

Le langage MPD

MPD est un langage basé sur les langages `C` et `Pascal`, mais avec des ajouts pour la programmation concurrente (parallèle, distribuée, par variables partagées, par échange de messages). Plus précisément, MPD est un descendant direct, avec une syntaxe *de style C*, du langage `SR`¹, lequel langage était basé principalement sur `Pascal`.

La syntaxe complète du langage MPD est présentée (de façon informelle) à l'URL suivant :

<http://www.cs.arizona.edu/mpd/synopsis.html>

Les différents opérateurs et fonctions prédéfinis du langage sont présentés en détail à l'URL suivant :

<http://www.cs.arizona.edu/mpd/SRbook.appendixC.ps>

1 Commentaires et séparateurs pour les déclarations et instructions

Deux sortes de commentaires :

1. Débutant par “#” et se terminant à la fin de la ligne.
2. À la C, donc indiqués par “/* ... */”.

Le caractère “;” est utilisé pour terminer une déclaration ou instruction, mais n'est nécessaire que si *plusieurs* items apparaissent sur le même ligne (un saut de ligne suffit pour terminer/séparer).

2 Déclarations

```
bool estTermine;
int a[n];                # int a[1:n];
int c[2:2*m+2];         # Bornes explicites
double c[n, n] = ([n] ([n] 1.0)); # Initialisation
```

- Les instructions et les déclarations peuvent être mélangées (comme en Java).
- Tout identificateur doit être déclaré *avant* de pouvoir être utilisé.
- On peut spécifier les bornes inférieure et supérieure du tableau, ou simplement spécifier la taille (contrairement à C, la borne inférieure est alors de 1).

¹<http://www.cs.arizona.edu/sr/>

3 Instructions séquentielles de contrôle

– Instruction SI et boucle TANTQUE :

```
if (condition) {
    instruction1;
    ...
    instructionn;
} else {
    instruction1
    ...
    instructionm;
}

while (condition) {
    instruction1;
    instruction2;
    ...
    instructionn;
}
```

- Contrairement à C ou Java, les accolades indiquant un bloc d'instructions *ne sont pas optionnelles*.

– Boucle POUR :

```
for [quantificateur1, ..., quantificateurm] {
    instruction1;
    ...
    instructionn;
}
```

– Quantificateurs possibles :

```
for [i = 0 to n-1] ...

for [i = 0 to n-1, j = 0 to n-1] ...

for [i = 0 to n-1]
  for [j = 0 to n-1] ...

for [i = 0 to n by 2] ...

for [i = 0 to n-1 st i != x] ...      # st = such that
```

Note : les variables d'itération (introduites dans les quantificateurs) n'ont pas besoin d'être déclarées au préalable. En fait, la portée d'une telle variable est limitée à la boucle.

4 Instructions de contrôle co

– Une instruction `co` permet l'exécution concurrente d'un groupe de processus créés de façon *dynamique*. Les formes de base les plus fréquemment utilisées sont les suivantes :

- Création de n processus, où chaque processus exécute *une* des instructions indiquées (exécution parallèle d'un groupe d'instructions), chacune des instructions correspondant à une invocation (appel) de procédure ou de processus :

```
co
    invocation1;
// ...
// invocationn;
oc
```

Note : Une *invocation* (forme de base) est un appel simple de procédure “ $p(a_1, \dots, a_k)$ ” ou encore un appel d’une fonction “ $x = p(a_1, \dots, a_k)$ ”. En d’autres mots, un `co` ne peut pas contenir de suite d’instructions complexes.

