Méthodologie de modélisation pour l'évaluation des performances des architectures parallèles

Jean-Pierre Prost

THESE

présentée par

Jean-Pierre PROST

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(*arrêté ministériel du 23 Novembre 1988*)

Spécialité: Informatique

==========

METHODOLOGIE DE MODELISATION POUR L'EVALUATION DES PERFORMANCES DES ARCHITECTURES PARALLELES

==========

Date de soutenance: 18 Septembre 1989

Jury:	**Jacques**	**MOSSIERE**	**(Président)**
	Gian-Franco	**BALBO**	
	Monique	**BECKER**	**(Directeur de Thèse)**
	Enrico	**CLEMENTI**	
	Paul	**FEAUTRIER**	**(Rapporteur)**
	Guy	**VIDAL NAQUET**	**(Rapporteur)**

Thèse préparée au sein du Laboratoire de Génie Informatique (Unité Génie Matériel)

Remerciements

Je tiens tout d'abord à exprimer toute ma gratitude à Madame Monique Becker, pour avoir dirigé cette thèse avec tant de dévouement et de gentillesse. Les nombreux conseils qu'elle a su me prodiguer ont grandement contribué à la qualité de mon travail.

Je remercie vivement Monsieur Jacques Mossière d'avoir bien voulu me faire l'honneur de présider le jury de cette thèse.

Je suis très reconnaissant envers Messieurs Paul Feautrier et Guy Vidal Naquet, rapporteurs de cette thèse, dont les remarques judicieuses m'ont permis d'enrichir ce rapport.

Je voudrais exprimer des remerciements tout particuliers à Monsieur Enrico Clementi, pour m'avoir accueilli pendant un an dans le département de calcul scientifique et d'ingénierie de IBM Kingston, dont il est responsable, et pour avoir accepté d'être membre du jury.

Je remercie également Monsieur Gian-Franco Balbo d'avoir bien voulu appartenir au jury.

Bien sûr, ce travail de thèse n'aurait pu être mené à bien sans la précieuse collaboration de nombreuses personnes pour les différentes études de cas, à qui j'adresse ici ma plus sincère gratitude:

- pour la modélisation des réseaux point-à-point de systèmes HP3000, mes remerciements vont notamment à Mademoiselle Josée Auber et Monsieur Jean-Charles Dubuis, responsables du projet pour Hewlett-Packard, et à Monsieur Marcel Cottin, avec qui je formais un binôme au sein de l'E.N.S.I.M.A.G., et qui fut un si talentueux collaborateur,

- pour les mesures de performances sur le réseau local d'IBM PC, j'adresse toute ma reconnaissance à Messieurs Thierry Falgon et Luc Simonet pour le travail important qu'ils ont fourni dans le cadre de leur projet d'année spéciale à l'E.N.S.I.M.A.G., et dont j'ai pu largement bénéficier, ainsi qu'à IBM France pour nous avoir prêté tout le support matériel nécessaire,

- pour la modélisation du calculateur scientifique FPS264, je remercie plus particulièrement Messieurs Christophe Dekoninck et Bruno Verrier pour le travail considérable et de qualité qu'ils ont su produire lors de leur projet de 3ème année à l'E.N.S.I.M.A.G., Messieurs Paul Caspi, Gérard Florin et Stéphane Natkin pour leur assistance théorique sur les réseaux de Petri stochastiques, ainsi que Messieurs Alain Nemoz et Yves Siret pour la mise à disposition gracieuse du FPS264 du Centre de Calcul Interuniversitaire de Grenoble,

- pour la modélisation du système ICAP/FPS264, ma gratitude est adressée surtout à Monsieur John Detrich, superviseur de mon projet pour IBM, à Messieurs Serge Cauberghs, Dave Folsom et Doug Logan pour l'aide qu'ils m'ont apportée lors de la conception du modèle, à Madame Raphaèle Herbin et Messieurs Luigi Brochard, Stéphane Gerbi, et Vijay Sonnad pour m'avoir fourni le code source des programmes d'application utilisés pour la validation du modèle, à Monsieur Michele Re, pour m'avoir initié aux outils graphiques disponibles au sein du département, et à Messieurs Steve Chin, Tom Henebery, Lou Saylor et John Simpson pour leur assistance technique lors des campagnes de mesures.

Je voudrais également exprimer mes plus vifs remerciements à Monsieur Zaphiris Christidis, travaillant au centre de recherche IBM de Yorktown Heights, pour m'avoir initié à Latex, que j'ai utilisé pour rédiger ce rapport.

Cette thèse a été partiellement réalisée au Département Informatique et Télématique de l'Institut National des Télécommunications, dirigé par Madame Monique Becker. J'ai pu apprécier, lors de mes nombreuses visites à ce département, l'accueil très chaleureux que m'ont réservé ses membres, et je les en remercie sincèrement.

Je tiens aussi à remercier mes collègues de l'équipe "Unité Génie Matériel" du Laboratoire de Génie Informatique pour l'ambiance amicale qu'ils y ont installée et qu'ils savent si bien maintenir.

Je voudrais associer à la réussite de ce travail tous mes amis de Grenoble et de Kingston, qui ont su m'apporter le soutien moral nécessaire pour surmonter les quelques épreuves qui ne manquent pas d'accompagner tout travail de thèse.

J'adresse enfin une pensée toute particulière à mes parents, frère et soeur, à qui je dédie cette thèse.

CHAPITRE 1

Introduction

A tous les stades de développement d'un projet informatique, que ce soit la conception d'une nouvelle architecture chez le constructeur ou l'implémentation d'une nouvelle application dans un service informatique, le recours à l'évaluation des performances permet de prédire le comportement du nouveau système (association architecture-applications), et par-là même, de mieux comprendre le système.

Lors de la conception, l'évaluation des performances permet d'analyser et de comparer plusieurs choix de réalisation et donc d'écarter très rapidement les alternatives les plus défavorables. On réduit ainsi les coûts de développement inutiles en focalisant son attention sur les choix les plus porteurs a priori [Bough88].

Tel que B. Domanski le décrit [Doman88], le responsable d'un système informatique gère son système suivant un cycle à trois phases, **l'expression des besoins, l'acquisition, le suivi du fonctionnement**.

En effet, un système informatique est conçu pour répondre à des besoins. Ces besoins peuvent s'exprimer par une demande de travail ou charge que l'on désire appliquer au système et une demande de service ou rapidité de réaction correspondante que l'on exige du système. Le premier effort est donc de bien **caractériser la charge du système**.

La deuxième phase consiste à faire une étude prospective sur le marché du système donnant un rapport service/charge optimal pour la gamme de prix que l'on s'est fixée. Cette recherche, appelée également **prévision de configuration**, doit prendre en compte, à court et moyen termes, l'augmentation inévitable de la charge et prévoir les conséquences de cet accroissement sur la qualité du service rendu par le système. Les extensions système nécessaires et leur occurrence dans le temps sont alors prévues. Parallèlement aux informations des constructeurs (très générales et donc peu adaptées aux besoins propres du responsable) et à des mesures comparatives sur les systèmes

pré-sélectionnés (qui requièrent la disponibilité des systèmes et surtout l'implémentation des applications sur les systèmes [Harms88,Wasse88]), l'évaluation des performances est sans aucun doute la solution la plus raisonnable, pour une investigation suffisamment large et peu coûteuse permettant véritablement l'élaboration d'un plan de gestion du système.

Une fois le système (ou une extension) installé(e), le responsable doit assurer le **suivi du fonctionnement du système**, du point de vue des performances. Ce suivi, appelé également gestion de performances, consiste à vérifier par des mesures fréquentes que les prévisions de service de la phase d'acquisition sont toujours satisfaites. Si elles ne le sont pas, ou bien la charge du système a augmenté et conformément au plan de gestion du système, une extension doit être installée, ou bien les prévisions de la phase d'acquisition sont erronées et les causes de cette erreur doivent être recherchées (imprécision dans l'évaluation des performances, mauvaise utilisation des ressources du système) et les actions correspondantes menées (révision du plan de gestion du système, recommandations aux utilisateurs). Dans le cas où le plan de gestion arrive à son terme (par un accroissement de la charge nécessitant un changement complet de système), un nouveau cycle est engagé.

Le cycle de vie d'un système informatique, tel que le conçoit Domanski, a donc pour pilier central l'évaluation de performances. Si l'on s'intéresse maintenant à la mise en oeuvre de l'évaluation des performances d'un système informatique, on peut être frappé des similitudes qu'elle présente avec le cycle de vie d'un système informatique.

En effet, la démarche habituelle pour évaluer les performances d'un système informatique est constituée d'un cycle à sept phases [Kienz79,Rusch82,Goldb83,Lubec88,Woodb88]:

1. l'identification du problème où les objectifs de l'étude sont formulés ainsi que les questions à résoudre,

2. la collection d'informations sur le système (configuration, système d'exploitation, caractérisation de la charge),

3. la conception d'un modèle du système,

4. l'estimation des paramètres d'entrée du modèle,

5. l'analyse ou résolution du modèle,

6. la vérification et la validation du modèle,

7. l'utilisation du modèle pour la prédiction de performances.

On peut alors regrouper les phases 1 et 2 sous la rubrique "expression des besoins", les phases 3, 4 et 5 sous le terme "modélisation" et les phases 6 et 7 sous la dénomination "suivi du fonctionnement" pour mettre en évidence l'analogie entre le cycle de vie et l'évaluation des performances d'un système informatique. Il est bon de souligner que la clé de voûte de l'évaluation de performances est la **modélisation**.

1.1 L'Evaluation des Performances

Décrivons maintenant chacune des phases constituant la démarche habituelle de mise en oeuvre de l'évaluation des performances des systèmes informatiques. Les points essentiels pour mener à bien chaque phase ainsi que les techniques et outils qui peuvent être employés sont détaillés. Nous abordons différents modes de résolution des modèles et divers types d'outils de mesures, sans pour autant avoir la prétention d'être exhaustifs.

1.1.1 Formulation des Objectifs

Lors de cette étape, on fixe les buts que l'on désire atteindre grâce à l'évaluation des performances du système. On précise non seulement les critères de performances recherchés, mais également les applications critiques du système ainsi que les composants de l'architecture qui nous intéressent en priorité. Aucune technique particulière ne s'applique à ce niveau de l'étude, mais une bonne connaissance du système informatique est déjà requise. Ajoutons simplement que les objectifs s'imposent parfois d'eux-mêmes lors du constat de performances déplorables sur un système existant. La finalité principale de l'étude est alors la recherche d'une solution pour améliorer les performances dégradées du système, qui peut résider à la fois dans une remise en question de l'architecture (ou tout du moins de sa configuration) et dans une modification des applications incriminées (voire du système d'exploitation).

1.1.2 Analyse du Système

A partir des objectifs définis à l'étape précédente, on analyse le système en considérant d'une part les composants principaux de l'architecture, d'autre part les programmes d'application qui feront l'objet de l'étude.

Concernant **l'architecture**, on détermine les différentes configurations qu'elle peut avoir, et pour chaque élément, des données à la fois qualitatives (par ex. le type de travail réalisé pour chaque application, la politique d'allocation) et numériques (par ex. la vitesse de traitement, le nombre d'unités) sont collectées.

Quant aux **programmes d'application**, on définit leurs principales caractéristiques et on quantifie les demandes de travail qu'ils requièrent de chaque composant du système. Si le nombre d'applications utilisées sur le système est prohibitif, on essaie de considérer des classes d'applications. Tous les éléments d'une même classe sont tels qu'ils possèdent des caractéristiques similaires. Le principal atout de cette approche est la possibilité de représenter toutes les applications d'une même classe par une d'entre elles seulement et donc d'obtenir une réduction considérable de données. Pour la constitution des classes, on peut utiliser des méthodes d'analyse de données (par ex. analyse en composantes principales, classification) ou avoir recours à une représentation graphique des caractéristiques qui permette une classification visuelle. Cette dernière méthode est utilisée par J.P. Bouhana pour classifier les requêtes adressées à un système transactionnel en considérant comme caractéristiques les demandes de lecture, d'écriture et de mise à-jour dans une base de données [Bouha88].

1.1.3 Conception du Modèle

Fort de l'analyse de la phase précédente et gardant bien présent à l'esprit les finalités de l'étude, on peut concevoir un premier modèle. Ce modèle est en fait constitué de deux modèles, **le modèle de l'architecture** d'une part, **le modèle des programmes d'application** d'autre part.

Comme le souligne Ruschitzka [Rusch82], plusieurs techniques de modélisation peuvent être utilisées. Parmi elles, citons la modélisation déterministe et l'optimisation combinatoire (adaptées à la détermination de stratégies d'allocation optimales ou sub-optimales), la modélisation par réseaux de files d'attente [Klein76,Coler82]. Il faut maintenant ajouter la modélisation par réseaux de

Petri, stochastiques [Flori78,Shapi79,Balbo84] ou temporisés [Holid85], ainsi que celle par réseaux d'automates stochastiques principalement utilisée pour l'évaluation des performances d'algorithmes distribués [Plate84]. Un nouveau formalisme de modélisation qui allierait les avantages d'approximation des réseaux de files d'attente et la richesse d'expression de l'espace d'état des réseaux de Petri est souhaité par P.J. Denning et G.B. Adams III [Denni87], dans leur énumération des insuffisances relevées en matière d'évaluation des performances des super-ordinateurs.

La principale difficulté du modéliseur est la détermination des éléments importants du système devant être pris en compte par le modèle. Le dialogue avec les utilisateurs et ingénieurs système, si primordial soit-il pour le modéliseur afin d'acquérir l'expertise sur le fonctionnement du système, est souvent difficile à établir. En effet, les spécialistes du système sont très rarement familiers avec l'évaluation des performances et ont du mal à appréhender les éléments cruciaux dont le modéliseur a besoin pour modéliser l'essentiel. Le modéliseur est souvent assailli par une foule de détails et c'est à lui qu'incombe la lourde tâche d'en extraire la substantifique moëlle (pour citer Rabelais) ayant trait aux performances du système.

Afin de démocratiser l'évaluation des performances des systèmes informatiques, des systèmes experts d'aide à la modélisation commencent à voir le jour [Deese88,Fang88,Feuga88] avec l'essor grandissant de ce domaine de l'intelligence artificielle. Denning et Adams III [Denni87] encouragent le recours aux systèmes experts dans l'utilisation de modèles analytiques ou autres. En effet, ces systèmes rendent transparent à l'utilisateur le modèle sous-jacent du système (bien souvent un réseau de files d'attente) en lui demandant par un dialogue interactif de décrire les composants de l'architecture et les caractéristiques des applications. Précisons cependant que chaque système expert ne s'applique actuellement qu'à l'étude d'un type bien spécifique d'architecture.

1.1.4 Estimation et Mesure des Paramètres d'Entrée

A partir du modèle du système, on déduit les paramètres d'entrée nécessaires à sa résolution. Notons que ces paramètres d'entrée concernent aussi bien le modèle de l'architecture que le modèle des programmes d'application.

Une fois ces paramètres définis, il s'agit d'en déterminer les valeurs numériques. On a recours pour ce faire soit à des mesures sur système réel (s'il existe) soit à une estimation (si le système ou l'application considérée est seulement en phase de conception). L'estimation peut parfois être arbitraire et s'avérer erronée lors de la réalisation du système ou de l'implémentation de l'application.

Pour effectuer les mesures sur système réel, on peut soit insérer dans le code des applications **des appels à des routines système** invoquant l'horloge pour faire des calculs de durées, soit avoir recours, si l'on en dispose, à des outils beaucoup plus performants (mais plus sophistiqués également), **les moniteurs**.

On distingue trois types de moniteurs [Herzo87].

- Les **moniteurs matériels** (ou "hardware") utilisent des sondes électroniques, connectées directement sur des voies de communication du système (par ex. un bus interne), capables de saisir des signaux binaires et de les acheminer vers un processeur de traitement qui met à jour des compteurs. Le principal avantage de ce type de moniteur est qu'il n'a aucune interférence avec le système testé. Cependant, l'interprétation des traces d'événements qu'il produit requiert souvent l'assistance d'un spécialiste. De plus, avec l'avènement des architectures VLSI ("Very Large Scale Integration"), le problème d'accessibilité des signaux semble devenir crucial [Nacht88].

- Les **moniteurs logiciels** (ou "software") sont des programmes de saisie de données et d'estampillage souvent résidant dans le système d'exploitation. Ils présentent l'avantage d'une interprétation des traces d'événements au niveau application (ou processus). Cependant, ils ont un impact non négligeable sur les performances des applications étudiées.

- Les **moniteurs hybrides** [Klar87,Mink88,Wybra88] sont un compromis des deux types de moniteurs précédents. Ils essaient d'allier les avantages des uns et des autres, c'est-à-dire de permettre une interprétation des traces au niveau application sans pour autant interférer de trop avec le système testé. Bien souvent, ils disposent d'un interface graphique qui permet de visualiser, en temps réel ou différé, les performances mesurées du système sous forme de courbes ou autres représentations graphiques [Wybra88].

Denning et Adams III [Denni87] soulignent l'absence d'une méthodologie tendant à incorporer dès la conception de nouvelles architectures les outils de mesures utiles à en estimer les performances.

1.1.5 Résolution du Modèle

U. Herzog distingue la résolution mathématique de la simulation [Herzo87]. Selon lui, l'approche mathématique doit être préférée à la simulation si son application est possible, car elle est beaucoup moins coûteuse en temps de calcul et permet donc d'obtenir des résultats beaucoup plus rapidement. Comme le souligne M.A. Salsburg [Salsb88], il ne faut cependant pas tomber dans l'excès, en simplifiant de manière outrancière le modèle afin qu'il puisse être résolu par une méthode analytique, et ainsi interdire une interprétation des résultats qui ait une signification vis-à-vis du système réel.

Les **méthodes analytiques** nécessitent pour leur application des hypothèses très strictes sur le modèle. On distingue des méthodes exactes et des méthodes approchées (généralement itératives). Pour des modèles encore plus complexes (notamment ceux faisant intervenir des synchronisations), seules des bornes supérieures ou inférieures peuvent être obtenues pour quelques critères de performances. Je ne ferai pas ici l'inventaire de l'ensemble des méthodes mathématiques existant à ce jour. Le lecteur peut, s'il le désire, se référer à [Houei88, p.14-16] pour une rapide description de ces méthodes et les références bibliographiques s'y rapportant. Précisons cependant que des outils de modélisation sous forme de réseaux de files d'attente [Véran84] ou sous forme de réseaux de Petri stochastiques [Flori85,Chiol87] implémentent bon nombre de ces méthodes et vérifient leur applicabilité avant leur emploi par l'utilisateur.

La **simulation** permet une modélisation beaucoup plus détaillée du système dans la mesure où elle ne requiert aucune hypothèse sur le système. Elle est donc très générale d'emploi et convient notamment à l'analyse de régimes transitoires du modèle ou à l'étude de modèles évanescents (où tous les clients ou jetons quittent le modèle dans un temps fini - phénomène typique de l'exécution d'un programme d'application). Pour mettre en oeuvre une simulation, on peut avoir recours à des progiciels de simulation [Véran84,Sauer86] ou utiliser n'importe quel langage de programmation. L'état de l'art en la matière est actuellement la simulation parallèle où plusieurs processeurs coopèrent pour simuler le modèle d'un système [Lubac84,Brine88].

Plusieurs techniques de simulation à événements discrets sont utilisées [Covin88].

- La technique **basée sur des distributions** emploie un modèle stochastique des programmes d'application pour diriger la simulation. Chaque événement est généré aléatoirement. Ainsi les branchements conditionnels sont résolus statistiquement. Cette technique est généralement

utilisée pour des systèmes non encore existants ou pour des études de performances à caractère général.

- La technique **basée sur des traces** consiste à exécuter chaque programme d'application sur une architecture similaire à celle du système étudié et à enregistrer une trace des événements caractérisant le programme d'application vis-à-vis du modèle que l'on a défini. Les traces conduisent ensuite la simulation.

- La technique **basée sur les instructions** consiste à simuler chaque instruction des programmes d'application ainsi que le temps nécessaire à l'exécution de l'instruction. Cette méthode est souvent très lente et requiert un espace mémoire considérable car elle doit maintenir une copie conforme de l'état du système réel.

- La technique **basée sur l'exécution** s'apparente à celle basée sur les instructions. Cependant, l'ordinateur hôte, où s'effectue la simulation, exécute véritablement les programmes d'application (éventuellement en pseudo-parallélisme). Pour ce faire, il est indispensable que le processeur de l'ordinateur hôte possède le même jeu d'instructions que le(s) processeur(s) du système simulé [Covin88,Wang81].

Un autre atout de la simulation est la possibilité d'une **animation** en temps réel ou différé [Hills86] du modèle du système, présentant de nombreux avantages sur l'approche classique (production de traces, de tableaux et de graphiques récapitulatifs en fin de simulation) à différents stades de développement et d'utilisation du simulateur [Johns88].

1. En phase de vérification du modèle, l'animation est un outil efficace d'aide à la détection d'erreurs de codage du modèle.

2. En phase de validation, elle constitue le support de communication essentiel entre le concepteur du modèle et l'expert du système réel, assurant une bonne compréhension entre ces deux interlocuteurs.

3. En phase d'utilisation du simulateur, elle facilite l'analyse du comportement du système par une visualisation possible d'événements simultanés et de phénomènes transitoires.

4. En phase de présentation du simulateur, dans un but d'explication des choix de modélisation ou à des fins didactiques, elle procure, par une représentation très réaliste du système modélisé, un médium de communication très appréciable.

1.1.6 Vérification et Validation

Avant l'utilisation effective du modèle pour la prédiction des performances du système qu'il représente, il reste à prouver la validité du modèle relativement aux objectifs initiaux que l'on s'était fixés.

Dans un premier temps, la **vérification du modèle** s'intéresse à la correction du programme de définition et de résolution du modèle. J.C. Comfort [Comfo87] a plus particulièrement étudié les tests de vérification des modèles de simulation. Selon lui, des méthodes similaires à celles employées dans les tests de logiciels peuvent être utilisées, telles que la technique du partitionnement des données d'entrée du programme en classes d'équivalence pour la construction de jeux de tests, l'analyse des valeurs limites. La simulation mettant bien souvent en oeuvre des phénomènes stochastiques, on s'efforcera de supprimer le caractère aléatoire des événements en considérant des intervalles d'inter-arrivées et des temps de service constants. On pourra également étudier le cas où les clients arrivent dans le modèle par vagues importantes, mais très espacées dans le temps, pour s'intéresser au comportement du modèle en situation de saturation passagère.

La **validation du modèle**, telle que la définit R.G. Sargent [Sarge82], est la justification que le modèle possède, à l'intérieur de son champ d'application, un degré de précision satisfaisant en accord avec les objectifs de son utilisation.

Sargent souligne que toute la difficulté dans le processus de validation est la détermination du degré de précision satisfaisant. Selon lui, la validité du modèle doit être vérifiée à deux niveaux.

Le **niveau conceptuel** a pour tâche de valider les hypothèses qui ont été considérées lors de l'analyse et de la modélisation du système.

Le **niveau opérationnel** s'attache à vérifier que, compte tenu des objectifs d'utilisation du modèle, ses caractéristiques principales représentent convenablement le système. Cette vérification s'opère essentiellement en comparant les résultats du modèle avec des mesures sur le système. Pour ce faire, on peut avoir recours à des tests statistiques (tests d'hypothèse ou intervalles de confiance) ou à des critères visuels à partir de représentations graphiques.

Sargent ainsi que Denning et Adams III [Denni87] déplorent cependant l'absence de méthodologie de validation des modèles qui puisse indiquer quelle technique de validation utiliser pour tel problème et permettre une reproductibilité des expérimentations.

Bien souvent, le procédé de validation est basé sur de **simples comparaisons** entre mesures et résultats de résolution du modèle, qui permettent **de constater plus que d'expliquer** les erreurs dues aux approximations de modélisation. Une tentative de diagnostic de l'origine des erreurs peut, dans certains cas, être entreprise, afin d'analyser les remèdes possibles pour que le modèle représente plus fidèlement le système réel.

1.1.7 Utilisation du Modèle pour Prédire les Performances du Système

Pour prédire les performances du système, on utilise le modèle que l'on résout pour différents jeux de paramètres d'entrée. Deux modes d'utilisation peuvent être a priori employés. En fonction des finalités de l'étude, on préfèrera un mode devant l'autre.

Le **mode déterministe** consiste à considérer chaque temps de service, chaque arrivée de client dans le modèle, chaque transition d'une station de service (ou d'une place) vers une autre comme des constantes ne dépendant que de la classe du client. Ce mode est principalement employé pour des **études de cas**. Son intérêt est la prise en compte très fidèle des événements se produisant dans le système, mais le nombre de paramètres d'entrée qu'il nécessite est très vite considérable. Précisons également que ce mode d'utilisation est communément employé lors de la phase de **validation du modèle**.

Le **mode probabiliste** utilise des lois de distribution de probabilités qui régissent l'occurrence et la durée des événements. C'est le mode privilégié de la **prédiction de performances**, dont l'objet est le plus souvent **une analyse exploratoire générale du comportement du système** plutôt que celle d'études de cas très spécifiques. Le dessein d'une telle analyse est surtout de **mettre en évidence certains phénomènes**, pathologiques ou non, leur explication pouvant nécessiter une **analyse ultérieure**. En d'autres termes, la prédiction de performances joue le rôle de **révélateur de phénomènes**, dont l'explication incombera, lors d'une phase ultérieure, aux spécialistes du système et des programmes applications, en collaboration avec l'évaluateur de performances. La principale difficulté de la mise en oeuvre de ce mode d'utilisation est la caractérisation des lois de probabilité reflétant le plus fidèlement le système réel.

1.2 Présentation du Rapport de Thèse

A partir de la décomposition en sept phases présentée précédemment, sur laquelle repose la méthodologie générale de modélisation des systèmes informatiques monoprocesseurs multiprogrammés, introduite en 1979 par M.G. Kienzle et K.C. Sevcik [Kienz79], nous avons réalisé quatre études de performances très précises sur des systèmes parallèles réels de types fort différents. Trois études ont donné lieu à une modélisation très détaillée et la quatrième à des mesures de performances analysées avec beaucoup de minutie. Que le lecteur nous permette de préciser qu'à ce jour, fort peu de modélisations ont été réalisées avec ce niveau de détail sur des systèmes parallèles réels.

Forts de l'expérience acquise au cours de ces études, nous présentons une méthodologie de modélisation adaptée à l'évaluation des performances des architectures parallèles, s'inspirant naturellement de celle mentionnée précédemment, mais présentant de nombreuses techniques convenant fort bien aux architectures parallèles. Parmi elles, citons les méthodes d'agrégation, l'utilisation de l'analyse des données pour la constitution de classes de programmes d'application, le découpage du modèle en deux parties, le modèle de l'architecture d'une part et le modèle des programmes d'autre part.

Notre approche de modélisation se veut un compromis entre un modèle très précis (relativement déterministe), validé pour quelques cas, et un modèle probabiliste, adapté à des études statistiques de caractère plus général. Ainsi, afin de pouvoir générer aléatoirement les événements corrélés propres aux architectures parallèles, tels que communications et synchronisations entre processus parallèles, une technique originale est introduite.

Ce rapport de thèse est divisé en six chapitres, ce chapitre d'introduction excepté. Il comporte en outre deux annexes.

Le chapitre deux présente la méthodologie de modélisation que nous préconisons pour évaluer les performances des architectures parallèles. Ce chapitre fait tout d'abord un rappel de la méthodologie de modélisation introduite par Kienzle et Sevcik. Il propose ensuite une classification des architectures parallèles, puis il présente, en suivant la décomposition en sept phases de toute analyse d'évaluation de performances, les techniques tout à fait adaptées aux architectures parallèles. Pour être plus expressive, chaque phase est agrémentée de nombreux exemples empruntés aux quatre études de cas, qui sont à l'origine de cette méthodologie.

Les quatre chapitres suivants détaillent les différentes études de cas, afin d'illustrer l'applicabilité de la méthodologie.

Ainsi, le chapitre trois est consacré à la modélisation des réseaux point-à-point de systèmes Hewlett-Packard HP3000, sous forme de réseau de files d'attente. Conçu pour être utilisé en conjonction avec les outils de mesures et d'évaluation de performances que Hewlett-Packard a développés pour les systèmes HP3000 isolés, le modèle applique une méthode d'agrégation bien adaptée à ce mode d'utilisation.

Le chapitre quatre présente l'étude de mesures de performances, que nous avons réalisée sur un réseau local d'IBM PC, afin de comparer les efficacités respectives de trois stratégies d'implémentation d'algorithmes parallèles sur ce type de réseau. Bien que cette étude de cas n'ait pas donné lieu à une modélisation, certains aspects de la modélisation, qui pourrait être faite pour ce système, sont abordés dans le chapitre deux.

Le chapitre cinq s'intéresse à la modélisation du calculateur scientifique FPS264 de Floating Point Systems. Cette modélisation est fort intéressante, **sur le plan méthodologique**, par son caractère hybride (deux modèles sont développés, l'un fort détaillé, à base de réseau de Petri stochastique, et l'autre très simple, dérivé du premier, à base de réseau de files d'attente), par la méthode itérative d'agrégation-désagrégation employée pour résoudre le modèle détaillé, et par la constitution de classes de programmes, à l'aide d'une technique empruntée à l'analyse de données.

Le chapitre six décrit la modélisation, sous forme de réseau de files d'attente, du système lCAP ("loosely Coupled Array of Processors"), conçu à IBM Kingston. Ce chapitre démontre tout l'intérêt d'une distinction nette entre le modèle de l'architecture et le modèle des programmes, permettant à la fois une modélisation déterministe et une modélisation probabiliste des programmes, sans remettre en question le modèle de l'architecture. Une analyse de prédiction de performances fort détaillée permet d'illustrer une utilisation possible du modèle développé et souligne ainsi tout l'intérêt de l'évaluation des performances. De plus, cette analyse de prédiction de performances donne un exemple d'application de la technique originale de génération aléatoire d'événements corrélés, introduite au chapitre deux.

Le chapitre sept conclut le présent rapport. Il rappelle les principaux points de la méthodologie introduite et présente quelques prolongements possibles à ce travail.

En outre, l'annexe A présente la structure générale, largement commentée, du programme de simulation du modèle du système ICAP, et l'annexe B donne une description de la plupart des références bibliographiques, qui agrémentent ce chapitre d'introduction.

CHAPITRE 2

Présentation de la Méthodologie

Ce chapitre présente tout d'abord la méthodologie élaborée en 1979 par M.G. Kienzle et K.C. Sevcik [Kienz79], avant l'essor des architectures parallèles.

Il s'intéresse ensuite au cas des architectures parallèles. Les principales raisons de leur essor actuel sont exposées et une classification de ces architectures est proposée. Une méthodologie de modélisation adaptée à de telles architectures est détaillée, avec de larges illustrations des différentes techniques la constituant et de nombreux renvois aux quatre études de cas présentées très en détail dans les chapitres suivants.

Ce chapitre propose enfin un tableau récapitulatif énumérant les points les plus importants de la méthodologie présentée.

2.1 Méthodologie Générale de Modélisation

Dès 1979, M.G. Kienzle et K.C. Sevcik proposent une méthodologie générale de modélisation pour l'évaluation des performances des systèmes informatiques [Kienz79], à l'époque essentiellement monoprocesseurs multiprogrammés. Ce paragraphe détaille cette méthodologie et le paragraphe suivant précise les adaptations qu'on peut lui appliquer pour qu'elle puisse également s'utiliser pour l'évaluation des performances des architectures parallèles.

La méthodologie de Kienzle et Sevcik repose sur le découpage en sept phases de l'évaluation de performances, détaillé au paragraphe 1.1. L'originalité de cette méthodologie est la considération de deux modèles du système, le **modèle conceptuel** (qui représente véritablement le système étudié, avec tous les détails nécessaires) et le **modèle opérationnel** (qui est une approximation du premier afin de permettre un temps de résolution non prohibitif). L'idée de Kienzle et Sevcik est de

pouvoir se ramener pour le modèle opérationnel à un réseau de files d'attente qui puisse se résoudre analytiquement.

La démarche de modélisation qu'ils préconisent est la suivante:

1. définir les buts de la modélisation, c'est-à-dire les critères de performances du système réel que l'on veut évaluer,
2. définir le modèle opérationnel, sous forme de réseau de files d'attente, dont la résolution puisse nous fournir les critères précisés en 1 (la méthode de résolution est également déterminée ici),
3. à partir des paramètres d'entrée du modèle opérationnel et compte tenu de la finalité de l'étude, définir le modèle conceptuel, dont le rôle principal est de fournir les paramètres d'entrée du modèle opérationnel à partir de ses paramètres d'entrée propres mesurables sur le système réel,
4. réaliser les mesures sur système réel qui permettent d'obtenir directement ou d'estimer les paramètres d'entrée du modèle conceptuel.

Le **modèle conceptuel**, que Kienzle et Sevcik définissent comme une représentation d'un point de vue logique de tous les composants et fonctions essentielles du système réel (configuration, système d'exploitation, programmes d'application, etc.), doit selon eux comporter:

- la caractérisation des programmes d'application par la détermination de leurs demandes de service aux différentes ressources du système, sans prise en compte des interactions avec le système d'exploitation,
- le schéma d'interférence entre les programmes d'application et le système d'exploitation, c'est-à-dire la représentation de toutes les interactions statiques ou dynamiques entre les programmes d'application et le système d'exploitation,
- le niveau de multiprogrammation pour chaque classe de travaux (interactif, batch),
- la matrice de charge qui donne, pour chaque programme d'application, les demandes de service aux diverses ressources du système, en imputant le surcoût ("overhead") système caractérisé dans le schéma d'interférence,
- les caractéristiques des ressources: discipline d'allocation, taux de service (pouvant dépendre de la charge du système) pour les ressources actives (CPU, disques par ex.).

Kienzle et Sevcik insistent enfin sur le fait que plus les mesures sur le système réel seront nombreuses et précises, plus le modèle conceptuel sera affiné et, par voie de conséquence, meilleure sera la détermination des paramètres d'entrée du modèle opérationnel.

Suite à cet exposé détaillé de la méthodologie, je voudrais faire quelques commentaires. Peu de détails sont donnés quant à la conception du modèle opérationnel. Quels éléments faut-il qu'il comprenne afin que la construction du modèle conceptuel soit possible ? Faut-il se restreindre forcément, pour le modèle opérationnel, à une structure telle qu'il puisse être résolu analytiquement ? Et cette restriction ne peut-elle pas conduire dans certains cas à une impasse pour la réalisation du modèle conceptuel ? Toutes ces remarques amènent à penser que le modèle opérationnel et le modèle conceptuel doivent être conçus de concert. En effet, une démarche tout aussi naturelle aurait été de construire d'abord le modèle conceptuel, d'en déduire un réseau de files d'attente, qui puisse bien sûr répondre aux buts de l'étude, et dont on soit assuré de pouvoir déterminer les paramètres d'entrée, et d'étudier, ensuite seulement, quelles approximations sont nécessaires à sa résolution analytique, en vérifiant si ces approximations ont un sens pour le système réel.

2.2 Cas des Architectures Parallèles

Afin de réduire le temps d'exécution des gros programmes d'application, le recours aux architectures parallèles est aujourd'hui indéniable. K.M. Chandy [Chand88] et P.C. Treleaven [Trele88] (parmi d'autres) se sont intéressés aux principales raisons de cet engouement pour les architectures parallèles:

- réduction de la taille des composants au silicium pour une même puissance de calcul,
- coût peu élevé par association de composants standards de plus en plus puissants (existence sur le marché de micro-processeurs 32 bits),
- grande extensibilité par accroissement incrémental de ressources,
- tolérance aux pannes par duplication de composants critiques et possibilité de fonctionnement en mode dégradé.

D'autres facteurs freinent cependant le plein essor des architectures parallèles:

- difficulté de programmation,
- réticence des entreprises à abandonner tous leurs programmes d'application séquentiels dont elles sont fort dépendantes, et ce malgré le développement de langages parallèles (par ex. OCCAM, XANADU [Frabo87]),
- manque de logiciels parallèles sur le marché.

Devant le nombre incalculable d'applications scientifiques écrites en FORTRAN et pour favoriser l'essor commercial des architectures parallèles, le développement de sur-langages de FORTRAN implémentant le parallélisme intra-système (ex. VM/EPEX-FORTRAN [Darem88] ou Parallel FORTRAN [Gentz88]), voire inter-système [Cleme88], s'intensifie. L'objectif est de permettre très simplement (voire automatiquement) l'implémentation du parallélisme inhérent à bon nombre de programmes séquentiels.

2.2.1 Classification des Architectures Parallèles

Tel que l'écrit O. McBryan [McBry87], il est difficile de concevoir une classification des architectures parallèles unique, tant leur nombre et leur variété sont importants. On peut distinguer les architectures SIMD ("Single Instruction Multiple Data stream") des architectures MIMD ("Multiple Instruction Multiple Data stream"), ou les architectures à mémoire globale partagée des architectures à mémoires locales distribuées, ou encore faire une classification d'après la structure du réseau d'interconnexion des processeurs.

R.W. Hockney [Hockn85] distingue les architectures SIMD et les architectures MIMD. Pour les architectures MIMD, il considère deux classes principales, les architectures commutées et les architectures réseaux. Parmi les architectures commutées, il distingue les architectures à mémoire(s) partagée(s) de celles à mémoires distribuées. Les architectures réseaux sont classées selon la topologie du réseau d'interconnexion et on distingue les réseaux maillés (anneaux, rectangles), les cubes (hypercubes), les réseaux hiérarchiques (arbres) et les réseaux reconfigurables.

K.M. Chandy [Chand88] considère quatre classes principales d'architectures parallèles:

1. les pipelines (ex. CRAY-1),
2. les architectures SIMD (ex. Connection Machine),
3. les architectures MIMD, constituées de plusieurs processeurs échangeant de l'information au travers de mémoires partagées (ex. BBN Butterfly Parallel Processor) ou par transmission de messages (ex. Intel iPSC),
4. et les autres architectures, de type SPMD ("Single Program Multiple Data" - ex. VM/EPEX-FORTRAN) ou "data-flow".

Pour P.C. Treleaven [Trele88], l'approche est moins classique. Il distingue cinq classes principales:

1. les systèmes transactionnels regroupant les systèmes parallèles UNIX, les systèmes parallèles de base de données, les systèmes tolérants aux pannes et les systèmes de communication,
2. les super-ordinateurs numériques comprenant entre autres les hypercubes,
3. les architectures VLSI ("Very Large Scale Integration"), à base de grilles de processeurs parallèles spécialisés (tableaux systoliques) ou à base de micro-processeurs d'usage général à architecture RISC ("Reduced Instruction Set Concept"), telles que le Transputer d'INMOS,
4. les ordinateurs de la 5ème génération, basés sur la programmation symbolique et utilisant des langages non-procéduraux (orienté-objet - ex. SMALLTALK, fonctionnel - ex. LISP, logique - ex. PROLOG, basé sur la connaissance - ex. OPS5),
5. et les systèmes neuroniques reposant sur un parallélisme massif et constitués d'un très grand nombre de processeurs très primitifs, soit spécialisés (modèle neuronique), soit programmables (modèle connectionniste).

Pour ma part, je désignerai sous le terme d'"architecture parallèle" tout système constitué de plusieurs processeurs, coopérant pour l'exécution de programmes parallèles ou répartis, écrits à l'aide de langages procéduraux (ex. FORTRAN, PASCAL, OCCAM, assembleur).

Sous ce terme sont ainsi regroupées à la fois les architectures parallèles classiques (selon la classification de R.W. Hockney) et les architectures distribuées de type réseau d'ordinateurs (à distance ou local).

La classification des architectures parallèles que je propose est la suivante:

1. les architectures distribuées, regroupant les réseaux d'ordinateurs, à distance ou locaux,
2. les architectures SIMD,
3. les architectures MIMD, distinguant les architectures commutées des architectures réseaux (classification de R.W. Hockney),
4. les autres architectures de type SPMD ou hybrides (ex. ICAP/FPS264 [Prost89a,Prost89b]).

2.2.2 Méthodologie Adaptée aux Architectures Parallèles

Avant de préciser les aménagements à la méthodologie présentée au paragraphe 2.1, nécessaires pour évaluer les performances des architectures parallèles, classifiées précédemment, introduisons les principaux éléments constituant le temps d'exécution d'un programme parallèle.

Principaux Eléments Constituant le Temps d'Exécution d'un Programme Parallèle

E. Clementi, D. Logan et J. Saarinen [Cleme88] soulignent que l'accélération résultant de l'exécution parallèle d'un programme d'application présente une valeur maximale pour un nombre de processeurs optimal. En effet, de par la loi d'Amdahl qui établit:

$$S(p) = \frac{1}{1 - X + X/p}$$

où $S(p)$ est l'accélération réalisée lors de l'exécution parallèle du programme sur p processeurs avec un pourcentage X du code parallélisé, ils montrent que $S(p)$ peut passer par un maximum si X est une fonction décroissante de p.

Viennent s'ajouter à ce phénomène deux autres paramètres qui influent directement sur l'accélération:

- les communications inter-processus nécessaires aux transferts de messages pour les architectures commutées à mémoires distribuées et aux synchronisations pour l'ensemble des architectures MIMD,
- le déséquilibre de charge entre tâches parallèles qui peut engendrer des attentes conséquentes à certains points de synchronisation; Clementi, Logan et Saarinen précisent qu'il peut être de deux natures, statique (i.e. lié à l'implémentation parallèle de l'application) et/ou dynamique (i.e. provoqué par l'exécution sur la même architecture d'autres programmes concurrents).

K.M. Chandy [Chand88] indique que les principaux problèmes liés aux performances des programmes parallèles sont le temps de latence associé à chaque transfert de message entre processus, le temps de commutation entre processus (si le nombre de processus est supérieur au nombre de processeurs), la contention pour l'accès aux voies de communication, et les entrées-sorties abondantes nécessaires à l'initialisation de chaque processus.

A la lecture de ces problèmes, et compte-tenu du fait que les synchronisations entre processus sont souvent implémentées par des transferts de messages ou des accès à des variables partagées, on mesure toute l'importance que revêt l'ensemble des communications nécessaires à l'exécution parallèle d'un programme. On peut donc, d'ores et déjà, attirer l'attention du lecteur sur le fait qu'un des éléments essentiels, pour réussir une évaluation des performances fidèle d'une architecture parallèle, est une très bonne caractérisation des communications entre processus.

Ainsi, on distingue deux constituants principaux dans le temps d'exécution d'un programme parallèle, le **temps de calcul proprement dit** et le **temps nécessaire aux communications**. Bon nombre d'études de performances sont basées sur ce principe [Broch87,Duda88,Hockn89].

Méthodologie de Modélisation des Architectures Parallèles

Nous proposons maintenant une méthodologie de modélisation adaptée aux architectures parallèles, en considérant successivement chaque phase constituant toute étude d'évaluation de performances (cf paragraphe 1.1).

Afin d'étayer notre présentation, les quatre études de cas, décrites en détail dans les chapitres 3, 4, 5 et 6, et qui sont à l'origine de cette méthodologie, illustrent les principaux aspects et techniques présentés, en précisant l'approche que nous avons utilisée dans chaque cas. Afin de faciliter leur référence ultérieure, ces différents points sont repérés dans la méthodologie de modélisation par des lettres majuscules placées entre crochets.

Avant d'exposer la méthodologie, donnons une brève description de ces quatre études de cas.

1. Les **réseaux point-à-point de systèmes HP3000** sont un exemple d'architecture distribuée de type réseau à distance [Auber87]. Ils sont formés de plusieurs systèmes Hewlett-Packard 3000, reliés deux à deux par des lignes de communication conformes au protocole X.25. Deux modes d'utilisation des applications sont possibles: le mode local, où l'utilisateur exécute l'application localement sur le système où il est connecté (aucune activité réseau n'est alors générée), et le mode distant, où l'utilisateur exécute l'application sur un site connecté (par un lien direct ou non) à son site de connexion (toute transaction provoque alors des communications inter-systèmes). Un modèle sous forme de réseau de files d'attente a été développé pour de tels réseaux, prenant en compte à la fois des applications interactives et des transferts de fichiers. Pour plus de détails, le lecteur est invité à se reporter au chapitre 3.

2. Les **réseaux locaux d'IBM PC** sont un exemple d'architecture distribuée de type réseau local [Prost88c]. Un tel réseau local a été utilisé pour implémenter deux algorithmes parallèles (produit de matrices, résolution de systèmes d'équations linéaires par la méthode de relaxation), selon trois stratégies différentes d'allocation des tâches de calcul aux IBM PC. Ces stratégies reposent, toutes trois, sur une décomposition de domaine, consistant à partitionner les données à traiter entre les différents processeurs présents sur le réseau, et sur une organisation hiérarchique des processeurs, de type maître-esclaves. Des mesures de performances ont permis de comparer les efficacités respectives de ces trois stratégies. Le lecteur dispose, au chapitre 4, d'une analyse fort détaillée de ces mesures. Bien que cette étude n'ait pas donné lieu à une modélisation pour des contraintes de temps, certains aspects de la méthodologie sont néanmoins illustrés à partir de cette étude de cas, grâce à une connaissance prévisionnelle suffisamment précise du modèle qui pourrait être conçu pour ce type de système parallèle.

3. Le **calculateur scientifique Floating Point Systems 264** est assimilable à une architecture MIMD de type commuté à mémoire partagée [Becke88]. Deux unités arithmétiques (un additionneur et un multiplieur) à virgule flottante peuvent opérer simultanément sur des

données stockées en mémoire principale ou dans une table des constantes. Un modèle sous forme de réseau de Petri stochastique a été conçu pour produire les paramètres d'entrée d'un modèle très simplifié du calculateur, sous forme de réseau de files d'attente constitué essentiellement des quatre unités fonctionnelles précédemment citées (additionneur, multiplieur, mémoire principale et table des constantes). Le lecteur dispose d'une présentation beaucoup plus détaillée au chapitre 5.

4. Le **système parallèle lCAP/FPS264** ("loosely Coupled Array of Processors FPS264") de IBM Kingston est une architecture MIMD commutée de type hybride [Prost89a,Prost89b]. Il est composé d'un ordinateur hôte IBM connecté à dix systèmes FPS264 (appelés "Attached Processors" ou APs) par des canaux de communication. Cinq mémoires de masse relient les APs entre eux selon une topologie en double anneau, chaque AP étant connecté à deux mémoires et chaque mémoire étant accessible depuis quatre APs. Ces mémoires seront qualifiées de mémoires locales. Enfin, une mémoire de masse globale est reliée aux dix APs. Les mémoires de masse peuvent être utilisées comme des boîtes à lettres, lorsque les APs communiquent par l'intermédiaire de messages (mode message), ou comme des mémoires partagées (mode partagé). Un seul de ces deux modes peut être utilisé tout au long de l'exécution d'un même programme d'application. Les APs, gérés par l'ordinateur hôte, peuvent opérer simultanément sur des données et des instructions différentes d'un même programme d'application parallèle. L'expression du parallélisme se fait par l'intermédiaire de directives spécifiques insérées par le programmeur dans le code FORTRAN (pour l'ordinateur hôte) ou APFTN64 (pour les FPS264). Ces directives sont ensuite traduites par un précompilateur en des appels à des routines systèmes implémentant la gestion des tâches parallèles et les communications et synchronisations inter-tâches. Un modèle sous forme de réseau de files d'attente a été réalisé afin de prédire les performances de programmes d'application s'exécutant concurremment sur le système. Le chapitre 6 est dédié à la présentation très détaillée de la modélisation de ce système.

FORMULATION DES OBJECTIFS

Lors de la formulation des objectifs, un dialogue doit s'établir avec tous les partenaires du système (ingénieurs système, opérateurs, utilisateurs), s'il existe, ou avec l'équipe de conception du futur système. Les finalités de l'étude doivent être énoncées clairement et doivent préciser les points

devant retenir une attention toute particulière. Tout au long du processus de modélisation, il faudra conserver ces objectifs bien présents à l'esprit afin que le modèle puisse y répondre.

ANALYSE DU SYSTEME

L'analyse du système se fait en étroite collaboration avec les ingénieurs système et les opérateurs. Une documentation exhaustive est souhaitable également. Tous les aspects matériels et logiciels du système en rapport avec les finalités de l'étude doivent être disséqués autant que faire se peut. En effet, il n'est pas rare, à ce stade de l'étude, que des problèmes de confidentialité viennent interdire l'accès à certaines informations capitales pour une bonne compréhension du système. Il est bien sûr souhaitable d'avoir connaissance de ces problèmes le plus tôt possible. C'est également à cette étape que l'on répertorie les outils de mesure dont on pourra user tant pour l'estimation des paramètres d'entrée du modèle que pour sa validation. On évite ainsi la conception d'un modèle dont la caractérisation des paramètres d'entrée s'avérera fort approximative, voire impossible. Compte tenu des remarques faites au paragraphe précédent, un effort particulier sera porté sur la caractérisation des communications et des synchronisations.

CONCEPTION DU MODELE

Contrairement à la démarche de Kienzle et Sevcik (cf paragraphe 2.1), je préconise dans un premier temps **la conception d'un modèle très précis du système** qui prenne en compte tous les aspects étudiés à l'étape précédente, dont la caractérisation n'a pas été interdite pour des raisons de confidentialité. Ce modèle constitue le **modèle conceptuel**. Pour chaque paramètre d'entrée du modèle, on s'interroge déjà sur la marche à suivre pour son estimation. Le modèle opérationnel sera déduit du modèle conceptuel ultérieurement.

On s'intéresse tout d'abord aux **aspects matériels** du système (processeurs, mémoires, voies de communication, etc.) afin d'obtenir un **modèle de l'architecture** [**A**].

Dans un deuxième temps, on étudie les **aspects logiciels**, c'est-à-dire d'une part les caractéristiques du **système d'exploitation** essentielles pour une bonne modélisation du comportement du système (politiques d'allocation, gestion des processus, gestion des mémoires, etc.), d'autre part les **programmes d'application** qui constituent la charge du système. On élabore ainsi le **modèle des programmes** [**B**].

Rappelons que si le nombre de programmes est très grand, on pourra classifier les applications en utilisant des techniques empruntées à **l'analyse des données** [C], telles que l'analyse en composantes principales [Becke88] (cf paragraphe 5.5.1) et ne considérer alors qu'une application représentative de chaque classe.

Dans le modèle des programmes, on distingue les deux éléments principaux qui constituent le temps d'exécution des programmes, **le calcul proprement dit** et **les communications**.

Pour modéliser les **tâches de calcul**, une **finesse** est à définir [**D**], en fonction des finalités de l'étude.

Par exemple, il n'est pas utile que le modèle représente l'exécution de chaque instruction du programme d'application si l'objet principal de l'étude est l'évaluation des coûts de communication du programme et l'effet d'un déséquilibre de charge entre tâches parallèles sur le temps d'exécution. Un excès de modélisation peut en effet entraîner des difficultés inutiles de résolution du modèle opérationnel.

Dans le cas du modèle du FPS264, les calculs s'opèrent au sein des deux unités arithmétiques (l'additionneur et le multiplieur). Aussi est-il nécessaire de considérer les programmes d'application à un niveau où l'on a accès à l'activité de ces unités arithmétiques. C'est pourquoi les programmes ont été analysés à partir du **code assembleur**. Une analyse lexicale du code a permis d'automatiser l'obtention des principales caractéristiques des programmes, telles que le degré de parallélisme, le pourcentage d'utilisation des unités fonctionnelles, la répartition des opérandes entre donnée issue de la mémoire principale ou de la table des constantes et résultat d'opération (addition ou multiplication) obtenu au cycle précédent (cf paragraphe 5.3.1). Puis des classes de programmes ont été constituées à partir d'une analyse en composantes principales des caractéristiques (cf paragraphe 5.5.1).

Dans le cas de ICAP/FPS264, l'intérêt de l'étude est plutôt porté sur le coût des communications et des synchronisations que sur une caractérisation très fine des tâches de calcul. Aussi, le modèle des programmes d'application considère chaque **bloc d'instructions FORTRAN (ou APFTN64)** délimité par des directives de communication ou de synchronisation comme un élément indivisible et le temps CPU de calcul de chaque bloc est seul requis pour caractériser l'élément (cf paragraphe 6.3.2).

Dans le cas du réseau local d'IBM PC, une approche semblable à celle du système lCAP semble appropriée. En effet, on est ici fort intéressé par le coût des communications. Les tâches de calcul, que l'on peut facilement isoler et dont le temps d'exécution (mesurable très aisément) seul nous importe, peuvent être également considérées comme des éléments indivisibles.

En ce qui concerne les réseaux point-à-point de systèmes HP3000, les applications considérées sont très complexes et font très peu appel à des tâches de calcul. Outre les échanges de messages sur les lignes de communication entre systèmes, le modèle des programmes d'application comporte des tâches de gestion des requêtes utilisateur, telles que des tâches d'acheminement des données ou résultats sur les lignes de communication, des tâches d'accès à des fichiers, des tâches de gestion des terminaux. Toutes les **tâches**, quelles qu'elles soient, **nécessaires à la gestion d'une même requête utilisateur** et séparant deux communications de données, sont regroupées pour ne former qu'un seul élément. Le temps de réponse pour l'exécution du groupe de tâches est déterminé par une modélisation plus fine de chaque système HP3000 présent sur le réseau en distinguant le temps CPU du temps d'accès disque. (cf paragraphe 3.3.2).

La **granularité** des tâches de calcul parallèles définit leur longueur en nombre d'opérations ou d'instructions de calcul à effectuer. Bien sûr, ce nombre conditionne, pour un programme d'application donné, le nombre de tâches parallèles à définir, appelé degré de parallélisme. Le modèle doit donc pouvoir **générer le degré de parallélisme** [**E**].

Dans le cas du modèle du FPS264, une partie du modèle détaillé, appelée **modèle de contrôle** (cf paragraphe 5.3.1), est dédié à la génération du degré de parallélisme.

Dans le cas du modèle du système lCAP/FPS264, le degré de parallélisme est statique pour chaque job et correspond au nombre d'APs qui lui sont attachés. Sa génération est assurée par un paramètre d'entrée du modèle. Le temps moyen d'exécution des tâches de calcul parallèles est bien sûr ajusté au degré de parallélisme.

Dans le cas du réseau local d'IBM PC, on est confronté à un problème supplémentaire, celui de processeurs hétérogènes, c'est-à-dire de processeurs dont les vitesses de calcul sont différentes. La notion de **processeur virtuel** [**F**] définit une unité de performance permettant l'appréciation relative des différents processeurs physiques. Le processeur physique le plus lent devient le processeur de référence et équivaut à un processeur virtuel. Chacun des autres processeurs physiques est "équivalent", compte tenu du rapport de sa vitesse de calcul propre avec celle du processeur de référence, à un certain nombre de processeurs virtuels. On peut alors définir une granularité des tâches de calcul, différente pour chaque processeur physique, qui est directement fonction du

nombre de processeurs virtuels auxquels ce processeur est équivalent. On assure ainsi un meilleur équilibrage de charge entre tâches de calcul parallèles. Le degré de parallélisme est dans ce cas exprimé en nombre de processeurs virtuels de calcul (cf paragraphe 4.5.1).

Pour les **communications**, le temps de **latence** ainsi que le temps de **transmission** doivent tous deux être modélisés [**S**].

Les attentes induites par un déséquilibre éventuel de charge entre tâches parallèles sont également prises en compte par une modélisation fidèle des attentes aux points de synchronisation.

ESTIMATION ET MESURE DES PARAMETRES D'ENTREE

L'estimation des paramètres d'entrée du modèle conceptuel est alors effectuée. Si possible, on essaie d'obtenir la valeur de ces paramètres par **mesure directe sur le système** (s'il existe bien sûr). Afin d'éviter les biais induits par une charge extérieure, on s'efforce d'opérer chaque mesure sur **système dédié**, sauf si l'objet de la mesure est l'influence d'une charge extérieure sur tel ou tel paramètre.

Si les mesures peuvent se faire à l'aide de moniteurs, les moniteurs hybrides répartis sont assurément les mieux appropriés [**G**]. Ils permettent l'enregistrement simultané sur chaque processeur de traces d'événements, sans influence trop forte sur les programmes d'application, et la synthèse automatique des différentes traces par un processeur de traitement dédié permet leur interprétation au niveau application [Klar87,Wybra88].

Pour la **détermination des temps de calcul**, les mesures se font de préférence sur un seul processeur. Si la tâche de calcul est parallèle et est basée sur une décomposition de domaine, on prend garde à ne considérer que la portion de données effectivement traitée par chaque processus.

Pour la **détermination des paramètres de communication et de synchronisation**, on a recours au **nombre minimum de processeurs nécessaires**, afin de réduire, autant que faire se peut, les problèmes de contention dûs à l'accès à des ressources partagées, problèmes qui seront pris en compte par le modèle et qui ne doivent donc pas être intégrés dans les paramètres d'entrée.

Dans le cas du modèle des réseaux point-à-point de systèmes HP3000, toutes les mesures ont été réalisées à l'aide d'un **moniteur logiciel**, SPEP ("System Performance Evaluation Project"), permettant la saisie de traces d'événements et leur réduction sous la forme d'un rapport lisible, qui facilite leur interprétation au niveau application grâce à une présentation des résultats sous forme d'histogrammes. Ce moniteur n'étant pas distribué, il a été **lancé sur chaque système du réseau** et un recoupement des rapports de réduction a été nécessaire, pour l'identification de chaque requête utilisateur et le suivi de son cheminement à travers le réseau. Le temps CPU consommé par chaque requête utilisateur ainsi que le nombre d'accès disque ont alimenté le modèle simplifié de chaque système HP3000, noeud du réseau (cf paragraphe 3.4.2). La résolution de ces modèles simplifiés a permis d'obtenir le temps de réponse moyen de chaque noeud pour la gestion de la requête utilisateur, ce temps de réponse étant à son tour paramètre d'entrée du modèle global du réseau.

Pour le modèle du réseau local d'IBM PC, le temps d'exécution des tâches de calcul, parallèles ou non, pourrait très bien être obtenu par mesure directe sur un IBM PC isolé.

Pour les communications, il serait nécessaire de faire une étude approfondie du protocole de communication utilisé sur ce type de réseau, en l'occurrence CSMA/CD ("Carrier Sense Multiple Access with Collision Detection"), pour estimer d'une part le temps de latence et d'autre part le coût des retransmissions de messages dues aux collisions et aux erreurs de transmissions. Le débit des lignes (2 mégabits par seconde) et la longueur des messages suffiraient à la détermination du temps de transmission.

Pour les synchronisations des tâches de calcul esclaves avec le maître, une étude très fine de l'ensemble des traitements s'effectuant sur le processeur maître devrait être réalisée.

Pour le modèle du FPS264, tous les paramètres d'entrée du modèle ont été déterminés à partir de **l'analyse lexicale du code assembleur** des programmes (cf paragraphe 5.4). Seul le temps de cycle a été obtenu à partir de la documentation fournie par Floating Point Systems. La synchronisation des unités fonctionnelles et des différents étages des pipelines a été garantie par la considération, dans le modèle sous forme de réseau de Petri stochastique, de transitions spécifiques, appelées **transitions synchronisées par l'horloge**, dont le tir est simultané et s'opère après un temps de cycle. De même, toutes les communications, entre la mémoire principale ou la table des constantes et les unités arithmétiques, s'effectuent en un temps de cycle (cf paragraphe 5.5.1). Ainsi, les communications et synchronisations n'ont fait l'objet d'aucune mesure.

Dans le cas de l'architecture ICAP/FPS264, une étude spécifique a permis de **définir des fonctions de coût pour les communications et les synchronisations** entre l'ordinateur-hôte et chaque AP d'une part, et entre APs d'autre part [Prost88b]. Des fonctions paramétrées ont tout d'abord été formulées à partir de la connaissance que l'on avait des mécanismes de communication et de synchronisation. Les coefficients de ces fonctions ont été déterminés par un **ajustement aux moindres carrés** [H] des fonctions aux mesures des temps d'exécution des communications et synchronisations étudiées (cf paragraphe 6.4.1). Les mesures ont été réalisées par des appels à l'horloge système.

Pour les communications, les fonctions prennent en compte un temps d'initialisation (ou temps de latence) et un temps de transmission (associé au débit des voies de communication).

Pour les communications hôte-AP, tous les appels à l'horloge système ont été effectués sur l'ordinateur-hôte afin de **pallier le problème des horloges asynchrones**. On aurait cependant pu réaliser les mesures en utilisant des appels à la fois sur l'hôte et l'AP et résoudre le problème des horloges asynchrones en appliquant l'une des deux méthodes introduites par A. Duda, G. Harrus, Y. Haddad et G. Bernard, qui permettent l'analyse globale des systèmes distribués à partir de traces d'événements locales [Duda86b]. Les principaux paramètres de la fonction de coût sont le nombre de données transmises, leur taille ainsi que le sens de transmission.

Pour les communications entre APs par l'intermédiaire des mémoires de masse (utilisées comme boîtes à lettres), les mesures ont été faites à partir d'appels à l'horloge système sur les APs. Pour la réception de message qui est bloquante, on a garanti que le message ait bien été envoyé par l'AP émetteur par une temporisation appropriée entre l'émission et la demande de réception. Les paramètres de la fonction de coût sont: la taille à transférer, le sens de transmission et le type de la mémoire de masse utilisée (locale ou globale).

Pour les accès aux mémoires de masse (utilisées comme mémoires partagées), les mesures ont été effectuées sur l'AP. La fonction de coût comporte, outre les paramètres de la fonction précédente, le nombre de blocs non contigus de données à transmettre. En effet, à chaque bloc de données contiguës correspond un accès physique à la mémoire de masse et donc un temps de latence propre.

Pour les synchronisations entre APs, on s'est intéressé uniquement aux temps nécessaires, **une fois tous les processus arrivés au point de synchronisation**, pour que le dernier processus soit libéré. Les mesures ont été effectuées sur chaque AP. Pour garantir que tous les processus soient effectivement arrivés au point de synchronisation, on a réalisé dans un premier temps une synchronisation forte (de type barrière) entre tous les processus avant de déclencher le chronomètre

et de réaliser la synchronisation qui nous préoccupait. Les paramètres de la fonction de coût sont le nombre de processus à synchroniser ainsi que le type de la mémoire de masse nécessaire au stockage des drapeaux de synchronisation.

L'ensemble de ces mesures a été **reproduit un grand nombre de fois** afin d'obtenir des valeurs moyennes et de pouvoir estimer la dispersion des mesures autour de ces valeurs moyennes [**I**]. Seules les mesures des communications hôte-AP ont fait preuve d'une grande dispersion, conditionnant quelque peu la qualité de la fonction de coût. Cette dispersion est imputable à une variation de la charge système importante sur l'ordinateur hôte, bien que les mesures aient toutes été réalisées en environnement dédié, afin justement de limiter ce phénomène. En effet, plusieurs processus systèmes sont nécessaires à la gestion des processus des programmes parallèles et entrent en compétition avec ces derniers pour l'attribution des processeurs de l'hôte, fonctionnant en temps partagé.

RESOLUTION DU MODELE

On construit le **modèle opérationnel** en simplifiant le modèle conceptuel, afin de pouvoir le résoudre si possible par une méthode analytique (basée sur la théorie des **réseaux de files d'attente** ou celle des **réseaux de Petri stochastiques**).

Diverses techniques de réduction des modèles analytiques sont communément employées. Parmi elles, citons les **méthodes d'agrégation** [**J**] et la **réduction de l'espace des états et de l'espace mémoire pour les réseaux de Petri stochastiques** [**K**].

La **méthode d'agrégation** a été introduite par H.A. Simon et A. Ando pour la résolution des grandes chaînes de Markov presque complètement décomposables [Simon61]. Cette technique consiste à analyser différents composants de la chaîne séparément, puis à combiner les solutions de ces composants pour obtenir une solution globale approchée de la chaîne initiale.

P.-J. Courtois a appliqué cette technique à l'analyse des systèmes informatiques modélisables sous la forme d'un réseau de files d'attente markovien presque complètement décomposable [Court77]. La méthode de Courtois consiste à faire apparaître, par permutations de lignes et de colonnes de la matrice stochastique de transition de la chaîne de Markov sous-jacente, m blocs diagonaux quasi-indépendants, c'est-à-dire tels que les éléments extérieurs aux blocs sont petits devant ceux des blocs. Chaque bloc diagonal est résolu séparément et les solutions des m blocs sont combinées pour former un nouveau système de m équations à m inconnues dont la solution est la

solution globale approchée de la chaîne initiale. Cette solution correspond aux probabilités des états du réseau de files d'attente en régime permanent.

La détermination du nombre d'agrégats ainsi que leur constitution peuvent bien sûr être arbitraires. Un algorithme de classification, basé sur la construction d'une dissimilarité entre lignes et colonnes de la matrice stochastique, a cependant prouvé tout son intérêt pour la détermination automatique des agrégats [Trico84,Tchue85].

