Paradigmes de base de la programmation concurrente

Table des matières

Programmation concurrente

– Programme concurrent

= programme qui contient *plusieurs* processus (ou *thread*s) qui *coopèrent*

Coopération ⇒ Communication, échange d'information

Deux principales façons de communiquer :

– Par l'intermédiaire de variables *partagées*
– Par l'échange de messages et de signaux (par ex., *canaux* de communication)

Note : On considère un *thread* comme étant un processus léger (*lightweight process*)

– Différents types de programmes concurrents

- Programme multi-contextes (*multi-threaded*) = contient plusieurs *thread*s

  Utilisation : Pour mieux organiser/structurer une application (plus grande modularité)

  Exemples : Système d'exploitation multi-tâches, fureteurs multi-tâches, interface personnes-machines vs. logique d'affaire

- Programme parallèle = chaque processus s'exécute sur son propre processeur

  Utilisation : Pour résoudre plus rapidement un problème ou pour résoudre un problème plus gros

  Exemples : Prévisions météorologiques, prospection minière, physique moderne, bio-informatique (génomique)

- Programme distribué = les processus communiquent entre eux par l'intermédiaire d'un réseau (⇒ délais plus longs)

  Utilisation : Pour répartir, géographiquement, les données et les traitements

  Exemples : Serveurs de fichiers, accès à distance à des banques de données

Note : les catégories précédentes *ne sont pas* mutuellement exclusives. Par exemple, de nombreuses machines parallèles modernes (qu'on utilise essentiellement pour développer des applications *parallèles*) sont des *multi-ordinateurs*, donc des machines composées d'un ensemble de processeurs (avec leur mémoire) réliés par un réseau.

0 Patrons de base de la programmation concurrente

Les différents types d'applications concurrentes peuvent généralement être réalisées en utilisant un certain nombre de *patrons* de base. Évidemment, certains de ces patrons sont plus appropriés pour certains types d'applications concurrente — par exemple, le patron client/serveur est souvent utilisé pour des applications distribuées. Toutefois, aucun de ces patrons n'est exclusif à un type d'application. Un programme parallèle, par exemple, pourrait donc être développé en utilisant un ou plusieurs des patrons présentés plus bas.

Les cinq (5) patrons de base de la programmation concurrente selon G.R. Andrews [And00] sont les suivants :

1. Parallélisme itératif : Programme contenant plusieurs processus avec des boucles, et communicant généralement par l'intermédiaire de variables partagées.
2. Parallélisme récursif : Programme avec procédures récursives exécutées en parallèle, avec communication par variables partagées.
3. Producteurs/consommateurs (filtres et pipelines) : Ensemble de processus qui communiquent entre eux de façon uni-directionnelle, généralement organisés sous forme de pipeline.
4. Clients/serveurs : Les processus clients font des requêtes et attendent les réponses ; les processus serveurs attendent les requêtes et y répondent. Communication *bi*-directionnelle.
5. Pairs interagissant (*interacting peers*) : Les processus exécutent (en gros) le même code et *s'échangent des messages* pour coopérer et accomplir leur tâche.

Remarque importante : plusieurs des exemples de programmes MPD présentés dans les pages qui suivent (plus précisément,ceux en anglais) sont tirés du site *web* de G.R. Andrews, le concepteur du langage MPD :

`http://www.cs.arizona.edu/mpd/programs/tutorial.html`

Consultez ce site *web* pour plus de détails sur le langage (ainsi que le chapitre "Le langage MPD") et pour d'autres exemples de programmes.

1 Parallélisme itératif

- Programme itératif = utilise des boucles pour examiner des données et calculer des résultats
- Programme parallèle itératif = contient deux ou plusieurs processus itératifs
- Chaque processus calcule les résultats pour un sous-ensemble des données, puis les résultats sont combinés
- Souvent utilisé pour des calculs *embarassingly parallel* = problèmes pouvant être décomposés en un grand nombre de parties indépendantes, générant ainsi une grande quantité de parallélisme
- Exemples : Deux versions de la multiplication de matrices :
  - Programme MPD 1 : version avec instruction `co` (processus créés de façon dynamique).

– Programme MPD 2 : version avec déclarations `process` (déclaration de processus en *arrière-plan* avec création semi-dynamique).

  Note : la clause `final` (Programme MPD 2) assure que la série d'instructions qui suit le `final` ne s'exécutera que lorsque *tous* les processus créés par le programme auront terminé ou seront bloqués.

Si on ignore le temps requis pour générer les matrices `a` et `b` ainsi que le temps pour imprimer le résultat (matrice `c`), donc si on ne considère que la partie effectuant la multiplication des matrices, ces deux algorithmes ont alors les caractéristiques suivantes :

– Programme MPD 1 : Exactement $\mathtt{n}^2$ processus sont créés par le programme (on ne compte pas le programme principal lui-même comme un processus). Chaque processus s'exécute en temps $\Theta(\mathtt{n})$ (boucle `for` de la procédure `inner`). Si on considère qu'un processeur distinct est associé à chaque processus, le coût de l'algorithme (voir chapitre suivant) sera donc $\Theta(\mathtt{n}^3)$ (n^2 processeurs durant un temps $\Theta(n)$).
– Programme MPD 2 : Même caractéristiques.

Programme MPD 1 Multiplication parallèle de matrices à granularité fine : version avec instruction `co` (processus créées de façon dynamique) [Exemple G.R. Andrews]

```
# matrix multiplication using co and fine-grained concurrency
# usage:  a.out size

resource matrix_mult()
  # read command line argument for matrix sizes
  int n; getarg(1,n);

  # declare matrices and initialize a and b
  real a[n,n] = ([n] ([n] 1.0)),
       b[n,n] = ([n] ([n] 1.0)),
       c[n,n];

  # compute inner product of a[i,*] * b[*,j]
  procedure inner(int i, int j) {
    real sum = 0.0;
    for [k = 1 to n] {
      sum += a[i,k] * b[k,j];
    }
    c[i,j] = sum;
  }

  # compute n**2 inner products concurrently
  co [i = 1 to n, j = 1 to n]
    inner(i,j);
  oc

  # print result, by rows
  for [i = 1 to n] {
    for [j = 1 to n] {
      writes(c[i,j], " ");
    }                          # one line per row
    write();                   # append a newline
  }
end
```

Programme MPD 2 Multiplication parallèle de matrices à granularité fine : version avec `process` (processus définis à l'aide de déclarations, donc création semi-dynamique) [Exemple G.R. Andrews]

```
# matrix multiplication using process declarations and fine-grained concurrency
# usage:  a.out size

resource matrix_mult()
  # read command line argument for matrix sizes
  int n; getarg(1,n);

  # declare matrices and initialize a and b
  real a[n,n] = ([n] ([n] 1.0)),
       b[n,n] = ([n] ([n] 1.0)),
       c[n,n];

  # compute inner product of a[i,*] * b[*,j]
  process inner[i = 1 to n, j = 1 to n] {
    real sum = 0.0;
    for [k = 1 to n] {
      sum += a[i,k] * b[k,j];
    }
    c[i,j] = sum;
  }

  # wait for processes to terminate, then print result, by rows
  final {
    for [i = 1 to n] {
      for [j = 1 to n] {
        writes(c[i,j], " ");
      }                          # one line per row
      write();                   # append a newline
    }
  }
end
```

Remarque sur le parallélisme itératif

Certains programmes ou procédures, bien qu'ils ne contiennent pas de boucles comme tel, entrent dans la catégorie des programmes concurrents avec parallélisme *itératif*. C'est le cas lorsque ces programmes *ne contiennent pas de récursivité* et ne correspondent pas non plus à aucun des autres paradigmes présentés plus bas. Par exemple, la procédure `multVectScal` suivante — elle multiplie un vecteur `a` de taille `n` par une constante `c` (un *scalaire*) pour produire un vecteur résultat `b` lui aussi de taille `n` — utilise une forme de parallélisme itératif :

```
procedure fois( int x, int y ) returns int z
{ z = x * y; }

procedure multVectScal( int a[*], int c, res int b[*], int n )
{
  co [i = 1 to n]
    b[i] = fois(a[i], c)
  oc
}
```

Ici, il s'agit de parallélisme itératif à *granularité (très) fine* (*fine granularity*). On dit que la granularité est *fine* lorsque les processus n'exécutent qu'un petit nombre d'instructions. Lorsque les processus exécutent un grand nombre d'instructions, on parle alors de *granularité grossière* (*coarse granularity*).

Notons que, dans cet exemple, il a été nécessaire d'introduire une fonction auxiliaire `fois` (pour multiplier deux entiers) puisque les restrictions syntaxiques de MPD font que seules des *activations* de fonctions ou procédures peuvent apparaître dans un `co`.

2 Parallélisme récursif

- Le parallélisme récursif est utile si le problème peut être résolu à l'aide d'un algorithme contenant de nombreux appels récursifs (par ex., diviser-pour-régner), appels récursifs pouvant s'évaluer de façon *indépendante* les uns des autres $\Rightarrow$ on fait les appels en parallèle

- Si trop de parallélisme (trop nombreux appels récursifs), alors l'arbre de récursion peut être tronqué (élagué, en anglais *pruned*), e.g., solution non récursive/non parallèle lorsque le sous-problème devient *suffisamment simple* — même principe que celui décrit à la Section 2.7 du manuel

- Exemples :
  - Intégration numérique, Programme MPD 3.
  - Factoriel, Programme MPD 4.

