Parallélisme

Brigitte Plateau - Anne Rasse - Jean-Louis Roch - Jean-Pierre Verjus

ENSIMAG

1995 - 1996

2

Table des matières

1 Introduction **7**

1.1 Problématique et bref historique 7

1.2 Pourquoi le parallélisme 9

1.3 Organisation du cours 10

2 Principes de Base **13**

2.1 Un exemple didactique 13

2.1.1 Quel algorithme? 14

2.1.2 Quel programme? 15

2.1.3 L'efficacité 17

2.2 Système parallèle et processus 18

2.2.1 La notion de processus 18

2.2.2 Etat d'un système parallèle 19

2.2.3 Relations entre processus 20

2.2.4 Le non déterminisme 21

2.2.5 La synchronisation 23

2.2.6 Vérification et comportements pathologiques 24

2.3 Exemples de problèmes de la programmation parallèle 25

2.3.1 Le produit itéré 25

2.3.2 Les lecteurs-rédacteurs 26

3 Eléments parallèles du langage ADA **27**

3.1 Introduction 27

3.2 Activation et état d'une tâche 28

3.3 Structures des tâches 30

3.3.1 La déclaration de spécification 30

3.3.2 Sélection et acceptation 31

3.3.3 Indexation d'une famille de points d'entrée 35

3.4 Définition des types de tâches 36

3.5 Autres possibilités du langage 36

3.6 Exemples de programmation en ADA 37

3.6.1 Exclusion mutuelle 37

3.6.2 Files d'attente de processus 38

3.6.3 Les pompiers 39

3.6.4 Les lecteurs-rédacteurs 40

3.6.5 Le produit itéré 42

3.7 Conclusion 42

4 Fondements de l'Algorithmique Parallèle **45**

4.1 Parallélisme de données et fonctionnel 46

4.1.1 Parallélisme fonctionnel 46

4.1.2 Parallélisme de données 48

4.1.3 Le mode Parallèle sur les données 48

4.1.4 Le mode Pipeline 49

4.2 Complexité Parallèle 50

4.2.1 Le modèle PRAM 51

4.2.2 Le travail d'un algorithme 54

4.2.3 La classification $\mathcal{NC}$ 57

4.2.4 Parallèle versus séquentiel 58

4.2.5 Le mode parallèle versus le mode séquentiel 60

4.3 Méthodes de base 61

4.3.1 Equilibre des travaux : le produit itéré et les préfixes 61

4.3.2 Graphe de dépendance et parallélisme : résolution d'un système linéaire triangulaire 66

4.3.3 Redondance et parallélisme : l'addition d'entiers 67

4.4 Programmation en ADA 69

4.4.1 Le produit itéré par la méthode non optimale 69

4.4.2 Le produit itéré optimal 70

4.4.3 Le calcul des préfixes 72

4.5 Conclusion 74

5 Construction d'Algorithmes Parallèles **79**

5.1 Algèbre linéaire 80

5.1.1 Produit matrice-matrice et matrice-vecteur 80

5.1.2 Résolution de système linéaire triangulaire 81

5.2 Retour sur l'addition d'entiers 83

5.2.1 Maîtriser la redondance 83

5.2.2 Un algorithme optimal pour l'addition d'entiers 84

5.3 Tri Parallèle 86

5.3.1 Tri par sélection 86

5.3.2 Tri par fusion et réseaux de tri 88

5.3.3 Fusion Bitonique 89

5.3.4 Fusion pair-impair 92

5.4 Conclusion 94

5.4.1 Algorithme synchrone sur architecture asynchrone 94

5.4.2 Modèle de mémoire 95

5.4.3 L'opérateur *fork* 96

5.5 Programmes ADA 97

6 Concurrence et Synchronisation **105**

6.1 Trois architectures de synchronisation 106

6.1.1 Exemple introductif : coopération entre producteur et consommateur 106

6.1.2 Spécification et variables d'état 109

6.1.3 Synchronisation centralisée ou distribuée 113

6.2 Synchronisation centralisée 114

6.3 Synchronisation mixte 117

6.3.1 Module partageable 117

6.3.2 Variables partagées et exclusion mutuelle ou sémaphore. 120

6.4 Synchronisation distribuée 126

6.4.1 L'exemple du producteur-consommateur 127

6.5 Architectures de synchronisation et de machines 132

6.6 Conclusion 134

7 Algorithmique distribuée **139**

7.1 Modèle de cohérence de données 141

7.1.1 Modèle de cohérence de la PRAM 142

7.1.2 Datation dans une machine parallèle 142

7.2 Solution centralisée 143

7.2.1 Principe 143

7.2.2 Programmation en ADA 144

7.3 Exclusion mutuelle distribuée : le jeton 146

7.3.1 Principe 146

7.3.2 La programmation en ADA 148

7.4 Diffusion atomique 153

7.4.1 Les horloges de Lamport 153

7.4.2 La diffusion atomique 154
7.4.3 La gestion des copies multiples 154
7.5 Mécanisme d'invalidation 155
7.6 Conclusion 157

8 Modèles et Spécifications **159**

8.1 Introduction 159
8.2 Modélisation des comportements 159
8.2.1 Graphe des exécutions. 159
8.2.2 Composition parallèle de processus. 160
8.2.3 Modélisation des contraintes de synchronisation à l'aide des réseaux de Petri. 164
8.3 Vérification qualitative 171
8.3.1 Introduction 171
8.3.2 Spécifications 172
8.3.3 Vérification de propriétés sur le graphe des marquages 174
8.4 Évaluation quantitative 179
8.4.1 Introduction 179
8.4.2 Approche probabiliste 181
8.4.3 Valuation du graphe des exécutions 182
8.4.4 La théorie de Markov 183
8.4.5 Réseaux de Petri Stochastiques 187
8.5 Bibliographie 188

9 Interblocage et Famine **189**

9.1 Introduction 189
9.1.1 Interblocage 189
9.1.2 Détection de la famine sur le graphe d'exécution. 191
9.1.3 Rappel des spécifications du problème des lecteurs-rédacteurs. 193

9.2 Lecteurs/rédacteurs avec synchronisation centralisée 193
9.2.1 Mise en défaut de l'approche naïve 193
9.2.2 Respecter la spécification. 195
9.2.3 Eliminer la famine 198
9.2.4 Programmer l'équité. 200
9.3 Lecteurs/rédacteurs avec synchronisation distribuée 203
9.3.1 Respecter la spécification. 203
9.3.2 Eliminer la famine - Programmer l'équité. 207

A Notions sur le langage ADA séquentiel **211**

A.1 Les déclarations 211
A.2 Les expressions 212
A.3 Les créations dynamiques d'objets 212
A.4 Les instructions 213
A.5 Les sous programmes 214

Chapitre 1

Introduction

1.1 Problématique et bref historique

Un système informatique est comme une usine dans laquelle le travail est partagé entre les divers employés. Les employés sont assimilés à des unités de calcul et les divers moyens de stockage (armoire à dossiers, magasin de pièces) à des mémoires, et l'usine est une image de ce qu'on appelle un système informatique pour le traitement *coopératif*. Dans cette usine, considérons divers modes d'organisation possibles du travail.

- L'usine est divisée en unités de production ou de gestion : un service comptable, un service de contrôle de qualité, deux unités de production. La coordination entre les services se fait par les quatre responsables de services, et les décisions globales se prennent à ce niveau qui n'implique que peu de personnes. Mis à part cette coordination, les unités sont autonomes et communiquent par des dossiers par exemple. L'efficacité d'une telle organisation tient à la fois à l'autonomie des unités et à la bonne coordination des chefs qui sont en petit nombre. Cette coordination est relativement peu fréquente et ne peut fonctionner que si il y a peu de participants. Cependant elle permet des relations peu codifiées entre les unités et donc une relative souplesse d'adaptation. En informatique, c'est ce qu'on appelle un *système informatique réparti* (ou distribué) et les unités sont des sous-*systèmes communicants*. La communication entre les unités se fait par transmission de dossiers que l'on assimile aux *messages* en informatique.

- A l'intérieur des unités (composées de 10 à 100 voire 1000 ouvriers), le travail peut être organisé de la même manière : un ouvrier construit une voiture entièrement ce qui le rend quasi autonome. Sa coordination avec ses collègues se situe au niveau du partage des outils. Ce n'est plus le mode d'organisation du travail utilisé dans les grosses unités de production, à cause de son inefficacité. A la place, on a mis en place une coordination – le travail à la chaîne – entre les ouvriers. Cette coordination est stricte et figée à l'avance, chacun effectuant une sous tâche et fournissant le résultat de son travail à certains de ses collègues. C'est ce qu'on assimile en informatique à un *ordinateur parallèle,* éventuellement massivement parallèle si le nombre des unités de calcul est grand.

La structure de la chaîne de montage est appelée la *topologie* du système parallèle. Si la chaîne de montage est linéaire, on dit que c'est un système parallèle *pipeliné*. Il fonctionne bien si la chaîne est alimentée de façon continue et si toutes les opérations durent le même temps. Si chaque ouvrier réalise la même tâche au même moment sur des pièces différentes, on dit que c'est un système parallèle *SIMD* (Single Instruction Multiple Data). Si au contraire, les ouvriers exécutent des tâches différentes au même moment on parle de système parallèle *MIMD* (Multiple Instruction Multiple Data). Si les ouvriers font globalement le même travail, mais qu'ils n'exécutent pas la même tâche au même moment, on parle de parallélisme *SPMD* (Single Program Multiple Data – c'est un cas particulier de MIMD). En général un système pipeliné est MIMD. La communication entre les ouvriers se fait soit par partage d'une zone de stockage où l'un dépose les pièces que l'autre prend, soit par envoi de lots de pièces. En informatique, on parlera respectivement de *mémoire partagée* ou d'envoi de *messages*.

Dans une grande usine, les divers types d'organisation du travail décrits ci-dessus coexistent et il en est de même dans les systèmes informatiques. Comme l'organisation des tâches est cruciale pour la bonne marche d'une entreprise, elle l'est pour tout ordinateur mettant en jeu 10, 100, voire 1000 ou plus, processeurs de calcul. *Cet ouvrage est une présentation des fondements nécessaires à la maîtrise de cette organisation.* Les outils de cette organisation s'appellent *coopération* et *communication*. Le problème essentiel en est la *concurrence*.

Les termes *systèmes parallèles* et *systèmes communicants* ont une signification qui est déterminée par l'usage. Dans cet ouvrage, on parlera de *parallélisme* de façon générique pour indiquer que des activités se déroulent en même temps et de *programmation parallèle* pour désigner les techniques de programmation associées à cette idée de parallélisme. C'est ainsi que doit être compris le titre de cet ouvrage.

Depuis une vingtaine d'années déjà, les ordinateurs dits séquentiels conçus pour être partagés entre plusieurs utilisateurs possèdent une unité de calcul et des unités d'entrée-sortie qui coopèrent, communiquent et sont en concurrence. Ils présentent donc une forme de parallélisme, même si celui-ci est réduit à un nombre faible d'unités. Plus récemment, les progrès des technologies de transmission de données (réseaux Transpac et Ethernet dans les années 70, réseaux à base de fibre optique dans les années 80, réseaux utilisant des technologies très rapides de transmission de données, de sons et d'images – réseaux FDDI ou ATM, bus optiques, réseau Internet à travers le monde – dans les années 90) ont rendu aisée la communication entre des ordinateurs distants. Les ordinateurs furent d'abord sommairement interconnectés pour échanger des messages puis des fichiers. Aujourd'hui on envisage un mode de fonctionnement plus intégré dans lequel une application peut s'exécuter sur plusieurs machines physiquement distantes (de quelques mètres à des milliers de kilomètres) et connectées : on parle d'applications réparties. Enfin, la réalisation des systèmes parallèles a émergé dans les années 1980 avec la baisse du coût des microprocesseurs et leur intégration plus poussée : le but est d'installer dans une même armoire plusieurs processeurs et d'utiliser ceux-ci pour toute application qui nécessite une grande puissance de calcul. Actuellement, les machines parallèles ont de quelques processeurs (moins de 10) jusqu'à quelques centaines (on parle de parallélisme massif). Elles permettent d'atteindre des puissances de calcul proches du TeraFlops (10^{12} opérations par seconde). A horizon 2007, grâce à l'évolution des technologies, les machines parallèles auront quelques milliers de processeurs, et permettront d'atteindre le PetaFlops (10^{15} opérations par seconde).

En conclusion, les systèmes répartis sont proposés pour la coopération de sites informatiques distants et les systèmes parallèles pour accroître la puissance de calcul. Bien sûr, toutes les combi-

naisons de systèmes parallèles et répartis sont possibles et voient le jour.

1.2 Pourquoi le parallélisme

L'analogie précédente avait pour but de faire sentir tout ce qu'on pouvait mettre sous le vocable traitement parallèle de l'information comparé à la production. Essayons maintenant de voir pourquoi le traitement de l'information peut nécessiter d'importantes structures analogues à des usines en production. Prenons le point de vue de l'ingénieur qui doit concevoir une solution informatique à un problème réel. Deux raisons motivent son choix d'une solution non séquentielle, c'est à dire parallèle ou répartie :

1. le problème informatique se pose de lui-même sous *une forme répartie* [AHV 83] : on parle de parallélisme de situation. Par exemple, la gestion d'une banque ayant plusieurs succursales exige un ordinateur sur chaque site pour le traitement des opérations locales mais aussi une coordination informatisée entre les sites pour le calcul de bilans globaux. On sait bien que les premières solutions informatisées pour la gestion, qui fonctionnaient sur une machine centrale et des terminaux distants, posaient des problèmes terribles d'engorgement d'accès aux heures d'affluence et de paralysie quand la machine avait une défaillance. Les solutions modernes font appel à l'informatique répartie.

2. le problème à résoudre est très gros, ou (et) sa résolution est trop longue par rapport aux temps de réponse attendus dans l'application [BT 89]. L'idée est d'utiliser de nombreux processeurs qui coopèrent afin de gagner en *rapidité*. De plus, si ces processeurs possèdent une mémoire propre, *la capacité de stockage globale est augmentée*. De nombreuses applications de l'informatique gagneraient à pouvoir calculer *plus rapidement* des problèmes *plus gros* : le calcul numérique et formel dans les applications industrielles (transport, nucléaire), le traitement de grosses bases d'information pour l'aide à la décision (diagnostic médical, systèmes financiers), la perception informatisée (traitement des images, de la parole), la résolution de problèmes difficiles de recherche opérationnelle (trafic aérien, gestion de production). Un exemple type est la prévision météorologique : les équations à résoudre ou les modèles à simuler à partir des données délivrées par les capteurs sont très complexes. Le météorologiste est pris entre deux types de contraintes :

   - d'une part, il veut pouvoir intégrer beaucoup de données dans des modèles complexes,
   - d'autre part, son temps de calcul est limité par la dernière heure de relevé de mesures et par l'heure du prochain bulletin.

   Dans l'état actuel des choses, pour être à même de calculer une solution avant que la prévision ne soit obsolète, ces modèles et équations doivent être simplifiés pour tenir compte de la puissance de calcul disponible. La prévision météorologique tirerait un bénéfice certain de moyens de calcul plus puissants en capacité de stockage et de calcul, apport qui peut lui être fourni par les machines parallèles.

Le parallélisme n'est pas le seul moyen qu'ait l'informatique aujourd'hui pour accroître les vitesses de calculs. Les autres solutions sont l'intégration plus poussée des circuits, la conception de circuits plus performants (dans l'accroissement de performance des circuits on retrouve le facteur

parallélisme dans la largeur des chemins de données ou des mots d'instruction), l'intégration de composants optiques, des modèles de calcul efficaces, etc. Ces avancées technologiques ne sont pas en contradiction avec l'idée de processeurs coopérants, car ceux-ci peuvent intégrer ces dernières avancées technologiques et y ajouter la puissance du parallélisme.

Bien entendu, si le problème à résoudre n'est pas gros, que l'utilisateur n'est pas pressé, et que le résultat n'a pas à être disponible sur plusieurs sites, il est préférable d'utiliser pour l'instant la machine dite "séquentielle" qui se trouve à proximité! En conclusion, coopération d'ordinateurs distants et taille des problèmes sont les motivations essentielles pour ce que nous conviendrons d'englober sous le terme de *système informatique parallèle.*

1.3 Organisation du cours

L'objectif de ce cours est de comprendre les concepts fondamentaux de la programmation parallèle et coopérative, indépendamment d'une architecture d'ordinateur. Certains argumenteront que c'est tâche impossible, mais si l'on se ramène à une analogie avec les langages séquentiels, on ne se préoccupe plus maintenant de la taille et du nombre de registres disponibles dans une unité de calcul pour concevoir un programme en ADA ou en C++. De même, on va essayer de ne pas se préoccuper du type de mémoire (partagée ou répartie) ou de la structure du réseau d'interconnexion, pour élaborer les concepts fondamentaux de la programmation parallèle. Par contre, la mise en oeuvre (ou l'implantation) efficace de ces concepts devra prendre en compte de façon très précise les caractéristiques de l'architecture.

Il pourrait apparaître une scission dans cet ouvrage entre l'étude de l'algorithmique parallèle applicative et l'étude de l'algorithmique parallèle des systèmes. Tout d'abord il faut être conscient que cette même scission existe en séquentiel : notons, par exemple, que tout système d'exploitation d'ordinateur séquentiel fonctionnant en mode multiprogrammé doit résoudre des problèmes de gestion de processus parallèles, sur une architecture matérielle particulière qui ne permet qu'une simulation du parallélisme en partageant le processeur entre les processus, alors que les utilisateurs conçoivent des programmes séquentiels.

Un point de vue unificateur (donc un peu simpliste) est celui du développeur de programme parallèle applicatif sur une architecture parallèle. Si sa machine dispose d'un langage de programmation parallèle performant et adapté à ses besoins il n'a à se préoccuper que de programmer son application dans le langage de haut niveau. Les problèmes de gestion des ressources, de coopération entre processus sont pris en charge par le compilateur de son langage et l'environnement d'exécution (i.e. tous les outils de partage dynamique de ressources) de sa machine parallèle. Si le langage utilisé n'est pas aussi performant qu'il le désire, le programmeur devra peut-être implanter lui même ses mécanismes de coopération. Il devra alors concevoir les deux niveaux de son programme : le niveau applicatif et le niveau de mise en oeuvre de la collaboration. De son côté, le concepteur du compilateur et de l'environnement d'exécution doit s'intéresser à la mise en oeuvre de la coopération, donc à la synchronisation et à la communication de processus.

On se limitera à une vue opérationnelle, basée sur des langages impératifs. Cette vue opérationnelle n'est pas forcément celle qui permet la meilleure abstraction ([Ban 90] et [CM 88]) du parallélisme, mais elle correspond aux concepts utilisés dans les langages existants (par exemple

ADA) et largement diffusés. On ne s'intéressera ici qu'aux langages dans lesquels le parallélisme est exprimé explicitement, par opposition à ceux où les compilateurs doivent détecter le parallélisme implicite dans les commandes du langage (par exemple HPF Fortran ou Prolog).

Ce cours présente donc les éléments qui permettent d'appréhender la programmation des systèmes parallèles et le plan en est le suivant :

- Dans le chapitre 2, nous introduisons les principes de base du parallélisme et présentons les quelques problèmes types qui seront repris en détail dans les chapitres suivants.
- Notre choix s'est porté sur le langage ADA pour exprimer les algorithmes que nous présentons dans ce cours. Le chapitre 3 décrit les opérateurs de ce langage qui permettent d'exprimer le parallélisme.
- Le chapitre 4 intitulé "Fondements de l'Algorithmique Parallèle", présente les concepts qui permettent de d'élaborer une solution parallèle à un problème donné.
- Le chapitre 5 montre de quelle manière l'analyse d'un problème doit être conduite pour mener à une solution algorithmique très parallèle.
- La mise en œuvre de la communication nécessite la connaissance de notions fondamentales de synchronisation et de la concurrence. Elles sont présentées dans le chapitre 6, à travers un mécanisme de gestion de la mémoire, l'exemple du "producteur-consommateur".
- Le chapitre 7, est consacré à l'algorithmique distribuée. Des principes fondamentaux d'algorithmique distribuée sont présentés et illustrés à travers la gestion de la cohérence de copies multiples.
- Dans le chapitre 8, nous proposons des modèles de représentation des systèmes parallèles communicants. Ces modèles ont pour objectif de spécifier leur comportement et de les valider ou de les évaluer.
- Dans le chapitre 9, nous étudions plus précisément les problèmes d'interblocage et d'équité liés à la synchronisation, à travers l'exemple du lecteur-rédacteur. Nous montrons en particulier comment les détecter sur les modèles présentés au chapitre précédent.
- Le chapitre **??** montre l'utilisation des différentes techniques présentées dans cet ouvrage pour la construction d'une application parallèle sur une architecture distribuée.
- En annexe A, nous présentons une brève introduction à la partie séquentielle du langage ADA.

Bibliographie

[**CM 88**] Parallel Program Design, Chandy & Misra, Addison Wesley, 1988.

[**AHV 83**] Synchronisation de programmes parallèles, F. André, D. Herman & J. P. Verjus, Dunod 1983.

[**BT 89**] Parallel and Distributed Computation, D. Bertsekas & J. Tsitsiklis, Prentice Hall, 1989.

[**Ban 90**] La programmation parallèle, J. P. Banatre, INRIA Didactique 1990.

Chapitre 2

Principes de Base

Programmer dans un langage *séquentiel* impératif classique, c'est maîtriser la correction et la complexité d'*une* séquence d'instructions. Cette séquence d'instructions est celle qui est exécutée par le processeur de calcul. Programmer dans un langage *parallèle* impératif, c'est maîtriser la correction et la complexité de *plusieurs* séquences d'instructions. Chaque séquence d'instructions est exécutée sur un processeur de calcul, et les processeurs fonctionnent en parallèle ou en mode multiprogrammé (i.e. plusieurs programmes séquentiels se partagent le même processeur). Au niveau de la pratique de la programmation, au lieu d'écrire un programme séquentiel pour un processeur, il faut d'une part écrire un programme séquentiel pour chaque processeur et exprimer la façon dont ils interagissent. L'ensemble des programmes séquentiels et des interactions s'appelle un programme parallèle. Les langages parallèles existants fournissent des instructions adéquates à ce type de description : elles permettent de spécifier le nombre de séquences d'instructions, ce que fait chacune, comment elles échangent de l'information et comment elles se coordonnent. On dit que le parallélisme est explicite dans le langage. On ne parle pas ici de la programmation utilisant des instructions vectorielles (c'est à dire des instructions qui font des opérations sur des vecteurs et qui s'exécutent sur des processeurs spécialisés, dit processeurs vectoriels). Il est clair cependant que parallélisme et utilisation d'instructions vectorielles peuvent être utilement combinées.

Dans la suite, nous donnons le vocabulaire et les idées qui nous semblent fondamentaux pour la compréhension de la programmation parallèle.

2.1 Un exemple didactique

Pour commencer, décrivons un exemple qui nous servira d'illustration par la suite : pour maîtriser un incendie, on dispose d'un robinet à eau, d'un nombre illimité de robots-pompiers (ce sont nos processeurs séquentiels), nommés $R[i]$, $i = 1, 2, 3, \ldots$ et d'un seau par pompier (les zones de stockage). Les pompiers sont capables de faire les actions suivantes :

- *remplir*: remplir un seau au robinet.
- *échanger-avec-R*: échanger un seau avec le robot R.

- *déverser*: verser un seau sur le foyer.
- *marcher*: aller du robinet au foyer ou du foyer au robinet.

Sur ces actions, les contraintes suivantes sont imposées : un pompier ne peut posséder qu'un seul seau à la fois ; il ne peut faire qu'une seule action à la fois ; à un moment donné un seul pompier peut utiliser le robinet pour remplir son seau et l'échange de seau ne peut se produire que si les deux pompiers concernés sont prêts à l'échange. Par contre l'action de marcher s'effectue sans contrainte, en parallèle (pas de collisions de pompiers) ainsi que l'action de verser sur le foyer. Au départ les pompiers disposent chacun d'un seau vide.

Le problème est de programmer le fonctionnement des pompiers afin qu'ils éteignent le foyer. On ne cherchera pas à programmer dans un premier temps l'arrêt des pompiers lorsque le feu est éteint. On se propose uniquement de déterminer l'algorithme exécuté par chaque pompier et leur coordination. Pour exprimer l'algorithme, des outils de programmation parallèle sont nécessaires.

2.1.1 Quel algorithme?

Il y a deux façons extrêmes de programmer les pompiers qui correspondent respectivement au cadre de coopération lâche des systèmes communicants et au cadre de coopération étroite des systèmes parallèles : soit chaque pompier remplit un seau, marche jusqu'au foyer, déverse son seau et revient au robinet, ou bien les pompiers font la chaîne jusqu'au foyer. On n'envisagera pas ici les solutions intermédiaires où la longueur de la chaîne nécessite que les pompiers marchent entre les échanges (de seau), ni celles où l'on mettrait plusieurs chaînes.

Solution 1 : autonomie des pompiers. Chaque pompier remplit un seau, marche jusqu'au foyer, déverse son seau et revient au robinet. Les pompiers sont donc des unités autonomes qui remplissent tous une tâche identique et qui doivent se *coordonner* (on dit se *synchroniser*) pour l'accès au robinet. Les pompiers forment un système d'entités *communicantes*, qui fonctionne en mode SPMD (on rappelle que ce sigle veut dire Single Program Multiple Data). L'efficacité de cette organisation du travail dépend des vitesses relatives des actions marcher, remplir et verser et du nombre de pompiers. Au départ, les pompiers attendent tous pour remplir leur seau. Si le temps nécessaire pour le remplissage de $P-1$ seaux est supérieur au temps nécessaire pour aller jusqu'au foyer, déverser et revenir, le premier pompier qui a rempli son seau sera revenu avant que le dernier soit parti. Il n'est donc pas utile d'avoir plus de P robots pompiers en parallèle. On dira alors qu'il y a un *goulot d'étranglement* pour l'accès au robinet. Ce goulot d'étranglement vient du fait que le robinet ne peut être utilisé que par un pompier à la fois : on dit que le robinet est une ressource dont l'usage est *critique* car il doit se faire en *exclusion mutuelle*. Il s'agit d'ajuster P de façon à ce que P pompiers aillent exactement P fois plus vite que 1 pompier : on dit que l'on a une *accélération d'ordre P*.

Solution 2 : la chaîne de pompiers. On prend un nombre de pompiers P tel qu'une chaîne puisse être construite du robinet au foyer, sans que les pompiers aient besoin de marcher ni que la chaîne fasse des zigzags (s'il y a un zigzag, cela veut dire que la chaîne de pompiers comprend trop d'individus pour le trajet à parcourir). En début de chaîne un pompier remplit les seaux, en bout

de chaîne un autre pompier les déverse sur le foyer et entre les deux, les pompiers échangent seaux vides contre seaux pleins. Cette solution de la chaîne n'est pas une généralisation de la solution séquentielle (1 pompier), mais dérive d'une idée différente : le parallélisme pipeline. La vitesse de la chaîne est déterminée par la plus longue des actions remplir, échanger, déverser. L'accélération que l'on obtient par rapport à l'utilisation d'un seul pompier dépend des vitesses relatives des actions échanger et marcher. Si un aller et retour du pompier dure aussi longtemps que $P-1$ échanges et que le remplissage est une action plus courte que l'échange, alors l'accélération sera d'ordre P. En effet, dans ce cas, $P-1$ échanges permettront de déverser P seaux sur le foyer dans le même temps qu'un pompier unique aurait versé son seau. Le temps de remplissage n'est alors pas déterminant.

Quelles remarques amène cet exemple ? D'abord, nous sommes contraints de construire une solution parallèle, puisqu'on dispose de plusieurs robots et non d'un seul. Ensuite, étant donné les actions que savent faire les robots, deux solutions radicalement distinctes (plus les solutions intermédiaires éventuellement) se présentent, c'est à dire deux *algorithmes* de résolution distincts : celui des pompiers autonomes en parallèle et celui de la chaîne de pompiers. Notre choix final sera déterminé par la meilleure accélération que l'on pourra obtenir de l'une ou l'autre des solutions, mais aussi par la fiabilité de la solution choisie. On voit que la solution de la chaîne est vulnérable à la défaillance d'un pompier (e.g. s'il laisse tomber chaque seau qu'il reçoit, alors que l'autre solution fonctionne aussi longtemps qu'il existe un pompier en état de fonctionnement (on parle de mode de fonctionnement dégradé).

2.1.2 Quel programme?

Une fois l'algorithme choisi, il s'agit de le programmer. Supposons que l'on ait choisi la solution 2 avec $P = 3$ et que nous ne disposions que d'un langage séquentiel pour commander le système parallèle composé des P robots. Dans ce langage, chaque action est préfixée par le nom du robot qui l'exécute. Voici un exemple de programme qui implante la solution 2, tous les seaux étant vides au départ :

Programme 1 :
R[1].remplir
R[1].échanger-avec-R[2]
Pour toujours
R[1].remplir
R[2].échanger-avec-R[3]
R[3].déverser
R[3].échanger-avec-R[2]
fin pour

Cette solution amène plusieurs remarques : on n'a pas pu exprimer que certaines actions peuvent se passer en parallèle (puisque ce langage est séquentiel), comme `R[1].remplir` et `R[2].echanger-avec-R[3]` dans la boucle, mais il a fallu leur imposer un ordre arbitraire. Pour écrire cette solution, on a raisonné sur l'état des pompiers (quels pompiers possèdent un seau vide ou plein ?) pour déterminer l'action suivante qu'ils peuvent exécuter. La généralisation à P pompiers ne sera pas immédiate à cause de cela. En effet, pour P pompiers, le nombre d'états possibles des seaux est 2^P et le

raisonnement devient rapidement difficile. Si l'état initial des seaux est différent, il faut réécrire ce programme. Il apparaît clairement à travers cet exemple d'école qu'un langage séquentiel n'est pas adapté pour la programmation de systèmes parallèles.

Une solution parallèle prend le point de vue de chaque pompier, chacun exécutant un programme séquentiel décrivant actions et coordination. La coordination se fait uniquement par les instructions d'échange. Le programme parallèle, composé de P programmes séquentiels peut se présenter ainsi :

```
Programme 2 :
  Exécuter en parallèle tous les R[i] pour i = 1..P :
  R[1] :
    Pour toujours
      R[1].remplir
      R[1].échanger-avec-R[2]
    fin pour
  R[i] : i = 2 .. P-1
    Pour toujours
      R[i].échanger-avec-R[i-1]
      R[i].échanger-avec-R[i+1]
    fin pour
  R[P] :
    Pour toujours
      R[P].échanger-avec-R[P-1]
      R[P].déverser
```

Cette solution est générale pour P pompiers et n'impose que le minimum de synchronisation au niveau des instructions d'échange entre voisins : le seul ordre imposé entre les actions des pompiers est celui relatif aux actions *échanger*. La programmation de la commande des robots est assez simple et intuitive. Cependant, il n'est pas plus aisé que précédemment d'appréhender l'évolution de l'état de tous les seaux de pompiers, car l'espace des états possibles est toujours 2^P. Si les pompiers ne travaillent pas tous à la même vitesse, il va être difficile de se représenter l'évolution de la chaîne dans son ensemble. Prouver que la programmation est juste (par exemple que les pompiers ne vont pas échanger deux seaux pleins ou deux seaux vides) nécessite une vérification formelle (voir chapitre 8).

Remarque. Si on ajoute la contrainte (extérieure au programme, mais imposée par l'environnement matériel par exemple) que les robots ne peuvent exécuter leurs actions que de façon globalement séquentielle (c'est à dire que l'on n'autorise qu'un seul pompier à agir à un instant donné), on retrouve alors le comportement décrit par le programme séquentiel *Programme 1*, mais également toutes les exécutions séquentielles possibles de ce type. Ceci illustre le fait qu'un programme parallèle peut engendrer des exécutions différentes (en particulier des exécutions séquentielles), suivant les contraintes de l'exécution sur la machine. Cependant toutes ces exécutions répondent au problème si le programme est juste.

En conclusion, identifier les critères de choix d'un algorithme, savoir le programmer et en connaître les techniques de vérification sont les objectifs de cet ouvrage.

2.1.3 L'efficacité

Le choix d'un algorithme est essentiellement fondé sur des critères de rapidité (d'autres critères peuvent intervenir, par exemple la sécurité, très importante dans certaines applications). On peut se donner trois types de mesure de cette rapidité :

- La première est *expérimentale* et consiste à comparer le temps d'exécution d'un algorithme séquentiel sur un processeur d'une machine parallèle et le temps d'exécution d'un algorithme parallèle sur cette même machine parallèle. Le rapport du temps séquentiel sur le temps parallèle donne ce qu'on appelle l'*accélération expérimentale* obtenue par l'implémentation de l'algorithme parallèle. On peut expérimenter ainsi l'accélération en fonction du nombre de processeurs utilisés. On appelle *degré de parallélisme* d'un algorithme le nombre maximal de processeurs qu'il peut utiliser (bien qu'on restreigne parfois son sens au nombre de processeurs effectivement utilisés). Pour un même problème, les différents algorithmes parallèles peuvent se comparer par leur courbe d'accélération expérimentale. Le problème est bien parallélisable quand la courbe est une fonction linéaire du degré de parallélisme. Cette méthode a le désavantage – lorsque l'on veut comparer des algorithmes – de donner une mesure combinée de la qualité de l'algorithme, de sa programmation, et de l'implantation du programme sur la machine.

- La seconde méthode est *théorique* et nécessite un modèle de calcul parallèle, comme le modèle RAM (Random Access Machine) utilisé en algorithmique séquentielle pour mesurer la complexité d'un algorithme et lui attribuer une classe. Le modèle le plus employé est le modèle *PRAM* (Parallel Random Access Machine). Schématiquement, dans ce modèle, les opérations élémentaires durent le même temps (typiquement, l'action élémentaire est de durée 1) et effectuent un calcul sur un nombre borné d'entrées. On se donne virtuellement un nombre non borné de processeurs. La question est alors de savoir pour quel nombre minimum de processeurs le temps de résolution minimum d'un problème de taille n est atteint? La notion de taille d'un problème ne dépend pas du type de programmation (séquentielle ou parallèle). C'est par exemple le nombre d'éléments pour un tri, la taille du vecteur pour un produit scalaire, etc... Pour mesurer la complexité, deux quantités sont généralement considérées : le nombre de processeurs utilisés et le temps nécessaire à la résolution du problème. Le meilleur algorithme est celui qui permet d'obtenir un temps minimum de résolution, et parmi les algorithmes qui dépensent ce temps minimum, celui qui nécessite un nombre minimum de processeurs (il arrive que ce meilleur algorithme nécessite un seul processeur). L'évaluation de ces minima peut être non triviale.

  Le problème de la calculabilité parallèle ne se pose pas différemment de celui de la calculabilité séquentielle. En effet, la notion théorique de fonction calculable introduite par Turing n'est pas liée à un modèle de calcul. Le fait de considérer un modèle de calcul parallèle n'augmente pas l'ensemble des fonctions que l'on peut théoriquement calculer. En particulier on peut montrer que tout problème qui peut se calculer en parallèle admet au moins une résolution séquentielle.

- La troisième méthode est aussi *théorique* mais d'une nature différente et poursuit des objectifs différents : les durées des opérations ou actions élémentaires ne sont pas prévisibles (contrairement au modèle précédent), mais varient autour d'une valeur moyenne suivant une loi de probabilité. Cette variabilité des durées induit des attentes entre les processus qui coopèrent

et échangent des données. Les *performances* de la coopération entre les processeurs sont recherchées en réduisant l'attente des processus (contrairement à la deuxième méthode, on ne compte pas des opérations élémentaires). Sous des hypothèses probabilistes (par exemple des hypothèses de Markov), le calcul de cette attente est possible. Par exemple, dans la solution 1 des pompiers, ce type de méthode permettrait d'évaluer l'accélération produite par l'apport d'un deuxième robinet d'approvisionnement qui réduirait l'attente pour l'accès au robinet, dans le cas où celui-ci est un goulot d'étranglement.

L'évaluation des performances de la coopération de processus ou le calcul de complexité sont deux mesures complémentaires de l'efficacité des algorithmes parallèles que nous aborderons respectivement dans les chapitres 4 et 8. Le premier point de vue est abandonné à cause de son aspect essentiellement pratique et aussi parce qu'il mesure non seulement la qualité d'un programme, mais aussi la qualité de son implantation sur la machine, sujet que nous n'abordons pas ici.

2.2 Système parallèle et processus

2.2.1 La notion de processus

Commençons par souligner la différence entre un programme qui est une description d'un ensemble d'actions et une exécution de ce programme qui est une réalisation particulière de ces actions : si le programme est écrit dans un langage séquentiel son exécution sera une suite d'actions appelée *processus*. Un programme est une entité statique, un processus est une entité dynamique : par exemple, si l'on compare l'un à une recette de cuisine, l'autre est une confection du plat. Prenons un programme séquentiel comportant un branchement conditionnel du type *si condition alors action_1 sinon action_2*. Tout processus d'exécution de ce programme comporte soit *action_1* soit *action_2*, alors que les deux apparaissent dans le programme. On voit bien l'intérêt de différencier les deux notions. Dans l'exemple précédent, *Programme 1* et *Programme 2* sont des programmes. L'exécution de *Programme 1* est un processus (c'est un programme séquentiel). L'exécution de *Programme 2* est l'ensemble de P processus, exécutions des P programmes séquentiels.

Historiquement, l'utilité de la notion de processus est apparue avec les systèmes d'exploitation, et le partage par plusieurs processus (chacun étant par exemple l'exécution d'un même programme séquentiel) d'une seule unité de calcul. Il s'est avéré utile de nommer et de représenter chacune de ces différentes exécutions pour décrire leur façon de partager l'unité centrale.

Un processus est modélisé par une *suite d'états-actions*. Un *état* du processus est le résultat de son exécution antérieure (état des registres, de la pile, etc...). Le passage d'un état à un autre se fait par une *action*. Dans cette représentation, une action est *indivisible*. Par indivisible, on veut dire que soit elle s'exécute entièrement, soit pas du tout, ou encore qu'elle n'admet pas d'interruption. On dit encore que les actions sont *atomiques*. Cette atomicité est souvent offerte au niveau du langage machine. Un *événement* est le début ou la fin d'une action. Un événement est aussi le début ou la fin d'un changement d'état. Un processus n'est *observable* qu'avant ou après une action. Une action a une durée, alors qu'un événement est de durée nulle. La notion de suite d'états-actions fait bien sûr référence au temps : les actions du processus doivent se faire dans un ordre temporel déterminé par le programme séquentiel qui décrit ce processus (voir en particulier, sur ce thème, le chapitre

8).

Un processus est donc un modèle : selon ce que l'on veut étudier, on peut se placer à des niveaux d'observation différents. A un niveau donné, certaines actions sont indivisibles, mais peuvent se décomposer en d'autres actions dans un modèle plus fin.

2.2.2 Etat d'un système parallèle

Un système parallèle peut donc être caractérisé à un instant donné par les états de chacun de ses processus à cet instant, l'état d'un processus donné ne pouvant être modifié qu'après la fin d'une action et avant le début de l'action suivante. L'évolution du système peut donc être visualisée par une séquence de "photographies", dans laquelle chaque photographie représente l'état de tous les processus ; le changement d'état du système correspond au passage d'une photographie à la suivante. Les *variables d'état* permettent de caractériser l'état du système à un instant : le changement d'état du système correspond alors à une modification de ces variables d'état (par une procédure).

Illustrons ces notions avec l'exemple des pompiers :

- l'état du processus (pompier) est caractérisé par l'état de la zone de stockage (le seau) : *Vide* ou *Plein*. Il est clair que cet état est parfaitement défini après ou avant chacune des actions *remplir*, *échanger* ou *déverser* et que seules ces actions le modifient. Entre les actions, le pompier est observable. En pleine action, il est impossible de savoir de façon sûre son état. Cet état peut être représenté par une *variable d'état état_de_mon_seau*, qui ne peut prendre que les deux valeurs *vide* ou *plein*.
- Dire que l'action *remplir* est atomique, c'est dire que les actions élémentaires *ouvrir-robinet*, *attendre-le-remplissage*, *fermer-le-robinet* sont indissociables. Si tel n'était pas le cas, dans la solution 1 (autonomie des pompiers), il se pourrait qu'un pompier ouvre le robinet et que l'un de ceux qui attendent pour remplir leur seau lui dérobe le robinet ouvert et remplisse son seau. Pour éviter ce type de comportement désagréable dans une tâche coopérative, l'atomicité des actions est imposée. Cette atomicité peut être assurée par la machine. Par exemple, le masquage d'interruption dans les processeurs classiques rend en général les opérations de transfert mémoire-registre et certaines autres instructions du langage machine atomiques. A l'inverse, dans un langage évolué de type Pascal, les instructions de type `c := a+b` ne sont pas atomiques. Cette instruction peut être interrompue entre le chargement de `a` et `b` et l'écriture de `c` en mémoire. Si lors de l'interruption, l'une des valeurs `a` ou `b` est modifiée, le résultat de l'instruction peut être incohérent. Ce problème de l'atomicité des affectations de variables sera traité en détail ultérieurement (chapitre 6).
- Dans *Programme 2*, le processus $R[1]$ est décrit par la suite des états-actions :

  `Vide-remplir ;  Plein-échanger ;  Vide-remplir ;  etc`

  Ce qui prend la représentation graphique de la figure 2.1.

Décrire les processus est essentiellement intéressant pour décrire les relations qu'ils ont entre eux. Un processus possède une relation d'ordre total sur son ensemble d'actions. Si plusieurs processus

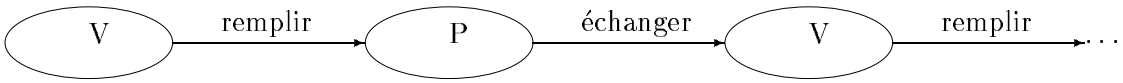


FIG. 2.1 – *Description du processus du premier pompier.*

coopèrent à la réalisation d'un même objectif, leur coopération induit de nouvelles relations d'ordre entre les actions de ces processus. Par exemple,

- s'il y a partage de ressource : dans la solution 1 du problème des pompiers, les robots qui remplissent leur seau ne peuvent le faire qu'en séquence. Ceci équivaut à dire qu'il existe un ordre (temporel) total entre les diverses actions *remplir* de chacun des robots. Cet ordre total peut être défini par exemple par une discipline d'accès au robinet (comme premier arrivé premier servi).
- s'il y a rendez-vous entre deux processus : un rendez-vous survient lors de l'exécution de l'instruction *échanger* des robots. Si l'un des deux pompiers n'est pas au rendez-vous, l'autre attend. Une fois le rendez-vous effectué (ce rendez-vous est une action commune des deux processus), chaque pompier passe à l'action suivante. Ainsi, le pompier $R[1]$ du programme *Programme 2* ne peut remplir le seau suivant qu'après avoir échangé le précédent, donc après que $R[2]$ ait échangé avec $R[3]$, etc.
- s'il y a création d'un nouveau processus. Ceci est une situation qui n'a pas été envisagée dans l'exemple des robots pompiers, mais qui aurait pu l'être. En effet, dans la solution de la chaîne des pompiers, cette chaîne aurait pu être mise en place par un algorithme du type : *tant qu'il y a la place entre le foyer et moi, inviter un nouveau robot pompier*. Le nouveau participant ne commence bien sûr son travail qu'après son invitation.

2.2.3 Relations entre processus

Pour étudier les relations entre processus, deux modèles du temps sont généralement utilisés : le modèle *synchrone* ou le modèle *asynchrone*. Le cadre synchrone est utilisé lorsque l'hypothèse que toutes les actions des processus ont la même durée (ou bien ont des durées multiples d'une durée unité) est justifiée. Pratiquement, une échelle de temps discrète est définie et à chaque top d'horloge toutes les actions possibles de différents processus se réalisent simultanément si elles doivent se réaliser, sinon pour certaines, ce sera au top d'horloge suivant, etc. A l'opposé, un cadre asynchrone est justifié lorsque les durées des actions ne sont pas multiples d'une même quantité. L'échelle de temps est alors un espace continu et dense (les réels en général) et ce cadre est accompagné de l'hypothèse qu'à un instant donné de cette échelle de temps dense, un seul événement de fin d'action est possible. Les actions (ou plus précisément les fins d'actions) sont *sérialisées* dans une exécution. Les durées sont inconnues, sinon imprévisibles : la représentation des processus doit tenir compte de tous les cas possibles de sérialisation.

Dans les deux cas de modèle de temps, pour représenter les exécutions possibles des processus, on utilisera une représentation de graphe orienté. L'interprétation de ces graphes est immédiate :

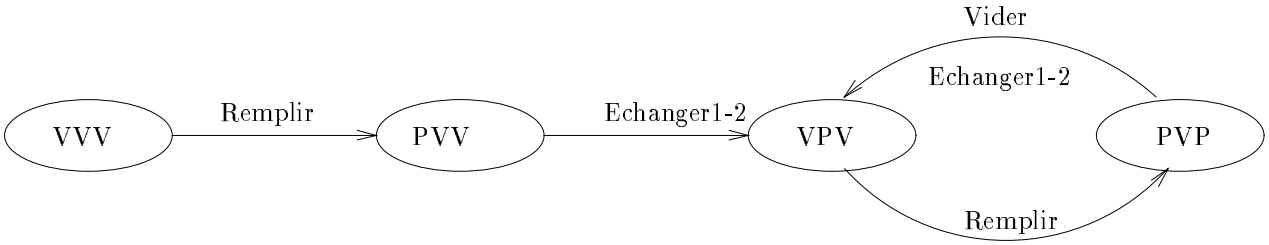


FIG. 2.2 – *Evolution de l'état de 3 pompiers dans une chaîne dans le cas d'une hypothèse synchrone où toutes les actions*

les exécutions possibles sont tous les cheminements possibles sur ce graphe. L'état *(P V V)* signifie que le premier pompier a un seau plein, les deuxième et troisième un seau vide.

Pour fixer les idées, construisons ces graphes pour l'exemple des pompiers, lorsque le programme ne se termine jamais : considérons la chaîne des pompiers dans le cadre d'une hypothèse synchrone où toutes les actions ont exactement comme durée 1. Ceci est représenté par le graphe de la figure 2.2. Au départ tous les seaux sont vides, puis il y a une phase d'initialisation de la chaîne, à la suite de quoi un seau est déversé par unité de temps sur le foyer. On suppose que toute action possible est réalisée immédiatement. Plusieurs actions sont exécutées dans la même unité de temps.

Lorsque cette chaîne est envisagée sous une hypothèse asynchrone la figure 2.3 représente l'ensemble des évolutions possibles. Dans ce cadre, une seule action se termine à un instant donné et la durée d'une action est indéterminée. Le scénario suivant est alors possible : supposons que la chaîne soit dans un état tel que tous les pompiers ont un seau plein sauf celui qui accède au robinet. Si le remplissage du seau prend un temps très long (coupure de l'arrivée d'eau) la chaîne atteint un état où tous les pompiers ont un seau vide. Cet état ne serait pas *atteignable* dans une hypothèse synchrone puisqu'il n'y a pas de chemin de retour possible dans l'état *VVV*. Ces deux modes de fonctionnement, synchrone et asynchrone induisent des comportements observables différents.

La construction et l'étude de ces graphes nous aidera à maîtriser la construction d'un programme parallèle.

2.2.4 Le non déterminisme

On appelle non-déterminisme une situation dans laquelle plusieurs actions étant possibles, une seule se réalise, ce choix n'étant pas toujours le même. Il faut noter que l'instruction *if* n'a pas cette propriété : pour un même état, le test donne le même résultat et l'action choisie est la même. Le non déterminisme peut provenir soit d'une instruction de contrôle qui provoque un choix non déterminé a priori d'actions, soit de durées d'actions non prévisibles. Dans la solution 1 du problème des pompiers,

– si l'attribution du robinet se fait par un robot dédié qui choisit au hasard l'un des pompiers en attente, on dit que ce robot choisit de façon indéterministe l'un d'entre eux.

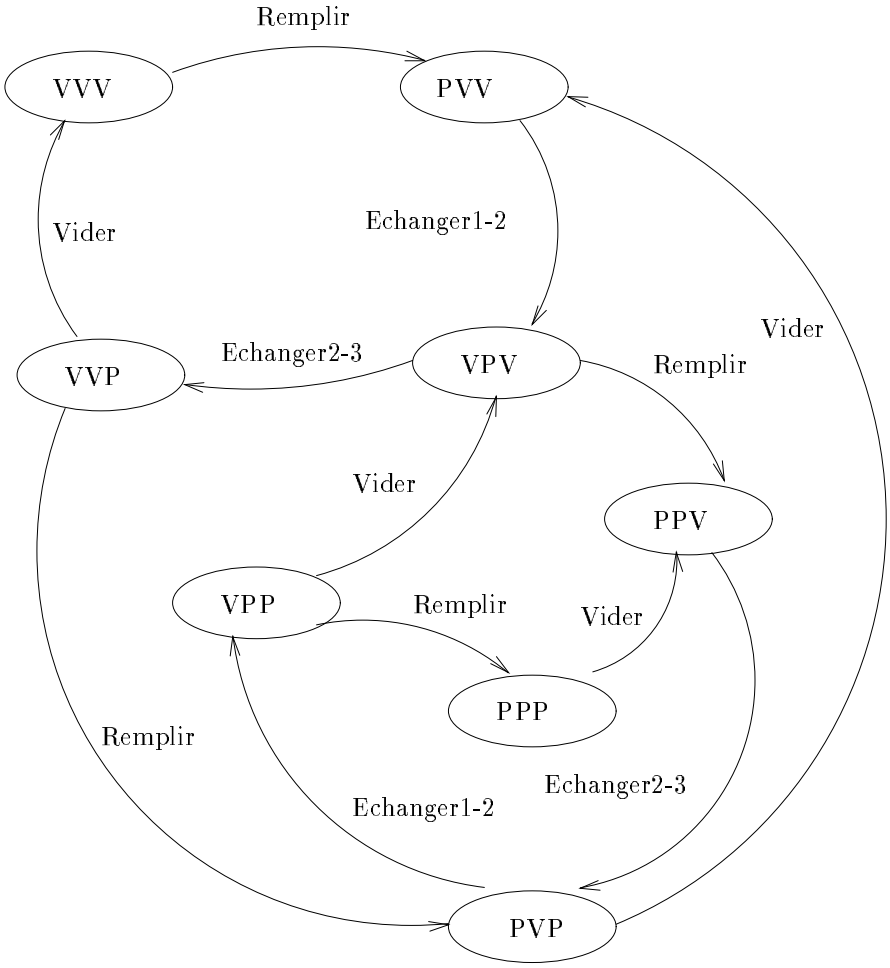


FIG. 2.3 – *Evolution de l'état de 3 pompiers dans une chaîne dans le cas d'une hypothèse asynchrone : un seule action de l'un des trois pompiers à la fois et des choix de parcours*

– si les durées des actions sont non prévisibles, l'identité des pompiers attendant la disponibilité du robinet ne peut pas être prévue à l'avance : la situation est dite indéterministe.

Pour un processus, faire face à des situations indéterministes, c'est savoir tester la situation courante et choisir de façon aléatoire une action parmi les alternatives possibles. Les langages séquentiels ne permettent pas de programmer de tels comportements. En effet, les tests d'alternatives (comme le *if* ou le *case*) sont programmés avec un certain ordre, et le premier test programmé et évalué à vrai est celui qui sera choisi à l'exécution, et non l'un quelconque parmi ceux qui sont vrais. La programmation parallèle donne cette dernière possibilité.

2.2.5 La synchronisation

La mise en parallèle de processus est l'exécution concurrente des actions qui les composent. Quand ces processus coopèrent à la même tâche ou bien partagent la même machine, ils sont soumis à des contraintes supplémentaires que l'on appelle contraintes de *synchronisation*. Les trois grandes causes de synchronisation sont :

– *le rendez-vous* : le rendez-vous est une action exécutable par plusieurs processus *simultanément*. Par exemple, l'échange des seaux des pompiers est une contrainte de synchronisation de type rendez-vous. Il faut (et il suffit) que tous les processus convoqués à ce rendez-vous soient parvenus à l'action de rendez-vous de leur programme pour que celui-ci se réalise. Si tel n'est pas le cas, les processus parvenus au rendez-vous sont bloqués (i. e. attendent). Le rendez-vous est une action atomique comme les autres.

– *l'exclusion mutuelle* : une ressource est dite utilisable en exclusion mutuelle si et seulement si un seul processus peut l'utiliser à la fois. Si un processus utilise la ressource et qu'un autre processus en fait la demande, ce dernier se met en attente (se bloque) jusqu'à ce que la ressource soit libérée. Cette attente se résout par l'existence d'une politique d'allocation, comme premier-arrivé-premier-servi. Le fait qu'il n'existe qu'un robinet utilisable par un pompier à la fois met cette ressource en exclusion mutuelle.

– *la section critique* : cette notion est l'analogue de l'exclusion mutuelle mais pour une suite d'actions. Une suite d'actions est dite en section critique si et seulement si un seul processus peut exécuter cette suite d'actions à la fois. Dans la solution 1, l'action de remplir est une section critique. Dans ce cas, le fait que la ressource soit en exclusion mutuelle est intimement lié au fait que les actions qui permettent d'utiliser la ressource sont en section critique.

Rappelons qu'un processus est formalisé par une suite d'actions et un état courant de son calcul qui résume son passé. Lorsque ce processus est soumis à des contraintes de synchronisation, on aura besoin de rajouter à l'état courant de son calcul, un information sur son statut : le processus est-il *bloqué* par une synchronisation ou au contraire *actif*? En plus le statut d'un processus peut être *pas-commencé* ou *terminé*.

2.2.6 Vérification et comportements pathologiques

La vérification de programmes séquentiels ou parallèles a un même objectif: prouver que les programmes se terminent (s'ils sont supposés se terminer) ou qu'ils ne peuvent pas se terminer (s'ils sont censés fonctionner indéfiniment) et qu'il rendent le résultat attendu ou qu'ils ont le comportement attendu. En plus des erreurs qui apparaissent de façon classique en programmation séquentielle, la programmation parallèle doit éviter en particulier deux écueils : les situations d'*interblocage* et de *famine*.

Lorsqu'à une synchronisation est associée une condition de franchissement, il arrive (plus souvent que l'on ne pense) des situations où le processus $R[1]$ attend qu'une condition soit réalisée par le processus $R[2]$ qui lui même attend $R[3]$, et ceci jusqu'au dernier processus $R[P]$ qui attend $R[1]$. La condition de franchissement ne se réalisera jamais puisqu'aucune action ne peut s'exécuter. L'exécution du programme est dite *bloquée* dans la mesure où elle est dans un état qu'il ne quittera jamais. En programmation parallèle, il est indispensable de pouvoir bloquer provisoirement un processus, mais il ne faut pas que cette possibilité amène à un blocage indéfiniment long de tous les processus, ce qu'on appelle une situation d'*interblocage*.

Voici un programme qui amène à une situation d'interblocage lorsque les pompiers s'amusent à faire une ronde avec les seaux :

Programme 2 :
Exécuter en parallèle tous les R[i] pour i = 1..P :
R[1] :
Pour toujours
R[1].échanger-avec-R[2]
R[1].échanger-avec-R[P]
fin pour
R[i] : i = 2 .. P-1
Pour toujours
R[i].échanger-avec-R[i+1]
R[i].échanger-avec-R[i-1]
fin pour
R[P] :
Pour toujours
R[P].échanger-avec-R[1]
R[P].échanger-avec-R[P-1]

L'interblocage vient de ce que $R[1]$ est prêt pour un rendez-vous avec $R[2]$, qui lui même attend $R[3]$, etc, et finalement $R[P]$ attend $R[1]$. Bien sûr, avec une simple inversion de deux instructions, la solution n'a plus d'interblocage.

Un processus est en état de *famine* s'il ne peut pas exécuter une certaine action et ne pourra jamais le faire. Dans la solution 1 des pompiers, supposons que la politique d'allocation du robinet aux pompiers soit de l'attribuer au pompier qui a le plus grand numéro parmi ceux qui sont en

attente. Alors un scénario *possible dans le cadre asynchrone* est que le pompier $R[1]$ ne soit jamais seul en attente du robinet. $R[1]$ n'obtiendra jamais le remplissage de son seau : il est en état de famine. Un programme parallèle dont une exécution mène à un état de famine pour un processus est en général à éviter.

2.3 Exemples de problèmes de la programmation parallèle

L'une des classification possible de l'algorithmique est la suivante. D'une part l'algorithmique que l'on nomme ici applicative parce que son objectif premier est de répondre à des domaines d'application qui posent des problèmes numériques ou plus généralement de traitement de données. Les *algorithmes applicatifs de base* sont les algorithmes numériques de l'algèbre linéaire, de la résolution d'équations ou bien les tris, recherches arborescentes, etc. D'autre part, on distinguera l'algorithmique des systèmes d'exploitation dont l'objectif est de rendre l'ordinateur utilisable de façon confortable, efficace et sûre pour la programmation applicative. Les *algorithmes systèmes de base* sont ceux qui permettent de partager une ressource en section critique ou de communiquer une donnée entre deux ou plusieurs processus (on retrouve la notion de processus communicants). Ces deux algorithmiques divergent nettement au niveau des objectifs : dans un cas il s'agit de calculer un résultat en utilisant le moins de ressources (mémoire, temps calcul) possible, dans l'autre il s'agit de coordonner les demandes de ressources des divers utilisateurs du système d'exploitation.

Les paragraphes précédents s'appuyaient sur un exemple d'école. Nous présentons maintenant deux exemples de problèmes, l'un provenant du calcul numérique : le *produit itéré*, l'autre des systèmes d'exploitation : les *lecteurs-rédacteurs*. Leur présentation utilise le vocabulaire et les notions qui viennent d'être présentées. Ils doivent rester présents à l'esprit pour comprendre les chapitre suivants où, avec d'autres, ils seront utilisés à titre d'illustration.

Ces problèmes ont des caractéristiques communes : des processus, qui, de fait, coopèrent à une tâche commune et qui au cours de leur collaboration, ont besoin d'échanger de l'information ou de partager des ressources. Entre les échanges d'information ou les demandes de ressources, les processus ont une activité autonome, dont la durée et le contenu ne sont connus que de l'application parallèle dont ils font partie.

2.3.1 Le produit itéré

Le premier exemple est une "brique de base" de la programmation applicative. Etant donné une opération à deux opérandes notée $\star$, associative, considérons l'expression :

$$a_1 \star a_2 \star \ldots \star a_n$$

appelée produit itéré de n valeurs. Le programme séquentiel qui résout ce problème s'écrit à l'aide d'un constructeur de répétition. Cette solution utilise la propriété d'associativité comme suit :

$$((\ldots((a_1 \star a_2) \star a_3) \star \ldots) \star a_n)$$

Une solution parallèle peut regarder ce produit itéré comme :

$$(\ldots(((a_1 \star a_2) \star (a_2 \star a_3)) \star ((a_5 \star a_6) \star (a_7 \star a_8))) \ldots)$$

Les produits $a_i \star a_{i+1}$, pour tout i impair, s'effectuent en parallèle, puis leurs résultats se multiplient deux à deux, etc, jusqu'à obtenir un seul nombre. Si l'on se place dans une hypothèse SIMD synchrone avec actions de durées unitaires, avec un nombre de processus égal à $\frac{n}{2}$ et chaque processus effectuant un produit, il suffit de $(\log_2 n)$ unités de temps pour résoudre ce problème. Le procédé utilisé ici pour réorganiser le travail de façon à ce qu'il s'effectue en parallèle est standard : il s'agit de considérer une propriété mathématique sous une forme qui permet d'exhiber du parallélisme.

2.3.2 Les lecteurs-rédacteurs

Lorsque des processus coopèrent à l'exécution d'une tâche, ils doivent naturellement échanger de l'information. Dans le domaine de l'algorithmique système, la tâche qu'ils résolvent ne nous intéresse pas. Seul le mode d'échange de l'information nous préoccupe. Ici, l'échange d'information est sous la forme d'un tableau d'affichage. Soient donc R processus qui peuvent afficher (on les appelle les rédacteurs), L processus qui peuvent consulter l'affichage (on les appelle les lecteurs) et une zone d'affichage unique (on l'appelle le tampon). Ces processus ont une activité autonome et de temps en temps échangent de l'information par ce tableau d'affichage. Les *contraintes de synchronisation* sont les suivantes :

- les lecteurs doivent pouvoir consulter la dernière information cohérente affichée dans le tampon, c'est à dire qu'aucun lecteur ne peut lire pendant qu'un rédacteur affiche,
- un seul rédacteur peut écrire à un instant donné dans le tampon, mais aucun rédacteur ne peut écrire lorsqu'un lecteur est en train de lire.

Les lecteurs n'ont aucun besoin de se synchroniser entre eux. Autrement dit, l'action d'afficher est une section critique et se fait en exclusion mutuelle avec les actions de consultation. Ces contraintes de synchronisation s'expriment à l'aide de variables d'états : soient N_R le nombre de rédacteurs en train d'afficher et N_L le nombre de lecteurs en train de consulter. Ces variables d'états peuvent se déduire simplement de l'état des divers processus du programme. Les conditions suivantes doivent toujours être vérifiées :

- condition de lecture : $N_R = 0$
- condition d'écriture : $N_R = 0$ et $N_L = 0$

La condition de franchissement pour l'action de consultation est $N_R = 0$ et la condition de franchissement pour l'action d'affichage est ($N_R = 0$ et $N_L = 0$). Une solution au problème des lecteurs-rédacteurs est un algorithme parallèle qui vérifie les contraintes de synchronisation ci-dessus mentionnées pour les actions d'affichage et de consultation. Naturellement, on demandera en plus à cet algorithme qu'il évite les situations d'interblocage et de famine, et qu'il soit le plus efficace possible. L'efficacité est obtenue en laissant le parallélisme maximum aux $L + R$ processus participants afin que consultations et affichages s'effectuent au plus vite. En effet il existe une solution évidente à ce problème qui satisfait les contraintes de synchronisation. C'est la solution qui consiste à gérer les deux types de processus dans une file FIFO (premier-arrivé-premier-servi) qui ne permet l'accès au tampon qu'à un seul des $L + R$ processus à tout instant. Cette solution est sans interblocage ni famine. Mais elle ne présente aucun parallélisme et n'est pas efficace.

Chapitre 3

Eléments parallèles du langage ADA

Pour exprimer des algorithmes séquentiels, on dispose de différents langages de programmation, comme les langages impératifs, fonctionnels et logiques. De la même manière, différents types de langage de programmation concurrente et parallèle ont chacun des domaines d'application adaptés, comme les systèmes d'exploitation, les applications scientifiques ou la programmation temps réel. L'objectif de ce cours est de proposer un langage qui soit convenable pour ces différents domaines. Il faut donc un langage généraliste et puissant. Notre choix s'est porté sur ADA.

