INF7440 : Devoir #4

(À remettre *au plus tard* jeudi 6 décembre 2007, à 21h00)

Les deux questions de ce devoir portent sur la ressource `DPR2`, dont l'interface est présentée dans l'extrait de code MPD 1 (p. 2). Cette ressource définit un module «générique» pour la résolution de problèmes à l'aide de la stratégie Diviser-Pour-Régner avec décomposition dichotomique (d'où le «2»).

Plus précisément, le module `DPR2` permet de traiter des problèmes pour lesquels les données du problème peuvent être représentées par un tableau d'entiers, et où le résultat (la solution du problème) *est un simple entier* (scalaire). Le domaine d'application de ce module est donc limité, mais le module peut quand même être utilisé pour résoudre divers problèmes, comme nous allons le voir dans ce devoir.

L'extrait de code MPD 2 présente une utilisation de ce module pour calculer le maximum d'une série de nombres.[1] La méthode appelée, `resoudreRecursif`, utilise du parallélisme récursif pour produire la solution — pour la mise en oeuvre de cette opération, voir le contenu du fichier `dpr2.mpd`. (Signalons que cette ressource illustre le style de programmation *basée* objets (i.e., avec des objets, mais sans héritage) qu'il est possible de faire en MPD.)

Deux questions portant 1) sur l'utilisation de ce module et 2) sur sa mise en oeuvre sont décrites plus bas.

À remettre

Vous devrez remettre votre solution aux deux exercices (fichiers `calcul-somme-max.mpd` et `dpr2.mpd`) en exécutant la commande suivante sur la machine `arabica` :[2]

```
oto rendre_tp tremblay INF7440 <codePerm> calcul-somme-max.mpd  dpr2.mpd
```

Contraintes de mise en oeuvre

Votre code devra avoir été compilé et exécuté de façon à assurer son 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` :

- `dpr2.mpd` : Ressource définissant l'en-tête de deux opérations (une déjà mise en oeuvre, l'autre que vous devez réaliser) et divers autres éléments (constantes, types, etc.).
- `calcul-max.mpd` : Les deux opérations (`resoudreSimpleMax` et `combinerMax`) permettant, dans le fichier `tests.mpd`, d'utiliser le module `DPR2` pour calculer la valeur maximum parmi une série de nombres.
- `calcul-somme-max.mpd` : Les *squelettes* (procédures bidons, sans mise en oeuvre) des deux opérations (`resoudreSimpleSommeMax` et `combinerSommeMax`) permettant, dans le fichier `tests.mpd`, d'utiliser le module `DPR2` pour calculer la somme maximum d'éléments contigus parmi une série de nombres.
- `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`.

[1] Il s'agit d'une version simplifiée du code qui vous est fourni dans les fichiers `tests.mpd` et `calcul-max.mpd`.
[2] Vous pouvez aussi utiliser l'URL suivant : `http://labunix.uqam.ca:8181/~oto/application-web`

```
resource DPR2
  # Type pour representer un probleme = tableau d'entiers.
  type Probleme = [*]int;

  # Type pour representer une solution = un simple entier.
  type Solution = int;


  # Type procedure permettant de resoudre directement un probleme simple.
  # PRECONDITION
  #   i <= j & j-i < tailleSimple
  # POSTCONDITION
  #   r contient la solution au sous-probleme x[i:j]
  optype ResoudreSimple( ref Probleme x, int i, int j, res Solution r );


  # Type procedure permettant de combiner deux sous-solutions.
  # PRECONDITION
  #   i <= j
  #   r1 represente la solution pour x[i:m]
  #   r2 represente la solution pour x[m+1:j]
  #     avec m = (i + j) / 2
  # POSTCONDITION
  #   r contient la solution combinant r1 et r2 pour le sous-probleme x[i:j]
  optype CombinerSolutions( ref Probleme x, int i, int j, Solution r1, Solution r2,
                            res Solution r );


  # Operation publique pour resoudre un probleme facon recursive.
  # PRECONDITION
  #   tailleSimple >= 1
  # POSTCONDITION
  #   r = la solution au probleme x
  op resoudreRecursif( Probleme x, int tailleSimple, res Solution r );

  # Operation publique pour resoudre un probleme sans recursion.
  # PRECONDITION
  #   tailleSimple >= 1
  # POSTCONDITION
  #   r = la solution au probleme x
  op resoudreIteratif( Probleme x, int tailleSimple, res Solution r );

body DPR2( cap ResoudreSimple rs, cap CombinerSolutions cs )

  # ARGUMENTS
  #   rs: procedure pour resoudre un probleme simple
  #   cs: procedure pour combiner deux sous-solutions au probleme de depart
  .
  .
  .
end
```

Code MPD 1: Interface du module `DPR2` (extraits du fichier `dpr2.mpd`).

```
procedure resoudreSimpleMax( ref Probleme x, int i, int j, res Solution r )
{
  r = low(int);
  for [k = i to j] {
    r = max( r, x[k] )
  }
}

procedure combinerMax( ref Probleme x, int i, int j, Solution r1, Solution r2,
                       res Solution r )
{
  r = max( r1, r2 );
}

procedure testMax1()
{
  nommerCasDeTest( "Test max 1" );
  int x[1] = ([1] 10);
  int r;
  cap DPR2 dpr2 = create DPR2( resoudreSimpleMax, combinerMax );
  dpr2.resoudreRecursif( x, 1, r );
  assertIntEquals( 10, r );
}

procedure testMax2()
{
  nommerCasDeTest( "Test max 2" );
  const int N = 16;
  int x[N] = (3, 4, 10, 5, -3, 0, 45, 34, 23, 21, 22, 12, 12, -1, 2, 9);
  int r;

  cap DPR2 dpr2 = create DPR2( resoudreSimpleMax, combinerMax );
  for [k = 0 to lg(N, 2)] {
    dpr2.resoudreRecursif( x, 2**k, r );
    assertIntEquals( 45, r );
  }
}

procedure suite1() returns SuiteDeTests suite
{ suite = mkSuiteDeTests( "Suite max", null, (testMax1, testMax2), null ); }
```

Code MPD 2: Utilisation du module DPR2 pour calculer le maximum d'une série de nombres.