Plus récemment, des méthodes itératives, proches de la méthode de Courtois, ont vu le jour. Elles sont communément appelées **méthodes itératives d'agrégation-désagrégation**. La phase d'agrégation correspond à la résolution de chaque bloc diagonal (appelé agrégat). La phase de désagrégation correspond à la résolution du système de m équations à m inconnues. Les deux phases se succèdent de manière itérative jusqu'à convergence. Notons que la convergence n'est pas vérifiée dans tous les cas. La méthode de F. Chatelin consiste à résoudre chaque agrégat par la méthode de la puissance [Chate84]. Cette méthode a été comparée avec succès à la méthode de Courtois pour la résolution d'un modèle du système radio téléphone automatique [Houei85]. Citons encore les méthodes de Y. Takahashi [Takah75], de R. Koury, D.F. McAllister et W.J. Stewart [Koury84], et de H. Vantilborgh [Vanti81], dont la convergence a été étudiée par W.-L. Cao et W.J. Stewart [Cao85].

Parallèlement à l'approche de ces méthodes d'agrégation que l'on peut qualifier de théorique, car basée sur la matrice stochastique de transition de la chaîne de Markov sous-jacente, une approche plus pratique de ces méthodes d'agrégation peut être appliquée directement à l'étude de systèmes réels par regroupement de files d'attente (ou de places) dont les interactions avec les files d'attente (ou les places) d'autres groupes sont limitées, voire inexistantes. Chaque groupe de files d'attente (ou de places) est remplacé par une file d'attente (ou une place) équivalente (phase d'agrégation), au sens où le définissent E. Gelenbe et G. Pujolle [Gelen82, p. 142-143]. L'avantage de cette approche est une détermination beaucoup plus aisée des agrégats, dans la mesure où les files d'attente (ou places) ont une signification physique par rapport au système réel. Ainsi, bien souvent, les agrégats correspondent à des unités fonctionnelles ou à des ressources physiques du système relativement indépendantes les unes des autres.

On appréhende déjà l'intérêt de cette approche pour les architectures parallèles, souvent composées d'unités fonctionnelles similaires (voire identiques). Ces unités fonctionnelles constituent en effet des agrégats de choix, d'autant plus que l'étude des agrégats correspondant à un même type d'unité fonctionnelle peut se réduire à celle d'un seul agrégat.

Cette approche de l'agrégation a été mise à profit lors de la modélisation des réseaux point-à-point de systèmes HP3000, où **chaque système HP3000 constitue un agrégat** que l'on résout séparément [Auber87]. Le temps de réponse de chaque agrégat pour le traitement de chaque requête utilisateur est ensuite considéré comme le temps de service moyen du délai représentant chaque noeud du réseau dans le modèle global pour cette même requête (cf paragraphe 3.4.2).

Pour le modèle du système FPS264 sous forme de réseau de Petri stochastique, une méthode itérative d'agrégation-désagrégation a été employée où **chaque unité fonctionnelle** (additionneur, multiplieur, mémoire principale, table des constantes) constitue un agrégat [Becke88].

Chaque itération est divisée en quatre phases. Chaque phase consiste à remplacer les sous-modèles de trois des quatre unités fonctionnelles par trois places "équivalentes" (le taux de la transition de sortie de chacune de ces places est égal à l'inverse du temps moyen de séjour, calculé à l'itération précédente, dans l'ensemble des places du sous-modèle de l'unité fonctionnelle correspondante). La résolution du modèle simplifié de chaque phase détermine pour l'unité fonctionnelle non agrégée le taux de la transition de sortie de la place "équivalente", représentant cette même unité fonctionnelle pour les phases ultérieures. Le processus itératif se termine lorsque la variation des taux de transition de sortie des places "équivalentes" est inférieure à un seuil suffisamment petit (cf paragraphe 5.5.1).

Un autre exemple de l'utilisation d'une méthode itérative d'agrégation-désagrégation est la modélisation du système radio téléphone automatique [Mazel88], où chaque relais constitue un agrégat.

Diverses techniques permettent de **réduire l'espace des états ainsi que l'espace mémoire** nécessaire au stockage du graphe des marquages des réseaux de Petri stochastiques.

Pour permettre une réduction du temps de génération du graphe des marquages et de l'espace mémoire nécessaire à son stockage, G. Chiola utilise des techniques de compilation, telles que le classement des marquages en cours de génération du graphe sous forme d'arbre binaire, un codage compact des marquages, l'élimination des places implicites, la maintenance d'une liste des transitions tirables [Chiol88].

De même, afin de réduire le graphe des marquages du modèle du système FPS264 (et assurer également la cohérence du modèle d'un point de vue sémantique), une modification du logiciel RDPS [Flori85] a été réalisée afin de rendre simultané le tir de toutes les transitions synchronisées par l'horloge [Becke88] (cf paragraphe 5.5.1).

La modélisation analytique ne peut malheureusement pas toujours s'appliquer. En effet, il est fréquent que la modélisation d'architectures parallèles doive prendre en compte des schémas complexes de synchronisation. La théorie des réseaux de files d'attente ne permet actuellement la résolution analytique que de schémas simples de synchronisation, tels que certains types de Fork-Join [Bacce86,Duda86a,Houei88]. La théorie des réseaux de Petri stochastiques peut, elle, prendre en compte des schémas complexes de synchronisation, mais elle est restreinte aux seuls temps de service exponentiels. De plus, tous les résultats qu'elle produit sont obtenus pour un fonctionnement du modèle en régime permanent et elle ne se prête pas à la modélisation de systèmes évanescents comme celui de l'exécution d'un programme d'application. C'est pourquoi, bien souvent, la simulation doit être employée. Dans ce cas, afin d'éviter un temps de résolution prohibitif, on s'évertue à réduire au maximum l'espace d'états du modèle par la considération d'un nombre minimum d'événements, en accord avec les finalités de l'étude.

VALIDATION

Pour valider le modèle, on procède à une **comparaison entre résultats obtenus par résolution du modèle et résultats de mesures**. Cette comparaison doit porter sur un **nombre maximum de critères de performance** [**L**]. Cependant, rares sont les systèmes parallèles disposant aujourd'hui d'outils suffisamment sophistiqués pour permettre les mesures requises [Denni87], et bien souvent la validation se limite à une comparaison de temps d'exécution.

Pour les architectures parallèles, une **démarche progressive de validation** [**M**] est souhaitable. On considère tout d'abord chaque programme d'application **en environnement dédié** [**N**], que l'on exécute sur un nombre croissant de processeurs. Si le système le supporte, on considère ensuite deux programmes d'application concurrents et on fait varier d'une part le nombre de processeurs exécutant chaque programme, d'autre part les contentions pour l'accès aux ressources entre les programmes. Puis, on s'intéresse à trois programmes d'application concurrents, et ainsi de suite.

On essaiera de **réduire au maximum le surcoût induit par l'outil de mesures** [**O**], notamment le code inséré directement dans les programmes d'application. En effet, ce surcoût peut entraîner des déséquilibres de charge supplémentaires entre tâches parallèles, responsables de performances dégradées que ne reflète bien sûr pas le modèle. Cela peut conduire à des conclusions erronées quant à la validité du modèle. Le phénomème inverse peut également se produire, où le surcoût viendrait compenser un déséquilibre de charge que le modèle ne détecte pas. C'est pourquoi,

si l'on ne dispose pas de moniteur, on limitera le nombre d'appels à l'horloge système pour la détermination de durées d'exécution afin de perturber le moins possible l'exécution des programmes d'application et on s'efforcera d'assurer une équi-répartition des appels parmi les différentes tâches parallèles.

Selon Denning et Adams III [Denni87], la qualité d'une validation devrait s'apprécier par la **reproductibilité des expérimentations** [P], au demeurant très difficile à assurer pour les architectures parallèles (cf paragraphe 6.6.3). De ce fait, plusieurs expérimentations identiques sont souhaitables, afin de permettre une analyse des résultats par valeurs moyennes. La **variabilité des mesures**, souvent caractérisée par l'écart type des résultats expérimentaux, permet de nuancer la comparaison avec les résultats de la modélisation. Malheureusement, le facteur temps bien souvent limite le nombre d'expérimentations et empêche une étude satisfaisante de la variabilité.

Pour les réseaux point-à-point de systèmes HP3000 (cf paragraphe 3.6), une **distinction entre sessions interactives et transferts de fichiers** a été considérée.

Pour les sessions interactives, le **temps de réponse moyen** de chaque requête interactive et le **débit moyen** de requêtes interactives traitées ont constitué les critères de performance. Les valeurs de ces critères, obtenues à partir du modèle, ont été comparées à celles qui ont été exhibées des rapports de réduction, produits par le moniteur logiciel des différents systèmes.

Pour les transferts de fichiers, seul le **temps de transfert global** a été considéré. La comparaison s'est faite là encore entre les résultats des simulations et certaines données des rapports de réduction.

Pour des problèmes de disponibilité du réseau-test, une seule série de mesures a été réalisée et la validation ne repose donc que sur un jeu de rapports de réduction.

Afin que l'analyse soit plus réaliste, une tâche de fond, présente sur chaque système du réseau-test, générait une charge système en temps CPU et accès disque évoluant dans le temps et comparable à celle qui avait pu être relevée sur les systèmes de réseaux réels.

Pour le modèle du calculateur scientifique FPS264 (cf paragraphe 5.6), la validation s'est effectuée en **plusieurs étapes**.

1. Tout d'abord, on a vérifié la **correction du modèle de contrôle**, représentant la distribution des différents degrés de parallélisme du programme d'application.

2. Puis, la **méthode de classification des programmes d'application** par une analyse en composantes principales a été validée en contrôlant que les critères de performances obtenus par le modèle pour deux programmes appartenant à la même classe étaient similaires.

3. La **convergence de la méthode itérative d'agrégation-désagrégation** utilisée a été constatée sur différents cas de figure.

4. Enfin, la validité de la méthode d'agrégation et du modèle a été étudiée en comparant le **temps d'exécution d'un programme représentatif de chaque classe**, mesuré sur le système réel, avec le temps de réponse donné, pour le même programme, par le modèle de files d'attente, dérivé du réseau de Petri stochastique.

En ce qui concerne le système ICAP/FPS264 (cf paragraphe 6.6), une **approche progressive** a été adoptée pour la validation du modèle. A la différence du système précédent, qui est mono-utilisateur et donc ne peut exécuter qu'un programme d'application à la fois, le système ICAP/FPS264 est multi-utilisateur et conçu pour gérer plusieurs programmes parallèles concurrents. La validation s'est effectuée en comparant le **temps global d'exécution de chaque programme parallèle**, mesuré sur le système, avec celui estimé par le modèle, pour les nombreux scénarios suivants:

- Dans un premier temps, on a considéré **un seul programme parallèle** s'exécutant sur le système, **en faisant varier le nombre d'APs** qu'il utilisait (de 1 à 10). Plusieurs programmes d'application, différant par le mode d'utilisation des mémoires de masse, ont été employés. L'un utilisait uniquement les canaux de communication entre l'hôte et les APs, sans aucun accès aux mémoires de masse. Deux autres avaient recours au mode message. Les deux derniers employaient le mode partagé.

- Dans un deuxième temps, on a pris en compte **deux programmes parallèles concurrents, sans contention pour l'accès aux APs** tout d'abord, puis **entrant en concurrence pour l'accès aux APs**.

- Ensuite, on est passé à des scénarios avec trois programmes parallèles concurrents, sans contention pour les APS, puis avec contention.

- Finalement, on a considéré des scénarios avec successivement quatre, cinq, et même huit programmes parallèles concurrents avec contention pour les APs.

Les scénarios à plusieurs programmes parallèles concurrents différaient les uns des autres par le mode d'utilisation des mémoires de masse et le nombre d'APs de chaque programme.

UTILISATION DU MODELE

Nous rappelons que le modèle peut être a priori employé suivant deux modes d'utilisation: le **mode déterministe** pour des études de cas bien précis et pour la validation, et le **mode probabiliste** pour des études exploratoires générales, permettant une prédiction des performances du système réel.

Bien sûr, dans le cas des architectures parallèles, le mode d'utilisation probabiliste n'est pas sans soulever quelques problèmes.

On veillera tout d'abord à **respecter les contraintes structurelles de l'architecture**, lors de l'allocation aléatoire des ressources, c'est-à-dire que l'on évitera d'allouer à un même processus des ressources qui soient incompatibles les unes avec les autres. Notamment, on s'attachera à maintenir une parfaite compatibilité entre les processeurs alloués et les voies de communication qui relient ces processeurs.

Lors de la génération aléatoire des événements corrélés, associés aux communications et aux synchronisations entre tâches parallèles, on veillera à **respecter la sémantique des programmes** et notamment à **prévenir l'occurrence d'étreintes fatales** [**R**].

Ainsi, pour les communications, à chaque envoi de message devra correspondre une réception du même message et inversement. Pour les synchronisations, tout processus impliqué dans le schéma de synchronisation devra exécuter, dans un temps fini, une requête de synchronisation avec le(s) processus adéquat(s).

Utilisation probabiliste ne doit donc pas signifier utilisation hasardeuse, et le risque d'une telle utilisation est grand pour les architectures parallèles si les règles énoncées ci-dessus ne sont pas respectées. Voici donc une technique originale assurant le respect de ces règles.

En ce qui concerne les communications, chaque fois qu'un envoi de message est généré, un processus destinataire est déterminé aléatoirement, selon une loi de répartition des différents destinataires possibles. La réception de ce message par le destinataire est alors générée dans le futur. On peut pour ce faire considérer que la réception s'effectue à l'issue d'un délai exprimé par un nombre de communications intermédiaires du processus récepteur, suivant une loi de probabilité. On procéde

de manière duale si le premier événement de la communication est une réception de message. Enfin, un pourcentage exprime la probabilité pour que le premier événement d'une communication soit un envoi de message. Pour les communications intermédiaires, on prend garde à ce que l'envoi de message (s'il est bloquant) ou la réception de message ne s'opère pas à destination ou en provenance d'un processus déjà bloqué. Un **graphe orienté des processus en attente** doit être maintenu et à chaque génération d'une nouvelle communication, on vérifie que la mise à jour de ce graphe ne produit pas un cycle. Tout cycle révèle une étreinte fatale et donc la nouvelle communication doit être abandonnée au profit d'une autre, que l'on génère selon le même procédé. Si aucune autre ne peut convenir (toute communication avec un autre processus entraînant un cycle dans le graphe), on supprime les communications intermédiaires.

Pour les synchronisations, on utilise un **tableau de sémaphores** $C(i,j)$ où i et j désignent une paire de processus et on adopte une approche analogue à celle des communications. Chaque fois qu'un processus i génère aléatoirement la fonction $P(i,j)$ ou $V(i,j)$ à destination d'un processus j, la fonction duale $V(j,i)$ ou $P(j,i)$ est générée par le processus j après un délai exprimé par un nombre aléatoire d'appels intermédiaires à des fonctions $P(j,k)$ ou $V(j,k)$. A chaque appel de fonction P ou V, le graphe orienté des processus en attente est mis à jour. Si l'appel d'une fonction P génère un cycle dans le graphe, l'appel est abandonné au profit d'un autre appel. Un appel est dans ce cas toujours possible, car la fonction V n'est pas bloquante.

Une illustration détaillée de cette méthode originale de génération aléatoire d'événements corrélés est présentée au paragraphe 6.7.1.

Lors d'une analyse de prédiction de performances d'un système réel, conçu pour être utilisé en environnement non dédié (comme le système ICAP par exemple), nous préconisons la démarche suivante [**T**]:

1. tout d'abord, considérer un job unique, modélisé de manière relativement déterministe, et étudier le comportement du système en environnement dédié (**étude sans charge**),

2. dans un deuxième temps, considérer le même job, mais cette fois-ci en compétition pour l'accès aux ressources du système avec d'autres jobs, modélisés aléatoirement, et reflétant, autant que faire se peut, la charge habituelle moyenne du système réel (**étude avec charge**).

Le lecteur peut constater que notre approche est un compromis entre le mode purement déterministe et le mode purement probabiliste, présentant l'avantage de faciliter l'analyse des

résultats avec charge par une comparaison avec les résultats sans charge. Une application de cette démarche est proposée au paragraphe 6.7.

Le dernier point méthodologique que nous voudrions souligner est la **prise en compte de la notion de processeur virtuel** [U] dans la détermination des critères de performances, tels que l'accélération ou l'efficacité de l'exécution parallèle d'un programme sur une architecture parallèle constituée de **processeurs hétérogènes** (cf paragraphe 4.5.2).

2.3 Tableau Récapitulatif

Les points essentiels abordés au paragraphe précédent sont résumés dans le tableau ci-dessous. Rappelons que les lettres majuscules placées entre crochets référencent les aspects les plus importants de la méthodologie de modélisation.

Phase	Points Importants de la Méthodologie
Conception du Modèle	- **Modèle conceptuel très précis** du système: + modèle de l'architecture [**A**] + modèle des programmes [**B**]: x tâches de calcul o **finesse** de modélisation [**D**] o génération du **degré de parallélisme** [**E**] o notion de **processeur virtuel** en cas de processeurs physiques hétérogènes [**F**] x communications et synchronisations o temps de latence et temps de transmission [**S**] - Recours à l'**analyse des données** pour classifier les programmes [**C**]
Estimation et Mesure des Paramètres d'Entrée	- Privilégier l'emploi de **moniteurs hybrides distribués** [**G**] - Possibilité d'employer de simples appels à l'horloge système + **problème des horloges asynchrones** pour la mesure des temps de communication entre systèmes: x mesures sur un seul système de communications bi-directionnelles x mesures sur les deux systèmes de communications uni-directionnelles (méthodes d'ajustement d'estampilles de Duda et al.) - Recours à des **méthodes d'ajustement aux moindres carrés** [**H**] - **Reproduire les mesures** plusieurs fois afin d'apprécier la dispersion des résultats autour de valeurs moyennes [**I**]

Phase	Points Importants de la Méthodologie
Résolution du Modèle	- Simplification du modèle conceptuel pour obtenir le **modèle opérationnel** - Diverses techniques de réduction: + grand intérêt des **méthodes d'agrégation** [**J**] + techniques de réduction de l'espace des états des réseaux de Petri stochastiques [**K**] - Résolution proprement dite (méthodes analytiques ou simulation)
Validation	- Comparaison entre résultats du modèle et valeurs mesurées sur le système + prise en compte d'un nombre maximum de critères de performances [**L**] + **démarche progressive** [**M**] + mesures sur **système dédié** [**N**] + étudier la **reproductibilité des mesures** [**P**] - **Limiter l'impact de l'outil de mesure** [**O**] sur l'exécution des programmes + supprimer les insertions de code inutiles (impressions surabondantes) + risque de déséquilibres de charge induits
Utilisation du Modèle	- Deux modes d'utilisation du modèle [**Q**]: + déterministe (étude de cas, validation) + **probabiliste** (étude exploratoire générale) x règles à respecter pour la **génération aléatoire d'événements corrélés** [**R**] (communications, synchronisations) - Prédiction des performances d'un système réel non dédié [**T**]: + comparaison entre étude sans charge et étude avec charge + compromis entre modes d'utilisation déterministe et probabiliste - Prise en compte de la notion de processeur virtuel: + calcul d'accélération et d'efficacité avec des processeurs hétérogènes [**U**]

Afin d'illustrer cette méthodologie, les chapitres 3, 4, 5 et 6 de ce rapport de thèse détaillent la modélisation ou les mesures de performances réalisées pour chacune des quatre architectures parallèles introduites précédemment:

1. les réseaux point-à-point de systèmes HP3000 (architecture distribuée de type réseau à distance) [Auber87],

2. un réseau local d'IBM PC (architecture distribuée de type réseau local) [Prost88c],

3. le calculateur scientifique Floating Point Systems 264 (assimilable à une architecture MIMD de type commuté à mémoire partagée) [Becke88],

4. le système expérimental ICAP/FPS264 de IBM Kingston (architecture MIMD commutée de type hybride) [Prost88a,Prost88b,Prost89a,Prost89b,Prost89c,Prost89d].

CHAPITRE 3

Réseaux Point-à-point de Systèmes HP3000

Ce chapitre présente la première illustration de la méthodologie de modélisation des architectures parallèles développée au chapitre précédent.

Cette illustration décrit un outil de modélisation de réseaux point-à-point de systèmes Hewlett-Packard HP3000 sous forme de réseau de files d'attente, en vue de l'évaluation de leurs performances.

Cet outil a été conçu pour être utilisé **en conjonction avec les outils Hewlett-Packard de mesures** (HPSPEP: System Performance Evaluation Project) **et d'évaluation de performances** (HPCAPLAN: CAPacity PLANning et Kasandra) déjà disponibles pour mesurer et prédire les performances des systèmes HP3000 isolés.

La modélisation est basée sur une **méthode d'agrégation en deux phases** [J]. La première phase réalise l'étude isolée de chaque système HP3000 présent sur le réseau (appelé site dorénavant), à l'aide des outils Hewlett-Packard mentionnés précédemment. La deuxième phase effectue l'analyse globale du réseau point-à-point en remplaçant dans le réseau de files d'attente chaque site par un délai équivalent.

Ce chapitre développe chacune des phases de la méthodologie de modélisation pour cette étude de cas. Faute de temps, aucune réelle prédiction de performances n'a cependant pu être réalisée.

3.1 Formulation des Objectifs

Le besoin en réseaux de communication de données est de plus en plus important du fait du développement de l'informatique répartie, qui réalise une distribution de l'intelligence sur le marché local.

Les concepteurs de tels réseaux ont pour souci permanent la recherche d'une solution optimale pour le client, c'est-à-dire d'une solution la mieux adaptée à ses besoins.

Cette optimisation procède en évaluant, de manière prédictive, les performances du futur réseau, en termes de temps de réponse, de charge et de débit. Le coût associé est ensuite calculé. En fonction de ce dernier, le client peut vouloir modifier quelque peu le réseau afin d'adapter ce coût à son budget. Une nouvelle évaluation de performances est alors nécessaire pour appréhender les dégradations ou améliorations éventuelles apportées par ce changement. Ce processus de modification-évaluation peut se répéter plusieurs fois avant que client et concepteur considèrent avoir atteint le meilleur compromis entre performances et coût.

Pour des réseaux conséquents, le nombre de paramètres de performances à prendre en compte est vite prohibitif au point de nécessiter le recours à un outil informatique d'évaluation de performances.

Afin de répondre à ce besoin croissant de la part des concepteurs, un outil prototype de modélisation des réseaux point-à-point de systèmes HP3000 a été développé. Le choix des systèmes HP3000 sur d'autres systèmes Hewlett-Packard a été guidé par l'importance de ces systèmes dans la base installée, en France comme à l'échelle mondiale, et par le nombre croissant de réseaux point-à-point interconnectant de tels systèmes.

La démarche naturelle a été d'essayer de tirer bénéfice des outils de mesure et d'évaluation de performances dont Hewlett-Packard disposait pour l'analyse des performances des systèmes HP3000 isolés. Ceci a à la fois guidé nos choix de modélisation et le mode de détermination des paramètres d'entrée du modèle.

3.2 Analyse des Réseaux Point-à-point Etudiés

Les réseaux étudiés (cf figure 3.1) sont constitués de plusieurs systèmes commerciaux HP3000 auxquels sont connectés des terminaux. Ces systèmes sont interconnectés par des lignes point-à-point "full duplex", le niveau liaison du protocole de communication étant conforme à l'avis X.25 du CCITT, et **aucune hiérarchie** n'existe entre les sites. Les transferts de données entre sites sont gérés par des processeurs spécialisés appelés "Intelligent Network Processors" (INP). Sur chaque

site il y a un INP par ligne point-à-point connectée au système HP3000, qui a la charge à la fois des émissions et des réceptions de messages, sans aucune priorité entre ces deux types d'opérations.

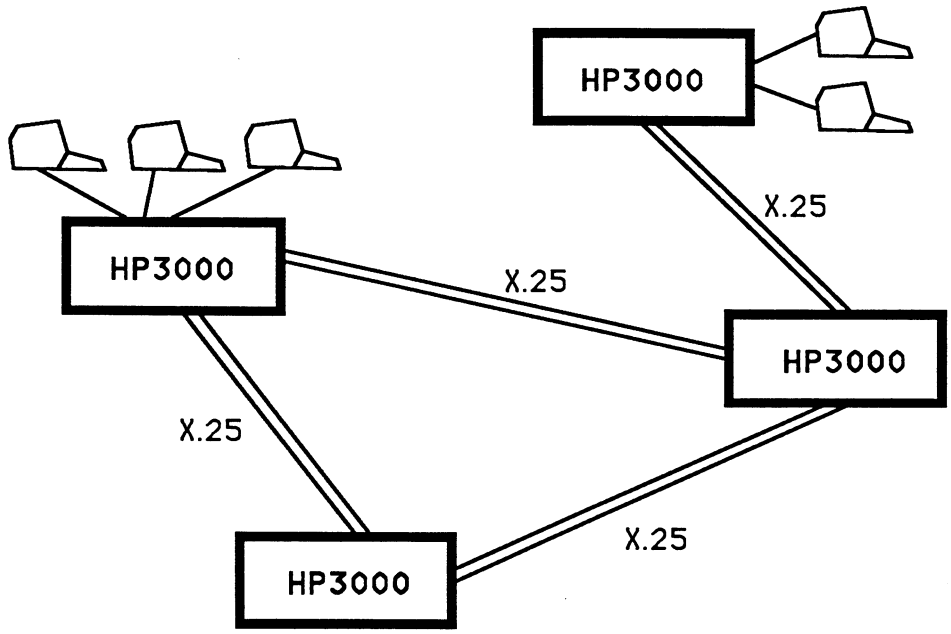

Figure 3.1: Exemple de Réseau Point-à-point de Systèmes HP3000

Un utilisateur en mode interactif peut exécuter une application soit localement sur son site de connexion (**mode local**), soit à distance (**mode distant**) sur un site accessible depuis son site de connexion au travers du réseau, par un lien direct ou par l'intermédiaire de un ou plusieurs **sites de transit**.

Pour une session interactive utilisateur, on peut donc distinguer trois types de sites (cf figure 3.2): le site de connexion, le(s) site(s) de transit et le site de destination (là où l'application est réellement exécutée).

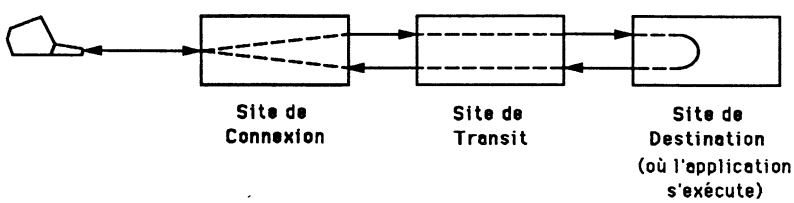

Figure 3.2: Différents Types de Sites

En mode distant, l'utilisateur doit, avant le lancement de son application, d'une part ouvrir une voie logique sur chaque ligne physique, qui sera empruntée par ses requêtes et les réponses émanant du site de destination, d'autre part établir une session sur chacun des sites visités par l'application.

Parallèlement à cette activité interactive, le réseau est également utilisé pour des transferts de fichiers entre sites connectés par une ligne directe uniquement.

Le lecteur peut, s'il désire plus d'information concernant l'architecture des systèmes HP3000 et des réseaux point-à-point les reliant, se reporter à [Prost86, chapitre 2].

3.3 Description du Modèle

Comme tout modèle d'une architecture parallèle, le modèle des réseaux étudiés est composé de deux composantes, le modèle de l'architecture proprement dite et le modèle des programmes, en l'occurrence, le **modèle du réseau** [**A**] et le **modèle des applications** [**B**], interactives ou transferts de fichiers.

3.3.1 Modèle du Réseau [A]

Comme je l'ai mentionné plus haut, l'approche que nous avons adoptée pour la modélisation a été dictée par l'existence chez Hewlett-Packard d'outils de mesure et d'évaluation de performances des systèmes HP3000 isolés.

Une **méthode d'agrégation** [**J**] simple, considérant chaque système HP3000 présent sur le réseau comme un agrégat, nous a donc paru appropriée et nous a conduits à considérer dans le modèle global du réseau chaque site comme une seule station.

Cette station **Site(i)** est une station à serveur infini (ou délai) dont le temps de service exponentiel moyen est égal au temps de réponse moyen déterminé par la résolution du modèle de l'agrégat qu'elle représente (cf paragraphe 3.4.2).

Pour être plus exact, un site est représenté non seulement par la station **Site(i)** (modélisant le système HP3000 isolé), mais aussi par autant de stations **INP(i,j)** que de lignes point-à-point

connectées au site (modélisant les INP qui gèrent tous les échanges du site avec l'extérieur).

Ces stations **INP(i,j)** sont des stations exponentielles à un seul serveur et à discipline de file FCFS ("First Come First Served"), pour bien rendre compte de l'absence de priorité entre émissions de messages et réceptions de messages et de la file d'attente unique occupée par les messages en attente.

La ligne point-à-point, "full duplex", reliant les sites **Site(i)** et **Site(j)** est modélisée par deux stations **Ligne(i,j)** et **Ligne(j,i)**. Ces stations sont des délais dont le temps de service moyen est la somme d'un délai de propagation constant (fonction uniquement de la longueur de la ligne) et d'un temps de transmission exponentiel (fonction de la longueur du message à transmettre et de la vitesse de transmission de la ligne). La loi exponentielle est communément employée pour des délais de transmission de messages en l'absence d'information complémentaire.

L'ensemble des terminaux connectés aux divers sites du réseau est modélisé par une seule station **Réflexion**, à serveur infini, dont le temps de service exponentiel moyen est le temps de réflexion moyen d'un utilisateur entre la réception d'une réponse système et la soumission d'une nouvelle requête interactive. Cette station n'est bien sûr pas visitée par les clients correspondant aux transferts de fichiers. L'emploi d'un délai est justifié par le fait qu'à chaque requête utilisateur correspond une seule réponse système immédiatement servie par l'utilisateur dès sa visualisation au terminal.

La figure 3.3 représente le modèle, sous forme de réseau de files d'attente ouvert, du réseau de test utilisé lors de la phase de validation et présenté au paragraphe 3.6. Ce modèle, bien que très simple, est cependant général, car il comporte les trois types de sites que l'on peut rencontrer sur un réseau quelconque (site de connexion, site de transit et site de destination). Précisons également que la modélisation est elle aussi tout à fait générale et peut prendre en compte des réseaux aussi complexes soient-ils. Nous ne présentons ici qu'un exemple simple afin qu'il soit facilement représentable.

3.3.2 Modèle des Applications [B]

Le modèle des applications représente les clients qui vont circuler à l'intérieur du réseau de files d'attente modélisant le réseau de systèmes HP3000.

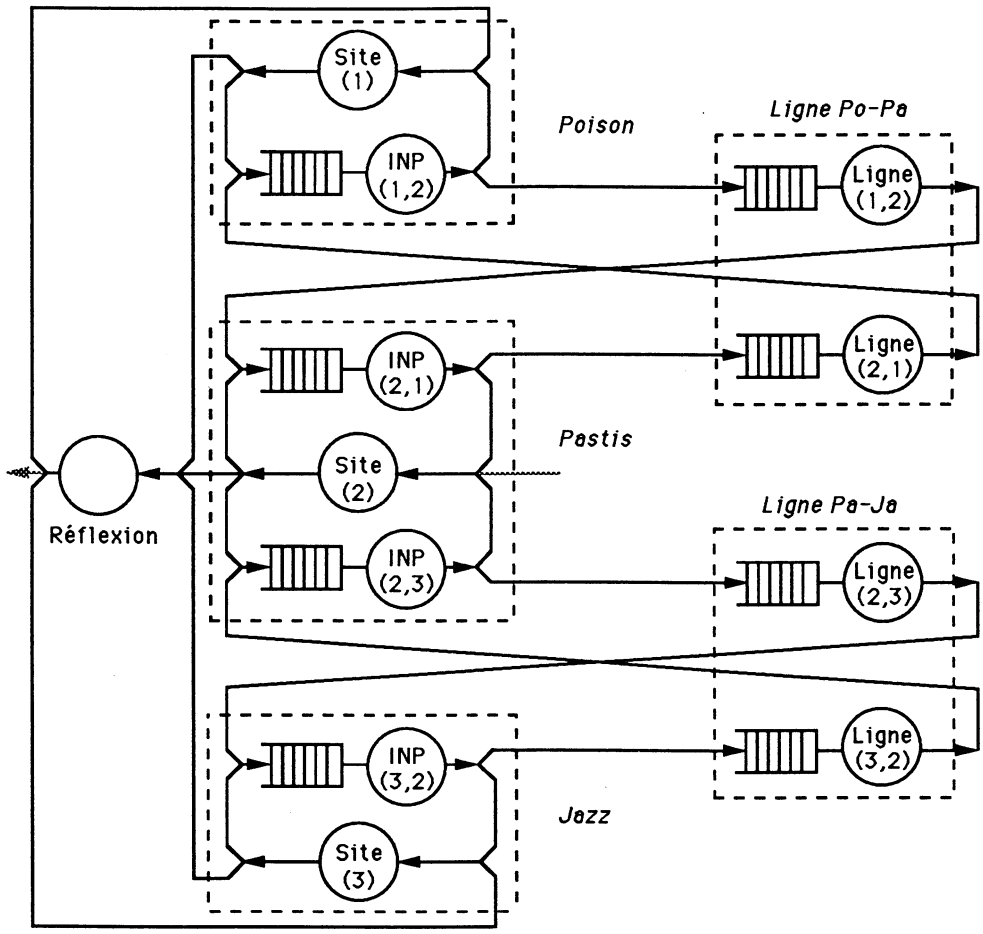

Figure 3.3: Modèle du Réseau de Test

Pour les applications interactives, un client est une requête utilisateur, lorsqu'il circule du site de connexion vers le site de destination, et une réponse du site de destination, au retour. L'association requête utilisateur - réponse système (appelée transaction) constitue un et un seul client. Ce client est ensuite géré par l'utilisateur qui réfléchit avant de lancer une nouvelle requête, représentée toujours par le même client. Il y a donc autant de clients interactifs que d'utilisateurs à un terminal sur le réseau.

Ces clients interactifs sont regroupés en classes définies comme suit: toutes les requêtes utilisateur émanant du même site de connexion et exécutant la même application sur le même site de destination constituent une seule classe de clients.

Les principaux paramètres associés à chaque classe de clients interactifs sont:

- la liste des sites visités,
- le temps CPU moyen et le nombre moyen d'accès disque pour servir une transaction respec-

tivement pour le site de connexion, le(s) site(s) de transit et le site de destination,

- le nombre moyen par transaction de caractères transmis sur les lignes à l'aller et au retour,
- le temps de réflexion moyen de l'utilisateur,
- et le nombre moyen de transactions constituant une session interactive.

Pour les transferts de fichiers, un client est un bloc de transfert (de taille fixe égale à 4000 octets, imposée par l'application DSCopy réalisant de tels transferts) à l'aller (du site source au site destination) et un message d'acquittement au retour. On néglige ici les réémissions de messages causées par des erreurs de transmission, dont la probabilité est inférieure à 0,2%.

A chaque transfert de fichier correspond donc un seul client qui constitue sa propre classe.

Les principaux paramètres associés à cette classe sont:

- les sites source et destination,
- le temps CPU moyen et le nombre moyen d'accès disque pour traiter un bloc de transfert respectivement pour le site source et le site destination,
- et la taille du fichier à transférer.

3.4 Estimation et Mesure des Paramètres d'Entrée

Pour estimer et mesurer la plupart des paramètres d'entrée nécessaires à la résolution du modèle, nous avons eu recours aux outils de mesure et d'évaluation de performances que Hewlett-Packard a développés pour les systèmes HP3000 pris isolément.

Ce paragraphe débute par une brève description de ces outils. Le lecteur pourra obtenir de plus amples détails en se référant à [Prost86, p. 6.2-6.7] ou aux manuels de référence cités dans la bibliographie.

Dans un deuxième temps, la modélisation de chaque site considéré comme un agrégat du modèle global est détaillée, ainsi que la détermination des paramètres d'entrée nécessaires à la résolution du sous-modèle ainsi obtenu. Rappelons que cette résolution permettra d'évaluer le

temps de réponse moyen de chaque site pour le traitement des clients de chaque classe, ce temps de réponse étant fonction de la position du site pour les clients considérés (site de connexion, site de transit ou site de destination). Ce temps de réponse servira, pour la classe de clients considérée, de temps de service moyen de la station à serveur infini remplaçant le sous-modèle dans le modèle global conformément à la méthode d'agrégation présentée au paragraphe précédent.

Enfin, ce paragraphe précise l'origine des autres paramètres d'entrée du modèle global. Pour une explication plus détaillée de leur détermination, le lecteur est invité à se reporter à [Prost86, p. 6.11-6.14].

3.4.1 Outils de Performance de Hewlett-Packard

Dans le souci constant d'adapter au mieux la configuration de ses systèmes aux besoins de ses clients, Hewlett-Packard a développé toute une gamme d'outils de mesure et d'évaluation de performances des systèmes HP3000, pris isolément. Ces outils sont conçus pour être utilisés conjointement. Le premier outil, **SPEP** [Spep], est un moniteur logiciel permettant l'échantillonnage de nombreuses données système. L'ensemble de ces données peut ensuite être géré et réduit par le second outil, **CAPLAN** [Capla]. Le troisième outil, **Kasandra** [Kasan], est un outil de modélisation et d'évaluation de performances des systèmes HP3000 qui utilise les données réduites fournies par CAPLAN comme paramètres d'entrée nécessaires à la résolution de son modèle sous forme de réseau de files d'attente sous-jacent.

System Performance Evaluation Project

SPEP [Spep] est un moniteur logiciel adapté aux systèmes HP3000 en tant que systèmes fermés. Comme le montre la figure 3.4, son utilisation est divisée en trois étapes successives.

La première étape, ou étape de **collection de données**, lance plusieurs utilitaires destinés à recueillir, à partir d'un échantillonnage réalisé toutes les minutes, toute l'information relative aux entrées-sorties, à l'activité des disques et du CPU et aux lignes de communications. La durée de cette phase est fixée par l'expérimentateur (dans le cas de nos mesures, elle a été fixée à 20 minutes afin de ne pas saturer l'espace disque qui nous était alloué).

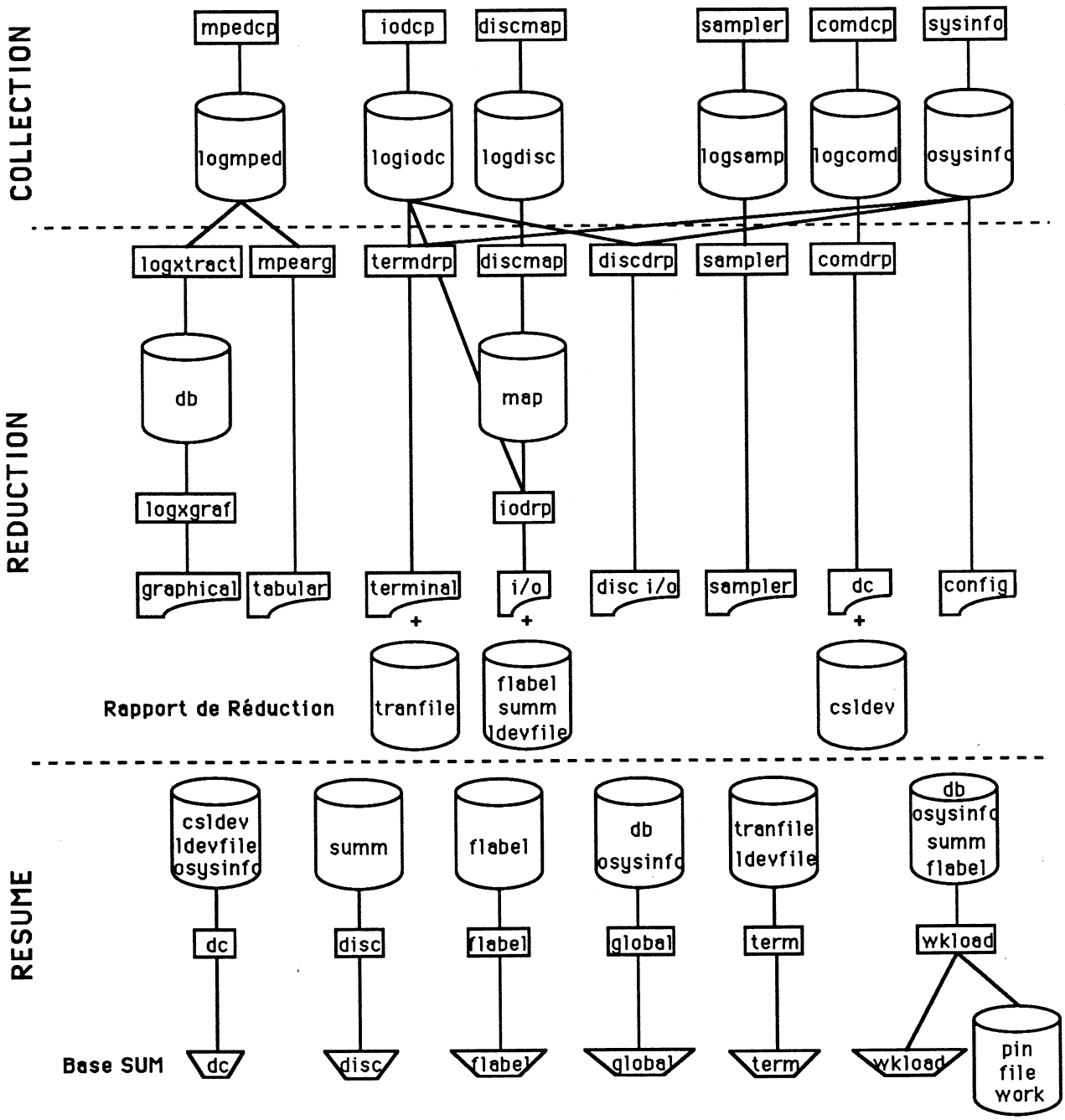

Figure 3.4: Synopsis de SPEP

La seconde étape, ou étape de **réduction**, condense l'information réunie lors de l'étape précédente et produit un rapport, appelé **rapport de réduction**, plusieurs fichiers (**tranfile**, **flabel**, **summ**, **ldevfile**, **csldev**) et une base de données (**db**). Le rapport de réduction contient de nombreuses tables et figures représentant l'ensemble de l'information recueillie lors de la collection de données.

La dernière étape, ou étape de **résumé**, crée, à partir des fichiers et de la base de données générés lors de la réduction, la base de données **SUM**, ainsi que quelques fichiers (**pin**, **file**, **work**) qui contiennent des informations essentielles telles que le temps de réponse moyen et le temps de réflexion moyen de chaque session interactive. La base de données SUM, quant à elle, est composée de plusieurs ensembles de données (a priori indépendants), chacun d'entre eux étant organisé selon une structure hiérarchique en plusieurs domaines. Des liens peuvent ensuite être créés entre ensembles de données et domaines, afin d'établir des relations spécifiques entre les composants de la base de données. La base de données SUM est exclusivement prévue pour être employée comme point d'entrée de l'outil CAPLAN.

Capacity Planning

CAPLAN [Capla] a été développé par Hewlett-Packard pour proposer à l'expérimentateur un gestionnaire automatique de la base de données SUM générée par SPEP. Ce gestionnaire a pour tâche de produire un rapport lisible, présentant les performances du système étudié pour chaque classe d'applications, que l'expérimentateur aura préalablement définie, à partir des applications présentes sur le système lors de la phase de collection.

Ce rapport ressemble au rapport de réduction fourni par SPEP, mais il présente l'avantage de scinder les informations entre les différentes classes d'applications. Les performances obtenues sont le débit du système, le nombre d'entrées-sorties disque et le temps CPU consommé par transaction, le taux d'utilisation du CPU, des disques, le temps de réflexion des utilisateurs à leurs terminaux et le temps de réponse du système. Pour chacune de ces performances, les deux premiers moments de ces variables aléatoires sont calculés et ils seront d'un grand intérêt pour la phase de modélisation réalisée par Kasandra (cf infra).

CAPLAN réalise également un bon interface avec l'outil analytique de modélisation sous forme de réseau de files d'attente et d'évaluation de performances des systèmes HP3000, Kasandra, en exprimant, dans la syntaxe de cet outil, l'ensemble des paramètres d'entrée du modèle du système.

Kasandra

Kasandra [Kasan] est un outil d'évaluation des performances des systèmes HP3000, basé sur la méthode "Mean Value Analysis" introduite par Reiser et Lavenberg [Reise80]. Kasandra utilise un modèle des systèmes HP3000 sous forme de réseau de files d'attente, dont les stations sont le CPU, les disques et les terminaux.

Trois classes de clients sont supportées: **terminal**, **transaction** et **batch**. La classe **terminal** correspond aux transactions interactives générées par les utilisateurs connectés au système. La classe **transaction** est une classe ouverte (les clients sont introduits dans le modèle en début de traitement et en sortent en fin de traitement), constituée par les transactions générées à l'extérieur du système (telles que toutes les transactions utilisateur en mode distant sur les sites de transit et les sites de destination). La classe **batch** correspond aux jobs s'exécutant en batch ou en tâches de fond, caractérisés par un niveau de priorité inférieur.

Comme nous l'avons dit précédemment, tous les paramètres d'entrée nécessaires à la résolution analytique du modèle du système sont fournis directement par CAPLAN. Cependant, l'utilisateur a la possibilité de modifier certains d'entre eux pour voir l'impact de ce changement sur les performances du système. En sortie, Kasandra fournit le temps de réponse et le débit du système, le taux d'utilisation des stations du modèle, globalement et par classe de clients. Kasandra peut en outre produire de nombreuses courbes représentant la variation de chaque critère de performance en fonction de celle d'un ou de deux paramètres d'entrée du modèle.

3.4.2 Modélisation des Agrégats du Modèle Global [J]

Notre intention première pour modéliser chaque site présent sur le réseau point-à-point étudié, soit chaque agrégat du modèle global, était d'utiliser toute la gamme d'outils présentée ci-dessus pour obtenir grâce à Kasandra un modèle de chaque site et les critères de performances associés, et notamment les deux premiers moments du temps de réponse de chaque système pour chaque classe de clients considérée (interactifs ou transferts de fichiers).

Malheureusement, à l'époque où les mesures ont été conduites, nous avons été dans l'impossibilité d'utiliser l'outil CAPLAN assurant l'interface entre la base de données SUM produite par SPEP et le modèle de chaque système construit par Kasandra. Conservant néanmoins la même

optique (et pensant que le problème de version de système d'exploitation pourrait être levé dans le futur), nous avons décidé de substituer au modèle, que Kasandra aurait fourni pour chaque système, un modèle fort grossier sous forme de réseau de files d'attente (cf figure 3.5). Le manque de raffinement du modèle est essentiellement dû à l'extrême difficulté que nous avons rencontrée pour déterminer les paramètres d'entrée du modèle à partir du rapport de réduction produit par SPEP (difficulté inexistante si l'emploi de CAPLAN avait pu être possible). En particulier, nous n'avons pas pu dériver les deuxièmes moments de ces paramètres, et nous avons donc eu recours à des serveurs exponentiels. Pour cette même raison, la station équivalente sur le plan des performances au modèle de l'agrégat, et remplaçant celui-ci dans le modèle global, aura une loi exponentielle pour loi des services.

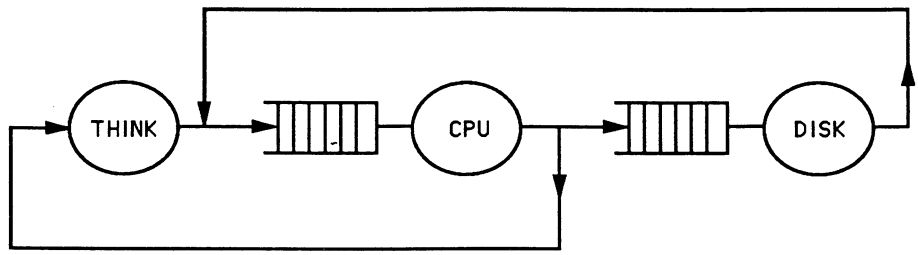

Figure 3.5: Modèle de Chaque Agrégat

Trois stations de service composent le modèle de chaque agrégat:

- la station **CPU** représente l'unité centrale du système; elle est à serveur unique et sa discipline de file est FCFS,
- la station **DISK** représente l'ensemble des unités de disque présentes sur le système; elle est à serveur unique et sa discipline de file est également FCFS,
- la station **THINK** modélise soit le temps de réflexion utilisateur si le système modélisé est un site de connexion pour la classe du client interactif en cours de service, soit le délai séparant le départ du client du système et son premier retour au système dans les autres cas; cette station est à serveur infini (délai).

Précisons également que les classes de clients sont exactement celles définies pour le modèle global, ce qui est nécessaire pour pouvoir appliquer la méthode d'agrégation.

Les principaux paramètres d'entrée du modèle sont le temps CPU moyen consommé et le nombre moyen d'accès disque réalisés par une transaction interactive ou par un bloc de transfert

pour chaque classe de clients, de même que le temps de réflexion moyen d'un utilisateur interactif, si le système est un site de connexion, ou le délai séparant deux visites du même client, dans les autres cas. Tous ces paramètres sont obtenus par synthèse des rapports de réduction produits par l'outil SPEP, qui doit être lancé sur chaque système présent sur le réseau. Cette synthèse est nécessaire lorsque les paramètres dépendent de la position du système relativement à la classe de clients considérée (site de connexion, site de transit ou site de destination). Dans ce cas, elle consiste en une moyenne des données fournies par chaque rapport de réduction, pondérée par le nombre de clients servis sur chaque site pour la classe de clients considérée.

Nous n'entrerons pas plus en détail dans la détermination des paramètres d'entrée du modèle de chaque agrégat. Pour plus d'information, le lecteur est invité à se reporter à [Prost86, p. 6.15-6.18 + annexes].

Quant au cheminement d'un client dans le modèle, il visite tout d'abord la station CPU pour une durée égale au temps moyen CPU consommé divisé par le nombre d'accès disque + 1. Ensuite, il effectue autant de visites à la station DISK et à la station CPU que le nombre d'accès disque. Il se rend alors à la station THINK, pour une durée égale au temps de réflexion moyen, si le système est son site de connexion, ou égale à la durée de ses demandes de service à l'extérieur du système avant son retour à ce système, dans les autres cas.

Enfin, la durée moyenne de chaque accès disque est estimée à 20 milli-secondes, en se basant sur l'information des constructeurs et sur certaines mesures de performances réalisées par le centre de recherche Hewlett-Packard de Cupertino.

3.4.3 Origine des Autres Paramètres d'Entrée du Modèle Global

Les autres paramètres d'entrée du modèle global, à savoir le temps de réflexion moyen d'un utilisateur pour les sessions interactives, le nombre moyen de caractères transmis sur les lignes de communication à l'aller et au retour par transaction, ainsi que le nombre moyen de transactions constituant chaque session interactive, sont déterminés là encore par une synthèse des rapports de réduction issus de SPEP (cf [Prost86, p. 6.12]).

Quant au temps de service moyen des stations **INP(i,j)**, il nous a été fourni par le centre de recherche Hewlett-Packard de Cupertino. Précisons seulement que ce temps inclut, pour les

émissions de messages, le temps de synchronisation nécessaire, avant le début de la transmission, avec l'INP situé à l'autre extrémité de la ligne.

3.5 Résolution du Modèle

Pour évaluer les performances du réseau de files d'attente décrit au paragraphe 3.3, nous avons écrit un programme QNAP ("Queueing Network Analysis Package" [Véran84]), nous permettant à la fois de décrire et de résoudre notre modèle.

Les stations de service définies à l'aide du programme sont exactement celles décrites lors de la modélisation, pour le modèle global comme pour le modèle des agrégats.

Afin d'expliquer plus en détail les classes de clients et les clients que nous avons considérés, les structures de données QNAP correspondantes sont présentées à la figure 3.6.

Deux catégories de classes de clients sont définies, **INT** pour les sessions interactives et **TRANS** pour les transferts de fichiers.

Les caractéristiques communes à tous les clients d'une même classe sont attachées à la classe de clients par un mécanisme d'attributs et de sur-typage propre à QNAP.

Nous définissons ainsi le type-objet **classe** et ses attributs:

- **type** représente la catégorie à laquelle appartient la classe,
- **classe_int** contient les attributs spécifiques de la classe si celle-ci appartient à la catégorie **INT**,
- et **classe_fich** contient les attributs spécifiques de la classe si celle-ci appartient à la catégorie **TRANS**.

Les caractéristiques de chaque client (qui sont dynamiques alors que celles des classes sont statiques) sont également attachées par un mécanisme d'attributs et de sur-typage.

Ainsi le type-objet **client** est défini avec les attributs suivants:

- **cclasse** représente la **classe** à laquelle appartient le client,

```
OBJECT   typ_travail;                          & type objet des modes de travail
         END;
         typ_travail sexec_local,              & mode local
                     sexec_distance;           & mode remote

OBJECT   typ_inp;                              & type objet des destinations a la sortie des INP
         END;
         typ_inp entrant,                      & requete entrante (a destination du systeme)
                 sortant;                      & requete sortante (a destination de la ligne)

OBJECT   typ_parcours;                         & mode de parcours dans le reseau
         END;
         typ_parcours aller,                   & mode aller
                      retour;                  & mode retour

OBJECT   typ_classe;                           & type objet des deux groupes de classes
                  END;
         typ_classe INT,                       & groupe de classes correspondant aux travaux interactifs
                    TRANS;                     & groupe de classes correspondant aux transferts de fichiers

OBJECT   typ_interactif;                       & attributs specifiques d'une classe "travail interactif"
         INTEGER nutilisateur,                 & nombre de clients de cette classe
                 long_chemin,                  & nombre de sites visites par les clients de cette classe
                                               & en mode aller
                 chemin(nsite),                & cheminement dans le reseau des clients de cette classe
                                               & en mode aller
                 application;                  & numero de l'application utilisee par la classe de clients
         REF     typ_travail mode_travail;     & mode de travail de la classe de clients
         END;

OBJECT   typ_trans_fich;                       & attributs specifiques d'une classe "transfert de fichier"
         INTEGER site_orig,                    & site origine du transfert
                 site_desti,                   & site destination du transfert
                 fichier;                      & numero du fichier concerne par le transfert
         REAL    temps;                        & duree de la simulation pour cette classe de clients
         END;

CLASS    OBJECT classe;                        & "sur-type" du type objet predefini CLASS
         REF     typ_classe type;              & groupe d'appartenance de la classe
         REF     typ_interactif classe_int;    & liste des attributs specifiques au groupe
                                               & de classes "travail interactif"
         REF     typ_trans_fich classe_fich;   & liste des attributs specifiques au groupe
                                               & de classes "transfert de fichier"
         END;

CUSTOMER OBJECT client;                        & "sur-type" du type objet predefini CUSTOMER
         REF     classe cclasse;               & classe a laquelle appartient le client
         INTEGER etape_courante,               & numero du site courant du client
                                               & dans le cas d'un travail interactif
                 nb_transfert;                 & nombre de blocs restant a transferer
                                               & dans le cas d'un transfert de fichier
         REF     typ_inp mode_inp;             & destination courante du client a la sortie de l'INP
         REF     typ_parcours mode_parcours;   & sens de cheminement du client dans le reseau
         END;
```

Figure 3.6: Structures de Données QNAP Utilisées

- **étape_courante** représente le numéro du site où le client se trouve (dans le cas d'une session interactive uniquement),
- **nb_transfert** représente le nombre de blocs restant à transférer (dans le cas d'un transfert de fichier uniquement),
- **mode_inp** précise la destination courante du client à la sortie de l'INP (émission ou réception de message),
- et **mode_parcours** indique si le client est en mode aller (du site de connexion vers le site de destination pour une session interactive, ou du site source vers le site destination pour un transfert de fichier) ou en mode retour.

Afin de limiter au maximum l'espace mémoire occupé par l'environnement QNAP lors de l'exécution de notre programme, les classes de clients et les clients eux-mêmes sont créés dynamiquement à l'aide de la fonction QNAP **NEW**.

A l'issue de la définition de notre modèle se pose le problème de la méthode à employer pour sa résolution. L'outil QNAP propose à l'utilisateur un large éventail de méthodes de résolution analytiques, exactes [Baske75,Reise80] ou approchées [Gelen77,Marie79], une méthode markovienne ainsi que la simulation à événements discrets.

L'avantage indéniable des méthodes analytiques sur la simulation est le temps de résolution du modèle beaucoup plus court.

Cependant, leur emploi est conditionné par des hypothèses rigoureuses que doit satisfaire le modèle, au niveau de sa topologie, des lois de service des stations, des disciplines de file d'attente des stations, du type même des stations.

Dans le cas de notre modèle, les demandes de service aux stations sont complexes et leur expression nécessite l'utilisation du langage algorithmique de QNAP, interdisant le recours à une méthode analytique. La simulation s'impose donc comme la seule alternative de résolution du modèle.

Dans un premier temps, le modèle de chaque agrégat est résolu par simulation afin d'obtenir le temps moyen de réponse pour chaque classe de clients du système HP3000 correspondant à l'agrégat. Le temps de réponse moyen pour chaque classe de clients est alors, conformément à la méthode d'agrégation décrite précédemment, considéré, dans le modèle global, comme le temps de service moyen, pour la même classe de clients, du délai exponentiel **Site(i)** représentant l'agrégat et équivalent sur le plan des performances au sous-modèle de l'agrégat.

Une nouvelle simulation résout le modèle global et permet d'obtenir les critères de performances du réseau point-à-point étudié suivants:

- pour chaque classe interactive, le temps de réponse global du réseau, le débit global du réseau exprimé en nombre de transactions par unité de temps, et le taux d'occupation des différentes stations du modèle par les clients représentant la classe interactive,

- pour chaque transfert de fichier, le temps total du transfert, le débit global du réseau exprimé en nombre moyen de blocs transférés par unité de temps et le taux d'occupation des différentes stations du modèle par le seul client représentant le transfert.

3.6 Validation

Afin d'étudier la validité du modèle et, ce faisant, celle de la méthode d'agrégation employée, une campagne de mesures a été effectuée sur le réseau de test représenté par la figure 3.7.

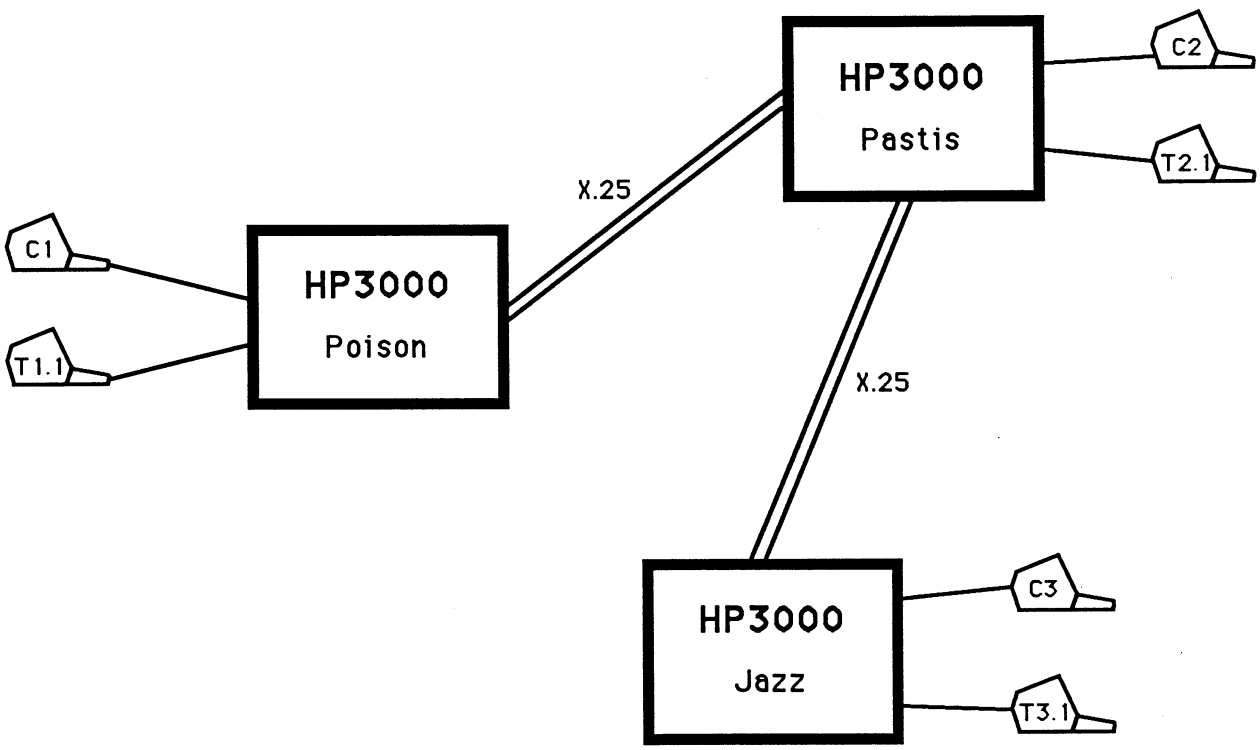

Figure 3.7: Réseau de Test

Cette campagne de mesures a permis, à partir d'un scénario très précis établi à l'avance, de mesurer, pour les classes de clients considérées, les critères de performances énumérés à la fin du paragraphe précédent. En comparant les valeurs mesurées de ces critères avec les résultats recueillis à l'issue de la simulation du modèle global représentant le réseau de test, nous avons pu apprécier la validité de la modélisation.

Ce paragraphe présente tout d'abord le scénario retenu pour les mesures et les classes de clients que nous avons considérées. Ensuite, la comparaison entre critères de performances mesurés

et critères de performances obtenus par simulation du modèle est détaillée et commentée. Enfin, des éléments relatifs à la validité du modèle en sont déduits.

3.6.1 Scénario et Classes de Clients

Cinq sessions interactives sont considérées:

1. l'utilisateur de la console C1 et celui du terminal T1.1 listent le catalogue des fichiers du répertoire HPSPEP sur le système Jazz (application LISTF),
2. l'utilisateur du terminal T2.1 liste l'espace disque libre sur Poison (application FREE5),
3. l'utilisateur de la console C3 liste le catalogue des fichiers du répertoire HPSPEP sur Poison (application LISTF),
4. et l'utilisateur du terminal T3.1 exécute sur Pastis l'application IFS, afin de visualiser sur son écran l'environnement de l'imprimante laser connectée à ce système.

Ces cinq sessions interactives ne forment en fait que quatre classes de la catégorie interactive **INT**, car les deux utilisateurs sur Jazz exécutent depuis le même site de connexion la même application sur le même site de destination par l'intermédiaire du même site de transit; ils appartiennent donc à la même classe de clients.

Au début de la campagne de mesures, l'outil SPEP est lancé sur chacun des trois systèmes présents sur le réseau et l'étape de collection de données est, comme nous l'avons dit précédemment, effective pendant 20 minutes. Pendant toute la durée de cette étape, chaque utilisateur envoie les mêmes requêtes afin d'obtenir un temps de réponse relativement homogène du réseau pour chaque transaction.

Parallèlement aux sessions interactives, deux transferts de fichiers (application DSCOPY) sont lancés dès le début de la phase de collection de données. Un fichier de 170632 octets est transféré de Poison vers Pastis et le même fichier est copié de Jazz vers Pastis. La longueur du fichier est suffisante pour garantir une durée raisonnable des transferts comparée à celle de la collection de données. Ces deux transferts de fichiers correspondent à deux classes de clients de la catégorie **TRANS**.

Le lecteur pourra être étonné de la simplicité du scénario réalisé pour la validation du modèle. Il faut cependant préciser que deux personnes physiques seulement contribuèrent à générer la charge correspondant aux 5 utilisateurs du scénario, de par l'impossibilité d'utiliser au moment des mesures des simulateurs de terminaux tels que AUTOMAN [Autom] ou TEPE/3000 [Tepe], conçus pour les systèmes HP3000.

3.6.2 Comparaison entre Critères Mesurés et Critères Simulés

La table suivante présente, pour chaque classe de clients de la catégorie interactive, les résultats obtenus à l'issue de la simulation du modèle, et les critères de performances mesurés par l'outil SPEP et obtenus à partir d'une synthèse des rapports de réduction des différents systèmes (cf [Prost86, paragraphe 6.2.5]).

L'erreur relative est calculée ainsi qu'un intervalle de confiance à 95% pour le temps de réponse. Cet intervalle de confiance permet d'appréhender l'influence induite par le simulateur lui-même sur l'erreur globale.

Deux critères de performances ont pu être comparés, le temps de réponse moyen du réseau et le débit moyen du réseau exprimés respectivement en secondes et en nombre de transactions par seconde.