Une connaissance minimum du langage ADA séquentiel est requise : le lecteur pourra se reporter à l'annexe A pour une présentation sommaire de la syntaxe du langage, et à des ouvrages comme [Boo 88] ou [OR 90] pour avoir une vue plus complète. Dans ce chapitre, nous ne présentons que la notion de tâche par laquelle on exprime le parallélisme en ADA, et nous terminons par des exemples.

3.1 Introduction

Les années 70 ont vu naître le concept de processus dans les langages de programmation. Comme on l'a déjà dit, un processus correspond à l'exécution d'un programme séquentiel. C'est donc une séquence d'actions totalement ordonnées dans le temps. Plusieurs processus peuvent être simultanément en activité : on dit qu'ils sont parallèles. En ADA, un processus est l'exécution d'une unité de programme appelée tâche.

Les tâches constituent l'un des trois types principaux d'unités de programme en ADA (les deux autres étant les sous-programmes et les paquetages). Un programme principal est implicitement une tâche, mais il est possible de déclarer des tâches dans n'importe quelle partie déclarative du programme principal, d'une procédure, d'une tâche, etc. Cette déclaration comprend une spécification et un corps. On dit qu'une tâche est activée lors de l'exécution d'un programme, lorsqu'un processus est créé pour l'exécuter. Les tâches sont activées après l'élaboration de leur corps, à la fin de la partie déclarative du parent. La tâche est alors dépendante de ce parent et, par transitivité de la relation de dépendance, de toutes les constructions (tâche, instruction bloc, sous-programme en cours d'exécution ou bien encore paquetage de bibliothèque) dont ce parent est lui-même dépendant.

Si la partie déclarative contient plusieurs déclarations de tâches, elles seront toutes activées à la fin de la partie déclarative, dans un ordre non spécifié. Ces tâches s'exécuteront en commençant par élaborer leur partie déclarative (qui peut elle-même contenir des déclarations de tâches) en parallèle[1] avec le programme principal.

Comme pour un bloc ou un sous-programme, une tâche est dite achevée lorsqu'elle a teminé l'exécution de la dernière instruction de son corps. Si une exception est levée lors de l'ex'ecution de sa suite d'instructions, la tâche est achevée si il n'y a pas de traitement associé à l'exception, ou, si il y en a un, lorsque elle en a fini l'exécution.

Les tâches peuvent communiquer entre elles. La communication des tâches ADA peut-être comparée à la communication téléphonique : l'une des tâches prend l'initiative de la communication et désigne son interlocuteur. Dans sa version simple cet appel est bloquant, et la tâche appelante "laisse sonner" jusqu'à obtenir une réponse. Lorsque la tâche appelée répond, elle décroche sans connaître l'identité de l'appelant. Sa réponse consiste à rendre un service (par exemple exécuter une procédure). La communication se termine pour les deux tâches simultanément lorsque la tâche appelée rend les résultats attendus. Chaque tâche "raccroche" le téléphone, et reprend son exécution suivant les actions décrites dans son corps. On parle de communication par *rendez-vous*. Ce type de communication s'oppose au type de communication par boîte aux lettres dont une illustration est le courrier. Dans ce dernier schéma, l'appelé lit la lettre quand il veut, et l'appelant n'est pas bloqué ; il envoie sa lettre et passe à autre chose.

3.2 Activation et état d'une tâche

Dans la terminologie des langages de programmation traditionnels, on appelle état d'une structure de données sa valeur à un instant donné. Typiquement, cet état évolue dans le temps. On associe aussi un état à une tâche. Comme les tâches sont des outils de structuration, l'état d'une tâche est l'une ou l'autre de ces informations :

- l'état des entités englobées (ces entités sont des structures de données, mais aussi peut-être des tâches, etc)
- une information sur son comportement parmi les diverses instructions (*select*, *accept*, *delay*, *terminate*) qu'elle traverse.

Cette dernière information est souvent appelée abusivement l'état de la tâche... et nous userons de cet abus.

Cet état peut prendre plusieurs valeurs (voir figure 3.1) :

ELABORATION, ACTIF, BLOQUE, ACHEVE, TERMINE.

Une tâche est dans l'état ACTIF après l'élaboration de son corps, à la fin de la partie déclarative du parent. Si plusieurs tâches sont déclarées dans la même partie déclarative, elles sont toutes activées en fin de partie déclarative, mais l'ordre d'activation n'est pas déterminé.

1. Le parallélisme est ici simulé par la multi-programmation.

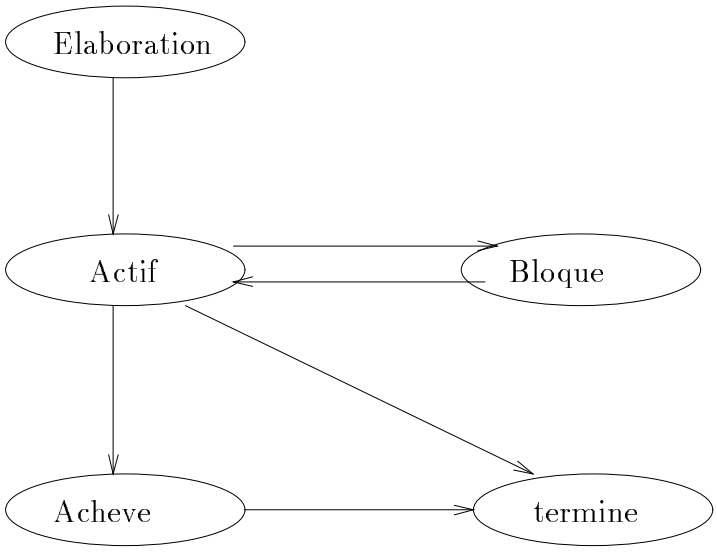


FIG. 3.1 – *Diagramme des transitions possibles d'une tâche.*

Avant d'être active une tâche est dans l'état ELABORATION et avant sa déclaration de spécification, elle n'a pas d'existence. Une tâche est dans l'état BLOQUE lorsqu'elle est en attente sur une entrée ou sur une acceptation ou encore dans une attente de *delay*. Elle est dans l'état ACHEVE si elle a atteint son instruction de fin.

Elle est dans l'état TERMINE si elle est achevée et si, lorsqu'elle a des tâches dépendantes, ses tâches dépendantes sont soit achevées, soit terminées[2].
Cette notion se généralise aux autres constructions ADA : ainsi, on ne sort d'une instruction bloc dont l'exécution est achevée que lorsque toutes ses tâches dépendantes sont terminées.

Remarque. Les états retenus ci-dessus ne sont relatifs qu'à l'état du processus exécutant les instructions d'un langage de programmation ADA. Certains modèles étendent cette vision en y incluant une vision de système d'exploitation. En effet ces tâches s'exécutent soit toutes sur des processeurs différents (il y a alors du parallélisme utilisable dans la machine), soit certaines partagent le même processeur. Parmi les tâches qui partagent le même processeur, une seule est véritablement en train d'être exécutée à un moment donné. Les autres sont peut-être dans l'état ACTIF suivant notre dénomination précédente, mais à cause du partage de la ressource processeur elles ne sont pas en exécution. Avec cette vision plus fine, l'état actif est découpé en deux sous états : EN_EXECUTION et ATTENTE (on veut dire en attente de la ressource processeur, ce qui est complètement différent de BLOQUE). C'est le mode de partage du processeur qui fait passer les tâches de l'un à l'autre des états ATTENTE et EN_EXECUTION.

A propos de cette remarque, on notera que parallélisme dans l'expression du programme n'implique pas forcément parallélisme dans la machine. De fait, parallélisme et multiprogrammation

2. Nous verrons au paragraphe 3.3.2 qu'une alternative de terminaison au sein d'une instruction de sélection peut permettre de terminer une tâche sans qu'elle ait atteint son instruction de fin.

(i.e. plusieurs tâches sur le même processeur) font souvent très bon ménage : pendant qu'une tâche est dans l'état BLOQUE, une autre peut utiliser le processeur.

3.3 Structures des tâches

La déclaration d'une tâche contient une partie spécification et un corps.

3.3.1 La déclaration de spécification

```
task pompier is
    entry echange_vide_plein ;
end pompier ;
```

Cette spécification donne un identificateur à la tâche et désigne les communications qu'il peut accepter en tant qu'appelé, qu'on appelle les points d'entrée de la tâche (*entry*). La syntaxe des points d'entrée est la même que celle des spécifications de procédures : c'est un nom suivi d'une liste (éventuellement vide) de paramètres avec leur mode d'utilisation. Voici la spécification d'une tâche chargée de gérer une structure de donnée de type pile.

```
task pile is
    entry empiler ( valeur : in element ) ;
    entry depiler ( valeur : out element ) ;
end pile ;
```

Bien qu'un appel de point d'entrée ressemble à un appel de procédure, les sémantiques sont très différentes. Si plusieurs tâches peuvent appeler le même sous-programme (on dit que le code du sous-programme est *réentrant*), une tâche ne répond qu'à une sollicitaion de point d'entrée à la fois et pour un seul appel. Les autres appels sont implicitement en attente. La sélection par une tâche du prochain appel à prendre est non spécifiée : on voit ici une des premières causes de l'indéterminisme du comportement à l'exécution des programmes ADA. Une tâche ADA peut aussi n'avoir aucune entrée et se contenter d'appeler d'autres tâches.

Les règles de visibilité des déclarations pour les structures de données et les procédures sont aussi valables pour les tâches. Ainsi, il est possible d'avoir deux tâches qui s'appellent mutuellement, pour autant que leurs spécifications soient connues lors de la définition de leurs corps.

L'appel d'une entrée de tâche se fait comme suit, avec le mode d'appel "positionnel" de variables :

```
    pompier.echange_vide_plein ;
```

```
pile.empiler(12) ;
pile.depiler(x) ;
```

3.3.2 Sélection et acceptation

La forme d'un corps de tâche est semblable à celle d'un corps de procédure : une partie déclarative suivie d'un bloc d'instructions. Il existe en ADA des instructions spécifiques de tâches : *accept* et *select* sont les deux principales.

Si l'on a déclaré un point d'entrée dans la spécification d'une tâche, son corps doit contenir au moins une fois l'instruction *accept*. Dans la tâche pompier, on doit avoir l'instruction :

```
accept echange_vide_plein do
   ... instructions ... ;
end echange_vide_plein ;
```

Si l'on reprend l'analogie avec le téléphone, l'appel d'un point d'entrée consiste, pour l'appelant, à décrocher le téléphone (et composer le numéro) ; l'acceptation consiste aussi à décrocher le téléphone, mais par l'appelé. Les communications sont typées : ici, la tâche pompier attend un appel de type `echange_vide_plein`. L'acceptation est une instruction dite bloquante : si aucun appel n'est en attente (i.e. si le téléphone ne sonne pas) la tâche attend sur l'instruction *accept* qu'un appel survienne.

Cette notion d'instruction bloquante marque une différence importante entre un langage séquentiel et un langage permettant d'exprimer le parallélisme. Un langage parallèle permet, au contraire d'un langage séquentiel, de stopper une exécution ou de la retarder et donc d'exprimer un ordre partiel entre les instructions exécutées.

On remarque que le point d'entrée `echange_vide_plein` n'a pas d'argument. La communication sert alors uniquement à la synchronisation. Avec des arguments, l'acceptation se fait comme un appel de procédure, avec le type et le mode de passage des arguments. Entre les mots clés *do* et *end* se situe le corps de l'acceptation, donc le corps du point d'entrée. A un même point d'entrée peut correspondre plusieurs acceptations, correspondant à des séquences d'instructions différentes.

Une tâche qui attend un appel (sur un *accept*) est bloquée jusqu'à ce qu'une tâche appelante ait effectué l'appel. La tâche appelée n'a aucun contrôle sur la durée de cette attente. Pour pouvoir (si on le désire), contrôler cette durée d'attente, ADA donne la possibilité de faire des acceptations sélectives et temporisées grâce à l'instruction *select*. L'acceptation sélective autorise la tâche appelée à exécuter l'appel seulement si une tâche est effectivement en attente sur cet appel. L'acceptation temporisée (grâce à l'instruction `delay`) autorise à retirer son acceptation éventuelle après une durée d'attente spécifiée.

L'exemple ci-dessous permet de présenter informellement la syntaxe et le fonctionnement de l'acceptation sélective.

```
task body pile is
    taille : constant integer := 100 ;
    struct_pile : array ( 1 .. taille ) of integer;
    tete_pile : integer;
begin
    tete_pile := 0 ;
    loop
       select
          accept empiler ( u : in element ) do
             if tete_pile < taille then
                tete_pile := tete_pile + 1 ;
                struct_pile ( tete_pile ) := u ;
             end if;
          end empiler ;
       or
          accept depiler ( u : out element ) do
             if tete_pile > 0 then
                u := struct_pile ( tete_pile ) ;
                tete_pile := tete_pile - 1 ;
             end if;
          end depiler ;
       end select;
    end loop;
end pile ;
```

Ce programme boucle indéfiniment sur une attente de l'un ou l'autre appel. Si les deux acceptations sont possibles, la tâche en sélectionne un seul et le choix est arbitraire : de nouveau l'indéterminisme apparaît ici. Si aucune n'est possible, la tâche est bloquée jusqu'à ce que l'un des *accept* soit possible (attente passive).

Plus précisément, une attente sélective (*select*) peut comprendre autant d'alternatives que désiré, chacune étant, à l'exception de la première, précédée par *or*. Une attente sélective (*select*) doit contenir au moins une alternative d'acceptation. Elle peut en outre contenir mais de manière exclusive :

- soit une (ou plusieurs) alternative de temporisation : *delay* `t`, suivie d'une suite d'instructions. `t` est une expression simple du type point-fixe prédéfini *duration* qui désigne une durée en secondes.

```
    select
        accept ...
    or  ...
    or  delay t;
        instructions ;
```

```
or  ...
end select;
```

– soit une alterntaive de terminaison : *terminate*.

```
select
    accept ...
or  ...
or  terminate;
end select;
```

– soit une partie *else*, suivie d'une suite d'instructions.

```
select
    accept ...
or  ...
else
    instructions ;
end select;
```

Alternatives gardées. Une alternative (*accept*, *delay* ou *terminate*) dans une instruction *select* peut aussi être associée à une condition booléenne appelée garde. L'écriture d'une telle alternative est alors la suivante :

when <condition> => <alternative>

Sémantique de l'attente sélective. Lors de l'exécution d'une instruction d'attente sélective *select*, les conditions associées aux alternatives gardées sont d'abord evaluées, dans un ordre arbitraire, ainsi que les expressions de délai pour les alternatives *delay*.
Les alternatives gardées pour lesquelles la condition est évaluée à faux sont dites fermées. Les autres alternatives (non gardées ou gardées avec une condition évaluée à vrai) sont dites ouvertes.

Si toutes les alternatives sont fermées et qu'il n'y a pas de partie *else*, l'exception PROGRAM_ERROR est levée.
Sinon, la sélection d'une alternative parmi celles ouvertes est effectuée de la façon suivante. On sélectionne en premier lieu les alternatives *accept* ouvertes pour lesquelles le point d'entrée associé est appelé par au moins une autre tâche : ces alternatives sont dites sélectionnées. Parmi toutes les alternatives sélectionnées, l'une d'entre elles est choisie arbitrairement et exécutée : l'acceptation correspondante est d'abord effectuée, puis les instructions qui la suivent.
Si aucune alternative ouverte n'est sélectionnée, différents cas sont à distinguer selon la présence ou non d'alternatives ouvertes *delay* ou de partie *else* :

Pas d'alternatives ouvertes autres que d'acceptation. L'instruction *select* est alors bloquante jusqu'à ce que l'une au moins des alternatives ouvertes puisse être sélectionnée.

Une alternative *delay* est ouverte. Une telle alternative est sélectionnée si aucune alternative d'acceptation ouverte ne peut l'être avant le délai indiqué[3]. Dans ce cas, les instructions qui suivent la clause *delay* dans l'alternative sont exécutées.
Lorsque plusieurs alternatives *delay* figurent dans le *select*, si aucune alternative d'acceptation ne peut être sélectionnée avant le délai minimum parmi ceux indiqués, un choix indéterministe est effectué pour sélectionner l'alternative *delay* qui sera exécutée parmi celles de délai minimum.

Avec partie *else*. Si toutes les alternatives sont fermées ou si aucune acceptation ouverte ne peut être immédiatement sélectionnée, les instructions de la partie *else* sont exécutées. Ainsi, une attente sélective qui comprend *else null* n'est pas bloquante.

L'alternative *terminate* est ouverte. Une telle alternative ne peut être sélectionnée que si la tâche dépend d'un parent dont l'exécution est achevée et dont les tâches dépendantes sont chacune soit terminée soit en attente sur l'alternative ouverte *terminate* d'une instruction *select*[4] .

L'utilisation de l'instruction *select* pemet d'écrire le corps de la tâche pile plus naturellement.

```
task body pile is
    taille : constant integer:= 100 ;
    struct_pile : array ( 1 .. taille ) of integer;
    tete_pile : integer;
begin
    tete_pile := 0 ;
    loop
       select
          when tete_pile < taille =>
             accept empiler ( u : in element ) do
                tete_pile := tete_pile + 1 ;
                struct_pile ( tete_pile ) := u ;
             end empiler ;
       or
          when tete_pile > 0 =>
             accept depiler ( u : out element ) do
                u := struct_pile ( tete_pile ) ;
                tete_pile := tete_pile - 1 ;
             end depiler ;
       or
          terminate;
       end select;
    end loop;
end pile ;
```

3. Si le délai est nul ou négatif, l'alternative *delay* est sélectionnée et exécutée si aucune alternative d'acceptation ouverte ne peut être immédiatement sélectionnée.

4. Par suite, l'alternative *terminate* ne peut être sélectionnée si il y a des appels en attente sur l'un des points d'entrée de la tâche.

Cette tâche n'accepte pas d'appel *empiler* si la structure de donnée est pleine ni *dépiler* si elle est vide. Les gardes sont bien sûr réévaluées à chaque passage dans la boucle. De plus la tâche se termine dès que la construction qui l'a créée (ainsi que ses tâches dépendantes) est achevée.

3.3.3 Indexation d'une famille de points d'entrée

Il est possible d'indexer une famille de points d'entrée, comme le montre l'exemple suivant.

```
type indice is range 1 .. 10;
task pile_multiple is
    entry empiler ( indice ) ( x: in element );
    entry depiler ( indice ) ( x: out element );
end pile_multiple;
```

Le corps de la tâche `pile_multiple` peut traiter l'indexage par une boucle :

```
task body pile_multiple is
    taille: constant integer:= 100;
    struct_pile: array ( indice, 1 .. taille ) of integer;
    tete_pile: array ( indice ) of integer;
    i: indice;
begin
    ...
    for i in indice loop
       select
          when tete_pile(i) < taille =>
             accept empiler ( i ) ( u: in element ) do
                tete_pile(i) := tete_pile(i) + 1;
                struct_pile ( i, tete_pile(i) ) := u;
             end empiler;
       or
          ...
       else
          null;
       end select;
    end loop;
end pile_multiple;
```

Dans cet exemple, les *accept* pour les divers points d'entrée d'indices différents sont visités dans l'ordre spécifié par la boucle. Les *select* doivent alors être utilisés dans leur version non bloquante. L'appel d'entrées indexées se fait par :

```
pile_multiple.empiler ( 1 ) ( 12);
```

pour empiler 12 sur la pile d'indice 1.

3.4 Définition des types de tâches

On peut définir des types de tâche. Un type de tâche est un type *limité-privé*: un objet d'un tel type ne peut pas apparaître comme cible d'une affectation, ni être passé en paramètre d'une procédure. Il est cependant possible de déclarer en paramètre de procédure un accès sur une tâche: ce dispositif permet de connaître l'identité de l'appelant en le passant par paramètre (via un accès) ou bien de renvoyer à l'appelant l'identité d'une tâche créée.

```
task type pompier is
    entry echange_vide_plein ;
end pompier ;
```

La définition du corps de type de tâche est identique à celui que l'on décrirait pour la déclaration directe d'une tâche. L'utilisation de ce type peut se faire comme suit:

```
robert : pompier ;
bataillon : array (1..20) of pompier ;                  -- un tableau de tâches de type pompier
type pt_pompier is access pompier ;                     -- un type accès à une tâche de type pompier
georges : pt_pompier ;                                  -- un accès à une tâche de type pompier
```

---

qui permettent de se référer à

```
robert.echange_vide_plein ;
bataillon(1).echange_vide_plein ;
georges := new pt_pompier ;                             -- création d'une instance de pompier
georges.echange_vide_plein ;
```

On dit que `georges` est un accès à une instance de tâche de type `pompier`. On utilise ce mécanisme de création dynamique de tâche lorsque le nombre de tâches n'est pas connu au départ et lorsque l'identité de cette tâche est nécessaire. La tâche créée devient active juste après l'exécution du *new*.

3.5 Autres possibilités du langage

D'autres possibilités sont ouvertes dans le langage mais nous ne les développerons pas ici ; par exemple:

- possibilité d'auto-destruction en cas de comportement anormal.

- possibilité de s'attribuer des priorités qui sont prises en compte dans une politique d'attribution du processeur en multiprogrammation.
- certains attributs d'une tâche sont accessibles comme son état (comme défini ci-dessus), le nombre d'appels en attente sur un point d'entrée, etc.
- utilisation d'un module standard de gestion du temps.
- des dispositifs pour contrôler les accès concurrents sur des variables (notamment en écriture)
- possibilité d'effectuer un appel de point d'entrée non bloquant (*select* chez l'appelant avec clause *else* ou *delay*)
- etc

3.6 Exemples de programmation en ADA

Cette présentation de ADA est peu formelle et essentiellement fondée sur l'exemple. Voici des programmes résolvant certains problèmes posés dans le chapitre introductif.

3.6.1 Exclusion mutuelle

Lorsque plusieurs processus s'exécutent en parallèle mais doivent exécuter certaines de leurs opérations en exclusion mutuelle avec les autres processus, il est nécessaire de mettre à disposition un outil garantissant l'exclusion mutuelle. Cette exclusion peut être définie en ADA à l'aide d'une tâche du type suivant :

```
task type exclusion_mutuelle is
    entry entrer;
    entry sortir;
end exclusion_mutuelle;

task body exclusion_mutuelle is
begin
    loop
       select
          accept entrer;
          accept sortir;
       or
          terminate;
       end select;
    end loop;
end exclusion_mutuelle;
```

Supposons déclarée une tâche `section_critique` de ce type :

```
section_critique : exclusion_mutuelle ;
```

Lorsque l'un des processus désire effectuer une séquence d'instructions <S> en exclusion mutuelle, il lui suffit allors d'exécuter les opérations suivantes :

```
section_critique.entrer ;                    -- pour entrer en section critique
<S> ;                                -- les instructions à effectuer en exclusion mutuelle
section_critique.sortir ;                    -- pour sortir de la section critique
```

Nous utiliserons cette technique pour implanter l'exclusion mutuelle en ADA dans le chapitre 6.

3.6.2 Files d'attente de processus

Les processus en attente sur un point d'entrée sont servis selon l'ordre FIFO. Cette propriété est très utile pour définir une file d'attente de processus, outil qui sera utilisé dans le chapitre 6.

Supposons que l'on désire écrire un outil permettant à des processus de se mettre en attente (i.e. d'arrêter toute activité) jusqu'à ce que d'autres processus viennent les réveiller, le premier processus réveillé étant celui qui est en attente depuis le plus longtemps. Un tel outil peut être écrit de la manière suivante par une tâche ADA, avec deux points d'entrée: `endormir` (pour se mettre en attente), et `réveiller` (pour réveiller un processus en attente) :

```
task type file_d_attente is
    entry reveiller ;
    entry endormir ;
end file_d_attente;

task body file_d_attente is
begin
    loop
       select
          accept reveiller ;
          accept endormir ;
       or
          terminate;
       end select;
    end loop;
end file_d_attente;
```

3.6.3 Les pompiers

Dans cette programmation des pompiers, on remarque la phase d'initialisation qui permet d'attribuer à chaque pompier une identité. On remarque aussi que la déclaration de `bataillon` avant le corps de tâche de pompier a permis d'utiliser cet identificateur – donc d'identifier les pompiers de la chaîne – dans ce corps de tâche.

```
procedure eteindre_incendie( taille_bataillon, seaux_necessaires : in integer ) is

   package ENTIER_IO is new INTEGER_IO(integer ); use ENTIER_IO;

   duree_remplir : constant duration := 0.2 ;
   duree_verser : constant duration := 1.0 ;
   type etat is (plein, vide) ;

   task type pompier is
      entry initialisation_chaine(j : in integer ) ;
      entry echange_mon_seau_ton_seau(
         mon_seau : in etat ; ton_seau : out etat) ;
   end pompier ;

   bataillon : array (1..taille_bataillon) of pompier ;

   task body pompier is

      id : integer;                                              -- son idendité
      avant : integer;                                -- L'idendité du pompier avec qui il échange
      seau : etat := vide ;

      procedure remplir_seau(s : in out etat) is
      begin
      delay duree_remplir ;
         put ("remplir : pompier ") ; put (id) ; new _line ;
         s := plein ;
      end remplir_seau ;

      procedure verser_sur_incendie(s : in out etat) is
      begin
         delay duree_verser ;
         put ("verser : pompier ") ; put (id) ; new _line ;
         s := vide ;
      end verser_sur_incendie ;


   begin
      accept initialisation_chaine(j : in integer ) do
         id := j ;
         avant := id - 1 ;
      end initialisation_chaine ;

      for nombre_de_seau in 1..seaux_necessaires loop
```

```
            if id = 1 then remplir_seau(seau) ;
            else bataillon(avant).echange_mon_seau_ton_seau(seau, seau) ;
            end if;
            if id = taille_bataillon then verser_sur_incendie(seau) ;
            else
               accept echange_mon_seau_ton_seau(
                  mon_seau : in etat ; ton_seau : out etat) do
                  ton_seau := seau ;
                  seau := mon_seau ;
                  put ("echanger : seau plein de ") ;
                  put (id); put (" avec seau vide de ") ;
                  put (id+1) ; new _line ;
               end echange_mon_seau_ton_seau ;
            end if;
         end loop;
      end pompier ;

begin                                                    -- début programme principal
      for k in 1..taille_bataillon loop
         bataillon(k).initialisation_chaine(k) ;
      end loop;
end eteindre_incendie ;
```

3.6.4 Les lecteurs-rédacteurs

Le problème posé définit *a priori* l'existence de L tâches lecteurs et R tâches rédacteurs. Une façon naturelle de programmer est de créer une tâche responsable de la gestion de l'accès au tableau d'affichage. Cette tâche joue le rôle de serveur dont les tâches lectrices et rédactrices sont clients. ADA est particulièrement bien adapté à ce type de programmation client-serveur. En effet, la dissymétrie du rendez-vous en ADA permet à l'appelant de nommer l'appelé, alors que l'appelé accepte des appels anonymes (sans connaître le nom de l'appelant). D'autres langages proposent des rendez-vous symétriques où l'appelé désigne aussi l'appelant.

```
procedure lecteurs_redacteurs is
      type contenu_affichage is ...
      task type redacteur is
      end redacteur ;
      task type lecteur is
      end lecteur ;

      task serveur_d_affichage is
         entry lire ( v : out contenu_affichage ) ;
         entry ecrire ( v : in contenu_affichage ) ;
      end serveur_d_affichage ;

      population_lecteur : array ( 1..L) of lecteur ;
      population_redacteur : array ( 1..L) of redacteur ;
```

```
   task body lecteur is
      procedure consommer_lecture ( p: in contenu_affichage ) is
      begin
         ...
      end consommer_lecture;
      v : contenu_affichage;
   begin
      while (...) loop
         serveur_d_affichage.lire ( v );
         consommer_lecture ( v );
      end loop;
   end lecteur;

   task body redacteur is
      procedure produire_information ( p: in contenu_affichage ) is
      begin
         ...
      end produire_information;
      v : contenu_affichage;
   begin
      while (...) loop
         produire_information ( v );
         serveur_d_affichage.ecrire ( v );
      end loop;
   end redacteur;

   task body serveur_d_affichage is
      valeur : contenu_affichage := valeur_initiale;
   begin
      loop
         select
            accept lire ( v: out contenu_affichage ) do
               v := valeur;
            end lire;
         or
            accept ecrire ( v: in contenu_affichage ) do
               valeur := v;
            end ecrire;
         or
            terminate;
         end select;
      end loop;
   end serveur_d_affichage;

end lecteurs_redacteurs;
```

Ici, la procédure principale ne comporte pas de partie impérative, mais les tâches sont actives à la fin de la partie déclarative.

Cette solution est très limitative en terme de parallélisme : un seul lecteur ou bien un seul

rédacteur peut accéder au tableau d'affichage à un instant donné. La tâche serveur impose une exclusion mutuelle qui n'est pas requise par l'énoncé du problème. On aurait pu le remarquer puisqu'aucune utilisation n'a été faite de ces fameuses contraintes de synchronisation évoquées dans le chapitre introductif. On remarquera aussi que cette solution au problème des lecteurs-rédacteurs autorise la famine de l'une ou l'autre de ces familles : il est possible que jamais un lecteur ne puisse lire, puisque la sélection est indéterministe (et de même pour les rédacteurs). Ce problème sera traité dans le chapitre 9.

3.6.5 Le produit itéré

Un algorithme pour exécuter le produit itéré de n nombres (on supposera que n est une puissance de 2 pour simplifier le problème) se fonde sur la propriété associative du produit pour d'abord multiplier les éléments deux par deux en parallèle, puis les résultats deux par deux entre eux, etc. Ce même algorithme a une expression récursive simple : pour faire le produit de n nombres, on exécute d'abord en parallèle deux produits de taille $n/2$ et on multiplie les résultats entre eux. Une expression en ADA de cet algorithme est donnée dans le chapitre 4.

3.7 Conclusion

Cette brève introduction aux outils ADA va nous servir de base de travail. Les chapitres 4 et 5 montrent la programmation d'algorithmes parallèles en ADA, les chapitres 6, 7 et 9 développent quelques concepts fondamentaux de la programmation coopérative, très utilisée dans le domaine des systèmes d'exploitation.

Il faut se rappeler que ADA est un langage extrêmement riche et que ce cours n'a pas pour objectif d'enseigner ADA mais uniquement de se servir de ADA pour faire comprendre les principes de l'algorithmique et de la programmation parallèle.

Exercices

1. Dans l'exemple de la file d'attente de processus, lorsqu'un processus appelle l'entrée réveiller et qu'il n'y a pas de processus endormi, il est bloqué. Proposer une implémentation qui évite ce blocage : on pourra utiliser l'instruction *select* else et faire apparaître un ordonnancement global de tous les processus, qu'ils appellent `endormir` ou `reveiller`.

Bibliographie

[**Boo 88**] Ingénierie du logiciel avec ADA, Grady Booch, InterEditions 1988.

[**OR 90**] Langage ADA et algorithmique, R. Ogor et R. Rannou, Hermes 1990.

[**ADA 83**] Reference Manuel for the ADA Programming Language, ALSYS, Janvier 1983.

Chapitre 4

Fondements de l'Algorithmique Parallèle

Nous allons montrer dans ce chapitre qu'il est possible de résoudre un problème donné, même lorsqu'il est énoncé sous une forme séquentielle (un ensemble de données en entrée auquel doit correspondre un ensemble de résultats en sortie), à l'aide d'algorithmes parallèles. On appelle *algorithmique parallèle* la conception de méthodes permettant de résoudre en parallèle tous les problèmes calculables du calcul séquentiel.

L'algorithmique parallèle n'est pas vraiment nouvelle : depuis longtemps, les circuits qui réalisent certaines primitives de machines séquentielles implantent un algorithme parallèle (par exemple la multiplication d'entiers). Cependant, la disponibilité effective d'architectures parallèles a donné un essor à l'algorithmique parallèle programmée, car elle permet d'envisager, pour de nombreux problèmes, de nouvelles solutions permettant de réduire de manière conséquente les temps de résolution.

Les différences existant entre les modes de calculs séquentiels et parallèles sont profondes et, par suite, les algorithmes permettant de résoudre un même problème pour chacune de ces deux grandes classes sont souvent fondamentalement différents. La vision parallèle est plus générale (puisque le calcul séquentiel est un sous cas du cas parallèle) et a permis de dégager de nouveaux algorithmes qui s'avèrent parfois intéressants sur des machines séquentielles.

Ce chapitre est destiné à mettre en évidence les fondements de l'algorithmique parallèle, c'est à dire les concepts qui permettent d'élaborer une solution parallèle à un problème donné. On examinera d'abord (section 4.1) quelles sont les "sources" de parallélisme, puis les modèles (section 4.2) permettant de quantifier la complexité d'un algorithme parallèle et enfin les méthodes de base (section 4.3) pour la construction d'algorithmes parallèles. Dans la section 4.4, nous appliquons ces méthodes via quelques programmes simples écrits en ADA. Nous verrons dans le chapitre suivant comment les techniques génériques présentées dans le présent chapitre autour d'exemples simples peuvent être utilisées pour résoudre des problèmes plus complexes.

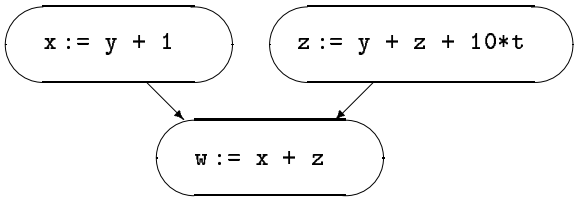


FIG. 4.1 – *Graphe de précédence d'un programme simple.*

4.1 Parallélisme de données et fonctionnel

On identifie deux types de source de parallélisme : le parallélisme fonctionnel et le parallélisme de données.

4.1.1 Parallélisme fonctionnel

Le graphe de précédence

Considérons le programme séquentiel :

---

x := y + 1 ;
z := y + z + 10 * t ;
w := x + z ;

L'obtention du résultat à partir des entrées x, y et z nécessite plusieurs opérations, certaines pouvant être effectuées en parallèle.

Le graphe orienté de la figure 4.1 s'appelle *graphe de précédence* ou *graphe de flot de contrôle.* Les noeuds du graphe sont les opérations à effectuer pour résoudre le problème et les arcs orientés les contraintes de précédences entre les opérations. Ce graphe orienté est l'image d'une relation d'ordre partiel. Les opérations ordonnées par cet ordre partiel doivent être exécutées séquentiellement (suivant l'ordre partiel) et les opérations non ordonnées par cet ordre partiel sont dites indépendantes et peuvent s'exécuter en parallèle.

Dans l'exemple précédent, les instructions `x := y+1` et `z := y+z+10*t` du programme séquentiel pourraient s'exécuter en parallèle, le résultat serait inchangé. Par contre l'instruction `w := x+z` doit impérativement attendre la fin des deux précédentes pour débuter. On dit qu'il y a des dépendances de données entre ces instructions.

Ce graphe est une vision statique du parallélisme d'un problème. La longueur de son plus long chemin donne une idée du nombre d'opérations qui doivent être exécutées de façon séquentielle donc de la durée de l'algorithme. La quantité de noeuds qui ne sont pas comparables par la relation d'ordre (ou plus précisément le diamètre du graphe) donne une idée du nombre maximum d'opérations que l'on peut effectuer en parallèle.

Remarque. L'interprétation que l'on donne ici au graphe (fig. 4.1) diffère sensiblement de celle qui avait été faite pour le graphe évoqué au chapitre 2. Le graphe des exécutions du chapitre 2 est une vision dynamique de l'ensemble des exécutions possibles et les noeuds sont les états possibles atteignables. Ces deux interprétations sont donc totalement distinctes.

On peut spécifier un algorithmme parallèle en donnant un ensemble d'opérations et leur graphe de précédence. La topologie (ou structure) de ce graphe (arbre, hypercube, grille, quelconque, etc) est une des caractéristiques importante de l'algorithme. Typiquement, la programmation d'un tel algorithme sera telle que les processus exécutent en séquence les opérations sur les chemins du graphe.

Le parallélisme fonctionnel

On peut construire un graphe de contrôle à partir d'un programme séquentiel grâce à la technique dite de *parallélisation automatique* . Cette méthode n'est pas abordée dans cet ouvrage mais le lecteur peut se reporter à [CT93]. Par contre, on peut mettre en évidence le parallélisme fonctionnel d'un problème (et non plus d'un programme séquentiel) en exhibant un ensemble d'opérations à effectuer et un graphe de précédence reliant ces opérations. Considérons un exemple générique, le "produit" itéré de n données par une loi associative. Dans un produit itéré, étant donné les n entrées $(a_0, \ldots, a_{n-1})$, il faut calculer en sortie :

$$\Pi = a_0 \star a_1 \star \cdots \star a_{n-1}$$

la loi $\star$ étant une loi associative comme la somme, le produit, le maximum, etc. Le fait que la loi soit associative est ici essentiel. Le produit itéré est utilisé par exemple pour le calcul d'un produit scalaire, de la factorielle, etc. .

La différence entre l'algorithme séquentiel classique et l'algorithme parallèle présenté ici réside dans la façon d'appliquer l'associativité. La solution séquentielle s'écrit naturellement par accumulation : le produit itéré de n éléments est le produit du premier élément avec le produit des $n-1$ derniers éléments.

```
Pi := a ( 0 ) ;
for i in 1 .. n - 1 loop
   Pi := Pi * a ( i ) ;
end loop ;
```

Le temps nécessaire à l'exécution de cet algorithme est linéaire en n. Son graphe de précédence étant une chaine, il ne peut être exploitable en parallèle puisque toutes les opérations sont totalement ordonnées.

Pourtant, il est facile de construire un algorithme parallèle illustré par le graphe de précédence de la figure 4.2, où nous avons considéré, pour simplifier, que n était une puissance de 2.

Cet algorithme peut se programmer à l'aide de n processus. Si l'on suppose que toutes les opérations ont la même durée unitaire, le temps nécessaire pour obtenir le résultat est $\log n$ (

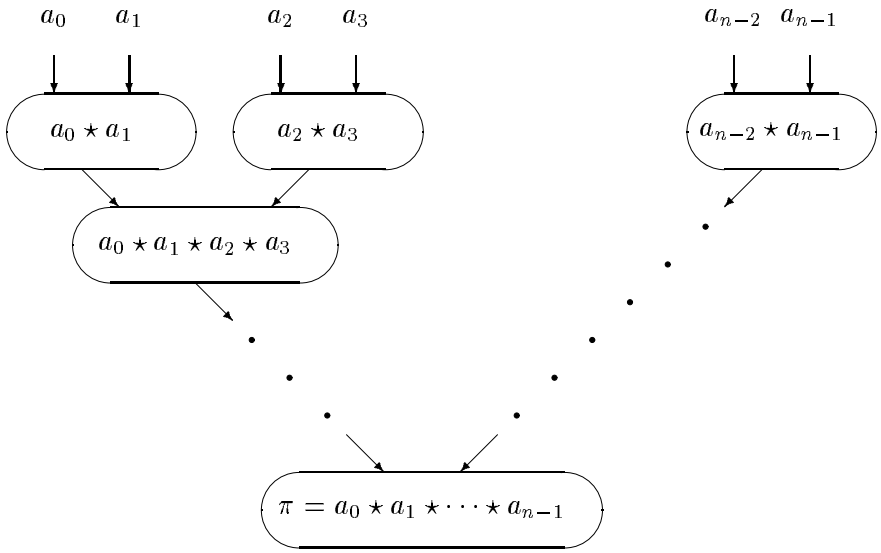


FIG. 4.2 – *Graphe de précédence du produit itéré par accumulation.*

dans tout ce qui suit la notation log est utilisé pour noter la fonction logarithme en base 2). Un programme ADA qui implante cet algorithme est présenté au paragraphe 4.4.

4.1.2 Parallélisme de données

La notion de parallélisme de contrôle est très générale. Elle consiste à découper un problème en opérations élémentaires et à indiquer l'ordonnancement de ces opérations. Il est particulièrement utilisé lorsque des opérations indépendantes s'exécutent sur les mêmes données ou bien sur des données différentes. Par contre, on parle de *parallélisme de données* lorsque la même opération (SIMD : Single Instruction Multiple Data) ou la même fonction (SPMD : Single Procedure Multiple Data) est effectuée sur des données en entrée différentes. Cette source de parallélisme est particulièrement fréquente lorsque l'on effectue des opérations sur des vecteurs (ce qui est souvent le cas en calcul scientifique).

Il existe deux façons d'exploiter ce parallélisme dû aux données, que nous présentons grâce à l'exemple du calcul suivant :

Calculer $r_i = a_i \star b_i \quad i = 0, \ldots, n-1$ avec $\star$ opérateur binaire quelconque

4.1.3 Le mode Parallèle sur les données

Les opérations $a_i \star b_i$ sont indépendantes. On peut donc les effectuer en parallèle : on a $n+1$ tâches indépendantes, la tâche i correspondant au calcul de $a_i \star b_i$. Le graphe de précédence est élémentaire car sans arc (voir fig. 4.3).

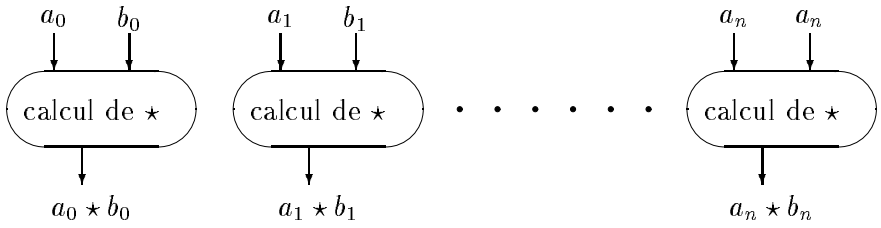


FIG. 4.3 – *Mode parallèle : tous les calculs indépendants sont effectués simultanément.*

Le temps total du calcul sur les $2n$ données est alors le temps du calcul sur 2 donnée si l'on dispose de n unités de calcul indépendantes.

Le parallélisme de données vu ainsi est une forme de parallélisme parfait. Cependant, il est rare qu'il apparaisse de façon aussi pure. Ainsi, dans un produit scalaire, les produits des coordonnées deux à deux font apparaître un parallélisme de données, mais l'obtention du résultat final – c'est une somme itérée – nécessite une fusion des résultats obtenus.

Cette forme de parallélisme est très fréquente et certains langages (CM*, HPF Fortran) disposent d'instructions spécifiques pour l'exprimer.

4.1.4 Le mode Pipeline

Supposons que l'opération $\star$ soit "complexe" à effectuer et puisse être découpée en k sous-calculs successifs $E_1, \ldots, E_k$ plus simples. On a alors :

$$a * b = E_k(E_{k-1}((\cdots(E_1(a,b))\cdots)))$$

Pour la même suite (a_i, b_i) qu'au paragraphe précédent, on enchaîne alors les calculs selon le principe suivant :

- à l'étape 1 : on calcule $r_0^{(1)} = E_1(a_0, b_0)$
- à l'étape 2 : on calcule $r_0^{(2)} = E_2(r_0^{(1)})$ et $r_1^{(1)} = E_1(a_1, b_1)$
- etc...

Au bout de k étapes, le résultat $r_0 = r_0^{(k)} = a_0 \star b_0$ est obtenu en sortie de E_k et on entre en régime stationnaire : on alimente la série d'opérateurs de manière continue, d'où le nom d'opération "pipe-line". Chaque opérateur élémentaire E_i est appelé "étage" du pipe-line. Le temps pour le calcul total (en supposant que tous les calculs E_i prennent un temps 1) est alors $k + n$ (au lieu de (kn) en calcul séquentiel) et le pipeline est d'autant plus efficace que n est grand devant k.

A titre d'exemple, l'addition de deux flottants peut être réalisée avec 4 étages successifs :

- E_1 : comparaison des exposants (soustraction des exposants)

- E_2 : alignement des mantisses (décalage)
- E_3 : addition des mantisses (sur 64 bits)
- E_4 : normalisation du résultat (rétablissement du format)

Le principe du mode pipeline est donc le découpage d'une opération s'effectuant en temps borné en sous-opérations de même durée. Le mode pipeline a été très efficacement exploité dans les architectures de type CRAY pour profiter du parallélisme de donnée. Il faut noter que le mode pipeline permet d'intégrer efficacement certaines dépendances entre les données (comme, par exemple, une propagation de retenue ou un produit scalaire), très difficiles à traiter en mode parallèle.

Le mode pipeline ne sera pas repris dans la suite de cet ouvrage. Nous l'avons présenté ici car ce mode d'organisation des calculs relève effectivement du parallélisme et qu'il peut être très utilement combiné avec le parallélisme de données ou le parallélisme de contrôle.

En conclusion, on se rappellera que le graphe de précédence est un outil fondamental de conception d'algorithme parallèle. Les sources de parallélisme sont le parallélisme de données sur les structures de données vectorielles (et matricielles) et le parallélisme de contrôle. Bien exploiter le parallélisme de contrôle consiste à minimiser les dépendances de données dans les graphes de contrôle.

4.2 Complexité Parallèle

De façon à pouvoir évaluer précisément la complexité (en nombre d'opérations, en espace-mémoire utilisé, ...) d'un algorithme parallèle et le comparer à un algorithme séquentiel résolvant le même problème, il est fondamental de définir un modèle de calcul parallèle permettant de le mesurer quantitativement. Une fois ce modèle choisi, il est intéressant d'établir une classification. Il y a ici une analogie avec l'algorithmique séquentielle et la classification en classes $\mathcal{P}$ (problèmes traitables en temps polynômial sur une machine de Turing déterministe) et $\mathcal{NP}$ (problèmes traitables en temps polynômial sur une machine de Turing non-déterministe).

Soit A_n un algorithme (séquentiel ou parallèle) calculant la solution d'un problème P_n (instance d'un problème général P ayant $n^{O(1)}$ entrées - $O(1)$ est une fonction majorée par une constante, et en particulier une constante). Comme exemples de problème P_n, on peut considérer le produit de deux entiers de n bits ($2n$ entrées), de deux matrices $n \times n$ à coefficients flottants ($2n^2$ entrées), de deux matrices $n \times n$ à coefficients entiers de n bits ($2n^3$ entrées), etc. Pour mesurer la qualité d'un algoritme A_n, il est nécessaire de dégager des critères de mesure. Il apparaît naturel de considérer le temps de l'algorithme, c'est à dire le temps qui s'écoule entre le lancement de l'exécution de l'algorithme et la fin de l'exécution de l'algorithme. Ce critère, bien qu'intuitif, n'est cependant pas le seul à prendre en compte, puisqu'un algorithme, pour être exécuté, a besoin d'un support matériel (lié au modèle de calcul) que le temps ne décrit pas.

Dans le cadre du calcul séquentiel, le modèle considéré est la machine de Turing (et ses variantes). En pratique, on utilise le modèle RAM (Random Access Machine), variation de ce modèle théorique mieux adaptée à la description d'algorithmes sur une machine séquentielle quelconque [Coo85, AHU 74].

4.2.1 Le modèle PRAM

De nombreux modèles sont proposés pour le calcul parallèle *synchrone*, liés à la variété des architectures parallèles (circuits, machines à mémoire partagée ou distribuée, machines hiérarchiques etc.). Il y a cependant deux modèles qui prédominent [Coo 85] : le premier (*PRAM*) met en évidence le nombre de processeurs, le deuxième (*circuit*) le nombre de connexions entre processeurs. Nous avons choisi ici de ne présenter que le modèle PRAM, qui est le plus utilisé en algorithmique parallèle et qui peut être vu comme une généralisation du modèle séquentiel RAM. Nous en donnons une première définition intuitive, puis une définition plus formelle.

Modèle d'architecture et relation avec le graphe de précédence

Intuitivement, le modèle PRAM correspond à différentes unités de calcul synchrones (du modèle RAM), en nombre infini, qui communiquent via une mémoire partagée. Ainsi, une PRAM est un ensemble infini de processeurs fonctionnant de manière synchrone et indicés $P_0, \ldots, P_n, \ldots$, une mémoire *globale* infinie et un programme fini [BDG 89]. Chaque processeur P_i a une mémoire *locale* infinie et possède son propre compteur ordinal, un accumulateur et un indicateur d'*activité* (vrai si le processeur exécute des instructions).

Le fonctionnement est le suivant : à chaque *pas*, toutes les unités accèdent à un nombre borné de données (généralement 2), font un calcul (de temps borné), et rendent ensuite un nombre borné de résultats (généralement 1). La notion de "pas" correspond à l'hypothèse synchrone avec durée unitaire des opérations.

Pour exécuter sur une PRAM un algorithme parallèle décrit par un ensemble d'opérations (chaque opération est de durée unitaire) et un graphe de précédence, on exécute au premier pas de calcul toutes les opérations qui n'ont pas d'arc de précédence entrant, puis au second pas toutes les opérations qui suivent (au sens du graphe) celles qui ont été exécutées au premier pas et ainsi de suite. On parle de *pas* de calcul ou d'*étape* de calcul (voir figures 4.1 ou 4.2).

Ce modèle s'adapte bien aux architectures parallèles dites à mémoire partagée, c'est à dire où tous les processeurs accèdent à une même mémoire. "Accéder" aux données veut alors dire "lire en mémoire" et "rendre" un résultat veut dire modifier une variable en mémoire. Ceci est shématisé dans la figure 4.4.

De nombreux algorithmes parallèles sont évalués sur ce modèle, qui correspond à une classe de machines parallèles. Cependant, ce modèle reste relativement imprécis et il est souvent nécessaire de préciser certaines caractéristiques importantes, notamment le mode selon lequel sont réglés les conflits de lecture et écriture par plusieurs processeurs à une même adresse dans la mémoire commune. On distingue ainsi trois sous-modèles (classés par ordre de puissance croissante) :

- EREW (Exclusive Read Exclusive Write ; lecture et écriture exclusives) : deux processeurs différents ne peuvent ni lire ni écrire à une même étape sur une même case mémoire.
- CREW (Concurrent Read Exclusive Write ; lecture concurrente et écriture exclusive) : des processeurs différents peuvent lire à une même étape le contenu d'une même case mémoire, mais ne peuvent écrire simultanément sur une même case.

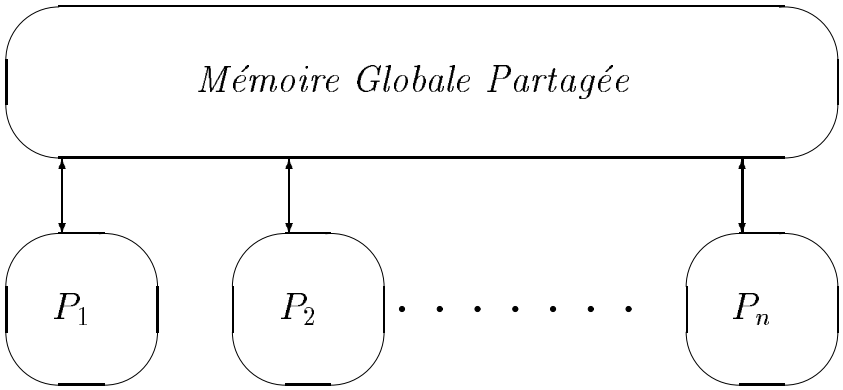


FIG. 4.4 – *Modèle PRAM*

– CRCW (Concurrent Read Concurrent Write ; lecture et écriture concurrentes) : à une même étape, différents processeurs peuvent lire ou écrire sur une même case. Dans le cas d'écritures concurrentes, plusieurs modes peuvent être utilisés pour préciser comment est réglé le conflit d'écriture (ARBITRAIRE, PRIORITAIRE ou COMMUNE).

Ces différents sous-modèles de PRAM sont cependant très proches et il existe des résultats permettant de passer d'un modèle à un autre. Dans ce qui suit nous travaillerons surtout avec le modèle PRAM-CREW qui est réaliste.

Du modèle théorique PRAM à la programmation en ADA

En ADA, la notion de mémoire *physique* locale ou globale se représente aisément par les notions de variables *logiquement* locales (à une procédure ou à une tâche) ou globales à un module. Le lien effectif (par une mise en oeuvre) entre ces deux concepts sera explicité plus tard (chapitres 6 et 7).

Le jeu d'opérations du modèle PRAM est réduit aux opérations booléennes, mais on peut sans difficulté se placer à un niveau d'abstraction plus élevé en considérant que la PRAM est capable d'exécuter toutes les opérations standards des langages de programmation.

Le parallélisme est réalisé par l'instruction `fork etiquette` : quand à un pas de calcul un proccesseur P_i exécute cette instruction, il *sélectionne* le premier processeur inactif P_j, efface la mémoire locale de P_j, copie son accumulateur Acc_i dans celui Acc_j de P_j, met le compteur ordinal de P_j à `etiquette` et incrémente de 1 son propre compteur ordinal C_i. Au top suivant, le processeur P_i exécute donc l'instruction qui suit le `fork`, alors que P_j exécute l'instruction étiquetée `etiquette` dans le programme. Pour la transmission de valeurs entre P_i et P_j en vue d'effectuer un calcul en parallèle, il suffit que P_i mette dans son accumulateur Acc_i l'adresse dans la mémoire globale du bloc contenant les paramètres en entrée du sous calcul. Parmi ces paramètres peut figurer l'adresse dans la mémoire globale où le processeur P_j mettra son résultat. P_i peut alors tester régulièrement cette position pour déterminer quand le processeur P_j a terminé le sous-calcul qui lui a été confié. Cette instruction modélise l'activation d'une tâche en ADA. Cette description opérationnelle de passage de paramètre constitue une bonne base pour comprendre ce qui se passe en ADA lors de l'activation d'une tâche mais aussi à l'appel des entrées.

Plus précisément, l'instruction `fork` peut être écrite relativement facilement en ADA par le biais d'une nouvelle tâche. Supposons que le processeur P_i désire effectuer en parallèle un calcul de la forme :

```
f ( donnees, resultat )
```

où f est une fonction ADA de profil :

```
f ( entrees : in type_entrees ; sorties : out type_sortie ) ;
```

Pour programmer un tel calcul parallèle en ADA, il suffit de définir un type de tâche `tache_f` avec deux points d'entrée :

```
task type tache_f is
    entry envoyer_donnees ( entrees : in type_entree ) ;
    entry recevoir_resultat ( sorties : out type_sortie ) ;
end tache_f ;
```

et le corps de tâche suivant :

```
task body tache_f is
    procedure f ( entrees : in type_entree ; sorties : out type_sortie) is ... ;
    arg : type_entree ;
    res : type_sortie ;
begin
    accept envoyer_donnees ( entrees : in type_entree ) do
       arg := entrees ;
    end envoyer_donnees ;
    f ( arg, res ) ;
    accept recevoir_resultat ( resultat : out type_sortie ) do
       resultat := res ;
    end recevoir_resultat ;
end tache_f ;
```

L'instruction *fork* , exécutée par P_i de la PRAM pour effectuer en parallèle le calcul *f*(*donnees*, *resultat*) peut alors s'écrire en ADA de la façon suivante :

```
    declare P_j : tache _f ;
    begin
       P_j.envoyer_donnees ( donnees ) ;
       -- ici on peut faire des calculs pendant que la tache P_j s'execute...
       P_j.recevoir_resultat ( resultat ) ;
    end ;
```

Nous utiliserons ce canevas générique de programmation dans tout ce chapitre et le suivant pour écrire des programmes ADA implémentant les algorithmes parallèles PRAM que nous construirons.

Dans le modèle de graphe de précédence des programmes, on voit bien comment cette action *fork* (de la PRAM) effectuée par le processeur P_i est représenté par un noeud du graphe de précédence, l'action suivante de P_i sera un autre noeud et la première action de P_j un troisième noeud. Il y a un arc de précédence entre l'action fork de P_i et la première action de P_j, correspondant à la dépendance de données. Il y a éventuellement un arc de précédence entre les deux actions de P_i si il y a dépendance de données entre ces deux actions.

Néanmoins, il n'existe pas à ce jour de compilateur ADA distribué, c'est à dire générant du code pouvant s'exécuter sur une architecture parallèle comportant plusieurs processeurs ayant chacun une mémoire locale, et se partageant une mémoire globale (ou dotés d'un dispositif d'envoi de message permettant de simuler une mémoire globable). Le parallélisme n'est donc qu'émulé sur un processeur séquentiel, et par conséquent les exécutions des programmes parallèles ADA proposés ne sont que des simulations séquentielles des exécutions parallèles possibles de ces programmes ; dans ce cadre, les notions de mémoires (physiquement) globale et locales du modèle PRAM ne sont pas pertinentes. Nous reviendrons sur cette remarque dans la conclusion du chapitre 5. Les chapitres suivants (notamment 6 et 7) définiront les outils nécessaires pour une implémentation réellement parallèle de ce modèle PRAM, outils qui doivent être utilisés pour la réalisation d'un compilateur des langages de types ADA sur machines physiquement parallèles.

On retiendra uniquement la définition donnée au paragraphe 2.1.1 du modèle PRAM pour concevoir et évaluer les algorithmes (nombre infini de processeurs synchrones dotés d'une mémoire globale partagée) et pour la programmation ADA, le shéma proposé ci-dessus.

4.2.2 Le travail d'un algorithme

Sur le modèle séquentiel RAM, deux caractéristiques (l'une temporelle, l'autre matérielle) sont considérées pour évaluer un algorithme séquentiel A_n résolvant P_n :

- le *temps*, noté $T_s(A_n)$, défini comme le nombre d'opérations effectuées sur des données bornées (ex : opérations flottantes, opérations sur des entiers machines, lecture ou écriture en mémoire d'un mot machine...). Pour les problèmes "faisables", ce temps séquentiel est un polynôme en n (classe $\mathcal{P}$).
- l'*espace mémoire*, noté $S(A_n)$ (S pour *space*[1]) défini comme le nombre de places en mémoire nécessaires à l'exécution de l'algorithme.

Par analogie, dans le cadre du calcul parallèle, on évalue un algorithme A'_n résolvant sur une PRAM le problème P_n grâce à deux caractéristiques, l'une matérielle, l'autre temporelle.

- La *surface* $H(A'_n)$ (H pour *hardware*), définie comme le nombre de processeurs utilisés lors du pas de calcul qui nécessite le plus de processeurs, est en général un polynôme en n.

1. Nous conservons ces notations anglosaxonnes – S ici, H plus bas – par souci de cohérence avec celles qui sont utilisées dans la plupart des publications.

– Le *temps* $T_{//}(A'_n)$ est le nombre de pas nécessaires à l'exécution de l'algorithme avec $H(A'_n)$ processeurs.

La plupart du temps, nous utiliserons $T_s(n)$, $S(n)$, $H(n)$ et $T_{//}(n)$ pour désigner ces grandeurs lorsqu'il n'y a pas d'ambiguité sur l'algorithme auxquelles elles se rapportent.

Remarque. Dans la suite, nous ne considèrerons les évaluations temporelles et matérielles qu'au niveau de leur ordre. Par abus de langage, les dénominations "efficace" ou "optimal" correspondent donc aux dénominations "asymptotiquement efficace" ou "asymptotiquement optimal" à cause des résultats donnés en O, donc connus à une constante multiplicative près.

A partir de ces deux critères, différentes quantités peuvent être introduites, qui seront utilisées par la suite.

Travail. On appelle *travail* d'un l'algorithme parallèle A_n la quantité notée $W(n)$ définie par :

$$W(n) = H(n)T_{//}(n)$$

Par extension, le travail d'un algorithme séquentiel (qui peut être vu comme un algorithme parallèle s'exécutant sur $H(n) = 1$ processeur) est défini comme son temps séquentiel.

Intuitivement, le travail d'un algorithme est l'aire d'un rectangle ayant pour côtés d'une part le nombre de processeurs qu'il utilise, et d'autre part son temps d'exécution. Dans le cas d'un algorithme séquentiel, il n'y a qu'un seul processeur. Si l'on considère le graphe de précédence de l'algorithme, dessiné de telle façon que les opérations exécutées sur la PRAM au premier pas soient sur une première ligne, puis celles exécutées au i-ème pas sur la i-ème ligne, ce graphe s'inscrit dans le rectangle comme la partie grisée de la figure 4.5. Il faut noter que dans l'algorithme parallèle, les $H(n)$ processeurs nécessaires de la PRAM exécutent à chaque instant une instruction :

– soit de l'algorithme proprement dit (partie grisée dans la figure),

– soit vide (partie blanche dans la figure).

Nous dirons qu'un algorithme parallèle est de *travail optimal* si son travail est du même ordre que le travail du meilleur algorithme séquentiel connu :

$$W(n) = O(T_s(n))$$

Parmi tous les problèmes, il est intéressant de déterminer les problèmes qui peuvent être résolus beaucoup plus rapidement en parallèle qu'en séquentiel. Un algorithme séquentiel s'exécute au moins en temps linéaire (sinon, il y a des entrées inutiles). Il apparaît donc naturel de considérer les problèmes qui peuvent être traités en parallèle en temps poly-logarithmique ($\log^{O(1)} n$), donc plus rapidement que tout algorithme séquentiel.

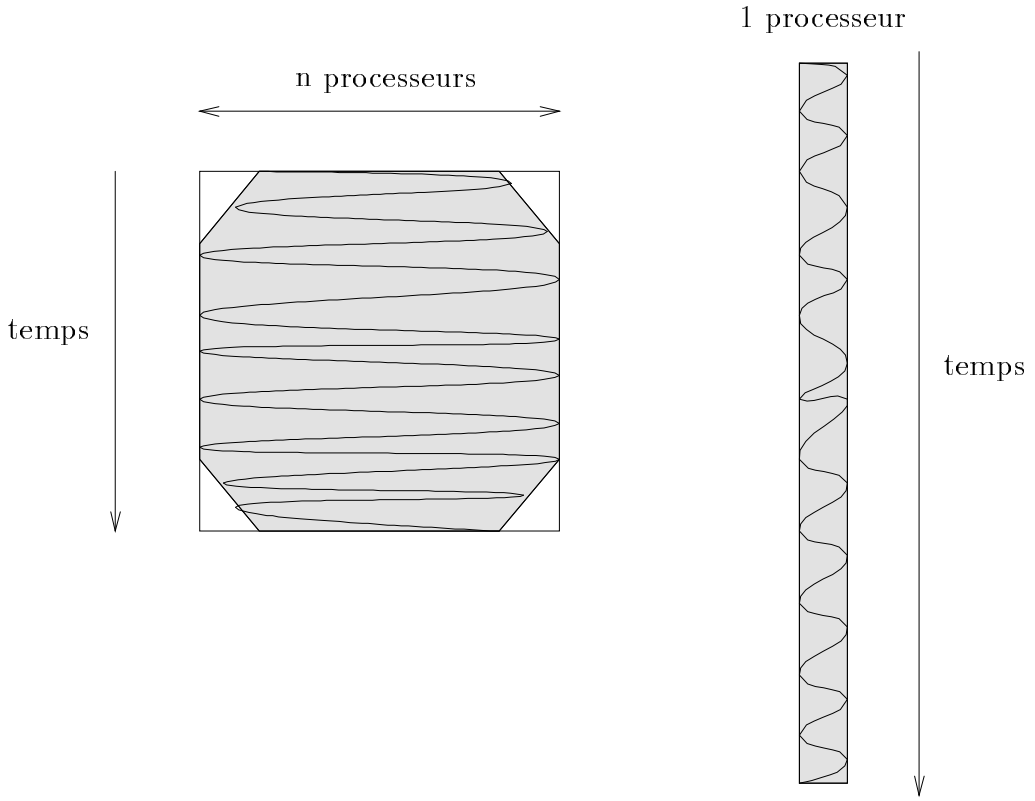


FIG. 4.5 – *Exemple de représentation du travail de n et de 1 processeur(s).*

Algorithme efficace Un algorithme parallèle est dit *efficace* si son temps d'exécution est poly-logarithmique ($O(\log^k n)$) et si son travail est le temps du meilleur algorithme séquentiel connu multiplié par un facteur poly-logarithmique :

$$\begin{cases} T_{//}(n) &= O(\log^{O(1)} n) \\ W(n) &= O(T_s(n) \log^{O(1)} n) \end{cases}$$

Un algorithme efficace permet donc d'obtenir un temps de résolution impossible à atteindre en séquentiel, avec un travail plus important mais raisonnable par rapport au meilleur algorithme séquentiel.