    Examinons le comportement de ce dernier algorithme lorsque `seuil = 1` (donc avec récursion jusqu'aux cas de base *triviaux*, c'est-à-dire un seul élément à traiter). Pour simplifier, supposons que $n = 2^k$:
    - Nombre total de processus : chaque niveau de récursion va générer deux nouveaux processus. L'appel pour le problème de taille n va donc générer deux (2) processus pour traiter les problèmes de taille $n/2$, lesquels vont générer (au total) quatre processus pour les problèmes de taille $n/4$, etc., jusqu'à ce que n processus pour les problèmes de taille `1` soient générés (au niveau k). Au total, on aura donc le nombre suivant de processus :

$$\sum_{i=1}^{k} 2^i$$

Programme MPD 3 Programme avec processus récursifs pour intégration d'une fonction [Exemple G.R. Andrews]

```
# a parallel recursive MPD program for approximating the integral
# of f(x) from x==a to x==b for f(x) = sin(x)*exp(x)

# usage:  a.out epsilon a b

resource parallel_quad()
  real epsilon; getarg(1, epsilon);
  real a, b; getarg(2, a); getarg(3, b);

  procedure f(real x) returns real fx {
    fx = sin(x) * exp(x);
  }

  procedure quad(real left, real right, real fleft, real fright, real lrarea)
                          returns real area {
    real mid = (left+right)/ 2;
    real fmid = f(mid);
    real larea = (fleft + fmid) * (mid-left) / 2;     # left area
    real rarea = (fmid + fright) * (right-mid) / 2;   # right area
    if (abs((larea+rarea) - lrarea) > epsilon) {
      # recurse to integrate both halves in parallel
      co larea = quad(left, mid, fleft, fmid, larea)
      // rarea = quad(mid, right, fmid, fright, rarea)
      oc;
    }
    area = larea + rarea;
  }

  int start = age();    # start time, in milliseconds
  real area = quad(a, b, f(a), f(b), (f(a)+f(b))*(b-a)/2);
  int finish = age();   # finish time, in milliseconds

  write("epsilon =", epsilon," a =", a, " b = ", b);
  write("area =", area, " time = ",finish-start);
end
```

Programme MPD 4 Fonction factoriel avec parallélisme récursif et élagage (utilisation d'une solution séquentielle pour problème simple)

```
resource factoriel()
  int n;     getarg(1, n);
  int seuil; getarg(2, seuil);

  procedure fact( int i, int j, int seuil ) returns int resultat
  {
    if (j - i <= seuil) {
      # Probleme simple: solution sequentielle et iterative

      resultat = 1;
      for [k = i to j] {
        resultat *=  k;
      }
    }
    else {
      # Probleme plus complexe: solution parallele et recursive

      int r1, r2;
      int mid = (i+j) / 2;

      co  r1 = fact(i,     mid, seuil);
      //  r2 = fact(mid+1, j,   seuil);
      oc
      resultat = r1 * r2;
    }
  }

  int resultat = fact( 1, n, seuil );
  write( "fact(", n, ") = ", resultat );
end
```

En d'autres mots, $2(n-1)$ processus seront créés au total (le premier appel par le programme principal n'est pas compté comme un processus).

* Nombre de processeurs : parmi les divers processus créés, à un instant donné, au plus n d'entre eux auront à être actifs de façon véritablement *concurrente* (pour traiter les n feuilles en parallèle). On peut donc utiliser n processeurs pour traiter l'ensemble des processus.
* Les divers processus à un même niveau de récursion peuvent tous s'exécuter en parallèle. Le temps total requis dépendra donc (asymptotiquement) du nombre de niveaux de récursion, c'est-à-dire, de la profondeur de l'arbre. Ceci peut être décrit par l'équation de récurrence suivante :

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= T(n/2) + \Theta(1) \end{aligned}$$

  Le temps d'exécution sera donc $\Theta(\lg\, n)$.
* Le coût sera $\Theta(n \lg\, n)$.

3 Producteurs/consommateurs (pipeline)

- Producteur = processus qui produit un flot (*stream*) de résultats
- Consommateur = processus qui reçoit un flot de données et le traite
- Filtre = processus qui consomme un flot en entrée et produit un nouveau flot en sortie
- Pipeline = séquence de filtres
- Exemple : Utilisation de processus et *pipes* sur Unix pour supprimer les commandes d'un fichier LaTeX, Script 1. Les commandes utilisées sont les suivantes (description produite par `man`) :
  - `sed` : "*The* `sed` *utility is a stream editor that reads one or more text files, makes editing changes according to a script of editing commands, and writes the results to standard output.*
  - `grep` : "*The* grep *utility searches files for a pattern and prints all lines that contain that pattern.*"

    Note : Rôle de l'option "`-v`" de `grep` : "*Print all lines except those that contain the pattern.*
  - `tr` : "*The* `tr` *utility copies the standard input to the standard output with substitution or deletion of selected characters.*".
  - `wc` : "*The* `wc` *utility reads one or more input files and, by default, writes the number of newline characters, words and bytes contained in each input file to the standard output.*".

    Note : Rôle de l'option "`-w`" de `wc` : "*Count words delimited by white space characters or new line characters.*"

4 Clients/serveurs

– Producteur/consommateur ⇒ flot uni-directionnel d'information (du producteur vers le consommateur)

– Client/serveur ⇒ flot *bi-directionnel*

Script 1 Script Unix pour la suppression des commandes LaTeX dans un fichier

```
cat $1.tex \
 | sed '/\\begin{programmeMPD}/,/\\end{programmeMPD}/d' \
 | sed '/\\begin{table}/,/\\end{table}/d' \
 | sed '/\\begin{picture}/,/\\end{picture}/d' \
 | grep -v "^%" \
 | tr "[~]" "[ ]" \
 | tr "[\t]" "[\n]"\
 | tr "[ ]" "[\n]"\
 | grep -v '\\'  \
 | wc -w
```

- Le client fait une requête au serveur puis attend la réponse
- Le serveur reçoit la requête du client, la traite, puis retourne une réponse

– Analogie :

- Client = procédure appelante
- Serveur = procédure appelée (par plusieurs autres)

– Exemples : systèmes d'exploitation, serveurs de fichiers, bases de données, RPC (*Remote Procedure Call*) et RMI (*Remote Method Invocation*, en Java)

– Style généralement utilisé dans les systèmes répartis et distribués plutôt que dans les programmes parallèles

5 Pairs interagissants

– *Peers* = pairs ("*n.m.* Personne semblable quant à la fonction, la situation sociale", selon le petit Robert)

– *Interacting peers*

⇒ processus égaux, interagissant généralement (mais pas toujours) de façon *symétrique* entre eux

= chaque processus exécute, *grosso modo*, le même algorithme (style SPMD = *Single Program Multiple Data*) et communique avec les autres processus de façon à calculer sa partie des résultats

⇒ communication par échange de messages

⇒ les données de chaque processus sont strictement privées

– Exemple : multiplication de matrices à l'aide d'échange de messages : Programmes MPD 5 et 6. Le programme utilise un processus coordonnateur (Programme MPD 5) et W processus "*esclaves*" (travailleurs) (Programme MPD 6). Chaque travailleur calcule N/W rangées du résultat (pour simplifier, on suppose que N est un multiple de W).

Programme MPD 5 Multiplication de matrices à l'aide de processus distribués interagissant par l'intermédiaire d'un coordonnateur [Exemple G.R. Andrews]

```
# Distributed matrix multiplication using message passing.
# Usage:  a.out N numWorkers

resource distributed_matrix_mult()
  # read command line arguments for matrix sizes and numWorkers
  int n; getarg(1, n);
  int numWorkers; getarg(2, numWorkers);
  if ((n % numWorkers) != 0)
    { write("N must be a multiple of numWorkers"); stop(1); }
  int stripSize = n/numWorkers;

  op data  [numWorkers] (real m[*,*]);    # channels to Workers
  op result[numWorkers] (real m[*,*]);    # channels to Coordinator

  process Coordinator {
    real a[n,n] = ([n] ([n] 1.0)),
         b[n,n] = ([n] ([n] 1.0)),
         c[n,n];
    # send strips of a[*,*] and all of b[*,*] to Workers
    for [w = 1 to numWorkers] {
      int startRow = stripSize*(w-1) + 1;
      int endRow = startRow + stripSize - 1;
      send data[w] (a[startRow:endRow,*]);
      send data[w](b[*,*]);
    }
    # gather results from Workers
    for [w = 1 to numWorkers] {
      int startRow = stripSize*(w-1) + 1;
      int endRow = startRow + stripSize - 1;
      receive result[w](c[startRow:endRow,*]);
    }
    # print results
    for [i = 1 to n] {
      for [j = 1 to n] { writes(c[i,j]); writes(" "); }
      write();
    }
  }
```

Programme MPD 6 Multiplication de matrices à l'aide de processus distribués (suite) [Exemple G.R. Andrews]

```
  process Worker[w = 1 to numWorkers] {
    real a[stripSize,n], b[n,n], c[stripSize,n];
    # receive strip of a and all of b from Coordinator
    receive data[w](a[*,*]);
    receive data[w](b[*,*]);
    # compute inner products of a[i,*] * b[*,j]
    for [i = 1 to stripSize, j = 1 to n] {
      real sum = 0.0;
      for [k = 1 to n]
        { sum += a[i,k] * b[k,j]; }
      c[i,j] = sum;
    }
    # send results back to Coordinator
    send result[w](c[*,*]);
  }

end distributed_matrix_mult
```

A Différentes façons de diviser-pour-régner

Comme on l'a vu dans la première partie du cours, il existe différentes façons d'utiliser la stratégie diviser-pour-régner. Dans le cas d'un programme concurrent, cela est d'autant plus vrai qu'il existe, pour une décomposition donnée en sous-problèmes, différentes façons d'associer les divers sous-problèmes à des processus. Voyons quelques exemples, qui présentent des façons différentes de calculer la somme des éléments d'un tableau.

A.1 Décomposition récursive

Programme MPD 7 Somme des éléments d'un tableau avec décomposition récursive

```
resource sommeTableau()
  # Lecture et verification du nombre d'elements a generer et a traiter
  int n; getarg(1, n);
  if (n <= 0) { write("*** Erreur: n <= 0" ); stop(1); }

  int a[1:n];             # Tableau dont on veut calculer la somme des elements.

  # Generation (aleatoire) des elements du tableau a.
  procedure generer( res int a[1:*], int n ) {
    for [i = 1 to n] { a[i] = int(random(1, n)) }
  }

  # Impression des elements du tableau a.
  procedure imprimer( int a[1:*], int n ) {
    for [i = 1 to n-1] { writes( a[i], ", " ) }
    write( a[n] );
  }

  # Somme (recursive) des elements a[i..j]
  procedure somme( int i, int j ) returns int resultat {
    # Utilise (en lecture seulement) la variable globale a.
    if ( i == j ) {
      resultat = a[i];
    } else {
      int r1, r2, m = (i + j) / 2;
      co   r1 = somme(i,   m);
      //   r2 = somme(m+1, j);
      oc
      resultat = r1 + r2;
    }
  }

  #
  # Programme principal
  #

  # Generation aleatoire des elements
  generer(a, n)

  # Calcul de la somme
  int s = somme( 1, n );

  # Impression des resultats
  imprimer( a, n );
  write( "somme", " = ", s );
end
```

Dans le programme MPD 7, chaque instance (récursive) de la procédure `somme` génère un processus distinct. Le nombre total de processus créés dépend donc de la taille du tableau à traiter.

A.2 Décomposition avec un nombre statique de processus

Dans le programme MPD 8, on crée un nombre fixe de processus — `NBPROCS`, un paramètre spécifié de façon statique, c'est-à-dire, au moment de la compilation. Pour simplifier, on suppose que le nombre d'éléments à additionner est un multiple du nombre de processus.

