Gestion dynamique d'une architecture cellulaire massivement parallèle

Youssef Latrous

▶ To cite this version:

Youssef Latrous. Gestion dynamique d'une architecture cellulaire massivement parallèle. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1995. Français. NNT : . tel-00005051

HAL Id: tel-00005051
https://theses.hal.science/tel-00005051

Submitted on 24 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Présentée par

Youssef LATROUS

pour obtenir le titre de **DOCTEUR**

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(Arrêté ministériel du 30 Mars 1992)

(Spécialité: **Informatique**)

GESTION DYNAMIQUE D'UNE ARCHITECTURE CELLULAIRE MASSIVEMENT PARALLÈLE

Date de soutenance : 12 Janvier 1995

Composition du jury:

Messieurs	Yves	CHIARAMELLA	Président
	Guy	MAZARE	Examinateurs
	Jean-Louis	ROCH	
	Jean-Luc	BASILLE	Rapporteurs
	Bernard	TOURSEL	

Thèse préparée au sein du **L**aboratoire de **G**énie **I**nformatique

Dédicaces

À mon épouse Safia,

À ma fille Meriem ,

À mes parents et beaux-parents,

À mes frères et amis,

À la mémoire d'un ami qui m'est très cher
le Pr Djillali BELKHENCHIR.

Remerciements

Je tiens à remercier Monsieur Yves CHIARAMELLA, Professeur à l'UJF et Directeur du LGI, pour l'honneur qu'il me fait en présidant le jury de cette thèse.

Je remercie Monsieur Guy MAZARE, Professeur à l'ENSIMAG et Directeur de cette Ecole, pour la confiance qu'il m'a témoignée et les conseils qu'il m'a prodigués qui m'ont permis de mener à terme ces travaux.

Je remercie très sincèrement Monsieur Jean-Luc BASILLE, Professeur à lENSEEIHT de l'Université de Toulouse 3 et responsable d'équipe à l'IRIT, ainsi que Monsieur BernardTOURSEL, Professeur à l'Université de Lille 1 et responsable d'équipe au LIFL, pour avoir accepté d'être rapporteurs de cette thèse.

Mes remerciements vont également à Monsieur Jean-LouisROCH, Maître de Conférence à l'ENSIMAG, pour l'intérêt qu'il a toujours porté à mes travaux et pour l'honneur qu'il me fait en participant à ce jury.

Je tiens à remercier tout particulièrement Monsieur MichelDELAUNAY, pour le temps qu'il m'a consacré tout au long de cette thèse, aussi bien pour la mise en forme de ce manuscrit que pour son assistance aux pires moments de la programmation...

Je n'oublierai pas tous mes collègues du laboratoire, sans qui ce travail n'aurait pas pu être réalisé dans d'aussi bonnes conditions. Qu'ils trouvent dans ces quelques lignes toute ma sympathie.

Résumé

Le modèle d'acteurs est un modèle de calcul concurrent qui semble bien adapté au modèle d'exécution des architectures massivement parallèles. Nous étudions son adéquation à une machine massivement parallèle à grain fin étudiée dans notre groupe; nous dégageons les mécanismes de base à intégrer au niveau de l'unité de routage de cette machine. Pour des modèles de programmation dynamiques de ce type, il est nécessaire d'établir une stratégie d'allocation dynamique de processus. Nous proposons et évaluons une idée originale pour l'allocation dynamique de processus dans une architecture massivement parallèle.

Dans le cas d'une machine à grain fin comme la notre, il est primordial qu'une telle fonction n'induise qu'un minimum de charge supplémentaire en termes de communications. Nous présentons une fonction intégrée de recherche de processeurs libres pour l'exécution des processus dynamiques. L'évaluation des principaux choix architecturaux a pu être menée grâce à un simulateur développé dans le cadre de ce travail. Les résultats obtenus, qui démontrent l'intérêt de l'approche, sont présentés dans cette thèse.

Mots clefs architecture massivement parallèle, parallélisme à grain fin, système intégré, modèle acteur, régulation dynamique de charge, réseaud'interconnexion, gestion des ressources.

Abstract

The actor model which is a concurrent computing model that seems to be well suited to the massively parallel architecture execution model. We investigate its adequacy to a massively parallel fine grain machine; we derive the basic mechanisms to be integrated at the routing unit level of this machine. For dynamic programming models of this kind, it is necessary to establish a strategy for dynamic process allocation. We propose and evaluate an original approach for dynamic process allocation in a massively parallel architecture.

In the case of a fine grain machine as ours, it is essential that such a function implies a minimum overhead in terms of communications. We introduce an integrated hardware function that takes in charge the search for free processors that will execute dynamic processes. The evaluation of the main architectural choices had been carried out with the help of a simulator developed as a part of this work. The results obtained in this thesis show the usefullness of this approach.

Key words massively parallel architecture, fine grain parallelism, integrated system, actor model, dynamic load balancing, interconnectionnetwork, resources management.

Table des Matières

Introduction ... 3

1. **Contexte de travail et objectifs** ... 9
   1.1. Le projet "Réseau Cellulaire Asynchrone Intégré" ... 9
      1.1.1. Motivations et historique du projet ... 9
         1.1.1.1 Classification des machines parallèles ... 11
         1.1.1.2 Les machines massivement parallèles ... 11
            1.1.1.2.1 Le GAPP ... 12
            1.1.1.2.2 La Connection Machine 2 ... 13
            1.1.1.2.3 La Machine MEGA ... 14
            1.1.1.2.4 La CM5 ... 15
            1.1.1.2.5 La SP1 ... 16
            1.1.1.2.6 La Paragon ... 17
         1.1.1.3 Le projet R.C.A.I ... 20
         1.1.1.4 Processeur dédié ... 21
         1.1.1.5 Processeur programmable ... 22
      1.1.2. Motivations pour une nouvelle approche ... 24
         1.1.2.1 Modèle statique ... 24
         1.1.2.2 Modèle dynamique: allocation dynamique de ressources ... 25
   1.2. Présentation du projet R.C.A.I ... 28
      1.2.1. Architecture du réseau ... 28
      1.2.2. La partie traitement ... 29
      1.2.3. La partie routage ... 31
         1.2.3.1 La stratégie de routage ... 32
         1.2.3.2 Interblocage et famine dans R.C.A.I ... 33
   1.3. Conclusion ... 37

2. **Modèles de programmation et d'exécution** ... 41
   2.1. Définition d'un modèle d'exécution ... 42
      2.1.1. Modèle physique ... 43
   2.2. Modèles centrés objets ... 45
      2.2.1. Définition du modèle objet ... 46
         2.2.1.1 Définition d'un objet ... 46
         2.2.1.2 Classes et instances ... 46
         2.2.1.3 Héritage ... 47
         2.2.1.4 Activité d'un objet ... 48
         2.2.1.5 Problèmes liés au modèle objet ... 49
            2.2.1.5.1 Gestion de la mémoire ... 49
            2.2.1.5.2 Parallélisme et synchronisation ... 49
      2.2.2. Définition du modèle acteur ... 50
         2.2.2.1 Définition d'un acteur ... 50
         2.2.2.2 Délégation ... 52
         2.2.2.3 Continuation ... 52
         2.2.2.4 Instanciation ... 53

2.2.2.5 Sélection de méthodes 53
2.2.2.6 Activité d'un acteur 54
2.2.3. Définition du modèle d'agents 56
2.2.3.1 Définition d'un agent 56
2.2.3.2 Différents types d'agents 57
2.2.3.3 Organisation des agents 57
2.2.3.4 Protocoles de communication entre agents 57
2.2.3.5 Granularité et modèle d'agents 57
2.2.4. Conclusion 59
2.3. Un modèle d'acteurs particulier: CANTOR 60
2.3.1. Les objets CANTOR 60
2.3.2. Structures de contrôle et structures de données dans CANTOR 61
2.3.3. Structure d'un programme CANTOR 61
2.3.4. Les expressions de CANTOR 62
2.3.4.1 Déclaration de variables locales 62
2.3.4.2 Instruction conditionnelle 62
2.3.4.3 Affectation 63
2.3.4.4 Instruction de communication 63
2.3.4.5 Répétition 64
2.3.5. Un exemple de programme CANTOR 65
2.4. Mise en œuvre du modèle acteur dans R.C.A.I 66
2.4.1. Quelques caractères implicites 66
2.4.1.1 Communication asynchrone 66
2.4.1.2 Indéterminisme 67
2.4.1.3 Fiabilité des liens de communication 67
2.4.2. Caractéristiques mises en œuvre 67
2.4.2.1 Système de tampons 67
2.4.2.2 Gestion des références (accointances) 70
2.4.2.2.1 Reconfiguration dynamique d'un réseau d'acteurs 70
2.4.2.2.2 Continuation 70
2.4.2.2.3 Délégation et "pattern matching" 71

3. **Allocation dynamique** 75
3.1. Régulation de la charge dans les systèmes massivement parallèles 75
3.1.1. Quelques définitions 76
3.1.1.1 Régulation de charge 76
3.1.1.2 Granularité 76
3.1.1.3 Communication 76
3.1.1.4 Localité 76
3.1.2. Modes de régulation de charge 78
3.1.2.1 Mode centralisé 78
3.1.2.2 Mode réparti 78
3.1.2.3 Mode mixte 79
3.1.3. Modes de transfert de processus 80

3.1.4. Mode d'échange d'informations 81
3.1.5. Critères pour l'évaluation de l'indice de charge 83
3.1.6. Distance et protocole de transfert de processus 84
3.1.7. Stratégies de régulation de charge 85
3.1.7.1 Allocation statique 85
3.1.7.2 Méthodes aveugles 85
3.1.7.3 Méthodes à jetons 85
3.1.7.4 Propagation du gradient 87
3.1.7.5 Méthode des enchères 88
3.1.7.6 Méthode micro-économique 89
3.1.7.7 Méthode stochastique 90
3.1.8. De la prédiction de quelques paramètres 92
3.1.9. Régulation de la charge dans la machine R.C.A.I 93
3.2. Mise en œuvre de l'allocation dynamique 96
3.2.1. Allocation de ressources dans la machine R.C.A.I 96
3.2.1.1 Algorithme en colimaçon 97
3.2.1.2 Algorithme de diffusion par ondes 101
3.2.1.2.1 Calcul des bornes des messages de réponses 102
3.2.1.2.2 Protocole de réservation des cellules 103
3.2.1.3 Comparaison des deux méthodes de recherche de cellules libres 104
3.2.2. Implémentation et intégration de la recherche par ondes 108
3.2.2.1 Structure des tampons 109
3.2.2.2 Structure du chemin de données 110
3.2.3. Une nouvelle proposition d'algorithme pour la régulation de charge dans la machine R.C.A.I: l'algorithme CLIMB 114
3.2.3.1 Principe de l'algorithme proposé 116
3.2.3.1.1 Un objet en descente 117
3.2.3.1.2 Un objet en montée 118
3.2.3.1.3 Etude des frottements 119
3.2.3.2 Application à la régulation de charge 121
3.2.3.3 Cas de la machine R.C.A.I 122
3.2.3.4 Quelques résultats 124
3.2.3.5 Conclusion 136
3.3. Ramasse-miettes et gestion de la mémoire 138
3.3.1. Nécessité d'un ramasse-miettes 138
3.3.2. Algorithme du compteur de références 139
3.3.3. Algorithme "Mark & Sweep" (Récupérer & Nettoyer) 141
3.3.4. Algorithme "Mark & Compact" (Marquer & Compacter) 141

3.3.5. Algorithme “Stop & Copy” (Arrêter & Copier) 142
3.3.6. Algorithmes “générationnels” 143
3.3.7. Un algorithme “générationnel” parallèle 145
3.3.8. Un algorithme parallèle non-bloquant 147
3.3.9. Conclusion 150
3.3.10. Un algorithme distribué pour la récupération d’acteurs 152

4. Autres problèmes liés aux limitations de R.C.A.I 159
4.1. Serveur de code 159
4.1.1. Définition de l’espace d’adressage 160
4.1.2. Différents types de copies de code 162
4.1.2.1 Duplication 163
4.1.2.2 Copie distante 163
4.1.2.3 Dernière phase de la copie de code 165
4.1.3. Exemple de gestion d’un espace d’adressage 165
4.2. Ramasse-miettes dans R.C.A.I 167
4.3. Limitation de l’espace d’adressage de R.C.A.I 171
4.3.1. Limites physiques de la machine R.C.A.I 171
4.3.2. Notion de relais 171
4.3.3. Introduction de relais pour l’extension de l'adressage 174
4.3.3.1 Cas d'un relais physique 176
4.3.3.2 Cas d'un relais logique 178
4.3.3.3 Différence entre les deux types de relais 179
4.3.3.4 Relation entre relais et recherche de cellules libres 181

5. Expérimentation et évaluations 187
5.1. L’environnement de simulation 188
5.1.1. Le simulateur 190
5.1.1.1 Chargement du code 190
5.1.1.2 Paramètrage d’une session 190
5.1.1.3 Exécution et extraction de mesures 190
5.2. Les benchmarks 192
5.2.1. Crible d’Erathostène 192
5.2.2. Tri d’entiers 194
5.2.3. Calcul de la suite de Fibonacci 197
5.2.4. Comparaison des modèles statique et dynamique 200
5.3. Quelques résultats globaux sur des points délicats 201

6. **Conclusion et perspectives futures** 211

Bibliographie 217

ANNEXE I: Classification des machines parallèles 229

ANNEXEII: Environnement de simulation 235

II.1. L'assembleur 235

II.1.1. Commentaires 236

II.1.2. Directives d'assemblage..236

II.1.3. Instructions 237

II.2. L'éditeur de liens...240

Introduction

L'orientation actuelle, de ce que l'on pourrait appeler communément les "super calculateurs", allant des calculateurs vectoriels aux machines massivement parallèles, tend à favoriser l'émergence des machines multiprocesseurs regroupant un certain nombre de processeurs classiques. C'est ainsi qu'une des machines parallèles actuelles, la SP1 d'IBM, est constituée d'un réseau de processeurs RISC RS6000 du même constructeur, alors que ce processeur fut conçu à l'origine pour la construction de stations de travail. A la base de cette tendance se trouve la conviction que l'assemblage de processeurs à haute performances pourrait supplanter les machines parallèles dont toute l'approche aura été pensée et réfléchie autour de concepts purement parallèles (par exemple l'iPSC d'Intel) dont les processeurs ne peuvent, à priori, être utilisés d'une manière efficace que sur des architectures parallèles (contrairement aux précédents).

Cependant, de récents travaux de [SCHÖNAUER93] ont clairement montré que les machines de la première approche doivent intégrer des solutions aux problèmes cruciaux soulevés par les architectures parallèles. En effet, l'absence d'une unité de routage indépendante par exemple, nécessiterait que le processeur soit interrompu pour chaque communication à entreprendre - ce qui n'est pas raisonnable. De même, les applications d'ingénierie, toujours selon ce même auteur, nécessitent d'énormes calculs vectoriels. Le défaut d'une unité vectorielle pénalise sérieusement ce genre d'architectures.

L'approche massivement parallèle semble être une meilleure alternative. La CM5 de Thinking Machine en est un bon exemple. Ces machines allient à une conception relativement simple par rapport aux précédentes, une puissance de calcul non négligeable. Un de leurs atouts est la *modularité* (l'augmentation de la puissance de ces machines est généralement aisée). Les machines citées plus haut présentent généralement un nombre limité de processeurs. Une des plus récentes contributions au monde des machines massivement parallèles a été faite par CRAY Research. Il s'agit du T3D bâti autour de processeurs RISC ALPHA de DEC. Cette

machine peut recevoir jusqu'à 1000 processeurs. Les concepteurs chez Kendall Square Research (KSR), quant à eux, proposent la KSR2. C'est une machine massivement parallèle à mémoire partagée qui comprend de 32 à5000 processeurs "maison".

Toutes ces propositions de machines massivement parallèles confirment la prochaine orientation des machines parallèles. Le souci actuel est d'affirmer leur adéquation à des domaines aussi peu investis par les machines parallèles que celui de la gestion (KSR2 supporte notamment les outils de gestion tels que COBOL ou ORACLE 7).

Les machines massivement parallèles sont extensibles: leur structure est habituellement conçue de telle façon à ce que l'ajout de processeurs supplémentaires n'entraîne qu'un minimum de manipulations (c'est le cas des Connections Machines qui sont organisées en "rack"). Ces machines adaptent la puissance de calcul aux besoins des applications en allouant le nombre nécessaire de processeurs.

Notre projet s'inscrit dans le cadre des machines massivement parallèles.

Jusqu'à ces dernières années, seul un parallélisme limité était exploité (de l'ordre de 64 à 128 processeurs). Le massivement parallèle concerne les machines qui intègrent 64K processeurs ou davantage. Cette approche était réservée aux machines SIMD (Single Instruction Multiple Data), telle que laCM2.

L'originalité du projet **R.C.A.I**[1] (Réseau Cellulaire Asynchrone Intégré) a été d'étudier une structure MIMD (Multiple Instruction Multiple Data) à grain fin: le processeur élémentaire est de taille très limitée, doté de peu de mémoire et associé à une unité de routage simple (10000 à 20000 transistors en tout), pour réaliser des machines de 100000 processeurs élémentaires.

[1]Le projet R.C.A.I (Réseau Cellulaire Asynchrone Intégré) a été soutenu par le Ministère de l'Enseignement supérieur et de la Recherche via le PRC-ANM (Programme de Recherche Commun en Architectures des Nouvelles Machines) et par le CNRS, via le GDR-ANM, dans le cadre des activités de recherche en architecture de machines, entre 1989 et 1993.

Les machines massivement parallèles rendent nécessaire l'élaboration d'environnements qui en faciliteront l'usage, et en tireront le meilleur parti. Lors du développement d'une application, l'utilisateur doit s'attaquer à la difficile tâche de répartir celle-ci sur les processeurs. Le plus simple est une préparation statique: préparation, lors de la compilation, d'un parallélisme statique; cependant cette approche correspond à une classe de problèmes limitée. Une autre approche, peut-être plus prometteuse, pour la répartition du code serait l'adoption du modèle d'exécution par acteurs [AGHA86a]: la répartition du code revient à l'allocation des acteurs aux processeurs.

Le but de nos travaux a été de proposer une mise en œuvre de la programmation suivant le modèle d'acteurs sur notre machine massivement parallèle R.C.A.I. L'étude du modèle d'acteurs a permis de dégager un certain nombre de points:

- toutes les fonctionnalités de ce modèle ne doivent pas nécessairement avoir une projection directe au niveau de l'architecture,
- le réseau de communication doit supporter d'autres modèles de communication que l'actuel échange point-à-point,
- le jeu d'instructions de la machine doit être étendu pour réaliser des fonctions de plus haut niveau.

Dans cet ordre d'idées, des notions telles que la délégation ou la continuation peuvent aisément être traduites par un protocole d'échange de messages entre les processus impliqués dans ces opérations.

Par contre, la création de nouveaux acteurs nécessite à la fois:

- une méthode de diffusion (au niveau du réseau de communication) de la requête pour la recherche d'une cellule libre,
- une instruction pour initier cette opération et récupérer le résultat de cette recherche.

Parmi les problèmes les plus importants soulevés par le modèle d'acteurs, nous pouvons citer:

- la nécessité d'élaborer un système de récupération d'espace (*garbage collecting*),
- une politique pour la régulation de la charge des processeurs (et éventuellement un mécanisme de migration d'objets),
- un système de désignation (*naming*) des entités pour les différencier.

Dans le premier chapitre de ce manuscrit, nous présentons le contexte dans lequel s'est inscrit cette thèse en en fixant les objectifs.

Dans le second chapitre, nous passons en revue les modèles centrés objets. Nous nous intéressons à un cas particulier des langages d'acteurs, à savoir le langage CANTOR. Nous exposons par la suite la mise en œuvre du modèle acteur sur R.C.A.I.

Dans le troisième chapitre, nous abordons le problème de l'allocation et de la restitution dynamique de ressources dans un environnement massivement parallèle. Après une présentation de l'état de l'art pour la régulation de charge des processeurs, nous présentons notre proposition pour réaliser cette opération: il s'agit de l'algorithme **CLIMB**.

Dans le chapitre suivant, nous nous attachons aux problèmes relatifs aux limites physiques de la machine R.C.A.I. Pour rompre les limites d'adressage des cellules, nous proposons une nouvelle notion, celle des **relais logiques**.

Le cinquième chapitre est dédié aux expérimentations et évaluations des mécanismes de gestion dynamique intégrés dans R.C.A.I.

Nous terminons par une conclusion sur l'intérêt de l'approche et quelques perspectives futures pour les travaux menés dans cette thèse.

Chapitre 1
Contexte de Travail & Objectifs

1. CONTEXTE DE TRAVAIL ET OBJECTIFS

1.1. Le projet “Réseau Cellulaire Asynchrone Intégré”

1.1.1. Motivations et historique du projet

Les machines massivement parallèles connaissent de plus en plus d’engouement. Toutes les machines récentes tendent à conforter cette orientation: la CM5 de Thinking Machine, la KSR2 de Kendall Square Research ou encore INTEL avec sa toute dernière machine la Paragon.

La question essentielle est de connaître l’opportunité de ces machines. Le nombre peut-il remplacer la puissance individuelle? Construire d’énormes machines avec de gros processeurs est difficilement envisageable à grande échelle car trop coûteux... Cependant la réplication massive de processeurs élémentaires ne dépend que de la possibilité d’en intégrer un nombre maximal. La technologie pour leur réalisation ne cesse de progresser, repoussant un peu plus loin à chaque étape les limites d’intégration sur le silicium (ou sur d’autres supports). De ce fait, il est réaliste d’espérer des machines intégrant plusieurs milliers de processeurs “rudimentaires”. Le problème soulevé alors par ce nombre d’individus est leur organisation et leur contrôle. Ce qui revient donc à établir entre eux des mécanismes de communications performants (dans tous les sens du terme) pour une meilleure circulation de l’information. La gestion d’une information cohérente est gage d’une évolution “saine” pour la réalisation d’une œuvre commune.

Les travaux menés au sein de notre équipe ont porté jusqu’à présent sur la définition d’une architecture simple, mais puissante, se contentant, à la plus petite échelle, de réaliser des opérations “élémentaires” (essentiellement des opérations arithmétiques et des opérations de communication). C’est la mise en œuvre d’un nombre important de ces éléments de base et surtout leur **coordination** qui en font toute sa puissance. Cela nous a toujours fait penser à une fourmilière où chaque individu, à la base, possède une activité rudimentaire. La coopération entre ces individus “élémentaires” en fait une

société organisée et cohérente, visant à réaliser un but commun: la survie et la reproduction de l'espèce!

Reportée à notre cas, cette image reste toujours valable. Les études menées par [RUBINI90] et [PAYAN91] ont montré l'efficacité d'une telle conception: la collaboration de tâches élémentaires peut réaliser des fonctions de haut niveau.

Historiquement, notre projet **R.C.A.I** (Réseau Cellulaire Asynchrone Intégré) a subi plusieurs phases: parti de machines dédiées à des applications spécifiques, il a ensuite évolué vers une machine programmable, capable de s'adapter à plusieurs modes de programmation (comme nous le verrons par la suite). Le point commun entre ces versions successives sont:

- une architecture en une grille ouverte à deux dimensions,
- un asynchronisme d'exécution entre les processeurs élémentaires,
- et une communication par messages comme unique moyen d'échange.

La tendance actuelle est, comme dans notre cas, vers le développement d'unités de routage indépendantes. Nous retrouvons dans cette catégorie le projet MOSAIC de Caltech [ATHAS87b] [ATHAS88], le projet MEGA [GERMAIN89] avec l'UGC (Unité de Gestion des Communications), le Transputer T9000 avec son processeur de routage associé le C104 [MAY93] ou encore le DCM (Direct-Connect Routing Module) de l'iPSC/2 [NUGENT88]. Le point commun entre tous ces routeurs est l'utilisation de la technique dite du ***wormhole***[2] [SEITZ84] [DALLY87], qui semble devenir de plus en plus un standard de fait...

Avant de voir en détail l'évolution de R.C.A.I vers une machine massivement parallèle MIMD à allocation dynamique de ressources, nous introduisons d'abord une classification des machines parallèles, puis nous présentons quelques unes des machines (à grain fin puis à gros grain) existantes sur le marché. Ces machines nous permettront de situer nos objectifs (cf. introduction).

[2]Se reporter à la définition de cette stratégie au paragraphe 1.2.3.1.

1.1.1.1. Classification des machines parallèles

La classification établie par [FLYNN72] est maintenant un standard établi. Afin de situer rapidement notre approche par rapport à cette classification, nous en donnons un bref rappel à ce niveau[3]. Cette classification est basée sur le séquencement des données et des instructions au niveau des processeurs:

- Les architectures **SISD** (Single Instruction Single Data): c'est la machine de Von Neumann traditionnelle; les instructions (lues à partir d'une mémoire vive) sont exécutées l'une à la suite de l'autre.
- Les architectures **SIMD** (Single Instruction Multiple Data): plusieurs unités de traitement sont supervisées par la même unité de contrôle: mode d'exécution synchrone. Les unités de traitement exécutent la même instruction sur des données distinctes.
- Les architectures **MIMD** (Multiple Instruction Multiple Data): cette dernière catégorie comprend l'ensembles des machines fortement couplées et communiquant par messages. La mémoire peut être soit partagée soit distribuée. L'ensemble des processeurs fonctionnent en mode asynchrone.

La classe de machines qui nous intéresse relève du groupe des machines MIMD. Dans notre contexte, chaque processeur est doté d'une mémoire locale, il n'existe pas de mémoire commune. L'originalité de notre approche tient à la petite taille du processeur élémentaire et à la finesse du grain de calcul. Pour mieux la situer, nous allons passer en revue les machines les plus proches. Nous en exposerons également d'autres plus éloignées, mais qui relèvent de la même classe de machines (MIMD).

1.1.1.2. Les machines massivement parallèles

Il existe plusieurs machines parallèles sur le marché. Ce n'est plus un mythe. Une des premières machines qui a eu un succès retentissant est la CM2 de Thinking Machine. A la simplicité de sa conception fut allié un environnement de programmation riche. Ce qui explique en grande partie sa réputation.

[3]Nous invitons le lecteur à se reporter à l'annexe I pour de plus amples informations.

1.1.1.2.1. Le GAPP

Le processeur GAPP (Geometrical Arithmetic Parallel Processor) peut être considéré comme une machine cellulaire [SANSONNET91]. Un circuit (processeur) comporte 72 processeurs élémentaires (1-bit), disposés en une matrice (grille) de 6 par 12 éléments. Chaque processeur élémentaire peut donc communiquer avec ses quatre voisins immédiats par des liens série (figure1.2).

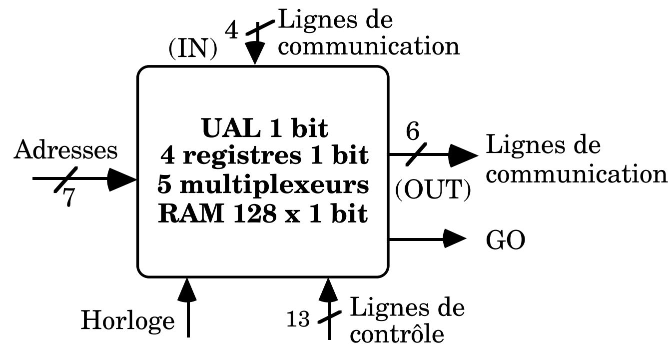


Fig.1.2. Le processeur GAPP.

Chaque processeur élémentaire est composé:

- d’une unité de calcul (UAL) un bit,
- d’un registre 4 bits,
- d’une mémoire vive de 128 bits,
- de 5 multiplexeurs,
- de 10 lignes de communication cardinales (entrées et sorties),
- et de 1 ligne de sortie globale (GO).

Les circuits emploient un multiple de matrices de 6 par 12 processeurs élémentaires.

Les structures ainsi composées sont notamment bien adaptées au traitement de bas niveau d’images.

1.1.1.2.2. La Connection Machine 2

On peut voir la Connection Machine 2 (**CM2**) comme étant un ensemble d'opérateurs du type GAPP. Cependant, la CM2 est équipée d'un réseau d'interconnexion, d'une unité micro-programmée avec un langage de micro-programmation et surtout d'outils logiciels.

La CM2 est un super-ordinateur massivement parallèle SIMD [TUCKER88] [DELASALLE90] [TREW91] [SANSONNET91].

Il existe deux types de processeurs dans la CM2 (figure1.1):

- les processeurs 1-bit, qui réalisent les opérations logiques, mais également les opérations entières et en virgule flottante,
- et les processeurs 32-bits, utilisés comme des accélérateurs vectoriels de calcul.

Les processeurs 1-bit sont groupés dans des "amas" (*clusters*) de 32 éléments, autour de chaque processeur vectoriel. Leur nombre est compris entre 8192 et 65536 processeurs élémentaires.

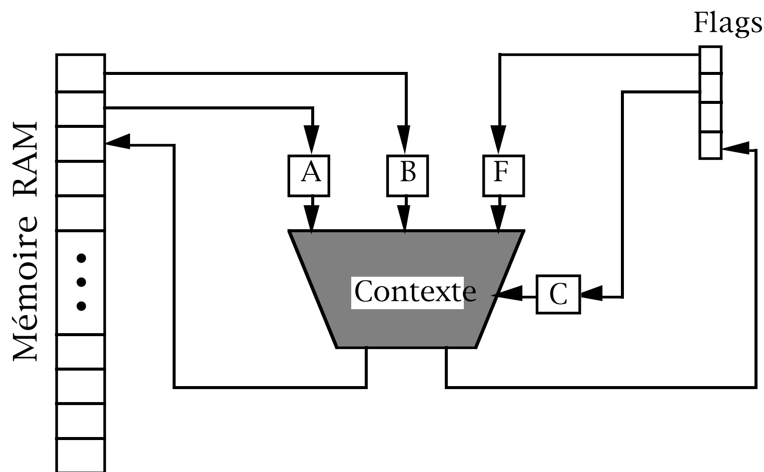
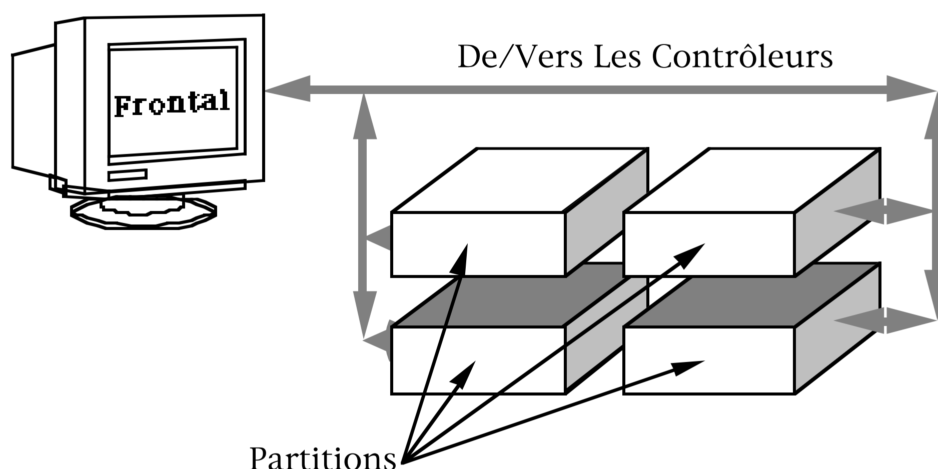


Fig.1.1. Le processeur élémentaire de la CM2.

Cet ensemble de processeurs est divisé en quatre partitions (de 16K processeurs chacune). On peut utiliser 1, 2 ou 4 de ces sous-ensembles (lorsque ceux-ci sont disponibles - libres).

Comme la CM2 est une machine SIMD, il y a quatre processeurs de contrôle - appelés séquenceurs (*sequencers*). Un contrôleur est associé à chaque partition, donc quatre programmes différents peuvent s'exécuter simultanément sur une CM2. Cependant le contrôle global est réalisé sur un ordinateur frontal. Ce processeur frontal envoie des instructions parallèles aux séquenceurs, qui, à leur tour, renvoient ces instructions aux processeurs 1-bit.



Chaque groupe de 16 processeurs 1-bit est associé à un routeur pour les communications distantes.

1.1.1.2.3. La Machine MEGA

Avec la machine MEGA, nous changeons de classe de machines. Il s'agit d'une machine MIMD avec un processeur 16 bits et une taille de mémoire plus importante.

La machine MEGA (Machine pour l'Expérimentation des Grandes Architectures) est une machine massivement parallèle. Cette machine a été développée par l'équipe Architecture et conception des Circuits Intégrés du LRI, dirigée par le Professeur Jean-Paul Sansonnet. MEGA est une machine MIMD à passage de messages [GERMAIN89].

Son architecture est un cube, une grille 3D (figure1.3). Le but visé par cette équipe est d'arriver à intégrer jusqu'à un million de processeurs élémentaires (PE). Chaque PE intègre une unité centrale, une mémoire locale et une unité de gestion des communications[4].

Cette machine a été conçue pour la résolution de problèmes d'intelligence artificielle. La contribution majeure de ce projet, de notre point de vue, est la définition d'une stratégie de routage adaptative: le **routage forcé**. Cette technique d'acheminement de messages permet une utilisation efficace des liens de communication [GERMAIN89].

[4]A ce sujet, il existe une grande similitude entre les projets R.C.A.I et MEGA : le nombre visé de processeurs à intégrer est du même ordre, ainsi que la séparation entre les unités de calcul et de communication.

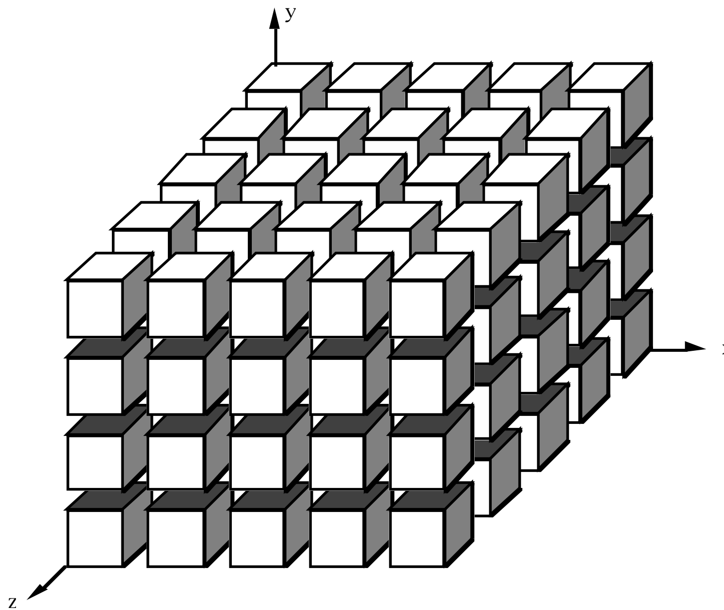


Fig.1.3. Structure du réseau tridimensionnel de MEGA.

1.1.1.2.4. La CM5

La CM5 est une machine MIMD à mémoire distribuée dont le nombre de processeurs peut comprendre de 32 à 2048 processeurs élémentaires. Chaque processeur élémentaire (dans sa version "*High-Performance Arithmetic Hardware*") est bâti autour d'un processeur Sparc (RISC), doté d'une mémoire de 32 Mo (divisée en quatre bancs indépendants de 8 Mo chacun) avec une taille de bus de 64 bits de données et 8 bits de protection de la mémoire ("*ECC code*"), et de quatre unités de calcul en virgule flottante (figure1.4).

L'architecture de la CM5 est constituée de trois réseaux (figure1.4):

- le réseau de données (*Data Network*), qui gère les communications point-à-point,
- le réseau de contrôle (*Control Network*), qui réalise les opérations globales, telles que la diffusion, la réduction ou les synchronisations,
- le réseau de diagnostic (*Diagnostic Network*), qui, non seulement permet de renseigner l'administrateur sur l'état du système, mais également de réaliser la reconfiguration nécessaire des réseaux de communication (les deux précédents) pour poursuivre le traitement en cas d'erreur.

Tout comme la machine de la même famille décrite plus haut (la CM2), la logithèque de la CM5 est assez bien fournie. Outre les langages de programmation, tels que le CMFortran, le C* ou le *Lisp, une bibliothèque

scientifique très riche est proposée, avec des environnements de programmation (Prism) et d'exécution (à la Unix).

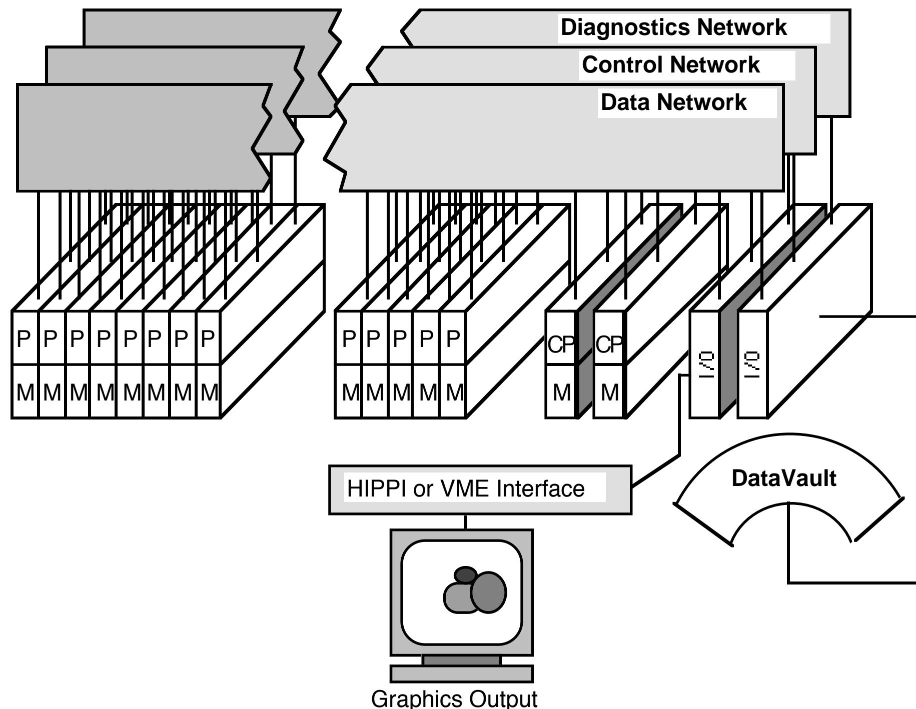


Fig.1.4. Architecture globale de la CM5.

1.1.1.2.5. La SP1

La SP1 (Scalable POWERparallel 1) est une machine parallèle d'IBM constituée de 8 à 64 processeurs [STUNKEL94] [CLAUSEL94]. Chaque processeur élémentaire est un RS6000 du même constructeur (utilisé notamment dans les stations de travail).

Dans le cas d'une machine SP1 à 64 processeurs, cet ensemble de processeurs est divisé en quatre groupes (clusters) avec deux étages de commutateurs (figure1.5). De chaque cluster sortent seize liens: douze vers le reste des clusters, et quatre vers des connexions externes (mémoire de masse ou autre). Chaque élément du switch possède 8x8 ports. Dans la figure1.5 les liens visualisés sont bidirectionnels; il faut donc multiplier par deux ce nombre.

Le réseau de communication de la SP1 est synchrone. La technique de routage utilisée est le wormhole. Le paquet de base a une taille d'un octet. Tous les messages doivent traverser quatre éléments du switch pour arriver à destination.

A titre d'exemple, nous donnons quelques caractéristiques de l'IBM 9076 SP1 acquis par l'IMAG. Cette machine est constituée de 32 nœuds. Chaque nœud possède une mémoire de 64 Mo et un disque dur de 1 Go. Il existe deux type de réseaux de communication:

- un réseau interne, le switch,
- et un réseau Ethernet standard pour les communications externes.

Les processeurs de cette machine sont divisés en deux groupes de 16 nœuds. Chacun de ces groupes est relié directement à un serveur de *boot* par le réseau Ethernet pour le chargement du système au démarrage.

1.1.1.2.6. La Paragon

La structure de cette machine est une grille 2D. Elle dispose de 56 à 4096 processeurs élémentaires (**PE**) [PIERCE94]. Chaque nœud de cette grille est formé (figure1.6):

- d'un processeur de calcul (i860 XP),
- d'une mémoire locale de 128 Mo,
- d'un processeur de communication (i860), dont la tâche est de préparer les messages et de les réceptionner,
- d'un processeur de routage, le PMRC (Paragon Mesh Routing Chip), dont la stratégie d'acheminement est le whormhole. Chaque PMRC est connecté physiquement à ses quatre voisins.

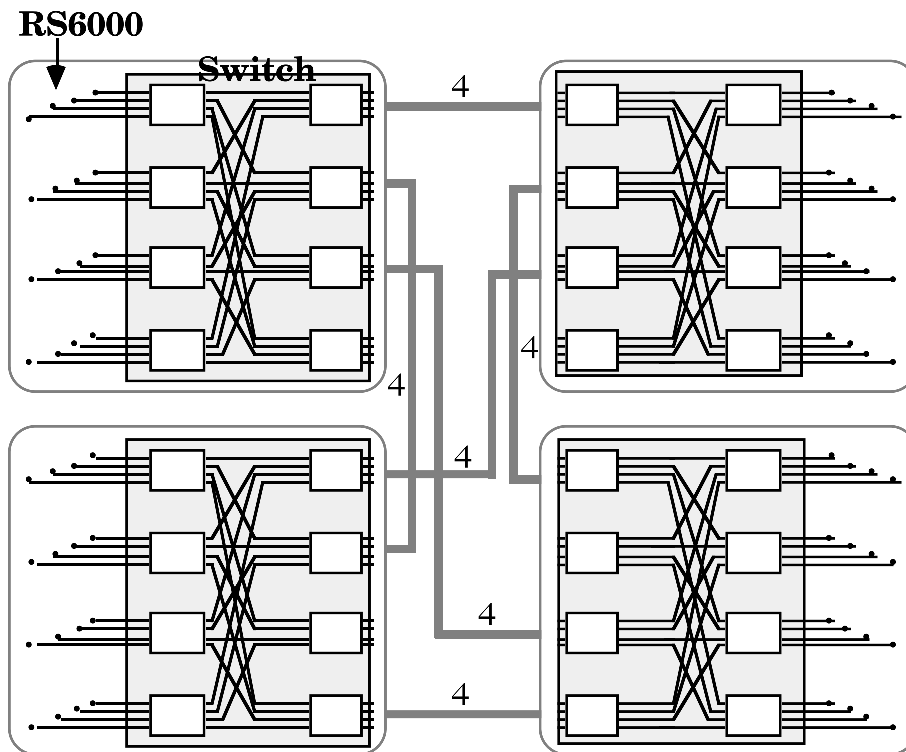


Fig.1.5. Architecture globale de la SP1.

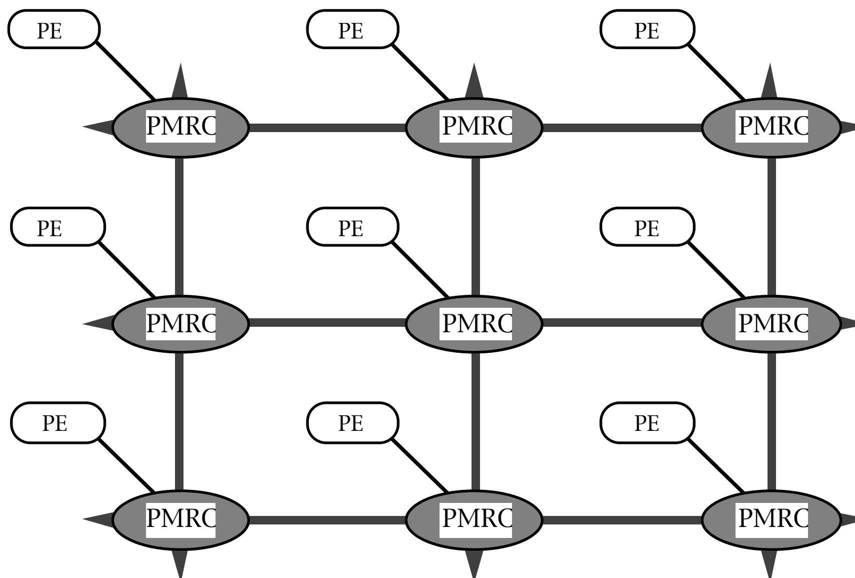


Fig.1.6. Architecture globale de la Paragon.

Cette présentation ne prétend pas être exhaustive, loin de là. Notre but est d'exposer les machines les plus proches de la nôtre, pour mieux la situer. La CM2 ainsi que le GAPP semblent très proches de notre conception au niveau de la taille et de la puissance de chaque processeur élémentaire (du moins par rapport aux premières versions du projet R.C.A.I). Les deux premières machines que nous venons de voir sont toutes deux régies en mode SIMD. Ce mode ne répond pas à la classe de problèmes dits "irréguliers", tels que les algorithmes récursifs ou l'intelligence artificielle. D'une manière générale, lorsqu'on doit exécuter du code différent sur les processeurs, ce mode est incompatible[5] (se reporter à la définition de ce mode).

D'autre part, les machines MIMD sont toutes de très grande capacité (le bus de données) et nécessitent beaucoup d'efforts, aussi bien humains que financiers, pour leur réalisation. Elles ne sont donc pas à la portée des milieux universitaires.

[5] Comme le signale [HILLIS_85], il est possible qu'une machine SIMD puisse simuler une autre machine MIMD en temps linéaire, en utilisant un interpréteur. Chaque processeur va interpréter les données qu'il reçoit comme des instructions. Cette solution reste tout de même très lourde et induit un *overhead* non négligeable pour sa réalisation !

1.1.1.3. Le projet R.C.A.I

Le développement du projet R.C.A.I a été suscité par l'élaboration d'une machine parallèle qui, tout en gardant une conception relativement simple, ne néglige nullement le côté performances. L'utilité d'une machine parallèle est d'arriver à résoudre des problèmes connus pour être gourmands en temps de calcul. La devise est alors de "diviser pour *résoudre*". Les structures en hypercube ou les grilles 3D présentent, au niveau conceptuel, énormément d'intérêts au niveau de leur réseau de communication. Cependant, le passage du modèle architectural au silicium est dans la plupart des cas coûteux, voire impossible pour des structures intégrant plusieurs milliers de processeurs élémentaires. Le nombre de lignes de communication à réaliser est assez volumineux et surtout irrégulier (certaines lignes sont plus grandes que d'autres). La projection la plus naturelle sur un plan de silicium est la grille 2D. Les lignes de communication entre éléments voisins sont toutes de même longueur. Elles présentent donc les mêmes caractéristiques électriques (délai de transmission, bande passante...).

Certains processeurs parallèles, tel que le GAPP, présentent l'inconvénient majeur de ne réaliser que des communications fixes. Seuls les processeurs voisins (directement reliés) peuvent entrer en communication.
Lors de la conception de la machine R.C.A.I, notre souci a toujours été d'établir des communications générales, où chaque processeur peut communiquer avec n'importe quel autre processeur.

Plusieurs tentatives ont été réalisées au cours du projet R.C.A.I [CORNU88] [OBJOIS88] [LATTARD89] [FAURE90] [PAYAN91] [RUBINI92] [KARABERNOU93]. Nous allons en retracer les plus importantes.

1.1.1.4. Processeur dédié

Les premières expériences du projet R.C.A.I ont débouché sur la conception de processeurs dédiés. Chaque processeur a été conçu pour la réalisation d'une fonction spécifique. Les deux premières applications pour ce processeur ont été:

- la simulation logique de circuits intégrés [OBJOIS88],
- et la reconstruction d'images [LATTARD89].

A cet égard, ces processeurs peuvent aisément être considérés comme des co-processeurs de haut niveau.

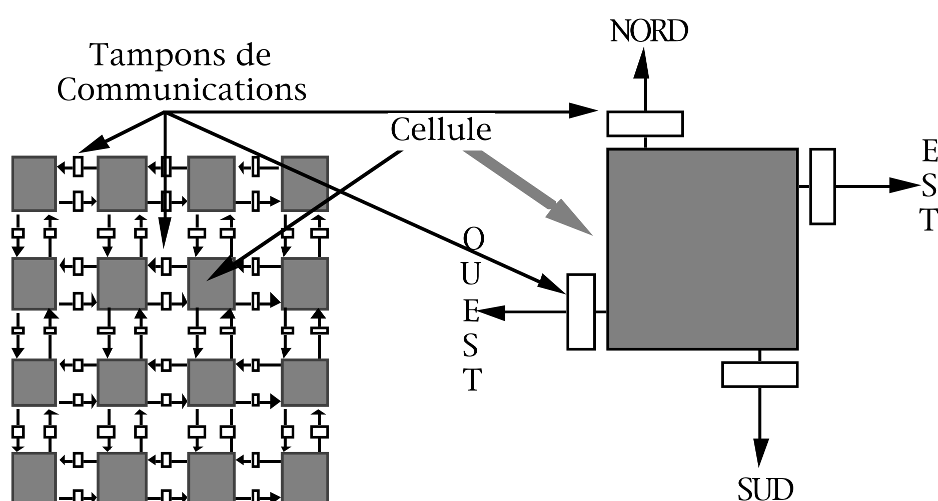


Fig.1.7. Structure en grille 2D de R.C.A.I.

Le point commun entre ces différentes conceptions sont:

- une intégration massive de processeurs élémentaires,
- une communication par messages entre ces derniers.

Dès cette époque, l'architecture en une grille 2D fut adoptée pour les raisons évoquées précédemment (figure1.7). Cette structure prit le nom de **réseau cellulaire**. Chaque cellule peut communiquer (au plus) avec ses quatre voisines immédiates par le biais de tampons (buffers) de communications.

Toutefois, l'unité de traitement du processeur élémentaire est réduite à sa plus simple expression: seules quelques opérations arithmétiques et logiques sont disponibles, augmentées d'instructions de communication primitives.

1.1.1.5. Processeur programmable

Un besoin pressant fut ressenti par les possibilités limitées du processeur de base. Une architecture programmable fut alors imaginée. Une machine hôte initialise le réseau et récupère à la fin de l'exécution les résultats. Chaque cellule du réseau est composée [KARABERNOU93] par:

- une unité de traitement du type MC6800,
- et d'une unité de routage indépendante.

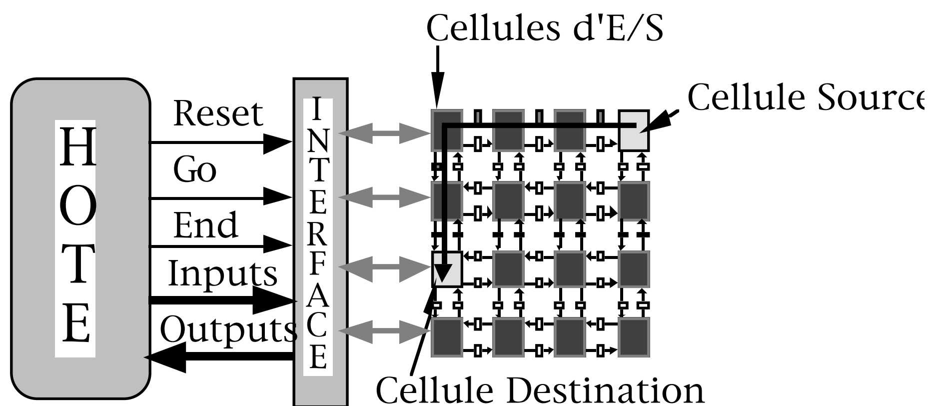


Fig.1.8. Architecture de R.C.A.I.

D'une manière conceptuelle, nous disposons de deux réseaux (figure1.9):

- un réseau de calcul, constitué de l'ensemble des unités de traitement,
- et un réseau de communication, dont la topologie est une grille 2D ouverte.

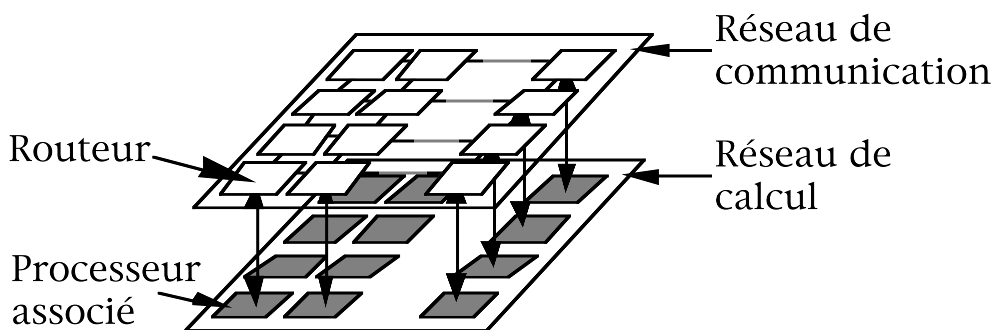


Fig.1.9. Niveau communication & Niveau calcul.

L'originalité de cette approche est la séparation entre les deux réseaux. De cette manière, chaque niveau réalise la fonction qu'il sait le mieux faire. Le second trait est la parallélisation des routeurs: cinq communications peuvent être réalisées simultanément au niveau de chaque cellule.

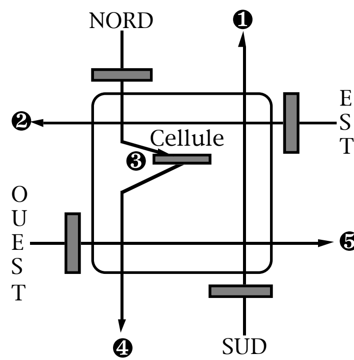


Fig.1.10. Cinq communication en parallèle.

Le mode de programmation de cette machine [PAYAN91] [RUBINI92] est **statique**: après avoir chargé le code au niveau de chaque cellule, celui-ci subsiste jusqu'à la fin d'exécution du programme.

Bien que ce mode réponde à une grande classe de problèmes[6] [RUBINI92], nous pouvons cependant porter les remarques suivantes:

- le temps nécessaire pour le chargement du code dans l'ensemble du réseau est très important comparé à celui de son exécution sur cette architecture,
- une partie du code chargé risque d'être utilisée sporadiquement, d'où une sous-utilisation des ressources,
- toute une classe de problèmes ne peut être abordée <u>efficacement</u> suivant cette approche: il s'agit notamment de systèmes d'intelligence artificielle qui évoluent en cours d'exécution [FERBER89] [HASSAS92].

Ces constatations nous ont donc amenés à considérer une nouvelle approche pour la programmation de ce réseau.

[6]Des algorithmes de tri, de traitement d'images, de réseaux de neurones...

1.1.2. Motivations pour une nouvelle approche

Devant ces limites, nous nous sommes demandés de quelle manière économiser du temps lors du chargement du code? La réponse qui nous est venue à l'esprit est de ne charger qu'une copie de chaque "bout de code". L'ensemble de ces morceaux de code constitue le programme formulé par l'utilisateur. Par la suite, ces bouts de code se dupliqueront suivant l'évolution du programme. Ce schéma d'exécution est donc tout à fait **dynamique**.

Nous allons revoir rapidement l'approche statique avant d'entamer celle dont il est question au niveau de cette thèse, à savoir l'approche dynamique.

1.1.2.1. Modèle statique

Plusieurs modèles d'exécutions se prêtent tout à fait aux architectures statiques. Le modèle à flot de données (DATA FLOW) en constitue un exemple typique [PAYAN91]. Dans ce modèle, on commence par extraire les opérateurs, puis à les affecter aux différents processeurs du réseau (figure1.11). Par la suite, les données circulent suivant un diagramme établi à la compilation durant toute l'exécution du programme correspondant (schéma statique).

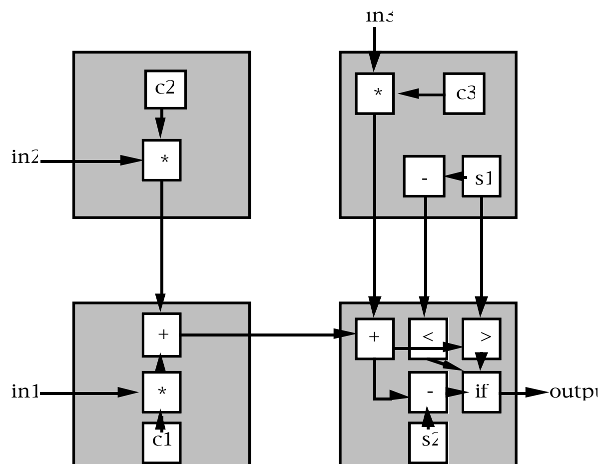


Fig. 1.11. Exemple du modèle à flot de données.