- Création dynamique d’une ensemble de processus spécifiés par un (ou des) quantificateur(s), un processus étant créé pour chaque valeur de `i` et où chaque processus utilise une valeur *distincte* de `i` :

```
co [i = 1 to n]
    invocation;
oc
```

– La forme générale d’une instruction `co` permet l’introduction de plusieurs branches parallèles, chacune débutée par un quantificateur, et où chaque invocation est suivie de l’exécution d’un bloc d’instructions :

```
co
    [i = a1 to b1] invocation -> blocInstructions
// [j = a2 to b2] invocation -> blocInstructions
// ...
oc
```

Chacun des blocs d’instruction (appelé *bloc de post-traitement*, en anglais *post-processing code*) s’exécute implicitement de façon *atomique*, donc un à la fois, et ce sans qu’aucun mécanisme particulier de synchronisation n’ait besoin d’être indiqué ou ajouté.

– Note importante concernant la terminaison d’une instruction `co` :

- Une instruction `co` (dans son ensemble) se termine uniquement lorsque *chacune* des instructions qui la composent est terminée (les processus enfants qui ont été créés ainsi que le bloc de post-traitement associé ont terminé leur exécution).

5 Processus et procédures

– Processus : *déclaré* un peu comme une procédure (pas d’arguments formels), mais amorcé *implicitement* et s’exécutant *en arrière-plan* (donc pas d’attente implicite, comme dans un `co`, qu’il se termine) :

```
process foo {
    instruction1;
    ...
    instructionm;
}
process bar [i = 1 to n] {
    instruction1;
    ...
    instructionm;
}
```

- La forme de gauche crée un unique processus.
- La forme de droite crée `n` processus distincts, avec chacun son propre `i`.

– Procédure et fonction :

```
proc foo( a ) {
    instruction1;
    ...
    instructionm;
}

procedure fooProcedure( var int a ) {
    instruction1;
    ...
    instructionm;
}

procedure bar( int v ) returns int resultat {
    # La variable resultat doit être définie (affectation) dans le corps de la fonction.
    instruction1;
    ...
    instructionm;
}
```

Différences entre `proc` et `procedure` :

- Une `procedure` est généralement utilisée pour définir une procédure locale et privée. Dans ce cas, on donne explicitement la *signature complète* de la procédure (types des arguments et du résultat).

Une *fonction* est simplement une procédure qui retourne un résultat, donc une procédure dont la signature se termine par “`returns unType uneVar`”. Une affectation à la variable *uneVar* doit se faire dans le corps de la fonction.

L’instruction `return` peut être utilisée pour terminer l’exécution d’une procédure ou fonction. Toutefois, cette instruction, contrairement à la version C, ne prend aucun argument.

- Une `proc` correspond plutôt à la mise en oeuvre d’une opération publique (`op`) exportée par la ressource (voir prochaine section). C’est alors l’`op` qui donne la signature, la déclaration du `proc` n’indiquant que le nom des arguments (donc pas leur type).

– Modes de passage des arguments :

- `val` : passage par valeur.
- `var` : passage par valeur/résultat (*copy-in/copy-out*).
- `res` : passage par résultat (*copy-out*).
- `ref` : passage par référence.

Note : le mode `val` est le mode par défaut (lorsqu’aucune annotation n’est indiquée).

6 Ressources et services exportés

- Chaque programme doit nécessairement définir au moins une ressource (`resource`).
- Une ressource peut exporter explicitement une ou plusieurs opérations, indiquées par le mot réservé `op`. La mise en oeuvre d’une telle opération se fait avec une procédure `proc` ou encore avec une instruction `in`.² Dans ce dernier cas, selon que l’appel est synchrone (`call`) ou non (`send`), il s’agit alors d’un rendez-vous (synchrone) ou d’un envoi

²Le mot réservé `in` est utilisé pour `input`. Notons qu’une instruction `receive` est considérée comme une forme simplifiée d’instruction `in` dans laquelle l’instruction `in` ne fait simplement que recevoir un message.

de message (asynchrone). La figure ?? (adaptée de AndrewsOls93, “The SR Programming Language”) illustre les différents types d’interaction entre processus disponibles en MPD. Dans le cadre du présent cours, nous n’utiliserons pas d’instructions `in` ; nous n’utiliserons donc que les mécanismes a) et b) de la Figure ??.