Algorithme optimal Un algorithme parallèle est dit *optimal* s'il est efficace et qu'il est de *travail optimal* (i.e. son travail est du même ordre que le travail du meilleur algorithme séquentiel connu) :

$$\begin{cases} T_{//}(n) &= O(\log^{O(1)} n) \\ W(n) &= O(T_s(n)) \end{cases}$$

Un algorithme parallèle optimal est donc particulièrement intéressant: non seulement il est très rapide en parallèle, mais sa simulation séquentielle (voir paragraphe 4.2.4 ci-après) n'est pas plus coûteuse (asymptotiquement tout au moins) que l'exécution du meilleur algorithme séquentiel.

Efficacité et accélération d'un algorithme parallèle Soit A_n^s le meilleur algorithme séquentiel connu résolvant P_n, et soit A_n^p un algorithme parallèle résolvant P_n. L'*efficacité* de l'algorithme A_n^p est définie comme le rapport du travail de A_n^s sur le travail de A_n^p :

$$e(A_n^{(p)}) = \frac{W(A_n^s)}{W(A_n^p)} = \frac{T_s(A_n^s)}{T_{//}(A_n^p) H(A_n^{(p)})}$$

Il ne faut pas confondre la notion d'algorithme efficace qui est une qualification d'un type d'algorithme et l'efficacité qui est une grandeur calculable pour tout algorithme. L'efficacité d'un algorithme efficace est égale à l'inverse d'un poly-logarithmique ($\log^{O(1)} n$ par définition.

L'*accélération* de l'algorithme A_n^p est définie comme le rapport du temps de A_n^s sur celui de A_n^p :

$$a(A_n^p) = \frac{T(A_n^s)}{T_{//}(A_n^p)}$$

Efficacité et accélération sont à manier avec précaution : le meilleur algorithme séquentiel connu est rarement l'algorithme prouvé le meilleur pour le problème P_n. C'est pourquoi, de manière expérimentale, on considère souvent comme algorithme séquentiel une simulation (plus ou moins bonne) de l'algorithme parallèle sur un seul processeur : mais dès lors, il est essentiel de préciser qu'il s'agit d'*efficacité expérimentale* et d'*accélération expérimentale*, et il est possible d'obtenir des efficacités supérieures à 1.

4.2.3 La classification $\mathcal{NC}$

Une question de base en calcul parallèle est de déterminer les problèmes intrinsèquement parallèles, c'est-à-dire qui peuvent être résolus "beaucoup plus" rapidement avec un "nombre raisonnable" de processeurs que séquentiellement avec un seul.

La classe $\mathcal{NC}$ [Coo 85][KR 90], formalisée par Nicholas Pippenger [Pip 79] (et nommée par Cook $\mathcal{NC}$ = "Nick's class"), est la classe des fonctions qui peuvent être résolues en temps poly-logarithmique (c'est-à-dire résolues plus rapidement qu'il ne faut de temps pour lire séquentiellement leurs entrées) sur une machine parallèle ayant un nombre polynomial de processeurs, autrement dit de surface raisonnable. $\mathcal{NC}$ peut donc être caractérisée par ces deux critères :

$$\left(\begin{array}{lcl} H(n) & = & n^{O(1)} \\ T_{//}(n) & = & \log^{O(1)} n \end{array} \right)$$

Une propriété fondamentale de $\mathcal{NC}$ est d'être *résistante* : elle reste la même quel que soit le modèle parallèle choisi.

Par définition, $\mathcal{NC}$ est un sous-ensemble de la classe $\mathcal{P}$ des fonctions qui peuvent être calculées séquentiellement en temps polynomial ; mais le résultat d'inclusion stricte de $\mathcal{NC}$ dans $\mathcal{P}$ qui est une conjecture, reste à démontrer.

De façon à distinguer plus précisément les problèmes à l'intérieur de $\mathcal{NC}$, $\mathcal{NC}$ est partitionnée en sous-classes. On distingue ainsi dans le cadre du modèle PRAM-CREW la classe $CREW^k$ des problèmes qui peuvent être résolus en temps $O(\log^k n)$ avec un nombre polynomial de processeurs d'une PRAM-CREW. $\mathcal{NC}$ est l'union pour k des problèmes dans $CREW^k$.

4.2.4 Parallèle versus séquentiel

Le but de ce paragraphe est d'affiner la comparaison entre un algorithme parallèle et un algorithme séquentiel résolvant un même problème.

Le principe de Brent.

On observera d'abord qu'il est toujours possible de simuler le comportement de plusieurs processeurs synchrones sur un seul : en ce sens, tout algorithme parallèle est exécutable séquentiellement. A l'inverse, si tout algorithme séquentiel peut être considéré comme un cas particulier (dégénéré) d'un algorithme parallèle s'exécutant sur un seul processeur (donc avec une surface de 1), on ne peut affirmer qu'en général il soit exécutable sur plusieurs processeurs en tirant parti du parallélisme (c'est à dire en améliorant son temps d'exécution). De ce point de vue, il est donc plus intéressant de disposer d'une version parallèle d'un algorithme, puisqu'on pourra l'exécuter sur un nombre quelconque de processeurs, y compris 1.

Plus précisément, on appelle simulation séquentielle d'un algorithme parallèle l'exécution sur un seul processeur des opérations - même vides - de l'algorithme parallèle. Pour ce faire, le processeur exécute d'abord toutes les opérations du premier pas de calcul de l'algorithme parallèle, puis toutes les opérations du deuxième pas, etc (voir figure 4.6). Par extension, étant donné un algorithme parallèle dont l'exécution est décrite sur $H(n)$ processeurs, on appelle simulation de cet algorithme sur $m < H(n)$ processeurs l'exécution sur les m processeurs des opérations de l'algorithme. Cette simulation se fait de façon analogue : on divise les opérations de chaque pas de calcul en m groupes d'opérations, chaque groupe étant à la charge de l'un des m processeurs. Les m processeurs exécutent séquentiellement les opérations d'un groupe, puis séquentiellement les groupes de chaque étapes.

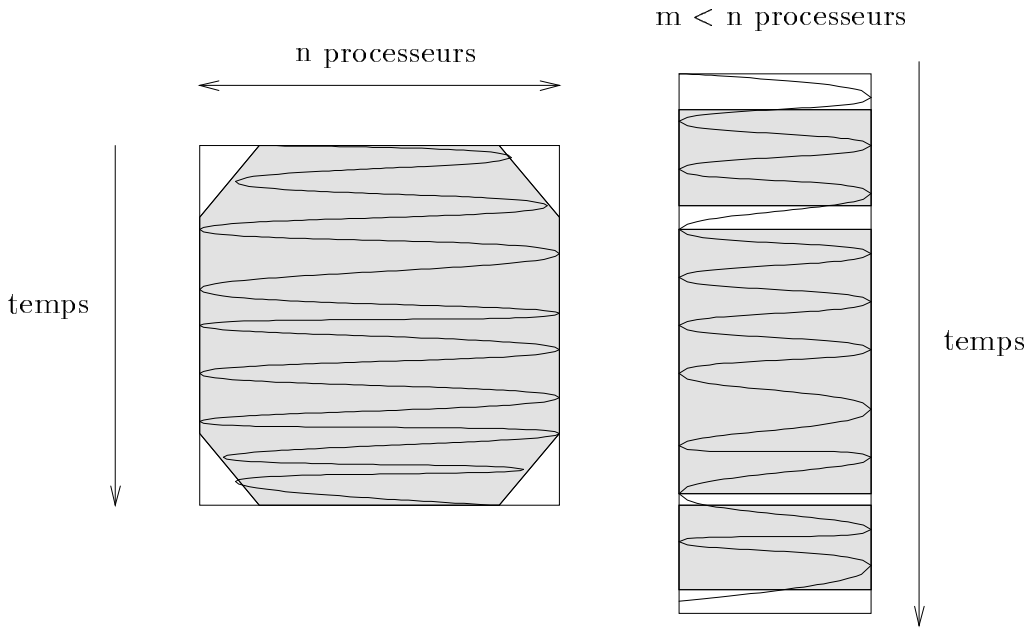


FIG. 4.6 – *Illustration d'une simulation de n processeurs sur $m < n$.*

On remarque que cette façon de faire peut induire pour les m processeurs des pas de calcul vide supplémentaires si m ne divise pas $H(n)$, mais ce nombre est au maximum de l'ordre de $T_{//}(n)$.

Le résultat essentiel est que le travail de l'algorithme parallèle est du même ordre que le temps d'exécution de sa simulation séquentielle, ou encore que le travail de sa simulation sur $m < H(n)$ processeurs. Cet énoncé est connu sous le nom de *principe de Brent*. Compte tenu de ce principe, le travail d'un algorithme parallèle est un invariant (en ordre au moins) lorsqu'on utilise un nombre de processeurs inférieur à $H(n)$.

Considérons A_n^s le meilleur algorithme séquentiel connu résolvant P_n. Alors, le principe de Brent peut également s'énoncer ainsi : le travail de tout algorithme parallèle A_n^p est supérieur à celui de A_n^s puisqu'au pire, le meilleur algorithme séquentiel serait l'algorithme dérivé de la simulation séquentielle de A_n^p dont on enlèverait les instructions vides. On peut donc écrire :

$$W(A_n^s) \leq W(A_n^p)$$

et pour tout m, $1 \leq m \leq H(A_n^p)$, l'algorithme A_n^p peut être exécuté en temps parallèle $O\left(mT_{//}(A_n^p)\right)$ avec $O\left(\frac{H(A_n^p)}{m}\right)$ unités de calcul.

C'est pourquoi la complexité de A_n^p sera dans la suite notée [notation inspirée de [KR 90]:

$$O_{//}\left(T_{//}(A_n^p), H(A_n^p)\right)$$

Ce principe de Brent est important car il montre aussi que le parallélisme explicité de façon théorique sur une PRAM peut être efficacement exploité sur une architecture de machine réelle à nombre fixé de processeurs.

A ce sujet, considérons un algorithme parallèle qui n'est pas de travail optimal : c'est le cas de l'algorithme de résolution d'un système matriciel triangulaire à partir de l'inversion de la matrice (voir chapitre 5), de complexité $O_{//}\left(\log^2 n, n^3\right)$, donc très rapide mais de travail non optimal. Ainsi il existe un algorithme séquentiel dont le temps (ici n^2) est négligeable devant le travail de l'algorithme parallèle (ici $n^3 \log^2 n$). Alors le temps de la résolution parallèle sur une architecture à p processeurs par cet algorithme parallèle (ici $\frac{n^3 \log^2 n}{p}$) sera, pour n assez grand, supérieur au temps séquentiel. Cet algorithme parallèle n'est intéressant que pour des petites valeurs de n ("petit" ici est défini par rapport à p). Un tel algorithme parallèle n'est donc pas portable sur une gamme extensible ("scalable" en anglais) de machines.

Par contre, l'implantation parallèle de l'algorithme introduit à partir du graphe de précédence (voir dans ce chapitre le paragraphe 4.3.2) est de complexité $O_{//}(n, n)$ et donc de travail optimal, et son temps de résolution parallèle sur une architecture à p processeurs est de l'ordre de $\frac{n^2}{p}$, ce qui est toujours inférieur au temps séquentiel.

Ainsi, indépendamment de l'appartenance à $\mathcal{NC}$ d'un algorithme parallèle, on s'intéressera en pratique surtout aux algorithmes de travail optimal. Un algorithme parallèle de travail optimal sera d'autant plus intéressant qu'il peut exploiter la disponibilité d'un grand nombre de processeurs (typiquement une fonction croissante de n). Ainsi, il sera possible (au moins pour des problèmes de grande taille) d'exploiter la présence de beaucoup de processeurs, et, indépendamment de ce nombre de processeurs ce qui assure une grande portabilité.

Pour reprendre l'exemple considéré ici, l'algorithme de complexité $O_{//}(n, n)$ pour la résolution matricielle triangulaire, peut *a priori* être intéressant pour un problème de taille n_0 donnée sur toute architecture comportant moins de n_0 processeurs ; ici, le parallélisme est surtout exploité pour diminuer le temps de résolution de systèmes de grande taille, donc pour des n_0 grands. Cet algorithme est donc très portable, à condition de pouvoir le simuler efficacement sur un nombre de processeurs $p < n_0$ donné.

4.2.5 Le mode parallèle versus le mode séquentiel

Avant d'essayer de construire des algorithmes, il est intéressant de préciser les relations entre temps et mémoire en séquentiel et temps et nombre de processeurs en parallèle. L'espace mémoire nécessaire à un algorithme séquentiel correspond à la taille de la pile qu'il est nécessaire d'utiliser pour pouvoir exécuter cet algorithme : cette taille est clairement en relation avec la dépendance qui existe entre les instructions successives de l'algorithme. Cette dépendance correspond en parallèle à autant de dépendances de données qui ne peuvent pas être éliminées : c'est pourquoi, l'espace mémoire séquentiel est lié au temps parallèle. De même, la surface parallèle est liée au nombre d'instructions qui devront être exécutées pour résoudre le problème, et donc cette surface parallèle est liée au temps séquentiel.

On a donc les relations suivantes : le temps parallèle est fortement corrélé à l'espace mémoire séquentiel, et le nombre de processeurs parallèles est fortement corrélé au temps séquentiel. Des relations précises (encadrements) ont été démontrées par Borodin [Bor 77]. Le résultat général est connu sous le nom de "*thèse du calcul parallèle*".

On peut visualiser cette propriété sur le problème du produit itéré, présenté dans la partie

précédente (paragraphe 4.1.1 et fig. 4.2). En séquentiel, la surface de cet algorithme -i.e. l'espace mémoire-, liée au parcours récursif, est égale à la profondeur de l'arbre ($\log n$), et la profondeur de cet arbre est justement égal au temps de l'algorithme parallèle. En ce qui concerne la surface parallèle - i.e. le nombre de processeurs - (n si l'on ne réutilise pas les processeurs et $n/2$ sinon), elle est très liée au temps séquentiel (dans ce cas, temps de parcours de l'arbre à n sommets).

Cette remarque est importante car elle montre que les approches algorithmiques séquentielle et parallèle sont fondamentalement différentes. Dans le cas séquentiel, on cherche d'abord à trouver un algorithme qui minimise le temps : puis, on essaye de réduire la surface de l'algorithme. Dans le cas parallèle, on procède exactement dans l'autre sens : on cherche d'abord à minimiser le temps parallèle (qui correspond à la surface séquentielle), puis on s'attache à réduire le nombre de processeurs (qui correspond essentiellement au temps séquentiel).

Il apparaît donc qu'à un algorithme séquentiel *rapide* (par exemple le produit de matrices par la méthode de Strassen), qui apporte un gain de temps par rapport à l'algorithme séquentiel standard, correspond un algorithme parallèle *économique*, c'est à dire qui utilisera pour un même temps d'exécution, moins de processeurs que l'algorithme parallèle standard (i.e. obtenu par parallélisation de l'algorithme séquentiel standard).

4.3 Méthodes de base pour la construction d'algorithmes parallèles

Différentes techniques peuvent être utilisées pour concevoir un bon algorithme parallèle. Les principales sont l'équilibre des travaux [Kal 89] (présentée ici sur le produit itéré, puis de manière plus fine sur le calcul des préfixes), l'analyse du graphe des dépendances (illustrée par l'élimination de Gauss) et l'introduction de redondance (montrée avec l'addition d'entiers). Ces méthodes seront utilisées et affinées dans le chapitre 5.

4.3.1 Equilibre des travaux : le produit itéré et les préfixes

Le produit itéré.

Nous avons vu (paragraphe 4.1.1 que le produit itéré se calcule séquentiellement en temps $O(n)$: cet algorithme est clairement optimal, puisqu'il faut déjà n étapes pour lire les entrées.

Nous avons proposé un algorithme parallèle de temps $O(\log n)$ avec n processeurs. Cet algorithme fonctionne sur la PRAM-CREW. Le problème du produit itéré appartient donc à $CREW^1$. Cet algorithme est efficace. Mais il n'est pas optimal, son travail $O(n \log n)$ étant d'un ordre supérieur au temps séquentiel. Nous allons montrer qu'il est cependant possible de le transformer pour obtenir un algorithme optimal, en faisant un bon compromis entre l'algorithme efficace et le meilleur algorithme séquentiel.

Considérons pour cela l'intérprétation du travail d'un algorithme parallèle comme étant la surface d'un rectangle ayant pour côtés d'une part le nombre de processeurs, et d'autre part le temps d'exécution. Lorsqu'un algorithme n'est pas optimal, cette surface n'est pas du même ordre de

grandeur que celle qui représente le travail de l'algorithme séquentiel : ceci provient des instructions vides exécutées par les processeurs à certains moments de l'exécution. L'idée de base dans la démarche d'"équilibre des travaux" est la suivante : on cherche à réduire la surface du rectangle en occupant à tout moment les processeurs. Pour celà, il faut modifier certaines phases de l'algorithme en tirant parti d'un "bon" algorithme séquentiel : à certaines phases de l'exécution, l'algorithme parallèle est remplacé par l'exécution en parallèle par chaque processeur de l'algorithme séquentiel. On parvient ainsi à réduire la surface du rectangle en supprimant des parties correspondant à des instructions vides. Le temps de l'algorithme parallèle reste du même ordre, mais le nombre de processeurs utilisés est plus faible.

Dans l'exemple du produit itéré, en comparant les algorithmes, on remarque que le travail parallèle est d'un facteur $O(\log n)$ supérieur au travail séquentiel. Le temps ($O(\log n)$ étant une borne inférieure sur le modèle CREW (les n données doivent inévitablement se rencontrer), nous allons essayer de garder ce temps (en ordre de grandeur) en réduisant le nombre de processeurs. L'obtention d'un algorithme optimal nécessite donc de n'utiliser que $O\left(\frac{n}{\log n}\right)$ processeurs. Cette diminution du nombre de processeurs peut être réalisée en groupant différentes opérations sur un même processeur, sur lequel sera alors exécuté -de manière optimale- l'algorithme séquentiel.

Groupons les n entrées en $\left(\frac{n}{\log n}\right)$ groupes ayant chacun $(\log n)$ éléments. A chaque groupe nous associons une tâche, qui calcule le produit itéré de son groupe par l'algorithme séquentiel. Comme il y a $(\log n)$ éléments dans chaque groupe, le temps de cette étape est $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs.

Puis, le produit itéré des $O\left(\frac{n}{\log n}\right)$ produits partiels ainsi obtenus peut être calculé par l'algorithme parallèle (4.1.1) en temps $O(\log n)$, en utilisant encore $O\left(\frac{n}{\log n}\right)$ processeurs. Le temps parallèle de ce nouvel algorithme est alors de :

$$T_{//}(n) = O(\log n)$$

et le nombre de processeurs de

$$H(n) = O\left(\frac{n}{\log n}\right).$$

La complexité de cet algorithme est donc

$$O_{//}\left(\log n, \frac{n}{\log n}\right)$$

et son travail est $O(n)$, c'est à dire du même ordre que le travail de l'algorithme séquentiel : il est donc *optimal*. Le programme ADA qui implante cet algorithme est donné dans le paragraphe suivant.

Le calcul des préfixes.

Le calcul des n préfixes d'une expression est une extension du problème précédent. Considérant n entrées $a_0, \ldots, a_{n-1}$ nous proposons de calculer les n sorties :

$$\Pi_0 = a_0, \Pi_1 = a_0 \star a_1, \ldots, \Pi_{n-1} = a_0 \star a_1 \star a_2 \star \cdots \star a_{n-1}$$

avec une loi $\star$ associative (+, x, *Max*, etc).

La solution séquentielle s'écrit naturellement :

$\Pi_0 := a_0$;
for i *in* 1..n-1 *loop* $\Pi_i := \Pi_{i-1} \star a_i$ *end loop* ;

Le temps nécessaire à l'exécution de cet algorithme est $O(n)$. Il est clairement optimal : à chaque pas de l'algorithme, on calcule un résultat qui apparaît en sortie. Néanmoins, il paraît essentiellement séquentiel : il est cependant dans $CREW^1$, et il existe même une solution parallèle optimale ! Autrement dit, il existe également une solution intrinsèquement parallèle.

Considérons la solution parallèle, en supposant $n = 2^p$ pour simplifier. L'algorithme correspondant peut être construit récursivement, par une approche "Diviser Pour Régner" :

- phase 1 : on réduit le problème en calculant en parallèle les produits des entrées groupées par 2. On calcule donc les quantités b_i :

$$b_0 = a_0 \star a_1, b_1 = a_2 \star a_3, \ldots, b_{\frac{n}{2}-1} = a_{n-1} \star a_n$$

- phase 2 : on résout le problème réduit en calculant (par un appel récursif des 3 phases décrites ici, jusqu'à obtenir un problème à deux entrées) les préfixes des $\frac{n}{2}$ valeurs b_i : (b_0), $(b_0 \star b_1$, ..., $(b_0 \star b_1 \star b_{\frac{n}{2}-1})$. Ainsi, on a calculé tous les Π_{2k+1} pour $k = 0, \ldots, \frac{n}{2} - 1$.

- phase 3 : à partir de la solution du problème réduit, on construit la solution du problème initial. Les résultats qui manquent sont tous ceux ayant un indice pair de termes : ils peuvent être obtenus en calculant : (a_0), $(\Pi_1 \star a_2)$, $(\Pi_3 \star a_4)$, ..., $\Pi_{2k-1} \star a_{2k}$, ... en parallèle en 1 pas de calcul.

Les différents calculs effectués en parallèle lors d'une même étape portent sur des données différentes : cet algorithme peut donc être exécuté sur une CREW-PRAM. Soit $T_{//}(n)$ le temps parallèle de cet algorithme avec $H(n)$ processeurs d'une CREW-PRAM. La complexité parallèle des différentes phases est (on se reportera à l'exemple donné figure 4.7) :

- phase 1 : en temps 1 en utilisant $\frac{n}{2}$ processeurs.

- phase 2 : en temps $T_{//}\left(\frac{n}{2}\right)$ avec $H\left(\frac{n}{2}\right)$ processeurs

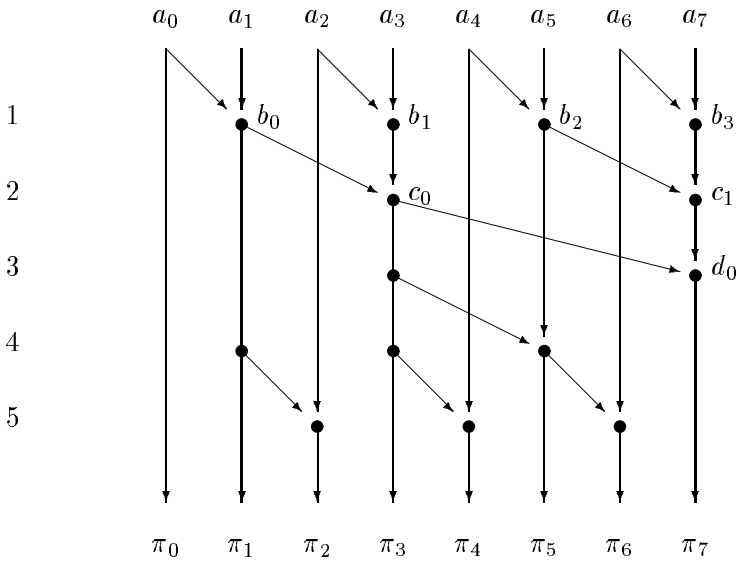
- phase 3 : en temps 1 en utilisant $\frac{n}{2}$ processeurs.

Finalement le temps total de l'algorithme est donné par la récurrence suivante, sachant que $T_{//}(2) = 1$:

$$T_{//}(n) = 2 + T_{//}\left(\frac{n}{2}\right) \quad \text{d'où} \quad T_{//}(n) = 2\log_2 n$$

et le nombre de processeurs nécessaires est, sachant que $H(2) = 1$:

$$H(n) = \text{Max}\left(\frac{n}{2}, H\left(\frac{n}{2}\right)\right) \quad \text{d'où} \quad H(n) = \frac{n}{2}$$



Etape 1 : Obtention de b_0, b_1, b_2 et b_3.
Etape 2 : Obtention de $c_0 = b_0 \star b_1$, et $c_1 = b_2 \star b_3$.
Etape 3 : Obtention de $d_0 = c_0 \star c_1$
Etape 4 : Obtention de b_0, $b_0 \star b_1$, $b_0 \star b_1 \star b_2$ et $b_0 \star b_1 \star b_2 \star b_3$.
Etape 5 : Obtention des préfixes : a_0, $a_0 \star a_1$, ..., $a_0, \star \cdots \star a_7$.

FIG. 4.7 – *Graphe d'exécution du calcul parallèle des préfixes.*

Le problème du préfixe parallèle appartient donc à $CREW^1$. Le graphe de précédence correspondant à cet algorithme est représenté sur la figure 4.7 pour $n = 8$. Les premier et dernier pas de calcul correspondent respectivement aux phases 1 et 3 de l'algorithme, et les pas intermédiaires à l'appel récursif développé dans la phase 2.

Un programme ADA qui implante cet algorithme est présenté au paragraphe 4.

Equilibre des travaux : Comme pour le produit itéré, il apparaît que l'algorithme obtenu est efficace mais non optimal. Nous allons donc reprendre la méthode d'équilibre des travaux décrite au paragraphe précédent. Divisons les entrées en $m = O\left(\frac{n}{\log n}\right)$ groupes ayant chacun $k = O(\log n)$ éléments. En parallèle, nous pouvons calculer les m quantités suivantes :

$A_0 = a_0 \star a_1 \star \cdots \star a_{k-1}$

$A_1 = a_k \star a_{k+1} \star \cdots \star a_{2k-1}$

$\ldots$

$A_{m-1} = a_{(m-1)k} \star a_{(m-1)k+1} \star \cdots \star a_{mk-1}$

Chaque calcul peut-être effectué en temps $O(\log n)$, grâce à l'algorithme séquentiel du produit itéré. Cette étape peut donc être calculée en temps $O(\log n)$ en utilisant $O\left(\frac{n}{\log n}\right)$ processeurs. Ayant obtenu les A_i, nous pouvons maintenant calculer leurs préfixes en utilisant l'algorithme parallèle décrit précédemment : cette étape prend un temps $O(\log n)$ en utilisant $O\left(\frac{n}{\log n}\right)$ processeurs. Nous obtenons ainsi les m termes B_i suivants :

$$B_0 = A_0, B_1 = A_0 \star A_1, \ldots, B_{m-1} = A_0 \star A_1 \star \cdots \star A_{m-1}$$

Il reste maintenant à calculer les termes manquants, c'est-à-dire :

$a_0, a_0 \star a_1, \ldots, a_0 \star a_1 \star \cdots \star a_{k-2}$

$B_0 \star a_k, A_0 \star a_k \star a_{k+1}, \ldots, B_0 \star a_k \star a_{k+1} \star \cdots \star a_{2k-2}$

etc

Pour ce faire, nous répartissons à nouveau la charge entre $m = O\left(\frac{n}{\log n}\right)$ processeurs. Le processeur i $(0 \leq i \leq m-1)$ calcule les valeurs :

$$B_{i-1} \star a_{ik}, B_{i-1} \star a_{ik} \star a_{ik+1}, \ldots, A_{i-1} \star a_{ik} \star \cdots \star a_{(i+1)k-2}$$

Le temps de cette étape est donc $k = O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs. Finalement, le temps parallèle total de ce nouvel algorithme est :

$$T_{//}(n) = O(\log n)$$

et le nombre de processeurs est :

$$H(n) = O\left(\frac{n}{\log n}\right).$$

La complexité de cet algorithme est donc :

$$O_{//}\left(\log n, \frac{n}{\log n}\right)$$

et son travail est $O(n)$, c'est-à-dire du même ordre que le travail de l'algorithme séquentiel : il est donc *optimal*.

Remarque. L'algorithme proposé ici ne fait intervenir aucune lecture concurrente; il fonctionne donc avec la même complexité sur une EREW-PRAM.

4.3.2 Graphe de dépendance et parallélisme : résolution d'un système linéaire triangulaire

Soit A une matrice $n \times n$ triangulaire inférieure inversible, à coefficients dans un corps K, et b un vecteur de K^n. Le problème est de déterminer le vecteur unique x de K^n tel que : $Ax = b$.

La parallélisation de ce problème est caractéristique de nombreux problèmes d'algèbre linéaire : la parallélisation des algorithmes séquentiels de factorisation (Gauss, Householder, Givens...) met en jeu les mêmes techniques.

La solution séquentielle s'exprime très facilement à partir de l'itération :

$$\text{Pour } i = 1 \ldots n \ : \quad x_i = \frac{b_i - \sum_{k=1}^{i-1} a_{i,k} x_k}{a_{i,i}}$$

Cet algorithme a une complexité $O(n^2)$ et est donc optimal (les n^2 coefficients de la matrice A devant être examinés).

Mais sa parallélisation semble difficile : il y a une dépendance très forte entre le calcul de x_i et celui de x_{i-1}. Néanmoins, pour chaque indice i, le calcul à effectuer est du type produit itéré, et cette phase peut être réalisé en temps $O_{//}\left(\log i, \frac{i}{\log i}\right)$. Les n phases s'effectuant séquentiellement, on obtient comme complexité globale sur une PRAM-CREW :

$$O_{//}\left(n \log n, \frac{n}{\log n}\right)$$

Pour trouver ce résultat, tous les $\log i$ sont majorés par $\log n$ et la mise en séquence implique l'addition des temps. Le nombre de processeurs nécessaires est égal au nombre maximum de processeurs nécessaires pour chacune des phases.

La parallélisation peut être améliorée si l'on regarde précisément le graphe de précédence des tâches de l'algorithme séquentiel. En effet, dès que x_i est calculé, il est possible de calculer en parallèle pour $k = i+1, \ldots, n$ tous les produits : $a_{k,i}x_i$, si l'on suppose que x_i peut être lu en parallèle par les processeurs chargés des calculs de x_k $(k = i+1, \ldots, n)$. En terme d'étapes, le schéma de calcul est alors le suivant (les calculs effectués en parallèle sur des processeurs différents sont séparés par "//") :

étape 1 : $x_1 = \frac{b_1}{a_{1,1}}$

étape 2 : $x_2 = \frac{b_2 - a_{2,1}x_1}{a_{2,2}}$ // $t_3 = b_3 - a_{3,1}x_1$ // $t_4 = b_4 - a_{4,1}x_1$ // ...

étape 3 : $x_3 = \frac{t_3 - a_{3,2}x_2}{a_{3,3}}$ // $t_4 = t_4 - a_{4,2}x_2$ // $t_5 = t_5 - a_{5,2}x_2$ // ...

étape 4 : $x_4 = \frac{t_4 - a_{4,3}x_3}{a_{4,4}}$ // $t_5 = t_5 - a_{5,3}x_3$ // $t_6 = t_6 - a_{6,3}x_3$ // ...

...

A chaque nouvelle étape (toutes les étapes sont effectuées en temps constant avec n processeurs), une nouvelle composante de x est calculée. Le vecteur x peut ainsi être calculé en temps $O(n)$ sur une CREW-PRAM de n processeurs.

L'algorithme donné ici présente l'avantage d'être de même travail en ordre que le meilleur algorithme séquentiel connu. Mais il ne permet pas de décider de l'appartenance à $\mathcal{NC}$ de ce problème car sa complexité en temps est $O(n)$. Nous verrons dans le prochain chapitre que par un tout autre algorithme, ce problème peut être en fait calculé très rapidement en temps $O(\log^2 n)$ sur une CREW-PRAM. Le problème considéré appartient donc bien en fait à $\mathcal{NC}$, mais même une bonne parallélisation de l'algorithme séquentiel, comme celle donnée ici, ne permet pas de le démontrer.

Remarque. L'algorithme proposé ici fait intervenir des lectures concurrentes; il ne fonctionne donc pas avec la même complexité sur une EREW-PRAM. Il est cependant possible de résoudre ce problème avec la même complexité sur une EREW-PRAM, en retardant certaines lectures (voir exercice 6).

4.3.3 Redondance et parallélisme : l'addition d'entiers

Une autre manière d'apporter du parallélisme dans un algorithme consiste à effectuer des calculs redondants, de façon à diminuer les dépendances de données, donc le temps. Cette démarche est contraire à celle utilisée en séquentiel où, lorsque deux calculs se ressemblent, on essaie de factoriser les termes communs pour ne les calculer qu'une fois. L'introduction de redondance est donc une approche souvent intéressante pour diminuer le temps de traitement parallèle, même si elle conduit, au premier abord tout au moins, à des algorithmes non optimaux. La duplication de certains calculs évités en séquentiel fait que le travail parallèle est souvent plus important. Nous allons expliciter cet apport de la redondance sur l'exemple de l'addition de deux entiers de n chiffres et cette technique générale sera exploitée différemment pour le tri dans le chapitre suivant.

On considère deux entiers A et B de n chiffres en base β, représentés par la séquence de leurs chiffres (poids forts en tête) :

$$A = [a_{n-1}, a_{n-2}, \ldots, a_0]_\beta = \sum_{i=0}^{n-1} a_i \beta^i \quad \text{avec} \quad \forall i \ : 0 \leq a_i < \beta$$

$$B = [b_{n-1}, b_{n-2}, \ldots, b_0]_\beta = \sum_{i=0}^{n-1} b_i \beta^i \quad \text{avec} \quad \forall i \ : 0 \leq b_i < \beta$$

et on cherche la séquence de $((n+1)$ chiffres : $R = [r_n, r_{n-1}, \ldots, r_0]_\beta$ représentant l'entier $R = A+B$.

L'algorithme séquentiel consiste à additionner deux à deux les chiffres de A et B. poids faibles d'abord, en faisant propager les éventuelles retenues (une retenue est soit nulle, soit égale à 1), ce qui peut être exprimé par la récurrence suivante :

$$\begin{cases} c_{-1} & = & 0 \\ r_k & = & (a_k + b_k + c_{k-1}) \mod \beta \\ c_k & = & (a_k + b_k + c_{k-1}) \text{ div } \beta \end{cases}$$

où *div* et *mod* désignent respectivement les opérations de division entière et de calcul de reste.

De cette relation de récurrence, il est très difficile d'extraire du parallélisme. La propagation de retenue rend cet algorithme très séquentiel : une nouvelle retenue c_k ne peut être calculée que si l'on connaît la retenue c_{k-1} précédente.

L'introduction de parallélisme nécessite donc d'éviter cette propagation séquentielle. Pour mieux étudier la précédence liée à cette propagation, l'idée est d'abord d'utiliser l'approche diviser pour régner de façon à n'avoir plus que deux calculs (i.e. un nombre borné d'instances de sous-problèmes) dépendants l'un de l'autre, puis de casser cette précédence en calculant *a priori* deux résultats possibles. Pour cela, on partitionne A et B en séparant poids forts (A_H et B_H - H pour "High"-) et poids faibles (A_L et B_L - L pour "Low"-). De même, on calcule le résultat R en séparant les poids forts (R_H) et les poids faibles (R_L). Deux calculs sont possibles pour R_H : l'un en supposant que la retenue entrante pour le calcul de $A_H + B_H$ est nulle, l'autre en supposant qu'elle est égale à 1. La sélection du bon résultat pourra être faite a posteriori, en regardant la retenue sortant effectivement du calcul de $A_L + B_L$. L'algorithme est comme suit :

phase 1 : en temps constant séparer en parallèle A en A_H, A_L et B en B_H, B_L.

phase 2 : calculer par trois appels récursifs en parrallèle $A_H + B_H$, $A_H + B_H + 1$ et $A_L + B_L$.

phase 3 : si $A_L + B_L$ génère une retenue fusionner les résultats de $A_L + B_L$ et $A_H + B_H + 1$; sinon fusionner$A_L + B_L$ et $A_H + B_H$. L'opération de fusion se fait en temps constant.

Il y a clairement ici redondance : les chiffres de poids fort de A et B sont additionnés deux fois. Mais cette redondance est intéressante puisqu'elle permet de rendre indépendants les trois appels à l'addition.

Le temps parallèle de ce nouvel algorithme sur une CREW-PRAM (il y a concurrence au niveau des lectures mais pas au niveau des écritures) est alors :

$$T_{//}(n) = T_{//}\left(\frac{n}{2}\right) + O(1) = O(\log n)$$

avec un nombre de processeurs :

$$H(n) = 3H\left(\frac{n}{2}\right) = O(n^{\log_2 3}) \simeq O(n^{1,58})$$

L'introduction de la redondance permet donc d'obtenir un temps parallèle très intéressant en $O(\log n)$.

Par contre le travail de cet algorithme - $O(n^{1,58} \log n)$ - est important devant la complexité séquentielle du problème - $O(n)$ -. L'algorithme parallèle obtenu est loin d'être *efficace*. Il permet cependant de montrer que l'addition d'entiers peut être résolu très rapidement en parallèle et appartient à la classe $\mathcal{NC}$. Nous verrons dans le prochain chapitre comment obtenir un algorithme efficace pour ce problème.

4.4 Programmation en ADA de quelques algorithmes

Le principe des algorithmes ayant été expliqué dans le paragraphe précedent, nous en donnons une traduction en ADA.

4.4.1 Le produit itéré par la méthode non optimale

On rappelle qu'il s'agit ici d'effectuer le produit itéré de n valeurs. Le programme parallèle ADA est conçu comme une fonction qui rend le résultat désiré. Cette fonction va générer à l'exécution n processus. Le tableau contenant les entrées (a_i) est partagé par tous les processus. Chaque processus reçoit en entrée les indices délimitant l'intervalle contenant les éléments dont il doit faire le produit. Si il y a deux éléments ou plus, il génère deux nouveaux processus pour faire les produits en parallèle. La structure de contrôle générale est donc la récursion.

```
generic
     type Element is private;                                   -- le type des éléments
     with function "*"( a, b: in Element) return Element;
                                                  -- la fonction produit de 2 elements
package PRODUIT_ITERE is
     type indice is new integer;                -- le type indice pour accéder à une séquence
     type Sequence is array (indice range <>) of Element;      -- le type séquence
     function ProduitItere (s: in Sequence) return Element;
end PRODUIT_ITERE;

package body PRODUIT_ITERE is

function ProduitItere (s: in Sequence) return Element is

     function ProduitParallele (indice_debut, indice_fin: in indice) return Element is
        task type TacheProduit is
           entry envoyer_donnees ( indice_debut, indice_fin: in indice);
           entry recevoir_resultat ( resultat: out Element);
        end TacheProduit;

        task body TacheProduit is
           res: Element;
           i, j: indice;
        begin
           accept envoyer_donnees ( indice_debut, indice_fin: in indice) do
              i:= indice_debut; j:= indice_fin;
```

```
            end envoyer_donnees;
            res := ProduitParallele (i,j);
            accept recevoir_resultat ( resultat: out Element) do
               resultat := res;
            end recevoir_resultat;
         end TacheProduit;

      begin
         if (indice_debut=indice_fin ) then return s (indice_fin );
         else
            CalculEnParallele:                                   -- bloc de calcul du produit itéré
            declare
               calcul1, calcul2 : TacheProduit;                  -- lancement de deux nouvelles tâches
               indice_milieu : indice;                           -- pour le découpage en deux intevalles
               res1, res2 : Element;                             -- les deux sous-résultats pour la fusion
            begin
               indice_milieu := (indice_debut+ indice_fin ) / 2;          -- Partition des entrées.
               calcul1.envoyer_donnees (indice_debut, indice_milieu);     -- Calcul en
               calcul2.envoyer_donnees (indice_milieu +1, indice_fin );   -- parallèle.
               calcul2.recevoir_resultat (res2);                          -- Réception des
               calcul1.recevoir_resultat (res1);                          -- résultats.
               return res1 * res2;                                        -- Calcul du résultat.
            end CalculEnParallele;
         end if;
      end ProduitParallele;
begin
      return ProduitParallele (s'first, s'last);
end ProduitItere;

begin
      null;
end PRODUIT_ITERE;
```

4.4.2 Le produit itéré optimal

Le programme ADA correspondant à cet algorithme est présenté ci-dessous. Par rapport au précédent, seule la partie calcul séquentiel change, notament lors de la condition d'arrêt pour le découpage récursif en processus.

```
generic
      type Element is private;                                   -- le type des éléments
      with function "*"( a, b: in Element) return Element;
                                                                 -- la fonction produit de 2 éléments
package PRODUIT_ITERE is
      type indice is new integer;                -- le type indice pour accéder à une séquence
      type Sequence is array (indice range <>) of Element;       -- le type séquence
```

```
    function ProduitItere (s: in Sequence) return Element;
end PRODUIT_ITERE;

package body PRODUIT_ITERE is

function ProduitItere (s : in Sequence) return Element is

    tg_max: integer;                                    -- taille du groupe pour un calcul séquentiel

    function ProduitParallele (indice_debut, indice_fin : in indice) return Element is
       task type TacheProduit is
          entry envoyer_donnees ( indice_debut, indice_fin : in indice);
          entry recevoir_resultat ( resultat : out Element);
       end TacheProduit;

       task body TacheProduit is
          res : Element;
          i, j, k : indice;
       begin
          accept envoyer_donnees ( indice_debut, indice_fin : in indice) do
             i := indice_debut; j:= indice_fin;
          end envoyer_donnees;
          res := ProduitParallele (i,j);
          accept recevoir_resultat ( resultat : out Element) do
             resultat := res;
          end recevoir_resultat;
       end TacheProduit;

    begin
       if (indice_fin - indice_debut < tg_max) then
          res := s(indice_debut) ;
          for k in (indice_debut+1, indice_fin) loop
             res := res * s(k) ;
          end loop;
          return res ;
       else
          CalculEnParallele :                                -- bloc de calcul du produit itéré
          declare
             calcul1, calcul2 : TacheProduit;                -- lancement de deux nouvelles tâches
             indice_milieu : indice;                         -- pour le découpage en deux intevalles
             res1, res2 : Element;                           -- les deux sous-résultats pour la fusion
          begin
             indice_milieu := (indice_debut+ indice_fin ) / 2;          -- 1. Partition des entrées.
             calcul1.envoyer_donnees (indice_debut, indice_milieu);     -- 2. calcul en
             calcul2.envoyer_donnees (indice_milieu +1, indice_fin );   -- parallèle
             calcul2.recevoir_resultat (res2);                          -- 3. réception des
             calcul1.recevoir_resultat (res1);                          -- résultats.
             return res1 * res2;                                        -- 4. Calcul du résultat.
          end CalculEnParallele ;
       end if;
    end ProduitParallele ;
begin
    tg_max := Log_base_2 (n) ;
```

```
    return ProduitParallele (s'first, s'last);
end ProduitItere;

begin
    null;
end PRODUIT_ITERE;
```

4.4.3 Le calcul des préfixes

Le programme suivant implémente exactement l'algorithme décrit dans ce chapitre -sans l'équilibre des travaux-, en respectant les notations quant au nom des variables. n est supposé égal à $2^k - 1$. Les tâches *Réduire* permettent de calculer les éléments b décrits dans la phase 1. Après l'appel récursif à la fonction *Préfixe*, la séquence `beta` contient les préfixes de la séquence `b` (phase 2). Les tâches *Construire* permettent de calculer les préfixes `pi` de la séquence initiale, à l'aide de `beta` (phase 3). L'exercice 4 montre que ce programme est du type EREW.

On remarque que la génération de processus est effectuée dynamiquement, ce qui fait passer la complexité temporelle du programme ADA à $O(\log^2 n)$ au lieu de $O(\log n)$ comme dans l'algorithme présenté. Cette remarque est générale et peu étonnante : en séquentiel, la programmation du préfixe récursif entraîne l'utilisation d'un espace mémoire de taille $O(\log n)$ alors qu'il peut facilement s'écrire itérativement avec un espace mémoire de 1. Cela illustre la remarque faite sur la comparaison entre espace mémoire séquentiel et temps parallèle.

Dans le cas présent, le problème pourrait être résolu si ADA permettait de créer en 1 pas de calcul un ensemble de taille polynômiale de processus à partir d'un même modèle de tâche en leur associant des données différentes. ADA permet de déclarer un tableau de tâches, mais il n'est pas possible d'associer des données propres à chacune des tâches du tableau. Il est donc nécessaire de lancer les processus un par un avec leurs arguments propres. L'exercice 3 propose une programmation où l'ensemble des processus est créé lors de l'initialisation, ce qui permet d'obtenir la complexité de l'algorithme "théorique" $O_{//}(\log n, n)$.

```
generic
    type Element is private;                                    -- le type des éléments
    with function "*"( a, b: in Element) return Element;
                                                    -- la fonction produit de 2 éléments
package PREFIXE_PAR is
    type indice is new integer;                  -- le type indice pour accéder à une séquence
    type Sequence is array (indice range <>) of Element;        -- le type séquence
    function Prefixe (a: in Sequence) return Sequence ;
end PREFIXE_PAR;

package body PREFIXE_PAR is
function Prefixe (a : in Sequence) return Sequence is
```

```
task type TacheProduit is
   entry donnees ( e1, e2 : in Element);
   entry resultat ( r : out Element);
end TacheProduit ;
n : indice := a'last - a'first +1;
b : Sequence (0..n/2-1);
beta : Sequence (0..n/2-1);
pi : Sequence(0..n-1);

task body TacheProduit is
   res, a, b : Element;
begin
   accept donnees ( e1,e2 : in Element) do a := e1; b := e2; end donnees ;
   res := a * b ;
   accept resultat ( r : out Element) do r := res; end resultat;
end TacheProduit ;

procedure Reduire ( nb_elements : in indice) is
   k : indice;
   calcul : array (0..nb_elements /2 -1) of TacheProduit ;
begin
   for k in 0..nb_elements /2-1 loop                          -- En parallèle en théorie
      calcul(k).donnees (a(2*k), a(2*k+1));
   end loop;
   for k in 0..nb_elements /2-1 loop                          -- En parallèle en théorie
      calcul(k).resultat (b(k) );
   end loop;
end Reduire ;

procedure Construire ( nb_elements : in indice) is
   k : indice;
   calcul : array (1..nb_elements /2 - 1) of TacheProduit ;
begin
   for k in 0..nb_elements /2-1 loop                          -- En parallèle en théorie
      pi(2*k+1):= beta(k);
   end loop;
   for k in 1..nb_elements /2-1 loop                          -- En parallèle en théorie
      calcul(k).donnees (beta(k-1), a(2*k));
   end loop;
   pi(0) := a(0);
   for k in 1..nb_elements /2-1 loop                          -- En parallèle en théorie
      calcul(k).resultat (pi(2*k ) );
   end loop;
end Construire ;

begin
   if n = 1 then pi(0):= a(0);                                -- condition d'arrêt
   elsif (n mod 2) = 1 then          -- n est impair : le dernier élément est traité à part.
      pi (0..n-2) := Prefixe(a(a'first..a'last-1));
      pi(n-1) := pi(n-2)* a(n-1) ;                  -- Le dernier préfixe manquant est calculé.
   else                  -- n est pair : on se ramène aux préfixes avec deux fois moins d'éléments
      Reduire(n);                                -- la séquence b a été calculée par effet de bord
      beta := Prefixe(b);                         -- la séquence beta contiendra les préfixes de b
```

```
            Construire(n);                    -- la séquence p a été calculée par effet de bord
         end if;
         return pi;
      end Prefixe ;
begin
      null;
end PREFIXE_PAR;
```

4.5 Conclusion

Nous disposons maintenant des outils de base pour nous lancer dans la programmation de problèmes en vraie grandeur. Comme en calcul séquentiel, il est essentiel de procéder d'abord à une étape de conception de l'algorithme. Là, il s'agit d'exploiter parallélisme de contrôle et de données, et d'affiner la qualité des algorithmes par des techniques de redondance ou d'équilibrage des travaux. Ensuite vient la programmation qui demande de bien posséder les primitives disponibles du langage.

Exercices

1. Montrer que le programme donné pour le produit itéré équilibré (4.4.2) peut s'exécuter sur une EREW-PRAM. Calculer sa complexité, en précisant le coefficient du terme de plus grand ordre (i.e. celui en $\log n$).

2. Modifier l'algorithme de calcul des préfixes proposé (4.4.3) pour qu'il implémente l'algorithme de travail optimal, en utilisant la même technique de programmation que celle proposée pour le produit itéré (4.4.2) : la structure du programme est conservée, seuls les tests d'arrêt de découpage dans les modèles de tâches *Réduire* et *Construire* sont modifiés.

3. Montrer que, dans le programme de calcul des préfixes 4.4.3, le coût de l'étape 1 par *Réduire* (et de la même façon de l'étape 3 par *Construire*) est dans ce programme de l'ordre de $\log n$ avec n processeurs. En déduire que ce programme s'exécute en temps $O(\log^2 n)$ avec n processeurs.
Expliquer comment écrire un programme ADA qui implémente l'algorithme proposé, et qui s'exécute bien en temps $O(\log n)$ avec $\frac{n}{\log n}$ processeurs.
Appliquer cette technique pour la programmation ADA de l'algorithme optimal de calcul des préfixes proposé.

4. Montrer que le programme de calcul des préfixes (4.4.3) peut s'exécuter sur une PRAM-EREW, même lorsqu'il est exécuté avec une hypothèse de temps asynchrone. Autrement dit, montrer qu'il n'est pas possible d'avoir simultanément plusieurs lectures ou écritures sur un même emplacement mémoire.

5. Pour le problème du calcul des préfixes, on propose la stratégie suivante: soit $(a_0, \ldots, a_{n-1})$ une séquence en entrée. On calcule récursivement en parallèle les préfixes de $S_1 = (a_0, \ldots, a_{\frac{n}{2}-1})$ et $S_2 = (a_{\frac{n}{2}}, \ldots, a_{n-1})$. Puis on multiplie à gauche en parallèle les préfixes obtenus pour S_2 par le dernier préfixe de la séquence S_1.

1. Montrer que cette stratégie appliquée directement conduit à un algorithme CREW.
2. Montrer qu'il s'exécute en temps $O(\log n)$ avec n processeurs.
3. Rendre l'algorithme EREW (en gardant bien sûr la même stratégie) et en conservant la même complexité. Les (a_i), $0 \leq i < n-1$ désignant les entréees du problème, on pourra considérer le problème du calcul de la suite suivante :

$$((\pi_0, \pi), (\pi_1, \pi), \ldots (\pi_{n-1}, \pi))$$

où $\pi_k = \prod_{i=0}^{k} a_i$ et $\pi = \pi_{n-1}$.

4. En appliquant la technique d'équilibre des travaux, rendre l'algorithme précédent optimal, i.e. construire un algorithme de complexité $O_{//}\left(\log n, \frac{n}{\log n}\right)$.

5. Ecrire un programme ADA implémentant l'algorithme de la question 3. Montrer que le programme ainsi obtenu s'exécute bien en temps $O(\log n)$ avec n processeurs, et non en $O(\log^2 n)$ comme le programme (4.4.3).

6. Construire un algorithme de complexité $O_{//}(n, n)$ pour la résolution d'un système linéaire triangulaire, qui respecte la contrainte EREW.

Bibliographie

[AHU 74] A.V. Aho, J. E. Hopcroft, J. D. Ullman, "The design and Analysis of Computer Algorithms", Addison-Wesley, 1974.

[BDG 89] J.L. Balcazar, J. Diaz, J. Gabarro, "Structural Complexity II", EATCS Monographs on Theoretical Computer Science, vol. 11, Springer-Verlag 1989

[Bor 77] A. Borodin, "On relating Time and Space to Size and Depth", SIAM Journal of computing, n° 5, p.733-744, 1977

[CT 93] M. Cosnard, D. Trystram, "Algorithmes et architectures parallèles", Intereditions 1993

[Coo 85] S.A. Cook, "A Taxonomy of Problems that have Fast Parallel Algorithms", Information and Control, vol. 64, pp2-22, 1985

[Csa 76] L. Csanky, "Fast Parallel Matrix Inversion Algorithms", SIAM J. of Computing, 5/4, 1976.

[CW 87]

D. Coppersmith, S. Winograd, "Matrix Multiplication via Arithmetic Progressions", Proc. 19th ACM Symp. Theory Comp. , p.1-6, 1987

[Kal 89] E. Kaltofen, "Parallel Algebraic Computing Design", Proc. Issac 89, Portland 1989

[Kog 81] P.M. Kogge, *"The Architecture of pipelined computers"*, Hemisphere Publishing, 1981.

[KR 90] R.M. Karp, V. Ramachandran, *"A survey of parallel algorithms for shared memory machines"*, Handbook of Theoretical Comp. Science, p. 869-941, North-Holland 1990

[Pip 79] N. Pippenger, "On Simultaneous Ressource Bounded", Proc. 20th Ann. IEEE Symp. on Fundations of Comp. Science, p 307-311, 1979

[Pip 90] N. Pippenger, "Communication Networks", Handbook of Theoretical Comp. Science, p. 805-833, North-Holland 1990

[Qui 88] M. J. Quinn, "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill, 1988.

[Ruz 81] W.L. Ruzzo, "On Uniform Circuit Complexity", Journal of Computer and System Sciences 22, 3, p. 365-383, 1981

[**Sch 71**] A. Schönhage, "Schnelle Berechnung von Kettenbruchentwicklungen", Acta Informatica vol 1, pp139-144, 1971.

[**SS 71**] A. Schönhage, V. Strassen, "Schnelle Multiplikation Grosser Zahlen", Computing vol. 7 p.281-292

[**Str 69**] V. Strassen, "Gaussain elimination is not optimal", Numerische Math, pp 54-56, 1969.

Chapitre 5

Construction d'Algorithmes Parallèles

Il y a essentiellement deux méthodes générales pour construire un algorithme parallèle, l'une consiste à détecter et à exploiter le parallélisme inhérent dans un algorithme séquentiel déjà existant, l'autre consistant à inventer un nouvel algorithme dédié au problème [Qui 88]. On a déjà vu que transformer de manière aveugle un algorithme séquentiel dans une forme parallèle est souvent une erreur. Par exemple, pour inverser une matrice carrée de dimension n, les algorithmes séquentiels intéressants conduisent à des algorithmes parallèles de coût temporel $O(n)$, alors qu'il existe des algorithmes parallèles beaucoup plus rapides qui s'exécutent en temps $O(\log^2 n)$.

Le chapitre précédent était consacré aux techniques de base pour transformer intelligemment un algorithme séquentiel, mais il n'en reste pas moins que certains algorithmes séquentiels contiennent très peu de parallélisme, alors que le problème peut lui-même être très parallèle. Nous allons dans ce chapitre nous intéresser à cet aspect des choses : comment faire l'analyse algorithmique d'un problème qui conduise à un algorithme très parallèle. Nous allons donc appliquer les techniques introduites au chapitre précédent (étude du graphe de précédence, découpe récursive, introduction de redondance) pour construire les algorithmes parallèles les plus performants possible.

Le premier travail algorithmique consiste à "mesurer" le parallélisme intrinsèque d'un problème, en essayant de prouver son appartenance à $\mathcal{NC}$ ou en le réduisant à un problème "standard" (P-complet par exemple). Cette démarche conduit souvent à construire des algorithmes qui, même s'ils peuvent être de temps polylogarithmiques, ne sont pas efficaces et encore moins optimaux car ils utilisent un nombre de processeurs très important. Le deuxième travail algorithmique consiste donc à limiter ce gaspillage de processeurs pour essayer de construire des algorithmes de travail optimal; souvent cette diminution du nombre de processeurs peut être réalisée en augmentant le temps de l'algorithme parallèle initial, pour tirer parti des meilleurs algorithmes séquentiels connus (cf. la technique d'équilibre des travaux introduite au chapitre 4).

La démarche parallèle s'apparente donc à la séquentielle, modulo l'inversion de la complexité espace-temps (cf chapitre 4, paragraphe 4.2.5), même si les techniques diffèrent.

De manière générale, les problèmes pour lesquels les sorties peuvent s'écrire formellement comme une expression algébrique "équilibrée" des entrées peuvent se paralléliser directement : la technique est du même type que celle utilisée pour le produit itéré ou le préfixe parallèle. Ces deux problèmes (sous une forme plus ou moins adaptée) apparaissent très souvent comme nous le verrons dans

ce chapitre. Cette formulation <*sorties*> = <*expressions équilibrées des entrées*> a permis la résolution de plusieurs problèmes. Malheureusement, l'expression des sorties en fonction des entrées est généralement très complexe. Ainsi, le calcul de x^{2^n}, calculable en séquentiel en temps n, est très difficile à paralléliser avec un nombre de processeurs polynomial. Ce schéma de calcul revient malheureusement dans de nombreux algorithmes.

Dans ce chapitre nous allons étudier trois types de problèmes :

- l'algèbre linéaire : la multiplication de matrices et l'algorithme de résolution de système linéaire présenté dans le chapitre précédent.
- la maîtrise de la redondance : l'introduction de redondance avait permis de résoudre partiellement le problème de l'addition d'entiers en construisant un algorithme très rapide mais non efficace. La limitation de cette redondance conduira à modifier cet algorithme pour en construire un qui soit optimal.
- le tri : différents algorithmes sont présentés, qui mettent en avant les techniques précédemment explicités : introduction de redondance, découpe récursive, analyse des dépendances et équilibre des travaux.

Parmi les problèmes importants et significatifs prouvés comme étant dans $\mathcal{NC}$ figurent [KR90] : le tri, les opérations arithmétiques ($+$, $-$, $*$, $/$) (entiers et polynômes), le pgcd de polynômes, certaines opérations matricielles (produit, inverse, déterminant, rang, formes normales), la reconnaissance des langages hors-contexte, la recherche d'un arbre de recouvrement minimal dans un graphe.

En conclusion de ce chapitre, nous aborderons le problème du passage d'un algorithme PRAM à celui d'un programme parallèle pouvant s'exécuter sur une architecture parallèle possèdant un certain nombre de processeurs.

5.1 Algèbre linéaire

5.1.1 Produit matrice-matrice et matrice-vecteur

Soient A et B deux matrices à coefficients bornés, c'est-à-dire sur lesquels les opérations arithmétiques peuvent être effectuées en temps constant, avec un résultat ayant une représentation de même taille (ex. flottants simple ou double précision).

Soit $C = AB$ leur produit : les n^2 coefficients $c_{i,j}$ de C peuvent être tous calculés indépendamment :

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

$c_{i,j}$ est donné par le produit scalaire de la ligne i de A par la colonne j de B. Ce calcul est tout à fait similaire au calcul du produit itéré : dans une première étape on effectue les produits de deux coefficients, en temps 1 sur n processeurs. Puis on effectue une somme itérée des n résultats

obtenus. Ceci peut être réalisé en temps $O(\log n)$ sur $O(\frac{n}{\log n})$ processeurs (grâce à l'équilibre des travaux).

Les n^2 coefficients de C pouvant être calculés en parallèle, on obtient finalement un algorithme de complexité

$$O_{//}\left(\log n, \frac{n^3}{\log n}\right)$$

Cet algorithme est optimal par rapport à l'algorithme classique de produit de matrices, puisque son travail est $O(n^3)$. Il peut être implémenté sur un modèle PRAM avec contraintes CREW : les coefficients des matrices en entrée sont lus de manière concurrente par les processeurs.

Dans le cas du produit d'une matrice de dimension n par un vecteur, on obtient de la même façon un algorithme optimal de complexité :

$$O_{//}\left(\log n, \frac{n^2}{\log n}\right)$$

Remarque. Il existe d'autres algorithmes séquentiels plus rapides pour le produit de matrices (par exemple, $O(n^{2,8})$ [Str 69] ou $O(n^{2,4})$ [CW 87]). Les parallélisations de ces algorithmes conduisent à des algorithmes parallèles de temps $O(\log n)$ avec un travail optimal par rapport à l'algorithme séquentiel initial.

5.1.2 Résolution de système linéaire triangulaire

Nous reprenons ici le problème abordé au chapitre 4. L'étude du graphe de précédence de l'algorithme séquentiel dit de "descente triangulaire" a permis d'obtenir un algorithme de complexité $O_{//}(n, n)$, donc de travail optimal. Mais cette étude ne permettait pas de conclure sur l'appartenance du problème à la classe $\mathcal{NC}$.

Pour aller dans ce sens, il faut contourner la dépendance qui existe entre chaque variable à calculer et qui implique un temps de calcul en $O(n)$.

La recherche d'un algorithme de type "Diviser pour paralléliser" nous conduit à découper la matrice en sous-blocs carrés de dimension $\frac{n}{2}$:

$$A = \begin{bmatrix} A_{1,1} & 0 \\ A_{2,1} & A_{2,2} \end{bmatrix}$$

ainsi que le vecteur inconnu x et le second membre b :

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

La matrice A étant supposée triangulaire inférieure et inversible, les matrices de dimension $\frac{n}{2}$ $A_{1,1}$ et $A_{2,2}$ le sont aussi. Cette remarque permet la construction directe d'un algorithme récursif :

1. Résoudre le système de dimension $\frac{n}{2}$: (S_1) $\quad A_{1,1}x_1 = b_1$

2. Résoudre le système de dimension $\frac{n}{2}$: (S_2) $\quad A_{2,2}x_2 = b_2 - A_{2,1}x_1$

Mais l'obtention d'un algorithme très parallèle nécessite de casser la précedence qui lie le calul de x_2 à celui de x_1 (pour le calcul du second membre du système $(S2)$).

