École Nationale Supérieure
d'Ingénieurs de Caen

6, Boulevard Maréchal Juin
F-14050 Caen Cedex, France

Pipeline logiciel ADA

Niveau	**2ème année**
Parcours	**Électronique et Physique Appliquée (majeure SATE)**
Responsable	**Emmanuel Cagniot** **Emmanuel.Cagniot@ensicaen.fr**

1 Introduction

L'objectif de ce projet est l'implémentation d'un *parallel design pattern* relativement méconnu ou négligé : `Pipeline`. Conséquence software du principe de fonctionnement des processeurs vectoriels, ce *pattern* présente la particularité de pouvoir s'appliquer à de nombreux problèmes tout en s'implémentant de façon simple. Nous allons donc l'utiliser dans le cadre d'un algorithme de tri de mauvaise réputation, le *bubble sort*. Cette implémentation sera réalisée en langage ADA et plus précisément dans sa dernière norme 2012. Ce langage, relativement méconnu et pourtant ancien (première norme en 1983), est abondamment utilisé dans le monde des systèmes critiques et des projets à très long terme (avionique, ferroviaire, bancaire, etc.) où la fiabilité et la tolérance aux pannes sont des notions essentielles ; il se caractérise par une tolérance zéro vis à vis des constructions programmatiques obscures ou exotiques.

Mots-clés : langage ADA, distribution GNAT GPL 2019, programmation générique, programmation par objets, programmation parallèle, programmation par contrats.

1.1 Opérateur pipeline

Commençons par rappeler le principe du pipeline.

Certaines instructions telles que les additions et les multiplications entières et/ou pseudo réelles sont présentes en grandes quantités dans les codes. De fait, elles sont traitées par des opérateurs câblés dédiés directement intégrés au processeur. Sous réserve d'accès simultanés à la mémoire (mémoires entrelacées composées de plusieurs modules), ces opérateurs utilisent un principe permettant d'accélérer l'exécution d'une séquence $\{i_n\}$ contenant n occurrences de la même instruction : le pipeline.

Le principe du pipeline consiste à :
— segmenter l'instruction en étapes si possible de même durée ;
— connecter les sorties de l'étape n aux entrées de l'étape $n+1$;
— synchroniser le passage des données entre étapes successives.

Sa mise en œuvre matérielle est réalisée comme suit :
— à chaque étape est associé un étage c'est à dire une ressource matérielle disposant de registres d'entrée et de sortie ;
— les étages sont synchronisés par une horloge ;
— l'ensemble des étages constitue le pipeline.

Considérons le cas concret d'une addition pseudo réelle. Cette instruction peut être segmentée en trois étapes consécutives :
— comparaison des exposants et alignement de la mantisse du plus petit nombre sur celle du plus grand (dénormalisation) ;
— addition des mantisses en virgule fixe ;
— normalisation du résultat (norme IEEE 754).

La figure 1 présente l'opérateur pipeliné correspondant.

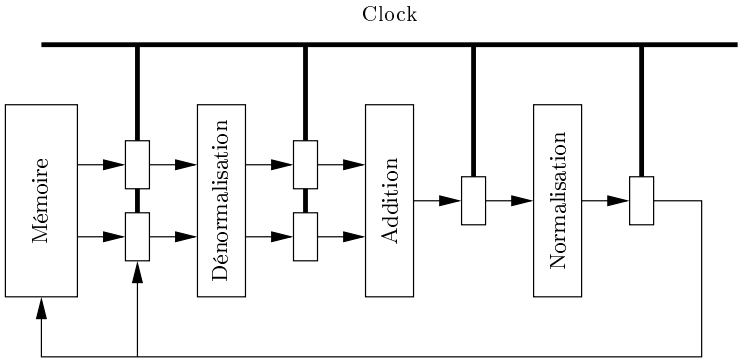

FIGURE 1 – Additionneur en virgule flottante.

Supposons que notre additionneur soit dans une configuration idéale, c'est à dire que chacun de ses étages nécessite la même durée de passage τ. Nous pouvons alors régler la période γ de l'horloge en choisissant $\gamma = \tau$.