Chacun de ces processus est responsable de calculer la somme d'un sous-intervalle du tableau `a` (plus précisément, le processus `i` doit faire la somme du sous-tableau allant de la borne `inf(i, n, NBPROCS)` jusqu'à la borne `sup(i, n, NBPROCS)` inclusivement).

Les résultats intermédiaires sont transmis par l'intermédiaire d'une variable partagée, à savoir le tableau `toti` (plus précisément, le résultat produit par le processus `i` est conservé à la position `i` du tableau `toti`). Ces résultats intermédiaires sont *ensuite* additionnés les uns avec les autres dans la boucle `for` du programme principal.

Les parties du code indiquées par `begin` et `final` s'exécutent, respectivement, *avant* le début de l'exécution des processus en arrière-plan et *après* la fin de l'exécution de ces processus.

A.3 Décomposition style "sac de tâches"

Dans le programme MPD 9, plutôt que de fixer le nombre de processus, on spécifie plutôt la taille désirée pour chacune des tâches. Ici, cette taille est spécifiée par la variable `tailleTache` (spécifiée au moment de l'appel du programme), qui indique la taille du sous-intervalle devant être traité par chacune des instances de la procédure `somme`. Pour simplifier, on suppose que le nombre total d'éléments du tableau est un multiple de la taille des tâches. Dans l'exemple du programme MPD 9, on crée un processus pour chacun des sous-intervalles (`nbProcs = n/tailleTache`). Comme on a plusieurs processus qui vont accéder, en lecture *et* écriture, une même variable partagée, on utilise donc un sémaphore pour en protéger l'accès (`totLibre`).

Une telle approche où on fixe la taille des tâches qu'on répartit ensuite entre les divers processus disponibles est habituellement utilisée dans le cadre d'une approche dite "*sac de tâches*" (*bag of tasks*). Lorsqu'on utilise cette stratégie, on crée un certain nombre de processus (nombre qui peut être déterminé par la structure de la machine, c'est-à-dire, par le nombre de processeurs). On crée aussi, de façon indépendante, un certain nombre de tâches (plus précisément, de *descripteurs* de tâches), tâches qu'on insère dans une structure de données appropriée, appelée *sac de tâches* (en anglais, *bag of tasks*, parfois aussi appelé, *task pool*). Lorsqu'un processus/processeur devient libre, il choisit alors une des tâches disponibles dans le sac et l'exécute. Dans certains cas, l'exécution d'une telle tâche génère alors de nouvelles tâches, lesquelles sont simplement ajoutées au sac. Le programme dans son ensemble se termine lorsqu'il ne reste plus *aucune* tâche à exécuter, c'est-à-dire lorsque le sac de tâches est vide. Le programme MPD 10 illustre l'allure générale d'une telle mise en oeuvre pour le problème du calcul de la somme des éléments d'un tableau.

Programme MPD 8 Somme des éléments d'un tableau avec décomposition statique (en fonction d'un nombre fixe de processus)

```
resource sommeTableau()
  const int NBPROCS = 10;     # Nombre de processus a creer

  # Lecture et verification du nombre d'elements a generer et a traiter
  int n;
  getarg(1, n);
  if (n <= 0 | (n % NBPROCS) != 0) {
    write("*** Erreur: n <= 0 | ( n %", NBPROCS, #") != 0" ); stop(1);
  }

  int a[1:n];

  int toti[1:NBPROCS];        # Tableau pour resultats intermediaires

  # Bornes inferieure et superieure du i-ieme intervalle.
  procedure inf( int i, int n, int nbProcs ) returns int r
  { r = (i-1) * (n / nbProcs) + 1 }
  procedure sup( int i, int n, int nbProcs ) returns int r
  { r = i * (n / nbProcs) }

  ...

  # Processus (iteratifs) pour la somme des elements d'un sous-intervalle
  process somme[i = 1 to NBPROCS] {
    int resultat = 0;
    for [k = inf(i, n, NBPROCS) to sup(i, n, NBPROCS)] {
      resultat += a[k];
    }
    toti[i] = resultat;
  }

  #
  # Programme principal
  #
  begin {
    # Generation aleatoire des elements
    generer(a, n);
  }

  # Calcul de la somme
  final {
    int tot = 0;
    for [i = 1 to NBPROCS] {
      tot += toti[i];
    }

    # Impression des resultats
    imprimer( a, n );
    write( "somme", " = ", tot );
  }
end
```

Programme MPD 9 Somme des éléments d'un tableau avec décomposition en tâches de taille fixe (style "sac de tâches")

```
resource sommeTableau()
  # Lecture et verification du nombre d'elements a generer et a traiter,
  # de meme que du nombre d'elements a traiter par processus.
  ...
  int tailleTache; getarg(2, tailleTache);
  if (tailleTache <= 0 | tailleTache > n | (n % tailleTache != 0) ) {
    write("*** Erreur: tailleTache <= 0 | tailleTache > ", n, "| n %", tailleTache, "!= 0" ); stop(1);
  }


  # La matrice a traiter.
  int a[1:n];

  int tot = 0;          # Variable globale mise a jour directement par les processus.
  sem totLibre = 1;     # Semaphore protegeant l'acces a tot.

  # Bornes inferieure et superieure du i-ieme intervalle.
  procedure inf( int i, int tailleTache ) returns int r
  { r = (i-1) * tailleTache + 1 }
  procedure sup( int i, int tailleTache ) returns int r
  { r = i * tailleTache }

  ...

  # Somme (iterative) des elements a[i..j] avec mise a jour de la var. globale
  procedure somme( int i, int j ) {
    int resultat = 0;
    for [k = i to j] {
      resultat += a[k];
    }
    P(totLibre); tot += resultat; V(totLibre)
  }

  #
  # Programme principal
  #

  # Generation aleatoire des elements
  generer(a, n);

  # Calcul de la somme.
  int nbProcs = n / tailleTache;
  co [i = 1 to nbProcs]
    somme( inf(i, tailleTache), sup(i, tailleTache) )
  oc

  # Impression des resultats
  ...
end
```

Programme MPD 10 Somme des éléments d'un tableau avec un véritable "sac de tâches"

```
resource sommeTableau()
  ...
  int nbProcs; getarg(3, nbProcs);
  if (nbProcs <= 0) { write("*** Erreur: nbProcs <= 0" ); stop(1); }
  ...

  #
  # Structures de donnees et operations pour gestion du sac de taches.
  #
  ...
  # Ajout d'une tache (sous-intervalle a[i..j]) dans le sac.
  procedure ajouterTache( int i, int j ) { ... }

  # Retrait d'une tache du sac.
  procedure obtenirTache( var int i, var int j ) returns bool disponible { ... }

  # Selection d'une tache et execution.
  procedure somme() {
    int i, j;
    while ( obtenirTache(i, j) ) {
      int resultat = 0;
      for [k = i to j] {
        resultat += a[k];
      }
      P(totLibre); tot += resultat; V(totLibre);
    }
  }

  ...

  # Ajout des taches dans le sac.
  for [i = 1 to n / tailleTache] {
    ajouterTache(inf(i, tailleTache), sup(i, tailleTache));
  }

  # Activation des taches.
  co [i = 1 to nbProcs]
    somme()
  oc
  # Se termine lorsque tous les processus ont termine, c'est-a-dire
  # lorsqu'ils ont tous detecte que le sac de taches etait vide.

  # Impression des resultats
  ...
end
```

B Une autre façon de classifier les paradigmes de programmation parallèle

Une autre classification intéressante des modèles de programmation parallèle, qui peut aussi aider à mieux comprendre certaines notions importantes de conception d'algorithmes parallèles, est celle qui distingue entre parallélisme de données, parallélisme de contrôle et parallélisme de flux. Ces trois styles de programmation parallèle sont présentés comme suit par Gengler, Ubéda et Desprez [GUD96, p. 97].

> - Le **parallélisme de données** (*data parallelism*) exploite comme source de parallélisme la régularité des données et applique en parallèle un même calcul à des données distinctes. [...]
> - Le **parallélisme de contrôle** (*control parallelism*) consiste à faire des choses différentes en même temps afin de générer du parallélisme. [...]
> - Le **parallélisme de flux** (*flow parallelism*) correspond à la technique du travail à la chaîne. Chaque donnée subit une séquence de traitements. C'est cette séquence qui est utilisée comme source de parallélisme et le parallélisme de flux réalise la séquence de traitements en mode pipeline en exploitant une régularité des données.

Ces différents styles, qu'on peut retrouver combinés dans un même programme ou algorithme, sont décrits plus en détails dans les sections suivantes.

B.1 Parallélisme de données

Le *parallélisme de données* correspond à l'application d'une même opération sur tous les éléments d'une collection de données *homogène*, c'est-à-dire dont les éléments sont tous de même type. Ces collections de données sont soit des listes (des séquences) — dans les langages fonctionnels par exemple — soit des tableaux — dans les langages de programmation impératifs plus *mainstream*. Dans les deux cas, on parle de structures de données *régulières*, par opposition aux structures de données dynamiques basées sur l'utilisation de pointeurs et donnant lieu à des arbres ou des graphes, donc des structures de formes non régulières. Un programme écrit dans le style parallélisme de données est donc composé d'une séquence d'applications d'opérations sur des collections. L'ensemble du travail effectué par le programme consiste donc en une série de phases de calcul, où chaque phase est une application d'une opération sur une collection, et où les différentes phases peuvent évidemment manipuler des collections différentes.

La forme la plus simple de parallélisme de données survient lorsque les calculs sur les différents éléments sont complètement *indépendants* les uns des autres, ce qui fait que l'ordre d'exécution n'a aucune importance et, donc, que tous les calculs peuvent se faire en parallèle.

Par exemple, soit l'opération consistant à multiplier chacun des éléments d'une collection par l'entier 2. En MPD, l'application d'une telle opération sur les éléments d'un tableau d'entiers `a` pour obtenir un tableau `b` pourrait alors s'écrire comme suit :

```
int a[n], b[n]
co [i = 1 to n]
  b[i] = 2 * a[i]
oc
```

Langages pour le parallélisme de données avec tableaux

Certains langages de programmation supportent *directement* le parallélisme de données sur des collections représentées par des tableaux. Par exemple, la même opération sur des tableaux (multiplication par 2 de chacun des éléments) pourrait s'écrire simplement comme suit en Fortran 90 :

```
integer A(n), B(n)
B = 2 * A
```

Ici, c'est le compilateur qui s'occupe de générer le code qui permettra l'exécution en parallèle des diverses multiplications et affectations. C'est aussi le compilateur qui s'occupera de *distribuer* les données entre les processeurs, en fonction des directives spécifiées par le programmeur, une tâche cruciale du processus de *programmation* parallèle lorsqu'on utilise le style parallélisme de données. Par exemple, en HPF (*High Performance Fortran*), la directive `DISTRIBUTE` permet d'indiquer au compilateur de quelle façon les éléments d'un tableau doivent être distribués entre les processeurs. Chaque processeur est alors responsable de calculer les éléments indiqués — on parle alors de la règle *owner computes*, i.e., "*the processor that "owns" a value is responsible for updating that value*" [Fos95]. La Figure 1 (adaptée de [Fos95, p. 254]) illustre les deux principaux types de décomposition et distribution d'un tableau à deux (2) dimensions 8×8 sur quatre (4) processeurs, à savoir distribution par bloc ou distribution cyclique (ou des combinaisons des deux modes).