	Temps de Réponse Moyen				Débit Moyen		
	R_m	R_s	+/−	$\frac{\lvert R_m - R_s \rvert}{R_m}$	T_m	T_s	$\frac{\lvert T_m - T_s \rvert}{T_m}$
LISTF (Poison → Jazz)	26.20	22.25	1.697	15%	0.0735	0.0813	11%
FREE5 (Pastis → Poison)	60.50	58.87	0.882	3%	0.0176	0.0164	7%
LISTF (Jazz → Poison)	20.40	22.18	1.427	9%	0.0434	0.0402	8%
IFS (Jazz → Pastis)	5.40	5.27	0.595	3%	0.1460	0.1481	2%

L'erreur est d'environ 9% pour l'ensemble des sessions interactives, et au vu de l'intervalle de confiance à 95%, la moitié de cette erreur semble être due au simulateur lui-même. De plus, les résultats sont les meilleurs pour la session interactive générant le plus d'activité sur le réseau (IFS). Les résultats sont donc très encourageants pour les sessions interactives, compte tenu du modèle très grossier utilisé pour modéliser chaque système HP3000. On peut en effet espérer de bien meilleurs résultats avec l'utilisation de Kasandra. De plus, une erreur de 9% en évaluation de performances est tout à fait acceptable.

Pour les transferts de fichiers, la table suivante présente la durée globale du transfert mesurée (en secondes) et celle simulée à l'aide du modèle. Seule cette durée globale a pu être obtenue à partir des rapports de réduction produits par SPEP.

	Durée Globale du Transfert		
	TG_m	TG_s	$\frac{\lvert TG_m - TG_s \rvert}{TG_m}$
DSCOPY (Poison → Pastis)	216	312.4	45%
DSCOPY (Jazz → Pastis)	371	291.9	22%

Les résultats sont ici beaucoup plus médiocres avec une erreur relative de l'ordre de 30%. Cette erreur peut en partie être expliquée par le fait que, dans la modélisation de chaque agrégat, nous avons dû estimer approximativement le délai séparant deux visites consécutives du même client de la catégorie transfert de fichier à cet agrégat. Il est clair que l'utilisation de CAPLAN aurait levé cette approximation. Cependant, il est difficile de dire actuellement dans quelle proportion l'approximation réalisée influe sur la durée globale du transfert.

Si l'on s'interroge sur la validité de la modélisation, il serait délicat d'être tout à fait affirmatif ou négatif, compte tenu du faible nombre de mesures réalisées et de la détermination de certains paramètres d'entrée et critères de performances rendue très laborieuse par l'impossibilité de la mise en oeuvre de l'outil CAPLAN, et par voie de conséquence de Kasandra. De plus, le nombre de critères de performances que nous avons pu comparer reste très limité [L].

Cependant, les résultats déjà obtenus semblent très encourageants pour les sessions interactives et n'invalident en rien la méthode d'agrégation tout à fait générale utilisée, consistant à remplacer chaque site par un délai équivalent, dont le temps de service moyen est obtenu par la résolution du sous-modèle représentant ce site. L'intérêt de cette campagne de mesures aura été sans nul doute de montrer les faiblesses du modèle des agrégats et la nécessité de l'emploi des outils CAPLAN et Kasandra pour une utilisation efficace du modèle global, tant du point de vue de la facilité de détermination de ses paramètres d'entrée que de la précision des résultats de simulation. En particulier, l'utilisation de Kasandra permettrait la détermination du deuxième moment du temps de service du délai représentant chaque agrégat, qui pourrait fort ne plus satisfaire à une loi exponentielle.

CHAPITRE 4

Mesures de Performances de Différentes Stratégies d'Implémentation de Programmes Parallèles sur un Réseau Local d'IBM PC

Afin d'implémenter des algorithmes parallèles sur un réseau local d'IBM PC, ce chapitre présente trois stratégies, différentes de par la politique d'allocation des tâches de calcul parallèles qu'elles appliquent.

Deux exemples d'algorithmes parallèles sont étudiés, le produit de matrices et la résolution de systèmes d'équations linéaires par la méthode itérative de relaxation.

Pour assurer un bon équilibrage de charge entre processeurs physiques **hétérogènes**, la notion de **processeur virtuel** est introduite [**F**].

Des mesures de performances ont été réalisées sur un réseau local, formé d'un IBM PC AT et de 2 IBM PC XT, pour comparer les efficacités respectives des trois stratégies d'implémentation, en termes de temps de réponse global du réseau pour exécuter chaque programme parallèle et d'accélération. Le calcul de l'accélération prend en compte la notion de **processeur virtuel** [**U**].

L'interprétation des résultats des mesures permet d'appréhender le champ d'application de chaque stratégie et de caractériser les raisons qui feront préférer une stratégie d'implémentation à une autre.

Ce chapitre est divisé en six paragraphes. Le paragraphe 4.1 présente le réseau local d'IBM PC. Le paragraphe 4.2 décrit les caractéristiques communes aux trois stratégies d'implémentation ainsi que les particularités de chacune. Les paragraphes 4.3 et 4.4 illustrent ces stratégies en présentant l'implémentation de deux algorithmes parallèles, le produit de matrices et la méthode de relaxation pour la résolution de systèmes d'équations linéaires. Le paragraphe 4.5 s'intéresse aux mesures des performances respectives des trois stratégies d'implémentation, réalisées sur un réseau local formé de trois IBM PC, et implémentant les deux algorithmes parallèles mentionnés précédemment. L'interprétation des résultats de mesures permet d'exhiber des règles générales pour l'utilisation des différentes stratégies, résumées au paragraphe 4.6, qui conclut ce chapitre.

4.1 Présentation du Réseau Local d'IBM PC

Ce paragraphe s'intéresse successivement aux aspects matériels et aux aspects logiciels du réseau local d'IBM PC.

4.1.1 Description Matérielle

Le réseau local d'IBM PC [IBMPC1] est un réseau local à large bande permettant à plusieurs IBM PC (tels que IBM PC, IBM PC XT, IBM PC AT) de communiquer (cf figure 4.1).

Les ordinateurs sont reliés entre eux par des câbles coaxiaux qui leur apportent la possibilité de communications bi-directionnelles à une vitesse de transmission de 2 méga-bits par seconde.

Chaque ordinateur doit posséder une carte d'adaptation pour être raccordé au réseau. Cet adaptateur est composé d'un modem, d'un interface d'entrée-sortie appelé BIOS ("Basic Input Output System"), et d'un processeur de communication propre qui s'occupe des communications avec les autres ordinateurs du réseau. Ce processeur est programmé pour réaliser les niveaux 1 (physique) à 5 (session) du protocole de communication.

Chaque adaptateur est connecté à une unité appelée "translateur de fréquences", qui effectue une modulation de fréquences, propre à soutenir des voies de communication simultanées entre tous les ordinateurs présents sur le réseau.

Sans avoir recours à des options d'extension du réseau, on peut relier jusqu'à 8 ordinateurs au même translateur de fréquences, la distance de chaque ordinateur au translateur étant limitée à 67 mètres.

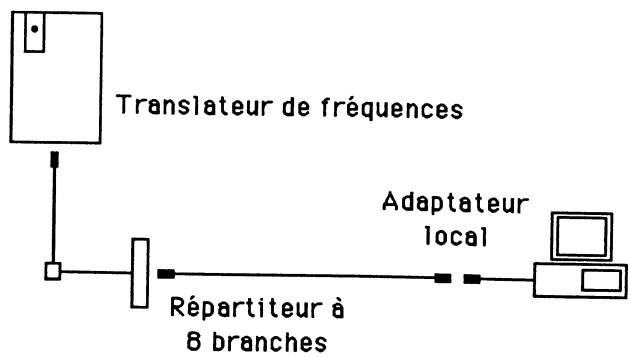

Figure 4.1: Configuration de Base d'un Réseau Local d'IBM PC

Afin de réguler les transmissions, le protocole d'arbitration pour l'accès au canal de communication est CSMA/CD ("Carrier Sense Multiple Access with Collision Detection"), qui résout, de manière équitable, les contentions dues au partage du canal de communication.

Le réseau que nous avons utilisé lors de la campagne de mesures (cf paragraphe 4.5.1) était en fait composé de deux IBM PC XT et d'un IBM PC AT, tous trois reliés au translateur de fréquences par un répartiteur à huit branches.

4.1.2 Programme d'Utilisation du Réseau Développé par IBM

IBM a développé un programme facilitant l'utilisation du réseau local d'IBM PC [IBMPC2]. Ce programme propose à l'utilisateur, par l'intermédiaire de menus interactifs ou de commandes DOS, les fonctionnalités d'une messagerie électronique et la possibilité de partage et d'utilisation à distance de périphériques (disques durs, répertoires de disques durs, imprimantes, traceurs de courbes).

Chaque ordinateur présent sur le réseau possède un nom propre et, en fonction de ses caractéristiques, il a accès à un ensemble de fonctionnalités, qui lui confère l'un des quatre types suivants:

1. le *redirecteur* doit posséder au moins 128 kilo-octets de mémoire vive; il peut utiliser à distance les périphériques partagés par les *serveurs* et envoyer des messages aux autres ordinateurs,

2. le *récepteur* doit avoir au moins 192 kilo-octets de mémoire vive; en plus des fonctionnalités offertes au *redirecteur*, il peut recevoir des messages en provenance des autres ordinateurs et les stocker dans un fichier lors de leur réception,

3. le *messager* doit posséder au moins 256 kilo-octets de mémoire vive; outre les fonctionnalités offertes au *récepteur*, il peut recevoir et rediriger les messages destinés à d'autres ordinateurs, et exécuter en tâche de fond le programme d'utilisation du réseau et y accéder à tout moment depuis n'importe quelle application,

4. le *serveur* doit avoir au moins 320 kilo-octets de mémoire vive; il a accès à toutes les fonctionnalités du *messager* et peut en outre partager son disque dur, ou simplement des répertoires particuliers de son disque dur, ainsi que l'ensemble des périphériques de sortie qui lui sont connectés (imprimantes, traceurs de courbes), avec les autres ordinateurs du réseau.

Cependant, les modes d'utilisation possibles pour ce programme (mode interactif ou commandes DOS) ne nous ont pas permis de l'utiliser pour implémenter les algorithmes parallèles. Nous avons donc eu recours directement au BIOS (cf paragraphe 4.4).

4.2 Description des Stratégies d'Implémentation

Les trois stratégies d'implémentation que nous avons développées sont basées sur une décomposition de domaine, appelée également **partitionnement de données**, dont le principe peut s'énoncer ainsi: à partir d'une tâche de calcul T opérant sur un domaine de taille N, on crée m sous-tâches t_i, $1 \leq i \leq m$, aussi identiques que possible les unes des autres, opérant sur une partition du domaine initial en m parties disjointes; la taille de chaque partie (appelée encore sous-domaine) sera aussi proche que possible de N/m; les m sous-tâches s'exécuteront en parallèle sur un nombre NC de processeurs, NC pouvant très bien être inférieur à m. L'objectif de la parallélisation est de réduire la durée globale de l'exécution de la tâche T initiale.

Le point principal de ce partitionnement est la définition des sous-tâches t_i, $1 \leq i \leq m$, de manière à garantir une indépendance maximale entre elles. En effet, plus ces sous-tâches seront indépendantes, moins nombreuses seront les communications et synchronisations requises à leur exécution en parallèle et par conséquent moins grand sera le surcoût ajouté au coût du calcul séquentiel de la tâche T. De plus, si l'on peut assurer une bonne homogénéité des sous-domaines, meilleur sera l'équilibrage de charge entre les processeurs exécutant en parallèle les sous-tâches.

Les trois stratégies d'implémentation d'algorithmes parallèles sur réseau local d'IBM PC que nous proposons diffèrent par la politique d'allocation des sous-tâches qu'elles appliquent.

Cependant, toutes trois reposent sur une hiérarchie de type **maître-esclaves** entre les ordinateurs du réseau. L'ordinateur maître, appelé dorénavant maître, se charge de la gestion des sous-tâches parallèles t_i, $1 \leq i \leq m$, en les allouant aux ordinateurs esclaves, appelés dorénavant esclaves, voire en s'allouant personnellement certaines d'entre elles.

4.2.1 Stratégie 1: Allocation Statique des Tâches Parallèles

Cette stratégie est basée sur un partitionnement du domaine en *NC* sous-domaines, *NC* représentant toujours le nombre d'ordinateurs présents sur le réseau.

Au début de l'exécution, le maître lance *NC-1* sous-tâches sur les *NC-1* esclaves et prend à sa charge la dernière sous-tâche, qu'il exécute. Lorsqu'il a terminé, il attend qu'il en soit de même pour tous les esclaves, en se mettant en attente de la réception de leurs résultats de calcul respectifs. Enfin, il combine les résultats partiels des différents esclaves avec les siens pour obtenir les résultats finals.

L'allocation des sous-tâches est donc purement **statique**, et effectué une seule fois au début de l'exécution du programme.

4.2.2 Stratégie 2: Allocation Dynamique avec Gestionnaire de Tâches Dédié

Pour cette stratégie, comme pour la suivante, le nombre m de sous-tâches n'est pas directement lié au nombre *NC* d'ordinateurs présents sur le réseau. m dépend surtout du problème à résoudre et sera choisi afin d'optimiser la granularité des sous-tâches parallèles. La plupart du temps, m sera choisi grand devant *NC*, pour éviter la famine d'un ordinateur, et si possible, m sera un multiple de *NC* de manière à réaliser un bon équilibrage de charge entre les ordinateurs.

Soulignons dès maintenant le fait que les stratégies 2 et 3 sont applicables même si l'on ne connaît pas parfaitement la taille N du domaine d'étude au début de l'exécution du programme.

Pour la stratégie 2, le maître est dédié à la gestion des sous-tâches et son seul rôle est de les allouer aux esclaves au fur et à mesure qu'ils en font la demande. Le maître ne calcule donc aucune sous-tâche.

Etant donné qu'il n'y a que *NC-1* esclaves, m sera dans ce cas un multiple de *NC-1*.

L'allocation des sous-tâches est cette fois-ci **dynamique**, et un esclave, dont la vitesse de calcul serait supérieure à celle des autres esclaves, ou dont les sous-tâches qu'il reçoit nécessiteraient moins de calcul que celles allouées aux autres esclaves, peut très bien être amené à calculer plus de sous-tâches que ses partenaires.

4.2.3 Stratégie 3: Allocation Dynamique avec Gestionnaire Interruptible

Cette troisième stratégie est analogue à la précédente. Cependant, le maître n'est plus dédié à la seule gestion des sous-tâches et entre en compétition avec les esclaves pour leur attribution.

Chaque fois qu'un esclave termine l'exécution d'une sous-tâche, il interrompt le calcul du maître en cours, afin que celui-ci lui alloue la prochaine sous-tâche à effectuer, s'il en reste une. Après l'attribution de la nouvelle sous-tâche, le maître reprend son calcul à l'endroit où il l'avait suspendu. Le nombre m de sous-tâches pourra être dans ce cas un multiple de NC.

On peut noter que cette stratégie réalise un degré de parallélisme supérieur à la stratégie précédente, par l'emploi de tous les ordinateurs pour le calcul des sous-tâches. Ce gain en parallélisme sera surtout sensible si le nombre d'ordinateurs présents sur le réseau est petit. Cependant, la mise en oeuvre de cette stratégie s'est avérée plus laborieuse à cause de la gestion des interruptions sur le maître.

4.3 Présentation des Algorithmes Parallèles

Illustrons brièvement les trois stratégies d'implémentation présentées au paragraphe précédent au travers des deux algorithmes parallèles que nous avons utilisés pour les mesures de performances (cf paragraphe 4.5).

Pour chaque algorithme, nous précisons les différentes stratégies que nous avons employées pour les implémenter sur le réseau local dont nous disposions, et associées à ces stratégies, les sous-tâches respectives. Pour la méthode de relaxation, nous rappelons en outre très brièvement son algorithme.

4.3.1 Produit de Matrices

Le problème consiste à calculer la matrice C à r lignes et s colonnes, produit des deux matrices A et B, respectivement à r lignes, q colonnes, et à q lignes, s colonnes.

Les trois stratégies ont été implémentées. Pour chaque stratégie, une sous-tâche consiste à calculer un nombre fixé l de lignes de la matrice C, soit à multiplier l lignes de la matrice A par la matrice B.

Pour la stratégie 1, le nombre l de lignes est directement fonction de r et de NC.

Pour les stratégies 2 et 3, l pourrait être égal à 1, mais nous avons préféré le considérer comme un paramètre d'entrée du programme, afin de pouvoir ajuster la granularité des sous-tâches.

Une fois lancées, les sous-tâches s'exécutent en parallèle, tout à fait indépendamment les unes des autres. Cette méthode de parallélisation est souvent définie sous le terme de décomposition géométrique.

4.3.2 Méthode de Relaxation

Soit à résoudre le système d'équations linéaires $Ax = b$, où A est une matrice carrée non singulière de dimension d, x et b des vecteurs à d composantes, et où x est la solution du système.

La méthode de relaxation [Stoer83] consiste à résoudre ce système par un processus itératif donné par l'expression suivante:

$$a_{jj}\, x_j^{(i+1)} = a_{jj}\, x_j^{(i)} + w \left(- \sum_{k=1}^{j-1} a_{jk}\, x_k^{(i+1)} - a_{jj}\, x_j^{(i)} - \sum_{k=j+1}^{d} a_{jk}\, x_k^{(i)} + b_j \right), \qquad 1 \leq j \leq d, \;\; i \geq 0$$

où $x_j^{(i)}$ représente la $j$ième composante du $i$ième itéré du vecteur x et w le coefficient de relaxation.

Dans le cas de notre implémentation parallèle, une sous-tâche consiste à calculer un nombre fixé de composantes contiguës du vecteur solution x, notées x_{i_1} à x_{i_2}.

A chaque itération i, l'ordinateur exécutant cette sous-tâche a besoin des composantes $x_1^{(i)}$ à $x_{i_1-1}^{(i)}$. En attendant de les recevoir des ordinateurs exécutant les sous-tâches associées, il calcule les termes de l'expression ne dépendant que des composantes du précédent itéré du vecteur x, $x^{(i-1)}$. Lorsqu'il a reçu toutes les composantes dont il a besoin, il calcule alors l'expression complète des composantes x_{i_1} à x_{i_2} et envoie leurs valeurs à tous les ordinateurs qui en ont besoin, c'est-à-dire les ordinateurs qui calculent les composantes de $x^{(i)}$ d'indices supérieurs à i_2.

A partir de la description de l'algorithme qui précède, le lecteur peut subodorer les dépendances très strictes qui existent entre les sous-tâches et imaginer la complexité qu'une allocation dynamique des sous-tâches pourrait entraîner. C'est pourquoi, pour cet algorithme, nous nous sommes limités à mettre en oeuvre la stratégie 1, qui présente l'avantage de garantir que chaque ordinateur calculera les mêmes composantes des itérés successifs du vecteur x.

4.4 Implémentation des Algorithmes Parallèles

Ce paragraphe présente la méthode que nous avons employée pour implémenter, suivant les différentes stratégies, les deux algorithmes parallèles décrits précédemment.

Comme nous l'avons expliqué au paragraphe 4.1.2, nous ne pouvions pas avoir recours, pour ce faire, au programme d'utilisation du réseau local d'IBM PC développé par IBM. La seule alternative était donc d'utiliser directement le BIOS ("Basic Input Output System").

Un programme d'application peut communiquer avec le BIOS par l'intermédiaire de commandes spéciales, comportant de nombreux champs, appelées blocs de contrôle du réseau ou *NCB* ("Network Control Blocks"). Ces commandes permettent de réaliser toutes les fonctionnalités offertes par le programme développé par IBM et bien d'autres encore. Leurs adresses sont transmises au BIOS par un mécanisme d'interruption à l'aide d'une paire bien spécifique de registres du processeur de l'ordinateur [IBMPC1].

Pour implémenter les algorithmes parallèles sur le réseau local d'IBM PC, nous avons tout d'abord créé une librairie regroupant toutes les procédures de transmission au BIOS des NCB nécessaires à nos besoins. Ensuite, à l'aide de cette librairie, nous avons mis en oeuvre les différentes stratégies d'implémentation décrites au paragraphe 4.2.

4.4.1 Librairie des Blocs de Contrôle du Réseau

Afin de présenter brièvement la librairie introduite précédemment, nous décrivons rapidement le format général d'un bloc de contrôle du réseau, et donnons la liste des fonctionnalités que les procédures de transmission au BIOS des NCB, composant la librairie, permettent de réaliser.

Format des Blocs de Contrôle du Réseau

Chaque PC présent sur le réseau peut recevoir jusqu'à 16 noms locaux, dont un au moins est permanent et représente son nom de réseau. Entre deux noms attribués à deux PC distincts, une voie logique de communication bi-directionnelle, appelée **session**, peut être établie et supprimée ensuite. Une communication entre deux PC ne peut avoir lieu qu'au travers d'une session préalablement établie. Chaque message envoyé vers un nom ne peut être reçu que par ce nom, lequel envoie un message d'acquittement au nom expéditeur lors de la réception du message sans erreur.

Lorsque le programme d'application s'exécutant sur un PC envoie au BIOS un bloc de contrôle du réseau, ce dernier place, lorsqu'il a fini d'exécuter la commande décrite par le bloc de contrôle, un code d'erreur dans un registre spécifique du PC. Ce code d'erreur est nul si la commande s'est déroulée sans erreur.

D'autre part, chaque fois que le programme d'application envoie un bloc de contrôle au BIOS, il peut décider soit d'attendre que la commande correspondante soit achevée avant de poursuivre son exécution (mode "wait"), soit de ne pas attendre et poursuivre aussitôt son exécution (mode "no wait").

En mode "no wait", deux alternatives sont cependant possibles. Dans le premier cas, lorsque la commande est achevée, le programme d'application l'ayant lancée est interrompu et dérouté sur une routine d'interruption (mode "no wait" interruptible), avant de revenir au point de l'exécution avant déroutement. Dans le deuxième cas, lorsque la commande est terminée, le PC reçoit un code de terminaison de commande dans un registre spécifique, que le programme d'application a la charge de tester si nécessaire (mode "no wait" simple).

Parmi les nombreux champs d'un bloc de contrôle du réseau, on retrouve le code de la commande, qui comporte deux valeurs possibles selon l'alternative retenue pour le mode "no wait" (interruptible ou simple), le nom local du PC adressant la commande au BIOS, le nom de la session concernée, l'adresse et la longueur d'un tampon de données utile pour les transferts de messages, la longueur du dernier message reçu, l'adresse de la routine d'interruption si le mode "no wait" interruptible est sélectionné, le code de terminaison de la commande ainsi que son code d'erreur.

Procédures Définies

Exposons tout d'abord les quelques raisons qui nous ont conduits à utiliser le langage de programmation Turbo Pascal pour implémenter les deux algorithmes parallèles.

A la lecture du paragraphe précédent, il est clair que le langage de programmation retenu doit pouvoir interfacer le langage assembleur du PC, notamment pour charger et lire des registres, activer et masquer les interruptions. D'autre part, il est souhaitable que le langage de programmation soit suffisamment évolué pour permettre une écriture aisée des algorithmes parallèles.

Turbo Pascal présente ces deux qualités. C'est un langage de haut niveau qui interface très aisément l'assembleur grâce à l'appel de procédures prédéfinies ou à l'insertion directe d'instructions assembleur dans le code source. De plus, son environnement intégré accélère considérablement le temps nécessaire à la mise au point des programmes.

Voici maintenant la liste des fonctionnalités que les procédures définies dans la librairie des blocs de contrôle du réseau permettent de réaliser:

- des manipulations de noms de réseau, telles que ajoût d'un nouveau nom, destruction d'un nom, interrogation sur l'existence d'un nom,
- la gestion des sessions (ouverture, fermeture, identification),
- et des communications entre PC (envoi de message, réception de message).

4.4.2 Mise en Oeuvre des Trois Stratégies

Les trois stratégies d'implémentation d'algorithmes parallèles, présentées au paragraphe 4.2, ont été mises en oeuvre de manière très modulaire, afin de simplifier le codage d'une nouvelle stratégie par la réutilisation possible d'une partie du code développé lors de la mise en oeuvre de la stratégie précédente.

Ainsi, chaque stratégie est composée de trois modules, ayant tous accès à la librairie des blocs de contrôle du réseau.

Le premier module est le module du maître. Bien qu'il dépende de la stratégie utilisée, sa structure est commune aux trois stratégies, et elle peut être énoncée comme suit:

1. lancer les esclaves,
2. mettre en oeuvre la stratégie considérée,
3. arrêter les esclaves.

La première partie et la troisième partie de ce module ne dépendent ni de la stratégie mise en oeuvre, ni de l'algorithme parallèle implémenté. Elles ont donc été regroupées dans un sous-module, appelé module de gestion des esclaves, accessible par le module du maître.

Le second module est le module des esclaves. Il ne dépend ni de la stratégie d'implémentation mise en oeuvre, ni de l'algorithme parallèle implémenté.

Le troisième module contient la sous-tâche, que chaque esclave devra exécuter lorsqu'il en fera la demande (pour les stratégies 2 et 3 à allocation dynamique de tâches), ou que le maître assignera statiquement à chaque esclave (stratégie à allocation statique des tâches). Ce module dépend à la fois de l'algorithme parallèle implémenté et de la stratégie d'implémentation utilisée. Cependant, le même module a pu être défini pour les stratégies 2 et 3 dans le cas du produit de matrices.

Nous détaillons maintenant le module de gestion des esclaves, commun à toute stratégie et tout algorithme parallèle, le module du maître pour chaque stratégie, le module des esclaves, commun également à toute stratégie et tout algorithme parallèle, et le module de la tâche esclave pour chaque stratégie.

Module de Gestion des Esclaves

Lancement des Esclaves

Pour lancer les esclaves, le maître essaie dans un premier temps d'établir une session avec chaque PC présent sur le réseau. A la fin de cette étape, il connaît le nombre et l'identification des esclaves.

Dans un deuxième temps, il envoie à chaque paire d'esclaves les commandes appropriées pour que ces deux esclaves établissent une session entre eux.

Précisons cependant que la seconde étape n'est pas nécessaire dans le cas du produit de matrices.

Arrêt des Esclaves

Pour arrêter les esclaves, le maître envoie à chacun d'entre eux l'ordre de fermer chaque session qu'il a établie au début de l'exécution du programme. Ensuite, le maître ferme à son tour les sessions qu'il avait établies lors de la première étape du lancement des esclaves.

Module du Maître

Stratégie 1: Allocation Statique

Après avoir lancé les esclaves, le maître détermine le sous-domaine de chaque sous-tâche. Puis, il envoie à chaque esclave sa sous-tâche, composée du nom du module de la tâche esclave à exécuter et des données à traiter.

Le maître demande alors à recevoir les résultats en provenance des esclaves et exécute sa propre sous-tâche. Lorsque cette dernière est terminée, il attend que tous les esclaves lui aient transmis leurs résultats. Enfin, il arrête les esclaves.

Précisons que toutes les commandes d'envoi et de réception de messages réalisées par le maître le sont en mode "no wait" simple afin, d'une part d'accélérer le chargement des sous-tâches sur les esclaves, et d'autre part de permettre au maître de continuer à exécuter sa propre sous-tâche lors de l'arrivée des résultats en provenance d'un esclave (dans le cas bien sûr où un esclave finirait sa sous-tâche avant le maître).

Stratégie 2: Allocation Dynamique avec Gestionnaire Dédié

Après avoir lancé les esclaves, le maître détermine le sous-domaine de chaque sous-tâche, puis envoie à chaque esclave le noyau commun à toutes les sous-tâches, composé du nom du module de la tâche esclave à exécuter et des données communes à toutes les sous-tâches.

Ensuite, de manière itérative pour chaque sous-tâche restant à traiter, le maître se met en attente du premier esclave prêt à travailler, il récupère les résultats de cet esclave (sauf si ce dernier est sur le point d'exécuter sa première sous-tâche), il lui adresse les données spécifiques à la sous-tâche courante, et il lui donne l'ordre de les traiter.

Lorsque toutes les sous-tâches ont été ainsi attribuées, pour chaque esclave venant lui réclamer une nouvelle sous-tâche, le maître récupère ses résultats et il lui donne l'ordre de se mettre en attente de l'ordre de fermeture des sessions. A l'arrivée des résultats de la dernière sous-tâche, le maître arrête les esclaves.

De nouveau, toutes les commandes d'envoi et de réception de messages sont exprimées en mode "no wait" simple afin, d'une part d'accélérer le chargement du noyau des sous-tâches sur les

esclaves, et d'autre part de permettre au maître de transmettre les données d'une nouvelle sous-tâche à un esclave lors de l'arrivée de résultats en provenance d'un autre esclave au même instant.

Pour que le maître puisse identifier le premier esclave prêt à travailler, un bloc de contrôle du réseau spécifique est défini pour chaque esclave et la recherche de l'esclave s'effectue en testant le code de terminaison de la commande associée à chaque bloc.

Stratégie 3: Allocation Dynamique avec Gestionnaire Interruptible

Après avoir lancé les esclaves, le maître procède comme lors de la stratégie précédente, en déterminant le sous-domaine de chaque sous-tâche et en envoyant à chaque esclave le noyau commun à toutes les sous-tâches, composé du nom du module de la tâche esclave à exécuter et des données communes à toutes les sous-tâches.

Ensuite, le maître s'alloue une sous-tâche, active les interruptions et commence à exécuter sa propre sous-tâche. Lorsque cette dernière est terminée, il masque les interruptions, vérifie s'il reste d'autres sous-tâches à traiter, si oui, il s'en alloue une nouvelle, réactive les interruptions et procède ainsi de manière itérative jusqu'à épuisement des sous-tâches.

Lorsque la dernière sous-tâche a été attribuée, il réactive les interruptions et se met en attente de l'arrivée des résultats en provenance des esclaves, puis arrête les esclaves.

Dans cette stratégie, le maître attribue les sous-tâches aux esclaves grâce à un mécanisme d'interruptions.

Contrairement à la stratégie précédente, le maître envoie à chaque esclave le noyau commun à toutes les sous-tâches en mode "no wait" interruptible. Lorsque chaque esclave a reçu ces données, le BIOS interrompt l'exécution du maître, qui est dérouté sur une routine d'interruption.

Cette routine recherche l'esclave qui a provoqué l'interruption du maître, demande à recevoir les prochains résultats en provenance de cet esclave en mode "no wait" interruptible, lui envoie les données propres à la sous-tâche courante et l'ordre de traiter cette sous-tâche en mode "no wait" simple.

Lorsque l'esclave aura achevé cette sous-tâche, il enverra les résultats correspondants au maître. Le BIOS du maître, détectant l'arrivée des résultats, interrompra une nouvelle fois l'exécution du maître, qui sera aussitôt dérouté sur la routine d'interruption.

Ce processus se répète itérativement jusqu'à épuisement des sous-tâches. Lorsque toutes les sous-tâches ont déjà été attribuées, la routine d'interruption envoie simplement à l'esclave, ayant

provoqué le déroutement du maître, l'ordre de se mettre en attente de l'ordre de fermeture des sessions.

Module des Esclaves

Ce module est lancé par l'utilisateur sur chaque esclave avant le lancement du module du maître sur le maître.

Chaque esclave, exécutant ce module, définit son propre nom de réseau et l'ajoute à sa table locale des noms de réseaux. Puis, il ouvre une session avec le maître et commence l'exécution d'une boucle.

Dans cette boucle, de manière itérative, l'esclave demande à recevoir un message en provenance du maître. Ce message contient le code d'un ordre, que l'esclave décode dès la réception du message. En fonction de l'ordre reçu, l'esclave réalise l'un des traitements suivants: ouverture d'une session avec un autre esclave, lancement de l'exécution du module de la tâche esclave, dont le nom est inclus dans le message, fermeture des sessions ouvertes auparavant avec d'autres esclaves et destruction de son propre nom de réseau. Toutes les commandes associées à ces traitements sont effectuées en mode "wait".

L'exécution de la boucle se termine lors de la destruction par l'esclave de son propre nom de réseau.

Module de la Tâche Esclave

Le module de la tâche esclave contient l'ensemble des traitements que chaque esclave doit réaliser lors de la réception en provenance du maître de l'ordre d'exécution de ce module. Ces traitements correspondent au calcul d'une ou de plusieurs sous-tâches t_i, $1 \leq i \leq m$, et à l'ensemble des communications entre le maître et l'esclave nécessaires à la gestion de ces sous-tâches. Quelle que soit la stratégie utilisée, l'esclave retourne, à la fin de l'exécution du module de la tâche esclave, dans la boucle du module des esclaves, où il se met en attente de la réception d'un nouveau message en provenance du maître.

Stratégie 1

Pour cette stratégie, le module de la tâche esclave réalise le traitement nécessaire au calcul d'une seule sous-tâche t_i.

Ce traitement consiste à recevoir le sous-domaine de la sous-tâche t_i (par exemple l lignes de la matrice A), à calculer cette sous-tâche afin d'obtenir les résultats correspondants (par exemple l lignes de la matrice C) et à envoyer ces résultats au maître.

Tous les transferts de messages se font en mode "wait".

Stratégies 2 et 3

Pour ces deux stratégies, le module de la tâche esclave est identique et réalise le traitement nécessaire au calcul de toutes les sous-tâches t_i qui sont attribuées à un même esclave.

Ce traitement consiste à recevoir les données communes à toutes les sous-tâches (par exemple la matrice B) et à attendre un ordre du maître, qui est soit un ordre de calcul, soit un ordre d'arrêt.

Si l'ordre reçu est un ordre de calcul, l'esclave demande à recevoir les données propres à la sous-tâche courante, calcule cette sous-tâche et envoie les résultats correspondants au maître, avant de se mettre en attente d'un nouvel ordre. Tous les transferts de messages s'effectuent en mode "wait".

Si l'ordre reçu est, au contraire, un ordre d'arrêt, l'esclave termine l'exécution du module de la tâche esclave.

4.5 Mesures de Performances

Nous présentons dans ce paragraphe les mesures de performances, que nous avons réalisées, sur un réseau local d'IBM PC, pour estimer les efficacités respectives des trois stratégies d'implémentation d'algorithmes parallèles sur ce type de réseau. Ces mesures sont basées sur les différentes implémentations des deux algorithmes parallèles présentés au paragraphe 4.3, le produit de matrices et la méthode de relaxation.

Après une brève description du réseau local d'IBM PC que nous avons utilisé pour réaliser la campagne de mesures, nous présentons les différentes configurations du réseau retenues pour chaque stratégie et chaque algorithme parallèle.

Puis, nous détaillons l'ensemble des mesures de performances obtenues, exprimées sous la forme des temps de réponse du réseau, pour l'exécution parallèle des différents algorithmes, et des accélérations associées.

Enfin, nous commentons et interprétons ces résultats afin d'exhiber les conditions favorisant l'utilisation d'une stratégie d'implémentation devant une autre.

4.5.1 Campagne de Mesures

Le réseau local que nous avons utilisé pour les mesures (cf figure 4.2) est composé d'un IBM PC AT, appelé SRV1 (serveur 1), d'un IBM PC XT avec disque dur, appelé SRV2 (serveur 2) et d'un IBM PC XT sans disque dur, appelé MSG3 (messager 3).

Les noms logiques des PC sur le réseau sont *Maître*, *Esclave1* et *Esclave2*. Les deux serveurs SRV1 et SRV2 peuvent jouer le rôle de maître. Leur disque dur sera partagé par les esclaves qui pourront y accéder à distance pour charger le module de la tâche esclave résidant sur le maître. Par contre, MSG3 est nécessairement un esclave, car il ne dispose pas de disque dur.

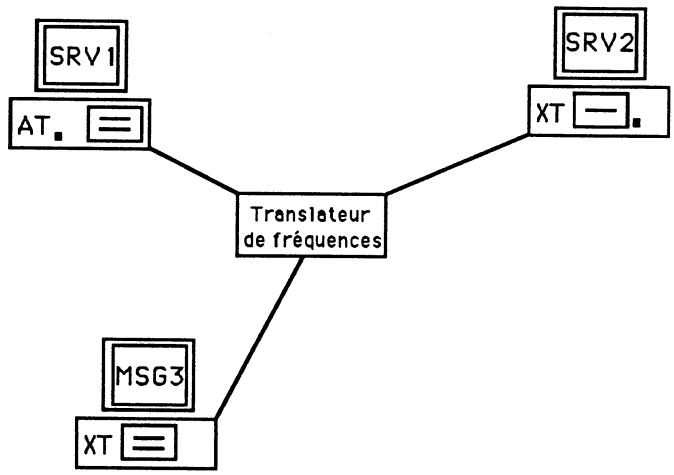

Figure 4.2: Réseau Utilisé lors de la Campagne de Mesures

Les deux algorithmes parallèles, présentés au paragraphe 4.3, ont été implémentés. Nous rappelons que pour le produit de matrices, les trois stratégies d'implémentation, décrites aux paragraphes 4.2 et 4.4, ont été mises en oeuvre, alors que seule la première stratégie a été utilisée pour la méthode de relaxation. Précisons enfin qu'à partir de maintenant, nous appellerons indifféremment chaque PC présent sur le réseau "ordinateur" ou "processeur physique".

Produit de Matrices

Les matrices A, B et C sont des matrices carrées de dimension 50. Cette dimension a la valeur maximale que nous avons pu considérer compte tenu de l'espace en mémoire vive disponible sur chaque PC. En effet, nous n'avons pas voulu avoir recours au disque pour le stockage des matrices, car cela aurait entraîné un déséquilibre de charge important entre les ordinateurs, du fait de performances d'accès de leurs disques respectifs fort différentes.

Du fait de la taille relativement réduite des matrices, nous avons préféré contrôler la granularité des sous-tâches par l'introduction d'un nouveau paramètre *NB_ITER*, plutôt que de modifier le nombre l de lignes de la matrice A composant chaque sous-domaine. *NB_ITER* représente le nombre de fois (entre 1 et 10) que chaque produit partiel des matrices A et B (constituant chaque sous-tâche) est effectivement calculé.

Stratégie 1

La stratégie 1 consiste à partitionner la matrice A en un nombre de blocs disjoints égal au nombre de processeurs physiques présents sur le réseau. Chaque bloc est composé de plusieurs lignes de la matrice A et chaque sous-tâche réalise *NB_ITER* fois le calcul des lignes correspondantes de la matrice C.

Cinq configurations de réseau ont été considérées:

- configuration 1*a*: seul SRV1 est utilisé comme *Maître* (le produit de matrices est calculé séquentiellement sur SRV1),
- configuration 1*b*: seul SRV2 est utilisé comme *Maître* (le produit de matrices est calculé séquentiellement sur SRV2),
- configuration 1*c*: SRV1 est utilisé comme *Maître* et MSG3 comme *Esclave1*,
- configuration 1*d*: SRV2 est utilisé comme *Maître* et MSG3 comme *Esclave1*,
- configuration 1*e*: SRV1 est utilisé comme *Maître*, SRV2 comme *Esclave1* et MSG3 comme *Esclave2*.

Introduisons maintenant la notion de **processeur virtuel**, permettant une comparaison des performances de processeurs physiques hétérogènes pour un programme d'application donné.

Soient $T_1, T_2, \ldots, T_p$ les temps d'exécution séquentielle respectifs du programme d'application considéré sur p processeurs physiques hétérogènes. Si T_k, $1 \leq k \leq p$, est le plus petit de ces temps, le processeur physique k devient le processeur de référence pour ce programme d'application et on le considère équivalent à un processeur virtuel. En formant les rapports $\frac{T_1}{T_k}, \frac{T_2}{T_k}, \ldots, \frac{T_p}{T_k}$, on définit les nombres de processeurs virtuels auxquels les p processeurs physiques sont respectivement équivalents, pour le programme d'application considéré. Il n'y a bien sûr aucune raison a priori pour que ces nombres soient entiers.

Cette notion de processeur virtuel définit ainsi une unité de performance de processeurs physiques hétérogènes pour un programme d'application donné. Il est clair que cette unité dépend du programme d'application considéré.

Ainsi, les configurations 1*a* et 1*b* permettent de déterminer le nombre de processeurs virtuels auxquels l'IBM PC AT est équivalent pour le produit de matrices. L'IBM PC XT, en tant que processeur physique le plus lent, est le processeur de référence et équivaut donc à un processeur virtuel.

Afin d'assurer un bon équilibrage de charge entre les sous-tâches, SRV1 se voit alloué dans les configurations 1*c* et 1*e* une charge de calcul (i.e. un nombre de lignes de la matrice C à calculer) directement proportionnelle au nombre de processeurs virtuels auxquels il est équivalent [**F**].

Stratégie 2

La stratégie 2 définit chaque ligne de la matrice A comme un sous-domaine. Une sous-tâche consiste donc à calculer *NB_ITER* fois une ligne de la matrice C. Le maître est dédié à l'allocation des sous-tâches aux esclaves. Précisons que deux ordinateurs au moins sont nécessaires pour mettre en oeuvre cette stratégie.

Nous avons considéré les cinq configurations de réseau suivantes:

- configuration 2*a*: seul SRV1 est utilisé comme *Maître* et MSG3 comme *Esclave1* (le produit de matrices est calculé "séquentiellement" sur MSG3),
- configuration 2*b*: SRV2 est utilisé comme *Maître* et MSG3 comme *Esclave1* (le produit de matrices est calculé "séquentiellement" sur MSG3),

- configuration 2*c*: SRV2 est utilisé comme *Maître* et SRV1 comme *Esclave1* (le produit de matrices est calculé "séquentiellement" sur SRV1),
- configuration 2*d*: SRV1 est utilisé comme *Maître*, SRV2 comme *Esclave1* et MSG3 comme *Esclave2*,
- configuration 2*e*: SRV2 est utilisé comme *Maître*, SRV1 comme *Esclave1* et MSG3 comme *Esclave2*.

Les configurations 2*a* et 2*b* permettent d'estimer l'influence de la vitesse de traitement du maître sur la gestion des sous-tâches. Les configurations 2*a* et 2*c*, ainsi que les configurations 2*d* et 2*e*, peuvent fournir des éléments de réponse à la question de savoir quel est le meilleur choix pour cette stratégie, un maître rapide et quelques esclaves virtuels, ou bien un maître relativement lent avec plus d'esclaves virtuels.

Stratégie 3

La stratégie 3 est similaire à la stratégie 2, mais en plus de la gestion des sous-tâches, le maître participe à leur calcul, au même titre que les esclaves, et il est interrompu chaque fois qu'un esclave achève sa sous-tâche. Chaque sous-tâche consiste toujours à calculer *NB_ITER* fois une ligne de la matrice C.

Pour cette stratégie, sept configurations de réseau ont été étudiées:

- configuration 3*a*: seul SRV1 est utilisé comme *Maître* (le produit de matrices est calculé séquentiellement sur SRV1),
- configuration 3*b*: seul SRV2 est utilisé comme *Maître* (le produit de matrices est calculé séquentiellement sur SRV2),
- configuration 3*c*: SRV1 est utilisé comme *Maître* et MSG3 comme *Esclave1*,
- configuration 3*d*: SRV2 est utilisé comme *Maître* et MSG3 comme *Esclave1*,
- configuration 3*e*: SRV2 est utilisé comme *Maître* et SRV1 comme *Esclave1*,
- configuration 3*f*: SRV1 est utilisé comme *Maître*, SRV2 comme *Esclave1* et MSG3 comme *Esclave2*,

- configuration $3g$: SRV2 est utilisé comme *Maître*, SRV1 comme *Esclave1* et MSG3 comme *Esclave2*.

Les configurations $3c$ et $3e$, ainsi que les configurations $3f$ et $3g$, peuvent apporter des éléments de réponse à la question de savoir s'il est préférable que le maître soit l'ordinateur le plus rapide ou non. Les configurations $3c$ et $2e$, ainsi que les configurations $3d$ et $2d$, sont intéressantes pour comparer les efficacités respectives des stratégies 2 et 3, car elles mettent en oeuvre, deux à deux, le même nombre de **processeurs virtuels de calcul**.

Méthode de Relaxation

Pour la méthode de relaxation, la matrice A est une matrice régulière carrée de dimension 49. Comme pour le produit de matrices, cette dimension a la valeur maximale que nous avons pu considérer, compte tenu de l'espace en mémoire vive disponible sur chaque PC, ne voulant pas avoir recours au disque pour le stockage des données.

Le seuil de convergence de la méthode itérative est initialisé au début de l'exécution du programme et on l'a fait varier entre 10^{-5} et 10^{-12}, afin d'augmenter le nombre d'itérations nécessaires à la convergence et observer ainsi le comportement du programme en fonction de sa durée d'exécution.

Seule la stratégie à allocation statique des sous-tâches a été mise en oeuvre pour les raisons évoquées au paragraphe 4.3.2. Décrite schématiquement, cette stratégie consiste à allouer statiquement à chaque processeur physique un ensemble séparé de composantes du vecteur solution x à calculer. Le test de convergence est effectué sur le maître, qui prend en charge les composantes d'indices supérieurs du vecteur x.

Cinq configurations de réseau ont été considérées:

- configuration Ia: seul SRV1 est utilisé comme *Maître* (le système d'équations linéaires est résolu séquentiellement sur SRV1),
- configuration Ib: seul SRV2 est utilisé comme *Maître* (le système d'équations linéaires est résolu séquentiellement sur SRV2),
- configuration Ic: SRV1 est utilisé comme *Maître* et MSG3 comme *Esclave1*,
- configuration Id: SRV2 est utilisé comme *Maître* et MSG3 comme *Esclave1*,

- configuration *Ie*: SRV1 est utilisé comme *Maître*, SRV2 comme *Esclave1* et MSG3 comme *Esclave2*.

Comme pour le produit de matrices, les configurations *Ia* et *Ib* permettent de déterminer le nombre de **processeurs virtuels** auxquels l'IBM PC AT est équivalent, sachant que l'IBM PC XT est là encore le processeur de référence et équivaut donc à un processeur virtuel.

Afin de garantir un bon équilibrage de charge entre les sous-tâches, SRV1 se voit alloué dans les configurations *Ic* et *Ie* une charge de calcul (c'est-à-dire un nombre de composantes du vecteur solution x à calculer) directement proportionnelle au nombre de processeurs virtuels auxquels il est équivalent [**F**].

4.5.2 Description des Résultats

Ce paragraphe détaille l'ensemble des résultats que nous avons obtenus lors de la campagne de mesures décrite précédemment.

Le principal critère de performance que nous avons mesuré est le temps de réponse du réseau pour exécuter le produit des matrices A et B ou pour résoudre le système d'équations linéaires $Ax = b$.

A partir de ce temps de réponse, nous avons déduit, pour chaque stratégie et pour chaque configuration, l'accélération réalisée par l'utilisation de plusieurs processeurs virtuels pour exécuter le programme parallèle. Cette accélération, notée $S(p)$, prend en compte la notion de **processeur virtuel** [**U**], et elle est définie par l'expression:

$$S(p) = \frac{T(1)}{T(p)}$$

où p représente le nombre de **processeurs virtuels de calcul** utilisés, $T(1)$ le temps de réponse avec un seul processeur virtuel et $T(p)$ le temps de réponse avec p processeurs virtuels.

Nous rappelons que, pour les deux programmes d'application étudiés, le processeur de référence est l'IBM PC XT, qui équivaut donc à un seul processeur virtuel.

Le temps de réponse $T(p)$ représente en fait le temps séparant la fin de l'établissement des sessions entre le maître et les esclaves et le début de la fermeture de ces mêmes sessions. Il a été mesuré par insertion de deux appels à l'horloge système dans le code source du module du maître. Précisons enfin qu'il est exprimé en secondes dans les tableaux de résultats présentés ci-après.

Produit de Matrices

Les tables 4.1, 4.2 et 4.3 présentent l'ensemble des résultats obtenus respectivement pour les stratégies 1, 2 et 3.

Pour chaque configuration (cf paragraphe 4.5.1) et chaque valeur de *NB_ITER*, ces tables donnent le nombre p de processeurs virtuels, le temps de réponse correspondant $T(p)$ ainsi que l'accélération $S(p)$.

La configuration $1b$ fournit les valeurs de $T(1)$, pour les différentes valeurs de *NB_ITER*.

Stratégie 1	Configurations (Nombre p de Processeurs Virtuels)				
	$1a(4.7)$	$1b(1.0)$	$1c(5.7)$	$1d(2.0)$	$1e(6.7)$
NB_ITER	$T(p)$	$T(p)$	$T(p)$	$T(p)$	$T(p)$
	$S(p)$	$S(p)$	$S(p)$	$S(p)$	$S(p)$
	14.90	70.03	13.24	37.07	11.86
1	4.70	1.00	5.29	1.89	5.90
	29.80	140.06	25.10	71.90	21.64
2	4.70	1.00	5.58	1.95	6.47
	44.70	210.09	37.68	106.78	32.35
3	4.70	1.00	5.58	1.97	6.49
	59.54	280.12	50.15	141.65	43.06
4	4.70	1.00	5.59	1.98	6.51
	74.48	350.15	62.67	176.42	53.83
5	4.70	1.00	5.59	1.98	6.50
	89.31	420.18	75.19	211.30	64.54
6	4.70	1.00	5.59	1.99	6.51
	104.25	490.21	87.66	246.12	75.24
7	4.70	1.00	5.59	1.99	6.52
	119.14	560.30	100.18	280.95	85.96
8	4.70	1.00	5.59	1.99	6.52
	134.02	630.32	112.70	315.82	96.72
9	4.70	1.00	5.59	2.00	6.52
	148.90	700.30	125.23	350.64	107.43
10	4.70	1.00	5.59	2.00	6.52

Table 4.1: Produit de Matrices: Stratégie 1

Pour les stratégies 2 et 3, les tables présentent en outre le nombre n de sous-tâches exécutées par chaque *processeur physique*. Ce nombre est très intéressant, car il permet d'apprécier la qualité de l'équilibrage de charge entre les ordinateurs. Bien sûr, il faut le relier au nombre de **processeurs virtuels** auxquels chaque ordinateur est équivalent pour le produit de matrices.

Précisons enfin que le nombre p représente, dans le cas de la stratégie 2, le nombre de **processeurs virtuels de calcul**, et qu'il n'inclut donc pas le(s) processeur(s) virtuel(s) du maître.

Stratégie 2		Configurations (Nb. Processeurs Virtuels de Calcul)									
NB_ITER		2a (1.0)		2b (1.0)		2c (4.7)		2d (2.0)		2e (5.7)	
		n	T(p) S(p)	n	T(p) S(p)	n	T(p) S(p)	n	T(p) S(p)	n	T(p) S(p)
1	SRV1 SRV2 MSG3	0 50	77.34 0.91	0 50	77.34 0.91	50 0	21.31 3.29	0 25 25	39.99 1.75	40 0 10	17.25 4.06
2	SRV1 SRV2 MSG3	0 50	147.04 0.95	0 50	147.20 0.95	50 0	37.35 3.75	0 25 25	74.54 1.88	40 0 10	31.19 4.49
3	SRV1 SRV2 MSG3	0 50	216.62 0.97	0 50	216.68 0.97	50 0	52.18 4.03	0 25 25	109.19 1.92	40 0 10	44.93 4.68
4	SRV1 SRV2 MSG3	0 50	286.32 0.98	0 50	286.32 0.98	50 0	67.94 4.12	0 25 25	144.46 1.94	40 0 10	58.94 4.75
5	SRV1 SRV2 MSG3	0 50	356.02 0.98	0 50	356.24 0.98	50 0	83.10 4.21	0 25 25	179.27 1.95	40 0 10	72.99 4.80
6	SRV1 SRV2 MSG3	0 50	425.78 0.99	0 50	425.78 0.99	50 0	98.87 4.25	0 25 25	214.32 1.96	40 0 10	86.67 4.85
7	SRV1 SRV2 MSG3	0 50	495.48 0.99	0 50	495.43 0.99	50 0	114.41 4.28	0 25 25	249.31 1.97	40 0 10	100.68 4.87
8	SRV1 SRV2 MSG3	0 50	564.97 0.99	0 50	565.30 0.99	50 0	129.74 4.32	0 25 25	284.29 1.97	40 0 10	114.80 4.88
9	SRV1 SRV2 MSG3	0 50	634.61 0.99	0 50	634.77 0.99	50 0	145.61 4.33	0 25 25	319.28 1.97	40 0 10	128.53 4.90
10	SRV1 SRV2 MSG3	0 50	704.31 0.99	0 50	704.52 0.99	50 0	160.60 4.36	0 25 25	354.27 1.98	40 0 10	142.75 4.91

Table 4.2: Produit de Matrices: Stratégie 2

Stratégie 3		Configurations (Nombre de Processeurs Virtuels)													
NB_ITER		3a (4.7)		3b (1.0)		3c (5.7)		3d (2.0)		3e (5.7)		3f (6.7)		3g (6.7)	
		n	T(p) S(p)	n	T(p) S(p)	n	T(p) S(p)	n	T(p) S(p)	n	T(p) S(p)	n	T(p) S(p)	n	T(p) S(p)
1	SRV1 SRV2 MSG3	50	17.52 4.00	50	73.44 0.95	41 9	15.71 4.46	26 24	38.62 1.81	39 11	16.69 4.20	35 8 7	13.01 5.38	33 9 8	14.22 4.92
2	SRV1 SRV2 MSG3	50	33.39 4.19	50	143.52 0.98	41 9	28.01 5.00	25 25	74.64 1.88	40 10	29.71 4.71	34 8 8	25.15 5.57	33 9 8	26.37 5.31
3	SRV1 SRV2 MSG3	50	49.27 4.26	50	213.61 0.98	41 9	40.59 5.18	25 25	109.30 1.92	40 10	43.29 4.85	34 8 8	36.31 5.79	34 8 8	36.26 5.79
4	SRV1 SRV2 MSG3	50	65.09 4.30	50	283.74 0.99	41 9	53.61 5.23	25 25	144.12 1.94	40 10	57.29 4.89	34 8 8	47.45 5.90	34 8 8	47.51 5.90
5	SRV1 SRV2 MSG3	50	81.73 4.28	50	353.77 0.99	41 9	67.12 5.22	25 25	179.00 1.96	40 10	71.30 4.91	34 8 8	58.55 5.98	34 8 8	58.61 5.97
6	SRV1 SRV2 MSG3	50	97.61 4.30	50	423.86 0.99	40 10	86.62 4.85	25 25	213.82 1.97	40 10	85.30 4.93	34 8 8	69.76 6.02	34 8 8	69.59 6.04
7	SRV1 SRV2 MSG3	50	113.53 4.32	50	493.83 0.99	40 10	100.57 4.87	25 25	248.87 1.97	40 10	99.25 4.94	34 8 8	80.91 6.06	34 8 8	81.02 6.05
8	SRV1 SRV2 MSG3	50	129.35 4.33	50	563.86 0.99	40 10	114.69 4.89	25 25	283.47 1.98	40 10	113.26 4.95	34 8 8	92.22 6.08	34 8 8	91.95 6.09
9	SRV1 SRV2 MSG3	50	145.17 4.34	50	633.68 0.99	40 10	128.63 4.90	25 25	318.35 1.98	40 10	127.26 4.95	34 8 8	103.26 6.10	34 8 8	103.20 6.11
10	SRV1 SRV2 MSG3	50	161.10 4.35	50	703.65 1.00	40 10	142.31 4.92	25 25	353.22 1.98	40 10	141.27 4.96	34 8 8	114.30 6.13	34 8 8	114.30 6.13

Table 4.3: Produit de Matrices: Stratégie 3

Afin de comparer les efficacités respectives des différentes stratégies d'implémentation, la figure 4.3 propose les courbes de l'accélération réalisée avec ces stratégies lorsque *NB_ITER* vaut 6. Notons que l'accélération atteinte pour la configuration 1*a* n'est pas portée sur la courbe associée à la stratégie 1, car cette accélération est idéale par définition.

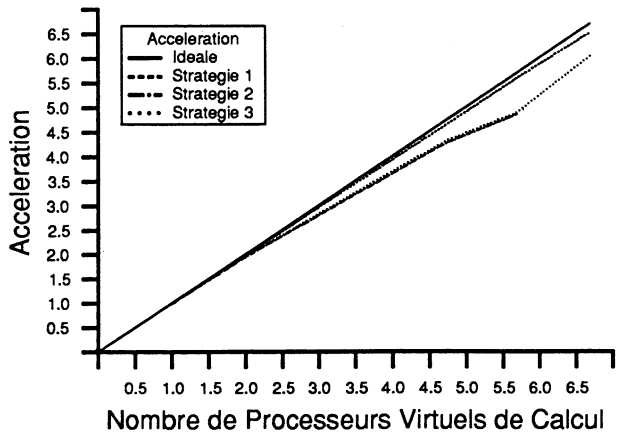

Figure 4.3: Efficacités Respectives des Stratégies d'Implémentation

Méthode de Relaxation

Comme pour le produit de matrices, la table 4.4 présente l'ensemble des résultats obtenus pour la méthode de relaxation, avec la stratégie 1. Les résultats sont donnés pour chaque configuration et chaque seuil de convergence.

Stratégie 1	Configurations (Nombre p de Processeurs Virtuels)				
	$Ia(3.9)$	$Ib(1.0)$	$Ic(4.9)$	$Id(2.0)$	$Ie(5.9)$
Seuil de Conv.	$T(p)$	$T(p)$	$T(p)$	$T(p)$	$T(p)$
Nb. Itérations	$S(p)$	$S(p)$	$S(p)$	$S(p)$	$S(p)$
10^{-5}	28.60	111.80	37.62	98.75	41.85
62 Iter.	3.91	1.00	2.97	1.13	2.67
10^{-6}	35.75	139.18	46.36	122.37	51.63
77 Iter.	3.89	1.00	3.00	1.14	2.70
10^{-7}	42.29	164.83	54.54	144.51	60.63
91 Iter.	3.90	1.00	3.02	1.14	2.72
10^{-8}	49.27	192.41	63.33	168.29	70.53
106 Iter.	3.91	1.00	3.04	1.14	2.73
10^{-9}	55.80	218.33	71.51	190.59	79.70
120 Iter.	3.91	1.00	3.05	1.15	2.74
10^{-10}	62.83	246.06	80.25	214.49	89.91
135 Iter.	3.92	1.00	3.07	1.15	2.74
10^{-11}	69.31	272.10	88.43	236.67	99.19
149 Iter.	3.93	1.00	3.08	1.15	2.74
10^{-12}	76.41	299.83	97.33	260.62	108.58
164 Iter.	3.92	1.00	3.08	1.15	2.76

Table 4.4: Méthode de Relaxation: Stratégie 1

La figure 4.4 propose la courbe des accélérations réalisées lors de l'exécution parallèle de la méthode de relaxation. Comme pour le produit de matrices, l'accélération atteinte pour la configuration *Ia* n'est pas portée sur la courbe car elle est idéale par définition.

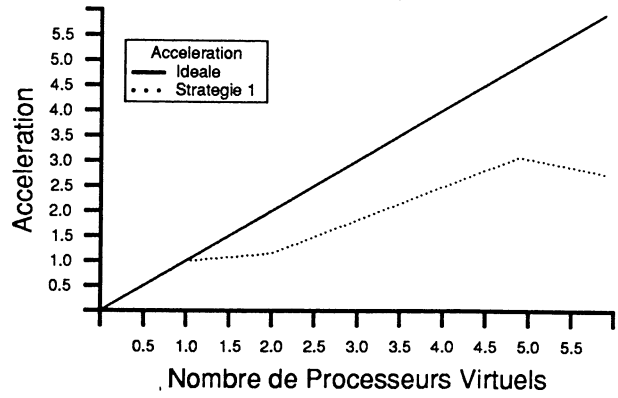

Figure 4.4: Méthode de Relaxation: Accélération

4.5.3 Interprétation des Résultats

Produit de Matrices

A partir de la figure 4.3, on peut tout d'abord constater que l'accélération atteinte est proche de sa valeur idéale, en particulier pour la stratégie 1.

A partir des tables 4.1, 4.2 et 4.3, il apparaît que l'accélération augmente avec la valeur de *NB_ITER*, soit avec la granularité des sous-tâches. Ce résultat est tout à fait typique des algorithmes parallèles; plus la granularité des tâches de calcul parallèles est forte, plus les communications et les synchronisations entre ces tâches sont espacées, et par conséquent moins le surcoût dû à la parallélisation est grand.

Les configurations 1*a* et 1*b* permettent de déduire que l'IBM PC AT est équivalent à 4,7 processeurs virtuels dans le cas du produit de matrices.

Si l'on s'intéresse à la question de savoir quel processeur physique est préférable pour le maître, il est naturel de prendre l'ordinateur le plus rapide pour la stratégie 1. Mais qu'en est-il pour les deux autres stratégies ?

Pour la stratégie 2, si l'on compare les résultats obtenus avec les configurations 2*a* et 2*c* d'une part, et avec les configurations 2*d* et 2*e* d'autre part, il est évident que le meilleur choix est d'utiliser

le processeur physique le plus rapide comme esclave (c'est-à-dire comme processeur de calcul), et que n'importe quel ordinateur plus lent peut s'occuper de la gestion des sous-tâches. On augmente ainsi considérablement le nombre de processeurs virtuels de calcul. En outre, les configurations $2a$ et $2b$ nous enseignent que, même si l'on maintient le même nombre de processeurs virtuels de calcul, il n'y a pratiquement aucun gain à utiliser le processeur physique le plus rapide comme maître.

Pour la stratégie 3, il apparaît, en comparant les résultats obtenus avec les configurations $3c$ et $3e$, que le maître devrait être le processeur le plus rapide si les sous-tâches sont de faible granularité, et qu'il devrait être un processeur plus lent lorsque les sous-tâches sont de granularité plus forte. Cependant, si l'on s'intéresse aux configurations $3f$ et $3g$, cette règle ne semble plus aussi évidente, et les deux choix paraissent équivalents dès que l'allocation des sous-tâches aux ordinateurs est identique dans les deux configurations. En réalité, il n'y a pas de loi générale pour cette stratégie, dans la mesure où les résultats sont étroitement liés aux nombres respectifs de sous-tâches que les différents ordinateurs obtiennent, en fonction de leur vitesse de traitement propre et de leur qualité (maître ou esclave). Il aurait été intéressant de réaliser d'autres mesures avec un nombre plus élevé d'ordinateurs présents sur le réseau. Malheureusement, nous ne disposions pas du matériel nécessaire.

Examinons maintenant l'équilibrage de charge entre les ordinateurs réalisé par les deux stratégies effectuant une allocation dynamique des sous-tâches.

Pour la stratégie 2 (cf table 4.2), on peut observer que le nombre de sous-tâches allouées à chaque ordinateur n'évolue pas avec la granularité.

Pour la stratégie 3 (cf table 4.3), dès que deux processeurs physiques au moins sont utilisés, l'allocation des sous-tâches aux ordinateurs n'est pas constante pour les petites valeurs de *NB_ITER*. Ce phénomène est dû au fait que le maître s'alloue des sous-tâches localement (c'est-à-dire sans aucune communication sur le réseau). Il a par conséquent le temps d'exécuter plusieurs sous-tâches, lorsque celles-ci sont de faible granularité, pendant que les esclaves reçoivent le noyau commun à toutes les sous-tâches. Le nombre de sous-tâches, que le maître peut ainsi exécuter, est directement lié à leur granularité. Lorsque celle-ci augmente, les communications entre le maître et les esclaves sont moins sensibles et l'attribution des tâches tend à se stabiliser.

Au contraire, pour la stratégie 2, tous les processeurs de calcul (c'est-à-dire les esclaves) sont sur un pied d'égalité, car ils reçoivent tous leurs sous-tâches au travers du réseau. La répartition des sous-tâches reste stable lorsque leur granularité varie, et elle réalise un très bon équilibrage de charge entre les ordinateurs (cf configurations $2d$ et $2e$).

Ajoutons que, lorsque la granularité des sous-tâches est plus forte, les répartitions des sous-tâches entre les ordinateurs sont les mêmes, avec les deux stratégies, pour des configurations similaires, telles que les configurations $2d$ et $3d$ et les configurations $2e$ et $3c$.

Si l'on veut comparer les stratégies 2 et 3, une question intéressante est de savoir quelle stratégie réalise la meilleure accélération pour un même nombre de processeurs virtuels de calcul.

Au premier abord, on pourrait être tenté de répondre que c'est la stratégie 2, car le maître doit, dans la stratégie 3, s'occuper de l'allocation des sous-tâches aux esclaves en plus de leur calcul.

Si l'on étudie de près les résultats obtenus avec les configurations $2d$ et $3d$ d'une part, et avec les configurations $2e$ et $3c$ d'autre part, on se rend compte que le surcoût induit sur le maître par la gestion des interruptions, dans la stratégie 3, est compensé par la réduction des communications sur le réseau, du fait que le maître s'alloue des sous-tâches localement.

Lorsque les sous-tâches sont de faible granularité, il apparaît que le maître, dans la stratégie 3, s'approprie plus de tâches que l'esclave lui correspondant n'en reçoit dans la stratégie 2, et de ce fait, l'accélération obtenue pour la stratégie 3 est supérieure à celle relevée pour la stratégie 2.

Même lorsque les répartitions des sous-tâches sont similaires pour les deux stratégies, la stratégie 3 semble légèrement plus efficace que la stratégie 2, ce qui apparaît sur la figure 4.3.

Si, maintenant, on tient compte du fait que la stratégie 2 requiert un ordinateur supplémentaire dédié à la gestion des sous-tâches, il est clair que la stratégie 3 est meilleure que la stratégie 2.