La durée d'exécution scalaire, noté $\tau_{scalaire}$, représente la durée d'exécution d'une séquence de n additions si notre opérateur se comporte selon le modèle d'exécution scalaire de VON NEUMAN, c'est à dire qu'il attend le résultat de l'addition i_p avant de lancer l'addition suivante i_{p+1}. Dans ce cas, nous avons tout simplement le résultat (désespérant) :

$$\tau_{scalaire} = n \times 3 \times \tau. \tag{1.1}$$

La latence, notée $\tau_{latence}$, représente la durée nécessaire à l'obtention du résultat de la première addition i_0 si notre opérateur est exploité en mode pipeline, c'est à dire qu'il exécute simultanément (régime de croisière) :
— une dénormalisation sur l'addition i_{p+2} ;
— une addition en virgule fixe sur l'addition i_{p+1} ;
— une normalisation sur l'addition i_p.

Dans notre cas, la latence représente la durée nécessaire à la traversée des trois étages de l'additionneur c'est à dire :

$$\tau_{latence} = 3 \times \tau. \tag{1.2}$$

Notre additionneur fonctionnant à présent en mode pipeline, le premier résultat est obtenu trois tops d'horloge plus tard que l'injection de l'addition i_0. Cependant, contrairement au mode de fonctionnement scalaire, les résultats suivants arrivent juste derrière à raison d'un nouveau résultat par top d'horloge, le

pipeline ayant atteint son régime de croisière. Par conséquent, la durée d'exécution d'une séquence de n additions est maintenant donnée par :

$$\tau_{pipeline} = \tau_{latence} + (n-1) \times \tau = (3+n-1) \times \tau. \tag{1.3}$$

Afin de mesurer le gain de rapidité procuré par le mode de fonctionnement pipeline par rapport au mode scalaire, nous définissons le facteur d'accélération (speed up) comme suit :

$$s_3^n = \frac{\tau_{scalaire}}{\tau_{pipeline}} = \frac{3 \times n}{3+n-1}. \tag{1.4}$$

Faisons à présent tendre le nombre d'additions n vers l'infini, c'est à dire que la séquence d'additions devient très longue ; il vient le résultat (magique cette fois) :

$$s_3^\infty = lim_{n \to \infty} \Big(\frac{3}{\frac{3}{n} + 1 - \frac{1}{n}} \Big) = 3, \tag{1.5}$$

ce qui signifie que dans le cas idéal, le mode de fonctionnement pipeline de notre additionneur est trois fois plus rapide que le mode de fonctionnement scalaire de Von Neuman. Par conséquent, un pipeline sera d'autant plus rapide que son nombre d'étages est élevé et que les temps de traversée sont relativement homogènes (configuration proche du cas idéal).

Transposons à présent ce principe en ingénieurie logicielle.

De nombreux problèmes se caractérisent par une chaîne de traitements successifs au travers de laquelle passent des données. L'exemple le plus connu est la commande `pipe` (symbole `'|'`) du système d'exploitation Unix.

Les traitements pouvant être nombreux, une parallélisation basée sur une boucle de type `forall` sur les données (*data parallelism*, premier type de parallélisation venant généralement à l'esprit) impliquerait l'écriture d'un code relativement complexe et donc difficilement maintenable. Pourvu que chaque traitement puisse être réalisé indépendamment des autres et qu'il ne mette à jour aucune structure de données partagée alors le problème considéré peut être parallélisé beaucoup plus efficacement (et surtout simplement) via le *parallel design pattern* `Pipeline`.

En général un *parallel design pattern* ne peut être complètement modélisé en UML (voire pas du tout) et ce pour plusieurs raisons :

— les compositions ou aggrégations sont parfois insuffisantes pour modéliser les dépendances entre classes ;
— les communications entre objets ne se résument pas à de simples invocations de méthodes ;
— etc.

Cependant, dans le cas du *pattern* `Pipeline`, il existe une forte proximité avec le *behavioral design pattern* `Chain of Responsability` dont la figure 2 présente le diagramme de classes.