Cette précédence peut être cependant réduite en redondance, si on calcule dès la premiére étape l'inverse $A_{2,2}^{-1}$ de $A_{2,2}$. En effet, dans ce cas on obtient $x_2 = A_{2,2}^{-1}(b_2 - A_{2,1}x_1)$. Mais calculer l'inverse de $A_{2,2}$ est -asymptotiquement- de même coût que le calcul de l'inverse A. Le problème de la résolution du système initial $Ax = b$ se ramène alors, par l'introduction de redondance, au calcul de l'inverse de A, puis au calcul de $x = A^{-1}b$. Cette dernière opération est un produit matrice-vecteur, qui peut (paragraphe 5.1.1) se faire très rapidement en parallèle.

La matrice A^{-1} s'écrit directement à partir de la découpe récursive de la matrice A :

$$A^{-1} = \begin{bmatrix} A_{1,1}^{-1} & 0 \\ -A_{2,2}^{-1}A_{2,1}A_{1,1}^{-1} & A_{2,2}^{-1} \end{bmatrix}$$

(il suffit d'effectuer le produit de la matrice A par la matrice ci-dessus pour remarquer que l'on obtient l'idendité).

Le calcul de A^{-1} se fait donc de manière très simple :

1. En parallèle : calcul des inverses $A_{1,1}^{-1}$ et $A_{2,2}^{-1}$
2. Calcul de la matrice produit $A_{2,2}^{-1}A_{2,1}A_{1,1}^{-1}$

La deuxième étape s'effectue avec une complexité (paragraphe 5.1.1) :

$$O_{//}\left(\log n, n^3\right)$$

On en déduit donc le temps parallèle pour calculer l'inverse :

$$T_{Inv//}(n) = T_{Inv//}\left(\frac{n}{2}\right) + \log n = \log n + \log\left(\frac{n}{2}\right) + \log\left(\frac{n}{4}\right) + \cdots = O(\log^2 n)$$

avec un nombre de processeurs :

$$H_{Inv}(n) = Max\left(2H\left(\frac{n}{2}\right), n^3\right) = n^3$$

(le premier membre du maximum croît linéairement, alors que le deuxième croît polynomialement).

Le calcul de l'inverse de A se fait donc avec une complexité parallèle

$$O_{//}\left(\log^2 n, n^3\right)$$

On en déduit que le calcul de x pour la résolution du système initial $Ax = b$ se fait avec la même complexité.

Il est à noter que l'algorithme d'inversion proposé ici est efficace, et peut être rendu optimal par un équilibre des travaux. Par contre, l'algorithme de résolution de système proposé est de travail

$O(n^3)$ alors que l'algorithme séquentiel initial est de complexité $O(n^2)$: cet algorithme n'est donc pas efficace. Son utilisation pratique pour la résolution de systèmes triangulaires présente donc un inconvénient majeur (voir principe de Brent, 4.2.4) : l'algorithme ainsi construit n'est pas de travail optimal, alors que celui présenté dans le chapitre 4 l'était. Par conséquent, sa simulation sur $p \leq n$ processeurs est bien moins performante – temps $= \frac{n^3}{p}$ – que la simulation de l'algorithme du chapitre 4 sur le même nombre de processeurs – temps $= \frac{n^2}{p}$ –.

Cet algorithme est cependant intéressant : il montre que le calcul de l'inverse d'une matrice triangulaire est dans $\mathcal{NC}$ (et même dans NC^2).

Remarque. Ce résultat de complexité temporelle est valide pour de nombreux problèmes d'algèbre lineaire, notamment l'inversion de matrices denses ([Csa 76] [KP 91]) et les formes normales de matrices ([RV 93]).

5.2 Retour sur l'addition d'entiers

5.2.1 Maîtriser la redondance

L'introduction de redondance sur le problème de l'addition d'entiers (voir chapitre 4) a permis d'obtenir un algorithme s'exécutant en temps $O(\log n)$ mais avec un nombre important de processeurs : $n^{1,58}$.

Cette technique a permis de casser le mécanisme de propagation de retenues, qui limitait le parallélisme. Il reste à étudier comment supprimer certains calculs redondants tout en gardant un haut degré de parallélisme.

Soient $A = [a_{n-1}, a_{n-2}, \ldots, a_0]_\beta$ et $B = [b_{n-1}, b_{n-2}, \ldots, b_0]_\beta$ deux entiers de n chiffres en base β (on supposera que n est une puissance de 2), représentés par la séquence de leurs chiffres, poids forts en tête (i.e. $A = \sum_{i=0}^{n-1} a_i \beta^i$ et $B = \sum_{i=0}^{n-1} b_i \beta^i$).

Notons :

$$A_L = [a_{\frac{n}{2}-1}, \ldots, a_0]_\beta \qquad A_H = [a_{n-1}, \ldots, a_{\frac{n}{2}}]_\beta$$
$$B_L = [b_{\frac{n}{2}-1}, \ldots, b_0]_\beta \qquad B_H = [b_{n-1}, \ldots, b_{\frac{n}{2}}]_\beta$$

et posons :

$$r = A_L + B_L = [c, r_{\frac{n}{2}-1}, \ldots, r_0]_\beta \qquad (c \in \{0,1\})$$
$$R = A_H + B_H = [R_{\frac{n}{2}}, R_{\frac{n}{2}-1}, \ldots, R_0]_\beta \qquad (R_{\frac{n}{2}} \in \{0,1\})$$

On peut remarquer la propriété suivante :

- si $(c = 0)$ alors : $A + B = [R_{\frac{n}{2}}, R_{\frac{n}{2}-1}, \ldots, R_0, r_{\frac{n}{2}-1}, \ldots, r_0]_\beta$
- si $(c = 1)$ alors : $A + B = [R_{\frac{n}{2}}, R_{\frac{n}{2}-1}, \ldots, R_{i+1}, 1 + R_i, 0, \ldots, 0, r_{\frac{n}{2}-1}, \ldots, r_0]_\beta$ où i est tel que :

$$\begin{cases} R_k = \beta - 1 & \text{pour } k = 0, \ldots i-1 \\ R_i \leq \beta - 2 \end{cases}$$

Dans le cas où c est nul, le calcul peut se faire facilement en parallèle. Dans le cas où $c = 1$, tout le problème est de déterminer l'indice i sur lequel la retenue va s'arrêter. Cet indice est caractérisé par :

$$i = \text{Min}\{k/R_k < \beta - 1\}$$

Posons :

$$t_k = \begin{cases} k & \text{si } R_k < \beta - 1 \\ +\infty & \text{sinon} \end{cases}$$

i est alors égal au minimum des t_k, et peut être obtenu par une opération de type produit itéré, avec comme loi associative de base le minimum de deux entiers. Une fois i calculé, il est facile d'obtenir les poids forts de $A + B$ à partir de R en utilisant la propriété ci-dessus.

Le schéma de l'algorithme est alors le suivant :

1. calcul de $r = A_L + B_L = [c, r_{\frac{n}{2}-1}, \ldots, r_0]_\beta$ $(c \in \{0, 1\})$ et de $R = A_H + B_H = [R_{\frac{n}{2}}, R_{\frac{n}{2}-1}, \ldots, R_0]_\beta$ $(R_{\frac{n}{2}} \in \{0, 1\})$
2. calcul de t_k pour $k = 0, \ldots, \frac{n}{2}$ – coût : $O_{//}(1, n)$ –
3. calcul de $i = \text{Min}\{t_k; k = 0, \ldots, \frac{n}{2}\}$: produit itéré – coût : $O_{//}\left(\log n, \frac{n}{\log n}\right)$ –
4. formation de $A + B$:

   si $c = 0$ alors $A + B = [R_{\frac{n}{2}}, R_{\frac{n}{2}-1}, \ldots, R_0, r_{\frac{n}{2}-1}, \ldots, r_0]_\beta$

   sinon $A + B = [R_{\frac{n}{2}}, R_{\frac{n}{2}-1}, \ldots, R_{i+1}, 1 + R_i, 0, \ldots, 0, r_{\frac{n}{2}-1}, \ldots, r_0]_\beta$

Le coût en processeurs de cet algorithme sur une PRAM-EREW est donc :

$$H(n) = \text{Max}\left(2H\left(\frac{n}{2}\right), \frac{n}{\log n}\right) = O\left(\frac{n}{\log n}\right)$$

et le temps est :

$$T(n) = T\left(\frac{n}{2}\right) + \log n = O(\log^2 n)$$

On a donc une complexité totale :

$$O_{//}\left(\log^2 n, \frac{n}{\log n}\right)$$

sur une PRAM-EREW.

On a donc énormément gagné en nombre de processeurs, mais on a perdu un peu en temps : néanmoins, l'algorithme obtenu est efficace.

5.2.2 Un algorithme optimal pour l'addition d'entiers

La perte en temps mentionnée ci-dessus est essentiellement dûe au fait que, pour chaque étape d'additions (il y en a $\log n$), on calcule une position de butée pour la retenue.

Pour lever cette contrainte, on observe que l'on peut calculer une fois pour toutes les valeurs de chacune des retenues qui apparaissent dans l'itération séquentielle présentée au chapitre 4 :

$$\left\{\begin{array}{lcl} c_{-1} & = & 0 \\ r_k & = & (a_k + b_k + c_{k-1}) \bmod \beta (k = 0, \ldots, n) \\ c_k & = & (a_k + b_k + c_{k-1}) \text{ div } \beta (k = 0, \ldots, n) \\ r_n & = & c_n \end{array}\right.$$

Dans ce but, on établit les équations algébriques qui dirigent le calcul des retenues : c_k ne peut valoir que 1 ou 0, et elle ne peut valoir 1 que si $a_k + b_k \geq \beta$ (génération de retenue), ou si c_{k-1} valait 1 et $a_k + b_k = (\beta - 1)$ (propagation de retenue).
Soit g_k (pour génération) et p_k (pour propagation) les deux booléens suivants :

$$g_k = 1 \quad \text{ssi } a_k + b_k \geq \beta$$

$$p_k = 1 \quad \text{ssi } a_k + b_k = \beta - 1$$

Les équations définissant les retenues (qui peuvent être considérées comme des booléens) sont alors :

$$c_{-1} = 0$$

$$c_k = g_k \vee (p_k \wedge c_{k-1})$$

Soit la fonction booléenne affine suivante :

$$T_k(x) = g_k \vee (p_k \wedge x)$$

On a :

$$c_k = T_k(T_{k-1}(\cdots(T_0(0))\cdots))$$

Or la composition de deux fonctions affines est stable et associative. Posons :

$$U_k = T_k \circ T_{k-1} \circ \cdots \circ T_0$$

U_k s'écrit comme une fonction de la forme :

$$U_k(x) = u_k \vee (v_k \wedge x)$$

et les fonctions U_k correspondent aux préfixes des fonctions T_k, avec comme loi associative la composition de deux fonctions booléennes.
En utilisant un algorithme de préfixe parallèle, on en déduit que U_k (c'est à dire ses coefficients u_k et v_k) peuvent être calculés avec un coût :

$$O_{//}\left(\log n, \frac{n}{\log n}\right)$$

La retenue $c_k = U_k(0)$ est alors :

$$c_k = u_k \vee (v_k \wedge 0) = u_k$$

Le schéma de l'algorithme est alors le suivant :

1. Calcul de $s_i = a_i + b_i$ $(i = 0, \ldots, n-1)$ – coût : $O_{//}(1, n)$ –

2. calcul de $g_i = 1$ si $(s_i \geq \beta)$, 0 sinon
   calcul de $p_i = 1$ si $(s_i = \beta - 1)$, 0 sinon – coût : $O_{//}(1, n)$ –

3. Calcul de u_k et v_k (préfixe parallèle) – coût : $O_{//}\left(\log n, \frac{n}{\log n}\right)$ –
   On a alors $c_{-1} = 0$ et $c_k = u_k$ $(k = 0, \ldots, n-1)$.

4. Calcul de $r_i = (s_i + c_{i-1}) \bmod \beta$, $(i = 0, \ldots, n-1)$ et $r_n = c_{n-1}$ – coût : $O_{//}(1, n)$ –

On a donc ainsi obtenu un algorithme optimal pour l'addition de deux entiers, s'exécutant en temps $\log n$ avec $\frac{n}{\log n}$ processeurs d'une EREW-PRAM.

5.3 Tri Parallèle

Un ordinateur passant en moyenne 25% de son temps à trier, on comprend l'intérêt porté par les chercheurs aux algorithmes de tri, y compris parallèles et ce dès 1954 [Knuth 3 p.227]. En séquentiel, la complexité de ce problème est $\Omega(n \log n)$ (borne inférieure atteinte, par exemple par partition-fusion).

Nous construisons ci-dessous, à partir de la méthode séquentielle du tri par insertion, un premier algorithme parallèle qui, bien que non efficace, permet de montrer que ce problème peut être traité en temps $\log n$. Puis, nous donnons deux algorithmes efficaces de tri parallèle de temps $\log^2 n$, construits à partir de l'algorithme séquentiel partition-fusion.

5.3.1 Tri par sélection

On se donne n éléments e_k $(k = 0, \ldots, n-1)$ d'un ensemble E totalement ordonné, et on voudrait obtenir les n éléments triés dans un tableau. Nous supposerons ici les éléments distincts, mais il est facile d'étendre cet algorithme au cas général.

La méthode séquentielle immédiate, même si ce n'est pas la plus efficace, est celle du tri par insertion : à chaque étape on prend un nouvel élément que l'on essaie de classer parmi les précédents éléments que l'on a déjà triés. Le coût séquentiel est $O(n^2)$. Exploitée sous cette forme en parallèle, cette méthode ne permet pas d'obtenir un temps parallèle inférieur à n, l'élément d'indice i dans le tableau initial ne pouvant être inséré dans le tableau trié que lorsque tous les éléments d'indices inférieurs ont été ordonnés.

Il est cependant possible de casser cette dépendance par l'introduction de redondance. En effet, l'élément e_i peut être directement comparé à tous les autres éléments: le nombre d'éléments qui lui sont inférieurs indique la position de l'élément e_i dans le tableau trié.

La comparaison de deux éléments étant effectuée en temps constant, l'algorithme se déroule en deux phases :

1. chaque élément e_k est comparé aux $n-1$ autres éléments. Pour cela, à chaque élément e_k est associé un groupe G_k de $n-1$ processeurs $G_{k,i}$ $(i \neq k)$: chaque processeur $G_{k,i}$ compare e_k

à l'élément e_i. Le résultat de cette comparaison est 1 si e_k est strictement plus grand que e_i, et 0 sinon. La complexité de cette étape est donc $O_{//}(1, n^2)$ sur une CREW-PRAM.

2. dans chaque groupe G_k, on effectue la somme des $n-1$ valeurs trouvées : on obtient ainsi la position de l'élément e_k dans la séquence triée.

Chaque calcul effectué dans la phase 2 se réduit donc à l'addition de n bits. Cette opération peut être réalisée par un algorithme du type produit itéré, avec comme opération de base la somme de deux nombres ayant au plus $\log n$ bits. D'après ce qui a été vu précédemment, cette somme est de complexité $O_{//}\left(\log\log n, \frac{\log n}{\log\log n}\right)$ sur une EREW-PRAM. Compte-tenu qu'une somme itérée avec des opérations en temps constant peut être réalisée en temps $\log n$ avec $\frac{n}{\log n}$ processeurs, le coût de cette étape est donc pour chaque groupe G_k : $O_{//}\left(\log n \log\log n, \frac{n}{\log\log n}\right)$, soit $O_{//}\left(\log n \log\log n, \frac{n^2}{\log\log n}\right)$ pour tous les groupes. Le travail total pour cette phase est alors $n^2 \log n$: la parallélisation de l'algorithme de tri par insertion n'est donc pas optimale.

Néanmoins il est possible de calculer la phase 2 en complexité $O_{//}\left(\log n, \frac{n^2}{\log n}\right)$ en utilisant la technique dite du *2 pour 3* . En effet, l'utilisation du produit itéré dans la phase 2 ignore le fait que les entrées sont sur 1 bit seulement : la complexité serait la même si le problème était l'addition de n nombres de $\log n$ bits. Or, additionner 3 nombres de k bits peut se ramener à additionner 2 nombres de $k+1$ bits. Soient :

$$a = [a_{k-1}, a_{k-2}, \ldots, a_0] \qquad b = [b_{k-1}, b_{k-2}, \ldots, b_0] \text{ et } c = [c_{k-1}, c_{k-2}, \ldots, c_0]$$

trois entiers de k bits dont on désire calculer la somme.

Pour tout i, $a_i + b_i + c_i$ est un entier de deux bits : soit u_i son bit de poids fort, v_i son bit de poids faible, et soient u et v les deux entiers de $k+1$ bits :

$$u = [u_{k-1}, u_{k-2}, \ldots, u_0, 0] \quad \text{et} \quad v = [0, v_{k-1}, v_{k-2}, \ldots, v_0]$$

On a alors (propriété du 2 pour 3) :

$$a + b + c = u + v$$

Additionner n nombres de 1 bit se ramène donc à additionner $\left(\frac{2n}{3}\right)$ nombres de 2 bits, et en réitérant cette réduction $(\log_{\frac{3}{2}} n)$ fois, deux nombres de $O(\log n)$ bits. Cette dernière addition peut être effectuée avec un coût $O_{//}\left(\log\log n, \frac{\log n}{\log\log n}\right)$. Le coût de la phase 2 est donc, après équilibre des travaux, $O_{//}\left(\log n, \frac{n}{\log n}\right)$ sur une EREW-PRAM pour chaque groupe, soit $O_{//}\left(\log n, \frac{n^2}{\log n}\right)$ pour tous les groupes.

De façon à uniformiser le modèle de calcul PRAM utilisé, il est possible de transformer la première étape (beaucoup plus courte que la deuxième, avec le même nombre de processeurs) pour pouvoir l'exécuter sur une EREW-PRAM au lieu d'une CREW. Ceci peut être réalisé en dupliquant chacune des n données pour en obtenir un exemplaire différent dans chaque groupe de processeurs : cette opération correspond à un produit itéré, avec comme opération de base une écriture en mémoire partagée (duplication). Le coût de la première étape devient alors $O_{//}\left(\log n, \frac{n^2}{\log n}\right)$ sur une EREW-PRAM.

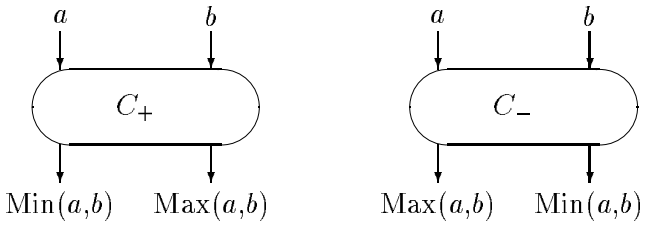


FIG. 5.1 – *Les deux comparateurs C_+ et C_-.*

Finalement, la complexité globale de cet algorithme de tri est $O_{//}(\log n, n^2)$ sur une EREW-PRAM : son temps est particulièrement intéressant. De plus, la parallélisation est optimale par rapport à l'algorithme séquentiel de départ, le tri par insertion. Mais le nombre de processeurs est très important et quoiqu'il appartienne à la classe $\mathcal{NC}$, il n'est pas efficace. Pour obtenir un algorithme de travail plus intéressant – de travail $O(n \log n)$ –, il est nécessaire de reconsidérer le problème en partant d'algorithmes meilleurs que le tri par insertion.

5.3.2 Tri par fusion et réseaux de tri

Un *réseau de tri* (Batcher, 1968) est le graphe de précédence d'un algorithme de tri dont la seule primitive est la comparaison de deux éléments (supposée effectuée en temps constant). Nous supposerons donc disposer d'un *comparateur* qui prend en entrée deux éléments quelconques, et qui les retourne triés en sortie. Nous utiliserons deux types de comparateurs, décrits en figure 5.1.

De nombreux algorithmes séquentiels de tri, et notamment ceux qui permettent d'atteindre la borne inférieure $O(n \log n)$, utilisent comme primitive de base la fusion de deux listes triées. Nous allons étudier la parallélisation de l'algorithme dit de partition-fusion, en étudiant surtout le problème posé par la fusion parallèle.

Fusion séquentielle

Soient $S_1 = (e_0, \ldots, e_{\frac{n}{2}-1})$ et $S_2 = (e_{\frac{n}{2}}, \ldots, e_{n-1})$ une partition de la séquence des n éléments en entrée en deux sous-séquences de $\frac{n}{2}$ éléments. L'algorithme récursif de tri par fusion consiste à trier les deux sous-séquences S_1 et S_2, puis à fusionner les deux séquences trieés T_1 et T_2 obtenues.

La complexité séquentielle de la *partition* est de *1* (avec un bon choix de structures de données), et celle de la *fusion* est de n. On en déduit que le temps de l'algorithme ci-dessus est :

$$T_{seq}(n) = 2T_{seq}\left(\frac{n}{2}\right) + n = O(n \log n)$$

L'analyse du graphe de dépendance de cet algorithme montre que les deux appels récursifs pour trier les deux sous-séquences S_1 et S_2 peuvent être exécutés en parallèle sur une EREW-PRAM.

Tout en gardant les procédures *Partition* et *Fusion* séquentielles, on peut donc construire un algorithme parallèle de tri prenant en compte l'indépendance de ces deux appels. La complexité

parallèle de cet algorithme, avec une fusion séquentielle en $O(n)$ est alors :

$$T_{//}(n) = T_{//}\left(\frac{n}{2}\right) + O(n) = O(n)$$

$$H(n) = \text{Max}\left(2H\left(\frac{n}{2}\right), 1\right) = O(n)$$

Cet algorithme n'est pas de travail optimal, le travail séquentiel étant en $O(n \log n)$. Il est donc nécessaire d'appliquer la technique d'équilibre des travaux pour obtenir un algorithme de travail optimal. Le seul gain apporté par la parallélisation est ici basé sur l'indépendance des deux appels récursifs. Or il ne sert à rien d'effectuer ces deux appels en parallèle si leur exécution séquentielle est de toute façon de temps supérieur au temps de la dernière fusion séquentielle, à savoir $O(n)$.

Or, pour un tri de $\left(\frac{n}{\log n}\right)$ éléments, $T_{seq}\left(\frac{n}{\log n}\right) = O(n)$. En "arrêtant" la parallélisation après la création de $\log n$ processus de tri (donc à une profondeur de $\log \log n$), on obtient donc un algorithme de complexité :

$$T_{//}(n) = T_{seq}\left(\frac{n}{\log n}\right) + O\left(\sum_{k=1}^{\log \log n} \frac{n}{2^k}\right) = O(n)$$

$$H(n) = \text{Max}\left(2H\left(\frac{n}{2}\right), 4H\left(\frac{n}{4}\right), \ldots, 2^{\log \log n}\right) = O(\log n)$$

D'où un algorithme de travail optimal de complexité $O_{//}(n, \log n)$ sur une EREW-PRAM.

La parallélisation de l'algorithme séquentiel optimal de tri par partition-fusion est donc ramenée à la parallélisation de la fusion de deux séquences triées. Or, l'algorithme de fusion classique utilisé en séquentiel est très difficile à paralléliser : le parcours des deux séquences est à chaque étape déterminé par la comparaison qui vient d'être effectuée. Les deux paragraphes suivants proposent deux algorithmes de fusion parallèle : la fusion bitonique et la fusion pair-impair.

5.3.3 Fusion Bitonique

Cette fusion est construite à partir d'une propriété vérifiée par certaines séquences, appelées bitoniques. La séquence $(a_0, \ldots, a_{m-1})$ est dite *bitonique* si et seulement si :

(1) soit il existe k $(0 \leq k \leq m-1)$ tel que la sous-séquence $(a_0, \ldots, a_k)$ est croissante, et la sous-séquence $(a_k, \ldots, a_{m-1})$ est décroissante.

(2) soit il existe un décalage d'indice tel que la condition (1) est réalisée.

L'algorithme qui suit est fondé sur la remarque suivante : soient S_1 et S_2 deux séquences triées que l'on désire fusionner. Si S_1 est triée par ordre croissant et S_2 par ordre décroissant, alors la séquence S obtenue directement par concaténation de S_1 et S_2 est bitonique. Le tri de cette séquence bitonique est exactement la fusion des deux listes. Cette fusion est appelée fusion bitonique [Akl 85].

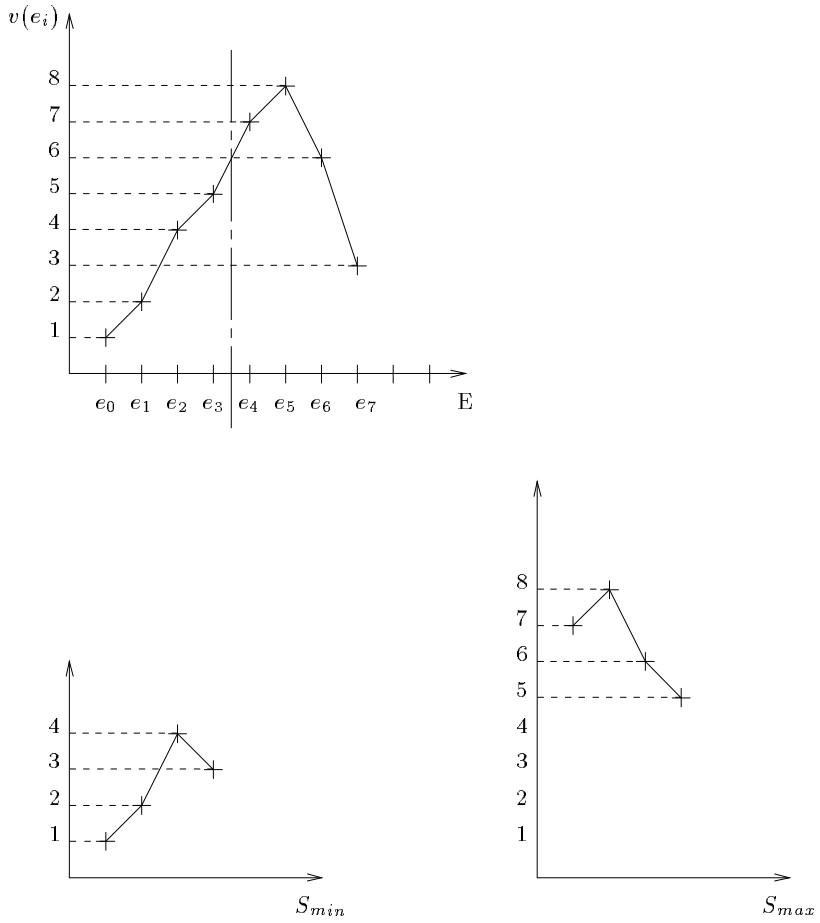


FIG. 5.2 – *Partition d'une séquence bitonique.*

Soit $S = (a_0, \ldots, a_{m-1})$ (on suppose pour simplifier $m = 2^p$) une séquence bitonique que nous supposerons vérifier (1) pour simplifier[1]. Autrement dit, S est telle que :

$$\begin{cases} a_0 \leq a_1 \leq \cdots \leq a_k & \text{i.e. } (a_0, \ldots, a_k) \text{ est croissante)} \\ a_k \geq a_{k+1} \geq \cdots \geq a_{m-1} & \text{i.e. } (a_k, \ldots, a_{m-1}) \text{ est décroissante)} \end{cases}$$

Posons $q = \frac{m}{2}$, et considérons alors les deux sous-séquences suivantes de S :

$$\begin{cases} S_{min} = (\ \text{Min}(a_0, a_q), & \text{Min}(a_1, a_{q+1}), & \ldots, & \text{Min}(a_{q-1}, a_{m-1}) \\ S_{max} = (\ \text{Max}(a_0, a_q), & \text{Max}(a_1, a_{q+1}), & \ldots, & \text{Max}(a_{q-1}, a_{m-1})) \end{cases}$$

Par définition, tout élément de S_{min} est inférieur à tout élément de S_{max}. Par ailleurs, on peut vérifier que les deux sous-séquences S_{min} et S_{max} sont bitoniques (comme le montre la figure 5.2).

Passer de S à (S_{min}, S_{max}) coûte $O_{//}(1, m)$. Les deux séquences bitoniques obtenues, S_{min} et S_{max}, ont chacune $\frac{m}{2}$ éléments. On applique donc le même principe à S_{min} et S_{max}, et on obtient ainsi, au bout de $p = \log m$ étapes, une séquence triée, avec une complexité $O_{//}(\log m, m)$. Cette étape récursive est schématisée par le réseau de tri de la figure 5.3.

1. Mais la proriété que nous allons énoncer reste valide lorsque S vérifie (2)

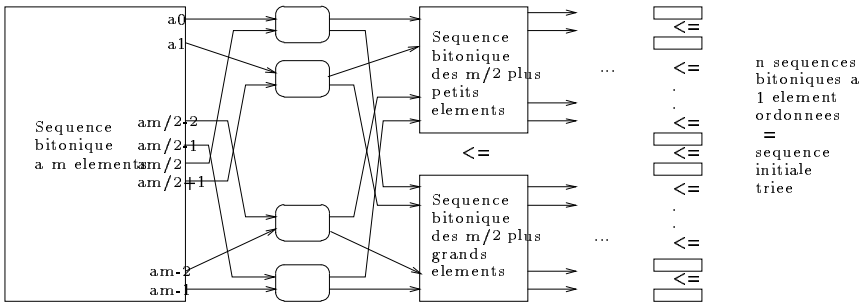


FIG. 5.3 – *Tri d'une séquence bitonique [Qui 88].*

Remarque. Considérant une séquence quelconque ayant n éléments, nous pouvons la considérer au départ comme $\frac{n}{2}$ séquences bitoniques de 2 éléments chacune. On applique en parallèle avec $m = 2$ la phase décrite ci-dessus, ce qui a pour complexité $O_{//}(1, n)$. On obtient $\frac{n}{4}$ séquences bitoniques de longueur 4, que l'on trie en appliquant de nouveau cette phase en parallèle, etc.

Le temps parallèle de l'algorithme de fusion bitonique est alors :

$$T_{\text{Fusion}}(n) = T_{\text{Fusion}}\left(\frac{n}{2}\right) + O(1) = O(\log n)$$

et le nombre de processeurs :

$$H_{\text{Fusion}}(n) = \text{Max}\left(2H_{\text{Fusion}}\left(\frac{n}{2}\right), n\right) = O(n)$$

On obtient donc un algorithme de fusion parallèle de complexité sur une EREW-PRAM :

$$O_{//}(\log n, n)$$

L'algorithme de tri bitonique est alors basé principalement sur la procédure de fusion bitonique de deux séquences S_1 et S_2 triées en sens contraires pour former une séquence T triée. Sa complexité en temps est :

$$T_{\text{Tri}}(n) = T_{\text{Tri}}\left(\frac{n}{2}\right) + T_{\text{Fusion}}(n) + O(1) = O(\log^2 n)$$

et le nombre de processeurs nécessaires :

$$H_{\text{Tri}} = \text{Max}\left(2H_{\text{Tri}}\left(\frac{n}{2}\right), H_{\text{Fusion}}(n)\right) = O(n)$$

L'algorithme de tri bitonique est donc efficace et de complexité sur une EREW-PRAM :

$$O_{//}\left(\log^2 n, n\right)$$

La figure 5.4 montre un réseau de tri implantant la fusion bitonique d'une séquence de 8 éléments.

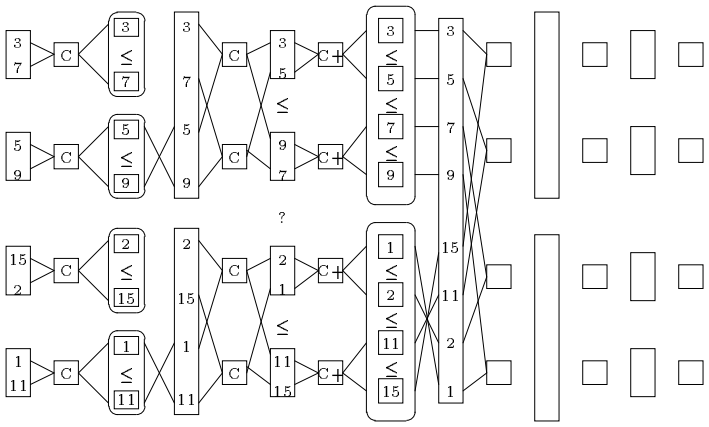


FIG. 5.4 – *Tri bitonique d'une séquence à 8 éléments.*

Remarque. La structure du graphe de précédence, qui correspond à ce tri s'appelle un perfect-shuffle (ou mélange parfait : tout le monde rencontre tout le monde). Il est très important, et apparaît dans de nombreux algorithmes, dont la transformée de Fourier rapide.

5.3.4 Fusion pair-impair

On propose ici un autre type de fusion. Soient

$$X = (x_1, x_2, \ldots, x_{2m}) \quad \text{et} \quad Y = (y_1, y_2, \ldots, y_{2m})$$

deux séquences triées par ordre croissant comportant chacune $2m$ éléments d'un ensemble totalement ordonné.

Soient alors X_{impair} et Y_{impair} les deux sous-séquences de rang impair de X et Y définies par :

$$X_{\text{impair}} = (x_1, x_3, \ldots, x_{2m-1}) \quad \text{et} \quad Y_{\text{impair}} = (y_1, y_3, \ldots, y_{2m-1})$$

et X_{pair} et Y_{pair} les deux sous-séquences de rang pair de X et Y :

$$X_{\text{pair}} = (x_2, x_4, \ldots, x_{2m}) \quad \text{et} \quad Y_{\text{pair}} = (y_2, y_4, \ldots, y_{2m})$$

On définit récursivement la fusion pair-impair des deux séquences X et Y triées et ayant le même nombre d'éléments qui est une puissance de 2, par l'algorithme pseudo-ADA suivant.

La procédure *PartitionPairImpair* permet de séparer une séquence X en entrée en deux sous-séquences X_{pair} et X_{impair} comme décrit ci-dessus.

```
procedure FusionPairImpair ( X, Y : in Sequence ; T : out Sequence ) is
    Z, W : Sequence ;
begin
    if ( Cardinal( X ) = 1 ) then
```

```
        T := Sequence( Min( x(1), y(1) ), Max( x(1), y(1) ) );
    else
        PartitionPairImpair( X, X_impair, X_pair );
        PartitionPairImpair( Y, Y_impair, Y_pair );
        FusionPairImpair( X_impair, Y_impair, Z );
        FusionPairImpair( X_pair, Y_pair, W );
        -- Soient (Z(1), ..., Z(2m)) et (W(1), ..., W(2m)) les éléments des séquences Z et W.
        -- On calcule alors la séquence T ainsi definie :
        T := Sequence( Z(1), Min(Z(2), W(1)), Max(Z(2), W(1)), Min(Z(3), W(2)), ...,
              Min(Z(k), W(k-1)), Max(Z(k), W(k-1)), ..., Max(Z(2m), W(2m-1)), W(2m) );
    end if;
end FusionPairImpair;
```

La séquence T ainsi obtenue est triée [Akl 85] : cette propriété peut être démontrée par récurrence, et provient du calcul de Z et W pour lequel on a séparé les éléments pairs et impairs de deux listes triées X et Y. On peut vérifier que chaque élément de la suite est comparé directement ou indirectement à tout autre élément.

Le calcul final de T peut être effectué en temps 1 en utilisant $4m$ processeurs. On en déduit donc que la complexité temporelle de l'algorithme de fusion pair-impair est :

$$T_{\text{Fusion}}(n) = T_{\text{Fusion}}\left(\frac{n}{2}\right) + O(1) = O(\log n)$$

et le nombre de processeurs :

$$H_{\text{Fusion}}(n) = \text{Max}\left(2H_{\text{Fusion}}\left(\frac{n}{2}\right), n\right) = O(n)$$

donc une complexité finale :

$$O_{//}(\log n, n)$$

A partir de cette fusion pair-impair, il est facile de construire un algorithme parallèle efficace pour le tri, comme expliqué précédemment à partir de la fusion bitonique. Finalement, la complexité du tri pair-impair ainsi obtenu est sur une EREW-PRAM :

$$O_{//}\left(\log^2 n, n\right)$$

Remarque. Les deux algorithmes de tri présentés (bitonique et pair-impair) peuvent être rendus de travail optimal (exercices 7 et 9), mais sont de complexité temporelle $O(\log^2 n)$, alors que l'algorithme de tri par sélection montre qu'il est possible d'atteindre un temps parallèle $O(\log n)$. Des réseaux optimaux de tris, de complexité $O_{//}(\log n, n)$ ont été trouvés par Ajtai, Komlòs et Szemerédi en 1983 [Pippenger, HTCS]. Il existe par ailleurs, sur le modèle PRAM, un autre algorithme optimal pour le tri de complexité $O_{//}(\log n, n)$) [Col 88] : cette algorithme est basé sur un mécanisme de fusion de temps constant.

Néanmoins, le facteur devant le $\log n$ est important, et en pratique, le tri bitonique -après équilibre des travaux- reste le plus pertinent.

5.4 Conclusion : de l'algorithmique PRAM à la programmation sur une machine réelle

Les algorithmes que nous avons décrits dans ce chapitre et le précédent peuvent être exécutés sur la PRAM théorique définie au chapitre 4 et avec la complexité qui leur a été associée.

Cependant, lorsque l'on désire les programmer sur une architecture parallèle possédant un nombre fini de proccesseurs (que ce soit un réseau de stations de travail ou une machine parallèle), les hypothèses sur lesquelles repose le modèle PRAM posent différents problèmes :

- le modèle théorique PRAM est basé sur des hypothèses synchrones qui ne peuvent être vérifiées en pratique.
- le modèle PRAM introduit un opérateur *fork*, de temps constant, qui permet l'affectation des processus aux processeurs, et suppose l'existence d'une mémoire partagée dont l'accès est supposé effectué en temps constant lui-aussi.

Après avoir explicité ces différents problèmes, nous expliquons pourquoi les techniques introduites pour l'algorithmique PRAM restent valides pour la conception d'algorithmes parallèles, à condition de disposer d'autres outils pour l'implantation d'une PRAM sur une architecture parallèle donnée. D'ailleurs, la tendance aujourd'hui est de munir les architectures parallèles de systèmes d'exploitation qui permettent de cacher les caractéristiques internes de l'architecture. Cette tendance justifie de travailler sur des modèles généraux, intégrant les opérateurs qui seront fournis par les futurs systèmes d'exploitation des architectures parallèles et qui rendent -entre autre- les services d'une PRAM. Les chapitres suivants sont consacrés à l'introduction de ces différents outils.

5.4.1 Algorithme synchrone sur architecture asynchrone

Ce synchronisme ne peut être assuré que par la présence d'une horloge globale (logicielle ou matérielle) et n'existe pratiquement que dans les architectures dites SIMD synchrones où tous les processeurs effectuent la *même* opération à chaque top de l'horloge globale. Ce mode opératoire est très restrictif au niveau algorithmique mais peut être utilisé efficacement pour des algorithmes de traitement d'image par exemple (les mêmes opérations sur tous les pixels). La majorité des architectures de machines parallèles disponibles sur le marché n'offre pas d'horloge globale ni de séquenceur global d'instructions, les processeurs fonctionnant intrinsèquement de manière asynchrone sur des instructions différentes à un instant donné.

En général, un processeur fournit un ensemble d'opérations de base qui sont toutes de durée différentes (lecture ou écriture dans la mémoire locale, multiplication de flottants ou de booléen par exemple, opérations d'accès à la mémoire partagée ou *fork*) : l'alignement de toutes les opérations sur celle de durée la plus grande est possible si l'on suppose que les opérations sont de durée bornée[2], mais est *pratiquement* très pénalisant pour les performances globales de la machine. Néanmoins, pour le calcul de la complexité, qui est toujours connue à une constante multiplicative près, l'évaluation d'un algorithme sur une PRAM synchrone (dans le sens où les durées de toutes

2. Ce qui est raisonnable si l'on prend des précautions -voir chapitre 6 et 7

les opérations sont alignées sur la plus longue) est raisonnable. En effet, cette évaluation permet de se dégager des spécificités architecturales et d'obtenir *l'ordre de grandeur* voulu, invariant quelles que soient les caractéristiques en terme de temps des opérations de base. Il est donc important de définir des modèles théoriques généraux, abstraction de grandes classes d'architectures, comme le modèle PRAM présenté ici, qui permettent la construction d'algorithmes "portables" [3].

5.4.2 Modèle de mémoire

Concernant la mémoire, locale ou globale, le modèle PRAM repose sur l'hypothèse d'un accès en temps constant unitaire. Ceci n'est pas verifié par les architectures pour lesquelles l'accès à la mémoire physique globale est en général plus long. De plus, les mémoires (locales et globales) sont en général paginées, ce qui rend fausse l'hypothèse du temps d'accès constant.

Cependant, les systèmes d'exploitation de machines parallèles fournissent des opérateurs d'accès en mémoire globale et locale qui optimisent le temps d'accès en mémoire, en respectant les propriétés de mémoire locale (accessible par un seul processeur) et globale (accessible par tous les processeurs [4].

Le premier opérateur consiste à mettre dans la mémoire locale du processeur les variables de la mémoire partagée qui sont accédées par les tâches (ou processus) du processeur : on parle de *placement de données*. Lorsqu'une donnée est manipulée par plusieurs tâches, il est parfois possible de placer tous les processus concernés sur le même processeur (on parle de placement de processus dirigé par les données). Mais souvent, cette stratégie entraîne une perte de parallélisme effectif. Dans le cas général, les processus concernés sont donc sur des processeurs distincts (placement dirigé par le parallélisme de contrôle). Si ces processus utilisent les mêmes données, ils peuvent

- soit tous posséder un exemplaire propre de la même donnée en mémoire locale. Il est donc nécessaire d'assurer une cohérence entre tous les exemplaires de cette donnée, lorsque celle-ci est concuremment lue ou écrite par les processus. Le coût effectif de cet algorithme de maintien de la cohérence est alors crucial pour les performances effectives du programme,
- soit partager un exemplaire en mémoire globale. Il est alors nécessaire de mettre en place un mécanisme de contrôle d'accès de façon à rendre les opérations de lecture et d'écriture atomiques. Le coût de l'accès à cette mémoire globale doit être contrôlé de façon à éviter qu'il devienne un goulot d'étranglement,
- soit de mettre en place un mécanisme de circulation des données entre les mémoires locales des tâches concernées. Là aussi un coût additionnel est à prendre en compte.

Pour un algorithme PRAM pour lequel il y a partage de données, les trois mécanismes sont envisageables. Le meilleur choix est dépendant de l'architecture cible. Poussant le raisonnement plus loin, il est possible d'avoir deux algorithmes A1 et A2 résolvant le même problème et A1

3. Il existe de nombreux autres modèles parallèles que le modèle PRAM, qui permettent de mieux prendre en compte certaines classes d'architectures : par exemple, les HRAM [CT 93] modélisent une machine parallèle synchrone dont les processeurs sont connectés selon une topologie d'hypercube. Les études algorithmiques sur l'un des ces modèles permettent de construire des algorithmes plus fins et mieux adaptés à des architectures de la même classe -même si l'hypothèse de fonctionnement sysnchrone est toujours considérée-. Par là-même, ils perdent toutefois en généralité.

4. On dit aussi mémoire partagée.

meilleur que A2 au sens du travail PRAM, mais que l'implantation de A1 sur une architecture cible soit toujours plus coûteuse que l'implantation de A2 à cause du coût de la primitive fork ou du coût de la gestion mémoire.

De façon analogue en séquentiel, une bonne bibliothèque de calcul sur les entiers – comme par exemple Gnu-MP, ou Pari – doit proposer trois procédures de multiplication, implémentations respectives des algorithmes : standard – $O(n^2)$ –, Karatsuba – $O(n^{1,58})$ – et Schonhagge-Strassen – $O(n \log n \log\log n)$ –. Dès lors, le choix de la meilleure procédure pour une taille d'entiers donnée est non portable et dépend réellement de l'architecture cible (machine, compilateur, système d'exploitation).

5.4.3 L'opérateur *fork*

L'opérateur *fork* de la PRAM permet implicitement d'affecter un nouveau processeur à tout nouveau processus et ceci en temps constant. Il est donc essentiel de disposer d'un tel opérateur sur toute architecture cible pour garantir la portabilité des algorithmes définis sur une PRAM. De plus, l'exécution sur le nombre de processeurs donné de la machine cible d'un algorithme construit pour fonctionner sur plus de processeurs est rendue possible par la multiprogrammation, c'est à dire l'exécution concurrente de plusieurs processus placés sur (ou affectés à) un même processeur. L'implantation de l'opérateur *fork* doit prendre en compte le fonctionnement en mode multiprogrammé des processeurs. L'implantation de *fork* peut être vue comme l'opérateur de base associé à l'architecture cible qui permet la construction du principe de Brent. Il consiste en effet à allouer à tout processus de l'algorithme parallèle un processeur (vu comme une ressource) de l'architecture sur lequel le processus pourra s'exécuter, de façon à ce que, au moins approximativement, le principe de Brent soit vérifié pour garantir l'efficacité de l'algorithme. Si un algorithme peut s'exécuter en temps $T(n)$ avec $H(n)$ processeurs d'une PRAM, alors l'opérateur *fork* d'une architecture à p processeurs doit donc être tel que si $p \leq H(n)$ proccesseurs sont utilisés sur l'architecture, alors le temps de résolution effective est proche de $\left(\frac{T(n)H(n)}{p}\right)$. Il doit de plus prendre un temps approximativement constant pour que le calcul de complexité effectué sur la PRAM reste valide. Nous aborderons ce problème dans le chapitre **??**.

L'affectation d'un processus à un processeur peut se faire de deux manières : statique et dynamique. Nous commentons maintenant ces deux techniques.

Affectation statique

Il est parfois possible, par une analyse du programme source, de connaître le graphe de précédence des processus, ainsi que leur durée d'exécution sur l'architecture cible. Pour reprendre l'exemple précédent, une telle analyse est possible si l'on désire construire un programme qui résolve des systèmes triangulaires de taille n_0 constante. Dans ce cas, des techniques d'ordonnancement [CT 93] permettent de définir une affectation de tout processus à un processeur de l'architecture de façon à optimiser la durée effective du programme sur cette architecture. Les placements associés aux opérateurs *fork* sont ainsi pré-calculés, le pré-calcul étant valide pour toutes les exécutions sur l'architecture. On parle de *placement statique* des processus.

Affectation dynamique

Dans la plupart des cas, il n'est pas possible de connaître ce graphe des processus, parce que ce graphe a une structure qui dépend des données (taille, valeur). Dans l'exemple du système triangulaire à résoudre, l'algorithme peut être utilisé dans une bibliothèque numérique pour la résolution d'un système de taille quelconque, la taille du système n'étant connue que lors de l'appel de la fonction correspondante. D'autre part, le programme peut contenir des branchements conditionnels entraînant ou non la création de processus, la valeur du test ne pouvant être connue que lors de l'exécution. L'affectation d'un processus à un processeur ne peut alors être calculée par l'opérateur *fork* que lors de la création du processus. On parle alors de *placement dynamique*.

Le problème de l'affectation d'un processus à un processeur s'apparente alors de beaucoup à celui de l'allocation répartie de ressources. Cependant, ici le problème introduit des contraintes supplémentaires, l'opérateur *fork* devant être de temps constant, et devant prendre en compte pour sa décision d'affectation le coût de l'exécution d'un processus sur un processeur (notamment lorsque les processeurs sont hétérogènes, et lorsque le placement implique des communications de données nécessaires d'un processus à un autre, par la mémoire partagée par exemple). Il doit donc être capable d'adapter la granularité de l'algorithme (qui souvent introduit un parallélisme très fin, comme dans le cas des algorithmes présentés pour les préfixes ou le tri), à celle de la machine : il est par exemple inutile de vouloir paralléliser un calcul si le coût de la mise en place des processus nécessaires à sa résolution, y compris le coût des accès en mémoire partagée, est supérieur à celui d'un calcul séquentiel.

Les techniques introduites (analyse des précédences, découpe récursive, introduction de redondance, élimination de la redondance) sont générales, et servent de cadre à la construction d'un programme parallèle pour une architecture parallèle quelconque; l'adaptation à l'architecture se situe à un niveau d'optimisation plus fin. On ne compare plus alors les algorithmes par leur complexité asymptotique (notation $O_{//}$), mais par leur *coût effectif*, paramétré par le temps des opérations de base. L'adéquation d'un bon algorithme parallèle à une architecture parallèle se situe alors essentiellement à un niveau d'équilibrage des travaux dans le sens où il s'agit de comparer d'une part le coût effectif pour créer une nouvelle tâche en parallèle (coût de l'opération fork, coût de passage des paramètres et de partage des données, etc) et d'autre part le coût d'une exécution en séquence de cette tâche.

Les chapitres suivants (6 et 7) sont consacrés à l'introduction des outils qui permettent la résolution de problèmes typiques d'implantation d'une PRAM sur une machine réelle. Dans tous les cas, il reste essentiel de pouvoir être assuré que la complexité théorique évaluée sur un modèle théorique -tel la PRAM- reste valide sur l'architecture cible sur laquelle est implantée l'algorithme. Pour cela, il est nécessaire d'être capable d'évaluer le coût de l'algorithme sur une architecture asynchrone avec ses propres caractéristiques : quel type de mémoire est disponible et avec quels outils de gestion, quels sont les outils de placement et leur coût, etc. Le chapitre 9 introduit les techniques de base permettant cette évaluation.

5.5 Programmes ADA

Nous donnons ici la programmation en ADA de l'algorithme de tri bitonique.

```
generic
     type Element is private;                                          -- le type des éléments
     with function "<"( a, b : in Element) return boolean; -- la fonction comparaison de 2 éléments
package TRI_BITONIQUE is
     type indice is new integer;                          -- le type indice pour accéder à une séquence
     type Sequence is array (indice range <>) of Element;                -- le type séquence
     type OrdreTri is (Croissant, Decroissant);
     procedure Trier(sens: in OrdreTri; s: in out Sequence);
end TRI_BITONIQUE;

     with TEXT_IO; use TEXT_IO;

package body TRI_BITONIQUE is

     package ENTIER_IO is new INTEGER_IO(integer ); use ENTIER_IO;

     procedure Trier(sens: in OrdreTri; s: in out Sequence) is

        temps, nombre_de_processeurs : integer:= 0;
        procedure TriParallele (sens : in OrdreTri; indice_debut, indice_fin : in indice) is


           task type TacheTri is
              entry envoyer_donnees (sens : in OrdreTri; indice_debut, indice_fin : in indice);
              entry attendre_resultat;
           end TacheTri;


           task body TacheTri is
              sens_tri : OrdreTri;
              i, j : indice;
              begin
                 accept envoyer_donnees (sens : in OrdreTri;
                    indice_debut, indice_fin : in indice) do
                    i := indice_debut; j:= indice_fin; sens_tri:= sens;
                 end envoyer_donnees;
                 TriParallele (sens_tri,i,j);
                 accept attendre_resultat;
              end TacheTri;

           function Inverse(s:OrdreTri) return OrdreTri is
              begin
                 case s is
                    when Croissant => return Decroissant;
                    when Decroissant => return Croissant;
                 end case;
              end Inverse;

           procedure Fusion(sens : in OrdreTri; i,j,k : in indice) is
```

```
-- fusion des deux sous-séquences triées (i..j) et (j+1..k)
-- on suppose ici que :
-- * s(i..j) est triée dans l'ordre sens
-- * s(j+1..k) est triée dans l'ordre inverse de sens
-- * s(i..k) sera triée en sortie dans l'ordre sens
-- * les deux sous-séquences en entrée ont le même nombre d'éléments


task type FusionBitonique is
   entry envoyer_donnees (sens : in OrdreTri; indice_debut, indice_fin : in indice) ;
   entry attendre_resultat ;
end FusionBitonique;

task type PorteMinMax is
   entry input (s: in OrdreTri; a,b : in Element);
   entry output ( min, max: OUT Element);
end PorteMinMax;

fusion1, fusion2 : FusionBitonique;                    -- lancement de deux nouvelles tâches

task body FusionBitonique is
   sens_tri : OrdreTri;
   i, j : indice;
   begin
      accept envoyer_donnees (sens : in OrdreTri;
         indice_debut, indice_fin : in indice) do
         i := indice_debut; j:= indice_fin; sens_tri:= sens;
      end envoyer_donnees;
      if (NOT(i=j)) then Fusion(sens_tri,i,(i+j) / 2, j); end if;
      accept attendre_resultat ;
   end FusionBitonique;


procedure MinMax(sens: in OrdreTri; a,b : in out Element) is
   minimum, maximum: Element;
   begin
      if (a<b) then
         case sens is
            when Croissant => minimum:= a; maximum:= b;
            when Decroissant => minimum:= b; maximum:= a;
         end case;
      else
         case sens is
            when Croissant => minimum:= b; maximum:= a;
            when Decroissant => minimum:= a; maximum:= b;
         end case;
      end if;
      a := minimum; b := maximum;
   end MinMax;

task body PorteMinMax is
   e1, e2 : Element;
   sens : OrdreTri;
```

```
   begin
      accept input (s: in OrdreTri; a,b : in Element) do
         e1 := a; e2 := b; sens := s;
      end input ;
      MinMax(sens, e1, e2);
      accept output ( min, max : OUT Element) do
         min := e1; max := e2;
      end output ;
   end PorteMinMax;


begin
   CalculBitonique :
      -- on sépare les min et les max en temps 1 en le faisant en parallèle.
      -- Ici les boucles loop ada sont insuffisantes : elles permettent
      -- cependant de confirmer la theorie.
      --
      -- Voici ce qu'on ecrirait en séquentiel
      -- begin
      -- for ind in 0..(j-i) loop
      -- MinMax(sens, s(i+ind), s(j+1+ind) );
      -- end loop;
      -- end CalculBitonique;

      -- Ici : Min Max en parallèle
      declare
         Bitoniser : array (0..j-i) of PorteMinMax;        -- les portes MinMax nécessaires
      begin
         for ind in 0..(j-i) loop
            Bitoniser(ind).input(sens, s(i+ind), s(j+1+ind));
         end loop;
         for ind in 0..(j-i) loop
            Bitoniser(ind).output(s(i+ind), s(j+1+ind));
         end loop;
      end CalculBitonique;


   if (i = s'first) then
      temps := temps + 1;
      nombre_de_processeurs := nombre_de_processeurs +
         INTEGER((s'last - s'first + 1) / 2);
      put ("- temps : ");put (temps);
      put (" nombre de processeurs : ");put (nombre_de_processeurs);
      new _line;
   end if;

   -- maintenant, tous les éléments de s(i..j) sont plus petits que
   -- ceux de s(j+1..k). De plus les deux sous-séquences sont bitoniques :
   -- il suffit donc de les fusionner.
   fusion1.envoyer_donnees (sens, i, j);
   fusion2.envoyer_donnees (sens, j+1, k);
   fusion2.attendre_resultat;
   fusion1.attendre_resultat;
```

```
            end Fusion;


         begin
            if (indice_debut<indice_fin ) then
               TriEnParallele :                                          -- bloc de tri parallèle
               declare
                  tri1, tri2 : TacheTri;                          -- lancement de deux nouvelles tâches
                  indice_milieu : indice := (indice_debut+ indice_fin ) / 2;     -- 1. Partition des entrées
               begin
                  tri1.envoyer_donnees (sens, indice_debut, indice_milieu);              -- 2. calcul en
                  tri2.envoyer_donnees (Inverse(sens), indice_milieu +1, indice_fin);      -- parallèle.
                  tri2.attendre_resultat;                                              -- 3. attente des
                  tri1.attendre_resultat;                                                -- résultats
                  Fusion(sens, indice_debut, indice_milieu, indice_fin);
               end TriEnParallele;
            end if;
         end TriParallele;

      begin
         TriParallele (sens, s'first, s'last);
         new _line; put ("complexite : (comparez avec la theorie...)"); new _line;
         put (" nombre de portes MinMax (en les reutilisant)");
         put (nombre_de_processeurs/temps);new _line;
         put (" temps ");put (temps);new _line;
         new _line;
      end Trier;

begin
      null;
end TRI_BITONIQUE;
```

Exercices

1. En appliquant la technique d'équilibre des travaux à l'algorithme proposé pour l'inversion de matrices triangulaires, construire un algorithme optimal pour ce problème.

2. En base $\beta = 10$, calculer 782524+653493 par l'algorithme de Brent et Kung. On explicitera les valeurs g_i, p_i, u_i, v_i et c_i.

3. Ecrire un programme ADA implémentant l'algorithme de tri par sélection.

4. Montrer que la technique d'équilibre des travaux permet la construction d'un algorithme de tri par sélection de complexité :

$$O_{//}\left(\log n, \frac{n^2}{\log n}\right)$$

sur une EREW-PRAM. Modifier le programme ADA de l'exercice précédent pour qu'il implémente cet algorithme.

5. Ecrire des programmes parallèles ADA pour réaliser les primitives *Concat*, *Reverse*, et *Separer_Min_Max* nécessaires pour la fusion bitonique.

6. Ecrire un programme ADA implémentant l'algorithme de tri bitonique en utilisant pour la communication entre tâches non plus l'envoi ou la réception de séquence mais la communication d'indices définissant un intervalle dans un tableau figurant en mémoire partagée. Ce tableau contient en entrée la séquence à trier et en sortie la séquence triée. Montrer que l'algorithme obtenu respecte bien la contrainte EREW pour l'accès en lecture ou écriture sur une case quelconque du tableau.

7. Construire un algorithme optimal de complexité $O_{//}\left(\log n \frac{n}{\log n}\right)$ pour la fusion bitonique de deux listes triées. En déduire un algorithme de tri de complexité $O_{//}\left(\log^2 n, \frac{n}{\log n}\right)$ et donc de travail optimal. Modifier le programme ADA de l'exercice précédent pour qu'il implémente cet algorithme.

8. Ecrire un programme ADA implémentant l'algorithme de tri basé sur la fusion pair-impair :

1. en s'inspirant de celui donné précédemment pour le tri bitonique.
2. en utilisant pour la communication entre tâches non plus l'envoi ou la réception de séquence mais la communication d'indices définissant un intervalle dans un tableau figurant en mémoire partagée. Ce tableau contient en entrée la séquence à trier et en sortie la séquence triée. Montrer que l'algorithme obtenu respecte bien la contrainte EREW pour l'accès en lecture ou écriture sur une case quelconque du tableau.

9. Construire un algorithme optimal de complexité $O_{//}\left(\log n \frac{n}{\log n}\right)$ pour la fusion pair-impair de deux listes triées. En déduire un algorithme de tri de complexité $O_{//}\left(\log^2 n, n\right)$ – et donc de travail optimal –. Modifier le programme ADA de l'exercice précédent pour qu'il implémente cet algorithme.

Bibliographie

[**Akl 85**] S. G. Akl, "Parallel Sorting Algorithms", Academic Press, 1985.

[**BK 83**] R. P. Brent, H. T. Kung, "Systolic VLSI Arrays for Linear Time Gcd Computation", IFIP-VLSI 83, Anceau ed. North Holland, 1983.

[**CT 93**] M. Cosnard, D. Trystram, "Algorithmes et Architectures Parallèles", Interéditions 1993

[**Col 88**] R. Cole, "Parallel merge Sort", SIAM J. Computing, vol.6, p. 733-744, 1977

[**Coo 85**] S.A. Cook, "A Taxonomy of Problems that have Fast Parallel Algorithms", Information and Control, vol. 64, pp2-22, 1985

[**Csa 76**] L. Csanky, "Fast Parallel Matrix Inversion Algorithms", SIAM J. of Computing, 5/4, 1976.

[**CW 87**] D. Coppersmith, S. Winograd, "Matrix Multiplication via Arithmetic Progressions", Proc. 19th ACM Symp. Theory Comp. , p.1-6, 1987

[**GBH 82**] J. Von Zur Gathen, A. Borodin, J.E. Hopcroft, "Fast Parallel Matrix and Gcd Computations", Inf. Control 52, p. 241-256 , 1982

[**Kal 89**] E. Kaltofen, "Parallel Algebraic Computing Design", Proc. Issac 89, Portland 1989

[**KMR 87**] R. Kannan, G. Miller, L. Rudolph, "Sublinear Parallel Algorithm for Computing the Greatest Common Divisor of Two Integers", SIAM J. Comput. vol 16, 1, Fév. 1987

[**KP 91**] E. Kaltofen, V. Pan, "", à paraître.

[**KR 90**] R.M. Karp, V. Ramachandran, *"A survey of parallel algorithms for shared memory machines"*, Handbook of Theoretical Comp. Science, p. 869-941, North-Holland 1990

[**MKR 88**] G.L. Miller, E. Kaltofen, V. Ramachandran, "Efficient Parallel Evaluation of Straight-Line Code and Arithmetic Circuits", SIAM J. of Computing, 17 / 4 pp. 687-695, 1988

[**Qui 88**] M. J. Quinn, "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill, 1988.

[**Sch 71**] A. Schönhage, "Schnelle Berechnung von Kettenbruchentwicklungen", Acta Informatica vol 1, pp139-144, 1971.

[**Sie 90**] F. Siebert-Roch, "Forme Normale d'Hermite - Méthodes de Calcul et Parallélisation", Thèse de doctorat, INPG 1989.

[**SS 71**] A. Schönhage, V. Strassen, "Schnelle Multiplikation Grosser Zahlen", Computing vol. 7 p.281-292

[**Str 69**] V. Strassen, "Gaussain elimination is not optimal", Numerische Math, pp 54-56, 1969.

Chapitre 6

Mise en oeuvre d'une mémoire partagée

Nous nous intéressons maintenant à l'implantation d'une PRAM (ou d'un programme ADA qui implante un algorithme PRAM) sur une machine réelle et plus spécifiquement au problème de l'implantation du partage de donnée. Le niveau d'observation du calcul parallèle va changer par rapport à ce qu'on a vu jusqu'à présent. Désormais, on considère des processus qui calculent (ces processus sont anonymes et on ignore ce qu'ils calculent) et partagent des données. Ce partage se fait soit par variables globales soit par passage de paramètres dans les appels d'entrées de tâches. Bien évidement, ce partage de données entre processus parallèles ne peut se faire sans contrôle :

- pour une variable globale partagée , qui est lue et modifiée par plusieurs processus, on peut vouloir imposer plusieurs disciplines d'accès : tous les accès sont strictement séquentiels (la variable est alors une ressource critique), ou bien les accès sont gérés par une discipline de lecteur-rédacteur (voir chapitre 2), ou bien encore chaque processus a une copie de la variable globale dans la mémoire de son processeur et un algorithme de maintien de la cohérence des copies est mis en place, etc.
- pour les appels d'entrées, ces appels sont stockés dans une file d'attente gérée avec une discipline FIFO par la tâche réceptrice. La gestion de cette file s'apparente au problème des producteurs-consomateurs avec plusieurs producteurs et un seul consommateur, et la contrainte qu'elle est de taille bornée. C'est avec l'exemple des producteurs consommateurs que nous allons illustrer les concepts de ce chapitre.