De même, dans le modèle fonctionnel, le nombre des fonctions est établi au départ et reste identique jusqu'à la fin de l'exécution. Les données (paramètres) sont fournies aux fonctions qui délivrent en retour des résultats à d'autres fonctions (figure1.12). Il n'y a pas de génération de nouvelle fonction en cours d'exécution.

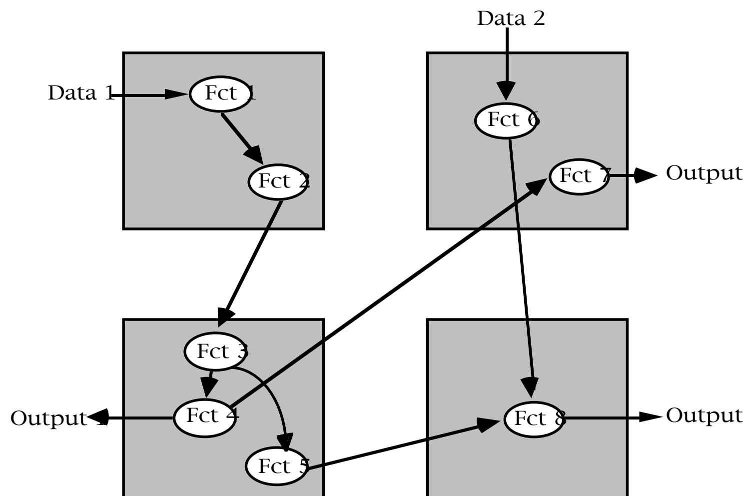


Fig. 1.12. Exemple du modèle fonctionnel.

1.1.2.2. Modèle dynamique: allocation dynamique de ressources

D'un autre côté, il existe un tout autre modèle d'exécution qui s'adapte d'une manière naturelle aux architectures parallèles. Il s'agit du modèle à base d'acteurs [LATROUS95] [AKTOUF93]. Très brièvement, il s'agit d'entités autonomes (aussi appelées *agents*) et qui coopèrent par échange de messages. Le comportement d'un acteur est indéterministe: il est fonction de son état courant et des messages en entrée.

Le modèle d'acteurs peut reproduire pratiquement tous les modèles d'exécution connus [AGHA86b]: cela va du modèle séquentiel, au modèle fonctionnel, en passant par le modèle à flots de données ou les modèles de l'intelligence artificielle à base de règles [HASSAS92].

Cependant, de par sa nature même, le modèle d'acteurs ne peut être exploité efficacement sur une architecture statique. Le schéma d'exécution du modèle d'acteur est éminemment dynamique. Tout au long de l'évolution d'un tel schéma, il y a création et destruction d'acteurs. De même, de nouvelles compositions d'acteurs ou des changements des liens (de communication) entres ceux-ci peuvent intervenir. Ces opérations ne peuvent être intégrées statiquement, à moins de recourir à des artifices incompatibles avec le modèle statique.

L'intégration de mécanismes dynamiques suppose une allocation dynamique des ressources. Les méthodes d'allocation statique (dites aussi

méthodes de placement) sur un réseau de processeurs parallèles fournissent de très bons résultats [TALBI93]. Les techniques issues de celles utilisées jadis dans les circuits intégrés - échanges de paires, placement constructif ou itératif [LATROUS88], ou encore le recuit simulé [BOKHARI81] ou de méthodes plus récentes, telles que les méthodes génétiques [TALBI93] - ont toujours gardé leur attrait lors de leur adaptation pour le placement de tâches sur les architectures parallèles [CHEN90] [BAIARDI89] [GHOSH89] [KRAVITZ87]...

Le point commun entre ces différentes approches est l'hypothèse d'une connaissance, partielle ou totale, de paramètres qualificatifs et/ou quantitatifs de l'application cible. Ces paramètres, qui entrent dans l'évaluation d'une fonction de coût, peuvent comprendre:

- la taille du réseau des entités à placer,
- le coût des communications sur les liens les reliant,
- la mémoire susceptible d'être utilisée,

ainsi que d'autres facteurs jugés critiques[7] pour l'évaluation de cette fonction de coût. Cette dernière caractérise un placement en lui affectant une valeur. Les différentes valeurs ainsi dégagées permettent de comparer une série de placements pour une même application afin d'en choisir le (ou les) meilleur(s).

Dans les systèmes dynamiques, cette connaissance à priori est difficilement extractible à partir des éléments du système (voire impossible à établir). Le nombre d'entités à générer ne peut pas être connu à l'avance. De même, il est pratiquement impossible d'associer préalablement un coût aux communications. Les communications dépendent:

- de la **distance** qui sépare les entités en relation; celle-ci est conditionnée par l'affectation des entités mises en jeu dans cette communication aux ressources disponibles au moment de leur génération,
- de la **charge des liens** de communication; c'est-à-dire, le nombre total de messages en cours de transit dans le réseau à l'instant où cette opération prend effet.

Ces remarques nous ont tout naturellement conduit à considérer l'allocation dynamique de ressources: à la demande et suivant la disponibilité de celles-ci. Ceci ne signifie nullement (comme nous le verrons par la suite)

[7]Tels que des critères de regroupement de nœuds, la distance maximale autorisée entre certains nœuds, les entrées/sorties...

qu'il n'y a pas lieu de tenter d'optimiser cette allocation. Par exemple, la distance séparant des acteurs reliés (par des liens de communications ou des références croisées) constitue un facteur important à minimiser.

Contrairement aux travaux menés par [DALLY84] [DALLY87], notre but n'est pas de proposer une machine dédiée. Les travaux de DALLY et de son équipe ont conduit à l'élaboration d'une machine-langage centrée objets[8]: le MDP (Message Driven Processor). Son objectif est la réduction du temps requis pour la gestion des messages dans le réseau. Pour cela, une unité spécialisée a été développée pour l'interprétation des messages (figure1.13).

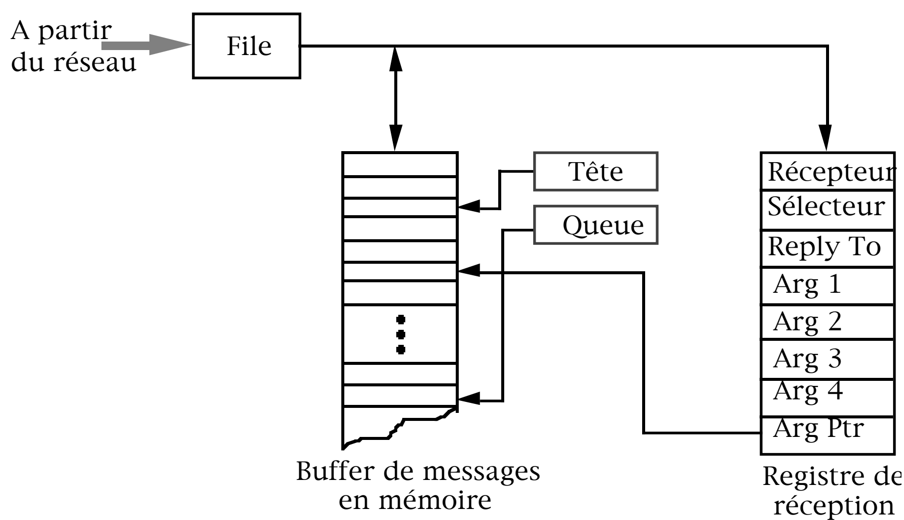


Fig.1.13. Réception d'un message dans le MDP.

L'inconvénient d'une telle machine est, à l'évidence, un modèle d'exécution figé. Toute modification de celui-ci entraîne inévitablement celle de l'unité correspondante. De plus, cette unité est très onéreuse en silicium (de l'ordre de 1/3 de la surface totale, sans compter le système de routage attenant).

Nous allons étudier, dans les sections suivantes, d'une manière plus détaillée, nos modèles dynamiques d'exécution et de programmation pour une machine de ce type.

[8]cf.§2.2 pour la définition de la notion de langages à objets.

1.2. Présentation du projet R.C.A.I

Pour situer la cadre dans lequel est mis en œuvre ce modèle, il est utile de rappeler le projet R.C.A.I jusqu'au point où il a été mené avant le début de nos travaux: il s'agit de l'approche statique.

1.2.1. Architecture du réseau

La machine cellulaire R.C.A.I se présente sous forme d'une grille 2D ouverte où chaque cellule est reliée à ses quatre voisines immédiates (figure1.14).

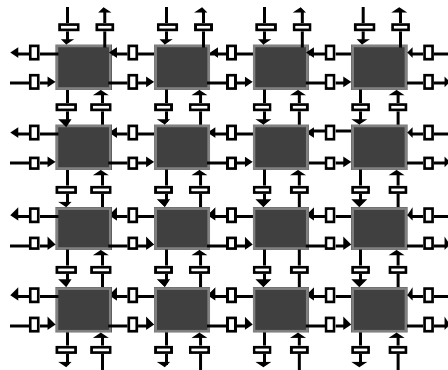


Fig.1.14. Réseau d'interconnexion.

Chaque cellule du réseau a été conçue en deux parties:

- la partie traitement (un processeur de calcul doté d'une mémoire locale),
- la partie routage (bloc de transmission et de réception de messages).

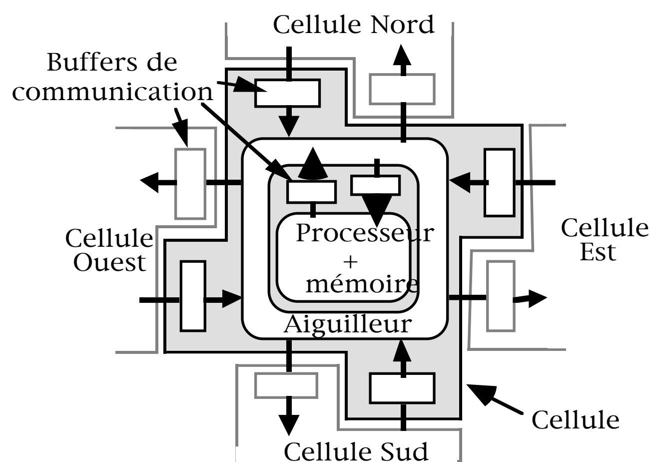


Fig.1.15. Structure d'une cellule.

1.2.2. La partie traitement

La partie traitement est à base d'un microprocesseur 8 bits du type MC6800, associée à une mémoire locale de 256 octets: il n'y a pas de mémoire globale.

Les modes d'adressage prévus dans la machine R.C.A.I sont issus en grande partie des modes classiques du MC6800. Il s'agit notamment des modes:

- d'adressage inhérent,
- d'adressage immédiat,
- d'adressage indexé, et
- d'adressage indexé post-incrémenté.

Un autre mode, dit mode d'adressage court, a été introduit pour réduire le nombre de cycles pour les accès mémoire, et pour optimiser le code des programmes [RUBINI92]. Dans ce mode, chaque instruction est composée de deux champs de 4 bits chacun:

- les premiers 4 bits définissent le code opération,
- les 4 bits restants déterminent le déplacement par rapport à la première page de la mémoire ($0) dans le cas des instructions LDA et STA et par rapport à la dernière page de la mémoire ($F) dans le cas des instructions de communications PUT, GET et TRY.

Le jeu d'instructions est également inspiré de celui du processeur MC6800. En résumé, nous donnons la classification suivante[9]:

Arithmétiques :		Accumulateur :	
ADD	addition	LDA	chargement accumulateur
SUB	soustratction	STA	stockage accumulateur
ADC	addition avec retenue		
SBC	soustraction / retenue	**Communication :**	
CMP	comparaison	GET	attente du message et lecture
INC	incrémentation	PUT	envoie message interne
DEC	décrementententation	TRY	test arrivée message
TST	comparaison avec 0	SEND	envoie message
CLR	Mise à 0		
MUL	multiplication	**Branchements :**	
		Bcc	branchements conditionnels
Opérations logiques :			
XOR	ou exclusif	**Opérations registres :**	
OR	ou logique	LDI	chargement du registre index
AND	et logique	CLC	mise à 0 de la retenue
NOT	negation	SEC	mise à 1 de la retenue
Décalage & Rotation :		**Opérations Registre à Registres :**	
ROL	rotation gauche	TAPC	transfert A à PC
ROR	rotation droite	TPCA	transfert PC à A
ASR	décalage arithm droite	TIA	transfert I à A
ASL	décalage arithm gauche	TAI	transfert A à I

D'autre part, nous disposons de quelques instructions 16-bit:

Instructions 16-bit :	
LDAW	chargement accumulateur étendu
STAW	stockage accumulateur étendu
ADCW	addition 16-bit /C
SBCW	substraction 16-bit / C

[9]Pour plus de précisions en ce qui concerne le jeu d'instructions de la machine R.C.A.I., nous conseillons au lecteur de se reporter aux travaux de [RUBINI_92] et de [KARABERNOU_93].

La machine R.C.A.I a été dotée, dans sa version précédente, de quatre instructions de communication:

Instructions de communication :

SEND <msg>	envoi d'un message <msg>
GET <addr>	attente d'un message à <addr> ensuite le retirer du réseau de communication
PUT <addr>	génération d'un message à <addr> pour une synchronisation locale
TRY <addr>	teste la présence d'un message à <addr>, recommencer sinon

1.2.3. La partie routage

Le système de communication a été conçu autour de deux parties:

- le récepteur et
- l'émetteur (arbitre).

Les cellules communiquent entre elles par passage de messages. La structure du message inclut quatre champs (figure1.16):

- l'adresse relative (Dx, Dy), mise à jour à chaque transition d'une cellule à une autre par décrémentation (un message dont l'adresse relative est nulle signifie qu'il est arrivé à destination, et passe ainsi de la partie routage à la mémoire locale de la cellule),
- l'étiquette (TAG) qui correspond à une adresse dans la mémoire locale du processeur où la donnée doit être rangée et
- la donnée à échanger (DATA) .

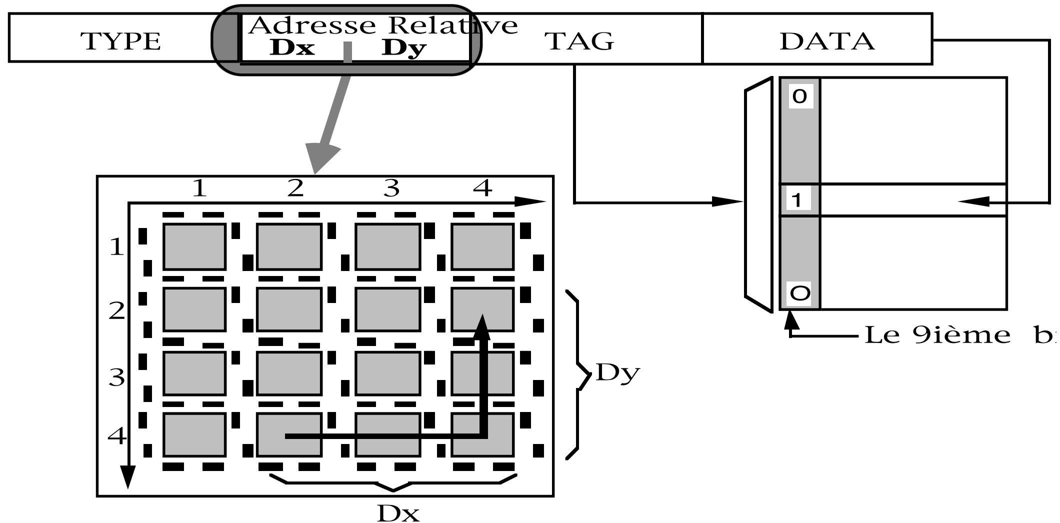


Fig.1.16. Structure d'un message.

Remarque: Le type du message fait partie de la dernière version de la machine R.C.A.I. Il sera explicité le moment venu.

1.2.3.1. La stratégie de routage

Il existe principalement deux modes de routage dans le cas général:

- le routage adaptatif (ou semi-adaptatif), et
- le routage déterministe.

Le premier mode présente l'avantage de tenir compte de l'engorgement du réseau de communication. Lorsqu'il n'y a aucune restriction sur les liens de sortie à partir d'un nœud nous parlons d'un routage **adaptatif**. Autrement, lorsque le nombre de liens de sortie autorisés est restreint, le routage est dit **semi-adaptatif**.

A l'opposé, le routage **déterministe** définit, d'une manière fixe, les liens de communication entre les différents processeurs, comme son nom l'indique.

La différence entre ces deux modes est la complexité engendrée par le maintien de l'état du réseau dans la première approche. De ce fait, son intégration (au niveau silicium) demande davantage de surface que la méthode retenue dans la machine R.C.A.I. Dans le cas d'une machine massivement parallèle, la partie routage doit consommer le moins possible de silicium (de l'ordre de 20 à 30% de la surface totale).

La stratégie de routage adoptée dans R.C.A.I est déterministe. Il s'agit de la technique dite du ***wormhole*** [SEITZ84] [DALLY87]. Sommairement, cette méthode consiste à diviser un message en plusieurs paquets avant de les introduire dans le réseau de communication: commutation de paquets. La tête du message creuse un “tunnel” pour le reste des paquets. Ce passage est refermé par la queue pour libérer les liens de communication aux prochains messages. De cette façon, il y a un recouvrement entre les différents paquets. Chaque message est réparti sur plusieurs cellules successives (suivant le nombre de paquets qui le composent).

Cette stratégie de routage a été implantée au niveau de l'**aiguilleur** dans la machine R.C.A.I (figure1.15).

1.2.3.2. Interblocage et famine dans R.C.A.I

Nous présentons les règles suivantes, inhérentes au réseau de cellules R.C.A.I:

① Les tampons d'entrée au niveau d'une même cellule sont inspectés successivement en vue d'être lus, toujours dans le même ordre.

② Un message qui transite dans un sens au niveau d'une cellule ne peut en bloquer d'autres qui transiteraient dans les autres sens (mais bloquent toutes les communications dans ce même sens).

③ Un message est toujours acheminé d'abord selon son déplacement en abscisse (d_X), puis en ordonnée (d_Y).

④ Un message arrivé à destination est immédiatement retiré du réseau (rangé en mémoire).

⑤ Les messages présentent le même nombre de paquets (même longueur).

⑥ Les paquets qui constituent un message empruntent tous le même chemin au travers des cellules.

⑦ Lorsqu'un message a été sélectionné pour être acheminé vers sa prochaine destination, il doit être traité en entier avant de passer au suivant (s'il y en a un en attente).

Ces règles nous permettent de dégager deux caractéristiques de notre réseau:

- la première est l'absence de **famine**; en effet, la règle (1) assure qu'au bout d'un temps fini, chaque entrée, au niveau d'une même cellule, sera prise

en compte, et à condition que la seconde caractéristique (règle 2) soit vérifiée (comme nous le montrerons par la suite),

- la seconde caractéristique est l'absence d'**interblocage** ; pour cela, définissons d'abord cette notion. Nous dirons qu'il y a interblocage lorsqu'un groupe ou l'ensemble des cellules n'arrivent plus à acheminer les messages sur leur entrée vers les cellules voisines. En général, ceci revient à dire qu'il existe un circuit dans le réseau.

Avant de montrer l'inexistence d'un tel cycle, nous allons considérer quelques définitions.

Soit G (C, L) un graphe orienté dont l'ensemble des sommets C correspond à l'ensemble des cellules:

$$C = \{c_i \mid i \in [0 , (n-1)^2]\}$$

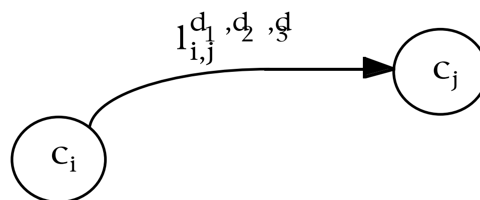
n^2 étant le nombre de cellules du réseau (de n×n cellules).

L est l'ensemble des arcs, tel que:

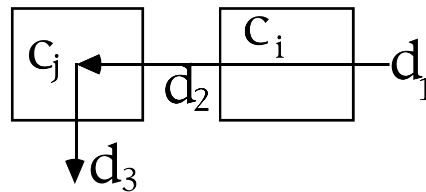
$$L = \{l_{i,j}^{d_1,d_2,[,d_3]} \mid (i,j) \in [0,(n-1)]^2 \text{ et } (d_1,d_2,d_3) \in \{N,S,W,E\}, \ d_1 \neq d_2 \neq d_3\}$$

N:*nord*, *E*:*est*, *W*:*ouest*, *S*:*sud*

Nous dirons qu'il existe un lien $l_{i,j}^{d_1,d_2,d_3}$ entre deux cellules c_i et c_j, lorsqu'un message au départ de la cellule c_i (ou seulement de passage au niveau de celle-ci), selon une direction (d_1, d_2) arrivant à la cellule c_j doit changer de direction en d_3.



Remarquons que la direction d_3 peut être éventuellement vide lorsqu'une des deux directions d_x ou d_y du déplacement est nulle (ou que l'on ne change pas de direction).



Les directions d_1, d_2, et d_3 reflètent donc les déplacements en abscisse et en ordonnée, comme par exemple:

$$d(d_1,d_2) = d_x \quad \& \quad d(d_2,d_3) = d_y$$

Le choix d'un tel modèle est justifié par le fait que, dans le cas qui nous concerne, une communication est décrite par un couple de déplacements (d_x,d_y).

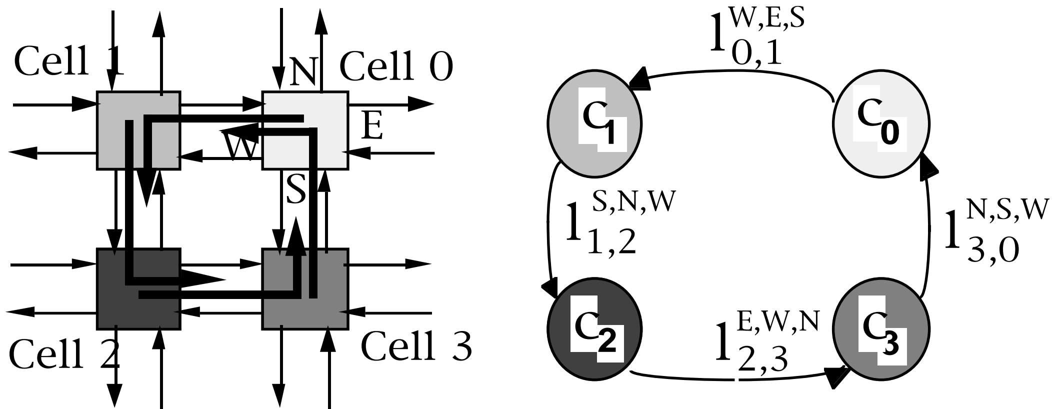
Trois nœuds importants interviennent donc dans cette description:

- le nœud de départ du message,
- le nœud d'arrivée,
- et surtout le nœud au niveau duquel s'opère le changement de direction (d_x en d_y ou d_y en d_x).

De plus, la règle(2) spécifie que deux communications distinctes dans deux sens différents ne peuvent interférer.

Par conséquent, pour mettre en évidence un cas de conflit, il faudrait choisir un modèle qui reflète la concurrence pour l'accès à une ressource critique par deux ou plusieurs nœuds à la fois: à savoir les tampons d'entrée/sortie des cellules en question.

Un circuit typique serait donc:



Ce cycle peut être traduit dans notre modèle par la suite d'arcs:

$$\text{cycle} = \left(l^{W,E,S}_{0,1},\ l^{S,N,W}_{1,2},\ l^{E,W,N}_{2,3},\ l^{N,S,W}_{3,0}\right)$$

Un tel cas ne peut se produire dans notre réseau eu égard aux règles(2) et (3). En effet, un cycle de ce type signifie que l'on procède à un moment donné à un déplacement (d_y, d_x). Or ce genre de déplacement ne peut avoir lieu d'après la règle (3). Les deux éléments $l^{S,N,W}_{1,2}$ et $l^{N,S,W}_{3,0}$ ne sont pas légaux car les changements de direction (S,N) et (N,S) ne sont pas permis d'après cette même règle.

Par conséquent, notre réseau, comme tous les réseaux qui réalisent un routage incrémental, suivant une dimension toujours croissante, est exempt de cas d'interblocage. De plus, notre réseau garantit l'absence de famine grâce à la première règle.

1.3. Conclusion

Rappelons brièvement les motivations qui ont présidé à ces travaux ainsi que les objectifs que nous nous sommes fixés.

Parmi les motivations, citons les plus importantes:

- tout d'abord, économiser le temps de chargement du code sur une machine massivement parallèle,
- traiter la classe des problèmes dits irréguliers (telle que l'IA),
- gérer efficacement les ressources du système (par le biais de l'allocation et de la restitution dynamiques des ressources).

Dans le cadre du projet R.C.A.I, les objectifs que nous nous sommes assignés peuvent être résumés dans les points suivants:

- la spécification d'un modèle de programmation: le modèle acteur; ce modèle, par les concepts sous-jacents, s'adapte tout naturellement à la programmation de notre réseau,
- la mise en œuvre de ce modèle au niveau de la machine (qui correspond au modèle d'exécution),
- l'allocation dynamique de ressources, ainsi que leur récupération,
- et finalement apporter une solution à quelques limites héritées de la version statique de R.C.A.I.

Chapitre 2
Modèles de programmation & d'exécution

2. MODÈLES DE PROGRAMMATION ET D'EXÉCUTION

Le projet R.C.A.I. nous a légué un certain nombre de contraintes, sur lesquelles est construite notre solution, ainsi:

- l'architecture du réseau est une grille 2D ouverte,
- les processeurs sont asynchrones,
- il n'existe pas de références globales entre les processeurs; les distances sont relatives et exprimées par des déplacements en abscisse et en ordonnée,
- il n'y a qu'un seul moyen de communication: le passage de messages,
- la durée de transmission est imprévisible,
- le grain de programmation est fin.

En prenant en compte toutes ces spécificités, nous allons passer en revue les modèles qui peuvent s'adapter à une gestion dynamique de ce réseau.

Avant d'entamer cette étude, nous définissons la notion de modèle d'exécution et le modèle physique de R.C.A.I.

Le modèle logique est présenté en plusieurs parties (à travers les différents chapitres de ce manuscrit). D'abord en définissant le modèle de programmation correspondant; puis en traitant les aspects dynamiques d'allocation et de restitution d'objets. En ce qui concerne le modèle logique, le point le plus important est la définition de l'espace d'adressage d'un acteur dans notre contexte. Il s'agit de ce que nous désignons par *serveur de code* (cf.§4.1).

2.1. Définition d'un modèle d'exécution

Un modèle d'exécution consiste en un ensemble d'abstractions qui fournit une vue simplifiée d'une machine.

Une machine doit comporter un minimum de mécanismes qui permettent de supporter un ou plusieurs modèles d'exécution.

Il existe des mécanismes capables de supporter divers modèles d'exécution parallèle. Ces derniers nécessitent, de façon globale, des mécanismes efficaces pour:
- la communication,
- la synchronisation,
- ainsi que la désignation.

Un autre mécanisme prend de plus en plus d'ampleur dans la spécification de divers systèmes, et semble s'imposer en tant qu'élément de base à fournir dans tout système pour faciliter la programmation au niveau le plus haut. Il s'agit du mécanisme de "*ramasse-miettes*" connu sous le nom de **garbage collector**.

Un mécanisme est dit **universel** lorsqu'il peut être utilisé pour supporter la majorité des modèles d'exécution existants. De même, un mécanisme est dit **primitif** lorsqu'il peut être implémenté d'une manière efficace au niveau physique [DALLY89].

Ces mécanismes permettent de ne pas s'attacher aux aspects physiques de la machine lors de la spécification du modèle d'exécution de plus haut niveau. Dans le cas qui nous concerne, l'ensemble de ces mécanismes doit être mis en œuvre dans un contexte de machine parallèle.

L'interface de machine parallèle (**IMP**) discutée dans [DALLY89] correspond à un ensemble de mécanismes universels reliant les systèmes de programmation aux machines (figure2.1).

L'IMP facilite la portabilité des systèmes de programmation parallèles. Cette interface sépare ainsi les problèmes liés à la machine (implémentation des mécanismes primitifs) de ceux liés aux modèles de programmation (niveau logiciel).

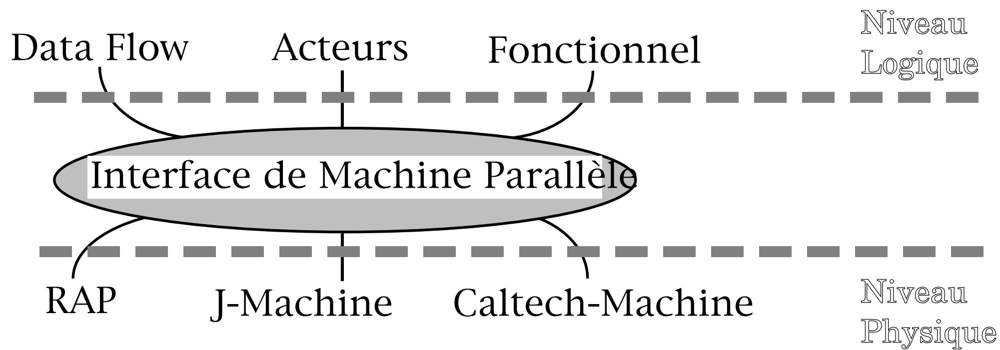


Fig.2.1. L'Interface de Machine Parallèle (IMP).

Nous présentons maintenant le modèle physique relatif à notre machine R.CA.I.

2.1.1. Modèle physique

Ce modèle inclut un ensemble de mécanismes suffisants pour supporter les modèles d'exécution les plus connus (modèle acteur, modèle dataflow, modèle fonctionnel...):

- la machine abstraite est composée d'un ensemble **N** de nœuds; un nœud i de cet ensemble est noté N[i],

- chaque nœud contient une mémoire locale **M**; cette mémoire est constituée d'un ensemble de mots M[0],...,M[k-1] (k est donc la taille de cette mémoire),

- un mot mémoire peut contenir une valeur, une référence (pointeur - adresse) ou une valeur particulière ⊥ (nil). La valeur ⊥ signifie "*valeur indéfinie ou mot mémoire en attente de donnée*". Nous reflétons cette opération au niveau de R.C.A.I en associant à chaque mot mémoire un neuvième bit de validité [KARABERNOU93] [RUBINI92]. Si ce bit est positionné à 0, la valeur de ce mot mémoire correspond à ⊥. S'il est positionné à la valeur 1, la case mémoire correspondante contient la dernière valeur reçue (figure1.16).

Ainsi, nous écrivons:

> M[k].valid ← 0, pour suspendre une tâche et la mettre en attente d'une donnée,
>
> M[k].valid ← 1, pour reprendre la tâche en question,

- une tâche sur un nœud N[i] peut:
  - modifier le contenu de sa zone mémoire,
  - envoyer un message à un autre nœud,
  - se mettre en attente d'un ou de plusieurs messages (attente active).

- si une tâche tente de lire la valeur ⊥, elle suspend son traitement, et se met en attente de l'arrivée d'une donnée dans cette zone.

2.2. Modèles centrés objets

Nous reprenons à ce niveau l'étude de modèles de programmation potentiels pour la gestion dynamique de R.C.A.I.

Les langages à objets peuvent être classés suivant la définition attribuée à la notion d'objet [FERBER87] [MASINI89]:

- La perspective **structurelle**: dans cette famille, l'objet est considéré comme étant un type de données qui définit un modèle pour la structure de ses représentants "physiques" et des opérations applicables à cette structure. Les langages qui appartiennent à ce premier groupe sont appelés *langages de classes*. La classe s'apparente à un type abstrait de données et décrit un ensemble d'objets qui partagent la même structure et le même comportement. Elle sert de moule pour générer ses représentants physiques (dits *instances*).

- La perspective **conceptuelle**: l'objet est une unité de connaissance qui représente le prototype d'un concept. Les langages appartenant à cette famille sont appelés *langages de frames*. Un frame est un prototype qui décrit une situation ou un objet standard. L'ensemble des frames est organisé en une hiérarchie d'héritage où, à la différence des classes, tout objet est à la fois un représentant des frames dont il est issu et un générateur de frames plus spécialisés. Les langages de cette catégorie sont essentiellement utilisés pour la représentation de connaissances [MINSKY75].

- La perspective **acteur**[10]: dans cette dernière catégorie, l'objet est une entité autonome et active. Cette famille regroupe tous les *langages dits d'acteurs*, issus des travaux menés au M.I.T par C.HEWITT [HEWITT77]. Un *acteur* est un objet *actif* qui communique avec les autres acteurs par envois de messages asynchrones (contrairement aux deux précédentes classes).

[10]Le modèle agent est une extension du modèle acteur comme nous le verrons au §2.2.3.

2.2.1. Définition du modèle objet

2.2.1.1. Définition d'un objet

Le terme de **programmation orientée objets** ("*object-oriented programming*") désigne un mode de conception de logiciel caractérisé par:

- une organisation en entités relativement autonomes (les "**objets**") qui utilisent une forme particulière d'interaction mutuelle (les "**messages**"),
- un mécanisme de spécification et de génération des objets au moyen de modèles génériques qui décrivent leur structure et leur comportement (les "**classes**").

2.2.1.2. Classes et instances

Les objets qui présentent les mêmes spécifications sont rangés dans une même représentation: la **classe**. La structuration en classes est une représentation d'un type abstrait. C'est à partir de ce moule que nous générons des exemplaires. Ces exemplaires sont appelés **instances**. L'opération de génération est appelée **instanciation d'objets**. Toutes les instances d'une classe peuvent présenter les mêmes caractéristiques définies par cette classe.

Il existe deux types d'instanciation:

- la méthode classique: construction d'un objet à partir des spécifications de sa propre classe (figure2.2),

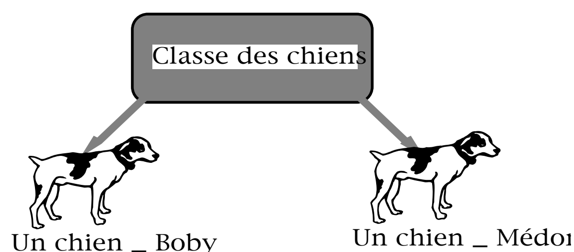


Fig.2.2. Instanciation.

- la construction d'un objet terminal par copie à partir d'un prototype en changeant éventuellement la valeur de champs, d'où le nom de **copie différentielle**. Tout objet est considéré comme une classe dans cette approche (figure2.3).

Fig.2.3. Copie différentielle.

2.2.1.3. Héritage

La spécialisation des classes définit la notion d'héritage (en réalisant une hiérarchie en classe et sous-classe). Plusieurs sous-classes peuvent hériter des méthodes (et des attributs) des classes supérieures. Chacune de ces sous-classes peut ajouter à son tour de nouvelles méthodes (ou de nouveaux attributs) qui lui sont propres ou redéfinir celles (ou ceux) dont elle a hérité.

[BAILLY87] donne la définition suivante de l'héritage:

"...les deux notions d'héritage et d'instanciation sont parfois confondues en un même mécanisme...il convient toutefois de les différencier car ils n'ont pas la même signification. Une relation classe-instance est assimilable à un lien d'appartenance élément-ensemble. L'héritage doit plutôt être considéré comme représentant une inclusion entre un ensemble et un sous-ensemble."

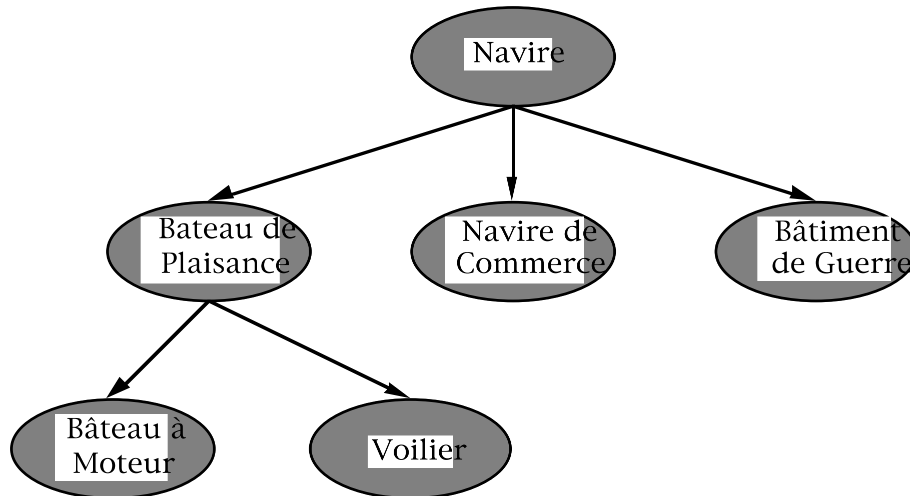


Fig.2.4. Notion de classe/sous-classe et d'héritage.

Il existe deux types de recherche de méthodes héritées:

- recherche *statique*: chaque classe contient un dictionnaire complet de toutes les méthodes (locales ou héritées); chaque entrée dans le dictionnaire pointe directement le code de la méthode correspondante,
- recherche *dynamique*: la recherche de la méthode héritée se fait par exploration de l'arbre d'héritage des classes (en remontant dans la branche jusqu'à trouver l'information sollicitée).

2.2.1.4. Activité d'un objet

Les seules opérations permises sur un objet sont:

- le changement de l'état de l'objet,
- l'envoi de messages,
- et la création de nouveaux objets.

Les objets sont normalement au repos, en attente de l'arrivée d'un message, d'où la notion de passivité des objets.

Un objet ne peut traiter qu'un message à la fois. Le nombre de messages envoyés ainsi que le nombre d'objets créés en réponse à la réception d'un seul message est fini.

2.2.1.5. Problèmes liés au modèle objet

La mise en œuvre de la programmation par objets pose divers problèmes qui sont encore largement ouverts:

2.2.1.5.1. Gestion de la mémoire

Les problèmes de gestion de la mémoire sont ceux de tout système comportant des entités créées et détruites dynamiquement (gestion d'un tas, ramasse-miettes). Ils sont compliqués par la possibilité de partage d'objets ou de parties d'objets (notamment les descriptions de classes) et par la nécessité de liaison dynamique inhérente au mode d'exécution spécifié, qui amène à conserver des informations de liaison jusqu'au moment de l'exécution. Ce problème n'est pas spécifique au modèle à objets. Il est commun aux trois modèles décrits dans cette section.

2.2.1.5.2. Parallélisme et synchronisation

Le modèle général ci-dessus ne fournit pas d'explications quant au parallélisme d'exécution entre les objets, bien que la terminologie (en particulier le terme "message") puisse suggérer une exécution parallèle. La plupart des langages développés jusqu'ici utilisent en fait un modèle séquentiel: Smalltalk [GOLDGBERG83], Ada [11], ou Flavors [WEINREB80]... Le parallélisme n'a été effectivement abordé qu'à partir du modèle d'acteurs.

[11] Ada n'est pas un langage à objets à proprement parler. Certains auteurs le considèrent comme tel eu égard à ses propriétés d'encapsulation, en regroupant dans une même entité (*package*) les données et les procédures qui les manipulent. Ceci n'est toutefois pas une règle, seulement une possibilité dans ce langage.

2.2.2. Définition du modèle acteur

Les techniques de programmation évoluent, et se tournent de plus en plus vers la réalisation de systèmes concurrents. Les langages d'acteurs, formés de petites entités **autonomes** qui communiquent par **envois de messages**, offrent alors un cadre simple et pratique pour développer des applications.

La programmation qui en découle transforme radicalement notre vision de l'informatique en ne concevant plus les programmes comme des monstres monolithiques mais comme des communautés de petits spécialistes qui, par interaction et dialogues, aboutissent au résultat souhaité.

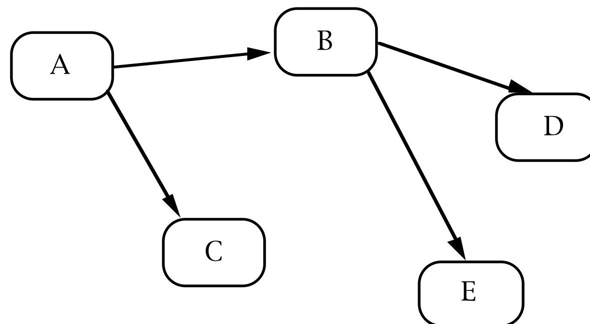


Fig.2.5. Structure d'acteurs.

2.2.2.1. Définition d'un acteur

Le terme d'**acteur** est utilisé par certains auteurs comme synonyme d'objet. Il exprime une notion d'action alors que le mot objet est passif.

La théorie des acteurs a été développée au M.I.T par C.HEWITT [HEWITT77]. Son but est de modéliser une communauté d'experts indépendants et autonomes. Ces experts (les acteurs) sont des entités cohérentes qui possèdent leur propre environnement et qui communiquent par message. Les acteurs se reproduisent par duplication d'eux-mêmes (copie identique) ou par *copie différentielle* (copie originale enrichie par de nouvelles caractéristiques).

Ce modèle introduit des concepts nouveaux par rapport aux autres systèmes d'objets dans le domaine de l'héritage, de l'instanciation et du passage de message (délégation, continuation...).

Un acteur est d'une façon générale composé:

- des données locales, appelées **accointances**, qui correspondent aux acteurs que celui-ci connaît explicitement,
- d'un comportement (**script**), qui définit les actions que l'acteur entreprend au cours de son existence, en fonction des événements qui surviennent. Contrairement à une classe, le comportement n'est pas défini par des méthodes mais par un *script*, unique, qui *filtre* les messages reçus par l'acteur et active la partie appropriée du comportement.

Les capacités d'un acteur sont limitées. Il est incapable de résoudre des problèmes importants à lui seul. Toute la puissance des systèmes d'acteurs tient plus aux capacités de communication entre acteurs qu'à des performances intrinsèques liées à chacun d'entre eux.

En fait, un acteur ne sait presque rien faire, si ce n'est répondre à des messages par d'autres messages ou en donnant naissance à d'autres acteurs.

L'objectif de cette famille de langages est de proposer un langage de programmation d'ordinateur multiprocesseur. Il est très orienté vers l'aspect multitâches et parallélisme.

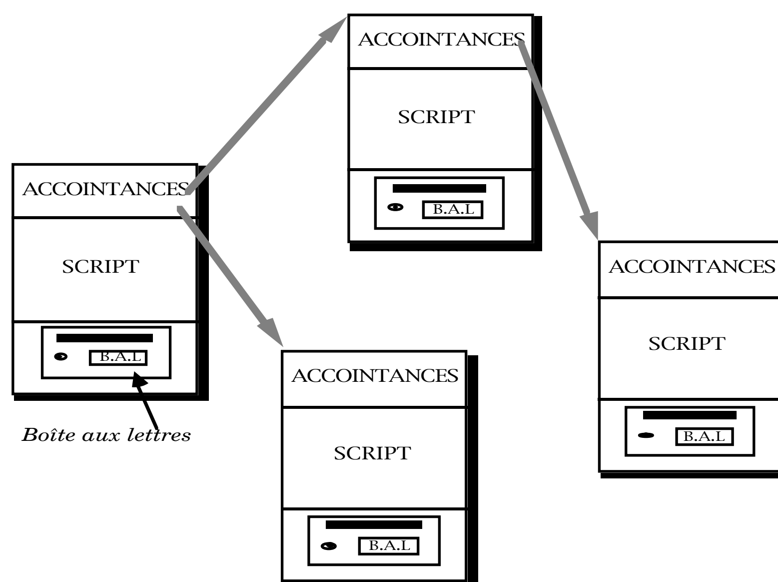


Fig.2.6. Modèle acteur.

2.2.2.2. Délégation

Les acteurs sont caractérisés par un **mandataire**. Le mandataire est l'acteur auquel on renvoie tous les messages incompris.

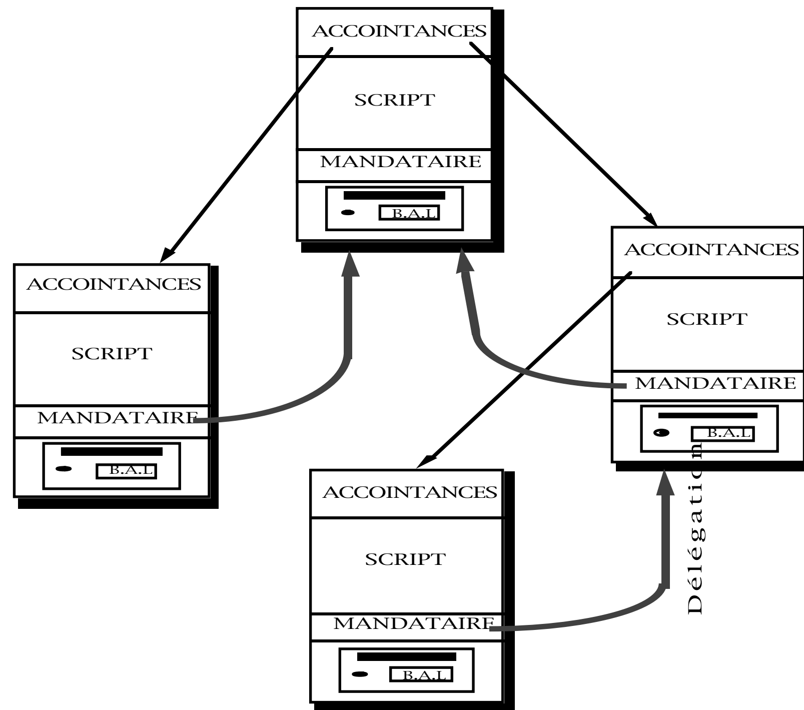


Fig.2.7. Notion de délégation.

Cette caractéristique permet de réaliser la notion d'héritage dans le modèle d'acteurs.

Le mandataire (appelé également *proxy*) correspond dans la hiérarchie d'héritage au père de l'acteur en question. C'est un héritage dynamique. Ce n'est qu'au moment de l'exécution qu'il y a passage de message du "fils" vers le "père", pour la prise en charge d'un message incompris. Il n'existe pas de notion de dictionnaire de méthodes, construit statiquement à la compilation.

2.2.2.3. Continuation

Une première idée introduite par C.HEWITT est la **continuation**: la réponse à un message n'est pas (forcément) retournée à l'envoyeur mais à un tiers (une accointance), spécifié par le message lui même. Ce modèle se rapproche de l'idée de processus, et s'adapte parfaitement à la programmation multitâche ou multiprocesseur.

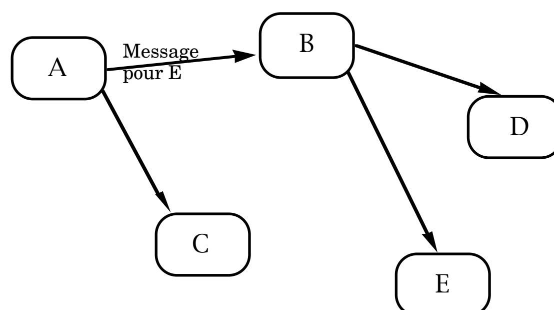


Fig.2.8. Opération de continuation.

Un autre aspect marquant vient de l'uniformité des concepts. Instances, classes, métaclasses sont remplacés par un unique concept: l'acteur. Les messages eux-mêmes sont des acteurs. Les mécanismes d'instanciation et d'héritage sont différents.

2.2.2.4. Instanciation

L'instanciation est en fait une recopie d'un acteur prototype (***copie stricte ou différentielle***). La notion d'héritage, comme nous l'avons écrit plus haut, n'existe pas dans le modèle acteur. Elle y est remplacée par le mécanisme de **délégation**. C'est un moyen différent de partager la connaissance entre plusieurs acteurs: l'héritage classique (à la Smalltalk [GOLDBERG85]) est un mécanisme câblé dans le langage alors que la délégation est un passage de message.

2.2.2.5. Sélection de méthodes

La dernière originalité de ce modèle est la généralisation des sélecteurs. Les messages ne sont pas reconnus d'après un mot clé (*sélecteur*) mais par **filtrage** ("*pattern matching*"). Ainsi, contrairement au modèle à objets, ce n'est plus le sélecteur qui déclenche la méthode approprié. Considérons par exemple les deux méthodes suivantes:

<**Concatener** x y> & <**Concatener** x y z>

La seconde méthode n'est déclenchée que s'il y a correspondance entre le message en entrée et le "prototype" qui la définit. Le message suivant <Concatener "chaîne1" "chaîne2" "chaîne3"> déclenche donc la méthode <Concatener x y z>.

L'interface d'un acteur, appelée **intention**, définit le ***contrat*** que l'acteur a passé avec le monde extérieur. L'acteur est seul responsable de la

manière dont il remplit son contrat. Il ne connaît que l'intention de ses accointances, qui peuvent être considérées comme une abstraction de sa représentation physique. Nous retrouvons donc la notion classique d'abstraction de données: un acteur est défini par son comportement et non par sa représentation physique.

2.2.2.6. Activité d'un acteur

La structure des acteurs peut être étendue à la décomposition suivante en trois sections:

- la première (**Données**) contient les données locales qui définissent son état,
- la seconde (**Comportement**) décrit son comportement de base, ce qu'il ne cesse de faire,
- et la troisième (**Méthodes**) caractérise ses réactions à la réception de messages.

Cette dernière définition s'éloigne de celle proposée par C.HEWITT, mais semble se rapprocher davantage du modèle de classe, enrichie par la notion d'activité des acteurs.

Un acteur qui n'a pas de comportement propre est dit **passif**: sa seule préoccupation est de répondre à des messages qui lui sont envoyés.

Les acteurs, dit **actifs**, agissent indépendamment des communications qu'ils entretiennent avec d'autres acteurs.

Les méthodes peuvent être considérées comme de petites procédures locales à l'entité et déclenchées lors de la réception d'un message. L'effet de ces méthodes est soit de modifier les valeurs des données locales, soit d'envoyer d'autres messages.

Quelle que soit la structure adoptée pour décrire un acteur (en deux ou trois sections), la valeur d'une accointance n'est jamais modifiée par une affectation, mais grâce à un *changement d'état* (en règle générale, l'instruction ***become***).

Pour conclure ce modèle, nous dirons que l'ensemble des notions présentées dans ce paragraphe augmente considérablement la souplesse de la programmation. Mais il faut être conscient que, dans bien des cas, cette

souplesse nécessite de la part du programmeur une grande rigueur. Il n'est guidé, en effet par aucune méthodologie. D'autre part, la souplesse se paie souvent en temps de calcul.

Le modèle d'acteurs est suffisant pour aborder une large classe d'applications (système d'intelligence artificielle, simulation de systèmes complexes...). Cependant, d'autres types d'applications, tels que les systèmes complexes d'intelligence artificielle, nécessitent un enrichissement de ce modèle par la notion de connaissance: c'est le modèle d'agents.

2.2.3. Définition du modèle d'agents

Les travaux que [KORNFELD79] a mené en collaboration avec C.HEWITT [KORNFELD80] ont dégagé de nouvelles bases quant à l'approche de la résolution distribuée de problèmes. Elle est fondée sur la métaphore de la communauté scientifique. La résolution d'un problème est basée sur la notion de "proposition/réfutation". Elle est le résultat d'une négociation entre plusieurs individus: proposition, objection, contre-proposition, approbation...

Chacun des individus qui intervient dans cette démarche est appelé agent. L'ensemble des agents en coopération constituent alors un système "multi-agents".

2.2.3.1. Définition d'un agent

[FERBER89] donne la définition suivante d'un agent:

> *"On appelle agent, une entité réelle ou abstraite qui est capable d'agir sur elle-même et son environnement, qui dispose d'une représentation partielle de cet environnement, qui, dans un univers multi-agents, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de sa connaissance et des interactions avec les autres agents".*

La différence fondamentale entre le modèle d'agent et le modèle d'acteurs est que, dans le premier, il existe une connaissance (expertise), qui est absente dans le second modèle.

Certains auteurs ajoutent la notion de responsabilité et d'engagement [COHEN90] [BURMEISTER91]. Un agent qui s'engage pour la réalisation d'une fonction, suivant un contrat, a la responsabilité de la mener à terme. Pour cela, il peut coopérer avec d'autres agents.

Dans ce contrat, nous pouvons spécifier des conditions d'engagement (pré-conditions) ainsi que des clauses portant sur les conditions que vérifie le résultat une fois que cet engagement est honoré (post-conditions).

2.2.3.2. Différents types d'agents

D'après la définition précédente, nous pouvons distinguer deux types d'agents [RUSSEL89] [KNIGHT9?]:

- les **agents délibératifs**, les objectifs à atteindre sont explicites et font l'objet d'un raisonnement,
- les **agents réactifs**, dans ce cas, les objectifs sont implicites et définis par les échanges entre les agents.

2.2.3.3. Organisation des agents

[FERBER89] distingue deux types d'organisations d'agents:

- l'**approche sociale**: les agents sont supposés doués d'une "intelligence" et capables d'avoir une perception de leur environnement; chacun des agents a un but qu'il cherche à satisfaire (en tenant compte de la connaissance que d'autres agents lui communiquent),
- l'**approche biologique**: dans cette conception, il n'y a pas de présomption quant à l'intelligence des agents. Comme le souligne [FERBER89],

> *"l'intelligence émerge de l'interaction d'un grand nombre d'agents qui individuellement ne disposent d'aucune intelligence"*.

L'échange de données entre les différents agents détermine la progression dans la résolution d'un problème. Généralement, tous les agents ont une structure et un mécanisme de résolution identiques. Ce sont donc des systèmes primaires où la coopération est le seul lien d'organisation.

2.2.3.4. Protocoles de communication entre agents

La structuration d'un ensemble d'agents en société ou en une organisation primaire, nécessite la définition d'un protocole de communication et d'interaction. Les interactions entre les agents peuvent relever:

- soit d'un protocole de résolution, qui correspond à l'ensemble des interactions qui contribuent à la progression dans la résolution d'un problème; ce sont tous les mécanismes qui portent sur les données;
- soit d'un protocole d'organisation, constitué des interactions mises en œuvre pour la structuration des agents en vue de résoudre des conflits internes d'organisation. Ceci correspond aux attitudes (décisions) des différents agents face à un problème. Par la suite, le protocole de résolution adéquat est appliqué pour la résolution du problème.

2.2.3.5. Granularité et modèle d'agents

On peut dire que l'approche sociale correspond à une structuration à gros grain. Chaque agent exhibe d'importantes capacités: il est capable de prendre des décisions (suivant ses propres intentions) sans manifestation d'événements dans son environnement.

A l'opposé, l'approche biologique se rapproche davantage du modèle connexionniste (réseaux de neurones). Les agents sont de petite taille (grain fin), et généralement en grand nombre [FERBER91]. La relation des agents avec leur environnement est essentielle. Elle détermine, suivant un comportement prédéfini, la progression de la résolution d'un problème.

2.2.4. Conclusion

Il existe une certaine similitude entre les modèle d'acteurs et d'agents. Tous deux présentent une autonomie et un système d'échange par messages. La dichotomie entre acteurs actifs et acteurs passifs a une projection dans le modèle d'agents par la division des agents en agents délibératifs et agents réactifs, respectivement.

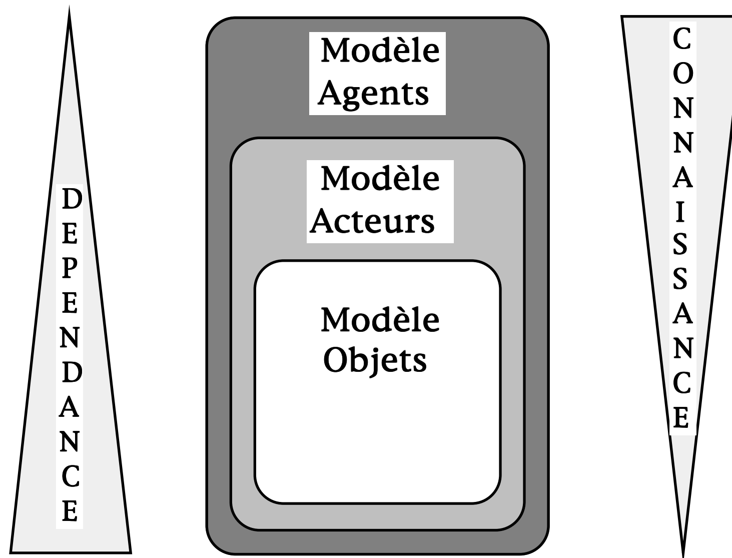


Fig.2.9. Hiérarchie au niveau concepts des modèles orientés objets.