En conception objet, le *behavioral design pattern* `Chain of Responsability` permet à un client d'envoyer une commande (requête) sans nommer explicitement l'objet qui la traitera. Plus précisément, le client émet une requête à l'élément placé en tête d'une liste constituée d'objets susceptibles de pouvoir la satisfaire. Si ce premier élément peut traiter la requête alors il le fait. Dans le cas contraire, il la transmet à l'élément suivant dans la liste et ainsi de suite. Ce *pattern* permet de réduire la dépendance entre le client et l'objet qui traitera la requête puisque ce dernier n'est jamais explicitement nommé.

Si nous transposons le *pattern* `Chain of Responsability` dans le monde du pipeline, les objets de la liste représentent les étages tandis que les requêtes représentent les données qui circulent au travers de ces étages. Cependant, la différence réside dans le fait qu'une requête, qui n'est pas émise par un client mais par le premier étage du pipeline, subit des modifications au fil des étages.

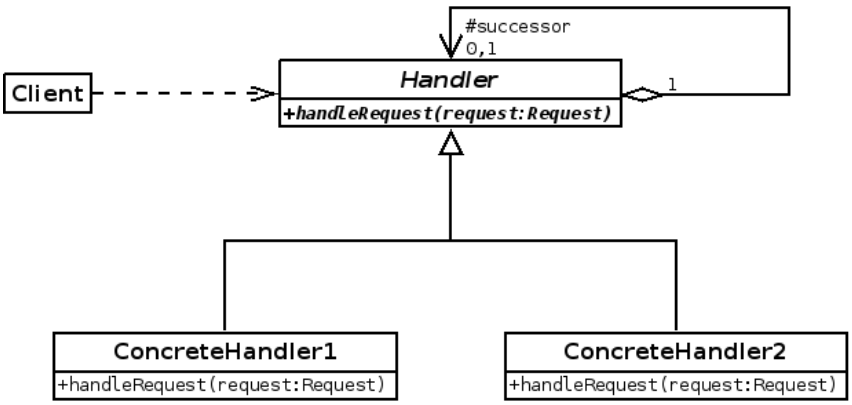

FIGURE 2 – Diagramme de classes du behavioral design pattern `Chain Of Responsability`.

Toujours par analogie avec les *design patterns* exploités en conception objet, `Pipeline` serait classé dans la famille des *structural design patterns* puisqu'il permet d'obtenir un traitement complexe par composition de traitements élémentaires. Dès lors, les bénéfices pour le programmeur sont évidents, ce dernier n'ayant besoin que de :

- décrire chaque traitement élémentaire indépendamment des autres (mise au point et debugging simplifiés à l'extrême) ;
- composer les traitements élémentaires pour obtenir le pipeline réalisant le traitement total ;
- laisser faire le pipeline pour l'aspect parallélisme.

En résumé, le programmeur s'occupe de la partie séquentielle (le fonctionnement des étages) de son code et n'aborde que peu ou prou la partie parallèle inhérente au fonctionnement du pipeline.

1.2 Algorithme du *bubble sort*

Venons-en à présent à l'algorithme qui nous intéresse : le *bubble sort* dont le pseudo-code est donné ci-dessous pour la relation d'ordre « strictement inférieur à ».

Procedure 1 *bubble sort*

```
Input : vect(1 : n)
Ouput : vect(1 : n)
  continue := true
  while continue do
    continue := false
    for i := 1 to n - 1 do
      if Vect(i) >= Vect(i + 1) then
        tmp := Vect(i)
        Vect(i) := Vect(i + 1)
        Vect(i + 1) := tmp
        continue := true
      end if
    end for
  end while
```

Sa complexité au pire est $\mathcal{O}(n^2)$ si n représente le nombre d'éléments à trier. Comme les comparaisons d'éléments se font selon le schéma (e_i, e_{i+1}), une parallélisation de type `forall` (le premier réflexe) est in-envisageable. Cependant, bien qu'il ne soit forcément évident, nous pouvons envisager un schéma de type *pipeline* :

1. un étage de fragmentation qui consiste à segmenter le conteneur à trier en petit morceaux qui sont envoyés à l'étage de tri. Les fragments sont alloués dynamiquement ;
2. un étage de tri qui ordonne chaque petit fragment reçu via la version séquentielle du *bubble sort* puis l'envoie à l'étage de fusion (complexité au pire en $\mathcal{O}(n^2)$ mais nombre d'éléments n très petit) ;
3. un étage de fusion qui agglomère le fragment reçu à la partie du conteneur déjà triée. Dans la pratique l'étage de fusion risque de devenir le goulet d'étranglement puisque la complexité d'une fusion de deux listes ordonnées de tailles respectives m et n est $\mathcal{O}(n+m)$. Il existe bien sûr des algorithmes parallèles dédiés à cette problématique mais ce n'est pas ici notre sujet. Les fragments alloués par l'étage de fragmentation sont dés-alloués dans l'étage de fusion.