Définir une discipline d'accès à une entité (variable globale ou entrée), c'est imposer aux processus parallèles des contraintes de synchronisation. Dans ce chapitre, nous allons montrer qu'il existe plusieurs façons de concevoir cette synchronisation. A chaque façon de voir correspond un type d'outil approprié. En théorie, on peut mettre en œuvre toute architecture logicielle de synchronisation sur n'importe quelle architecture matérielle, mais cette mise en oeuvre n'est pas forcément efficace.

Dans le paragraphe 1, nous présentons l'exemple qui sert de fil conducteur tout au long du chapitre : le problème du "producteur-consommateur", avec un seul producteur et un seul consom-

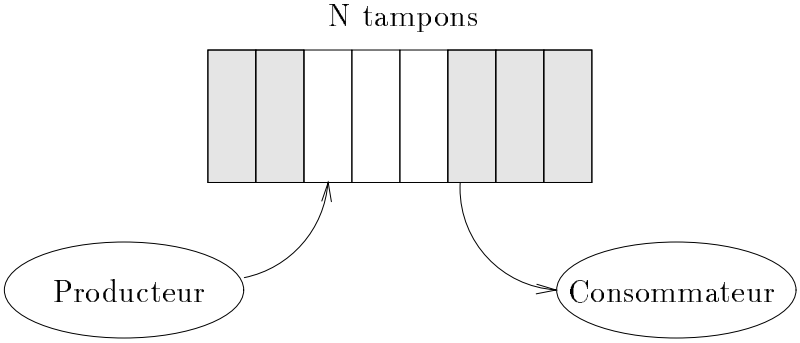


FIG. 6.1 – *Le producteur consommateur*

mateur pour simplifier. Nous identifions les principaux points de synchronisation entre les processus coopérants et différentes manières de représenter le système formé par ces deux processus. Les trois paragraphes qui suivent proposent trois architectures de synchronisation : centralisé, par variable partagée et distribuée. Dans le paragraphe 5, nous évoquons rapidement comment ces architectures "abstraites" de synchronisation peuvent s'adapter aux architectures matérielles sur lesquelles les systèmes de processus concurrents ou parallèles sont mises en œuvre.

6.1 Trois architectures de synchronisation

6.1.1 Exemple introductif : coopération entre producteur et consommateur

Considérons un couple de processus où l'un, le producteur, envoie des messages à l'autre, le consommateur. Les messages produits, non encore consommés, sont stockés dans des tampons. Ceux-ci sont au nombre fixe de N et utilisés circulairement : un tampon dont le message a été consommé peut être réutilisé (voir figure 6.1).

Dans ce chapitre, nous nous limiterons au cas d'un processus de type producteur et un processus de type consommateur qui communiquent via un tampon à N cases. Dans le chapitre suivant, des éléments permettront d'étudier comment les solutions proposées se généralisent au cas de P processus producteurs et C processus consommateurs.

Le prototypage d'un programme de coopération entre un producteur et un consommateur peut donc s'écrire en ADA de la façon suivante : l'action *traiter* (respectivement *créer*) ne concerne que le processus consommateur (respectivement producteur), et l'action *lire* (resp. *écrire*) nécessite une synchronisation entre les processus. Cette structure conviendra encore lorsque nous considérerons plusieurs producteurs et plusieurs consommateurs. Dans un premier temps, le tampon (la ressource) est supposé visible par l'ensemble des processus. Le programme suivant met en place les tâches *producteur* et *consommateur* d'un programme ADA pour ce problème.

```
procedure producteur_consommateur is
```

```
   N integer:= constante;
   type contenu_tampon is integer;                    -- ou un autre type au choix
   tampon: array (1.. N) of contenu_tampon;
   task producteur is
   end producteur;
   task consommateur is
   end consommateur;
   task body consommateur is
      elt: contenu_tampon;
      procedure traiter (p: in contenu_tampon) is
      begin ...
      end traiter;
      procedure lire ( p: out contenu_tampon) is
      begin ...
      end lire;
   begin
      loop
         lire (elt);
         traiter (elt);
      end loop;
   end consommateur;
   task body producteur is
      elt: contenu_tampon;
      procedure creer (p: out contenu_tampon) is
      begin ...
      end creer;
      procedure ecrire (p: in contenu_tampon) is
      begin ...
      end ecrire;
   begin
      loop
         creer (elt);
         ecrire (elt);
      end loop;
   end producteur;
begin
   null;
end producteur_consommateur;
```

La ressource tampon est définie par deux primitives **concurrentes** de manipulation pour la lecture et l'écriture. Ces deux primitives peuvent être exécutées en parallèle, à la seule condition de manipuler des **cases différentes**. On peut implanter ces primitives en rajoutant au programme précédent deux tâches *read* et *write*. Celles-ci gèrent des indices de production et de consommation par incrémentation modulo N après lecture ou écriture d'un tableau circulaire. Le *tampon* n'est accessible qu'à partir de ces deux primitives.

```
task read is
     entry element (v: out contenu_tampon);
end read;
task write is
     entry (element (v: in contenu_tampon);
end write;

task body read is
     icons: integer:= 1;
begin
     loop
        accept element (v out: contenu_tampon) do
           v := tampon (icons);
        end element;
        icons := (icons mod N) + 1;
     end loop;
end read;

task body write is
     iprod: integer:= 1;
begin
     loop
        accept element (v in: contenu_tampon) do
           tampon (iprod) := v;
        end element;
        iprod := (iprod mod N) + 1;
     end loop;
end write;
```

Ainsi, les procédures *lire* et *écrire* du programme précédent deviennent :

```
procedure lire (p: out contenu_tampon) is
begin
...
read.element (p);
...
end lire;

procedure ecrire(p: in contenu_tampon) is
begin
     ...
     write.element (p);
```

```
    ...
end ecrire;
```

Dans ce programme ADA, il existe 4 tâches concurentes avec des synchronisations dues à l'appel d'entrée entre *consommateur* et *read* d'une part et *producteur* et *write* d'autre part. Mais il n'existe aucun contrôle d'accés aux tampons. Plus précisément, ce contrôle doit ne permettre une consommation que si au moins une production (non encore consommée) a eu lieu, et ne permettre une production que s'il reste des cases non occupées dans le tampon. Mais il est important d'autoriser l'exécution parallèle d'une production et d'une consommation lorsqu'elles concernent deux cases distinctes du tampon.

Dans la suite, la spécification des synchronisations correspondantes à ce contrôle est explicitée à travers la notion de variables d'état. En effet, si pour les algorithmes vus dans les chapitres 4 et 5, les variables (ou données) font partie de l'énoncé du problème, dans les algorithmes définissant des synchronisations, la définition des variables est à la charge du concepteur de l'algorithme et ce choix peut se révéler important.

6.1.2 Spécification et variables d'état

La spécification d'un problème de synchronisation fait intervenir :

- des variables définissant l'état du système,
- des conditions, fonctions de ces variables d'état et qui déterminent l'évolution des processus en certains points particuliers, appelés points de synchronisation,
- et enfin des modifications à effectuer sur ces variables, aux points de synchronisation.

Par ailleurs, nous faisons l'hypothèse, au niveau de la spécification, que les évaluations des conditions, comme les modifications des variables d'état, effectuées par un observateur extérieur sont cohérentes, c'est-à-dire sans interférences entre elles ; l'observateur est supposé infiniment rapide (ou ce qui revient au même, il a le pouvoir d'arrêter le temps) pour évaluer une condition. En effet, si tel n'est pas le cas, la valeur des variables peut changer lors de cette évaluation.

Dans une mise en œuvre réelle, il nous faut respectivement :

- choisir un mode de représentation des variables d'état,
- définir comment effectuer les évaluations des conditions et les modifications des variables, à la place de l'observateur-contrôleur hypothétique, en respectant ce principe de cohérence.

Les variables d'état : précautions d'emploi

La méthode de base pour assurer ce principe de cohérence (éviter les interférences entre les évaluations des conditions de franchissement d'un point de synchronisation et les modifications des variables d'état) consiste à placer les variables dans une mémoire unique partagée et d'y accéder en **exclusion mutuelle.** Chaque opération devient atomique (ou indivisible), ce qui est une façon de simuler le fait que dans la spécification du problème l'opération doit être instantanée. En effet, si une opération sur une variable d'état est effectuée en exclusion mutuelle de façon atomique par un processus, le temps s'arrête, pour les autres processus en ce qui concerne cette variable qu'ils ne peuvent absolument pas accéder.

Un tel mécanisme d'exclusion mutuelle existe dans pratiquement chaque machine et permet d'avoir l'atomicité sur des instructions élémentaires du processeur (par masquage d'interruption par exemple). Par contre, ce mécanisme existant au niveau des instructions élémentaires du processeur n'est pas transféré au niveau de toutes les instructions d'un langage évolué. Aussi les langages parallèles offrent des primitives (ou instruction atomique) en exclusion mutuelle (par exemple le **accept** en ADA).

Au delà de cette méthode particulièrement bien adaptée aux systèmes centralisés (ceux qui sont mis en oeuvre sur une machine disposant d'une mémoire unique partagée par un nombre quelconque de processeurs), il faut considérer le cas des systèmes distribués. Dans ce cas, on peut placer les variables d'état dans la mémoire de l'une des machines et il faut définir ce que représente un mécanisme d'exclusion mutuelle distribué. Un autre cas est celui où l'on distribue les variables d'état dans les différentes mémoires des machines constituant le système et il faut définir un moyen d'en assurer la cohérence des accès et modifications.

Les variables d'état : mode d'emploi

Une spécification d'un problème de synchronisation amène à exprimer des conditions qui autorisent ou non l'exécution d'une action en fonction de l'état du système. Dans la suite,

condition (x) signifie condition pour autoriser l'exécution de l'action x.

Ces conditions s'expriment en fonction des variables d'état. L'état du système étant caractérisé par les actions que les processus ont déjà exécutées et par celles qui sont en cours d'exécution, un premier choix immédiat pour les variables d'état est un ensemble de compteurs qui mémorisent le début et la fin de chaque action. Ce choix privilégie une représentation du système par l'état de ses processus. Un autre choix, que nous illustrerons également sur l'exemple des *producteur_consommateur*, consiste à représenter le système par l'état des ressources partagées (ici le tampon), ce qui est suffisant quand le nombre limité de ressources est la cause des problèmes de synchronisation. Ces deux visions sont souvent duales.

Dans notre exemple, les problèmes de synchronisation se situent au niveau des actions de lecture et d'écriture dans le tampon. Il est naturel, pour ces actions, de distinguer ce qui concerne la synchronisation – obtention d'autorisation d'accès, mise à jour des variables d'état – de l'opération d'accès au tampon. Ainsi, les procédures *lire* et *écrire* se décomposent de la manière suivante :

```
procedure ecrire (p: in contenu_tampon) is
begin
     attendre que condition_ecriture soit vraie
     mettre a jour les variables d'etat: une ecriture a commence
     ecrire dans le tampon
     mettre a jour les variables d'etat: l'ecriture est terminee
end ecrire;

procedure lire (p: out contenu_tampon) is
begin
     attendre que condition_lecture soit vraie
     mettre a jour les variables d'etat: une lecture a commence
     lire dans le tampon
     mettre a jour les variables d'etat: la lecture est terminee
end lire;
```

Dans un premier temps, exprimons plus précisément la synchronisation en fonction du choix des variables d'état.

Les variables d'état : représentation de l'état des processus

L'évolution des processus est représentée au moyen de quatre variables correspondant aux quatre compteurs (initialisés à 0) :

- *nb_début_écriture* représentant le nombre de débuts d'écritures,
- *nb_fin_écriture* représentant le nombre de fins d'écritures,
- *nb_début_lecture* représentant le nombre de débuts de lectures,
- *nb_fin_lecture* représentant le nombre de fins de lectures.

En fonction de ces variables, la synchronisation s'exprime de la manière suivante :

- *condition* (écriture) : *nb_début_écriture* - *nb_fin_lecture* $< N$
- une écriture a commencé : *nb_début_écriture* := *nb_début_écriture* + 1
- l'écriture est terminée : *nb_fin_écriture* := *nb_fin_écriture* + 1
- *condition* (lecture) : *nb_fin_écriture* - *nb_début_lecture* > 0
- une lecture a commencé : *nb_début_lecture* := *nb_début_lecture* + 1

– la lecture est terminée : *nb_fin_lecture* := *nb_fin_lecture* + 1

Note. Les compteurs utilisés peuvent croître indéfiniment, ce qui n'est pas implémentable sur une machine réelle ; aussi faut-il choisir de représenter ces variables entières comme des entiers modulo ; dans le cas ci-dessus, il est nécessaire et suffisant de choisir des entiers modulo $N + 1$. Donc, il suffit de représenter les quatre compteurs comme des variables entières et de les incrémenter modulo $N + 1$.

Les variables d'état : représentation de l'état des ressources

Dans ce cas, le système est représenté par l'état des ressources. Dans notre exemple, l'occupation du tampon peut être représentée par deux variables *nombre_de_plein* et *nombre_de_vide*, initialisées respectivement à 0 et à N, ce qui correspond au changement de variables suivant par rapport à la solution précédente :

– *nombre_de_plein* = *nb_fin_écriture*- *nb_début_lecture*

– *nombre_de_vide* = N + *nb_fin_lecture*- *nb_début_écriture*

En fonction de ces variables, la synchronisation est spécifiée comme suit :

– *condition* (écriture) : *nombre_de_vide* > 0

– une écriture a commencé : *nombre_de_vide* := *nombre_de_vide* - 1

– l'écriture est terminée : *nombre_de_plein* := *nombre_de_plein* + 1

– *condition* (lecture) : *nombre_de_plein* > 0

– une lecture a commencé : *nombre_de_plein* := *nombre_de_plein* - 1

– la lecture est terminée : *nombre_de_vide* := *nombre_de_vide* + 1

Cet ensemble de variables d'états donne une représentation moins fine de l'état du système, mais cependant suffisante pour exprimer la synchronisation. Dans la suite, nous choisirons chaque fois que ce sera possible, les variables *nombre_de_vide* et *nombre_de_plein* qui correspondent à une vision plus simplificatrice (voire plus intuitive) de l'état du système.

Profitons de cet exemple pour illustrer l'importance de l'atomicité des opérations de contrôle. Dans un langage comme ADA, l'opération *nombre_de_vide* := *nombre_de_vide* - 1 n'est pas atomique (i.e. l'addition n'est pas atomique) car elle est composée des opérations assembleur :

```
load nombre_de_vide ; add -1 nombre_de_vide ; store nombre_de_vide ;
```

Imaginons que cette instruction soit interrompue après le **load** par l'exécution de *nombre_de_vide* := *nombre_de_vide* + 1 puis reprenne ensuite. Le résultat est la séquence d'instructions assembleur :

load *nombre_de_vide* ; **load** *nombre_de_vide* ; **add** +1 *nombre_de_vide* ; **store** *nombre_de_vide* ;
add -1 *nombre_de_vide* ; **store** *nombre_de_vide* ;

L'opération d'addition sur la variable *nombre_de_vide* n'a pas été enregistrée ce qui n'est pas du tout l'effet voulu. Ceci aurait été évité en assurant l'atomicité de l'opération ou d'une séquence d'opération qui la contient.

Il reste maintenant à programmer effectivement les procédures *lire* et *écrire*. Dans l'analyse que nous avons faite, nous avons pris soin de dissocier ce qui concerne la synchronisation – attendre qu'une condition soit vraie, mettre à jour les variables d'état – qui concerne l'autorisation d'accès à la ressource et ce qui concerne l'accès au tampon proprement dit.

6.1.3 Synchronisation centralisée ou distribuée

La synchronisation doit d'une part garantir l'utilisation correcte de la ressource, d'autre part mettre en œuvre le maximum de parallélisme dans l'utilisation de cette ressource. Dans ce paragraphe, trois modes de synchronisation sont présentés et illustrés à travers l'exemple du *producteur_consommateur*. Exprimer la synchronisation repose sur le choix d'une part de la représentation des variables d'état et d'autre part sur la représentation des instructions de contrôle . On appelle instruction de contrôle les instructions qui manipulent les variables d'état, stoppent et redémarrent les processus. En première approche, il existe deux expressions extrêmes de la synchronisation (ou du contrôle) :

- **centralisée** : les variables d'état et les instructions de contrôle sont gérées par un processus spécifique, chargé de gérer les variables d'état et d'exécuter les instructions de contrôle lorsqu'il est invoqué par les processus à synchroniser. Ce mode de contrôle est celui qui est préconisé dans ADA, car ce langage offre les bonnes primitives de synchronisation pour le faire. Ce processus spécifique est appelé *serveur*.
- **distribuée** : les variables d'état et les instructions de contrôle sont distribuées : elles sont respectivement gérées et effectuées par les processus du système parallèle. Les variables d'état sont dites *locales* et leurs valeurs sont des données échangées entre les processus, et les instructions de contrôle sont placées dans le texte des processus.
- Entre ces deux extrêmes, une troisième solution peut être envisagée: les variables sont regroupées au sein d'une même entité statique (i.e. pas un processus mais un module ou une mémoire partagée par exemple) et les instructions de contrôle sont directement effectuées par les processus. Deux variantes syntaxiques sont alors possibles pour la représentation des instructions : elles sont soit **dispersées** dans le texte des processus, soit **regroupées** au sein d'un même module (auquel on a donné le nom de moniteur).

Ces différentes formes de synchronisation sont valides et intéressantes dans tous les cas, mais pas efficaces dans le même cadre d'exécution : on ne programme efficacement pas de la même façon

un parallélisme simulé sur un seul processeur (multi-programmation) et un parallélisme effectif sur un réseau de processeurs. Nous présentons maintenant les trois architectures de synchronisation esquissées ci-dessus, en allant de la vision centralisée (la plus naturelle), à la vision distribuée. L'adéquation des différentes architectures de synchronisation et des architectures matérielles sera abordée au paragraphe 5.

Le tableau ci-dessous schématise les choix disponibles :

6.2 Synchronisation centralisée

Dans ce mode de synchronisation, on regroupe dans un processus créé à cet effet la déclaration des variables d'état et toutes les instructions qui les manipulent, sous une forme qui colle de très près à la spécification. Ce processus est sollicité par les processus coopérants lorsque ceux-ci arrivent à des instructions nécessitant de la synchronisation. Ce processus particulier peut être vu comme un serveur: il agit comme un accepteur (ou interpréteur) du langage défini sur les points de synchronisation (voir figure 6.2). Par définition, un processus ne peut exécuter qu'une seule instruction à la fois, ce qui assure l'exclusion mutuelle des opérations de contrôle. Ce processus est le seul à pouvoir manipuler ces variables de contrôle, ce qui assure l'atomicité des opérations (en effet une opération sur une variable d'état ne pourra pas être interrompue par une autre opération sur cette même variable d'état par un autre processus puisque aucun autre processus n'est à même de le faire). Par contre, il faut assurer une exclusion mutuelle des sollicitations envoyées par les processus. Ces sollicitations sont en général des appels dont un seul est accepté à la fois par le serveur. Lorsque les processus effectuent un appel, ils doivent attendre que le serveur l'accepte.

Les instructions utiles à l'écriture d'une solution du type serveur centralisé utilisent donc les fonctionnalités suivantes :

- *appel au serveur* : cette instruction est utilisée par les processus coopérants et *provoque leur blocage* jusqu'à l'acceptation du serveur ;
- *acceptation par le serveur* , quand la condition d'avancement du processus appelant (client) est satisfaite, traitement de l'appel et *relance* du processus client. Le blocage d'un processus

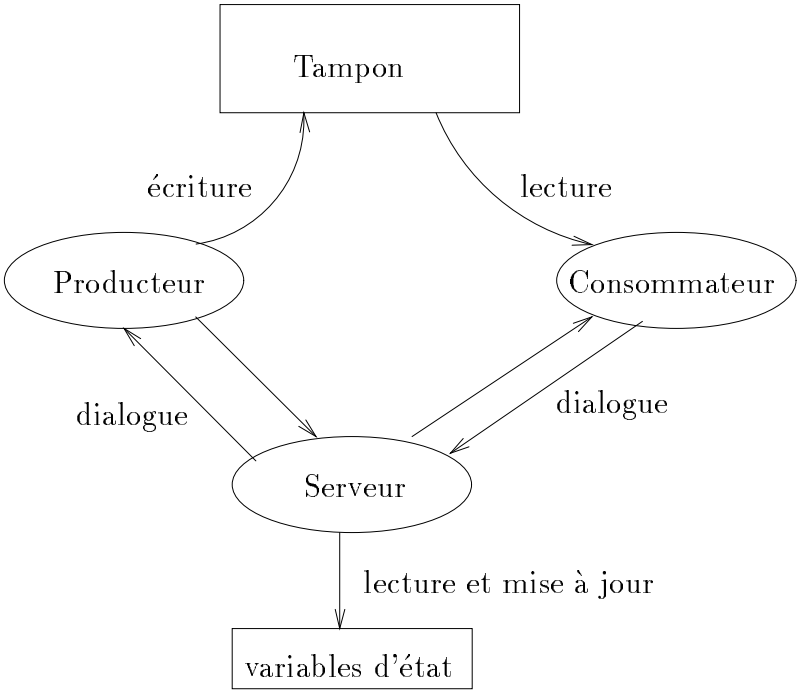


FIG. 6.2 – *producteur-consommateur centralisé*

peut être plus ou moins long suivant la valeur des conditions d'avancement. Le serveur est amené, lorsque les valeurs des variables d'état changent, à réexaminer les conditions associées aux processus bloqués, ou tout au moins les conditions qui risquent d'avoir été modifiées par le changement d'état.

Pour l'exemple du producteur-consommateur, cette architecture de synchronisation peut s'exprimer aisément à l'aide d'une nouvelle tâche *serveur* , au même niveau que les tâches *producteur* et *consommateur*. Ceci est shématisé dans la figure 6.2. Les entités actives (les processus) sont représentés par des cadres aux coins arrondis et les entités passives (variables, modules) par des cadres aux coins anguleux. Les relations directes entre ces entités sont représentées par des flèches.

La tâche serveur est donc la suivante :

```
task serveur is
     entry debut_lecture ;
     entry fin_lecture ;
     entry debut_ecriture ;
     entry fin_ecriture ;
end serveur ;

task body serveur is
     nombre_de_plein : integer:= 0 ;
```

```
    nombre_de_vide : integer := N ;
begin
    loop
       select
          when (nombre_de_plein > 0) =>              -- un consommateur peut consommer
             accept debut_lecture ;
             nombre_de_plein := nombre_de_plein - 1 ;
       or
          accept fin_lecture ;
          nombre_de_vide := nombre_de_vide + 1 ;
       or
          when (nombre_de_vide > 0): =>                  -- un producteur peut produire
             accept debut_ecriture ;
             nombre_de_vide := nombre_de_vide - 1 ;
       or
          accept fin_ecriture ;
          nombre_de_plein := nombre_de_plein + 1 ;
       end select ;
    end loop ;
end serveur ;
```

Les corps des procédures *lire* et *écrire* s'écrivent alors :

```
procedure lire (p : out contenu_tampon) is
begin
    serveur.debut_lecture ;        -- attendre condition_lecture et gestion des variables d'etat
    read.element (p) ;                                                  -- lire dans le tampon
    serveur.fin_lecture ;                                  -- gestion des variables d'etat
end lire ;

procedure ecrire (p : in contenu_tampon) is
begin
    serveur.debut_ecriture;        -- attendre condition_ecriture et gestion des variables d'etat
    write.element (p) ;                                                  -- ecrire dans le tampon
    serveur.fin_ecriture ;                                 -- gestion des variables d'etat
end ecrire ;
```

Il faut remarquer que le langage ADA est fait pour que la synchronisation soit exprimée de façon centralisée à l'aide d'un serveur. Les instructions qui permettent de gérer la synchronisation sont celles qui sont nécessaires à la construction d'un serveur : d'une part les appels d'entrées sont gérés en exclusion avec une file FIFO, d'autre part l'envoi de message est disymétrique (l'appelant envoie un message – le client est roi – et le serveur le stocke dans sa file d'attente). D'autres langages (comme OCCAM) proposent des envois de messages symétriques : les deux processus doivent s'appeler

mutuellement (par des primitives send et receive exécutées simultanément par les deux processus) pour que l'envoi de message ait lieu. De plus l'appelant en ADA connaît l'identité du serveur qu'il appelle mais le serveur ne connaît pas le nom de ses clients. Dans d'autres langages, la symétrie est préservée et les primitives *send* et *receive* doivent nommer chacun le processus partenaire ou bien nommer un objet, appelé canal qui leur permet de communiquer.

6.3 Synchronisation mixte

Dans les solutions présentées dans ce paragraphe, les variables d'état sont regroupées comme précédemment, mais cette fois les instructions de contrôle sont distribuées, c'est-à-dire qu'elles sont directement exécutées par les processus du système parallèle. Il n'y a donc plus de processus serveur (qui seul a accès aux variables d'état) et on dit que les **variables sont partagées.** Deux approches sont possibles : dans la première, les instructions de contrôle sont regroupées au sein d'un module partageable, dans la deuxième, elles sont dispersées dans le texte des processus.

6.3.1 Module partageable

Du point de vue syntaxique, la forme est analogue à celle de la solution précédente. Un module regroupe variables et procédures de contrôle (voir figure 6.3). Les processus coopérants appellent les procédures aux moments nécessaires. Ces appels remplacent les envois de messages au processus serveur. Cependant, il y a maintenant possibilité d'exécution parallèle entre plusieurs procédures. Il est donc nécessaire d'instaurer une exclusion mutuelle entre les procédures qui accèdent aux mêmes variables. De plus, lors de l'exécution de certaines procédures, le processus doit se bloquer si la condition d'avancement n'est pas satisfaite. Inversement, des opérations de libération doivent être réalisées.

Les primitives relatives à cette mise en œuvre sont donc les suivantes :

- *appel de procédure de module* : cette primitive est un appel de procédure en exclusion mutuelle avec toutes les autres procédures du module. Ceci assure l'atomicité des opérations de contrôle et bien sûr leur exclusion mutuelle.
- *blocage d'un processus* : si le processus en cours d'exécution d'une procédure de contrôle trouve une condition de progression fausse, il doit *se bloquer en libérant l'exclusion mutuelle* sur le module ;
- *retour de procédure* : la fin d'une procédure relâche l'exclusion mutuelle ;
- *libération de processus* : lorsque des modifications sont apportées aux variables d'état, les conditions de progression des processus doivent être réévaluées et les processus bloqués doivent être libérés si ces conditions sont vérifiées. Là se pose un des problèmes majeurs de cette solution : la libération des processus doit maintenir le principe de l'exclusion mutuelle. Les processus libérés doivent donc s'exécuter les uns après les autres et sans interférence avec le processus qui les a libérés.

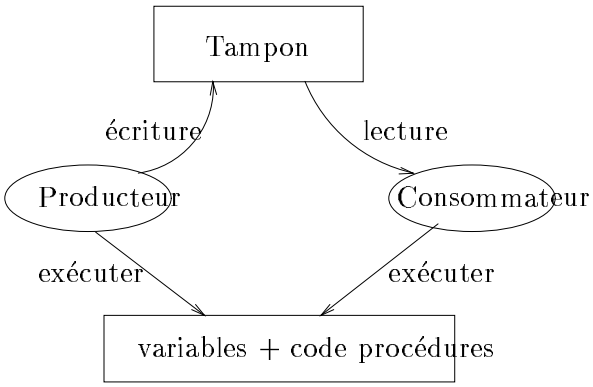


FIG. 6.3 – *module partageable*

Ce mode de synchronisation correspond à la notion de moniteur introduite par Hoare en 1974 [Hoa 74] et il est relativement difficile à exprimer en ADA. Aussi en donnons-nous une expression "approchée" en ADA.

Le Moniteur

Un moniteur est une entité syntaxique qui regroupe les objets définis pour la synchronisation (variables d'état, files d'atttente) et les procédures qui les manipulent. Par définition, les variables déclarées dans le moniteur ne sont accessibles que par les procédures du moniteur. Parmi ces procédures, certaines sont internes aux moniteurs, d'autres peuvent être appelées de l'extérieur, elles constituent les **points d'entrée** du moniteur.

Deux types de synchronisation sont définis dans un moniteur :

- exclusion mutuelle, réalisée de manière implicite, entre les procédures du moniteur.
- définition explicite de variables de synchronisation, appelées *conditions*, sur lesquelles ont peut appliquer trois opérateurs : *wait*, *signal* et *empty*.

L'opération *wait c* permet d'exprimer des attentes sur la condition *c* en bloquant les processus qui exécutent cette primitive dans la file d'attente associée implicitement à chaque condition. Cette file est gérée en mode FIFO (First In, First Out) L'opérateur *empty c* permet de tester si la file relative à la condition *c* est vide. L'opération *signal c* libère un processus en attente sur la condition *c*, s'il y en a au moins un, est sans effet sinon. Pour que l'exclusion mutuelle soit toujours vérifiée, le processus qui a exécuté l'instruction *signalc* se trouve à son tour bloqué et ceci tant qu'il y a un processus "en activité dans le moniteur", c'est-à-dire en train d'exécuter une instruction du moniteur. Lorsqu'un processus "quitte" le moniteur (en exécutant une instruction de fin de procédure ou une instruction *wait*), on vérifie, avant d'introduire un nouveau processus en attente sur un point d'entrée, qu'il n'y a plus de processus bloqués sur une condition qui se trouverait validée. Nous illustrons l'utilisation de cet outil pour un producteur-consommateur (avec une syntaxe "à la ADA").

```
Moniteur PROD_CONS is
    nombre_de_plein : integer := 0 ;
    nombre_de_vide : integer := N ;
    file_prod , file_cons : condition ;

    procedure debut_consommation is
    begin
       if nombre_de_plein = 0 then wait file_cons; end if;
       nombre_de_plein := nombre_de_plein - 1 ;
    end debut_consommation ;

    procedure fin_consommation is
    begin
       nombre_de_vide := nombre_de_vide + 1 ;
       signal file_prod;
    end fin_consommation ;

    procedure debut_production is
    begin
       if nombre_de_vide = 0 then wait file_prod ; end if;
       nombre_de_vide := nombre_de_vide - 1 ;
    end debut_production ;

    procedure fin_production is
    begin
       nombre_de_plein := nombre_de_plein + 1 ;
       signal file_cons;
    end fin_production ;
end PROD_CONS;
```

Les corps des procédures lire pour les processus consommateurs et ecrire pour les processus producteurs s'écriraient alors :

```
procedure lire (p : out contenu_tampon) is
begin
    debut_consommation ;
    read.element (p) ;
    fin_consommation ;
end lire ;
```

```
procedure ecrire (p: in contenu_tampon) is
begin
     debut_production ;
     write.element (p) ;
     fin_production ;
end ecrire ;
```

On remarque que dans la forme du moins les écritures en terme de moniteur et de serveur de ce problème sont très similaires. Elles procèdent de la même idée : encapsuler dans un serveur ou dans un module les instructions de contrôle, l'activation du serveur ou du module gérant l'exclusion mutuelle.

Il semble que la gestion de la file d'attente des processus soit plus naïve donc plus simple dans le cas du serveur. Afin de montrer plus finement la relation entre les deux, une programmation du moniteur en ADA est proposée en annexe de ce chapitre.

Limites du moniteur

Du point de vue efficacité, toutes les procédures d'un moniteur sont en exclusion mutuelle, ce qui n'est pas toujours nécessaire (cette remarque vaut aussi pour le serveur). C'est vrai dans notre exemple où il est nécessaire de placer sous la même exclusion mutuelle uniquement les couples *début_consommation* / *fin_production* d'une part et *fin_consommation* / *début_production* d'autre part, puisqu'ils travaillent sur les mêmes variables d'état.

Mais un des principaux inconvénients du concept de moniteur est l'utilisation de la primitive *signal*. Ces instructions de signalisation peuvent apparaître n'importe où dans le corps d'une procédure, ce qui peut donner des politiques de libération de processus inattendues et sont sources d'erreurs difficiles à détecter. Les variantes les plus intéressantes des moniteurs sont celles qui fournissent des opérations de signalisation implicites, placées systématiquement en fin de procédure (ce qui est le cas dans notre exemple). On trouve une formulation totalement implicite dans les modules de contrôle de Robert et Verjus [RoV 77].

6.3.2 Variables partagées et exclusion mutuelle ou sémaphore.

Dans cette solution, les variables partagées sont manipulées directement par les processus, les instructions de contrôle étant placées dans les programmes particuliers des processus (voir figure 6.4) et non plus encapsulées dans les procédures d'un moniteur. A chaque point, seules les instructions nécessaires sont incluses. Comme dans la solution précédente, les processus s'exécutent en parallèle et l'exclusion mutuelle doit être programmée pour assurer, quand elle est utile, l'indivisibilité des opérations de contrôle. Cette fois, l'exclusion peut porter sur des séquences quelconques d'instructions (et non plus sur des procédures entières).

Le premier outil associé à cette synchronisation est le *sémaphore* introduit par Dijkstra en 1968[Dij 68]. Un sémaphore S est une structure de données composée d'un compteur s à valeurs

entières et d'une file d'attente *f*, initialement vide, que l'on considérera comme gérée en mode FIFO. La valeur initiale de *s* peut être positive ou nulle. Après l'initialisation, deux primitives permettent de manipuler le sémaphore *S*: *P* et *V*. Ces primitives sont indivisibles et mutuellement exclusives. Leur sémantique est la suivante :

P (S) :
    s := s - 1 ;
    si s < 0 alors mettre le processus en attente dans la file f
    fsi ;

V (S) :
    s := s + 1 ;
    si s <= 0 alors reveiller un processus de la file f
    fsi ;

Un processus qui exécute *P* est bloqué si *s* est négatif ou nul. Dans ce cas, seule une opération *V* peut le libérer. Quand *s* est négatif, sa valeur absolue représente le nombre de processus en attente, quand il est positif, il indique le nombre maximum d'exécutions simultanées possibles.

Sémaphore et section critique

La programmation d'une séquence d'instructions en exclusion mutuelle ou section critique *SC* est la suivante avec un sémaphore: un sémaphore *MUTEX* est déclaré avec valeur initiale 1 pour représenter le nombre maximum d'exécutions simultanées de la séquence *SC*. Avant toute exécution de *SC*, un processus doit effectuer l'instruction *P*(*MUTEX*) qui le bloque si un processus est déjà en cours d'exécution de *SC*. A la fin de la séquence *SC*, la primitive *V*(*MUTEX*) est appelée, libérant éventuellement un processus qui rentre à son tour en section critique.

Cet outil est suffisant pour exprimer la synchronisation de processus en univers centralisé (c'est-à-dire dans un système où les processus se partagent une mémoire commune). On protège l'accès aux variables de synchronisation par la définition d'un sémaphore (un par variable ou groupe de variables) et les processus exécutent les primitives *P* et *V* respectivement avant et après chaque procédure de synchronisation. La figure 6.4 montre cette architecture de synchronisation pour l'exemple du producteur-consommateur.

Dans cette programmation on reconnaît les instructions qui étaient dans le moniteur, maintenant dans le texte même des processus. La déclaration du sémaphore d'exclusion mutuelle et les corps des procédures *lire* pour les processus consommateurs et *écrire* pour les processus producteurs s'écrivent alors :

MUTEX : Semaphore(s => 1) ;
nombre_de_plein, nombre_de_vide : *integer* := N, 0 ;

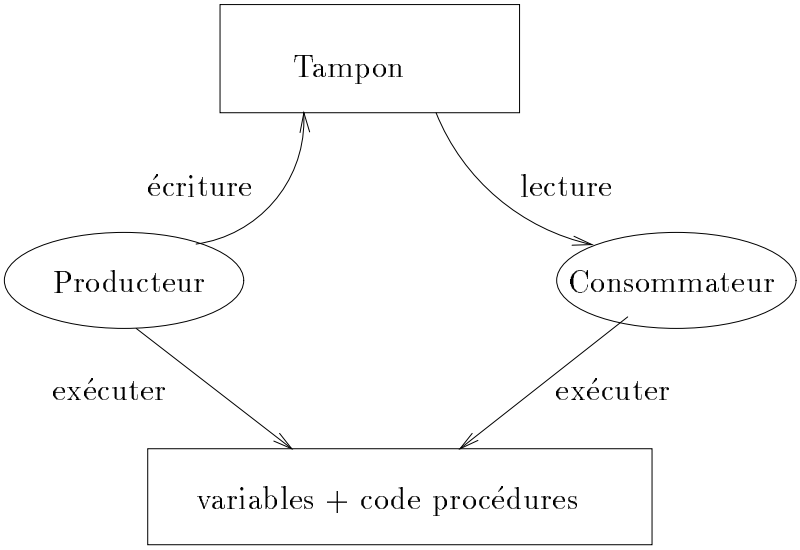


FIG. 6.4 – *Synchronisation mixte*

```
procedure lire (p : out contenu_tampon) is
    autorisation_lire : boolean := false ;
begin
    while (not autorisation_lire) loop                    -- boucle d'attente active
       P(MUTEX) ;                  -- exclusion mutuelle pour l'acces la la variable partagee
       if (nombre_de_plein <> 0) then
          autorisation_lire := true ;
          nombre_de_plein := nombre_de_plein - 1 ;
       end if ;
       V(MUTEX);                                          -- liberation de l'exclusion mutuelle
    end loop ;
    read.element(p) ;
    P(MUTEX) ;
    nombre_de_vide := nombre_de_vide - 1 ;
    V(MUTEX) ;
end lire ;

procedure ecrire (p : in contenu_tampon) is
    autorisation_ecrire : boolean := false ;
begin
    while (not autorisation_ecrire) loop                  -- boucle d'attente active
       P(MUTEX) ;
       if (nombre_de_vide <> 0) then
          autorisation_ecrire := true ;
          nombre_de_vide := nombre_de_vide - 1;
       end if ;
       V(MUTEX) ;
```

```
    end loop;
    write.element (p);
    P(MUTEX);
    nombre_de_plein := nombre_de_plein + 1;
    V(MUTEX);
end ecrire;
```

Sémaphore et représentation de l'état d'un système.

Les sémaphores peuvent en outre être utilisés également directement dans quelques cas particuliers pour représenter les variables de synchronisation, quand celles-ci se résument à des compteurs prenant des valeurs positives ou nulles et que les conditions de blocage de processus peuvent s'exprimer comme un test à zéro sur ces variables.

Pour le problème du producteur-consommateur, cette utilisation est particulièrement adaptée et permet d'aboutir à la programmation ci-dessous :

```
plein : Semaphore (s => 0);
vide : Semaphore (s => N);

procedure lire (p : out contenu_tampon) is
begin
    P(plein);
    read.element (p);
    V(vide);
end lire;

procedure ecrire (p : in contenu_tampon) is
begin
    P(vide);
    write.element(p);
    V(plein);
end ecrire;
```

Programmation d'un sémaphore en ADA.

On peut montrer que les divers outils de synchronisation sont équivalents dans le sens où il est possible de programmer les uns avec les autres. En ADA, l'outil de base est le serveur. A l'aide de deux serveurs de type *exclusion_mutuelle* et *file_d_attente* introduits au chapitre 3, un sémaphore

peut être programmé en ADA de la façon suivante : le sémaphore est un enregistrement comprenant un compteur et deux tâches. Des procédures P et V de manipulation sont définies.

```
type Semaphore (s : positive := 0) is
     record
        em : exclusion_mutuelle ;
        f : file_d_attente ;
     end record ;

procedure P (S : in Semaphore) ;
procedure V (S : in Semaphore) ;

procedure P (S : in Semaphore) is
begin
     S.em.entrer ;                                  -- appel au serveur de section critique
     S.s := S.s - 1 ;
     if (S.s < 0) then
        S.em.sortir ;
        S.f.endormir ;                                -- liberation de la section critique
     else
        S.em.sortir
     endif ;                                        -- avant de s'endormir si c'est le cas
end P ;

procedure V (S : in Semaphore) is
begin
     S.em.entrer ;
     S.s := S.s + 1 ;
     if (S.s <= 0) then
        S.em.sortir ;
        S.f.reveiller ;                               -- liberation de la section critique
     else
        S.em.sortir
     endif ;                              -- avant de reveiller un autre processus si c'est possible
end V;
```

Cette programmation en ADA nous permet d'avoir une vision plus fine des interactions entre la section critique et les actions de se bloquer (s'endormir) ou de réveiller un processus concurrent.

Limites des sémaphores

Les sémaphores sont des outils primitifs bien adaptés à la synchronisation. Cependant, ils présentent deux inconvénients majeurs mis en évidence dans les exemples précédents :

- les programmes sont peu lisibles car le nombre d'opérations sur sémaphores à effectuer peut être important et ces opérations sont éparpillées dans le texte des processus ;
- les solutions correctes sont difficiles à construire, à moins de programmer très systématiquement à partir des spécifications, ce qui donne alors des solutions peu efficaces du point de vue performance, du fait de l'usage systématique de sections critiques (cf. remarque sur les moniteurs en 3.1.1).

Pour illustrer le premier point de vue, considérons un programme utilisant les deux sémaphores E et L, initialisés à 1. Ce programme exécute en parallèle p exemplaires du processus P_i et q exemplaires du processus Q_j (p et q sont quelconques). Les deux modèles de chacun de ces processus sont les suivants :

```
nl: integer:= 0 ;
task Pi is
end Pi;
task Qj is
end Qj;

task body Pi is
begin
     P(L) ;
     nl := nl+1;
     if nl = 1 then P(E) ;
     V(L) ;
     operation 1;
     P(L) ;
     nl := nl - 1;
     if nl = 0 then V(E) ;
end Pi;

task body Qj is
begin
     P(E) ;
     operation 2;
     V(E) ;
end Qj;
```

Pour analyser ce programme, il faut essayer de retrouver des "formes" connues de combinaisons d'opérations P et V sur un sémaphore. Le sémaphore L, initialisé à 1, assure l'indivisibilité (par une forme de type "section critique") des opérations de comptage du nombre de processus de type P_i en cours d'exécution, grâce à la variable entière globale *nl*. Le sémaphore E , initialisé à 1, est requis (opération$P$) avant chaque opération 2 et libéré après. Ce sémaphore est également modifié par le plus ancien processus de type P_i parmi ceux qui sont en cours d'exécution et par le dernier. E assure donc une exclusion mutuelle, d'une part entre deux quelconques processus de type Q_j, et d'autre part, entre un processus de type Q_j et un au moins processus de type P_i.

Autrement dit, on peut représenter les conditions de fonctionnement de ce système ainsi :

- condition pour exécuter (*opération* 1) : *nb d'opération* 2 = 0
- condition pour exécuter (*opération* 2) : *nb d'opération* 1 + *nb d'opération* 2 = 0

ce qui, nous le verrons dans le chapitre suivant est caractéristique d'un système de lecteurs (les P_i) - rédacteurs (les Q_j).

En conclusion, cette notion de synchronisation mixte est importante pour deux raisons : d'abord une raison historique, puisque les sémaphores ont été dans les premiers outils de synchronisation évolués, suivis par les moniteurs quand le concept de modularité est apparue dans les langages de programmation. La notion de moniteur est, comme on l'a vu, peu éloignée de celle de serveur. La deuxième raison est que les processeurs modernes offrent tous une primitive de synchronisation de base, du même type que le sémaphore (quoique encore plus simple), c'est-à-dire basée sur une variable partagée et accessible par plusieurs processeurs. Certains offrent le **test_and_set/unset** , d'autre le **Lock/unlock.** Le premier couple de primitives s'applique à une variable booléenne et **test_and_set** teste sa valeur à 0, si c'est le cas la met à 1 et rend au processus la valeur 0, sinon rend la valeur 1 et **unset** affecte 0 à la variable**.** L'usage courant de cette primitive est que le processus se bloque ou non en fonction de la valeur de la variable. Le **Lock/unlock** s'applique à n'importe quelle variable et permet à un processus de la verrouiller, c'est-à-dire d'en garder l'usage exclusif jusqu'au déverrouillage. Avec ces outils primitifs, on peut construire des sémaphores, des moniteurs ou des serveurs.

6.4 Synchronisation distribuée

Dans les solutions précédentes de synchronisation mixte, seules les instructions de contrôle étaient distribuées. Nous allons maintenant voir dans quelles conditions les variables peuvent être réparties au sein de chaque processus ou au sein de plusieurs serveurs qui coopèrent (ce qui permettra éventuellement la répartition des processus sur des machines reliées par un réseau et sans mémoire commune, donc pour lesquelles la notion de variable partagées est difficilement implantable efficacement).

6.4.1 L'exemple du producteur-consommateur

Reprenons notre exemple du producteur-consommateur et reportons nous aux spécifications données aux paragraphes 1.2.3 et 1.2.4. Nous avons déjà remarqué que l'exclusion mutuelle unique proposée précédemment n'est pas nécessaire. En effet, lorsque le producteur teste la condition pour *écrire* en début de production (en faisant, selon le choix des variables d'état, (*nb_début_écriture* - *nb_fin_lecture* < *N*) ou (*nombre_de_vide* >*0*), l'exclusion mutuelle doit être assurée avec le consommateur qui modifie la variable *nb_fin_lecture* (en faisant *nb_fin_lecture*:=*nb_fin_lecture* + 1) ou respectivement *nombre_de_vide* (en faisant *nombre_de_vide*:= *nombre_de_vide* - 1). L'autre exclusion mutuelle concerne le début de consommation et la fin de production via la variable *nb_fin_écriture* ou *nombre_de_plein*. On peut donc distribuer les variables d'état dans deux serveurs qui gèrent respectivement les variables *nombre_de_vide* pour l'un et *nombre_de_plein* pour l'autre (ou les compteurs croissants correspondants).

Si l'on fait une observation plus fine de la spécification donnée au paragraphe 1.2.3 utilisant les compteurs croissants (*nb_début_écriture, nb_fin_écriture, etc ...)*, avec l'hypothèse que la lecture prise de valeur d'une variable est atomique (le **load** est en général atomique dans toutes les machines) on constate que :

- les compteurs *nb_début_écriture* et*nb_début_lecture* ne sont utilisés que par un processus, le producteur et le consommateur respectivement. Leur cohérence ne pose aucun problème.
- la variable *nb_fin_lecture* est utilisée simultanément par les deux processus.

  Cependant, l'opération *nb_fin_lecture*:= *nb_fin_lecture* + 1 interfère avec l'opération *nb_début_écriture* - *nb_fin_lecture* < *N,* sans mettre en défaut la cohérence du système. En effet, si l'incrémentation de la variable est commencée mais non terminée (elle a été lue mais non encore modifiée) au moment où sa valeur est prélevée pour évaluer *nb_début_écriture* - *nb_fin_lecture* < *N,* l'inégalité vraie avant l'incrémentation est a fortiori vraie après l'incrémentation.

Cette dernière propriété permet de progresser dans le processus de répartition en l'interprétant de façon différente. En effet, la variable *nb_fin_lecture* (par exemple, la même chose est vraie pour la variable *nb_fin_écriture*) peut être dupliquée, l'exemplaire original étant géré par le processus consommateur, et sa copie accessible par le processus producteur. Il suffit que toute incrémentation de l'original soit reportée sur la copie. Un retard sur le report de l'incrémentation ne met pas en défaut la condition de synchronisation. Donc par "copie" on entend une variable qui prend la même suite de valeurs que l'original, mais dans ce cas, on peut **distribuer complètement le contrôle** (voir figure 6.5) dans le sens où variables d'état et instructions de contrôle sont dispersés dans divers processus.

Nous proposons une solution à ce problème en ADA avec le principe suivant :

- une tâche *producteur*, qui encapsule une variable *nb_début_écriture* et une copie de la variable de comptage *nb_fin_lecture* qui s'appelle *my_nb_fin_lecture* (on aurait pu utiliser le même identificateur *nb_fin_lecture* pour la copie puisque sa portée est limitée à la tâche, ce choix n'étant motivé que par la lisibilité). Cette tâche gère elle-même ses droits d'accès en écriture et la mise à jour de sa copie *my_nb_fin_lecture* lorsque ce droit d'accès ne lui est pas favorable. On

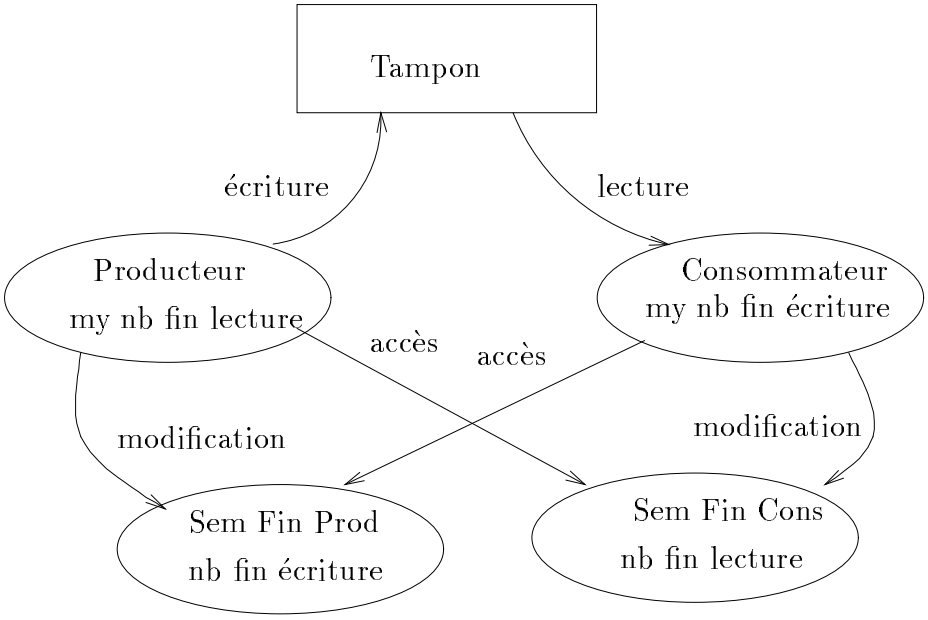


FIG. 6.5 – *Producteur/comsommateur sur une architecture répartie*

appelle ça de la mise à jour "paresseuse". L'avantage est de ne demander la mise à jour que si besoin est, le désavantage est l'attente induite à ce moment là. Des alternatives sont de mettre à jour périodiquement ou lors de temps morts, mais ceci sort du cas simple que nous traitons. La mise à jour du compteur de fin d'écriture se fait par appel à un serveur *Sem_FinCons*.

- une tâche *consommateur*, symétrique de la précédente, qui encapsule une variable *nb_début_lecture* et une copie de la variable de comptage *nb_fin_écriture* qui s'appelle *my_nb_fin_écriture*. Cette tâche gère elle-même ses droits d'accès en lecture et la mise à jour de sa copie *my_nb_fin_écriture* lorsque ce droit d'accès ne lui est pas favorable. La mise à jour du compteur de fin de lecture se fait par appel à un serveur *Sem_FinProd*.

- une tâche *Sem_FinCons* qui encapsule la variable *nb_fin_lecture* et accepte les demandes d'incrémentation de la tâche *consommateur* et les demandes de consultation de la tâche *producteur*. Ces demandes se font donc en exclusion mutuelle dans notre implantation alors qu'on a vu que ceci n'était pas nécessaire. Ceci étant, il se peut, pour de multiples raisons, dont l'indéterminisme de l'instruction **accept**, que la valeur obtenue par consultation ne soit pas la dernière mise à jour. On remarque qu'un mécanisme est mis en place (par la variable booléenne *nouvelle_valeur*), pour donner priorité aux incréments par rapports aux consultations. Sans ce dispositif il est possible que le **select** choisisse indéfiniment de sélectionner une consultation, qui ne donnera pas de valeur différente, donc cette consultation sera réitéré et le programme ne progressera pas. Ce cas est appelé une famine et sera détaillé dans le chapitre suivant.

- une tâche *Sem_FinProd* qui est symétrique de la précédente.

- les tâches *read* et *write* sont inchangées par rapport à la version du paragraphe 1.1

```
task producteur is
end producteur;

task consommateur is
end consommateur;

task Sem_FinCons is
     entry Valeur(v : out integer );
     entry Incrementer;
end Sem_FinCons;

task Sem_FinProd is
     entry Valeur(v : out integer );
     entry Incrementer;
end Sem_FinProd;

task read is
     entry element( v: out contenu_tampon);
end read;

task write is
     entry element( v: in contenu_tampon);
end write;

tampon : array (1..N) of contenu_tampon;

task body producteur is
     nb_debut_ecriture, my_nb_fin_lecture : integer := 0;
     elt : contenu_tampon;
     procedure creer (p : out contenu_tampon) is
     begin ... end creer;

     procedure ecrire (p: in contenu_tampon) is
        autorisation_ecrire : boolean := false;
     begin
     while (not autorisation_ecrire) loop                    -- boucle d'attente active
        if (nb_debut_ecriture - my_nb_fin_lecture < N) then
           autorisation_ecrire := true;
           nb_debut_ecriture := nb_debut_ecriture + 1;
        else
           Sem_FinCons.Valeur (my_nb_fin_lecture);
        end if;
     end loop;
     write.element (p);
     Sem_FinProd.Incrementer;
```

```
     end ecrire;

begin
     loop
        creer (elt);
        ecrire (elt);
     end loop;
end producteur;

task body consommateur is
     nb_debut_lecture, my_nb_fin_ecriture: integer:= 0;
     elt: contenu_tampon;
     procedure traiter (p: in contenu_tampon) is
     begin ... end traiter;

     procedure lire (p: out contenu_tampon) is
        autorisation_lire: boolean:= false;
     begin
        while (not autorisation_lire) loop                    -- boucle d'attente active
           if (my_nb_fin_ecriture - nb_debut_lecture > 0) then
              autorisation_lire:= true;
              nb_debut_lecture:= nb_debut_lecture + 1;
           else
              Sem_FinProd.Valeur(my_nb_fin_ecriture);
           end if;
        end loop;
        read.element (p);
        Sem_FinCons.Incrementer;
     end lire;

begin
     loop
        lire(elt);
        traiter(elt);
     end loop;
end consommateur;

task body Sem_FinCons is
     FinConsommation: integer:= 0;
     nouvelle_valeur: boolean:= false;
begin
     loop
        select
           when nouvelle_valeur => accept Valeur(v: out integer ) do
              v:= FinConsommation;
              nouvelle_valeur:= false;
           end Valeur;
```

```
        or
           accept Incrementer do
              FinConsommation := FinConsommation + 1 ;
              nouvelle_valeur := true;
           end Incrementer ;
        end select ;
     end loop ;
end Sem_FinCons ;

task body Sem_FinProd is
     FinProduction : integer := 0 ;
     nouvelle_valeur : boolean := false;
begin
     loop
        select
           when nouvelle_valeur => accept Valeur(v : out integer ) do
              v := FinProduction ;
              nouvelle_valeur := false;
           end Valeur ;
        or
           accept Incrementer do
              FinProduction := FinProduction + 1 ;
              nouvelle_valeur := true;
           end Incrementer ;
        end select ;
     end loop ;
end Sem_FinProd ;

task body read is
     icons : integer := 0 ;
begin
     loop
        accept element(v : out contenu_tampon) do
           v := tampon ( icons) ;
        end element;
        icons := (icons + 1) mod N ;
     end loop ;
end read ;

task body write is
     iprod : integer := 0 ;
begin
     loop
        accept element(v : in contenu_tampon) do
           tampon(iprod) := v ;
        end element ;
        iprod:=(iprod+ 1) mod N ;
```

```
    end loop;
end write;
```

On obtient un programme découpé en tâches ayant chacune des variables locales et pouvant s'exécuter sur des processeurs qui ne partage par de mémoire commune, l'appel d'entrée pouvant être implémenté par un protocole sur un réseau informatique. Cet exemple montre qu'on peut se passer du mécanisme d'ensemble de variables en copie unique et partagée, manipulée en exclusion mutuelle. On duplique au contraire certaines variables en assurant une cohérence lâche mais contrôlée, entre les deux exemplaires.

Cepandant, la méthode qui a été suivie ici, qui est très instructive pour l'exemple, n'est pas systématiquement applicable (par exemple au cas de multiples producteurs et consommateurs). Ici, nous avons pris le serveur centralisé et nous avons essayé d'éclater dans différentes tâches variables d'état et instructions de contrôle qui les manipulent.

En général, l'intérêt de la synchronisation centralisée est d'être simple à réaliser et l'intérêt de la synchronisation distribuée de résister à la non disponibilité (ponctuelle ou de longue durée) d'une tâche. Dans toute solution centralisée, si la tâche serveur est sur une machine qui devient indisponible (charge forte, panne, etc) rien ne fonctionne plus. Dans la solution présentée ci-dessus, les processus producteur et consommateur ont une certaine licence face à l'indisponibilité de processeurs supports (cet exemple-là n'étant pas un modèle du genre). L'art de construire des algorithmes de synchronisation distribuée s'appelle algorithmique distribuée (réference).

6.5 Architectures de synchronisation et architectures de machines

Les architectures de synchronisation présentées font apparaître deux critères distinctifs :

- les variables d'état du système sont centralisées ou distribuées,
- les instructions de contrôle (modification des variables d'état) sont centralisées dans un moniteur ou un serveur ou sont distribuées dans les processus.

Ces critères peuvent être directement associés aux critères caractéristiques des architectures parallèles :

- la mémoire physique est partagée ou bien distribuée,
- l'unité de calcul est unique ou plusieurs unités existent sans aucune hiérarchie entre elles.

On peut en conséquence distinguer trois niveaux d'environnement parallèle :

- le niveau processeur : le parallélisme est un concept qui permet de rendre compte de l'exécution en temps partagé de plusieurs tâches de façon à utiliser au mieux une même ressource

(le processeur, une imprimante...). Dans ce cas, il y a une seule mémoire et une seule unité de calcul : le parallélisme (on dit pseudo-parallélisme) sera géré efficacement par une synchronisation centralisée avec des variables d'état centralisées sur une zone mémoire accessible par tous les processus. Les instructions de contrôle étant exécutées par un unique processeur peuvent être centralisées au sein d'un même processus serveur (synchronisation centralisée), qui donne l'avantage d'une programmation très lisible et plus naturelle. Mais il est aussi possible de les faire exécuter par les différents processus (synchronisation mixte). Au niveau de la lisibilité, la solution d'un module regroupant les instructions de contrôle (un moniteur par exemple) sera préférée : comme les instructions de contrôle sont les mêmes pour tous les processus, elles peuvent être regroupées au sein d'un même module. La solution "instructions dispersées" (sémaphores) présente cependant l'avantage d'être plus proches des possibilités offertes par les architectures.

- le niveau grappe de processeurs connectés par une mémoire partagée : la présence de plusieurs unités de calcul suggère la distribution des instructions de contrôle. La solution la plus directe est alors une synchronisation mixte : les variables d'état sont placées en mémoire partagée, et les instructions de contrôle sont distribuées. Les instructions de contrôle peuvent être regroupées au sein d'un même module (moniteur), mais la solution n'est réellement envisageable que si les processeurs exécutant les instructions sont identiques, contrainte qui n'est pas nécessaire lorsqu'on utilise des instructions dispersées (exclusion mutuelle par sémaphores). De plus, cette dernière solution permet de décrire plus finement la synchronisation, et donc d'avoir, en général, des programmes plus efficaces : il faut cependant prendre soin à la programmation, l'utilisation brutale des sémaphores conduisant souvent à des programmes illisibles, et donc difficiles à maintenir.

- le niveau réseau de processeurs sans mémoire partagée : la distribution des instructions de contrôle sur les différentes unités permet d'exploiter au mieux les différentes unités. La mémoire étant distribuée, la synchronisation distribuée, avec variables d'état distribuées sur les différents processeurs, apparaît naturellement. Sur ce type d'architecture, il est possible de trouver un outil permettant de simuler une mémoire partagée entre les processeurs. Avec cet outil, la description d'une synchronisation mixte avec variables d'état centralisées est rendue possible, mais ne permet pas une expression aussi fine de la synchronisation que la synchronisation distribuée : l'exemple de la distribution du contrôle dans l'exemple du producteur_consommateur illustre clairement ceci. Ce cas de figure est étudié avec un point de vue plus général dans le chapitre suivant.

Pour une application complète s'exécutant sur une architecture donnée et faisant intervenir différents niveaux de parallélisme, il est fréquent d'utiliser différentes expressions de la synchronisation, chacune adaptée au niveau considéré.

Considérons par exemple une application s'exécutant sur un réseau de processeurs. De manière générale, deux niveaux de parallélisme peuvent alors intervenir :

- le niveau processeur : l'efficacité est apportée par la possibilité d'exécuter en pseudo-parallélisme plusieurs tâches (voir l'exemple donné dans le chapitre applications). A ce niveau, une synchronisation centralisée ou mixte est bien adaptée.

- le niveau réseau : la synchronisation distribuée est alors la mieux adaptée.

6.6 Conclusion

Les expressions de synchronisation présentées ici ne sont pas historiquement apparues dans cet ordre. Chronologiquement, une première catégorie d'outils, les sémaphores, est apparue à l'époque où synchroniser revenait à mettre en œuvre le travail de notre observateur (ou contrôleur), ce qui était facile car les processus se partageaient une mémoire commune où l'on pouvait représenter l'état du système et, une fois comprise l'idée d'exclusion mutuelle, faire fonctionner de manière cohérente les opérations de contrôle.

Une seconde catégorie d'outils est née du fait qu'il fallait lever cette hypothèse d'existence d'une mémoire commune, lorsque les réseaux et les machines à mémoire distribuée sont apparus. Le contrôleur de synchronisation (si tant est que le problème posé permette de formuler son expression) doit être réparti entre les processus et la synchronisation entre eux repose sur des mécanismes de **communication**. Par ailleurs, de nouveaux problèmes apparaissent, pour lesquels le point de vue d'un observateur est peu utile : l'algorithme de contrôle a pratiquement comme seule fonction d'assurer un protocole de communication entre entités. Enfin, lorsqu'il s'agit d'algorithmique parallèle, le problème du contrôle devient souvent secondaire par rapport aux problèmes de manipulation optimale des données.

Annexe : "Programmation" en ADA du moniteur.