On sera cependant prudent lors de cette affirmation. En effet, toutes les mesures ont été effectuées avec un nombre réduit d'ordinateurs, et le maître, dans la stratégie 3, n'était donc pas surchargé par la gestion des interruptions. Avec un nombre plus élevé d'esclaves, les résultats de cette comparaison entre les efficacités respectives des stratégies 2 et 3 pourraient être inversés, du fait d'une gestion des interruptions plus complexe, qui pourrait même s'avérer être une source d'erreurs dans certains cas. Malheureusement, il ne nous a pas été possible de réaliser des mesures avec un plus grand nombre d'ordinateurs, susceptibles de confirmer ou d'infirmer cette thèse.

La figure 4.3 nous permet de comparer les efficacités respectives des trois stratégies d'implémentation du produit de matrices.

Il apparaît clairement que la stratégie 1 est la meilleure des trois stratégies. Précisons cependant qu'elle requiert une bonne adéquation entre l'algorithme implémenté et le nombre de processeurs physiques utilisés, afin d'assurer un bon équilibrage de charge.

Par contre, les stratégies 2 et 3 permettent un partitionnement dynamique de la tâche initiale et un ordonnancement dynamique de l'exécution des sous-tâches obtenues. De plus, ces deux stratégies garantissent un bon équilibrage de charge entre les ordinateurs de calcul, quelles que soient les durées respectives des sous-tâches et quelles que soient les vitesses de calcul respectives des ordinateurs.

Les mesures, que nous avons effectuées, montrent enfin que la stratégie 3 semble légèrement plus efficace que la stratégie 2, lorsqu'on ne tient compte que des processeurs virtuels de calcul et que le nombre de processeurs physiques est réduit.

Méthode de Relaxation

Contrairement au produit de matrices, la méthode de relaxation illustre ce qui peut se produire lorsqu'une bonne parallélisation est délicate à implémenter.

L'accélération réalisée avec la stratégie 1 (cf figure 4.4) n'est plus une fonction croissante du nombre de processeurs virtuels utilisés. La courbe présente un maximum.

Si l'on observe les résultats de la table 4.4, on peut noter que l'accélération ne dépend pas du nombre d'itérations nécessaires à la convergence. Une légère accélération apparaît lorsqu'on utilise deux processeurs virtuels (configuration Id), mais, de 3,9 à 5,9 processeurs virtuels, l'accélération décroît régulièrement (configurations Ia, Ic et Ie).

Le problème est lié au fait qu'à chaque itération, tous les ordinateurs, sauf le premier, doivent attendre les valeurs des composantes du vecteur x calculées par leurs prédécesseurs, avant de pouvoir terminer le calcul des composantes du vecteur x qui leur sont propres. Une forte sérialisation des sous-tâches en découle, et le surcoût dû aux transferts de données entre ordinateurs devient très vite plus important que le faible gain réalisé par la parallélisation des sous-tâches de calcul.

4.6 Conclusion

Nous avons présenté dans ce chapitre trois stratégies d'implémentation d'algorithmes parallèles sur un réseau local d'IBM PC, basées sur un partitionnement de données et une structure hiéarchique de type maître-esclaves entre les ordinateurs présents sur le réseau.

Ces trois stratégies ont été illustrées au travers de deux algorithmes parallèles, le produit de matrices et la méthode itérative de relaxation.

Nous avons introduit la notion de **processeur virtuel** nécessaire à assurer un bon équilibrage de charge entre processeurs physiques de calcul hétérogènes. Cette notion a permis en outre d'exprimer l'accélération réalisée lors de l'exécution parallèle des deux algorithmes cités précédemment sur un réseau comportant des ordinateurs hétérogènes.

Grâce à une campagne de mesures, nous avons comparé les efficacités respectives des trois stratégies d'implémentation et essayé de caractériser les raisons qui peuvent inciter l'utilisateur à mettre en oeuvre une stratégie plutôt qu'une autre.

Ainsi, la stratégie à allocation statique des sous-tâches est la plus efficace et doit être employée si le programme d'application permet son utilisation. Pour ce faire, le domaine d'étude de l'application ainsi que le nombre d'ordinateurs présents sur le réseau doivent être compatibles. De plus, si ces ordinateurs sont hétérogènes, une connaissance assez précise de leurs vitesses respectives de traitement, pour le programme d'application considéré, est nécessaire, afin de réaliser un bon équilibrage de charge entre eux (application de la notion de processeur virtuel).

Les deux stratégies à allocation dynamique des sous-tâches permettent une définition dynamique des sous-tâches et un ordonnancement dynamique de leur exécution. Elles peuvent donc être employées dans de nombreux cas. Elles réalisent de manière automatique un bon équilibrage de charge entre les ordinateurs, quelles que soient les durées respectives des sous-tâches et quelle que soit l'hétérogénéité des ordinateurs. La stratégie où le maître n'est pas dédié à la gestion des sous-tâches sera utilisée de préférence, si le réseau comporte moins d'une quinzaine d'ordinateurs.

Une modélisation de ce type d'architecture parallèle pourrait permettre d'élargir le cadre des résultats présentés ici. L'implémentation d'autres algorithmes parallèles pourrait être modélisée et la complexité du réseau local étudié pourrait être augmentée à loisir, entraînant ainsi une généralisation des règles d'utilisation des différentes stratégies présentées dans ce chapitre.

CHAPITRE 5

Floating Point Systems 264 (FPS 264)

Ce chapitre présente la modélisation du calculateur scientifique Floating Point Systems 264, dont le but premier est la conception d'un modèle sous forme de réseau de files d'attente, susceptible de constituer, pour le système ICAP (cf chapitre 6), le sous-modèle de chaque FPS264, considéré comme un agrégat du modèle global.

En fait, faute de temps d'une part, et à cause d'une modélisation des programmes différente pour le système ICAP de celle utilisée ici d'autre part (la modélisation des programmes pour le système ICAP a en effet été simplifiée à l'extrême, quant aux sections de calcul sur les FPS264, du fait de la très grande complexité du système global), le modèle décrit dans ce chapitre n'a pu être utilisé à ce jour comme agrégat du modèle global du système ICAP. Néanmoins, son utilisation future n'est nullement compromise, ainsi que nous l'expliquons au paragraphe 6.5.2.

Cette modélisation est très intéressante de par la **modélisation hybride** employée (basée sur un réseau de Petri stochastique dont la résolution fournit les paramètres d'entrée d'un modèle simplifié du calculateur sous forme de réseau de files d'attente) et une **méthode itérative d'agrégation-désagrégation** [J] soulignant tout l'intérêt de telles méthodes pour l'accélération de la résolution d'un modèle, voire la faisabilité même de sa résolution, rendue impossible par une méthode classique pour des contraintes de temps ou à cause d'erreurs d'arrondis, très critiques pour de gros systèmes.

Un autre intérêt de cette modélisation est la définition de classes de programmes qui permet l'obtention d'un modèle simplifié du calculateur sous forme de réseau de files d'attente pour chaque classe de programmes. Ainsi, pour un programme quelconque, après détermination de sa classe par des techniques empruntées à l'analyse des données [C], on utilisera le modèle du calculateur correspondant à cette classe pour modéliser l'exécution du programme sur chaque FPS264 du système ICAP.

De même qu'au chapitre 3, nous développons maintenant chaque phase de la modélisation. Précisons cependant qu'aucune prédiction de performances n'a été effectuée du fait des objectifs de cette étude.

5.1 Formulation des Objectifs

Comme nous l'avons mentionné précédemment, le but premier de cette étude est la conception d'un modèle sous forme de réseaux de files d'attente du calculateur scientifique FPS264, aussi simple que possible, afin qu'il puisse être utilisé pour modéliser, en tant qu'agrégat, chaque FPS264 du système parallèle ICAP, dont la modélisation fait l'objet du chapitre 6.

Pour diverses raisons, déjà énoncées, l'utilisation de ce modèle a été différée. Nous aborderons brièvement, au paragraphe 6.5.2, les modifications à apporter au modèle du système ICAP, pour qu'il puisse bénéficier du sous-modèle décrit dans ce chapitre et représenter plus finement chaque FPS264 présent sur le système.

Afin de mieux prendre en compte les nombreuses synchronisations nécessaires entre les dix unités fonctionnelles du calculateur pour qu'elles opèrent simultanément au rythme de l'horloge système, une modélisation très détaillée du calculateur sous forme de réseau de Petri, stochastique [Flori78,Shapi79,Balbo84] ou temporisé [Holid85], nous a semblé bien adaptée. Disposant de la chaîne de programmes RDPS [Flori85], un logiciel de résolution de réseaux de Petri stochastiques, nous avons opté pour une modélisation basée sur de tels réseaux.

Cependant, connaissant le temps nécessaire à la résolution de réseaux conséquents, nous étions assurés que la complexité de notre modèle compromettrait son utilisation pour représenter chaque FPS264 dans le modèle global du système ICAP. Notre intention a donc été de définir des classes de programmes et, pour chacune d'entre elles, un réseau de files d'attente, dérivé du réseau de Petri stochastique, susceptible de modéliser simplement l'exécution, sur chaque FPS264 du système ICAP, d'un programme quelconque appartenant à cette classe.

5.2 Analyse du Système

Le calculateur scientifique FPS264 peut être considéré comme un processeur de calcul à 64 bits très performant accessible à partir d'un ordinateur hôte, qui le considère comme un périphérique, avec lequel il échange des programmes binaires (code à exécuter) et des données (cf figure 5.1).

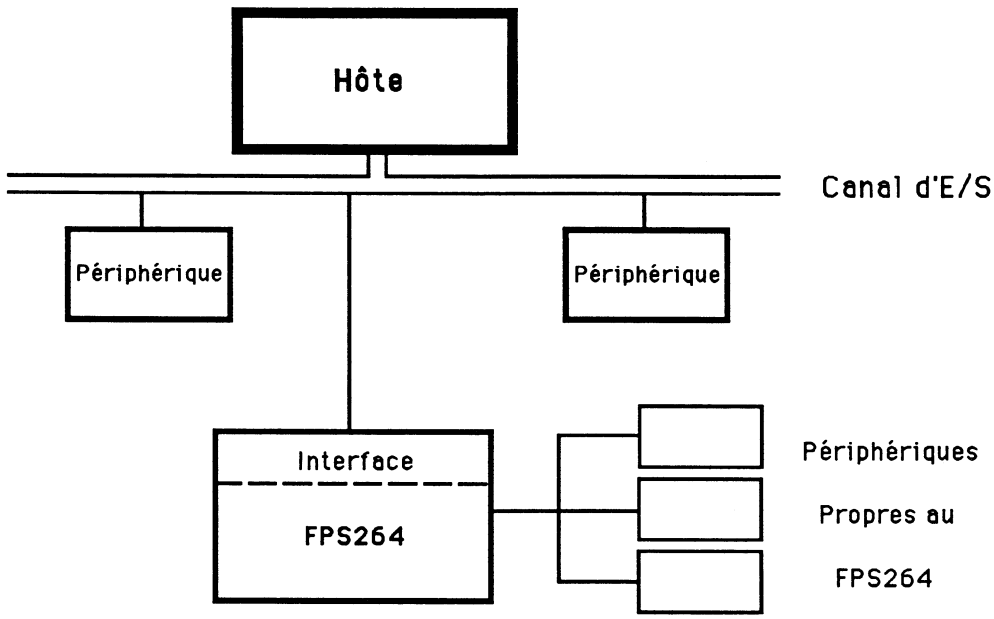

Figure 5.1: Mode d'Utilisation du FPS264

L'architecture générale du FPS264 est composée (cf figure 5.2):

- d'une unité de contrôle dont le temps de cycle est de 53 ns (avec préchargement du mot d'instruction suivant),
- d'une mémoire principale de 1536 kilomots de 64 bits, pipelinée sur trois étages permettant un accès à chaque temps de cycle (l'avancement des données dans le pipeline est automatique à chaque temps de cycle),
- d'une table des constantes de 1536 kilomots de 64 bits, pipelinée sur deux étages (l'avancement des données dans le pipeline est automatique à chaque temps de cycle),
- d'une mémoire cache formée de 4 bancs de 256 kilomots chacun,

- de deux unités arithmétiques à virgule flottante opérant sur des mots de 64 bits: un additionneur pipeliné sur deux étages et un multiplieur pipeliné sur trois étages (l'avancement des données dans ces deux pipelines est réalisé, en l'absence de nouvelles données alimentant le pipeline, par les deux instructions *FAPUSH* et *FMPUSH* pour l'additionneur et le multiplieur respectivement),
- d'une unité arithmétique et logique ("Scratch Pad Address Unit") calculant sur un tableau de 63 registres de 32 bits et utilisée essentiellement pour des calculs d'adresse,
- de deux bancs de 32 registres de données de 64 bits,
- et de plusieurs bus (bus de données, bus d'adresses) reliant entre elles les unités fonctionnelles.

L'unité de contrôle décode les mots d'instructions de 64 bits permettant de réaliser jusqu'à dix opérations pendant le même temps de cycle:

- une addition à virgule flottante,
- une multiplication à virgule flottante,
- une opération sur l'unité arithmétique et logique,
- deux accès mémoire dont un correspond au préchargement du mot d'instruction suivant,
- quatre accès aux registres de données (deux lectures et deux écritures),
- et une opération de branchement.

Une addition et une multiplication pouvant se terminer à chaque temps de cycle (après initialisation des pipelines), la performance maximale que peut atteindre le FPS264 est supérieure à 38 millions d'opérations en virgule flottante par seconde.

On définit le **degré de parallélisme** comme le nombre d'opérations réalisées pendant le même temps de cycle. Il est bien souvent inférieur à sa valeur maximale de 10, du fait de contentions pour l'accès aux différents bus.

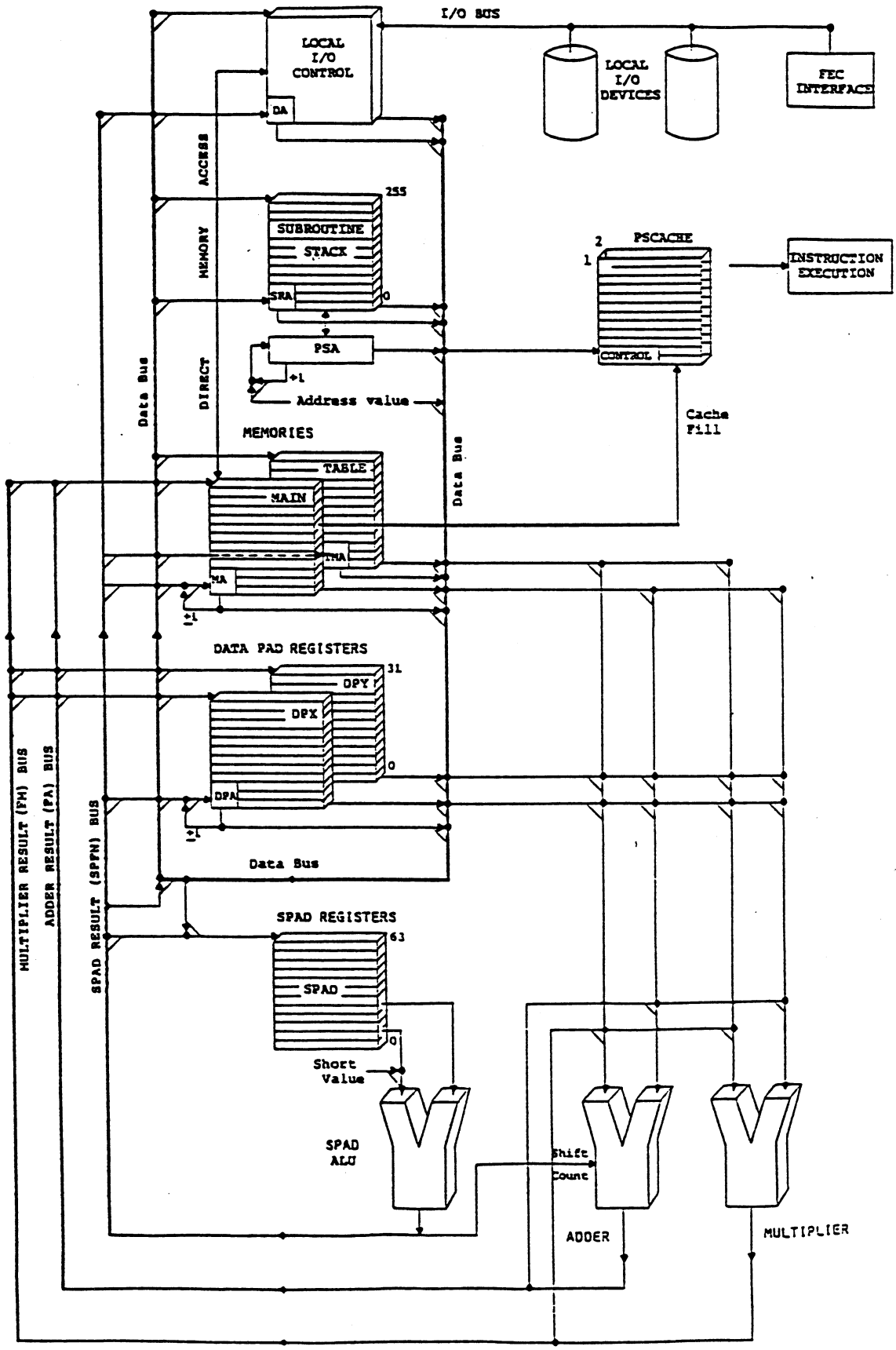

Figure 5.2: Architecture Générale du FPS264

5.3 Description du Modèle

Ainsi que nous l'avons expliqué précédemment, nous avons effectué une modélisation hybride en deux phases:

1. la première phase a consisté en une modélisation très détaillée de l'architecture du FPS264 et des programmes sous forme de réseau de Petri stochastique,
2. et la deuxième phase en une modélisation très simple du même système sous forme de réseau de files d'attente, les paramètres d'entrée du deuxième modèle étant déduit des critères de performances obtenus à l'issue de la résolution du premier modèle.

Nous allons successivement décrire chacune de ces phases, en présentant, ainsi que nous l'avons fait au chapitre 3, le modèle de l'architecture [**A**] d'une part, et le modèle des programmes [**B**] d'autre part.

5.3.1 Modélisation par Réseau de Petri Stochastique

Tout d'abord, rappelons très sommairement ce qu'est un réseau de Petri stochastique. Pour une information plus complète, le lecteur est invité à se reporter à la littérature (par exemple [Flori78,Shapi79,Balbo84,Flori85]).

Un réseau de Petri stochastique est constitué, comme tout réseau de Petri, de places connectées à des transitions par des arcs. Chaque place peut contenir des jetons et chaque arc est agrémenté d'un poids, valeur entière par défaut égale à 1.

Une transition est tirable lorsque chacune de ses places antécédentes contient le nombre de jetons correspondant au poids de l'arc qui la relie à la transition.

Lorsque la transition est tirée (voir condition de tir ci-après), chaque place antécédente à la transition perd le nombre de jetons indiqué par le poids de l'arc la reliant à cette transition et chaque place successeur de la transition reçoit un nombre de jetons égal au poids de l'arc la reliant à la transition.

La particularité des réseaux de Petri stochastiques réside dans l'attribution à chaque transition d'un **taux de tir**. Une transition tirable ne sera effectivement tirée qu'après un délai dont la durée suit une loi exponentielle de taux moyen égal au taux de tir de la transition. Un type particulier de

taux de tir, appelé **taux infini**, est utilisé pour modéliser un délai nul avant le tir de la transition associée. En d'autres termes, toute transition à taux de tir infini est tirée dès qu'elle est tirable.

Afin de modéliser des **branchements probabilistes**, on utilise plusieurs transitions connectées à une même place ou un même groupe de places et on pondère le même taux de base commun à toutes les transitions (représentant l'inverse du temps moyen de séjour d'un jeton dans la (ou les) place(s) antécédentes aux transitions) par la probabilité de branchement propre à chaque transition. Le taux moyen de n lois exponentielles en parallèle étant égal à la somme des n taux moyens, le taux moyen du branchement probabiliste est bien égal au taux de base commun à toutes les transitions.

Après ces quelques rappels, décrivons le modèle sous forme de réseau de Petri stochastique du calculateur scientifique FPS264. Très schématiquement, on peut dire que les places du réseau de Petri stochastique constituent le modèle de l'architecture, alors que les jetons correspondent au modèle des programmes.

Modèle de l'Architecture [A]

Le modèle de l'architecture est composé principalement de quatre sous-modèles représentant respectivement les **quatre unités fonctionnelles** que sont **l'additionneur** (*ADD*), **le multiplieur** (*MUL*), **la mémoire principale** (*MM*) et **la table des constantes** (*TM*).

Le bus de données (*DB*) et **les deux bancs de 32 registres de données** (*Dpx* et *Dpy*) sont modélisés comme des ressources, partagées par les quatre unités fonctionnelles précédentes. Ainsi, une opération telle que *Dpx* = *DB* (qui signifie "charger le registre Dpx avec la donnée présente sur le bus de données") n'est pas prise en compte par le modèle, alors qu'une opération telle que *FMUL Dpx, MD* (qui signifie "multiplier le contenu du registre Dpx avec le contenu du registre de la mémoire principale") l'est. Pendant la durée de la deuxième opération, le registre Dpx est considéré comme occupé.

L'unité arithmétique et logique ("Scratch Pad Address Unit") ainsi que ses 63 registres de 32 bits sont purement et simplement ignorés dans le modèle, car cette unité a un temps de cycle de 12 nanosecondes et son traitement peut être considéré comme instantané devant le temps de cycle de base (53 ns) du calculateur. De ce fait, le bus d'adresses est lui aussi absent du modèle.

Nous détaillons maintenant les modèles du bus de données, du multiplieur et de la mémoire principale. Les modèles des deux bancs de registres de données sont similaires à celui du bus de données, celui de l'additionneur est analogue à celui du multiplieur et le modèle de la table des constantes est semblable à celui de la mémoire principale.

Modèle du Bus de Données

Ce modèle est représenté par la figure 5.3.

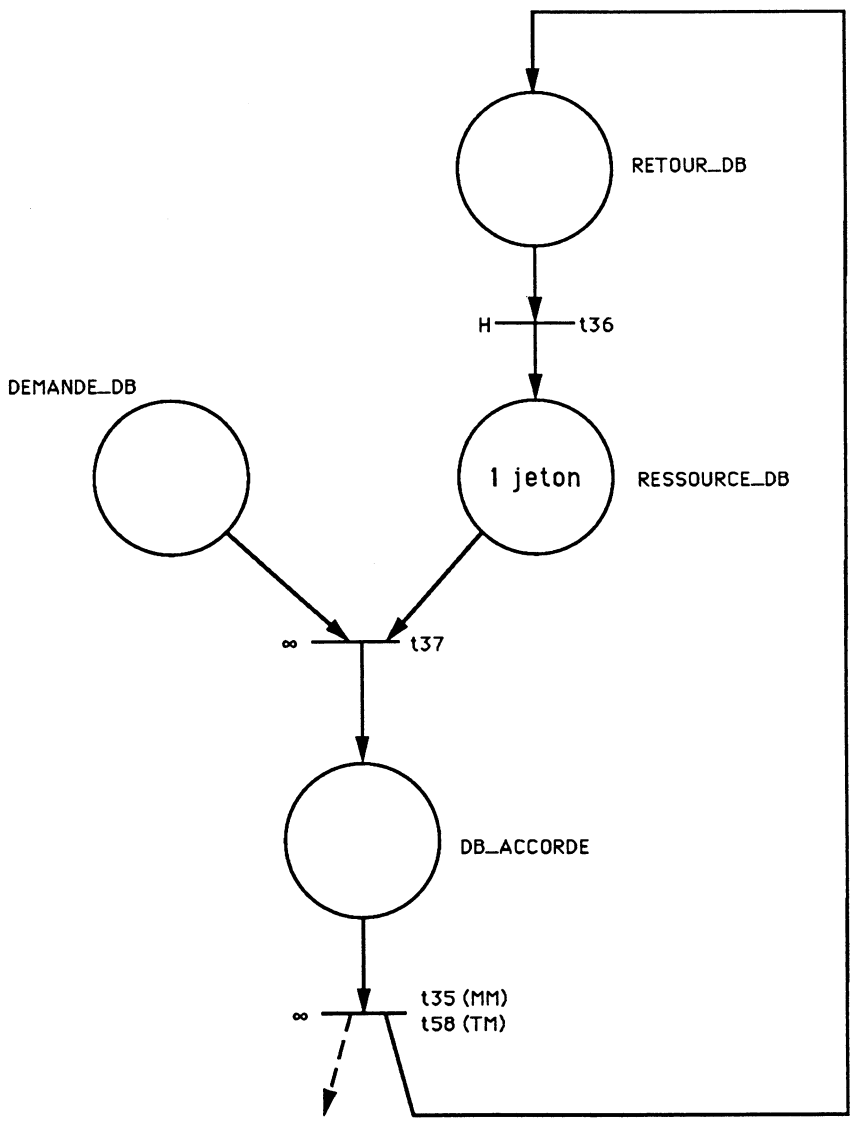

Figure 5.3: Modèle du Bus de Données (DB)

Lorsqu'une opération nécessite l'utilisation du bus de données, une demande d'occupation du bus est signifiée par l'ajoût d'un jeton dans la place *DEMANDE_DB*.

Si le bus est libre, c'est-à-dire si un jeton est présent dans la place *RESSOURCE_DB*, la transition *t37* est immédiatement tirée (taux infini). Le bus de données est alors alloué pour un accès en mémoire principale (transition *t35*) ou en table des constantes (transition *t58*).

Le jeton modélisant l'occupation du bus transite immédiatement dans la place *RETOUR_DB* où il séjourne pour une durée égale à un temps de cycle du calculateur. Afin de modéliser le **fonctionnement synchrone** de toutes les unités fonctionnelles à la cadence de l'horloge du calculateur, nous avons introduit un nouveau type de transition (dont la transition *t36* est un exemple), appelée **transition synchronisée par l'horloge** et représentée par un taux de tir constant (noté H) égal à la fréquence de l'horloge du calculateur. Ce faisant, nous avons étendu la définition des réseaux de Petri stochastiques classiques (cf paragraphe 5.5.1).

Modèle du Multiplieur

Ce modèle est constitué de trois parties (cf figure 5.4):

1. la première partie enregistre les demandes de multiplication et détermine les opérandes gauche et droit de chaque multiplication,
2. la deuxième partie modélise les trois étages du pipeline du multiplieur,
3. et la troisième partie achemine le résultat de la multiplication vers la bonne destination et déclenche le traitement de la prochaine instruction assembleur.

Détaillons maintenant chacune de ces trois parties.

La première partie résout tous les problèmes de contention pour l'accès au bus de données et aux deux bancs de registres de données.

Les deux bus d'entrée du multiplieur (l'un pour l'opérande gauche et l'autre pour l'opérande droit) ne doivent permettre qu'un accès par temps de cycle. Ceci a été modélisé par des arcs inhibiteurs entre les deux places *Demande_bus_F1*, *Demande_bus_F2*, et les transitions qui leur sont antécédentes. Ces arcs inhibiteurs n'ont pas été représentés sur la figure 5.4 afin que celle-ci reste lisible.

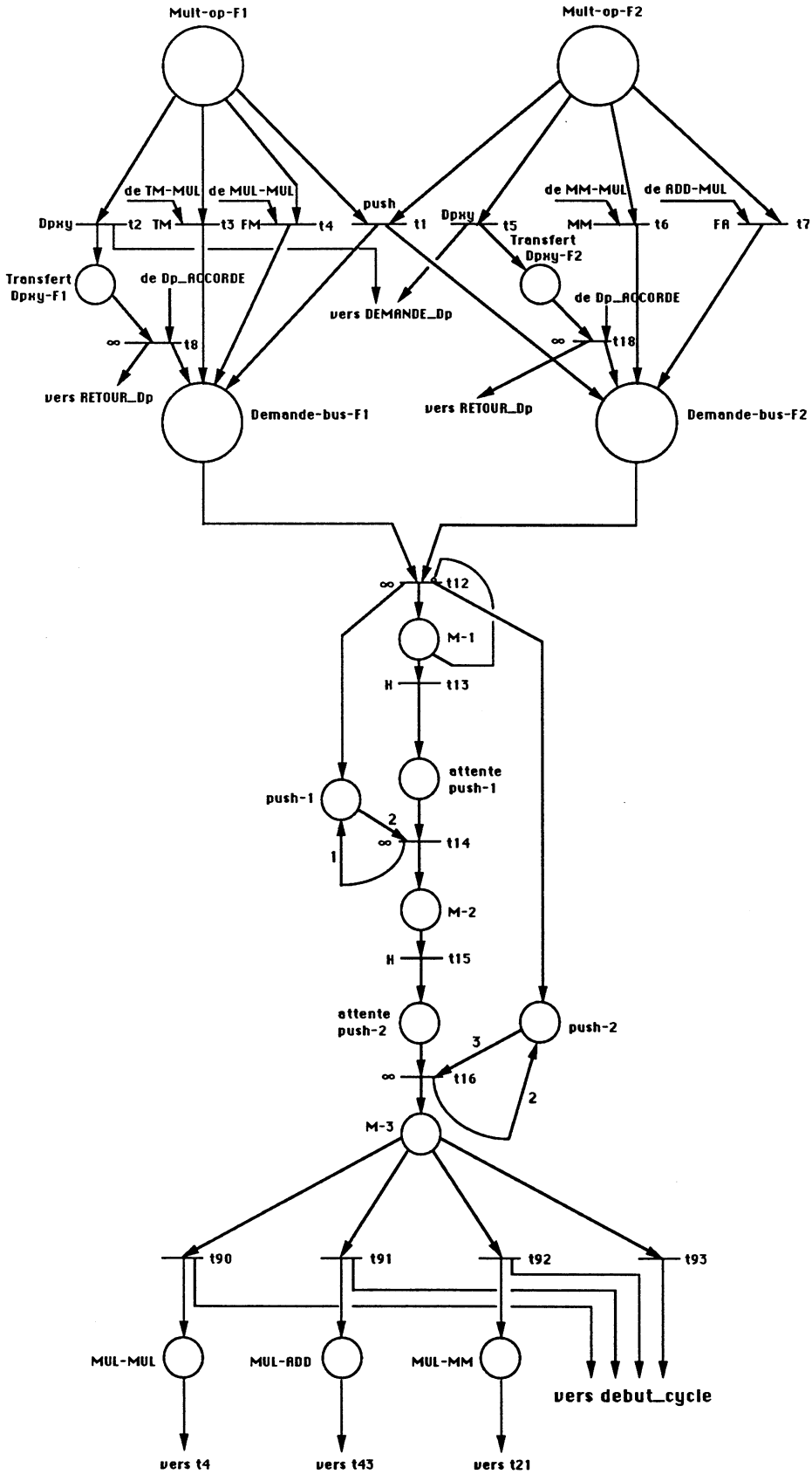

Figure 5.4: Modèle du Multiplieur (*MUL*)

La deuxième partie représente l'avancement des données dans le pipeline du multiplieur. Lorsqu'une nouvelle multiplication commence (transition *t12*), un jeton est envoyé dans le premier étage du pipeline (place *M_1*) d'une part, et un deuxième jeton est envoyé dans la place *push_1* d'autre part.

Si une multiplication précède cette nouvelle multiplication, un jeton est présent dans la place *attente_push_1* et il y a 2 jetons dans la place *push_1*. La transition *t14* est alors tirée immédiatement, un jeton reste dans la place *push_1*, et le second étage du pipeline reçoit un nouveau jeton (place *M_2*) pour modéliser l'avancement d'un étage de la première multiplication. A cet instant, les deux premiers étages du pipeline sont en cours de traitement pour une durée égale à un temps de cycle (les transitions *t13* et *t15* étant synchronisées par l'horloge).

De même, à l'arrivée de la nouvelle multiplication (transition *t12*), un jeton a également été envoyé dans la place *push_2*, si bien qu'une troisième multiplication en attente dans la place *attente-push_2* peut avancer jusqu'au troisième étage du pipeline (transition *t16* tirée immédiatement).

La troisième partie envoie le résultat de la multiplication vers la bonne destination (déterminée par le modèle des programmes - cf infra).

Ainsi, si la transition *t90* est tirée, le résultat devient l'opérande gauche d'une nouvelle multiplication (la transition *t4* est tirable). Le tir de la transition *t91* produit un opérande pour l'additionneur, celui de la transition *t92* une donnée qui sera stockée en mémoire principale.

D'autres résultats (transition *t93*) ne sont pas acheminés vers une unité fonctionnelle car ils correspondent en fait aux résultats virtuels d'opérations assembleur *FMPUSH*, qui forcent l'avancement des données dans les étages du pipeline, en l'absence de nouvelles multiplications alimentant le multiplieur (transition *t1*).

En fin de multiplication, une nouvelle instruction assembleur est prise en compte par le modèle des programmes par l'envoi d'un jeton dans la place *debut_cycle*.

Modèle de la Mémoire Principale

Le modèle de la mémoire principale (cf figure 5.5) comporte également trois parties, semblables aux trois parties du modèle du multiplieur:

1. la première partie précise si l'accès est une lecture ou une écriture, et s'il s'agit d'une écriture, elle détermine l'origine de la donnée,

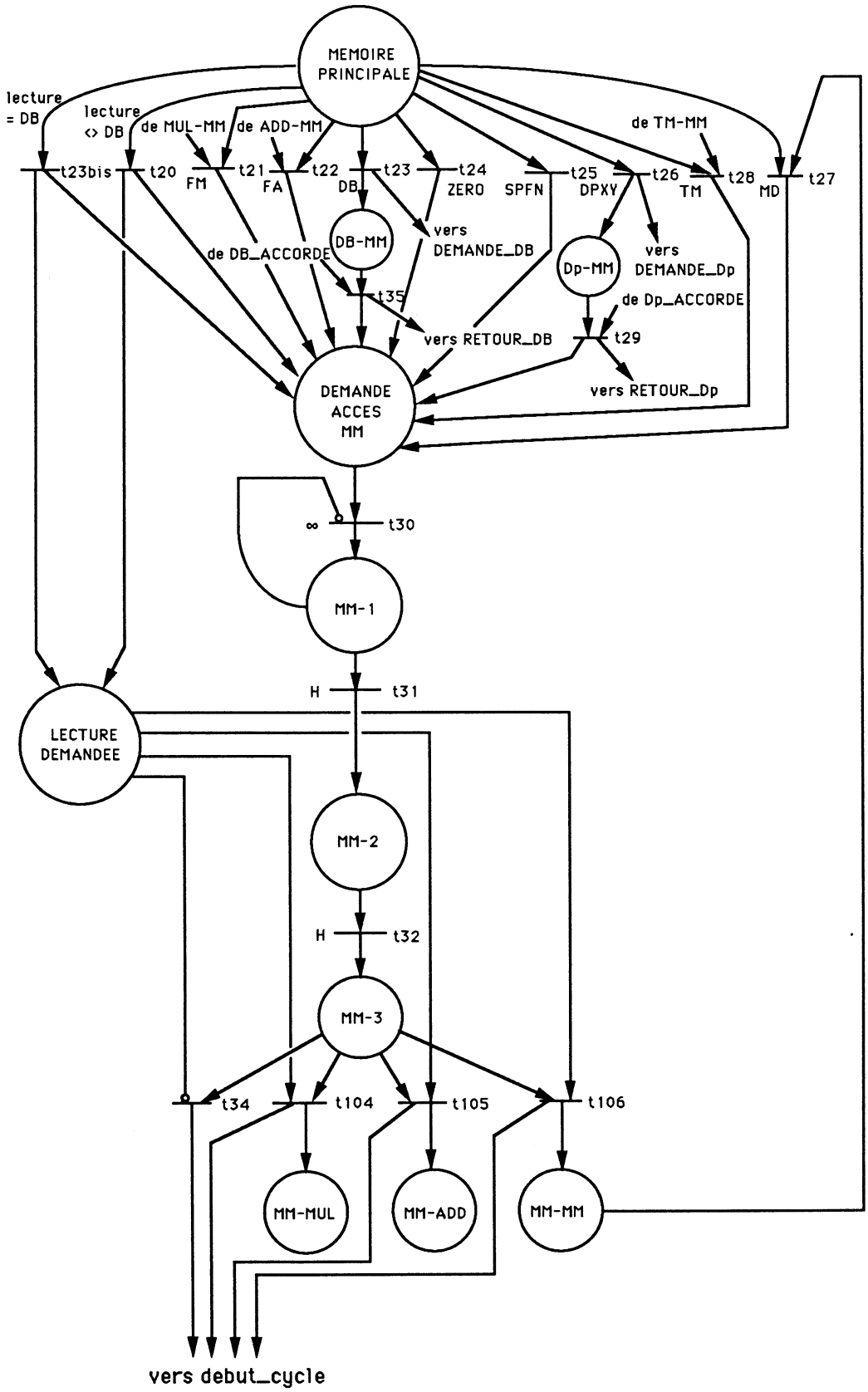

Figure 5.5: Modèle de la Mémoire Principale (*MM*)

2. la deuxième partie modélise les trois étages du pipeline,

3. la troisième partie se charge de l'acheminement de la donnée lue, si l'accès était une lecture, et déclenche le traitement de la prochaine instruction assembleur.

Lorsque l'accès est une lecture, un jeton est envoyé dans la place *LECTURE_DEMANDEE*. Après passage dans les trois étages du pipeline représentés par les places *MM_1*, *MM_2* et *MM_3* respectivement, le jeton présent dans la place *MM_3* s'associe à celui présent dans la place *LECTURE_DEMANDEE* pour modéliser l'acheminement de la donnée lue vers les autres unités fonctionnelles (transitions *t104*, *t105* et *t106*) et une nouvelle instruction assembleur est lancée (un jeton est envoyé dans la place *debut_cycle*).

Lorsque l'accès est une écriture, le jeton présent dans la place *MM_3* est directement envoyé dans la place *debut_cycle* pour lancer une nouvelle instruction assembleur (transition *t34*).

De nombreux travaux [Jalby,Lenfa,PenCh87] ont montré l'influence des conflits d'accès mémoire sur les performances d'algorithmes implantés sur des supercalculateurs. Malheureusement, la comptabilité système du FPS264 ne permet pas de calculer la fréquence d'occurrence des conflits d'accès mémoire, qui dépendent en outre directement des données du programme. Le modèle ne prend donc pas en compte les problèmes d'accès concurrents à un même banc de la mémoire principale.

Modèle des Programmes [B]

La modélisation des programmes doit pouvoir permettre d'alimenter le modèle de l'architecture en jetons. Le modèle de l'architecture étant essentiellement basé sur les unités fonctionnelles du calculateur, le niveau de représentation des programmes nécessite la connaissance très précise de l'utilisation de chaque unité fonctionnelle par le programme. Seul le code assembleur des programmes peut répondre à cette exigence. La modélisation des programmes est par conséquent basée sur leur **code assembleur APAL64 [D]**.

Afin de pouvoir **tester un large éventail de programmes**, aussi différents que possible quant à l'utilisation qu'ils font des unités fonctionnelles, il nous a semblé judicieux d'utiliser les procédures, écrites directement en assembleur APAL64, constituant la **bibliothèque mathéma-**

tique APMATH64 développée par Floating Point Systems.

A partir de l'étude du code assembleur de chaque procédure, on exhibe les caractéristiques constituant le modèle de cette procédure:

- **la probabilité de chaque degré de parallélisme** (les valeurs possibles pour le degré de parallélisme sont inférieures ou égales à 4, car seulement 4 unités fonctionnelles sont prises en compte dans le modèle), calculée sur l'ensemble des instructions composant le code assembleur, en tenant compte des niveaux d'imbrication des boucles éventuelles,
- **la répartition des opérations sur les quatre unités fonctionnelles**,
- et **la répartition de l'origine des opérandes pour chaque unité fonctionnelle**; cette répartition permet de définir des probabilités de branchement vers telle ou telle unité fonctionnelle pour chaque résultat d'une opération de lecture en mémoire, d'une addition ou d'une multiplication, et par là même, d'imposer un ordre de séquencement des opérations dans le modèle; par exemple, si l'on observe une addition utilisant comme opérande une sortie de la mémoire principale, on sait qu'une lecture en mémoire principale a précédé l'addition.

Nous verrons au paragraphe 5.4.1 la méthode employée pour obtenir de manière automatique ces caractéristiques à partir du code assembleur.

Exposons maintenant comment, à partir des caractéristiques du programme définies ci-dessus, générer les jetons dont la circulation à l'intérieur du modèle de l'architecture représentera l'exécution du programme sur le calculateur scientifique.

Pour ce faire, on a construit un réseau de Petri stochastique, appelé **modèle de contrôle** (cf figure 5.6), qui interconnecte les différents sous-modèles du modèle de l'architecture et qui assure véritablement le contrôle de la circulation des jetons à l'intérieur du modèle, soit de l'exécution du programme sur le calculateur.

Ce modèle est constitué de deux étages:

1. le premier étage détermine à chaque temps de cycle le nombre d'opérations à lancer simultanément en respectant les probabilités relatives des différents degrés de parallélisme possibles,
2. le deuxième étage répartit ces opérations sur les unités fonctionnelles à partir de la répartition exhibée lors de l'étude du code assembleur du programme.

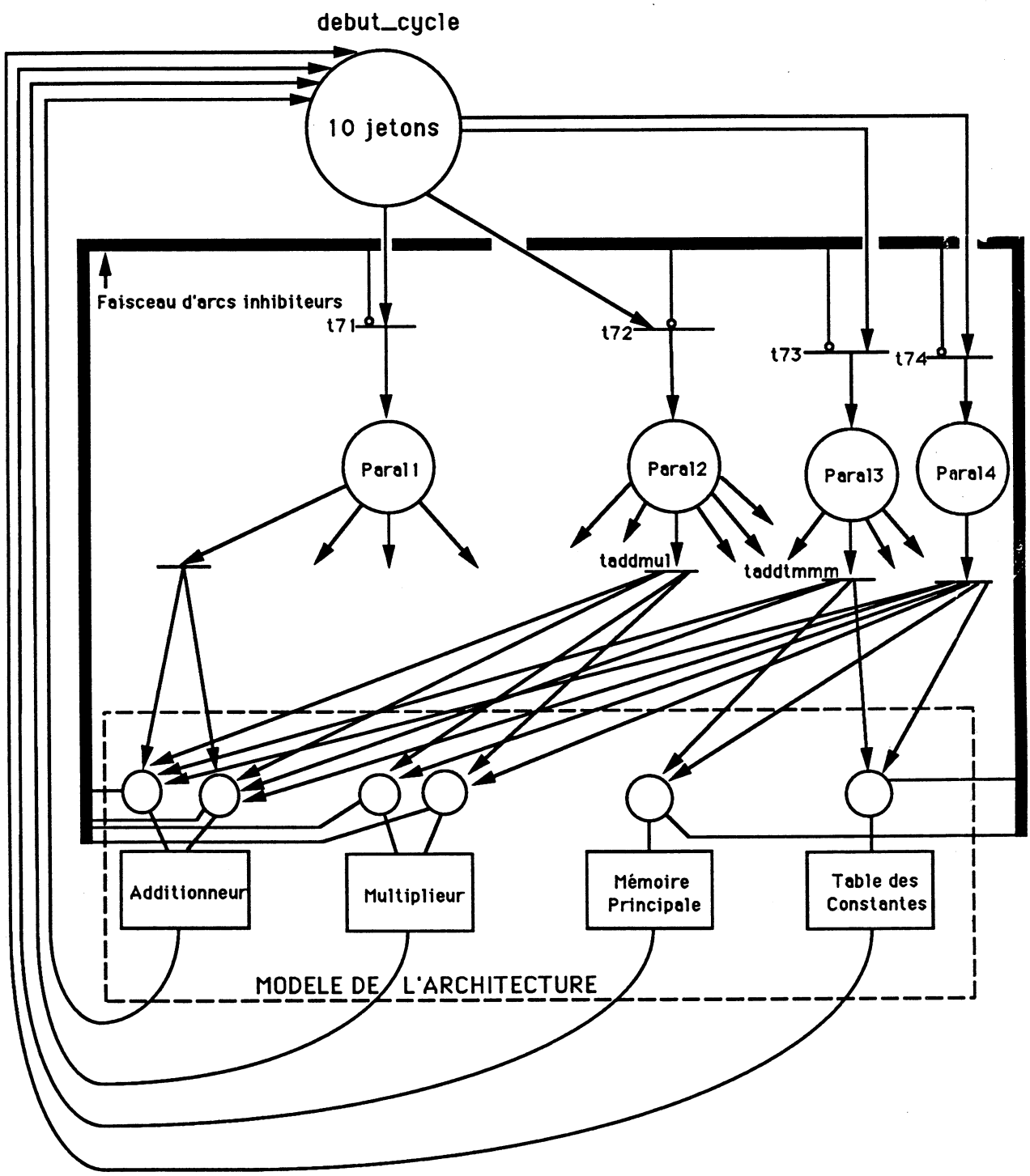

Figure 5.6: Modèle de Contrôle

Un faisceau d'arcs inhibiteurs, reliant ces deux étages, assure en outre la séquentialité de deux groupes consécutifs d'opérations et garantit qu'un seul groupe est lancé à chaque temps de cycle.

Détaillons maintenant chacun de ces deux étages.

Génération du Degré de Parallélisme [E]

Comme nous l'avons précisé plus haut, le degré de parallélisme est, à chaque temps de cycle, compris entre 1 et 4. La génération du degré de parallélisme [**E**] s'effectue donc à l'aide de quatre places, *Paral1*, *Paral2*, *Paral3* et *Paral4*, qui correspondent respectivement à des degrés de parallélisme de 1, 2, 3 et 4.

La place *debut_cycle* représente le début d'une nouvelle instruction assembleur, lancée au début de chaque temps de cycle. Cette place sera initialisée avec un nombre de jetons égal au nombre maximal d'opérations que l'on peut effectuer en parallèle sur les quatre unités fonctionnelles modélisées du calculateur, en tenant compte des différents étages des pipelines, soit avec un total de 10 jetons.

Les transitions *t71*, *t72*, *t73* et *t74* représentent un branchement probabiliste entre les différents degrés de parallélisme possibles, grâce à l'attribution à chacune d'entre elles d'un taux de tir égal à la probabilité relative du degré de parallélisme que sa place successeur représente.

Lorsque la transition précédant la place *Parali* est tirée, i jetons sont transférés de la place *debut_cycle* vers la place *Parali*.

Répartition des Opérations sur les Unités Fonctionnelles

Lorsque le degré de parallélisme du temps de cycle courant est i, les i jetons présents dans la place *Parali* doivent être répartis sur les quatre unités fonctionnelles. Il y a *C(4,i)* choix possibles, représentés par les *C(4,i)* arcs partant de la place *Parali*. Chacun de ces arcs est connecté à une transition d'où partent i groupes formés de 1 ou de 2 arcs, chaque groupe se dirigeant vers une des i unités fonctionnelles utilisées simultanément (le nombre d'arcs composant chaque groupe correspond au nombre d'opérandes de l'unité fonctionnelle où ils aboutissent). Les taux de tir de ces transitions

sont calculés de manière à respecter les probabilités relatives d'utilisation de chaque groupement de i unités fonctionnelles, afin de réaliser, comme à l'étage précédent, un branchement probabiliste.

Prenons un exemple afin d'illustrer notre propos. Supposons que le degré de parallélisme du temps de cycle courant est 2.

Parmi les six choix possibles de répartition des deux jetons présents dans la place *Paral2* sur les quatre unités fonctionnelles, le choix de la paire {additionneur, multiplieur} est représenté par l'arc se dirigeant vers la transition *taddmul*.

Si l'on considère maintenant les probabilités d'utilisation simultanée de deux unités fonctionnelles suivantes:

- additionneur et multiplieur: 50%,
- additionneur et mémoire principale: 30%,
- multiplieur et table des constantes: 20%,

les taux de tir des 6 transitions dont la place antécédente est *Paral2* seront:

- *taddmul*: 0,5,
- *taddmm*: 0,3,
- *tmultm*: 0,2,
- *taddtm*: 0,
- *tmulmm*: 0,
- *tmmtm*: 0.

Ajoutons enfin que les taux de tir des transitions de sortie pour chacune des unités fonctionnelles sont calculés de manière à respecter les probabilités de branchement des résultats de cette unité fonctionnelle vers les opérandes possibles de l'ensemble des unités fonctionnelles et à réaliser ainsi un branchement probabiliste.

5.3.2 Modélisation Simplifiée par Réseau de Files d'Attente

Afin de réduire le temps de la résolution du modèle de chaque FPS264 considéré comme un agrégat pour le système ICAP, un modèle simplifié du calculateur est souhaitable. C'est pourquoi nous avons dérivé du réseau de Petri stochastique présenté ci-dessus un modèle sous forme de réseau de files d'attente très simple.

Modèle de l'Architecture [A]

Le modèle de l'architecture (cf figure 5.7) est un réseau de files d'attente ouvert, composé de quatre stations de service exponentiel représentant les quatre unités fonctionnelles modélisées précédemment, l'additionneur, le multiplieur, la mémoire principale et la table des constantes. Chacune de ces stations a un nombre de serveurs égal au nombre d'étages que comporte son pipeline, et son temps de service moyen est égal au temps de séjour moyen d'un jeton dans les places du réseau de Petri stochastique modélisant l'unité fonctionnelle correspondante.

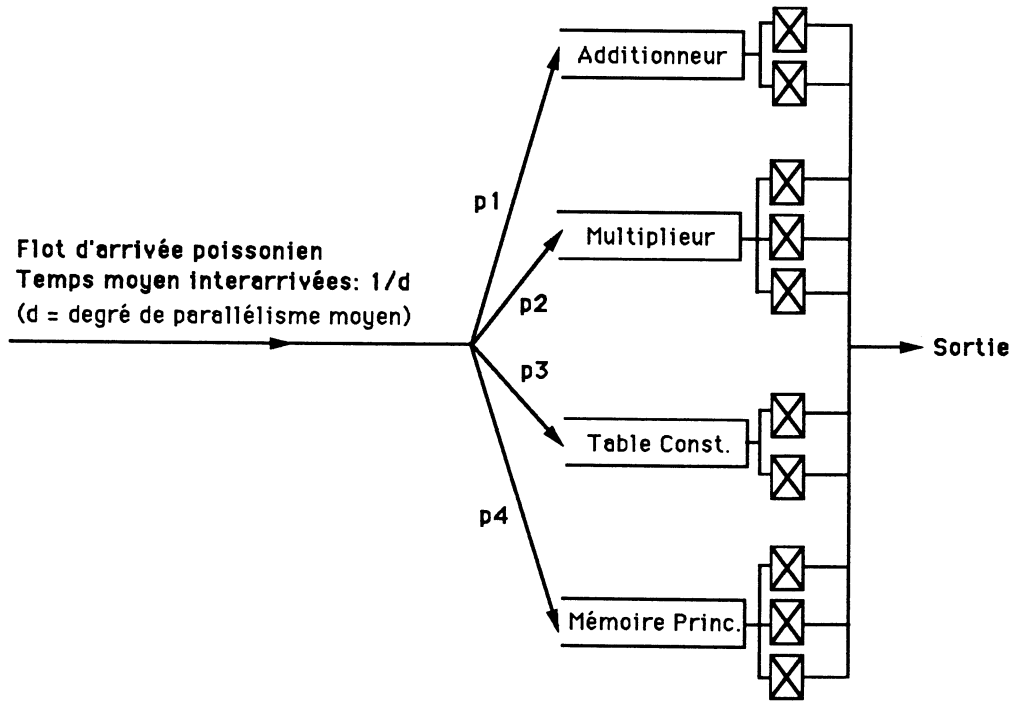

Figure 5.7: Modèle Simplifié du FPS264

Modèle des Programmes [B]

Un client circulant dans le réseau de files d'attente représente une opération effectuée sur le calculateur.

On considère que l'unité de temps est le temps de cycle de l'horloge du calculateur. Si d représente le degré de parallélisme moyen du programme, soit le nombre moyen d'opérations initialisées à chaque temps de cycle, le flot des arrivées de clients est donc caractérisé par un temps moyen des interarrivées de $1/d$.

Les probabilités de branchement p_i vers les différentes stations sont obtenues à partir de la répartition des opérations du programme sur les quatre unités fonctionnelles (cf paragraphe 5.4.1).

Ajoutons pour finir qu'il peut sembler que la dernière des caractéristiques des programmes (telles qu'elles sont définies dans le modèle des programmes du réseau de Petri stochastique), à savoir la répartition de l'origine des opérandes des unités fonctionnelles, est ignorée dans ce modèle. En fait, cette caractéristique est prise en compte dans le temps de service exponentiel des stations, dans la mesure où elle constitue un paramètre d'entrée du réseau de Petri stochastique dont la résolution permet la détermination de ce temps de service.

5.4 Estimation et Mesure des Paramètres d'Entrée

De la même manière, nous considérons successivement le modèle détaillé du calculateur sous forme de réseau de Petri stochastique et celui simplifié du calculateur sous forme de réseau de files d'attente.

5.4.1 Modèle Détaillé du Calculateur

Pour le modèle de l'architecture, le seul paramètre d'entrée que le modèle requiert est la **durée du temps de cycle** de l'horloge du calculateur, donnée fournie par Floating Point Systems (53 ns).

Pour le modèle des programmes, les paramètres d'entrée sont les **caractéristiques du programme étudié**, telles que définies précédemment, à savoir la probabilité de chaque degré de parallélisme, la répartition des opérations sur les quatre unités fonctionnelles du modèle et la répartition de l'origine des opérandes de chaque unité fonctionnelle.

Afin d'automatiser la détermination de ces caractéristiques, un **analyseur lexical** a été écrit à l'aide de l'utilitaire UNIX, LEX.

On définit le **poids d'une ligne assembleur APAL64** (constituant de base d'un programme) comme le nombre de fois que cette ligne sera exécutée compte tenu du niveau d'imbrication de la boucle dans laquelle elle est contenue (niveau 0 si elle n'est pas contenue dans une boucle).

Il est clair que le poids d'une ligne assembleur dépend des nombres de fois respectifs que les boucles imbriquées la contenant seront exécutées, ces nombres étant bien souvent fonction des variables du programme. L'analyse lexicale ne permet qu'une étude statique du programme, et en aucun cas, ne peut prendre en compte les valeurs dynamiques des variables du programme. Par conséquent, l'analyseur lexical n'est pas à même de calculer le poids réel de chaque ligne assembleur.

Cependant, dans le cas qui nous intéresse ici, on se limite aux procédures de la bibliothèque mathématique APMATH64, et les principales variables du programme sont les paramètres de la procédure. On observe que, le plus souvent, les indices de boucle sont des fonctions très simples de la taille des vecteurs ou des matrices passés en paramètre et que les nombres de fois respectifs que les boucles de la procédure sont exécutées sont du même ordre de grandeur. On a considéré qu'ils étaient rigoureusement identiques et défini le poids de chaque ligne assembleur comme le niveau d'imbrication de la boucle qui la contient. C'est bien sûr une faiblesse de notre analyseur de programmes, qui ne porte ici pas trop à conséquence, compte tenu de la structure, somme toute fort simple, des programmes étudiés.

L'analyseur lexical opère en deux passes:

1. la première passe détermine, à partir de la structure des boucles du programme (représentée sous la forme d'un arbre) le poids associé à chaque ligne assembleur,

2. la deuxième passe détermine les caractéristiques du programme, en tenant compte des poids respectifs des lignes assembleur.

Détaillons maintenant un peu plus le mode de calcul de chaque caractéristique.

Probabilité de chaque Degré de parallélisme

Dans chaque ligne assembleur, on repère le nombre i ($1 \leq i \leq 4$) d'unités fonctionnelles utilisées. On incrémente, du poids de la ligne assembleur analysée, la variable n_i qui comptabilise le nombre de fois que i unités fonctionnelles sont utilisées simultanément. La probabilité q_i, que le degré de parallélisme du programme soit égal à i, est déterminée par l'expression:

$$q_i = \frac{n_i}{n_1 + n_2 + n_3 + n_4}.$$

Répartition des Opérations sur les Unités Fonctionnelles

Lors de l'analyse de chaque ligne assembleur, on incrémente, du poids de cette ligne, la variable a_i associée à chaque unité fonctionnelle comptabilisant le nombre d'accès à cette unité fonctionnelle. La probabilité p_i d'accès à l'unité fonctionnelle i est alors donnée par l'expression:

$$p_i = \frac{a_i}{a_1 + a_2 + a_3 + a_4}.$$

Répartition de l'Origine des Opérandes de Chaque Unité Fonctionnelle

Pour chaque accès à l'unité fonctionnelle i, on repère le nombre et l'origine j des opérandes. Pour chaque opérande, on incrémente alors, du poids de la ligne assembleur contenant cet accès, la variable $o_{i,j}$ comptabilisant, pour l'unité fonctionnelle i, le nombre de fois où l'un des opérandes est issu de l'unité fonctionnelle j. On en déduit d'une part, les probabilités relatives $r_{i,j}$ des origines j possibles des opérandes de l'unité fonctionnelle i, par la formule:

$$r_{i,j} = \frac{o_{i,j}}{\sum_{j=1}^{4} o_{i,j}},$$

et d'autre part, les probabilités $s_{j,i}$ des destinations i possibles des résultats obtenus à la sortie de l'unité fonctionnelle j, par la formule:

$$s_{j,i} = \frac{o_{i,j}}{\sum_{i=1}^{4} o_{i,j}}.$$

5.4.2 Modèle Simplifié du Calculateur

Pour le modèle de l'architecture, le temps de cycle de l'horloge du calculateur est également requis (53 ns). Le temps de service moyen des stations correspondant à chaque unité fonctionnelle est, quant à lui, obtenu à l'issue de la résolution du modèle sous forme de réseau de Petri stochastique. Il est égal à la somme des temps moyens de séjour d'un jeton dans les places du réseau constituant l'itinéraire obligatoire du jeton lorsqu'il visite l'unité fonctionnelle.

Pour le modèle des programmes, les paramètres d'entrée sont les probabilités p_i calculées pour le modèle détaillé du calculateur d'une part, et le degré de parallélisme moyen du programme étudié d'autre part, donné par la formule:

$$d = q_1 + 2q_2 + 3q_3 + 4q_4.$$

5.5 Résolution du Modèle

Comme au paragraphe précédent, nous considérons successivement le modèle détaillé sous forme de réseau de Petri stochastique et le modèle simplifié sous forme de réseau de files d'attente.

5.5.1 Modèle Détaillé du Calculateur

Au vu de la description du modèle détaillé, on peut présager du nombre gigantesque d'états composant le graphe des marquages du réseau de Petri stochastique.

Si P représente le vecteur des probabilités d'états (P_i représente la probabilité du marquage i), résoudre le réseau de Petri stochastique consiste à trouver la solution non identiquement nulle du système linéaire $AP = P$, où A est la matrice de transition de la chaîne de Markov associée au réseau de Petri stochastique.

Afin de permettre la résolution de ce système par une méthode itérative de type Gauss-Seidel, avec une précision suffisante, il est nécessaire de réduire le nombre d'états du réseau.

Pour ce faire, nous avons utilisé deux méthodes originales, l'une consistant à **modifier le logiciel RDPS [K]** pour une représentation efficace du mode de fonctionnement synchrone des

unités fonctionnelles du calculateur, l'autre implémentant une **méthode itérative d'agrégation-désagrégation [J]**.

De plus, afin d'obtenir des modèles simplifiés pouvant convenir à une classe de programmes, plutôt qu'à un programme bien spécifique, limitant ainsi le nombre de résolutions nécessaires du modèle détaillé, nous avons défini et caractérisé des classes de programmes grâce à une analyse en composantes principales [C] effectuée sur un large échantillon de procédures de la bibliothèque APMATH64.

Les deux méthodes de réduction du nombre d'états du modèle détaillé ainsi que la classification des programmes font l'objet des trois paragraphes qui suivent.

Modification du Logiciel RDPS [K]

Afin de réduire le nombre d'états du réseau de Petri stochastique et aussi de modéliser le fonctionnement synchrone du calculateur, nous avons défini deux types de transitions:

1. les **transitions "synchronisées par l'horloge"**, dont le taux de tir est représenté par la lettre H sur les modèles, et dont la règle de mise à feu est la suivante: toute transition synchronisée par l'horloge tirable est tirée au bout d'une durée constante égale à un temps de cycle d'horloge,
2. les transitions standards, dont la mise à feu est tout à fait classique.

Le mode de génération du graphe des marquages du logiciel RDPS a été modifié afin de permettre le tir simultané de toutes les transitions synchronisées par l'horloge tirables et d'empêcher la génération de tous les états intermédiaires obtenus par le tir séquentiel de chacune de ces transitions.

Illustrons par un exemple la réduction du nombre d'états ainsi réalisée: un réseau de Petri très simple comportant 7 places, 7 transitions et 17 arcs a conduit à la génération de 182 états sans transition synchronisée par l'horloge et seulement à la génération de 35 états lorsque deux transitions ont été synchronisées par l'horloge, soit une réduction de l'ordre de 80%.

Bien sûr, cette modification des règles de mise à feu ne doit pas s'opérer sans une vérification de la persistance des transitions standard vis-à-vis des transitions synchronisées par l'horloge. En d'autres termes, on doit s'assurer que la séquence de tir des transitions synchronisées par l'horloge ne

modifie pas la "tirabilité" des transitions standard, entraînant alors une suppression illicite d'états. Cette vérification, fort coûteuse en temps, n'a cependant été réalisée que pour deux séquences de tir possibles des transitions synchronisées par l'horloge.

Méthode Itérative d'Agrégation-Désagrégation [J]

Remarquons tout d'abord que les quatre unités fonctionnelles modélisées (l'additionneur, le multiplieur, la mémoire principale et la table des constantes) ne sont pas fortement corrélées. En effet, leurs interactions ont lieu en fin de calcul, lorsque le résultat obtenu sur une unité fonctionnelle devient l'opérande d'une autre unité fonctionnelle.

Afin d'accélérer la résolution du modèle, nous avons élaboré la méthode itérative d'agrégation-désagrégation représentée par la figure 5.8.

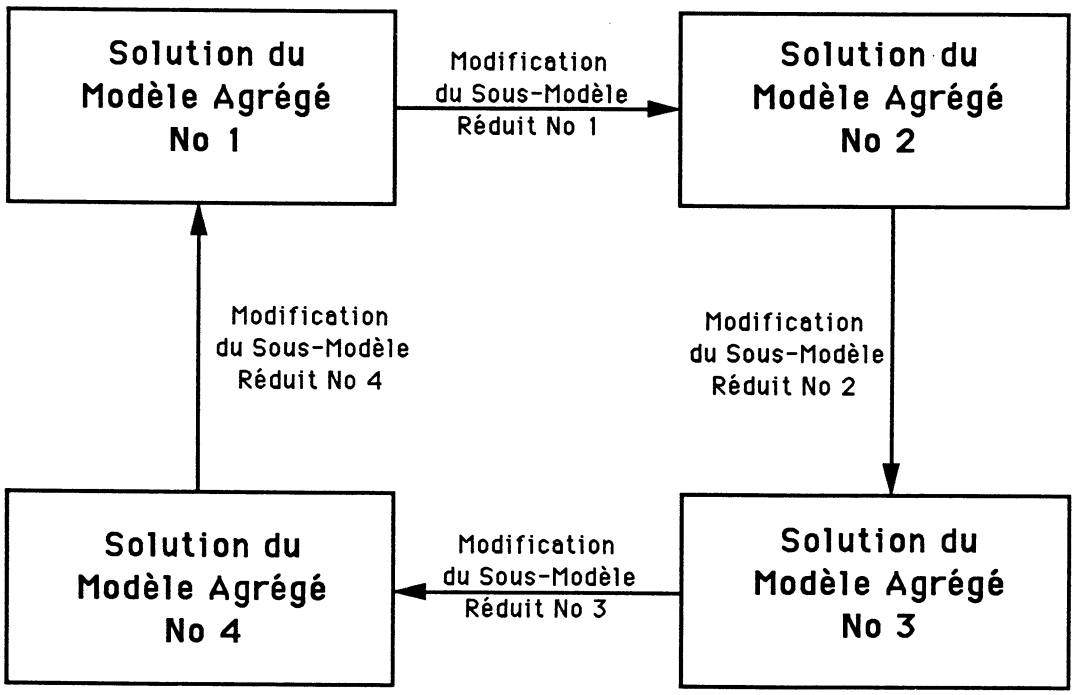

Figure 5.8: Méthode Itérative d'Agrégation-Désagréagtion

Quatre modèles "agrégés" sont considérés (cf figure 5.9). Le modèle numéro un (respectivement 2, 3 et 4) correspond au modèle global, dans lequel chacun des sous-modèles des trois unités fonctionnelles, autres que l'additionneur (respectivement le multiplieur, la mémoire principale et la table des constantes), est remplacé par une place unique, appelée place-agrégat. Le taux de tir de la transition de sortie de la place-agrégat est égal à l'inverse du temps de séjour moyen d'un jeton dans l'unité fonctionnelle associée, ce temps de séjour ayant été calculé à l'itération précédente à partir d'un des autres modèles agrégés.