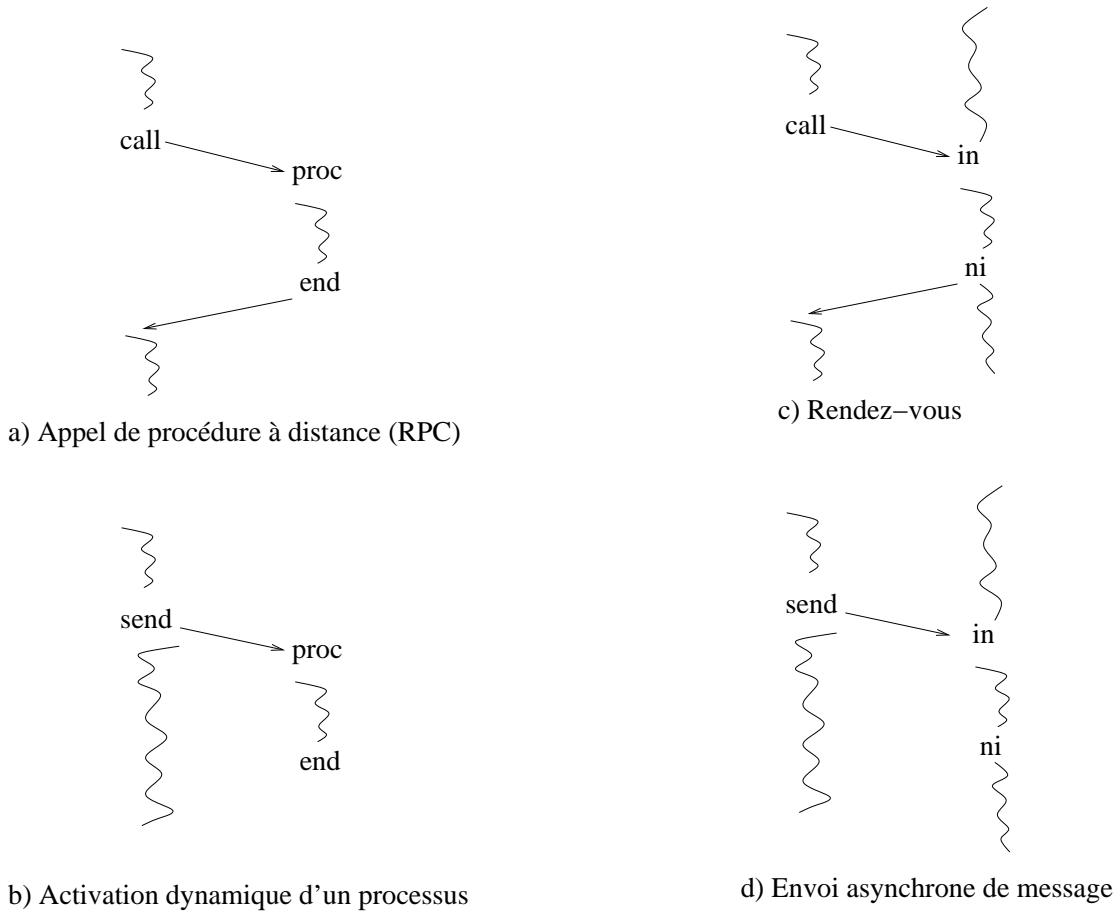


Figure 1: Les différents mécanismes d’activation et de synchronisation entre processus

- Exemple :

```
resource quick()

int MAXCARS = 120, MAXLIGNES = 10000;

op trier(var string[*] a[1:~]) # Declaration anticipée (forward) de trier()

# Corps principal de la ressource
...

proc trier(a) {
  # Definition effective de l'operation.
  ...
}
end quick
```

7 Opérations d'invocation et de synchronisation

7.1 Invocations

Deux mécanismes d'activation (voir Figure ??) :