On peut dire que le modèle d'agents est une extension du modèle d'acteurs. Il reprend toutes les caractéristiques du premier et lui additionne des facultés de plus haut niveau. D'ailleurs, historiquement, le modèle d'agents a découlé des travaux concernant les acteurs [KORNFELD80].

En ce qui nous concerne, il existe une bonne adéquation entre le modèle d'acteurs et les spécifications faisant partie du cahier des charges (cf.introduction). Nous nous limiterons donc, dans notre étude, à ce modèle. C'est dans ce contexte que nous allons développer un modèle d'acteurs particulier: le langage CANTOR. C'est ce langage que nous avons sélectionné pour illustrer nos propos. Par la suite, nous explicitons la mise en œuvre du modèle acteur dans l'architecture R.C.A.I.

2.3. Un modèle d'acteurs particulier: CANTOR

CANTOR est un langage qui peut être défini comme une variante (adaptation) du modèle d'acteurs. C'est autour de ce langage qu'a été développé le projet MOSAIC de Caltech [ATHAS86]. Un des résultats de ce projet est l'élaboration du MDP (Message Driven Processor). Cette unité interprète directement les messages et réalise une recherche rapide des méthodes spécifiées dans les différents messages.

Les objets CANTOR sont des "agents" (unités) de calcul indépendants. Ces objets ne communiquent que par passage de messages. Ce langage a été élaboré dans le but de décrire des traitements concurrents.

2.3.1. Les objets CANTOR

Un objet CANTOR est composé:

- d'un ensemble d'accointances (variables locales - privées),
- d'une boîte à lettres pour le stockage des messages, et
- d'un script (comportement) activé en fonction des messages reçus.

Les objets sont normalement au repos, en attente d'un événement pour les activer. Toutefois, les objets dans ce système ne sont pas "éternels". Après avoir traité un message, chaque objet doit:

- ou se positionner dans un nouvel état qui lui permet de recevoir d'autres messages,
- ou s'auto-détruire, et quitter de la sorte le système.

La réaction à la réception d'un message est standard. Ces actions incluent:

- l'envoi de nouveaux messages,
- la création de nouveaux objets,
- le déclenchement d'opérations de base: opérations arithmétiques, affectations...

La communication entre les objets est asynchrone. CANTOR établit une séparation temporelle entre le nom d'un objet et son instance. Une référence à un objet peut avoir lieu avant même sa création effective. Cependant, pour qu'un objet puisse accepter un message, l'objet doit avoir été instancié auparavant.

2.3.2. Structures de contrôle et structures de données dans CANTOR

CANTOR ne fournit aucune structure de données de haut niveau. Les tableaux, structures (RECORD) ou piles, par exemple, sont inexistantes dans ce langage. Ces données doivent donc être construites explicitement à partir d'objets de CANTOR.

De même, les structures de contrôle sont complètement absentes dans ce langage. Les structures de boucle (telle que l'expression *while*) doivent être construites par des opérations de passation de messages.

2.3.3. Structure d'un programme CANTOR

La structure générale d'un programme CANTOR peut prendre l'une des deux formes suivantes[12]:

```
<Nom_Objet> (<Liste_de_Paramètres>)    ::
  *[  <Corps_Progamme_Persistant>
  ]
```

```
<Nom_Objet> (<Liste_de_Paramètres>)    ::
  [  <Corps_Progamme_Temporaire>
  ]
```

La différence entre les deux structures est la longévité des objets qu'elles définissent. Dans la première, l'objet persiste tout au long de l'exécution du programme. Dans l'autre, l'objet s'auto-détruit après exécution de la dernière instruction dans le corps de l'objet.

La liste des paramètres, dont la portée est locale, peut être vide.

[12]Les crochets [et] ainsi que le symbole * font partie de la syntaxe de ce langage.

Le corps d'un programme est généralement constitué d'une suite de blocs de la forme suivante :

```
        <Début_Corps_Programme>
( <Variable_Communication> { , <Variable_Communication>}* )
        <Reste_Corps_Programme>
```

Les variables de communication sont les variables attendues pour engager le traitement qui suit cette expression. Ces variables, ainsi que les paramètres, sont considérés comme des variables globales à la définition en cours (corps du programme).

Le reste du corps d'un programme (respectivement d'un objet), est composé des différentes expressions développées dans le paragraphe qui suit.

2.3.4. Les expressions de CANTOR

2.3.4.1. Déclaration de variables locales

L'expression suivante permet de réserver un emplacement pour recevoir la valeur d'une expression. Ces variables sont locales au bloc en cours (délimité par les deux symboles **[** et **]**).

```
let  <Nom_Variable>   = <Expression>
```

Il faut rappeler que les variables CANTOR sont typées dynamiquement, au moment de leur création.

2.3.4.2. Instruction conditionnelle

La conditionnelle se présente sous la forme classique[13]:

```
if   <Expression_Booléenne>
  then   <Expression>+
  [else   <Expression>+]
fi
```

[13]Les crochets qui entourent l'expression *else* signifient que celle-ci est facultative.

2.3.4.3. Affectation

L’affectation ne peut avoir lieu que dans des variables réservées au moyen de l’instruction **let**, ou faisant partie de la liste des paramètres ou de la liste des variables de communication.

<Nom_Variable> **:=** *<Expression>*

2.3.4.4. Instruction de communication

Il n’existe qu’une seule forme d’instruction pour la communication. Elle est structurée de la manière suivante:

send **(** *<Expression_1>* **)** **to** *<Expression_2>*

La valeur de l’expression qui détermine la destination (Expression_2) doit correspondre à une référence. La valeur particulière **self** pour cette expression dénote une référence à soi-même.

La première expression est une liste de constantes, variables ou références à d’autres objets. Cette liste peut être vide (cas d’un signal).

Les références à des objets peuvent donc être communiqués entre les acteurs, établissant par la même occasion des liens dynamiques (à l’exécution).

2.3.4.5. Répétition

L'itération peut être obtenue au moyen de deux styles de programmation:

- en recourant à l'instruction conditionnelle:

```
repeat_relay  (where, what)   ::
[ (i)
    if    i > 0
      then
        send  (what)   to  where
        send  (i - 1)  to  repeat_relay  (where, what)
    fi
]

[ (console)
    send  (5)  to  repeat_relay  (console, "Hello World")
]
```

- en utilisant l'instruction **repeat**:

```
repeat_relay  (where, what)   ::
[ (i)
    if    i > 0
      then
        send  (what)   to  where
        send  (i - 1)  to  self
        repeat
    fi
]

[ (console)
    send  (5)  to  repeat_relay  (console, "Hello World")
]
```

- en utilisant les objets persistants, combinés avec l'instruction **exit**:

```
repeat_relay  (where, what)   ::
*[  (i)
    if    i > 0
      then
        send  (what)   to  where
        send  (i - 1)  to  self
      else
        exit
    fi
]

[ (console)
    send  (5)  to  repeat_relay  (console, "Hello World")
]
```

Dans ces trois écritures [ATHAS86], il s'agit d'écrire la même chaîne de caractères (en l'occurrence "Hello World") cinq fois de suite.

2.3.5. Un exemple de programme CANTOR

Nous considérons à ce niveau un exemple de programme CANTOR qui a été traduit en assembleur et implanté sur R.C.A.I. Il s'agit du crible d'Erathostène pour la détermination des nombres premiers, dont nous reparlerons au chapitre des évaluations.

```
; Programme principal
  [
    (console)     ; Attente de la variable "console"

      send   (2000) to   Generator   (console, NULL, 3)
  ]
Generator (console, Cell, next)  ; Générateur de nombres
[
  (limite)

    [ if   ( next   < limite )
         then   if   ( cell    ==   NULL   )
                  then   ; Rechercher une cellule libre,
                         ; puis lui affecter le code de Eratos,
                         ; pramètres :console & le nbr premier 2
                    cell   = Eratos (console,   2)
               fi
               send   (next) to   cell   ()
               let next     = next   + 2
               repeat   ; Répeter le bloc le plus interne
      fi
    ]
  ]
Eratos (console,  sieve) ; Code du crible d'Erathostene
[
  let    Cell   = NULL
  [
    (prime)
    if   ( prime  <  0  )
      then   if   ( cell   !=   NULL   )
             then   send   (-1)   to   cell
           fi
           send   (sieve)  to console
      else   let  advance  = prime
             let  unknowm  = prime
             let  Accu     = prime  / sieve
             let  Accu     = Accu   * sieve
             let  Accu     = Accu   - advance
             if   ( Accu   !=   0  )
               then   if   ( cell    ==   NULL   )
                        then   cell   = Eratos (console, unknown)
                    fi
                    send   (advance)   to   cell   ()
             fi
             repeat
    fi
  ]
]
```

2.4. Mise en œuvre du modèle acteur dans R.C.A.I

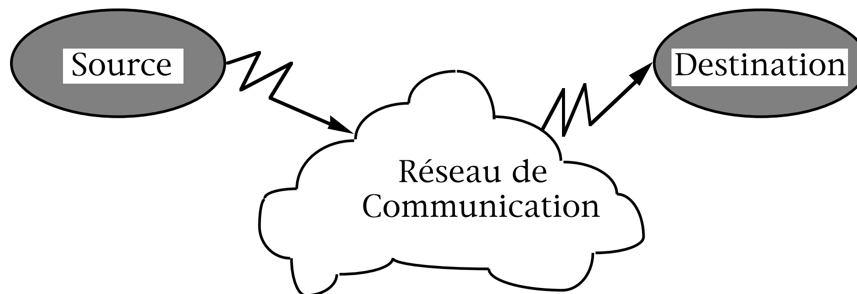
Pour les besoins de cette thèse, nous nous sommes limités dans notre choix à un langage d'acteurs simple, du type CANTOR [ATHAS86]. Comme nous l'avons vu précédemment, ce langage allie les avantages de la programmation par acteurs à la simplicité des moyens mis en œuvre pour sa définition. Ce qui nous importe le plus dans nos travaux, c'est davantage la mise en avant de mécanismes de base pour une implantation efficace des langages d'acteurs, que l'élaboration d'un langage à proprement parler qui serait doté de toutes les nouvelles spécifications qui augmentent la puissance de ce modèle [AGHA93].

2.4.1. Quelques caractères implicites

Certaines caractéristiques des langages d'acteurs se retrouvent implicitement dans notre machine, à commencer par une exécution parallèle, réelle, des différentes entités. Les autres traits sont énoncés dans ce qui suit.

2.4.1.1. Communication asynchrone

Les communications dans la machine R.C.A.I sont, par définition, asynchrones [KARABERNOU93]. L'émetteur et le récepteur ne sont pas tenus de se synchroniser pour établir une communication.



Tout message transféré au réseau de communication est acheminé de proche en proche, suivant la disponibilité des tampons intermédiaires, jusqu'à sa destination. Le message est ensuite rangé en mémoire sans intervention du processeur de calcul.

2.4.1.2. Indéterminisme

L'échange d'information entre deux nœuds du réseau, même voisins, peut prendre des délais variables. Les flux de données circulant sur les liens de communication qui relient ces nœuds ont une incidence directe sur les délais de propagation. Par conséquent, un même programme peut fournir des résultats différents suivant l'ordre d'arrivée des messages aux différents acteurs. De tels systèmes sont donc indéterministes.

2.4.1.3. Fiabilité des liens de communication

Le modèle d'acteurs nécessite un réseau de communication fiable [AGHA86b]. Tous les messages émis doivent être délivrés à leur destination. Aucune perte de message n'est tolérée.

Dans l'état actuel de notre machine, nous sommes dans l'obligation d'assumer l'hypothèse suivante:

Hypothèse

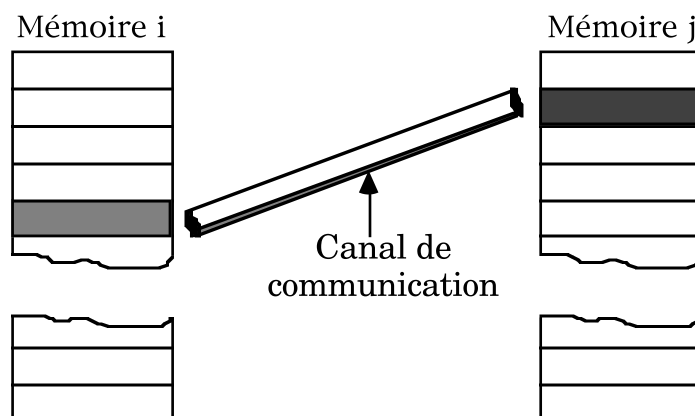
Tous les messages introduits dans le réseau de communication sont délivrés à leur destination, sans perte ni erreur.

2.4.2. Caractéristiques mises en œuvre

2.4.2.1. Système de tampons

Le modèle d'acteurs est basé sur la notion de boîte à lettres. Les messages destinés à un acteur sont rangés dans une file, suivant leur ordre d'arrivée. L'acteur puise dans cette structure les messages avant de leur appliquer l'opération de "*pattern matching*".

Dans notre cas, nous avons conservé le système de communication de la version statique de notre machine. Dans cette version, chaque message intègre dans sa description l'adresse mémoire où il est rangé une fois parvenu à destination. Une position mémoire de ce type est assimilée à un lien de communication virtuel, établi entre les deux acteurs.



Les canaux de communication sont déterminés à la compilation de l’instruction de haut niveau “SEND”:

$$\text{SEND}\left(\begin{Bmatrix}\text{Constante}_i\\ \text{Variable}_i\\ \text{Référence}_i\end{Bmatrix}\left[,\begin{Bmatrix}\text{Constante}_i\\ \text{Variable}_i\\ \text{Référence}_i\end{Bmatrix}\right]^*\right)$$
$$\text{TO}\quad \langle \text{Nom_Acteur}\rangle, \langle \text{Sélecteur}\rangle$$

Cette instruction est traduite par la séquence d’instructions de bas niveau de la figure2.11.

Remarque:

L’éditeur de liens se charge de résoudre les références croisées.

En adoptant cette stratégie nous avons réduit la notion de boîte à lettres à sa plus simple expression: sa taille correspond à l’ensemble de ce que nous avons appelés les variables de communication. Toutefois, le réseau de communication est sollicité pour étendre la notion de boîte à lettres. En effet, les communications sont complètement asynchrones. Il n’y a pas d’attente entre deux communications adressées sur un même canal. Le réseau de communication sert de tampon temporaire pour stocker les messages non encore traités au niveaux des acteurs. Il faut rappeler, qu’à l’arrivée d’un message, celui-ci ne peut être rangé en mémoire que si la case mémoire correspondante est libre. C’est-à-dire, en se référant à notre modèle physique développé précédemment, nous devons avoir:

$$\mathit{Mémoire}\left[\text{Tag}_j\right].\text{valid} \neq \perp$$

Sachant que Tag_j est l’adresse en mémoire du canal de communication.

```
; Acteur Récepteur
        ORG  0
; Déclarations système
Tag_1:  DS   1
...
Tag_i:  DS   1

; Reste du code

        GET  Tag_1
...
        GET  Tag_i
...
        END
```

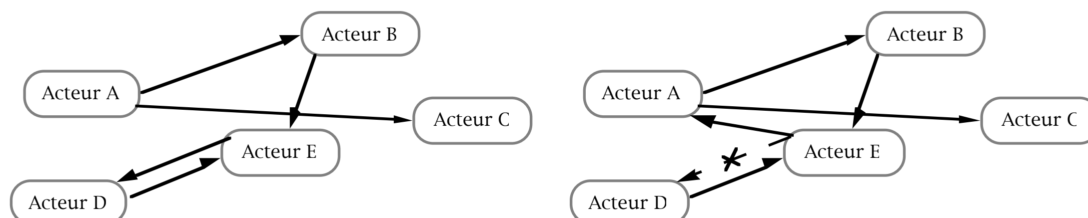
```
; Acteur Emetteur
        ORG  0
; Déclarations système
Var_1:  DC   adr_1
        DC   Tag_1
        DC   val_1
...
Var_i:  DC   adr_i
        DC   Tag_i
        DC   val_i

; Reste du code

        SEND  Var_1
...
        SEND  Var_i
...
        END
```

Fig.2.10. Traduction d'une instruction de communication de haut niveau en assembleur.

2.4.2.2. Gestion des références (accointances)

2.4.2.2.1. Reconfiguration dynamique d'un réseau d'acteurs

Les systèmes à base d'acteurs sont dynamiques. Chaque création de nouvelles entités engendre une structure différente. Cependant, les liens créés dynamiquement peuvent également être modifiés dynamiquement.



Pour réaliser cette opération, l'acteur doit envoyer un message à l'acteur concerné, sur un canal de communication. Ce canal régit une référence à un acteur tiers. A la réception de ce message, l'interface du routeur de la cellule écrase la référence précédante en faveur de la nouvelle. La mise en œuvre de la reconfiguration dynamique est donc assez triviale dans notre système. Toutefois, d'autres considérations nous ont contraint à abandonner pour le moment cette faculté. En effet, le ramasse-miettes actuel est basé sur un compteur de références. Par conséquent, nous ne pouvons pas modifier la structure du réseau d'acteur sans mettre à jour le compteur de références correspondant. L'intégration d'un tel mécanisme nécessite de revoir en profondeur la procédure du ramasse-miettes.

2.4.2.2.2. Continuation

Un trait particulier des langages d'acteurs est la notion de continuité (figure2.12). Le résultat à un message n'est (nécessairement) pas renvoyé à l'émetteur, mais à un autre acteur spécifié dans le message lui-même. Les acteurs impliqués dans cette opération ne sont pas pour autant interrompus. Chacun des acteurs progresse individuellement en fonction des messages et requêtes qu'il reçoit.

Tout comme au niveau du point précédent, il suffit de recourir à un message, adressé à un emplacement particulier de la zone système (associée à chaque acteur dans le réseau). En écrasant l'ancien contenu de cette zone, nous désignons ainsi l'acteur à qui remettre la réponse.

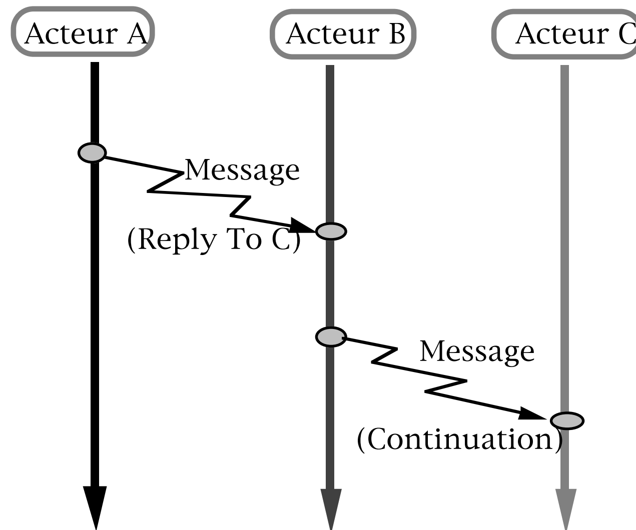


Fig.2.11. Notion de continuité.

2.4.2.2.3. Délégation et "pattern matching"

Le principe de la délégation est de renvoyer tous les messages incompris à l'acteur mandataire.

Dans le cas de langages d'acteurs "réduits", tel que CANTOR, la notion de "pattern matching" n'existe pas. Dans ces conditions, la sélection de méthode se fait par le biais d'un sélecteur (et non plus par pattern matching). De ce fait, il n'y a pas lieu d'intégrer cette notion dans notre contexte. Notre but, il faut le rappeler, n'est pas de proposer un langage d'acteurs en soi, mais d'étudier l'allocation dynamique de ressources dans un environnement massivement parallèle, nécessaire pour ce modèle.

Chapitre 3
Allocation dynamique

3. ALLOCATION DYNAMIQUE

3.1. Régulation de la charge dans les systèmes massivement parallèles

Il a été clair, dès l'avènement des machines massivement parallèles, qu'un mécanisme (adaptatif) pour la régulation de charge était nécessaire pour l'allocation de ressources à ces systèmes dynamiques.

Dans la littérature, de nombreux auteurs se sont longtemps posé la question du choix entre la **régulation de charge** ("*load balancing*") et le **partage de charge** ("*load sharing*") [KRUEGER87b]. Alors que la première approche tente de maintenir une charge égale sur l'ensemble du réseau, la seconde a pour mission de tout simplement prévenir l'inactivité "superflue" des processeurs, en assurant qu'il n'y a pas de processeurs inactifs (ou sous-utilisés) lorsqu'il y a des processus en attente d'exécution. L'utilisation de l'une ou l'autre technique dépend de la connaissance (ou de la prédiction dans certains cas) des paramètres caractérisant le système cible.

Un système global de régulation de charge doit être en mesure d'estimer, d'une quelconque façon, la charge moyenne courante du système (ou une approximation dans un futur proche), pour pouvoir acquérir ou se décharger d'une ou plusieurs tâches afin de se maintenir à la charge moyenne du système.

3.1.1. Quelques définitions

3.1.1.1. Régulation de charge

Pour une utilisation efficace d'un réseau de processeurs, il est nécessaire de distribuer uniformément le travail sur l'ensemble des processeurs. En règle générale, cette opération peut être réalisée d'une manière **statique** ou **dynamique**, par un système **câblé** ou **programmé**. L'équilibrage de la charge d'un système est désigné sous le nom de "*régulation de la charge*".

3.1.1.2. Granularité

La granularité peut être définie comme étant la taille des unités de travail allouées aux processeurs. Les réseaux de processeurs conventionnels sont dits à **gros grain**; le travail y est alloué par programme entier. Il existe des machines dites à **grain fin** (CM2, Manchester DataFlow Machine - MDM, R.C.A.I ...) qui allouent de très petites unités de traitement de quelques instructions seulement, voire d'une seule instruction machine dans le cas de la MDM.

Bien que les machines à gros grain traitent de manière plus efficace de grandes séquences d'instructions, elles n'exploitent pas cependant tout le parallélisme potentiel dans un programme, comme c'est généralement le cas dans les machines à grain fin.

3.1.1.3. Communication

Il est maintenant communément admis que les communications sont souvent le facteur limitant dans les systèmes parallèles. Ces limites peuvent être liées à la **bande passante** qui est le débit maximal d'un canal de communication, et à la **latence,** qui constitue le temps total pour qu'une communication parte de sa source et arrive à sa destination.

Ces limites peuvent causer l'inactivité du processeur ou provoquer un coût supplémentaire (*overhead*) lors de changements de contexte de processus. Nous dirons qu'une machine est "***tolérante à la latence***" si son architecture permet une poursuite d'exécution "naturelle" en attendant une communication (c'est le cas de la machine R.C.A.I).

3.1.1.4. Localité

Pour réduire le coût de communication, il est nécessaire d'exploiter la localité en regroupant ensemble les entités reliées par des liens de communication.

Trois approches ont été adoptées dans ce sens [SARGEANT86]:

- **localité nulle** ; cette catégorie ignore complètement cette notion en partant du fait que cette propriété est trop difficile à exploiter, donc, autant l'ignorer en assurant plutôt une meilleure fonction de communication (telle que le projet Manchester DataFlow Machine),
- **localité partielle** ; dans cette approche la notion de localité est étendue à l'architecture des machines. L'objectif dans cette catégorie est de tenter de regrouper les unités de mémoire et les processeurs "physiquement" par des liens de communication fortement couplés. Le but final est de réduire le temps nécessaire aux «switchs» pour établir les connexions. Les machines visées à ce niveau sont celles possédant un réseau de communication "dense" (tel le Transputer ou la FlagShip),
- **localité totale** ; à ce niveau, les processeurs et les mémoires sont considérées fortement liées et le réseau d'interconnexion "souple", telle une grille par exemple. Dans ce cas, il est primordial d'exploiter les liens de voisinage des différents processeurs pour tirer parti de la notion de localité.

3.1.2. Modes de régulation de charge[14]

3.1.2.1. Mode centralisé

Comme dans tous les systèmes centralisés, c'est un processeur particulier, dit **processeur maître**, qui s'occupe du maintien de l'état global du système pour l'allocation de processus aux autres processeurs, dits **processeurs esclaves**, suivant leur état de charge.

La simplicité de tels algorithmes ne résout pas l'éternel problème du goulot d'étranglement que constitue le processeur central. Un autre problème est celui de la tolérance aux pannes: un tel système serait en complète paralysie suite à une panne du processeur maître. Par similitude avec le problème de la régénération du jeton dans un système réparti, un remède pour la tolérance aux pannes serait de dupliquer le processeur central sur plusieurs autres nœuds. Néanmoins le nombre de messages nécessaires pour la mise à jour de l'état du système sera multiplié par autant de fois qu'il y a de copies du processeur maître. De plus, ce remède nécessitera inévitablement un mécanisme complexe de diffusion sélective ("*one to many*") pour plus de performance.

3.1.2.2. Mode réparti

Dans cette catégorie, un même algorithme est exécuté sur l'ensemble des processeurs. Un réseau composé de ce type de processeurs sera beaucoup plus résistant aux pannes par reconfiguration du système. L'extension de ce dernier peut être facilement appréhendée dans cette approche.

Le problème majeur que pose la répartition d'une information est le maintien de sa cohérence.

---

[14] On pourra trouver une très bonne classification de la régulation de charge dans [XU93] et [GOSCINSKI91]. Nous en proposons une autre à ce niveau.

3.1.2.3. Mode mixte

Une solution intermédiaire consiste à hiérarchiser le système. Chaque groupe dispose de son propre processeur maître.

Cette hiérarchie peut être réalisée de deux manières:

- hiérarchie à plusieurs niveaux, ou **structure de compétition**; dans ce cas chaque niveau est responsable des niveaux inférieur. En cas d'incapacité à satisfaire une demande de service par un niveau inférieur, il y a transfert de celle-ci au niveau supérieur immédiat. On aboutit ainsi à une gestion arborescente de demandes d'allocation de processeurs.

- hiérarchie à un niveau, ou **structure de coopération**. Dans ce cas, le système sera divisé en plusieurs partitions. Chaque partition est indépendante des autres. En cas d'échec pour l'allocation d'une ressource, une demande de coopération sera diffusée vers les maîtres des autres partitions.

Lorsque la profondeur de l'arborescence dans le premier cas, ou le nombre de partitions dans le second cas, augmente, la duplication des nœuds de maintien de l'état des processeurs de chaque niveau ou partition devient un sérieux problème pour la tolérance aux pannes.

Toutefois lorsque le nombre de ces entités augmente, la taille de l'information à maintenir diminuant, leur duplication sera toujours possible. En effet, nous pourrons supposer que ces algorithmes pourront cohabiter avec des processus esclaves. Ils ne monopolisent plus les processeurs. De surcroît, plus le nombre de partitions ou niveaux augmente, plus le nombre de copies de processeurs maîtres nécessaire pour la tolérance aux pannes diminue.

3.1.3. Modes de transfert de processus

La régulation de charge peut agir de deux manières:

- en opérant sous le mode **non préemptif**, à ce moment on réalise un **placement**[15] de processus. La décision de transfert n'est prise qu'à la naissance du processus, suivant l'état de charge du processeur qui le supporte. Dans ce cas, une fois alloué à un processeur, le processus ne pourra plus migrer ailleurs, assurant ainsi une certaine stabilité du système.

- en opérant sous le mode **préemptif**, et on parle alors de **migration** de processus. La décision de transférer un processus peut affecter même un processus en cours d'exécution.

La migration est beaucoup plus coûteuse que le placement, puisque le contexte du processus, qui doit l'accompagner vers le processeur cible, devient beaucoup plus complexe après le début d'exécution. Ceci ajoute donc un volume d'informations supplémentaires qui doivent accompagner le processus.

L'opération de migration doit rester transparente vis à vis des processus qui communiquent avec ce dernier (nous ne devons pas redéfinir l'espace d'adressage de ceux-ci). Dans la cas contraire, il faudra prévoir un mécanisme de ré-acheminement des messages adressés à ce processus, tel le **mécanisme de relais** que nous proposons dans la machine R.C.A.I (se reporter au paragraphe4.3).

La question posée à ce niveau est de savoir [KRUEGER88], vu les coûts de gestion occasionnés par la complexité du mécanisme de migration de processus, si le placement à lui seul ne peut pas réguler la charge du système?

La réponse a été que la migration ne peut apporter de meilleures performances que dans le cas où le système comporte beaucoup plus de processus que de nœuds.

---

[15]Dans le sens où un processus est alloué une fois pour toutes à un site. Le mode préemptif réalise également un placement de processus. Il s'agit juste de faire la différence entre les deux modes.

3.1.4. Mode d'échange d'informations

Le maintien de la cohérence des données d'un algorithme de régulation de charge repose sur le type d'échange d'informations entre les différents processeurs [EAGER86].

Cet échange peut être (cf.les diagrammes3.1 et 3.2):

- **actif** :dans ce cas, lorsqu'un processeur désire prendre une décision de placement de processus, il va interroger les processeurs concernés sur leur propre état (***sender-initiated***),
- **passif** : à ce moment, lorsqu'un changement intervient au niveau d'un processeur, celui-ci le communique automatiquement aux processeurs concernés (***receiver-initiated***).

Ces deux modes d'échange d'information induisent un échange de données trop fréquent, pas toujours nécessaire pour prendre une décision.

Une solution intermédiaire serait de réaliser un échange **périodique**. Ce mode d'échange n'est valable que dans le cas du mode passif, puisque laissé à l'initiative du processeur où s'opère le changement d'état.

Dans cette variante on ne communiquera un nouvel état que lorsque celui-ci a changé d'une manière significative, eu égard à une fonction définie (taux relatif, suite exponentielle... [KUCHEN91]).

Dans le diagramme 3.2, le voisinage d'un nœud peut être défini soit par l'ensemble des nœuds qui lui sont connectés physiquement, soit par un critère de proximité logique (tous les nœuds de distance inférieure ou égale à deux, par exemple).

En ce qui concerne la migration de processus, toujours dans le même diagramme, il faut s'assurer de la stabilité du système. En effet, il faut éviter qu'un processus ne migre perpétuellement d'un nœud vers un autre. Une politique de désignation de candidats à l'immigration doit donc être établie. Elle peut reposer simplement sur l'utilisation d'un compteur qui limite le nombre de déplacements d'un processus.

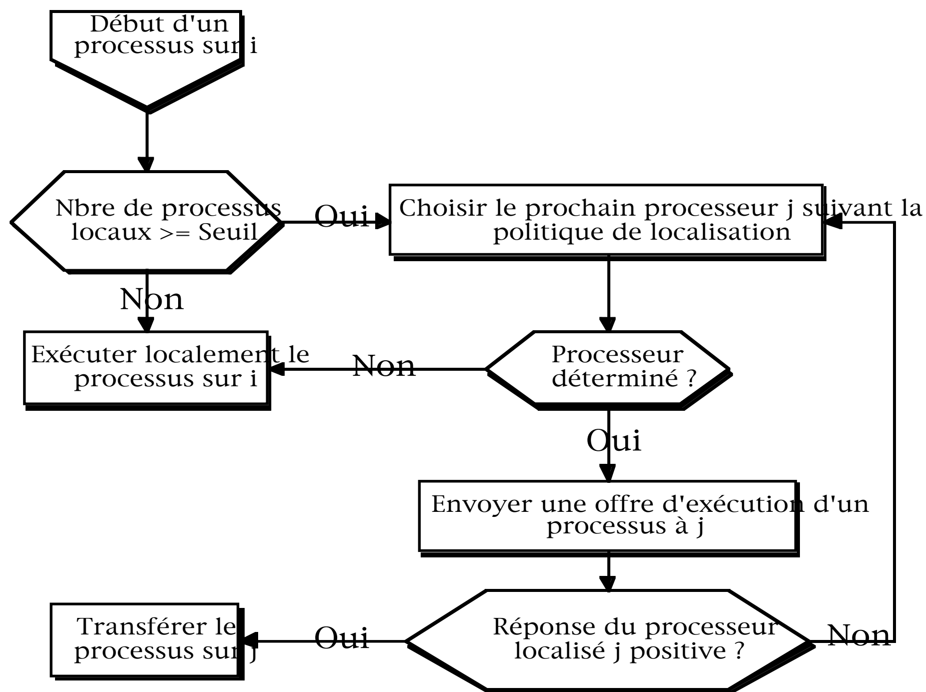


Diagramme 3.1. Stratégie “sender initiated”

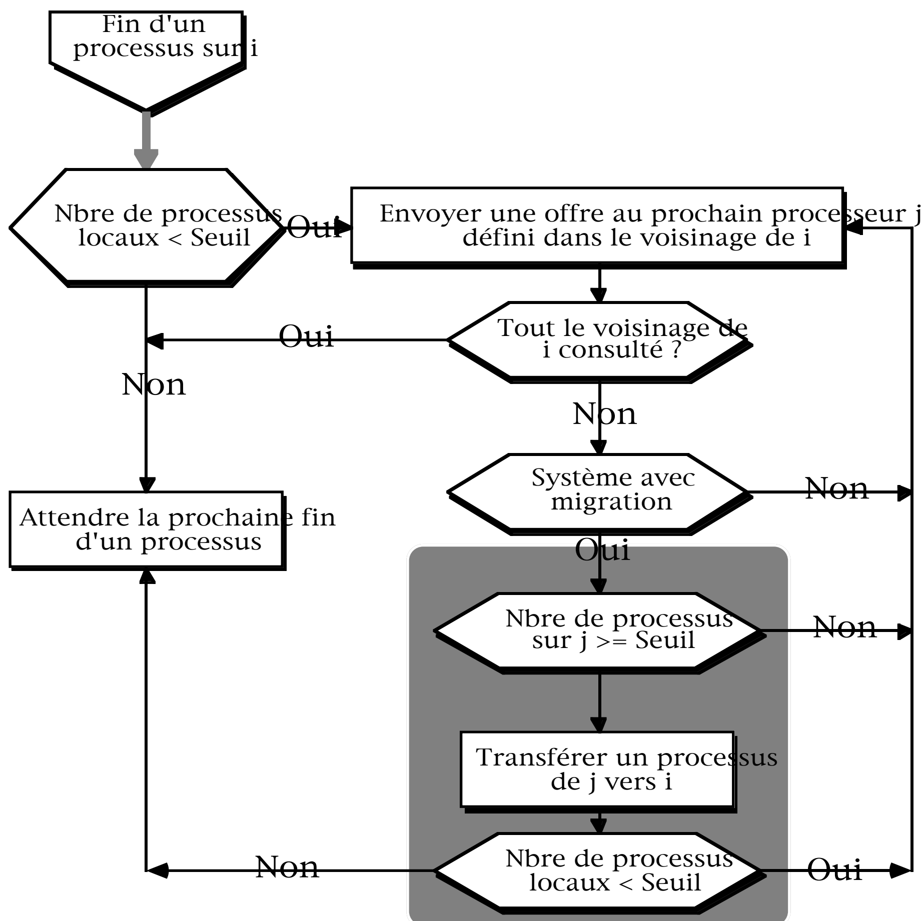


Diagramme 3.2. Stratégie “receiver initiated”

3.1.5. Critères pour l'évaluation de l'indice de charge

L'évaluation de l'indice de charge doit choisir entre, calculer une valeur exacte en se basant sur un maximum de critères, ou réaliser une approximation non optimale, mais satisfaisante pour une prise de décision rapide en se basant cette fois-ci uniquement sur quelques critères jugés significatifs.

Parmi les critères d'évaluation de l'indice de charge, nous pouvons citer entre autres:

- la taille de la file d'attente pour l'unité de calcul (CPU),
- la taille de la file d'attente des messages à traiter (MESS),
- la taille de la file d'attente des entrées/sorties (E/S),
- la taille de la mémoire dynamique utilisée (même si cet indicateur n'est pas exhaustif en ce qui concerne le temps d'exécution d'un processus)...

Une corrélation doit être trouvée entre certains de ces critères pour constituer une approximation significative de l'indice de charge d'un processeur.

Dans la pratique, tout dépend en fait du type d'applications visées: si celles-ci demandent beaucoup de temps de calcul, la taille de la file d'attente CPU peut constituer un bon indicateur pour l'évaluation de l'indice de charge; par contre, lorsqu'il s'agit d'applications qui effectuent énormément d'entrées/sorties, tout porte à privilégier la taille de la file d'attente E/S pour déterminer l'indice de charge du processeur.

De même, le taux de communication généré par l'application peut apporter un élément supplémentaire pour l'évaluation de l'indice de charge. En effet, suivant que la communication est continue ou sporadique, la prise en compte de la taille de la file des messages en attente peut s'avérer judicieuse ou désuète.

3.1.6. Distance et protocole de transfert de processus

Le grain de parallélisme a un impact direct sur ce point. Plus le grain est gros, plus le volume d'information à transférer est important. D'un autre côté, plus le grain est fin, plus le réseau de communication est sollicité par tous les messages de synchronisation et d'échange d'information. De ce fait, cette dernière approche contribue à un accroissement de la charge du réseau de communication.

Intuitivement, nous pouvons penser que dans le cas d'un grain fin, nous avons tout intérêt à donner une importance particulière au critère de la localité (distance).

Deux idées concourent à ce niveau:

- les réseaux dits à topologie fixe (CM5, Parsytec, R.C.A.I...),
- les réseaux à reconfiguration dynamique (tel le Transputer).

Dans le premier cas, c'est à la charge de l'algorithme d'allocation de déterminer le processeur le moins chargé, et le plus proche possible (nous reviendrons sur ce point par la suite).

En revanche, dans le second cas, la reconfiguration du réseau de communication va permettre de minimiser au plus la distance entre les différents processeurs qui souhaitent opérer un transfert de processus des uns vers les autres. Pour cela, il faut disposer d'un processeur maître (contrôleur) qui, au vu des différentes commandes de placement, restructure instantanément le réseau de manière à mettre en liaison directe un maximum de couples de processeurs esclaves (workers) impliqués dans cette opération.

Néanmoins les réseaux à reconfiguration dynamique (Transputers) n'autorisent que des structures de faible dimension [WAILLE91] et ne sont donc pas adaptés aux architectures massivement parallèles. Le passage à des architectures de plusieurs milliers de processeurs rend impossible un réarrangement fiable (non bloquant) du réseau de commutation. De même, le fait d'avoir à réarranger ce réseau à chaque communication entraîne inévitablement une sérialisation des communications, d'où perte de parallélisme effectif!

3.1.7. Stratégies de régulation de charge

3.1.7.1. Allocation statique

La manière la plus triviale pour réaliser une régulation de charge est d'allouer statiquement les différentes entités aux processeurs. Cette opération peut être réalisée soit de manière automatique, transparente à l'utilisateur, soit au contraire en l'assistant dans cette démarche, ou pire en la laissant à la charge de ce dernier. L'allocation statique suppose tout de même une certaine adéquation entre le problème à traiter et l'architecture cible. En exemple, nous pouvons considérer le traitement d'une image par un réseau de processeurs organisés en une grille où chacun d'eux serait responsable d'un pixel de l'image.

Toutefois, ce type de projection ne peut pas être trouvé systématiquement dans les programmes généraux. C'est le cas des programmes logiques. Nous ne pouvons faire d'a priori sur les buts à évaluer. Dans ce cas, une régulation de charge dynamique est nécessaire.

3.1.7.2. Méthodes aveugles

Ces méthodes ne reposent généralement sur aucune connaissance de la charge des autres processeurs. Pour cette raison, elles n'augmentent pas la charge du réseau de communication.

Une approche serait de sélectionner aléatoirement une distribution de processeurs destination à partir d'un ensemble de candidats [EAGER86b] [WANG85] [YUM81]...

Dans la régulation aléatoire **globale**, tous les nœuds du systèmes sont éligibles.

Dans la régulation aléatoire **locale**, seuls les nœuds topo-logiquement adjacents peuvent être candidats.

Cependant, ces méthodes ne semblent pas avoir un large consensus bien qu'elles soient très simples à implémenter [SARGEANT86]: en effet, des simulations ont montré leur inefficacité, tant au niveau des machines à gros grain, qu'au niveau des machines à grain fin. Ceci est du principalement au fait qu'elles ne prennent pas en compte l'état de charge du système lors de l'établissement de la liste des candidats.

3.1.7.3. Méthodes à jetons

Dans cette méthode, un "**contrôleur du réseau**" récupère les annonces d'aptitude à recevoir du travail, émises par l'ensemble des processeurs.

Lorsqu'un processeur est en mesure de fournir du travail en parallèle, il adresse une demande d'allocation au contrôleur. En retour, le contrôleur lui attribue un jeton référençant le processeur prêt à recevoir ce processus. Le processus y est alors transféré avec tout son contexte. Cependant, en cas d'indisponibilité de jetons, le processeur demandeur réalise le traitement en séquentiel, assurant par la même occasion un contrôle automatique du parallélisme. Néanmoins, cette méthode n'a été appliquée que dans le cas d'une machine à gros grain KABU-WAKE [SOHMA85], où chaque processeur ne peut traiter qu'un processus à la fois.

Une variante de cette méthode a été appliquée à la machine PIM-D [ITO86]. Chaque processeur possède une file de jetons des autres processeurs. Lorsqu'il dispose d'un processus à allouer, il puise dans la file le premier jeton. Le processus est exécuté sur le processeur correspondant à ce jeton. A la terminaison du processus, le jeton est restitué à la fin de la queue. La régulation de charge y est effectuée implicitement: un processeur occupé rendra tardivement le jeton à sa source. De plus, la notion de localité (distance) peut y être introduite en favorisant les processeurs voisins: initialement, on fournit dans la file locale davantage de jetons pour les processeurs voisins que pour ceux distants.

Néanmoins, cette technique, ainsi que sa variante, présentent le gros inconvénient de ne pas désigner explicitement les processeurs inactifs.

3.1.7.4. Propagation du gradient

Les méthodes précédentes ont soulevé la nécessité d'une **connaissance** de l'état de charge des autres processeurs pour être en mesure de réaliser une "bonne" régulation de la charge du système. Cette connaissance suppose donc un échange d'information entre les différents processeurs.

Une des méthodes pour réaliser cette fonction d'échange d'information est la propagation du gradient de pression à ses voisins. Cette méthode a d'abord été appliquée à une grille [LIN87]. Chaque processeur est au courant de l'état de charge de ses quatre voisins immédiats. Ainsi, à chaque étape, un processus sera orienté vers le voisin le moins chargé, et graduellement diffusera au travers du réseau (par analogie avec les pressions de gaz). Pour éviter de se bloquer dans des minima locaux, [LIN87] propose de propager son gradient affecté par les pressions des voisins immédiats. La pression p_i propagée d'un processeur i est définie comme suit:

$$p_i = \min\{g_i, 1 + \min\{p_j | d_{i,j} = 1\}\}$$

sachant que $d_{i,j}$ est la distance entre les processeurs i et j. Le gradient local g_i du processeur i est évalué de la manière suivante:

$$g_i = t_i + memPrs / (1 - memInUse)$$

avec t_i le nombre de tâches, $memPrs$ une constante (paramètre du simulateur positionnée à la valeur 0,01 dans [LIN87]), et $memInUse$ est la portion de mémoire utilisée (comprise dans l'intervalle]0,1[).

Trois états sont définis pour caractériser localement une pression:

- ***light*** : peut accepter toute nouvelle création de processus; la pression locale est maintenue à 0,
- ***moderate*** : dans ce cas le processeur évalue sa propre pression en tenant compte de celles des voisins; la création de la nouvelle entité est tout de même acceptée localement,
- ***heavy*** : saturation locale, le processeur tente de transférer la tâche courante vers un de ses voisins les moins saturés; la pression locale est évaluée comme dans l'état précèdent.

Dans tous les cas, la nouvelle pression locale n'est communiquée aux voisins que si celle-ci a changée depuis sa dernière mise à jour.

Le seul inconvénient de cette méthode est que la propagation du gradient à l'ensemble du système risque d'être relativement longue...

3.1.7.5. Méthode des enchères

Les algorithmes à **enchères** sont une autre forme de la régulation de charge [STANKOVIC84b][SMITH80].

Dans ces méthodes, les tâches sont affectées aux processeurs suivant leur capacité à réaliser un tel travail.

A la naissance d'un processus, une requête d'enchères est diffusée vers les autres processeurs. Chacun de ceux-ci communique à son tour le coût d'exécution de la tâche à son niveau. Le processeur demandeur choisit donc la meilleure offre pour y placer la nouvelle tâche.

Le problème qui se pose dans cette approche est la multitude de réponses à une seule requête. La solution à ce problème est que, lorsque le processeur demandeur retient la meilleure offre à sa requête d'enchère, il réserve le processeur correspondant [ROSS90]. Ce dernier peut accepter ou rejeter cette réservation, suite aux changements intervenus durant cet échange de messages. En cas d'échec, l'opération d'enchères est reprise. L'inconvénient de cette solution est le risque de famine si une requête est toujours supplantée par une autre plus “proche” des sites qui répondent à ces requêtes.

Bien que ces méthodes fournissent une solution sous-optimale, elles sont cependant largement plus extensibles et adaptables que beaucoup d'autres méthodes. De plus, les facteurs impliqués dans l'évaluation des enchères n'ont pas été très bien étudiés. [STANKOVIC84b] utilise des connaissances a priori, comme par exemple, les fichiers utilisés par un processus, le coût pour l'accès à ces fichiers localement et à distance, l'effet de groupage ou de séparation de processus coopérants, etc.

3.1.7.6. Méthode micro-économique

Dans cette approche très originale [FERGUSON88], les processeurs ainsi que les tâches qui doivent s'exécuter dessus, sont modélisés par des **agents économiques**. Chaque agent tente d'auto-satisfaire ses propres besoins en ressources, du mieux qu'il peut. Il n'y a pas d'approche globale dans ce système. En outre, le but d'un tel système est de définir les besoins des agents et les règles régissant leur service de telle manière à atteindre une amélioration des performances globales grâce à une "*main invisible*".

Les différents agents agissent en compétition pour la satisfaction de leur besoins sans aucune coopération: ce système est régi par une politique de prix structurée.

Deux types d'agents coexistent dans ce système: les **agents clients**, associés aux tâches à exécuter qui *achètent* du service, et les **agents serveurs**, représentés par les processeurs qui *vendent* du service.

Chaque agent client qui entre dans le système est doté d'un portefeuille avec une somme d'argent initiale. L'agent peut migrer au travers du système à la recherche du processeur offrant le service au prix le plus bas. Toutefois, pour garantir une **stabilité** du système (c'est à dire que les agents ne passent pas leur temps à migrer), à chaque passage au travers d'un lien de communication, l'agent doit régler une certaine somme d'argent (fonction de la charge de celui-ci).

Les agents serveurs vendent entre autres du temps CPU, de la bande passante pour la communication et de la mémoire. Ces agents fixent leur prix indépendamment des autres: leur but étant " *mercantile*", ils essayent de maximiser leur gain. Un paradoxe est qu'ils ne tentent pas de réguler la charge ou de minimiser le temps de réponse.

La "*publicité*" est autorisée dans ce système. Chaque processeur peut publier ses prix dans un "*tableau d'affichage*", maintenu par les processeurs voisins.

Eu égard à ce tableau, et suivant le budget de l'agent client et sa demande actuelle en ressources, il lance une enchère vers le processeur de prix le plus attractif. L'enchère est calculée de la manière suivante:

$$e_i = r_i - o_j$$

r_i est le reste d'argent de l'agent client sur le processeur P_i, et o_j est la meilleure offre de prix reçu du processeur P_j.

De cette façon, lorsque le processeur P_j reçoit plusieurs enchères, il choisira celle qui lui "*rapporte*" le plus.

Lorsqu'une enchère remporte la sélection, l'offre de l'agent client doit retourner jusqu'à la source, tout en réglant ses droits de passage au travers des liens de communication. La charge des liens de communication est une préoccupation persistante dans cet algorithme, puisque c'est un coût prépondérant dans les systèmes parallèles.

Les prix sont maintenus selon la loi de l'offre et de la demande. Les plus bas prix correspondent aux ressources les moins sollicitées, et vice versa.

La question que l'on peut se poser à ce niveau, est de savoir si une méthode qui ne repose que sur une structure de compétition peut fournir des performances satisfaisantes dans un système distribué?

Les auteurs montrent au travers de leur article, que la réponse est positive en ce qui concerne le problème de la régulation de charge.

Toutefois, cette méthode suppose une connaissance des coûts d'exécution et de communication des processus, ainsi que les temps de service CPU, que le système n'est pas toujours en mesure de connaître dans un environnement dynamique.

3.1.7.7. Méthode stochastique

[HAILPERIN88] propose une nouvelle méthode de régulation de charge dynamique basée sur une analyse statistique de séries temporelles. Chaque nœud estime une moyenne majorée de l'indice de charge du système en exploitant des connaissances passées des indices de charge de chacun des autres processeurs. Le but de chaque nœud est d'approcher cette valeur.

Cette méthode a été appliquée dans un système qui exhibe des indices de charge périodiques, sur une architecture qui assure un routage virtuel (du type "*wormehole*") et un mécanisme de communication "*one-to-many*" performant.

Périodiquement, chaque processeur propage son information vers un échantillon aléatoire des autres processeurs. [HAILPERIN88] montre que la distribution ainsi générée par cette communication suit une loi binomiale, pour rapidement converger vers une distribution de Poisson. Ainsi, il assure

par cette même occasion un volume d'information suffisant pour établir une bonne approximation de la charge moyenne du système, même lorsque la taille de l'échantillon est petite.

A la réception d'un message d'information, le processeur intègre celle-ci dans sa propre connaissance, puis utilise le modèle des séries temporelles (fourni à priori suivant une expérimentation portée sur ce système) pour estimer la charge moyenne courante du réseau. Il compare ensuite la charge prédite avec sa propre charge et celle reçue de l'émetteur. Si le processeur récepteur semble relativement sous chargé par rapport à l'émetteur, une demande de tâche sera émise en retour.

Comme toutes les méthodes basées sur les “enchères”, quelques précautions doivent être prises pour éviter une congestion du réseau de communication. Le processeur sous chargé ne propose plus de transfert de tâches jusqu'à ce qu'il reçoive du processeur surchargé soit une tâche soit une excuse (si une meilleure enchère a été obtenue d'un autre processeur).

Une autre possibilité serait que le processeur émetteur réserve le processeur sous chargé; mais ceci suppose qu'il y ait échange de messages supplémentaires.

Le seul inconvénient de cette méthode est qu'il n'y a pas de désignation de l'objet à transférer. Ce mécanisme a été surtout développé pour un système temps réel où il est primordial d'établir une distinction nette entre les tâches prioritaires et les autres.

3.1.8. De la prédiction de quelques paramètres...

Les données peuvent-elles fournir un élément significatif pour pouvoir appréhender le problème de la régulation dynamique de la charge d'un réseau de processeurs? L'étude statique d'un programme peut-elle dégager des caractéristiques qui peuvent être exploitées dans le but de faciliter par la suite la tâche de régulation de charge?

Autant de questions qui sont à l'ordre du jour, dans les préoccupations du domaine des systèmes parallèles et distribués [MEHRA93] [ROCH92] [KUNZ91] [DEVARAKONDA89] [ZHOU88] ...

Une première ébauche de réponse est de dire que ce problème est intimement lié à son expression. A savoir, les modèles de programmation et d'exécution utilisés, ainsi que l'architecture cible dans une moindre mesure, puisque elle intervient en fin de cycle, au moment de l'exécution. De même, les applications visées peuvent apporter un élément supplémentaire dans l'élaboration d'un cadre d'extraction de paramètres susceptibles d'être opportuns lors du placement dynamique.

S'agissant de la programmation logique par exemple, un graphe de dépendance des littéraux d'une clause, ainsi que les relations entre les différents arguments d'une même clause peuvent fournir une estimation de la mémoire requise pour la liaison des variables, ainsi que le nombre maximal de clauses qui peuvent s'exécuter simultanément (le cas des boucles n'est pas considéré à ce niveau).

Dans le cas de la programmation par objet, il est toujours possible de dégager le graphe d'exécution du réseau d'objets, en se référant à leur interdépendance (désignations entre objets). L'analyse de ce graphe permet d'établir pour chaque objet le degré de parallélisme qu'il pourra générer à chaque moment (degré du sommet correspondant). Ainsi, au moment de l'exécution, le système réserve (de préférence et dans la mesure du possible) autant d'espace que le degré du nœud correspondant. Ce procédé entraîne fatalement un gaspillage inutile de ressources critiques.

Néanmoins, la complexité du problème relègue pour le moment celui-ci à l'arrière plan en attendant des jours meilleurs!

3.1.9. Régulation de la charge dans la machine R.C.A.I

Après l'exposé de quelques techniques de régulation de la charge, une question se pose quant à l'intégration d'un tel mécanisme au niveau de l'architecture R.C.A.I: peut-on, ou à la limite, doit-on intégrer la procédure de régulation/partage de la charge?

Posée de cette façon, la question n'est pas d'un grand intérêt si nous ne situons pas le contexte d'une telle fonction. Rappelons que l'architecture R.C.A.I relève du grain fin et qui, pour l'instant, ne peut prendre en charge qu'un seul processus par processeur. Par conséquent, les unités de calcul appelées à s'exécuter sur cette machine sont de très petite taille. Leur durée de vie est relativement très courte (de l'ordre de quelques centaines de cycles en moyenne). La question qui se pose alors c'est l'opportunité d'une fonction qui, dans certains cas, peut être complexe. Le temps nécessaire pour calculer un emplacement "optimal" au processus est dans beaucoup de cas égal au temps requis pour mener à terme l'exécution du processus proprement dit. Ceci nous amène à priori à ignorer la régulation de charge dans de telles conditions.

Cependant, il ne faut pas perdre de vue que pour s'exécuter, un processus doit disposer auparavant d'un emplacement, un processeur libre, prêt à le prendre en charge. Néanmoins, le fait que les processeurs ne peuvent, pour le moment, exécuter qu'un seul processus à la fois, rend inutile la réalisation de la régulation de charge. En effet, le partage de charge suffit dans ce cas à équilibrer la charge globale du système [KRUEGER87a] [KRUEGER87b]. Ce point sera discuté dans la prochaine section.

La condition sine qua non pour pouvoir réaliser de la régulation de charge est la disponibilité de la notion de multiprogrammation au niveau du processeur élémentaire. Le souci omniprésent à chaque niveau de la conception de la machine R.C.A.I est d'arriver à n'intégrer que des fonctions simples. Vu la finesse de grain de notre machine, il est essentiel d'éviter l'utilisation de techniques qui nécessitent, pour leur réalisation, l'interruption des processus. Ce qui conduit à éliminer d'ores et déjà toutes les méthodes de multiprogrammation qui feraient appel à des moyens sophistiqués pour la gestion de la mémoire. C'est ainsi que nous sommes arrivés à décider, qu'au

lieu d'introduire un noyau pour l'allocation, la récupération et le tassement de la mémoire lorsque cela s'impose, il est plus judicieux d'éclater la mémoire en plusieurs blocs. Cette procédure revient à paginer, en quelque sorte, la mémoire et de n'allouer que des pages mémoire et non des fragments de celle-ci. Dans ce cas, chaque bloc est alloué à un seul processus. Cette structuration rend inutile le tassement de la mémoire (l'unité d'allocation étant la page mémoire).