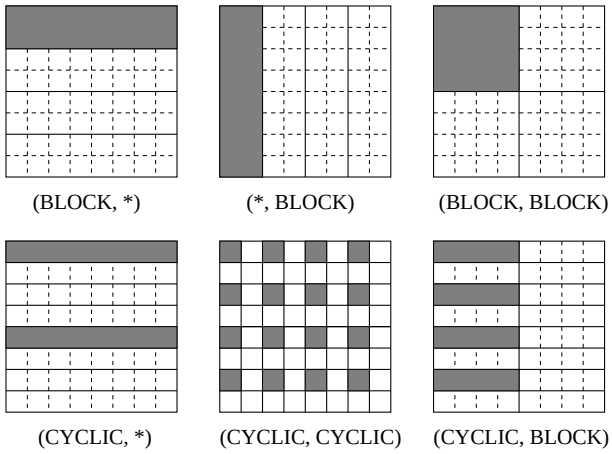


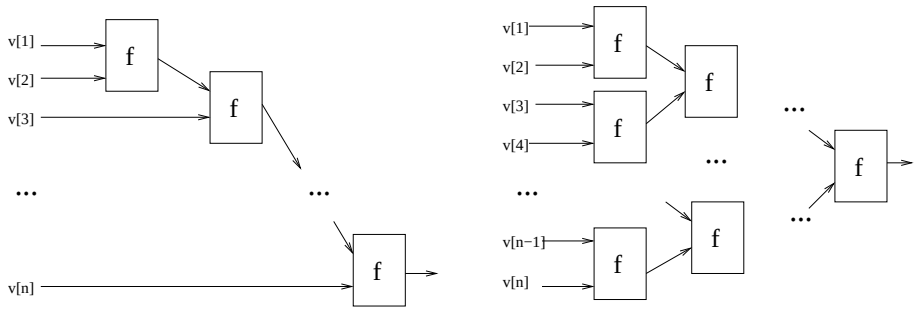
Figure 1: Différents types de distribution des données pour un tableau 8×8 sur quatre (4) processeurs en HPF. Les données pour le processeur no. 1 sont en gris.

En fait, la décomposition des données est le pendant pour le parallélisme de données de la décomposition en tâches pour le parallélisme de contrôle. Nous n'aborderons pas plus à fond la question de la distribution des données ici, cette question étant plutôt du ressort d'un cours de *programmation [parallèle]* plutôt qu'un cours de conception d'algorithmes [parallèles] — entre autres parce qu'elle dépend fortement de l'architecture de la machine parallèle utilisée.

Les principaux types d'opérations sur les collections

Les opérations sur les collections sont généralement de l'un des types suivants :

- α-*notation* : ce type consiste à appliquer une opération sur chacun des éléments d'une collection pour obtenir un résultat qui est lui-même une collection, de taille identique. Dans la terminologie des langages fonctionnels, on parle alors d'un opération



(a) Schéma béta

(b) Schéma béta−logarithmique

Figure 2: Schémas de réduction β et β-logarithmique

de type `map`. Par contre, certains auteurs français parlent plutôt d'un schéma de type *α-notation* [GUD96].

Plus spécifiquement, soit f une opération unaire (un argument) et $C = [c_1, \ldots, c_n]$ une collection de taille n. L'application de f à la collection C produira alors la collection R de taille n satisfaisant la propriété suivante :

$$R = [f(c_1), \ldots, f(c_n)]$$

On peut aussi généraliser l'idée aux opérations k-aire, c'est-à-dire avec k arguments. Soit les k collections $C^1, \ldots, C^k$ toutes de taille n. L'application de g, une fonction de k arguments, sur ces k collections produira alors la collection R de taille n satisfaisant la propriété suivante :

$$R = [g(c_1^1, \ldots, c_1^k), \ldots, g(c_n^1, \ldots, c_n^k)]$$

- *β-réduction* : ce type consiste à appliquer une opération *binaire* sur les divers éléments de la collection pour obtenir un résultat qui est un *scalaire*. Dans la terminologie des langages fonctionnels, on parle alors d'un opération de type `fold`.

  Par exemple, soit $\oplus$ une opération binaire et $C = [c_1, \ldots, c_n]$ une collection. L'application de $\oplus$ sur C via une β-réduction produira alors le résultat r satisfaisant la condition suivante :

  $$r = ((((c_1 \oplus c_2) \oplus c_3) \oplus \ldots) \oplus c_n)$$

  Dans le cas où $\oplus$ est associative — ce qui est habituellement le cas pour les opérations utilisées dans une telle β-réduction, par exemple, $+$, $*$, `MAX`, `MIN` — l'ordre d'application de l'opération $\oplus$ n'a donc pas d'importance. On peut donc parenthéser cette expression de façon différente sans changer le résultat[1]. Par exemple, pour $n = 8$, on peut alors parenthéser l'expression comme suit :

  $$(((c_1 \oplus c_2) \oplus (c_3 \oplus c_4)) \oplus ((c_5 \oplus c_6) \oplus (c_7 \oplus c_8)))$$

  Comme l'illustre la Figure 2 (adaptée de [GUD96, p. 107]), on peut alors obtenir une évaluation parallèle qui pourra s'effectuer en temps logarithmique plutôt qu'en temps séquentiel, d'où le terme *β-réduction logarithmique*.

[1] Et si ignore, dans le cas des nombres à virgule flottante, les questions d'arrondissement

- Calcul de préfixes : pour ce type, on applique une opération binaire associative sur les divers éléments de la collection pour obtenir un résultat qui sera une autre collection de même taille, où le $i$ème éléments de la collection résultante est obtenu par une série d'applications de l'opération binaire sur les i premiers éléments. Plus précisément, soit $\oplus$ une opération binaire et $C = [c_1, \ldots, c_n]$ une collection. Le calcul de préfixes sur C avec $\oplus$ produira alors le résultat R satisfaisant la condition suivante, donc tel que $R_i = c_1 \oplus c_2 \oplus \ldots \oplus c_i$:

  $$R = [c_1, c_1 \oplus c_2, c_1 \oplus c_2 \oplus c_3, \ldots, c_1 \oplus \ldots \oplus c_{n-1}, c_1 \oplus \ldots \oplus c_{n-1} \oplus c_n]$$

  On verra au chapitre sur les algorithmes PRAM que le calcul des préfixes est très intéressant car il peut s'exécuter efficacement (en temps logarithmique) et de nombreuses autres opérations peuvent être réalisées, de façon parallèle efficace, par l'utilisation d'un calcul de préfixes approprié — en fait, les deux premiers types d'opération sur des collections (α et β) peuvent s'exprimer comme des calculs de préfixe.

B.2 Parallélisme de contrôle

Le parallélisme de contrôle consiste à décomposer le travail à effectuer en différentes tâches, à identifier celles qui peuvent s'exécuter en parallèle, puis finalement à *ordonner*, à l'aide de structures de contrôle appropriées, l'exécution de ces diverses tâches — certains auteurs utilisent aussi le terme *task graph model* [GGKK03], ou encore *parallélisme de tâches*.

Parmi les exemples vus précédemment, ceux présentés à la section 2 sur le parallélisme récursif sont des programmes parallèles basés sur la parallélisme de contrôle, la décomposition en tâches se faisant par l'intermédiaire de la stratégie diviser-pour-régner (récursive).

B.3 Parallélisme de flux

Le parallélisme de flux est défini comme suit [GUD96, p. 132] :

> Le parallélisme de flux correspond au principe du *travail à la chaîne*. Une séquence d'opérations doit être appliquée en cascade à une série de données similaires. Les opérations à réaliser sont associées à des éléments de calculs chaînés de façon à ce que l'entrée d'une opération soit la sortie de l'opération précédente. Le parallélisme de flux n'est donc rien d'autre qu'un fonctionnement en mode *pipeline* des éléments du calcul.

Ce type de parallélisme correspond donc au paradigme *producteur–consommateur* vu précédemment.

Notons que la frontière entre parallélisme de flux et parallélisme de données peut parfois être floue. Ainsi, si l'ensemble des données du flux est entièrement disponible en mémoire, il est alors généralement aisé passer du mode parallélisme de flux au mode parallélisme de données, chacun des filtres du pipeline pouvant alors simplement être vu comme une application de style parallélisme de données (de type α ou β).

B.4 Un exemple : évaluation d'un polynôme en une série de points

On veut évaluer un polynôme $p(x) = a+bx+cx^2+dx^3$ sur un ensemble de n valeurs $v_1, \ldots, v_n$. En d'autres mots, on veut calculer : $a+bv_1+cv_1^2+dv_1^3, a+bv_2+cv_2^2+dv_2^3, \ldots, a+bv_n+cv_n^2+dv_n^3$. On suppose pour simplifier qu'il s'agit d'un polynôme à coefficients entiers et à valeur entière.

Algorithme séquentiel

L'algorithme suivant, exprimé en MPD, permet d'effectuer les évaluations désirées, où `v` contient les points d'évaluation et `r` retourne les résultats associés :

```
procedure evaluerPolynome( int a, int b, int c, int d, int v[n],
                           res int r[n] )
{
  for [i = 1 to n] {
    r[i] = a + b*v[i] + c*v[i]**2 + d*v[i]**3
  }
}
```

Parallélisme de données

La procédure suivante[2] permet, dans le style parallélisme de données, d'évaluer le polynôme $p(x) = a + bx + cx^2 + dx^3$ sur l'ensemble de n valeurs :

```
procedure evaluerPolynome( int a, int b, int c, int d, int v[n],
                           res int r[n] )
{
  co [i = 1 to n]
    r[i] = a + b*v[i] + c*v[i]**2 + d*v[i]**3
  oc
}
```

Il s'agit ici d'un schéma parallèle de type α. Du point de vue *algorithmique pure*, donc avec une machine parallèle idéale et sans contrainte, cette approche est intéressante car elle s'exécute en temps $\Theta(1)$ ... sauf qu'elle nécessite n processeurs.