- **call operation**(*arg*₁, ..., *arg*_{*k*}) : Activation *synchrone* d'une op. Si cette op est définie par un **proc** (ce sera le cas pour la plupart des exemples que nous verrons, à l'exception de ceux utilisant des canaux de communication), l'effet est donc semblable à un appel de procédure usuel. L'instruction **call** dans l'appelant ne se termine que lorsque l'appelé a terminé son exécution.
- **fork operation**(*arg*₁, ..., *arg*_{*k*}) : Activation *asynchrone*, donc aucune attente par l'appelant pour que l'appelé se termine. Le parent (appelant) et l'enfant (appelé) s'exécutent de façon complètement indépendante.

Note : Pour une op définie par un **proc**, un **fork** est en fait une abréviation d'un **send**.

7.2 Synchronisation avec sémaphore

Sémaphore = sorte *spéciale* de variable contenant un nombre entier *non-négatif* et manipulée uniquement par deux opérations atomiques : P et V.

- **sem s = val** : Déclaration d'un sémaphore **s** avec une valeur initiale de **val**.
sem s[n] = ([n] val) : Déclaration d'une collection indexée (tableau) de **n** sémaphores (constante ou variable), tous initialisés avec la valeur initiale **val**.
- **P(s) = Passeren** \approx passer, prendre \Rightarrow Décrémente la variable, à moins qu'elle ne soit déjà 0.
Si déjà 0, alors le processus appelant bloque (suspend son exécution) jusqu'à ce qu'un événement approprié, signalé par **V(s)**, survienne.
- **V(s) = Vrijgeven** \approx relâcher \Rightarrow Incrémente la variable.
Si un ou plusieurs processus étaient bloqués sur le sémaphore, alors l'un d'entre eux est réactivé (ce qui a immédiatement pour effet de le remettre à 0, puisque l'opération **P** qui était bloquée peut alors se compléter).

7.3 Invocations et synchronisation par échange de messages

- **send operation**(*arg*₁, ..., *arg*_{*k*}) : Appel asynchrone = envoi d'un message (formé de **k** champs) par l'intermédiaire d'un canal de communication. Ce canal est défini par un **op**, mais sans **proc** correspondant.
Ici, on dit que l'appel, et l'envoi, sont effectués de façon *asynchrone*. Ceci signifie que l'opération **send**, du point de vue de l'expéditeur, a simplement pour effet d'ajouter le message approprié dans le canal de communication, et donc que l'opération se complète de façon immédiate, sans attente. En d'autres mots, l'expéditeur n'a pas à attendre que le message soit reçu par un autre processus.
- **receive operation**(*var*₁, ..., *var*_{*k*}) : Réception d'un message par l'intermédiaire d'un canal de communication. Si aucun message n'est disponible, l'opération *bloque*, reste en attente jusqu'à ce qu'un message soit disponible sur le canal. Si un message est disponible, il est extrait du canal et ses divers champs sont extraits et affectés aux variables indiquées.

Contrairement à un **send**, qui est une opération non bloquante, un **receive** est donc une opération *bloquante*.

Un autre point important à souligner concernant les canaux de communication est que la séquence des messages transmis par l'intermédiaire d'un canal possède la propriété suivante :

Propriété FIFO : si un processus donné P envoie un premier message m_1 puis un autre message m_2 sur le même canal, alors le message m_1 sera reçu *avant* m_2 si la réception se fait par un même processus Q .

Par contre, cette propriété n'est pas valable si les deux messages m_1 et m_2 sont envoyés par deux processus distincts. Dans ce cas, l'ordre de réception sera *non déterministe*.

Pour des exemples, voir les programmes MPD 5 et 6 du chapitre sur les "Paradigmes de base de la programmation concurrente" ainsi que l'appendice B de ce même chapitre "Paradigmes de base de la programmation concurrente (suite) — B. Filtres et canaux de communication".