A.1 Implantation des mécanismes du moniteur en ADA

Pour coller au mieux à l'outil originel, nous introduisons en ADA un module moniteur comportant une structure de donnée et cinq opérateurs :

```
package MONITEUR is
    type condition;
    procedure entree_moniteur;
    procedure sortie_moniteur;
    procedure wait(c: in out condition);
    function empty(c: in condition) return boolean;
    procedure signal(c: in out condition);
end MONITEUR;
```

Une condition est un enregistrement avec deux champs : tâche de type file d'attente de processus (dont le texte est donné au chapitre 3) et un compteur qui donne le nombre de processus en attente sur la condition.

```
type condition is
    record
       cptr: integer:= 0;
       file: file_d_attente;
```

```
    end record;
```

Pour l'implémentation des opérateurs, nous introduisons en ADA une tâche *exclusion_moniteur* qui permet de garantir qu'un seul processus n'est actif à un instant donné : cette tâche assure l'*exclusion mutuelle* pour la manipulation des variables d'état du module et des conditions ainsi que la gestion des blocages et des réveils. Nous avons choisi pour ce faire une discipline standard de gestion pour laquelle un processus qui réveille un autre processus par une primitive signal, se suspend lui-même jusqu'à ce que le processus réveillé ait quitté le moniteur. Ceci pouvant être récursif. Cette tâche, qui gère la discipline d'entrée dans le moniteur, comprend quatre points d'entrée :

```
task type exclusion_moniteur is
    entry entrer;
                        -- lorsqu'un nouveau processus entre dans le moniteur, et devient actif.
    entry interrompre;
                         -- lorsque le processus actif dans le moniteur est interrompu suite a
                          -- un appel a wait (il reste dans le moniteur mais n'est plus actif).
    entry reprendre;
                          -- pour rendre actif un processus devenu inactif suite a un appel a
                         -- signal, des lors qu'il n'y a plus de processus actif dans le moniteur.
    entry sortir;
                                                  -- lorsque le processus actif quitte le moniteur.
end exclusion_moniteur;
```

L'implémentation de cette tâche est la suivante :

```
task body exclusion_moniteur is
begin
    loop
      select                    -- Un processus et un seul est rendu actif dans le moniteur.
        accept reprendre;
          -- La priorite est donnee aux processus qui sont mis en
      else                                          -- attente apres avoir effectue un signal,
        select                          -- en defaveur des nouveaux processus qui desirent
          accept entrer;                                          -- entrer dans le moniteur
        or
          terminate;              -- Il y a alors terminaison si tous les processus clients du
        end select;                             -- moniteur sont en attente de terminaison
      end select;
      select                -- Un processus est actif : l'usage du moniteur est interdit jusqu'a :
```

```
         accept interrompre;                    -- qu'il soit rendu inactif par un appel a wait
      or
         accept sortir;                         -- ou qu'un processus quitte le moniteur
      end select;
   end loop;
end moniteur;
```

Une implémentation possible du module MONITEUR est alors

```
package body MONITEUR is
   controleur_synchro: exclusion_moniteur;
   procedure entree_moniteur is
   begin
      controleur_synchro.entrer;
   end entree_moniteur;
   procedure sortie_moniteur is
   begin
      controleur_synchro.sortir;
   end sortie_moniteur;
   procedure wait(c: in out condition) is
   begin
      c.cptr := c.cptr + 1;
      controleur_synchro.interrompre;
      c.file.endormir;
   end wait;
   function empty(c: in condition) return boolean is
   begin
      return (c.cptr <> 0);
   end empty;
   procedure signal(c: in out condition) is
   begin
      if not (empty(c)) then                    -- un processus est en attente sur
                                                -- la condition : il est reveille, et le
         c.cptr := c.cptr - 1;                  -- signalant mis en attente
         c.file.reveiller;
         controleur_synchro.reprendre;
      endif;
   end signal;
end MONITEUR;
```

A.2 Application au problème des producteurs et consommateurs

Les procédures offertes par le paquetage MONITEUR sont alors accessibles par les tâches *producteur* et *consomateur*, dont l'implémentation est très semblable à celle proposée en pseudo-ADA au paragraphe 3.1.1. Les variables *nombre_de_plein* et *nombre_de_vide* sont partagées par tous les processus, mais l'accès à l'une de ces variables, comme aux conditions associées au moniteur, est garanti en exclusion mutuelle par le moniteur. L'instruction ADA : PRAGMA SHARED (nom de variable) est donc ici inutile. La solution proposée est valide pour plusieurs producteurs et plusieurs consommateurs.

```
procedure producteurs_consommateurs is
-- on inclut ici : la declaration de tableau et les primitives read et write
                                    -- la declaration des types de tache producteur et consommateur
                          -- la declaration des P processus producteurs et des C processus consommateurs
                                                                  -- Module Controleur (ou moniteur)

    with MONITEUR; use MONITEUR;
    nombre_de_plein : integer:= 0 ;
    nombre_de_vide : integer:= N ;
    file_p : condition;
    file_c : condition;

    procedure debut_consommation is
    begin
       entree_moniteur ;
       if (nombre_de_plein = 0) then
          wait(file_c);
       endif;
       nombre_de_plein := nombre_de_plein - 1 ;
       sortie_moniteur ;
    end debut_consommation ;

    procedure fin_consommation is
    begin
       entree_moniteur ;
       nombre_de_vide := nombre_de_vide + 1 ;
       signal(file_p) ;
       sortie_moniteur ;
    end fin_consommation ;

    procedure debut_production is
    begin
       entree_moniteur ;
       if (nombre de vide = 0) then
          wait(file_p);
       endif;
       nombre_de_vide := nombre_de_vide - 1 ;
       sortie_moniteur ;
```

```
    end debut_production ;

    procedure fin_production is
    begin
       entree_moniteur ;
       nombre_de_plein := nombre_de_plein + 1 ;
       signal(file_c);
       sortie_moniteur ;
    end fin_production ;

                        -- Les corps des procedures lire pour les processus consommateurs et
                                -- ecrire pour les processus producteurs sont les memes que
                                                        -- dans le paragraphe precedent.

end producteurs_consommateurs ;
```

Chapitre 7

Mise en oeuvre d'une mémoire virtuellement partagée

Nous poursuivons l'étude entreprise au chapitre précédent sur les moyens de se doter d'une mémoire de type PRAM-CREW sur une machine physiquement parallèle. Ce chapitre traitait des problèmes de concurrence d'accès à des variables (ou zones de mémoire comme le tampon du producteur-consommateur). Les processus étaient en concurrence et on a vu comment la gestion de cette concurrence pouvait se faire par la manipulation de variables d'états. Ces variables de contrôle pouvaient être globales (dans le programme parallèle), locales dans une seule partie de code (on parlait de solution centralisée) ou encore locales à plusieurs parties de codes (on parlait de solution distribuée). Dans les solutions distribuées présentées, les variables d'état étaient réparties dans différents modules, mais non dupliquées. Il existe des formes de distribution du contrôle qui nécessitent la duplication. Tout ceci a trait à des structures de programmes. On ne se préoccupait pas de savoir si la mémoire physique était physiquement globale ou pas. La dernière section de ce chapitre est une ouverture sur les architectures de mémoire (partagée ou distribuée) les plus adaptées à tel ou tel type d'algorithme de contrôle.

Maintenant nous allons nous intéresser aux problèmes spécifiques liés à la mise en oeuvre d'une mémoire PRAM-CREW sur une machine *ne possédant pas* de mémoire physiquement partagée, ce qui est le cas de toutes les architectures qui dépassent quelques dizaines de processeurs. Les processeurs disposent alors chacun d'une mémoire locale et sont reliés entre eux par un *réseau d'interconnexion* qui leur permet de communiquer. On dit que le réseau est totalement connecté lorsque chaque processeur peut communiquer physisquement avec tout autre. Si le réseau ne permet pas de le faire physiquement, un mécanisme logiciel de routage doit y pallier. Rappelons, à ce propos, la typologie en usage :

- les architectures UMA (Uniform Memory Access) sont à mémoire partagée,
- les architectures NUMA (Non Uniform Memory Access), sont à mémoire distribuée et tout processeur peut accéder directement à la mémoire locale d'un autre processeur via le réseau d'interconnexion, les temps d'accès étant non uniformes car dépendants de la distance,

- les architectures NORMA (NO Remote Memory Access), sont à mémoire distribuée et les accès directs aux mémoires distantes sont impossibles et les échanges se font par passage de message toujours via le réseau.

Les architectures de machines parallèles NUMA ou NORMA sont dites à mémoire distribuée.

Dans ce chapitre, les problèmes seront discutés dans le cadre de l'architecture NORMA. Ceci n'est pas une perte de généralité puisqu'il est facile de simuler des échanges de messages sur une architecture NUMA.

Un message informatique est une suite de bits codée comprenant une information (données ou contrôle) et une adresse de destination. Ces messages permettent à deux (ou plus) processeurs d'échanger de l'information, puisque ceux-ci n'ont pas de mémoire commune, donc d'emplacement mémoire partagé pour le faire. On appelle *algorithme distribué* un algorithme de contrôle où les processeurs (ou les processus) s'échangent de l'information par passage de message.
On suppose que le réseau est fiable dans le sens où tout message envoyé est reçu intact. De la même façon que pour l'activité de calcul où nous n'avons pas envisagé les pannes de processeurs, nous n'envisageons pas les pannes de réseau. Ceci est important à signaler car l'une des grandes difficultés de l'algorithmique distribuée est de traiter les pannes [BBK91]. En supposant qu'elles sont inexistantes, nous simplifions grandement le problème. La durée de transmission d'un message est finie (puisque le message est délivré), mais non nécessairement bornée. Dans la pratique, lorsque le temps de transmission d'un message est trop long, des mécanismes spécifiques se mettent en place. Cette durée variable est l'une des spécificités de l'algorithmique distribuée: ainsi lorsque plusieurs processeurs envoient par message des modifications - ou mises à jour - d'une même variable sur une même mémoire, l'ordre dans lequel elles seront effectuées est imprévisible. C'est l'un des indéterminismes dus aux durées de transit dans le réseau.

Dans ce livre où nous utilisons ADA, les envois de messages sont effectués lors des appels d'entrée et des retours de résultat. Ces actions sont atomiques et se font sur rendez-vous : un protocole d'envoi de messages sous-jacent assure ces propriétés lorsque les tâches ne résident pas sur le même site (on utilise le terme site pour processeur, afin de bien insister sur le fait que les processeurs sont physiquement distants), ainsi que le transfert des informations. Mais les durées de ces actions sont inconnues. De plus, deux tâches distinctes peuvent être dans deux cas de figures : soit elles sont implantées sur le même processeur et partagent la même mémoire donc peuvent avoir des variables partagées aisément (donc avec des alogorithmes de contrôle de la concurrence d'accès similaires à ceux du chapitre précédent), soit sont implantées sur des processeurs différents et ne partagent pas la même mémoire physique : elles ne peuvent pas directement partager des variables (i.e. emplacement mémoire). Elles peuvent aisément transmettre des valeurs via des messages.

L'objectif de ce chapitre est de montrer comment les tâches distribuées sur différents processeurs peuvent tout de même "partager des variables". On essaiera bien sûr de trouver des solutions qui préservent le parallélisme et la performance.

Lorsque l'architecture comporte des mémoires locales à chaque processeur, deux schémas ex-

trèmes sont possibles pour les variables qui sont partagées entre les processeurs [1].

- chaque variable est implantée dans une seule mémoire locale et ce processeur agit comme un serveur qui gère les accès à cette variable. Les tâches implantées sur ce processeur font des accès directs à la mémoire et les tâches implantées sur des sites distants envoient par message des requêtes de lecture et récupèrent par message des valeurs, ou envoient des messages de requêtes d'écriture (on dit mise à jour dans ce contexte) contenant la nouvelle valeur.

- une copie de chaque variable est implantée dans toutes les mémoires. Tout processeur qui veut lire une variable accède à sa copie locale. Mais tout processeur qui veut modifier une variable doit, à plus ou moins long terme, modifier l'ensemble des copies de la variable : on appelle cela *maintenir la cohérence de copies mulptiples.*

Dans le cas d'une seule copie, le prix à payer est un goulot d'étranglement au niveau du serveur centralisé. Si la grande majorité des accès sont des lectures, cette solution n'est peut-être pas à privilégier puisqu'on perd tout parallélisme. L'intérêt de la solution centralisée (une seule copie) est sa simplicité. Pour les copies multiples, le prix à payer est la modification de toutes les copies à chaque écriture. De plus, dans le cas de problèmes qui manipulent de gros ensembles de données, cette redondance mémoire peut coûter cher, voire être impossible.

Entre ces deux extrêmes, des solutions hybrides peuvent être trouvées avec des replications partielles sur un sous ensemble de processeurs, ce sous ensemble pouvant varier dynamiquement au cours de l'exécution du programme.

Dans la section suivante nous explicitons les problèmes sémantiques posés par les copies multiples de variable. Ensuite, nous proposons diverses solutions pour la mise en oeuvre d'une PRAM-CREW sur une mémoire distribuée. Ces solutions sont génériques dans le sens où elles mettent en oeuvre des principes généraux d'algorithmique distribuée, principes qui ne font pas intervenir de spécificités architecturales.

7.1 Modèle de cohérence de données

Le chapitre précédent a montré que l'accès à une mémoire partagée pose des problèmes de concurrence d'accès que l'on résout généralement par des techniques d'exclusion mutuelle. Ces problèmes subsistent en environnement distribué. D'autres problèmes se posent liés à la distribution de la mémoire.

1. Dans ce chapitre, on gardera la terminologie processeur ou site pour parler de tous les processus (ou tâches en ADA) qui s'exécutent sur un même processeur en temps partagé et qui partagent la même mémoire locale.

7.1.1 Modèle de cohérence de la PRAM

Les principes d'une PRAM-CREW sont les suivants :

- les écritures se font en exclusion mutuelle (avec toute autre opération- à chaque pas de temps) et suivant un ordre qui peut être connu de tous les processeurs.
- toute lecture lit la dernière valeur écrite et les lectures peuvent se faire en parallèle pendant la même unité de temps.

Le modèle PRAM-CREW fait l'hypothèse d'une échelle de temps commune afin d'ordonnancer globalement les écritures et donner un sens à la notion de "dernière valeur". Les opérations de lecture et d'écriture ont implicitement une *date d'occurence* qui est la date à laquelle les processeurs les émettent. Plusieurs lectures sur la même variable peuvent avoir la même date, mais une écriture ne partage sa date avec aucune autre opération.

Sur une mémoire (réelle) avec un unique accès (même si elle est partagée par plusieurs processeurs), il est relativement facile d'implanter une PRAM-CREW (avec des instructions de lecture et d'écriture atomiques), à ceci près que toutes les lectures en parallèle sur la PRAM sont faites séquentiellement dans un ordre quelconque sur la mémoire réelle. Aussi, dans la réalité on perd l'hypothèse de temps synchrone pour passer à l'hypothèse asynchrone. Or sur une mémoire distribuée, formée de plusieurs modules de mémoire, chacun ayant un seul accès, on ne sait pas comparer directement la date d'une lecture émise par le processeur P1 pour une variable située dans la mémoire locale de P2 avec les dates des écritures émises par P2 pour sa mémoire locale. Il faudra donc donner un sens non trivial à la notion d'ordre sur les écritures et de "dernière valeur écrite".

Remarquons qu'une PRAM-CREW demande l'implantation d'un algorithme de type lecteurs-rédacteurs pour chaque variable (lectures en parallèle et écritures en exclusion mutuelle) assortie d'une règle de priorité : les lectures et écritures ont implicitement une date utilisée comme suit:

- les écritures se font séquentiellement dans l'ordre de leur datation,
- toute lecture accède à la valeur produite par l'écriture immédiatement antérieure.

Ces principes définissent ce qu'on appelle un *modèle de cohérence* de la mémoire. La PRAM-CREW nous définit un modèle de cohérence particulier, mais il en existe d'autres [Mos93] plus ou moins contraignants et utilisables dans des contextes différents.

7.1.2 Datation dans une machine parallèle

Tous les processeurs possèdent de façon standard une horloge (un compteur modulo) et les systèmes d'exploitation permettent de lire ces valeurs avec plus ou moins de précision. Si les processeurs sont identiques, les horloges sont semblables mais le système physique qui est utilisé est

sensible à plusieurs paramètres (par exemple la température) et supporte une imprécision à la fabrication. On observe donc des *dérives* entre horloges sur des processeurs distincts. Chaque processeur est capable de dater un événement relativement à son horloge locale, mais les dates obtenues sur différents processeurs ne sont pas comparables. Ceci est vrai surtout lorsque les événements sont proches dans le temps, qui est le cas qui nous intéresse ici.

Un autre aspect de la question est relatif à l'observation de l'état global de la machine. Dans une machine parallèle où les processeurs sont reliés via un réseau qui impose un délai, chaque processeur ne peut connaître les actions du processeur distant que via le transfert de messages de délai inconnu. Dans le contexte de la mémoire, si P2 veut lire une valeur, il lui est impossible de savoir si P1 a émis une écriture de date inférieure (en supposant que l'on a résolu le problème de la datation). Cette information lui parviendra, mais avec un délai inconnu. Jusqu'à quand P1 devra-t-il attendre avant d'effectuer sa lecture?

Pour implanter cette PRAM, il faudra donc résoudre le problème *l'absence d'ordonnancement temporel global* et *d'observation de l'état global* des processeurs.

Il existe des algorithmes qui permettent de construire une horloge globale entre plusieurs processeurs. Cette horloge fournit une date à chaque événement quelque soit son site d'occurence. Elle permet de comparer les dates d'événements qui sont apparus sur des processeurs distants et éventuellement de quantifier des délais de transfert de messages. Ces horloges peuvent être *logiques* [Ray92], auquel cas elles définissent un ordre total entre les événements des divers processeurs, sans relation avec le temps physique, ou *physique* [MTr95] auquel cas elles attribuent à chaque événement une date qui définit un ordre global et dont les écarts traduisent les délais temporels.

Dans ce chapitre, nous verrons comment utiliser ces horloges ou bien comment construire un mécanisme implicite de datation en utilisant par exemple l'ordre des arrivées d'appel à une tâche.

Dans la suite, nous allons donner différentes solutions pour implanter une PRAM-CREW sur une mémoire distribée, de complexité croissante, en discutant la façon dont elles implantent le modèle de cohérence de la PRAM-CREW et leur points forts et faibles en terme de performance.

7.2 Solution centralisée

7.2.1 Principe

Dans la solution centralisée, la variable partagée réside dans une seule mémoire. Les accès à cette variable sont gérés par un serveur. Les exigences de cohérence du modèle PRAM-CREW sont satisfaites par des contraintes d'accès de type lecteur-rédacteur (lectures en parallèle et écritures en exclusion mutuelle) sur chaque variable. L'ordre temporel utilisé pour la discipline d'accès est l'ordre *d'arrivée* des messages à ce serveur et non pas l'ordre d'émission. Dans le modèle PRAM théorique, ces deux ordres (ordre d'émission par les processeurs et ordre d'exécution sur la mémoire) sont identiques. Dans une machine réelle NORMA, l'ordre d'émission des messages n'est pas connu simplement, alors que l'ordre de réception sur un site (ou processeur) peut-être connu simplement par le serveur destinataire. Ainsi, la notion de "dernière valeur écrite" prend un sens relativement

à cet ordre de réception.

7.2.2 Programmation en ADA

La programmation en ADA de ce serveur donnée dans ce chapitre est inspirée de versions du lecteur-rédacteur extensivement traitées ailleurs dans ce livre: une version naive au chapitre 2.3.2 et des versions plus élaborées au chapitre 9.

Dans ce programme ADA, la donnée partagée est encapsulée dans un paquetage "memoire_globale_solution_centr
Les accès à ce paquetage se font via deux procédures "lire" et "écrire" appelables en parallèles. Les tâches utilisatrices - on les appellera tâches clientes- ne figurent pas dans le programme, elles font des appels à ces procédures pour accéder à la donnée partagée.

Le contrôle d'accès à la donnée partagée est fait par deux tâches "acces_serveur" et "serveur". La tâche "acces_serveur" n'a qu'une seule entrée afin que l'ordre (FIFO par rapport à leur date d'arrivée) des acceptations à cette entrée définisse l'ordonnancement des accès à la variable (écritures et accès à la dernière valeur écrite). En dehors de ce séquencement, la tâche "acces_serveur" ne fait rien d'autre que de transmettre les requêtes à la tâche "serveur". On a donc ici un mécanisme implicite de datation qui se réduit à l'ordonnancement des appels en ADA.

La tâche "serveur" implante un protocole de lecteurs-rédacteurs, c'est à dire autorise des accès parallèles en lecture et séquentiels en écriture. En effet:

- si les variables NR et NL sont nulles et si la tâche "accès_serveur" envoit une suite de requêtes de lectures, celles ci seront acceptées les unes après les autres. Chaque acceptation de "autoriser_lecture" a pour effet de débloquer la tâche "accès_serveur" qui elle-même débloque son client sur l'entrée "requete". Ce client peut alors effectuer son opération sur la donnée partagée. La tâche "accès-serveur" peut alors accepter la requête suivante de lecture. Le parallélisme des lectures se trouve au niveau de l'instruction "v := variable_partagée" qui peut s'exécuter en parallèle, surtout si la donnée partagée est de grosse taille et que sa durée de lecture est longue par rapport aux durées des acceptations "requete" et "autoriser_lecture".
- Dans le même temps, la tâche "serveur" accepte en alternance dans la même alternative des appels "fin_lecture".
- Dès que la tâche "accès_serveur" accepte et transmet à "serveur" une requête d'écriture, cette requête ne sera acceptée par la tâche "serveur" que lorsque toutes les fins de lectures auront mis la varible d'état NL à zero. Toutes les autres requêtes sont à ce moment en attente dans la file de la tâche "accès_serveur". Maintenant, le seul scénario possible est que le client de la requête d'écriture, enfin débloqué, exécute son écriture et fasse un appel "fin_écriture". La variable d'état NR sera remise à zero et la tâche "serveur" pourra accepter une nouvelle entrée, soit "autoriser_lecture", soit "autoriser_écriture".

---

package memoire_globale_solution_centralisee *is*

```
   type contenu_variable is ...                          -- primitives de gestion de la variable
   procedure lire (v : out contenu_variable);
   procedure ecrire (v : in contenu_variable);
end memoire_globale_solution_centralisee ;

package body memoire_globale_solution_centralisee is

   type mode_acces is (lecture, ecriture) ;
   variable_partagee : contenu_variable := valeur_initiale ;

   task acces_serveur is
      entry requete (mode : in mode_acces) ;
   end acces_serveur ;

   task serveur is
      entry autoriser_lecture ;
      entry fin_lecture ;
      entry autoriser_ecriture ;
      entry fin_ecriture ;
   end serveur ;

   procedure lire (v : out contenu_variable) is
      info : contenu_variable ;
   begin
      acces_serveur.requete(lecture) ;
      v := variable_partagee ;
      serveur.fin_lecture ;
   end lire ;

   procedure ecrire (v : in contenu_variable) is
      info : contenu_variable ;
   begin
      acces_serveur.requete(ecriture) ;
      variable_partagee := v ;
      serveur.fin_ecriture ;
   end ecrire ;

   task body acces_serveur is
   begin
      loop
         accept requete (mode : in mode_acces) do
            case mode is
               when lecture => serveur.autoriser_lecture ;
               when ecriture => serveur.autoriser_ecriture ;
            end case;
         end requete ;
      end loop;
   end acces_serveur ;

   task body serveur is
      NR, NL: integer := 0 ;                             -- les variables d'etat gerees par le serveur
   begin
      loop
```

```
        select
          when (NR = 0) => accept autoriser_lecture ;
          NL := NL + 1 ;
        or
          accept fin_lecture ;
          NL := NL - 1 ;
        or
          when (NR + NL = 0) => accept autoriser_ecriture ;
          NR := 1 ;
        or
          accept fin_ecriture ;
          NR := 0 ;
        end select;
      end loop;
    end serveur ;

end memoire_globale_solution_centralisee ;
```

Comme nous l'avons déjà souligné, cette solution n'est acceptable, en terme de performances, que si le traffic des requêtes pour cette variable est faible, sinon le serveur centralisé est un goulot d'étranglement pour le programme tout entier. Par contre, cette solution, par sa simplicité, est préconisée pour toute variable partagée peu utilisée et critique.

Les sections suivantes traitent de copies multiples.

7.3 Exclusion mutuelle distribuée : le jeton

7.3.1 Principe

On suppose que la variable partagée est dupliquée sur P processeurs de la machine (pas forcément tous, mais en nombre fixe). La solution centralisée met en place une exclusion mutuelle pour l'accès à une copie unique de variable. Des copies multiples peuvent également être maintenues cohérentes grâce à une exclusion mutuelle.

Une façon simple de programmer une exclusion mutuelle en environnement distribué est d'utiliser un message de contrôle spécifique appelé *jeton*. Plus précisément, les processeurs sont numérotés de 0 à $P-1$, ce qui les organise en *anneau logique* de communication. On appelle anneau logique tout schéma de communication où chaque processeur ne peut recevoir de message que de son prédécesseur et ne peut envoyer de message qu'à son successeur. Le prédécesseur du processeur i est le processeur $(i-1) \bmod P$ et son successeur le processeur $(i+1) \bmod P$. Ce schéma de communication est possible sur tout réseau de communication totalement connecté. Initialement, un processeur particulier- par exemple le processeur 0- a seul le droit d'émettre le jeton. Selon la règle de l'anneau logique, il l'émet vers le processeur 1, qui l'émet vers 2, etc. Un processeur ayant reçu le jeton et ne l'ayant pas encore retransmis est le seul dans cet état. Cet état est donc un état d'exclusion mutuelle.

Nous donnons ci-dessous l'algorithme du jeton en ADA. Le "message" qui circule est matérialisé par les appels successifs aux entrées "jeton" : Le programme principal appelle l'entrée jeton du site 0, qui après avoir exécuté sa section critique en exclusion mutuelle appelle l'entrée jeton du site 1, et ainsi de suite. On a ainsi les images successives d'un message jeton qui circule de site en site.

```
procedure jeton is

   task type serveur_jeton is
      entry TonSite(tid: in integer );
      entry jeton;
   end serveur_jeton;

   task body serveur_jeton is
      mon_site: integer;
      mon_voisin: integer;
   begin
      accept TonSite(tid: in integer ) do
         mon_site := tid;                          -- Idendité du serveur associé
      end TonSite;
      mon_voisin := (mon_site + 1 ) mod P;         -- vers qui le jeton sera transmis


      loop
         accept jeton;                             -- on reçoit le jeton
         ....                                      -- instructions exécutées en section critique
         les_serveur_jeton(mon_voisin).jeton;      -- on passe le jeton au voisin
         ...
      end loop;

   end serveur_jeton;

   les_serveurs_jeton: array (0.. P-1) of serveur_jeton;

begin
   for i in 0.. P-1 loop                           -- identification des sites
      les_serveurs_jeton.TonSite(i);
   end loop;

   les_serveurs_jeton(0).jeton;                    -- départ du jeton
end jeton;
```

L'algorithme distribué qui gère la cohérence de données dupliquées en utilisant le jeton est basé sur les idées suivantes :

- utiliser localement sur chaque site la version centralisée pour chaque copie de variable afin d'imposer l'ordre sur les requêtes locales et la discipline d'accès des lecteurs-rédacteurs.
- l'exclusion mutuelle du jeton est utilisée pour effectuer les écritures. Lorsqu'un site veut effectuer une écriture, il attend le jeton. Si le jeton n'est pas porteur d'une écriture lui-même,

le site réalise l'écriture sur la mémoire locale (après la fin de toutes les lectures locales) et transmet le jeton à son voisin. Ce jeton contient l'écriture à reporter. Le site voisin réalise l'écriture du jeton (laissant en attente pour un prochain passage de jeton ses écritures locales) et ainsi de suite, jusqu'à ce que le jeton ait effectué un tour complet. Le jeton n'est à ce moment là plus porteur d'écriture. Un site n'a le droit d'émettre une nouvelle écriture que lorsque la dernière écriture (émise par un site quelconque et transportée par le jeton) est connue de tout les processeurs de l'anneau.

- Les lectures sont effectuées localement en exclusion mutuelle avec les écritures. Les lectures sur les différents sites peuvent se passer en parallèle (en plus du parallélisme possible localement, comme auparavant). La dernière valeur écrite est celle qui a été écrite lors du dernier passage de jeton.

Dans cet algorithme, l'ordre des écritures et la notion de "dernière valeur écrite" est donné par une combinaison du séquencement provoqué par la circulation du jeton et de la politique locale sur chaque site réalisée par l'algorithme centralisé.

7.3.2 La programmation en ADA

Le programme ADA est construit comme suit : le paquetage encapsule les P copies de la variable partagée. Tout appel aux procédures de lecture et d'écriture doit préciser le numéro de la copie concernée en paramètre qui est aussi le numéro du site. Sur chaque site, on a trois tâches : "acces_serveur", "serveur", "serveur_jeton".

La tâche "acces_serveur" a le même rôle et le même code que précédement. Elle ordonne les requêtes de lecture et d'écriture locales sur chaque site.

La tâche "serveur" met en oeuvre la politique des lecteurs-rédacteurs pour la copie locale de la variable partagée. Par rapport à la solution centralisée, on note les modifications suivantes : cette tâche accepte en priorité les appels du "serveur_jeton" de son site (grâce à la construction "select ...else").

- Si l'appel est "mise_a_jour", le jeton est porteur de l'écriture à reporter localement. Les lectures en cours doivent alors se terminer, le "serveur-jeton" est débloqué, effectue l'écriture et signifie la fin de cette écriture à la tâche "serveur" par l'appel de "fin_mise_a_jour". Tout ceci s'effectue séquentiellement.

- Si l'appel du jeton est "ecriture_possible", le jeton n'est pas porteur de nouvelle valeur et une écriture locale est possible. Pour que cette écriture soit effectivement réalisée, il faut, selon la discipline locale, que ce soit la première requête en attente sur un appel de la tâche "accès_serveur". Si c'est le cas, toutes les lectures en cours sont terminées avant d'autoriser l'écriture. La tâche "serveur_jeton" sait, grâce à un booléen, si une écriture a été possible.

- Les appels non prioritaires sont les appels de lecture et sont traités comme auparavant entre les passages du jeton.

- La variable d'état NR a été supprimée puisque les débuts et fins d'écriture sont maintenant inscrits séquentiellement dans le code[2].

La tâche "serveur_jeton" agit après chaque réception de jeton, c'est à dire lors de l'activation de son entrée "jeton". Le jeton transporte deux données : la dernière valeur écrite et combien de sites ont reporté cette mise à jour sur leur copie locale, dans la variable nommée "tour" :

- si "tour" vaut P, alors la tâche "serveur_jeton" autorise son "serveur" local à effectuer une écriture par appel de l'entrée "autoriser_ecriture". Si le booléen paramètre revient avec la valeur vrai, une écriture locale est lancée, et le "serveur_jeton" attend que le client lui transmette la valeur qu'il insère dans le jeton avec un nombre de sites visités à zéro. Si le booléen paramètre revient avec la valeur FAUX, le jeton est retransmis tel quel.
- si "tour" vaut P-1, le jeton est revenu au site qui a été à l'origine de la dernière mise à jour. Il renvoie le jeton au suivant en assignant à "tour" la valeur P : il permet ainsi de donner priorité aux écritures à venir sur les autres processeurs afin d'éviter la famine.
- si "tour" est strictement inférieur à P-1, il faut répercuter la mise à jour du jeton localement et retransmettre le jeton en incrémentant le nombre de sites visités.

Le corps du paquetage sert aux déclaration des tableaux de tâches et à l'initialisation : Des appels font connaitre à chacune des tâches leur numéro de site. Un appel à la tâche de numéro 0 permet d'initialiser le premier tour du jeton.

```
package memoire_globale_solution_jeton is
    type contenu_variable is ...                          -- primitives de gestion de la variable
    procedure lire (v : out contenu_variable, site : in integer );
    procedure ecrire (v : in contenu_variable, site : in integer );
end memoire_globale_solution_jeton ;

package body memoire_globale_solution_jeton is

    type mode_acces is (lecture, ecriture) ;
    variable_partagee : array (0..P-1) of contenu_variable := valeur_initiale ;
                                          -- cette valeur initiale est identique sur tous les sites

    task type serveur_jeton is
       entry TonSite(site : in integer );
       entry jeton (valeur : in contenu_variable , compteur : in integer ) ;
       entry a_transmettre (nouvelle_valeur : in contenu_variable) ;
    end serveur_jeton ;

    task type acces_serveur is
       entry TonSite(site : in integer );
       entry requete (mode : in mode_acces) ;
    end acces_serveur ;
```

2. Lors de l'exécution, ils se font toujours séquentiellement, mais dans le code précédent, cette séquentialité était cachée par un "select" et des gardes.

```
task type serveur is
   entry autoriser_lecture ;
   entry fin_lecture ;
   entry autoriser_ecriture ;
   entry fin_ecriture ;
   entry mise_a_jour ;
   entry fin_mise_a_jour ;
   entry ecriture_possible ( ecriture_faite : out boolean );
end serveur ;


les_acces_serveur : array (0..P-1) of acces_serveur ;
les_serveur : array (0..P-1) of serveur ;
les_serveur_jeton : array (0..P-1) of serveur_jeton ;


procedure lire (v : out contenu_variable , mon_site : in integer ) is
   info : contenu_variable ;
begin
   les_acces_serveur(mon_site).requete(lecture) ;
   v := variable_partagee(mon_site) ;
   les_serveur(mon_site).fin_lecture ;
end lire ;

procedure ecrire (v : in contenu_variable , mon_site : in integer ) is
   info : contenu_variable ;
begin
   les_acces_serveur(mon_site).requete(ecriture) ;
   variable_partagee (mon_site) := v ;
   les_serveur_jeton (mon_site).a_transmettre (v);
   les_serveur(mon_site).fin_ecriture ;
end ecrire ;

task body acces_serveur is
begin
   accept TonSite (mon_site : in integer ) ;
   loop
      accept requete (mode : in mode_acces) do
         case mode is
            when lecture => les_serveur(mon_site).autoriser_lecture ;
            when ecriture => les_serveur(mon_site).autoriser_ecriture ;
         end case;
      end requete ;
   end loop;
end acces_serveur ;

begin

task body serveur_jeton;
   mon_site : integer;
   site_voisin : integer;
   ecriture_faite : boolean;                    -- vrai si mise a jour faite lors du passage du jeton
```

```
   val_jeton : contenu_variable ;                          -- valeur contenue dans le jeton
   tour : integer;              -- nombre de sites visites par le jeton depuis la derniere ecriture

begin
   accept TonSite(tid : in integer ) do
      mon_site := tid ;                                    -- Idendité du serveur associé
   end TonSite ;
   site_voisin := (mon_site + 1 ) mod P ;                  -- vers qui le jeton sera transmis

   loop
      accept jeton (valeur_courante : in contenu_variable, compteur : in integer ) do
         val_jeton := valeur_courante ;
         tour := compteur ;
      end jeton ;

      case tour is
         when P =>                -- Tous les sites sont a jour et on peut autoriser une ecriture
            les_serveur (mon_site).ecriture_possible( ecriture_faite ) ;
            if ecriture_faite then                         -- une ecriture a ete faite localement
               accept a_transmettre (nouvelle_valeur : in contenu_variable) do
                  val_jeton := nouvelle_valeur ;           -- recuperation de la nouvelle valeur
               end a_transmettre ;
               tour := 0 ;                                 -- aucun autre site n'est a jour
            end if;

         when P-1 =>              -- Je suis celui qui a realise la derniere ecriture et elle a ete
                                  -- repercutee partout: je passe la main.
            tour := P ;

         when others =>           -- Ecriture locale de la valeur ecrite dans le jeton.
            les_serveur (mon_site).mise_a_jour ;
            variable_partagee(mon_site) := val_jeton ;
            les_serveur(mon_site).fin_mise_a_jour
            tour := tour + 1 ;
      end case;

      les_serveur_jeton(site_voisin).jeton (val_jeton , tour ) ;
   end loop;
end serveur_jeton ;

task body serveur is
   NL: integer:= 0 ;                                       -- la variable d'etat geree par le serveur
begin
   loop
      select
         accept mise_a_jour do
            while NL > 0 loop
               accept fin_lecture;
               NL := NL -1 ;
            end loop;
         end mise_a_jour ;
         accept fin_mise_a_jour ;
      or
```

```
            accept ecriture_possible ( ecriture_faite : out boolean ) do
               select
                  accept autoriser_ecriture do
                     while NL > 0 loop
                        accept fin_lecture;
                        NL := NL -1 ;
                     end loop;
                  end autoriser_ecriture;
                  ecriture_faite := TRUE
               else
                  ecriture_faite := FALSE ;
               end select;
            end ecriture_possible ;
            accept fin_ecriture ;
         else
            select
               accept autoriser_lecture ;
               NL := NL + 1 ;
            or
               accept fin_lecture ;
               NL := NL - 1 ;
            or
            end select;
      end loop;
      end serveur ;


begin
      for i in 0.. P-1 loop
         les_acces_serveur(i).TonSite(i) ;
      end loop;
      for i in 0.. P-1 loop
         les_serveur_jeton (i).TonSite(i) ;
      end loop;
      les_serveur_jeton(0).jeton (valeur_initiale , P-1 ) ;                    -- depart du jeton

end memoire_globale_solution_jeton ;
```

Dans cet algorithme, l'ordre global est construit comme suit :

- les requêtes d'écriture sont séquencées par le jeton: le premier site, soit P_j qui a une écriture en tête de file de son "serveur_accès", l'effectue, puis le jeton fait un tour pour la répercuter. Le processus recommence à partir de P_{j+1} à rechercher la première écriture en tête de file.

- le jeton insère donc dans les sites locaux des écritures prioritaires qui modifient, pour les requêtes de lecture, la dernière valeur écrite. Dans ce cas, c'est la date d'arrivée du jeton, par rapport à la date de prise en compte des lectures par la tâche "serveur", qui est déterminante. On remarque que les lectures postérieures à une requête d'écriture locale sont en attente si l'écriture est elle-même en attente dans la file des entrées à "acces_serveur".

Cet ordre doit être manipulé avec précaution lorsque le programmeur a en tête un fonctionnement synchrone de type PRAM théorique de son programme. Cet algorithme trouve une utilisation raisonnable dans le cas d'un faible traffic global d'écriture puisque les lectures pourront s'effectuer en parallèle. Sinon, le temps de circulation du jeton est très pénalisant : dans cet algorithme les écritures sont faites et répercutées sur toutes les copies séquentiellement. Celui-ci est particulièrement pénalisant lorsqu'un seul processeur manipule la variable partagée, mais qu'il doit faire circuler tout de même le jeton afin de répercuter cette mise à jour. Cette solution est intéressante lorsque le traffic des requêtes est équilibré entre les clients et que les variables partagées manipulées sont peu modifiées.

7.4 Diffusion atomique

On suppose toujours que la variable partagée est dupliquée sur P processeurs de la machine. L'idée dans cette section est de se rapprocher du fonctionnement de la PRAM synchrone de telle sorte que les lectures se fassent en parallèle sur les mémoires locales et que toutes les écritures soient répercutées sur tous les processeurs dans le même ordre. Un moyen de faire ces écritures est d'utiliser un mécanisme de diffusion *atomique et uniforme*. Etant donné un groupe de processeurs, une diffusion est une opération de communication où l'un des processeurs envoie une même valeur au reste du groupe de processeurs. Cette diffusion est uniforme s'il existe un ordre total sur ces diffusions connu de tous les processeurs. Cet ordre servira d'ordre sur les écritures. Il existe de nombreux algorithmes de diffusion dans la littérature [BBK91]. Nous en proposons un basé sur la construction d'horloges logiques.

7.4.1 Les horloges de Lamport

Nous introduisons ici les horloges logiques de Lamport [Lam78]. Chaque processeur P_i dispose d'un entier h_i, son horloge, initialisée à 0. Le doublet (valeur de l'horloge, numéro de site) est appelée estampille. Cette horloge logique évolue de la façon suivante :

- chaque fois que le processeur émet une lecture ou une écriture, il incrémente son horloge de 1, et l'opération de lecture ou d'écriture porte l'estampille correspondante,
- lorsque le processeur envoie un message, celui-ci véhicule la valeur courante de son horloge et le numéro de site émetteur, qui devient alors l'estampille du message,
- lorsque le processeur recoit un message, il met à jour son horloge locale au maximum de la valeur courante de son horloge locale et de l'horloge de l'estampille du message puis lui ajoute 1.

$$h_i := max(h_i, horloge\_message) + 1 \tag{7.1}$$

L'événement de réception du message porte alors cette nouvelle estampille.

Avec cette horloge, les événements sont ordonnés suivant une certaine logique : les opérations locales ordonnées suivant l'ordre temporel et physique. Tout message a une date de réception qui

porte une valeur d'horloge supérieure à celle de tous les événements locaux au site de réception et antérieurs, et aussi supérieure à la valeur d'horloge d'émission.

Si deux événements sur deux sites distincts portent la même valeur d'horloge, on décide que celui qui a le numéro de site le plus grand porte l'estampille la plus grande. Cette façon d'arbitrer a peu d'importance et ne sert pas dans notre algorithme, car les événements dont nous comparons les estampilles sont, soit tous sur le même site, soit causalement dépandants- c'est à dire qu'il y a eu des échanges de messages entre sites pour ces événements, ce qui rend l'égalité des estampilles impossible.

7.4.2 La diffusion atomique

Le protocole de diffusion atomique est utilisé dans le cadre de la diffusion d'une requête d'écriture. Son principe est le suivant [BBK91]:

- dans une première phase, l'émetteur de l'écriture envoie un message signifiant une intention d'écriture à chacun des sites destinataires et attend de chacun d'eux un message d'accusé de réception. Ces messages sont bien sûr porteurs d'estampilles de Lamport,
- lorsqu'il a recu tous les accusés de réception, l'émetteur prend la plus grande des estampilles des accusés de réception, notée h_{max}, et envoie à tous les processeurs (y compris lui même) l'ordre d'écriture muni de cette estampille h_{max}.

Cet algorithme appartient à la classe des algorithmes de consensus (ici consensus sur l'estampille de l'écriture) *à trois phases*: la phase de diffusion des intentions d'écriture, la phase d'envoi des accusés de réception, et la phase d'envoi de l'ordre d'écriture. Il faut $3 * (P - 1)$ messages pour parvenir à un accord. Beaucoup d'autres protocoles sont basés sur ce principe.

7.4.3 La gestion des copies multiples

Avec ces estampilles et le protocole de diffusion, la gestion des copies multiples devient :

- les lectures recoivent une estampille au moment de leur apparition comme événement local,
- les écritures recoivent une estampille (la même pour tous les sites) donnée par le protocole de diffusion (voir ci-dessous)
- les lectures-écritures se font effectivement sur les mémoires dans l'ordre des estampilles en conservant le parallélisme possible des lectures.

Il reste un problème à élucider. Chaque processeur traite les demandes de lecture-écriture dans l'ordre des estampilles. Par exemple, à un instant donné, ces requêtes portent les horloges 12, 20, 45 sur un processeur P_i. Ces estampilles sont des entiers non consécutifs. Avant de traiter la requête

munie de l'horloge 12, le processeur doit être assuré qu'aucune requête avec une estampille inférieure ne va lui parvenir dans le futur, cette requête ne pouvant être qu'une écriture venant d'un autre site (puisque les lectures sont effectuées localement et les opérations locales- écritures locales comprises- sont connues sans délais). Pour cela, chaque site conserve dans un tableau l'horloge du dernier message reçu de chaque autre processeur. On note h_{min} le minimum courant de ce tableau. Les requêtes à traiter ne sont effectivement répercutées sur la mémoire que lorsque leur estampille est inférieure à h_{min}. On est ainsi assuré que les inversions d'ordre ne sont pas possibles.

Ce protocole utilise un mécanisme d'estampille et retarde les requêtes jusqu'à être sûr que l'ordre sera bien respecté partout. Du point de vue des performances, ce retard peut être très pénalisant si le calcul est très synchrone, i.e. que chaque processeur doit attendre que les écritures soient effectivement réalisées avant de continuer le calcul. Il est aussi très coûteux lorsque le traffic de requêtes en écriture est fort. En effet chaque écriture, avec toutes ses recopies coûte cher en nombre de messages. Le rendement de cette solution est bon si le traffic de requêtes en écriture est faible. Si le traffic des requêtes sur la variable partagée est déséquilibré, cet algorithme est inefficace dans le sens ou beaucoup de recopies sont faites sur des versions de variable qui ne seront pas utilisées. Par exemple, on peut modifier n fois la version de P_i sans que P_i ne visite cette variable en lecture. Il aurait été suffisant de n'effectuer que le n-ième changement. Si le réseau impose des délais de transmission réguliers entre les divers processeurs, l'ordre d'exécution des requêtes doit s'approcher de leur ordre d'émission.

Les réseaux de machines parallèles peuvent offrir du matériel permettant d'effectuer les diffusions qui peuvent ainsi devenir peu coûteuses. Ce dispositif matériel peut aussi par la même occasion garantir un ordre total sur toutes les diffusions, quelque soit leur émetteur. Si c'est le cas, la mise à jour des copies multiples peut se faire en une phase avec une simple diffusion portant l'estampille de l'émetteur.

7.5 Mécanisme d'invalidation

L'intérêt des copies multiples de variables est de préserver le parallélisme d'un programme en autorisant l'accès parallèle en lecture aux données sur des mémoires distinctes. Le prix à payer est le maintien de la cohérence des copies multiples lors d'une écriture. Dans cette section nous proposons une solution qui tente de limiter ce prix au strict nécessaire [NLo91]. On suppose que l'on dispose d'un mécanisme logiciel ou matériel pour faire des diffusions atomiques et uniformes.

Dans cette solution chaque copie d'une variable partagée a un état :

- elle est *valide* si la valeur est à jour ou *invalide* sinon,
- elle est *libre* si cette copie est l'unique copie *valide* et *liée* sinon (i.e. soit *invalide*, soit *valide* mais non unique).

Le protocole fonctionne comme suit:

- lors de l'occurrence d'une lecture locale sur P_i, si la copie locale est *valide*, cette valeur est

retournée imédiatement. Si la copie locale est *invalide*, une demande de lecture est envoyé à un processeur qui possède une copie *valide*, soit P_j. Cette valeur valide est alors recopiée localement et la copie locale prend l'état *valide*. Cette copie restera *valide* jusqu'à recevoir un ordre d'invalidation. Si la copie de P_j est *libre* elle passe dans l'état *liée*.

- lors de l'occurrence d'une écriture sur P_i, si la copie locale est *libre*, cette écriture est effectuée directement sur la copie locale. Si la donnée est *liée*, un ordre d'invalidation est diffusé à tous les processeurs accompagné de l'identité P_i du processeur possédant la nouvelle valeur *valide*, puis l'écriture s'effectue sur la copie locale qui passe dans l'état *valide* et *libre*.

Cette solution a un bon fonctionnement, même lorsque le traffic de requête est déséquilibré et qu'un seul processeur manipule la donnée. Dans ce cas, elle est *libre* et toutes les opérations de lecture-écriture se passent localement. Comme nous l'avions annoncé, la gestion des copies multiples est dite "paresseuse" dans le sens où on les modifications ne sont répercutées sur les diverses copies qu'au moment où elles sont nécesaires[3]. Le coût supplémentaire est un coût mémoire pour stocker l'état des diverses données.

Concernant le modèle de cohérence de la PRAM-CREW, les écritures sont ordonnancées globalement par le mécanisme de diffusion atomique et uniforme. Les lectures accèdent bien la dernière valeur écrite, i.e celle reçue après le dernier ordre d'invalidation diffusé.

Du point de vue des performances, cette solution semble préférable lorsque les accès aux variables présentent des propriétés de localité spatiale des références à la mémoire. On dit que les accès à une variable possèdent une localité temporelle, comme dans les programmes séquentiels, lorsque le programme accède les mêmes variables pendant un laps de temps avant de se déplacer sur une autre zone de données. Cette propriété est la base de l'utilisation des mécanismes de pagination. Elle n'est pas un cas très favorable en calcul parallèle, sauf si une seule tâche (ou seulement les tâches d'un site) accède à cette variable dans le programme parallèle pendant ce laps de temps. On dit que les accès ont une localité spatiale lorsque cette localité temporelle se déplace de tâche en tâche, ou bien lorsque chaque tâche travaille sur sa partie de donnée (le parallélisme de donnée) sans interaction avec les autres. La localité spatiale est favorable au calcul parallèle.

Ce type de solution est inspiré de mécanismes qui existent pour les caches [Ste90] de mémoire distribuée qui sont implantés par des dispositifs matériels. Même pour la gestion de mémoire, un mécanisme de diffusion matériel efficace semble indispensable pour atteindre des performances raisonnables [SZh90].

3. Dans la terminologie anglo-saxonne, on appelle "write-through" ou bien "write-update" les protocoles de maintien de la cohérence de copies multiples qui mettent à jour immédiatement toute les copies par diffusion. On appelle "write-back" ou "write-invalidate" les protocoles qui ne mettent à jour les copies que lorsque cela est nécessaire.

7.6 Conclusion

Dans ce chapitre, nous avons décrit 4 solutions pour gérer des données partagées sur une mémoire distribuée : une solution centralisée, une solution basée sur une exclusion mutuelle distribuée, une solution basée sur les horloges de Lamport et la diffusion et enfin une solution basée sur des mécanismes d'invalidation. Ces solutions utilisent des principes généraux de l'algorithmique distribuée : serveur centralisé, jeton, diffusion, etc. Pour le contrôle de la gestion de la mémoire, ces algorithmes couvrent la quasi-totalité des solutions possibles.

Cepandant, le contrôle de la gestion de la mémoire n'est pas le seul problème à résoudre. Un autre problème est celui de la *granularité*. Dans tout le chapitre, on a parlé de "variable partagée" : celle-ci peut être soit une entité connue du programme comme un booléen, un enregistrement, ou tout un tableau, soit une unité de gestion mémoire comme la page ou le segment. La taille et le type des variables partagées sont cruciaux pour les performances de l'algorithme. Par exemple, si la variable est de taille trop petite, le risque est de faire fonctionner l'algorithme de maintien de la cohérence trop souvent. Si on groupe ces variables, l'algorithme fonctionne 1 fois pour le groupe; mais si le groupe est trop gros, les recopies sont trop nombreuses pour un pourcentage d'utilisation faible des données du groupe. Un compromis difficile est à trouver en fonction du type d'application et des performances de la machine parallèle.
La granularité peut être définie par des découpages logiques (un objet, un tableau, une zone de tableau, etc.) ou à partir d'entités physiques comme la page. La première option permet de mieux s'adapter au programme et se gère au niveau de l'environnement de programmation. La deuxième option permet une automatisation plus simple qui peut se situer au niveau du système d'exploitation.

On peut aussi penser à utiliser différents algorithmes de maintien de la cohérence pour divers types de variable. La solution qui semble émerger actuellement [NLo91] est de réserver ces algorithmes de maintien de la cohérence à un sous ensemble de variables critiques (variables de synchronisation par exemple) afin d'en limiter le coût.

Bibliographie

[Ray92] M. Raynal, "Synchronisation et état global dans les systèmes répartis", Ed Eyrolles, 1992.

[MTr95] E. Maillet, C. Tron, "On efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems", Journal of Parallel and Distributed Computing, vol 28, pp 84-93, 1995.

[BBK91] R. Balter, J.P. Banâtre, S. Krakowiak (eds), "Construction des systèmes d'exploitation répartis", Collection Didactique INRIA, 1991.

[NLo91] B. Nitzberg, V. Lo, "Distributed Shared Memory : a survey of Issues and Algorithms", IEEE Computer , 1991.

[Mos93] D. Mosberger, "Memory Consistency Models", ACM Operating System Review, vol 27, nb 1, 1991.

[**Lam78** L. Lamport, "Time, Clocks, and the ordering of Events in a Distributed System", Communication of the ACM, vol 21, nb 7, 1978.

[**SZh90**] M. Stumm, S. Zhou, "Algorithms Implementing Distributed Shared Memory", IEEE Computer, 1990.

[**Ste90**] P. Stenstrom, "A survey of Cache Coherence schemes for Multiprocessors", IEEE Computer, 1990.

Chapitre 8

Modèles et Spécifications

8.1 Introduction

La conception d'un programme comprend des phases de *spécification*, *programmation*, *vérification* et *évaluation des performances*. La spécification consiste à expliciter les propriétés attendues du programme à l'exécution. Ces propriétés concernent aussi bien des aspects qualitatifs comme l'absence de blocage et de famine (cf. chapitre 2) ou l'adéquation du programme au service que l'on attend de lui, que des aspects liés à ses performances.

Les phases de vérification et d'évaluation consistent à s'assurer de la conformité entre les exécutions du programme et ses spécifications. Pour vérifier cette conformité, il est nécessaire de pouvoir *modéliser* les exécutions du programme, c'est-à-dire de les représenter par un objet mathématique qui permettra une analyse formelle. Ce modèle est un graphe appelé *graphe des exécutions*. Intuitivement, ce graphe représente toutes les exécutions possibles du programme.

Ce chapitre est organisé de la manière suivante :

- le paragraphe 2 présente les modèles de graphes et leur génération à partir de réseaux de Petri.
- le paragraphe 3 est consacré à leur utilisation dans le cadre de la vérification de propriétés.
- le paragraphe 4 est consacré à l'utilisation des réseaux de Petri dans le cadre de l'évaluation de performances.

8.2 Modélisation des comportements d'un programme parallèle

8.2.1 Graphe des exécutions.

Rappelons qu'une exécution d'un programme séquentiel est un *processus*, c'est-à-dire une séquence (finie ou infinie) état-action-état... Les états d'un processus, ou états du programme à

l'exécution, sont le résultat des actions antérieures du processus : état des registres, valeurs des variables, du contrôle..., et les actions sont ce qui provoque le changement d'état.

Pour étudier les propriétés de la composition parallèle de programmes, on a besoin de modéliser les exécutions de cette composition parallèle : quels sont les *états* et les *actions* lors d'une exécution, et dans quel *ordre* les actions s'exécutent.

- Les *états* sont les éléments du produit cartésien de l'ensemble des états des composants (processus) : l'état du système global est défini par les états de chaque composant.
- Les *actions* sont les actions de l'un et/ou de l'autre des composants.

On se restreint ici à la composition parallèle asynchrone : dans ce modèle, on ne peut faire aucune hypothèse sur la durée des actions des processus, et on pose comme impossible que deux actions qui s'exécutent en parallèle se terminent au même instant ; en d'autres termes, on suppose que la probabilité de cet événement est nulle. Par conséquent, un changement d'état est provoqué par la fin de l'exécution d'exactement *une* action d'un des composants : lors d'une exécution, les événements de fin d'action sont strictement ordonnés dans le temps. Ainsi, une exécution parallèle est aussi une séquence état-action-état...

Le problème est de connaître *l'ordre* dans lequel s'exécutent les actions, ou plutôt tous les ordres possibles. Dans le cas d'un processus, on connaît un ordre total (unique) sur l'occurrence des action : à partir de ;'état initial (et pour une séquence d'entrées donnée), la séquence des états-actions caractérisant une exécution est toujours la même (en faisant abstraction des entrées-sorties). Dans le cas de la composition parallèle, on ne connaîtra généralement qu'un ordre partiel. Celui-ci provient :

- de l'ordre total sur les actions des processus
- des contraintes de synchronisation.

En d'autres termes, cela signifie que deux exécutions de la même composition parallèle ne donneront pas nécessairement la même séquence d'états-actions.

Dans le paragraphe 2.2, nous allons voir comment représenter les séquences d'exécution possibles de la composition parallèle de deux processus en fonction de différentes contraintes de synchronisation.

Dans le paragraphe 2.3, on présentera le modèle des Réseaux de Petri, et on étudiera une méthode pour construire le graphe de toutes les exécutions d'un système à partir de sa modélisation par des Réseaux de Petri.

8.2.2 Composition parallèle de processus.

Considérons l'exécution parallèle des deux processus séquentiels finis $P_1 et P_2$ (figure 8.1).

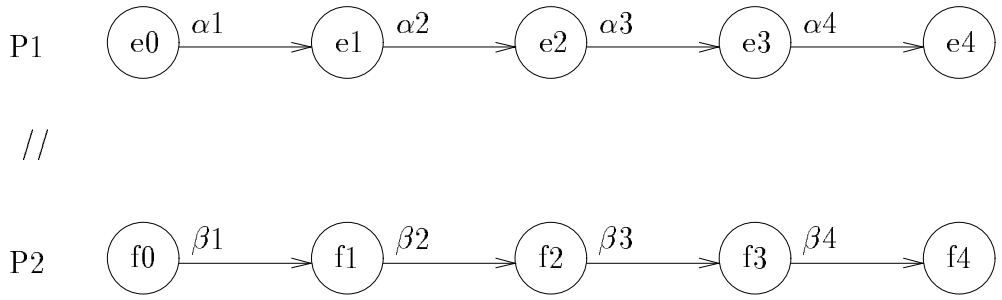


FIG. 8.1 – *Processus séquentiels finis en parallèle*

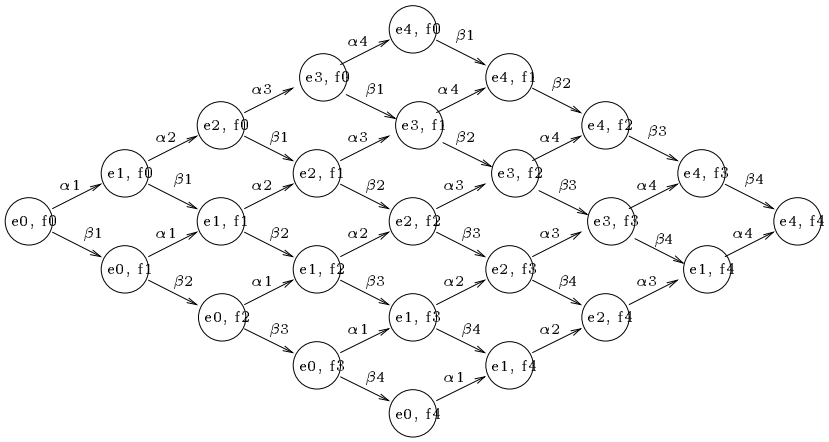


FIG. 8.2 – *Graphe des exécutions de P1//P2 sans contraintes de synchronisation*

Parallélisme sans synchronisation.

Les processus évoluent de façon totalement indépendante, on ne sait pas si telle action se passe avant, pendant, ou après telle autre ; plus précisément, compte tenu de notre hypothèse d'asynchronisme, on ne sait si une action donnée se termine avant ou après telle autre. Dans ce cas, la seule contrainte que doit respecter l'ordre des événements de fin d'action est l'ordre relatif des actions des deux processus : l'exécution $P_1//P_2$ est l'une des séquences d'actions qui respecte cette relation d'ordre.

On représente l'ensemble de ces séquences par un graphe dont les états sont les couples d'états des deux processus. Une séquence d'exécution possible est l'un des chemins de ce graphe (figure 8.2) depuis l'état initial (e_0, f_0) jusqu'à l'état final (e_4, f_4) : chacune des séquences d'actions de ces chemins est un *entrelacement* des séquences d'actions de P_1 et P_2. On appelle ce graphe *graphe des exécutions* du système $P_1//P_2$.

Ce cas est le cas le plus général (et le plus rare en pratique) : il s'agit de parallélisme sans aucune contrainte de synchronisation. Nous avons vu que les contraintes de synchronisation induisent des contraintes sur les relations d'ordre entre les actions des différents processus. L'ensemble des

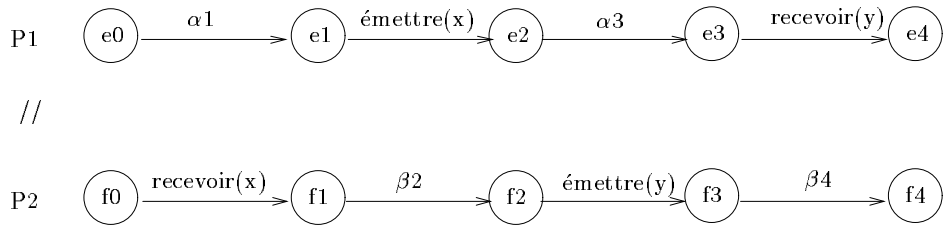


FIG. 8.3 – *Communication par rendez-vous*

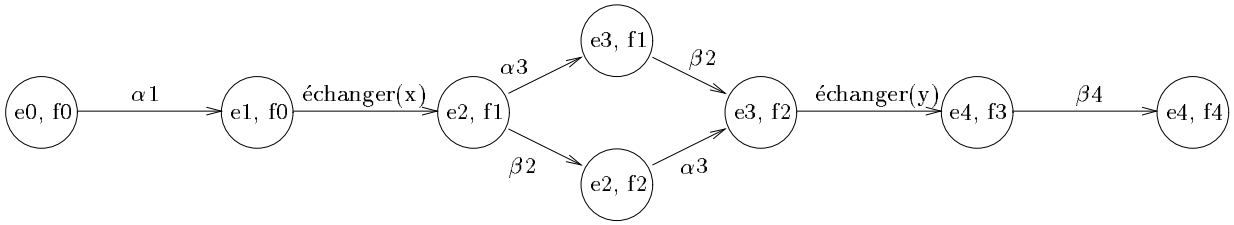


FIG. 8.4 – *Graphe des exécutions avec communication par rendez-vous*

séquences d'exécutions possibles se restreint alors à celles qui respectent ces relations d'ordre introduites. Examinons quelques unes de ces contraintes.

Communications par rendez-vous.

Imaginons par exemple que les processus communiquent par échange de variables : un échange est constitué de l'émission d'un message par un des processus et de la réception correspondante par l'autre processus (figure 8.3). Ces actions d'émission et de réception doivent se faire simultanément.

Les contraintes de synchronisation imposées par les deux rendez-vous sont les suivantes :

Les actions *émettre* et *recevoir* doivent se passer en même temps (modulo notre hypothèse d'asynchronisme ; cet abus de langage signifie que *émettre* ne peut s'exécuter que si le processus récepteur est prêt à exécuter l'instruction *recevoir*; bien entendu, dans ce cas particulier, il est évident que l'instruction *émettre* sera terminée avant l'instruction *recevoir*; un autre point de vue est de considérer qu'après la fin de l'action *émettre*, la seule action qui peut être observée est recevoir); il s'agit en fait d'une même action que nous appellerons *échanger*. Pour exécuter cette action les deux processus doivent être prêts à le faire: ainsi, P_2 n'exécute aucune action avant que P_1 n'ait exécuté a_1. Après le premier échange, les actions a_3 et b_2 s'exécutent dans un ordre indifférent. Mais lorsque P_1 a exécuté a_3, il attend que P_2 ait exécuté b_2 pour échanger la variable y, et réciproquement : là encore, il y a un rendez-vous entre les deux processus. Le graphe des exécutions de $P_1//P_2$ devient celui de la figure 8.4

Remarquons que les contraintes de synchronisation rendent certains états du produit cartésien inaccessibles. Elles imposent une restriction sur les séquences d'exécution possibles.