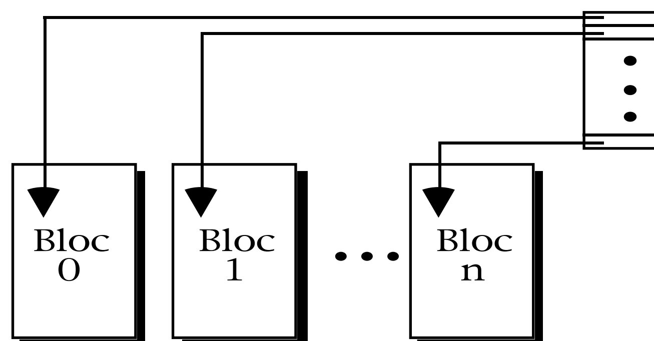


Fig.3.1. Partition de la mémoire.

Le problème qui se pose alors est de déterminer la taille du bloc de base (ainsi que le nombre de ces blocs par cellule). D'après les études menées par [ATHAS 86], travaux dont nous nous sommes inspirés pour élaborer notre modèle d'exécution d'acteurs, il semble que la taille du bloc soit relativement petite (la taille d'un acteur est en moyenne de l'ordre d'une quinzaine d'instructions - de haut niveau).

La partition de la mémoire n'est que virtuelle. En effet, il suffit d'associer à une mémoire normale, contiguë, une structure de table pour l'allocation des différentes pages de la mémoire (figure3.1). Les requêtes de cellules libres seront traduites par un couple

$$(p_i \ , \ b_j)$$

qui signifie que pour cette demande, nous allouons sur le processeur p_i, le bloc mémoire b_j. La charge d'un processeur sera alors le nombre de blocs déjà alloués.

Basé sur cette structure, nous proposons un nouvel algorithme dans les sections qui suivent, que nous avons baptisé **CLIMB**, par association aux sauts qu'il réalise.

Nous allons dans un premier temps exposer la solution qui a été retenue dans la version actuelle. Par la suite, nous détaillerons notre proposition pour la régulation de la charge comme perspective d'amélioration des performances de la machine R.C.A.I.

3.2. Mise en œuvre de l'allocation dynamique

Durant son exécution, un processus peut réclamer une ou plusieurs ressources pour mener à bien la tâche qui lui aura été assignée. Une ressource dans notre cas est un noyau exécutif (composé d'une mémoire, d'une unité centrale et de mécanismes de communication et de synchronisation).

La demande de ressources se décompose en plusieurs phases [LATROUS94a]. La première étape consiste à émettre une requête de localisation d'une cellule libre. Une fois cette cellule trouvée, il faut y charger la partie de code (processus ou encore acteur) à exécuter. Les variables locales de ce processus (contexte) doivent être initialisées à des valeurs cohérentes avant d'entamer son exécution.

Nous détaillons dans la suite de cette section chacun des points qui viennent d'être évoqués.

3.2.1. Allocation de ressources dans la machine R.C.A.I

Toute demande de ressource est transmise au réseau de communication qui se charge de la propager à l'ensemble des cellules, puis de rendre la réponse à la cellule émettrice. De cette manière, le processeur de calcul est déchargé de cette fonction qui trouve naturellement sa place au niveau de l'unité de routage.

Nous pouvons imaginer plusieurs algorithmes pour la diffusion dans une grille [TSAI94] [TAKKELLA94] [MICHALLON93] [BERMOND92] [TOUZENE92]... Cependant, il ne faut pas perdre de vue que notre but est d'arriver à intégrer tous ces mécanismes avec une consommation minimale de silicium. Par conséquent, nous devons renoncer à tous les algorithmes basés sur des tables de routages volumineuses. De même, les algorithmes qui nécessitent des messages intermédiaires de grande taille sont à écarter.

Le nombre de messages nécessaires dans le réseau pour réaliser la recherche de cellules libres est au minimum d'un seul message. Ce message doit couvrir l'ensemble du réseau. A l'opposé, dans le cas de l'utilisation de plusieurs messages, l'exploration du réseau est alors répartie entre ceux-ci.

Un exemple de la recherche au travers d'un message unique est l'algorithme en colimaçon.

L'utilisation de plusieurs messages requiert, au niveau du réseau de communication, un mécanisme de diffusion pour être efficace. L'algorithme dit de diffusion par ondes constitue un exemple de cette stratégie que nous développons ici.

3.2.1.1. Algorithme en colimaçon

Dans cette méthode, il n'y a qu'un seul message par demande dans le réseau. Le message décrit une spirale autour de la cellule émettrice (figure3.2).

Fig.3.2. Principe de l'algorithme en colimaçon.

L'adresse relative par rapport à la cellule émettrice est entretenue (par des incrémentations et des décrémentations) de sorte à pouvoir rendre la réponse à celle-ci. De même, un contrôle est effectué sur le calcul de l'adresse de retour, pour éviter que cette demande ne dépasse l'espace d'adressage de la cellule source.

Cet algorithme n'est pas aussi trivial qu'il le paraît. Lors de sa mise en œuvre, nous avons relevé deux cas qui ont nécessité un traitement particulier. Les cellules de bord requièrent un traitement spécifique. En effet, dans une grille ouverte[16], les cellules de bord doivent simuler la poursuite de la

[16]Dans un tore, ce type de problèmes ne se présente pas, puisque chaque cellule peut être considérée comme le centre du réseau par un simple changement de repère.

recherche afin de ne pas perturber le déroulement normal de l'algorithme (figure3.3). Nous désignons cette opération par une *sortie virtuelle du réseau*.

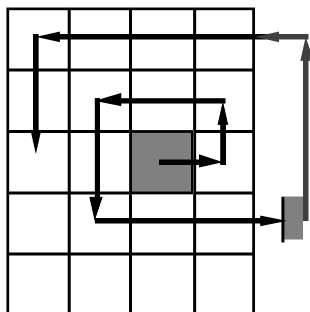


Fig.3.3. Cas délicat pour l'algorithme en colimaçon.

L'autre cas concerne les réseaux de petite taille. Un message de requête dans ce genre de réseau risque de tourner indéfiniment lorsque toutes les cellules de bord sont momentanément occupées.

En effet, si un réseau est de taille réduite, il faut assurer que si un message de recherche réalise quatre fois de suite l'opération de sortie virtuelle du réseau, il doit être supprimé. Autrement, ce message circulera sur la couronne externe des cellules sans dépasser l'espace d'adressage de la cellule source, donc sans risquer d'être extrait du réseau lorsque cela se produit.

Nous développons dans ce qui suit, une version complète de cet algorithme.

```
/*   INITIALISATIONS       */
   Length          =  1    ;  NextLength      =  1    ;
   OutputDirection  =  WEST  ;  Remain_Dx       =  0    ;  Remain_Dy  =  0  ;
void   SnailSearch   ()
{    /*   ALGORITHME                    */
   Length    =   Length   -  1  ;
   if    (  Length  ==    0  )
      then  {  switch  (  InputDirection    )
               {case    NORTH    :  OutputDirection  =  WEST     ;  break    ;
                case    EAST     :  OutputDirection  =  NORTH    ;  break    ;
                case    SOUTH    :  OutputDirection  =  EAST     ;  break    ;
                case    WEST     :  OutputDirection  =  SOUTH    ;  break    ;  }
               switch  (  InputDirection  )
               {case    NORTH    :  case  SOUTH    :
                  NextLength =  NextLength      +  1       ;  break    ;
                default          :  break          ;  }
               switch  (  OutputDirection   )
               {case    EAST     :  case  WEST     :
                  Length         =  NextLength    -  Remain_Dx           ;
                  break          ;
                case    NORTH    :  case  SOUTH    :
                  Length         =  NextLength    -  Remain_Dy           ;
                  break          ;                                          }
            }
      else  {  switch  (  InputDirection    )
               {case    NORTH    :  OutputDirection  =  SOUTH    ;  break    ;
                case    EAST     :  OutputDirection  =  WEST     ;  break    ;
                case    SOUTH    :  OutputDirection  =  NORTH    ;  break    ;
                case    WEST     :  OutputDirection  =  EAST     ;  break    ;  }
   /*   Border  cells  handling.  */
   Loop     =  TRUE     ;
   while    (  Loop  )
   {  Loop  =  FALSE       ;
      switch  (  OutputDirection  )
      {case    EAST     :  case  WEST     :
         if    (  IsBorderCell  (OutputDirection,   Dx,   Dy)   )
            then  /*   Detect the ring is already checked ?    */
            {  Remain_Dx        =  Length                              ;
               Length           =  NextLength -  Remain_Dy             ;
               OutputDirection  =  (  OutputDirection  ==    EAST)      ?
                                             SOUTH    :  NORTH         ;
               Loop             =  TRUE                                ;
               switch  (  InputDirection    )
               {case    NORTH    :
                  if    (  NextLength >=    ToBorder_Dy    )
                     then  StopSnail   |=      0x01        ;  /*    0001  */
                  break          ;
                case    EAST     :
                  if    (  NextLength >=    ToBorder_Dx    )
                     then  StopSnail   |=      0x02        ;  /*    0010  */
                  break          ;
                case    SOUTH    :
                  if    (  NextLength >=    ToBorder_Dy    )
                     then  StopSnail   |=    0x4           ;  /*    0100  */
                  break          ;
                case    WEST     :
                  if    (  NextLength >=    ToBorder_Dx    )
                     then  StopSnail   |=    0x08          ;  /*    1000  */
                  break          ;                         }
            }
         break          ;
```

```
      case   NORTH   :  case  SOUTH   :
        if    (  IsBorderCell  (OutputDirection,   Dx,   Dy)   )
          then
          {  Remain_Dy     =  Length                             ;
             Length           =  NextLength -  Remain_Dx          ;
             OutputDirection  =  (  OutputDirection  ==    NORTH)  ?
                                         EAST     :     WEST     ;
             Loop             =  TRUE                            ;
             switch (  InputDirection   )
             {case   NORTH   :
               if    (  NextLength    >=    ToBorder_Dy   )
                  then  StopSnail     |=    0x01       ;  /*    0001  */
               break         ;
              case   EAST     :
               if    (  NextLength    >=    ToBorder_Dx   )
                  then  StopSnail     |=    0x02       ;  /*    0010  */
               break         ;
               case   SOUTH   :
               if    (  NextLength    >=    ToBorder_Dy   )
                  then  StopSnail     |=    0x04       ;  /*    0100  */
               break         ;
              case   WEST    :
               if    (  NextLength    >=    ToBorder_Dx   )
                  then  StopSnail     |=    0x08       ;  /*    1000  */
               break         ;                          }
          }
        break                 ;                          }
    }
}
```

Algorithme du colimaçon.

3.2.1.2. Algorithme de diffusion par ondes

L'algorithme de diffusion par ondes est parmi les plus simples à mettre en œuvre. Dans notre cas, cet algorithme ne nécessite aucun découpage du message avant son envoi. Les algorithmes qui recourent à des découpages de messages pour exploiter au maximum tous les liens de communication sont souvent confrontés à des problèmes de débit de ces liens [MICHALLON93]. A priori, dans notre cas, le réseau de communication ne présente pas de problème de débit [KARABERNOU93]. De plus, une telle stratégie requiert un mécanisme de reconstruction des messages.

Dans l'approche par ondes (figure3.4), la demande de cellule est diffusée dans le réseau de communication suivant une onde autour de la cellule émettrice. Une cellule libre qui reçoit un message de requête répond à la cellule émettrice et interrompt par la même occasion la diffusion à son niveau.

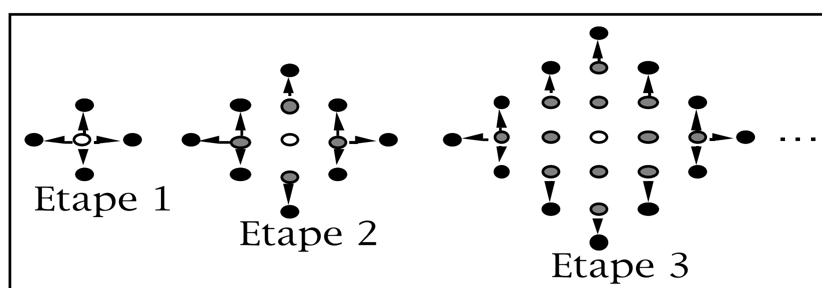


Fig.3.4. Principe de l'algorithme de diffusion par ondes.

L'inconvénient majeur de cette méthode est le traitement des réponses multiples. Pour y remédier, nous avons introduit une horloge logique pour dater chaque demande. A la réception d'une première réponse, l'horloge est avancée d'une unité. De cette façon, toutes les réponses antérieures à la date courante seront ignorées (plus exactement annulées comme nous le verrons par la suite).

L'algorithme qui en résulte est très simple, comme nous pouvons le constater dans la fonction suivante:

```
void  WaveSearch  (  void   )
{      /*    ALGORITHME                */
 Switch (  InputDirection   )
 {case    NORTH    :  BroadcastTo   (SOUTH, NIL,     NIL)     ;  break    ;
  case    EAST     :  BroadcastTo   (NORTH, WEST,    SOUTH)   ;  break    ;
  case    SOUTH    :  BroadcastTo   (NORTH, NIL,     NIL)     ;  break    ;
  case    WEST     :  BroadcastTo   (NORTH, EAST,    SOUTH)   ;  break    ;  }
}
```

Algorithme de diffusion par ondes.

3.2.1.2.1. Calcul des bornes des messages de réponses

Dans le but de limiter la prolifération des messages dans le réseau, et eu égard aux spécifications matérielles de R.C.A.I, la recherche par onde est bornée. Elle est limitée par la taille physique de l'adresse. Dans R.C.A.I, cette taille est de 8 bits (cf.§1.2.3). A chaque progression dans une direction, nous incrémentons ou décrémentons le champ correspondant de l'adresse de retour. Lorsqu'une des deux directions provoque un débordement, le message correspondant est retiré du réseau et détruit[17]. De cette manière, nous réduisons considérablement la multiplication des messages de réponse en retour.

Nous allons aborder à présent l'évaluation de ces bornes. Plusieurs messages sont introduits dans le réseau pour une même recherche. Ce nombre ***n*** peut être borné par la relation:

$$1 \leq n \leq 2(2 * k + 1)$$

où k est la distance maximale pouvant être parcourue dans un sens ou dans l'autre de l'axe des abscisses (figure3.5).

[17]Le cas particulier des cellules aux extrémités est traité au paragraphe 4.3. Il s'agit de la notion de relais.

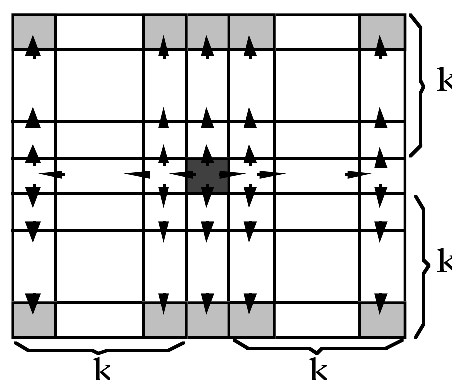


Fig.3.5. Limites de la diffusion.

Dans le cas qui nous concerne, k prend au maximum la valeur de 14, ce qui correspond au plus à 48 messages[18] (cf.§4.3). La valeur 14 correspond en fait à un niveau de relais que nous détaillerons au chapitre 4.

Un tel nombre de messages est négligeable dans R.C.A.I compte tenu des capacités du routeur [RUBINI92]. De plus, ce cas extrême signifie qu'autour de la cellule émettrice, toutes les cellules sont occupées, à l'exception des cellules de bord. Ce cas est assez rare dans la pratique...

3.2.1.2.2. Protocole de réservation des cellules

A la réception d'un message de recherche de cellule libre, la cellule interroge l'unité de gestion de la "régulation de charge" pour savoir si la cellule en question est libre, prête à recevoir un processus. A ce moment, deux réactions sont possibles:

- celle qui consiste à répondre par l'affirmative à une première demande et à refuser les suivantes; pour cela, la cellule se déclare comme étant *réservée* dès la réception de cette demande (ce schéma correspond dans une certaine mesure à celui adopté dans [HAILPERIN88] au §3.1.7.7),
- l'autre attitude consiste à répondre par l'affirmative tant que cette cellule est effectivement libre. A partir du moment où une autre cellule aura confirmé sa réservation, la cellule répond alors par la négative.

Dans le premier cas, une cellule qui reçoit une réponse invalide à sa demande (la date de cette confirmation est antérieure à l'horloge locale) annule sa requête par un message de type "*CANCEL*". Ce message, une fois

[18]En réalité, k a pour valeur 7 (taille physique de chacune des deux parties d'une adresse dans R.C.A.I). Ce qui induit 30 messages au maximum.

arrivé à destination, libère la cellule cible qui devient alors disponible pour les requêtes suivantes.

L'inconvénient de cette méthode est évidemment la réquisition, temporaire, de cellules sans être exploitées. Ceci pénalise donc les cellules qui ne trouvent pas de cellules libres à leur demande. De ce fait, il est nécessaire de pouvoir réitérer la recherche après plusieurs échecs successifs, laissant ainsi le temps aux cellules qui en ont réquisitionnées d'autres de les libérer.

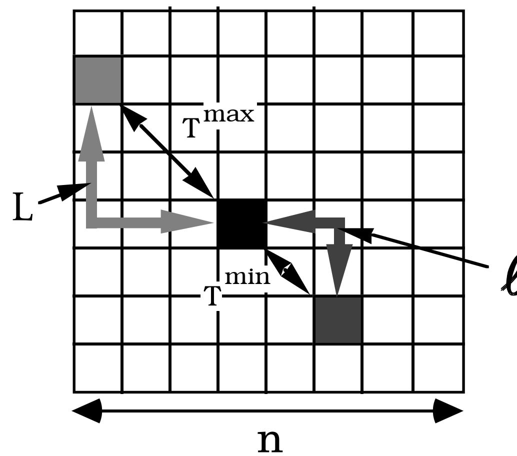
La seconde approche nécessite par contre pour chaque recherche un protocole plus contraignant. Après une diffusion de la requête, la cellule attend un message de type "*FREE*" (cellule libre, prête à recevoir un processus). Elle envoie ensuite un message de type "*CONFIRM*" (pour certifier sa demande) ou "*CANCEL*" (si elle a déjà reçu une réponse). Dans le cas d'une confirmation, la cellule attend un message de type "*LOAD*" qui confirme à son tour que la cellule cible est réellement prête à recevoir du code.

Ce protocole en quatre phases entraîne une multiplication du nombre de messages mis en œuvre pour chaque demande. C'est la raison pour laquelle nous avons opté pour la première solution.

3.2.1.3. Comparaison des deux méthodes de recherche de cellules libres

L'avantage de la recherche en colimaçon est qu'il n'existe qu'un seul message dans le réseau pour chaque appel. Cependant, cette technique procède d'une manière séquentielle. De plus, elle ne prend pas en compte les débits des liens de communication.

La seconde solution présente l'avantage de trouver non seulement la première cellule libre la plus proche, mais également celle où les débits des canaux de communication sont les moins chargés.



Les temps d'acheminement de messages peuvent être formulées comme suit:

$$t_{com}^{max} = \left[L + (\nu - 1)\right] \left[\beta + \frac{m}{\nu}\tau_c\right]$$

$$t_{com}^{min} = \left[l + (\nu - 1)\right] \left[\beta + \frac{m}{\nu}\tau_c\right]$$

avec m la longueur du message à transmettre, ν la taille d'un paquet, β le temps d'amorçage de la communication et τ_c le temps élémentaire de transfert.

En simplifiant les écritures précédentes, nous obtenons les équations suivantes:

$$t_{com}^{max} = s + t_{elem} * L$$

$$t_{com}^{min} = s + t_{elem} * l$$

avec:

$$s = (\nu - 1) \left(\beta + \frac{m}{\nu}\tau_c\right)$$

$$t_{elem} = \beta + \frac{m}{\nu}\tau_c$$

$$L = l + l^{\copyright} \quad (\text{car} \quad L \rangle \; l)$$

L'avantage de la solution par diffusion d'une requête peut être expliqué par le fait qu'une cellule éloignée répond dans un laps de temps plus long qu'une cellule proche, pour des débits de communication similaires sur les liens respectifs reliant les deux cellules. De cette façon, nous pouvons écrire:

$$L \rangle l \Rightarrow t_{com}^{max} \rangle t_{com}^{min}$$

Par contre, une cellule qui se trouve sur des liens de communication saturés met plus de temps à faire parvenir sa réponse à la cellule émettrice. Dans le même temps, une cellule, bien qu'éloignée, mais qui se trouve sur des liens de communication peu saturés, répond plus rapidement. En considérant *tsup* le temps supplémentaire induit par la charge des liens de communication, les équations qui découlent de cette constatation s'écrivent comme suit:

$$t_{com}^{min} = s + t_{elem} * l + t_{sup},$$

$$t_{com}^{max} = s + t_{elem} * \left(l + l^{\copyright}\right),$$

$$t_{com}^{max} = s + t_{elem} * l + t_{elem} * l^{\copyright},$$

$$t_{com}^{max} = t_{com}^{min} - t_{sup} + t_{elem} * l^{\copyright}$$

$$t_{sup} \rangle t_{elem} * l^{\copyright} \Rightarrow t_{com}^{min} \rangle t_{com}^{max}$$

En résumé, le tableau ci-dessous reprend les différents éléments comparatifs des deux méthodes.

	Nombre de messages	Minimise la distance	Régulation de la charge des communications	Risque d'Interblocage	Complexité
Recherche en colimaçon	1	oui	non	oui	Beaucoup de contrôle
Recherche par ondes	$1 \leq n \leq 30$	oui	oui	non	Simple à intégrer

Les constatations relevées dans ce tableau nous ont amené à écarter la recherche en colimaçon, surtout pour les cas d'interblocage qu'elle peut occasionner. En effet, la stratégie mise en œuvre ne respecte plus les règles établies au paragraphe1.2.3.2, qui garantissaient l'absence d'interblocage. On se convaincra facilement au vu de l'exemple de la figure3.6 de la réalité de ce risque.

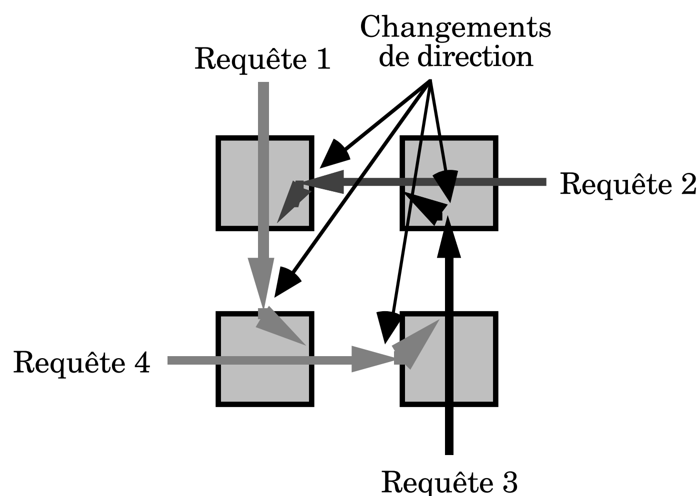


Fig.3.6. Cas simple d'interblocage lors de quatre recherches simultanées en colimaçon.

Du fait que l'algorithme du colimaçon ne respecte plus un routage incrémental, routage toujours dans le même sens, en abscisse puis en ordonné par exemple, des cas d'inteblocage peuvent facilement survenir, comme nous l'avons montré au §1.2.3.2.

3.2.2. Implémentation et intégration de la recherche par ondes

De par sa nature même, cet algorithme impose une structure différente pour les routeurs NORD/SUD et EST/OUEST. En effet, dans le cas des routeurs NORD et SUD, il suffit de retransmettre le message reçu sur un lien dans la direction opposée. Dans le cas des routeurs EST et OUEST, il faut diffuser ce message dans les trois directions restantes.

L'opération de diffusion peut être réalisée soit de manière séquentielle, soit en parallèle.

Dans la première situation, la solution est assez évidente. En suivant un ordre préétabli, la diffusion de la requête est acheminée vers la prochaine direction dès que le message sur la direction précédente a été complètement transféré. Avant d'être acheminé vers une quelconque direction, le message de recherche est stocké dans une structure de sauvegarde (réutilisée successivement pour les différents transferts, cf.figure3.7).

Par contre, s'agissant de l'approche parallèle, il faut prévoir trois structures indépendantes - une pour chaque direction. Cette approche présente l'avantage de ne pénaliser aucune des trois directions. Toutefois, un premier problème soulevé par cette méthode est celui du traitement de plusieurs requêtes successives. Le fait qu'un des tampons soit saturé, entraîne nécessairement la sauvegarde de toute la suite des requêtes pour pouvoir être réexpédiées plus tard. A moins de pouvoir poser une borne sur la longueur de cette liste, et qui soit tout de même modérée, il est impossible de pouvoir intégrer efficacement une telle solution. De plus, le recours à une structure triple entraîne un coût supplémentaire non négligeable au niveau silicium.

Par conséquent, pour des considérations d'optimisation et de simplicité en vue d'une intégration peu coûteuse sur silicium, nous avons retenu la solution séquentielle.

Nous allons donc discuter de cette solution dans ce qui suit.

3.2.2.1. Structure des tampons

La structure des tampons de réception des requêtes est représentée dans la figure3.7.

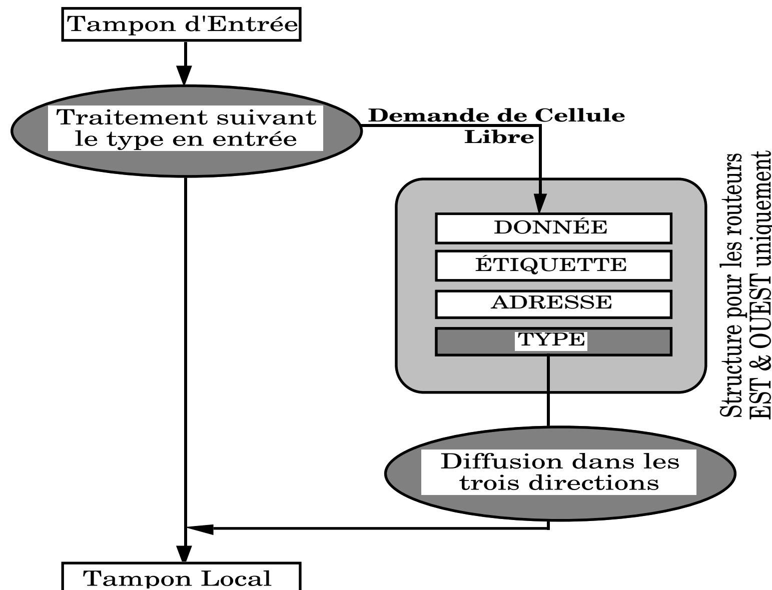


Fig.3.7. Structure des tampons.

La localisation de cellules libres est amorcée par l'instruction FIND. Le processus débute par la construction d'un message de type "*FIND*" de la manière suivante:

FIND *ADRESSE_EN_MÉMOIRE*

Cette instruction génère un message dont la structure est détaillée dans la figure3.8.

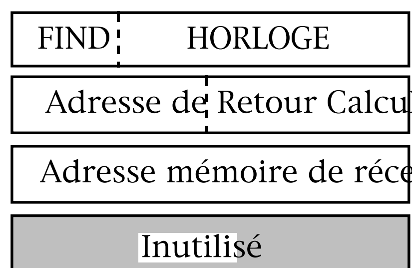


Fig.3.8. Structure d'un message de recherche.

L'adresse de retour est initialisée à la valeur (0,0). Le champ *ADRESSE_EN_MÉMOIRE* correspond à l'adresse mémoire de réception (où sera rangée la réponse à la requête courante). Le processeur se met ensuite en attente sur cette position mémoire (tant que le neuvième bit reste à zéro).

M[*ADRESSE_EN_MEMOIRE*].valid $\leftarrow$ 0

En forçant le contenu de cette case mémoire à la valeur $\perp$ (nil), nous imposons une attente sur ce canal de communication (synchronisation): la tentative de lecture de la valeur $\perp$ entraîne la mise en attente d'une donnée à cette position mémoire du processus courant (voir modèle d'exécution). A la réception d'une réponse, le processus qui a exécuté cette instruction poursuit normalement son traitement.

3.2.2.2. Structure du chemin de données

Le chemin de données proposé par [KARABERNOU93] a été modifié pour intégrer les unités spécialisées pour la gestion de l'allocation dynamique de ressources (figure3.9).

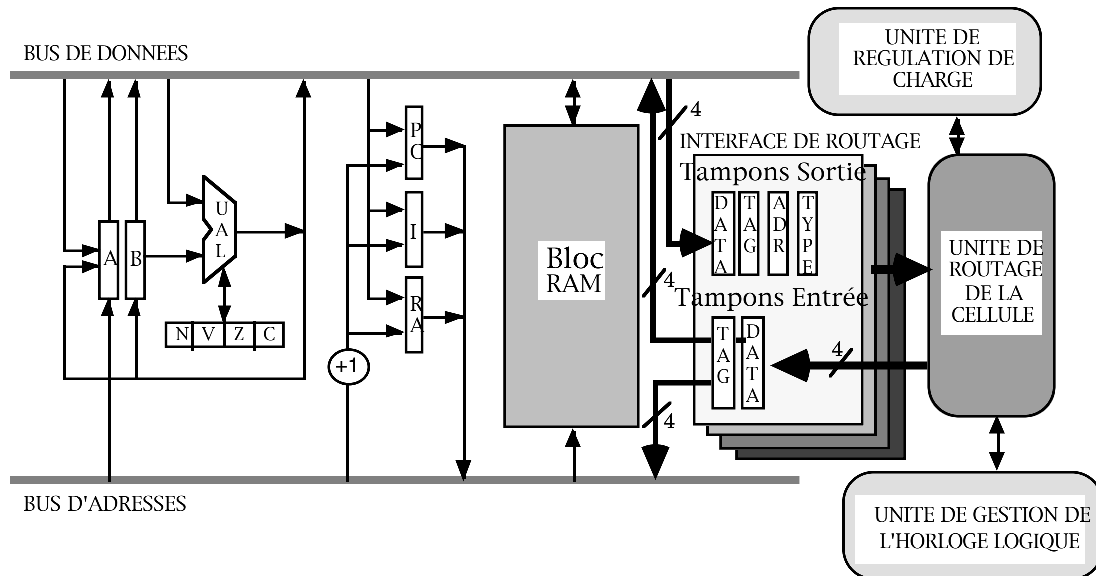


Fig.3.9. Diagramme du chemin de données.

De la même manière, nous avons également modifié les blocs d'entrée et de sortie pour les adapter aux nouvelles exigences de la gestion dynamique du réseau (figures3.10 & 3.11).

L'unité de routage de la cellule dispose de quatre tampons d'E/S indépendants et parallèles [LATROUS94b] pour augmenter le degré de parallélisme de la recherche de cellules libres (cf.§3.2.2.3).

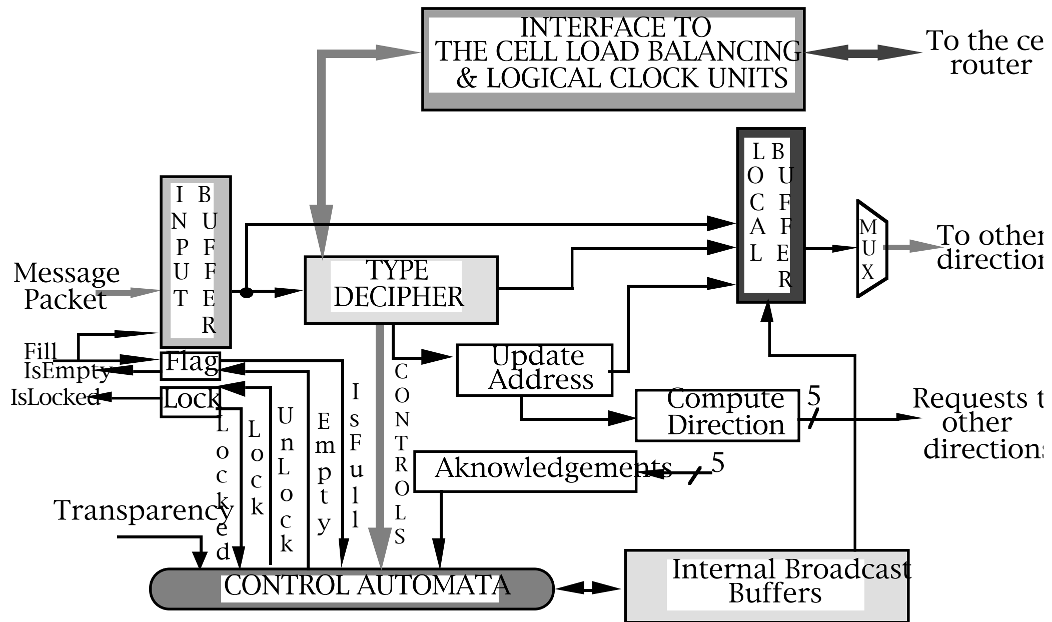


Fig.3.10. Synoptique du block d'entrée (le récepteur).

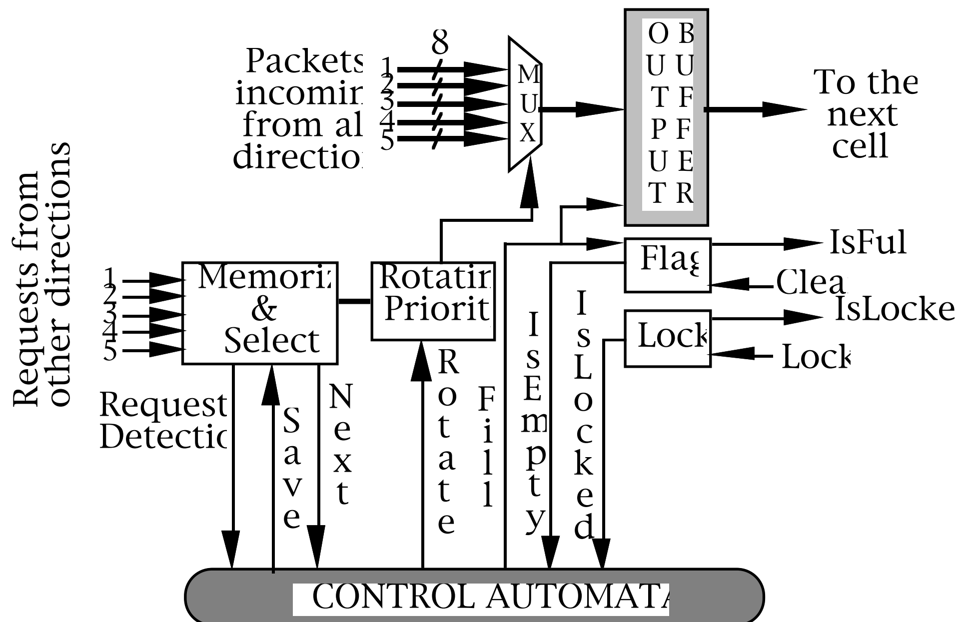


Fig.3.11. Synoptique du block de sortie (l'arbitre).

Il faut noter que les quatre routeurs d'entrée de la cellule sont indépendants. Par conséquent, pour assurer l'exclusion mutuelle lors de la réception simultanée de plusieurs demandes, le routeur se réfère à une seule unité de la régulation de charge. Toutes les interrogations sur l'état de la cellule sont adressées à cette unité. C'est la seule entité habilitée à répondre à ces questions (figure3.12).

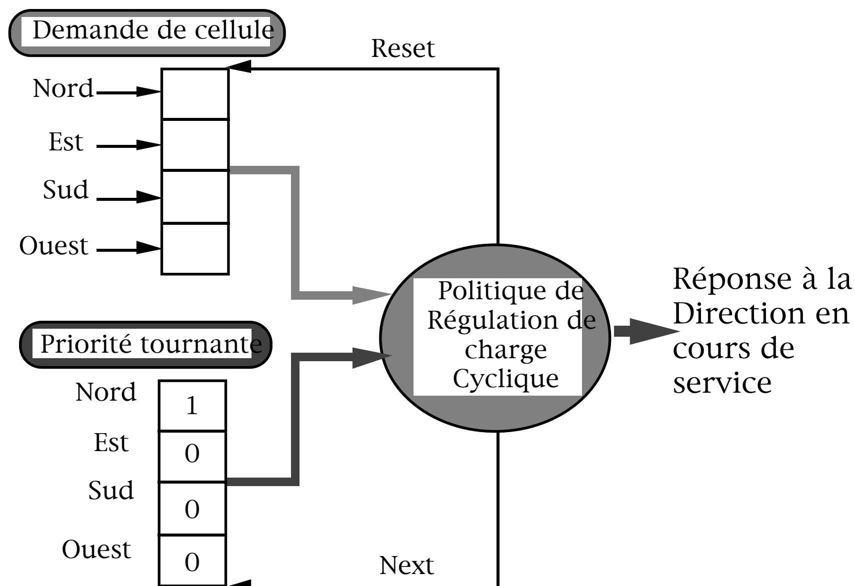


Fig.3.12. Vue fonctionnelle de l'unité de régulation de charge.

L'instruction de recherche de cellules libres émet simultanément dans les quatre directions un message de requête et se met en attente d'une réponse. Chacune des quatre directions est indépendante des autres (figure3.13).

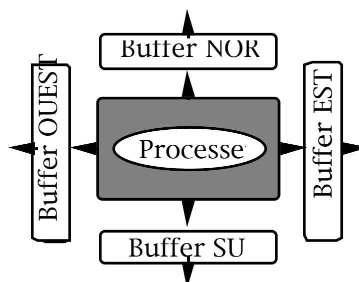


Fig.3.13. Tampons de sortie parallèles.

Ainsi, si l'une de ces directions est bloquée en attente de la libération du tampon de sortie correspondant, en raison d'une ancienne communication

non encore acheminée, les autres directions diffusent tout de même la demande de la cellule dans le réseau.

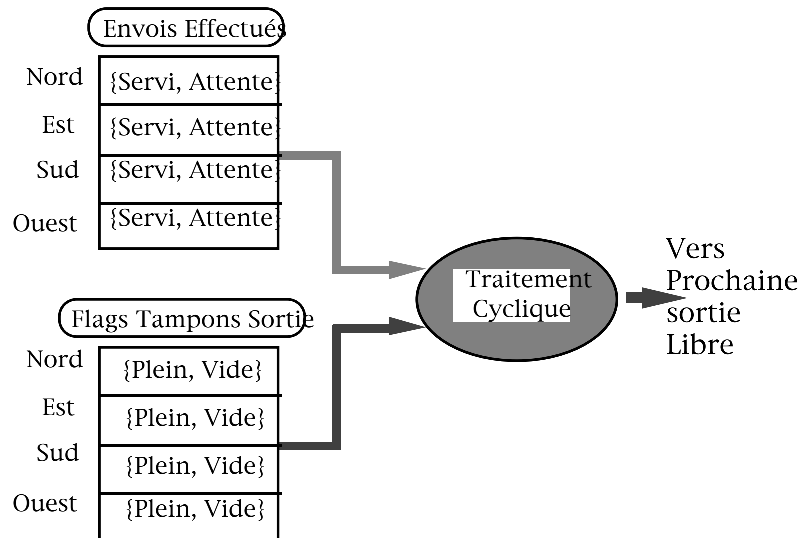


Fig.3.14. Envoi parallèle (cyclique) de FIND.

De plus, cette approche permet de maximiser le parallélisme de l'opération de recherche. Pour cela, l'instruction FIND procède d'une manière cyclique. Elle cherche perpétuellement la prochaine sortie libre, toujours dans le même sens (figure3.14).

Toutefois, avant et après chaque envoi (également une fois tous les envois effectués), le routeur de la cellule procède au test d'arrivée d'une réponse (positive) à l'un des messages déjà acheminés (figure3.15).

```
if  (M[ADRESSE_EN_MÉMOIRE].valid)
  then
     • Arrêter la recherche
     • Branchement à l'instruction suivant
fi
```

Fig.3.15. Attente d'arrivée d'une réponse pour l'instruction FIND.

Si une réponse a déjà été reçue, les envois restants sont abandonnés afin de ne pas saturer le réseau de communication. En effet, tout message "périmé" qui arrive à une cellule doit être annulé par un autre message de type "CANCEL", d'où une multiplication de messages superflus.

3.2.3. Une nouvelle proposition d'algorithme pour la régulation de charge dans la machine R.C.A.I: l'algorithme CLIMB

La régulation de charge par le biais de techniques qui nécessitent une connaissance globale s'avère incompatible avec les contraintes imposées par les systèmes massivement parallèles [HAILPERIN88]. En effet, le temps requis pour collecter une information globale est trop important pour être réaliste (vu le nombre élevé des processeurs). C'est le cas notamment de l'algorithme du gradient [LIN87] (cf.§3.1.7.4) qui doit diffuser une nouvelle pression à l'ensemble du réseau. Il faut donc privilégier les méthodes qui reposent sur une connaissance partielle (très restreinte) pour réaliser une régulation de charge dans les systèmes massivement parallèles. [HAILPERIN88] propose l'utilisation d'un modèle stochastique basé sur les séries temporelles. Bien que les résultats rapportés semblent satisfaisants, l'heuristique développée s'applique, comme le souligne l'auteur, à des systèmes qui présentent des charges périodiques (répétitives), tels que les systèmes temps réels.

Il nous a donc semblé possible d'améliorer l'algorithme du gradient. Nous proposons à notre tour un nouvel algorithme qui repose uniquement sur la connaissance de la charge des voisins immédiats et qui ne suppose à priori aucun type d'applications spécifiques.

Le point critique dans l'algorithme de Lin est la mise à jour de l'information dans le réseau. Cette mise à jour nécessite dans le pire cas une diffusion de cette donnée à l'ensemble des cellules. Dans le cas de la machine R.C.A.I, le nombre de messages nécessaires à une telle opération est de **2n** pour un réseau de $n \times n$ cellules. Nous avons alors imaginé un autre algorithme, **CLIMB**, qui nécessite au plus **4** messages pour cette même opération.

Un autre critère important, pour la comparaison de ces deux algorithmes, est la qualité de l'information disponible à un moment donné pour prendre une décision. Dans le cas de l'algorithme du gradient, une information disponible à un instant donné au niveau d'un processeur peut être erronée. En effet, entre le moment où cette information aura été communiquée au processeur en question et celui où ce dernier décide de lui

envoyer un processus, d'autres processeurs ont déjà pu faire appel à ce même processeur pour se décharger de leur surplus de travail. De cette manière, et dans le cas dynamique, il peut survenir le cas où un processeur ait à réitérer plusieurs fois sa requête avant de la voir satisfaite. Par contre, dans le cas de notre algorithme, les décisions sont prises successivement par les différents processeurs qui voient passer cette requête eu égard à la connaissance, toujours fiable, de leur propre charge et de celles de leurs voisins immédiats[19].

Pour terminer, signalons que l'algorithme du gradient ne prend pas en considération les communications. Or, dans les systèmes (massivement) parallèles, ce facteur tient une importance prépondérante dans les performances globales d'une machine de ce type. L'algorithme CLIMB est en mesure d'inclure ce facteur dans son principe (cf. paragraphe 3.2.3.2).

D'autre part, notre algorithme reprend également une philosophie inhérente au recuit simulé: la possibilité de s'échapper de minima locaux pour approcher un minimum global du système considéré. Dans cette optique, notre algorithme permet à un objet, suivant l'énergie cinétique acquise durant son mouvement, de remonter des côtes, pour s'échapper de minima locaux.

Brièvement, notre algorithme repose sur la notion de quantité de mouvement. En se déplaçant, un objet acquiert de l'énergie cinétique qui lui permet de se mouvoir entre les “sommets” du plan définis par les charges des processeurs (figure3.16).

Cette analogie: la minimisation de la fonction de coût avec la fonction d'énergie potentielle sur une surface dans l'espace soumis à un champ de gravitation, est parlante et fréquemment employée. Nous allons donc la reprendre.

[19]Au niveau silicium, on peut imaginer que l'échange d'information entre deux cellules voisines puisse s'effectuer directement par une ligne électrique qui indique un processus en plus ou en moins au niveau de la cellule en question.

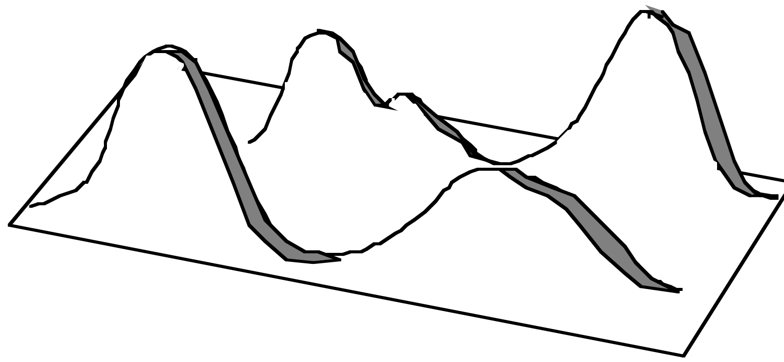


Fig.3.16. Plan défini par les charges des processeurs.

3.2.3.1. Principe de l'algorithme proposé

Le principe général de l'algorithme proposé repose sur deux mouvements essentiellement: un objet peut descendre ou remonter une côte suivant les inclinaisons du plan défini par les charges respectives des différents processeurs du réseau.

Il s'agit de calculer, dans tous les cas de figure, la valeur de l'énergie cinétique acquise durant ce mouvement sur une certaine distance[20].

Lors de l'application de cet algorithme aux systèmes parallèles, nous simplifions largement les formules, ce qui nous permettra d'appréhender plus facilement le cas de frottements proportionnels.

---

[20]On ne considérera, pour l'écriture des équations, que le cas de frottements constants. En effet, le cas de frottements proportionnels à la masse d'un objet en déplacement nécessite des écritures d'équations très difficiles à déchiffrer.

3.2.3.1.1. Un objet en descente

Dans ce premier cas, les frottements vont contribuer à ralentir l'allure de l'objet, avec une accélération dans le sens du mouvement (positive).

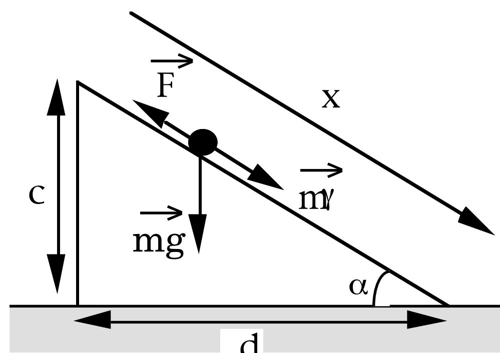


Fig.3.17. Un objet progressant le long d'une pente.

La valeur de l'énergie cinétique est calculée de la sorte:

$$E_c = m\gamma.x + E_{c_0}$$

avec

$$\gamma = g.\sin\alpha - \frac{F}{m}$$

sachant que **F** est la force de frottement supposée constante, γ l'accélération de l'objet de masse m, et g l'attraction dans le système considéré. E_{c_0} représente l'énergie cinétique initiale.

Il en découle l'expression suivante:

$$E_c = (m.g.\sin\alpha - F)x + E_{c_0}$$

3.2.3.1.2. Un objet en montée

Les frottements et l'accélération sont cette fois-ci dans le même sens, contraire à celui de la trajectoire de l'objet. L'objet se meut grâce à l'énergie (initiale) acquise au préalable. A cause de l'action conjuguée des frottements et de l'accélération, l'objet ralentit son allure jusqu'à une (éventuelle) immobilisation.

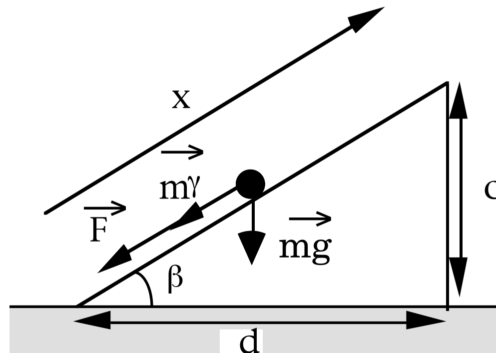


Fig.3.18. Un objet progressant le long d'une côte.

La valeur de l'énergie cinétique est calculée de la sorte:

$$E_c^{©} = -m\gamma^{©}.x + E_{c_0}^{©}$$

avec

$$\gamma^{©} = g.\sin\beta + \frac{F}{m}$$

Ce qui donne l'équation ci-après:

$$E_c^{©} = -\left(m g.\sin\beta + F\right)x + E_{c_0}^{©}$$

Pour simplifier l'écriture de ces expressions, considérons les deux équations suivantes:

$$\sin\theta = \frac{c}{x} \quad \left(\theta \in \{\alpha, \beta\}\right)$$

et

$$x = \sqrt{d^2 + c^2}$$

les équations s’écrivent donc de la manière suivante:

$$E_c = c\left(mg - F\sqrt{1 + \frac{d^2}{c^2}}\right) + E_{c_0}$$

$$E_c^{\copyright} = -c\left(mg + F\sqrt{1 + \frac{d^2}{c^2}}\right) + E_{c_0}^{\copyright}$$

3.2.3.1.3. Etude des frottements

Concernant les bornes de la force de frottement, elles sont définies de la façon suivante:

$$tg\alpha = \frac{c}{d} \quad , \quad c = 0,1,2...ma$$

d’où

$$\alpha \in \left[0, \frac{\pi}{2}\right[$$

En considérant la relation suivante:

$$mg.\sin\alpha > F \geq 0$$

nous obtenons:

$$si(\alpha = 0) \qquad = F0 \quad ,obligatoirem$$
$$si(\alpha \neq 0) \qquad F \in \left]0, \frac{1}{\sqrt{2}}\right[$$

Le cas où (F = 0) et (α = 0) est exclu, car il peut induire une oscillation à l’infini (figure3.19). Pour une même hauteur (h) des deux côtés (avec des pentes différentes ou égales), et en l’absence de frottement pour ralentir l’évolution de l’objet, celui-ci va réaliser des va-et-vient incessants sur chacune des pentes (entre les deux points définissant la hauteur h sur chacune des deux pentes).

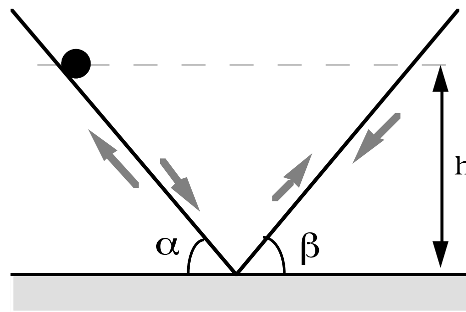


Fig.3.19. Oscillation infinie d'un objet en l'absence de frottement.

Dans le cas où les processus sont de différentes masses, celle-ci doit respecter la relation suivante:

$$c * m * g - F\sqrt{c^2 + d^2} > 0$$

avec

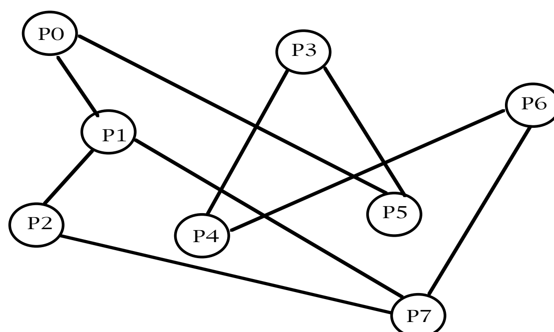
$$E_{c_0} = E^{©}_{c_0} = 0$$

d'où:

$$m > \frac{F}{g}\sqrt{1 + \frac{d^2}{c^2}}$$ ❐

3.2.3.2. Application à la régulation de charge

Les systèmes parallèles peuvent être considérés comme un graphe G=(P, L), où **P** est l'ensemble des nœuds (processeurs) et **L** l'ensemble des arêtes (liens de communication) reliant les différents nœuds.



Chacun des processeurs peut générer de nouveaux processus, accepter des processus d'autres processeurs et détruire des processus locaux. Le nombre de ceux-ci détermine au niveau de chacun des processeurs la charge de ce dernier.

La différence de charge entre deux processeurs voisins (liés physiquement) sera notée **c** dans les différentes équations. Tous les processeurs directement connectés seront séparés par une distance considérée uniforme et notée **d**. Cependant dans le cas de réseaux distribués cette distance peut intervenir pour différencier les processeurs susceptibles d'entrer en compétition lors de la recherche d'un processeur capable d'accepter une nouvelle tâche. Chaque processus sera affecté d'une masse **m**. La gravitation **g** dans ce système peut varier d'un processeur à un autre, suivant des critères à définir: le type du processeur et/ou du processus, la nature du processeur recherché (comme par exemple un processeur graphique pour le traitement d'une image)...

Au terme de l'évolution de l'objet, celui-ci s'immobilise dans un minimum de la fonction d'énergie. Il ne nous est pas possible de démontrer, de façon théorique, que ce minimum local correspond généralement à une solution de bonne qualité. Cependant, nous nous efforcerons de le mettre en évidence par des simulations (cf. paragraphe 3.2.3.4).

Il est possible d'incorporer la charge de communication d'un lien dans les différentes équations. Pour cela, suivant la charge constatée sur un lien donné, il faut multiplier par un facteur **k** (proportionnel à la charge des

communications sur ce lien) la différence de charge **c** entre les deux processeurs liés par ce canal.

3.2.3.3. Cas de la machine R.C.A.I

Dans le cas de notre machine, où les processeurs sont homogènes (non différenciés), la gravitation est la même pour tous. La distance entre les processeurs voisins est considérée comme identique. De ce fait, on peut écrire:

$$d = g = 1$$

Les équations décrites plus haut s'écrivent maintenant:

$$E_c = mc - F\sqrt{1+c^2} + E_{c_0} \quad \textbf{(1)}$$

$$E_c^{©} = -mc - F\sqrt{1+c^2} + E_{c_0}^{©} \quad \textbf{(2)}$$

$$m \rangle F \sqrt{1 + \frac{1}{c^2}} \quad \textbf{(3)}$$

$$F \langle \frac{m}{\sqrt{1+\frac{1}{c^2}}} \quad \textbf{(4)}$$

De plus, si l'on considère que les frottements sont proportionnels à la masse:

$$F = k*m \text{ avec } k \in [0,1] \quad (k \text{ est un réel})$$

les équations (1) et (2) se réécrivent définitivement comme suit:

$$E_c = m\left(c - k*\sqrt{1 + c^2}\right) + E_{c_0} \quad \textbf{(1)}$$

$$E_c = -m\left(c + k*\sqrt{1 + c^2}\right) + E_{c_0} \quad \textbf{(2)}$$

Nous donnons dans la suite l'algorithme synthétique de CLIMB avant de présenter les résultats de son évaluation.

```
/*   L'algorithme  CLIMB  pour  la  régulation  de  la  charge.  */

short     CLIMB  ( NetWork   *cell  , int    input  )
{
  int    potentialDiff,  toCell,  ddp,   i      ;
  float  energy,         initialEnergy          ;
  short  status                                 ;

  switch ( input  )
  {case  NORTH  : initialEnergy = cell -> NorthIn.energy ; break ;
   case  EAST   : initialEnergy = cell -> EastIn.energy  ; break ;
   case  SOUTH  : initialEnergy = cell -> SouthIn.energy ; break ;
   case  WEST   : initialEnergy = cell -> WestIn.energy  ; break ;
   case  CELL   : initialEnergy = 0                      ; break ;
  }

  /*   Calculer la DDP entre les cellules adjacentes.             */
  potentialDiff =  0      ;
  ddp           =  0      ;
  energy        =  0      ;
  toCell        =  UNDEF  ;
  status        =  _NOP   ;

  for  (  i      =  0;   i  <  4;   i++  )
  {  /*   Le sens du mouvement est incorporé directement dans     */
     /*   l'expression qui suit.                                  */
     potentialDiff = cell -> cellLoad - cell -> NeighboorLoad [i] ;

     if   (  toCell ==   UNDEF  )
       then {  toCell        =  i             ;
               ddp           =  potentialDiff ;
            }
       else {  if   (  potentialDiff > ddp  )
                 then {  toCell =  i             ;
                         ddp    =  potentialDiff ;
                      }
            }
  }

  if   (  toCell      !=   UNDEF  )
    then
    {  if   (  toCell ==   CELL   )
         then /*   Minimum local, cellules voisines saturées.   */
           status    =  _STORE                                  ;
         else /*   Calculer l'énergie à acquérir.               */
         {  energy    =  (ddp   *  __Masse__   *  __Gravitation__)
                      -  ((SQRT   (1   +  POW    (ddp,  2)))
                      *  __Frottements__)
                      +  initialEnergy                          ;

            if   (  energy    >=   0  )
              then status    =    _CLIMBE                       ;
              else status    =    _STORE                        ;
         }
    }

  return (status)           ;

}  /*   End  CLIMB.        */
```

3.2.3.4. Quelques résultats

Nous présentons dans ce paragraphe les résultats préliminaires de l'algorithme CLIMB.