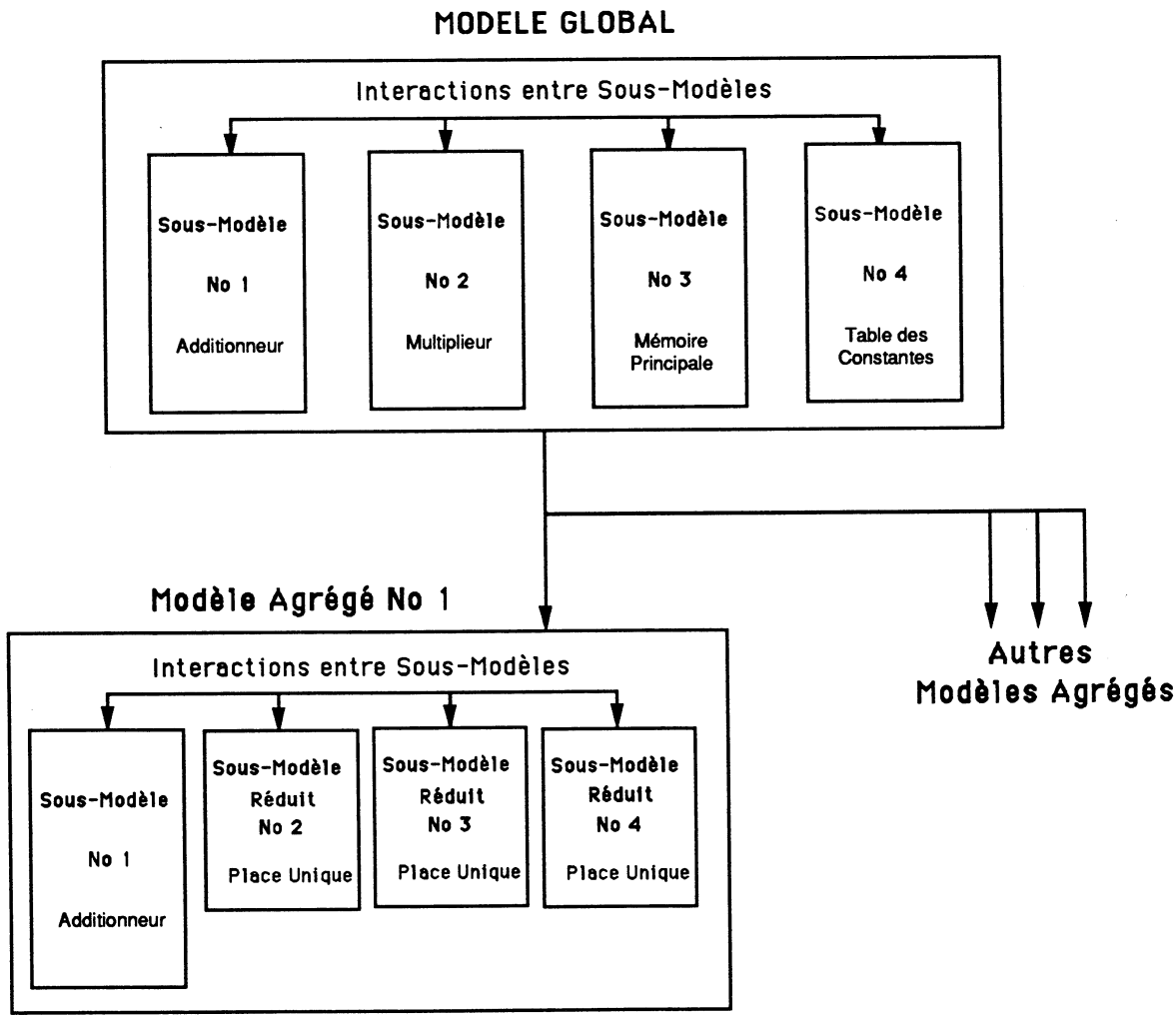

Figure 5.9: Modèle Global et Modèles Agrégés

Dans chaque modèle agrégé, le modèle de contrôle de même que les interactions entre les quatre sous-modèles ne sont pas modifiés, ou très légèrement afin de permettre leur interconnexion.

Les quatre modèles agrégés sont résolus itérativement, chaque itération se déroulant en quatre étapes (cf figure 5.8).

L'étape numéro i résout le modèle agrégé numéro i et fournit le taux de sortie moyen de la place-agrégat qui représentera, aux trois étapes suivantes, la seule unité fonctionnelle non agrégée à cette étape.

Pour la première itération, le taux de sortie de chaque place-agrégat est initialisé avec l'inverse du nombre d'étages composant le pipeline de l'unité fonctionnelle correspondante.

Le processus itératif converge lorsque la différence relative des taux de sortie de chaque place-agrégat obtenus pour deux itérations consécutives, est inférieure à un seuil fixé (proche de 0).

La vitesse de convergence de cette méthode approximative de résolution du modèle global du calculateur est analysée au paragraphe 5.6.

Classification des programmes [C]

Afin d'obtenir des modèles simplifiés du calculateur (sous forme de réseau de files d'attente), qui puissent modéliser l'exécution d'une classe de programmes et limiter ainsi le nombre de résolutions nécessaires du modèle détaillé, une analyse en composantes principales (ACP) a été effectuée sur un échantillon de 36 procédures de la bibliothèque APMATH64, à l'aide du logiciel d'analyse de données ITCF [Fouca84].

Les procédures analysées réalisent des calculs matriciels (multiplication de matrices, addition de vecteurs, ...), mais aussi des traitements plus complexes (résolution de systèmes linéaires, résolution d'équations différentielles, transformée de Fourier, ...). Elles sont écrites en assembleur APAL64 de manière à exploiter au mieux les ressources du calculateur.

Choix des Variables pour l'ACP

Les variables considérées pour l'ACP sont toutes obtenues à partir de l'analyse lexicale des procédures:

- tout d'abord les probabilités q_i $(1 \leq i \leq 4)$ de chaque degré de parallélisme,

- puis des variables représentant les probabilités respectives d'utilisation des 4 unités fonctionnelles, conditionnées par le degré de parallélisme:
  - lorsque le degré de parallélisme est 1,
    1. A représente la probabilité d'utiliser l'additionneur,
    2. M représente la probabilité d'utiliser le multiplieur,
    3. S représente la probabilité d'utiliser la mémoire principale,
    4. et T représente la probabilité d'utiliser la table des constantes,
  - lorsque le degré de parallélisme est 2,
    1. AM représente la probabilité d'utiliser simultanément l'additionneur et le multiplieur,
    2. AS représente la probabilité d'utiliser simultanément l'additionneur et la mémoire principale,
    3. AT représente la probabilité d'utiliser simultanément l'additionneur et la table des constantes,
    4. MS représente la probabilité d'utiliser simultanément le multiplieur et la mémoire principale,
    5. MT représente la probabilité d'utiliser simultanément le multiplieur et la table des constantes,
    6. et ST représente la probabilité d'utiliser simultanément la mémoire principale et la table des constantes,
  - lorsque le degré de parallélisme est 3,
    1. AMS représente la probabilité d'utiliser simultanément l'additionneur, le multiplieur et la mémoire principale,
    2. AMT représente la probabilité d'utiliser simultanément l'additionneur, le multiplieur et la table des constantes,
    3. AST représente la probabilité d'utiliser simultanément l'additionneur, la mémoire principale et la table des constantes,
    4. et MST représente la probabilité d'utiliser simultanément le multiplieur, la mémoire principale et la table des constantes.

Ainsi, chaque procédure de la bibliothèque APMATH64 est représentée par un point de l'espace $\Re^{18}$ et le but de l'ACP est de projeter cet espace sur des plans afin de mettre en évidence des groupements de points qui constitueront les classes de programmes.

Résultats de l'ACP

Voici l'ensemble des résultats de l'ACP réalisée sur les 36 procédures de la bibliothèque APMATH64.

Degré de Parallélisme

Les probabilités obtenues pour chaque degré de parallélisme sont:

$$q_1 = 0,9\ , \quad q_2 = 0,09\ , \quad q_3 = 0,01\ , \quad q_4 = 0.$$

Le fait que le degré de parallélisme maximal de 4 ait une probabilité nulle a permis une réduction du nombre d'états du modèle de contrôle, grâce à une nouvelle modification des règles de mise à feu des transitions du logiciel RDPS, consistant à ignorer les transitions à taux de tir nul.

Utilisation des Unités Fonctionnelles

Lorsque le degré de parallélisme est 1, il apparaît que les deux unités les plus fréquemment employées sont l'additionneur et la mémoire principale. On remarque aussi que la mémoire principale est utilisée environ 6 fois plus souvent que la table des constantes.

Lorsque le degré de parallélisme est 2, les valeurs des variables non nulles sont:

$$AM = 0,6\ , \quad AS = 0,26\ , \quad MS = 0,14.$$

Lorsque le degré de parallélisme est 3, la seule variable non nulle est *AMS*.

Comme précédemment, les variables nulles ont permis une réduction en conséquence du nombre d'états du modèle détaillé, par la non mise à feu des transitions à taux de tir nul correspondantes.

Inertie des Axes Principaux

L'inertie des trois premiers axes principaux est la suivante:

1. premier axe principal: 35,5%,
2. deuxième axe: 22,5%,
3. troisième axe: 19,4%.

Avec une inertie de l'ordre de 60% pour les deux premiers axes principaux et de plus de 75% pour les trois premiers axes, les résultats de l'ACP sont bons, compte tenu du nombre élevé de variables utilisées.

Interprétation des Résultats de l'ACP

Au vu des résultats de l'ACP, il apparaît que les ressources les plus critiques, pour l'ensemble des 36 procédures analysées, sont l'additionneur et la mémoire principale, qui sont le plus souvent employées. Cependant, elles sont relativement peu utilisées simultanément ($AS = 0{,}26$).

L'interprétation des trois premiers axes principaux est la suivante:

1. le premier axe représente l'utilisation simultanée de l'additionneur et du multiplieur,
2. le deuxième axe représente le degré de parallélisme,
3. et le troisième axe représente l'utilisation simultanée de l'additionneur et de la mémoire principale.

A partir de ces constatations et des projections sur les trois premiers plans principaux, nous avons pu dégager 4 classes principales de programmes:

1. la classe 1 correspond aux programmes qui ont un faible degré de parallélisme moyen,
2. la classe 2 regroupe les programmes qui utilisent surtout l'additionneur et le multiplieur simultanément,

3. la classe 3 est constituée par les programmes utilisant surtout l'additionneur et la mémoire principale simultanément,
4. et la classe 4 se compose des programmes au plus fort degré de parallélisme moyen.

Afin de limiter le nombre de résolutions du modèle détaillé, une procédure représentative de chaque classe a été choisie:

1. pour la première classe, la procédure *ICAMAX* qui recherche l'indice de l'élément maximal d'un vecteur,
2. pour la deuxième classe, la procédure *SASUM* qui calcule la norme d'un vecteur,
3. pour la troisième classe, la procédure *MMTMUL* qui multiplie deux vecteurs,
4. et pour la dernière classe, la procédure *FUN3* qui réalise l'interpolation d'une fonction par un polynôme d'ordre 3.

Du fait que les variables de l'ACP sont obtenues à partir des caractéristiques des programmes, telles qu'elles ont été définies au paragraphe 5.3.1, on est assuré que les éléments d'une même classe ont des caractéristiques voisines. Par conséquent, le cheminement des jetons à l'intérieur du réseau de Petri stochastique modélisant l'exécution de ces éléments dans le calculateur variera peu d'un élément à l'autre. Cette hypothèse de similarité de comportement est d'ailleurs vérifiée lors de la validation (cf paragraphe 5.6).

5.5.2 Modèle Simplifié du Calculateur

Rappelons tout d'abord la nécessité de la résolution préalable du modèle détaillé pour un représentant de chaque classe de programmes afin d'obtenir les temps de service moyens des stations modélisant chaque unité fonctionnelle.

Le modèle simplifié du calculateur a été décrit à l'aide du logiciel QNAP ("Queueing Network Analysis Package" [Véran84]). Pour chaque classe de programmes, on a simplifié le modèle au maximum, éliminant les stations des unités fonctionnelles très peu utilisées (probabilités de branchement proches de 0), voire inutilisées, et on a effectué une simulation.

L'ensemble des résultats obtenus est présenté et commenté dans le paragraphe suivant.

5.6 Validation

La validation de cette modélisation hybride est basée sur une campagne de mesures réalisée à partir des 36 procédures de la bibliothèque APMATH64 utilisées pour la constitution des classes de programmes.

Comme nous l'avons décrit au paragraphe 5.5.1, quatre classes de programmes ont été exhibées et pour chaque classe, un représentant a été déterminé.

Pour des contraintes de temps de calcul et donc de coût, nous n'avons pu résoudre de manière exacte le modèle détaillé que pour le représentant de la première classe. Pour le représentant de la deuxième classe, seule la méthode itérative d'agrégation-désagrégation a pu permettre sa résolution, grâce à la réduction du nombre d'états du modèle qu'elle engendre. Enfin, pour les deux classes restantes, la résolution du modèle global n'a pas semblé raisonnable d'un point de vue financier. A titre d'exemple, le temps de résolution d'un modèle relativement simple comportant 1800 états est d'environ 25 mn CPU sur un VAX 785. Actuellement, le logiciel RDPS [Flori85] permet la résolution de modèles comportant jusqu'à 10000 états. Une extension à 100000 états est en cours de développement.

Pour les représentants des deux premières classes, nous avons résolu par simulation, à l'aide du logiciel QNAP [Véran84], le modèle simplifié du calculateur, dont les paramètres d'entrée ont été fournis par la résolution précédente du modèle détaillé.

La validation elle-même s'est opérée en plusieurs étapes.

Tout d'abord, nous avons effectué quelques vérifications préliminaires.

Nous avons dans un premier temps isolé le **sous-modèle de contrôle** du modèle détaillé et nous avons vérifié que le degré de parallélisme moyen modélisé [**E**] était bien identique au degré de parallélisme moyen déterminé à partir de l'analyse lexicale des programmes tests. De même, nous avons comparé les répartitions des opérations sur les unités fonctionnelles réalisées par la modélisation avec celles issues de l'analyse lexicale des programmes tests et nous avons obtenu une erreur relative de l'ordre de 8%. Compte tenu de cette erreur tout à fait acceptable, nous avons considéré que le sous-modèle de contrôle était valide.

Dans un deuxième temps, nous avons étudié l'influence de la modification des règles de mise à feu des transitions dans le logiciel RDPS sur la précision de la résolution du modèle. La différence relative entre la précision obtenue à partir du réseau complet et celle obtenue à partir du réseau

réduit fut toujours comprise entre 0,01% et 1%, validant ainsi les nouvelles règles de mise à feu.

Ensuite, nous nous sommes attachés à vérifier l'hypothèse selon laquelle les programmes appartenant à la même classe ont un comportement similaire sur le modèle détaillé du calculateur. Pour ce faire, nous avons considéré deux procédures appartenant à la première classe et avons résolu le modèle détaillé pour chacune d'entre elles. La différence relative entre les temps moyens de séjour d'un jeton dans les sous-modèles des différentes unités fonctionnelles a été inférieure à 18% avec une moyenne de 6,3% sur l'ensemble des unités fonctionnelles.

Il semble donc valide de pouvoir remplacer un programme par un autre programme de la même classe sans entraîner une modification trop importante des temps de séjour moyens d'un jeton dans les sous-modèles des unités fonctionnelles.

Rappelons que ces temps moyens de séjour sont les résultats de la résolution du modèle détaillé qui seront utilisés comme paramètres d'entrée du modèle simplifié pour la classe de programmes étudiée. Plus précisément, ces résultats déterminent les temps moyens de service des stations du modèle simplifié modélisant les différentes unités fonctionnelles du calculateur.

Nous avons également étudié la validité de la méthode de modélisation hybride employée, en comparant le temps de réponse du calculateur pour exécuter la procédure *ICAMAX* mesuré sur le FPS264 du Centre Interuniversitaire de Calcul de Grenoble et le temps de réponse simulé par le modèle simplifié du calculateur pour cette même procédure. L'erreur relative obtenue a été de l'ordre de 10%, ce qui est tout à fait raisonnable compte tenu de la complexité du modèle. Afin d'estimer également la validité de la méthode itérative d'agrégation-désagrégation utilisée, nous avons procédé de même avec le représentant de la deuxième classe, *SASUM*, dont la résolution du modèle détaillé a nécessité le recours à cette méthode d'agrégation. L'erreur relative a été de l'ordre de 14%, soit légèrement supérieure au cas précédent, mais elle reste cependant très acceptable et semble valider la méthode d'agrégation employée.

Le dernier point de notre expérimentation a été d'étudier la vitesse de convergence de la méthode itérative d'agrégation-désagrégation utilisée, pour évaluer ainsi son efficacité.

Lors de la résolution du modèle détaillé pour la procédure *SASUM*, nous avons constaté une convergence à 10^{-3} en quatre itérations. Chaque itération ne comportait que deux étapes, car seuls l'additionneur et la mémoire principale étaient corrélés et ont donc été modélisés dans la phase d'agrégation-désagrégation. Le sous-modèle du multiplieur, quant à lui, a été résolu séparément.

Enfin, pour donner un ordre de grandeur du gain de temps, procuré par l'application de la méthode d'agrégation, nous avons mesuré le temps de résolution du modèle détaillé et avons obtenu 18 mn CPU. Compte tenu des 10000 états et 37000 arcs environ qui constituent ce modèle, on peut estimer à quatre heures CPU le temps qui serait nécessaire à sa résolution directe. On se rend compte, à partir de cet exemple, de tout le bénéfice apporté par cette méthode d'agrégation.

5.7 Conclusion

La modélisation hybride du calculateur scientifique, présentée dans ce chapitre, a permis d'illustrer trois points importants de la méthodologie de modélisation:

1. l'intérêt de la distinction entre le modèle de l'architecture [**A**] et le modèle des programmes [**B**], qui permet une décomposition du problème complexe de la modélisation d'un tel système et une meilleure caractérisation des paramètres d'entrée nécessaires à la résolution du modèle,

2. l'efficacité des méthodes itératives d'agrégation-déagrégation [**J**] pour accélérer, voire simplement rendre possible, la résolution du modèle, par une réduction conséquente de l'espace des états du modèle; ces méthodes d'agrégation sont souvent caractérisées par une grande vitesse de convergence, même si la preuve de cette convergence est empirique dans de nombreux cas,

3. l'importance de la constitution de classes de programmes, par l'application de méthodes empruntées à l'analyse des données [**C**], qui permet de caractériser le comportement d'un élément quelconque de la classe par l'étude approfondie d'un seul représentant de cette classe.

Certes, la phase de validation, qui présente bien tous les aspects de la modélisation méritant d'être validés, peut sembler superficielle, par le petit nombre de procédures qui ont donné lieu à une résolution du modèle détaillé. Cela est indéniable, mais la seule raison en est un besoin en temps de calcul prohibitif, qui a malheureusement limité le nombre de tests effectués. Nous sommes bien sûr conscients de cet état de fait et insistons plus sur **l'intérêt méthodologique** de la modélisation que sur l'intérêt pratique du modèle obtenu.

Nous montrerons néanmoins, au chapitre suivant, comment le modèle simplifié du calculateur pourrait être utilisé en tant que sous-modèle de chaque FPS264, considéré comme un agrégat, pour le modèle global du système ICAP.

CHAPITRE 6

Loosely Coupled Array of Processors FPS264 (lCAP/FPS264)

Ce chapitre présente une modélisation sous forme de réseau de files d'attente du système lCAP/FPS264. Les points importants de cette modélisation sont:

1. une **distinction très nette entre le modèle de l'architecture [A] et le modèle des programmes [B]**, qui a permis deux approches différentes pour le modèle des programmes, sans remettre en question le modèle de l'architecture:
   - tout d'abord, une **modélisation déterministe des programmes [Q]** par trace d'exécution a été employée pour la validation du modèle, permettant une représentation très fidèle des programmes d'application utilisés lors des mesures,
   - puis, une **modélisation probabiliste des programmes [Q]** a été mise à profit pour la phase de prédiction de performances, permettant une étude beaucoup plus générale du comportement du système, soumis à une charge moyenne,
2. une **analyse de prédiction de performances** très détaillée, mettant en évidence les avantages d'une évaluation de performances devant de simples mesures, par la possibilité d'envisager des extensions ou des modifications système, d'étudier de nombreuses alternatives d'implémentations possibles pour un programme, sans aucune intervention sur le système réel, et de produire de nombreux critères de performances, difficiles (voire impossibles) à obtenir à partir de mesures,
3. et une illustration, lors de la phase de prédiction de performances, de la **technique de génération aléatoire d'événements corrélés [R]** (en l'occurrence émissions et réceptions de messages), garantissant l'absence d'étreinte fatale entre processus parallèles.

Bien sûr, les autres aspects de la méthodologie sont également illustrés dans cette modélisation.

Même si aucune méthode d'agrégation n'est employée ici, nous rappelons que chaque FPS264 présent sur le système pourrait être considéré comme un agrégat du modèle global, et l'utilisation du modèle présenté au chapitre précédent, moyennant quelques aménagements quant au modèle des programmes (cf paragraphe 6.5.2), pourrait alors permettre la résolution du sous-modèle représentant chaque agrégat [J].

Nous montrerons que la modélisation actuelle ne remet pas en question cette possibilité. Seulement, les aménagements nécessaires n'ont pu être réalisés pour des contraintes de temps. En outre, la résolution de chaque agrégat pourrait s'avérer fort coûteuse, compte tenu de la différence notable qui peut exister entre une procédure de la bibliothèque mathématique APMATH64 et un programme d'application utilisé en production. Enfin, l'objectif principal du modèle du système ICAP est plus l'étude de l'overhead induit par les communications et synchronisations entre processus parallèles qu'une caractérisation très fine de chacun des processus.

Conformément à la méthodologie proposée au chapitre 2, nous détaillons maintenant chacune des sept phases qui ont composé cette modélisation.

6.1 Formulation des Objectifs

Le système parallèle ICAP ("loosely Coupled Array of Processors", que l'on peut traduire par réseau de processeurs faiblement couplés) a été conçu à IBM Kingston pour permettre la résolution, en parallèle sur plusieurs processeurs, de problèmes nécessitant de très gros calculs intensifs, dans des domaines d'application aussi variés que la chimie théorique, la mécanique quantique, la mécanique statistique ou la dynamique des fluides. Le système est prévu pour satisfaire les besoins en calcul très importants de l'approche de la "simulation globale", consistant à simuler de manière réaliste des problèmes complexes [Cleme86].

Depuis 1983, naissance de ICAP, l'environnement du système a été en constante évolution, aussi bien sur le plan matériel que sur le plan logiciel, afin d'améliorer ses performances et sa facilité d'utilisation [Detri88,Cleme88].

Pour implémenter un programme d'application sur le système, l'utilisateur part du code source séquentiel FORTRAN, qu'il subdivise en portions de code qui resteront séquentielles et en portions de code qui pourront s'exécuter en parallèle sur plusieurs processeurs. Il définit ainsi un certain nombre de tâches de calcul, qui devront pouvoir communiquer et se synchroniser entre elles. Pour ce faire, l'utilisateur dispose de directives spécifiques, qu'il insère dans le code source séquentiel FORTRAN. Un précompilateur a été développé afin de traduire ces directives en appels de routines FORTRAN ou assembleur [Chin85,Calta87].

Pour étudier les performances d'un tel système, plusieurs approches sont possibles .

La première solution est la réalisation de mesures sur le système. De nombreuses mesures de performance ont été effectuées sur le système ICAP (citons [Coron88]), afin de comparer les temps de réponse obtenus pour différentes stratégies d'implémentation d'un même programme. Mais, l'interprétation de ces mesures doit se limiter au programme et aux données utilisés et ne permet pas une prédiction du temps de réponse du même programme avec un nombre différent de processeurs parallèles ou d'autres données. Si l'on veut s'intéresser à un autre programme, de nouvelles mesures sont nécessaires en milieu dédié, empêchant tout calcul de la part des autres utilisateurs.

Une autre alternative est la caractérisation de chaque élément composant le temps de réponse du système pour exécuter un programme d'application. On pourra distinguer le temps de calcul propre, le temps de communication entre processus et le temps de synchronisation entre tâches parallèles, et déduire des formules plus générales donnant le temps de réponse pour une implémentation bien précise du programme d'application.

Ainsi que le souligne M. Becker dans [Becke87], cette approche a déjà été conduite pour des systèmes multiprocesseurs, semblables à ICAP. P. Leca [Leca87] s'est surtout intéressé à déterminer l'algorithme optimal pour résoudre un problème donné sur le système. R.W. Hockney [Hockn89], L. Brochard [Broch87], F. Faurie et moi-même [Broch88] avons défini des expressions générales pour l'accélération d'algorithmes spécifiques sur des systèmes multiprocesseurs.

Cependant, les prérequis communs à toutes ces études sont la nécessité que le système soit dédié à l'exécution du programme considéré et celle qu'il n'y ait aucun recouvrement entre les différents éléments constituant le temps de réponse du système pour l'exécution du programme.

Tant que le système opère en environnement dédié et que les processeurs parallèles sont à couplage lâche, les approches précédentes sont valides.

Or, le système ICAP a été conçu pour être utilisé dans un environnement de multiprogrammation et il évolue de plus en plus vers un couplage serré des processeurs parallèles (adjonction de mémoires partagées, d'un bus d'interconnexion). Par conséquent, les approches précédentes sont insuffisantes pour une bonne connaissance du comportement du système dans son environnement actuel.

Afin d'être plus proche de la réalité, et donc plus constructif, il est nécessaire de prendre en compte les problèmes de contention pour l'accès aux ressources. Une modélisation par réseau de files d'attente est une solution possible au problème de l'évaluation des performances du système ICAP.

L'objectif du modèle est tout d'abord de donner à l'analyste la possibilité de prédire le temps de réponse du système pour l'exécution de différents programmes d'application, et ceci sous diverses conditions de charge du système.

En outre, les critères de performances calculés à l'issue de la résolution du modèle apportent bien d'autres informations quant au comportement du système, telles que les taux d'occupation des ressources composant le système, les temps d'attente des processus, coopérant à l'exécution du programme, dûs à des contentions pour l'accès aux ressources partagées ou aux synchronisations entre tâches parallèles.

L'analyste peut également estimer l'impact d'extensions du système ou de modifications dans l'implémentation du programme d'application sur les performances de l'exécution du programme, par de multiples résolutions du modèle avec des paramètres d'entrée différents. Des questions plus générales encore peuvent être adressées, telles que la granularité optimale d'un programme d'application donné, et associé à cette granularité, le nombre optimal de processeurs parallèles à utiliser lors de l'exécution du programme.

La liste, proposée ici, des réponses que peut procurer l'utilisation du modèle, n'est bien sûr pas exhaustive. Nous avons simplement voulu énumérer certains points qui nous paraissaient dignes d'intérêt.

Nous ne résoudrons pas ici le problème plus général qui serait de déterminer quelles peuvent être les caractéristiques de charge du système ICAP susceptibles d'en optimiser son utilisation et quelles peuvent être les raisons de favoriser une application devant une autre, par le recours à une exécution parallèle plutôt que séquentielle. Cependant, le modèle, que nous présentons, constitue une première étape essentielle vers la possibilité future de répondre à de telles questions.

6.2 Analyse du Système

Dans ce paragraphe, nous présentons successivement l'architecture du système ICAP, la structure générale d'un programme d'application s'exécutant en parallèle sur ce système et les fonctionnalités du gestionnaire de ressources du système.

6.2.1 Description de l'architecture du système ICAP

Précisons tout d'abord que l'architecture décrite ici, et prise en compte par la modélisation, correspond à la configuration du système ICAP du premier semestre 1988, que nous appellerons ICAP/FPS264. Depuis le second semestre 1988, le système ICAP a évolué vers une configuration constituée de plusieurs ordinateurs multiprocesseurs IBM 3090, appelée ICAP/3090 et décrite en détail dans [Logan88].

Le système ICAP/FPS264 (appelé indifféremment ICAP dorénavant) est composé d'un **ordinateur hôte IBM 3081** (appelé simplement hôte), auquel sont attachés **10 calculateurs scientifiques FPS264** de Floating Point Systems (appelés APs - "Attached Processors") au moyen de canaux de 3 méga-octets par seconde de débit (appelés **canaux hôte-APs**).

Le système comporte également 5 mémoires de masse de Scientific Computing Associates (SCA), chacune d'entre elles étant accessible par 4 APs (on les appellera **mémoires de masse locales**), et une mémoire de masse de SCA accessible par les 10 APs (appelée **mémoire de masse globale**). Cependant, à un instant donné, un seul accès à une même mémoire de masse est possible. La capacité de stockage de chaque mémoire de masse locale est de 32 méga-octets et celle de la mémoire de masse globale de 512 méga-octets.

Les APs sont connectés à deux mémoires de masse locales chacun, suivant une structure en double anneau (cf figure 6.1). Chaque AP peut transférer des données depuis ou vers les mémoires de masse locales ou globale à une vitesse maximale de 38.5 méga-octets par seconde (1 mot tous les quatre temps de cycle).

Le système d'exploitation de l'hôte est "Virtual Machine/System Product" (VM/SP), système de multiprogrammation dans lequel chaque session utilisateur ou chaque programme s'exécutant en mode "batch" (tous deux appelés indistinctement "jobs") se voit attribué une **machine virtuelle**, représentation virtuelle de l'ordinateur qui semble dédié à l'exécution exclusive du job.

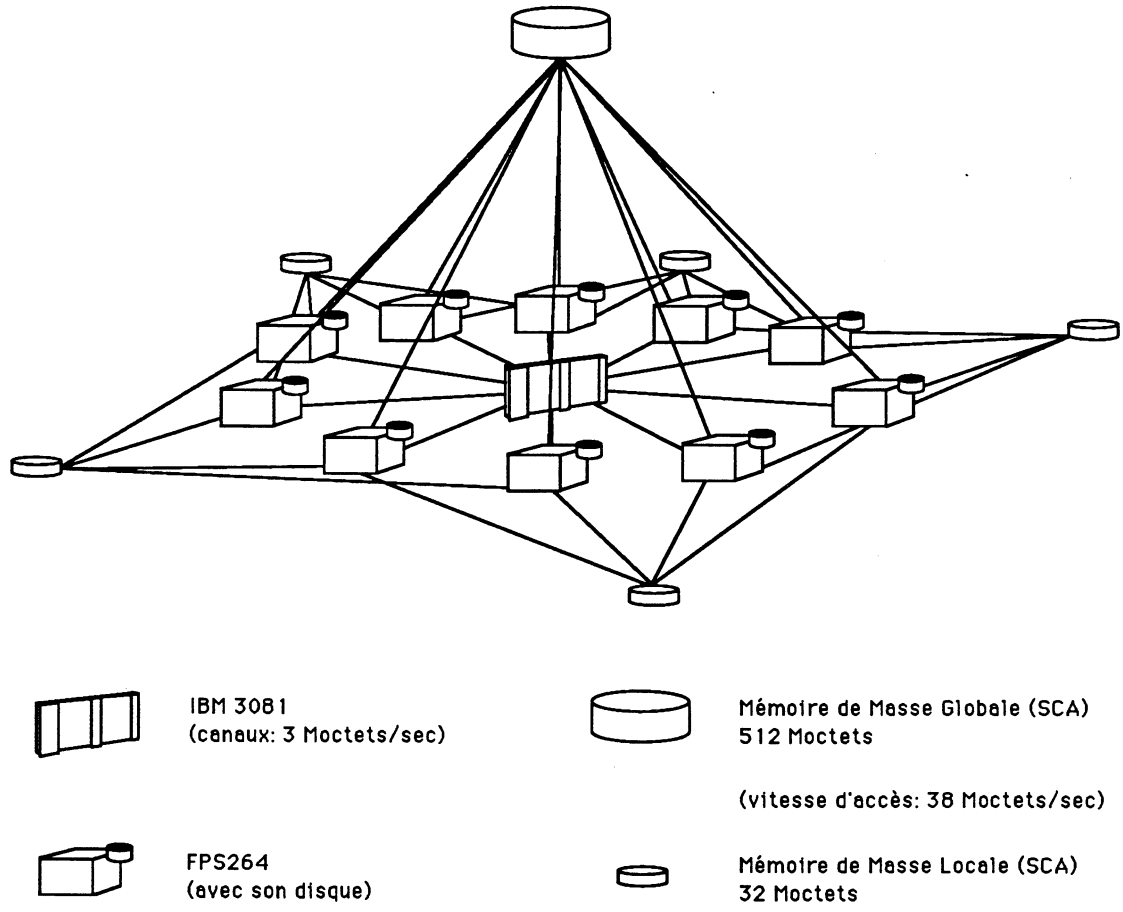

Figure 6.1: Configuration du Système ICAP/FPS264

Le système d'exploitation de chaque AP est un système mono-utilisateur, développé par FPS. Par un mécanisme appelé "**rollin/rollout**", le calculateur FPS264 peut interrompre, à la fin de chaque tranche de temps (fixée à une minute pour le système ICAP), l'exécution du job de l'utilisateur unique présent en mémoire, au bénéfice d'un autre job utilisateur. Dans ce cas, le premier job voit tout son environnement (état des registres, données et code en mémoire) sauvegardé sur le disque attaché au calculateur (phase de "rollout"), et le deuxième job voit à son tour son environnement restauré depuis le disque (phase de "rollin"). Cette opération de "rollin/rollout" est fort coûteuse et peut durer jusqu'à 11 secondes.

6.2.2 Structure Générale des Programmes d'Application Parallèles

Un programme d'application parallèle s'exécutant sur le système ICAP est composé d'un **programme maître** et d'un certain nombre de routines esclaves, appelées **AProutines**.

Le programme maître, écrit en FORTRAN77, constitue le programme principal de l'application et sera exécuté sur l'hôte, dans la machine virtuelle de l'utilisateur ayant lancé le programme, appelée **machine virtuelle maître** ("Master VM"). L'exécution du programme maître sera appelée **tâche maître**, ou plus simplement encore **maître**.

Les AProutines, écrites en APFTN64 (langage FORTRAN propre aux FPS264), seront appelées à partir du programme maître et pourront s'exécuter en parallèle sur plusieurs APs.

A l'initialisation du job, une machine virtuelle est créée sur l'hôte pour chaque AP que l'utilisateur désire attacher au job. Ces machines virtuelles, appelées **machines virtuelles esclaves** ("Slave VMs"), peuvent communiquer avec la machine virtuelle maître grâce à un utilitaire du système VM/SP, appelé "Virtual Machine Communication Facility" (VMCF).

Chaque machine virtuelle esclave sera ensuite connectée, en fonction de la disponibilité des APs déterminée par le gestionnaire de ressources ("Resource Manager"), à un calculateur FPS264 particulier et pourra communiquer avec celui-ci grâce à des utilitaires standard du système FPS (cf figure 6.2).

L'initialisation du job se termine par le chargement du code des AProutines dans chaque machine virtuelle esclave.

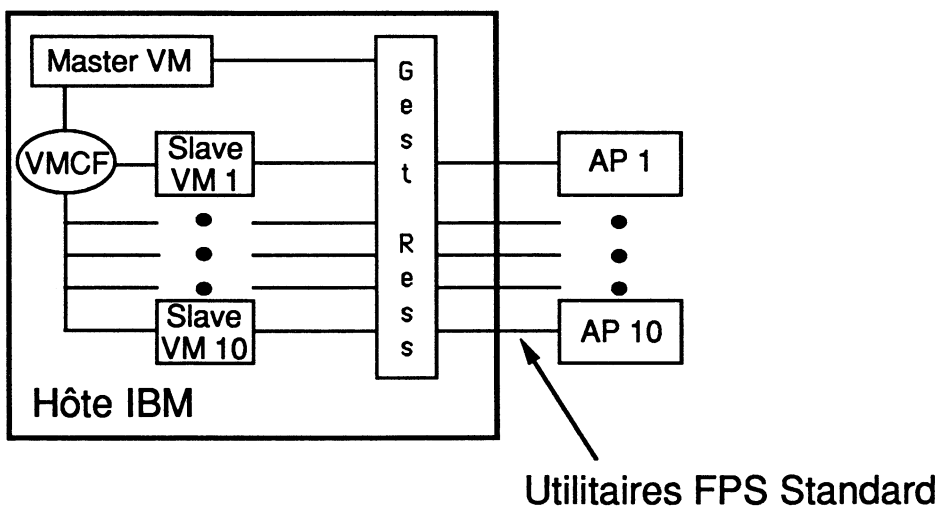

Figure 6.2: Implémentation d'un Programme d'Application Parallèle

Lors du lancement de la première exécution en parallèle d'une AProutine sur les APs, le code des AProutines est transféré par chaque machine virtuelle esclave vers l'AP qui lui est attaché. Chaque AP exécute alors le code de l'AProutine requise, avec des données qui lui sont propres.

L'exécution de l'AProutine en parallèle sur les différents APs définit donc des tâches parallèles, que nous appellerons **tâches esclaves**, ou plus simplement encore **esclaves**.

Ainsi que nous l'avons dit au paragraphe 6.1, un précompilateur traduit en FORTRAN77 (pour le programme maître) ou APFTN64 (pour les AProutines) les directives que l'utilisateur a inséré dans son code source initial (prévu pour une exécution séquentielle), pour définir et lancer les différentes tâches esclaves et pour exprimer toutes les communications et les synchronisations nécessaires entre ces tâches d'une part et entre la tâche maître et les tâches esclaves d'autre part.

Nous résumons ici les principales directives. Pour une information plus détaillée, le lecteur est invité à se référer à [Chin85,Calta87].

On distingue les directives de contrôle et les directives pour l'utilisation des mémoires de masse.

Les **directives de contrôle** sont utilisées pour:

- initialiser les machines virtuelles esclaves et leur affecter les APs requis (*START*),
- libérer les machines virtuelles esclaves et les APs qui leur sont attachés (*FINISH*),
- lancer l'exécution d'une AProutine sur un ou plusieurs APs en parallèle (*EXECUTE*),
- et synchroniser le maître et les esclaves en forçant le maître à attendre la fin de l'activité des esclaves (*WAIT*), soit la fin de l'exécution sur chaque AP de l'AProutine lancée auparavant.

Les **directives pour l'utilisation des mémoires de masse** sont divisées en deux catégories, dépendant précisément du mode d'utilisation des mémoires de masse:

- en **mode partagé**, les mémoires de masse sont utilisées comme de réelles mémoires partagées par les APs qui leur sont connectés, et permettent la réalisation de synchronisations entre les esclaves; des structures de données locales (*LOCAL*) et partagées (*SHARED*) peuvent être définies dans les mémoires de masse; chaque AP peut alors accéder sa propre copie des données locales et les données partagées grâce à des transferts (*MOVE*) entre sa mémoire propre et la mémoire de masse partagée; des synchronisations fortes, de type barrière (*BARRIER*), peuvent également être réalisées entre les esclaves d'un même job, grâce à l'utilisation par les APs attachés au job de drapeaux stockés dans la mémoire de masse partagée,

- en **mode message**, les mémoires de masse jouent le rôle de boîtes à lettres pour des communications asynchrones entre esclaves; l'utilisateur définit tout d'abord une topologie de réseau d'interconnexion entre les APs attachés à son job (*BULK MESSAGE NETWORK*); en fonction de cette topologie, le gestionnaire de ressources détermine, comme nous le verrons au paragraphe suivant, une configuration de chemins de données optimale à établir entre les APs au travers des mémoires de masse; les esclaves peuvent alors envoyer (*SEND*) et recevoir (*RECEIVE*) des messages à destination ou en provenance d'autres esclaves; précisons enfin que l'envoi de message est totalement asynchrone, alors que la réception de message n'est satisfaite que lorsque l'envoi du message attendu a été effectué.

Nous donnons ci-dessous la structure générale d'un programme maître, en précisant entre accolades les principaux paramètres accompagnant chaque directive:

```
      PROGRAM Master
      Section de calcul hôte   (instructions standard de FORTRAN77)
C$    START                    {nombre d'APs demandés,
                               identification éventuelle des APs demandés,
                               classe du job}
C$    BULK                     {mode d'utilisation des mémoires de masse}
      Section de calcul hôte
C$    EXECUTE ON               {nom de l'AProutine à exécuter,
                               identification des esclaves devant l'exécuter,
                               liste des paramètres effectifs}
      Section de calcul hôte
C$    WAIT FOR                 {identification des esclaves à attendre}
      Section de calcul hôte
      .
      .
      .
C$    FINISH
      Section de calcul hôte
      STOP
      END
```

Précisons que chaque section de calcul hôte peut être vide.

De même, voici la structure générale d'une AProutine:

```
        APROUTINE Rout(par1, ..., par2, ..., par3, ...)
        Section de calcul AP   (instructions standard de APFTN64)
        APIN arg1,...
        APIO arg2,...
        APOUT arg3,...
        Section de calcul AP
C$AP    Directive              (accès à une mémoire de masse)
        Section de calcul AP
        .
        .
        .
        RETURN
        END
```

L'instruction APFTN64 *APIN* permet de déclarer la liste des paramètres qui seront passés uniquement de la tâche maître aux tâches esclaves au début de l'exécution de l'AProutine. De même, l'instruction *APOUT* énumère la liste des paramètres que les esclaves renverront au maître à la fin de l'exécution de l'AProutine. Quant à l'instruction *APIO*, elle donne la liste des paramètres qui doivent être transmis au début et à la fin de l'exécution de l'AProutine.

Comme pour le programme maître, chaque section de calcul AP peut être vide.

Les directives que l'on peut rencontrer dans le code source d'une AProutine sont les suivantes:

```
C$AP  SEND      {identification de l'esclave destinataire,
                 taille du message}
C$AP  RECEIVE   {identification de l'esclave expéditeur,
                 taille du message}
C$AP  MOVE      {sens du transfert de données,
                 données à transférer}
C$AP  BARRIER
```

Entre accolades figurent les principaux paramètres à préciser pour chaque directive.

A titre d'illustration, nous proposons ci-dessous l'exemple d'un programme parallèle très simple, utilisant les mémoires de masse en mode message. Chaque esclave est identifié par le système à l'aide d'un numéro logique référencé par la variable *ISL*. Pour la correction de ce programme, on suppose que la procédure *PrecSuiv (ISL)* affecte à la variable *PREV* (respectivement *NEXT*) le numéro logique de l'esclave précédant (respectivement suivant) l'esclave courant de numéro logique *ISL*, conformément à la topologie en anneau (*RING*) définie pour le réseau d'interconnexion des APs attachés au job.

Programme Maitre

```
      PROGRAM PROG1
      Section de calcul hote 1
C$    START
C$    BULK MESSAGE NETWORK ('RING')
      Section de calcul hote 2
C$    EXECUTE ON ALL: ROUTINE(A,B,C,D)
C$    WAIT FOR ALL
      Section de calcul hote 3
C$    FINISH
      Section de calcul hote 4
      STOP
      END
```

AProutine

```
      APROUTINE ROUTINE(A,B,C,D)
      INTEGER PREV,NEXT,LNG
      APIN A,B,C
      APOUT D
      PrecSuiv (ISL)
      DO 10 I=1,5
C$AP     SEND (A,1) TO SLAVE PREV
         Section de calcul AP 2
C$AP     RECEIVE (D,LNG,1) FROM SLAVE NEXT
10    CONTINUE
      Section de calcul AP 3
      RETURN
      END
```

Cet exemple de programme parallèle sera repris au paragraphe 6.3.2 pour une illustration du modèle des programmes.

6.2.3 Fonctionnalités du Gestionnaire de Ressources

Le gestionnaire de ressources a pour tâche d'allouer aux jobs présents dans le système ICAP les ressources qu'ils demandent, principalement des APs et de l'espace dans les mémoires de masse. Il est constitué de deux machines virtuelles, le **gestionnaire d'APs** et le **gestionnaire de mémoires de masse**. Nous décrivons maintenant le travail effectué sans relâche par ces deux machines virtuelles.

Le Gestionnaire d'APs

Notre objectif n'est pas de fournir une liste exhaustive de toutes les tâches réalisées par le gestionnaire d'APs, mais plutôt d'insister sur les points intéressants de la politique d'allocation des APs.

Lors du lancement d'un job parallèle sur le système ICAP, l'utilisateur doit spécifier au gestionnaire de ressources, d'une part le nombre d'APs qu'il désire attacher à son job, d'autre part le temps d'exécution qu'il estime pour ce job. En fonction de ce temps estimé, une classe, appelée **classe de priorité** est attribuée au job. A chaque classe est associée une priorité et un temps maximal d'exécution.

Le gestionnaire d'APs alloue les APs aux différents jobs présents dans le système à partir d'un calcul de poids pour chaque job. Les poids des jobs évoluent dynamiquement tout au long de leur exécution. Le job de poids maximal reçoit les APs qu'il a demandés, au prix de "rollouts" des jobs de poids inférieur les utilisant.

Le poids de chaque job est fonction de la priorité de sa classe, du temps maximal d'exécution de sa classe, de son temps d'attente courant pour l'accès aux APs (composante dynamique), du nombre de jobs de même classe que lui présents dans le système (deuxième composante dynamique) et du nombre d'APs qu'il demande ou utilise. Les coefficients de cette fonction de poids (appelée **fonction de poids dynamique des jobs**) permettent de donner une importance relative aux diverses composantes, et ainsi de moduler la politique d'allocation des APs.

Lors du lancement d'un job parallèle, l'utilisateur peut également préciser sur quels APs physiques il veut que les tâches esclaves de son job s'exécutent. S'il ne le fait pas (on parle alors de **requête non spécifique**), le gestionnaire d'APs prendra la décision pour lui et affectera les APs qui lui paraissent les mieux appropriés, compte tenu de la charge courante du système, pour minimiser les contentions d'accès aux APs. Cependant, cette allocation statique ne s'effectue que lors de l'initialisation du job, et elle peut s'avérer moins optimale dans les heures qui suivent, compte tenu d'une évolution néfaste de la charge.

Afin de limiter le nombre de "rollin/rollouts" de jobs sur chacun des APs (processus très coûteux comme nous l'avons vu précédemment), deux restrictions sont appliquées à la politique d'allocation des APs présentée ci-dessus:

- tout job appartenant à la classe de priorité maximale ne peut être en aucun cas "rollouté", quels que soient les poids des autres jobs; son temps estimé d'exécution est en fait de l'ordre de quelques minutes seulement,
- tout job *presque terminé* ne sera pas non plus "rollouté"; on considère qu'un job est presque terminé lorsqu'un certain pourcentage de son temps d'exécution estimé s'est écoulé (ce pourcentage dépend bien sûr de la classe du job).

Ces concepts sont à la base de la politique d'allocation des APs qu'applique inlassablement la machine virtuelle du gestionnaire d'APs. Pour une plus large information relative à cette politique d'allocation, le lecteur pourra se référer aux deux rapports de recherche [Russo87a,Russo87b], qui expliquent en détail les raisons des choix qui ont été faits. [Russo87b] présente également la méthode, basée sur des simulations, qui a été suivie pour ajuster les coefficients de la fonction de poids dynamique des jobs. De plus, le lecteur trouvera dans [Caube88] une description complète de l'interface utilisateur du gestionnaire d'APs.

Le Gestionnaire de Mémoires de Masse

Le rôle du gestionnaire de mémoires de masse est d'allouer lors de l'initialisation de chaque job parallèle l'espace dans les mémoires de masse dont il a besoin. Cette allocation statique est fonction du mode avec lequel le job désire utiliser les mémoires de masse.

En mode partagé, une seule zone mémoire est affectée au job dans une mémoire de masse unique. Cette zone mémoire constitue la mémoire partagée par les APs attachés au job. La taille de cette zone mémoire est déterminée à partir de celle des variables locales et partagées définies respectivement à l'aide des directives *LOCAL* et *SHARED*. Dans la mesure du possible, le gestionnaire de mémoires de masse affectera une zone contenue dans une mémoire de masse locale, considérant que les accès à la mémoire de masse globale sont susceptibles de générer le plus de contention entre jobs parallèles.

En mode message, une zone mémoire sera allouée pour chaque chemin de données requis entre deux APs attachés au job, compte tenu de la topologie du réseau d'interconnexion entre les APs spécifiée par l'utilisateur. Les différentes zones mémoires seront réparties sur un nombre maximal de mémoires de masse afin de réduire les contentions d'accès à une même mémoire de masse pour ce job. De même que pour le mode partagé, le recours aux mémoires de masse locales sera privilégié.

6.3 Description du Modèle

Comme lors des modélisations précédentes, le modèle du système est composé du modèle de l'architecture [**A**] d'une part, et du modèle des programmes [**B**] d'autre part. Nous détaillons maintenant chacun de ces deux modèles.

6.3.1 Modèle de l'Architecture [A]

Le modèle de l'architecture est constitué par les stations de service du réseau de files d'attente, représenté sous une forme simplifiée à la figure 6.3.

Ce modèle ouvert est composé de 32 stations, dont les caractéristiques sont les suivantes:

- *ENTRY* représente la station source du réseau et modélise l'arrivée des jobs dans le système; les temps des interarrivées de jobs sont définis de manière déterministe pour la validation (à partir des traces d'exécutions de programmes composant le modèle des programmes), ou suivant une loi exponentielle pour des études plus générales (comme pour l'analyse de prédiction de performances décrite au paragraphe 6.7),
- l'hôte est décomposé en deux stations:
  - la station *HT* est dédiée aux sections de calcul hôte des jobs, et plus généralement, elle modélise la consommation CPU des jobs sur l'hôte; en d'autres termes, cette station modélise l'activité CPU de la machine virtuelle maître et des machines virtuelles esclaves de chaque job; sa discipline de file est celle du tourniquet ("round robin"), avec un quantum de quelques millisecondes; cette station comporte un serveur unique, dont le temps de service est défini de manière déterministe à partir d'un attribut des clients (cf modèle des programmes),

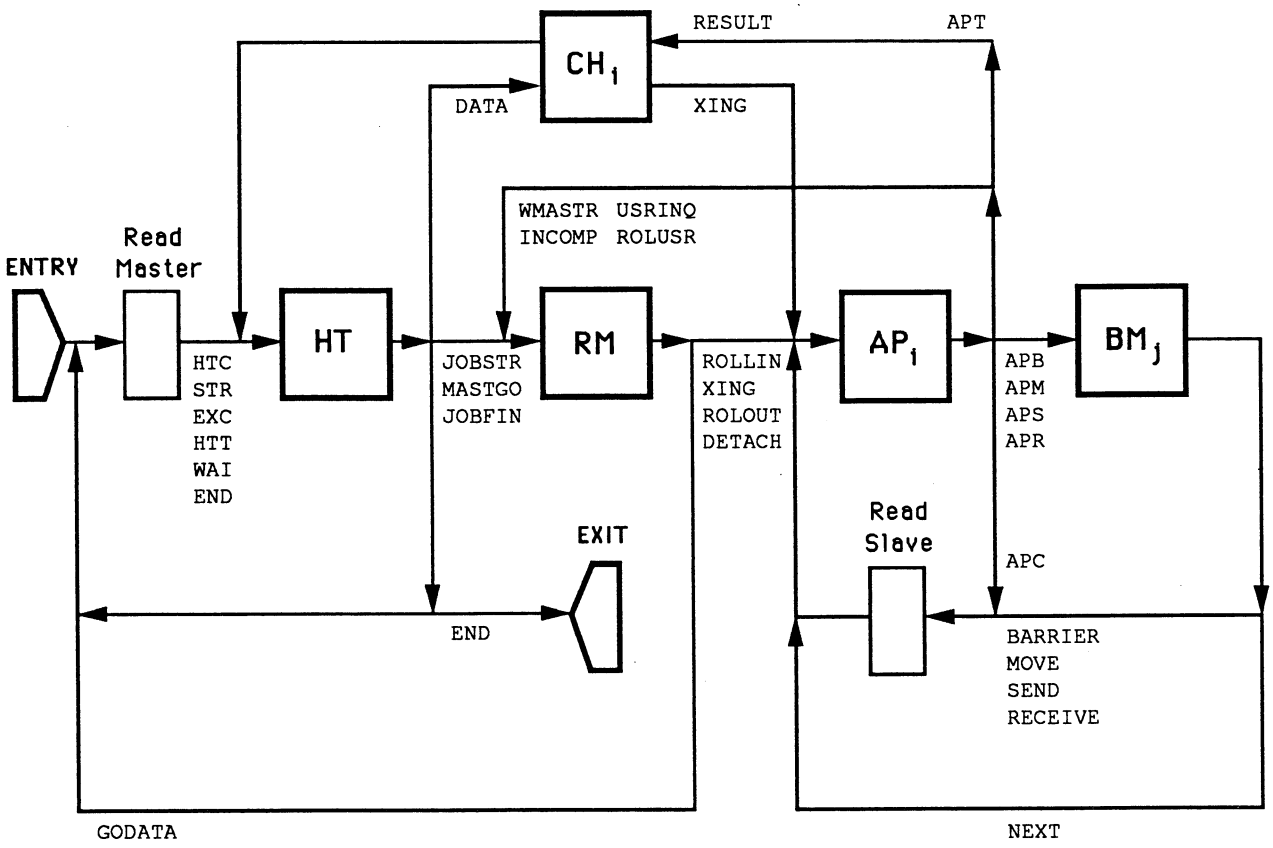

Figure 6.3: Modèle du Système ICAP/FPS264

- la deuxième station modélisant l'hôte est la station *RM* ("Resource Manager") qui représente les deux machines virtuelles du gestionnaire de ressources; du fait que le temps CPU consommé par ces machines virtuelles pour le traitement de chaque job s'est avéré fort difficile (pour ne pas dire impossible) à mesurer, nous avons décidé de le négliger; cette station est donc un délai (station à serveur infini) de temps de service nul, qui modélise seulement les fonctionnalités du gestionnaire de ressources.

Certes, cette modélisation de l'hôte est une forte simplification imposée par un manque d'information; cependant, en dédiant l'un des deux CPUs de l'IBM 3081 aux machines virtuelles du gestionnaire de ressources, on affecte une plus grande priorité à ces machines virtuelles qu'aux machines virtuelles attachées aux jobs, ce qui est le cas sur le système réel; une autre simplification a été nécessaire; en effet, la connaissance du système que nous avons acquise

ne nous a pas permis de modéliser les machines virtuelles esclaves des jobs; seule la machine virtuelle maître est donc prise en compte par le modèle, le temps CPU consommé par les machines virtuelles esclaves d'une part, et par les nombreuses communications entre la machine virtuelle maître et les machines virtuelles esclaves d'autre part, étant négligé,

- les stations CH_i, $i = 1, 10$, représentent les canaux hôte-APs; ce sont des délais du fait qu'il y a un canal hôte-AP par AP et que toute donnée devant être transférée entre la machine virtuelle esclave et l'AP qui lui est attaché peut donc être transmise immédiatement; le temps de service d'un client à ces stations est défini de manière déterministe, à partir du débit du canal hôte-AP, de la taille des données à transférer et de la direction du transfert,

- les stations AP_i, $i = 1, 10$, représentent les calculateurs scientifiques FPS264; ce sont des délais du fait que le système d'exploitation des FPS264 est mono-utilisateur et que l'attente nécessaire à l'accès aux APs est modélisée au niveau de la station *RM*; le temps de service d'un client à ces stations est défini de manière déterministe à partir d'un attribut du client (cf modèle des programmes); précisons dès maintenant que ce temps de service pourrait être obtenu, par application d'une méthode d'agrégation fort simple décrite au paragraphe 6.5.2, à partir de la résolution du modèle du FPS264, considéré comme un agrégat pour le modèle du système lCAP,

- les stations BM_j, $j = 1, 6$, représentent les 5 mémoires de masse ("Bulk Memories") locales ($j \leq 5$) et la mémoire de masse globale ($j = 6$); ce sont des stations à serveur unique car un seul accès aux mémoires de masse peut être satisfait à un instant donné; la discipline de file est FCFS ("First Come, First Served"); de nouveau, le temps de service d'un client à cette station est défini de manière déterministe, à partir du type d'accès réalisé (correspondant à la directive exécutée - *SEND*, *RECEIVE*, *MOVE* ou *BARRIER*), du type de la mémoire de masse accédée (locale ou globale), du nombre de mots transférés ou du nombre d'APs impliqués dans la synchronisation,

- la station *EXIT* modélise les terminaisons de jobs; chaque client arrivant à cette station sort immédiatement du modèle, sans attente ni service aucuns,

- les stations *Read Master* et *Read Slave* sont deux stations supplémentaires, utilisées uniquement pour le modèle déterministe des programmes (sous forme de traces d'exécutions de programmes - voir paragraphe suivant), qui modélisent la lecture de l'enregistrement suivant

de la trace d'exécution du programme maître (respectivement de la trace d'exécution d'une tâche esclave); ces stations sont de simples délais à temps de service nul.

6.3.2 Modèle des Programmes [B]

Le modèle des programmes est, rappelons-le, constitué des clients qui visitent les stations du réseau de files d'attente. Ces clients sont généralement regroupés en classes et on leur associe des attributs qui représentent leurs caractéristiques propres, qui sont fonction de leur classe d'appartenance.

Le modèle des programmes présenté ci-dessous n'échappe pas, bien sûr, à cette règle. Les clients pris en compte dans le modèle ont deux origines. Les uns sont une représentation très précise des jobs parallèles. Les autres modélisent, pour chaque job, les communications entre le gestionnaire de ressources, la machine virtuelle maître et les APs exécutant les tâches esclaves.

Compte tenu de la description du modèle de l'architecture (cf paragraphe 6.3.1) et de celle des programmes d'application (cf paragraphe 6.2.2), il nous a semblé raisonnable de considérer chaque section de calcul (hôte ou AP) comme un même client ainsi que chaque directive que l'utilisateur insère dans le code source. Nous notons donc une différence de finesse de la représentation des tâches de calcul [**D**] avec celle que nous avons utilisée pour le modèle des programmes du FPS264 (cf paragraphe 6.3.1), où l'instruction assembleur était l'élément de base.

A ce choix, deux raisons majeures: la complexité suffisante du modèle ainsi obtenu d'une part, et l'objectif de la modélisation qui est plus de caractériser le comportement des jobs parallèles sur le plan des communications et des synchronisations entre tâches que sur le plan du temps de calcul des tâches parallèles esclaves.

Néanmoins, nous montrerons au paragraphe 6.5.2 que ce choix ne remet pas en question l'utilisation possible du modèle du FPS264, par application d'une méthode d'agrégation [**J**] fort simple.

L'optique que nous avons retenue pour la représentation très précise des jobs parallèles est d'utiliser, pour la validation du modèle, des traces d'exécutions des programmes d'application. La **trace d'exécution d'un programme d'application** est constituée d'un ensemble de fichiers. Le premier fichier, appelé **fichier maître**, représente l'activité de la tâche maître. Ensuite, pour chaque AP attaché au job et chaque APRoutine définie dans le programme d'application, un fichier

spécifique, appelé **fichier esclave**, représente l'activité de chaque tâche parallèle. Chaque enregistrement d'un fichier contient la description d'un client du modèle des programmes, constituée par sa classe et ses attributs propres.

Nous verrons, lors de l'analyse de prédiction de performances (cf paragraphe 6.7.1), que les traces d'exécutions de programmes ont été remplacées par des chaînes de caractères, appelées chaînes de programmes, et des lois de distribution de probabilité, afin de permettre une modélisation probabiliste [**Q**] des jobs parallèles, nécessaire à une étude plus générale du comportement du système.

Quelle que soit l'optique retenue (traces d'exécutions de programmes ou modélisation probabiliste des programmes), nous avons distingué trois catégories principales de classes de clients:

1. les classes regroupant les clients associés à l'activité de la tâche maître de chaque job,
2. les classes regroupant les clients associés à l'activité des tâches esclaves de chaque job,
3. les classes regroupant les clients modélisant les communications entre le gestionnaire de ressources, la machine virtuelle maître et les APs exécutant les tâches esclaves.

Nous énumérons maintenant les différentes classes de clients, en distinguant chacune des trois catégories. Pour chaque classe de clients, nous précisons entre accolades les principaux attributs associés. Notons que, pour chacune des classes des deux premières catégories, le nom de la classe est aussi l'identificateur de l'enregistrement qui représente chaque client de cette classe dans le fichier maître (respectivement dans les fichiers esclaves), et que les attributs du client constituent les divers champs de cet enregistrement.

Ajoutons également que chaque client a un attribut supplémentaire, l'identificateur du job parallèle dont il modélise partiellement l'exécution.

Précisons enfin que les noms des différentes classes de clients sont placés sur la figure 6.3 sur les liaisons entre stations de service que les clients de ces classes empruntent.

Les classes des clients associés à l'activité de la tâche maître sont les suivantes:

- *HTC* représente chaque section de calcul hôte {temps CPU nécessaire à son exécution},
- *STR* modélise la directive *START* du programme maître {nombre d'APs à attacher au job, identification des APs si c'est une requête spécifique, taille du code et des données locales des AProutines, classe du job, temps d'exécution estimé, mode d'utilisation des mémoires de masse, topologie du réseau d'interconnexion des APs attachés au job pour le mode message},

- *EXC* représente chaque directive *EXECUTE* du programme maître {identification de l'AProutine à exécuter, identification des APs devant l'exécuter}; précisons que la directive *FINISH* est également modélisée par un client de cette classe, avec 0 comme identification de l'AProutine, et 100 comme identification des APs concernés, signifiant que tous les APs attachés au job sont requis,
- *HTT* et *DATA* représentent les transferts de données de la tâche maître vers les tâches esclaves, au début de l'exécution de chaque AProutine {nombre de paramètres effectifs de l'AProutine à transférer, taille de ces paramètres}; seul l'enregistrement de type *HTT* est présent dans le fichier maître,
- *WAI* modélise chaque directive *WAIT* du programme maître {identification des tâches esclaves avec lesquelles la tâche maître doit se synchroniser (100 signifiant une synchronisation avec toutes les tâches esclaves)},
- *END* représente la fin du programme maître.

Les classes des clients associés à l'activité des tâches esclaves sont les suivantes:

- *APC* représente chaque section de calcul AP {temps CPU nécessaire à son exécution},
- *APB* et *BARRIER* modélisent chaque directive *BARRIER* de synchronisation entre tâches esclaves; précisons que seul l'enregistrement de type *APB* est présent dans les fichiers esclaves,
- *APM*, *MOVE* et *NEXT* modélisent chaque directive *MOVE* de transfert de données depuis ou vers la mémoire de masse partagée {sens du transfert, taille des données à transférer, nombre de blocs non contigus de données}; à chaque bloc de données contiguës correspond un transfert physique de données; ainsi *MOVE* correspond au transfert du premier bloc de données contiguës, et le cas échéant, *NEXT* à celui des blocs suivants; seul l'enregistrement de type *APM* est présent dans les fichiers esclaves,
- *APS* et *SEND* modélisent chaque directive *SEND* d'envoi de message d'une tâche esclave vers une autre {taille du message, identification de la tâche destinataire}; seul l'enregistrement de type *APS* est présent dans les fichiers esclaves,
- *APR* et *RECEIVE* modélisent chaque directive *RECEIVE* de demande de réception de message d'une tâche esclave en provenance d'une autre {taille du message, identification de la tâche expéditrice}; seul l'enregistrement de type *APR* est présent dans les fichiers esclaves,

- *APT* et *RESULT* représentent les transferts de données de chaque tâche esclave vers la tâche maître, à la fin de l'exécution d'une AProutine {nombre de paramètres effectifs de l'AProutine à transférer, taille de ces paramètres}; seul l'enregistrement de type *APT* est présent dans les fichiers esclaves.

Les classes des clients modélisant les communications entre le gestionnaire de ressources, la machine virtuelle maître et les APs exécutant les tâches esclaves, sont les suivantes:

- pour les communications entre le maître et le gestionnaire de ressources:
  - *JOBSTR* représente le message que le maître adresse au gestionnaire de ressources lors de l'exécution de la directive *START*, pour lui préciser les ressources (APs et espace dans les mémoires de masse) dont le job parallèle a besoin {mêmes attributs que pour la classe *STR*},
  - *MASTGO* représente le message que le maître envoie au gestionnaire de ressources lors de l'exécution de chaque directive *EXECUTE*, pour lui signaler qu'il est prêt à envoyer aux APs les données nécessaires aux tâches parallèles pour exécuter l'AProutine {mêmes attributs que pour la classe *EXC*},
  - *GODATA* représente le message que le gestionnaire de ressources envoie au maître, dès qu'un AP est prêt à recevoir les données dont la tâche esclave, qui lui est attribuée, a besoin pour s'exécuter {identification de l'AP}; le maître envoie alors les données requises,
  - *JOBFIN* représente le message que le maître envoie au gestionnaire de ressources lors de l'exécution de la directive *FINISH*, pour lui signifier qu'il peut libérer à la fois les APs qui sont attachés au job et l'espace que le job occupe dans les mémoires de masse,
- pour les communications entre le gestionnaire de ressources et les APs exécutant les tâches esclaves:
  - *ROLLIN* représente le message que le gestionnaire de ressources envoie à un AP pour lui demander de "rolliner" un job {identification du job},
  - *INCOMP* représente la réponse de l'AP lorsqu'il a terminé son opération de "rollin",
  - *WMASTR* représente le message qu'un AP envoie au gestionnaire de ressources chaque fois qu'il est oisif, pour lui signaler qu'il est prêt à recevoir de nouvelles données en provenance du maître, c'est-à-dire à exécuter une nouvelle tâche esclave,

- *USRINQ* représente le message que chaque AP envoie au gestionnaire de ressources à la fin de chaque tranche de temps (dont la durée a été fixée à une minute pour le système ICAP), pour lui demander s'il doit "rollouter" le job dont il s'occupe actuellement ou s'il peut au contraire continuer de traiter ce job,
- *XING* et *ROLOUT* sont les deux réponses possibles du gestionnaire de ressources pour signaler respectivement à l'AP qu'il peut continuer avec le même job ou qu'il doit "rollouter" le job dont il s'occupe actuellement,
- *ROLUSR* est le message qu'un AP envoie au gestionnaire de ressources, chaque fois qu'il termine une opération de "rollout", pour lui signaler qu'il est maintenant prêt à prendre en charge le job de poids supérieur,
- *DETACH* est le message que le gestionnaire de ressources envoie à tous les APs d'un même job, pour leur dire que le maître n'a plus besoin d'eux (le gestionnaire de ressources vient de recevoir le message *JOBFIN* de la part du maître).