2 Progression du travail

La première chose à faire est de récupérer l'environnement gratuit GNAT GPL 2019 (GNAT *Community Edition*) via l'url (choisissez la distribution associée au système d'exploitation et l'architecture sur lesquels vous souhaitez travailler) :

`https://www.adacore.com/download/`

Cette distribution est un « tout en un », éditée une fois par an et qui propose à la fois l'intégralité des outils de développement GNAT mais également les bibliothèques utilisées par ces outils (celles de votre système ne sont pas utilisées car peut être plus anciennes ou plus récentes).

Une fois l'installation effectuée, commencez à vous familiariser avec l'environnement de développement GPS (GNAT *Programming Studio*) et le langage ADA (ceux d'entre vous ayant déjà codé en VHDL devraient logiquement avoir une sensation de « déjà vu ») au travers du cours en ligne (très bien écrit malgré quelques petites coquilles) :

`https://openclassrooms.com/fr/courses/900279-apprenez-a-programmer-avec-ada`

L'idée n'est pas d'ingérer toutes les notions proposée dans ce cours mais juste d'effectuer les premiers pas avec ADA : vous y reviendrez lors de chaque étape de notre progression. Profitez-en également pour comprendre la structure d'un script de compilation `GPRBuild` comme celui présenté en figure 3.

```
1  project Build is
2
3     for Source_Dirs use ("src/**");
4     for Object_Dir use "obj";
5     for Exec_Dir use "bin";
6     for Main use ("demo.adb");
7
8     package Compiler is
9        for Default_Switches ("ada") use ("-gnat12",
10                                         "-gnata",
11                                         "-gnaty",
12                                         "-gnatg",
13                                         "-O3");
14    end Compiler;
15
16 end Build;
```

FIGURE 3 – Exemple simple de script de compilation `GPRBuild`.

2.1 Programmation générique

Notre application aura besoin de deux opérations de base : trier un tableau d'éléments selon l'algorithme du *bubble sort* et fusionner deux tableaux triés en un seul. Il faut donc commencer par définir ces deux opérations sous la forme de deux sous-programmes indépendants.

Afin de pouvoir gérer des tableaux de n'importe quel type, de n'importe quelle longueur et pouvant être indicés par autre chose que des entiers naturels, nous optons pour des sous-programmes génériques permettant d'opérer sur des tableaux non contraints (bornes inconnues) dont les éléments sont d'un type privé `Element` (au sens ADA du terme) et dont les indices sont d'un type discret `Position` quelconque (entiers, types énumérés, etc.). La relation d'ordre utilisée, symbolisée par l'opérateur `<`, est au choix du programmeur.

La figure 4 donne l'exemple de la spécification d'un sous-programme générique `Merge`.

```
1  generic
2
3     type Element is private;
4     --  Type représentant les éléments manipulés.
5
6     type Position is (<>);
7     --  Type représentant les indices des tableaux manipulés (ici tout type
8     --  discret).
9
10    type Array_Of_Elements is array (Position range <>) of Element;
11    --  Type représentant les tableaux manipulés (ici des tableaux non
12    --  contraints).
13
14    with function "<"
15      (Left, Right : Element)
16       return Boolean
17      is <>;
18    --  Relation d'ordre souhaitée (par défaut "strictement inférieur à").
19
20 procedure Merge
21   (Left, Right :     Array_Of_Elements;
22    Result      : out Array_Of_Elements);
23 --  Fusionne deux tableau préalablement ordonnés dans un tableau cible.
24 --  L'implémentation proposée ici permet de gérer le cas où l'un des tableaux
25 --  à fusionner représente la partie basse du tableau cible. Cependant, elle ne
26 --  permet pas de gérer le cas "in-place" dans lequel l'un des tableaux à
27 --  fusionner représente la partie basse du tableau cible et le second sa
28 --  partie haute.
```

FIGURE 4 – Spécification d'un sous-programme générique `Merge`.