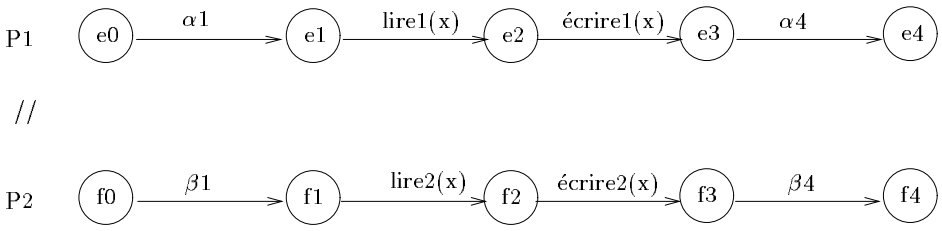


FIG. 8.5 – *Section critique*

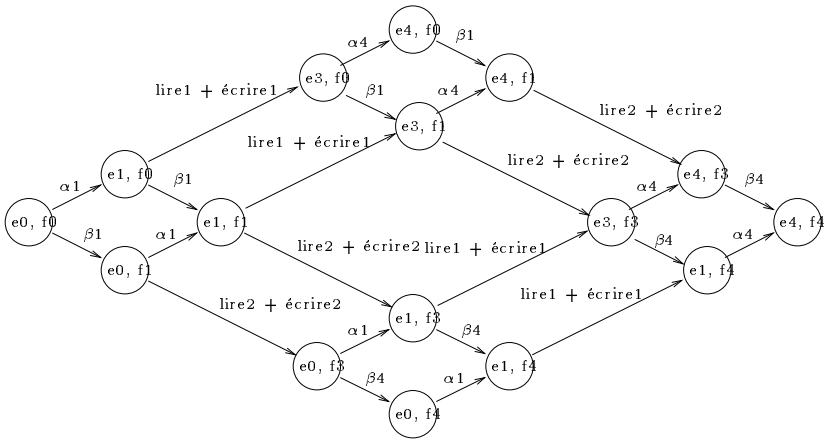


FIG. 8.6 – *Graphe des exécutions avec sections critiques*

Sections critiques

L'utilisation des sémaphores d'exclusion mutuelle pour mettre une suite d'actions en section critique a pour conséquence que cette suite d'actions devient une action indivisible.

Par exemple, si les suites d'actions $a_2 - a_3$ et $b_2 - b_3$ sont des accès à une même variable devant s'effectuer en exclusion mutuelle (figure 8.5), le graphe des exécutions possibles devient celui représenté sur la figure 8.6.

Encore une fois, il faut noter, compte-tenu de notre hypothèse d'asynchronisme, que le graphe ci-dessus n'exprime pas exactement que la suite d'opérations *lire*$_i$-*écrire*$_i$ doit être terminée avant que ne commence la suite *lire*$_j$-*écrire*$_j$. Il exprime moins que cela : c'est la fin de l'exécution de *lire*$_1$ (ou *lire*$_2$) qui provoque le changement d'état, donc le choix de l'une ou l'autre branche ; si l'on veut rester fidèle à la sémantique de l'exclusion mutuelle, il suffit de rajouter avant l'opération *lire*$_i$, une opération d'*entrée section critique* ; c'est le processus qui aura terminé le premier l'exécution de cette instruction qui aura l'autorisation de rentrer en section critique.

En résumé, la sémantique de l'opérateur // dépend de celle des synchronisations entre les

différents processus. Ces contraintes peuvent être de différente nature :

- rendez-vous : un processus doit attendre que les autres membres du rendez-vous soient présents pour franchir une transition
- exclusion mutuelle : interdiction de franchir la transition qui accède à une ressource si un autre processus l'utilise
- coopération (style producteurs-consommateurs) : le franchissement d'une transition dépend de l'histoire du système.

D'une façon générale, l'exécution d'une action par l'un des processus dépend non seulement de l'état du processus concerné, mais aussi de l'état des autres processus en parallèle. Pour construire l'ensemble des exécutions correspondant à la mise en parallèle de processus, il faut tenir compte de ces différents modes de synchronisation. Parmi les différents formalismes permettant de modéliser la mise en parallèle, nous étudions l'un des plus anciens et des plus utilsés, celui **des réseaux de Petri**.

8.2.3 Modélisation des contraintes de synchronisation à l'aide des réseaux de Petri.

Les réseaux de Petri ont été développés au début des années 60 par C.A. Petri. Leur but était de proposer un modèle formel pour les concepts d'actions asynchrones et concurrentes. Ils ont fait l'objet de nombreux travaux dans les années 70, et des recherches sur le sujet se poursuivent encore à l'heure actuelle. Leur intérêt provient aussi bien de l'étendue de leurs résultats théoriques que de la diversité de leurs applications. Nous donnons dans ce paragraphe une brève présentation informelle des réseaux de Petri. Pour une présentation complète et une synthèse des résultats et applications on peut se reporter à [Bra 82].

Présentation des Réseaux de Petri sur un exemple : l'exemple des pompiers.

Dans une première approche, un Réseau de Petri est un ensemble d'automates (les composants du système), auquel s'ajoutent des contraintes de synchronisation. Chaque pompier, pris indépendamment de toute contrainte, est modélisé par un automate, comme sur la figure 8.7

Lorsque le système s'exécute, le processus correspondant à chaque pompier est une séquence acceptée par l'automate correspondant. Dans chacun des automates, les états sont caractérisés par l'état du seau : V, lorsque le seau est vide, P, lorsqu'il est plein ; les transitions sont étiquetées par les actions qui modifient l'état du seau : *remplir*, *échanger*, *vider*. L'ensemble des pompiers est représenté par l'ensemble des automates, auquel on ajoute des conditions de franchissement de certaines transitions : les transitions qui doivent se passer simultanément (nous ferons désormais cet abus de langage, voir remarques du § 2.2) sont considérées comme une seule transition, et on représente par un trait qui les relie, le fait que ces transitions doivent s'exécuter en même temps (figure 8.8).

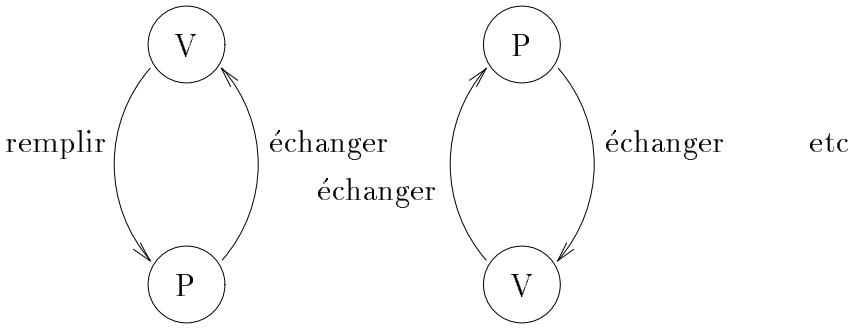


FIG. 8.7 – *Automate pour un pompier.*

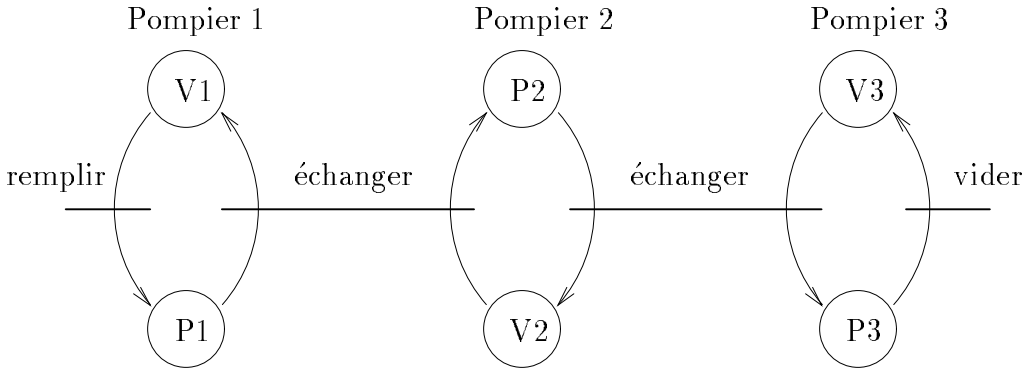


FIG. 8.8 – *Automate pour trois pompiers.*

Le réseau d'automates obtenu est le réseau de Petri modélisant le comportement des pompiers. Pour définir les états et l'évolution du réseau, les conventions utilisées sont les suivantes :

- Les *états des composants* sont appelées *places*. Par exemple, le réseau des trois pompiers comporte 6 places, que nous appellerons V_1, P_1, V_2, P_2, V_3 et P_3.

- L'*état du système* est défini par la présence de *jetons* dans les places. Par exemple l'état dans lequel le premier et le troisième pompier ont un seau plein, et le deuxième un seau vide est représenté sur la figure 8.9.

  Un état est aussi appelé *marquage*. On représente aussi un marquage par un vecteur indexé par les places du réseau, et tel que la composante d'indice p est le nombre de jetons dans la place p. L'état du réseau représenté sur la figure 8.9 correspond au vecteur $[V_1, P_1, V_2, P_2, V_3, P_3] = (0, 1, 1, 0, 0, 1)$.

- Une *transition* peut être exécutée si et seulement si il y a au moins un jeton dans chacune de ses places origine. Après exécution de cette transition, chacune de ses places origines perd un jeton, et chacune de ses places d'arrivée en gagne un. Par exemple, la figure 8.10 représente l'état d'un réseau de deux pompiers avant et après exécution de la transition *échanger*.

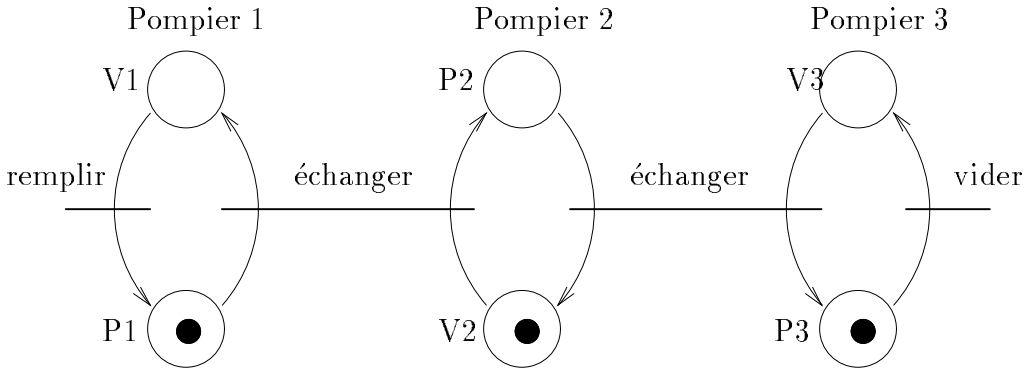


FIG. 8.9 – *État d'un système.*

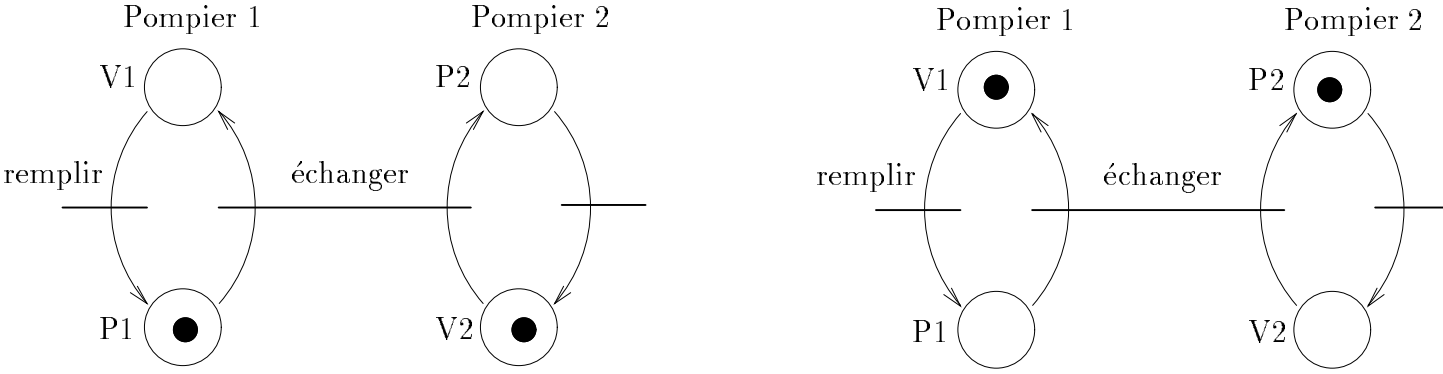


FIG. 8.10 – *Exécution de la transition échanger.*

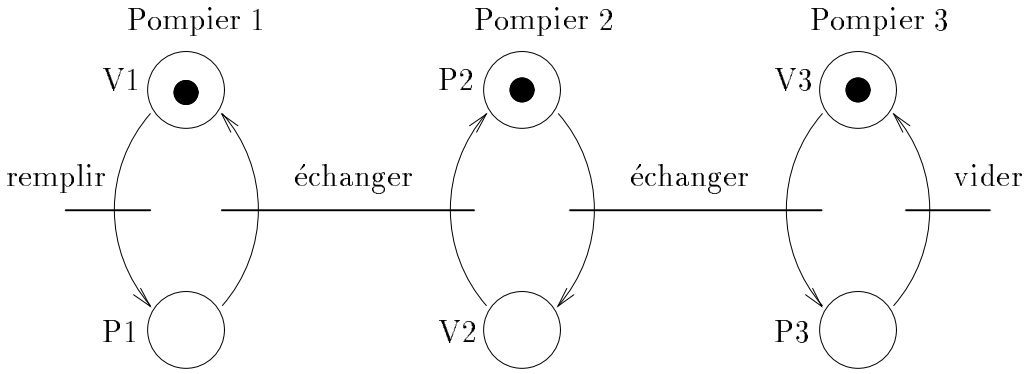


FIG. 8.11 – *État d'un réseau ayant deux successeurs.*

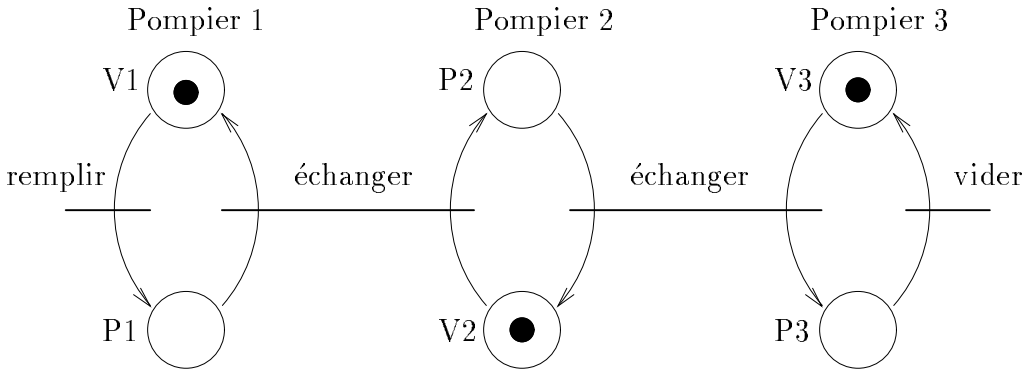


FIG. 8.12 – *Le réseau de Petri des trois pompiers et son marquage initial.*

Remarquons que dans un état donné, les conditions de franchissement de plusieurs transitions peuvent être vraies simultanément. Dans ce cas, l'état du réseau peut avoir plusieurs successeurs : par exemple, le réseau de la figure 8.11 est dans un état ayant deux successeurs selon que la transition suivante est *remplir seau*, ou *échanger* (entre les pompiers 2 et 3), autrement dit le système est indéterministe.

– Un réseau de Petri est entièrement défini par la donnée de ses places, de ses transitions, et de son *marquage* (ou état) *initial*. Ainsi, le système des trois pompiers ayant initialement un seau vide est défini par la figure 8.12.

Séquence d'exécution et graphe des exécutions dans un Réseau de Petri.

Comme nous l'avons évoqué à plusieurs reprises, une exécution du système est une suite état-transition-état... Un état d'un réseau de Petri est un marquage de ses places. Dans chaque marquage, une ou des transitions peuvent être franchies, après lesquelles on atteint un des marquages successeurs possibles du marquage courant. Par conséquent, un algorithme de construction du graphe des

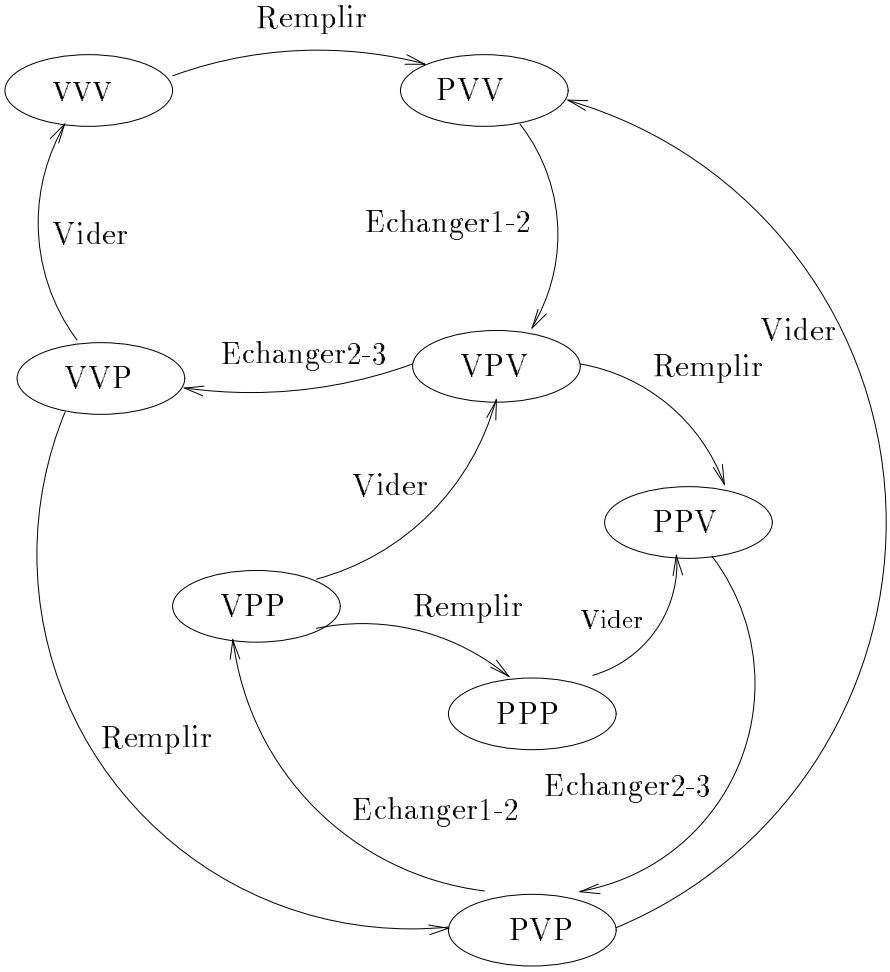


FIG. 8.13 – *Graphe des exécutions du système des trois pompiers.*

exécutions consiste à calculer de proche en proche à partir du marquage initial, tous les marquages du réseau.

Pour le système de trois pompiers, la figure 8.13 représente le graphe des exécutions. Les états sont représentés de manière abrégée plus lisible, en donnant l'état du seau de chaque pompier. Par exemple, l'état initial est noté $(V\ V\ V)$ à la place de $(1,0,1,0,1,0)$, l'état donné en exemple au paragraphe précédent est noté $(P\ V\ P)$ à la place de $(0,1,1,0,0,1)$. Bien évidemment, le graphe des exécutions est identique à celui donné au chapitre 2.

Cas général

Dans l'exemple des pompiers, chaque marquage du système est tel qu'il y a au plus une marque dans chaque place. Dans le cas général, le nombre de marques peut être quelconque. On définit

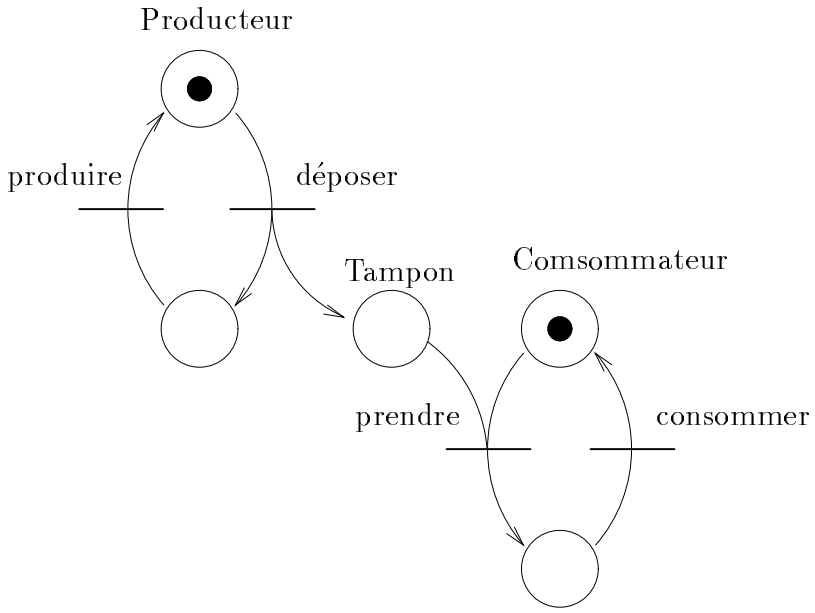


FIG. 8.14 – *Producteurs-consommateurs avec tampon non borné.*

alors les notions de transition et d'exécution de transitions de la manière suivante :

- la condition de franchissement d'une transition est la présence d'*au moins une* marque dans chacune des places origines de cette transition.
- lorsque la transition est franchie, l'état successeur est celui dans lequel une marque a été retirée de chaque place origine et une marque a été ajoutée dans chaque place en sortie.

Modélisation d'autres types de synchronisation à l'aide de Réseaux de Petri.

Avec l'exemple des pompiers, on a vu comment modéliser la communication par rendez-vous. On donne ici deux exemples modélisant d'autres types de coopération : les producteurs-consommateurs, et l'exclusion mutuelle sur l'accès à un ressource.

- Les producteurs-consommateurs (figure 8.14)

  Le producteur et le consommateur sont des processus qui communiquent par un tampon dans lequel le producteur dépose des messages qui sont lus par le consommateur. Ce dernier ne peut lire un message que s'il y en a au moins un dans le tampon : une des places origine de la transition *prendre* est le tampon ; le franchissement de cette transition retire un jeton (un message) du buffer. Remarquons que ce réseau ne permet pas de modéliser le fonctionnement du tampon en Fifo, par exemple.

  Une caractéristique de ce réseau, pourtant simple est qu'il a un nombre infini d'états possibles. En effet, le producteur peut déposer autant de messages qu'il le désire, c'est-à-dire que le

FIG. 8.15 – *Producteurs-consommateurs avec tampon borné.*

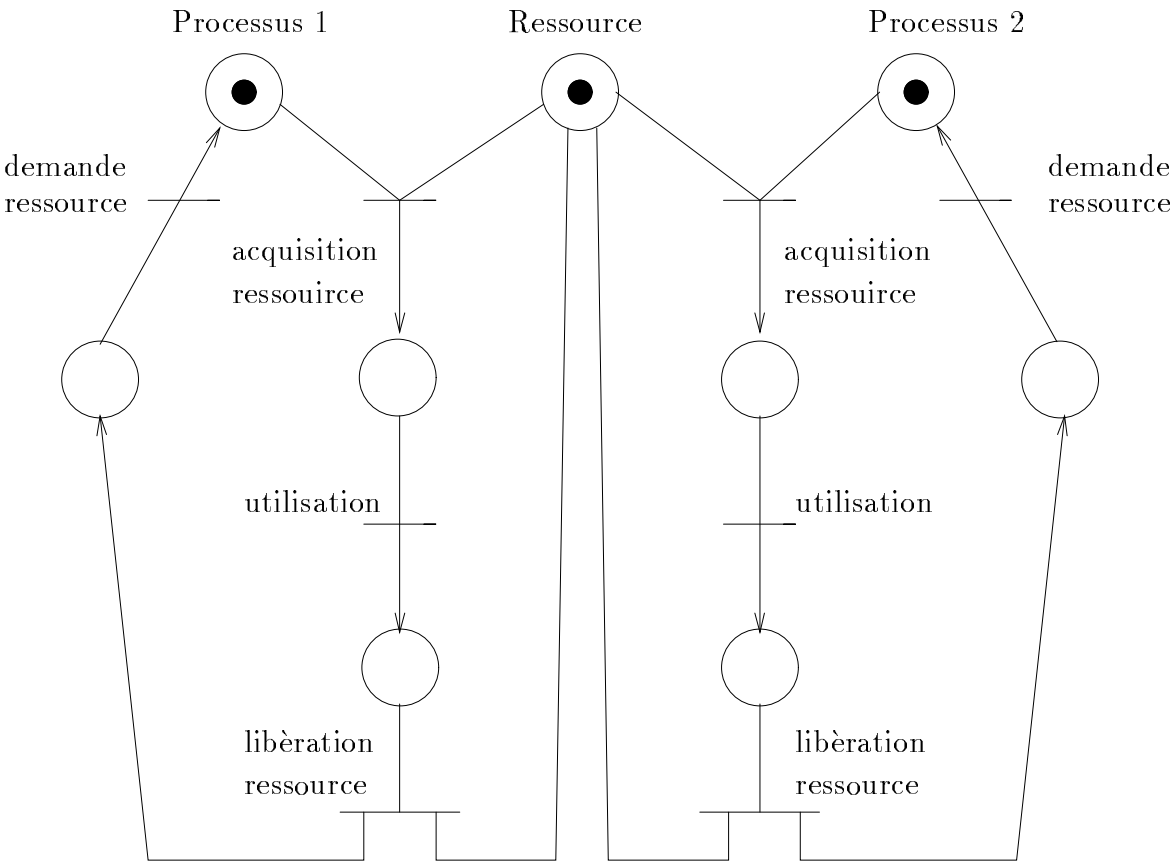


FIG. 8.16 – *Exclusion mutuelle.*

tampon n'est pas borné. Le nombre de jetons dans ce dernier n'étant pas borné, le nombre d'états est infini : on dit que le réseau n'est pas borné.

Le réseau représenté dans la figure 8.15 modélise une variante dans lequel le nombre de "cases" du tampon est borné à 2. Ceci introduit une autre condition de synchronisation sur la limitation des ressources (les cases du tampon) disponibles. Intuitivement, il y a une symétrie dans les actions de prise et de dépôt de message : la première retire une case pleine et produit une case vide, la seconde retire une case vide et produit une case pleine.

– L'exclusion mutuelle sur l'accès à une ressource (figure 8.16)

  Ce réseau modélise deux processus qui se partagent une ressource en exclusion mutuelle. Ce réseau peut se généraliser aisément à un nombre quelconque de processus, et à un nombre quelconque n de ressources : il suffit dans ce cas de modifier le marquage initial en mettant n jetons dans la place "Ressource".

Conclusion

Nous n'avons vu ici qu'un bref aperçu des réseaux de Petri en tant qu'outils de modélisation. Au même titre que les automates pour les systèmes séquentiels, ils sont particulièrement bien adaptés à la description du contrôle des systèmes parallèles, notamment les conflits, le séquencement et les mécanismes de base de communication. Ils sont couramment employés pour modéliser la partie commande d'automatismes industriels, facilitant le passage du modèle à sa réalisation.

Cependant, l'intérêt majeur des réseaux de Petri provient de la possibilité de construire des méthodologies d'analyse autant quantitatives que qualitatives, fondées sur les résultats théoriques nombreux. Nous n'aborderons pas dans ce cours les méthodes d'analyse des réseaux proprement dits. Par contre le paragraphe suivant est consacré à l'analyse du graphe des exécutions d'un système communicant. Ce graphe peut être, comme nous l'avons vu, obtenu de manière automatique en construisant le graphe des marquages d'un réseau de Petri modélisant ce système.

Pour que cette phase de génération de graphe soit automatisable dans le cas général, il est nécessaire que le graphe produit soit fini. De même, les méthodes d'analyse que nous présentons dans les paragraphes suivants s'appliquent à tous les graphes finis. Il existe des extensions de ces méthodes pour des graphes infinis de structure particulière. Remarquons que dans le cas général, les programmes ont un nombre infini d'états, du fait, par exemple qu'on utilise une variable à valeur dans N, ce qui rend le problème de la vérification formelle indécidable. Cependant, l'étude des protocoles, par exemple, dans les réseaux et systèmes distribués montre que ce sont surtout les algorithmes de contrôle qui sont complexes et l'analyse du contrôle de ces systèmes peut être faite en faisant abstraction des données: les variables de contrôle ne prenant généralement qu'un nombre fini de valeurs, ce modèle peut être fini. C'est à la modélisation de ce type de systèmes que l'on s'intéresse dans ce chapitre.

8.3 Vérification qualitative

8.3.1 Introduction

On étudie dans ce paragraphe l'expression et la vérification de propriétés logiques des systèmes communicants. Ces propriétés sont généralement appelées spécifications du programme. A ce niveau, nous ne ferons pas de distinction entre les propriétés intrinsèques des exécutions (comme l'absence de blocage), et les propriétés de service (ce que l'on veut observer). Le programme et ses spécifications décrivent un ensemble de comportements : pour le premier, il s'agit de ses exécutions, pour les secondes, il s'agit des comportements que l'on attend du programme. Ainsi, le formalisme utilisé pour l'expression des spécifications peut être le même que celui utilisé pour décrire le programme, mais on peut aussi utiliser un formalisme plus abstrait. Nous verrons deux exemples de tels formalismes pour l'expression des spécifications : dans le premier cas, il s'agit d'automates, dans le second, de formules logiques. La vérification consiste à comparer les comportements décrits par le programme et par les spécifications. Les spécifications sont une description plus abstraite que le programme, et représentent une classe de programmes: vérifier consiste à tester l'appartenance du programme à cette classe. Lorsque programme et spécifications sont décrits par le même formalisme, cette comparaison se fait modulo une relation d'équivalence appropriée : les comportements décrits

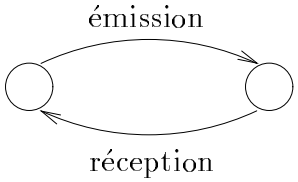


FIG. 8.17 – *automate emetteur-recepteur.*

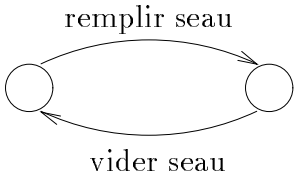


FIG. 8.18 – *Automate remplir-vider.*

doivent être équivalents. Dans le cas où les spécifications sont décrites par des formules logiques, il s'agit de vérifier que les comportements du programme satisfont (sont un modèle de) ces formules.

8.3.2 Spécifications

Spécifications comportementales

Ces spécifications décrivent, modulo une certaine abstraction, les comportements souhaités du programme, on peut les représenter par un automate S. Par exemple, un émetteur et un récepteur communiquent, et on veut que leur comportement limité aux émissions et aux réceptions soit une suite infinie alternant émission et réception. Cette spécification est représentée par un automate comme sur la figure 8.17

Le graphe du programme peut être vu lui aussi comme un automate P qui accepte l'ensemble de ses exécutions (modulo la même abstraction). La vérification consiste à tester si $L(P) \subset L(S)$, et la technique employée est une généralisation de celle utilisée pour résoudre ce problème dans le cas des automates réguliers.

Exemple. On peut vérifier que le comportement du système des trois pompiers limité aux actions "remplir seau" et "verser seau" n'est pas accepté par l'automate de la figure 8.8, parce qu'on peut par exemple, remplir deux fois successivement le seau dans le système décrit et non dans cet automate.

Spécifications de type logique

Des spécifications de type logique sont des propriétés globales des comportements du système, telles que l'absence de blocage, l'exclusion mutuelle : par exemple, on peut exprimer le fait que les processus $P_1 ... P_n$ doivent être mis en exclusion mutuelle pour écrire dans une même zône mémoire, par la formule :

Pour tout *instant* $\exists i, P_i$ écrit $\Rightarrow \forall j \neq i, P_j$ n'écrit pas.

Une propriété d'invariance comme "la variable x ne peut prendre que les valeurs 1 et 2" peut être exprimée de manière analogue. Un formalisme adapté à ce type de spécifications est l'expression par des formules de **logique temporelle**. Les logiques temporelles sont des extensions de la logique classique par des opérateurs modaux permettant de construire des formules dont l'interprétation dans un état dépend de l'interprétation dans les autres états : par exemple, il ne suffit pas de savoir que la variable x vaut 1 dans l'état q, mais il faut aussi savoir sa valeur dans tous les états accessibles à partir de q. La valeur de vérité d'une formule n'est pas quelque chose d'absolu, mais dépend du temps.

Généralement, une spécification d'un système comporte à la fois des propriétés de type comportemental et de type logique, qui sont complémentaires. Le formalisme des automates peut sembler mieux adapté à l'expression des premières, et les logiques temporelles aux secondes, mais leur pouvoir d'expression est en réalité similaire.

On considère maintenant un exemple de logique temporelle à sémantique arborescente, et nous montrerons comment exprimer des propriétés dans cette logique, et les vérifier sur le graphe du programme.

Définition d'une logique temporelle.

On définit une logique permettant d'exprimer les propriétés du modèle du comportement d'un système parallèle. Ce modèle est le graphe des exécutions décrit au cours du paragraphe 2. On le représente par un quadruplet $(S, q_0, \rightarrow, A)$ où :

- S est l'ensemble fini des états du programme lors des exécution : valeurs des variables ...
- q_0 est un élément de S, l'état initial du programme
- A est un ensemble d'actions élémentaires du programme
- $\rightarrow$ est une relation de transition sous-ensemble de $S \times A \times S$

Rappelons que tout chemin dans ce graphe à partir de l'état initial est une séquence d'exécution du programme.

Syntaxe. La logique temporelle considérée est formée à partir du calcul propositionnel (prédicats de base, $\vee, \wedge, \neg$), et d'opérateurs modaux notés *toujours* et *inévitable.*

Sémantique. Les formules expriment des propriétés des exécutions des processus considérés à travers leurs états. La sémantique est donnée par une relation de satisfaction $\models$ entre états du programme et formules et se définit par induction structurelle sur les formules. On en donne ici une définition informelle pour les prédicats et les opérateurs de la logique :

- un prédicat sur les variables du programme est vrai dans les états où les valeurs des variables satisfont ce prédicat, par exemple, $q \models$ "$x = y$" si dans l'état q, les valeurs des variables x et y sont égales.
- un prédicat sur les actions adjacentes à un état comme *après*(a) est vrai dans les états atteints après exécution de l'action a ; de la même manière, *avant*(a) est vrai dans les états à partir desquels l'action a peut être exécutée.

A de tels prédicats s'ajoutent les prédicats prédéfinis *vrai* (satisfait par tous les états), *init* (qui caractérise l'état initial du système) et *puits* (qui est satisfait par les états n'ayant pas de successeur dans le graphe).

Aux opérateurs de la logique, on associe la sémantique suivante :

- $\vee$, $\wedge$, $\neg$: sémantique habituelle.
- $q \models$ *toujours* f est vrai si la formule f est satisfaite par tous les états du graphe accessibles à partir de q.
- $q \models$ *inévitable* f est vrai si sur tout chemin suivi à partir de q, il existe un état où la formule f est satisfaite (voir figure 8.19).

Expression de propriétés dans cette logique

Considérons un sémaphore binaire d'exclusion mutuelle (figure 8.16) :

Exprimons la propriété d'exclusion mutuelle dans la logique temporelle que nous avons définie :

La ressource x_2 doit n'être utilisée au plus que par un utilisateur. En termes de marques du réseau de Petri, il doit y avoir au plus une marque dans x_4 ou dans x_5 : $(x_4 = 0 \vee x_5 = 0)$. On veut que cette propriété soit toujours vraie à partir de l'état initial, c'est-à-dire que ce dernier vérifie la formule *toujours*$(x_4 = 0 \vee x_5 = 0)$.

8.3.3 Vérification de propriétés sur le graphe des marquages

Exemple introductif

Si la propriété à vérifier est exprimée par la formule f, on calcule l'ensemble $|f| = \{q / q \models f\}$. Par exemple, l'ensemble $|$ *toujours*$(x_4 = 0 \vee x_5 = 0)|$ peut être calculé sur le graphe des marquages

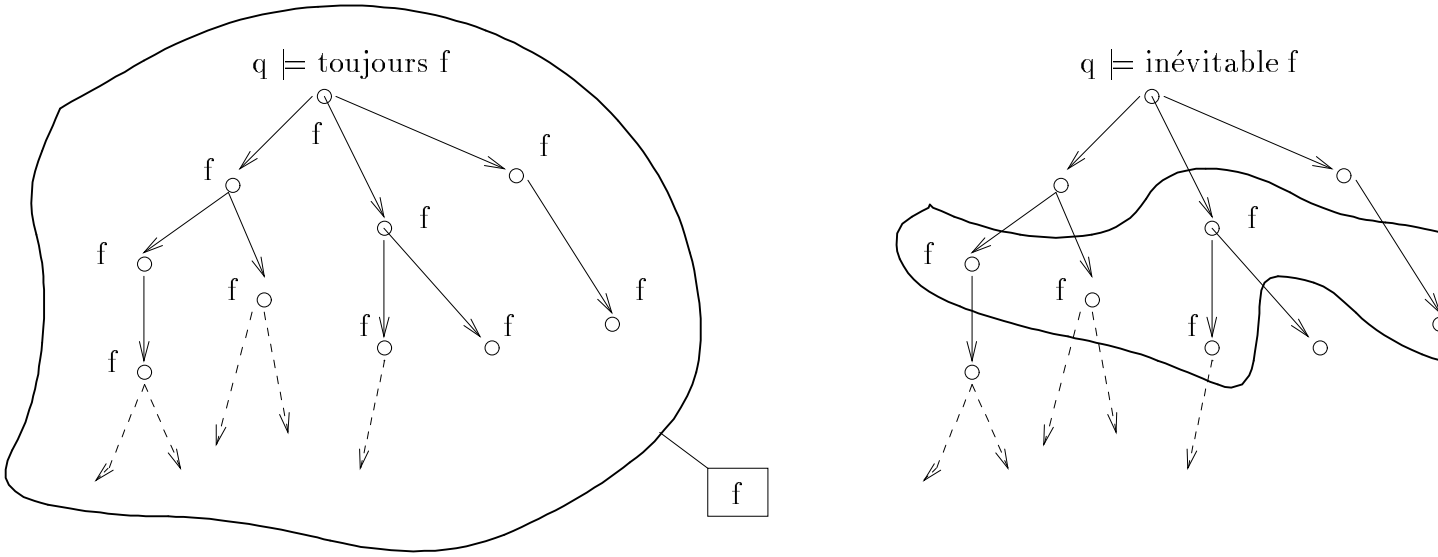


FIG. 8.19 – *Sémantique des opérateurs "toujours" et "inévitable".*

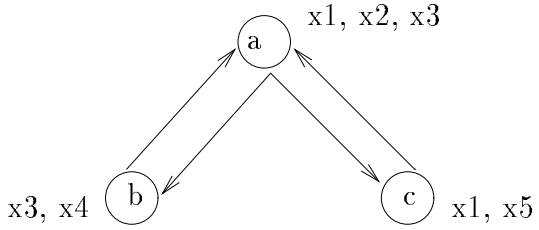


FIG. 8.20 – *Graphe des exécutions du sémaphore d'exclusion mutuelle.*

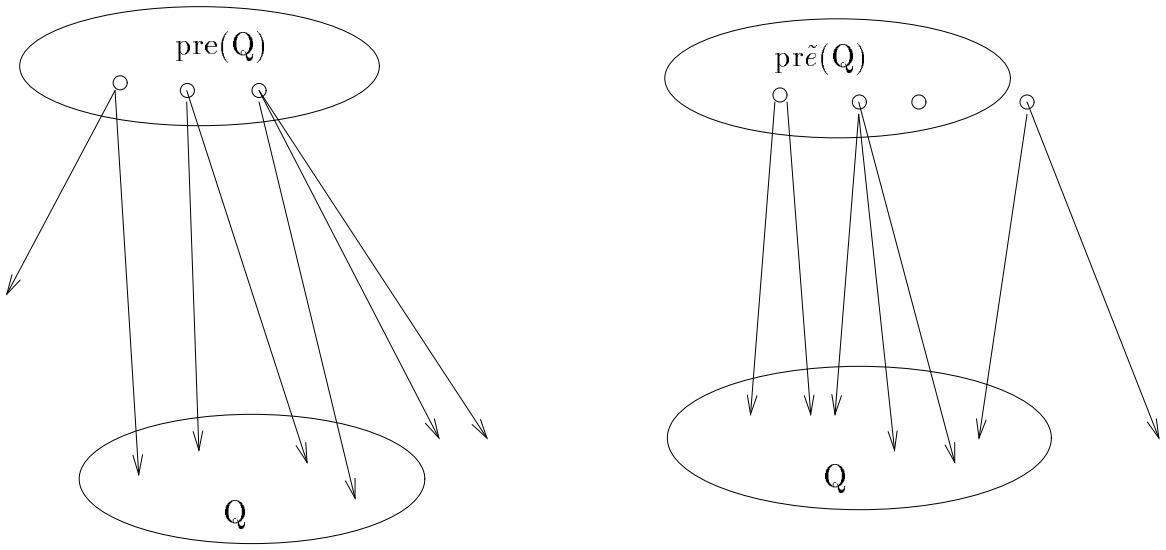


FIG. 8.21 – *Opérateurs pre et pretilda.*

du réseau de Petri de l'exclusion mutuelle (figure 8.20), dans lequel les états sont étiquetés par celles des variables x_i du programme qui ont pour valeur 1 (les autres ont pour valeur 0).

L'ensemble des états du graphe est $S = \{a, b, c\}$. La formule : $x_4 = 0 \vee x_5 = 0$ est vraie pour tous les états, y compris l'état initial, et il est donc évident que a (l'état initial) satisfait *toujours*$(x_4 = 0 \vee x_5 = 0)$ La propriété d'exclusion mutuelle est donc vraie.

En ce qui concerne la propriété : $x_4 > 0$, elle n'est vraie que pour l'ensemble d'états $\{b\}$ et on ne peut pas étendre cet ensemble avec des états dont b est successeur *inévitable*. Par suite, *inévitable*$(x_4 > 0)$ n'est pas satisfait par l'état initial et une famine est possible.

On devine, à travers cet exemple, que le calcul esquissé repose sur une définition récurrente d'ensembles d'états. Dans le cas de la formule *toujours f*, on partira de l'ensemble de tous les états satisfaisant f, dont on cherchera le plus grand sous-ensemble stable par la relation de transition (dans l'exemple, c'est S). En ce qui concerne la formule *inévitable f*, au contraire, on part de l'ensemble des états qui satisfont f, et on cherche à l'étendre à tous les états qui conduisent inévitablement à ceux-ci, jusqu'à trouver un ensemble stable. La méthode générale repose de fait sur la détermination d'un point fixe d'une fonction sur des ensembles définis récursivement.

Méthode générale

Si Q est un ensemble d'états, on note $pre(Q)$ l'ensemble des états dont un successeur appartient à Q, et $pr\tilde{e}(Q)$ l'ensemble des états dont tous les successeurs sont dans Q (figure 8.21). Par convention, les états puits (qui n'ont pas de successeur) appartiennent à $pr\tilde{e}(Q)$. Par conséquent, l'ensemble des états dont tous les successeurs sont dans Q et qui ont au moins un successeur dans Q est l'ensemble $pre(Q) \cap pr\tilde{e}(Q)$.

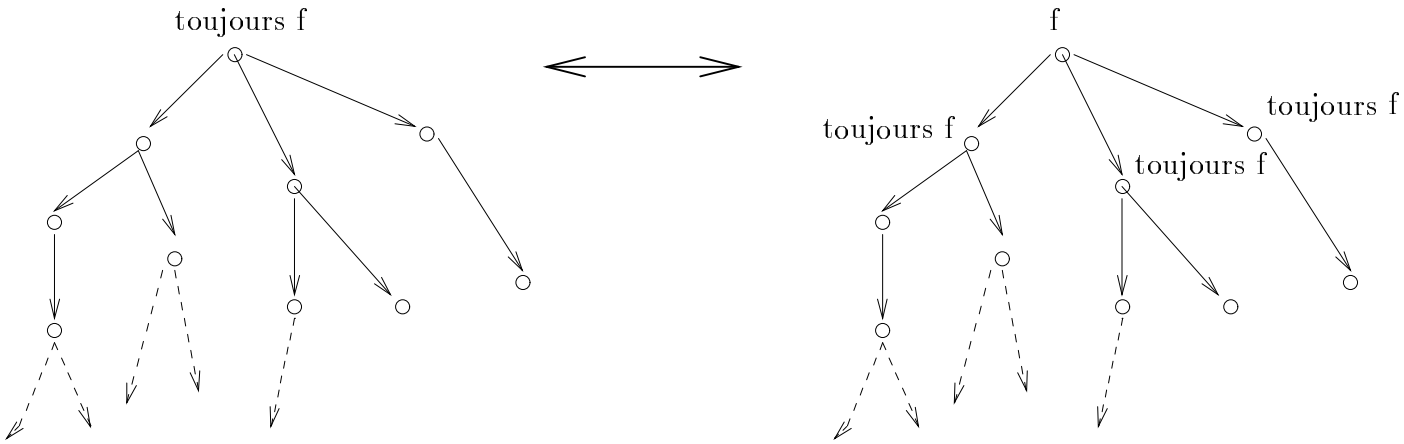


FIG. 8.22 – *Propriété de l'opérateur toujours.*

L'opérateur *toujours* a la propriété suivante (figure 8.22)

$(q \models f \ et \ q \in pr\tilde{e}(toujours\ f)) \Leftrightarrow q \models toujours\ f$

Note (pour avoir $q \models$ *toujours f*, il suffit d'avoir ($q \models f$ et que tous les successeurs de q satisfassent *toujours f*, et réciproquement).

On a en particulier :

$|toujours\ f| \subset |f|$,

et on peut montrer que l'ensemble $|$ *toujours f* $|$ est le plus grand point fixe de l'équation :

$X = |f| \cap pr\tilde{e}(X)$

En ce qui concerne l'opérateur *inévitable*, sa propriété s'énonce (figure 8.23) :

$(q \models f \ ou \ q \in pr\tilde{e}(|inévitable\ f|) \ et \ \ q \in pre(|inévitable\ f|)) \Leftrightarrow q \models inévitable\ f.$ (2)

Note pour avoir $q \models$*inévitable f*, il suffit que $q \models f$, ou que chacun des successeurs de q satisfassent *inévitable f*, et qu'il y en ait au moins un, et réciproquement.

On a :

$|f| \subset |$ *inévitable f* $|$

et on peut montrer que l'ensemble $|$*inévitable f*$|$ est le plus petit point fixe de l'équation :

$Y = |f| \cup (pr\tilde{e}(Y) \cap pre(Y))$

Soient F et G les fonctions de l'ensemble des parties de S dans lui-même définies par :

$F(X) = |f| \cap pr\tilde{e}(X)$

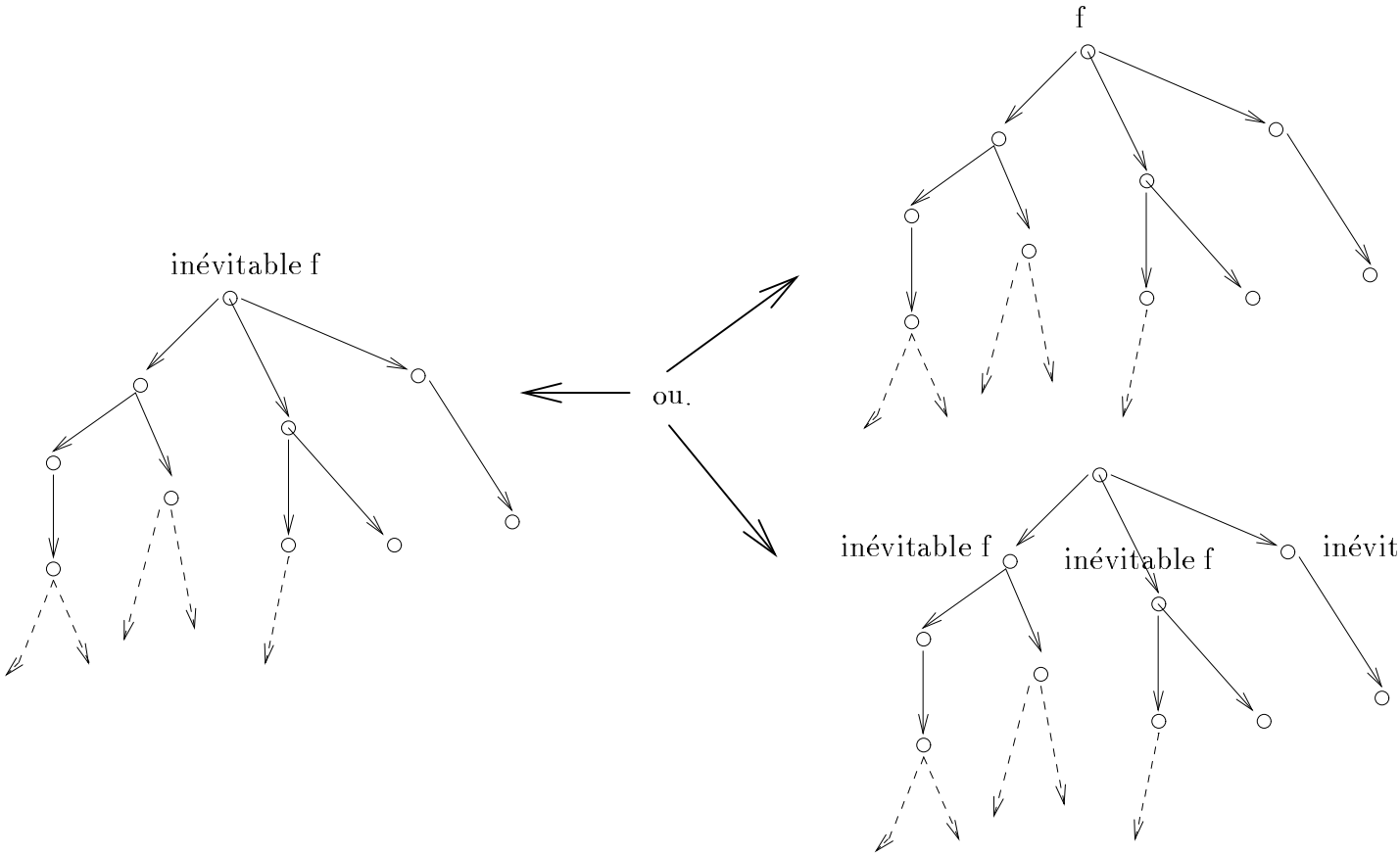


FIG. 8.23 – *Propriété de l'opérateur inévitable.*

et

$G(Y) = |f| \cup (\tilde{pre}(Y) \cap pre(Y))$

Ces fonctions étant des fonctions croissantes dans un treillis fini sont continues, par conséquent, on déduit du théorème de Kleene :

$|toujours\ f| = \bigcap_{i=0}^{\infty} F(S)$

et $|inévitable\ f| = \bigcup_{i=0}^{\infty} G(\emptyset)$

Ces équations nous donnent un moyen de calculer effectivement les ensembles $|$ *toujours* $f|$ et $|$*inévitable* $f|$ en itérant le calcul des fonctions F et G jusqu'à stabilisation.

Exemple

Calculons l'ensembles $|$ *toujours*$(\ x_4 = 0 \vee x_5 = 0)|$ sur le graphe des marquages du réseau de Petri de l'exclusion mutuelle (figure 8.20).

L'ensemble des états du graphe est $S = \{a, b, c\}$. Soit $G = (x_4 = 0 \vee x_5 = 0)$; on a : $|g| = \{a, b, c\}$, par conséquent, la fonction F correspondante est définie par :

$F(X) = \{a, b, c\} \cap \tilde{pre}(X)$

Calculons la suite X_i définie par :

$X_0 = S$

et $X_{i+1} = F(X_i)$

$X_0 = \{a, b, c\}$; $\tilde{pre}(X_0) = \{a, b, c\}$

$X_1 = \{a, b, c\} \cap \tilde{pre}(X_0) = X_0$,

par conséquent :

$|$ *toujours*$(\ x_4 = 0 \vee x_5 = 0)| = \{a, b, c\}$

On en déduit que l'état a (l'état initial) satisfait *toujours*$(x_4 = 0 \vee x_5 = 0)$ et donc que la propriété d'exclusion mutuelle est vraie.

8.4 Évaluation quantitative

8.4.1 Introduction

L'un des objectifs de tout programme parallèle est d'être plus rapide que le programme séquentiel équivalent. Dans ce but, il faut que la synchronisation nécessaire à l'échange d'information dans

un programme parallèle ne soit pas d'un coût prédominant par rapport au calcul. Deux facteurs caractérisent ce coût :

- le nombre de points de synchronisation
- le délai induit par chaque point de synchronisation

Dans ce paragraphe, nous ne traitons que le second point. La conception d'un algorithme de contrôle présentant aussi peu de synchronisation que possible a été traitée dans le chapitre 6. On cherche ici à évaluer le retard induit par les points de synchronisation et plus généralement la vitesse d'un programme parallèle. Cette vitesse calculée est à comparer aux spécifications du programme en terme de performances.

Pour l'évaluation des performances en général [PF 89], et en particulier dans le cas des systèmes parallèles, deux approches générales sont possibles :

- par modélisation,
- et par mesure.

Dans le second cas, le programme parallèle est écrit, implémenté, et un relevé de mesure (on dit que l'on instrumente le programme) permet de calculer les délais induits par chaque point de synchronisation. Dans le premier cas, qui nous intéresse, le programme peut-être modélisé a priori afin de prédire ses performances (bien sûr, il peut l'être aussi après implémentation afin de comprendre son comportement). Les modèles sur lesquels nous travaillons sont basés sur ceux que nous avons présentés dans le paragraphe 2, auxquels on ajoute des informations sur la durée des actions. Deux traitements de ces modèles sont possibles :

- la simulation,
- et le calcul analytique.

La simulation est la reproduction par un programme informatique (prenons-le séquentiel pour simplifier) du comportement du modèle. Une simulation comprend un relevé de mesure du modèle simulé afin d'en évaluer les performances. La simulation est aussi une méthode pour explorer un graphe des exécutions et vérifier des propriétés d'un programme modélisé. Nous ne traiterons pas ici cette technique, qui n'a rien de spécifique aux modèles de calcul parallèle.

Faire un calcul analytique signifie poser les équations vérifiées par le modèle et les résoudre. Bien sûr, pour poser des équations, il nous faut d'abord poser des hypothèses de calcul. Dans ce paragraphe, nous présentons l'un des modèles les plus simples, basés sur les hypothèses de Markov.

La suite de ce chapitre est organisée comme suit : nous rappelons d'abord dans le paragraphe 8.4.2 les notions de probabilités indispensables à la compréhension de ce chapitre ; dans le paragraphe 8.4.3, nous exposerons alors le théorème de calcul fondamental de la théorie de Markov et son application à un graphe d'exécution ; enfin, dans le paragraphe 8.4.4, nous présentons les réseaux de Petri stochastiques et dans le paragraphe 8.4.5 un exemple afin de comprendre la démarche générale de résolution d'un modèle.

8.4.2 Approche probabiliste

Lorsqu'il est nécessaire de représenter les durées dans un programme, trois cas se présentent: soit ces durées sont connues exactement (on dit alors que ces durées sont déterministes), soit ces durées sont connues par leur loi de probabilité, soit ces durées sont totalement inconnues. Le troisième cas est le cadre d'étude du paragraphe précédent sur la vérification : on veut vérifier des propriétés quelle que soit la durée des actions en présence. Il n'est pas possible, par contre, de faire de l'évaluation des performances dans ce cadre, puisque rien n'est valué dans les hypothèses. Dans le deuxième cas, les durées exactes ne sont pas connues, mais on sait chiffrer l'incertitude. Le premier cas d'un point de vue mathématique n'est qu'un cas particulier du second : l'incertitude est nulle. Dans les deux cas, la probabilité avec laquelle la durée concernée se trouve dans un intervalle fixé peut être estimée. Ces probabilités, pour tout intervalle, forment ce qu'on appelle la *distribution de probabilité* ou *loi de probabilité* de la durée. La durée elle-même s'appelle une *variable aléatoire*, ce qui veut exactement dire que cette quantité n'est connue que par sa distribution de probabilité. La théorie des probabilités nous permet de calculer la moyenne des variables aléatoires. Grâce à des théorèmes appelés *Loi des grands nombres*, il est démontré que cette moyenne est la même que celle que l'on calcule par une moyenne arithmétique (bien sûr, sous certaines hypothèses). Dans la suite, on note $P(x \leq X \leq z)$ la probabilité que la variable aléatoire X soit comprise entre x et z. Cette probabilité est un réel compris entre 0 et 1. L'ensemble des valeurs de $P(x \leq X \leq z)$, pour tout x, z positifs, constitue la distribution de probabilité de la variable aléatoire X.

Puisque l'on considère des durées, deux types d'échelle de temps sont possibles. L'échelle de temps est discrète lorsque toutes les durées sont multiples d'une durée unitaire, et continue lorsque toutes les valeurs réelles sont permises pour les durées. Ne sera traitée ici que le cas de l'échelle de temps continue, l'autre aspect étant à peu de choses près identique. L'échelle de temps continue s'apparente naturellement à l'hypothèse de modèle asynchrone, ce qui sera justifié par la suite.

Étant donné un programme dont la durée de vie est X, les quantités qui nous intéressent sont les probabilités que le programme se termine avant z : $P(0 \leq X \leq z)$, que l'on notera $P(X \leq z)$. Parmi l'ensemble des lois de probabilités qui modélisent des durées (donc des quantités positives), celles qui vont nous être utiles sont celles dites *sans mémoire*, c'est-à-dire telles que pour tout z et tout x :

$$P(X \leq x + z \mid X \geq x) = P(X \leq z).$$

Dire que la variable aléatoire X est sans mémoire, signifie que sachant que X a déjà excédé la valeur x, sa durée de vie ultérieure a la même distribution que sa durée de vie totale lorsqu'il a démarré. En d'autres termes, le fait que ce programme ait déjà tourné pendant x ne nous apporte pas de renseignement supplémentaire sur son futur.

Une variable aléatoire X, à valeurs dans les réels positifs a une *distribution exponentielle* si et seulement si il existe un réel$a$ positif tel que, pour tout $x \leq z$ positifs :

$$P(x \leq X \leq z) = \int_x^z ae^{-at}dt = e^{-ax} - e^{-az}$$

a est appelé le *taux de la distribution exponentielle* et la fonction ae^{-at} en est la *densité*.

Les moyenne (ou espérance) $E(X)$ et variance $V(X)$ sont :

$$E(X) = \int_0^\infty tae^{-at}dt = \left[e^{-at}\left(t + \frac{1}{a}\right)\right]_0^\infty = \frac{1}{a}$$

FIG. 8.24 – *Processus séquentiels finis synchronisés.*

$$\begin{aligned} V(X) &= E[(X - E(X))^2] = E[X^2 - 2XE(X) + E^2(X)] = E(X^2) - E^2(X) \\ &= \int_0^\infty t^2 a e^{-at} dt - \frac{1}{a^2} = \left[-e^{-at}\left(t^2 + 2\frac{t}{a} + \frac{1}{a^2}\right)\right]_0^\infty - \frac{1}{a^2} = \frac{1}{a^2} \end{aligned}$$

L'inverse de la moyenne de cette variable aléatoire est donc son taux au sens du taux de la variable aléatoire exponentielle définit ci-dessus. Cette dénomination a le sens physique suivant : si un phénomène répétitif a une durée exponentielle de taux a, alors la durée moyenne entre deux occurrences successives est $\frac{1}{a}$ et le nombre d'occurrences de ce phénomène par unité de temps (ce qu'on appelle aussi taux) est a. Sa variance est importante puisqu'égale au carré de sa moyenne. Il est facile de vérifier que si X est une variable aléatoire de distribution exponentielle (on dit couramment variable aléatoire exponentielle) $P(X = x) = 0$. On peut vérifier que cette loi est sans mémoire.

$$P(X \leq x + z \mid X \geq x) = \frac{P(x \leq X \leq x + z)}{P(X \geq x)} = \frac{e^{-ax} - e^{-a(z+x)}}{e^{-ax}} = 1 - e^{-az} = P(X \leq z)$$

Dans la suite *toutes les durées utilisées ont des distributions exponentielles.* La raison principale en est que cette hypothèse nous permet de travailler dans le cadre de Markov.

8.4.3 Valuation du graphe des exécutions

Une évaluation quantitative suppose que les durées des actions définies dans le modèle de comportement soient connues. Le modèle de comportement utilisé est le graphe des exécutions défini dans le paragraphe 8.2 de ce chapitre. On appellera graphe des exécutions valué, ce graphe auquel on ajoute des informations probabilistes sur la durée des actions. Les distributions des durées sont donc exponentielles et les paramètres de ces distributions sont connues. Ce sont des données du modèles. Certaines d'entre elles peuvent être prises comme paramètres et le modèle résolu pour différentes valeurs des paramètres.

Ceci nous amène à définir plus précisément ce que veut dire définir des durées pour les différentes actions de processus en parallèle. Considérons la mise en parallèle de deux processus et les graphes de deux types d'exécution, comme dans le paragraphe 8.2, figures 8.24 et 8.25.

La structure des graphes est identique mais les étiquettes sur les arcs ne sont plus des noms d'actions. Ce sont les taux de transition associés aux durées de ces actions ; autrement dit, la

FIG. 8.25 – *Graphe des exécutions avec communication par rendez-vous.*

transition est effectuée avec une durée aléatoire, selon le taux indiqué. Plus précisément, on a les hypothèses suivantes :

- Dès qu'une action est possible, elle démarre.
- Une action a une durée de distribution exponentielle et toutes ces variables aléatoires sont indépendantes. Sous ces hypothèses, deux variables aléatoires exponentielles et indépendantes sont égales avec la probabilité 0. En d'autres termes, deux actions ne peuvent se terminer en même temps (remarquer l'adéquation avec l'hypothèse de sérialisation du modèle asynchrone).
- Sur le graphe des exécutions, le changement d'état apparaît à la fin de l'action. Un changement d'état dans le graphe correspond à l'événement de fin d'action.
- Quand deux actions sont en concurrence (sur le schéma ci-dessus, les actions de taux a_3 et b_2), la première qui se termine provoque le changement d'état correspondant.
- Il faut remarquer l'importance de l'hypothèse *sans mémoire* de la loi exponentielle. Si dans le graphe ci-dessus, les actions se terminent dans l'ordre a_3 puis b_2, le chemin suivi dans le graphe est : (e_2, f_1), (e_3, f_1), (e_3, f_2). Le scénario d'exécution est précisément : dans l'état (e_2, f_1), après la synchronisation d'échange de taux c, les deux actions de taux a_3 et b_2 démarrent. Celle de taux a_3 se termine d'abord et ce processus se bloque en vue du prochain échange (de taux d). Dans l'état (e_3, f_1), l'action de taux b_2 *continue*, mais la loi exponentielle étant sans mémoire, la distribution de probabilité du temps restant est toujours exponentielle de taux b_2. C'est donc ce taux qui apparaît sur l'arc de continuation de l'action se terminant en (e_3, f_2). Une situation symétrique est modélisée sur le chemin (e_2, f_1), (e_3, f_1), (e_3, f_2).