Considérons une grille 2D de 10x10 cellules et un certain nombre de processus à créer par chaque processeur. Dans la table3.1 nous donnons la configuration sur laquelle nous avons réalisé nos évaluations, c'est à dire le nombre de processus à créer au niveau de chacun des processeurs. Le nombre de ces processus a été généré aléatoirement entre les bornes 0 et 200. La durée d'exécution de tous les processus est identique. Le comportement du réseau faisant appel à l'algorithme CLIMB a été simulé de façon très simple, et a fourni les résultats de la table3.2: le nombre de processus exécuté par chaque processeur.

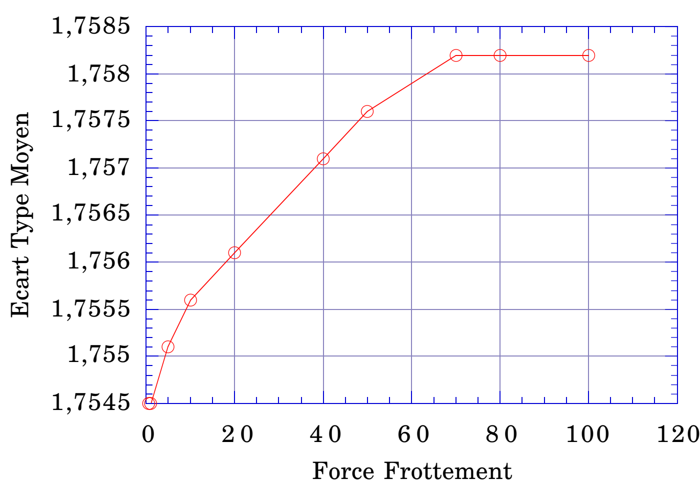


Fig.3.20. Ecart type moyen des solutions en fonction de la masse des objets et de la force de frottements[21].

[21]Pour chaque valeur de la force de frottement, on réalise une moyenne des solutions obtenues en faisant varier la masse.

38	158	113	115	51	27	**10**[*]	**19**[*]	**12**[*]	86
149	167	84	60	25	143	89	**183**[*]	137	166
166	178	95	111	167	54	31	145	82	136
124	105	**194**[*]	102	51	67	154	53	161	96
28	**188**[*]	85	143	167	**6**[*]	165	**3**[*]	162	34
153	120	44	**3**	162	118	22	127	57	145
79	**183**[*]	151	94	70	59	48	157	29	106
6[*]	**190**[*]	138	116	14	62	116	25	**181**[*]	134
143	22	33	136	160	179	71	**1**[*]	136	61
160	**5**[*]	129	44	75	93	114	139	88	121

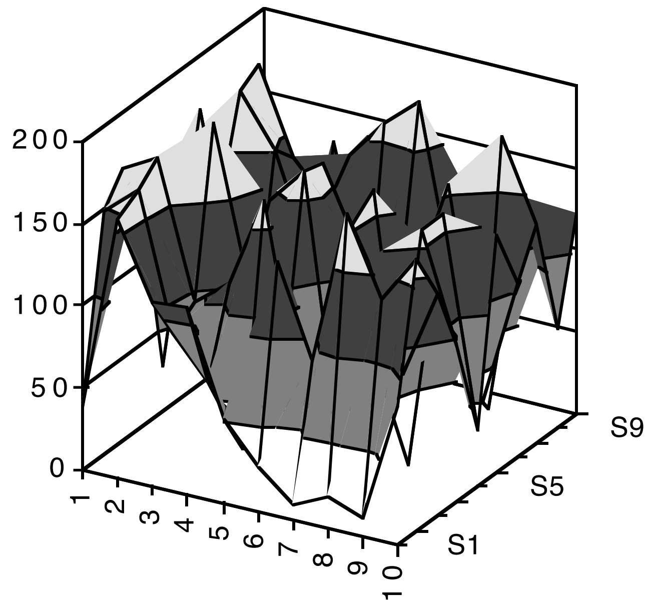
Table3.1. Nombre de processus à créer par processeur.

100	101	101	100	99	98	99	99	99	98
100	101	102	101	100	99	99	100	99	99
101	102	102	101	100	99	99	99	98	98
101	102	**103**	102	101	100	99	98	98	97
102	102	102	101	100	99	98	97	98	97
101	102	101	100	99	98	97	96	97	96
100	101	101	100	99	98	97	97	98	97
100	101	100	100	99	98	97	97	98	97
99	100	99	99	98	98	97	97	97	96
98	99	98	98	97	97	97	96	96	96

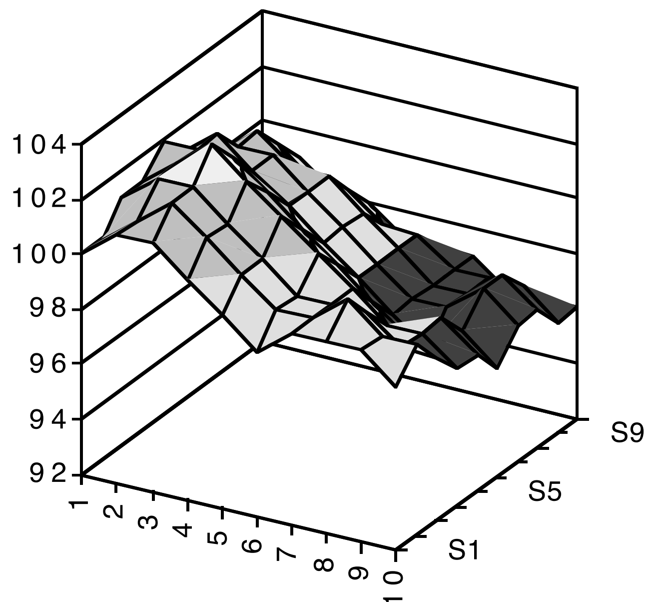
Table3.2. Charge des processeurs après application de l'algorithme CLIMB (nombre de processus exécutés par chacun des processeurs).

[*] Les valeurs en gras correspondent aux 20% des plus faibles et des plus fortes charges du réseau.

Nous donnons dans les figures qui suivent la représentation graphique des tables3.1 et 3.2 respectivement. La première correspond à une allocation sans application d'une régulation de charge sur les processeurs. Dans la seconde, nous pouvons constater le résultat obtenu en utilisant l'algorithme CLIMB.



Représentation graphique de la table3.1.



Représentation graphique de la table3.2.

Le nombre total de processus à générer est de 9904 unités, situant ainsi la moyenne à 99,04 processus par processeur.

Pour dégager la qualité des solutions obtenues après application de notre algorithme, nous calculons l'écart type de celles-ci par rapport à la moyenne du réseau. La valeur moyenne de cet écart type est de 1,7567. Cette valeur constitue l'écart moyen, constaté sur chacun des processeurs, par rapport à la moyenne globale du réseau. Dans le cas présent, ceci représente une différence de 2 processus en moyenne sur les processeurs du réseau. Ce résultat est très stable pour différentes valeurs de l'ensemble des paramètres (figure3.20).

Une première conclusion s'impose au vu des tables3.1 et 3.2. Les résultats de la table3.2 sont très satisfaisants, faisant apparaître de très faibles disparités. Toutefois, nous allons analyser plus finement l'effet de chaque paramètre.

Dans un premier temps, nous montrons qu'il n'est pas nécessaire, pour obtenir une même qualité de solution (voire meilleure), de laisser un objet osciller entre les cellules au delà d'une certaine valeur (figure3.21). Nous considérons donc le réseau de la table3.1 dans lequel nous faisons varier uniquement la masse de l'objet.

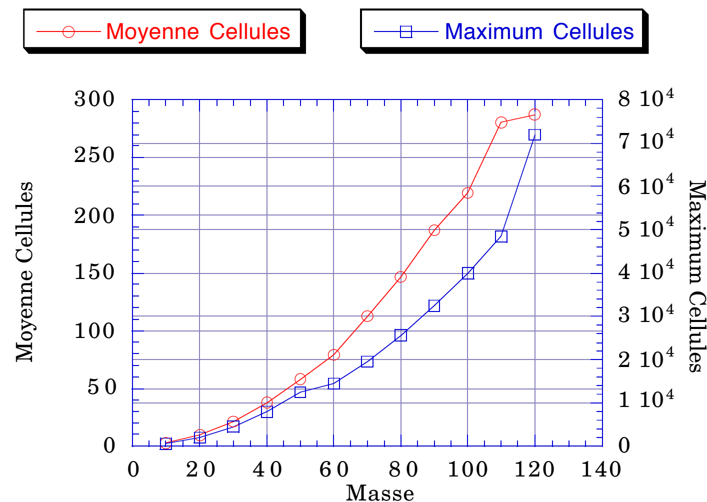


Fig.3.21. Transitions entre cellules en absence de toute limitation des oscillations.

En imposant une borne aux transitions possibles entre les cellules pour un objet, nous réduisons d'une manière significative le nombre d'oscillations entre les cellules du réseau (nous passons de 8×10^4 à 10 transitions au plus) sans pour autant nuire à la qualité finale du résultat obtenu.

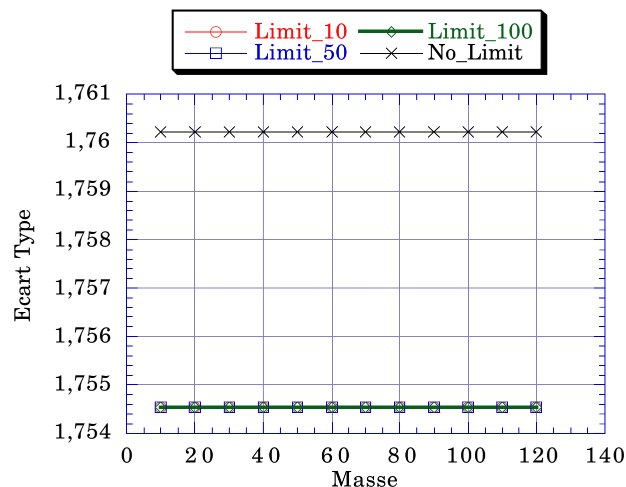


Fig.3.22. Comparaison de différentes bornes pour les transitions d'un objet dans le réseau[22].

Cela est dû principalement au fait que les objets s'éloignent en moyenne de 0,0519 unités de leur source. La distance maximale de la source constatée

[22] Les courbes associées aux valeurs 10, 50 et 100 du nombre d'oscillations maximales sont pratiquement confondues sur ce graphique.

dans cet exemple est de 13 unités. Les résultats correspondants à la qualité des solutions suivant le nombre de transitions fixées sont reportés dans la figure3.22. Dans la suite nous considérons que les objets sont limités à 100 transitions au maximum.

Il est important d'expliciter quelques notions pour la compréhension ultérieure des figures. La distance moyenne notée dans celles qui suivent ont une relation directe avec le nombre de processus transférés sur d'autres processeurs (que le processeur source), mais ne constitue pas en soi un indicateur direct sur leur nombre. En effet, une valeur du genre 0,06 unités signifie que les processus s'éloignent en moyenne de cette valeur de leur source. Cependant, cela ne signifie nullement qu'il y a 6% des processus qui s'éloignent de leur source (comme on pourrait le penser à priori).

Pour s'en convaincre, nous allons vérifier cette relation en considérant successivement une même configuration avec des charges initiales des processeurs différentes (figure3.23). Les valeurs de l'axe des abscisses désignent, pour chacune d'elles, la charge des processeurs avant l'application de l'algorithme CLIMB. Cette charge est générée aléatoirement entre les bornes 0 et la valeur considérée.

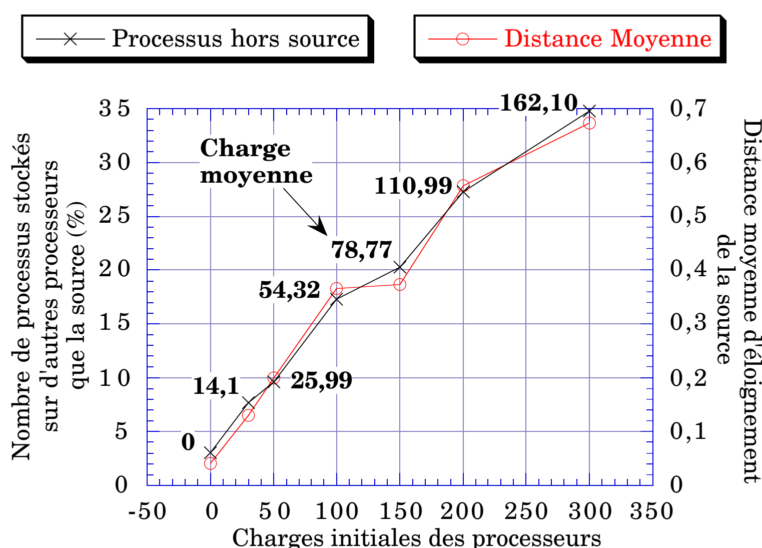


Fig.3.23. Relation entre la distance d'éloignement et le nombre de processus ayant effectivement quitté leur processeur source.

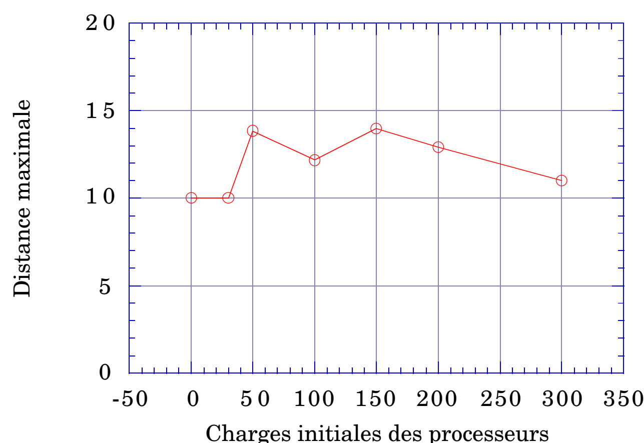


Fig.3.24. Distance maximale d'éloignement de la source en fonction des charges initiales des processeurs.

La moyenne des charges des processeurs à cet instant est reportée à côté du point relatif sur la courbe. L'application de notre algorithme intervient alors[23] et fournit sur la seconde courbe le nombre de processus qui ont effectivement quitté leur source vers d'autres processeurs.

La distance maximale d'éloignement des processus de leur source est schématisé dans la figure3.24. Dans le reste des évaluations effectuées dans ce chapitre, le nombre de processus qui quittent leur processeur source reste faible en raison de l'efficacité de l'algorithme CLIMB dès les premières applications. En effet, CLIMB tend à uniformiser à chaque étape la charge des différents processeurs, en se rapprochant tout le temps de la moyenne de la charge du réseau[24]. De ce fait, les processus au niveau d'un même processeur, qui se trouvent dans un voisinage de même charge (ou de plus forte charge), restent au sein du processeur source. C'est ce qui explique la faible valeur de la moyenne des distances relevées dans les différentes simulations qui suivent. La figure3.23 montre qu'en partant de charges initiales différentes sur les processeurs du réseau, cette distance s'accroît chaque fois que cette différence prend de l'ampleur.

---

[23]Nous utilisons la même configuration de la table 3.1 avec des processeurs déjà chargés.

[24][SALETORE_90] et [KALE_88] suggèrent d'ailleurs de procéder à la régulation de la charge à chaque nouvelle création de processus sans attendre de dépasser un seuil pour activer cette opération (comme dans le cas du gradient par exemple).

Nous allons tenter à présent de mettre en avant l'influence des différents paramètres sur la qualité des solutions obtenues après la mise en œuvre de notre algorithme: la masse de l'objet, la gravitation exercée sur celui-ci ainsi que la force de frottement qui lui est appliquée.

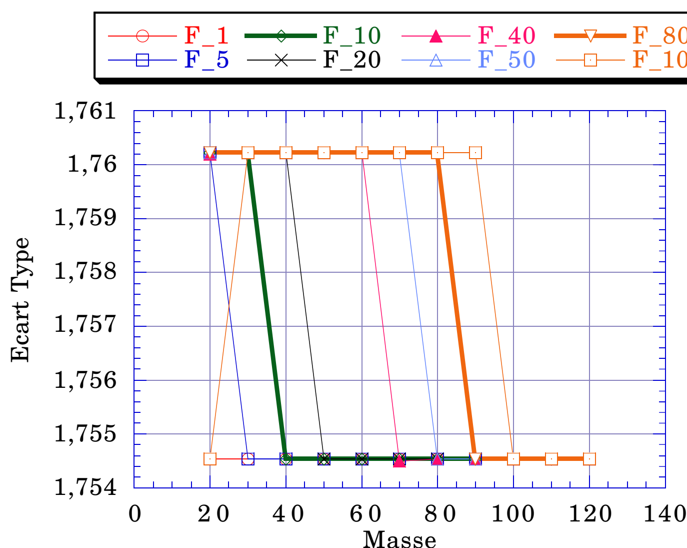


Fig.3.25. Qualité des solutions en fonction de la masse des objets et de la force de frottement.

La figure3.25 montre l'écart type pour différentes valeurs de la masse et de la force de frottement au niveau du réseau décrit précédemment (table3.1). Nous pouvons noter que l'ensemble des courbes convergent vers une même solution. Cela signifie que pour arriver à de meilleures solutions, la force de frottement doit rester suffisamment faible (par rapport à la masse de l'objet) pour ne pas altérer la qualité des résultats de l'algorithme CLIMB.

Par ailleurs, la figure3.26 montre que parallèlement, il n'est pas nécessaire de trop s'éloigner de la source pour obtenir les meilleurs solutions. En effet, ce contraste peut s'expliquer par le fait qu'un objet qui s'éloigne de la source perd rapidement de l'énergie (surtout lorsqu'il est léger) et ne peut donc plus s'échapper de minima locaux dans lesquels il risque de tomber.

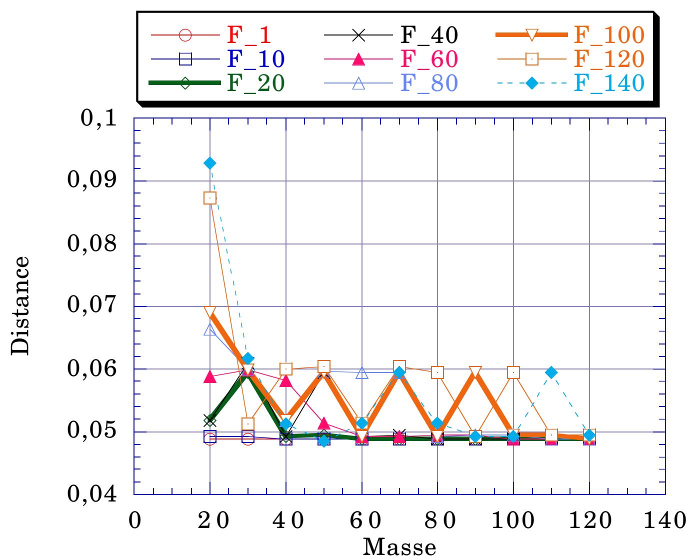


Fig.3.26. Éloignement des objets de leur source en fonction de leur masse et de la force de frottement.

Le nombre moyen de cellules visitées est reporté au niveau de la figure3.27. Nous constatons que pour atteindre une certaine efficacité de la solution, il faut permettre aux objets de transiter par plusieurs cellules. Le nombre de ces cellules reste toutefois limité (2,5 cellules en moyenne dans le cas considéré).

Nous pouvons noter que plus la force de frottement est élevée, moins un processus peut s'éloigner du processeur source (figure3.27). Cet état peut être contrebalancé par une charge plus forte, comme nous le constatons sur cette même figure.

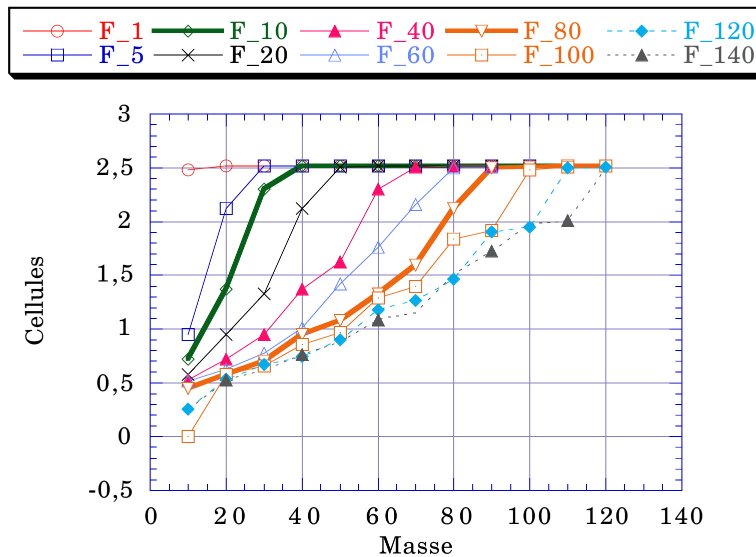


Fig.3.27. Nombre de cellules visitées en fonction de leur masse et de la force de frottement.

Les résultats obtenus, en faisant varier la gravitation, semblent corroborer les résultats précédents. En effet, pour empêcher un objet de trop s'éloigner de la source, et ainsi converger vers la moyenne du réseau, il faut que la gravitation soit suffisamment importante - par rapport à la masse (figures3.28, 3.29 et 3.30).

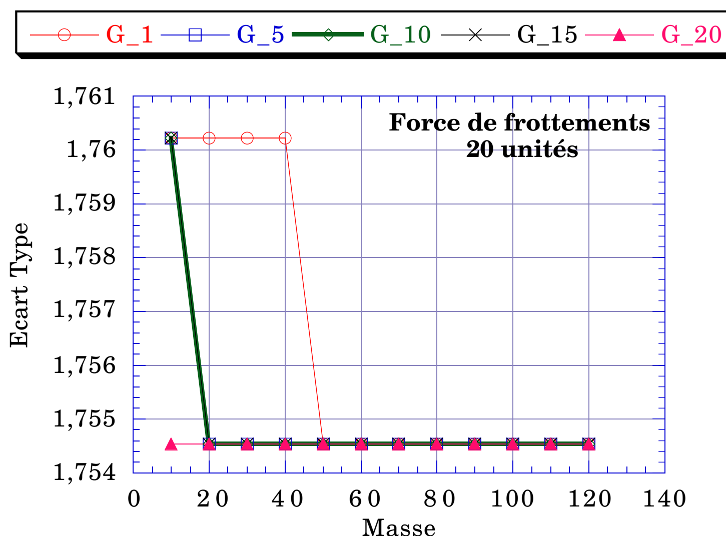


Fig.3.28. Qualité des solutions en fonction de la masse des objets et de la gravitation.

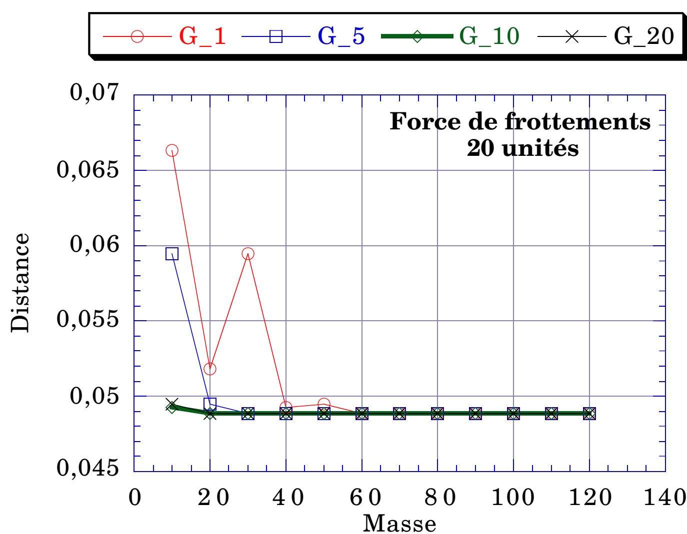


Fig.3.29. Éloignement des objets de leur source en fonction de leur masse et de la gravitation.

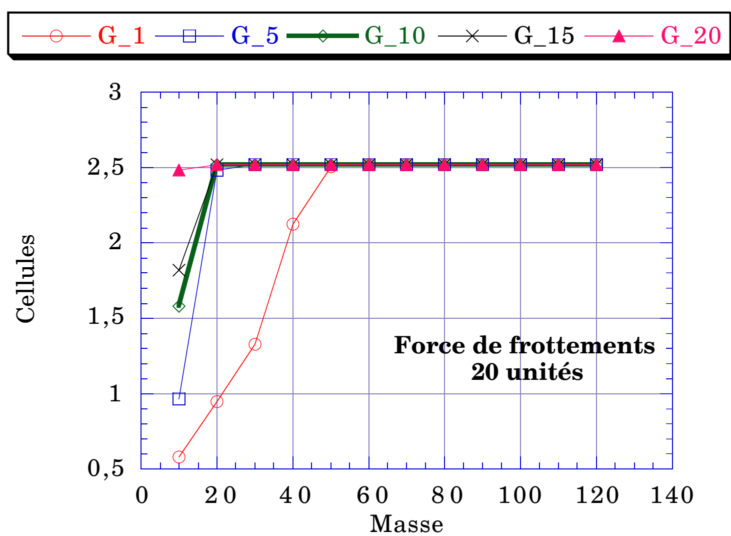


Fig.3.30. Nombre de cellules visitées en fonction de leur masse et de la gravitation.

A titre d'illustration, nous donnons l'écart type correspondant à des mesures de frottements proportionnels à la masse d'un objet.

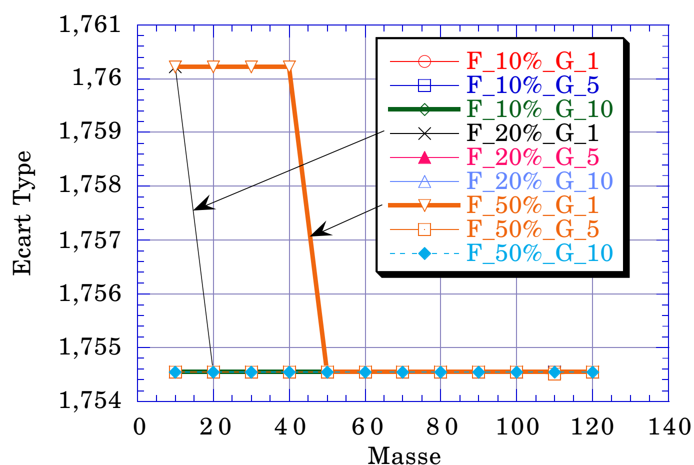


Fig.3.31. Qualité des solutions en fonction de frottements proportionnels à la masse des objets et de la gravitation.

La figure3.31 exprime le fait que nous pouvons compenser des frottements trop importants par rapport à la masse d'un objet par une force de gravitation plus faible. C'est le cas des forces de frottement à 50% et à 20% de la masse de l'objet considéré.

3.2.3.5. Conclusion

Pour clore ces résultats préliminaires, nous montrons que cet algorithme est efficace pour différentes tailles de réseaux (figures3.33 et3.34). Pour cela, nous considérons des réseaux de 20x20 et 40x40 cellules. Chaque processeur peut générer, aléatoirement, un nombre de processus compris entre 0 et 100.

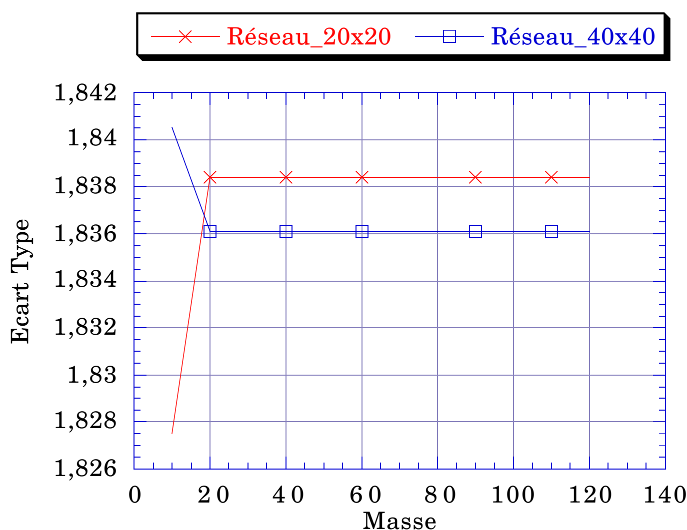


Fig.3.32. Qualité des solutions pour de grandes tailles de réseaux.

Nous pouvons noter que l'écart type se situe entre les valeurs 1 et 2 (figure3.32); ceci signifie qu'en moyenne, il y a une différence d'au plus deux processus sur les différents processeurs, par rapport à la moyenne du réseau.

En conclusion, nous dirons que l'ensemble des résultats présentés dans cette partie montrent l'efficacité de notre algorithme.

Nous n'avons pas comparé CLIMB à d'autres algorithmes pour la régulation de charge pour la simple raison que les résultats que fourniraient ces derniers ne peuvent pas être significativement meilleurs.

Cependant, l'intérêt principal de notre algorithme, outre ses résultats de bonne qualité, est le faible nombre de messages échangés, aussi bien pour la recherche d'un processeur peu chargé, que pour la mise à jour des éléments d'information nécessaires pour prendre une bonne décision en vue de réguler la charge du réseau.

A titre d'exemple, dans le cas du réseau de 40x40 cellules, nous avons une première distribution aléatoire de processus, reflétée par la figure 3.33.

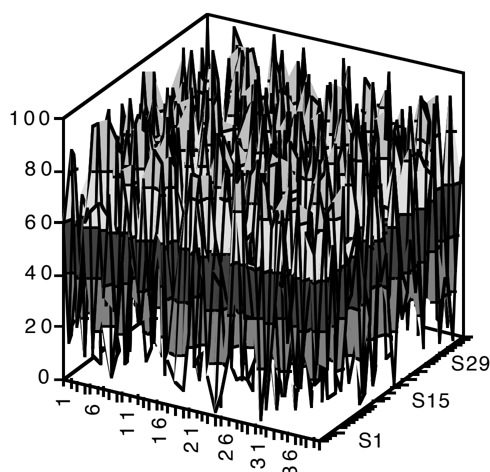


Fig.3.33. Un réseau 40x40 cellules sans régulation de charge.

L'application de l'algorithme CLIMB donne le résultat de la figure 3.35. Nous constatons une distribution uniforme très nette des charges sur les différents processeurs.

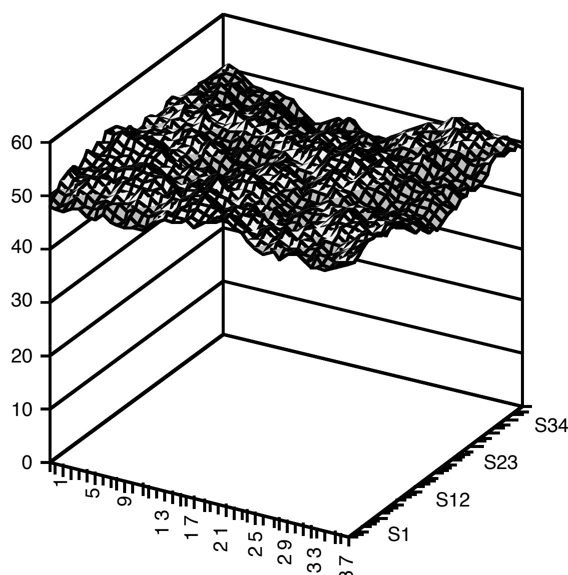


Fig.3.35. Un réseau 40x40 cellules après application de l'algorithme CLIMB pour la régulation de charge.

3.3. Ramasse-miettes et gestion de la mémoire

3.3.1. Nécessité d'un ramasse-miettes

La réclamation d'objets inactifs est nécessaire dans les systèmes où la ressource mémoire est considérée comme étant limitée. Dans un premier temps nous considérons la notion d'objet au sens le plus large du terme[25].

La fonction d'un collecteur d'objets inactifs[26] est de retrouver, par un quelconque mécanisme, tous les objets inactifs. Un objet est dit inactif s'il n'est plus référencé par aucun processus du système considéré. A l'opposé, tout objet potentiellement accessible par un ou plusieurs processus, via une série de pointeurs, est dit actif. De tels objets doivent être préservés par l'opération de collection d'objets inactifs. La notion d'activité est une propriété globale à partir du moment où tout objet peut être référencé par plusieurs processus.

Dans un environnement centralisé, une liste commune sera gérée par l'ensemble des objets créés dans le système. Par contre, dans un environnement distribué, la tâche du collecteur devient plus ardue. La répartition de la liste des objets du système sur l'ensemble des sous-systèmes qui le constituent, nécessite une gestion de la cohérence des informations enregistrées au niveau de chacune des sous-listes.

En résumé, nous dirons qu'un collecteur doit réaliser essentiellement les deux fonctions suivantes:

- déceler les objets inutilisés dans le système (détection),
- reprendre l'espace qu'ils occupaient pour une utilisation future par d'autres processus (récupération)[27].

La détection d'objets inactifs repose sur le principe d'accessibilité: en partant d'entités dites "racines" - censées exister tout au long de l'exécution

---

[25]La notion d'objet est étendue de la simple position mémoire (entier, caractère, structure de donnée...) jusqu'aux "objets" définis suivant l'approche orientée objets.

[26]Nous désignerons par la suite le collecteur d'objets inactifs par simplement **collecteur**.

[27]Les considérations de l'émiettement de la mémoire qui pourraient en résulter ne sont pas niveau.

d'un programme, tous les objets à leur portée seront désignés actifs. L'espace occupé par les autres objets doit être récupéré.

Le processus de récupération est généralement déclenché lorsqu'une demande de mémoire est insatisfaite.

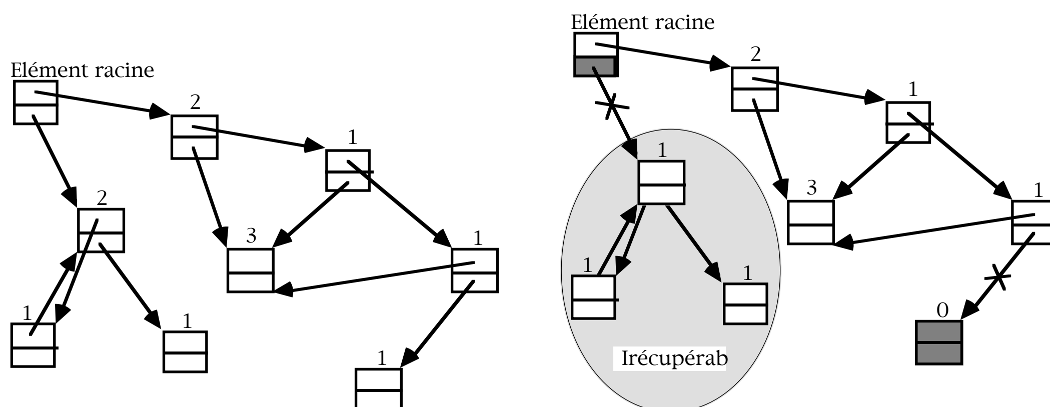
Une fois le processus de récupération engagé, le collecteur ne s'intéressera qu'aux “variables globales”. En effet, les objets locaux à une portion de code sont considérés actifs tant que celle-ci l'est également.

Il existe différentes méthodes pour réaliser la récupération d'espace inutilisé. Nous allons en exposer quelques unes, des plus connues à celles qui le sont moins.

3.3.2. Algorithme du compteur de références

Dans cette approche, chaque objet est doté d'un compteur qui représente le nombre de références qui lui sont faites [DEUTSCH76] [DeTREVILLE90]. A chaque nouvelle référence à un objet, son compteur est incrémenté de 1. De même, lorsqu'une référence à un objet est supprimée, son compteur est décrémenté de 1. Au moment où celui-ci passe à zéro, l'espace que cet objet occupait est récupéré, puisque cet état indique qu'il n'y a plus de références à cet objet (inaccessible par les autres objets). Les objets ainsi récupérés sont généralement liés dans une liste d'objets libres pour être réutilisés ultérieurement. A la récupération d'un objet tous les compteurs des éléments qu'il pointe sont décrémentés de 1.

L'avantage de cette méthode est qu'elle fait partie intégrante du système, et évolue avec lui en temps réel, sans pour autant induire de traitements supplémentaires excessifs. Les seules opérations nécessaires sur un objet sont les incrémentations et décrémentations, ainsi que le test d'un compteur. Cependant, si les objets ont une duré de vie éphémère, les opérations de mise à jour augmentent proportionnellement avec le nombre de ces objets et ainsi deviennent préjudiciable pour le bon déroulement du programme en question.



L'inconvénient majeur de cet algorithme est l'impossibilité de récupérer des structures circulaires. Dans un circuit formé par deux objets ou plus, leur compteur de références ne passe jamais à zéro et ne sont donc pas récupérables.

Nous pouvons proposer notamment deux remèdes à ce problème:

- Le premier consiste à introduire un second mécanisme de récupération d'espace (parmi ceux décrits par la suite). Cette seconde méthode est lancée à l'échec d'une réclamation d'espace. Toutefois, cette solution nuit aux applications temps réel, puisque ce sont généralement des méthodes qui supposent l'arrêt momentané de l'application en vue du nettoyage de la mémoire.

- Le second remède est soit d'interdire tout simplement la création de tels circuits en imposant un style de programmation “arborescent” (CANTOR), soit de les réduire à quelques modèles [BOBROW80] ce qui, dans les deux cas, limite le type d'applications envisageables.

3.3.3. Algorithme “Mark & Sweep” (Récupérer & Nettoyer)

Cet algorithme procède en deux phases [HAYES91] [ZORN90]:

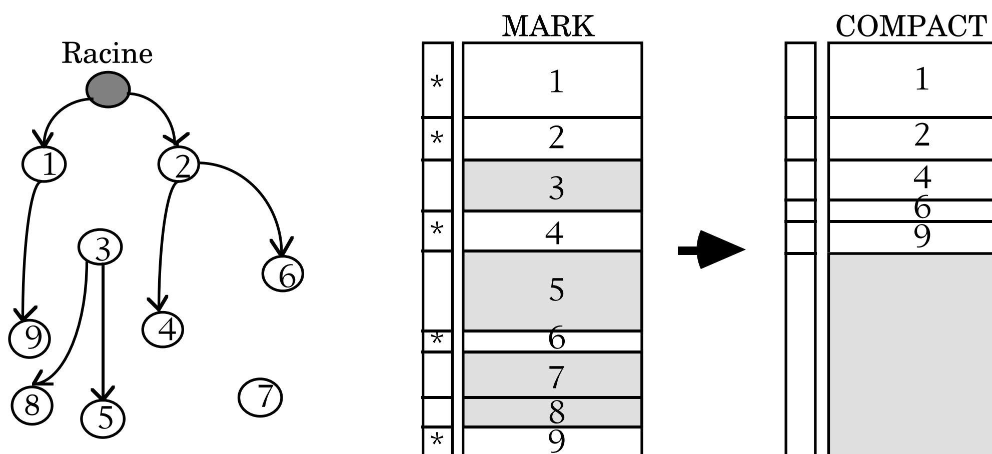
- Marquage: en partant des éléments racines, puis de proche en proche, tous les objets accessibles sont marqués suivant une certaine méthode (en altérant leur état, en les enregistrant dans une structure spécifique...).
- Récupération: une fois les différents objets marqués, tous ceux qui ne l’ont pas été sont considérés désuets et ainsi, leur espace est enregistré (en règle générale) dans une liste d’objets libres.

Lorsque nous gérons des éléments de mémoire de tailles différentes, un inconvénient de cette méthode est l’émiettement de la mémoire. Ce problème se rencontre également dans la stratégie précédente. La solution à ce problème relève du type de gestion de la mémoire adopté (BEST FIT, WORST FIT...) et nécessite donc un traitement supplémentaire.

La seconde difficulté, entraînée par la précédente, est le mélange de “générations” d’objets: des objets récemment créés sont mêlés à de plus anciens, pénalisant ainsi la localité des références. Par conséquent, ces méthodes ne sont pas adaptées à la gestion de la mémoire virtuelle. Les références dispersées à travers différentes pages impliquent d’incessants vas-et-vients entre ces pages.

3.3.4. Algorithme “Mark & Compact” (Marquer & Compacter)

Le problème de la localité des références est résolu par cette autre approche. Après avoir marqué dans une première phase tous les objets accessibles à partir d’objets racines, les objets sont ensuite “compactés” [COHEN83]. Tous les objets actifs sont déplacés vers le haut (ou vers le bas) de la mémoire jusqu’à ce qu’ils ne forment plus qu’un seul bloc contigu (laissant ainsi en un seul bloc le reste de la mémoire libre).



La localité des références est ainsi sauvegardée: les objets sont “reconstitués” suivant leur ordre de création, et ne sont pas entremêlés à des objets plus récents.

L’inconvénient de cette méthode est la nécessité de réaliser plusieurs passages sur chacun des objets: une première fois pour les marquer, une seconde pour leur trouver un nouveau emplacement en mémoire, et enfin une phase de translation des adresses. Si beaucoup d’objets survivent à une opération de récupération, le temps nécessaire pour réaliser cette dernière est par conséquent important. Il existe différentes variantes de cet algorithme [COHEN83].

3.3.5. Algorithme “Stop & Copy” (Arrêter & Copier)

Tout comme l’algorithme précédent, celui-ci réalise le compactage des objets dans une même zone contiguë. Cependant, contrairement à l’algorithme “mark & compact”, les phases de marquage puis de copie sont réalisées en une seule étape. La mémoire est généralement découpée en deux zones distinctes: la zone courante (*from space*), où sont alloués les nouveaux objets, et la zone future (*to space*), qui représente la zone destination des objets qui survivent à l’opération de ramasse-miettes. Durant cette opération, les objets sont progressivement copiés vers la zone future au fur et à mesure qu’ils sont accédés [ZORN90] [CHENEY70]. Une fois cette opération terminée, les deux zones intervertissent leur rôle respectif: la zone courante devient la zone future et inversement.

3.3.6. Algorithmes “générationnels”

La dernière famille d’algorithmes de récupération d’objets concerne les algorithmes dits générationnels (“*Generational Garbage Collection Algorithms*“) [HAYES91] [DeTREVILLE90] [UNGAR84]. Ces derniers répartissent les objets en plusieurs “générations” suivant l’âge de ces objets. L’âge d’un objet correspond au temps depuis lequel cet élément est présent dans le système. Ces algorithmes sont capables de réaliser l’opération de récupération dans des délais très courts, en concentrant leur effort de recherche sur les générations les plus jeunes (c’est-à-dire les objets récemment alloués). Cette catégorie d’algorithmes par du fait que les objets les plus récents ont des durées de vie nettement plus courtes que celle des objets les plus anciens [APPEL89] [UNGAR88].

Une zone mémoire est attribuée à chaque génération d’objets. Tous les nouveaux objets sont alloués dans la zone mémoire réservée à la génération la plus jeune. Les objets sont “promus” des jeunes générations vers de plus anciennes suivant leur capacité (traduite par le nombre de fois) à résister (survivre) à plusieurs opérations de récupération.

Généralement, l’opération de récupération, pour une génération k, est engagée lorsque le taux d’espace libre restant est en deçà d’un seuil fixé pour l’ensemble des générations ou, au contraire, ce seuil est fixé indépendamment pour chaque génération [LIEBERMAN83] [MOON84].

Pour permettre la récupération d’une génération k sans pour autant accéder aux générations plus anciennes, une table de références en arrière au niveau de la génération k (qui regroupe toutes les références à partir des objets dans les anciennes générations vers ceux de la génération k) doit être mise à jour pour maintenir la cohérence des pointeurs lors de la collecte des objets dans k. Un “ensemble de rappel” (*remembered set*) est utilisé pour indiquer les objets anciens qui effectuent ces références en arrière. De même, il est important de garder trace des références en avant (depuis la génération k vers les générations plus anciennes), toujours pour assurer la cohérence des références. Cependant, l’explosion de telles tables nécessite de prendre des précautions quant à leur utilisation. La solution proposée par [LIEBERMAN83], par exemple, consiste à collecter toutes les générations i, tel que $i \leq k$, afin d’éviter d’inclure dans les tables de rappel les références à partir des générations plus jeunes qui sont les plus nombreuses dans le système.

Nous nous proposons à présent d'aborder la récupération de la mémoire dans un environnement parallèle.

Les premiers travaux sur les algorithmes parallèles de ramasse-miettes débutèrent avec l'étude des approches "au-vol" pour la récupération d'espace. Dans cette approche, deux processeurs distincts opèrent sur une mémoire commune. L'un (le "*mutateur*") réalise le travail "effectif" (il s'agit de l'application proprement dite) et l'autre (le "*collecteur*") récupère l'espace inutilisé. Les algorithmes développés pour le ramasse-miettes au-vol ont été utilisés plus souvent comme des exemples dans les techniques de preuve de programmes parallèles que dans les systèmes réels [BEN_ARI84] [DIJKSTRA78]. Ces algorithmes ont parallélisé les traditionnels algorithmes "*mark-and-sweep*". Ils héritèrent donc des inconvénients de ces algorithmes, et en premier lieu, la nécessité d'accéder à un grand nombre d'objets, aussi bien inactifs qu'actifs.

Cet inconvénient, comme nous l'avons vu auparavant, a été corrigé par les méthodes dites de "*copy collection*" (tels que les algorithmes "*stop & copy*").

Par la suite, les algorithmes générationnels ont affiné et réduit la copie d'objets stables, en concentrant la majeure partie des efforts sur les générations d'objets "jeunes".

Nous illustrons l'approche parallèle par deux algorithmes dus respectivement à [SHARMA91] et à [HERLIHY93]. Le premier concerne une parallélisation de l'approche générationnelle et le second propose une solution qui ne repose sur aucun système de synchronisation pour mener à bien sa tâche, contrairement au précédent.

3.3.7. Un algorithme "générationnel" parallèle

La parallélisation de l'approche "générationnelle" peut intervenir à deux niveaux:

- l'opération de récupération peut être entreprise en parallèle avec le programme courant,
- ou encore, et l'auteur [SHARMA91] met l'accent davantage sur cet aspect, les tâches parallèles du "ramasseur" (collecteur) peuvent opérer en concurrence, chacune étant chargée de la récupération des objets inactifs dans une génération distincte.

La méthode présentée considère la parallélisation d'un ramasse-miettes générationnel sur une architecture multiprocesseurs à mémoire partagée.

Le collecteur est composé de plusieurs entités qui travaillent en parallèle pour exploiter au mieux l'architecture sous-jacente. Le calcul peut évoluer concurremment avec le collecteur pendant la phase de collecte. L'avancement (promotion) d'un objet au travers des générations repose sur le nombre de fois que ce dernier survit aux différentes phases de collecte.

Tout comme dans le cas de [LIEBERMAN83], la collecte d'une génération k nécessite celle de toutes les générations i, tel que $i \leq k$, pour les mêmes raisons évoquées au paragraphe3.3.6. Toutefois, la collecte de chacune des générations est réalisé en parallèle. Cette parallélisation rend nécessaire l'utilisation de verrous pour assurer la synchronisation entre les différentes tâches parallèles. Pour éviter le cas d'interblocage, causé par une attente circulaire sur les verrous, l'auteur utilise une méthode de prévention: pour chaque génération, il existe différentes classes de verrous qui contrôlent chacune l'accès à différentes variables.

De plus, pour éviter la prolifération des "ensembles de rappel" (cf.§3.3.6), cette technique recourt à une table globale de rappel pour tout le système. Cette table contient toutes les pages (mémoires) qui possèdent des objets qui ont des références en arrière vers des générations plus jeunes que la leur. Lors d'une collecte des générations 1 à k, les pages mémoires désignées par l'ensemble de rappel sont parcourues uniquement lorsqu'elles appartiennent à des générations plus anciennes (donc plus grandes que k).

La mémoire est divisée en plusieurs générations, dont une particulière: le "vieil" espace (*Old Space - OS*), qui contient les objets les plus anciens (suite à de multiples promotions de génération en génération). Les autres générations sont divisées en deux parties: la partie courante et la partie future (figure3.35). La première partie contient une zone appelée "nouvelle

zone", où sont alloués tous les nouveaux objets (de cette génération) ainsi que les objets promus depuis les générations plus jeunes vers elle. La partie "future" est utilisée lors de la collecte de cette génération (en inter changeant à ce moment les deux parties[28]).

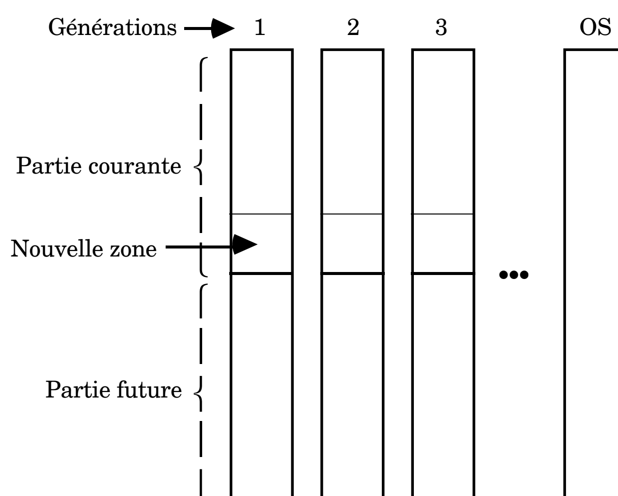


Fig.3.35. Division de l'espace mémoire dans [SHARMA91] **.**

Le principe général de cet algorithme peut être résumé comme suit. Lorsque l'espace libre est jugé insuffisant (eu égard à un seuil) pour assurer les prochaines demandes en mémoire, le processus de réclamation est enclenché. L'application est alors temporairement suspendue et le collecteur se met, éventuellement, en attente de la fin d'exécution d'une précédente opération de ramasse-miettes. Il intervertit les deux zones de chaque région (une par génération) dans les k premières générations. Puis, il copie les objets racines des ces premières générations dans la zone appelé "partie courante". Un objet racine est défini comme étant tout objet accessible directement par les processus de l'application. Le collecteur détermine ensuite et marque non-explorées toutes les pages de l'ensemble de rappel qui se trouvent dans des générations plus anciennes que k (car ces pages contiennent des objets qui doivent être examinés). Il initie en parallèle le processus de ramasse-miettes pour chacune des générations comprises entre 1 et k. Finalement, le collecteur reprend l'activité suspendue de l'application.

Pour préserver la cohérence des pointeurs, toutes les pages non encore examinées sont verrouillées, interdisant ainsi l'accès aux objets de cette page tant que celle-ci n'a pas été traitée par le collecteur.

[28]Dans la littérature, cette opération est souvent désignée sous le nom de *flip*.

3.3.8. Un algorithme parallèle non-bloquant

Comme nous venons de le voir, dans [SHARMA91] [APPEL88] et bien d'autres, les algorithmes de ramasse-miettes pour les multiprocesseurs à mémoire partagée reposent d'une façon générale sur une synchronisation globale (qui peut se présenter sous différents aspects) pour préserver une cohérence entre les différentes références entre objets. Néanmoins, une telle synchronisation globale n'est pas appropriée à des architectures asynchrones: lorsqu'un processus est arrêté ou mis en attente, d'autres processus, corrects, seront incapables de progresser (car liés à ces derniers). D'autre part, un algorithme de gestion de l'allocation mémoire est non-bloquant si, en absence d'épuisement de ressources (il s'agit typiquement de la mémoire), un processus en pleine allocation ou récupération de mémoire peut subir un retard considérable sans pour autant forcer d'autres processus à se bloquer.

[HERLIHY93] présente un algorithme de copie incrémentale non-bloquant pour la récupération de la mémoire: la synchronisation dans cet algorithme est établie en appliquant à la mémoire partagée les opérations de base *read*, *write* et *compare&swap*. Cet algorithme n'utilise aucune notion de verrou ou d'attente active; de même, un processus ne peut pas observer ou modifier les variables locales ou registres d'un autre processus, ni recourir à des interruptions inter-processus.

Le modèle de mémoire adopté par l'auteur est composé de trois aspects:

L'architecture sous-jacente

Dans cet algorithme, l'auteur considère des architectures MIMD sur lesquelles n processus asynchrones partagent une mémoire globale. Cependant, chaque processus possède une zone privée (la pile et les registres), inaccessible aux autres processus. Les opérations primitives de la mémoire sont:

- *read*, qui copie une valeur de la mémoire partagée vers sa propre zone privée,
- *write*, qui réalise l'opération inverse de la primitive précédente,
- *compare&swap*, pour réaliser une synchronisation de base et dont le code de la primitive est le suivant:

```
compare&swap  (w: world, old, new: value) returns (boolean)
  if  w = old
    then  w := new
          return  true
    else  return  false
  endif
end compare&swap.
```

Le niveau application

Il s'agit à ce niveau de la sémantique de la gestion de la pile mémoire au niveau application. Une application possède des variables locales et partage, avec d'autres processus, un ensemble d'objets. Du point de vue de l'application, un objet apparaît sous la forme d'un vecteur de valeurs, de taille fixe. Une valeur peut être un booléen, un entier ou un pointeur vers un autre objet.

Les opérations de base à ce niveau sont:

- *create (s)*, qui crée un objet de taille *s* et retourne le pointeur sur cet objet,
- *fetch (x, i)*, qui prend en argument un pointeur vers un objet *x* et un indexe *i* dans cet objet, et retourne la valeur du composant ainsi indéxé,
- *store (x, i, v)* qui range la valeur *v* dans l'objet *x* à la position *i*.

L'aspect structurel

Cet aspect concerne la structuration de la mémoire partagée, dans le but de supporter la sémantique du niveau application.

La mémoire est partitionée en *n* régions contiguës: une pour chaque processus. Un processus peut accéder à n'importe quelle position mémoire, mais, par contre, ne peut allouer ou collecter de l'espace mémoire que dans sa propre région uniquement.

Un objet est représenté sous la forme d'une liste chaînée de *versions*. Chaque version est contenue dans la région mémoire d'un seul processus. Chaque version pointe vers la suivante. La dernière, qui ne pointe sur aucune autre, est la *version courante*. N'importe quel processus peut déterminer le processus au niveau duquel réside une adresse *x* par le biais de la fonction *owner(x)* .

Les primitives de gestion de la mémoire à ce niveau sont:

- *find_current(x)* , détermine la version courante d'un objet x,
- *fetch(x, i)* , lit le contenu du composant indexé par i dans l'objet x,
- *store(x, i, v)* , modifie l'objet x en créant et chaînant cet élément dans une nouvelle version courante.

L'utilisation de plusieurs versions d'un même objet permet de réaliser des mises à jours concurrentes sur un même objet sans avoir recours à l'exclusion mutuelle. De même, cette organisation permet de "déplacer" un objet, lors de la phase de copie de l'opération de récupération de la mémoire, en le chaînant, sans avoir à le verrouiller.

Le principe de cet algorithme est celui des algorithmes qui procèdent par copie d'un espace mémoire vers un autre. Chaque région (associée à un processus) est partagée en plusieurs zones contiguës: une seule zone "toSpace", zéro ou plus de zones "fromSpace", et zéro ou plus de zones libres (se reporter au §3.3.5 pour l'origine de ce découpage).

Basé sur le modèle que nous venons de décrire, l'auteur propose une stratégie, non-bloquante, de parcours des différentes zones mémoires lors de l'opération de ramasse-miettes. Cette opération est initiée lorsqu'un seuil du taux d'occupation de la zone libre d'allocation est dépassé.

3.3.9. Conclusion

La fonction de ramasse-miettes est un service qui est soit intégré dans le langage de programmation (tel que LISP, EIFFEL...), soit offert par l'environnement d'exécution. L'introduction d'une telle fonction permet au programmeur de s'affranchir des considérations d'allocation/restitution de ressources et de se concentrer davantage sur les aspects fonctionnels de son application. Ceci permet en outre d'accroître la productivité et la fiabilité du logiciel qui en résulte.