Tous les composants logiciels liés à notre implémentation pipeline du *bubble sort* seront rattachés à un paquetage générique et maître `Pipeline`, paramétré par les mêmes entités que les deux routines décrites ci-dessus. Ce paquetage propose, par le biais d'un sous-paquetage, un type `Chunck` représentant un pointeur de tableaux et deux routines permettant de les allouer/désallouer dynamiquement. La figure 8 présente sa spécification.

2.2 Programmation par objets

Bien que ce ne soit pas une obligation, nous choisissons d'imiter la bibliothèque C++ THREADING BUILDING BLOCKS d'INTEL en découplant la fonction d'un étage (programmation séquentielle) de son

```
generic

   type Element is private;
   --  Type représentant les éléments manipulés.

   type Position is (<>);
   --  Type représentant les indices des tableaux manipulés (ici tout type
   --  discret).

   type Array_Of_Elements is array (Position range <>) of Element;
   --  Type représentant les tableaux manipulés (ici des tableaux non
   --  contraints).

   with function "<"
     (Left, Right : Element)
      return Boolean
     is <>;
   pragma Unreferenced ("<");
   --  Relation d'ordre souhaitée (par défaut "strictement inférieur à").

package Pipeline is
   --  Paquetage maître auquel se rattachent tous les composants logiciels
   --  liés à une implémentation pipeline de l'algorithme du bubble sort.

   package Chunk is
      --  Sous-paquetage définissant un type non objet représentant un
      --  fragment de tableau circulant d'un étage à un autre.

      type Instance is access Array_Of_Elements;
      --  Notre type représentant in fine un pointeur vers un fragment de
      --  tableau alloué sur le tas.

      function Allocate
        (An_Array : Array_Of_Elements)
         return Instance;
      --  Alloue un fragment sur le tas puis en retourne le pointeur
      --  correspondant.

      procedure Deallocate
        (A_Chunk : in out Instance);
      --  Désalloue un fragment du tas.

   end Chunk;

end Pipeline;
```

FIGURE 5 – Spécification du paquetage maître `Pipeline`.

support parallèle (ici les tâches ADA). Par conséquent, nous définissons une interface `Stage` censée être un équivalent de la classe abstraite `filter` de TBB. Cette interface propose une méthode `Run` prenant un fragment en entrée et fournissant un fragment en sortie. La figure 6 présente la spécification du sous-paquetage correspondant.

```
1  generic
2  package Pipeline.Stage is
3     -- Sous-paquetage définissant une interface représentant un étage de
4     -- pipeline.
5
6     type Services is limited interface;
7     -- Notre interface.
8
9     procedure Run
10       (A_Stage : in out Services;
11        A_Chunk : in out Chunk.Instance)
12    is abstract;
13    -- Reçoit un fragment en entrée, effectue un traitement à partir de
14    -- ce fragment puis fourni un fragment en sortie.
15
16 end Pipeline.Stage;
```

FIGURE 6 – Spécification du sous-paquetage générique `Stage`.

Une fois notre interface en place, ne reste plus qu'à définir trois classes concrètes qui implémentent cette interface (trois sous-paquetages de `Pipeline.Stage`) :

`Splitting` : qui représente l'étage de fragmentation. Sa redéfinition de la méthode `Run` ne tient pas compte du fragment reçu. Elle segmente le tableau à trier en petit fragments qu'elle alloue dynamiquement. La valeur spéciale `null` fourni en sortie signifie que la fragmentation est terminée. Afin de ne pas se retrouver confronté au problème des paramètres tableaux `aliased` sur lesquel ADA semble un peu pointilleux, nous fournissons le tableau à trier (et non pas son adresse via un pointeur) en tant que constante générique (`An_Array`) du sous-paquetage `Pipeline.Stage.Splitting` correspondant. Celui-ci prévoit également un paramètre supplémentaire : une constante positive `Chunk_Max_Size` représentant la capacité de stockage maximum d'un fragment. Cette constante est initialisé à la valeur 8 par défaut ;

`Sorting` : qui ordonne les éléments du fragment reçu via une implémentation du bubble sort ;

`Merging` : qui fusionne le fragment reçu avec la partie déjà ordonnée du tableau avant de le désallouer. Pour les mêmes raisons que l'étage de partitionnement, le tableau à trier est fourni en tant que variable générique (`An_Array`) du sous-paquetage `Pipeline.Stage.Merging` correspondant.