8 Opération pour mesurer le temps d'exécution

- `age()` : Retourne un entier qui indique, en millisecondes, le temps écoulé depuis que la ressource locale a été créée.

9 Opérations pour accéder aux arguments du programme et aux E/S

- `getarg(i, a)` : retourne dans `a` le i ème argument spécifié au moment de l'appel du programme. Si $i = 0$, alors le nom de l'exécutable est retourné.
- `numargs()` : retourne le nombre d'arguments indiqués à l'appel du programme (sans compter le nom du programme lui-même).
- `read(x, ...)` et `read(f, x, ...)` : lit des valeurs de `stdin` ou du fichier `f` et les met dans les arguments indiqués. Retourne le nombre de valeurs lues, ou EOF si la fin de fichier est rencontrée avant qu'une valeur ne soit lue.
- `write(x, ...)` et `write(f, x, ...)` : formate et écrit sur `stdout` ou sur le fichier `f`. La valeur écrite pour la variable `x` est `string(x)` (une chaîne formatée en fonction du type de la variable). Un blanc est inséré entre chaque item et une fin de ligne est ajoutée.
- `writes` : comme `write`, mais sans blanc entre les items et sans fin de ligne.
- `scanf` et `printf` : comme en C (avec les mêmes spécifications de format).

10 Compilation et exécution de programmes MPD

– Étapes pour compiler et exécuter un programme MPD :

1. Créer un fichier `foo.mpd` (l'extension `mpd` doit être utilisée).
2. Compiler avec la commande "`mpd foo.mpd`".
3. Exécuter le programme en spécifiant ses arguments.

Par défaut, le nom de l'exécutable est `a.out`. On peut spécifier le nom du fichier à utiliser avec `mpd -o nomExecutable foo.mpd`

Note : À l'étape 2, un répertoire `MPDinter` est créé pour conserver divers fichiers générés par le compilateur.

– Sur une machine à plusieurs processeurs (par exemple, `arabica`, qui en possède six), il est possible de spécifier le nombre de processeurs pouvant être utilisés pour exécuter le programme en définissant la variable d'environnement `MPD_PARALLEL`. Par défaut, la valeur de cette variable est 1. Pour la modifier, il suffit d'utiliser l'une des commandes suivantes :

- Avec `csh/tcsh` :

```
setenv MPD_PARALLEL 4
```

- Avec `bash` :

```
MPD_PARALLEL=4
export MPD_PARALLEL
```

– Pour plus de détails sur la compilation et l'exécution de programmes MPD, consultez l'URL suivant :

<http://www.cs.arizona.edu/mpd/howto.html>

– Note : Il est très facile, si vous utilisez une machine Linux, d'installer le compilateur MPD sur votre machine. Voir le site *web* de MPD.

11 Opérations externes et accès à l'environnement

Il est possible d'utiliser des fonctions *externes*, fonctions pouvant être définies dans d'autres langages, y compris des fonctions standards de la librairie C. Par exemple, l'extrait de code suivant permet de déterminer le nombre effectif de processeurs spécifié par la variable d'environnement `MPD_PARALLEL`, et ce en utilisant la fonction pré-définie `getenv` (provenant de `<stdlib.h>`) :

```
external getenv(string[*]) returns string[2];

int nbProcs = 0;
if (getenv("MPD_PARALLEL") != "") {
    nbProcs = int(getenv("MPD_PARALLEL"));
}
```

A Les différentes façons de créer des processus

– Il existe différentes façons, en MPD, de créer des processus :

- De façon *statique* ou *semi-dynamique*, à l’aide de déclarations `process` : de tels processus sont créés au moment où les déclarations `process` sont rencontrées et *élaborées*. Les processus ainsi créés s’exécutent *en arrière-plan* (en *background*).

