différentes positions des séquences.

Plus précisément, le *score* d'un alignement entre deux séquences est donné par la somme des scores aux différentes positions. Si les deux éléments à une position donnée sont identiques, le score est donné par `valMatch`, un nombre positif. Si l'un des éléments est un tiret, alors le score est donné par `valGap`, un nombre négatif. Autrement (deux éléments différents, dont aucun n'est un *gap*), le score est donné par `valMismatch`, aussi un nombre négatif.

Par exemple, soit les valeurs suivantes [Cha04] :

- `valMatch` $= +8$
- `valMismatch` $= -5$
- `valGap` $= -3$

L'exemple qui suit présente alors le score pour un alignement possible, *non optimal*, entre les deux séquences «CTTAACT» et «CGGATCAT» :

C	–	–	–	T	T	A	A	C	T		
C	G	G	A	T	C	A	–	–	T		
+8	−3	−3	−3	+8	−5	+8	−3	−3	+8	=	+12

Notons par $S(i, j)$ le score pour un alignement global optimal entre $[a_1, \ldots, a_i]$ et $[b_1, \ldots, b_j]$. La valeur de $S(i, j)$, avec des valeurs initiales appropriées, peut être définie par l'équation de récurrence suivante :

$$S(i,j) = max \begin{cases} S(i-1,j) + \texttt{valGap} \\ S(i,j-1) + \texttt{valGap} \\ S(i-1,j-1) + score(a_i, b_j) \end{cases}$$

$$score(a,b) = \begin{cases} \texttt{valMatch} & \text{si } a = b \\ \texttt{valMismatch} & \text{si } a \neq b \end{cases}$$

La figure 1 donne, sous forme tabulaire, un alignement *optimal* entre les séquences «CTTAACT» et «CGGATCAT».

		C	G	G	A	T	C	A	T
	0	-3	-6	-9	-12	-15	-18	-21	-24
C	-3	8	5	2	-1	-4	-7	-10	-13
T	-6	5	3	0	-3	7	4	1	2
T	-9	2	0	-2	-5	5	-1	-4	9
A	-12	-1	-3	-5	6	3	0	7	6
A	-15	-4	-6	-8	3	1	-2	8	5
C	-18	-7	-9	-11	0	-2	9	6	3
T	-21	-10	-12	-14	-3	8	6	4	**14**

Figure 1: Un exemple d'alignement optimal et son score (tiré de [Cha04]).

Ce que vous devez faire

Pour cet exercice, vous devez développer et coder (en MPD) **deux versions** d'une fonction qui détermine le coût optimal (maximum) d'un alignement global entre deux séquences. L'interface de ces fonctions est présentée dans l'extrait de Code MPD 1 (p. 5).

Plus précisément, les deux versions à développer sont les suivantes :

1. Une version avec *parallélisme récursif* basée sur une approche diviser-pour-régner (sans mémorisation) : `alignerRec`.
2. Une version avec *parallélisme itératif* (à granularité fine) basée sur une approche de programmation dynamique (ascendante) : `alignerPD`.

Dans les deux cas, il suffit de trouver le *score* de l'alignement optimal, et non d'identifier l'alignement lui-même.

À remettre

Vous devrez remettre votre programme final (fichier `aligner.mpd` : voir plus bas) en exécutant la commande suivante sur la machine `arabica` : [2]

`oto rendre_tp tremblay INF7440` <*codePerm*> `aligner.mpd`

Questions additionnelles concernant l'analyse des algorithmes

a. Pour l'algorithme avec parallélisme récursif, choisissez une *opération barométrique* et indiquez les équations de récurrence appropriées pour le temps d'exécution.

   Ensuite, *n'essayez pas de résoudre cette équation directement.* Déterminez plutôt la complexité asymptotique du temps d'exécution en examinant l'arbre des appels et en déterminant sa profondeur maximale.

b. Pour l'algorithme de programmation dynamique avec parallélisme itératif, donnez les caractéristiques de l'algorithme parallèle résultant, c'est-à-dire, temps d'exécution, nombre (maximum) de processeurs et coût. Indiquez aussi si l'algorithme est optimal en coût.

   Note : Pour le temps d'exécution, avant de donner l'expression Θ, indiquez *explicitement* l'expression représentant le nombre (exact) de fois où les opérations que vous analysez sont éxécutées. Indiquez aussi s'il s'agit d'opérations élémentaires ou d'opérations barométriques ; dans ce dernier cas, justifiez brièvement le choix de cette opération ou de ces opérations.

[2] Vous pouvez aussi utiliser l'URL suivant : `http://labunix.uqam.ca:8181/~oto/application-web`.