La figure 7 présente la spécification (incomplète) de la classe `Splitting`.

2.3 Programmation parallèle

Le codage des étages étant terminé, nous pouvons maintenant passer à l'implémentation parallèle proprement dite. L'idée est de fournir à l'utilisateur une routine générique `Pipelined_Bubble_Sort`, semblable à `Bubble_Sort` mais comportant un paramètre supplémentaire : une constante `How_Many_CPUs` indiquant le nombre de processeurs à utiliser. La valeur par défaut de ce nouveau paramètre est le nombre de processeurs total de la machine. En fonction de l'argument fourni pour ce paramètre lors de l'invocation, notre nouvelle fonction détermine l'implémentation à utiliser. La figure 8 présente sa spécification.

Là encore, nous recourons à l'objet pour définir nos différentes implémentations parallèles. Nous commençons par définir une interface `Implementation` (sous-paquetage `Pipeline.Implementation`) propo-

```
generic

   An_Array        : Array_Of_Elements;
   --  Le tableau à fragmenter.

   Chunk_Max_Size : Positive := 8;
   --  Taille maximum d'un fragment de tableau.

package Pipeline.Stage.Splitting is
   --  Sous-paquetage représentant l'étage d'entrée dont le rôle est de
   --  fragmenter le tableau à trier.

   type Instance is new Stage.Services with private;
   --  Notre classe.

   overriding
   procedure Run
     (A_Stage : in out Instance;
      A_Chunk : in out Chunk.Instance);
   --  Redéfinition. La valeur du fragment en sortie vaut systématiquement
   --  null s'il n'y a plus rien à fragmenter.

private

   type Instance is new Stage.Services with
      record
      --  A vous de savoir quel(s) attribut(s) ajouter ...
      end record;
   --  Structure interne de notre classe.

end Pipeline.Stage.Splitting;
```

FIGURE 7 – Spécification (incomplète) du sous-paquetage générique `Splitting`.

```
1  with System.Multiprocessors; use System.Multiprocessors;
2
3  generic
4
5     type Element is private;
6     --  Type représentant les éléments manipulés.
7
8     type Position is (<>);
9     --  Type représentant les indices des tableaux manipulés (ici tout type
10    --  discret).
11
12    type Array_Of_Elements is array (Position range <>) of Element;
13    --  Type représentant les tableaux manipulés (ici des tableaux non
14    --  contraints).
15
16    with function "<"
17      (Left, Right : Element)
18       return Boolean
19      is <>;
20    --  Relation d'ordre souhaitée (par défaut "strictement inférieur à").
21
22 procedure Pipelined_Bubble_Sort
23   (An_Array       : in out Array_Of_Elements;
24    How_Many_CPUs :         CPU := Number_Of_CPUs);
25 --  Ordonne les éléments du tableau fourni en argument via une implémentation
26 --  parallèle de l'algorithme du bubble sort basée sur une pipeline logiciel
27 --  à trois étages. Le nombre de CPUs par défaut est celui de l'architecture
28 --  sous-jacente.
```

FIGURE 8 – Spécification du sous-programme générique `Pipelined_Bubble_Sort`.

sant une unique méthode `Process` prenant un tableau à trier en paramètre. La figure 9 présente sa spécification.

```
1  generic
2  package Pipeline.Implementation is
3     -- Sous-paquetage définissant une interface représentant une
4     -- implémentation pipeline particulière de l'algorithme du bubble sort.
5
6     type Services is limited interface;
7     -- Notre interface.
8
9     procedure Process
10      (An_Implementation :         Services;
11       An_Array          : in out Array_Of_Elements)
12    is abstract;
13    -- Ordonne les éléments d'un tableau.
14
15 end Pipeline.Implementation;
```

FIGURE 9 – Spécification du sous-paquetage générique `Implementation`.