Parallélisme de contrôle

Pour un index `i` donné, l'évaluation du polynôme $a + bx + cx^2 + dx^3$ en un point v_i demande d'évaluer diverses sous-expressions. L'évaluation de chacune de ces sous-expressions peut alors être vue comme une tâche indépendante, certaines pouvant s'effectuer en parallèle, alors que d'autres ne le peuvent pas (lorsqu'il y a des dépendances entre sous-expressions). On obtient alors le segment de code présenté dans l'Exemple de code 1, qui introduit du parallélisme de granularité *très fine* (tâche = évaluation d'une sous-expression simple, style machine à flux de données (*dataflow*)).

Ici, le temps d'exécution est $\Theta(n)$. Par contre, au plus trois (3) processeurs sont requis. Évidemment, dans un programme plus complexe, on identifierait des tâches de granularité moins fine ... et on utiliserait aussi le parallélisme de boucles (de données), puisque toutes les itérations sont indépendantes les unes des autres.

Parallélisme de flux

L'évaluation d'un polynôme à l'aide du style parallélisme par flux repose sur la méthode de Horner d'évaluation des polynômes. Le polynôme $p(x) = a + bx + cx^2 + dx^3$ peut être évalué comme suit :

$$p(x) = (((((d * x) + c) * x) + b) * x) + a$$

[2] Notez qu'il ne s'agit pas d'un segment de code MPD valide : en MPD, seules des invocations de fonctions ou de procédures peuvent apparaître dans le corps d'une instruction `co`.

```
procedure evaluerPolynome( int a, int b, int c, int d, int v[n],
                           res int r[n] )
{
  for [i = 1 to n] {
    co
      t1 = v[i]**2
      t2 = v[i]**3
    oc
    co
      t3 = b * v[i]
      t4 = c * t2
      t5 = d * t3
    oc
    co
      t6 =  a + t3
      t7 = t4 + t5
    oc
    r[i] = t6 + t7
  }
}
```

Exemple de code 1: Procédure pour évaluer un polynôme en une série de points avec parallélisme de contrôle (de granularité (*très*) fine)

Soit alors les fonctions suivantes :

$$\begin{aligned} f_0\ (x,\ y) &= (x, y * x + d) \\ f_1\ (x,\ y) &= (x, y * x + c) \\ f_2\ (x,\ y) &= (x, y * x + b) \\ f_3\ (x,\ y) &= (x, y * x + a) \end{aligned}$$

L'évaluation du polynôme $p(x)$ peut alors être exprimée comme suit :

$$\begin{aligned} p(x) &= r_2 \\ &\quad \texttt{where}\ (r_1, r_2) = f_3\ (f_2\ (f_1\ (f_0\ (x, 0)))) \end{aligned}$$

Or, l'expression $f_3\ (f_2\ (f_1\ (f_0\ (x, 0))))$ peut être simplement vue comme une composition séquentielle des fonctions, c'est-à-dire comme une sorte de pipeline où on transmet l'argument à f_0 qui transmet son résultat à f_1 et ainsi de suite. Dans un langage fonctionnel comme Miranda [Tur85, Tur86], une telle composition de fonction s'exprime de façon élégante à l'aide de l'opérateur de composition de fonctions «.» : `(f3 . f2 . f1 . f0)`.

Le programme Miranda 1 illustre l'utilisation d'une telle approche. Dans ce programme, on combine en fait parallélisme de données — avec les fonctions `map` et `foldr1` (schéma α appliqué à une collection représentée par une liste) et β-réduction — et parallélisme de flux — les différents points d'évaluation peuvent être traités en parallèle dans les diverses fonctions composées à l'aide de «.». On remarque aussi que les différentes fonctions $\texttt{f}_i$ sont toutes définies à l'aide d'une fonction auxiliaire `f`, qui prend en argument le `coef`ficient associé à la fonction $\texttt{f}_i$.

Le script 2 présente du code Unix — scripts *C-shell* et Perl — réalisant l'évaluation de polynômes avec parallélisme de flux, dans un style qui reflète de façon fidèle le programme Miranda.

Programme Miranda 1 Évaluation d'un polynôme à l'aide de parallélisme de données et parallélisme de flux

```
|| Alias/synonyme, pour illustrer le fait que la fonction "activee"
|| avec "fork" est (comme) executee en parallele, via un pipe.
|| Plus precisement, l'effet de "fork f" est de transformer une fonction qui
|| travaille sur un scalaire en une fonction qui traite un flux de donnees.

fork = map


|| La fonction suivante permet de connecter, "en (pseudo)parallele",
|| une serie de fonctions.
||
|| Le sens est inverse par rapport a l'ordre des fonctions, i.e.,
|| la fonction en tete de liste est la derniere dans le pipeline.
||
|| Exemple:
||   forkAndConnect [inc, fois2, neg] xs
||   = ((fork inc) . (fork fois2) . (fork neg)) xs
||   = (map inc (map fois2 (map neg xs)))

forkAndConnect xs = foldr1 (.) (map fork xs)



|| La fonction pour evaluer un polynome avec coefficients a, b, c et d
|| sur une collection de valeurs (une liste).

evaluerPolynome a b c d
  = (fork snd) . forkAndConnect [f a, f b, f c, f d] . (fork init)
    where
      init x       = (x, 0)
      f coeff (x, y)    = (x, y * x + coeff)

|| Note: snd (x, y) = y



|| Exemple d'utilisation:
main = evaluerPolynome 1 2 3 4 [0, 1, 2, 10]
```

Script 2 Scripts Unix (*C-shell scripts* et Perl) réalisant l'évaluation de polynômes avec parallélisme de flux, dans le style du programme Miranda.

```
% cat eval-poly.csh
#!/bin/csh -f

set a = $1
set b = $2
set c = $3
set d = $4

init | f $d | f $c | f $b | f $a | snd

---------------------------------------------------------------

% cat init
#!
sed 's/\(.*\)/\1 0/'


---------------------------------------------------------------

% cat f
#!/usr/bin/perl

$coeff = $ARGV[0];

while (<STDIN>) {
        /([0-9]+)[ ]+([0-9]+)/;
        $x = $1;
        $y = $2;
        $r = $y * $x + $coeff;
        print "$x $r\n";
}

---------------------------------------------------------------

% cat snd
#!
sed 's/\(.*\) \(.*\)/\2/'
```

C Encore une autre façon de classifier les paradigmes de programmation parallèle

Une autre façon intéressante de comprendre les diverses approches de programmation parallèle est celle présentée par Carriero et Gelernter dans leur article «*How to write parallel programs: A guide to the perplexed*» [CG89], où l'on distingue entre les trois approches suivantes :

1. Parallélisme de résultat
2. Parallélisme de spécialistes
3. Parallélisme d'agenda

C.1 Parallélisme de résultat

Lorsqu'on utilise du *parallélisme de résultat*, on identifie le parallélisme *en partant du produit final*, i.e., en partant du résultat désiré et en tentant de le décomposer en morceaux indépendants. On associe donc à chaque travailleur la tâche de produire un morceau du résultat final.

Exemple : un programme (granularité fine) qui calcule le produit de deux matrices où chaque processus calcule une des entrées de la matrice résultat — voir Programme MPD 1 (section 1, p. 4).

C.2 Parallélisme de spécialistes

Lorsqu'on utilise du *parallélisme de spécialistes*, on identifie diverses tâches *spécialisées* requises pour effectuer le travail global. On associe alors à chaque travailleur une de ces tâches spécifiques et on tente de les faire travailler en parallèle – les divers processus exécutent donc des tâches tout à fait distinctes.

Exemple : un programme de type producteur/consommateur (pipeline avec filtres), tel que le Script Unix 1 (section 3, p. 10).

C.3 Parallélisme d'agenda

Lorsqu'on utilisie du *parallélisme d'agenda*, on identifie les différentes activités qui doivent être exécutées, possiblement selon un certain ordre (total ou partiel). Les travailleurs, généralistes (sans aucune spécialisation) et uniformes, exécutent alors ces diverses tâches.

Généralement, au début du programme, un processus *maître* initialise le calcul en activant un certain nombre de travailleurs identiques ainsi qu'en créant un certain nombre de tâches. Les travailleurs exécutent alors les premières tâches de la liste des tâches. Lorsqu'un travailleur termine une tâche, il détermine alors la tâche suviante à faire dans la liste et l'exécute. Au cours de l'exécution d'une tâche, il est aussi possible qu'une ou plusieurs nouvelles tâches soient générées.

Exemple : un programme pour calculer la somme des éléments d'un tableau à l'aide d'une décomposition avec sac de tâches — voir Programme MPD 10 (section A.3, p. 17).

C.4 En résumé

Donc, en résumé, ces trois approches se distinguent de la façon suivante :

1. Parallélisme de résultat : Met l'accent sur la structure/forme du résultat désiré $\Rightarrow$ on obtient du parallélisme en calculant en parallèle les divers éléments du résultat.

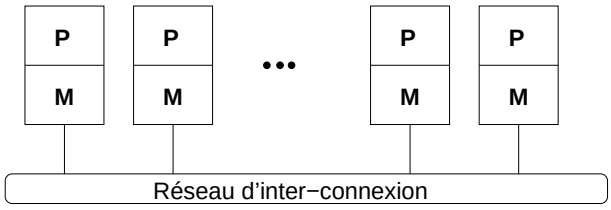
2. Parallélisme de spécialistes : Met l'accent sur la composition de l'équipe de travail $\Rightarrow$ on obtient du parallélisme en construisant un pipeline ou un réseau de processus communiquants, chaque processus étant spécialisé dans l'exécution d'une sorte de tâche spécifique.

3. Parallélisme d'agenda : Met l'accent sur la collection de tâches à être effectuées $\Rightarrow$ on obtient du parallélisme en associant plusieurs processus (généraliste) à la réalisation des diverses tâches en cours.

D Diviser-pour-régner avec filtres et canaux de communication

On a vu à la section 5, programmes MPD 5 et 6, que les canaux de communication pouvaient être utilisés pour résoudre un problème de façon parallèle à l'aide d'une approche de style *pairs interagissant*. Dans cette approche, un certain nombre de processus sont créés et chaque processus traite une partie des données et produit une partie des résultats, et ce de façon complètement indépendante des autres processus. Cette indépendance entre les processus tient au fait qu'aucune variable n'est partagée entre les processus. Les données initiales du problème sont plutôt gérées par un processus maître (coordonnateur) qui les transmet aux autres processus par l'intermédiaire de canaux de communication ; une fois les résultats produits par ces derniers processus, ils sont ensuite retournés au processus maître. Les canaux de communication utilisés pour transmettre les données et résultats sont alors les seuls éléments qui sont *partagés* entre les processus.