Globalement, le ramasse-miettes se présente sous l'une des trois formes suivantes:

- Les algorithmes avec compteur de références; ces algorithmes comptent le nombre de références pour chaque objet. Les objets dont le compteur passe à zéro sont récupérés. Les algorithmes à compteur de références pures sont incapables de récupérer des structures de données circulaires.
- Les algorithmes de traçage; ces derniers parcourent les structures de données afin de déterminer les objets inaccessibles. Les objets accessibles sont marqués, les autres sont récupérés.

  Nous pouvons aussi ranger dans cette famille les algorithmes de coloriage. En effet, leur principe revient à marquer les objets, suivant une certaine stratégie, puis à récupérer tous les objets d'une certaine couleur [KAFURA90].
- Les algorithmes générationnels; ces algorithmes, qui semblent retenir de plus en plus l'attention des chercheurs ([HAYES91] [DeTREVILLE90] [APPEL89] [UNGAR88]...), sont généralement plus rapide que ceux de la seconde famille. Leur principe est de diviser les objets en plusieurs classes, en fonction de leur âge. Les objets sont le plus souvent récupérés parmi les générations les plus jeunes. L'espace de recherche d'objets inutilisés est ainsi réduit.

En ce qui concerne les algorithmes parallèles de ramasse-miettes, la plupart considèrent une mémoire globale distribuée [HERLIHY93] [SHARMA91][LANGENDOEN92] ... Lorsque ce n'est pas le cas, on suppose un système de désignation globale, distribué sur les nœuds du réseau [AGHA94] ou encore la gestion de la pile est centralisée [LADIN92].

Ce que nous pouvons retenir des deux algorithmes parallèles que nous avons présentés dans ce chapitre[29], c'est que le passage d'un milieu monoprocesseur vers un milieu multiprocesseurs à mémoire commune partagée nécessite une coopération entre les différentes entités du collecteur distribué, et surtout, un mécanisme de synchronisation. La philosophie inhérente à chaque approche (méthode de copie ou répartition des objets par génération) reste inchangée lors de ce passage.

Nous avons donc pensé qu'il serait possible de considérer, à la manière de [KAFURA90], un ramasse-miettes adapté au modèle acteur. Toutefois, nous allons plus loin que la solution proposée par Kafura en proposant une stratégie parallèle (distribuée). Rappelons que la méthode de Kafura ,élaborée pour un contexte monoprocesseur, repose sur un algorithme de coloriage qui détermine les acteurs accessibles de ceux qui ne le sont pas (suivant leur couleur). Ces derniers doivent restituer l'espace qu'ils occupent.

Nous proposons dans ce qui suit un nouvel algorithme distribué pour la récupération d'espace dans un environnement parallèle.

[29]Il existe beaucoup d'autres algorithmes, basés sur les techniques traditionnelles: les algorithmes de copie et ceux dits générationnels [NETTLES93] [LANGENDOEN92] [LeSERGENT92] [LADIN92] [LESTER??] [MANCINI88] [KUNG77]...

3.3.10. Un algorithme distribué pour la récupération d'acteurs

Nous considérons à ce niveau l'application des algorithmes de ramasse-miettes dans le cas de R.C.A.I. Dans le cadre de nos travaux, il s'agit de récupérer des acteurs qui n'entrent plus dans le calcul d'une fonction en cours d'évaluation. Rappelons que dans cette première version du projet R.C.A.I à gestion dynamique, les notions d'acteurs et de cellules sont confondues puisqu'il y a précisément un acteur par cellule. Récupérer un acteur inutilisé revient donc à récupérer la cellule qui le contient.

Les algorithmes de récupération, basés sur la notion de marquage ("*mark & sweep*", "*mark & compact*", ...), posent, dans un milieu distribué, un sérieux problème. En effet, en supposant qu'une opération de récupération ait été initiée par un quelconque élément du système, le problème qui survient est celui de la détection de la terminaison de cette opération. A quel moment peut-on être sûr que tous les éléments racines, pour simplifier la chose, ont procédé au marquage de l'ensemble des éléments qui leur sont potentiellement accessibles? Cette question soulève également celle de la connaissance mutuelle des éléments racines. Chacun d'eux doit être mis au courant de la création et de la destruction de ces éléments particuliers. Ceci représente donc, a priori, le seul élément de contrôle possible dans un milieu distribué.

Pour résoudre ce problème, nous proposons la solution décrite dans la suite de ce paragraphe.

Définition:

Un acteur est dit "racine" lorsqu'il est apte à rendre compte à l'hôte de son état ou de celui d'autres acteurs.

Nous supposerons que cette caractéristique est détectée au moment de la compilation du source, en relevant des références à des entités externes (références externes), tel que *console*, *fichier*... Tout acteur de ce type sera marqué comme étant un *acteur-racine*. De même, lors de l'exécution, tout acteur à qui un autre acteur communique une référence externe est marqué à son tour *acteur-racine*.

Pour réaliser le contrôle de terminaison, dont nous avons donné une ébauche plus haut, nous avons besoin d'un autre type d'acteur: l' *acteur contrôleur*. L'acteur contrôleur est en quelque sorte un super *acteur-racine*. Il garde trace des premiers *acteurs-racines* et décide du moment de procéder à la récupération des cellules inutilisées. Cet acteur est la racine de l'arbre constitué par les *acteurs-racines* (figure3.36).

La création et la destruction d'acteurs-racines entraîne une modification de l'arborescence formée par les acteurs de ce type (figure3.37). Ces modifications correspondent à des mises à jour de pointeurs entre les *acteurs-racines*.

Une création d'un acteur-racine correspond à une duplication d'un autre acteur de ce type. C'est donc cet acteur qui garde trace de l'existence de ce dernier.

Lorsqu'un acteur-racine est détruit, il doit passer à l'acteur-racine père l'identité du (des) prochain(s) acteur(s)-racine(s) sur lequel (lesquels) il pointe.

Lorsqu'un acteur, à la recherche d'une cellule libre, s'aperçoit qu'aucune cellule n'est disponible, il diffuse une demande de récupération de cellules.
L'acteur contrôleur qui reçoit cet ordre va déclencher en parallèle l'opération de ramasse-miettes. Il envoie à tous ses nœuds fils un ordre de récupération, qui à leur tour renvoient cet ordre à leur propre descendance. Un acteur-racine qui reçoit un ordre de récupération envoie un signal à tous les acteurs qu'il peut atteindre, directement ou indirectement, l'ordre de se marquer comme "valide". Les acteurs racines se marquent "valides" dès la réception de l'ordre d'initiation de cette opération. Nous réalisons ainsi une **fermeture transitive** à partir des nœuds racines. Lorsque chaque élément des sous-arbres constitués des acteurs-racines aura répondu avoir contacté et reçu un accusé de réception de la part des acteurs qu'il connaît, l'acteur-racine de cette sous-arborescence renvoie un accusé de réception à son père.

De ce fait, l'opération prend fin lorsque les acteurs racines, directement reliés à l'acteur-contrôleur, auront tous renvoyé un accusé. A la réception de tous ces accusés, l'acteur-contrôleur diffuse l'ordre, à toutes les cellules non marquées, de se déclarer libres.

Durant cette opération, toutes les demandes d'une autre opération de récupération sont ignorées. De même, à la diffusion de la première requête de

récupération, toutes les cellules inhibent d'éventuelles demandes de cette nature, jusqu'à réception de l'ordre de libération.

Il faut remarquer que, contrairement aux algorithmes de marquage cités dans les paragraphes précédents, cet algorithme ne nécessite pas l'arrêt de l'application pour réaliser cette opération. Les acteurs réagissent aux événements qui leurs sont envoyés en s'auto-détruisant ou tout simplement en poursuivant leur traitement. Il n'y a que la demande d'autres opérations de récupération qui sont inhibées, afin de ne pas saturer le réseau inutilement.

Cet algorithme est repris dans la figure3.38.

Avant d'implémenter cette solution et d'en mesurer l'efficacité, nous nous sommes attachés à régler un certain nombre d'autres problèmes d'implémentation; c'est pourquoi des résultats d'évaluation précis n'ont pu être obtenus sur cet algorithme. La solution retenue dans R.C.A.I est beaucoup plus simple comme nous le verrons par la suite[30].

Avant d'aborder l'exposé de la solution préconisée dans R.C.A.I, nous devons définir quelques éléments. Ces éclaircissements vont apparaître au début du prochain chapitre. Ils seront suivis par notre proposition pour la récupération d'acteurs dans une machine massivement parallèle à grain fin, R.C.A.I.

[30]La solution que nous présenterons pour R.C.A.I repose sur la notion de compteur de références. Cette solution devra être combinée avec celle exposée à ce niveau pour obtenir un meilleur rendement. Ainsi, lorsqu'un acteur se trouve dans l'incapacité de découvrir une cellule libre, alors qu'il en existe potentiellement certaines occupées par des acteurs inactifs, il s'agit à ce moment là de cellules qui n'ont pas pu être récupérées par le biais de leur compteur - car elles forment un cycle. Toutefois, cette solution ne pourra être effective que si ce genre de cycle sont permis par le langage d'acteur retenu.

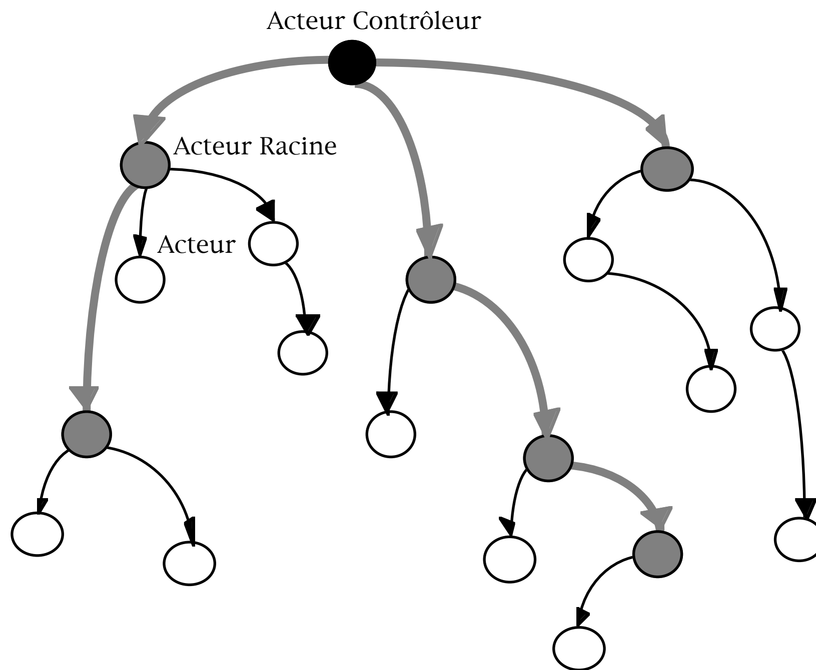


Fig.3.36. Structure des acteurs d'une application.

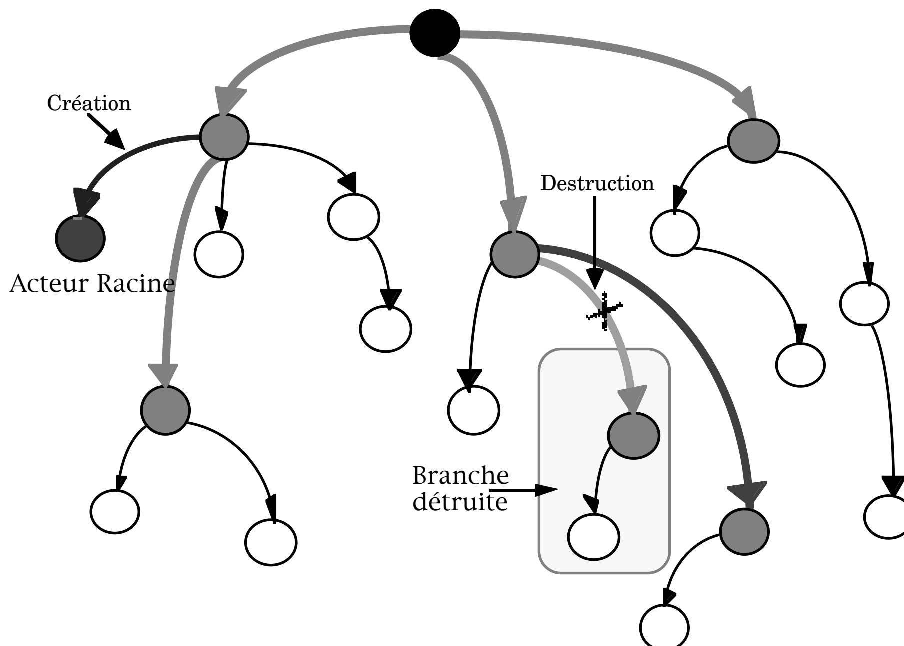


Fig.3.37. Mise à jour de l'arborescence de contrôle de la récupération parallèle d'acteurs.

```
/* Pour Toutes Les Cellules */
A l'échec d'une recherche de cellule libre
 { Si ( Diffusion_Requête == AUTORISÉE )
    Alors • Diffuser la demande de récupération
   FinSi
 }

A la réception d'une demande de récupération
 { Marque_Cellule  <- NON_VALIDE
   Diffusion_Requête <- NON_AUTORISÉE
 }

A la réception d'un ordre de ramasse-miettes
 { Si ( Marque_Cellule == VALIDE )
    Alors Diffusion_Requête <- AUTORISÉE
    Sinon Etat_Cellule   <- LIBRE
   FinSi
 }

A la réception d'un ordre de récupération
 { Marque_Cellule <- VALIDE

   Pour_Tout ( Acteur )
   Faire • Envoyer ordre de récupération
   FinFaire

   Attendre_Tout (Acteur )

   • Envoyer accusé à l'acteur père
 }

/* Pour Toute Cellule Acteur-Racine */
A la réception d'un ordre de récupération
 { Marque_Cellule <- VALIDE

   Pour_Tout ( Acteur_Racine & Acteur )
   Faire • Envoyer ordre de récupération
   FinFaire

   Attendre_Tout (Acteur_Racine & Acteur )

   • Envoyer accusé à l'acteur-racine père
 }

/* Pour La Cellule Acteur-Contrôleur */
A la réception d'une demande de récupération
 { Marque_Cellule <- VALIDE

   Si ( Récupération_En_Cours == VRAI )
    Alors • Ignorer la requête
    Sinon
       Pour_Tout ( Acteur-Racine )
       Faire • Envoyer ordre de récupération
       FinFaire

       Attendre_Tout ( Acteur-Racine )

       • Diffuser l'ordre de rammasse-miettes

       Récupération_En_Cours <- FAUX
   FinSi
 }
```

Fig.3.38. Algorithme du ramasse-miettes parallèle.

Chapitre 4
Autres problèmes liés aux limitations de R.C.A.I.

4. AUTRES PROBLÈMES LIÉS AUX LIMITATIONS DE R.C.A.I

Quelques-uns des problèmes de mise en œuvre sont intimement liés aux limitations de notre machine. Il s'agit notamment de l'espace d'adressage qui est relativement réduit dans R.C.A.I. Ces problèmes mettent en exergue l'algorithmique parallèle en relation avec notre architecture et sa spécificité. Il faut être toujours à la recherche de solutions simples (qui peuvent être intégrées facilement) et indépendantes de la taille du réseau (à cause de l'espace d'adressage) et de la taille du grain visé (très fin, en relation avec la mémoire de faible taille).

4.1. Serveur de code

Durant l'exécution d'un acteur, il y a création et destruction d'acteurs intermédiaires, pour l'évaluation d'une fonction commune. L'opération de création se déroule en plusieurs phases:

- recherche d'une cellule libre,
- copie du code correspondant,
- mise à jour des espaces de références des acteurs concernés par cette création,
- initialisation de l'acteur qui vient d'être créé,
- déclenchement à distance de l'exécution de l'acteur.

La recherche de cellules libres a été abordée dans le chapitre précédent. A présent, nous nous intéressons à un problème particulier: la désignation du code à copier. Trois cas sont possibles:

- la copie du code se trouve à l'extérieur du réseau; dans ce cas, à chaque création d'un acteur nous faisons appel à l'hôte qui copie le code correspondant dans la cellule spécifiée. Il est évident que cette solution est très coûteuse à cause du nombre d'E/S à effectuer pour mener à terme l'application.
- La copie du code est réalisée à partir d'une cellule particulière; chaque acteur différent est affecté à une cellule du réseau. Une telle démarche pose l'épineux problème du goulot d'étranglement que constitue cette cellule.

Une autre approche consiste à distribuer plusieurs copies d'un même acteur, répartissant ainsi la charge sur ces cellules. C'est cette solution que nous développons dans cette section.

4.1.1. Définition de l'espace d'adressage

Comme nous l'avons déjà vu, lors de la définition du modèle d'exécution, nous dégageons à la compilation le graphe non orienté d'acteurs de l'application (figure4.1).

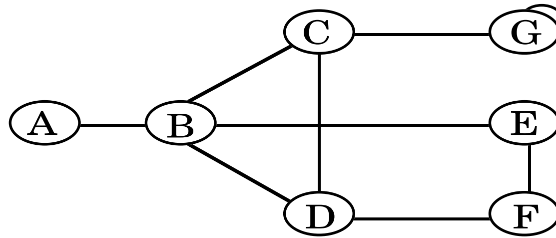


Fig.4.1. Réseau d'acteurs.

Pour chaque acteur nous dressons la table suivante qui définit entre autre son espace d'adressage que nous notons Ω. Chaque entrée de cette table correspond à une référence d'un acteur de ce type dans le système.

	Acteur i	
Acteur i.1	Ref. 1	Ensemble Ω
Acteur i.2	Ref. 2	
	...	
Acteur i.j	Ref. j	
Δ	⊥	Acteur délégué
∑	n	Compteur des références

Fig.4.2. Définition de l'espace d'adressage d'un acteur.

En considérant l'exemple de la figure4.1, la construction de toutes les tables de références des différents acteurs donne le résultat de la figure4.3.

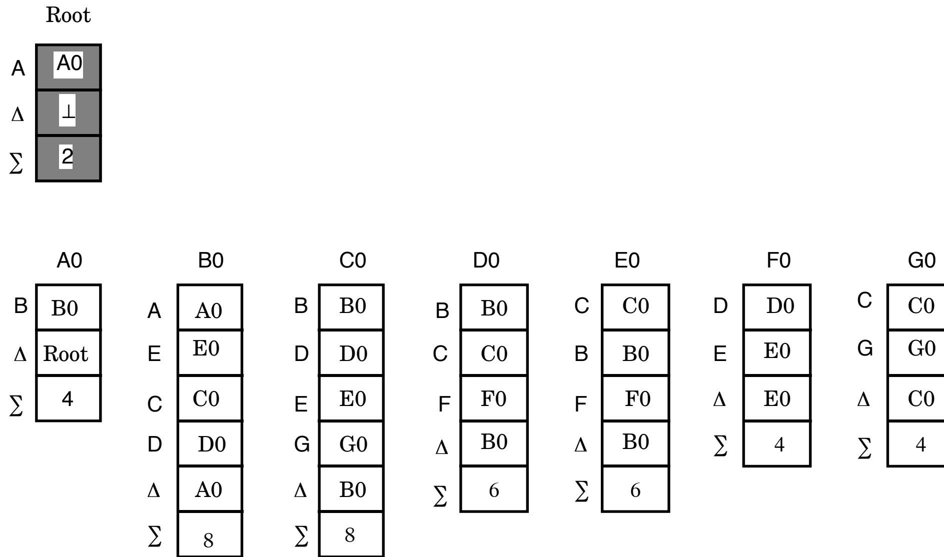


Fig.4.3. Exemple d'espaces d'adressage d'acteurs d'un graphe.

Eu égard à ces tables de références, chaque acteur qui désire créer un nouvel acteur consulte sa propre table pour dégager la dernière référence (ou *prototype*) connue de cet acteur dans l'ensemble Ω.

Un nouvel acteur signale à chacun des éléments de l'ensemble Ω sa création de telle sorte que chacun puisse mettre à jour son propre compteur de références.

L'acteur à l'origine de cette création informe l'acteur qui correspond à la dernière référence connue de ce prototype (qui figure dans son ensemble Ω) de sa décision de ne plus garder de lien avec lui pour une éventuelle demande de copie. Une fois qu'il l'aura averti, cet acteur remplace l'ancienne référence dans son ensemble Ω par la nouvelle.

L'espace d'adressage (ensemble des références avec lesquelles une entité est en mesure de communiquer) se présente sous la forme reportée dans la figure4.4.

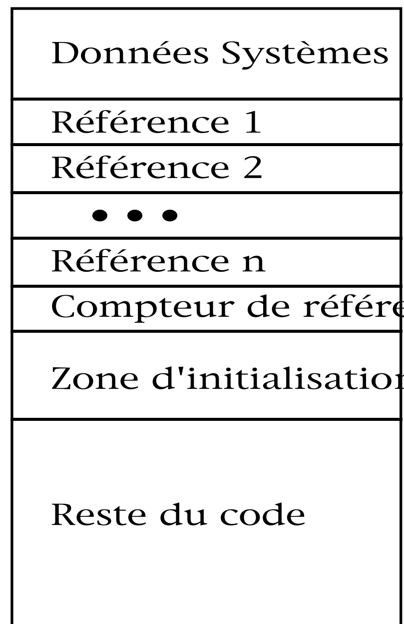


Fig.4.4. Représentation de l'espace d'adressage.

4.1.2. Différents types de copies de code

Comme nous l'avons évoqué précédemment, la création d'un acteur débute par la recherche d'une cellule libre. Une fois cette cellule trouvée, il faut procéder à la copie du code correspondant. Deux cas de figure apparaissent (figure4.5):

- soit il s'agit d'une duplication de l'entité elle-même,
- soit il s'agit d'une copie à partir d'une autre cellule.

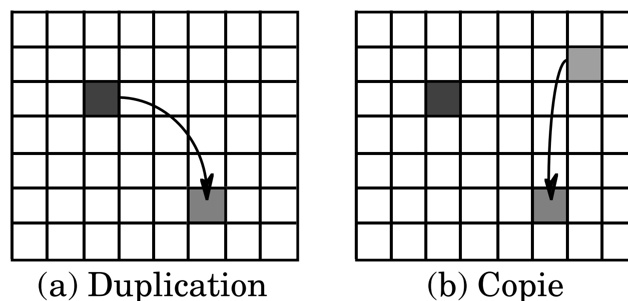


Fig.4.5. Serveur de code.

La différence entre les deux types de copies est évidente. L'une est déclenchée localement, alors que l'autre provoque, à distance, l'amorce de l'opération de copie (qui ne correspond pas à la progression normale de l'acteur impliqué dans cette action).

4.1.2.1. Duplication

Cette opération est provoquée par l'intermédiaire de l'instruction MOVE. La syntaxe correspondante est la suivante:

MOVE <adresse>

Cette instruction procède de la manière suivante. Le compteur associé à cette instruction est initialisé à la taille du code à transférer. Cette donnée est indiquée dans la zone système (figure4.4). Les mots mémoires sont acheminés l'un à la suite de l'autre à leur destination (indiquée dans le contenu du mot mémoire *<adresse>* de l'instruction), jusqu'au passage à zéro du compteur. Ce schéma est explicité dans la figure4.6.

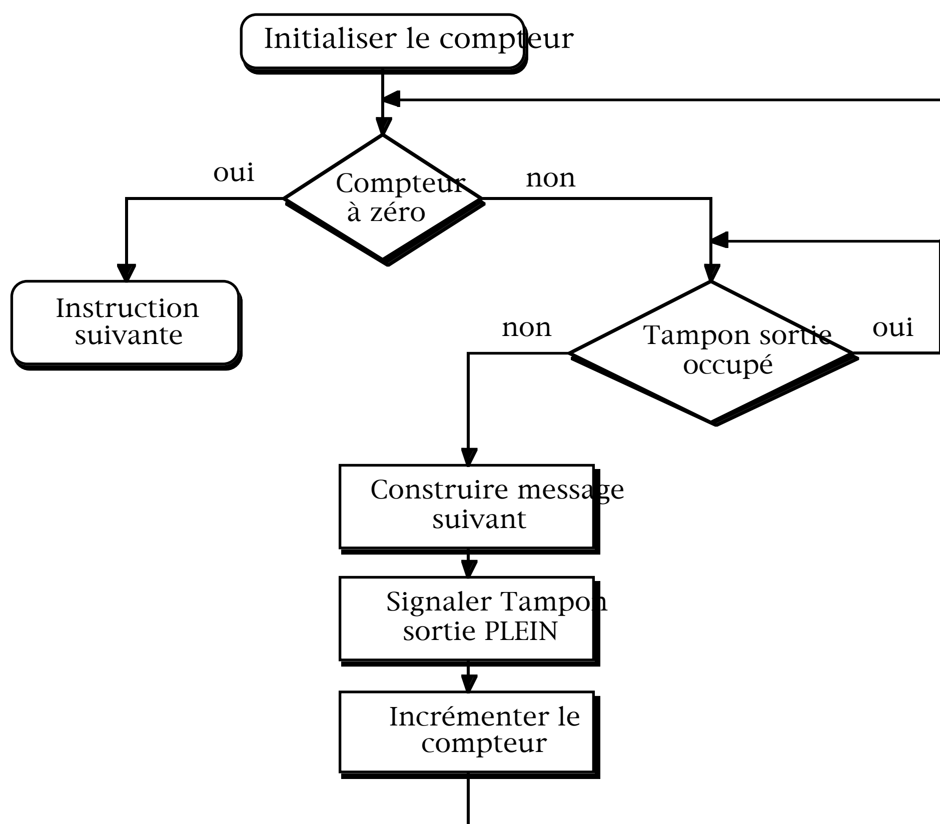


Fig.4.6. Duplication de code.

Cette instruction est bloquante. En effet, tant que le code n'aura pas été transféré en entier, nous ne passons pas à l'instruction suivante.

4.1.2.2. Copie distante

Cette opération est provoquée par l'intermédiaire de l'instruction COPY. La syntaxe correspondante est la suivante:

COPY <adresse>

Contrairement à la première instruction, cette dernière ne fait pas partie du code de la cellule où elle est interprétée (exécutée). La cellule qui accomplit cette opération reçoit un ordre (message) pour une copie différentielle. Ce message provoque une interruption au niveau de la cellule. A partir de ce moment, le schéma d'exécution est, à peu de choses près, semblable au précédent: nous commençons par initialiser un compteur, puis les mots mémoires sont transférés successivement à leur destination. Toutefois, cette instruction n'est pas bloquante pour la cellule mise à contribution dans cette action. En effet, il y a partage de la ressource que constitue le tampon de sortie entre la structure gérant la copie de code et le routeur de la cellule.

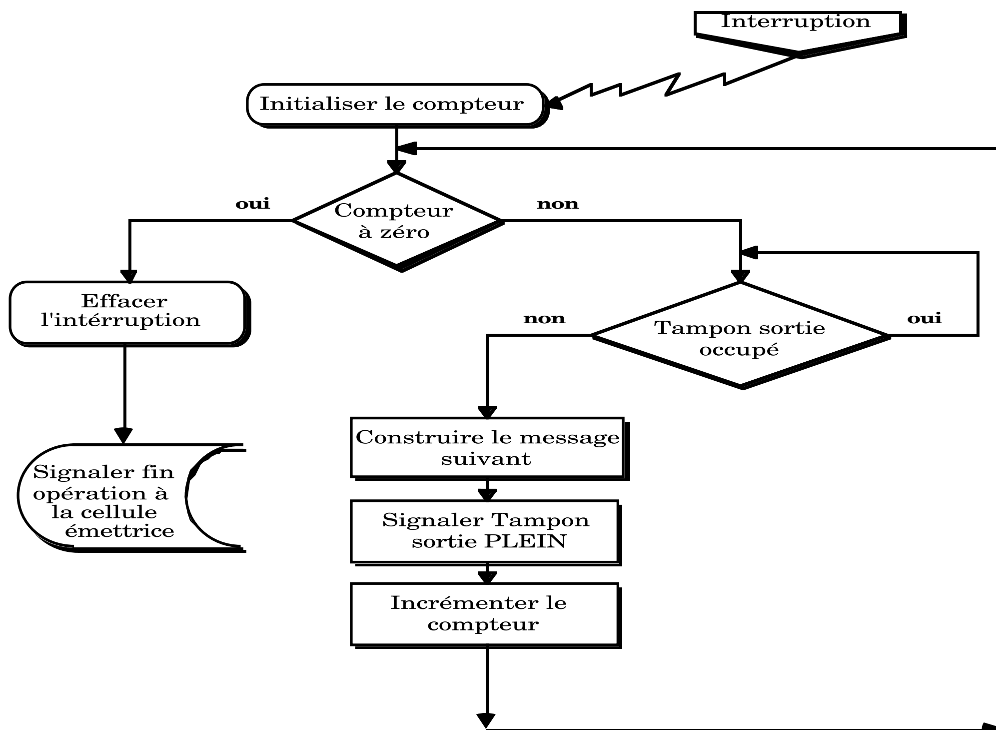


Fig.4.7. Copie de code à distance.

4.1.2.3. Dernière phase de la copie de code

Aussi bien dans l'une que dans l'autre instruction, l'opération de copie est terminée par l'envoi d'un message particulier: le message de début d'exécution sur le processeur (cellule) cible. Ce message est en fait interprété comme un signal, puisqu'il se contente de demander au processeur en question de se brancher à son premier point d'entrée. L'adresse de ce point d'entrée spécial est intégrée dans les données système de chaque acteur.

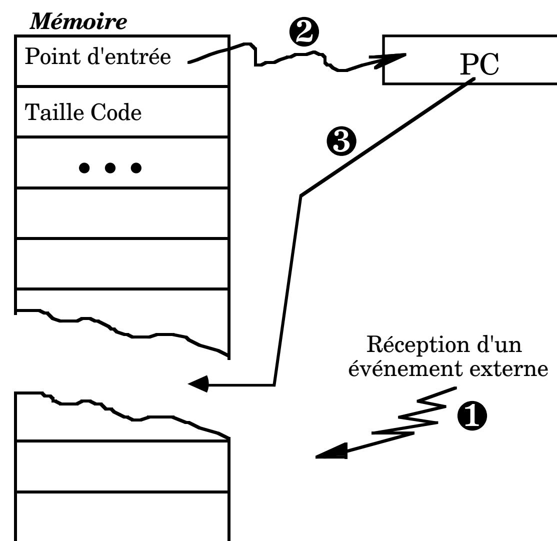


Fig.4.8. Déclenchement d'exécution à distance.

La section de code qui correspond à cette adresse est la phase d'initialisation. En effet, il faut pouvoir assurer pour chaque acteur un environnement cohérent pour qu'il puisse entamer correctement son exécution. La réaction ultérieure aux messages qu'il recevra en sera profondément influencée. La valeur des différentes variables locales détermine l'action à entreprendre par l'acteur vis à vis des requêtes des autres acteurs.

4.1.3. Exemple de gestion d'un espace d'adressage

L'entité nouvellement créée hérite de tout l'espace de références de l'entité génératrice. Cette dernière met à jour son propre espace par celles récemment créées. Elle informe ensuite celles précédemment référencées à cet usage que cette dernière ne garde plus trace de leur existence pour des copies ultérieures. De cette façon, nous évitons d'introduire des goulots d'étranglement pour la désignation des "modèles" (ou prototypes) des entités à générer.

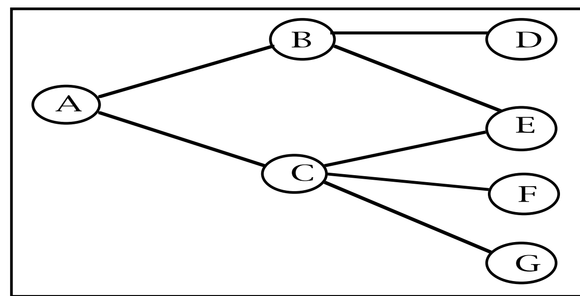


Fig.4.9. Exemple d'un graphe d'acteurs extrait d'une application.

C'est à partir d'un graphe d'acteurs que l'espace d'adressage est extrait pour chacun d'eux. Par exemple, pour l'acteur C, l'espace de références est le suivant:

$$\Omega_C = \{E,F,G,A\}$$

Supposons la création, au niveau de l'acteur C, d'une nouvelle entité (E') d'un acteur dont le prototype connu actuellement par celui-ci est désigné par E. Ω_C devient alors:

$$\Omega_C = \{E^{©},F,G,A\}$$

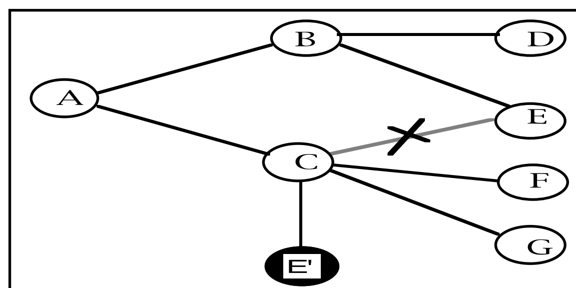


Fig.4.10. Le nouveau graphe d'acteurs.

La prochaine création d'un acteur du type E au niveau de ce même acteur se fera à partir de E'. Pendant ce temps, au niveau de l'acteur B qui n'a pas encore procédé à une création de ce type, celle-ci se fera toujours à partir de la référence E qu'il a de ce même acteur.

4.2. Ramasse-miettes dans R.C.A.I

A la manière du langage HAL [KIM92] [HOUCK92], qui dispose d'une instruction spéciale ***suicide***, nous avons également introduit une instruction spéciale de restitution de l'espace inutilisé. Il s'agit de l'instruction **END**. Un acteur qui souhaite libérer la cellule qu'il occupe le réalise par le biais de cette instruction.

La finesse du grain de R.C.A.I. nous interdit le recours à des solutions complexes. C'est pour cette raison que nous avons élaboré une technique basée sur la notion de compteur de références. Cette stratégie est connue pour être simple à implémenter et à gérer d'une manière distribuée (cf.§3.3.2). C'est cette solution que nous développons dans ce qui suit.

Initialement, les liens entre les acteurs sont bidirectionnels. C'est à dire qu'ils expriment la notion de "*connaître un acteur*" et "*être connu d'un acteur*". C'est la raison pour laquelle les compteurs de références sont doublés à l'origine.

A la création d'un nouvel acteur, son compteur de références est automatiquement positionné au cardinal de son ensemble de références Ω dont il a hérité. Ce compteur est augmenté de un qui correspond au fait qu'il est connu par l'acteur qui l'a créé. De même, l'acteur qui procède à cette création engendre un lien vers cet acteur, c'est à dire qu'il incrémente de un son propre compteur de références. Le nouvel acteur communique ensuite à chacun des éléments de son ensemble Ω sa propre création.

Lorsqu'un un acteur reçoit l'annonce de la création d'un autre acteur, il incrémente de un son propre compteur de références, puisqu'il est référencé par ce dernier.

A la rencontre de l'instruction **END**, l'acteur informe l'ensemble des éléments de son espace de références de son souhait de quitter le système.

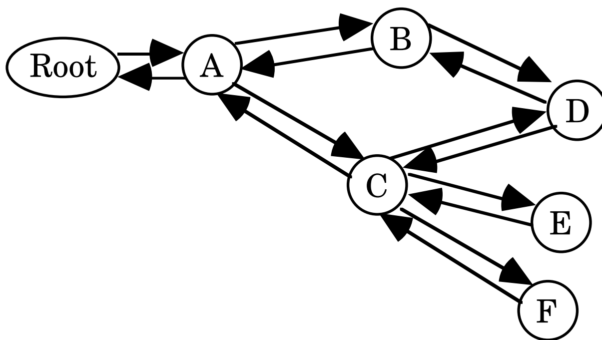
Un acteur qui reçoit une annulation pour un lien décrémente de un son compteur de références (puisqu'il n'est plus référencé par ce dernier). Au

même moment, l'acteur qui envoie cette annulation décrémente de un son compteur de références.

L'acteur se met ensuite en attente de l'annulation de tous les acteurs qui le référencent (mais qui ne font pas partie de son ensemble Ω).

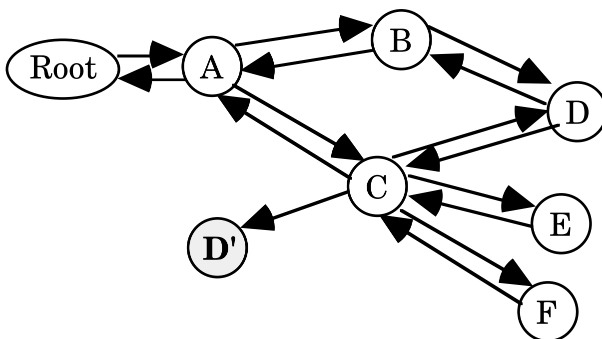
Ainsi, un acteur n'est autorisé à abandonner le système que si son compteur de références est à zéro.

Pour illustrer ce mécanisme, nous allons considérer un exemple basé sur le graphe d'acteurs suivant avec les ensembles de références de chaque acteur (Ω) et le compteur de références associé (∑):



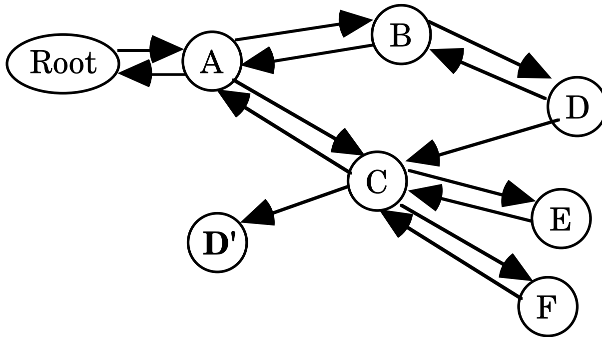
	Ω	Ref.	∑
A	{B,C,Root}		6
B	{A,D}		4
C	{A,D,E,F}		8
D	{B,C}		4
E	{C}		2
F	{C}		2

Supposons la création, au niveau de l'acteur C, d'un nouvel acteur D' à partir du prototype D. Il y a donc création d'un nouveau lien bidirectionnel (entre les acteurs C et D').



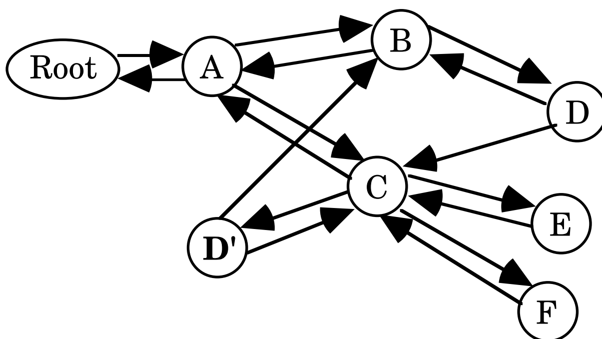
	Ω	Ref.	∑
A	{B,C,Root}		6
B	{A,D}		4
C	{A,D,E,F}	D'	**9**
D	{B,C}		4
E	{C}		2
F	{C}		2
D'			

L'acteur C informe l'acteur D de la rupture de leur liaison et remplace cette référence par la nouvelle vers D'.



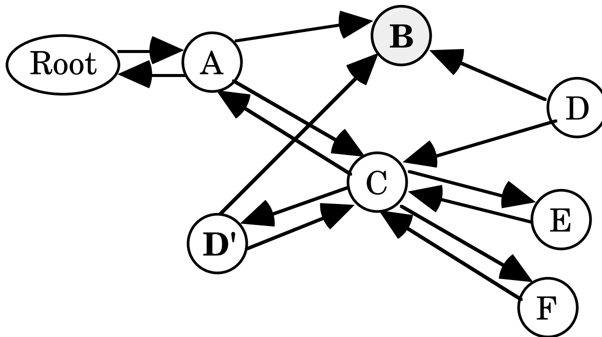
	Ω	Ref.	Σ
A	{B,C,Root}		6
B	{A,D}		4
C	{A,D,E,F}	D'	**8**
D	{B,C}		**3**
E	{C}		2
F	{C}		2
D'			

Pendant ce temps, l'acteur D' a hérité de l'espace de référence de D. D' signale ensuite à tous les éléments de Ω sa création. Ce qui donne comme nouvelle représentation:



	Ω	Ref.	Σ
A	{B,C,Root}		6
B	{A,D}		**5**
C	{A,D,E,F}		**9**
D	{B,C}		3
E	{C}		2
F	{C}		2
D'	**{B,C}**		**2+1**

Supposons maintenant que l’acteur B a atteint la fin de son exécution (l’instruction END). Il signale donc aux éléments de son ensemble Ω sa fin et met à jour son compteur de références. Nous obtenons alors:



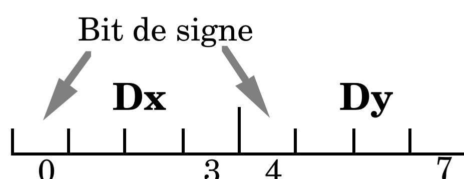
	Ω	Ref.	Σ
A	{B,C,Root}		**5**
B			**3**
C	{A,D,E,F}		9
D	{B,C}		**2**
E	{C}		2
F	{C}		2
D'	{B,C}		3

L’acteur B se met par la suite en attente de l’autorisation de quitter le système des acteurs A, D et D’ (pour que son compteur de références passe à zéro).

4.3. Limitation de l'espace d'adressage de R.C.A.I

4.3.1. Limites physiques de la machine R.C.A.I

Tous les problèmes qui ont été soulevés jusqu'à présent (recherche de cellules libres, gestion des serveurs de code,...) sont accentués par la réduction (physique) de l'espace d'adressage de la machine R.C.A.I. Rappelons que toute référence à une cellule distante s'écrit d'après le format suivant:



Le déplacement, dans l'une ou l'autre direction (abscisse ou ordonnée), reste toujours de longueur bornée (parce que le nombre de bits est limité). Dans R.C.A.I la borne est petite (±7 unités) et le problème se pose avec beaucoup d'acuité. Il faut donc trouver un moyen pour augmenter la zone de recherche pour pouvoir offrir un plus grand espace de recherche pour chaque cellule, et accroître de ce fait la probabilité de trouver une cellule libre (s'il y en a au moins une).

Dans le cas d'un réseau statique, un tel adressage est efficace même si le nombre de bits utilisé semble réduit [PAYAN91] [RUBINI92]!

4.3.2. Notion de relais

Pour élargir l'espace d'adressage dans le réseau R.C.A.I plusieurs stratégies peuvent être explorées:

- la première étant celle reposant sur un système de routage performant en considérant un routage du type *whormhole* où une adresse se présente sur plusieurs mots successifs (comme c'est le cas dans [FLAIG87]),
- l'autre solution est de considérer qu'il existe une entité, physique ou logique, que nous appellerons par la suite ***relais***, dont le rôle est de faire suivre un message entre deux cellules (ou plus) distantes.

Dans un premier temps, nous avons considéré la première solution qui consiste à offrir un mécanisme d'adressage dynamique du type whormhole, indépendant de la taille du réseau.

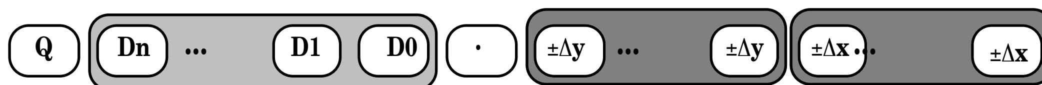


Fig.4.11. Utilisation d'une adresse de taille variable.

Dans cette approche, l'acheminement de messages se fait pas à pas, suivant une direction, puis l'autre: suite de déplacements (dx_1, dx_2, ... dx_n) puis (dy_1, dy_2, ... dy_m). Chacune des entités de déplacement dx_i ou dy_j est de longueur fixe *k*.

La valeur de cette taille *k* est assez délicate à déterminer. A première vue, un message arrivant à une cellule a deux possibilités:

- continuer son chemin par l'une des trois sorties restantes,
- ou ne plus progresser si la cellule en question est la cellule destinataire du message.

Quatre cas au total: un codage sur 2 bits est donc suffisant. Néanmoins, en supposant une adresse sur n bits, le nombre de paquets du message peut atteindre $2^{\frac{n}{2}-2}$ paquets pour la description de la seule adresse (comme nous allons l'expliquer dans ce qui suit).

D'une façon générale, considérons un codage sur k bits par paquet qui spécifie une portion de l'adresse, et une adresse à coder sur n bits (n>k).

Rappelons qu'une adresse dans notre contexte est définie par un couple (d_x,d_y). Chacune des deux parties est munie d'un bit de signe. De plus, en considérant le cas le plus simple -sans optimisation du codage-, un paquet de k bits nécessite, en plus du déplacement, un bit de signe et un bit de direction (abscisse ou ordonnée).

Pour dégager le nombre maximal de paquets, il faut considérer le rapport suivant (rapport de quantité d'information utile):

$$\frac{Valeur_maximale_de_l'adresse}{Valeur_maximale_représentable_sur_un_paquet}$$

Ce qui donne

$$Dx_{max} = Dy_{max} = 2^{\frac{n}{2}-1} \Rightarrow @_{max} = 2 * 2^{\frac{n}{2}-1} = 2^{\frac{n}{2}}$$

La taille maximale du paquet à laquelle nous retranchons les bits de signe et de direction est donnée par:

$$paquet_{max} = 2^{k-2}$$

D'où le rapport final:

$$\frac{2^{\frac{n}{2}}}{2^{k-2}} = 2^{\frac{n}{2}-k+2} \quad ❑ \qquad \textbf{(1)}$$

Pour garantir qu'il n'y a pas de perte d'information lors du codage (l'adresse à représenter doit être de taille supérieure à celle sur laquelle elle devra être représentée), nous devons satisfaire la règle suivante:

$$\frac{n}{2} - k + 2 > 0$$

d'où

$$n > 2(k-2) \quad ❑ \qquad \textbf{(2)}$$

Lorsque $n \gg k$ nous aurons une adresse de taille relativement grande d'après la relation **(1)**.

De plus, un tel mécanisme nécessiterait d'inclure des drapeaux pour différencier aussi bien la tête, que le corps et la queue d'un message; chacune des parties correspond respectivement à l'adresse (et éventuellement à la taille du message), à l'information à transmettre et à la fin du message (dans le cas où la taille du message n'est pas fournie au préalable).

La taille du message est variable, d'où une augmentation de la complexité du routeur. Cette solution pose donc le problème de la borne supérieure sur la longueur totale du message.

La seconde solution consiste à augmenter l'espace d'adressage en conservant une longueur fixe des messages. On introduit (statiquement ou dynamiquement) des entités pour réaliser une continuité dans l'adressage entre deux cellules éloignées (figure4.12). L'avantage de cette méthode est de pouvoir garder le système de routage actuel dont la particularité principale est sa simplicité et ses performances [RUBINI92].

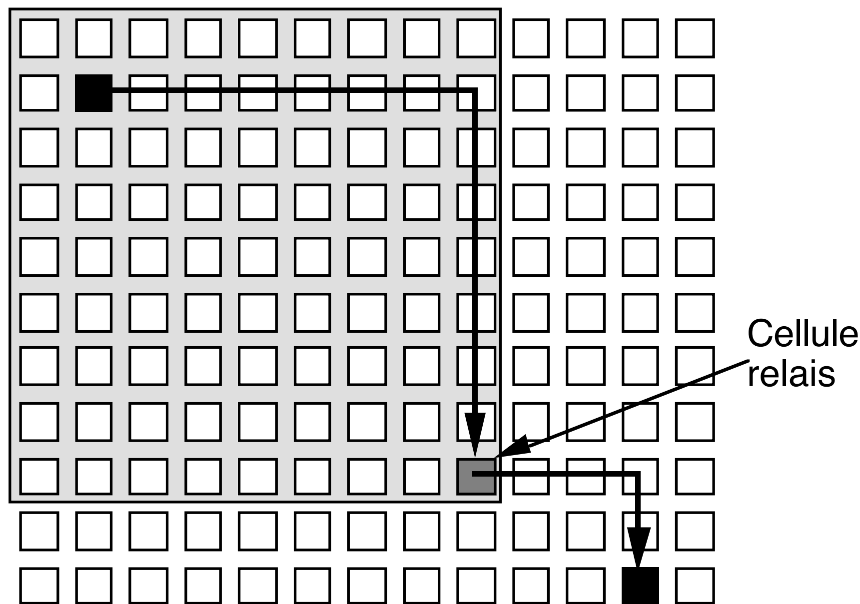


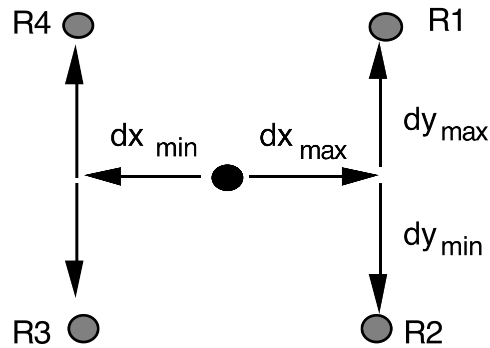
Fig. 4.12. Notion de relais.

4.3.3. Introduction de relais pour l'extension de l'adressage

L'introduction de relais est effectuée lorsque les conditions suivantes sont satisfaites:

$$d_x = \begin{cases} d_x^{max} \\ d_x^{min} \end{cases} \quad \& \quad d_y = \begin{cases} d_y^{max} \\ d_y^{min} \end{cases}$$

Ces conditions signifient que nous sommes arrivé à la limite de l'adressage de la cellule. Ce qui, schématiquement, peut être reproduit par la figure ci-après:



Signalons dès à présent qu'il existe deux types de relais:

- les relais introduits lors de la recherche d'une cellule libre,
- les relais dus à la communication de références d'une cellule à une autre ("*changement de coordonnées*").

Dans le premier cas, lorsque nous arrivons aux frontières d'adressage d'une cellule, l'opération est reprise par une des quatre cellules formant cette limite, à la demande de la cellule émettrice. De cette manière, nous pouvons, par programmation des relais, étendre l'espace d'adressage d'une cellule à l'ensemble du réseau.

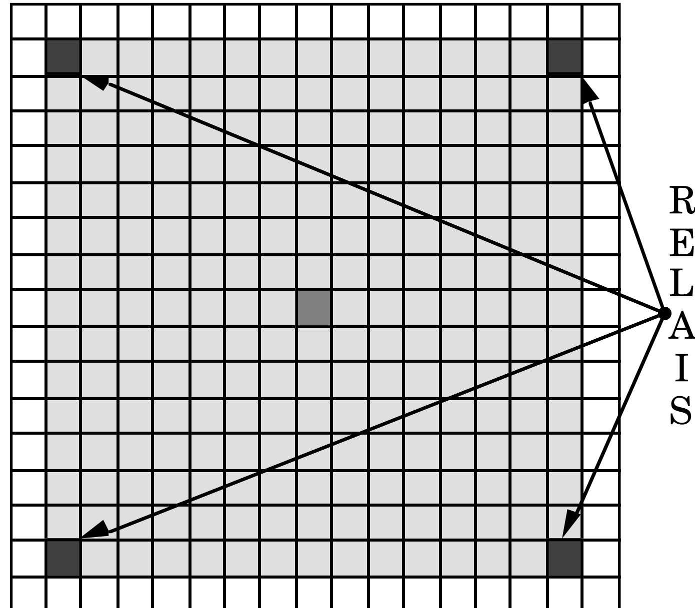


Fig. 4.13. Position des relais par rapport à une cellule.

Comme nous l'avons spécifié auparavant, un relais est une entité logique ou physique. Développons cette notion.

4.3.3.1. Cas d'un relais physique

Dans le cas d'un réseau statique, [PAYAN91] a proposé d'utiliser des cellules particulières du réseau appelées **relais**. Dans ce cas, on réserve une cellule à cet effet par pavé de 8x8 cellules. Ces cellules sont utilisées pour des adressages de longueur supérieure à 7 (ou inférieure à -7). Dans cette version statique de R.C.A.I, chaque bout de code est placé suivant la technique du recuit-simulé qui minimise la longueur entre les cellules les plus reliées. De ce fait, pour le peu de cellules qui n'ont pas pu être placées dans le voisinage immédiat des cellules auxquelles elles sont reliées, une zone tampon au niveau d'une cellule particulière, connue à l'avance, sera utilisée chaque fois que ceci est nécessaire. Le nombre de telles cellules tampon a été évalué à au plus2% du réseau entier [PAYAN91].

Une telle solution n'est cependant pas envisageable dans le cas d'un réseau dynamique. En effet, nous ne pouvons pas faire d'a priori sur l'évolution du système: celui-ci est complètement imprévisible (par hypothèse). Il faudrait donc dresser d'une façon active des relais (lors de la recherche de cellules libres ou lors d'un échange de références). Par conséquent, il n'est pas possible de réserver des cellules au préalable à cet usage: l'évolution du système pourrait ne jamais recourir aux relais. Ces cellules seraient alors tout simplement inexploitées, d'où une sous-utilisation du réseau.

Finalement, nous avons opté pour une solution intermédiaire qui, tout en gardant les avantages de la simplicité de la solution adoptée dans le cas du réseau statique, ne limite pas la taille de l'espace d'adressage.

Dans cette solution, chaque cellule est dotée d'une structure pour la retransmission des messages au niveau des cellules intermédiaires. Il s'agit d'une table “associative”: chaque entrée dans cette table contient l'adresse de la cellule vers laquelle devra être réacheminé le message correspondant.

Dans le cas le plus simple, chaque cellule ne peut être le relais que de quatre autres cellules au plus. Il suffit, pour s'en convaincre, de considérer la figure4.13 dans le sens inverse. En ne considérant qu'un seul niveau d'extension et qu'un seul relais autorisé par cellule, la table de correspondance est constituée de seulement quatre éléments.

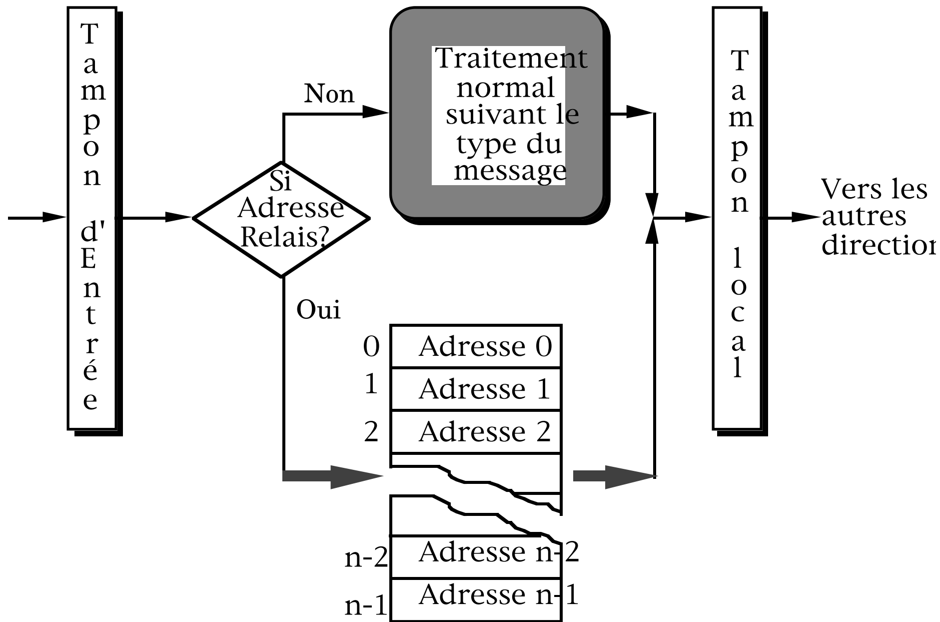


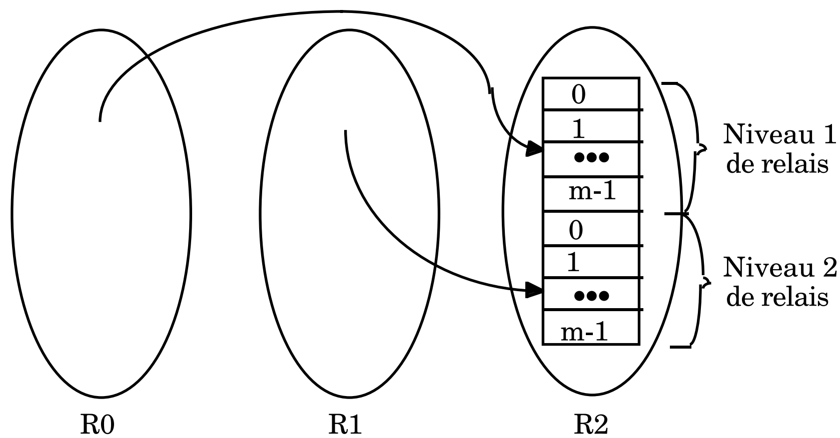
Fig. 4.14. Relais physique.