Le fil directeur d'une implémentation parallèle est de ne jamais créer plus de tâches ADA que le nombre de processeurs indiqué par l'utilisateur. Nous définissons ensuite deux classes concrètes représentant des implémentations :

`Dual_CPU` : dédié au cas bi-processeur (sous-paquetage `Pipeline.Implementation.Dual_CPU`). Comme la charge cumulée des étages de fragmentation et de tri peut compenser celle de l'étage de fusion, nous définissons deux types tâches (internes à la rédefinition de la méthode `Process`) et une tâche représentante pour chacun. Le premier type (`Merging_Stage_Manager`) propose une entrée `RDV` permettant de transmettre un fragment à l'étage de fusion défini en tant que variable locale de la tâche. Cette dernière s'arrête lorsque la fragmentation est terminée. Le second (`Splitting_Sorting_Manager`) ne propose aucune entrée (donc immédiatement exécutable) et encapsule les étages de fragmentation et de tri en tant que variables locales. Une tâche de ce type désalloue chaque fragment reçu et s'arrête lorsqu'il n'y a plus aucune demande de rendez-vous ;

`More_Than_Two_CPUs` : dediée aux machines comportant plus de deux processeurs (sous-paquetage `Pipeline.Implementation.More_Than_Two_CPUs`). Nous définissons ici trois types tâches :

1. `Splitting_Manager` : qui encapsule l'étage de fragmentation. Ce type propose une unique entrée `RDV` permettant d'obtenir un nouveau fragment. L'allocation de ce dernier est réalisé pendant le rendez-vous. Une seule et unique tâche représente ce type et celle-ci s'arrête lorsqu'il n'y a plus de demande de rendez-vous ;
2. `Merging_Manager` : analogue à celui de l'implémentation précédente dédiée aux bi-processeurs. Une seule tâche représente ce type ;
3. enfin, un type `Sorting_Manager` ne proposant aucune entrée (donc immédiatement exécutable). Toute tâche de ce ce type demande un rendez-vous à la tâche gérant l'étage de fragmentation pour obtenir un fragment, le tri puis sollicite un rendez-vous auprès de la tâche gérant l'étage de fusion pour la lui transmettre ; elle s'arrête lorsque le fragment reçu vaut la valeur spéciale `null`. Les tâches de ce type sont rassemblées dans un tableau dont la taille est le nombre de processeurs moins deux (un processeur pour la fragmentation, un autre pour la fusion). Ce sont ces dernières qui donnent le tempo.

In fine, ne reste plus qu'à écrire le programme principal `demo.adb` permettant de comparer implémentation séquentielles et parallèles.

2.4 Programmation par contrats

La fiabilité d'un code peut reposer sur plusieurs techniques :

— la programmation « défensive ». Le code se protège en permanence contre les erreurs logicielles (dues à une mauvaise utilisation de la part du programmeur), matérielles (dues à un circuit défectueux ou capricieux) ou arithmétique (par exemple une division par zéro) grâce au mécanisme des exceptions. En cas d'erreur, le contrôle est immédiatement transmis à une portion du code chargée de gérer cette erreur afin de reprendre l'exécution dans un état stable. Cette technique impose de tester en permanence si une opération peut être exécutée avant de l'exécuter et ce même si l'utilisateur est certain de son fait (aucun risque) ;

— les « tests » c'est à dire qu'a toute opération `Op` sont associées deux routines `can_do_op` et `do_op`. Si l'utilisateur est certain de son fait alors il appelle directement la seconde. S'il ne l'est pas alors il commence par invoquer la première. Cette technique est généralement généralisée par le biais des assertions (mot-clé `assert` dans la plupart des langages). Celles-ci restent actives le temps du développement puis sont neutralisées lors de la mise en production du code ;

— les « contrats ». Le problème des assertions est qu'elles sont directement implantées au niveau des instructions (et donc noyées au milieu de dizaines de milliers de lignes). Par conséquent, la façon d'utiliser le morceau de code correspondant est indiqué au niveau des commentaires (s'ils existent évidemment). Les contrats permettent de résoudre ce problème en installant les assertions directement au niveau des spécifications du sous-programme et non pas de ses instructions. Comme pour les assertions classiques, les contrats sont maintenus actifs pendant la phase de développement et peuvent être supprimés lors de la mise en production. Le second avantage par rapport aux assertions est qu'ils peuvent être totalement ou partiellement levés. En règle générale, seules les post-condition et les invariants de types sont levés tandis que les pré-conditions sont maintenues ;