Une telle approche de programmation concurrente par échange de messages est souvent utilisée à cause des contraintes de l'architecture sur laquelle le programme est exécuté. Ainsi, supposons que l'on travaille sur une machine de type *multi-ordinateurs avec mémoire distribuée* (avec utilisation d'un réseau comme mécanisme d'inter-connexion) comme l'illustre la figure suivante :



Supposons aussi qu'aucune forme de mémoire partagée (matériel ou logiciel) ne soit supportée sur cette machine.[3] Dans ce cas, c'est alors l'approche de programmation par échange de messages qui correspond de façon la plus fidèle à l'architecture sous-jacente, donc qui devrait permettre d'exploiter efficacement la machine (meilleure correspondance entre le modèle ou paradigme de programmation et l'architecture de la machine).

Dans la section D.1 qui suit, nous allons examiner de quelle façon l'approche de programmation concurrente par échange de messages entre processus concurrents peut être intéressante pour certains problèmes qui, *a priori*, *n'ont rien à voir avec le parallélisme ou la concurrence*. Plus précisément, nous allons examiner comment le patron producteurs/consommateurs (introduit à la section 4), qui utilise des flux de données (transmis via des canaux de communication) et des «filtres» peut être utilisé pour résoudre des problèmes qui, traités dans un paradigme purement séquentiel, pourraient être relativement difficiles à résoudre. En d'autres mots, nous allons donc voir comment cette approche de programmation concurrente peut être vue et utilisée comme un nouveau *mécanisme de modularisation*, comme une nouvelle façon d'appliquer la stratégie *diviser-pour-régner* (sans récursion).

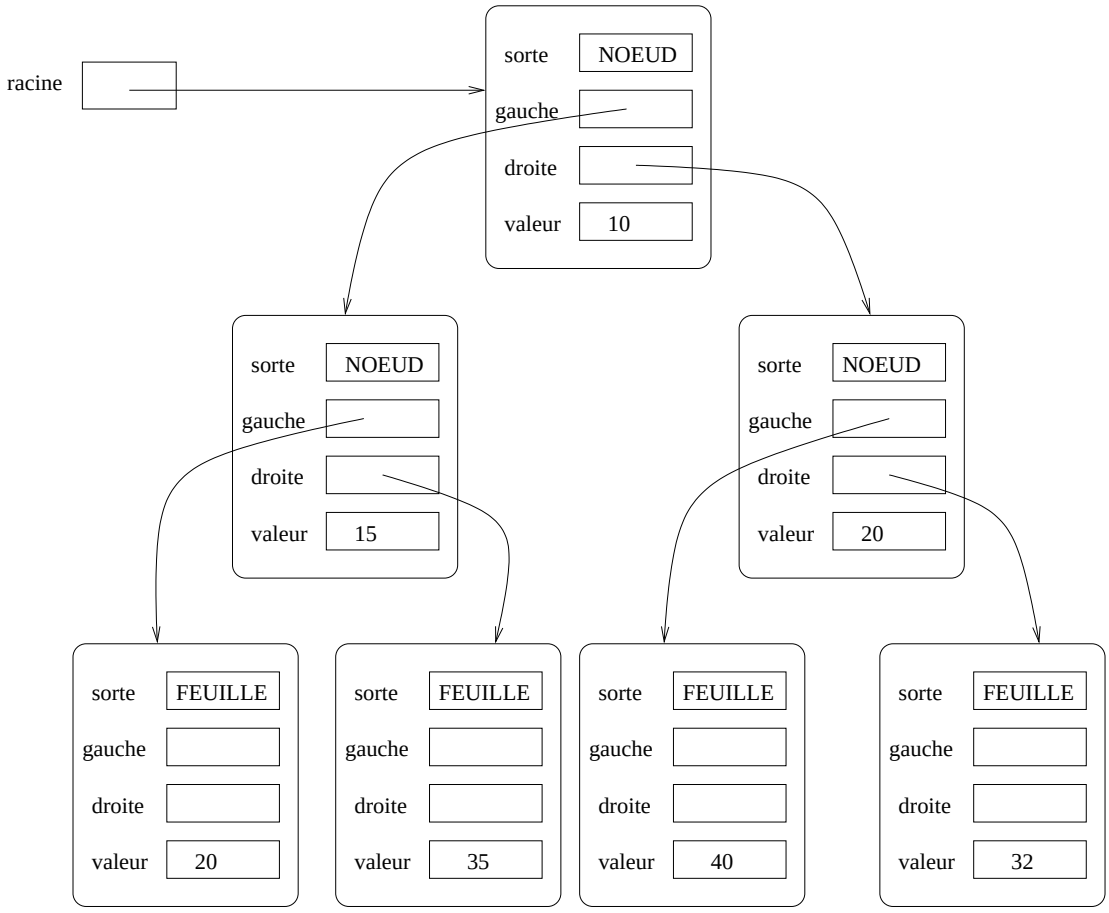
Programme MPD 11 Constantes, types et procédure pour la manipulation d'arbres binaires

```
# Constantes et types pour des arbres binaires.
const int FEUILLE = 0;
const int NOEUD   = 1;

type Arbre = ptr ArbreRec;
type ArbreRec = rec(
                  int sorte;
                  Arbre gauche, droit;
                  int valeur;
                );


# Procedure pour construire un arbre binaire (complet) aleatoire (pour tests).
procedure construireArbre( int n ) returns Arbre a
{
  if ( n == 0 ) {
    a = new(ArbreRec);
    a^.sorte = FEUILLE;
    a^.valeur = int(random(1, 100));
  } else {
    a = new(ArbreRec);
    a^.sorte = NOEUD;
    a^.valeur = int(random(1, 100));
    a^.gauche = construireArbre(n-1);
    a^.droit  = construireArbre(n-1);
  }
}

# Procedure pour imprimer un arbre.
procedure imprimer( Arbre a )
{
  if (a^.sorte == FEUILLE) {
    writes( "[", a^.valeur, "]" );
  } else {
    writes( "({", a^.valeur, "} " );
    imprimer( a^.gauche);
    imprimer( a^.droit );
    writes( ")" );
  }
}
```



Résultat produit par un appel à `imprimer` pour cet arbre :

```
({10} ({15} [20][35])({20} [40][32]))
```

Figure 3: Un exemple d'arbre produit par un appel à la procédure `construireArbre(2);`

D.1 Traitements sur les feuilles d'un arbre binaire

Le programme MPD 11 présente un certain nombre de constantes, types et procédures (écrites en MPD) pour la manipulation d'arbres binaires. Pour éviter d'introduire un type `union`, on remarque que tous les noeuds de ces arbres, tant les noeuds internes que les feuilles, contiennent les mêmes quatre champs. En pratique, il est entendu que seuls les noeuds internes (`sorte = NOEUD`) ont leur champ `gauche` et `droit` définis. Par contre, le champ `valeur` est bien défini pour toutes les sortes de noeud. La figure 3 présente l'allure d'un tel arbre.

D.1.1 Calculer les sommes partielles des feuilles paires

Le premier problème que nous allons examiner est le suivant : étant donné un arbre, on veut calculer toutes les sommes partielles des *feuilles* (pas les noeuds internes) dont le champ `valeur` est pair. Ainsi, pour l'arbre de la figure 3, on voudrait produire les suites de valeurs suivantes : 20, 60 ($= 20 + 40$), puis 92 ($= 20 + 40 + 32$) — la feuille dont le champ `valeur` est 35 est ignorée parce qu'impaire, alors que les valeurs des noeuds internes sont toujours ignorées.

Bien qu'il soit possible d'écrire une procédure récursive qui fasse le travail désiré, il est un peu plus difficile de définir une telle procédure qui soit *générique*, au sens de pouvoir facilement être adaptée pour faire des traitements divers sur les feuilles. Dans un langage séquentiel classique ou orienté objet, une façon possible d'obtenir une solution générique serait de définir un *itérateur*, c'est-à-dire, une procédure qui génère l'ensemble des noeuds (ou feuilles) de l'arbre.[4] On générerait alors les noeuds ou feuilles et on les traiterait dans une boucle en effectuant le traitement approprié. C'est un peu cette approche que nous allons utiliser, mais en générant, et en traitant, les divers éléments par l'intermédiaire de processus et de canaux de communication.

Le programme MPD 12 ($1^{ière}$ et $2^{ième}$ parties) présentent une première version d'un programme pour effectuer la tâche indiquée plus haut. Quatre (4) processus sont créés par le programme (de façon statique) et communiquent par l'intermédiaire de trois (3) canaux :

1. `GenererFeuilles` : initie le parcours de l'arbre en appelant la procédure récursive `genererFeuillesRec`, laquelle envoie chacune des feuilles rencontrées dans le canal `nombre`.

2. `FiltrerImpairs` : reçoit des nombres sur le canal `nombre` et ne transmet, sur le canal `nombrePair`, que ceux qui sont pairs (donc filtre, au sens de «*supprime*», ceux qui sont impairs).

3. `Cumuler` : reçoit sur le canal `nombrePair` une série de nombres et transmet sur le canal `sommePartielle` les diverses sommes partielles de tous les nombres reçus.

4. `EcrireNombres` : reçoit sur le canal `sommePartielle` une série de nombres et les écrit sur `stdout`.

Dans tous les cas, chacun des processus transmet et propage la valeur `EOS` pour indiquer la fin du flux, ce qui permet alors aux divers processus de se terminer correctement.

[3] Certaines machines parallèles récentes, bien qu'elles soient basées sur le modèle *physique* ci-dessus, introduisent malgré tout une forme de support — matériel, logiciel, ou une combinaison des deux — pour réaliser, au niveau logique, une mémoire partagée : on parle alors de *mémoire partagée distribuée* (en anglais : DSM = *Distributed Shared Memory*). Au niveau logique (vision du programmeur de haut niveau), la machine est alors perçue comme une machine à mémoire partagée, même si ce n'est pas le cas au niveau physique.

[4] En Java, on parlerait d'une `Enumeration`.