S’il s’agit d’un unique processus (pas de quantificateurs) ou si les bornes sont spécifiées à l’aide de constantes, on peut alors parler de processus déclarés de façon *statique*. Autrement, si les bornes sont spécifiées par des variables (définies dans les instructions qui précèdent, par exemple, voir le programme MPD 2 (chapitre sur les “Paradigmes de base de la programmation concurrente”) où `n` est défini par l’instruction `getarg`, donc au moment de l’appel du programme), on parle alors de processus créés de façon semi-dynamique.

- De façon *dynamique*, avec les diverses formes d’instruction `co` : de tels processus, anonymes, sont créés au moment où l’instruction `co` s’exécute. L’exécution de l’instruction `co` ne se termine que lorsque les processus enfants ainsi créés ont eux-mêmes terminé leur exécution.

Un processus peut aussi être créé de façon dynamique à l’aide de l’instruction `fork`. En termes de création des processus (voir les programmes MPD de l’exemple de code ??), une instruction `co` peut parfois être remplacée par une instruction `for` dont les invocations se font avec `fork` (plutôt qu’avec `call`). Toutefois, l’effets en termes de terminaison des processus n’est évidemment pas le même, puisque le `co` ne se termine que lorsque les processus enfants se terminent, ce qui n’est pas le cas avec une série de `fork` (processus en arrière-plan) à l’intérieur d’une boucle `for`.

Des exemples illustrant la création statique vs. dynamique de processus sont présentés plus bas (sections ?? et ??). La section ??, quant à elle, présente une analogie entre les différentes formes de déclaration de variables tableaux et de déclaration de processus.

A.1 Analogie entre les diverses formes de déclaration de tableaux et de déclaration de processus

Les diverses formes de déclaration et création de processus (statique, semi-dynamique et dynamique) sont semblables à ce qu’on retrouve dans de nombreux langages de programmation au niveau des déclarations de variables, tel que cela est illustré dans le programme MPD ??. On remarque qu’il est possible en MPD, comme cela peut aussi se faire en C, d’allouer un tableau en spécifiant sa taille de façon complètement dynamique. Un type pour de tels tableaux dynamiques de taille variable peut être déclaré à l’aide d’une déclaration de la forme suivante :

```
“type TableauDyn = [*] int” ;
```

Dans ce cas, c’est alors au moment de l’allocation du tableau (sur le tas) avec `new` que l’on peut spécifier la taille de ce tableau (`new([n] int)`).

Programme MPD 1 Divers modes d'allocation des tableaux en MPD

```
resource AllocationsTableaux()
# Type pour tableau d'entiers de taille variable
type TableauDyn = [*] int;

# Procédures auxiliaires
procedure init( ref int a[*], int n )
{ for [i = 1 to n] { a[i] = int(random(1, n)); } }

procedure imprimer( int a[*], int n )
{ for [i = 1 to n] { write( a[i] ); } }

# Procédures effectuant l'allocation de tableaux.
procedure allocSemiDynamique( int n )
{
  # Le tableau a est alloué sur la pile (allocation semi-dynamique).
  int a[n];
  init( a, n );
  imprimer( a, n );
  writes( "lb(a) = ", lb(a), ", ub(a) = ", ub(a), "\n" );

  # L'espace pour le tableau a est libéré lorsque la procédure se termine.
}

procedure allocDynamique( int n )
{
  # Le tableau a est alloué sur le tas, à l'aide d'un appel explicite
  # à l'instruction new. Comme il s'agit d'un tableau de taille variable,
  # la taille du tableau doit aussi être spécifiée.
  ptr TableauDyn a = new([n] int);
  init( a, n );
  imprimer( a, n );
  writes( "lb(a) = ", lb(a), ", ub(a) = ", ub(a), "\n" );

  # L'espace pour le tableau a n'est pas libéré, puisqu'il a été alloué sur le tas.
}

# Programme principal

# Allocation statique du tableau a.
int a[10];
init( a, 10 );
imprimer( a, 10 );
writes( "lb(a) = ", lb(a), ", ub(a) = ", ub(a), "\n" );

# Allocation semi-dynamique: 20 spécifie la taille du tableau qui sera créé
# dans la procédure (allocation sur la pile).
allocSemiDynamique( 20 );

# Allocation dynamique: une instruction new sera utilisée dans la procédure,
# le tableau (de 12 éléments) étant alors alloué sur le tas.
allocDynamique( 12 );
end
```

```

resource proc1()
  int n; getarg(1, n)
  int a[n][n];

  process p[i = 1 to n, j = 1 to n] {
    a[i, j] = i*j;
  }

  final {
    ecrire(a);
  }
end

resource proc2()
  int n; getarg(1, n)
  int a[n][n];

  procedure p( int i, int j ) {
    a[i, j] = i*j;
  }

  for [i = 1 to n, j = 1 to n] {
    fork p(i, j);
  }

  final {
    ecrire(a);
  }
end

resource proc3()
  int n; getarg(1, n)
  int a[n][n];

  procedure p( int i, int j ) {
    a[i, j] = i*j;
  }

  co [i = 1 to n, j = 1 to n]
    call p(i, j);          # Le call aurait pu etre omis
  oc

  ecrire(a);              # Le final n'est plus necessaire
end