Dans cet exemple, les durées des actions sont connues (en terme probabiliste), par contre les attentes provoquées par les rendez-vous d'échange, qui n'apparaissent pas comme des actions dans le modèle mais dont les durées sont bien réelles restent à évaluer et c'est ce que nous allons chercher à faire.

8.4.4 La théorie de Markov

Un graphe des exécutions permet de visualiser toutes les exécutions possibles de processus parallèles. C'est une vision statique et globale. Une vision dynamique est plus adaptée à l'évaluation des

FIG. 8.26 – *Graphe bouclé sur lui-même.*

performances : le temps s'écoule, et *un* cheminement du programme dans ce graphe des exécutions est représenté par une variable aléatoire $Y(t)$. $Y(t)$ est une variable aléatoire qui est égale, à tout instant t, à l'un des états de ce graphe d'exécution. C'est une variable aléatoire car ce cheminement n'est pas toujours le même, puisque justement le graphe des exécutions représente l'ensemble des cheminements possibles ; mais on peut calculer sa loi de probabilité, en calculant la probabilité du passage par un état donné. On admettra que sous les hypothèses précédemment émises, ($Y(t)$ est une *chaîne de Markov homogène.*

Une chaîne de Markov est une suite de variables aléatoires (l'indice de la suite est le temps, donc un réel positif et la valeur aléatoire prise par $Y(t)$ est l'état dans le graphe). Cette suite est telle que pour prédire en terme probabiliste son avenir (ce qui va se passer après l'instant t) l'état présent est suffisant, toute son histoire passée n'apporterait pas d'information supplémentaire. En termes mathématiques, pour tout t, r positifs et tout ensemble d'états E :

$$P\left(X(t+r) \in E \mid X(s), s \leq t\right) = P\left(X(t+r) \in E \mid X(t)\right)$$

Connaître tous les $X(s)$, $s \leq t$, n'apporte pas plus d'information que de connaître seulement $X(t)$. Cette chaîne est *homogène* parce que les durées des actions de base ne dépendent pas du temps. C'est leur enchaînement qui dépend du temps.

Notre objectif est de calculer des caractéristiques probabilistes de cette chaîne de Markov. Si ces caractéristiques à un instant t nous intéressent, on parle de probabilité *transitoire*. Si par contre, on s'intéresse à ces probabilités quand t tend vers l'infini, on parle de probabilité *stationnaire*. On note $Y(\infty)$ la variable aléatoire représentant l'état du programme dont l'évolution est représenté par le graphe des exécutions dans l'état stationnaire. Dans cet ouvrage, on se limitera à l'étude des probabilités stationnaires, avec l'hypothèse que le programme dure infiniment longtemps. Si tel n'est pas le cas, il suffit de compléter le programme et de le faire boucler sur lui-même infiniment afin d'obtenir des résultats sur un passage de boucle. Par exemple, pour l'exemple précédent, on étudiera le graphe de la figure 8.26.

Les exemples types Producteur-Consommateur, Lecteurs-Redacteurs, etc, et en général tous les problèmes de contrôle sont des programmes qui durent indéfiniment.

Etant donné un ordre arbitraire sur les états du graphe des exécutions, on note ces états de 1 à n. De façon bi-univoque une matrice de transition est associée à ce graphe, comme sur la figure ci dessous pour le graphe ci-dessus. Les taux de transitions de l'état i à l'état j sont sur la ligne i

et la colonne j de cette matrice, et les termes diagonaux sont ajustés de telle façon que la somme des éléments d'une ligne soit nulle. Notons T cette matrice qui est l'outil de base de calcul dans la théorie de Markov et s'appelle matrice de transition de la chaîne de Markov. Un vecteur de probabilité est représenté par un vecteur ligne. Par exemple la matrice de transition de la mise en parallèle des processus précédents est la suivante :

$$\begin{bmatrix} -a_1 & a_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -c & c & 0 & 0 & 0 & 0 \\ 0 & 0 & -a_3 - b_2 & a_3 & b_2 & 0 & 0 \\ 0 & 0 & 0 & -b_2 & 0 & b_2 & 0 \\ 0 & 0 & 0 & 0 & -a_3 & a_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & -d & d \\ b_4 & 0 & 0 & 0 & 0 & 0 & -b_4 \end{bmatrix}$$

L'ordre des états est : (e_0, f_0), (e_1, f_0), (e_2, f_1), (e_3, f_1), (e_2, f_2), (e_3, f_2), (e_4, f_3).

Nous allons utiliser le théorème central suivant, sous les hypothèses précédemment citées :

Si le graphe des exécutions est fortement connexe et si il existe une solution non nulle π (π est un vecteur ligne et π_i est sa i-ème composante) au système :

$$\begin{cases} \pi T & = & 0 \\ \sum_{i=1}^{n} \pi_i & = & 1 \end{cases}$$

Alors : $$\pi_i = P(Y(\infty) = i) = \lim_{t \to \infty} \frac{\mathbf{1}_{(Y(t)=i)}}{t}$$

La première égalité indique queπ_i est la probabilité stationnaire recherchée, et la deuxième égalité que cette probabilité stationnaire est bien égale à la proportion de temps que $Y(t)$ a passé dans l'état i. $\mathbf{1}(Y(t) = i)$ est la fonction caractéristique de l'état i qui vaut 1 quand $Y(t) = i$ et 0 sinon.
Si de plus le graphe des exécutions est fini, l'existence de cette solution est assurée. Pour le calcul, ce théorème est assortie de la propriété suivante :
Sous les hypothèses du théorème, et si la solution existe, cette solution est limite de la suite :

$$\pi^{(n)} = \pi^{(n-1)} + \pi^{(n-1)} T$$

quel que soit le vecteur de départ $\pi^{(0)}$ [1].

Exemple . Prenons l'exemple des pompiers (figure 8.27). Pour ne pas faire de calcul trop lourds, limitons le nombre des pompiers à deux. Les paramètres r, e et v représentent respectivement les taux des actions de remplissage, échange et vidage de chaque pompiers. Le graphe des exécutions munis des temporisations est représenté sur la figure 8.28.

La matrice de transition T est, sachant que nous numérotons de 1 à 4 les états VV, PV, VP et

1. L'indice supérieur est un indice de suite. $\pi_i^{(n)}$ désigne donc la i-ième composante du n-ième élément de la suite.

FIG. 8.27 – *Processus pompiers.*

FIG. 8.28 – *Graphe des exécutions associé à la figure .*

PP:

$$\begin{bmatrix} -r & r & 0 & 0 \\ 0 & -e & e & 0 \\ v & 0 & -r-v & r \\ 0 & v & 0 & -v \end{bmatrix}$$

La solution de $\pi T = 0$ sous la condition $\sum_{i=1}^{4} \pi_i = 1$:

$$\begin{aligned} -r\pi_1 + v\pi_3 &= 0 \\ r\pi_1 - e\pi_2 + v\pi_4 &= 0 \\ e\pi_2 - (r+v)\pi_3 &= 0 \\ r\pi_3 - v\pi_4 &= 0 \end{aligned}$$

Ces équations sont dépendantes et équivalentes après substitution à :

$$\pi_1 = \frac{v}{r}\pi_3, \quad \pi_2 = \frac{r+v}{e}\pi_3, \quad \pi_4 = \frac{r}{v}\pi_3$$

La condition de normalisation $\sum_{i=1}^{4} \pi_i = 1$ donne :

$$\pi_3 = \frac{1}{\frac{v}{r} + \frac{r+v}{e} + \frac{r}{v}}$$

L'ensemble des probabilités peut donc être calculé. La probabilité avec laquelle les pompiers ont tous les deux un seau vide est π_1. Sa variation peut être étudiée en fonction des valeurs des paramètres e, v et r ou bien sa valeur peut être calculée pour une instanciation de ces paramètres. Le taux

FIG. 8.29 – *Réseau de Petri pour trois pompiers.*

FIG. 8.30 – *Graphe des marquages associé*

de vidage des seaux est égal au produit de la vitesse de vidage par la probabilité que l'on soit dans un état où le vidage est possible soit : $(\pi_3 + \pi_4)v$. Le nombre de seaux pleins en moyenne est $2\pi_4 + \pi_3 + \pi_2$. De la même façon, on peut déduire divers indices de performance.

8.4.5 Réseaux de Petri Stochastiques

Un réseau de Petri Stochastique est un réseau de Petri valué où les durées des actions ont été définies par leur taux (le cadre de l'étude est toujours Markovien). Prenons l'exemple des trois pompiers (figure 8.29).

Le graphe des marquages (ou graphe des exécutions) peut-être calculé sous l'hypothèse d'asynchronisme comme il a été montré dans le paragraphe 8.2. Les actions y sont sérialisées et chaque arc du graphe correspond à l'exécution d'une action. Pour obtenir le graphe des exécutions valué, le taux de l'action est simplement associé à l'arc correspondant, et nous sommes ramenés au cadre étudié dans le paragraphe précédent. Le graphe des exécutions valué représente le comportement d'un processus de Markov homogène dont nous voulons étudier le comportement stationnaire. Les réseaux de Petri nous ont aidé à spécifier le modèle, mais le graphe des marquages valué sert à l'analyse. Bien sur, cette transformation réseau de Petri valué-graphe des marquages valué peut être aisément automatisé. La figure 8.30 montre le graphe des marquages valué du réseau de Petri des trois pompiers.

8.5 Bibliographie

[**Bra 82**] G.W. Brams, "Réseaux de Petri : Théorie et Pratique", Tomes 1 et 2. Masson Ed. 1982

[**JR 87**] C. Jard, M. Raynal, "Specification of Properties is Required to Verify Distributed Algorithms". Technical Report 651, INRIA, Centre IRISA, Rennes, February 1987.

[**PF 89**] G. Pujol, S. Fida. "Modèles de systèmes et de réseaux", Tomes 1 et 2, Eyrolles 1989.

Chapitre 9

Interblocage et Famine

9.1 Introduction

Parmi les propriétés qu'un système de processus parallèles doit respecter, deux sont élémentaires : le non blocage et l'absence de famine. Le système est bloqué si un processus au moins est en attente alors qu'aucune activité n'est observable dans le système. Cette mauvaise propriété résulte d'une mauvaise conception du système (sauf s'il s'agit d'une forme particulière qu'on cherche à donner à la terminaison de l'ensemble des processus). Un processus est en état d'attente infinie ou de famine si, alors qu'il attend qu'une condition soit satisfaite, il n'est jamais activé, bien que le système ne soit pas bloqué. Une erreur de conception peut provoquer cet état : la condition de franchissement d'un point de synchronisation n'est jamais satisfaite. Une seconde raison, plus subtile, peut intervenir : une condition d'autorisation est indéfiniment ignorée par le système parce qu'il privilégie de libérer les processus en attente sur une autre condition, toujours satisfaite quand la première l'est.

9.1.1 Interblocage

Dans ce paragraphe, nous montrons un exemple de programme pouvant mener à une situation d'interblocage, comment la détecter sur le graphe des exécutions et une solution correcte du problème.

L'exemple choisi est le suivant. Des processus clients (par exemple 2) ont besoin de ressources communes pour exécuter une tâche. Les ressources doivent être utilisées en exclusion mutuelle, et leur sont allouées par un serveur. Le cycle d'exécution d'un processus est de la forme : demander les ressources nécessaires une par une - travailler - relâcher les ressources.

Les ressources sont numérotées et au nombre de 2. L'un des processus noté P1 demande les ressources 2 et 1 dans cet ordre ; l'autre processus (P2) demande successivement les ressources 1 et 2 (figure 9.1).

On concoit aisément et le graphe des exécutions le prouve (figure 9.2), que ce système peut se bloquer : par exemple si le processus 1 ayant acquis la ressource 2, le processus 2 obtient la ressource

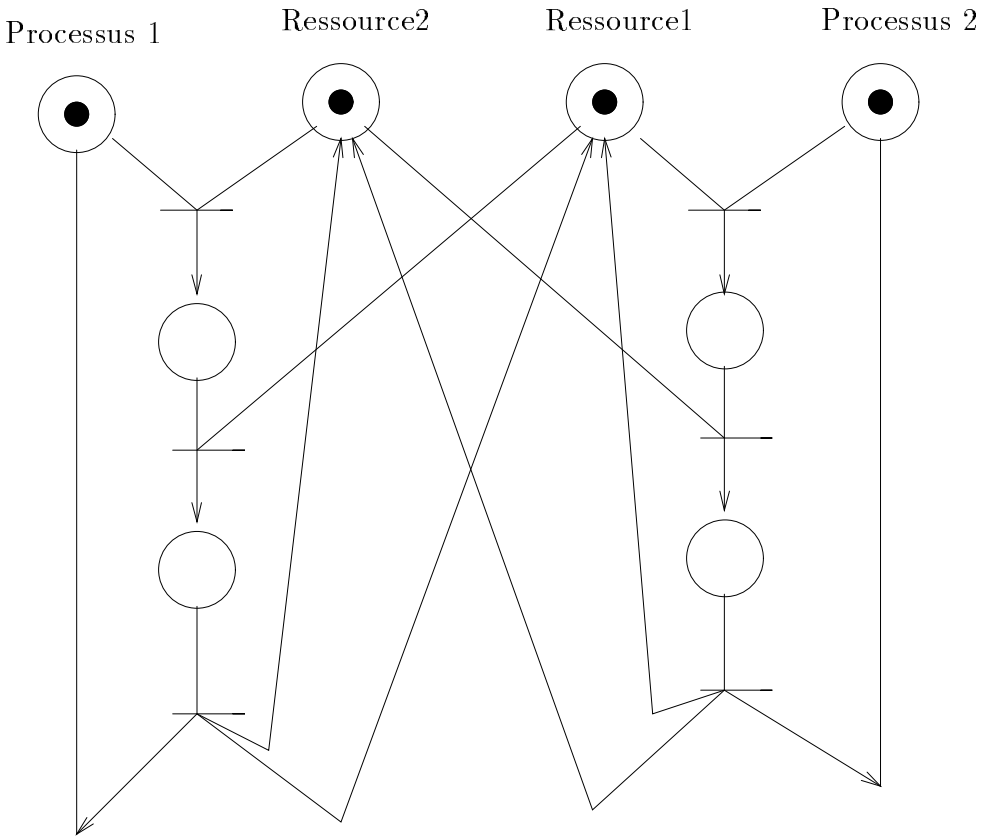


FIG. 9.1 – *Réseau de Petri modélisant l'allocateur de ressources*

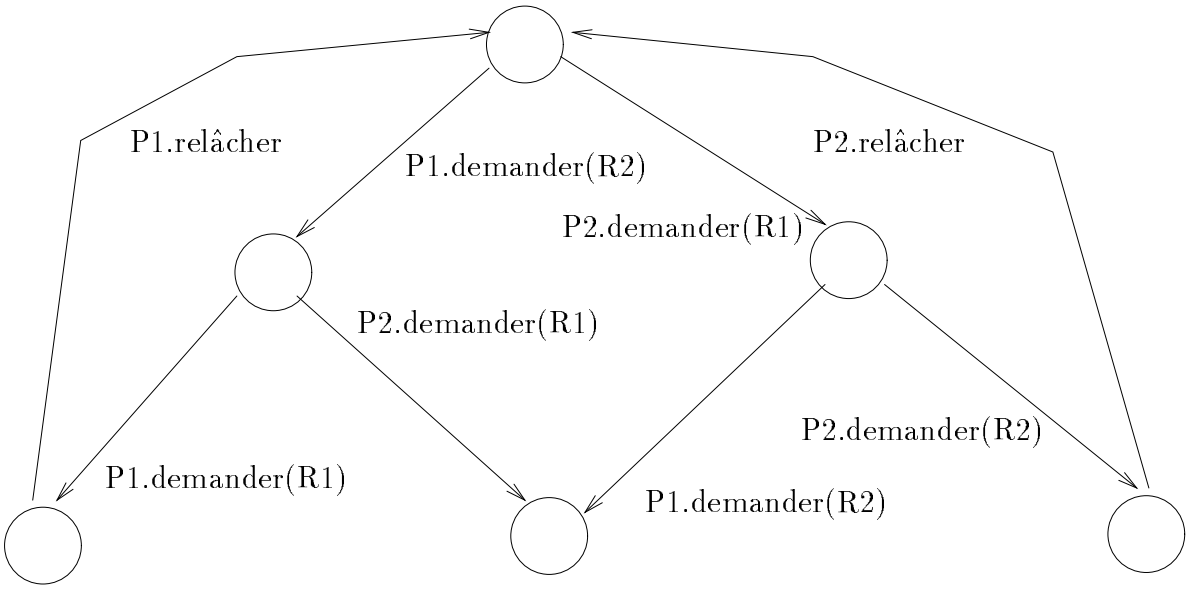


FIG. 9.2 – *Graphe des éxécutions montrant un interblocage*

1 avant le processus 1. Cette situation de blocage se traduit par un état puits dans le graphe des exécutions : aucun des processus ne peut plus évoluer, l'ensemble du système est définitivement coincé.

Une solution correcte pour ce problème consiste à imposer aux processus de demander les ressources dans l'ordre croissant de leurs numéros. Dans le cas de 2 ressources, il est clair que le premier processus ayant obtenu la ressource 1 pourra demander avec succès la ressource 2. Le nouveau graphe des exécutions ne comporte pas d'état puits (figure 9.3). Cette solution se généralise à un nombre quelconque de processus et de ressources.

On remarquera cependant que cette solution peut entraîner un état de famine. L'étude de la famine fait l'objet de la suite de ce chapitre.

9.1.2 Détection de la famine sur le graphe d'exécution.

S'il est possible de modéliser le comportement du programme comme nous l'avons vu au cours du chapitre précédent, cette situation peut être détectée. Reprenons le graphe des exécutions du sémaphore d'exclusion mutuelle introduit au chapitre 8 (figure 8.20).

Pour qu'il n'y ait pas de famine, il est souhaitable que chaque utilisateur puisse prendre toujours inévitablement la ressource, c'est-à-dire que l'état initial doit (devrait) vérifier les formules

toujours (*inévitable* $(x_4 > 0)$) et *toujours*(*inévitable* $(x_5 > 0)$).

Calculons l'ensemble $|$*toujours* (*inévitable* $(x_4 > 0)$)$|$ sur le graphe. L'ensemble des états du

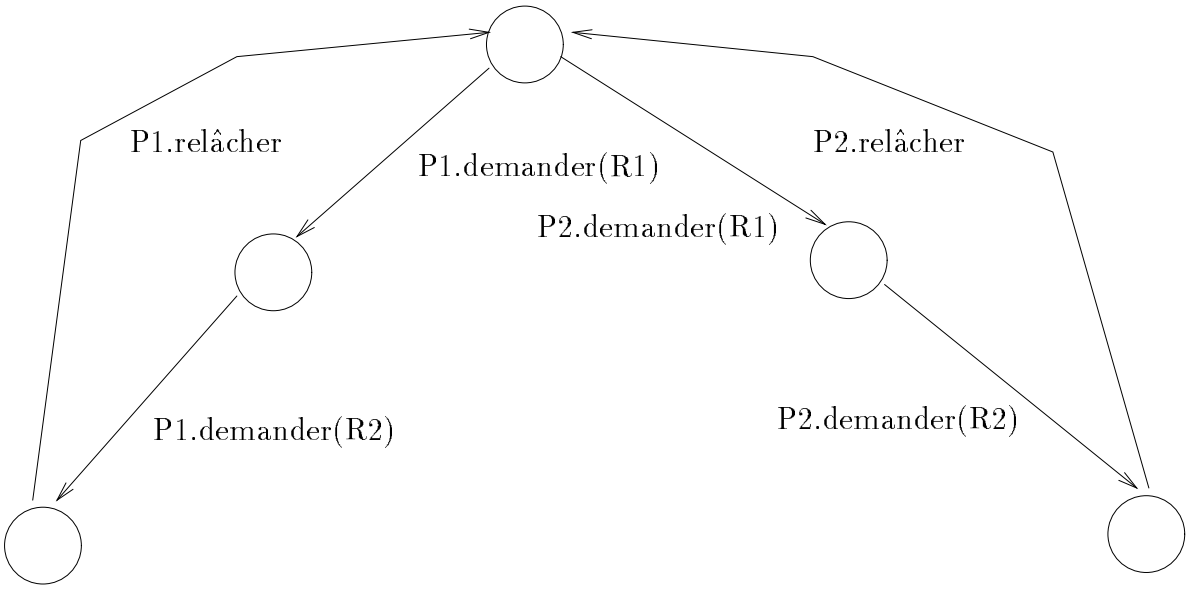


FIG. 9.3 – *Graphe des éxécutions de la solution correcte*

graphe est $S = \{a, b, c\}$.

- Soit g =*inévitable* $(x_4 > 0)$. Calculons $|g|$.

  Soit $h = x_4 > 0$; on a: $|h| = \{b\}$, par conséquent, la fonction G correspondante est définie par :

  $G(Y) = \{b\} \cup \tilde{pre}(Y) \cap pre(Y)$

  Calculons la suite Y_i définie par:

  $Y_0 = \emptyset$,

  et

  $Y_{i+1} = G(X_i)$

  - $Y_0 = \emptyset$; $pre\ (Y_0) = \emptyset$; $\tilde{pre}(Y_0) = \emptyset$;
  - $Y_1 = \{b\} \cup (\tilde{pre}(Y_0) \cap pre\ (Y_0)) = \{b\}$; $pre\ (Y_1) = \{a\}$; $\tilde{pre}(Y_1) = \emptyset$;
  - $Y_2 = \{b\} \cup (\tilde{pre}(Y_1) \cup pre(Y_1)) = Y_1$

  par conséquent:

  $|g| = \{b\}$

- Calcul de $|$ *toujours* $(g)|$. La fonction F correspondante est définie par:

  $F(X) = \{b\} \cap \tilde{pre}(X)$

  Calculons la suite $X_i$définie par:

  $X_0 = S$

  et

$X_{i+1} = F(X_i)$

- $X_0 = \{a, b, c\}$; $\widetilde{pre}(X_0) = \{a, b, c\}$;
- $X_1 = \{b\} \cap \widetilde{pre}(X_0) = \{b\}$; $\widetilde{pre}(X_1) = \emptyset$;
- $X_2 = \{b\} \cap \widetilde{pre}(X_1) = \emptyset$; $\widetilde{pre}(X_2) = \emptyset$;
- $X_3 = \{b\} \cap \widetilde{pre}(X_2) = X_2$

par conséquent :

$|toujours\ (inévitable\ (x_4 > 0))| = \emptyset$

On en déduit qu'une famine est possible.

Dans ce chapitre, nous allons étudier différentes manières de rendre équitable un système de processus concurrents. Dans ce but, nous choisissons un exemple type de synchronisation, celui des lecteurs-rédacteurs, avec deux modes de synchronisation : centralisé et distribué.

9.1.3 Rappel des spécifications du problème des lecteurs-rédacteurs.

Soient R processus rédacteurs, L processus lecteurs et une zone d'affichage unique (le tampon). Ces processus ont une activité autonome et de temps en temps échangent de l'information par cette zone d'affichage. On note NR et NL respectivement le nombre de rédacteurs en train d'afficher et de lecteurs en train de consulter à un instant donné. Les *contraintes de synchronisation* sont les suivantes :

- condition de lecture : $NR = 0$
- condition d'écriture : $NR = 0$ et $NL = 0$

9.2 Lecteurs/rédacteurs avec synchronisation centralisée

Une solution centralisée consiste à ajouter un processus serveur au système parallèle formé par les lecteurs et rédacteurs : le serveur doit garantir que les spécifications du problème sont garanties à tout instant.

9.2.1 Mise en défaut de l'approche naïve

Considérons le programme suivant, destiné à mettre en œuvre cette solution centralisée :

```
procedure lecteurs_redacteurs_2.1 is
    type contenu_affichage is ...
    task type redacteur;
```

```
task type lecteur;

task serveur_d_affichage is
   entry lire(v: out contenu_affichage);
   entry ecrire(v: in contenu_affichage);
end serveur_d_affichage;

task body lecteur is
   procedure consommer_lecture (p: in contenu_affichage) is
   begin
      ...
   end consommer_lecture;
   info: contenu_affichage;
begin
   loop
      serveur_d_affichage.lire(info);
      consommer_lecture(info);
   end loop;
end lecteur;

task body redacteur is
   procedure produire_information (p: out contenu_affichage) is
   begin
      ....
   end
   produire_information;

   info: contenu_affichage;
begin
   loop
      produire_information( info);
      serveur_d_affichage.ecrire(info);
   end loop;
end redacteur;

task body serveur_d_affichage is
   valeur: contenu_affichage:= valeur_initiale;
begin
   loop
      select
         accept lire(v: out contenu_affichage) do
            v:= valeur;
         end lire;
      or
         accept ecrire(v: in contenu_affichage) do
            valeur := v;
         end ecrire;
```

```
        end select;
      end loop;
    end serveur_d_affichage;

    population_lecteur : array (1..L) of lecteur ;
    population_redacteur : array (1..R) of redacteur;

end lecteurs_redacteurs ;
```

Cette programmation respecte bien la spécification du problème : mais elle présente deux inconvénients majeurs :

- d'une part, elle impose des conditions beaucoup plus fortes que les spécifications : en effet, avec cette solution, on a toujours :

  condition de lecture = = condition d'écriture : $NL + NR = 0$;

  la solution n'autorise donc pas le parallélisme maximal : elle interdit plusieurs lectures simultanées, ce qui est pourtant autorisé par la spécification.

- d'autre part, elle n'est pas équitable. Il se peut très bien qu'un même lecteur soit autorisé à lire de manière infinie : tous les autres processus (les rédacteurs et les autres lecteurs) se trouvent alors en état de famine.

Dans un premier temps, il est fondamental de respecter le plus précisément possible la spécification, en autorisant le maximum de parallélisme. Dans un deuxième temps, nous rendrons la solution équitable.

9.2.2 Respecter la spécification.

La spécification autorise plusieurs lectures simultanées : il est donc nécessaire que le serveur n'effectue pas l'action *lire* lui même, mais qu'il donne seulement au lecteur l'autorisation de lire. Pour garantir la spécification, il doit connaître cependant le nombre de lecteurs en action, ce qui implique que chaque lecteur l'informe en fin de lecture.

L'écriture, qui doit s'exécuter en exclusion mutuelle avec toute autre action de lecture ou d'écriture, pourrait être effectuée par le serveur (comme dans le précédent programme). Cependant, cette solution sera écartée car elle distingue le comportement des lecteurs de celui des rédacteurs au niveau du serveur. Pour des raisons d'homogénéité, il est donc naturel que le serveur accorde les autorisations d'entrée en action et soit informé des fins d'action par tous les processus, qu'ils soient lecteurs ou rédacteurs.

Cette solution montre nettement le caractère d'**arbitre** du serveur : en aucun cas il n'effectue les actions (comme dans la solution précédente). Il ne fait que gérer les modifications des variables d'état et garantir que la spécification du problème est bien respectée au cours de l'exécution.

Pour la programmation, nous introduisons deux procédures *read* et *write* pour la lecture et l'écriture. Ces deux procédures peuvent être appelées à n'importe quel moment par n'importe quelle tâche : le rôle de la tâche serveur est de garantir que leur utilisation est conforme aux spécifications.

```
procedure lecteurs_redacteurs_2.2 is
   type contenu_affichage is ...                    -- primitives de gestion de l'affichage
   valeur : contenu_affichage := valeur_initiale ;

   procedure read (v : out contenu_affichage) is
   begin
      v:= valeur;
   end read ;
   procedure write (v : in contenu_affichage) is
   begin
      valeur := v ;
   end write ;

   task type redacteur ;
   task type lecteur ;

   task serveur is
      entry autoriser_lecture ;
      entry fin_lecture ;
      entry autoriser_ecriture ;
      entry fin_ecriture ;
   end serveur ;

   task body lecteur is
      procedure consommer_lecture ... ;
      info : contenu_affichage ;
   begin
      loop
         serveur.autoriser_lecture ;
         read (info) ;
         serveur.fin_lecture ;
         consommer_lecture(info) ;
      end loop ;
   end lecteur ;

   task body redacteur is
      procedure produire_information ... ;
      info : contenu_affichage ;
   begin
      loop
         produire_information( info) ;
         serveur.autoriser_ecriture ;
```

```
        write (info) ;
        serveur.fin_ecriture ;
     end loop ;
  end redacteur ;

  task body serveur is
  NR, NL: integer:= 0 ;                    -- les variables d'etat gerees par le serveur
  begin
     loop
        select
           when (NR = 0)=>
              accept autoriser_lecture do
                 NL := NL + 1 ;
              end autoriser_lecture ;
        or
           accept fin_lecture do
              NL:=NL-1;
           end fin_lecture ;
        or
           when (NR + NL = 0) =>
              accept autoriser_ecriture do
                 NR := 1 ;
              end autoriser_ecriture ;
        or
           accept fin_ecriture do
              NR:=0;
           end fin_ecriture ;
        end select ;
     end loop ;
  end serveur ;

  population_lecteur ...                                  -- inchange

end lecteurs_redacteurs ;
```

Ce programme peut être facilement amélioré en temps de réponse du serveur : en effet, les modifications des variables d'état internes au serveur ne concernent pas les appelants. Il est donc inutile de bloquer l'appelant pendant la mise à jour de ces variables : le serveur effectue ces modifications après avoir débloqué l'appelant. L'appel sur un point d'entrée du serveur correspond alors à une communication atomique.

L'implantation de la tâche *serveur* peut donc être réécrite comme suit :

```
task body serveur is
   NR, NL ...
```

```
begin
    loop
      select
        when (NR = 0) => accept autoriser_lecture;
        NL := NL + 1;
      or
        accept fin_lecture;
        NL := NL - 1;
      or
        when (NR + NL = 0) => accept autoriser_ecriture;
        NR := 1;
      or
        accept fin_ecriture;
        NR := 0;
      end select;
    end loop;
end serveur;
```

9.2.3 Eliminer la famine

Le programme précédent respecte bien les spécifications, mais la famine reste possible : par exemple, une série de lectures peut se prolonger indéfiniment sans interruption même si une écriture est en attente. Cette famine provient du choix indéterministe qu'effectue le serveur avec l'instruction **select** pour l'accès au tableau d'affichage.

Les files d'attente sur un point d'entrée ADA sont gérées en FIFO, ce qui signifie que le choix fait lors d'un **accept** sur un point d'entrée est équitable, puisqu'il respecte le principe premier arrivé - premier servi. Par conséquent, il y a d'une part équité entre tous les lecteurs et d'autre part équité entre tous les rédacteurs.

Eliminer la famine revient donc à programmer une solution dans laquelle :

- une série de lectures ne peut se prolonger indéfiniment si une écriture est en attente
- une série d'écritures ne peut se prolonger indéfiniment si une lecture est en attente

Une première façon – qui n'est pas la plus pertinente comme nous le verrons peu après – pour programmer l'équité consiste à imposer que toutes les requêtes (qu'elles proviennent indifféremment des lecteurs ou des rédacteurs) soient traitées selon une politique "premier arrivé - premier servi". Cela peut être programmé en ADA en utilisant la sémantique de l'opération ACCEPT, et en ajoutant un nouveau processus avec un seul point d'entrée chargé d'acheminer dans l'ordre FIFO les requêtes vers le serveur.

Dans la solution ci-dessous, ce nouveau processus s'appelle *accès_serveur*, et possède un unique point d'entrée, *requête*. Le serveur de la solution précédente reste inchangé.

```
procedure lecteurs_redacteurs_2.3 is
                          -- on garde les primitives de gestion de l'affichage de la solution 2.2
                                                  -- les specifications des taches sont inchangees
   type mode_acces is (lecture, ecriture) ;
   task acces_serveur is
      entry requete (mode : in mode_acces) ;
   end acces_serveur

   task body acces_serveur is
   begin
      loop
         accept requete (mode : in mode_acces) do
            case mode is
               when lecture => serveur.autoriser_lecture ;
               when ecriture => serveur.autoriser_ecriture ;
            end case ;
         end requete ;
      end loop ;
   end acces_serveur ;

   task body lecteur is
      procedure consommer_lecture ... ;
      info : contenu_affichage ;
   begin
      loop
         acces_serveur.requete(lecture) ;
         read (info) ;
         serveur.fin_lecture ;
         consommer_lecture(info) ;
      end loop ;
   end lecteur ;

   task body redacteur is
      procedure produire_information ... ;
      info : contenu_affichage ;
   begin
      loop
         produire_information(info) ;
         acces_serveur.requete(ecriture) ;
         write (info) ;
         serveur.fin_ecriture ;
      end loop ;
   end redacteur ;

   task body serveur is ...                                  -- le meme que precedemment
```

```
    population_lecteur ...                                    -- inchange

end lecteurs_redacteurs;
```

Cette solution possède d'une part le défaut de décentraliser le contrôle du système entre deux processus (tous deux pouvant communiquer avec n'importe quel lecteur ou rédacteur), alors que l'on essayait de construire ici une solution centralisée. D'autre part, et cela est plus grave, les processus demandent l'autorisation de lire ou d'écrire à un processus (ici *accès_serveur)*, alors que la fin des lectures ou des écritures est adressée à un autre processus (ici *serveur*) : la solution perd donc en lisibilité. En outre, les programmes des lecteurs et rédacteurs ont été modifiés, et cette solution perd donc aussi en maintenabilité.

9.2.4 Programmer l'équité.

Une autre façon plus générale est de compléter la spécification de façon à intégrer l'équité. Le problème de la solution non équitable venait de la suite infinie possible de lectures ou d'écritures.

Ce problème peut être résolu en étudiant les deux cas qui entraînent la famine :

- lorsqu'un nouveau lecteur demande à lire alors que des lecteurs sont déjà en cours de lecture et que des rédacteurs sont en attente pour écrire.
- lorsqu'un nouveau rédacteur demande à écrire alors qu'un autre rédacteur vient de terminer d'écrire et qu'il existe des lecteurs en attente.

Une mauvaise politique dans l'une de ces deux situations peut en effet entraîner respectivement pour chacun des deux cas :

- qu'une série de lectures se prolonge indéfiniment sans interruption alors qu'un rédacteur est en attente d'écriture.
- qu'une série d'écritures se prolonge indéfiniment sans interruption alors qu'un lecteur est en attente de lecture.

Or ces deux situations peuvent être évitées respectivement de la façon suivante :

- avant d'autoriser une nouvelle lecture : si un rédacteur est en attente, il doit être prioritaire sur le nouveau lecteur. Cela peut être implanté par l'algorithme suivant : on attend que tous les lecteurs en cours finissent leur lecture, pour ensuite autoriser le rédacteur en attente. Lorsque ce rédacteur se termine, on laisse passer le lecteur que l'on a bloqué.

– lorsqu'on termine une écriture : si des lectures sont en attente, au moins une d'entre elles – la première – doit être prioritaire sur les autres écritures éventuellement en attente. Cela peut être implanté par l'algorithme suivant : lorsque le rédacteur a fini d'écrire, on débloque un lecteur en attente s'il en existe. Comme toutes les lectures peuvent s'exécuter en parallèle, il est intéressant de débloquer tous les lecteurs en attente de lecture : cela ne peut introduire une famine de rédacteur, car il ne peut pas y avoir plus de L lecteurs en attente.

Il est clair que d'autres variantes sont possibles pour traiter les deux cas de famine énoncés ci-dessus. Nous avons choisi ici de traiter ces deux cas de manière différente, de façon à présenter des techniques différentes de programmation : la famine des rédacteurs est évitée en modifiant l'algorithme d'entrée en section critique d'un lecteur, et la famine des lecteurs en modifiant l'algorithme de sortie de section critique d'un rédacteur. Les autres solutions (basées par exemple sur une modification de la sortie de section critique d'un lecteur au lieu de l'entrée) se programmeront donc en utilisant des schémas qui figurent dans le programme proposé ci-dessous.

De manière générale, ce type de solution peut être programmé en ADA en utilisant l'attente sélective (**select**) avec défaut (**else**) : dans une telle instruction, si aucune alternative **accept** gardée n'est ouverte, et si aucun appel n'est en attente sur l'une des alternatives **accept** ouvertes, alors c'est la partie **else** du **select** qui est exécuté.

Par rapport à la première solution non équitable, seule l'implantation du serveur est modifiée : la spécification du serveur reste inchangée, et les lecteurs et rédacteurs ne sont pas affectés par cette modification de la politique. la maintenabilité reste assurée. en outre, la solution reste centralisée, comme cela était désiré : la politique de gestion du parallélisme est totalement décrite par le serveur. cette solution apparaît donc clairement comme un raffinement de la solution centralisée non équitable.

Il est important de remarquer qu'un même point d'entrée peut être traité de différentes manières (plusieurs **accept** pour un même **entry**).

```
procedure lecteurs_redacteurs_2.4 is
                          -- primitives de gestion de l'affichage : idem versions precedentes
                  -- taches redacteur et lecteur, specification tache serveur : idem version 2.2

   task body serveur is
      NR, NL : integer:= 0;                          -- les variables d'etat gerees par le serveur
   begin
      loop
         select
            when (NR = 0) =>
               accept autoriser_lecture do
                  select
                     accept autoriser_ecriture do
                -- si un redacteur attend, on lui permet d'ecrire avant d'autoriser la lecture :
                                                -- donc il faut que les lecteurs en cours aient fini.
                        while (NL /=0 ) loop
```

```
                        accept fin_lecture
                        NL := NL - 1;
                     end loop;                    -- on peut maintenant autoriser l'ecriture.
                  end autoriser_ecriture
                  NR := 1;                                   -- le nouveau redacteur ecrit
                  accept fin_ecriture
                  NR := 0;
               else
                  null;                              -- il n'y a pas de redacteur en attente.
               end select;                          -- on peut maintenant autoriser la lecture.
            end autoriser_lecture;
            NL := NL + 1;
         or
            accept fin_lecture
            NL := NL - 1;
         or
            when (NR + NL = 0) => accept autoriser_ecriture
            NR := 1;
         or
            accept fin_ecriture
            NR := 0;
            declare IlResteDesLecteursEnAttente : boolean := true;
            begin                                    -- on autorise tous les lecteurs en attente.
               while (IlResteDesLecteursEnAttente ) loop
                  select
                     accept autoriser_lecture
                     NL := NL + 1;
                  else
                     IlResteDesLecteursEnAttente := false;
                  end select;
               end loop;
            end;
         end select;
      end loop;
   end serveur;

   population_lecteur ...                                          -- inchange

end lecteurs_redacteurs;
```

9.3 Lecteurs/rédacteurs avec synchronisation distribuée

9.3.1 Respecter la spécification.

Selon ce modèle, chaque processus doit prendre une décision en fonction d'une variable qui lui est locale. L'état du système est représenté par l'ensemble de ces variables locales augmenté d'une donnée qui propage une image retardée de l'état du système, en circulant de processus en processus. Cette donnée est appelée *jeton*. De manière à la faire circuler de processus en processus, et pour augmenter la clarté, de nouveaux processus, qui peuvent être vus comme des serveurs locaux, seront chargés de la modifier, en traitant les requêtes des processus à synchroniser.

A chaque processus, on associe donc un nouveau processus, jouant le rôle d'un serveur local, qui interprète et modifie l'état du système. Dans ce but, on définit un type de serveur : *serveur_local*, mais comme le comportement du serveur local est lié à la nature du processus (lecteur ou rédacteur) qui le sollicite, nous distinguons deux nouveaux types de serveurs locaux : *serveur_lecteur* et *serveur_rédacteur*. Les processus de ces types sont déclarés dans des tableaux (préfixés par *le_*) :

- *le_serveur_lecteur*(i) est le serveur associé au processus *le_lecteur*(i)
- *le_serveur_rédacteur*(i) est le serveur associé au processus *le_rédacteur*(i)

Pour l'implantation, nous avons choisi d'associer :

- d'une part un processus lecteur ou rédacteur et son serveur au niveau de la procédure générale *lecteurs_rédacteurs*. Cette procédure appelle successivement toutes les tâches lecteurs et rédacteurs qu'elle a engendrées, en leur précisant quel est leur serveur. Cela est implanté par un appel sur une entrée spécifique *TonServeurEst*.
- d'autre part un serveur local et son serveur voisin (le suivant dans le sens des aiguilles d'une montre sur le dessin), à qui il doit communiquer le jeton, au niveau de la procédure générale *lecteurs_rédacteurs*. Cette procédure appelle successivement toutes les tâches serveurs en leur précisant quel est leur voisin. Cela est implanté par un appel sur une entrée spécifique *TonVoisinEst*.

L'état du système circule entre processus serveurs locaux, de voisin à voisin, grâce à un point d'entrée : *Jeton*(*nombre_lecteurs, nombre_rédacteurs* : **in** *integer*).

Ce premier programme est obtenu par construction systématique à partir de la spécification du problème. On pourra se référer au premier serveur centralisé correct du précédent paragraphe : le programme des tâches lecteurs et rédacteurs est notamment inchangé, tandis que le corps des serveurs est relativement affecté bien que leur structure générale reste la même.

```
procedure lecteurs_redacteurs_3.1 is
                        -- primitives de gestion de l'affichage : idem versions centralisees
```

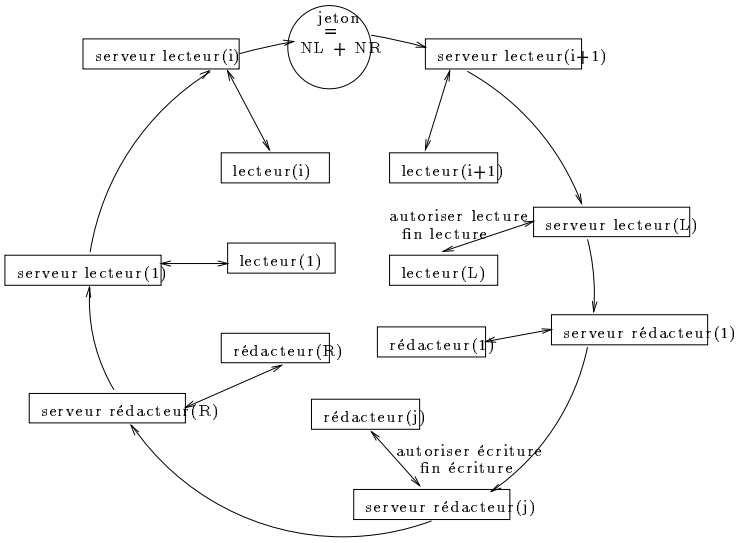


FIG. 9.4 – *Synchronisation distribuee avec Jeton circulant*

```
task type redacteur is
   entry TonServeurEst(mon_serveur : in serveur_local) ;
end redacteur ;

task type lecteur is
   entry TonServeurEst(mon_serveur : in serveur_local) ;
end lecteur ;

task type serveur_local is
   entry TonVoisinEst(mon_voisin : in serveur_local) ;
   entry jeton(nblect, nbred : in integer ) ;
   entry autoriser_lecture ;
   entry fin_lecture ;
   entry autoriser_ecriture ;
   entry fin_ecriture ;
end serveur_local

task body lecteur is
   procedure consommer_lecture ...;
   info : contenu_affichage ;
   serveur : access serveur_local ;
begin
   accept TonServeurEst(mon_serveur : in serveur_local) do
      serveur.all := mon_serveur ;
   end TonServeurEst ;
   loop
      serveur.autoriser_lecture ;
      read (info) ;
```

```
      serveur.fin_lecture ;
      consommer_lecture(info) ;
   end loop ;
end lecteur ;

task body redacteur is
   procedure produire_information ... ;
   info : contenu_affichage ;
   serveur : access serveur_local ;
begin
   accept TonServeurEst(mon_serveur : in serveur_local) do
      serveur.all := mon_serveur ;
   end TonServeurEst ;
   loop
      produire_information(info) ;
      serveur.autoriser_ecriture ;
      write (info) ;
      serveur.fin_ecriture ;
   end loop ;
end redacteur ;

task body serveur_local is
   NL; NR: integer
   voisin : access serveur_local ;
begin
   accept TonVoisinEst(mon_voisin : in serveur_local) do
      voisin.all := mon_voisin ;
   end TonVoisinEst ;
   loop
      select
         accept jeton( nblect, nbred : in integer ) do
            NL := nblect ; NR := nbred ;                          -- on reccoit le jeton
         end jeton ;
         voisin.jeton(NL, NR) ;                                   -- on passe le jeton au voisin
      or
         accept autoriser_lecture do
            accept jeton(nblect, nbred : in integer ) do
               NL := nblect ; NR := nbred ;                       -- on reccoit le jeton
            end jeton ;
            while ( NR /= 0) loop                  -- on attend que la fin des ecritures
               voisin.jeton(NL, NR) ;                           -- on passe le jeton au voisin
               accept jeton( nblect, nbred : in integer ) do
                  NL := nblect ; NR := nbred ;                   -- on reccoit le jeton
               end jeton ;
            end loop ; -- NR = 0 : on peut autoriser la lecture, mais on
            voisin.jeton(NL+ 1, 0) ; -- revoit le jeton                          d'abord
         end autoriser_lecture
```

```
            or
               accept fin_lecture do
                  accept jeton(nblect, nbred : in integer ) do
                     NL := nblect ; NR := nbred ;                    -- on reccoit le jeton
                  end jeton ;
                  voisin.jeton(NL- 1, NR) ;                  -- on passe le jeton au voisin
               end fin_lecture ;
            or
               accept autoriser_ecriture do
                  accept jeton( nblect, nbred : in integer ) do
                     NL := nblect ; NR := nbred ;                    -- on reccoit le jeton
                  end jeton ;              -- on attend que la fin des lectures et des ecritures
                  while (NL + NR /= 0) loop
                     voisin.jeton(NL, NR);                   -- on passe le jeton au voisin
                     accept jeton(nblect, nbred : in integer ) do
                        NL := nblect ; NR := nbred ;                 -- on reccoit le jeton
                     end jeton ;
                  end loop ;
               end autoriser_ecriture ;        -- on peut autoriser l'ecriture : mais on debloque
                                               -- d'abord le redacteur, car il bloque tout le monde
               voisin.jeton(0, 1) ;
            or
               accept fin_ecriture do
                  accept jeton( nblect, nbred : in integer ) do
                     null ;             -- on reccoit le jeton mais on sait deja qu'il vaut (0,1) !!
                  end jeton ;
                  voisin.jeton(0, 0) ;                       -- on passe le jeton au voisin
               end fin_ecriture ;
            end select ;
         end loop ;
      end serveur_local ;

le_lecteur : array (1..L) of lecteur ;
le_serveur_lecteur : array (1..L) of serveur_local ;
le_redacteur : array (1..R) of redacteur ;
le_serveur_redacteur : array (1..R) of serveur_local ;

begin
      for i in 1.. L loop
         le_lecteur(i).TonServeurEst(le_serveur_lecteur(i)) ;
         if (i = L) then
            le_serveur_lecteur(L).TonVoisinEst(le_serveur_redacteur(1)) ;
         else le_serveur_lecteur(i).TonVoisinEst(le_serveur_lecteur(i+1)) ;
         end if;
      end loop;
      for i in 1..R loop
         le_redacteur(i).TonServeurEst(le_serveur_redacteur(i));
```

```
        if (i = R) then
           le_serveur_redacteur(R).TonVoisinEst(le_serveur_lecteur(1)) ;
        else le_serveur_redacteur(i).TonVoisinEst(le_serveur_redacteur(i+1)) ;
        end if;
     end loop;

     le_serveur_redacteur(1).jeton (0, 0 );                    -- on donne le depart du jeton.

end lecteurs_redacteurs;
```

On peut en fait simplifier le parcours du jeton, en constatant que lorsqu'un rédacteur écrit, il bloque par définition tous les autres processus. ainsi, le jeton peut rester bloqué chez le rédacteur tant qu'il n'a pas terminé son écriture.

le serveur local peut donc être écrit plus efficacement comme suit :

le typage de l'entrée jeton devient :

entry *jeton* (*nblect*: **in** *integer*) ;

La modification du corps de la tâche *serveur_local* est laissée en exercice.

9.3.2 Eliminer la famine - Programmer l'équité.

Le programme précédent respecte bien les spécifications, mais la famine reste possible.

Cette famine (provenant du **select** dans les serveurs locaux décrits précédemment) prend les deux formes vues précédemment avec la synchronisation centralisée :

- une série de lectures peut se prolonger indéfiniment même si une écriture est en attente.
- une série d'écritures peut se prolonger indéfiniment même si une lecture est en attente.

Mais en plus, une famine plus sournoise provient de l'introduction du jeton dans les serveurs : en effet, les serveurs locaux peuvent très bien ne faire que circuler le jeton (en acceptant toujours la première alternative du **select**), et en ignorant par conséquent la requête provenant des processus qu'ils contrôlent.

Ce type de famine est improbable en pratique : en effet, si un processus a émis une requête à son serveur, il existe nécessairement une durée non nulle pendant laquelle le serveur sera en attente sur **select**, avec l'alternative de traitement de requête acceptable, et l'alternative de réception du jeton ouverte mais non acceptable. Cette durée est due au fait que lorsqu'un serveur s a fini d'émettre le jeton à son serveur voisin, le serveur voisin effectue d'une part certains calculs avant de le transmettre, et d'autre part, le délai de transmission du jeton entre tous les autres processus serveurs, avant le retour sur le serveur S, n'est pas nul.

Cependant, de manière théorique, aucune hypothèse ne peut être faite sur l'implantation du **select** et il est préférable de se garantir par une programmation robuste qui peut être obtenue en utilisant, ici aussi, une alternative **else** dans le **select**.

De manière générale, un serveur local a la forme suivante :

```
task body serveur_local is
                                                    -- declarations locales...
begin
                                                    -- initialisation...
    loop
       select
          accept jeton(etat: in etat) do ...
          end jeton
       or
          accept requete(req: in requete) do ...
          end requete
       end select;
    end loop;
end serveur_local;
```

avec éventuellement le traitement de plusieurs requêtes distinguées (comme c'est le cas dans le serveur local pour les lecteurs-rédacteurs).

pour éviter que l'alternative de réception du jeton soit éternellement prise, il suffit de donner priorité dans le select à l'acceptation de la requête :

```
select
    accept requete(req: in requete) do ...
    endrequete
else
    accept jeton(etat: in etat) do ...
    end jeton
end select;
```

L'inconvénient de ce programme est qu'il limite le parallélisme : si lorsqu'on entre dans le select aucune requête n'est parvenue, alors on se bloque sur l'attente du jeton, même si une nouvelle requête que l'on pourrait traiter arrive avant que l'alternative d'acceptation du jeton ne soit acceptable. une solution plus pertinente consiste donc à intégrer dans le else un nouveau select entre l'acceptation du jeton et celle d'une requête, de la façon suivante :

```
select
```

```
        accept requete(req : in requete) do ...
        end requete ; ...
else
        select
            accept requete(req : in requete) do ...
            end requete
            accept jeton(etat : in etat) do ...
            end jeton ; ...
        end select ;
end select ;
```

La famine dûe à l'introduction du jeton peut donc être facilement et systématiquement enlevée.

Une fois cette famine évitée, l'introduction du jeton circulant et la définition d'une politique équitable entre le traitement du jeton et celui des requêtes locales, entraîne la définition d'une politique équitable pour l'allocation des ressources entre les processus clients. Ainsi, dans l'exemple des lecteurs-rédacteurs : un rédacteur ne peut écrire que s'il possède le jeton. Or, le jeton ne peut passer directement de rédacteur en rédacteur qu'au plus (*R-1*) fois : ensuite, il ira nécessairement vers un lecteur, et, par suite, une série d'écritures ne peut se prolonger indéfiniment sans interruption par une lecture. La famine des lecteurs a donc été évitée grâce à l'introduction du jeton.

Il est clair qu'on peut varier à l'infini les politiques : nous recommandons simplement au lecteur de veiller, lorsqu'il introduit une nouvelle politique pour lever tel défaut, de ne pas introduire un nouveau défaut. Dans de nombreux cas, la simplicité sera la meilleure règle, et dans tous les cas on ne s'abstiendra pas d'utiliser les modèles de validation et d'évaluation que nous avons présenté au chapitre 8.

Annexe A

Notions sur le langage ADA séquentiel

Dans cette annexe, nous présentons brièvement les notions simples de ADA séquentiel. Le but de cet annexe est de faciliter la lecture du document pour un lecteur ne connaissant pas ce langage, mais ne constitue pas un cours sur ce langage.

Le langage ADA intègre de nombreux concepts avancés de la programmation : des outils pour la programmation modulaire, une distinction (quelquefois un peu lourde) entre le mode d'emploi d'un objet (appelé *spécification*) et son implantation (appelée *corps*), les traitements des exceptions, des mécanismes de généricité et la notion de tâches. Une tâche en ADA est un ensemble d'instructions que doit exécuter un processus. Un programme ADA séquentiel ne comporte qu'une tâche alors qu'un programme ADA parallèle en comporte plusieurs. L'exécution d'un programme ADA parallèle demande autant de processus qu'il y a de tâches définies dans le programme. Le lecteur devra se référer à des ouvrages comme [1] et [2] pour avoir une vue complète du langage. Ce qui suit n'est qu'un bref balayage de la syntaxe élémentaire. Les notions de *paquetages*, de *généricité*, de *surcharge* et d'*exceptions*, par exemple, ne seront pas présentées dans ce chapitre, mais pourront être utilisées à titre d'exemples dans certains programmes.

Nous décrivons ici des éléments de ADA séquentiel. Ces notions sont communes à Pascal, C, etc. Les mots écrits en italique sont les mots réservés du langage.

A.1 Les déclarations

Toute unité de programme peut comporter des déclarations de type, de constante ou de variable. Par exemple :

```
type couleur is ( bleu, blanc, rouge, vert, jaune) ;
type table is array ( 1 .. 100) of integer;
type tmatrice is array ( 1 .. 100, 1..500) of float;
```

```
type pointeur is access couleur;
type data is record
     mois: integer range 1..12;
     an: integer;
end record;
drapeau: boolean:= false;
x,y,toto: float:= 0;
epsilon: constant adafloat:= 0.1;
```

On a défini ci-dessus cinq types : un type énuméré, deux types tableau, un type pointeur sur un élément de type couleur et un type enregistrement ; `drapeau` est une variable de type booléen initialisée à faux et `epsilon` est une constante de type flottant. Les types **integer**, **boolean** et **character** sont des types prédéfinis du langage. `table`(20..30) désigne un sous tableau de `table`. On peut aussi définir des types de tableaux appelés non contraints :

```
type vecteur is array ( integer range <> ) of float;
v: vecteur(1..4):= (1.0, 2.0, 3.5, 4.2);
n: integer:= 4;
w: vecteur(1..n);
```

La déclaration de type spécifie le type de l'indice mais pas la taille du tableau. En revanche, cette taille devra être spécifiée lors de la déclaration de l'objet. Par exemple les déclarations ci-dessus fixent la taille à 4 : la première en donnant une valeur initiale, la deuxième en indiquant l'intervalle d'indexage.

A.2 Les expressions

Pour ces types prédéfinis, il existe des opérateurs prédéfinis : logiques (*and* , *or* , *not xor*), de relation (=, /=, >, >=, etc), additifs (+, -, &), multiplicatifs (*, /, *mod* , *rem*), autres (**, *abs* , etc). A partir de ces opérateurs on peut construire des expressions comme `epsilon*`(3 + `x`).

A.3 Les créations dynamiques d'objets

Ayant déclaré une variable de type accès (on dirait pointeur en Pascal), la création dynamique de structures de données peut se faire comme suit :

```
type acces_entier is access integer ;
a_entier1 : acces_entier ;
 i : integer := 0;
 a_entier1 := new integer ;
a_entier1.all := i;
```

L'appel à l'allocateur dynamique se fait par **new** suivi du type de l'objet accédé. Avant l'affectation, `a_entier1` avait la valeur prédéfinie **null**. L'objet accédé est désigné par l'identificateur de pointeur suivi de l'extension **all**. Si l'objet accédé est un enregistrement, la sélection d'un champ de cet enregistrement se fait par une extension composée simplement d'un point suivi de l'identificateur de champ. On peut construire des structures chaînées de la manière suivante :

```
type cellule ;
type  lien is access  cellule ;
type  cellule is record
     cl : integer  ;
     suivant : lien ;
end record
a_liste : lien ;
a_liste : = new  cellule ;
a_liste. cl : = 6 ;
a_liste.suivant : = new  cellule ;
...
```

La dernière instruction ajoute une cellule chaînée à la précédente.

A.4 Les instructions

L'instruction de base est l'affectation : `toto := x + y ;`

L'instruction vide se traduit par : *null* ;

Les instructions composées sont la conditionnelle, le choix multiple et la boucle. Des exemples d'utilisation sont donnés ci-dessous :

1. la conditionnelle :

```
if (note < 12) then put ( "est septembrise"); end if;
```

   ou bien

```
if ( note < 12 ) then
     put ( "est septembrise") ;
else
     put ( "part en vacances") ;
end if;
```

Il existe une forme étendue de la conditionnelle avec autant de **elsif** que nécessaire :

if ... then ... elsif ... then ... else ... end if

2. le choix multiple : avec `couleur_table` variable de type `couleur` :

```
case couleur_table is
    when bleue  => x := 0;
    when rouge  => x := 0;
    when others => null;
end case;
```

Le case peut se trouver sans la clause **others**. Dans ce cas, il faut s'assurer que l'une des alternatives **when** est vraie dans tous les cas sinon une erreur à l'exécution peut intervenir.

3. la boucle : il en existe deux types. Pour l'une le test d'arrêt se fait à partir d'une expression logique. L'autre utilise un compteur et un intervalle de variation.

```
while (x > 0) loop
    x := x - 1;
end loop;

for i in 1..10 loop
    x : x * x;
end loop;
```

Il est possible de construire des boucles infinies entre les mots clés **loop** et **end loop** (des exemples sont donnés dans la suite).

A.5 Les sous programmes

Les sous programmes sont l'outil de base de la modularité. Il en existe deux types : **procedure**et **function**. La déclaration d'un sous programme peut se faire en deux étapes : la déclaration de la spécification du sous programme (i.e. son mode d'emploi) et la déclaration de son corps (i.e. son implantation).
Voici quelques exemples de déclarations de spécification (on parle aussi d'en-tête de procédure) :

```
procedure aleatoire;
procedure stocker (information : in ada integer; tampon : in out tab_entier);
function factoriel (a : in integer ) return integer ;
function mult (a, b : in matrice) return matrice;
```

Dans ces déclarations, on distingue trois modes de passage de paramètres :

- **in**: le paramètre formel est une constante à l'intérieur de la procédure. Il ne peut pas être modifié par le sous programme. Ce mode correspond approximativement au passage par valeur en Pascal.
- **out**: le paramètre formel est une variable qui ne peut pas être lue par le sous-programme, mais seulement écrite (passage de paramètre par résultat).
- **inout**: le paramètre formel est une variable qui peut être lue et modifiée (passage de paramètre par donnée-résultat).

Lorsqu'une fonction est déclarée, le type du résultat suit le mot clé **return**. Les fonctions ne peuvent avoir que des paramètres **in**.

Le corps d'un sous programme reprend intégralement la partie spécification (ce qui permet d'omettre la déclaration de spécification quand elle n'est pas indispensable), puis contient d'éventuelles déclarations et enfin la partie impérative délimitée par les mots clés **begin... end**. Par exemple, si `cas` est le type énuméré (`indefini, impossible, defini`) :

```
function SorteEquation ( a, b: in float ) return cas is
     -- Ici, il peut y avoir des declarations
begin
     if (a=0) and (b=0) then
        return indefini ;
     elsif (a=0) then
        return impossible ;
     else
        return defini ;
     end if;
end SorteEquation ;

procedure ResoudreEquation (a,b: in float; x: out float ) is
begin
     x:= b / a;
end ResoudreEquation ;
```

L'appel du sous-programme est composé du nom du sous-programme suivi éventuellement de la liste des paramètres effectifs. Il existe plusieurs syntaxes possibles, dont la suivante, dite "positionnelle" : la correspondance paramètre effectif - paramètre formel se fait suivant la position dans la liste. Un appel de procédure est une instruction alors que l'appel d'une fonction rend un résultat qui doit être utilisé dans une instruction.
Exemple : si `tableau_de_points` est une variable de type tableau d'entiers `stocker` la procédure définie ci-dessus, nous pouvons écrire :

```
stocker ( 100, tableau-de-points ) ;
```

En réutilisant la fonction `SorteEquation` et la procédure `ResoudreEquation` (à noter l'introduction d'un bloc structuré nommé `bloc`, avec déclaration de variable locale au bloc) :

```
bloc :
    declare
       solution : float
    begin
       case SorteEquation (10, 15) is
          when impossible => null;
          when defini =>
             put ( "la solution est" );
             ResoudreEquation ( 10, 15, solution );
             put ( solution );
          when indefini => null;
       end case;
    end bloc;
```

Ceci conclut notre brève présentation des outils simples du langage ADA séquentiel. Pour terminer, nous présentons un exemple de programme avec un programme principal :

```
with TEXT_IO; use TEXT_IO;
procedure calculateur is
    package FLOTTANTS_IO is new FLOAT_IO(float ); use FLOTTANTS_IO;
    a, b, result : float;
    opr : character;
begin
    while not end_of_file(standard_input) loop
       get (a); get (opr); get (b);
       case opr is
          when '+' => result := a + b; put ( result );
          when '-' => result := a - b; put ( result );
          when '*' => result := a * b; put ( result );
          when '/' => result := a / b; put ( result );
          when others => put ( " Erreur : operateur illegal ");
       end case;
    end loop;
end calculateur;
```

Ici le programme principal est la procédure *calculateur*. Elle utilise une librairie standard d'entrée-sortie TEXT_IO pour des fichiers texte. Les commentaires commencent par un double tiret et se terminent en fin de ligne. end_of_file et standard_input sont des fonctions du module TEXT_IO et retournent respectivement un booléen et le nom du fichier standard d'entrée-sortie. Les caractères s'expriment entre simples apostrophes.