Un type d'enregistrement supplémentaire peut se rencontrer aussi bien dans le fichier maître que dans les fichiers esclaves, le type *BAC*. Contrairement aux autres enregistrements, il ne correspond pas à un client du modèle, mais il est utilisé pour simuler les boucles présentes dans le programme maître ou dans les AProutines. Ses trois champs représentent respectivement le niveau d'imbrication de la boucle, le nombre de fois qu'elle doit encore être itérée, et le nombre de lignes du fichier que le pointeur de fichier courant doit remonter, si le nombre d'itérations restantes n'est pas nul.

Nous terminons la description du modèle des programmes en présentant les traces d'exécutions de programmes qui seraient définies lors de l'exécution parallèle sur 3 APs du programme d'application *PROG1* introduit à la fin du paragraphe 6.2.2. Etant donné que ce programme comporte une seule AProutine, 3 fichiers esclaves seraient considérés en plus du fichier maître.

Fichier maitre	Fichier esclave 1	Fichier esclave 2	Fichier esclave 3
'HTC'	'APC'	'APC'	'APC'
tps_calcul1_hote	tps_calcul1_ap1	tps_calcul1_ap2	tps_calcul1_ap3
'STR'	'APS'	'APS'	'APS'
tps_directive_start	1	1	1
taille_code_donnees	3	1	2
classe_job	'APC'	'APC'	'APC'
tps_execution_estime	tps_calcul2_ap1	tps_calcul2_ap2	tps_calcul2_ap3
3	'APR'	'APR'	'APR'
7	1	1	1
3	2	3	1
10	'BAC'	'BAC'	'BAC'
tps_initialisation_job	1	1	1
2	5	5	5
1	12	12	12
1	'APC'	'APC'	'APC'
'RING'	tps_calcul3_ap1	tps_calcul3_ap2	tps_calcul3_ap3
'HTC'	'APT'	'APT'	'APT'
tps_calcul2_hote	1	1	1
'EXC'	8	8	8
100			
1			
'HTT'			
3			
24			
'WAI'			
100			
'HTC'			
tps_calcul3_hote			
'EXC'			
100			
0			

```
'WAI'
100
'HTC'
tps_calcul4_hote
'END'
```

Remarquons que les temps d'exécution d'une même section de calcul i sur les 3 APs peuvent très bien être différents, car les données affectées aux différents APs sont distinctes; ainsi, on peut avoir $tps\_calculi\_ap\_1 \neq tps\_calculi\_ap\_2 \neq tps\_calculi\_ap\_3$, $i = 1, 3$.

6.4 Estimation et Mesure des Paramètres d'Entrée

Dans ce paragraphe, nous distinguons le modèle de l'architecture et le modèle des programmes, et expliquons, pour chaque modèle, la manière avec laquelle nous avons obtenu ses paramètres d'entrée propres.

6.4.1 Modèle de l'Architecture

Les paramètres d'entrée requis pour le modèle de l'architecture sont de deux types:

- d'une part, toute une foule de données relatives à la configuration courante du système ICAP,
- d'autre part, quelques formules permettant un calcul fort simple du temps nécessaire à la réalisation d'une communication ou d'une synchronisation quelconque sur le système ICAP; ces formules seront utilisées pour calculer le temps de service des clients modélisant les directives réalisant ces communications et ces synchronisations.

Précisons maintenant la nature des paramètres propres à la configuration, qui sont déterminés à partir d'une simple observation du système et d'une étude de la politique d'allocation des APs, ainsi que la méthode qui nous a permis d'obtenir les formules permettant de caractériser le coût des communications et des synchronisations.

Nature des Paramètres Propres à la Configuration du Système

Les principaux paramètres, relatifs à la configuration du système et nécessaires à la résolution du modèle, sont les suivants:

- le nombre d'APs,
- le nombre de mémoires de masse et leur taille,
- le réseau d'interconnexion entre les APs et les mémoires de masse,
- le débit des canaux hôte-APs,
- la vitesse maximale des transferts de données entre APs et mémoires de masse,
- la durée maximale d'une opération de "rollin/rollout",
- le nombre de classes de jobs et leur priorité,
- le temps maximal d'exécution admis pour un job de chaque classe,
- les coefficients de la fonction de poids dynamique des jobs,
- et les seuils de non "rollout" pour les jobs *presque terminés* de chaque classe.

Evaluation du Coût des Communications et des Synchronisations

Faisons tout d'abord un bref rappel des directives qui permettent de réaliser les communications et les synchronisations sur le système ICAP:

- la directive *EXECUTE* réalise une communication entre la tâche maître et chacune des tâches parallèles esclaves qu'elle engendre,
- la directive *WAIT* réalise une synchronisation entre la tâche maître et les tâches esclaves; elle force en fait la tâche maître à attendre la fin de l'activité des tâches esclaves,
- la directive *SEND* réalise un envoi de message d'une tâche esclave vers une autre tâche esclave,
- la directive *RECEIVE* réalise une réception de message d'une tâche esclave en provenance d'une autre,

- la directive *MOVE* réalise un transfert de données depuis ou vers une mémoire de masse à partir d'une tâche esclave,
- et la directive *BARRIER* réalise une synchronisation, de type barrière, entre toutes les tâches parallèles esclaves.

Pour chacune de ces directives, nous avons dans un premier temps mesuré sur le système réel le temps CPU nécessaire à son exécution, en faisant varier les paramètres qui lui sont propres (nombre de tâches esclaves, taille des données, type de la mémoire de masse, ...).

Dans un deuxième temps, nous avons construit, à partir de la connaissance que nous avions du système ICAP, une formule simple permettant le calcul du temps CPU nécessaire à l'exécution de cette directive en fonction de ses paramètres propres.

La troisième et dernière étape a consisté à ajuster les coefficients constants de ces formules [**H**], afin de minimiser le carré des différences entre les coûts calculés et les temps CPU mesurés, différences obtenues pour les différents jeux de paramètres.

Nous ne reproduisons pas ici l'ensemble des résultats de notre étude. Le lecteur pourra se reporter à [Prost88b] pour en avoir une description exhaustive. Nous allons simplement illustrer la démarche employée pour les deux directives *MOVE* et *BARRIER*.

Directive de Communication MOVE

Pour les mesures, nous avons lancé sur un seul AP le petit programme suivant:

```
For    i:= 1 To 1000 Do
       appelle_horloge_système
       C$AP MOVE (n,m)
       appelle_horloge_système
       calcule_temps_CPU_consommé
End
calcule_moyenne_temps_CPU_consommés
```

n représente ici le nombre d'octets transférés et m le nombre de blocs non contigus de données. On a exécuté de nombreuses fois [**I**] ce même programme pour différentes valeurs de n et de m. Nous avons considéré successivement des accès en lecture et des accès en écriture, aussi bien vers une mémoire de masse locale que vers la mémoire de masse globale.

Notre connaissance du système ICAP nous a permis d'établir la formule suivante pour le calcul du coût de la directive MOVE [**S**]: $T_{move} = T_{init} + T_{bloc} * N_{bloc} + T_{mot} * N_{mot}$ (α)

où:

- T_{init} représente le temps d'initialisation globale du transfert de données (première composante du temps de service d'un client de classe APM à la station AP_i),
- T_{bloc} représente le temps d'initialisation du transfert de chaque bloc de données contiguës (deuxième composante du temps de service d'un client de classe APM à la station AP_i ou temps de service d'un client de classe $NEXT$ à la station AP_i),
- N_{bloc} représente le nombre de blocs non contigus de données (à chaque bloc de données contiguës, excepté le premier, correspond un client de classe $NEXT$),
- T_{mot} représente le temps nécessaire au transfert d'un mot de données,
- etN_{mot} représente la taille totale des données devant être lues ou écrites dans la mémoire de masse partagée; le temps donné par l'expression $T_{mot} * N_{mot} / N_{bloc}$ représente le temps de service du client de classe $MOVE$ (correspondant au transfert du premier bloc de données) ou de chaque client de classe $NEXT$ (correspondant aux blocs de données suivants) à la station BM_j.

L'ajustement au sens des moindres carrés [**H**], dont la qualité peut être appréciée sur la figure 6.4, a déterminé les valeurs suivantes pour les coefficients de la formule (α), l'unité étant la micro-seconde:

Mémoire de Masse	Locale		Globale	
Type d'Accès	Lecture	Ecriture	Lecture	Ecriture
T_{init}	77.24	77.24	83.69	83.69
T_{bloc}	59.28	59.28	66.35	66.35
T_{mot}	0.215	0.224	0.215	0.224

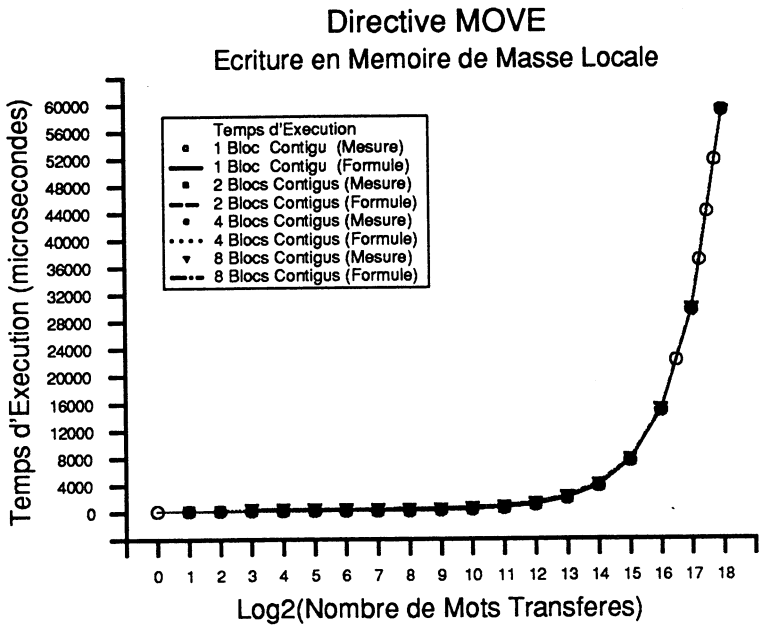

Figure 6.4: Directive *MOVE*: Comparaison entre Coût Mesuré et Coût Calculé

On remarque que les accès en lecture sont moins coûteux que ceux en écriture. De même, les accès vers une mémoire de masse locale sont plus rapides que ceux vers la mémoire de masse globale. De plus, nous pouvons noter la prépondérance des temps d'initialisation pour des transferts de données de petits volumes. On arrive aux mêmes conclusions avec les directives *SEND* et *RECEIVE* [Prost88b].

Directive de Synchronisation BARRIER

Pour la directive *BARRIER*, nous nous sommes intéressés au coût de la synchronisation en elle-même, en excluant de ce coût toute attente préalable des tâches parallèles (il est clair que l'attente sera prise en compte automatiquement par le modèle, par la mise en attente à la station AP_i des clients de classe *APB* jusqu'à l'arrivée du dernier d'entre eux).

Pour mesurer le coût de la synchronisation sans attente, nous avons considéré le petit programme suivant, lancé en parallèle sur un nombre variable d'APs:

```
For    i:= 1 To 1000 Do
       C$AP BARRIER
       appelle_horloge_système
       C$AP BARRIER
       appelle_horloge_système
       calcule_temps_CPU_consommé
End
calcule_moyenne_temps_CPU_consommés
```

La première directive BARRIER garantit que toutes les tâches parallèles sont synchronisées avant l'exécution de la deuxième directive. Le coût associé à la deuxième directive est donc bien le temps nécessaire pour libérer chaque tâche parallèle à partir de l'instant où toutes les tâches sont arrivées au point de synchronisation. Le maximum des coûts enregistrés sur les différents APs a été pris en compte afin de caractériser le coût maximal de cette synchronisation. Nous avons successivement utilisé comme mémoire partagée une mémoire de masse locale (limitant à 4 le nombre de tâches parallèles) et la mémoire de masse globale (permettant jusqu'à 10 tâches parallèles).

La formule que nous avons considérée a priori est la suivante:

$$T_{barrier} = T_{sync} + T_{ap} * N_{ap} \qquad (\beta)$$

où:

- T_{sync} représente la composante du coût de la synchronisation, qui ne dépend pas du nombre de tâches parallèles,
- T_{ap} représente le surcoût par tâche parallèle participant à la synchronisation,
- et N_{ap} représente le nombre de tâches parallèles qui prennent part à la synchronisation.

$T_{barrier}$ représente le temps de service du client de classe *BARRIER* à la station BM_j, modélisant la synchronisation.

L'ajustement aux moindres carrés [H], de qualité légèrement moins bonne que dans le cas précédent (du fait de fluctuations plus importantes des mesures), a permis la détermination des coefficients de la formule (β) suivants, l'unité étant toujours la micro-seconde:

Mémoire de Masse	Locale	Globale
T_{sync}	109.4	123.3
T_{ap}	2.73	3.22

On remarque d'une part que le coût de la synchronisation de type barrière entre tâches parallèles varie très peu avec le nombre de tâches, d'autre part que l'utilisation d'une mémoire de masse locale est préférable à celle de la mémoire de masse globale. Cette dernière remarque ainsi que la dernière formulée pour la directive MOVE expliquent la politique d'allocation des mémoires de masse appliquée par le gestionnaire de ressources.

6.4.2 Modèle des Programmes

Pour le modèle des programmes, l'ensemble des paramètres d'entrée nécessaires sont contenus dans les traces d'exécutions des programmes.

Actuellement, le fichier maître et les fichiers esclaves sont construits manuellement, à partir d'une analyse minutieuse du code source des programmes d'application, d'où une somme de travail assez considérable, même pour de petits programmes.

Les temps CPU des diverses sections de calcul, hôte ou AP, ont été, quant à eux, mesurés sur le système réel pour tous les jobs parallèles utilisés lors de la validation. Ce n'est certes pas la meilleure solution pour des analyses plus générales. Cette solution ne peut d'ailleurs être utilisée pour l'étude d'extensions système.

D'autres alternatives sont néanmoins possibles tout en conservant l'optique des traces d'exécutions de programmes:

- la plus simple d'entre elles est sans aucun doute d'estimer la durée des différentes sections de calcul, mais ce n'est certes pas la plus satisfaisante,
- une autre possibilité, beaucoup plus satisfaisante, est l'étude du code assembleur généré par le compilateur FORTRAN77 pour l'hôte (respectivement le compilateur APFTN64 pour les

APs) pour chaque section de calcul; en fonction de ce code, de la connaissance du nombre de temps de cycle nécessaires à l'exécution de chaque instruction élémentaire, et de la durée du temps de cycle de la machine, on peut calculer le temps CPU nécessaire à l'exécution de chaque section de calcul; cette méthode reste toutefois fort fastidieuse et la prise en compte de données dynamiques telles que le nombre d'itérations des différentes boucles du programme est fort difficile; on se rend compte en outre de la similitude de cette méthode avec celle employée pour le modèle du FPS264 (cf chapitre 5),

- la troisième possibilité est de recourir pour les sections de calcul hôte à des mesures (il est raisonnable de penser que les extensions systèmes ne concerneront pas l'hôte; si elles le concernent, on aura recours à des estimations), et pour les sections de calcul AP à l'utilisation du modèle du FPS264; nous montrerons au paragraphe 6.5.2 que cette solution consiste en fait à utiliser une méthode d'agrégation fort simple (de par sa nature), mais très ardue quant à son application.

Nous ne disposons pas à ce jour de méthode miracle, et bien souvent, un compromis entre estimation et mesures restera la solution la plus raisonnable, notamment pour des études générales où la nécessité d'une modélisation très fine des sections de calcul n'est pas requise, et où seules les durées relatives des sections de calcul sont importantes. C'est cette optique qui a prévalu pour l'analyse de prédiction de performances présentée au paragraphe 6.7.

6.5 Résolution du Modèle

Ainsi que nous l'avons déjà dit, pour résoudre un modèle sous forme de réseau de files d'attente, plusieurs méthodes peuvent a priori être utilisées, les méthodes analytiques, exactes ou approchées, et la simulation.

Les méthodes analytiques font appel à des hypothèses strictes que doit satisfaire le modèle. En l'occurrence, la discipline de file des stations AP_i, induite par la politique d'allocation des APs fort complexe réalisée par le service de la station RM, de même que les schémas de synchronisation engendrés par les directives $WAIT$, $RECEIVE$ et $BARRIER$, invalident l'utilisation des méthodes analytiques connues à ce jour.

La simulation, quant à elle, est toujours utilisable, mais à un coût de développement et d'exécution beaucoup plus important. Contraints de l'utiliser pour résoudre notre modèle, nous avons écrit un programme de simulation en FORTRAN77, dont le développement s'est opéré en plusieurs étapes:

1. tout d'abord, nous avons écrit un programme simulant uniquement la station de service RM, qui modélise le gestionnaire de ressources; il était en effet intéressant, au-delà du fait que ce simulateur était nécessaire au programme de simulation du modèle global, de disposer d'un outil permettant de mieux comprendre la politique d'allocation des APs fort complexe, voire d'ajuster cette politique par un changement de la fonction de poids dynamique des jobs ou par une simple modification de ses coefficients,

2. ensuite, pour chacune des autres stations, nous avons développé un module spécifique,

3. l'interconnexion des différents modules a constitué la phase finale du développement et a permis l'obtention du programme de simulation global.

La principale raison, qui nous a conduits à écrire le programme de simulation à partir d'un langage de programmation standard tel que FORTRAN77, est le fait que le code source du gestionnaire de ressources était écrit dans ce langage.

De larges portions de ce code ont donc pu être incluses dans le programme de simulation avec des modifications mineures, accélérant d'un facteur non négligeable le temps de développement.

De plus, nous maintenions ainsi un parallèle plus étroit entre le code du gestionnaire de ressources et le code du simulateur de la station RM, permettant ainsi une répercussion fort aisée de toute modification apportée au code du gestionnaire de ressources sur le code du simulateur, ou une mise à jour tout aussi facile de la politique d'allocation des APs du gestionnaire de ressources si l'utilisation du simulateur mettait en évidence une amélioration sensible du comportement du système que pourrait engendrer cette mise à jour.

Une autre raison pour ce choix a été une gestion optimisée des synchronisations entre clients, optimisation que ne permettait pas le recours à des logiciels de description et de résolution de réseaux de files d'attente, tels que QNAP [Véran84] ou RESQ [Sauer86].

Ajoutons enfin que la portabilité du programme de simulation était ainsi garantie. Elle fut d'ailleurs mise à profit, certaines simulations nécessaires à la validation du modèle ayant été exécutées sur un IBM 3084 à IBM Kingston, d'autres sur un IBM PS/2 à Grenoble.

Nous attirons maintenant l'attention du lecteur sur le fait que ce premier programme de simulation, adapté à une simulation à base de traces d'exécutions de programmes (constituant le modèle des programmes), a dû être assez largement modifié lors de l'analyse de prédiction de performances, par suite d'une modification du modèle des programmes (cf paragraphe 6.7.1), nécessaire pour une modélisation probabiliste des programmes d'application. Nous avons alors migré vers le langage Turbo Pascal, pour une meilleure structuration des données et la rapidité de mise au point sur micro-ordinateur procurée par son environnement intégré. Nous présentons à l'annexe A la structure générale de ce deuxième simulateur du système ICAP.

La suite de ce paragraphe présente plus en détail le simulateur du gestionnaire de ressources, puis ouvre une parenthèse pour expliquer comment le modèle du FPS264, décrit au chapitre 5, pourrait être utilisé comme sous-modèle de chaque station AP_i, par application d'une méthode d'agrégation fort simple.

6.5.1 Simulateur du Gestionnaire de Ressources

Rappelons tout d'abord que le simulateur du gestionnaire de ressources, nécessaire bien sûr au simulateur du modèle global, a été développé séparément pour produire un outil permettant de mieux comprendre la politique d'allocation des APs fort complexe, voire d'ajuster cette politique par un changement de la fonction de poids dynamique des jobs ou par une simple modification de ses coefficients. Son code, volontairement très proche de celui du gestionnaire de ressources réel, facilite les répercussions sur ce dernier de modifications relatives à la politique d'allocation des APs (voire des mémoires de masse), que des simulations auront révélées comme pouvant être bénéfiques.

Le but de ce paragraphe est essentiellement de présenter les fonctionnalités de ce simulateur. Ainsi sont brièvement abordées d'une part la description des données dont il a besoin en entrée et des résultats qu'il produit en fin de simulation, d'autre part la présentation succincte de son interface graphique, qui a été développé à l'aide du logiciel graPHIGS [Graph85], pour augmenter de manière considérable son pouvoir d'analyse du comportement du gestionnaire de ressources.

Pour de plus amples détails, le lecteur est invité à se reporter au rapport de recherche [Prost88a], qui constitue véritablement le guide de l'utilisateur de ce simulateur. En particulier, il pourra y trouver la description très complète de l'interface graphique. Ce rapport présente, en outre, un générateur aléatoire des entrées du simulateur, qui permet facilement l'utilisation du simulateur pour des études statistiques plus générales sur le comportement du gestionnaire de ressources.

Entrées-Sorties du Simulateur

La figure 6.5 donne le schéma des entrées-sorties du simulateur.

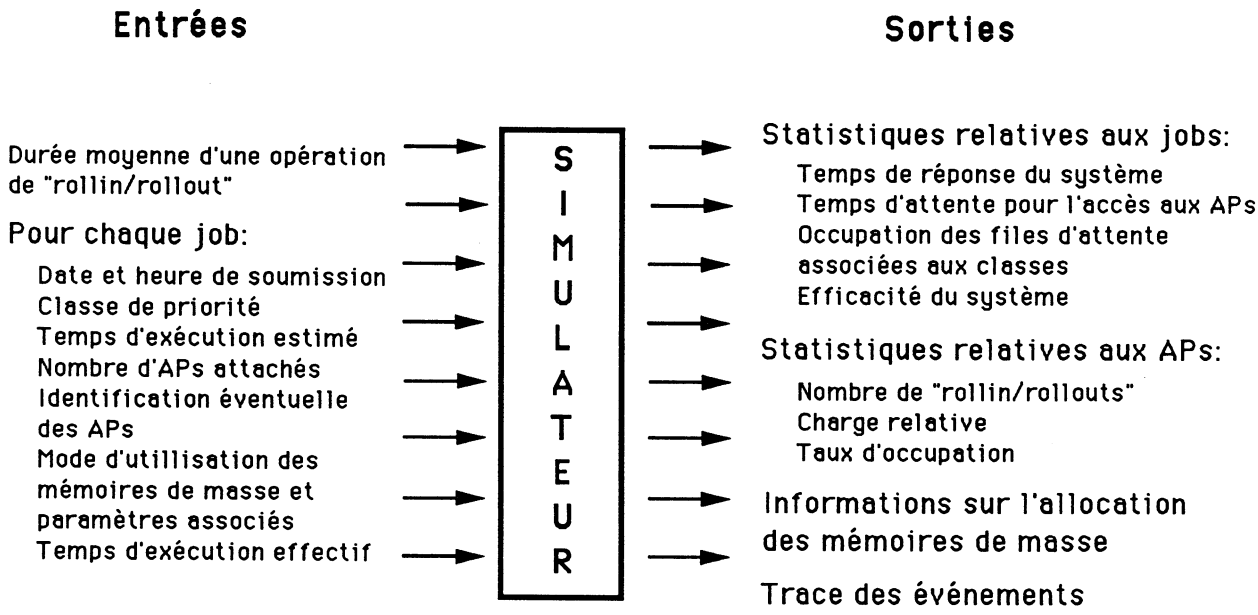

Figure 6.5: Entrées-Sorties du Simulateur du Gestionnaire de Ressources

En entrée, le simulateur requiert:

- la durée moyenne d'une opération de "rollin/rollout",
- et pour chaque job:
  - les date et heure de sa soumission au système,
  - sa classe de priorité,
  - le temps estimé de son exécution,
  - le nombre d'APs à lui attacher,
  - l'identification éventuelle de ces APs (pour une requête spécifique),
  - son mode d'utilisation des mémoires de masse,
  - et suivant ce mode, quelques paramètres supplémentaires propres (par exemple la topologie du réseau d'interconnexion des APs).

A cette liste de paramètres, correspondant en tout point aux paramètres que l'utilisateur doit spécifier lors du lancement de son job sur le système réel, il faut ajouter un paramètre supplémentaire, le temps effectif d'exécution du job.

En effet, l'exécution du job n'est pas simulée par ce simulateur, seuls les événements relatifs à l'utilisation des ressources sont pris en compte. Il est donc nécessaire de connaître pour chaque job l'instant où il va se terminer.

Il est clair cependant que ce paramètre ne sera plus nécessaire au simulateur du système global, qui, lui, simulera toute l'activité du job.

La considération d'une part du temps estimé de l'exécution du job, d'autre part du temps effectif de son exécution, permet la simulation de la **règle de pénalisation des jobs** appliquée par le gestionnaire de ressources, qui peut s'énoncer ainsi: "tout job, dont le temps effectif d'exécution dépasse, de plus d'un certain pourcentage, le temps estimé par l'utilisateur, se voit attribuer la classe de priorité minimale et est mis en attente derrière tous les jobs présents alors dans cette classe". Cette règle de pénalisation a pour but d'éviter une sous-estimation volontaire, de la part des utilisateurs, de la durée d'exécution de leur job, leur permettant d'accéder ainsi à une classe de priorité supérieure.

Les principaux résultats fournis à l'issue de la simulation sont:

- des statistiques relatives aux jobs, telles que:
  - le temps de réponse du système pour exécuter chaque job, avec la moyenne pour chaque classe de priorité,
  - le temps d'attente pour l'accès aux APs, pour chaque job, avec la moyenne pour chaque classe de priorité,
  - l'occupation des files d'attente associées aux différentes classes de priorité,
  - et l'efficacité du système vis-à-vis de chaque job, définie comme le rapport du temps d'exécution effectif sur le temps de réponse du système (plus cette quantité est proche de 1, meilleure est l'efficacité), avec la moyenne pour chaque classe de priorité; il est souhaitable que l'efficacité du système (qui est liée étroitement à la satisfaction de l'utilisateur ayant soumis le job) soit croissante avec la priorité de la classe du job, c'est-à-dire avec un temps d'exécution décroissant,

- des statistiques relatives aux APs, par AP, avec une moyenne sur tous les APs, telles que:
  - le nombre de "rollin/rollouts", permettant une évaluation du coût de ces opérations, qui constituent un gaspillage de temps pour l'utilisation des APs,
  - la charge relative, définie comme la durée relative pendant laquelle chaque AP a exécuté un job, la charge de l'AP le plus chargé étant normalisée à 1; cette quantité permet d'appréhender d'éventuels déséquilibres de charge entre APs, inhérents à la politique d'allocation des APs pour les requêtes non spécifiques,
  - et le taux d'occupation, défini comme le rapport du temps d'exécution (incluant les opérations de "rollin/rollout") sur la durée de la simulation; voisine de la précédente, cette donnée permet une autre estimation de l'équilibrage de charge entre APs, et, en la comparant à la précédente, une évaluation du temps gaspillé sur chaque AP,
- des informations sur l'allocation des mémoires de masse aux différents jobs, avec notamment, dans le cas du mode message, les différents chemins de données pour les communications entre APs,
- une trace de tous les événements relatifs à l'allocation des APs, identique à celle que produit sur le système réel le gestionnaire de ressources, permettant, entre autres, une reproduction de phénomènes étranges observés sur le système réel, ou une comparaison très fine entre le comportement du système réel et celui du simulateur, après une légère modification de la politique d'allocation des APs dans le simulateur; on doit souligner que la lecture de cette trace reste assez ardue pour un non-initié, cependant elle s'est avérée d'une grande aide lors du "debugging" du simulateur, comme du gestionnaire de ressources réel.

Interface Graphique

Certes, les résultats fournis à l'issue de la simulation, énumérés au paragraphe précédent, sont riches d'enseignement sur la politique d'allocation des APs qu'applique le gestionnaire de ressources, pour une charge du système caractérisée par les entrées du simulateur.

Toutefois, comme toute statistique globale sur l'ensemble d'une simulation, ils peuvent masquer certains aspects transitoires du comportement du gestionnaire de ressources. Il est vrai que la lecture de la trace des événements peut pallier ce problème, mais au prix de longues heures de

dur labeur. C'est pourquoi, il nous a semblé fort intéressant de pouvoir disposer d'un interface graphique (cf figure 6.6), permettant, **en temps réel simulé**, de visualiser l'évolution de l'état de charge du système ainsi que l'ensemble des actions réalisées par le gestionnaire de ressources.

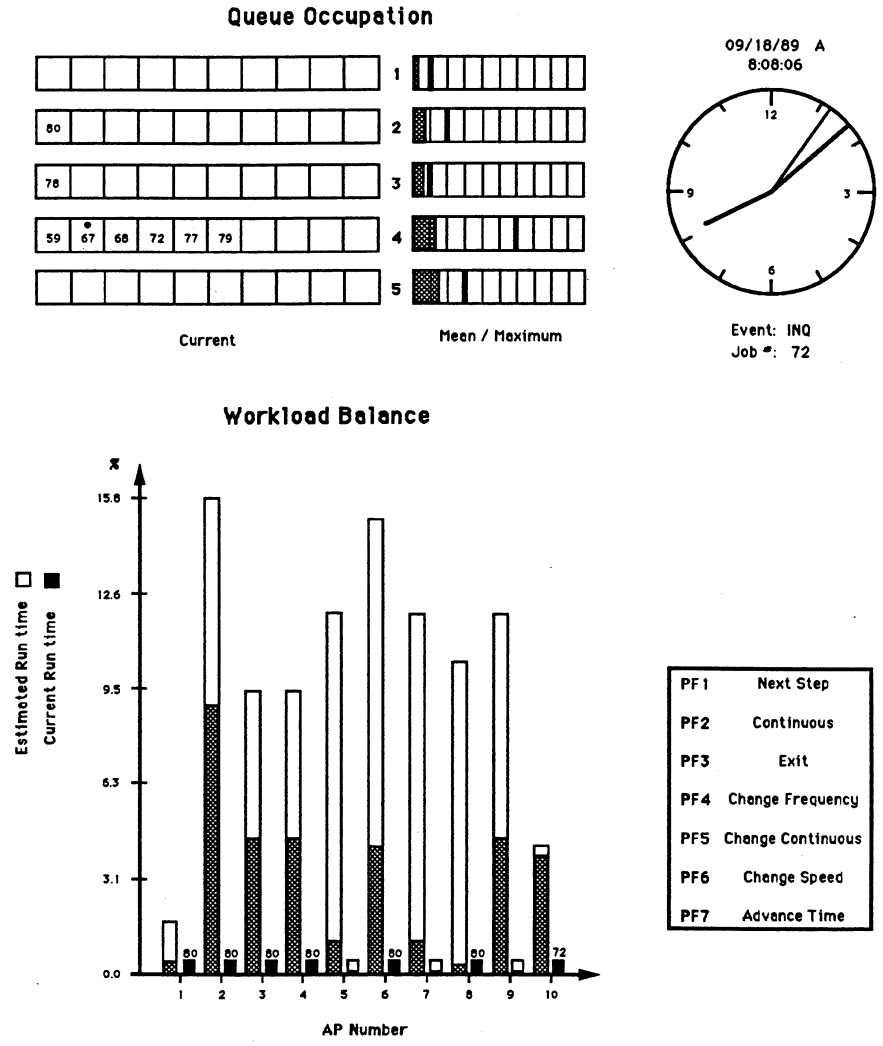

Figure 6.6: Interface Graphique du Simulateur du Gestionnaire de Ressources

Cet interface graphique, mis à jour lors de l'occurrence de chaque événement simulé, est divisé en quatre zones:

1. la zone supérieure gauche, intitulée "Queue Occupation", représente l'occupation des files d'attente associées aux 5 classes de priorité; chaque job est désigné par un numéro, surmonté d'un point si la requête est spécifique et une couleur lui est associé suivant son état (actif, en attente d'APs, devant perdre ou acquérir ses APs en fin de tranche de temps, etc.),

2. la zone inférieure gauche, intitulée "Workload Balance", permet d'estimer l'équilibrage de charge entre les APs, en comparant la hauteur des barres verticales, représentant, pour les différents APs, la somme des temps d'exécution des jobs, estimés par les utilisateurs; chaque barre se remplit au fur et à mesure de l'avancement de l'exécution du job occupant l'AP correspondant; en outre, un petit carré, à la droite de chaque barre, signale si l'AP est actuellement actif (carré plein) ou en attente (carré vide), le(s) job(s) susceptible(s) de l'utiliser ne pouvant accéder à tous les APs dont il(s) a (ont) besoin,

3. la zone supérieure droite donne le temps actuel simulé ainsi que l'événement courant, et le numéro du job qui a provoqué cet événement,

4. et la zone inférieure droite rappelle à l'utilisateur la signification des 7 touches de fonction dont il dispose; ces touches de fonction lui permettent respectivement:

   (a) de mettre à jour la représentation graphique en simulant le prochain événement (mode pas à pas),

   (b) de lancer la mise à jour de la représentation graphique pour plusieurs événements consécutifs, avec une temporisation entre les mises à jour successives (mode continu),

   (c) d'arrêter l'interface graphique (la simulation se poursuit en mode transparent),

   (d) de modifier certaines caractéristiques d'affichage (fréquence de mise à jour - exprimée en nombre d'événements simulés -, nombre de mises à jour consécutives en mode continu, durée de la temporisation en mode continu),

   (e) et de différer la prochaine mise à jour de la représentation graphique jusqu'à l'occurrence du premier événement postérieur à une heure spécifiée.

Nous rappelons que le lecteur dispose d'une description fort détaillée de cet interface graphique dans [Prost88a], où il pourra, entre autres, prendre connaissance des améliorations envisagées pour cet interface.

6.5.2 Utilisation du Modèle du FPS264 [J]

Nous évoquons dans ce paragraphe comment le modèle du FPS264, décrit au chapitre 5, pourrait être utilisé lors de la résolution du modèle du système ICAP, et nous insistons sur les modifications du modèle du FPS264 nécessaires à cette utilisation.

Rappelons que, pour des contraintes de temps d'une part, et vu la complexité de ces modifications d'autre part, nous avons préféré renoncer dans un premier temps à l'utilisation de ce modèle.

La question de l'applicabilité de l'approche que nous allons décrire maintenant reste d'ailleurs totalement ouverte, dans la mesure où le temps de résolution du modèle détaillé du FPS264, sous forme de réseau de Petri stochastique, lorsqu'elle est nécessaire, pourrait très bien se révéler très coûteuse, donc peu raisonnable, voire impossible (de par le nombre d'états constituant le graphe des marquages), tout en tenant compte des réductions induites sur le graphe des marquages, par l'application des nouvelles règles de mise à feu des transitions synchronisées par l'horloge et de la méthode itérative d'agrégation-désagrégation introduites au paragraphe 5.5.1.

Présentons donc l'utilisation possible du modèle du FPS264, qui aurait pour principal avantage de se substituer à la mesure sur le système réel ou à l'estimation du temps d'exécution de chaque section de calcul AP.

L'utilisation du modèle du FPS264 pour la résolution du modèle du système ICAP est basée sur une méthode d'agrégation fort simple: on considère chaque AP comme un agrégat du modèle global du système ICAP. Chaque agrégat ainsi formé est résolu, à l'aide des deux modèles détaillé et simplifié du FPS264, pour chaque section de calcul AP du job afin d'en déterminer le temps d'exécution. Ce temps est ensuite inséré dans la trace d'exécution du programme étudié comme paramètre de la section de calcul AP correspondante.

Plus précisément, pour chaque AProutine définie dans le programme d'application du job, on isole chaque section de calcul AP. On en récupère le code assembleur généré par le compilateur APFTN64 (grâce à une option du compilateur). Puis, on effectue l'analyse lexicale de ce code assembleur afin de définir les caractéristiques de la section de calcul AP. Grâce à une ACP réalisée sur ces caractéristiques, on détermine la classe de programmes dont la section de calcul se rapproche le plus, parmi celles définies à partir des procédures de la bibliothèque APMATH64.

Ensuite, de deux choses l'une: ou la section de calcul AP semble appartenir à l'une des quatre classes déjà définies: on peut alors utiliser directement le modèle simplifié obtenu pour cette classe et le résoudre par simulation; ou elle semble éloignée de chacune des quatre classes: dans ce cas, elle devient le représentant de sa propre classe (qui comptera peut-être d'autres éléments lors de l'analyse des autres sections de calcul AP de ce job ou de celles d'autres jobs), et l'on doit résoudre, dans un premier temps, le modèle détaillé, et dans un deuxième temps, le nouveau modèle simplifié, déduit de la résolution du modèle détaillé.

On se rend compte de la somme de travail nécessaire à l'application de cette méthode. En outre, cette méthode présente la même faiblesse que celle déjà constatée au paragraphe 5.4.1, à savoir le fait que l'analyse lexicale fournit uniquement les caractéristiques statiques des programmes, entraînant une approximation non négligeable dans la détermination du poids de chaque ligne assembleur composant le code analysé.

Dans le cas qui nous intéresse, l'approximation faite pour les procédures de la bibliothèque APMATH64, qui pouvait se justifier, risque fort de ne plus être loisible sur un code quelconque. Par conséquent, une modification majeure est ici nécessaire pour le modèle des programmes du FPS264, qui doit être basé sur une **analyse dynamique du code assembleur**, et non plus sur une analyse lexicale statique. Cette analyse dynamique risque fort d'être très ardue à réaliser de manière automatique, et j'avoue humblement que je ne suis pas à même de proposer une solution acceptable à ce jour.

6.6 Validation

Rappelons que la phase de validation consiste à comparer les critères de performances du système, estimés à l'aide du modèle, avec ceux obtenus par mesure sur le système réel. Ainsi que nous le soulignions dès le paragraphe 1.1.6, la comparaison a pour principal objectif de **mettre en évidence les erreurs** dues aux approximations de modélisation. Le diagnostic de l'origine des erreurs constatées est dressé dans la mesure du possible. Bien sûr, si une différence négligeable apparaît entre estimations et mesures, le processus de validation se limite à la comparaison.

La démarche de validation que nous avons suivie peut se résumer comme suit:

1. tout d'abord, on a réalisé une campagne de mesures pour déterminer, sur le système ICAP dédié [**N**], le temps d'exécution de plusieurs jobs de types différents, sous diverses conditions de charge; pour ce faire, on a défini plusieurs scénarios, de durée limitée, afin d'obtenir un nombre très important de données dans le temps qui nous était imparti pour réaliser nos mesures (4 à 5 heures); la campagne de mesures nous a permis également, lors d'une série de mesures bien distincte [**O**], de déterminer tous les paramètres nécessaires à la constitution des traces d'exécutions de programmes associées à chaque scénario,

2. à partir des données recueillies à l'étape précédente pour chaque scénario, on a construit les traces d'exécutions de programmes, c'est-à-dire les fichiers maître et esclaves de chaque job constituant le scénario,

3. dans un troisième temps, on a lancé le programme de simulation du modèle, en utilisant, pour chaque scénario, l'ensemble des traces d'exécutions de programmes le caractérisant,

4. nous avons ensuite comparé, pour chaque job de chaque scénario, le temps d'exécution mesuré et le temps d'exécution estimé à partir de la simulation,

5. le calcul et l'analyse de la différence relative entre temps mesuré et temps estimé nous ont permis de déduire les conditions dans lesquelles le programme de simulation donne une bonne évaluation des performances du système, ainsi que les raffinements à apporter au modèle pour réduire les erreurs relevées et étendre donc sa validité.

La suite de ce paragraphe est consacrée à la description de ces différentes étapes. On justifie tout d'abord le choix des programmes utilisés pour la campagne de mesures. Les différents scénarios considérés sont présentés, en distinguant deux types de scénarios, les scénarios à job unique et ceux à jobs multiples. Enfin, l'essentiel des nombreux résultats obtenus est analysé et largement commenté, avant de statuer quant à la validité du modèle.

6.6.1 Choix des Programmes d'Application

Cinq programmes d'application ont été sélectionnés, pour leur durée d'exécution très courte d'une part (de l'ordre de quelques minutes tout au plus), et pour les différents modes d'utilisation des mémoires de masse qui les caractérisent:

1. *PROG1* implémente une méthode parallèle de résolution d'un problème de flux transsonique, basée sur une décomposition du domaine de discrétisation [Broch85]; ce programme n'utilise pas les mémoires de masse, si bien que toutes les communications entre tâches parallèles esclaves se font par l'intermédiaire de la tâche maître; *PROG1* peut s'exécuter sur un, deux, trois, quatre, six, huit ou dix APs tout en maintenant un bon équilibrage de charge entre tâches parallèles,

2. *PROG2* implémente la même méthode que *PROG1*; cependant, les communications entre tâches parallèles se font directement au travers des mémoires de masse par utilisation du

mode message; lors de la campagne de mesures, des pannes ont limité à six le nombre d'APs attachés à un même job en mode message, si bien que *PROG2* n'a pu être exécuté qu'avec deux, trois, quatre ou six APs,

3. *PROG3* implémente une méthode parallèle de résolution des "shallow water equations", qui utilise également le mode message [Capot86]; du fait de pannes, seuls deux, trois, quatre, cinq ou six APs ont pu être attachés à un job exécutant *PROG3*,

4. *PROG4* et *PROG5* implémentent tous deux une méthode multi-grilles parallèle [Herbi87], *PROG4* utilisant une méthode de résolution directe au niveau de la grille la moins fine et *PROG5* une méthode itérative (Gauss-Seidel); le mode partagé est utilisé pour stocker dans la mémoire de masse partagée les grilles des différents niveaux, accessibles ainsi par toutes les tâches parallèles; ces deux programmes sont conçus pour être exécutés sur deux ou quatre APs.

Grâce à ces cinq programmes, on a pu prendre en compte les trois différentes manières possibles de réaliser des communications et des synchronisations entre tâches parallèles, soit par l'intermédiaire de la tâche maître, soit au travers des mémoires de masse en mode message, soit en accédant une mémoire de masse partagée. De plus, ces cinq programmes présentent l'avantage de pouvoir supporter une parallélisation plus ou moins importante, et ceci avec les mêmes données d'entrée, procurant ainsi une plus grande flexibilité dans la définition des scénarios.

Ces cinq programmes, choisis volontairement pour leur temps d'exécution très court, ne reflètent cependant pas exactement le type des programmes s'exécutant habituellement sur le système ICAP. Les programmes les plus fréquemment utilisés en production sont des gros programmes de simulation moléculaire et leur exécution nécessite couramment **plusieurs jours de temps CPU**. Le lecteur comprendra pourquoi il nous était difficile d'utiliser de tels programmes pour la validation.

A la différence des programmes de validation, les programmes couramment utilisés en production sont caractérisés par un **temps d'initialisation tout à fait négligeable** devant leur temps global d'exécution, et par une **granularité beaucoup plus forte** des tâches parallèles esclaves et des tâches de calcul entre les communications inter-esclaves.

C'est pourquoi le lecteur ne devra pas être étonné de constater que le temps d'exécution des programmes de validation augmente rapidement avec le nombre d'APs attachés au job (conduisant à une décélération du temps d'exécution parallèle) au lieu de diminuer comme l'on pourrait s'y attendre. Cela est dû d'une part à un temps d'initialisation du job souvent prépondérant dans son temps d'exécution global, d'autre part à la faible granularité des tâches parallèles esclaves. Le surcoût des communications et synchronisations entre tâches compense alors le gain apporté par la parallélisation des sections de calcul.

Précisons enfin que les efficacités relevées lors de l'exécution parallèle des programmes de production sont de l'ordre de 95% à 98% jusqu'à 10 APs attachés. Le système ICAP est donc tout à fait **performant dans son environnement de production**. De plus, son évolution au fil des années, caractérisée par l'addition des mémoires de masse pour limiter les communications hôte-APs, a permis d'étendre considérablement le spectre des applications pouvant s'exécuter en parallèle de manière efficace sur le système. Le lecteur pourra d'ailleurs constater que les programmes *PROG2* et *PROG3*, utilisant essentiellement des communications inter-APs au travers des mémoires de masse, ainsi que les programmes *PROG4* et *PROG5*, qui utilisent une mémoire de masse en tant que mémoire partagée, sont caractérisés par une exécution parallèle beaucoup plus efficace que le programme *PROG1*, qui réalise de manière itérative des communications hôte-APs et ne recourt pas à l'utilisation des mémoires de masse. Cette efficacité serait d'ailleurs renforcée, sauf pour le programme *PROG1*, si l'on faisait abstraction du temps d'initialisation des jobs, ce qui est l'optique retenue pour l'analyse de prédiction de performances, présentée au paragraphe 6.7.

6.6.2 Présentation des Scénarios

Deux types de scénarios ont été considérés:

- chaque scénario du premier type est constitué d'un job unique; chaque programme d'application a été exécuté avec les différents nombres d'APs qu'il pouvait supporter; les mesures ont été répétées deux fois, afin de permettre une estimation de la variabilité des temps d'exécution mesurés pour un même job [**P**],

- chaque scénario du deuxième type (47 au total) est composé de plusieurs jobs concurrents; deux à huit jobs concurrents ont été successivement considérés; à cause des pannes évoquées précédemment, et afin de maîtriser les contentions pour l'accès aux APs entre les jobs concurrents, seules des requêtes spécifiques ont été effectuées; on a pu ainsi distinguer deux classes de scénarios à jobs multiples, la classe où les jobs d'un même scénario ne rivalisent pas pour l'accès aux APs (car ils utilisent tous des APs différents) et celle où ils doivent se partager des APs communs; pour des impératifs de temps, nous n'avons malheureusement pas pu dupliquer les mesures.

L'analyse des résultats, qui suit, fait la distinction entre ces deux types de scénarios, et pour le deuxième type entre les deux classes de jobs concurrents. La simulation des scénarios à l'aide du modèle a comporté environ **50000 événements pour les scénarios à job unique** et **le million d'événements** a été dépassé lors de la simulation des scénarios constitués de 8 jobs concurrents.

6.6.3 Résultats Obtenus à Partir des Scénarios à Job Unique

Nous présentons ci-dessous l'ensemble des résultats obtenus lors de la campagne de mesures et à l'issue des simulations pour les scénarios à job unique. Les résultats sont regroupés par programme d'application, et pour chaque programme d'application, ils sont présentés en fonction du nombre d'APs attachés, noté *NAP*.

Afin de comparer les temps d'exécution mesurés avec ceux estimés à partir de la simulation du modèle, nous donnons, en plus de ces deux temps (exprimés en secondes), leur différence relative, appelée **erreur**.

Du fait d'une grande variabilité dans le temps d'initialisation du job pour un même nombre d'APs et d'un manque d'information pour en permettre une modélisation fidèle, nous n'avons pas simulé ce temps, mais l'avons considéré comme un nouveau paramètre d'entrée du modèle, que nous avons estimé à partir de la trace d'événements du gestionnaire de ressources. Par conséquent, nous avons défini une nouvelle quantité, appelée **erreur corrigée**, qui est en fait la différence relative des deux temps d'exécution, l'un mesuré et l'autre estimé, desquels on a déduit le temps d'initialisation du job. De plus, la trace du gestionnaire de ressources, de par sa conception, entraîne une erreur absolue, pouvant atteindre deux secondes, dans l'estimation du temps d'initialisation du job. Nous présentons donc l'erreur corrigée avec sa borne inférieure, sa valeur moyenne, et sa borne supérieure.

Analyse des Résultats du Programme *PROG1*

Les résultats obtenus pour le programme *PROG1* sont les suivants:

NAP	Mesure	Simulation	Erreur	Initialisation		Erreur Corrigee	
1	29.2525	29.4127	0.55	12	-4.60	0.93	7.14
1	29.2486	30.4127	3.98	13	0.95	7.16	14.19
2	39.0157	39.8972	2.26	19	-0.56	4.40	9.89
2	39.0116	39.8972	2.27	19	-0.54	4.43	9.92
3	50.9590	49.3516	-3.15	23	-9.00	-5.75	-2.25
3	47.6153	46.3516	-2.65	20	-7.91	-4.58	-0.99
4	67.8297	65.1015	-4.02	32	-10.12	-7.61	-4.96
4	64.4148	62.1015	-3.59	29	-9.10	-6.53	-3.82
6	102.9497	94.6333	-8.08	48	-16.65	-15.13	-13.56
6	98.5304	91.6333	-7.00	45	-14.48	-12.88	-11.23
8	135.1523	122.6654	-9.24	61	-17.95	-16.84	-15.70
10	163.0520	147.2049	-9.72	69	-17.72	-16.85	-15.96

Si l'on s'intéresse aux résultats des mesures tout d'abord, il est clair que leur variabilité augmente avec le nombre d'APs et ne peut pas être expliquée seulement par celle du temps d'initialisation. Cette variabilité a bien sûr un effet négatif sur la qualité de la simulation [**P**], qui se traduit par une croissance de l'erreur corrigée avec le nombre d'APs. Si cette erreur est acceptable jusqu'à 4 APs, elle devient conséquente ensuite et n'est alors pas entièrement justifiée par la variabilité des mesures.

Une autre cause est la modélisation fort simplifiée de tous les phénomènes qui se produisent sur l'hôte, et notamment le fait que l'on ait négligé les machines virtuelles esclaves (cf paragraphe 6.3.1). Or le programme *PROG1* réalise toutes les communications entre tâches parallèles par l'intermédiaire de la tâche maître, engendrant de nombreuses contentions entre machines virtuelles pour l'accès au CPU ainsi que de fréquentes communications entre elles. Le modèle de l'hôte ne peut malheureusement pas prendre en compte tous ces phénomènes, dont l'importance augmente avec le nombre d'APs. Il n'est donc pas surprenant que la simulation sous-estime le temps d'exécution du job, avec une erreur de l'ordre de 20% pour 10 APs.

Analyse des Résultats du Programme *PROG2*

Etudions maintenant les résultats du programme *PROG2*:

NAP	Mesure	Simulation	Erreur	Initialisation		Erreur Corrigee	
2	26.8817	27.1686	1.07	19	-8.03	3.64	18.70
2	27.8529	27.1686	-2.46	19	-17.09	-7.73	4.02
3	27.4617	27.9899	1.92	20	-5.58	7.08	23.65
3	27.9203	27.9899	0.25	20	-10.43	0.88	15.46
4	36.9033	37.3153	1.12	29	-6.60	5.21	20.45
4	38.1345	37.3153	-2.15	29	-17.95	-8.97	2.22
6	51.2166	51.5040	0.56	42	-6.97	3.12	15.67
6	55.0398	54.5040	-0.97	45	-13.91	-5.34	5.14

Les résultats de simulation sont ici bien meilleurs. La variabilité des mesures est moindre; elle est due essentiellement à celle du temps d'initialisation du job. L'erreur corrigée est acceptable quel que soit le nombre d'APs, d'autant plus que 0 est toujours compris entre ses bornes inférieure et supérieure.

Analyse des Résultats du Programme *PROG3*

Les résultats obtenus avec le programme *PROG3* sont les suivants:

NAP	Mesure	Simulation	Erreur	Initialisation		Erreur Corrigee	
2	98.2655	98.1390	-0.13	19	-1.40	-0.16	1.12
2	98.2796	98.1390	-0.14	19	-1.42	-0.18	1.10
3	79.6539	79.8642	0.26	23	-1.37	0.37	2.17
3	82.3524	81.8642	-0.59	25	-2.55	-0.85	0.91
4	78.6668	77.7640	-1.15	32	-3.99	-1.93	0.21
4	78.8958	77.7640	-1.43	32	-4.45	-2.41	-0.29

5	77.3420	75.7232	-2.09	36	-6.18	-3.92	-1.53
5	81.6223	80.7232	-1.10	41	-4.56	-2.21	0.25
6	81.7955	82.4550	0.81	46	-0.93	1.84	4.77
6	82.0834	81.4550	-0.77	45	-4.28	-1.69	1.03

Du fait d'une moindre importance du temps d'initialisation dans le temps total d'exécution, les résultats sont tout à fait satisfaisants. Dans tous les cas, l'erreur corrigée est inférieure à 6,5%.

Si l'on compare ces résultats à ceux obtenus avec le programme *PROG2*, on constate donc une diminution de l'erreur avec la durée du job simulé, tout du moins pour les programmes d'application utilisant le mode message. Ceci est de bon augure quant à la validité du modèle pour ce type de programmes.

De plus, la variation des temps d'exécution mesurés avec le nombre d'APs est entièrement reproduite par la simulation. Jusqu'à 5 APs, le temps d'exécution décroît, puis il commence à augmenter. De nouveau, ceci est dû à la granularité des tâches parallèles de calcul qui diminue avec le nombre d'APs jusqu'à un point tel que le surcoût induit par les communications devient prépondérant.

Analyse des Résultats des Programmes *PROG4* et *PROG5*

Si l'on analyse maintenant les résultats obtenus avec le programme *PROG4*:

NAP	Mesure	Simulation	Erreur	Initialisation		Erreur Corrigee	
2	23.9094	24.2276	1.33	19	-11.54	6.48	33.72
2	23.8824	24.2276	1.45	19	-11.13	7.07	34.65
4	34.3655	35.5279	3.38	30	3.03	26.63	64.25
4	38.5550	37.5279	-2.66	32	-26.83	-15.67	-0.49

et avec le programme *PROG5*:

NAP	Mesure	Simulation	Erreur	Initialisation		Erreur Corrigee	
2	23.4074	24.2364	3.54	19	-3.16	18.81	53.68
2	23.3960	24.2364	3.59	19	-2.96	19.12	54.19
4	34.0349	34.7087	1.98	29	-5.41	13.38	41.48
4	34.0382	34.7087	1.97	29	-5.46	13.31	41.37

on remarque que le temps d'initialisation du job est largement prépondérant et explique la large amplitude de l'intervalle contenant l'erreur corrigée. De ce fait, les résultats sont difficiles à interpréter. Même si, dans la plupart des cas, 0 appartient à cet intervalle, nous ne pouvons estimer raisonnablement la précision de la simulation. Il semble néanmoins que le simulateur soit moins précis que dans le cas des jobs utilisant le mode message, mais seulement 5 secondes sont simulées sur 25 à 30 secondes de temps total d'exécution.

6.6.4 Résultats Obtenus à Partir des Scénarios à Jobs Multiples

Les résultats sont présentés sous une forme analogue à celle des résultats des scénarios à job unique, les temps étant toujours exprimés en secondes. Cependant, seule la valeur moyenne de l'erreur corrigée est donnée ici, car les bornes inférieure et supérieure sont beaucoup moins faciles à déterminer. En effet, elles dépendent non seulement de la variabilité du temps d'initialisation du job considéré, mais également de celle du temps d'initialisation des autres jobs composant le scénario.

Nous ne décrirons pas les résultats obtenus à partir des 47 scénarios, mais seulement certains d'entre eux, sélectionnés pour leur représentativité. Nous analyserons successivement les résultats obtenus avec les scénarios à deux jobs, sans contention pour l'accès aux APs, ceux à deux jobs, rivalisant pour l'accès à des APs communs, puis ceux à cinq jobs avec contention, et enfin ceux à huit jobs avec contention.

Scénarios à Deux Jobs Sans Contention

Programme	NAP	Mesure	Simulation	Erreur	Initial.	Erreur Corrigee
PROG1	4	134.6184	112.9924	-16.06	52	-26.18
PROG1	4	134.4656	111.2140	-17.29	51	-27.86
PROG2	2	59.8923	60.0474	0.26	50	1.57
PROG1	6	111.1144	99.3080	-10.63	52	-19.97
PROG2	4	53.0683	53.3153	0.47	45	3.06
PROG3	3	99.2854	98.8642	-0.42	42	-0.74
PROG4	2	39.6001	39.8451	0.62	34	4.38
PROG2	3	42.5674	42.3258	-0.57	34	-2.82

Les erreurs corrigées sont proches de celles obtenues pour les mêmes programmes d'application dans le cas des scénarios à job unique. Toutefois, lorsqu'un job exécutant le programme *PROG1* fait partie du scénario, il semble avoir une légère incidence sur un autre job exécutant le même programme. En fait, même si les deux jobs ne rivalisent pas pour l'accès aux mêmes APs, ils rivalisent pour celui au CPU de l'hôte. Et comme le programme *PROG1* consomme beaucoup plus de temps CPU sur l'hôte que les autres programmes d'application, lorsque deux jobs exécutent *PROG1*, une forte contention se produit sur l'hôte, qui n'est malheureusement pas reflétée par le simulateur pour les raisons déjà exposées. Ceci est à l'origine de la légère augmentation de l'erreur corrigée relevée dans le premier cas (proche de 28%).

Scénarios à Deux Jobs Avec Contention

Programme	NAP	Mesure	Simulation	Erreur	Initial.	Erreur Corrigee
PROG1	8	142.8506	125.6416	-12.05	61	-21.02
PROG1	8	227.6702	191.1018	-16.06	65	-22.48

PROG1	6	114.6020	103.6333	-9.57	57	-19.04
PROG2	4	117.0193	104.4283	-10.76	54	-19.98
PROG2	6	77.0642	78.0565	1.29	64	7.60
PROG3	6	121.4800	120.6197	-0.71	70	-1.67
PROG2	3	31.1382	30.9874	-0.48	23	-1.85
PROG4	4	37.5108	37.5381	0.07	32	0.50

Les erreurs corrigées sont comparables à celles du cas précédent. Cela tend à prouver que la simulation reflète bien les problèmes de contention pour l'accès aux APs.

Cependant, si l'on compare le deuxième cas présenté ici avec le deuxième cas sans contention, une forte augmentation de l'erreur corrigée est remarquable pour *PROG2* dans le cas avec contention.

La raison en est toute simple: le job exécutant *PROG2* doit attendre que le job exécutant *PROG1* se termine car il rivalise pour l'accès à un AP commun et appartient à une classe de priorité inférieure. Du fait que le temps d'exécution est sous-estimé par la simulation pour le job exécutant *PROG1*, le temps d'attente du deuxième job en est d'autant réduit, et par voie de conséquence son temps d'exécution est lui aussi sous-estimé.

Scénarios à Cinq Jobs Avec Contention

Programme	NAP	Mesure	Simulation	Erreur	Initial.	Erreur Corrigee
PROG4	4	91.5649	90.5381	-1.12	85	-15.64
PROG5	2	91.9374	90.3820	-1.69	85	-22.42
PROG4	2	95.6310	95.3108	-0.33	90	-5.69
PROG1	8	180.9348	159.2111	-12.01	96	-25.58
PROG5	4	194.9853	170.5165	-12.55	100	-25.76

PROG1	4	113.3374	103.2849 (1)	-8.87	70	-23.20
PROG3	6	169.9940	172.0660 (2)	1.22	73	2.14
PROG1	1	203.2978	203.1526 (3)	-0.07	70	-0.11
PROG3	2	198.4642	218.0230 (5)	9.86	77	16.10
PROG4	2	202.9331	210.6503 (4)	3.80	78	6.18

Dans les deux cas présentés ci-dessus, les erreurs corrigées sont similaires à celles rencontrées auparavant et peuvent être justifiées principalement par la sous-estimation du temps d'exécution simulé des jobs exécutant le programme *PROG1*.

Cependant, le second cas est intéressant, car il met en évidence un ordonnancement différent des terminaisons de jobs, causé par un décalage entre temps mesuré et temps simulé.

Dans l'exemple présenté, le second job exécutant *PROG3*, que nous appellerons *JOB4*, se termine normalement avant le job exécutant *PROG4*, appelé *JOB5*. Or, la simulation considère que *JOB5* se termine avant *JOB4*.

L'explication est facile à obtenir à l'aide d'une comparaison rapide de la trace d'événements du gestionnaire de ressources et de celle du simulateur. Le second job exécutant *PROG1*, désigné par *JOB3*, appartient à la classe de priorité la plus forte, utilise l'AP numéro 5 et s'exécute. *JOB5* a la même priorité que *JOB3* et utilise les APs numéro 5 et numéro 6. Il est cependant obligé d'attendre que *JOB3* se termine avant de pouvoir accéder à l'AP numéro 5. *JOB4* est de priorité inférieure et utilise les APs numéro 6 et numéro 7. Comme il n'est pas en concurrence avec *JOB3*, il s'exécute également. *JOB3* se termine. *JOB5* est alors susceptible d'obtenir les APs dont il a besoin en fin de tranche de temps, et *JOB4* est, lui, susceptible de perdre ses APs en fin de tranche de temps, au bénéfice de *JOB5*, de priorité supérieure. En réalité, *JOB4* se termine avant la fin de la tranche de temps. De ce fait, *JOB5* se termine en dernier.

La simulation, quant à elle, décèle la fin de la tranche de temps sur l'AP numéro 6 avant que *JOB4* se termine, à cause d'une sur-estimation du temps d'exécution de ce job. A cet instant, *JOB4* perd le bénéfice de l'AP numéro 6 et se met en attente. Il ne reprendra son exécution que lorsque *JOB5* aura terminé, si bien que la simulation conclut à la terminaison de *JOB5* avant celle de *JOB4*.

Ce type de divergence entre réalité et simulation ne peut être évité. Certes, la probabilité de son occurrence peut être limitée en augmentant la précision de la simulation pour chaque job. Mais, elle ne sera jamais nulle, dans la mesure où il y aura toujours persistance d'une erreur entre temps mesuré et temps simulé, aussi infime soit-elle. Ajoutons néanmoins que ce problème ne s'est

rencontré qu'à deux reprises en 47 simulations. La probabilité de son occurrence ne semble donc pas être trop importante.

Scénarios à Huit Jobs Avec Contention

Programme	NAP	Mesure	Simulation	Erreur	Initial.	Erreur Corrigee
PROG5	2	125.8937	128.1637	1.80	84	5.42
PROG5	4	121.6024	122.6608	0.87	95	3.98
PROG4	2	123.1129	122.7202	-0.32	89	-1.15
PROG1	3	209.6445	209.5544	-0.04	99	-0.08
PROG2	2	234.8672	233.3026	-0.67	72	-0.96
PROG3	2	245.9801	248.6601	1.09	94	1.76
PROG4	4	235.8960	220.9294	-6.34	83	-9.79
PROG3	2	280.8734	283.0368	0.77	73	1.04
PROG5	2	96.2432	95.4164	-0.86	90	-13.24
PROG2	4	110.8513	108.8948	-1.77	100	-18.03
PROG4	2	110.9333	107.0102	-3.54	101	-39.49
PROG1	1	133.8270	124.5731	-6.91	98	-25.83
PROG5	4	133.8521	136.7354	2.15	106	10.35
PROG3	3	189.6644	188.9474	-0.38	102	-0.82
PROG3	2	237.6921	241.0547	1.41	90	2.28
PROG1	4	240.2224	238.7012	-0.63	95	-1.05

Avec huit jobs parallèles concurrents, les résultats de la simulation ne semblent pas se dégrader. Le premier cas est même meilleur que la plupart des cas rencontrés précédemment. Le deuxième cas montre une erreur corrigée proche de 40% pour le job exécutant *PROG4*, ce qui peut sembler fort important. Cependant, il faut relativiser cette erreur avec le fait que le temps d'initialisation de ce job est prépondérant et que l'incertitude associée à ce temps n'est sûrement pas négligeable, comparée au temps effectivement simulé.

6.6.5 Commentaires sur la Validité du Modèle

Les résultats obtenus pour les jobs exécutant le programme *PROG1* prouvent la faiblesse de la modélisation de l'activité de l'hôte et montrent que les approximations faites sont trop simplificatrices. De ce fait, la validité du modèle, pour des programmes semblables à *PROG1*, c'est-à-dire utilisant de nombreuses communications entre tâches parallèles par l'intermédiaire de la tâche maître, ou pour des programmes réalisant de longues sections de calcul sur l'hôte, est sujette à caution.

Par contre, les résultats obtenus pour les programmes *PROG2* et *PROG3* prouvent la validité du modèle pour la représentation des jobs utilisant les mémoires de masse en mode message, avec une précision accrue lorsque la durée du job simulé augmente.