Par contre, nous pouvons obtenir une imbrication de plusieurs niveaux de relais: chacun étend l'espace d'adressage du relais précédent.

De ce fait, dans le cas général, cette table présente une taille de:

$$table_{size} = (4 * n * m) \text{ élémen}$$

n correspond au nombre de niveaux autorisés par cellule et **m** le nombre d'entrées allouées au niveau de ce relais pour une même cellule (figure suivante).



4.3.3.2. Cas d'un relais logique

Dans cette autre approche, c'est une entité logique qui prend en charge l'extension de l'espace d'adressage. Cette entité est un acteur particulier, appelé **acteur relais**.

Tout comme dans l'approche développée auparavant, lorsque les conditions de dépassement de la capacité d'adressage des cellules sont satisfaites, une interruption sera générée au niveau de la cellule en question. L'acteur relais prend alors la main afin de réserver une entrée pour cette demande, après approbation par la cellule source (figure4.15).

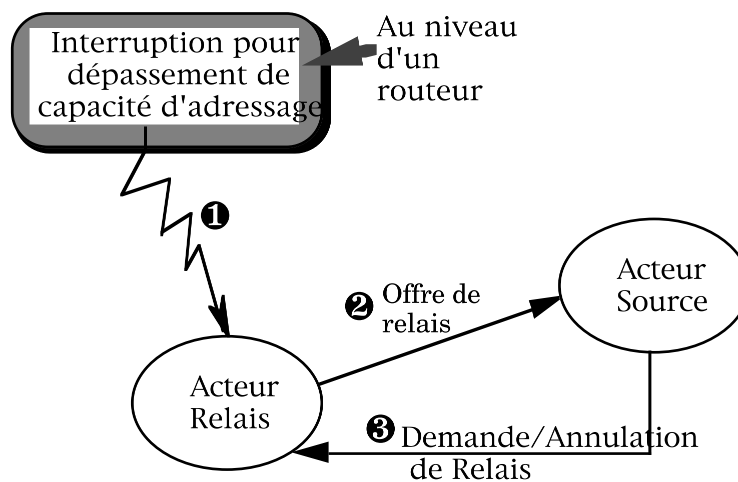


Fig. 4.15. Relais logique.

L'annulation d'un relais peut survenir lorsqu'une cellule, qui a demandé une cellule libre, a déjà reçu une réponse favorable à sa requête de la part d'une autre cellule.

4.3.3.3. Différence entre les deux types de relais

La différence entre les deux approches relève de l'éternel problème de la différence entre une fonction intégrée rapide, mais figée, et une fonction logicielle plus lente, mais flexible.

L'avantage des relais physiques est la rapidité avec laquelle seront converties les adresses relais en adresses physiques. De plus, ce mécanisme reste transparent vis à vis de l'application en cours d'exécution. Il n'y a pas d'interruption des processus en cours, comme c'est le cas dans la seconde solution, pour la traduction des adresses.

L'intérêt des relais logiques est leur extensibilité. D'un point de vue conceptuel, cette solution ne présente aucune limite (autre que la mémoire disponible dans le réseau). Comme nous l'avons exposé plus haut, chaque cellule ne peut être que le relais de quatre autres cellules. De ce fait, chaque relais logique de base peut être considéré comme présentant la structure suivante:

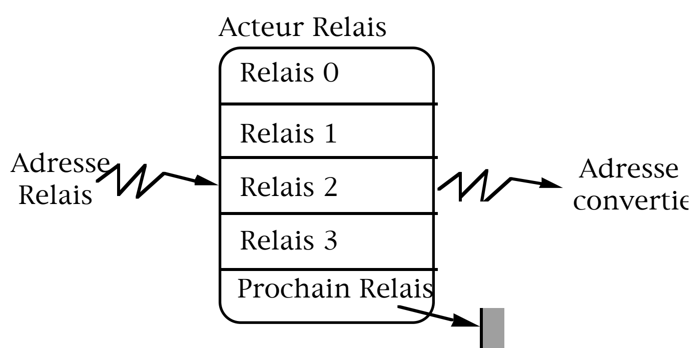


Fig. 4.16. Structure d'un relais logique multi-niveaux.

A chaque demande d'un niveau de relais supplémentaire, nous procédons à la création d'un nouvel acteur. De cette manière, il y a extension d'un niveau pour toutes les cellules concernées. Les cellules qui nécessitent par la suite un nouveau relais font toujours appel à la même cellule.

Trois cas de figure se présentent lorsqu'une cellule reçoit une demande de relais:

- la cellule dispose localement d'une entrée libre pour résoudre ce problème d'adressage,
- la cellule ne dispose pas d'une entrée, dans ces conditions la cellule crée un nouvel acteur relais qui prend en charge ce litige,
- finalement, la cellule ne dispose pas localement d'une entrée, mais peut déléguer ce conflit à un autre acteur relais (qu'elle aura créé dans une étape antérieure), de niveau supérieur qui devra prendre en compte ce conflit. Ce dernier cas est récursif et aboutit à l'un des deux points précédents.

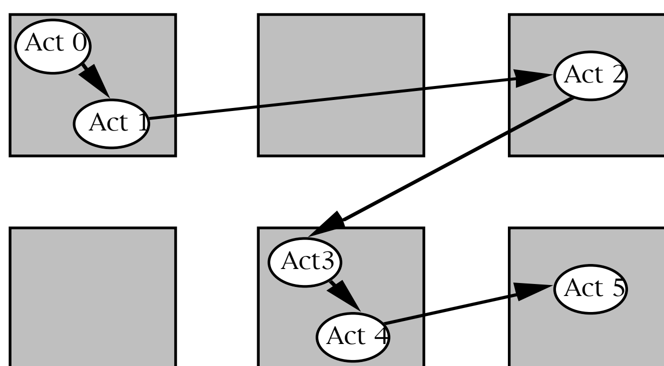


Fig. 4.17. Structure à six niveaux de relais logique pour une même cellule

Il est indéniable que l'approche par des relais logiques est la plus adaptée pour une programmation aisée au niveau le plus haut et une meilleure exploitation des ressources du système. Cette solution est très flexible et peut évoluer avec les spécifications de la machine actuelle.

Cependant, dans l'état actuel de la machine, la solution retenue est celle basée sur les relais physiques avec un seul niveau supplémentaire d'adressage. Cette limitation, comme nous l'avons déjà vu, offre, tout de même, un espace d'adressage relativement important pour chaque cellule. La solution faisant appel à des relais logiques ne peut être mise en œuvre que lorsqu'un langage d'acteur aura été défini pour notre machine (puisque faisant partie intégrante de ce modèle).

4.3.3.4. Relation entre relais et recherche de cellules libres

Les relais, surtout les relais logiques, ne présentent a priori aucune limite pour l'extension de l'espace d'adressage de chaque cellule à l'ensemble du réseau. Cependant, il n'est pas réaliste d'augmenter cet espace au delà d'un ou de deux niveaux. En effet, l'accroissement du nombre de relais par cellule augmenterait d'une manière drastique à chaque niveau supplémentaire.

La figure4.18 fournit le nombre de cellules adressables (au maximum) à partir d'une cellule en fonction du nombre de transitions possibles dans une direction (notée k dans la figure). Avec aucun relais, la valeur de k est 7 et le nombre de cellules potentiellement adressables est de 900 cellules. Pour un niveau, la valeur est de 14 et le nombre de cellules atteint la valeur de 3364 cellules. A partir du second niveau de relais on dépasse les 7000 cellules adressables.

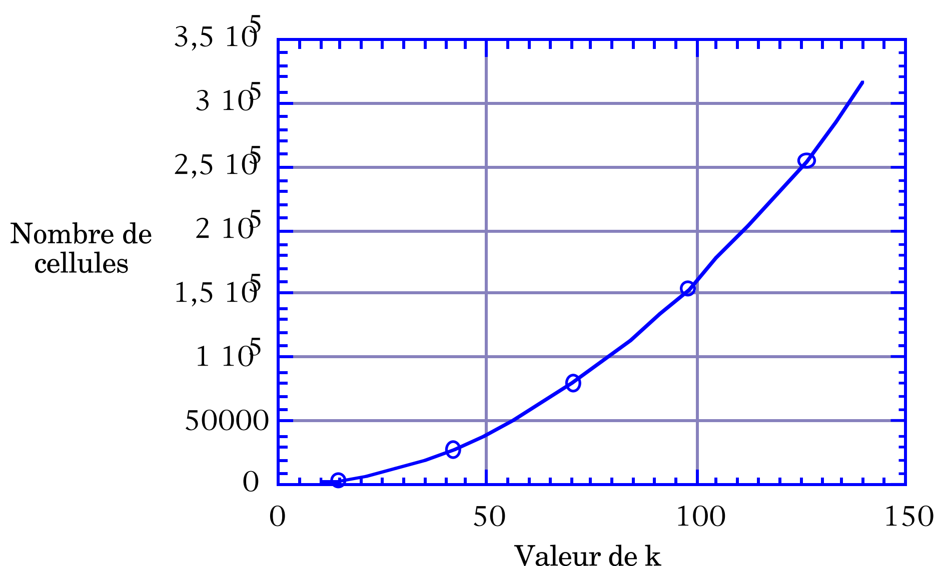


Fig. 4.18. Nombre de cellules adressables en fonction du nombre de niveaux de relais possibles.

Pour revenir à la procédure de recherche de cellules libres, il est essentiel de limiter l'explosion du nombre de messages de recherche dans le réseau[31].

[31]Nous avons déjà abordé ce sujet au paragraphe 3.2.1.2.1. Nous le complétons à ce niveau après avoir introduit la notion de relais.

Comme il vient d'être exposé, un relais est créé à la demande de la cellule émettrice. A chaque relais supplémentaire correspond un pavé (potentiel) additionnel de recherche de 254 cellules.

A la demande de la mise en place d'un relais, une cellule à la recherche d'une cellule libre doit savoir à quel moment autoriser cette opération. En effet, si la cellule, dès la réception de ce message, répond positivement, on risque d'étendre inutilement la recherche à un niveau supérieur, alors qu'au niveau courant on n'est pas encore sûr qu'il n'existe pas de cellules libres. De ce fait, toutes les réponses aux demandes de mise en place de relais sont mises en attente, jusqu'à la réception de la réponse des quatre cellules extrêmes (figure4.19). Ces réponses sont soit des demandes de mise en œuvre de relais soit des signaux d'échec de recherche (lorsqu'une cellule se trouve au bord du réseau).

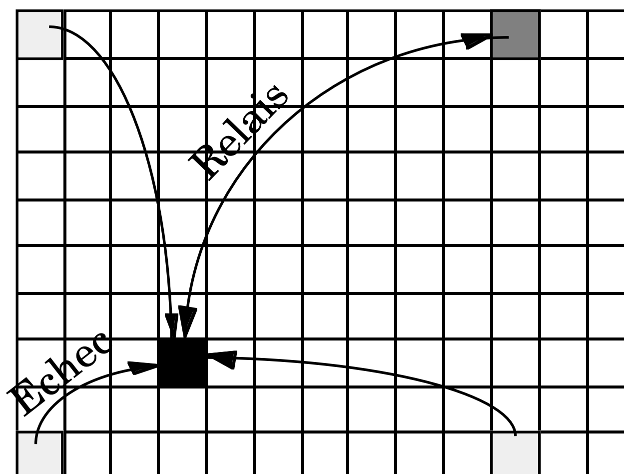


Fig.4.19. Protocole d'établissement de relais.

Nous pouvons imaginer plusieurs stratégies à ce niveau. Des stratégies de prévention, comme celle que nous venons de présenter, ou des stratégies de guérison. Dans ce dernier cas, une cellule qui reçoit une demande pour la création d'un relais y répond positivement, sans attendre. A la réception d'un message lui signalant la présence d'une cellule libre, elle devra mettre en œuvre un mécanisme pour intercepter et annuler les demandes en cours. Pour cela, il faut prévoir une priorité entre les différents messages pour que les messages d'annulation puisse progresser plus rapidement que les messages de demande de cellules libres.

Cependant, nous penchons beaucoup plus pour les méthodes de prévention qui, comme dans ce cas, nécessitent moins de ressources. Elles reposent sur des protocoles plus simples à mettre en place.

Chapitre 5
Expérimentation & évaluations

Bon nombre de raisons ont justifié le temps non négligeable consacré à la réalisation d'un simulateur de bas niveau pour l'évaluation des performances des solutions proposées. Parmi ces raisons, nous en citons les plus importantes:

- Le caractère imprévisible du comportement des programmes parallèles rend extrêmement difficile l'élaboration de modèles suffisamment fiables pour se passer de simulateurs. Les équipes qui travaillent sur l'évaluation des machines parallèles ont mesuré cette difficulté [TRON94] [KITAJIMA93]. Il paraît tout particulièrement délicat d'établir un modèle pour les communications dans un réseau "réel" où les liens de communication ne sont pas tous sollicités de la même manière aux mêmes instants (charges différentes).
- Le choix entre plusieurs options d'implantation, que ce soit au niveau du jeu d'instructions qu'au niveau du réseau de communication, nécessite d'en évaluer les performances sur des exemples "réels".
- La mise en évidence de phénomènes très fins ou difficilement prévisibles nécessite une simulation fine.

Pour illustrer ce dernier point, nous pouvons mentionner le comportement du réseau par rapport au protocole de recherche de cellules libres. Dans certaines conditions, ce protocole peut induire des cas d'interblocage. Rappelons que le message de recherche de cellule est composé de quatre paquets successifs. Dans une première version du simulateur, on réalisait un recouvrement entre l'envoi de la requête et la réponse à celle-ci, émise dès réception du premier paquet (figure5.1).

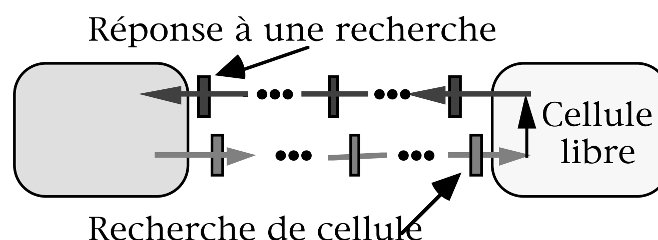


Fig.5.1. Recouvrement entre une demande de cellule et sa réponse.

Mais la simulation fine a permis de montrer que, lors d'un échange entre deux cellules, il pouvait se produire le cas d'interblocage de la figure5.2.

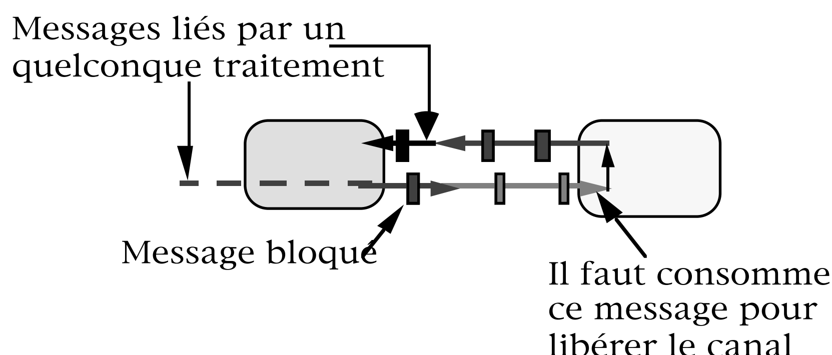


Fig.5.2. Cas d'interblocage.

Cette constatation nous a conduits à considérer ce protocole différemment: tout message d'une requête est d'abord consommé (extrait du réseau de communication) avant d'y répondre.

Ces raisons nous ont donc poussés à réaliser un simulateur fondé sur le cycle de base de la machine cible. La difficulté principale réside alors dans le déverminage de programmes parallèles.

5.1. L'environnement de simulation

L'environnement de simulation est construit autour de trois modules (figure5.3):

- un assembleur pour la description du code des différents algorithmes,
- un éditeur de liens , et
- le simulateur proprement dit.

Le temps qui nous était imparti ne nous a pas permis d'élaborer un compilateur pour un langage d'acteur du type CANTOR (cf.§2.3). Dans cette première expérimentation, nous avons défini un assembleur pour l'écriture des programmes (voir annexeII). Nous sommes cependant tout à fait conscients de la difficulté que représente la programmation à un niveau aussi bas.

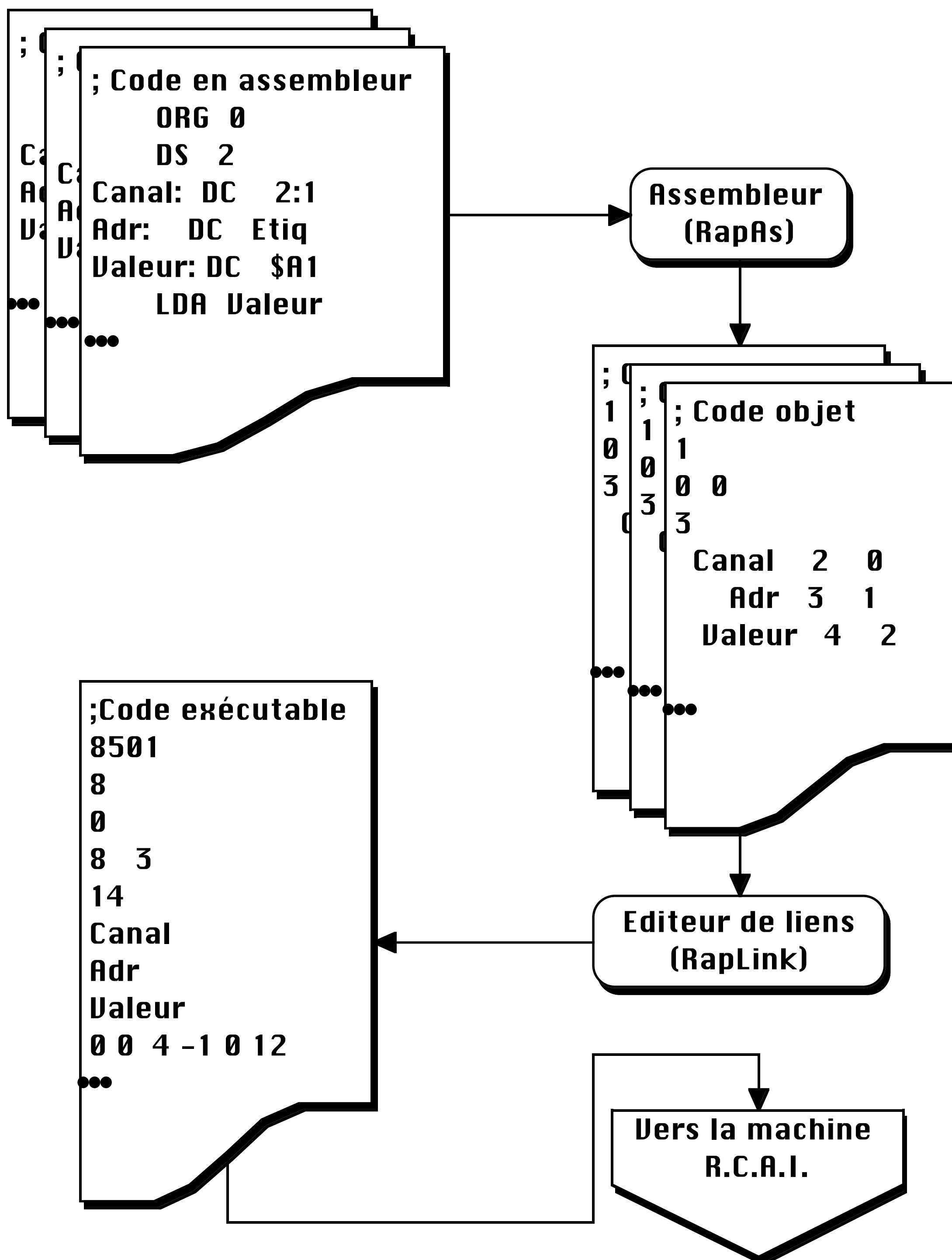


Fig.5.3. Schéma d'exécution du processus de simulation.

L'assembleur fournit du code intermédiaire ainsi que la table des symboles pour chaque fichier source spécifié en entrée de ce module.

L'éditeur de liens résout les références croisées extraites à partir de ces tables. De plus, en combinant tous les fichiers intermédiaires engendrés par l'assembleur dans une première étape, l'éditeur de liens génère un fichier unique de code exécutable. Les instructions sont réparties suivant leur fonction (calcul, communication, recherche de cellule libre...). Chaque groupe d'instructions détermine, à l'exécution, la nature de l'activité de la cellule qui exécute une des instructions de ce groupe.

5.1.1. Le simulateur

Le processus de simulation est découpé en trois phases:

- le chargement du code et la spécification du fichier des données,
- le paramètrage de la session courante, et finalement
- l'exécution du code chargé et la prise de mesures suivant les indications de l'utilisateur.

5.1.1.1. Chargement du code

A la demande de l'utilisateur, le fichier du code objet est chargé suivant les spécifications contenues dans ce dernier. Il faut signaler que si une même position mémoire, au niveau d'une cellule donnée, est référencée à plusieurs reprises dans le fichier, c'est la dernière valeur lue qui est conservée (les autres ayant été écrasées). Lorsqu'une cellule est déclarée active, son registre PC est initialisé à la valeur indiquée par la même instruction (voir le listing de la figure5.6). Les autres cellules, les cellules inactives, sont marquées libres.

5.1.1.2. Paramètrage d'une session

Le paramètrage d'une session consiste à spécifier principalement trois valeurs:

- le temps limite du routeur, au-delà duquel il faut retirer du réseau le message ayant provoqué le débordement du compteur et le réexpédier ultérieurement,
- le temps limite de la recherche, au bout duquel on relance la recherche d'une cellule libre,
- le nombre de relances de la recherche de cellules libres.

5.1.1.3. Exécution et extraction de mesures

Une fois le code chargé, on peut exécuter le programme selon trois modes:

- pas à pas,
- durant un certain nombre de cycles,
- ou d'une manière continue.

Pour suivre le déroulement d'une exécution, nous proposons plusieurs vues:

- une vue de l'activité du réseau,
- une vue du réseau de communication et des messages qui y transitent,
- une trace d'une ou plusieurs cellules,
- une vue des registres d'un ensemble de cellules.

L'extraction de mesures est réalisée à la demande de l'utilisateur sur la période et suivant l'intervalle d'observation qu'il fixe. On enregistre l'activité moyenne par zone d'activité pour l'ensemble du réseau (à chaque intervalle d'observation spécifié).

5.2. Les benchmarks

Pour évaluer les performances des solutions proposées dans ce manuscrit, nous avons retenu les exemples suivants:

- le crible d'Erathostène,
- le tri d'une suite d'entiers,
- et le calcul de la suite de Fibonacci.

5.2.1. Crible d'Erathostène

Le crible d'Erathostène[32] permet d'obtenir l'ensemble des nombres premiers compris entre 2 et une borne fournie par l'utilisateur. Le principe de cet algorithme est très simple. Un acteur spécifique, acteur racine, génère l'ensemble des entiers compris entre **2** et la borne **n** et les fait parvenir au premier maillon de la chaîne (figure5.4).

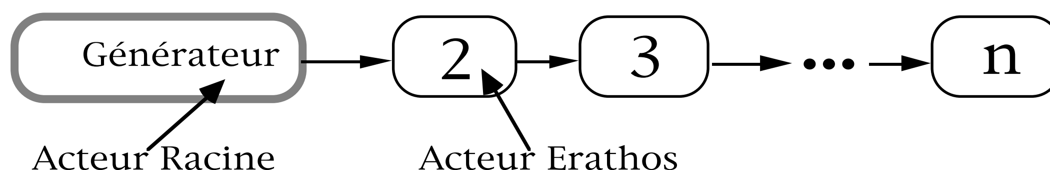


Fig.5.4. Crible d'Erathostène.

Lorsqu'un acteur reçoit une valeur, il la divise par le nombre premier qu'il représente. Si celle-ci n'est pas multiple du nombre premier en question, elle est renvoyée vers le prochain maillon de la chaîne. Par contre, lorsqu'une valeur a été reconnue multiple du nombre premier courant, elle est détruite. Dans le cas où l'acteur Erathos courant est le bout de la chaîne, il procède à une recherche de cellule libre pour s'y dupliquer. Après quoi, il communique à ce nouvel acteur le nombre premier qui vient d'être trouvé.

Pour signaler la fin du traitement, le générateur envoie la valeur -1 au premier élément du crible. Chaque acteur qui reçoit celle-ci va la renvoyer au prochain acteur avant de se suicider. Après un suicide, la cellule est libre et peut donc être réutilisée par d'autres acteurs du crible. La figureII.1 (de l'annexeII) reprend le code assembleur de l'acteur Erathos.

Le générateur de nombres (acteur racine) produit les entiers entre 2 et 255. Cet exemple a été testé sur un réseau de 8x3 cellules uniquement. Les

---

[32]Le code source de cet exemple a été présenté dans le paragraphe 2.3.5.

cellules libérées sont réutilisées par d'autres acteurs Erathos. De plus, c'est le réseau de communication qui sert de tampon pour l'absorption des entiers générés par l'acteur racine. Les communications sont, dans le cas du modèle dynamique, complètement asynchrones.

La figure5.5 retrace la surcharge (*overhead*) occasionné par le déplacement de code dans cet exemple. Nous remarquons que ces coûts supplémentaires pour la gestion de la distribution de code sont en deçà des 4,5% de l'activité totale du réseau. Ceci est dû essentiellement au bon recouvrement entre le calcul proprement dit du crible d'Erathostène et celui de la copie de code vers d'autres cellules.

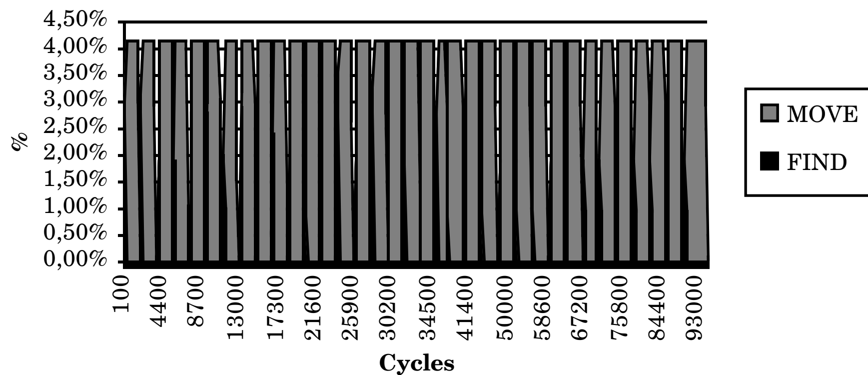


Fig.5.5. Activité des instructions de recherche de cellules libres et de déplacement de code.

Le chargement de code du réseau dure à peu près 28000 cycles dans une première phase (figure5.6). L'activité du réseau commence à baisser à partir de ce moment. Cette seconde phase correspond au vidage du réseau (sortie des résultats).

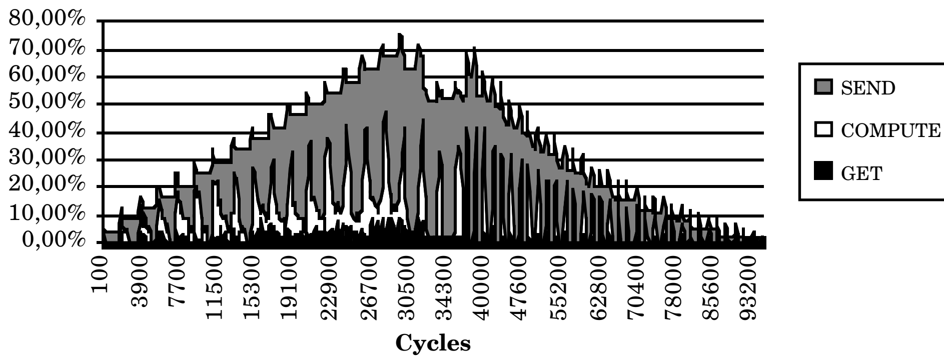


Fig.5.6. Activité des instructions de calcul et de communication.

5.2.2. Tri d'entiers

Il nous fallait aborder cet aspect algorithmique du fait que les algorithmes de tri sont souvent cités dans les différents benchmarks. Cependant, vu la nature de notre modèle d'exécution, nous ne pouvons pas utiliser les algorithmes de tri basés sur les réseaux (statiques) de tri [QUINN87] ou les algorithmes basés sur la circulation de données uniquement (le code étant supposé déjà présent sur place).

Nous avons alors mis en œuvre un algorithme simple, qui suit le même schéma d'exécution que le précédent (figure5.7). Chaque acteur correspond à un entier. Lorsqu'un acteur reçoit un entier il le compare à lui-même. Si celui-ci est plus grand il le renvoie au prochain. Sinon, l'acteur devient ce nouvel entier et fait parvenir au prochain acteur son ancienne valeur.

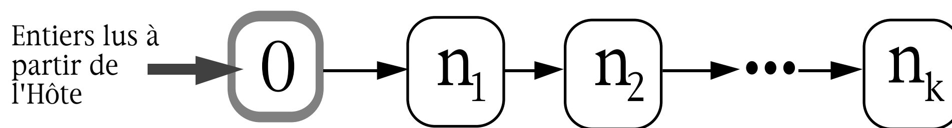


Fig.5.7. Tri d'entiers.

Dans le cas où l'acteur en cours de traitement est en bout de chaîne, il procède à la recherche d'une cellule libre dans une première phase. Puis, il s'y duplique et lui communique la valeur adéquate.

De la même manière que précédemment, il existe une valeur spéciale (-1) qui provoque la terminaison des acteurs inutilisés.

Dans cet exemple, nous trions 150 entiers, sur un réseau de 8x8 cellules. Chaque cellule prend en charge un seul entier.

Le coût du déplacement de code dans cet exemple reste similaire au précédent pour les mêmes raisons évoquées plus haut (figure5.8).

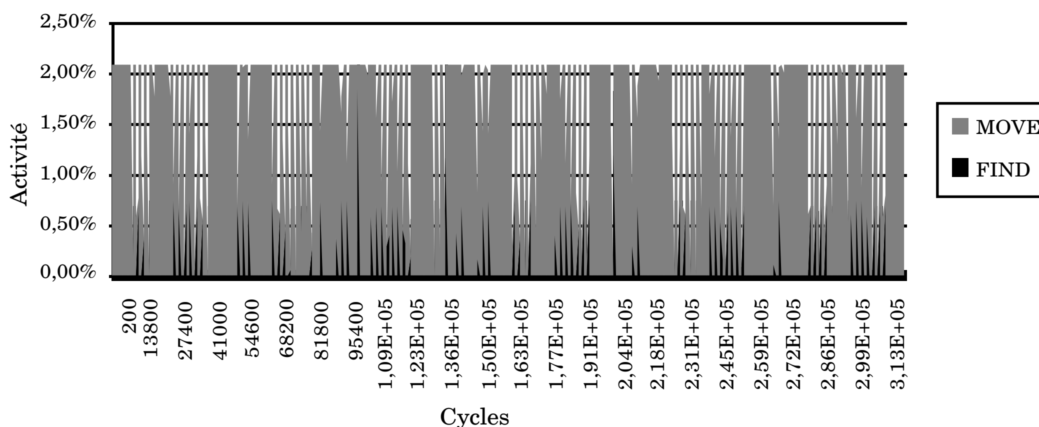


Fig.5.8. Activité des instructions de recherche de cellules libres et de déplacement de code.

L'importance de l'activité des instructions d'envoi de messages est la conséquence des arrêts nécessaires en bout de chaîne pour laisser le temps à ce dernier de copier le code correspondant au nouvel élément vers sa destination. A partir du moment où le nombre de ces éléments est suffisant pour absorber tous les messages en entrée (vers le cycle 72000), cette activité commence à décliner.

Aussi bien dans le cas du crible d'Erathostène que celui du tri, le coût dû au déplacement de code reste stable. En effet, comme nous l'avons signalé auparavant, il n'y a qu'une cellule à tout moment dans le réseau qui réalise l'opération de copie de code.

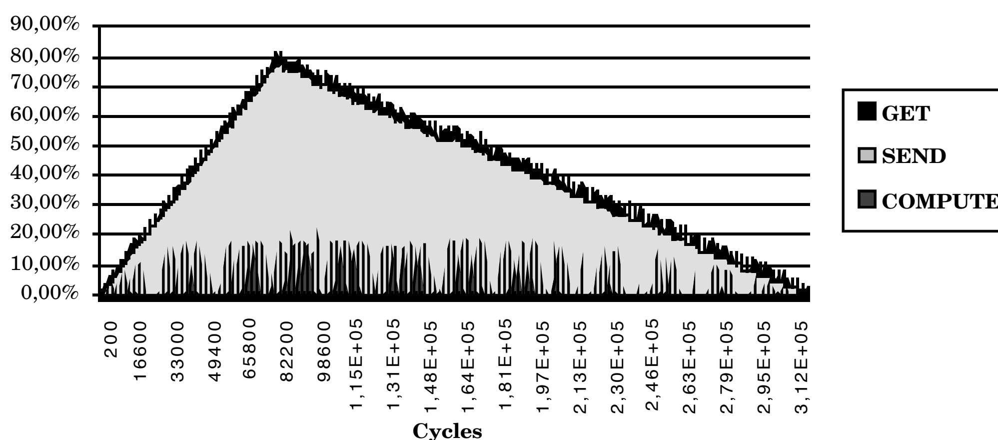


Fig.5.9. Activité des instructions de calcul et de communication.

Nous constatons que pour ces deux exemples très simples, le calcul ne représente qu'une mince portion de l'activité totale des processeurs, et que les communications sont plus importantes[33]. Cependant, les opérations d'allocation consomment peu de temps.

Le comportement de ces deux algorithmes est peu dynamique: il y a création d'acteurs au début, ensuite davantage de communications. Toutefois, même dans de telles configurations, l'allocation dynamique de processeurs présente un net avantage par rapport à l'allocation statique: le calcul se développe de lui-même à l'intérieur du réseau de cellules, sans nécessiter le chargement préalable de toutes les cellules (cf.§5.2.4).

Nous allons maintenant étudier un exemple dont le comportement en terme de créations d'acteurs est beaucoup plus dynamique.

[33]Ceci est dû au principe même de ces algorithmes qui font circuler des données (donc des messages) pour, dans un premier cas soit éliminer ces nombres soit les abouter en fin de chaîne comme nouveaux nombres premiers, et dans un autre cas, permuter des nombres afin de les classer.

5.2.3. Calcul de la suite de Fibonacci

Contrairement aux deux exemples précédents, le calcul de la suite de Fibonacci fournit un arbre binaire (figure5.10). Il y a donc concurrence pour la recherche de cellules libres.

La formule de calcul de la suite de Fibonacci est:

$$\begin{cases} Fib(0) = 0 \\ Fib(1) = 1 \\ Fib(n) = Fib(n-1) + Fib(n-2) \end{cases}$$

Chaque acteur Fib qui reçoit une valeur (supérieure à 2) crée deux nouveaux acteurs Fib auxquels il communique les valeurs appropriées. Il se met ensuite en attente des résultats avant de les renvoyer à celui qui l'a créé.

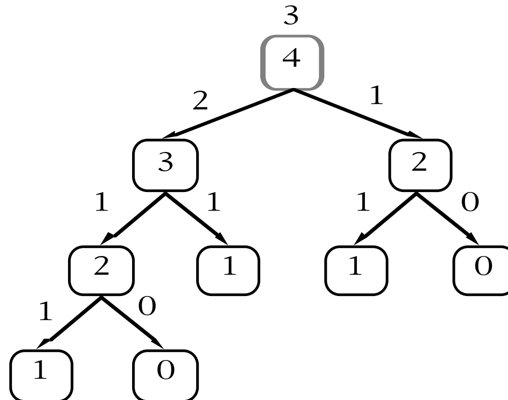


Fig.5.10. Calcul de la suite de Fibonacci (pour n = 4).

Contrairement aux deux exemples précédents, le nombre d'acteurs à la recherche de cellules libres est variable et assez important. Par conséquent, l'activité correspondante est normalement plus importante. Dans le cas du calcul de la suite de Fibonacci de 9, sur un réseau de 10x10 cellules, cette activité présente un maximum de 12% (figure5.11). Cependant, cette activité n'est que de 5,07% en moyenne de l'activité totale du réseau. Elle reste donc relativement faible.

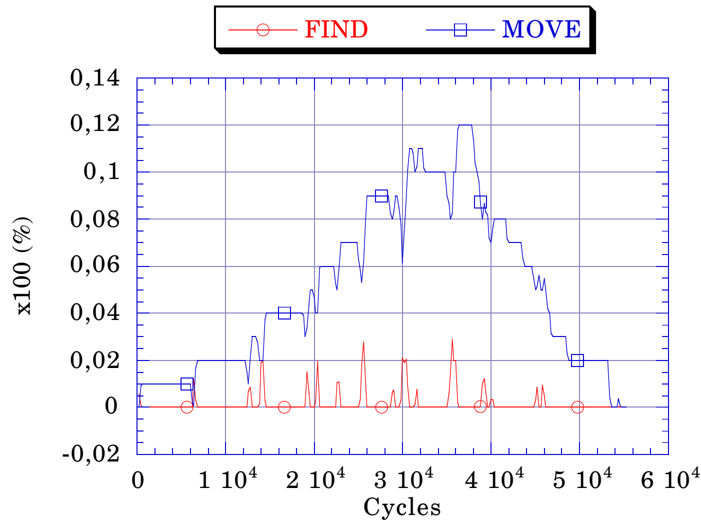
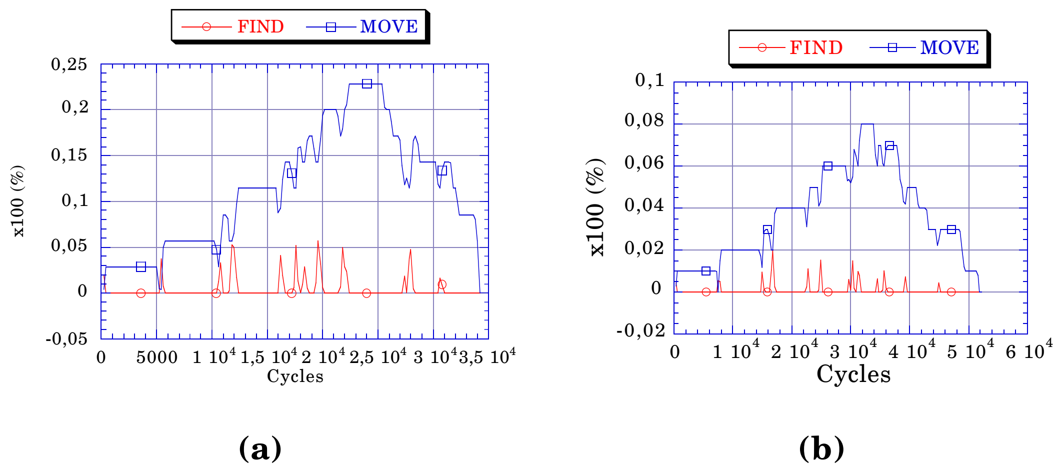


Fig.5.11. Activité des instructions de recherche de cellules libres et de déplacement de code (Fib (9) dans un réseau 10x10 cellules).

En revanche, l'exécution de cet algorithme sur des réseaux de plus petite taille peut nécessiter plus de recherche (figure5.12). En effet, une même quantité d'activité sur des réseaux de dimensions différentes induit une activité relative (par rapport à la taille du réseau en question) différente.



(a) **(b)**

Fig.5.12. Activité des instructions de recherche de cellules libres et de déplacement de code: (a) Fib(8) sur un réseau de 10 x10 cellules, (b) Fib(8) sur un réseau de 7x5 cellules.

Dans la figure5.13, le nombre de processus en attente de messages est relativement important. Cela est dû au schéma d'exécution de la suite de Fibonacci. Chaque acteur qui crée deux nouveaux acteurs se met automatiquement en attente des résultats.

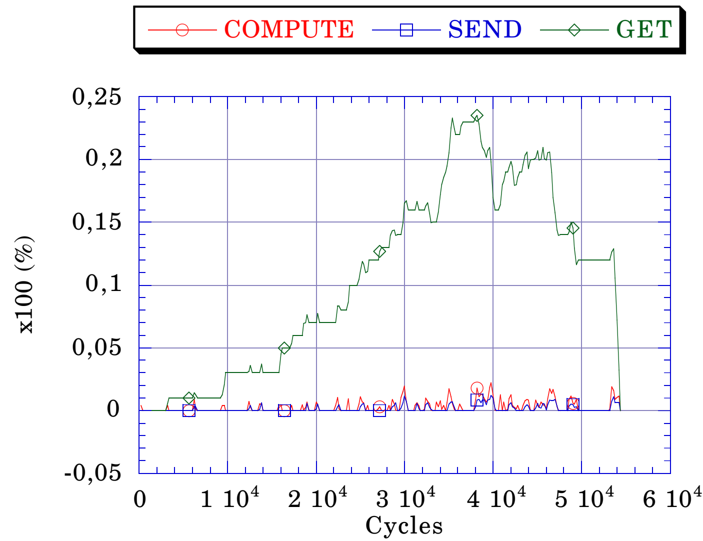


Fig.5.13. Activité des instructions de calcul et de communication.

5.2.4. Comparaison des modèles statique et dynamique

Un des objectifs de cette thèse a été de réduire le temps nécessaire au chargement de code. Nous allons mettre en évidence ce gain en considérant les exemples du tri de 150 entiers et du calcul de la suite de Fibonacci de 8. Les E/S entre le réseau et le système hôte sont réalisées au taux de 500 Ko/s.

Le point le plus important à noter est celui du nombre de cellules nécessaires à l'évaluation de ces fonctions. Dans le cas statique, il aurait fallu 67 cellules pour le calcul de la suite de Fibonacci de 8 (en suivant le développement strict de cette fonction). Dans la version dynamique, nous n'en utilisons que 35. Nous réalisons donc une économie non négligeable de ressources grâce à la réutilisation de cellules qui ne servent plus (fonction de ramasse-miettes).

De plus, dans les deux exemples, nous ne chargeons que le code d'une seule cellule. Le développement de chaque fonction, par la suite, génère le nombre nécessaire d'acteurs pour l'évaluation de la fonction correspondante. De ce fait, le temps de chargement du code et de son exécution dans la version dynamique est très inférieur à celui nécessaire uniquement au chargement du code correspondant dans la version statique (table5.1).

	Code Chargé	I/O (500 K/s)		Nombre de Cellules utilisées (Dyn.)	Cellules utiles (Stat.)	Overhead du déplacement de code (en moyenne)
		Dyn.	Stat.			
Tri d'entiers (8x8 cells)	1 cellule	0,5ms 31ms	82ms	64 (43%)	150	1,5%
Calcul Fibonacci (Fib:8, 7x5 cells)	1 cellule	0,5ms 3,7ms	40ms	35 (52%)	67	11,3%

Table5.1. Comparaison des modèles dynamique et statique [34].

Pour terminer, nous pouvons remarquer que le coût supplémentaire occasionné par le déplacement de code reste relativement faible par rapport à l'activité totale du réseau.

[34] Les parties grisées représentent le temps d'exécution de l'algorithme dans la version dynamique.

5.3. Quelques résultats globaux sur des points délicats

Dans cette section, nous mettons en évidence la relation qui existe entre la fréquence du processeur de communication (le routeur) et celle du processeur de calcul.

La version statique de R.C.A.I., dont un prototype a été réalisé par S.M.KARABERNOU [KARABERNOU93], fixait la fréquence d'horloge du processeur à 10MHz et celle du routeur à environ 20MHz. Les évaluations menées par P.RUBINI [RUBINI92] ont montré que le réseau était suffisamment performant et qu'il n'était pas nécessaire d'en augmenter les performances. La batterie de simulations qui suit va nous permettre de montrer que ceci n'est plus valable dans la version dynamique.

Pour cela, nous considérons le cas plus général des benchmarks utilisés dans ce chapitre, à savoir celui du calcul de la suite de Fibonacci. En effet, comme nous l'avons vu auparavant, cette fonction génère un arbre binaire à l'exécution. Il y a donc plusieurs cellules à la fois qui concourent pour la recherche de cellules libres.

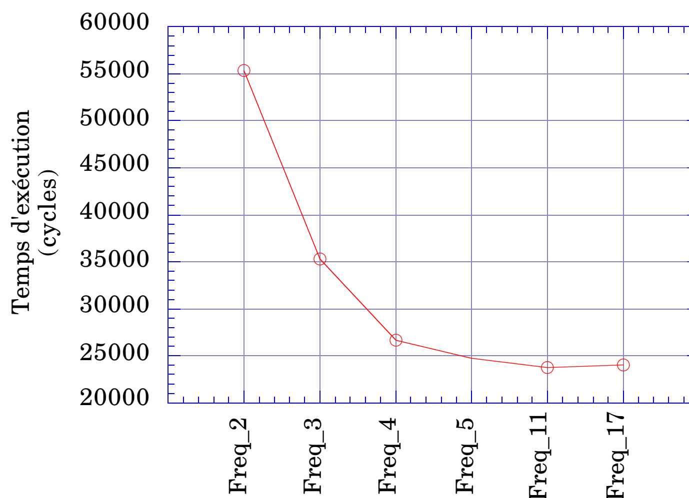


Fig.5.14. Comparaison des performances suivant le rapport des fréquences du routeur et du processeur.

Nous calculons la suite de Fibonacci pour 9 sur un réseau de 16x16 cellules. Ce calcul génère 109 éléments. Nous avons réalisé les simulations pour des fréquences du routeur 2, 3, 4, 5, 11 et enfin 17 fois plus rapide que celle de l'unité de calcul (processeur).

Nous remarquons qu'il y a un gain de 36,22 % de temps d'exécution entre une fréquence double (la fréquence actuelle du routeur du prototype de la version statique) et une fréquence triple (figure5.14). Il y a donc tout intérêt à ce que le routeur soit beaucoup plus performant que le processeur de calcul. En effet, le nombre important de messages de mouvement de code entre les cellules nécessite que cette opération soit réalisée en un temps minimal pour ne pas pénaliser le calcul sur les processeurs. Cependant, comme il ressort de la figure5.14, il est inutile que cette fréquence aille au-delà du quintuple (ce qui correspond à une fréquence du routeur de 50MHz). Le gain après cette valeur est à l'évidence beaucoup moins important.

Cependant, nous devons vérifier que ces résultats restent valables dans le cas du transfert d'un nombre différent de messages de code. Pour cela nous considérons le calcul de la suite de Fibonacci de 8 sur un réseau de 16x16 cellules. Nous prenons en compte successivement les tailles de 125, 150, 200 et 250 messages de code à transférer à chaque nouvelle création d'acteurs (figure5.15).

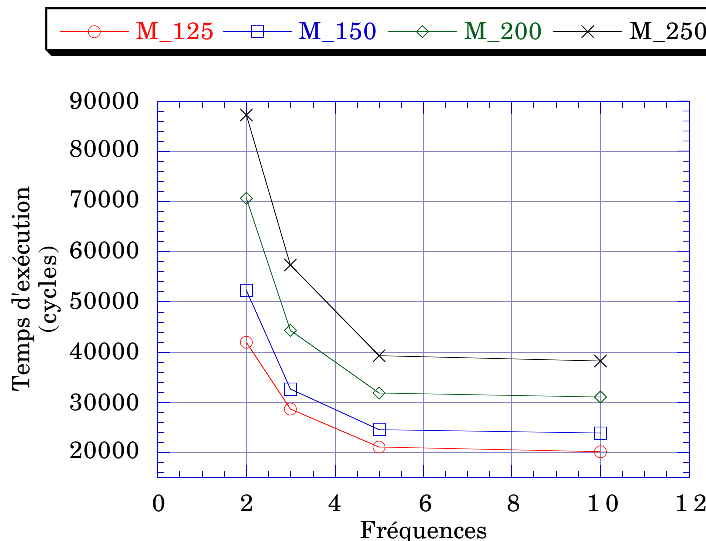


Fig.5.15. Temps d'exécution en fonction de la fréquence du routeur et du nombre de messages (de code) transférés.

Ce dernier graphique confirme bien qu'en augmentant les performances du routeur nous arrivons à améliorer nettement celles du réseau. De la même manière que dans la figure5.14, le seuil du gain de performances se situe bien entre une fréquence quadruple et une fréquence quintuple.

Eu égard à ces différentes fréquences du routeur, nous allons voir la répartition des messages de données, de recherche de cellules libres et de duplication de code. Nous donnons dans un premier temps cette répartition pour une fréquence double, c'est à dire la même fréquence que celle du routeur de la version statique, pour le déplacement de 125 messages de code à la fois (par acteur créé).

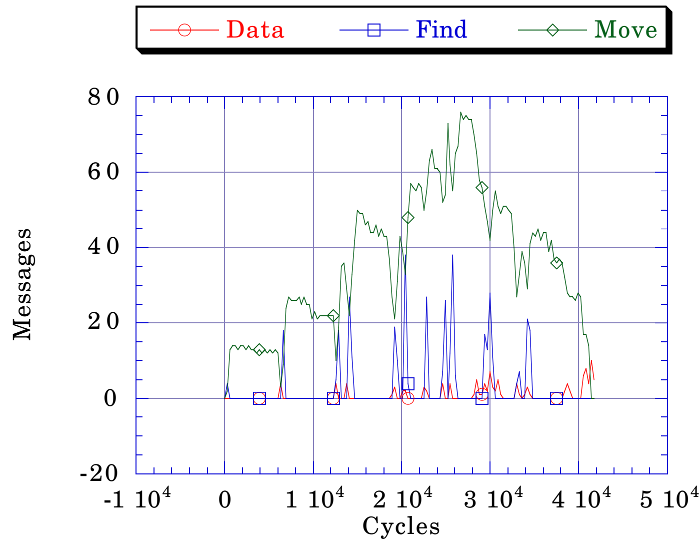


Fig.5.16. Répartition des différents messages.

Nous voyons clairement sur la figure5.16 que les messages de duplication de code sont prépondérants (cf.table5.2).

Type message	Total	Observation
Données	116	Données & résultats
Code	5040	(125+1)*40 acteurs
Recherche	389	**9,49** messages en moyenne par acteur

Table5.2. Répartition des différents types de messages pour l'exécution de la suite de Fibonacci de 8 sur un réseau de 16x16 cellules.

Par conséquent, dans toutes les simulations qui suivent, nous ne reprenons que les messages de déplacement de code pour la comparaison des performances suivant des fréquences différentes du routeur (figures5.17, 5.18, 5.19 & 5.20). Toutes ces figures montrent qu'à chaque fréquence supérieure, il y a une plus grande concentration de messages sur un même intervalle de temps et un gain de temps d'exécution significatif.

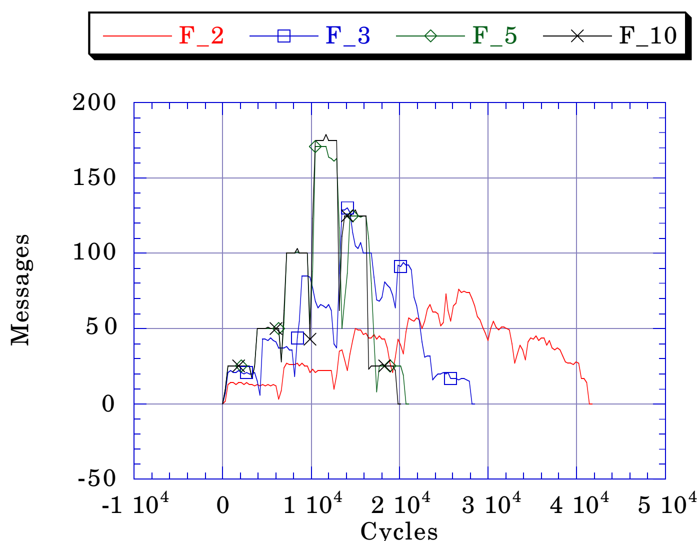


Fig.5.17. Répartition des messages de code pour 125 messages à déplacer à différentes fréquences.

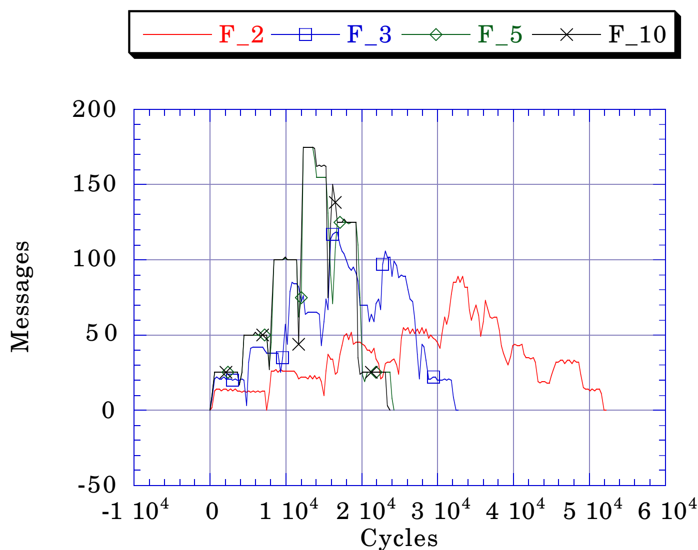


Fig.5.18. Répartition des messages de code pour 150 messages à déplacer à différentes fréquences.

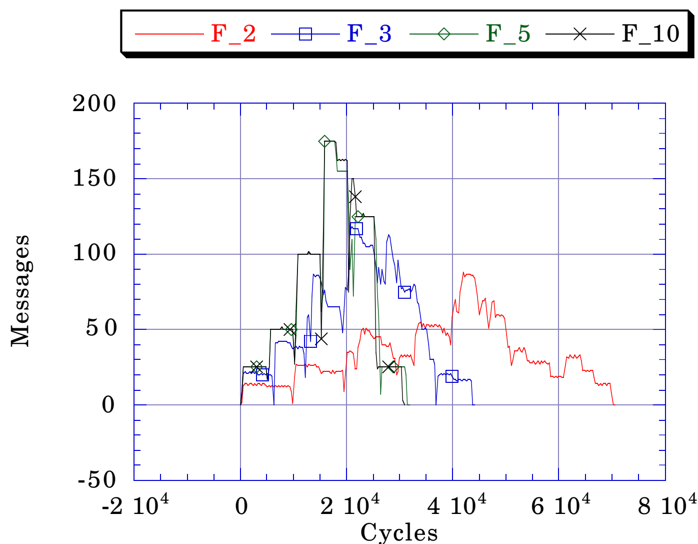


Fig.5.19. Répartition des messages de code pour 200 messages à déplacer à différentes fréquences.

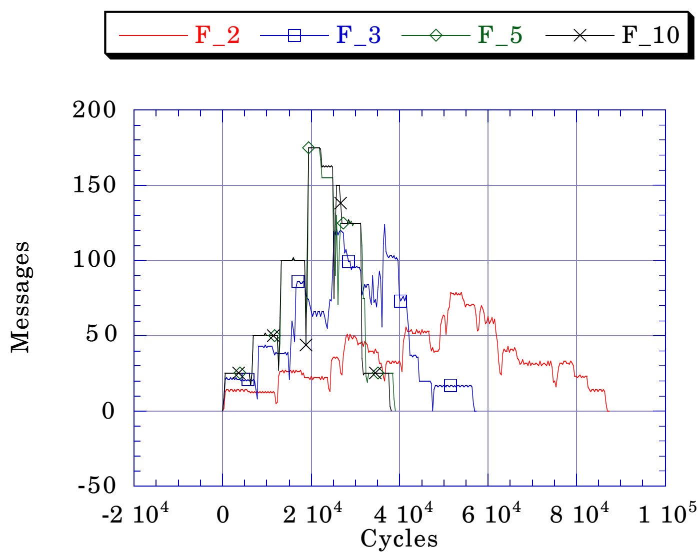


Fig.5.20. Répartition des messages de code pour 250 messages à déplacer à différentes fréquences.