— les « tests unitaires ». L'idée est ici de comparer notre réponse à un système « impulsion-réponse » : nous soumettons le code à des jeux d'entrées (plusieurs centaines de milliers parfois) dont nous connaissons les réponses. Si le code échoue sur un jeux entrées alors c'est qu'il est faux. Tout le problème consiste à définir les jeux d'entrées couvrant tous les cas d'utilisation possibles. La mise en œuvre des tests unitaires se fait en générale par le biais d'un *framework* logiciel (il ne s'agit pas à proprement parler d'un mécanisme du langage considéré).

ADA propose à la fois les exceptions et les contrats (ceux-ci étant basés sur les exceptions) : c'est à ces derniers que nous allons nous essayer.

Ces contrats peuvent être implantés au sommet c'est à dire, dans notre cas, au niveau du sous-programme `Pipelined_Bubble_Sort`. Par exemple, il n'impose rien sur le tableau à trier en entrée (pas de pré-condition) mais garantit en sortie (post-condition) que celui-ci est trié. Pour ce faire, il faut vérifier que les éléments sont ordonnés et que les fréquences du tableau en entrée (le nombre d'occurrences de chaque élément) sont identiques à celles du tableau en sortie. Cependant, est-il possible avec une telle approche de lever les contrats en phase de production ou, dit autrement, est-on certain que notre fonction ne présente aucune faille ? Vu que nous avons eu recours au parallélisme à l'intérieur de la fonction, c'est assez difficile à affirmer. Dans ce cas particulier (et uniquement dans ce cas), il vaut mieux procéder de la façon suivante :

— placer les contrats sur les opérations basiques (ici les sous-programmes `Bubble_Sort` et `Merge`) ;

— garantir que l'implémentation parallèle est déterministe (sujet de la section suivante).

Définissez deux contrats sous la forme de deux nouveaux sous-programmes génériques `Is_Sorted` et `Have_Same_Frequencies`. Les sous-programmes `Bubble_Sort` et `Merge` garantissent tous deux que le tableaux en sortie est trié. Le sous-programme `Merge` impose en entrée que la taille du tableau `Result` soit égale à la somme des tailles des tableaux `Left` et `Right` ; il garantit en sortie que les fréquences cumulées des tableaux `Left` et `Right` sont celles du tableau `Result` en sortie (regardez du côté des conteneurs associatifs (*map*) proposés par la bibliothèque standard).

Une fois au point, supprimez les post-conditions tout en conservant les pré-conditions (voir *pragma* `Assertion_Policy` à placer au début des spécifications).

2.5 Profil `RAVENSCAR`

Il est déjà difficile de garantir la fiabilité d'une application séquentielle lorsque le nombre de configurations pouvant survenir est combinatoire : imaginez ce que cela donne lorsque le parallélisme entre en jeu. Pour y parvenir, il est nécessaire de supprimer les constructions « non prouvables »formellement, par exemple l'absence d'inter-blocage (interdiction des rendez-vous entre tâches), les priorités dynamiques de tâches, etc. Le profil `RAVENSCAR` définit (par le biais de *pragmas*) l'ensemble des constructions parallèles interdites. Il s'agit d'un chemin de croix pour le programmeur mais, avec au bout, la certitude que son application est prouvable formellement et donc exempte de failles.

Si vous vous sentez d'attaque, écrivez une seconde version de `Pipelined_Bubble_Sort` basée sur le profil `RAVENSCAR` (il vous faudra absolument « tout casser » et faire communiquer vos tâche par boîte à lettre unique c'est à dire via les objets protégés). Vous l'aurez compris, ici l'objectif n'est plus tant la performance que la fiabilité d'exploitation.

Notez qu'ADA possède un « petit frère » uniquement dédié à ce cas de figure c'est à dire l'écriture de codes critiques parallèles, temps-réels, etc. et prouvables formellement par le biais d'un système de contrats très perfectionné : le langage SPARK.