Contraintes de mise en oeuvre

Vos algorithmes devront avoir été compilés et exécutés de façon à assurer leur bon fonctionnement. Plus précisément, vous trouverez sur le site *web* du cours divers éléments que vous devez utiliser : `http://www.info.uqam.ca/~tremblay/INF7440/Devoirs` :

- `aligner.mpd` : Ressource (`global`) définissant l'en-tête des deux versions des fonctions et divers autres éléments (constantes, types, variables, etc.).
- `tests.mpd`: Cas et suites de tests, définis à l'aide de `MPDUnit`.
- `MPDUnit.mpd` et `MPDUnit-body.mpd` : Cadre de tests pour MPD.
- `OpsAuxiliaires.mpd` : Fichier avec opérations auxiliaires, entre autres, `assert`.
- `Makefile`.

  Les principales commandes sont les suivantes :

  - «`make`» : compile les fichiers requis par le programme `tests.mpd`.
  - «`make tests`» : compile les fichiers requis puis exécute le programme de tests — l'exécutable est dans `a.out`.

    Cette commande exécute les tests pour les deux versions (récursive et programmation dynamique). Pour exécuter les tests pour une seule des versions, il suffit d'exécuter directement la commande `a.out` avec l'argument approprié.
  - «`make clean`» : fait le ménage, c'est-à-dire, supprime les fichiers qui peuvent être générés de façon automatique.

Quelques indices

- En MPD, une `string` est simplement un tableau de caractères. Les divers éléments d'une `string` peuvent donc être obtenus à l'aide d'une opération d'indexation, le premier élément étant à l'index `1`.
- Les boucles `for` et les instructions `co` de création d'instances multiples de processus concurrents peuvent utiliser des espaces d'indexation *à deux (ou plusieurs) dimensions* :

```
for [i = 1 to n, j = 1 to m] { ... }

co [i = 1 to n, j = 1 to m]
 ...
oc
```

- Dans un espace d'indexation, il est possible d'utiliser une clause `st`, qui spécifie une contrainte qui doit être satisfaite pour que l'élément sélectionné soit effectivement traité par le `for` ou le `co`. Par exemple, la clause d'indexation suivante couvre toutes les paires d'index `i` et `j` telles que `i != j` :

```
for [i = 1 to n, j = 1 to m st i != j] {
  ...
}
```

  De façon générale, n'importe quelle condition, simple ou complexe, peut apparaître après `st`.

- Lors de l'analyse d'un `for` ou `co` avec clause `st`, vous pouvez ne considérer dans l'analyse que les valeurs d'index pour lesquelles la contrainte `st` est satisfaite.

- Dans la version avec programmation dynamique et parallélisme itératif à granularité fine, examiner les dépendances de données de façon à bien identifier ce qui peut être évalué en parallèle. De façon générale,dans une telle forme de parallélisme, n'importe quel ensemble d'éléments peut possiblement être évalués de façon parallèle, pas uniquement des singletons, des lignes ou des colonnes...
- Le fichier `aligner.mpd` contient une fonction `id`, utile pour satisfaire la contrainte syntaxique de MPD qui veut qu'un `co` ne puisse contenir que des invocations de fonctions de procédures — entre autres, un appel à `max` n'est pas considéré comme un appel de fonction, `max` étant une primitive du langage (comme `abs`, `+`, etc.). L'utilisation d'une expression de type `int` comme argument à cette fonction permet donc d'utiliser n'importe quelle expression (entière) comme affectation à une variable dans un `co`.

Références

[Ben00] J. Bentley. *Programming Pearls (Second Edition).* Addison-Wesley, 2000.

[Cha04] K.-M. Chao. Dynamic programming strategies for analyzing biomolecular sequences. In *Selected Topics in Post-genome Knowledge Discovery.* Singapore University Press, 2004. Disponible sur `http://www.lacim.uqam.ca/~chauve/Enseignement/INF7440/A04/COURS4/DOC-Biomolecular-Sequences.pdf`.


```
global Aligner

  # Nombre maximum de caracteres dans les sequences a aligner.
  const int MAX_CARS = 32;

  # Les types pour les sequences et leurs elements.
  type Sequence = string[MAX_CARS];
  type Element  = char;


  # Type procedure d'alignement : utile pour les tests.
  optype ProcAligner = (Sequence, int, Sequence, int) returns int;


  # Deux variantes de procedure d'alignement global.
  op alignerRec( Sequence g1, int n1, Sequence g2, int n2 ) returns int r;
  op alignerPD ( Sequence g1, int n1, Sequence g2, int n2 ) returns int r;


  # Operation pour specifier les trois scores possibles pour des elements.
  # Doit etre appelee avant d'effectuer un appel a alignerRec ou alignerPD.
  op definirScores( int match, int mismatch, int gap );

body Aligner
  # Certains elements (definirScores, score) sont deja definis.

  # D'autres sont a definir: alignerRec, alignerPD.
```

Code MPD 1: En-tête définissant l'interface des opérations à mettre en oeuvre.