- `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 itérative). Pour exécuter les tests pour une seule des versions, il suffit d'exécuter directement la commande `a.out` avec les arguments appropriés.

    De plus, différents sous-ensembles de tests ont aussi été définis, pour faciliter le développement, étape par étape, des deux questions :

    * `testsr1` : Tests pour la version récursive utilisant toujours la valeur 1 comme taille de problème simple.
    * `testsr` : Tests pour la version récursive avec différentes valeurs pour la taille d'un problème simple.
    * `testsi1` : Tests pour la version itérative utilisant toujours la valeur 1 comme taille de problème simple.
    * `testsi` : Tests pour la version itérative, avec différentes valeurs pour la taille d'un problème simple.
  - «`make clean`» : fait le ménage, c'est-à-dire, supprime les fichiers qui peuvent être générés de façon automatique.

---

1. Utilisation du module `DPR2` pour calculer la somme maximum d'éléments contigus (20 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.

Dans l'exercice no. 1 du devoir 3, nous avons vu une procédure récursive diviser-pour-régner pour résoudre le problème de trouver la somme maximum d'éléments contigus dans

```
float alg4c()
{   int i;
    float maxsofar = 0, maxendinghere = 0;
    for (i = 0; i < n; i++) {
        maxendinghere += x[i];
        maxendinghere = max(maxendinghere, 0);
        maxsofar = max(maxsofar, maxendinghere);
    }
    return maxsofar;
}
```

Algorithme 2: Algorithme itératif pour le calcul de la somme maximum d'éléments contigus (tiré du site Web de Bentley).

une séquence d'entiers. Cette procédure, présentée dans l'algorithme 1, est l'une des nombreuses variantes présentées par Bentley dans son livre «*Programming pearls*» [Ben00] et disponibles sur son site Web : `http://www.cs.bell-labs.com/cm/cs/pearls/sketch08.html`. L'algorithme 2, exprimé en `C` et tiré directement du site Web de Bentley, présente une autre variante, la plus simple et la plus efficace.

En vous inspirant de ces deux algorithmes, complétez les deux procédures du fichier `calcul-somme-max.mpd` qui vont permettre, dans les cas de tests du fichier `tests.mpd`, d'utiliser le module `DPR2` pour résoudre le problème de trouver la somme maximum d'éléments contigus dans une série d'entiers :

```
procedure resoudreSimpleSommeMax
          ( ref Probleme x, int i, int j, res Solution r )

procedure combinerSommeMax
          ( ref Probleme x, int i, int j, Solution r1, Solution r2, res Solution r )
```

Donnez les caractéristiques de l'algorithme parallèle résultant, en supposant que l'argument utilisé pour spécifier la taille d'un problème simple lors de l'appel à `dpr2.resoudreRecursif` est une *constante* c :

a. Temps d'exécution.

b. Nombre de processeurs.

c. Coût. Optimalité en coût?

d. Travail. Optimalité en travail?

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

Pour l'optimalité en coût ou en travail, comparez avec l'algorithme 2 (p. 5).

2. Mise en oeuvre de `resoudreIteratif` avec parallélisme itératif simulant le calcul ascendant d'une récursion, *mais sans récursion* (20 pts)

Le module `DPR2.mpd` exporte une opération `resoudreIteratif`, dont l'interface est exactement la même que `resoudreRecursif` :

```
op resoudreIteratif( Probleme x, int tailleSimple, res Solution r );
```

Écrivez le code MPD permettant de mettre en oeuvre cette opération, et ce en utilisant la stratégie de dédoublement récursif/réduction β-logarithmique vue en cours :

- Exercice (fait en classe) sur la mise en oeuvre de l'opération `reduce` : `http://www.info2.uqam.ca/~tremblay/INF7440/SolutionsExercices/map-reduce.mpd`.
- Sections 3 et 4 des notes de cours sur le modèle PRAM.

En d'autres mots, il faut que le comportement de cette opération soit semblable à celui de la version récursive, mais *sans utiliser de récursion.*

Répondez ensuite aux questions suivantes :

a. Donnez les caractéristiques de l'algorithme parallèle résultant pour le calcul de la somme maximum d'éléments contigus (donc comme à la question précédente), en supposant que l'argument utilisé pour spécifier la taille d'un problème simple lors de l'appel à `dpr2.resoudreIteratif` est une constante c :

   (i) Temps d'exécution.

   (ii) Nombre de processeurs.

   (iii) Coût. Optimalité en coût?

b. Quel serait le coût si la taille indiquée pour un problème simple était plutôt lg `n`? L'algorithme serait-il alors optimal en coût, si on le compare à l'algorithme 2? Expliquez brièvement.

Références

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