Programme MPD 12 Calcul de la somme des feuilles paires : version avec processus statiques ($1^{ère}$ partie)

```
# L'arbre traite par le programme.
Arbre racine;

#
# Les canaux de communication
#
const int EOS  = -1;    # Constante pour signaler la terminaison des canaux.

op nombre         ( int n );
op nombrePair     ( int n );
op sommePartielle ( int n );


# Procedure auxiliaire pour le parcours (recursif) d'un arbre
# avec envoi de la feuille dans le canal approprie.
procedure genererFeuillesRec( Arbre a )
{
  if (a^.sorte == FEUILLE) {
    send nombre(a^.valeur);
  } else {
    # a.sorte == NOEUD
    genererFeuillesRec( a^.gauche );
    genererFeuillesRec( a^.droit  );
  }
}


# Les differents processus (filtres)

# Le processus source (le generateur, le producteur)
process GenererFeuilles {
  genererFeuillesRec( racine ); # On genere les feuilles.
  send nombre( EOS );           # Toutes les valeurs des feuilles ont ete transmises.
}

# Filtre qui supprime les nombres impairs
process FiltrerImpairs {
  int n;
  receive nombre( n );
  while ( n != EOS ) {
    if (n % 2 == 0) {
      send nombrePair( n );
    }
    receive nombre( n );
  }
  send nombrePair( EOS );
}
```

Programme MPD 12 Calcul de la somme des feuilles paires : version avec processus statiques ($2^{ième}$ partie)

```
  # Filtre qui cumule les sommes partielles
  process Cumuler {
    int tot = 0, n;

    receive nombrePair( n );
    while ( n != EOS ) {
      tot += n;
      send sommePartielle( tot );
      receive nombrePair( n );
    }
    send sommePartielle( EOS );
  }

  # Filtre ("sink") pour imprimer les resultats
  process EcrireNombres {
    int n;
    receive sommePartielle( n )
    while ( n != EOS ) {
      write( n );
      receive sommePartielle( n );
    }
  }


  #
  # Programme principal: On construit l'arbre puis on laisse les processus s'executer.
  #
  begin {
    int n; getarg(1, n);  # Nombre de niveaux desires
    if (n < 0) {
      write( "*** Erreur: n < 0" );
      stop(1);
    }
    racine = construireArbre(n);
    write( "L'arbre construit est le suivant:" );
    imprimer(racine); write(); write();

    write( "Les sommes partielles des feuilles paires sont les suivantes : ");
  }
end
```

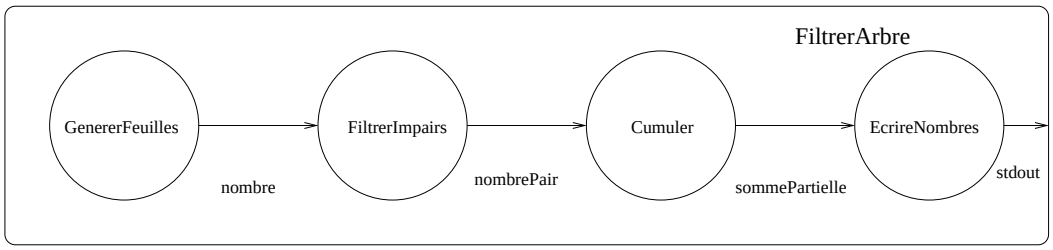


Figure 4: Diagramme de flux de données pour le programme de traitement des feuilles

Graphiquement, ces divers processus peuvent être représentés tel qu'illustré à la figure 4, c'est-à-dire sous forme d'un diagramme de flux de données (DFD, ou *Data Flow Diagrams*), une forme de diagramme «à la mode» dans les années «*pré-objets*» (fin 70, début 80, dans le contexte de l'analyse structurée).

D.1.2 Calculer les sommes partielles des feuilles paires puis trouver la feuille de valeur maximum

Supposons maintenant qu'à l'intérieur du même programme, il nous faut, après avoir calculé les sommes partielles des feuilles paires d'un arbre, trouver la feuille (paire ou impaire) de valeur maximum. Une solution avec création statique de processus ne nous permettrait évidemment pas de réutiliser les divers processus. Il serait plutôt nécessaire, et de toute façon plus général, de créer de façon *dynamique* les processus appropriés, et ce en indiquant à chacun de ces processus, de façon explicite, le canal de communication qui doit être utilisé pour communiquer avec ses «collaborateurs».

Le programme MPD 13 ($1^{ière}$, $2^{ième}$ et $3^{ième}$ parties) présente le code MPD pour une version où les processus requis sont créés explicitement à l'aide d'instructions `co`. Un groupe de trois (3) canaux de communication (`op can[3](int n)`) est défini (prog. MPD 13, $1^{ière}$ partie) puis est utilisé et réutilisé dans les deux traitements. Ces deux traitements se font alors de façon consécutive à l'intérieur du même programme (programme MPD 13, $3^{ième}$ partie). Ceci revient à créer, séquentiellement, deux pipelines combinant un groupe de processus :

- Premier traitement (somme des feuilles paires) : Quatre processus sont créés et utilisent les trois canaux (`can[1..3]`).
- Deuxième traitement (recherche du maximum) : Trois processus sont créés et utilisent deux des trois canaux (`can[1..2]`).

On remarque que les processus qui sont de véritables filtres, donc qui transforment un flux de données en un autre (par ex., `filtrerImpairs`, `cumuler`, `trouverMax`), reçoivent en argument, lors de leur activation, deux (2) canaux de communication : un premier pour lire (recevoir) le flux de données d'entrée, le deuxième pour transmettre le flux de données de sortie. Par contre, les processus qui sont strictement des producteurs (des *sources*, par ex., `genererFeuilles`) ou des consommateurs (des *puits*, par ex., `ecrireNombres`) ne reçoivent évidemment qu'un seul canal de communication.

Programme MPD 13 Calcul de la somme des feuilles paires et recherche du maximum : version avec processus dynamiques ($1^{ère}$ partie)

```
resource FiltrerArbre()
  #
  # Les canaux de communication
  #
  const int EOS  = -1;    # Constante pour signaler la terminaison des canaux.

  op can[3]( int n );


  # Procedure auxiliaire pour le parcours (recursif) d'un arbre
  # avec envoi de la feuille dans le canal approprie.
  procedure genererFeuillesRec( Arbre a, int cOut )
  {
    if (a^.sorte == FEUILLE) {
      send can[cOut](a^.valeur);
    } else {
      # a.sorte == NOEUD
      genererFeuillesRec( a^.gauche, cOut );
      genererFeuillesRec( a^.droit,  cOut );
    }
  }


  # Les procedures pour les divers processus (filtres)

  procedure genererFeuilles( Arbre a, int cOut )
  {
    genererFeuillesRec( a, cOut );
    send can[cOut]( EOS );
  }

  procedure  filtrerImpairs( int cIn, int cOut )
  {
    int n;
    receive can[cIn]( n );
    while ( n != EOS ) {
      if (n % 2 == 0) {
        send can[cOut]( n );
      }
      receive can[cIn]( n );
    }
    send can[cOut]( EOS );
 }
```

Programme MPD 13 Calcul de la somme des feuilles paires et recherche du maximum : version avec processus dynamiques ($2^{ième}$ partie)

```
procedure cumuler( int cIn, int cOut )
{
  int tot = 0, n;

  receive can[cIn]( n );
  while ( n != EOS ) {
    tot += n;
    send can[cOut]( tot );
    receive can[cIn]( n );
  }
  send can[cOut]( EOS );
}

procedure trouverMax( int cIn, int cOut )
{
  int max, n;

  receive can[cIn]( n );
  if ( n != EOS ) { max = n; }
  while ( n != EOS ) {
    if (n > max ) { max = n; }
    receive can[cIn]( n );
  }
  send can[cOut]( max );
  send can[cOut]( EOS );
}

# Filtre ("sink") pour imprimer les resultats
procedure ecrireNombres( int cIn )
{
  int n;
  receive can[cIn]( n )
  while ( n != EOS ) {
    write( n );
    receive can[cIn]( n );
  }
}
```

Programme MPD 13 Calcul de la somme des feuilles paires et recherche du maximum : version avec processus dynamiques ($3^{ième}$ et dernière partie)

```
  #
  # Programme principal: On construit l'arbre puis on active les processus.
  #
  int n; getarg(1, n);  # Nombre de niveaux desires
  if (n < 0) {
    write( "*** Erreur: n < 0" );
    stop(1);
  }
  Arbre racine = construireArbre(n);
  write( "L'arbre construit est le suivant:" );
  imprimer(racine); write(); write();

  # Premier traitement sur l'arbre: On produit les sommes partielles
  # de toutes les feuilles paires.
  write( "Les sommes partielles de feuilles paires sont les suivantes: ");
  co
     genererFeuilles( racine, 1 );
  // filtrerImpairs( 1, 2 );
  // cumuler( 2, 3 );
  // ecrireNombres( 3 );
  oc
  write();

  # Deuxieme traitement sur l'arbre: On cherche la valeur maximum
  # parmi toutes les feuilles.
  write( "La feuille de valeur maximum est la suivante:" );
  co
     genererFeuilles( racine, 1 );
  // trouverMax( 1, 2 );
  // ecrireNombres( 2 );
  oc
end
```

D.1.3 Calculer les sommes partielles des feuilles paires (bis) : version Unix

L'utilisation de processus concurrents qui communiquent par l'intermédiaire de canaux de communication est une stratégie utilisée fréquemment lorsqu'on travaille dans un environnement Unix. Un *pipe* Unix est une forme de canal de communication. Un programme Unix typique qui lit sur `stdin` et écrit sur `stdout` est un filtre (avec un canal d'entrée et un canal de sortie). De même, un programme qui génère des données sur `stdout`, sans lire sur `stdin`, peut être vu comme un producteur (source). L'exemple de code 2 plus bas présente une version Unix avec *pipes* d'un programme semblable à celui présenté précédemment (génération des feuilles, filtrage des valeurs impaires, puis cumul et impression des sommes partielles).