```

Exemple de code 1: Différentes façons, sensiblement équivalentes, de créer un groupe de processus

A.2 Les programmes présentés dans l'exemple de code MPD ??

Une déclaration (statique) d'un ensemble de processus (avec des quantificateurs) peut être considérée comme une abréviation d'un ensemble d'instantiations avec `fork` d'une procédure appropriée. Strictement en termes des processus créés, l'effet est aussi le même que pour une instruction `co` avec un appel synchrone (`call`) de cette procédure. Les exemples présentés dans l'extrait de code ?? sont donc, en termes des processus créés, équivalents entre eux.

Une première différence apparaît toutefois au niveau de la terminaison : dans les deux premières variantes, ce sont des processus s'exécutant en arrière-plan qui sont créés. Plus précisément, pour `proc1`, n^2 processus en arrière-plan sont créés au moment où la déclaration des processus `p` est rencontrée, alors que pour `proc2`, ces processus sont activés *explicitement* par l'intermédiaire des instructions `fork` à l'intérieur de la boucle `for`. Comme ces processus sont créés en arrière-plan et qu'on veut imprimer le résultat `a` uniquement lorsque tous les processus auront complété leur tâche, on doit alors utiliser une clause `final` pour protéger et retarder l'exécution de l'instruction `ecrire`. Dans le cas de `proc3`, par contre, les n^2 processus sont activés par l'intermédiaire de l'instruction `co` ; l'exécution de l'appel à `ecrire` s'effectue alors lorsque cette instruction `co` se termine, c'est-à-dire lorsque tous les processus activés en parallèle ont terminé leur exécution.

Une autre différence importante entre ces trois versions apparaît aussi lorsqu'on fait une analyse asymptotique, et abstraite, du temps d'exécution des algorithmes correspondants. Ainsi, une instruction `for` s'exécute strictement de façon séquentielle, alors qu'une instruction `co` ou un ensemble de déclarations de `process`, en termes d'analyse d'algorithme, génère un ensemble de processus de façon concurrente. Les caractéristiques de performance de ces trois versions pourraient donc être décrites comme suit, en ignorant dans chaque cas le temps pour écrire le résultat :

- `proc1` : crée n^2 processus concurrents, chacun s'exécutant en temps $\Theta(1)$. Le temps total requis sera donc $\Theta(1)$.
- `proc2` : crée n^2 processus concurrents, chacun s'exécutant en temps $\Theta(1)$. Toutefois, comme l'activation des processus se fait par l'intermédiaire d'une boucle `for`, le temps total requis dépendra du temps pour exécuter la boucle `for`, donc sera $\Theta(n^2)$ (n^2 itérations, chacune de temps $\Theta(1)$).
- `proc3` : crée n^2 processus concurrents (activés par l'intermédiaire du `call` dans le `co`), chacun s'exécutant en temps $\Theta(1)$. Le temps total requis sera donc $\Theta(1)$.