En ce qui concerne les jobs utilisant les mémoires de masse en mode partagé, l'interprétation n'est pas évidente. Comme nous l'avons déjà souligné, les programmes *PROG4* et *PROG5* utilisés avaient un temps total d'exécution composé pour une part largement prépondérante du temps d'initialisation du job, non réellement simulé et empreint d'une forte incertitude. De ce fait, le temps effectivement simulé est dérisoire et l'écart constaté avec le temps mesuré pourrait provenir uniquement de l'incertitude sur le temps d'initialisation. Des jobs plus longs auraient pu pallier ce problème. Cependant, leur utilisation lors de la campagne de mesures aurait eu pour conséquence une réduction du nombre de cas étudiés.

Si l'on s'intéresse enfin aux résultats obtenus à partir des scénarios à jobs multiples, on se rend compte que l'erreur entre mesures et résultats de simulation est similaire à celle enregistrée avec les scénarios à job unique. Que ce soit avec ou sans contention pour l'accès à des APs communs, l'erreur reste stable avec un nombre croissant de jobs concurrents et s'établit entre 15% et 20%, tout en étant souvent amplifiée par la sous-estimation des temps d'exécution simulés pour les jobs exécutant le programme *PROG1*. Le modèle du système ICAP semble donc tout à fait valide pour la représentation de l'exécution de jobs en environnement non dédié, soit dans l'environnement normal du système ICAP.

Nous terminons maintenant l'étude de la validité du modèle en abordant les améliorations qu'il faudrait apporter au modèle pour une meilleure prise en compte des contentions pour l'accès au CPU de l'hôte et une réelle modélisation du temps d'initialisation des jobs.

Pour ce faire, une information très détaillée sur l'ensemble des machines virtuelles présentes sur l'hôte est requise. Les machines virtuelles esclaves doivent être modélisées ainsi que toutes les communications entre elles et la machine virtuelle maître.

Une autre machine virtuelle, que nous n'avons pas encore mentionnée, est le gestionnaire des machines virtuelles esclaves. Cette machine virtuelle a de nombreuses interactions avec le gestionnaire des APs et avec l'ensemble des machines virtuelles maîtres et esclaves. Dans notre modèle, nous l'avons purement et simplement ignorée, faute d'information suffisante et pensant que son activité était négligeable.

Il en a été de même du gestionnaire des travaux "batch", qui gère l'ensemble des jobs que les utilisateurs soumettent au système en mode "batch". Or, cette machine virtuelle fut utilisée pour tous les scénarios à jobs multiples, puisque son utilisation représentait le seul moyen de soumettre plusieurs jobs simultanés à partir d'un seul terminal. Comme toute autre machine virtuelle, le gestionnaire des travaux "batch" consomme du temps CPU, et entre donc en concurrence avec toutes les autres machines virtuelles présentes sur l'hôte pour l'accès au CPU. Elle communique en outre plusieurs fois avec le gestionnaire des machines virtuelles esclaves, notamment au moment de l'initialisation des jobs soumis en mode "batch".

Une analyse très détaillée de l'activité relative de chaque machine virtuelle doit être entreprise, afin de connaître quelles sont, parmi toutes ces machines virtuelles, les plus importantes, sur le plan des performances de l'hôte, et quelles communications entre ces machines virtuelles sont les plus coûteuses. Seuls ces éléments prépondérants seront ensuite intégrés au modèle actuel du système ICAP.

Comme toujours en évaluation de performances, la question-clé est donc: "Qu'est-ce qui est négligeable et qu'est-ce qui ne l'est pas ?". Cependant, même si l'on connaît la réponse, le problème se pose alors en ces termes: "Sachant que cet élément n'est pas négligeable, suis-je à même de caractériser de manière suffisamment présise son interaction avec les autres éléments du modèle et serai-je capable de fournir au modèle les paramètres d'entrée qu'il requiert pour cet élément ?".

Le lecteur peut donc mesurer toute la difficulté qui accompagnera la réalisation des améliorations énoncées précédemment, et se rendre compte que l'effort ainsi investi n'est cependant pas assuré d'être couronné de succès.

6.7 Prédiction des Performances du Système

Ce paragraphe montre comment le modèle du système ICAP peut être utilisé pour des analyses de prédiction de performances.

Rappelons tout d'abord ce que nous entendons par prédiction de performances. Prédire les performances d'un système consiste à en estimer les critères de performances, tels que temps de réponse, temps d'attente, taux d'occupation, lorsqu'une charge de travail spécifique lui est soumise ou lorsque certains de ses composants sont améliorés ou remplacés.

Ainsi que nous l'avons dit au paragraphe 1.1.7, l'objet de la prédiction de performances est d'abord de **mettre en évidence certains phénomènes**. Leur explication pourra, dans certains cas, nécessiter une **analyse ultérieure** que l'évaluateur de performances réalisera en collaboration avec les spécialistes du système et des applications incriminées.

Dans l'étude qui suit, toutes les raisons expliquant les phénomènes constatés ne sont donc pas systématiquement fournies. De plus, les explications détaillées auraient nécessité une présentation beaucoup plus fouillée à la fois du système ICAP et des programmes d'application utilisés. Nous avons délibérément préféré présenter au lecteur le genre de phénomènes, qui peuvent être révélés grâce à une analyse de prédiction de performances, plutôt que lui fournir une explication très détaillée de chacun d'entre eux. Cependant, dans de nombreux cas, leur explication est synthétisée en une ou deux phrases.

Dans le cas du système parallèle ICAP, la charge est divisée en deux composantes, **un job principal** et **plusieurs jobs d'arrière-plan**.

Le job principal est réellement le job auquel nous nous intéressons. Il est modélisé **très précisément et de manière la plus déterministe possible**.

Au contraire, les jobs d'arrière-plan représentent une charge additionnelle soumise au système. Chacun d'entre eux est modélisé **aléatoirement** suivant des lois de distribution de probabilité spécifiques.

La prédiction de performances est réalisée en deux étapes [**T**]. La première étape ne considère **que le job principal (sans charge additionnelle)**. La seconde étape prend en compte **le job principal et les jobs d'arrière-plan composant la charge additionnelle**. Les jobs d'arrière-plan sont générés de manière à refléter **la charge moyenne du système ICAP dans son environnement utilisateur habituel**.

A chaque étape, on fait varier plusieurs paramètres d'entrée, aussi bien dans le modèle des programmes que dans le modèle de l'architecture:

1. dans le modèle des programmes, en ce qui concerne le job principal, **la granularité des tâches parallèles** peut décroître en augmentant le nombre d'APs attachés au job [**E**]; **l'équilibrage de charge entre tâches parallèles** peut se dégrader par une augmentation du coefficient de variation de la loi de distribution représentant les sections de calcul AP,

2. dans le modèle de l'architecture, des extensions système, telles que l'addition d'une **deuxième mémoire de masse globale**, l'addition d'**APs** (jusqu'à 20) et de **mémoires de masse locales** (jusqu'à 10) supplémentaires, l'accélération de la **vitesse de calcul de l'hôte ou de celle des APs**, l'accélération de la **vitesse d'accès aux mémoires de masse** (par un facteur de 1 à 8), peuvent être prises en compte; des modifications du système, telles que l'augmentation ou la réduction de la **tranche de temps sur l'hôte ou sur les APs** sont étudiées également.

Bien sûr, cette liste de paramètres d'entrée variables ne se veut pas exhaustive et beaucoup d'autres alternatives peuvent être considérées de même que de nombreuses combinaisons de celles-ci. L'objectif de ce paragraphe est seulement de décrire le type d'analyse de performances que l'on peut mener à partir de ces quelques alternatives.

A chaque étape et pour chaque ensemble de paramètres d'entrée, plusieurs simulations (10 dans notre cas) sont réalisées afin d'obtenir des résultats qui garantissent une bonne représentation des lois de distribution de probabilité à partir d'une génération de nombres aléatoires, et, dans le cas de la seconde étape (avec charge additionnelle), qui reflètent bien le comportement du job principal dans l'environnement utilisateur habituel du système ICAP. Un nombre supérieur à 10 simulations aurait été préférable, mais le nombre de 10 nous a semblé un bon compromis à cause des contraintes de temps de simulation. L'analyse présentée ci-dessous a nécessité pas moins de 350 heures de calcul sur un micro-ordinateur IBM PS/2 de la Série 60, équipé d'un co-processeur arithmétique, chaque alternative de l'analyse avec charge conduisant à la simulation d'environ **5 millions d'événements en moyenne** contre seulement 100000 pour chaque alternative de l'analyse sans charge.

Les critères de performances calculés à l'issue de chaque simulation sont les suivants:

- le temps de réponse global du système pour l'exécution du job principal,

- le temps d'attente du job principal pour obtenir les APs qu'il requiert,
- le temps de calcul du job principal sur l'hôte et sur chaque AP,
- le temps des communications (attente exceptée) émises sur chaque AP par le job principal; ce temps est imputable soit à des communications entre la tâche maître et les tâches esclaves (utilisation des canaux hôte-APs) soit à des communications ou des synchronisations entre les tâches esclaves (au travers des mémoires de masse),
- le temps d'attente du job principal dû aux contentions pour l'accès au processeur de l'hôte ou pour l'accès aux mémoires de masse,
- le temps d'attente du job principal dû aux synchronisations des tâches parallèles sur l'hôte (directive WAIT) et sur chaque AP (directive BARRIER ou directive RECEIVE), défini comme suit:
  - pour chaque directive WAIT, ce temps d'attente est le temps séparant l'arrivée des résultats d'AProutine issus du premier AP et celle des résultats en provenance du dernier AP,
  - pour chaque directive BARRIER, ce temps d'attente est le temps séparant l'arrivée au point de synchronisation du premier AP et celle du dernier AP,
  - pour la directive RECEIVE, ce temps d'attente est le temps séparant l'instant où la directive RECEIVE a été exécutée et celui où elle est satisfaite (réception des données); ce temps est nul si les données sont déjà disponibles lorsque la directive RECEIVE est exécutée,
- enfin, le taux d'occupation de chaque élément du système (hôte, APs, canaux hôte-APs, mémoires de masse), par le job principal d'une part (noté *JOB0*), et par les jobs d'arrière-plan d'autre part (notés *Others*).

Ensuite, tous ces résultats sont moyennés sur l'ensemble des 10 simulations et sont présentés à l'utilisateur sous la forme suivante, tous les temps étant exprimés en secondes:

```
Mean results after 10 simulation runs
                    Response time :      90.573171
                  AP waiting time :      27.152775
```

Server	Computation	Communic.	Utilization		Contention	Synchro.
			JOBO	Others		
HOST	1.966000		0.022	0.189	1.389805	0.200281
AP 1	35.812123	0.614990	0.705	0.103	0.010731	22.080036
AP 2	35.535154	0.736691	0.705	0.103	0.046358	22.192819
AP 3	35.188717	0.621656	0.705	0.116	0.008144	22.626540
CH 1		0.499952	0.006	0.000		
CH 2		0.500005	0.006	0.000		
CH 3		0.500007	0.006	0.000		
BM 1		0.230076	0.003	0.000	0.027507	
BM 3		0.345114	0.004	0.000	0.076001	
BM 4		0.230076	0.003	0.000	0.039116	
BM 5		0.197208	0.002	0.000	0.017490	
BM 6		0.256518	0.003	0.002	0.030962	

La suite de ce paragraphe est subdivisée en cinq parties. La première partie décrit une nouvelle modélisation des programmes d'application, mieux adaptée à la représentation des jobs d'arrière-plan. La deuxième partie s'intéresse à la génération aléatoire des jobs d'arrière-plan. Les parties trois et quatre présentent et commentent les résultats, obtenus à l'issue des simulations, respectivement pour les étapes un et deux de la prédiction de performances. La cinquième partie conclut ce paragraphe en exhibant quelques résultats d'ordre plus général qui peuvent être tirés de cette analyse.

6.7.1 Nouvelle Modélisation des Programmes d'Application

Afin de représenter les jobs qui seront générés aléatoirement [**Q**], les structures de données définies précédemment (cf paragraphe 6.3.2) pour modéliser les programmes d'application (bien adaptées à une simulation conduite à partir de traces d'exécutions de programmes, et dès lors bien adaptées à une simulation déterministe) ont été légèrement modifiées. Les nombreux fichiers d'entrée (introduits au paragraphe 6.3.2) sont remplacés par de simples chaînes de caractères, que nous appellerons **chaînes de programmes**, décrites ci-dessous.

Le lecteur peut se référer à l'annexe A pour prendre connaissance de la structure générale du programme de simulation, écrit en Turbo Pascal. Seule ici est reproduite la structure de données associée à chaque programme d'application:

```
program_structure_type =
   record
      expected_time : double;
      master        : record
                      program_string : string[maxi_program_length];
                      nb_iterations  : array [1..maxi_nesting_levels]
                                       of distribution;
                      computation_1  : distribution;
                      computation_2  : distribution;
                      nb_arguments   : distribution;
                      code_size      : distribution
                      end;

      slave         : record
                      program_string : string[maxi_program_length];
                      nb_iterations  : array [1..maxi_nesting_levels]
                                       of distribution;
                      computation    : distribution;
                      case bulk_mode : bulk_mode_type of
                         shared  : (read_percent : 1..100;
                                    block_length : distribution;
                                    nb_blocks    : distribution);
                         message : (send_percent           : 1..100;
                                    message_length         : distribution;
                                    intermediate_transfers : distribution)
                      end
   end;
```

Le modèle de chaque programme d'application est composé de trois éléments principaux:

1. *expected_time* représente le temps d'exécution estimé du programme tel qu'il est fourni par l'utilisateur lorsqu'il soumet son job au système,
2. *master* modélise le programme maître,
3. et *slave* modélise le programme esclave.

Précisons que, dans l'implémentation actuelle du programme de simulation, le programme esclave ne peut être constitué que d'une seule AProutine. Cette restriction est essentiellement due au seuil infranchissable des 640 kilo-octets de mémoire accessible par le système d'exploitation PCDOS, sous lequel le programme tourne actuellement.

Le programme maître *master* est composé de:

- une chaîne de caractères qui peut être vue comme une représentation condensée du code source du programme maître (*program_string*),
- le nombre d'itérations à exécuter pour chacune des boucles présentes à chaque niveau d'imbrication (*nb_iterations*),
- deux types différents de sections de calcul hôte (*computation_1* et *computation_2*),
- le nombre de paramètres formels de l'AProutine esclave (*nb_arguments*),
- et la taille occupée par le code exécutable et les données du programme esclave (*code_size*).

Mis à part le premier élément, tous ces éléments sont donnés sous la forme d'une distribution de probabilité définie comme suit:

```
distribution =
   record
      case distribution_nature : distribution_type of
         standard    : (mean      : double;
                        coeff_var : double);
         uniform     : (min_unif    : double;
                        max_unif    : double;
                        is_discrete : boolean);
```

```
        discrete    : (min_discr : integer;
                       max_discr : integer;
                       proba     : proba_array_type)
    end;
```

Comme le lecteur peut le voir, les distributions de probabilité supportées par le programme de simulation sont:

- les distributions constantes (*standard* avec un coefficient de variation *coeff_var* nul),
- les distributions exponentielles (*standard* avec *coeff_var* égal à 1),
- les distributions hyper-exponentielles (*standard* avec *coeff_var* supérieur à 1),
- les distributions hypo-exponentielles (*standard* avec *coeff_var* inférieur à 1),
- les distributions uniformes (*uniform*) sur l'intervalle [*min_unif*,*max_unif*] entier ou réel (suivant que *is_discrete* est vrai ou faux),
- et les distributions discrètes (*discrete*) où *proba* est un tableau contenant les probabilités des différentes valeurs entières incluses dans l'intervalle [*min_discr*,*max_discr*].

De manière analogue, le programme esclave *slave* est composé de:

- une chaîne de caractères (*program_string*), représentation condensée du code source de l'AP-routine esclave,
- le nombre d'itérations à exécuter pour chacune des boucles présentes à chaque niveau d'imbrication (*nb_iterations*),
- le temps nécessaire pour exécuter chaque section de calcul AP (*computation*) lorsqu'un seul AP est attaché au job; si plus d'un AP est attaché au job, le temps requis pour exécuter chaque section de calcul AP est supposé être égal au temps nécessaire avec un seul AP divisé par le nombre d'APs attachés au job (cette approximation est notamment valide pour des programmes parallèles basés sur une décomposition de domaine),

- et, en fonction du mode d'utilisation des mémoires de masse, deux ensembles distincts d'éléments sont considérés:
  - si le programme utilise le mode partagé, trois éléments supplémentaires sont définis:
    1. *read_percent* représente la probabilité pour un accès à la mémoire de masse partagée d'être une lecture de données depuis la mémoire partagée plutôt qu'une écriture de données,
    2. *block_length* définit la longueur de chaque bloc de données transmis vers ou depuis la mémoire partagée,
    3. et *nb_blocks* indique le nombre de blocs non contigus de données à transférer (chaque bloc de données contiguës correspondant à un transfert physique de données),
  - si le programme s'exécute en mode message, trois éléments supplémentaires sont également définis:
    1. *send_percent* représente, pour chaque communication entre deux APs, la probabilité que l'expéditeur envoie son message avant que le récepteur ne se mette en attente de celui-ci,
    2. *message_length* indique la longueur du message à envoyer ou à recevoir,
    3. et *intermediate_transfers*, utilisé pour les jobs d'arrière-plan uniquement, définit le délai, exprimé en nombre de communications intermédiaires, séparant l'instant où le receveur se met en attente de la réception du message (respectivement l'instant où l'expéditeur envoie son message) et l'instant où l'expéditeur envoie le message attendu (respectivement l'instant où le receveur récupère le message déjà envoyé); pour être plus explicite, chaque fois que AP_i envoie un message à (respectivement demande à recevoir un message de) AP_j $(i \neq j)$, une directive *RECEIVE* en provenance de (respectivement une directive *SEND* à destination de) AP_i est mise en attente pour AP_j, pour une durée exprimée par un nombre aléatoire de communications intermédiaires entre AP_j et AP_k ($k \neq j$ quelconque); cependant, si l'une des communications intermédiaires satisfait la directive mise en attente, le délai est annulé et la communication a lieu immédiatement; la prévention d'occurrence d'étreinte fatale est, conformément à la règle [**R**] présentée au paragraphe 2.2.2, garantie à chaque génération aléatoire de communication entre APs par maintenance d'un graphe des processus en attente et vérification que la nouvelle communication ne va pas créer

un cycle dans ce graphe; si tel est le cas, la communication est rejetée et une autre communication est générée.

Notons que les éléments *nb_iterations*, *computation*, *block_length*, *nb_blocks*, *message_length* et *intermediate_transfers* sont des distributions de probabilité.

Décrivons maintenant la structure des chaînes du programme maître (*master.program_string*) et du programme esclave (*slave.program_string*).

Dans la chaîne du programme maître, les caractères ou groupes de caractères suivants peuvent être rencontrés:

- N indique que les mémoires de masse ne sont pas utilisées,
- S signifie que les mémoires de masse sont utilisées en mode partagé,
- $Mn[L|R|S|N]^n$ précise que les mémoires de masse sont utilisées en mode message; n représente le nombre de réseaux définis et pour chacun d'entre eux, un caractère supplémentaire détermine sa topologie (L signifie ligne double, R anneau double, S étoile et N réseau maillé complet),
- *C1* représente une section de calcul hôte de type 1,
- *C2* représente une section de calcul hôte de type 2,
- E indique le lancement de l'exécution de l'AProutine sur tous les APs attachés aux jobs,
- W représente le point de synchronisation où le maître attend que les tâches esclaves soient terminées,
- *(* indique le début d'une boucle,
- *)* indique la fin d'une boucle (l'imbrication des boucles est représentée par des paires de parenthèses imbriquées *(...(...)...)*).

Notons une autre limitation de l'implémentation actuelle du programme de simulation. Le maître ne peut lancer l'exécution d'une AProutine que sur tous les APs attachés au job. Symétriquement, il ne peut se mettre en attente que de tous les APs attachés au job. Cette restriction, due à un temps de développement du programme de simulation malheureusement trop court, n'est

cependant pas critique, car la plupart des codes utilisés à IBM Kingston utilise le mode de synchronisation maître-esclaves supporté par le programme de simulation actuel.

Pour illustrer notre propos, la chaîne du programme maître du job principal utilisé pour la prédiction de performances est la suivante: *M1LC1EWC2*, et elle s'interprète comme suit:

1. le mode message est utilisé,
2. un seul réseau est défini et sa topologie est une ligne double,
3. après une section de calcul hôte de type 1, l'exécution de la seule APRoutine est lancée sur tous les APs attachés au job,
4. le maître se met en attente de la fin d'exécution de l'APRoutine sur chacun des APs,
5. et pour finir, le maître exécute une section de calcul de type 2.

Dans la chaîne du programme esclave, les caractères ou groupes de caractères suivants peuvent être rencontrés:

- *C* représente une section de calcul AP,
- *(* indique le début d'une boucle,
- *)* indique la fin d'une boucle,
- en mode partagé uniquement:
  - *A* représente un accès à la mémoire partagée; la direction du transfert de données est déterminée par *read_percent*,
  - *R* représente une lecture depuis la mémoire partagée,
  - *W* représente une écriture vers la mémoire partagée,
  - *B* indique un point de synchronisation de type *BARRIER*; en d'autres termes, chaque tâche parallèle atteignant ce point doit attendre que toutes les autres tâches parallèles l'aient également atteint avant de pouvoir poursuivre l'exécution de l'APRoutine,
- et en mode message uniquement:
  - *T* représente une communication; la direction de la communication (*SEND* ou *RECEIVE*) est déterminée par *send_percent*; le correspondant est désigné aléatoirement; si la communication duale de celle-ci n'a pas encore eu lieu, elle est mise en attente pour une

durée exprimée en nombre de communications intermédiaires (ainsi que nous l'avons décrit précédemment); une prévention d'étreinte fatale garantit qu'aucun blocage ne sera la conséquence de cette communication [**R**],

- *S+* indique un envoi de message vers le voisin de droite s'il existe; si l'esclave n'en a pas, cette action est simplement ignorée,
- *S-* indique un envoi de message vers le voisin de gauche s'il existe; si l'esclave n'en a pas, cette action est simplement ignorée,
- *S=* indique un envoi de message vers n'importe quel esclave avec équiprobabilité; là encore, la communication duale est éventuellement mise en attente et une prévention d'étreinte fatale est effectuée [**R**],
- *R+* indique une réception de message en provenance du voisin de droite s'il existe; si l'esclave n'en a pas, cette action est simplement ignorée,
- *R-* indique une réception de message en provenance du voisin de gauche s'il existe; si l'esclave n'en a pas, cette action est simplement ignorée,
- *R=* indique une réception de message en provenance de n'importe quel esclave avec équiprobabilité; là encore, la communication duale est éventuellement mise en attente et une prévention d'étreinte fatale est effectuée [**R**].

Notons que l'utilisateur du programme de simulation a la charge d'assurer la cohérence de toutes les communications entre APs, lorsqu'il a recours aux actions de type $S[+|-]$ et $R[+|-]$, car aucune communication duale n'est mise en attente dans ce cas.

A fin d'illustration, voici la chaîne du programme esclave du job principal utilisé pour la prédiction de performances: *C(CS-R+S+R-CS-S-R+R+S+S+R-R-C)*.

Elle exprime que chaque AP exécutant l'AProutine effectue tout d'abord une section de calcul AP, puis itérativement (pour un nombre de fois spécifié par *nb_iterations*):

1. effectue une section de calcul AP,
2. envoie un message à son voisin de gauche s'il en a un,
3. reçoit un message de son voisin de droite s'il en a un,
4. envoie un message à son voisin de droite s'il en a un,
5. reçoit un message de son voisin de gauche s'il en a un,

6. effectue une section de calcul AP,
7. envoie successivement deux messages à son voisin de gauche s'il en a un,
8. reçoit successivement deux messages de son voisin de droite s'il en a un,
9. envoie successivement deux messages à son voisin de droite s'il en a un,
10. reçoit successivement deux messages de son voisin de gauche s'il en a un,
11. et effectue une dernière section de calcul AP.

6.7.2 Génération Aléatoire des Jobs d'Arrière-Plan

Faisons tout d'abord l'inventaire des jobs d'arrière-plan représentant la charge moyenne du système lCAP.

Trois catégories principales de programmes sont considérées:

1. la première catégorie correspond aux programmes n'utilisant pas les mémoires de masse,
2. la deuxième catégorie contient les programmes utilisant les mémoires de masse en mode partagé,
3. et la dernière catégorie est constituée par les programmes utilisant les mémoires de masse en mode message.

A chaque catégorie de programmes sont associées **une chaîne du programme maître et une chaîne du programme esclave** bien spécifiques.

On tient également compte des **cinq classes de priorité** propres au système lCAP et définies en fonction du temps d'exécution prévisionnel des jobs. A chaque classe de priorité sont associées une **loi de répartition des nombres d'APs**, une **loi de distribution des interarrivées de jobs** et une **loi de probabilité régissant la catégorie d'appartenance du programme** exécuté par un job de la classe. En l'absence de données statistiques permettant une meilleure caractérisation du phénomène aléatoire des soumissions de jobs au système, nous avons considéré que ce phénomène est "poissonien" et donc la loi de distribution des interarrivées de jobs pour chaque classe est une **loi exponentielle**. Cependant, la moyenne de cette loi est fonction de la classe du job.

Compte tenu de ces trois catégories de programmes et des cinq classes de jobs, nous pouvons définir **quinze jobs d'arrière-plan génériques**, caractérisés par un mode d'utilisation des mémoires de masse spécifique (définissant aussi les chaînes de programmes maître et esclave), par une classe de priorité particulière (et, associé à cette classe, un temps d'exécution prévisionnel propre, défini par une **distribution**), par une moyenne spécifique de la loi exponentielle des interarrivées et par une loi de répartition particulière des nombres d'APs attachés. Ces jobs sont néanmoins génériques, car toutes les actions définies dans les chaînes de programmes maître et esclave, bien que fixées pour chaque catégorie de programmes, sont régies par des lois de distribution qui confèrent **à chaque instantiation du même job générique des paramètres différents** et donc un comportement différent.

En début de simulation (avec charge), l'instant d'arrivée d'un job d'arrière-plan est généré pour chaque classe de priorité en fonction de la loi de distribution des interarrivées de cette classe.

Lors de l'arrivée du job d'arrière-plan dans le système, l'instant d'arrivée du prochain job d'arrière-plan appartenant à la même classe est généré. Si le job principal n'est pas présent dans le système et si sa classe correspond à la classe du job d'arrière-plan arrivant dans le système, alors l'instant de la prochaine arrivée dans le système du job principal est généré conformément à la loi de distribution des interarrivées de la classe du job principal. La catégorie du programme que le job d'arrière-plan doit exécuter est définie conformément à la loi de répartition des catégories de programmes pour sa classe. Le nombre d'APs attachés au job est déterminé en fonction de la loi de répartition des nombres d'APs pour sa classe. En d'autres termes, dès son arrivée dans le système, le job d'arrière-plan (il en est d'ailleurs de même pour le job principal) se voit attribué tous les paramètres qui lui sont propres et qui dépendent soit de sa classe de priorité, soit de la catégorie d'appartenance du programme qu'il doit exécuter.

De par cette génération aléatoire des jobs d'arrière-plan et de leurs instants d'arrivée, nous pouvons **reproduire**, si les lois de distribution sont bien choisies, **la charge moyenne du système ICAP dans son environnement utilisateur habituel**. Par la méthode de génération des instants d'arrivée du job principal adoptée, nous garantissons **l'unicité de la présence du job principal dans le système** ou son absence à tout instant. De plus, nous sommes à même de simuler le comportement de ce job sous diverses conditions de charge du système et d'obtenir ainsi **les performances moyennes du job dans l'environnement de tous les jours du système ICAP**.

6.7.3 Analyse des Résultats des Simulations Sans Charge

Lorsqu'on simule à charge nulle, seul le job principal est pris en compte. A l'instant zéro, le job principal est soumis au système. Dès la fin de son exécution, il est soumis de nouveau et sans délai au système. Cette procédure est répétée afin d'obtenir les résultats de 10 simulations indépendantes garantissant à tout instant la présence unique du job principal dans le système. Les résultats des dix simulations sont alors moyennés afin de diminuer le biais induit sur les lois de distribution de probabilité par le générateur de nombres aléatoires.

Afin de permettre une comparaison avec les mesures réalisées pour la validation du modèle dans le cas des exécutions avec job unique, nous avons choisi pour job principal le job *PROG3*, qui présentait les résultats les plus valides.

Nous nous sommes tout d'abord intéressés à voir l'impact de la nouvelle modélisation des programmes d'application (qui est maintenant probabiliste alors que précédemment elle était basée sur des fichiers contenant la trace d'exécution du programme).

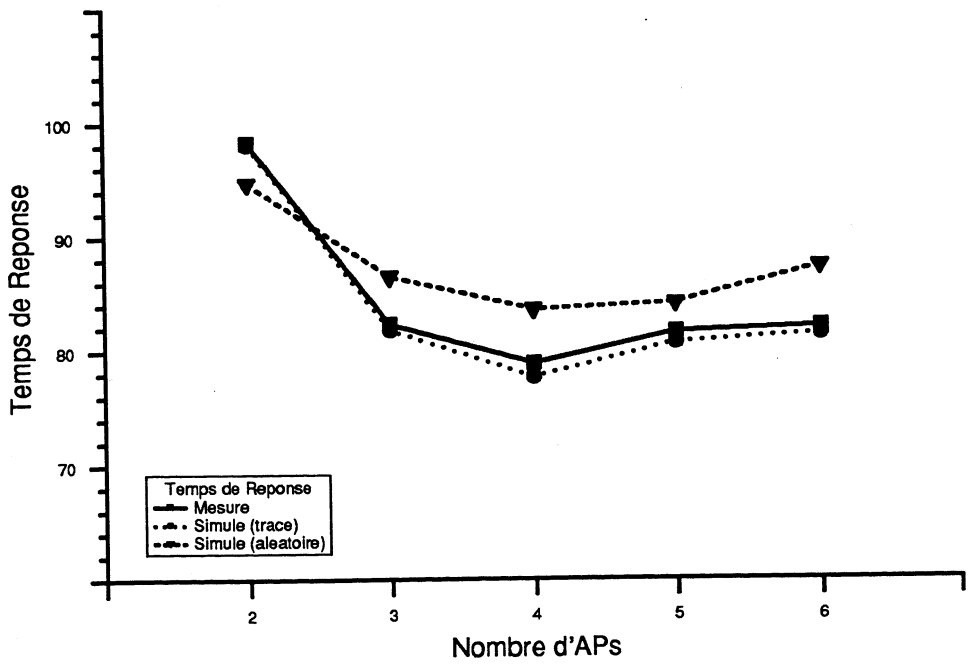

Figure 6.7: Comparaison des Temps de Réponse Mesurés, Simulés par Trace et Simulés Aléatoirement pour le programme *PROG3*

La figure 6.7 représente les temps de réponse du système (exprimés en secondes) mesurés, simulés par trace d'exécution et simulés aléatoirement, pour le programme *PROG3*, le nombre d'APs attachés variant de 2 à 6.

Il est certain que le décalage entre valeurs mesurées et résultats de simulation est plus grand lorsqu'on a recours à une simulation aléatoire que lorsqu'on utilise une trace d'exécution du programme. L'erreur relative est de l'ordre de 5% dans le premier cas alors qu'elle n'est que de 1% dans le second.

Cependant, l'allure générale de la courbe des temps de réponse mesurés est conservée et notamment, le nombre d'APs optimal pour assurer un gain en temps d'exécution est obtenu pour la même valeur dans les deux cas, à savoir 4 APs, en accord avec la valeur mesurée.

La figure 6.8, quant à elle, représente les mêmes résultats auxquels on a soustrait le temps d'initialisation du job qui est, faute d'information suffisante, trop simplement modélisé par un délai.

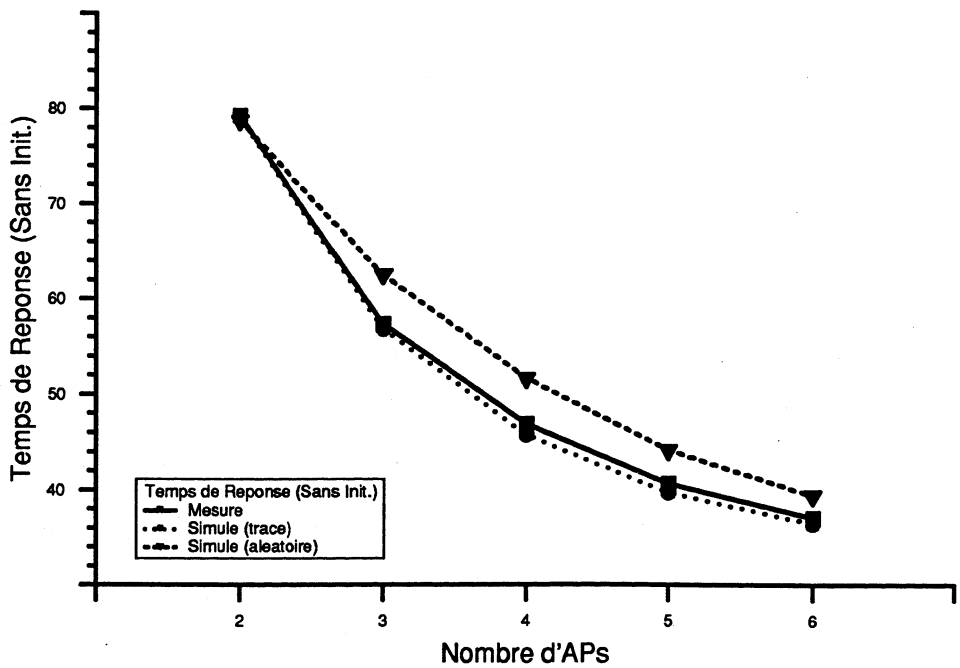

Figure 6.8: Comparaison des Temps de Réponse (Sans Initialisation) Mesurés, Simulés par Trace et Simulés Aléatoirement pour le programme *PROG3*

L'erreur relative est toujours prépondérante dans le cas de la simulation aléatoire avec 8% contre 2%. Cependant, l'allure générale de la courbe est beaucoup plus proche de celle obtenue à partir des valeurs mesurées et une tendance à converger pour les valeurs d'APs supérieures est décelable. De toute façon, une erreur relative inférieure à 10% est tout à fait acceptable.

Cette fois-ci, les temps obtenus décroissent régulièrement avec un nombre croissant d'APs, ce qui prouve l'importance du temps d'initialisation dans la durée totale de l'exécution de ce job. Il est clair que si le programme esclave du job principal comportait plus d'itérations, le temps d'initialisation pourrait être négligé devant la durée totale d'exécution.

Pour des contraintes de temps de simulation et afin d'étudier de nombreuses alternatives, nous avons volontairement limité ce nombre d'itérations à 100. Mais comme nous le verrons ultérieurement, il est cependant préférable de négliger ce temps d'initialisation pour une comparaison plus fine des différentes alternatives et une meilleure estimation du comportement d'un job plus long.

La suite de ce paragraphe présente (sous la forme d'histogrammes très expressifs) et commente les résultats obtenus lors de trois séries de simulations dont le dessein est respectivement d'étudier l'impact sur les performances du programme:

1. de la **granularité des tâches parallèles**; pour ce faire, on fait varier le **nombre d'APs attachés au job** de 2 à 20 (notons que les nombres d'APs supérieurs à 10 correspondent à des extensions du système ICAP), tout en divisant la durée de chaque section de calcul AP par le nombre d'APs considérés,

2. de l'**équilibrage de charge entre tâches parallèles**; pour ce faire, on fait varier le **coefficient de variation des tâches de calcul AP** de 1.032 (sa valeur mesurée) à 0 (équilibrage parfait),

3. d'**extensions système**, telles que:

   - l'addition d'une **deuxième mémoire de masse globale**,
   - l'accroissement de la **vitesse de calcul de l'ordinateur hôte** (par un facteur 2, 4 et 8),
   - l'accroissement de la **vitesse de calcul des APs** (mêmes facteurs),
   - l'accroissement de la **vitesse d'accès aux mémoires de masse** (mêmes facteurs).

Granularité des Tâches Parallèles

Plutôt que de produire des courbes comportant une information globale sur le temps de réponse (avec ou sans initialisation) du système pour exécuter le job principal, notre choix s'est porté sur une représentation par histogrammes (cf figures suivantes), qui ont l'avantage de pouvoir synthétiser sur la même figure et avec beaucoup plus de précision un nombre important de résultats de simulations.

Ainsi, la figure 6.9 représente le temps de réponse du système pour l'exécution du job principal pour un nombre croissant d'APs attachés (de 2 à 20 APs). Rappelons qu'un nombre supérieur à 10 APs correspond à une extension du système ICAP. Afin de maintenir la topologie du réseau d'interconnexion entre APs et mémoires de masse (en double anneau), nous avons augmenté en conséquence le nombre de mémoires de masse locales (6 pour 12 APs, 8 pour 16 APs et 10 pour 20 APs).

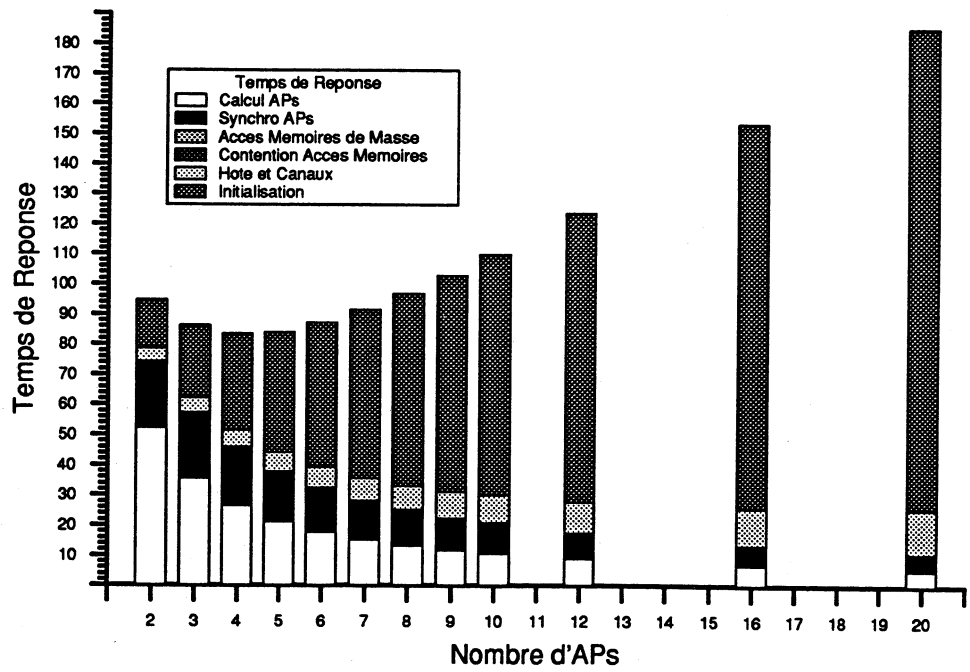

Figure 6.9: Influence de la Granularité des Tâches Parallèles: Temps de Réponse

Grâce à la représentation par histogramme, nous sommes à même de décomposer le temps de réponse en six composantes:

1. le temps moyen de calcul sur chaque AP,
2. le temps moyen d'attente par AP dû aux synchronisations avec d'autres APs (directives *RECEIVE*),
3. le temps moyen d'accès par AP aux mémoires de masse, représentant le temps de communication moyen par AP avec d'autres APs (attente exclue),
4. le temps moyen d'attente par AP dû aux contentions d'accès aux mémoires de masse (pour les communications avec d'autres APs),
5. le temps global d'exécution sur l'hôte et des communications hôte-APs,
6. et le temps d'initialisation du job.

Précisons que tous les temps sont exprimés en secondes, pour cet histogramme aussi bien que pour tous les autres histogrammes présentés dans cette analyse de prédiction de performances.

L'intérêt de cette représentation est une estimation visuelle de l'importance relative de chacune des composantes et une comparaison beaucoup plus aisée des différentes alternatives.

A l'aide de la figure 6.9, nous appréhendons toute l'importance du temps d'initialisation du job par rapport à son temps d'exécution total. Il devient, compte tenu du nombre limité d'itérations du programme esclave du job considéré, très vite prépondérant (à partir de 6 APs), limitant radicalement l'utilisation d'une petite granularité pour les tâches parallèles. Le lecteur pourra noter que l'on retrouve le nombre optimal de 4 APs pour un gain du temps d'exécution du job.

Toutefois, si l'on fait abstraction du fait que le job ne comporte que très peu d'itérations, on peut alors négliger le temps d'initialisation du job et obtenir l'histogramme de la figure 6.10, qui ne comporte que les 5 premières composantes de l'histogramme précédent. Le type de cet histogramme sera dénoté type A. Cet histogramme est très utile pour étudier une projection du job simulé (certes très court) sur un plus grand nombre d'itérations.

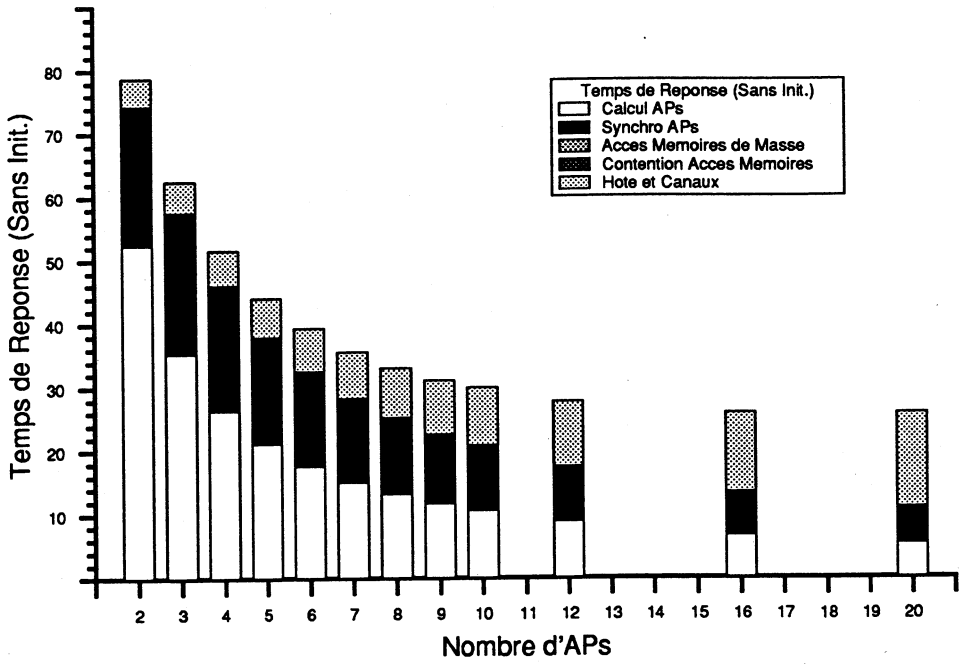

Figure 6.10: Influence de la Granularité des Tâches Parallèles: Temps de Réponse (Sans Initialisation)

Sur cet histogramme, les différentes composantes sont beaucoup plus faciles à comparer, exception faite de celles relatives aux mémoires de masse, qui sont trop petites pour apparaître et témoignent donc de l'importance très faible des communications entre APs dans le temps d'exécution global du job. On s'aperçoit que pour un nombre croissant d'APs, le temps moyen de calcul AP décroît ainsi que le temps moyen dû aux synchronisations. Cependant, la part relative des synchronisations tend à augmenter. Le principal phénomène est sans aucun doute l'augmentation constante du temps hôte-canaux, qui devient prépondérant à partir de 16 APs, et qui est la conséquence d'un gain infime en temps d'exécution global (sans initialisation) du job. On peut même être certain qu'un nombre d'APs supérieur à 20 conduirait à une perte de temps. On peut donc conclure que, pour le job principal seul, le nombre raisonnable d'APs à attacher est de 6 ou 8 et qu'il est vraiment inutile de prendre plus de 12 APs. Nous rappelons que cette conclusion fait bien sûr abstraction du temps d'initialisation, ainsi que l'ensemble des conclusions qui suivront dans cette analyse de performances.

On peut maintenant s'interroger sur les raisons du phénomène décelé précédemment et considérer un nouvel histogramme, dénoté de type B, et présenté à la figure 6.11.

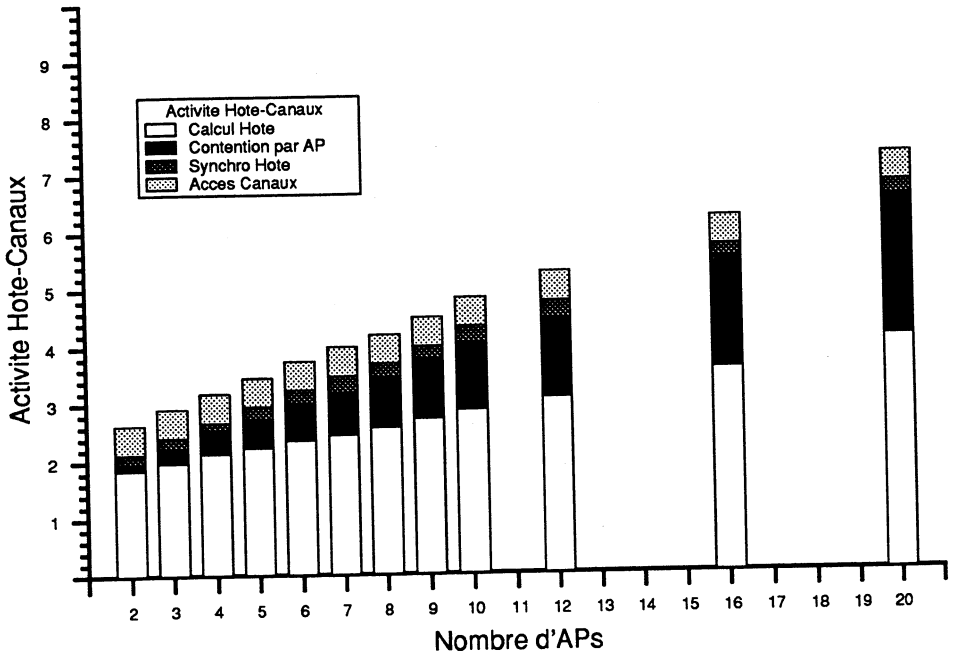

Figure 6.11: Influence de la Granularité des Tâches Parallèles: Activité Hôte-Canaux

Cet histogramme est constitué des quatre composantes suivantes:

1. le temps CPU consommé par le job sur l'hôte, regroupant le temps des sections de calcul hôte, mais également le temps CPU nécessaire au lancement de l'exécution de l'AProutine sur les APs, à la réception des résultats en provenance des APs,
2. le temps moyen d'attente par esclave dû aux contentions pour l'accès au CPU,
3. le temps d'attente dû aux synchronisations maître-esclaves en fin d'exécution de l'AProutine,
4. et le temps moyen de communication de chaque AP avec l'hôte (utilisation des canaux hôte-APs).

On se rend compte, à la vue de cet histogramme, que les composantes de l'activité hôte-canaux, qui augmentent avec le nombre d'APs de manière notable, sont: le temps dû aux contentions pour l'accès au CPU et le temps CPU consommé par le job sur l'hôte, dans une proportion plus marquée pour la première des deux. Ceci n'est pas pour nous étonner dans la mesure où le nombre de tâches entrant en compétition pour l'accès au CPU devient plus important lors du lancement de

l'AProutine sur les APs et de la réception des résultats en provenance des APs. Le temps moyen de communication hôte-AP ne varie pratiquement pas et le temps de synchronisation semble croître faiblement pour un nombre d'APs passant de 2 à 6, pour ne plus sensiblement varier ensuite. Le phénomène décelé à partir de l'histogramme de type A est donc expliqué.

Le lecteur aura sans doute constaté que les composantes relatives à l'accès aux mémoires de masse étaient, du fait de leurs petites valeurs, complètement masquées dans l'histogramme de type A. C'est pourquoi, nous considérons maintenant un troisième histogramme, dénoté de type C, restriction de l'histogramme de type A aux deux composantes relatives aux mémoires de masse (cf figure 6.12).

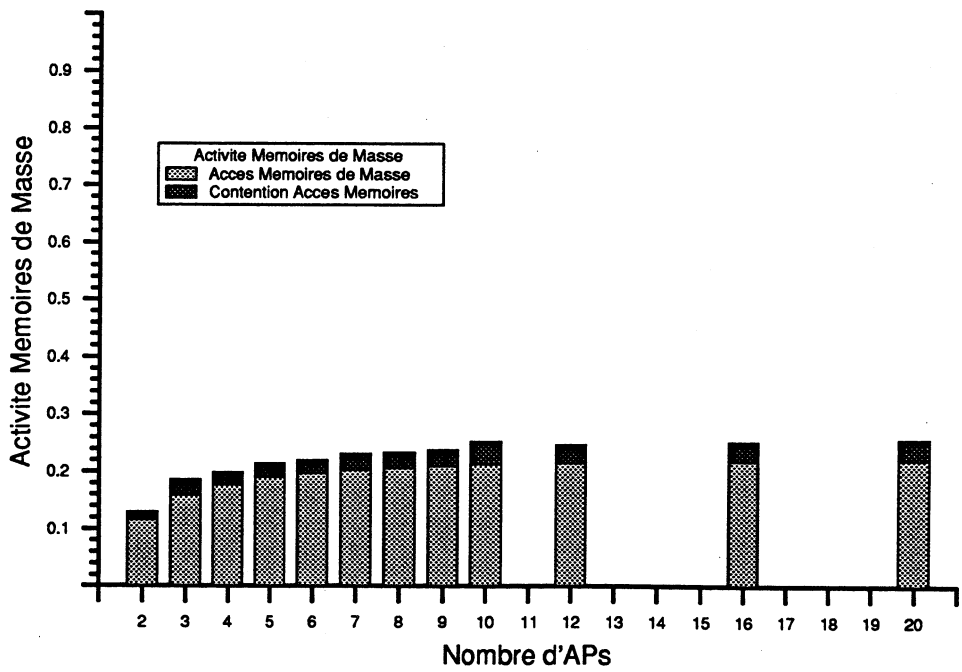

Figure 6.12: Influence de la Granularité des Tâches Parallèles: Activité Mémoires de Masse

Au vu de cet histogramme, on s'aperçoit que le temps moyen de communication par AP croît très faiblement avec le nombre d'APs pour se stabiliser dès que ce nombre atteint 10 APs. De très légères fluctuations sont notables pour le temps moyen d'attente par AP dû aux contentions d'accès aux mémoires de masse, avec peut-être une faible augmentation avec le nombre d'APs, mais rien de bien significatif.

Pour toutes les autres alternatives décrites précédemment (modification de l'équilibrage de charge, extensions système), nous présenterons les résultats de simulation sous la forme des trois histogrammes de types respectifs A, B et C. Les commentaires seront cependant plus succincts afin de mieux mettre en relief les points les plus importants.

Equilibrage de Charge entre Tâches Parallèles

L'histogramme de type A (cf figure 6.13) nous montre que le temps total d'exécution du job diminue avec un équilibrage de charge croissant, avec pour principale raison une réduction conséquente du temps moyen d'attente par AP dû aux synchronisations avec les autres APs.

Ce temps de synchronisation disparaît même tout à fait dans le cas de l'équilibrage de charge parfait (coefficient de variation des sections de calcul AP nul).

De plus, on peut remarquer qu'il est préférable, pour le job considéré, d'améliorer l'équilibrage de charge plutôt que d'augmenter la granularité des tâches parallèles, du moins à partir de 4 APs.

Le lecteur notera en outre que deux nombres sont portés sur l'axe des x, le deuxième figurant entre parenthèses. Ces nombres représentent respectivement le nombre d'APs attachés au job et le coefficient de variation des sections de calcul AP.

L'interprétation des nombres figurant sur l'axe des x des histogrammes suivants se fera de manière analogue.

A partir de l'histogramme de type B (cf figure 6.14), on voit qu'un gain sur l'hôte est également réalisé du fait d'une diminution du temps de synchronisation maître-esclaves avec un meilleur équilibrage de charge.

Cependant, ce gain en coût de synchronisation est bien inférieur au gain constaté, à partir de l'histogramme de type A, pour le coût de synchronisation entre esclaves, car le programme maître, contrairement à l'AProutine esclave, n'est pas itératif (voir les chaînes des programmes maître et esclave décrites au paragraphe 6.7.1). La synchronisation maître-esclaves n'a lieu par conséquent qu'une seule fois dans l'exécution du job, limitant ainsi le gain.

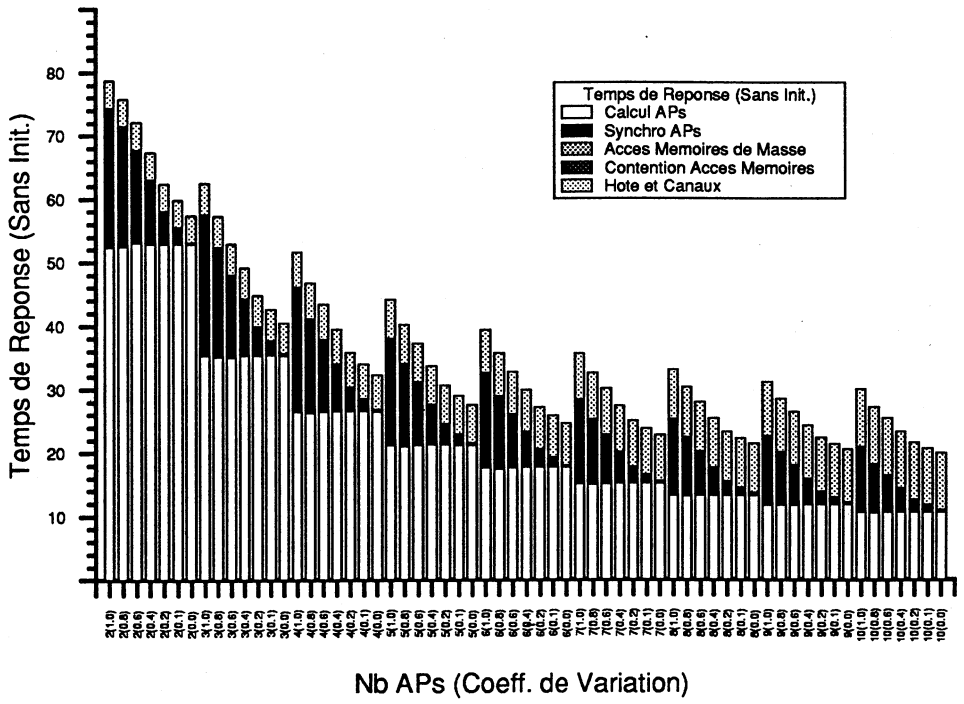

Figure 6.13: Influence de l'Equilibrage de Charge entre Tâches Parallèles: Temps de Réponse (Sans Initialisation)

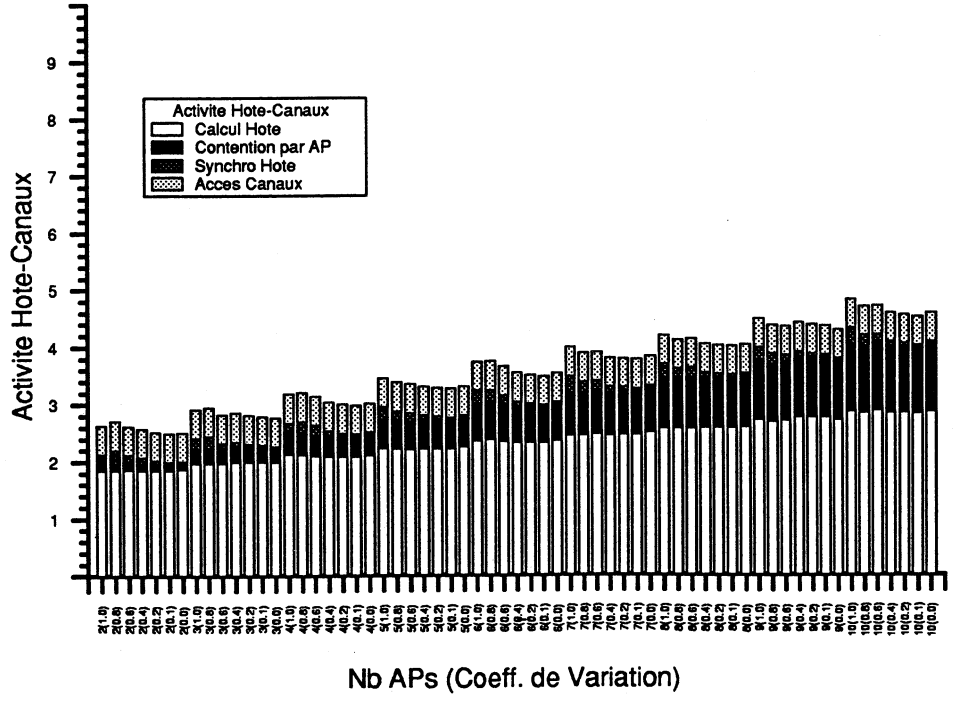

Figure 6.14: Influence de l'Equilibrage de Charge entre Tâches Parallèles: Activité Hôte-Canaux

En ce qui concerne l'activité des mémoires de masse (cf figure 6.15), un phénomène intéressant est notable. Le temps moyen d'attente par AP dû aux contentions pour l'accès aux mémoires de masse croît très faiblement avec une amélioration de l'équilibrage de charge, avec cependant une progression beaucoup plus marquée pour l'équilibrage de charge parfait. Cela nous conduit à la conclusion plus générale suivante: l'équilibrage de charge entre tâches parallèles est bénéfique pour des programmes nécessitant de nombreuses synchronisations entre tâches, cependant il peut générer une contention pour l'accès aux ressources partagées et pourrait donc être préjudiciable si le temps d'accès aux ressources partagées était moins performant ou les échanges de données entre APs plus volumineux.

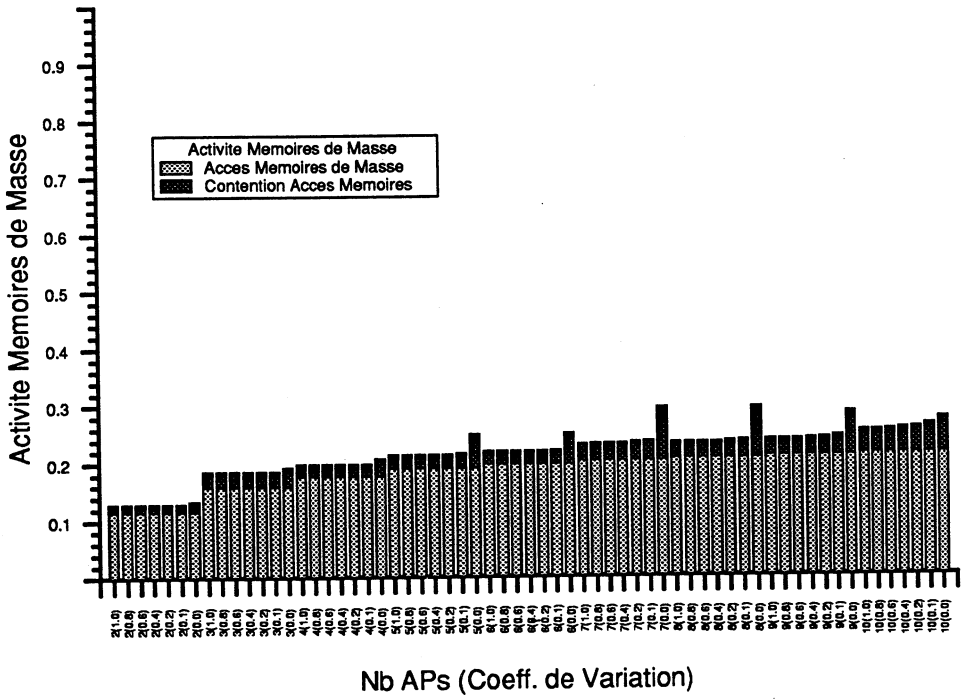

Figure 6.15: Influence de l'Equilibrage de Charge entre Tâches Parallèles: Activité Mémoires de Masse

Extensions Système

Addition d'une Mémoire de Masse Globale

Pour le job considéré, l'addition d'une mémoire de masse globale n'apporte pratiquement aucune variation dans les performances du job sur le système (cf figures 6.16 et 6.17).

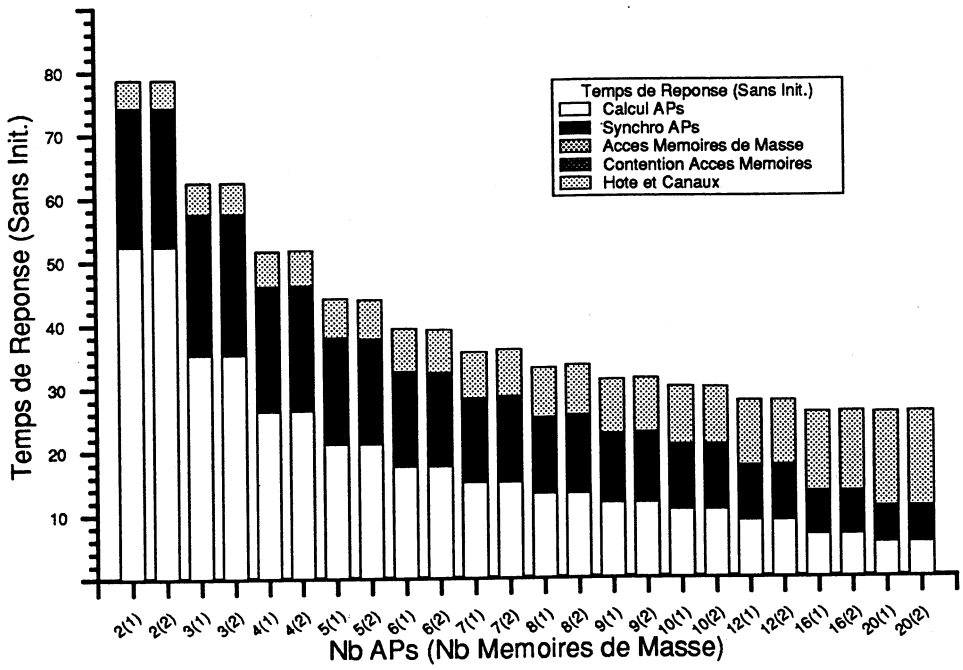

Figure 6.16: Addition d'une Mémoire de Masse Globale: Temps de Réponse (Sans Initialisation)

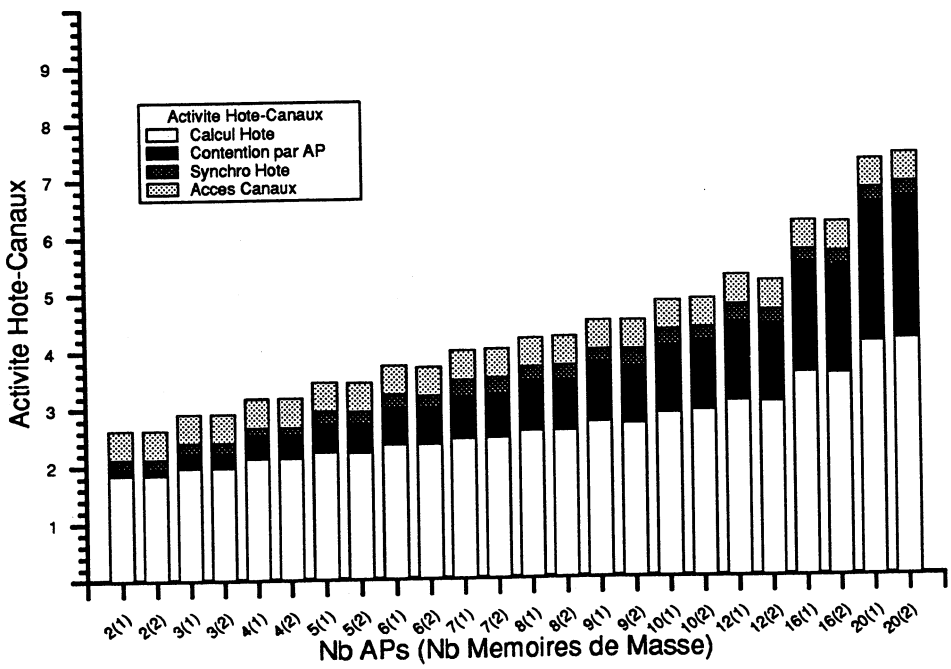

Figure 6.17: Addition d'une Mémoire de Masse Globale: Activité Hôte-Canaux

Toutefois, on observera une très légère réduction des contentions d'accès aux mémoires de masse (cf figure 6.18) due au nombre plus important de chemins de données entre APs (ceci est bien sûr propre au mode message).