Le programme dans son ensemble, équivalent au programme MPD présenté plus haut, est défini par la composition parallèle des quatre (4) programmes suivants — qui pourraient être regroupés dans un *shell script* et ou être appelés directement via la ligne de commande :

```
generer-feuilles 5 | filtrer-impairs | cumuler | ecrire-nombres
```

Soulignons que ce pipeline (ligne du bas) regroupe un ensemble de programmes écrits *dans des langages différents* :

- `generer-feuilles` : programme MPD compilé avec la commande suivantes :

  ```
  mpd -o generer-feuilles generer-feuilles.mpd
  ```

- `filtrer-impairs` : *shell script* faisant appel à l'utilitaire `grep`.
- `cumuler` : script *perl*.
- `ecrire-nombres` : *shell script* faisant appel à la commande `more`.[5]

[5] Notons que le script `ecrire-nombres`, puisqu'il s'agit d'un simple un appel à `more`, aurait pu être omis, car par défaut l'impression est effectuée lors d'un envoi sur `stdout`. Ce processus n'a été introduit de façon explicite dans le pipeline que pour rendre plus claire la correspondance avec l'exemple initial.

```
Fichier generer-feuilles.mpd (extraits)
---------------------------------------
resource GenererFeuillesArbre()
  # A compiler avec "mpd -o generer-feuilles generer-feuilles.mpd"

  # Constantes, types et procedures pour des arbres binaires.
  ...

  # Procedure auxiliaire pour le parcours (recursif) d'un arbre
  # avec envoi de la feuille sur stdtout.
  procedure genererFeuillesRec( Arbre a )
  {
    if (a^.sorte == FEUILLE) {
      write(a^.valeur);           # On ecrit simplement sur stdout
    } else {
      genererFeuillesRec( a^.gauche );
      genererFeuillesRec( a^.droit  );
    }
  }

  #
  # Programme principal: On construit l'arbre puis on genere
  # les feuilles sur stdout.
  #
  int n; getarg(1, n);  # Nombre de niveaux desires
  if (n < 0) { write( "*** Erreur: n < 0" ); stop(1); }

  genererFeuillesRec( construireArbre(n) ); # On genere les feuilles.
end

-------------------------------------------------------------------------------

Fichier filtrer-impairs          Fichier cumuler            Fichier ecrire-nombres
-----------------------          ---------------            ----------------------
#!                               #!/usr/bin/perl            #!
grep -v "[0-9]*[13579]"          $tot = 0;                  more
                                 while(<>){
                                   $tot += $_;
                                   print $tot, "\n";
                                 }

-------------------------------------------------------------------------------

Appel (par ex., dans un shell script ou sur la ligne de commande)
=================================================================
generer-feuilles 5 | filtrer-impairs | cumuler | ecrire-nombres
```

Exemple de code 2: Calcul des sommes partielles des feuilles paires d'un arbre binaire à l'aide de processus et *pipes* Unix

E Parallélisme de style «sac de tâches»

E.1 Qu'est-ce que le style «sac de tâches»?

On sait que, règle générale, il est préférable dans un programme parallèle d'avoir un nombre limité de processus, plus ou moins en correspondance avec le nombre de processeurs disponibles. Dans certains cas, on est capable de décomposer dès le départ le problème en un nombre limité de sous-problèmes et d'associer à chacun un processus. Par exemple, lorsqu'on utilise le parallélisme de résultat, ce qui conduit naturellement à la création d'un grand nombre de petites tâches (comme pour la multiplication de polynômes dans le devoir #1), on peut simplement décider de regrouper ces diverses petites tâches pour en obtenir des plus grosses, réduisant ainsi le nombre de processus requis. Malheureusement, ce n'est pas pour tous les problèmes qu'une telle approche peut être utilisée, c'est-à-dire où l'on peut fixer le nombre de processus, et ce dès le départ du programme.

De plus, même lorsqu'il est possible de faire une telle affectation statique des tâches aux divers processus, il peut arriver que les diverses tâches ainsi générées ne requièrent pas la même quantité de travail. La charge de travail entre les processeurs peut donc être *déséquilibrée*, ce qui signifie dire que certains des processus/processeurs termineront leur exécution possiblement longtemps avant les autres, réduisant ainsi l'efficacité (parallèle) du programme.

La stratégie de *parallélisme d'agenda*, contrairement à celle de parallélisme de résultat, met plutôt l'accent sur la collection des tâches qui doivent être effectuées. Dans cette strategie, on identifie les différentes activités qui doivent être exécutées, possiblement dans un certain ordre (total, donc séquentiel, ou partiel, donc avec concurrence possibles). On obtient alors du parallélisme en créant un certain nombre de processus concurrent, nombre généralement fixé au départ, lesquels processus vont exécuter les diverses tâches, possiblement en nombre variable, au fur et à mesure où ces diverses tâches deviendront disponibles pour exécution. On va donc décomposer le problème initial en un plus ou moins grand nombre de tâches, qu'on représentera à l'aide de ce qu'on appelle *un sac de tâches* (*a bag of tasks*). Ce sac, qui représente les tâches qui restent à être effectuées, peut être une structure de données complexe, mais peut aussi simplement être représenté par une (ou quelques) variable(s) qui permet(tent) d'identifier la prochaine tâche à effectuer parmi la collection de tâches.

Plus précisément, lorsqu'on utilise cette stratégie de création et gestion du parallélisme, on crée dès le départ un certain nombre de processus, nombre qui peut être déterminé par la structure de la machine (c'est-à-dire, par le nombre de processeurs). On crée aussi, de façon indépendante, un certain nombre de tâches (plus précisément, on crée/génère des *descripteurs* de tâches), tâches qu'on insère dans une structure de données appropriée, i.e., le *sac de tâches* (en anglais, on dit aussi *task pool*). Lorsqu'un processus (processeur) est libre, il choisit alors une des tâches disponibles dans le sac et l'exécute. Dans certains cas, l'exécution d'une tâche génère de nouvelles tâches, lesquelles sont alors ajoutées au sac. Le programme dans son ensemble se termine lorsqu'il ne reste plus *aucune* tâche à exécuter, c'est-à-dire lorsque le sac de tâches est vide et que tous les processus sont inactifs (donc aucun processus ne générera de nouvelle tâches). Signalons que, pour certains problèmes, la principale difficulté de l'utilisation d'un sac de tâches est justement de réussir à détecter le moment où le sac est vide et où toutes les tâches ont terminé leur exécution.

Dans une approche avec sac de tâches, chaque processus a généralement l'allure suivante — on verra plus bas une façon légèrement différente (plus élégante et plus simple) d'obtenir un effet équivalent dans le cas où de nouvelles tâches ne sont pas ajoutées en cours d'exécution, donc dans le cas où le test de déterminer si le sac est vide peut se faire de façon immédiate et directe :

```
process executerTaches[i = 1 to nbProcs] {
  termine ← false
  while (! termine) {
    obtenir une tâche du sac
    if (il restait une tâche à exécuter) {
      exécuter la tâche obtenue, possiblement en générant
         de nouvelles tâches et en les ajoutant dans le sac
    } else {
      termine ← true
    }
  }
}
```

Les principaux avantages de l'approche avec sac de tâches sont les suivants :

- Généralement facile à utiliser pour générer du parallélisme (d'agenda) ;
- On peut facilement faire varier le nombre de processus en fonction du nombre de processeurs, ce qui conduit à des programmes qui peuvent fonctionner sur des machines avec un nombre variable de processeurs (*scalability*).
- Permet généralement une meilleure distribution de la charge (*load balancing*).

E.2 Somme de deux vecteurs

Programme MPD 14 Somme des éléments d'un tableau avec un sac de tâches

```
# Nombre d'elements des tableaux.
int n; getarg(1, n);

# Taille de chacune des taches.
int tailleTache; getarg(2, tailleTache);

# Nombre de processus a creer.
int nbProcs; getarg(3, nbProcs);

# Assertion:
#  n >= 1 & 1 <= tailleTache <= n

#
# Structures de donnees et operations pour gestion du sac de taches
#
int suivant, borneSup;

# Initialisaton du sac de taches.
procedure initSacDeTaches( int bi, int bs )
{ suivant = bi; borneSup = bs;  }

# Retrait d'une tache du sac
procedure obtenirTache( res int i, res int j ) returns bool disponible
{
  i = FA( suivant, tailleTache );
  j = min( i + tailleTache - 1, borneSup );
  disponible = (i <= borneSup);
}

# Selection d'une tache et execution
procedure effectuer_somme( ref int a[*], ref int b[*], ref int c[*] )
{
  int i, j;
  while ( obtenirTache(i, j) ) {
    for [k = i to j] {
      c[k] = a[k] + b[k];
    }
  }
}

#
# La procedure de sommation avec un sac de taches.
#
procedure somme( int a[*], int b[*], ref int c[*], int n )
{
  initSacDeTaches( 1, n );

  # Activation des taches
  co [i = 1 to nbProcs]
    effectuer_somme( a, b, c )
  oc
}
```

Le programme MPD 14 présente une mise en oeuvre parallèle utilisant un sac de tâches pour le problème du calcul de la somme de deux vecteurs. Dans ce programme, plutôt que de fixer le nombre de processus et d'associer à chacun une «large» tranche des tableaux, on spécifie plutôt la taille désirée pour chacune des tâches, taille qui sera telle qu'on aura un plus grand nombre de tâches que de processus — chaque tâche correspondra donc à

une (relativement) «*petite*» tranche du tableau. Ici, cette taille est spécifiée par la variable `tailleTache` (spécifiée au moment de l'appel du programme), qui indique la taille de la tranche du résultat devant être traité par chacune des instances de la boucle `for` dans la procédure `effectuer_somme`. Notons qu'il n'est pas nécessaire de supposer que le nombre total d'éléments du tableau est un multiple de la taille des tâches — les manipulations et comparaisons avec les bornes se font correctement dans la fonction `obtenirTache`.

Programme MPD 15 Interface de l'opération `FA`

```
global FetchAndAdd

  op FA( ref int x, int incr ) returns int r
  # EFFET
  #   < r = x; x += incr; >

body FetchAndAdd
  ... partie omise ...
end
```

Dans la procédure `somme`, on crée donc `nbProcs` processus, nombre spécifié au moment de l'appel du programme. Puisque ces `nbProcs` processus doivent interagir pour accéder aux variables représentant le sac de tâches, il nous faut donc un mécanisme de synchronisation. Ici, la manipulation *atomique* de la variable représentant la prochaine tâche du sac de tâches (variable `suivant`) est effectuée à l'aide d'une opération `FA` — *fetch-and-add* — (définie dans le module `FetchAndAdd`), dont l'interface est présentée dans le Programme MPD 15. Cette opération retourne la valeur avant l'appel de la variable spécifiée en argument (`x`), puis l'augmente de l'incrément indiqué (`incr`), le tout de façon atomique.

Références

[And00] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, Reading, MA, 2000. [QA76.58A57 2000].

[CG89] N. Carriero and D. Gelernter. How to write parallel programs—a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, Sept. 1989. [Tiré de [ST95]].

[Fos95] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995. `http://www-unix.mcs.anl.gov/dbpp`.

[GGKK03] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing (Second Edition).* Addison-Wesley, 2003.

[GUD96] M. Gengler, S. Ubéda, and F. Desprez. *Initiation au parallélisme—Concepts, architectures et algorithmes.* Masson, 1996. [QA76.58G45].

[ST95] D.B. Skillicorn and D. Talia. *Programming Languages for Parallel Processing.* IEEE Computer Society Press, 1995.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Prog. Lang. and Comp. Arch.*, pages 1–16. Springer-Verlag, LNCS-201, 1985.

[Tur86] D.A. Turner. An overview of miranda. *SIGPLAN Notices*, 21(12):158–166, Dec. 1986.