A.3 Génération d'une série de valeurs

Le programme MPD ?? génère une série de valeurs, où pour chacune on détermine si la valeur générée est paire ou non. La génération d'une nouvelle valeur et la vérification de la valeur précédente se font en parallèle. À chaque itération, deux processus sont donc créés. Le nombre total de processus créés sera donc $2(n-1)$ — mais soulignons que seulement deux processeurs seront requis, puisqu'au plus deux processus seront actifs en même temps — alors que le temps d'exécution sera $\Theta(n)$.

Le programme MPD ?? effectue la même tâche, mais où la génération des valeurs et la vérification se font dans des processus itératifs indépendants (parallélisme itératif). Au total, seulement deux processus sont créés. Toutefois, un mécanisme de synchronisation (avec sémaphores) doit être utilisé pour assurer que les deux processus procèdent au même rythme et que la vérification d'un item du tableau ne se fasse pas avant que l'élément ait été généré. Cette synchronisation font alors en sorte que le temps d'exécution sera $\Theta(n)$.

Note : G.R. Andrews, dans son livre "*Foundations of Multithreaded, Parallel, and Distributed Programming*", appelle ces deux approches "*co-inside-while*" (création dynamique)

Programme MPD 2 Génération et vérification d'une série d'items avec création *dynamique* de processus

```
resource GenererEtVerifier()
  int n; getarg(1, n);

  int elems[1:n];
  bool estPair[1:n];

  procedure generer(int i) {
    elems[i] = int(random(1, n));
  }

  procedure verifier(int i) {
    estPair[i] = (elems[i] % 2) == 0;
  }

  # On genere et verifie les divers elements
  generer(1);
  for [i = 1 to n-1] {
    co verifier(i)
    // generer(i+1)
    oc
  }
  verifier(n);

  # On imprime les resultats
  write("elems::" )
  for [i = 1 to n] {
    printf(" [%d]: %d\n", i, elems[i] );
  }
  write("estPair::" )
  for [i = 1 to n] {
    printf(" [%d]: %b\n", i, estPair[i] );
  }
}
end
```

Programme MPD 3 Génération et vérification d'une série d'items avec création *statique* de processus

```
resource GenererEtVerifier()
  int n; getarg(1, n);

  int elems[1:n];
  bool estPair[1:n];

  sem precedentTermine = 1;
  sem nouveauGenere = 0;

  process generer {
    int i;
    for [i = 1 to n] {
      P(precedentTermine); # Attendre
      elems[i] = int(random(1, n));
      V(nouveauGenere); # Signaler
    }
  }

  process verifier {
    for [i = 1 to n] {
      P(nouveauGenere); # Attendre
      estPair[i] = (elems[i] % 2) == 0;
      V(precedentTermine); # Signaler
    }
  }

  final {
    # On imprime les resultats
    write("elems:" );
    for [i = 1 to n] {
      printf(" [%d]: %d\n", i, elems[i] );
    }
    write("estPair:" );
    for [i = 1 to n] {
      printf(" [%d]: %b\n", i, estPair[i] );
    }
  }
end
```

et “*while-inside-co*” (création statique). D’un point de vue théorique, purement algorithmique, la version dynamique est plus intéressante. Au niveau pratique toutefois, sur une machine réelle, la dernière approche est généralement préférable, car elle réduit le nombre de processus qui sont créés (la création d’un processus est une opération relativement coûteuse). Dans bien des cas, la tâche d’un *programmeur parallèle* est alors de définir un algorithme parallèle idéal, puis d’utiliser ses connaissances des paradigmes et des techniques de programmation parallèle pour transformer cet algorithme idéal en un programme fonctionnant de façon efficace sur la machine cible.