Cette concentration de messages risque, dans le cas des messages de données, d'engendrer des cas de dépassement de la capacité de traitement des processus de l'ensemble de ces messages.

Il y a donc lieu à ce que la zone de réception de messages de données soit accrue. La question est de savoir dans quelle proportion doit s'opérer cet accroissement.

Dans la table5.3, nous reportons la moyenne du nombre de messages transférés dans un même intervalle de temps. Il s'agit du calcul de la suite de Fibonacci de 9. Ces moyennes sont calculées en fonction de la taille du code à déplacer et de la fréquence du routeur. Pour la démonstration qui nous concerne, nous considérons les messages de déplacement de code plutôt que ceux des données proprement dites. En effet, l'évaluation de la suite de Fibonacci pour un même nombre génère la même quantité de données. C'est la raison pour laquelle le nombre de ces messages ne peut constituer un élément pertinent pour l'évaluation de la taille des boîtes aux lettres (BAL) à affecter aux acteurs. Par contre, nous pouvons faire varier la taille du code à transférer à chaque nouvelle création d'un acteur. Parmi les messages de code transférés, il est évident qu'une grande partie (des messages ajoutés) ne fait pas réellement partie du code de la fonction de Fibonacci.

Nous allons donc évaluer la taille idéale de la BAL pour absorber les messages supplémentaires engendrés par les performances accrues du routeur (dans un même intervalle de temps).

En considérant que la version statique du réseau R.C.A.I réalisait un équilibre entre la capacité de traitement du processeur et celle d'acheminement de messages par le routeur[35], nous pouvons à ce moment là établir la relation suivante (eu égard à la table5.4) :

$$Taille_{BAL_Dynamique} = Taille_{BAL_Statique} * 1{,}59 \qquad (1)$$

Le coefficient 1,59 de la relation **(1)** correspond au rapport maximal lors du passage d'une fréquence double à une fréquence triple. Ce coefficient peut donc être majorée par 2 (puisqu'il s'agit du nombre de messages de la BAL).

Il suffit donc de doubler la taille de la boîte aux lettres pour absorber les messages engendrés par l'augmentation des performances du routeur.

[35] D'après les travaux de [RUBINI_92] ceci semble être le cas.

Taille du code transféré (nombre de messages)	**Fréquence 2**	**Fréquence 3**	**Fréquence 5**	**Fréquence 10**
125	36,000	52,500	70,986	74,118
150	34,514	54,909	73,659	75,500
200	34,068	54,324	75,140	77,308
250	34,384	52,292	76,641	78,437

Table5.3. Moyenne du nombre de messages transférés dans les mêmes délais suivant la taille du code à déplacer et la fréquence du routeur.

Taille du code transféré (nombre messages)	**Fréquence** passage de 2 à 3	**Fréquence** passage de 3 à 4	**Fréquence** passage de 5 à 10
125	1,46	1,35	1,04
150	1,59	1,34	1,02
200	1,59	1,38	1,02
250	1,52	1,47	1,02
Moyenne:	**1,54**	**1,39**	**1,03**

Table5.4. Rapports d'augmentation du nombre de messages suivant la fréquence du routeur.

Conclusion & perspectives futures

6. CONCLUSION ET PERSPECTIVES FUTURES

Cette thèse a concerné l'étude de la gestion dynamique d'un réseau cellulaire, massivement parallèle à grain fin. Le point de départ fut l'utilisation d'un langage d'acteur simple pour la programmation du réseau. Au cours de l'évolution de la thèse, des problèmes fondamentaux ont été mis en exergue. Dans la mesure du possible, nous avons apporté une solution à plusieurs d'entre eux. C'est ainsi que nous avons proposé à partir de la spécification du modèle d'acteurs un ensemble d'améliorations des fonctions de communication et du jeu d'instructions du processeur.

Dans une première partie, nous avons abordé les problèmes liés à la régulation de charge du réseau cellulaire. Nous sommes arrivés à la conclusion qu'il fallait, vu la finesse visée du grain, une solution qui requiert un minimum (voire pas du tout, comme c'est notre cas) d'échange d'informations pour le maintien d'une connaissance partielle de la charge des voisins immédiats, et qui, de préférence, soit intégrée dans le réseau de communication. La solution proposée repose sur la diffusion d'une requête de cellules libres. Nous avons également proposé une solution originale, à notre connaissance, pour la régulation de la charge dans une machine de plus gros grain.

Nous nous sommes aussi intéressés à la récupération de cellules qui ne servent plus pour les réutiliser. Cette fonction est transparente à l'utilisateur. Le mécanisme de ramasse-miettes repose sur l'algorithme classique du compteur de références. Nous avons proposé une solution plus générale mais que nous n'avons pas pu implémenter pour l'instant. Elle fait partie intégrante du modèle d'acteurs.

Nous avons mis en œuvre une solution originale pour la duplication de code, en installant plusieurs “serveurs” ou “prototypes” de code, dans le but de réduire la charge sur ces éléments pour la copie dynamique de code.

Un autre aspect de cette étude a été la prise en compte de mécanismes hérités des versions précédentes du réseau[36]. Il s'agit notamment de la limitation de l'espace d'adressage de chaque cellule (acteur). La solution que nous avons proposée pour ce problème a été intégrée au modèle d'acteurs, en ajoutant des acteurs spécifiques que nous appelons **acteurs-relais**[37]. Ces acteurs brisent ces restrictions.

Pour évaluer les performances des solutions proposées, nous avons développé un simulateur de bas niveau (cycle), un assembleur et un éditeur de liens. Les premiers résultats que nous avons obtenus montrent que la surcharge (*overhead*) générée par le mouvement du code est relativement faible en terme de temps des processeurs (temps CPU). Par contre, le réseau au niveau des communications est saturé tel qu'il a été imaginé dans la version statique, et nous avons vu qu'il faudrait en augmenter les performances pour obtenir un fonctionnement satisfaisant.

L'ensemble des résultats démontrent l'intérêt de l'approche, à condition toutefois de disposer d'un outil de communication suffisamment puissant.

L'avantage de cette approche est d'éviter le problème du chargement initial du code. Par ailleurs, il faut noter que, même si le réseau de communication interne est saturé par moment et ralentit de ce fait l'exécution de l'application, le résultat final est tout de même meilleur que celui obtenu en chargeant entièrement le code dans une première phase (cf.table5.1).

[36]Le but de cette thèse n'est pas de revoir entièrement l'architecture de la machine R.C.A.I, mais de proposer quelques améliorations pour l'adapter au modèle d'exécution dynamique.

[37]Comme nous l'avons déjà spécifié au niveau du chapitre IV, la solution retenue pour l'instant, dans cette première version dynamique de R.C.A.I, est celle basée sur des **relais physiques** (cf.§4.3.3.1).

Perspectives

Nous envisageons plusieurs perspectives possibles. La première, concernant le modèle de programmation, serait d'étudier les systèmes d'intelligence artificielle distribués (SIAD), suivant le modèle d'acteurs, et d'étudier leur adéquation à des architectures comme la nôtre.

Une étude menée dans le cadre du projet PVC/BOX [HEMERY94], laisse à l'utilisateur le choix d'une politique de transfert de processus, d'échange d'information et de localisation pour la régulation de la charge des processeurs. Il serait intéressant d'étudier son intégration dans un méta-langage d'acteur qui spécifierait pour chaque application devant s'exécuter tous les protocoles de communications[38].

Ces travaux se sont restreints à l'étude du cas particulier où une cellule n'abrite qu'un seul acteur. Le cas de plusieurs acteurs par cellule, qui constitue une suite logique de notre étude, nécessite l'intégration d'un mécanisme de régulation de charge. Cette intégration peut intervenir soit au niveau logiciel soit au niveau matériel.

De la même façon, il serait intéressant d'expérimenter et d'évaluer l'algorithme CLIMB sur des machines existantes du type TNode ou SP1.

Un tout autre aspect qu'il serait souhaitable d'aborder est celui du déverminage de programmes parallèles. Cette fonction est assez délicate à mettre en œuvre. Cependant, il existe beaucoup de travaux en cours sur ce sujet (tel que le projet TRACE du LMC/IMAG) à partir desquels chacun peut s'inspirer, ou mieux encore, collaborer avec ces équipes pour ouvrir de nouveaux horizons!

Nous terminerons par le rêve de toute équipe de recherche en architectures de machines: voir le jour d'un prototype sur silicium des mécanismes développés dans cette thèse!

[38] Il s'agit de la notion de **réflexivité** dans les SIAD [FERBER_90].

Bibliographie

BIBLIOGRAPHIE

[AGHA86a] G.AGHA, C.HEWITT, *Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism*, Dans *Interaction Languages, Structures and Protocols*, Juin1986.

[AGHA86b] G.AGHA, *ACTORS, a Model of Concurrent Computation in Distributed Systems*, The MIT Press Series in Artificial Intelligence, 1986.

[AGHA93] G.AGHA, I.A.MASON, S.SMITH & C.TALCOTT, *A Foundation of Actor Computation*, Stanford University Report, Juillet1993.

[AGHA94] G.AGHA, N.VENKATASUBRAMANIAN & C.TALCOTT, *Scalable Distributed Garbage Collection for Systems of Active Objects*, Document disponible par *ftp* au site *cs.uiuc.edu*, 1994.

[AHAMAD87] M.AHAMAD, M.H.AMMAR, J.BERNABEU, M.Y.KHALDI, *A Multicast Scheme for Locating Objects in a Distributed Operating System*, Technical Report #GIT-ICS-87/01, Janvier1987.

[AKOUTF92] Ch.AKTOUF, S.-M.KARABERNOU, Y.LATROUS & G.MAZARE, *RAP: Une Perspective de Machine Massivement Parallèle*, Revue Technologies Avancées, Vol.2, N °2, Décembre1992, pp16-21.

[AKTOUF93] Ch.AKTOUF, S.-M.KARABERNOU, Y.LATROUS & G.MAZARE, *RAP: A Fine Grain MIMD Machine* , Dans *Parallel Computing Trends and Applications*, Editeurs. G.R.JOUBERT, D.TRYSTRAM, F.J.PETERS & D.J.EVANS, **PARCO'93**, Elsevier Science B.V., Grenoble, Septembre1993, pp665-670.

[APPEL88] W.A.APPEL, J.R.ELLIS & K.LI, *Real-Time Concurrent Collection on Stock Multiprocessors*, **SIGPLAN'88**, Procs. of the Conf. on Programming Language Design & Implementation, Atlanta, Juin1988, pp11-20.

[APPEL89] A.APPEL, *Simple Generational Garbage Collection and Fast Allocation*, Software Practice and Experience, Vol.19, N°2, Février1989, pp171-183.

[ATHAS86] W.C.ATHAS & C.L.SEITZ, *CANTOR User Report, Version 2.0*, Computer Sc. Dept., California Inst. of Technology, Tech. Report, 5232:TR:1986.

[ATHAS87a] W.C.ATHAS & C.L.SEITZ, *Fine Grain Concurrent Computations*, Computer Sc. Dept., California Inst. of Technology, Tech. Report #5242:TR:1987.

[ATHAS87b] W.C.ATHAS & C.L.SEITZ, *Multicomputers*, Computer Science Departement, California Inst. of Technology, Tech. Report #5244:TR:1987.

[ATHAS88] W.C.ATHAS & C.L.SEITZ, *Multicomputers: Message-Passing Concurrent Computers*, IEEE Computer, Vol.21, N°8, Août1988, pp9-24.

[BAIARDI89] F.BAIARDI & S.ORLANDO, *Strategies for Massively Parallel Implementation of Simulated Annealing*, **PARLE'89**, LNCS, Vol.366, Springer-Verlag, Eindhoven, Netherlands, Juin1989, pp273-287.

[BAILLY87] C.BAILLY, J.-F.CHAILLINE, P.Y.GLOESS, H.-C.FERRI & B.MARCHESIN, *Les Langages Orientés Objets - Concepts, Langages, et Applications*, Cepadues-Editions, 1987.

[BANAWAN78] BANAWAN, *An Evaluation of Local Sharing in Locally Distributed Systems*, Technical Report #87-08-02, University of Washington, Seattle, Août1978.

[BEN_ARI84] M.BEN-ARI, *Algorithms for On-the-Fly Garbage Collection*, ACM Transactions on Programming Languages and Systems, Vol.6, N °3, Juillet1984, pp333-344.

[BERMOND92] J.-C.BERMOND, P.MICHALLON & D.TRYSTRAM, *Broadcasting in Wraparound Meshes With Parallel Monodirectional Links*, Parallel Computing 18 (1992), **PARCO** 625, North-Holland, pp639-648.

[BOBROW80] D.G.BOBROW, *Managing Reetrant Structures Using Refrence Counts*, ACM Transactions on Programming Languages and Systems, Vol.2, N °2, Juillet1980, pp269-273.

[BOKHARI81] S.H.BOKHARI, *On the Mapping Problem*, IEEE Trans. on Computer, Vol.C-30, N °3, Mars1981, pp207-214.

[BURMEISTER91] B.BURMEISTER & K.SUNDERMEYER, *Cooperative Problem Solving Guided by Intentions and Perception*, 3rd Modeling Autonomous Agent in Multi-Agent World Workshop, Kaiserslautern, 1991.

[CHEN90] G.-H.CHEN & J.S.YUR, *A Branch and Bound with Underestimates Algorithm for the Task Assignement Problem with Precedence Constraint*, The 10th Conference on Distributed Systems, Paris, France, Mai1990, pp494-501.

[CHENEY70] C.J.CHENEY, *A Nonrecursive List Compacting Algorithm*, Communications of the ACM, Vol.13, N °11, Novembre1970, pp677-678.

[CLAUZEL94] D.CLAUZEL, *Présentation de l'IBM 9076 Scalable POWERparallel 1*, IBM France, Lyon Ecully, Avril1994.

[COHEN83] J.COHEN & A.NICOLAU, *Comparison of Compacting Algorithms for Garbage Collection*, ACM Transactions on Programming Languages and Systems, Vol.5, N °4, Octobre 1983, pp532-553.

[COHEN90] P.R.COHEN & H.J.LEVESQUE, *Intention is Choice with Commitment*, Artificial Intelligence, Vol.42, 1990, pp213-261

[CORNU88] R.CORNU-EMIEUX, *Réseau Cellulaire Intégré: Etude d'une Architecture pour des Applications de CAO de VLSI*, Thèse Docteur INPG, Septembre1988.

[DALLY84] W.J.DALLY & J.T.KAJIYA, *An Object-Oriented Architecture*, Computer Sc. Dept., California Inst. of Technology, Tech. Report #5168:TR:1984.

[DALLY87] M.J.DALLY, L.CHAO, A.CHIEN, S.HASSOUN, W.HORWAT, J.KAPLAN, P.SONG, B.TOTTY & D.S.WILLS, *Architecture of a Message-Driven Processor*, Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture, June 1987, pp189-196.

[DALLY89] M.J.DALLY & D.S.WILLS, *Universal Mechanisms for Concurrency*, Lecture Notes in Computer Science, N°365, Edited by G.GOOS, J.HARTMANIS, *Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures*, Editeurs: E.ODIJK, M.REM & J.-C.SYRE, **PARLE'89**, Eindhoven, The Netherlands, June 1989, pp19-33, Proceedings - Springer Verlag.

[DELASALLE90] D.DELASSALE, D.TRYSTRAM & D.WENSEK, *Tout ce que Vous Voulez Savoir sur la Connection Machine (Sans Oser le Demander)*, Document Interne LMC, Avril1990.

[DeTREVILLE90] J.DeTREVILLE, *Experience with Concurrent Garbage Collectors for Modula-2+*, Technical Report #64, Digital Equipment Corp. Systems Research Center, Palo Alto, California, Août1990.

[DEUTSCH76] L.P.DEUTSCH & D.G.BOBROW, *An Efficient, Incremental, Automatic Garbage Collector*, Communications of the ACM, Vol.19, N °9, Septembre1976, pp522-526.

[DEVARAKONDA89] M.V.DEVARAKONDA & R.K.IYER, *Predictability of Process Resource Usage: A Measurement-Based Study on UNIX*, IEEE Transactions on Software Engineering, Vol.15, N °12, Décembre1989, pp1579-1586.

[DIJKSTRA78] E.W.DIJKSTRA, L.LAMPORT, A.J.MARTIN, C.S.SCHOLTEN & E.F.M.STEFFENS, *On-the-Fly Garbage Collection: An Exercise in Cooperation* , Communications of the ACM, Vol.21, N °11, Novembre1978, pp966-975.

[EAGER86] D.L.EAGER, E.D.LAZOWSKA & J.ZAHORJAN, *A comparaison of Receiver-Initiated and Sender-Initiated Adaptative Load Sharing*, Performance Evaluation 6, 1, Mars1986, pp53-68.

[EAGER86b] D.L.EAGER, E.D.LAZOWSKA & J.ZAHORJAN, *Adaptative Load Sharing in Homogeneous Distributed Systems*, IEEE Transactions on Software Engineering, Vol.SE-12, N °5, Mai1986, pp662-675.

[FAURE90] B.FAURE & G.MAZARE, *Neural Networks: A Cellular Architecture*, Procs. of the International Conference *"Neural Networks, Biological Computers or Electronic Brains"*, **AFCET**-Entretiens de Lyon, Mars1990.

[FERBER86] J.FERBER, *La Programmation par Acteur (1), l'Ordre par le Dialogue*, <ARTEFACT>, Micro-Systèmes, Mars1986, pp140-144.

[FERBER87] J.FERBER, *Des Objets aux Agents: Une Architecture Stratifiée*, Actes du 6ième Congrès Annuel de Reconnaissance de Formes & Intelligence Artificielle, Antibes, 1987, pp275-286.

[FERBER89] J.FERBER, *Objets et Agents: Une Etude des Structures de Représentation et de Communication en Intelligence Artificielle*, Thèse d'Etat, Université Pierre et Marie Curie (ParisVI), Juin1989.

[FERBER90] J.FERBER & P.CARLE, *Actors and Agents as Reflective Concurrent Objects: A MERING IV Perspective*, Laboratoire Formes et Intelligence Artificielle, Rapport Technique N°18/90, Juin1990.

[FERBER91] J.FERBER, *The Framework of ECO-Problem Solving*, dans *Decentralized Artificial Intelligence II*, Editeurs: Y.Demazeau & J.P.Müller, Elsevier Science Publishers, B.V. (North-Holland), 1991, pp181-195.

[FERGUSON88] D.FERGUSSON, Y.YEMINI & C.NIKOLAOU, *Microeconomic Algorithms for Load Balancing in Distributed Computer Systems*, IEEE, 1988, pp491-499.

[FLAIG87] C.M.FLAIG, *VLSI Mesh Routing Systems*, Computer Sc. Departement, California Inst. of Technology, Tech. Report #5241:TR:1987.

[FLYNN72] M.J.FLYNN, *Some Computer Organisations and Their Effectiveness*, IEEE Transaction on Computers, Vol.21, N°9, Septembre1972.

[GERMAIN89] C.GERMAIN-RENAUD, *Etude des Mécanismes de Communication pour une Machine Massivement Parallèle: MEGA*, Thèse de l'Université Paris-Sud, Décembre1989.

[GHOSH89] J.GHOSH & K.HWANG, *Mapping Neural Networks onto Message-Passing Multicomputers*, Journal of Parallel and Distributed Comp., Vol.6, N °2, 1989, pp291-330.

[GOLDBERG83] A.GOLDBEGR & D.ROBSON, *SMALLTALK-80: The Language and its Implementation*, Addison-Wesley, 1983.

[GOSCINSKI91] A.GOSCINSKI, *Distributed Operating Systems: The Logical Design*, Addison-Wesley, 1991, pp479-514.

[GRUNWALD90] D.C.GRUNWALD, B.A.A.NAZIEF & D.A.REED, *Empirical Comparison of Heuristic Load Distribution in Point-to-Point Multicomputer Networks*, The 5th Distributed Memory Computing Conference, Charleston, Avril1990, pp984-993.

[HAILPERIN88] M.HAILPERIN, *Load Balancing for Massively-Parallel Soft-Real-Time Systems*, Tech. Report #STAN-CS-88-1222 (aussi numéroté #KSL-88-62), Department of Computer Science, Stanford University, Septembre1988.

[HANDLERS77] W.HANDLERS, *The Impact of Classification Schemes on Computer Architecture*, Procs. of the International Conference on Parallel Processing, New York, Août1977, pp7-15.

[HASSAS92] S.HASSAS, *GMAL: Un modèle d'Acteurs Réflexif pour la Conception de Systèmes d'Intelligence Artificielle Distribuée*, Thèse Doctorat Lyon-1, 21 Septembre1992.

[HAYES91] B.HAYES, *Using Key Object Opportunism to Collect Old Objects*, Dans ACM SIGPLAN 1991 Conf. on Object-Oriented Programming Systems, Languages and Applications (**OOPSLA'91**), ACM Press, Pheonix, Arizona, Octobre1991, pp33-46.

[HEMERY94] F.HEMERY, *Etude de la Répartition Dynamique d'Activités sur Architectures Décentralisées*, Thèse de l'Université des Sciences et Technologies de Lille, Juin1994.

[HERLIHY93] M.HERLIHY & J.E.MOSS, *Non-Blocking Garbage Collection for Multiprocessors*, Tech. Report #CRL90/9, Digital Equipt. Corp., Septembre1993.

[HEWITT77] C.HEWITT & H.BAKER, *Actors and Continuous Functionals*, **IFIP** Working Conference on Formal Description of Programming Concepts, St Andrews, Canada, 1977.

[HEWITT82] C.HEWITT, P.de JONG, *Open Systems*, Artificial Intelligence Laboratory, MIT, Technical Report, A.I.Memo N°691, Décembre1982.

[HEWITT83] C.HEWITT & G.BARBER, *Semantic Support for Work in Organization*, Artificial Intelligence Laboratory, MIT, Technical Report, A.I. Memo N°719, Avril1983.

[HILLIS85] W.D.HELLIS, *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.

[HOUCK92] C.HOUCK & G.AGHA, *HAL: A High-level Actor Language and Its Distributed Implementation*, 21st International Conference on Parallel Processing (**ICPP'92**), St Charles, IL, Vol.II, Août1992, pp158-165.

[ITO86] N.ITO, M.SATO, M.KISHI, E.KUNO & K.ROKUSAWA, *The Architecture & Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D*, The 13th Annual Symposium on ComputerArchitecture, 1986.

[KAFURA90] D.KAFURA, D.WASHABAUGH & J.NELSON, *Garbage Collection of Actors*, **ECOOP/OOPSLA'90** Proceedings, European Conference on Object-Oriented Programming, Editeur: N. MEYROWITZ, ACM Press, 1990, pp126-134.

[KALE88] L.V.KALE, *Comparing the Performance of Two Dynamic Load Distribution Methods*, Dans International Conference on Parallel Processing, 1988.

[KARABERNOU93] S.-M.KARABERNOU, *Conception et Réalisation d'un Processeur pour une Architecture Cellulaire Massivement Parallèle Intégrée*, Thèse Docteur INPG, Juillet1993.

[KIM92] W.KIM & G.AGHA, *Compilation of a Highly Parallel Actor-Based Language*, The Fifth Workshop on Languages and Compilers for Parallel Computing, YALEU/DCS/RR-915, New Haven, CT, Septembre92, pp1-12.

[KITAJIMA93] J.P.KITAJIMA, C.TRON & B.PLATEAU, *ALPES: A Tool for the Performance Evaluation of Parallel Programs*, Dans: *Environments and Tools for Parallel Scientific Computing*, Editeurs: J.J.Dongarra & B.Tourancheau, Amsterdam, The Netherlands, 1993, pp213-228.

[KNIGHT9?] K.KNIGHT, *Are Many Reactive Agents Better Than a Few Deliberative Ones?* , Distributed AI, 199?, pp432-437.

[KORNFELD79] W.A.KORNFELD, *ETHER: A Parallel Problem Solving System*, Proc. of the 6th **IJCAI**, 1979.

[KORNFELD80] W.A.KORNFELD & C.HEWITT, *The Scientific Community Metaphor*, AILab.Memo N °641, MIT, Cambridge, 1980.

[KRAKOWIAK85] S.KRAKOWIAK, *Introduction à la Programmation par Objets*, **CORNAFION**, Mai 1985, Version 1, pp1-25.

[KRAVITZ87] S.A.KRAVITZ & R.A.RUTENBAR, *Placement by Simulated Annealing on a Multiprocessor*, IEEE Trans. on Computer-Aided Design, Vol.CAD-6, N °4, Juillet1987, pp534-549.

[KRUEGER87a] Ph.KRUGER & M.LIVNY, *When is the Best Load Sharing Algorithm a Load Balancing Algorithm?* , Computer Science Technical Report #694, Août1987.

[KRUEGER87b] Ph.KRUGER & M.LIVNY, *Load Balancing, Load Sharing and Performance in Distributed Systems*, Computer Science Technical Report #700, Août1987.

[KRUEGER88] Ph.KRUGER & M.LIVNY, *A Comparaison of Preemptive and Non-Preemptive Load Distributing*, The 8th International Conference on Distributed Computing Systems, Juin1988, California, pp123-130.

[KUCHEN91] H.KUCHEN & A.WAGNER, *Comparison of Dynamic Load Balancing Startegies*, Dans: *Parallel & Distributed Processing*, Editeur: K.BOUYANOV, North-Holland, Mars1991, pp303-314.

[KUNG77] H.T.KUNG & S.W.SONG, *An Efficient Parallel Garbage Collection System and its Correctness Proof*, Dept. of Computer Science, Carnegie-Mellon University, Septembre1977.

[KUNZ91] T.KUNZ, *The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme*, IEEE Transactions on Software Engineering, Vol.17, N °7, Juillet1991, pp725-730.

[LADIN92] R.LADIN & B.LISKOV, *Garbage Collection of a Distributed Heap*, The 12th Inter. Conf. on Distributed Computing Systems, Mokohama, Juin1992, pp708-715.

[LANGENDOEN92] K.G.LANGENDOEN, H.L.MULLER & W.G.VREE, *Memory Management for Parallel Tasks in Shared Memory*, LNCS637, **IWMM'92**, Inter. Workshop on Memory Management, France, Septembre1992, pp165-178.

[LATROUS88] Y.LATROUS & N.TALBI, *PICMOS: Un Logiciel de Placement et d'Interconnexion dans les Circuits Intégrés VLSI suivant l'Approche Building-Blocks*, Mémoire d'Ingénieur, INI-Alger (ex. CERI), Septembre1988.

[LATROUS94a] Y.LATROUS & G.MAZARE, *Code Distribution in A Parallel Fine Grain Machine*, First International Workshop on Parallel Processing, Bangalore, India, 26-30Décembre1994, *à paraître*.

[LATROUS94b] Y.LATROUS & G.MAZARE, *Integrating Dynamic Mechanisms in a Parallel Fine Grain Machine*, **ICM'94**, Procs. 6th International Conference on Microelectronics, Istanbul, Turquie, Septembre1994, pp112-120.

[LATROUS95] Y.LATROUS & G.MAZARE, *Distributing Code in a Parallel Fine Grain Machine Using the Actor Model*, Procs. of the 3rd Euromicro Workshop on Parallel and Distributed Processing, San Remo, Italie, 25-27Janvier1995, *àparaître* .

[LATTARD89] D.LATTARD, *Architecture Massivement Parallèle: Un Réseau de Cellules Intégré pour la Reconstruction d'Images*, Thèse Docteur INPG, Novembre1989.

[LeSERGENT92] T.LeSERGENT & B.BERTHOMIEV, *Incremental Multi-threaded Garbage Collection on Virtually Shared Memory Architectures*, LNCS637, **IWMM'92**, Inter. Workshop on Memory Management, France, Septembre1992, pp179-199.

[LESTER??] D.R.LESTER, *An Efficient Distributed Garbage Collection Algorithm*, LNCS365, pp207-223.

[LIEBERMAN81] H.LIEBERMAN, *A Preview of ACT1*, Artificial Intelligence Laboratory, MIT, Technical Report, A.I. Memo N°625, 1981.

[LIEBERMAN83] H.LIEBERMAN & C.HEWITT, *A Real-Time Garbage Collector Based on the Lifetimes of Objects*, Communications of the ACM, Vol.26, N °6, Juin1983, pp419-429.

[LIN87] F.C.LIN & R.M.KELLER, *The Gradient Model Load Balancing Method*, IEEE Transactions on Software Engineering, Vol. SE-13, N°1, Janvier1987, pp32-38.

[MANCINI88] L.V.MANCINI & S.K.SHRIVASTAVA, *Fault-Tolerant Reference Counting for Garbage Collection in Distributed Systems*, Technical Report #260, University of Newcastle upon Tyne, Juin1988.

[MASINI90] G.MASINI, A.NAPOLI, D.COLNET, D.LEONARD, K.TOMBE, *Les Langages à Objets, Langages de classes, Langages de Frames, Langages d'Acteurs*, InterEditions, 1990.

[MAY93] M.D.MAY, R.M.SHEPHERD & P.W.THOMPSON, *The T9000 Communications Architecture*, dans *Networks, Routers and Transputers: Function, Performance, and Applications*, Editeurs: M.D.MAY, P.W.THOMPSON & P.H.WELCH, Inmos Limited, SGS-THOMSON MicroElectronics, Février1993, pp15-38.

[MEHRA93] P.MEHRA & B.W.WAH, *Automated Learning of Worload Measures for Load Balancing on a Distributed System*, Procs. of the International Conference on Parallel Processing, Vol.III, Août1993, pp263-270.

[MEYER90] B.MEYER, *Conception et Programmation par Objets, Pour du Logiciel de Qualité*, InterEditions, 1990.

[MICHALLON93] P.MICHALLON, *Communications Globales dans les Grilles Toriques*, Thèse Docteur INPG, Février1993.

[MINSKY75] M.MINSKY, *A Framework for Representing Knowledge*, Dans: *The Psychology of Computer Vision*, Editeur: P.WINSTON, McGrawHill, 1975, pp221-281.

[MOOIJ89] W.G.P.MOOIJ & A.LIGTENBERG *Architecture of a Communication Network Processor*, LNCS N°365, Editeur: G.GOOS, J.HARTMANIS, **PARLE'89**, *Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures*, Editors: E.ODIJK, M.REM, J.-C.SYRE, Eindhoven, Proc. - Springer Verlag, Juin1989, pp238-250.

[MOON84] A.D.MOON, *Garbage Collection in a Large Lisp System*, ACM Symposium on LISP & Functional Programming, Texas, Août1984, pp235-246.

[NETTLES93] S.NETTLES, J.O'TOOLE & D.GIFFORD, *Concurrent Garbage Collection of Persistent Heap*, Tech. Report #CMU-CS-93-137 à Carnegie Mellon Univ., existe aussi au MIT sous #MIT-LCS-TR-569, avril1993.

[NGAI87] J.N.NGAI & C.L.SEITZ, *A Framework for Adaptative Routing*, Computer Sc. Dept., California Inst. of Technology, Tech. Report #5246:TR:1987.

[NI89] L.M.NI, Y.LAN & A.-H.ESFAHANIAN, *A VLSI Router Design for Hypercube Multiprocessors*, INTEGRATION, The VLSI Journal 7 (1989), Elsevier Science Publishers B.V., pp103-125.

[NUGENT88] S.F.NUGENT, *The iPSC/2 Direct-Connect Communication Technology*, The 3rd Conf. on Hypercube Concurrent Computers and Applications, 1988.

[OBJOIS88] Ph.OBJOIS, *Réseau de Cellules Intégré: Mécanisme de Communication Inter-Cellulaire et Application à la Simulation Logique*, Thèse Docteur INPG, Septembre1988.

[PAYAN91] E.PAYAN, *Etude d'une Architecture, Celllulaire Programmable: Définition Fonctionnelle et Méthodologie de Programmation*, Thèse Docteur INPG, Juin1991.

[PIERCE94] P.PIERCE & G.REGNIER, *The Paragon Implementation of th NX Message Passing Interface*, Scalabale High-Performance Computing Conference, Knoxville, Tennessee, Mai1994, pp184-190.

[QUINN87] M.J.QUINN, *Designing Efficient Algorithms for Parallel Computers*, Mc GrawHill Computer Science Series, 1987.

[ROCH92] J.L.ROCH, A.VERMEERBERGEN & G.VILLARD, *Cost Prediction for Load-Balancing: Application to Algebraic Computations*, Parallel Processing, **CONPAR'92**, VAPP V, Lyon, pp467-478.

[ROSS90] A.ROSS & B.McMILLIN, *Experimental Comparison of Bidding and Drafting Load Sharing Protocols*, The 5th Distributed Memory Computing Conference, Procs. Vol.II, Charleston, Avril1990, pp968-974.

[RUBINI92] P.RUBINI, *Programmation d'une Architecture Cellulaire Massivement Parallèle*, Thèse Docteur INPG, Juin1992.

[RUSSEL89] S.J.RUSSEL, *Execution Architectures and Compilation*, **IJCAI** Conference, 1989, pp15-20.

[SAAD89] Y.SAAD, M.H.SCHULTZ, *Data Communication in Parallel Architectures*, Parallel Computing 11,, Elsevier Science Publishers B.V. (North-Holland), 1989, pp131-150.

[SALETORE90] V.A.SALETORE, *A Distributed and Adaptative Dynamic Load Balancing Scheme for Parallel Processing of Medium-Grain Tasks*, The 5th Distributed Memory Computing Conference, Procs. VolII (*Architecture, Software Tools & Other General Issues*), South Carolina, 1990, pp994-999.

[SANSONNET91] J.P.SANSONNET, *Concepts d'Architectures Avancées*, Cours DEA Informatique, LRI Paris XI-Orsay, 1990-1991.

[SARGEANT86] J.SARGEANT, *Load Balancing, Locality and Parallelism Control in Fine Grain Parallel Machines*, Technical Report Series, UCMS-86-11-5, 1986.

[**SCHÖNAUER93**] W.SCHÖNAUER & H.HAFNER, *Supercomputer Architectures and their Bottlenecks*, **PARCO'93**, Dans: *Parallel Computing: Trends and Applications* , Editeurs: G.R.JOUBERT, D.TRYSTRAM, F.J.PETERS & D.J.EVANS, Elsevier Science B.V., Grenoble, Septembre1993, pp411-417.

[SEITZ84] C.L.SEITZ, *Concurrent VLSI Architectures*, IEEE Transaction on Computers Vol. C-33, N°12, Décembre1984, pp1247-1265.

[SHARMA91] R.SHARMA & M.L.SOFFA, *Parallel Generational Garbage Collection*, Procs. of the **OOPSLA'91** Conf. (SIGPLAN Notices, Vol.26, N °11), Arizona, Octobre1991, pp16-32.

[SMITH80] R.G.SMITH, *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver*, IEEE Transactions on Computer, Vol.C-29,N °12, Décembre1980, pp1104-1113.

[SOHMA85] Y.SOHMA, K.SATOH, K.KUMON, H.MASUZAWA & A.ITASHIKI, *A New Parallel Inference Mechanism Based on Sequential Processing*, Dans: *Fifth Generation Computer Architecture*, Editeur: J.WOODS, North Holland, 1985.

[STANKOVIC84a] J.A.STANKOVIC, *Simulations of Three Adaptative, Decentralized Controlled, Job Scheduling Algorithms*, Computer Networks 8, 1984, pp199-217.

[STANKOVIC84b] J.A.STANKOVIC & I.S.SIDHU, *An Adaptative Bidding Algorithm for Processe, Clusters and Distributed Groups*, Procs. of Inter. Conf. on Parallel Processing, Août1984, pp49-59.

[STUNKEL94] C.B.STUNKEL, D.G.SHEA, D.G.GRICE, P.H.HOCHSCHILD & M.TSAO, *The SP1 High-Performance Switch*, Procs. of the Scalable High-Performance Computing Conference, Knoxville, Tennessee, Mai1994, pp150-157.

[TAKKELLA94] S.TAKKELLA & S.SEIDEL, *Complete Exchange and Broadcast Algorithms for Meshes*, Scalabale High-Performance Computing Conference, Knoxville, Tennessee, Mai1994, pp422-428.

[TALBI93] E.-G.TALBI, *Allocation de Processus sur les Architectures Parallèles à Mémoire Distribuée*, Thèse Docteur INPG, Mai1993.

[TMC92] Thinking Machines Corporation, *CM5*, Technical Summary, Cambridge, Massachusetts, Janvier1992

[TOUZENE92] A.TOUZENE, *Résolution des Modèles Markoviens sur Machines à Mémoires Distribuées*, Thèse Docteur INPG, Septembre1992.

[TREW91] A.TREW & G.WILSON, *Past, Present, Parallel Survey of Available Parallel Computing Systems*, Springer-Verlag, 1991.

[TRON94] C.TRON & B.PLATEAU, *Modelling of Communication Contention in Multiprocessors*, Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation - Vienna, Austria, Editeurs: G.HARING & G.KOTSIS, Springer-Verlag, Berlin, LNCS794, Mai1994, pp406-424.

[TSAI94] Y.-J.TSAI & P.K.McKINLEY, *An Extended Dominating Node Approach to Collective Communication in All-Port Wormhole-Routed 2D Meshes*, Scalable High-Performance Computing Conference, Knoxville, Tennessee, Mai1994, pp199-206.

[TUCKER88] L.W.TUCKER & G.G.ROBERTSON, *Architecture and Applications of the Connection Machine*, IEEE Computer, Vol.21, N °8, Août1988, pp26-38.

[UNGAR84] D.M.UNGAR, *Generation Scavenging: A Non-Disruptive High-Performance Storage Reclamation*, **ACM SIGSOFT/SIGPLAN**, Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, Avril1984, pp157-167.

[UNGAR88] D.UNGAR & F.JACKSON, *Tenuring Policies for Generation-Based Storage Reclamation*, Procs. of the Conference on Object-Oriented Programming Systems, Languages, and Applications, **ACM SIGPLAN** Not.23, 11, Nov.1988, San Diego, Cal., Septembre1988, pp1-17.

[WAILLE91] Ph.WAILLE, *Architectures Parallèles à Connectique Programmable: Reconfiguration et Routage* , Thèse Docteur INPG, Septembre1991.

[WANG85] Y.T.WANG & J.T.MORIS, *Load Sharing in Distributed Systems*, IEEE Transactions on Computers, Vol.C-34, N°3, Mars1985, pp204-217.

[WEINREB80] D.WEINREB & D.MOON, *FLAVORS: Message Passing in the LISP Machine*, MIT, A.I.Memo N °602, Novembre1980.

[XU93] J.XU & K.HWANG, *Heuristic Methods for Dynamic Load Balancing in a Message-Passing Multicomputer*, Journal of Parallel and Distributed Computing, Vol.18, N °1, Mai1993, pp1-13.

[YUM81] T.YUM, *The Design & Analysis of a Semidynamic Deterministic Routing Rule*, IEEE Transactions on Communication, Vol.COM-29, Avril1981, pp498-504.

[ZHOU88] S.ZHOU & D.FERRARI, *An Empirical Investigation of Load Indices for Load-Balancing Applications*, Dans: **PERFORMANCE'87**, Editeurs: P.J.Courtois & G.Latouche, Elsevier Science Publishers B.V. (NorthHolland), 1988.

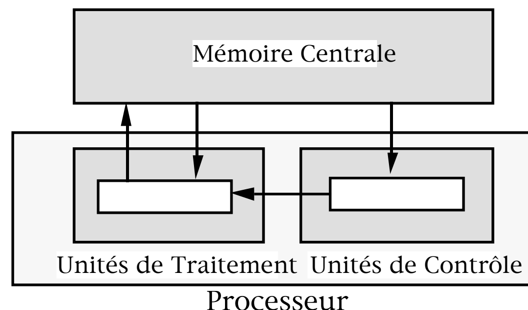
[ZORN90] B.ZORN, *Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection*, ACM Conference on LISP and Functional Programming, Nice, France, Juin1990, pp87-98.

Annexe 1
Classification des machines parallèles

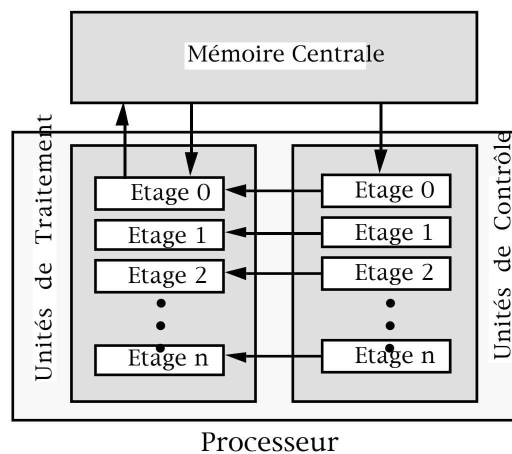
ANNEXE I: CLASSIFICATION DES MACHINES PARALLÈLES

Nul ne peut échapper à la classification traditionnelle établie par [FLYNN72]. Cette classification est basée sur le séquencement des données et des instructions au niveau des processeurs. Cette classification compte les catégories d'architectures suivantes:

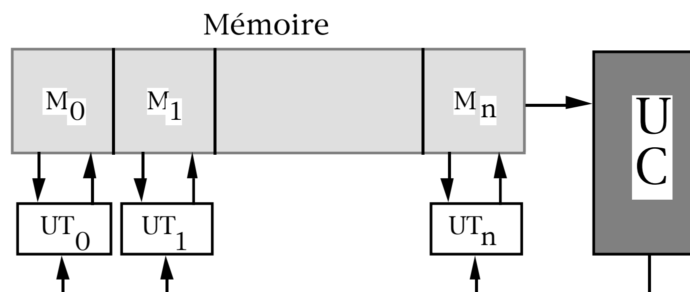
- Les architectures **SISD** (Single Instruction Single Data): c'est la machine de Von Neumann traditionnelle; les instructions (lues à partir d'une mémoire vive) sont exécutées l'une à la suite de l'autre.



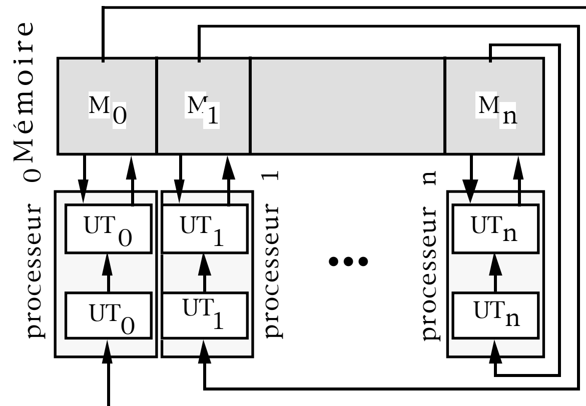
- Les architectures **MISD** (Multiple Instruction Single Data): ces machines sont plus connues sous le nom d'architectures *pipeline*. C'est le travail à la chaîne : plusieurs données (de même type en règle générale) sont introduites dans le circuit pour subir une série de traitements. L'efficacité de cette approche tient au recouvrement réalisé entre les différentes tâches (chaque tâche étant assignée à un étage du pipeline). Les machines dites VLIW (Very Large Instruction Word) sont une extension de ce principe : il y a recouvrement entre plusieurs instructions par mot mémoire [ZERROUK9?].



• Les architectures **SIMD** (Single Instruction Multiple Data): plusieurs unités de traitement (UT) sont supervisées par la même unité de contrôle (UC). Toutes les UT sont donc synchrones. L'UC diffuse une même instruction à l'ensembles des UT. Chaque UT opère sur des données distinctes, en provenance (éventuellement) d'un banc mémoire distinct. La mémoire est partagée entre les différentes UT.



• Les architectures **MIMD** (Multiple Instruction Multiple Data): la dernière catégorie est caractérisée par une duplication de plusieurs processeurs. On peut exécuter autant d'instructions qu'il y a de processeurs. La mémoire peut être soit partagée soit distribuée.



Cette classification ne tient compte ni du réseau de communication, ni du grain des applications à exécuter sur ces architectures. La topologie des machines parallèles définit, pour chacune, des schémas de communications plus ou moins bien adaptés. Les réseaux de communication en une grille, un hypercube ou une pyramide ne s'adressent pas tous aux mêmes applications. La topologie du réseau de communication, fixe ou reconfigurable, a une incidence importante sur les stratégies de routage et les modèles de programmation [WAILLE91].

De même, les machines systoliques, qui peuvent être considérées comme des automates ou des processeurs spécialisés, ne peuvent être classées dans aucune des catégories citées.

Pourtant, malgré toutes les constations que nous venons de faire, il n'y jusqu'à présent aucune autre classification qui ait reçu un consensus de la part de la communauté scientifique[39] !

---

[39] Il existe d'autres classifications, notamment celle de Händler [HANDLER77], mais qui ne sont pas très répandues dans le monde scientifique.

Annexe 2
Environnement de simulation

ANNEXEII: ENVIRONNEMENT DE SIMULATION

II.1. L'assembleur

Dans la version statique du projet R.C.A.I, lorsque les processeurs devaient recevoir du code plus ou moins différent, on utilisait des directives "d'assemblage conditionnel" pour développer le code pour l'ensemble des processeurs. Une directive d'assemblage conditionnel servait alors à spécifier un groupe de cellules homogènes (codées de la même manière), par le biais d'expressions arithmétiques des références relatives ou absolues des cellules.

Contrairement à la version statique, les cellules reçoivent toutes du code différent. De ce fait, des directives d'assemblage conditionnel sont tout à fait inutiles. Néanmoins, nous avons conservé la possibilité de charger un même code soit sur l'ensemble du réseau (adresse cellule (-1,-1) [40]), soit sur une même ligne (adresse cellule (x,-1)) soit sur une même colonne (adresse cellule (-1,x)). Cette approche peut être utile dans le cas de la programmation des cellules d'entrées/sorties par exemple, ou encore dans le cas de l'intégration d'un noyau système.

La directive d'assemblage s'écrit:

RapAs -d <répertoire> -c lig col <nom_fichier>

L'option (**-d**) indique le répertoire dans lequel l'assembleur cherche et sauvegarde ses fichiers d'entrée et de résultats. L'option (**-c**) spécifie la cellule (lig, col) réceptrice du code courant. <nom_fichier> est le fichier à assembler.

Chaque ligne du code assembleur peut représenter:

- un simple commentaire,
- une directive d'assemblage,
- ou une instruction arithmétique, logique, de branchement ou de communication.

---

[40] Une adresse de cellule se présente sous la forme (ligne, colonne).

II.1.1. Commentaires

Un commentaire est une suite de caractères alphanumériques, qui débute par le caractère spécial **;** et se termine par un retour à la ligne.

;<suite_caractères_alphanumériques>

II.1.2. Directives d'assemblage

Nous disposons de trois directives d'assemblage :

ORG <entier>

Cette directive indique une nouvelle valeur pour le compteur de génération de code. La valeur par défaut de ce compteur est 0.

<symbole> **:** **EQU** {<entier>, <référence>, <adresse>}

Cette directive spécifie des constantes symboliques, manipulées dans le reste du code.

[<symbole> **:**] **DS** <entier>

Cette directive réserve le nombre de positions mémoires spécifié. Le compteur de génération est incrémenté d'autant de fois que de mots réservés. La zone ainsi délimitée est initialisée à la valeur zéro. Elle est référencée par le symbole attenant.

[<symbole> **:**] **DC** {<entier>, <référence>, <adresse>}

Cette directive initialise une position mémoire à la valeur indiquée. Cette valeur peut être soit une valeur entière, une adresse relative d'une autre cellule ou une adresse en mémoire.

II.1.3. Instructions

Chaque instruction se présente sous la forme suivante:

[<label> **:**] <mnémonique> [<mode_adressage> <valeur>]

Dans cette définition, le champ *<label>* est une adresse de branchement. Les mnémoniques correspondent au jeu d'instructions de R.C.A.I. Quant aux modes d'adressage, ce sont ceux décrits dans le premier chapitre, lors de la présentation de la machine R.C.A.I.

Une instruction peut être suivie d'un commentaire.

Afin d'illustrer l'utilisation de l'assembleur, nous présentons dans ce qui suit l'algorithme du crible d'Erathostène ainsi que le code objet généré par cet outil (figuresII.1 et II.2).

```
; Programme cellule 0,1 (Crible d'Erathostène):

          ORG    0

          DS     2
CELL:     DC     0:0
          DC     SIEVE
ADVANCE:  DS     1
PPOST:    DC     0:0
          DC     PRIME
UNKNOWN:  DS     1
SIEVE:    DC     2
PRIME:    DC     0
INIT:     DC     0:0
          DC     CELL
          DC     0

NEXT:     GET    PRIME
          BMI    FIN          ; Réception de la valeur -1
          STA    ADVANCE      ; signal de la fin de traitement
          STA    UNKNOWN
          DIV    SIEVE
          MUL    SIEVE
          SUB    ADVANCE
          BEQ    NEXT
          LDA    CELL
          BNE    FORWARD
          FIND   CELL         ; Recherche d'une cellule libre
          MOVE   CELL         ; Duplication du code
          LDA    CELL
          STA    PPOST
          STA    INIT
          SEND   INIT
          SEND   CELL
          BRA    NEXT
FORWARD:  SEND   PPOST        ; Renvoyer la valeur non reconnue
          BRA    NEXT

FIN:      LDA    -1
          STA    UNKNOWN

          LDA    PPOST
          BEQ    FINISH       ; Si dernier élément de la chaîne

          SEND   PPOST        ; Envoyer la valeur -1 pour
                              ; terminer le traitement
          OUT    SIEVE

FINISH:   END
```

Fig.II.1. Listing de l'exemple du crible d'Erathostène en assembleur (cf.paragraphe2.3.5).

```
1         ;Validité du code objet (1: valide, 0: non valide)
1  0      ;Adresse absolue de la cellule réceptrice (lig,col)
11        ;Nombre de symboles
;TABLE DES SYMBOLES:   Symbole   Adresse  Rang
                       FORWARD    49       0
                          CELL     2       1
                       UNKNOWN     7       2
                          INIT    10       3
                          NEXT    13       4
                         SIEVE     8       5
                         PRIME     9       6
                         PPOST     5       7
                        FINISH    63       8
                           FIN    53      10
                       ADVANCE     4      15

12        ;Nombre de variables
;    Adresse      Symbole   Rang   Taille   Type    Valeur
                                            (0 ou   (entière ou
                                            1)       référence)
    0           __NONE__      9      2      0        2
    2               CELL      1      1      1        0       0
    3           __NONE__     11      1      0        8
    4            ADVANCE     15      1      0        1
    5              PPOST      7      1      1        0       0
    6           __NONE__     12      1      0        8
    7            UNKNOWN      2      1      0        1
    8              SIEVE      5      1      0        2
    9              PRIME      6      1      0        0
   10               INIT      3      1      1        0       0
   11           __NONE__     13      1      0        2
   12           __NONE__     14      1      0        0

;CODE:
;  PC    Mnemo.  Permission  Zone  Opérande
   13    286     3            2     6
   15    309     3            3    10
   17    263     3            2    15
   19    263     3            2     2
   21    294     3            2     5
   23    293     3            2     5
   25    265     3            2    15
   27    305     3            3     4
...
   39    263     3            2     7
   41    263     3            2     3
   43    287     3            2     3
   45    287     3            2     1
   47    299     3            3     4
   49    287     3            2     7
   51    299     3            3     4
   53    261     2            0    -1
   55    263     3            2     2
   57    261     3            2     7
   59    305     3            3     8
   61    287     3            2     7
   63    319     0           -1                 ; Fin du code
```

Fig.II.2.Listing du code objet généré par l'assembleur pour l'exemple de l'algorithme du Crible d'Erathostène.

II.2. L'éditeur de liens

La phase d'assemblage est terminée par la génération d'un code intermédiaire, dans lequel figure, pour chaque fichier, une table des symboles.

C'est à partir de ces tables que les références croisées sont résolues par l'éditeur de liens.

La directive d'éditions de liens est de la forme:

RapLink **-p** **-o** <nom_fichier_code> **-d** <répertoire>
-h <i> **-v** <j> **-m** <taille_mémoire> <fichier1>.obj <fichier2>.obj...

L'option (**-p**) indique à l'éditeur de liens RapLink de fournir un état de progression de cette opération. L'option (**-o**) spécifie le fichier résultat du code exécutable. L'option (**-d**) détermine le répertoire de travail. Les deux options suivantes (**-h**) et (**-v**) font état de la taille du réseau sur lequel sera placé ce code, respectivement le nombre de colonnes et le nombre de lignes. Finalement, l'option (**-m**) définit la taille de la mémoire locale de chaque processeur. On fournit par la suite l'ensemble des fichiers à traiter.

L'éditeur de liens génère, en combinant l'ensemble des fichiers objets de l'application, un fichier unique du code exécutable de R.C.A.I (figureII.3). Les instructions sont découpées en plusieurs groupes suivant leur fonction. Il y a principalement six zones d'activité[41]:

- zone d'inactivité, lorsque la cellule est libre,
- zone de calcul, pour l'ensemble des instructions arithmétiques, logiques et de branchement,
- zone d'envoi de message (instructions SEND et PUT),
- zone de réception de message (instructions GET et TRY),
- zone de recherche de cellules libres (instruction FIND), et
- zone de copie de code (instructions MOVE et COPY).

Lors de la génération du code exécutable (figureII.3), chaque groupe d'instructions est identifié par un code de zone, qui est exploité lors de l'exécution pour informer sur l'activité du réseau (cf.§5.1.3.3.).

[41]Une zone d'activité correspond au type de l'instruction en cours d'exécution.

```
8501                    ;Taille du code
8                       ;Taille du mot mémoire
0                       ;Version du processeur
8 3                     ;Taille du réseau
14                      ;Nombre de symboles
ADVANCE                 ;SYMBOLES
FIN
FINISH
PPOST
PRIME
SIEVE
NEXT
INIT
UNKNOWN
CELL
FORWARD
VALUE
DATA_OUT
LOOP
0 0 4 -1 0 5       ;Cellule active, contient du code (@ 5).
0 1 4 -1 0 -1      ;Cellule inactive, pas de code.
0 2 4 -1 0 -1      ;Cellule inactive, pas de code.
1 0 4 -1 0 13      ;Cellule active, contient du code (@ 13).
1 1 4 -1 0 -1      ;Cellules inactives, pas de code.
...
7 2 4 -1 0 -1
;Code de la cell(0,0):(Lig Col Zone Permission Adr Code)
0 0 0 1 0 5          ;Point d'entrée
0 0 0 1 1 22         ;Taille du code
0 0 0 1 2 1          ;Zone de donnée
...
0 0 1 1 5 38         ;Zone de code
...
0 0 4 1 21 129       ;Fin du code
0 0 4 0 22 0         ;Suite de la mémoire
...
0 0 4 0 255 0        ;Fin de la mémoire de la cellule (0, 0)
;Code de la cellule (0,1), 4 premiers mots car cellule vide.
0 1 4 0 0 0
0 1 4 0 1 0
0 1 4 0 2 0
0 1 4 0 3 0
...
;Code de la cellule (1, 0), correspond au précédent listing.
1 0 0 1 0 13         ;Point d'entrée
1 0 0 1 1 66         ;Taille du code
1 0 0 1 2 0          ;Zone de données
...
1 0 3 1 13 63        ;Zone de code
...
1 0 4 1 65 129       ;Fin du code
1 0 4 0 66 0         ;Reste de la mémoire vide
...
1 0 4 0 255 0        ;Fin de la cellule (1, 0).
1 1 4 0 0 0          ;Reste des cellules vides.
...
7 2 4 0 0 0
...
7 2 4 0 3 0          ;Fin du code exécutable
```

Fig.II.3. Listing du code exécutable pour l'exemple précédent.