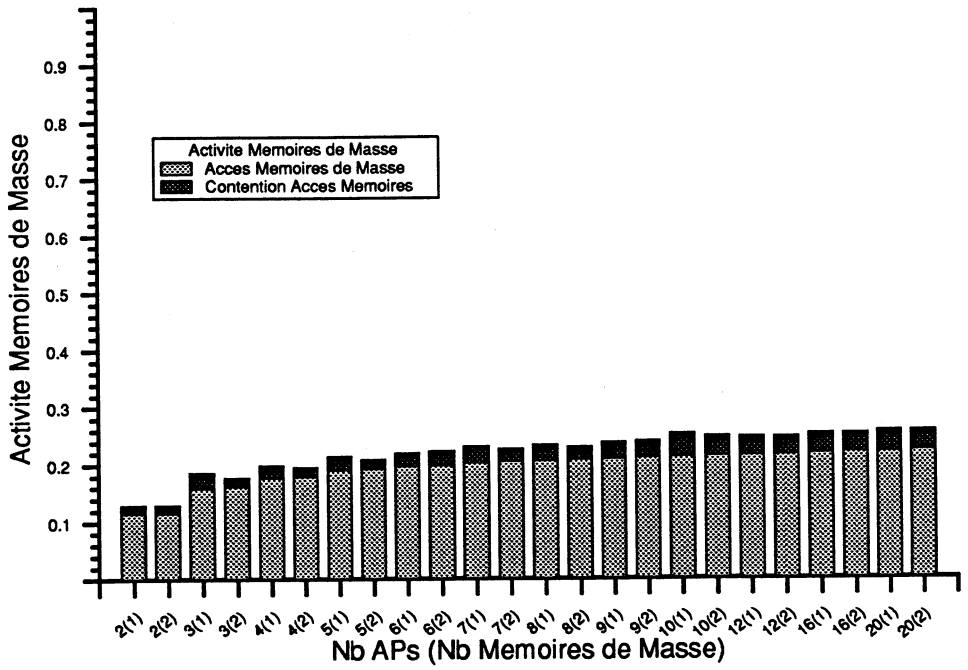

Figure 6.18: Addition d'une Mémoire de Masse Globale: Activité Mémoires de Masse

Accroissement de la Vitesse de Calcul de l'Hôte

L'accroissement de la vitesse de calcul de l'hôte n'a d'influence que sur l'activité de l'hôte (cf figure 6.19), par une réduction proportionnelle du temps CPU consommé par le job et du temps moyen d'attente des esclaves du fait des contentions pour l'accès au CPU (cf figure 6.20). L'accélération obtenue par l'utilisation d'un nombre croissant d'APs est régulière, quelle que soit la vitesse de calcul de l'hôte (cf figure 6.19).

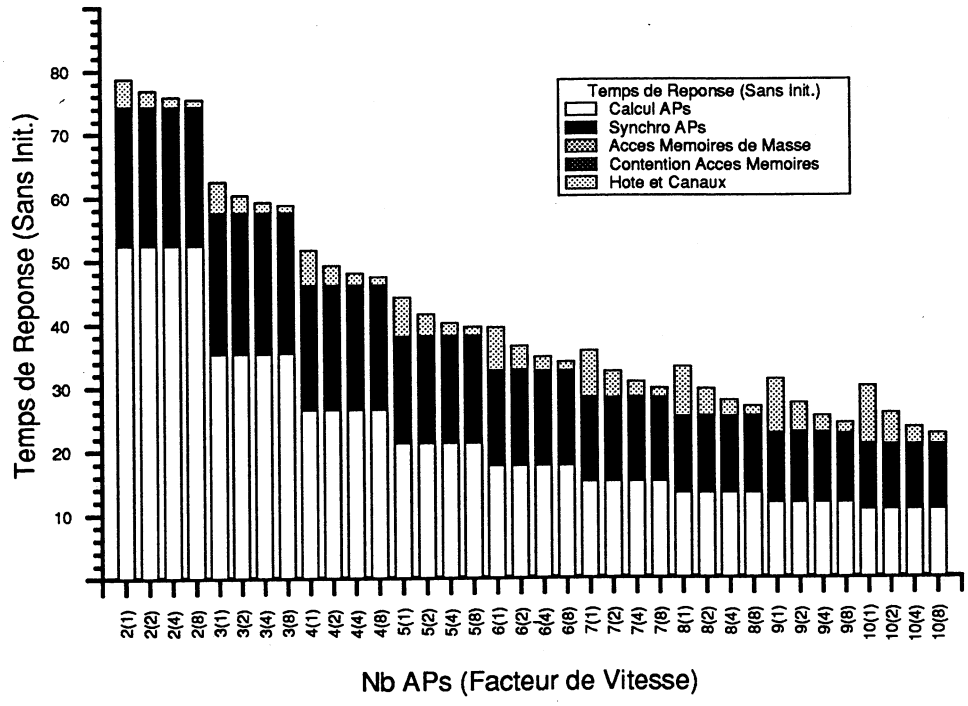

Figure 6.19: Influence de la Vitesse de l'Hôte: Temps de Réponse (Sans Initialisation)

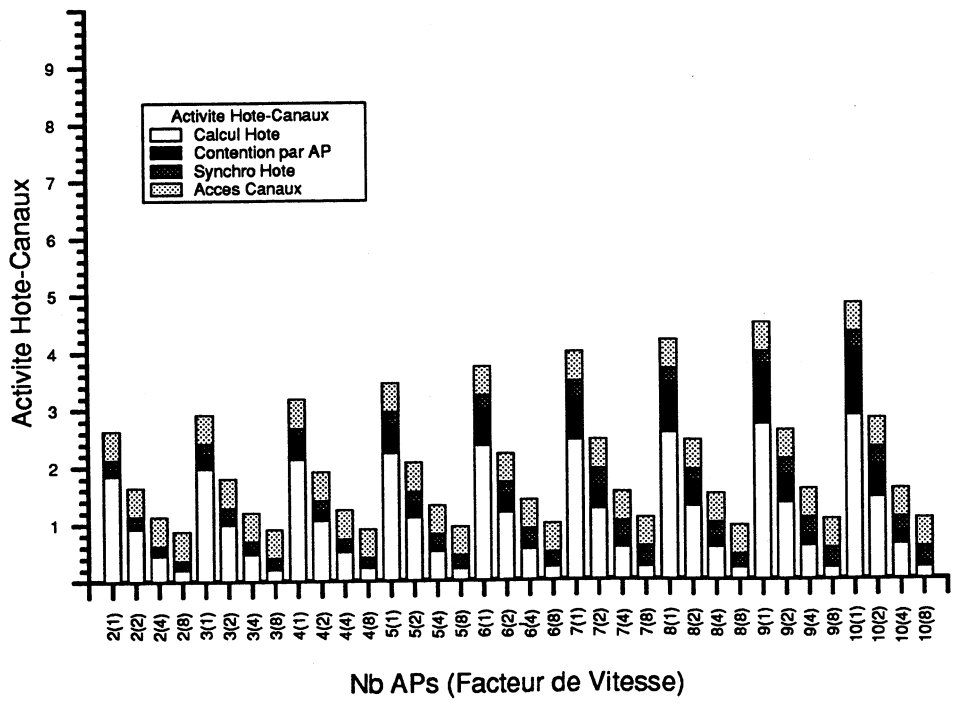

Figure 6.20: Influence de la Vitesse de l'Hôte: Activité Hôte-Canaux

Aucune influence n'est constatée au niveau des mémoires de masse (cf figure 6.21).

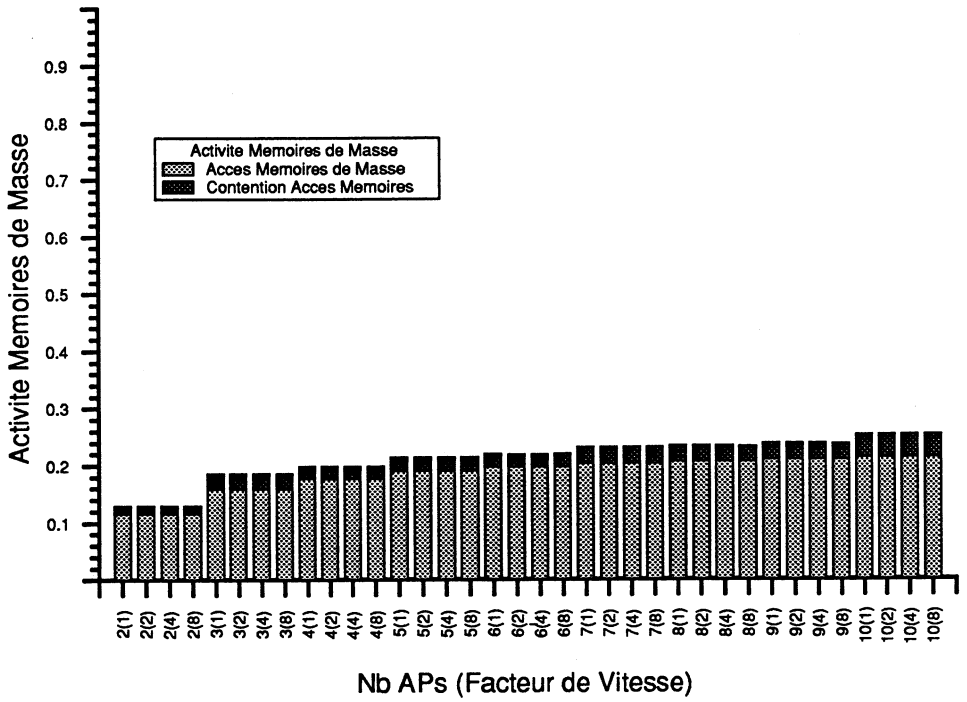

Figure 6.21: Influence de la Vitesse de l'Hôte: Activité Mémoires de Masse

Accroissement de la Vitesse de Calcul des APs

L'accroissement de la vitesse de calcul des APs a son influence bénéfique la plus marquée sur le temps de calcul AP et sur le temps de synchronisation entre APs, qui décroissent proportionnellement avec l'augmentation de la vitesse (cf figure 6.22).

Cependant, une amélioration est également sensible sur l'activité de l'hôte du fait d'une réduction du temps de synchronisation entre esclaves elle aussi proportionnelle à l'augmentation de la vitesse (cf figure 6.23).

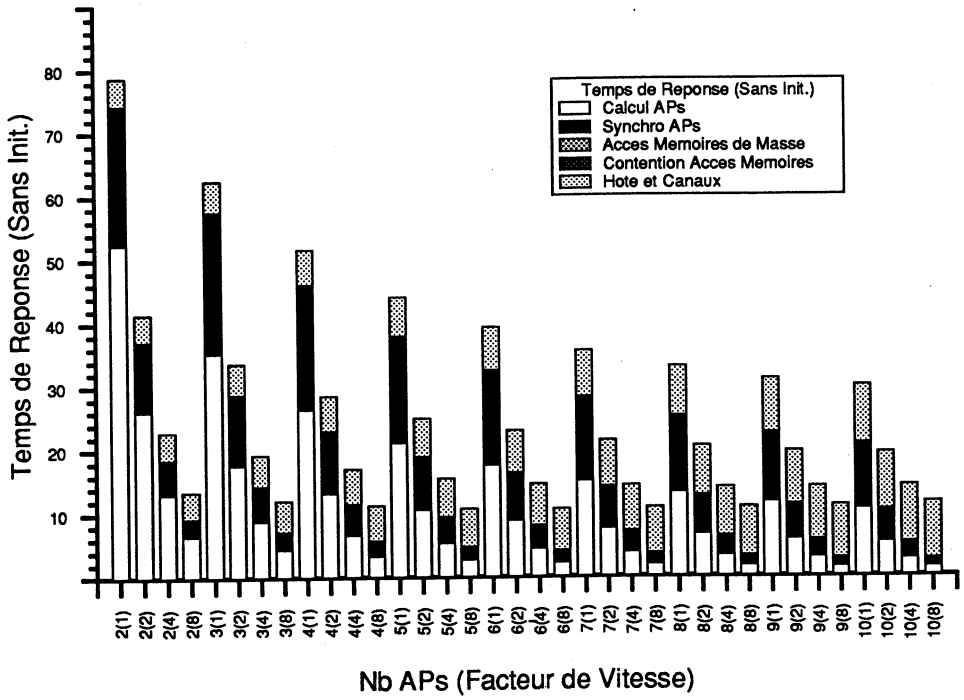

Figure 6.22: Influence de la Vitesse des APs: Temps de Réponse (Sans Initialisation)

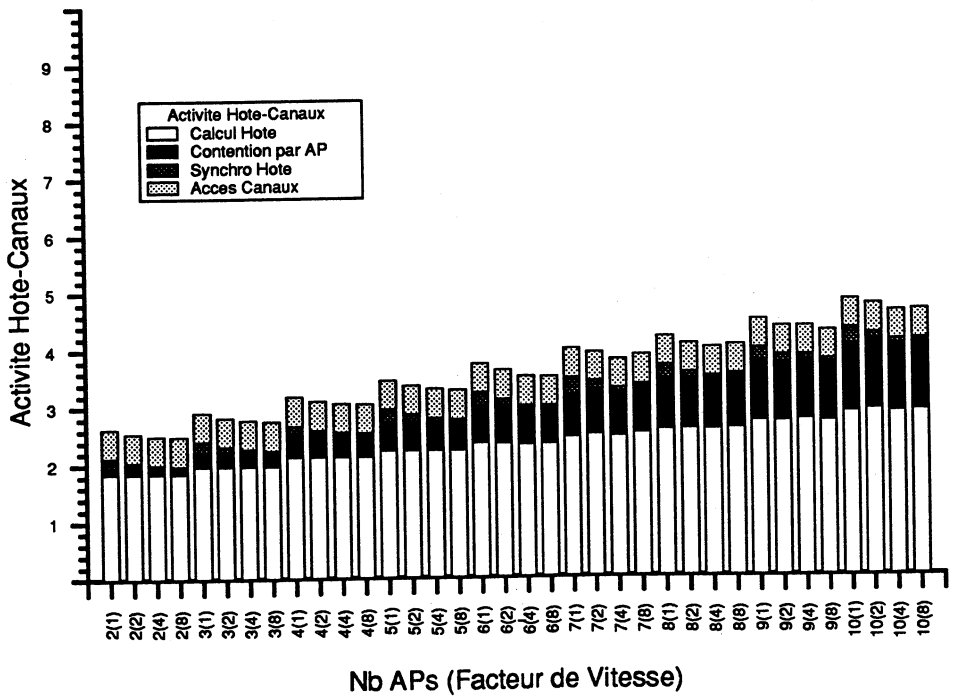

Figure 6.23: Influence de la Vitesse des APs: Activité Hôte-Canaux

Enfin, on constate une influence positive sur l'utilisation des mémoires de masse due à une réduction importante du temps de communication entre APs et à une diminution dans les mêmes proportions du temps moyen d'attente des APs du fait de contentions pour l'accès aux mémoires de masse (cf figure 6.24). Ces réductions sont attribuables à la diminution du temps d'initialisation de chaque communication, prépondérant pour de petits transferts (cf paragraphe 6.4.1).

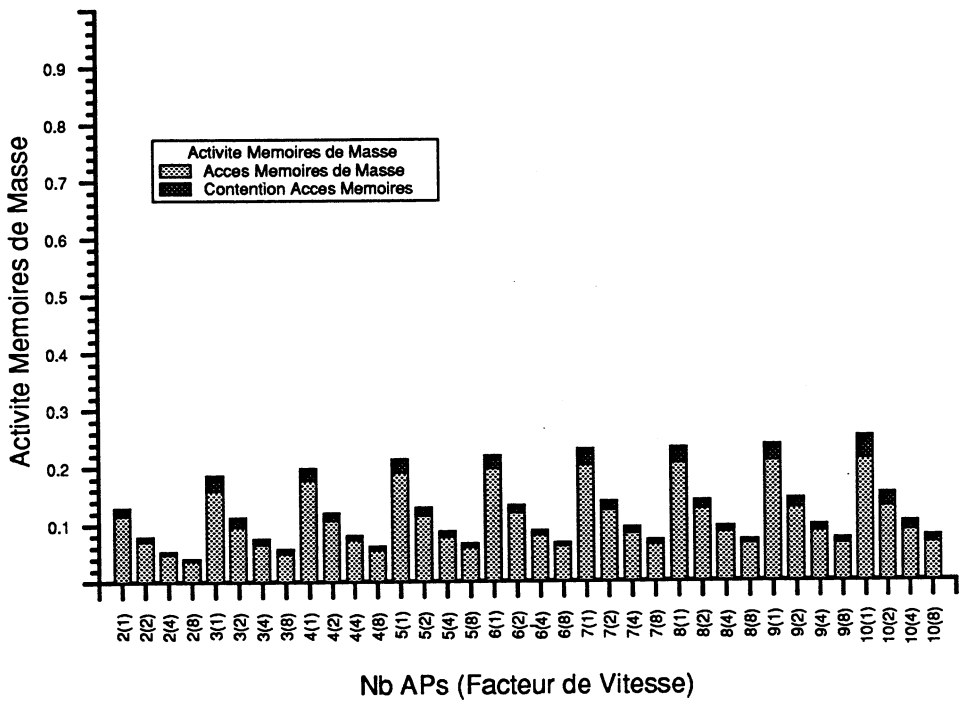

Figure 6.24: Influence de la Vitesse des APs: Activité Mémoires de Masse

Le fait d'augmenter la vitesse de calcul des APs semble être bénéfique sur tous les plans. Cependant, au vu de la figure 6.23, on pourra remarquer que l'accélération obtenue par l'emploi d'un nombre croissant d'APs n'est réelle que lorsque le facteur de vitesse reste inférieur à 4. En effet, pour les valeurs du facteur de vitesse de 4 et de 8, une décélération est même notable à partir de 7 APs. Il faut bien sûr rechercher la cause de ce phénomène dans l'activité de l'hôte qui croît en fonction du nombre d'APs utilisés (cf figure 6.24). Le gain procuré par la parallélisation sur le temps de traitement des APs est compensé par l'augmentation du temps de traitement sur l'hôte qui devient largement prépondérant pour de tels facteurs de vitesse.

Accroissement de la Vitesse d'Accès aux Mémoires de Masse

L'influence d'un accroissement de la vitesse d'accès aux mémoires de masse est très limitée dans la mesure où, comme nous l'avons souligné précédemment, l'importance de l'activité des mémoires de masse sur le temps d'exécution global du job considéré est infime (cf figure 6.25). Notons cependant que l'accélération obtenue par l'utilisation d'un nombre croissant d'APs, constatée à la figure 6.10, est toujours présente.

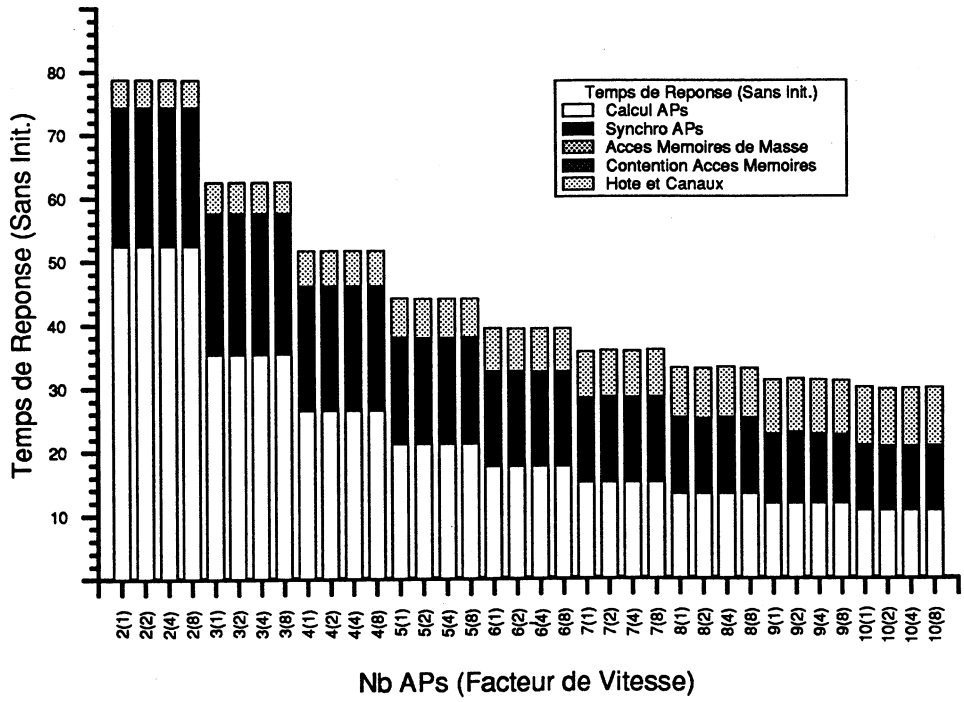

Figure 6.25: Influence de la Vitesse d'Accès aux Mémoires de Masse: Temps de Réponse (Sans Initialisation)

L'accroissement de la vitesse d'accès aux mémoires de masse n'a naturellement aucun effet sur l'activité hôte-canaux (cf figure 6.26), les fluctuations présentes étant sans aucun doute dues à la génération des nombres aléatoires.

Une très légère amélioration est sensible sur le figure 6.27, due à une réduction du temps de transfert des données entre les APs et les mémoires de masse. Cependant, ce temps de transfert étant court, pour de petits volumes de données, devant le temps d'initialisation de la communication, le gain enregistré ici est inférieur à celui constaté lors de l'accroissement de la vitesse de calcul des APs.

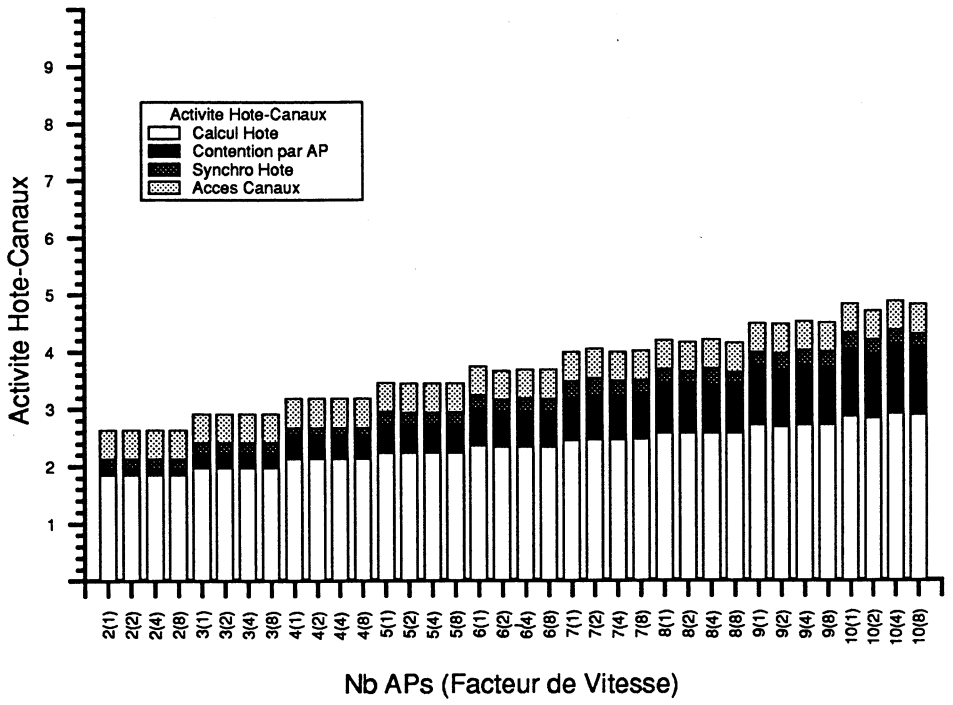

Figure 6.26: Influence de la Vitesse d'Accès aux Mémoires de Masse: Activité Hôte-Canaux

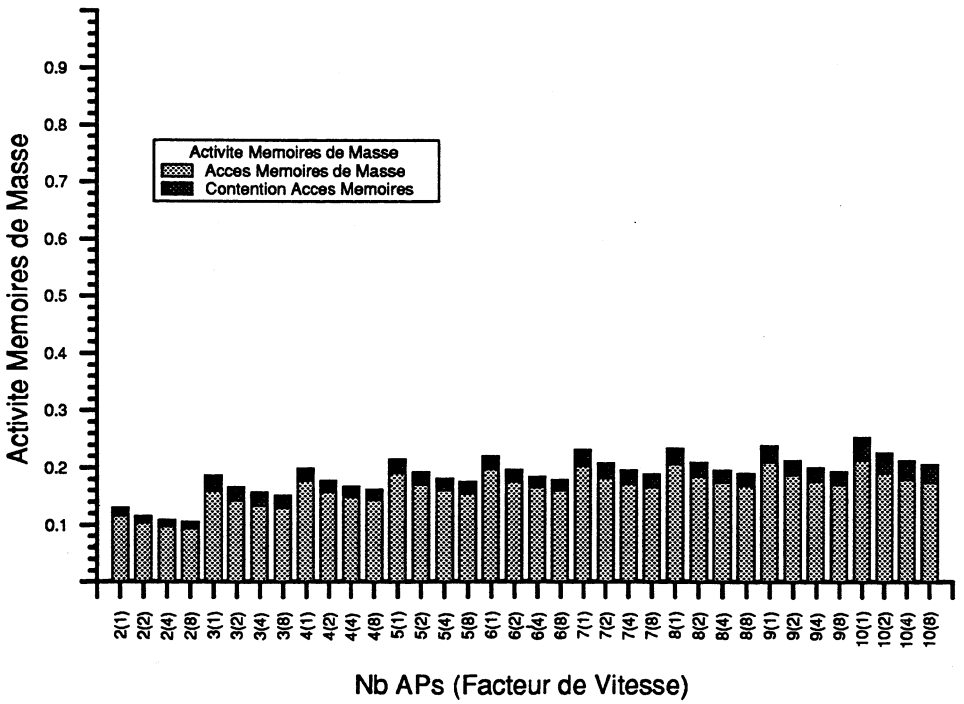

Figure 6.27: Influence de la Vitesse d'Accès aux Mémoires de Masse: Activité Mémoires de Masse

6.7.4 Analyse des Résultats des Simulations Avec Charge

Les simulations avec charge prennent en compte le job principal et l'ensemble des jobs d'arrière-plan. Ces derniers simulent la charge habituelle du système ICAP dans son environnement utilisateur. La génération aléatoire des jobs d'arrière-plan est réalisée conformément à la méthode exposée au paragraphe 6.7.2.

Les résultats des simulations sont, comme précédemment, présentés sous la forme d'histogrammes et concernent le job principal. Pour les mêmes raisons que lors des simulations sans charge, le temps d'initialisation du job principal n'est pas représenté. On retrouve ainsi les histogrammes de type A, B et C.

Cependant, pour les histogrammes de type A, une composante supplémentaire apparaît, appelée **contention pour l'accès aux APs**, et représente le temps d'attente moyen du job principal dû à l'occupation par un ou plusieurs jobs d'arrière-plan des APs qu'il doit utiliser. Cette composante, qui était bien sûr nulle pour les simulations sans charge, permet d'estimer l'influence de la charge externe sur l'accès aux APs.

De plus, afin de permettre une meilleure comparaison des résultats avec charge d'avec ceux sans charge, les histogrammes comportent deux parties. Dans leur partie gauche, on a reproduit les résultats sans charge, et dans leur partie droite, figurent les mêmes résultats, mais avec charge. Ces deux groupes de résultats sont facilement identifiables à partir des nombres d'APs portés sur l'axe des x, les nombres suivis de la lettre majuscule **C** correspondant aux résultats avec charge.

Un quatrième type d'histogramme, appelé type D, est également considéré. Il représente le taux moyen d'occupation des principales ressources, que sont l'hôte et les APs, par le job principal d'une part, et par l'ensemble des jobs d'arrière-plan d'autre part. Pour les jobs d'arrière-plan, les taux moyens d'occupation sont calculés sur les périodes de temps pendant lesquelles le job principal est présent dans le système. De plus, seuls les APs utilisés par le job principal sont pris en compte.

Ce type d'histogramme permet une meilleure évaluation de la concurrence entre le job principal et la charge représentée par les jobs d'arrière-plan et permet notamment une interprétation des temps moyens d'attente du job principal dûs respectivement aux contentions pour l'accès au CPU de l'hôte et à celles pour l'accès aux APs.

Seuls les taux moyens d'occupation de l'hôte et des APs sont représentés, car nous avons constaté que, pour le job principal étudié, les canaux hôte-APs et les mémoires de masse ne consti-

tuaient pas un goulot d'étranglement et que les contentions avec les jobs d'arrière-plan pour l'accès à ces ressources étaient presque inexistantes.

Comme pour les simulations sans charge, on a réalisé plusieurs séries de simulations, dont le dessein est d'étudier l'impact sur les performances du job principal, dans l'environnement utilisateur habituel du système lCAP:

1. de la **granularité des tâches parallèles**; pour ce faire, on fait varier le **nombre d'APs attachés au job** de 2 à 20 (notons que les nombres d'APs supérieurs à 10 correspondent à des extensions du système lCAP), tout en divisant la durée de chaque section de calcul AP par le nombre d'APs considérés,

2. de l'**équilibrage de charge entre tâches parallèles**; pour ce faire, on fait varier le **coefficient de variation des tâches de calcul AP** de 1.032 (sa valeur mesurée) à 0 (équilibrage parfait),

3. d'**extensions système**, telles que:

   - l'addition d'une **deuxième mémoire de masse globale**,
   - l'addition de **10 APs et 5 mémoires de masse locales supplémentaires** (afin d'obtenir une configuration en double anneau de 20 APs), augmentant ainsi le nombre d'APs disponibles pour l'ensemble des jobs,
   - l'accroissement de la **vitesse de calcul de l'ordinateur hôte** (par un facteur 2, 4 et 8),
   - l'accroissement de la **vitesse de calcul des APs** (mêmes facteurs),
   - l'accroissement de la **vitesse d'accès aux mémoires de masse** (mêmes facteurs).

4. de **modifications système**, telles que:

   - **la réduction ou l'accroissement de la durée de la tranche de temps sur l'hôte**,
   - **la réduction ou l'accroissement de la durée de la tranche de temps sur les APs**.

La suite de ce paragraphe présente et commente l'ensemble des résultats obtenus pour chaque série de simulations, en opérant une comparaison avec les résultats obtenus lors des séries similaires de simulations sans charge.

Granularité des Tâches Parallèles

Tout le problème de l'interprétation des résultats avec charge provient du fait que l'interférence des jobs d'arrière-plan avec le job principal est fonction des caractéristiques du job principal d'une part, mais également de la charge du système à l'instant précis de la soumission du job principal au système.

Bien que le job principal ait une priorité maximale, associée à sa classe, il peut se produire que l'instant d'arrivée du job nécessite l'attente d'une durée pouvant atteindre une minute (durée de la tranche de temps AP) avant l'accès aux APs qui lui sont destinés. Le job principal peut aussi être en concurrence avec un autre job de priorité maximale (donc non "rolloutable" par définition).

Tous ces facteurs peuvent entraîner, compte tenu du nombre restreint (10) de simulations du job principal avec les mêmes paramètres, de fortes variations du temps d'attente du job pour l'accès aux APs (cf figure 6.28), rendant plus difficile l'interprétation des histogrammes de type A. Pour ce type d'histogramme, on étudiera plus la tendance générale qui semble s'en dégager que chacun des cas représentés.

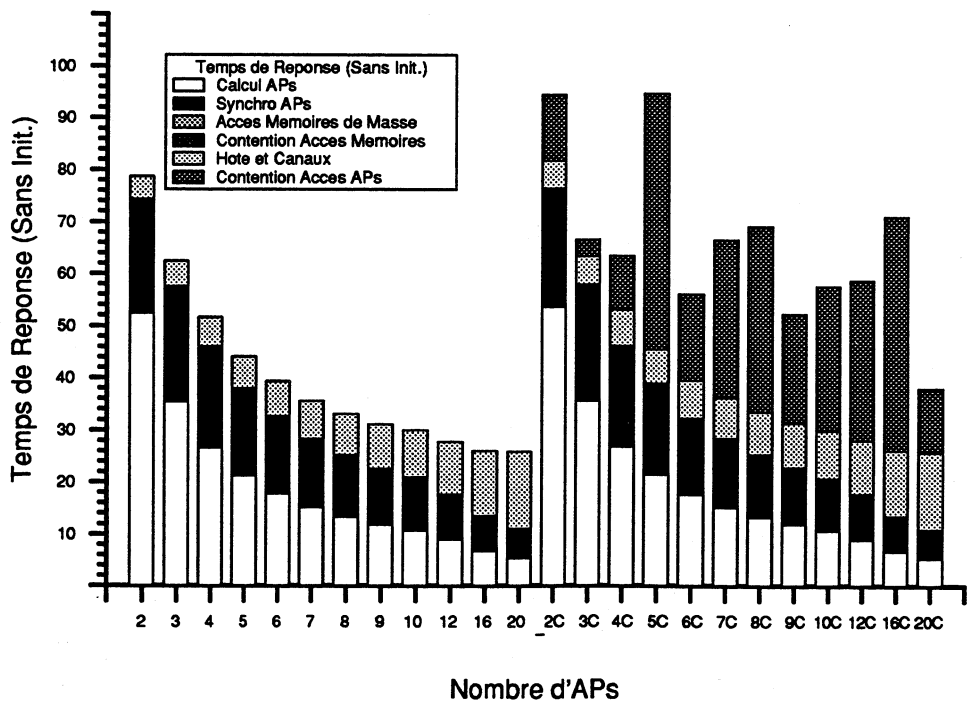

Figure 6.28: Influence de la Granularité des Tâches Parallèles: Temps de Réponse (Sans Initialisation)

Analysant l'histogramme de la figure 6.28, on remarque, en faisant abstraction du temps d'attente pour l'accès aux APs, que la forme générale de l'histogramme avec charge est identique à celle de l'histogramme sans charge.

Cependant, une légère augmentation de l'activité hôte-canaux peut être constatée, imputable à une plus grande attente moyenne par esclave pour l'accès au CPU de l'hôte, surtout sensible pour un petit nombre d'APs attachés au job principal (cf figure 6.29).

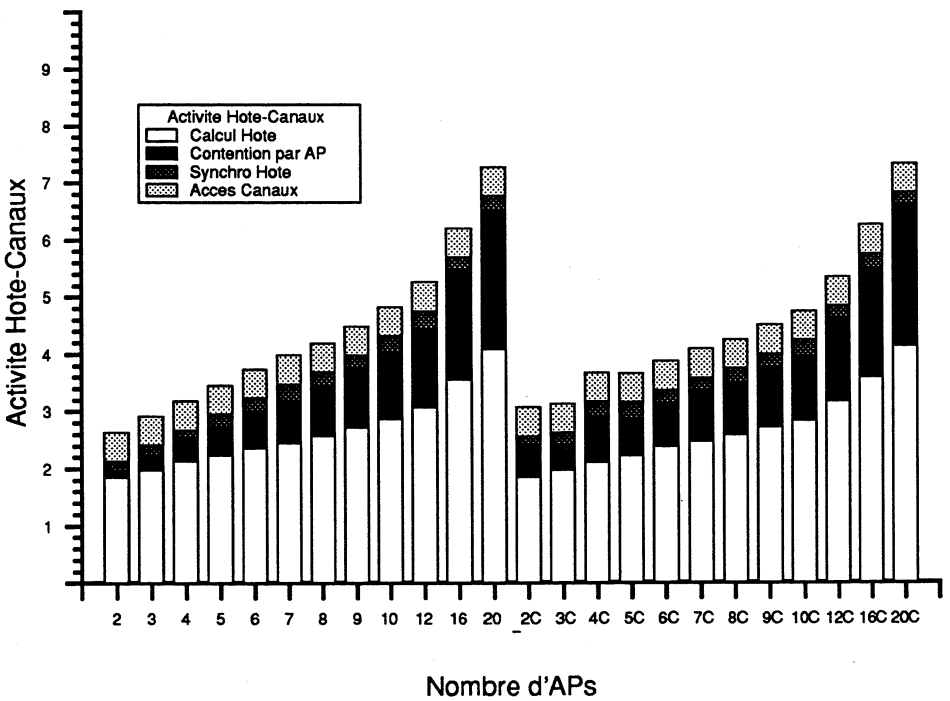

Figure 6.29: Influence de la Granularité des Tâches Parallèles: Activité Hôte-Canaux

Ce phénomène est explicable par le fait que cette attente est directement fonction du nombre de machines virtuelles en compétition pour l'accès au CPU de l'hôte. En effet, par le mécanisme du tourniquet pour l'attribution du CPU de l'hôte, l'influence des machines virtuelles esclaves des jobs d'arrière-plan sur chaque machine virtuelle esclave du job principal est en moyenne inversement proportionnelle au nombre de machines virtuelles esclaves du job principal, donc à la finesse de la granularité.

Si l'on s'intéresse maintenant au temps d'attente pour l'accès aux APs, il semble augmenter avec le nombre d'APs demandés, même si cette tendance n'est pas vérifiée pour chaque nombre d'APs, pour les raisons évoquées plus haut (cf figure 6.28).

L'histogramme de type D (cf figure 6.30) permet d'aboutir à la même conclusion, si l'on regarde la croissance du taux moyen d'occupation, par les jobs d'arrière-plan, de chaque AP attaché au job principal, avec le nombre d'APs utilisés par le job principal.

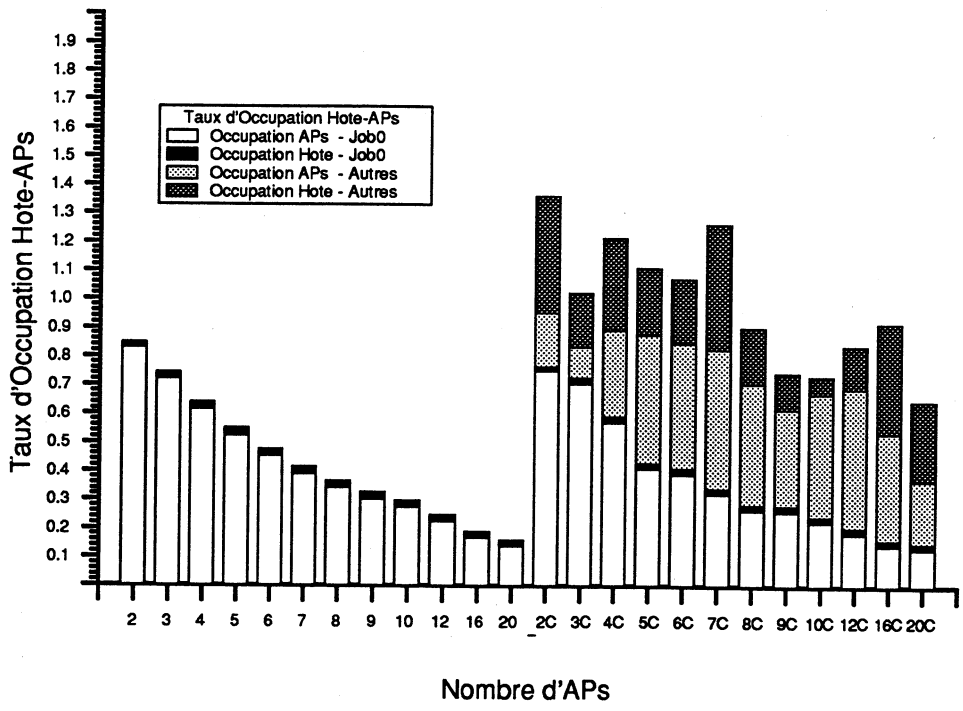

Figure 6.30: Influence de la Granularité des Tâches Parallèles: Taux d'Occupation Hôte-APs

Cette augmentation du temps d'attente limite le nombre d'APs pour lequel, temps d'initialisation du job principal exclu, une accélération du temps d'exécution est enregistrée. On peut considérer raisonnablement que 6 APs semblent être le nombre optimal d'APs à utiliser pour l'exécution d'un job identique au job principal, mais réalisant un nombre plus important d'itérations.

Notons également que les taux moyens d'occupation des APs et de l'hôte par le job principal avec charge sont légèrement inférieurs à ceux enregistrés dans le cas sans charge (cf figure 6.30), cette diminution provenant du temps d'attente du job principal plus important pour l'accès aux APs et au CPU de l'hôte dans le cas avec charge.

Quant à l'activité des mémoires de masse, aucune différence notable n'apparaît entre les résultats sans charge et ceux avec charge (cf figure 6.31).

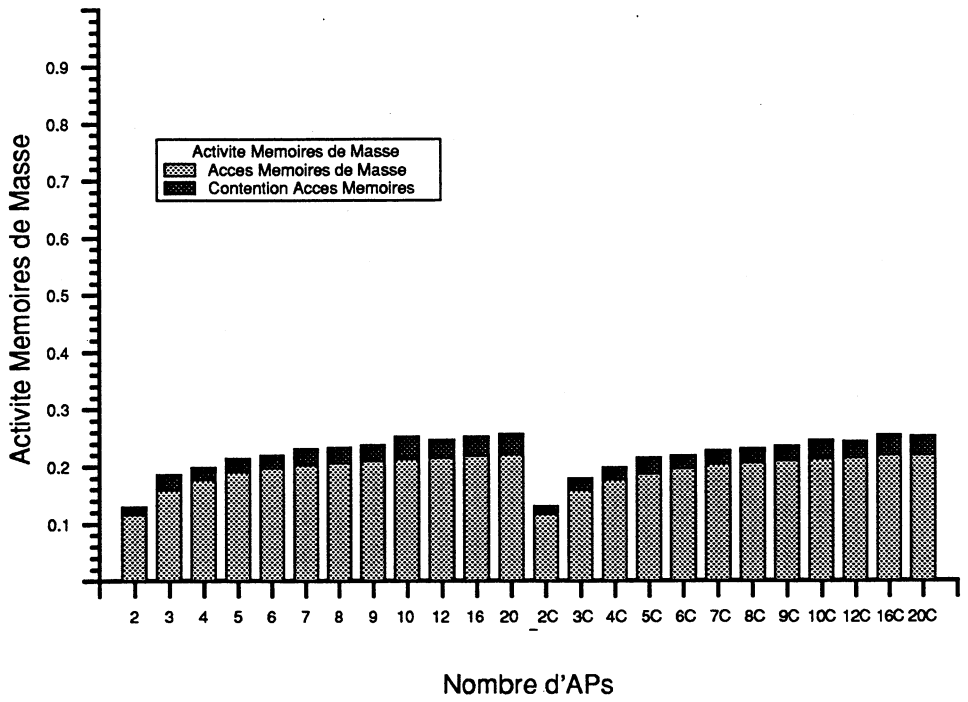

Figure 6.31: Influence de la Granularité des Tâches Parallèles: Activité Mémoires de Masse

Pour l'analyse des autres séries de résultats, nous ne présentons que les histogrammes révélant l'information la plus significative, afin de mieux insister sur les points importants de l'étude.

Equilibrage de Charge entre Tâches Parallèles

Au vu de la figure 6.32, on remarque une fois de plus l'identité entre la forme générale de l'histogramme avec charge (temps d'attente pour l'accès aux APs exclu) et celle de l'histogramme sans charge.

Un meilleur équilibrage de charge entre tâches parallèles réduit de manière significative les temps d'attente dûs aux synchronisations entre APs.

Cependant, ce gain peut être compensé par le temps d'attente pour l'accès aux APs et ne plus être constaté par l'utilisateur ayant soumis le job principal.

Il faut néanmoins ajouter que ce gain est bénéfique, globalement pour l'ensemble des jobs présents sur le système, grâce à la réduction pour le job principal du temps nécessaire de possession des APs, que peuvent donc utiliser plus rapidement les autres jobs.

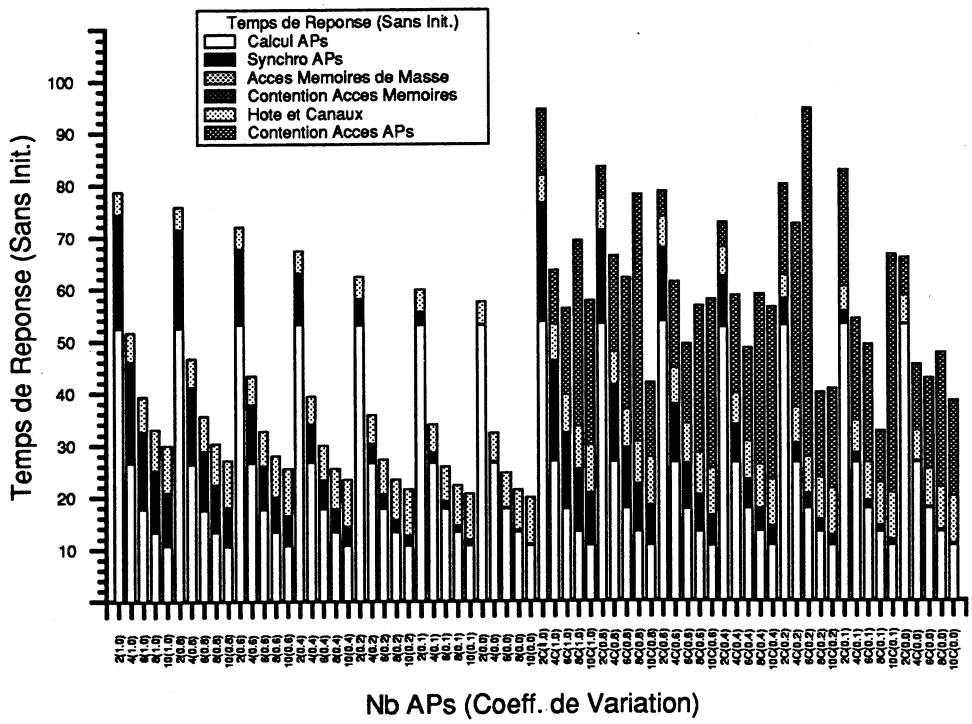

Figure 6.32: Influence de l'Equilibrage de Charge entre Tâches Parallèles: Temps de Réponse (Sans Initialisation)

On peut noter de nouveau que le nombre d'APs qui optimise le temps d'exécution du job principal (temps d'initialisation exclu) ne varie pas avec l'équilibrage de charge et semble toujours être de 6 APs.

Extensions Système

Addition d'une Mémoire de Masse Globale

L'addition d'une mémoire de masse globale ne contribue pas plus que dans le cas sans charge à une réduction sensible du temps d'exécution (cf figure 6.33).

Comme précédemment, on peut souligner que les communications entre APs sont loin d'être suffisantes, même en présence des jobs d'arrière-plan, pour provoquer un goulot d'étranglement au niveau de la mémoire de masse globale, que l'addition d'une seconde mémoire de masse globale viendrait atténuer. L'activité des mémoires de masse est en tout point identique qu'il y ait une ou

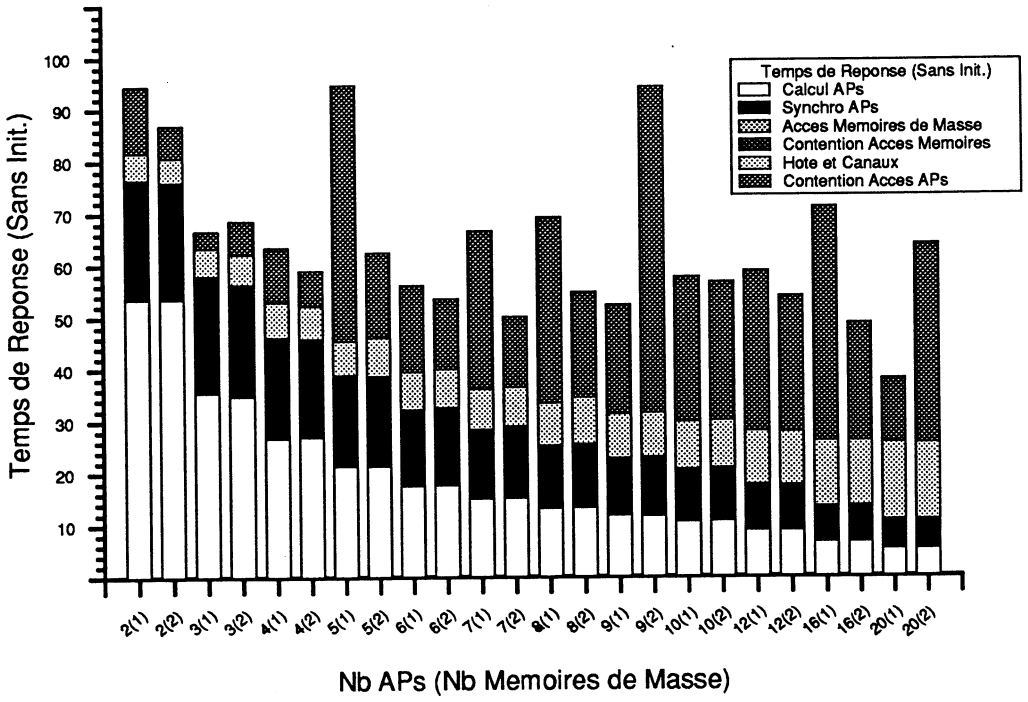

Figure 6.33: Addition d'une Mémoire de Masse Globale: Temps de Réponse (Sans Initialisation)

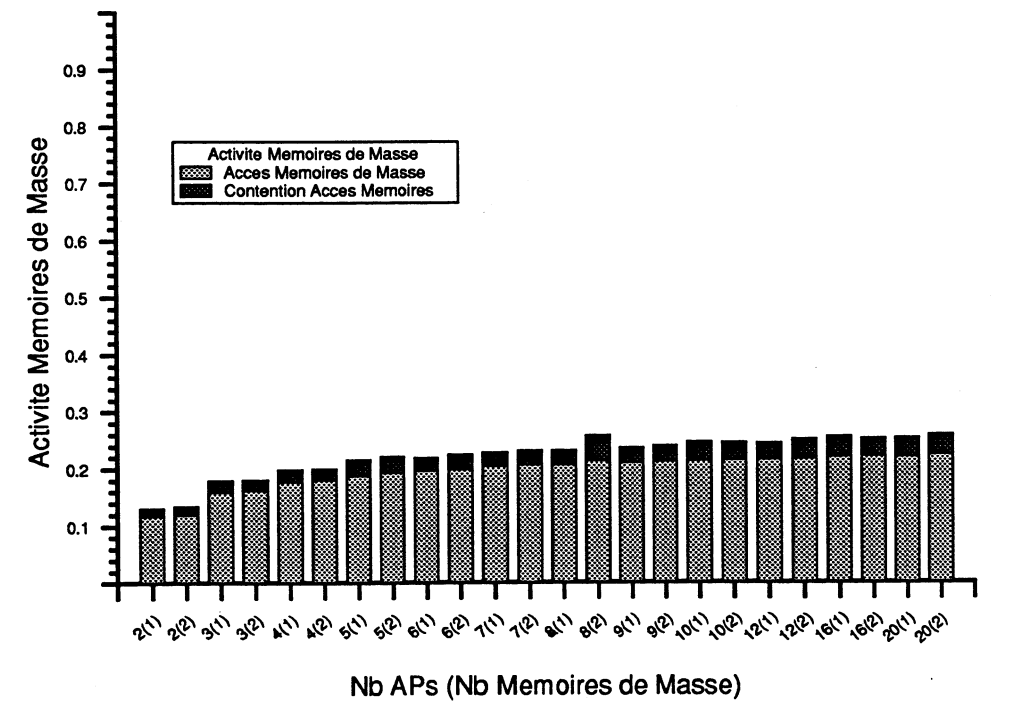

Figure 6.34: Addition d'une Mémoire de Masse Globale: Activité Mémoires de Masse

Accroissement du Nombre d'APs Disponibles

Le fait de doter le système ICAP de 10 APs et 5 mémoires de masse locales supplémentaires, tout en ne modifiant pas la charge qui lui est soumise, est très profitable pour le job principal, comme cela pourrait être mis en évidence également pour chacun des jobs d'arrière-plan. En effet, on note une réduction conséquente du temps d'attente pour l'accès aux APs, et ceci quel que soit le nombre d'APs attachés au job principal (cf figure 6.35).

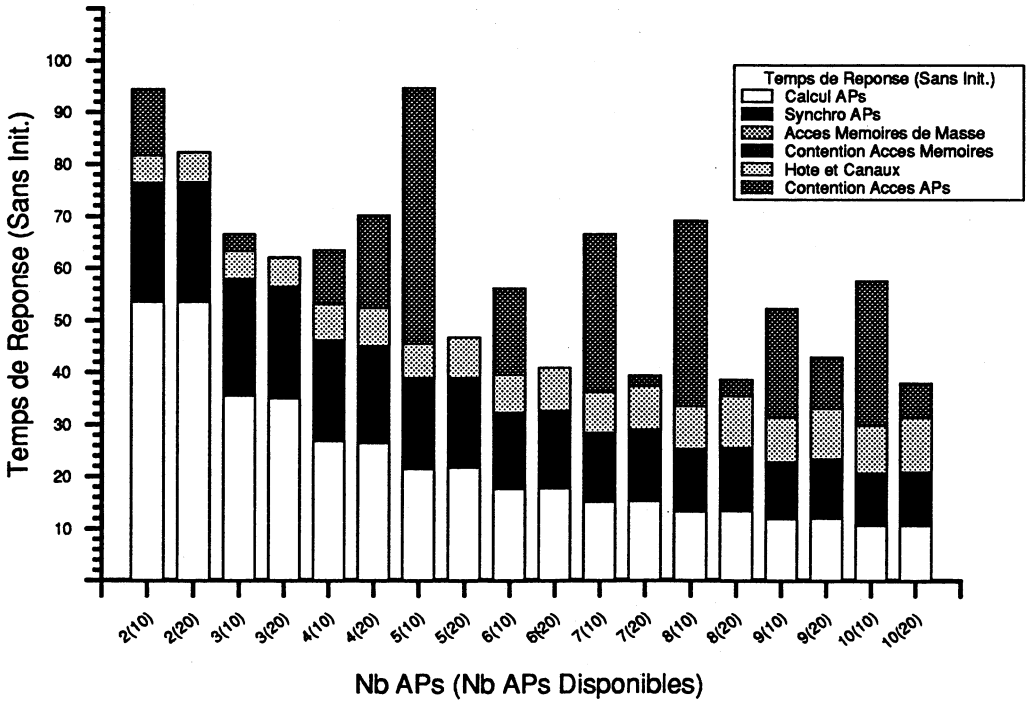

Figure 6.35: Influence du Nombre d'APs Disponibles: Temps de Réponse (Sans Initialisation)

Toutefois, comme lors de l'analyse de l'influence de la granularité des tâches parallèles, l'activité hôte-canaux augmente légèrement, à cause d'un petit accroissement de la contention pour l'accès au CPU de l'hôte, partagé entre un plus grand nombre de machines virtuelles esclaves actives (cf figure 6.36).

Ce phénomène se traduit sur l'histogramme de type D (cf figure 6.37), d'une part par une augmentation du taux moyen d'occupation des APs par le job principal, accompagnée d'une réduction du taux moyen d'occupation de ces mêmes APs par les jobs d'arrière-plan, d'autre part par un accroissement du taux moyen d'occupation de l'hôte par les jobs d'arrière-plan, qui ont une plus grande accessibilité aux APs et donc une plus grande activité parallèle à celle du job principal.

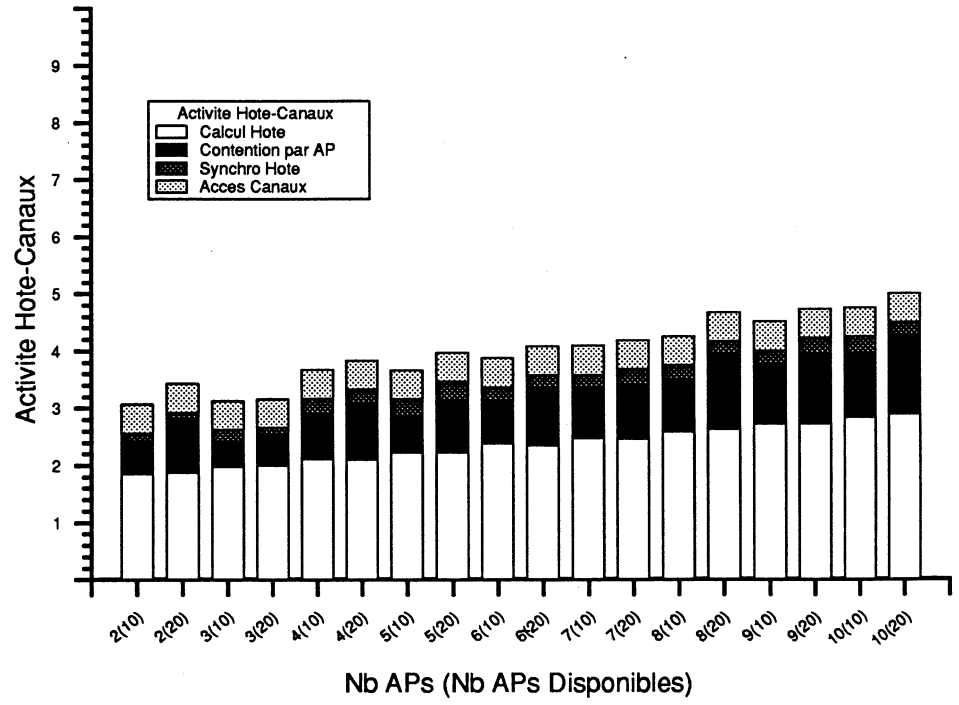

Figure 6.36: Influence du Nombre d'APs Disponibles: Activité Hôte-Canaux

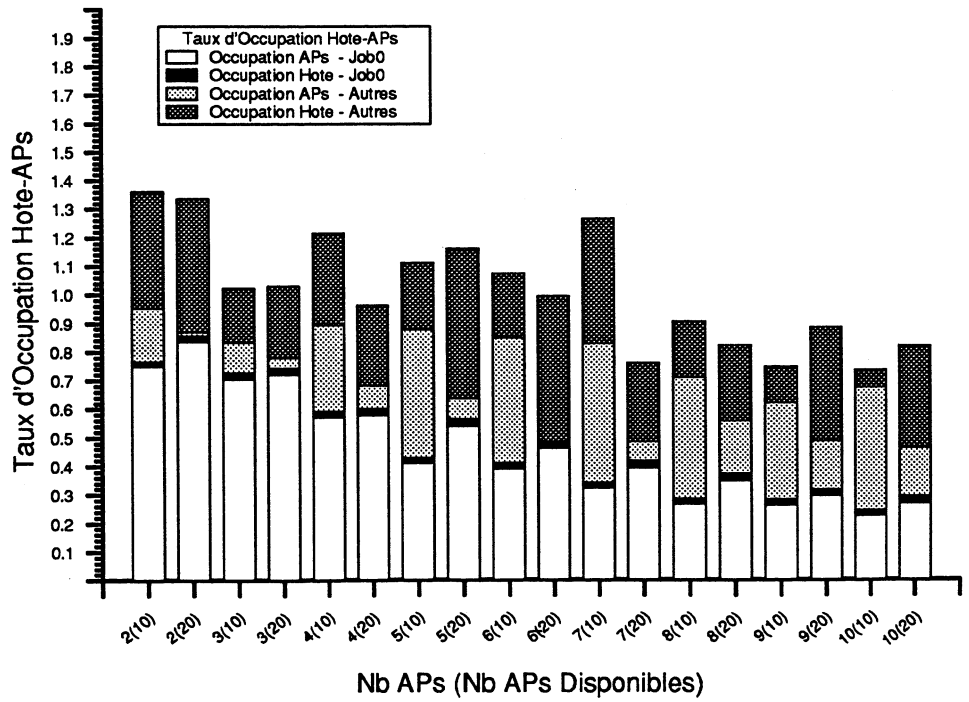

Figure 6.37: Influence du Nombre d'APs Disponibles: Taux d'Occupation Hôte-APs

Accroissement de la Vitesse de Calcul de l'Hôte

L'accroissement de la vitesse de calcul de l'hôte a, comme dans le cas sans charge, un effet positif sur le temps CPU consommé sur l'hôte par le job principal, ainsi que sur la contention entre esclaves pour l'accès au CPU (cf figure 6.38).

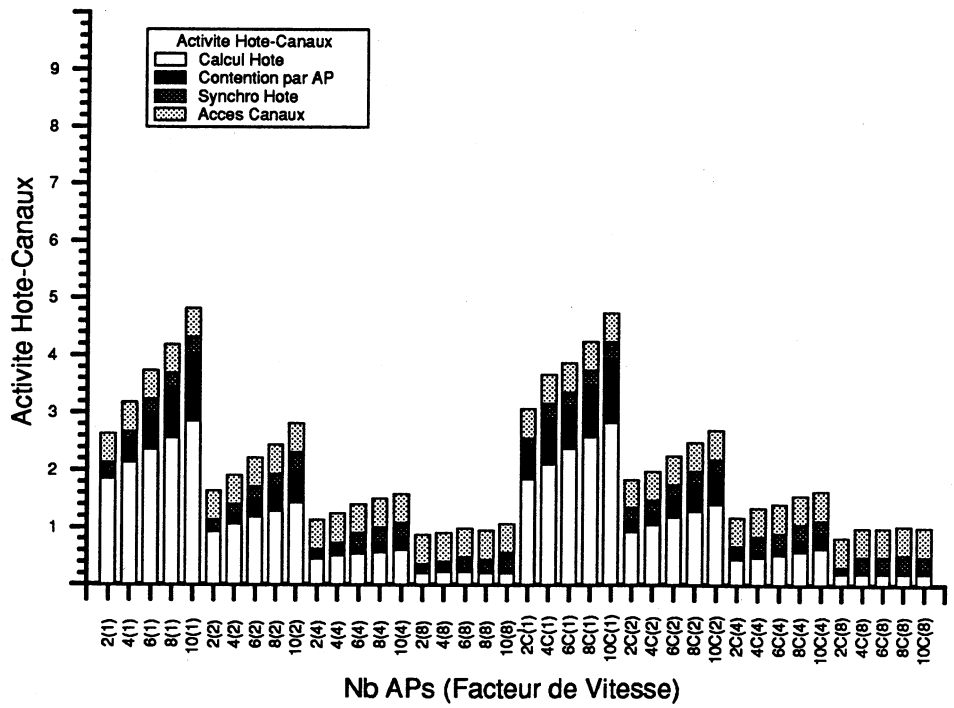

Figure 6.38: Influence de la Vitesse de l'Hôte: Activité Hôte-Canaux

L'histogramme des taux d'occupation (cf figure 6.39) reflète aussi cet effet bénéfique par une réduction des taux moyens d'occupation de l'hôte, pour le job principal comme pour les jobs d'arrière-plan.

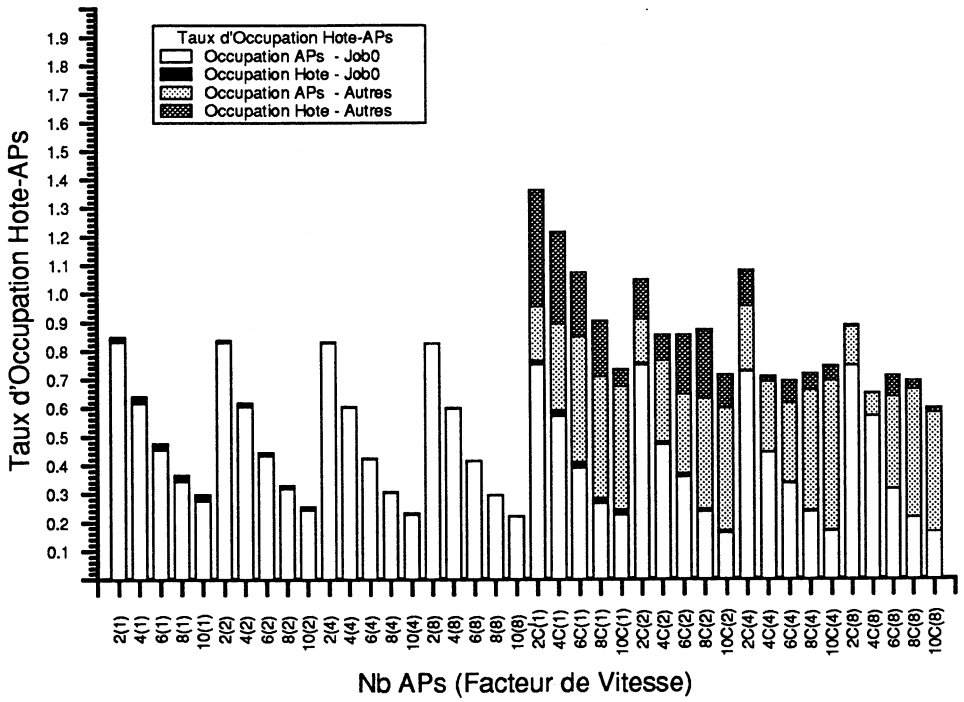

Figure 6.39: Influence de la Vitesse de l'Hôte: Taux d'Occupation Hôte-APs

Accroissement de la Vitesse de Calcul des APs

Le premier fait remarquable, lorsqu'on considère des APs plus rapides, est une diminution du temps d'attente du job principal pour l'accès aux APs (cf figure 6.40), ceci notamment pour un petit nombre d'APs attachés (2 ou 4).

Ce phénomène s'accompagne, comme pour le cas sans charge, d'une réduction du temps de calcul moyen sur chaque AP et du temps d'attente dû aux synchronisations.

On note, là encore, une décroissance importante des temps de communication entre APs (cf figure 6.41), expliquée par une réduction du temps d'initialisation sur chaque AP des communications (temps prépondérant pour de petits messages - cf paragraphe 6.4.1).

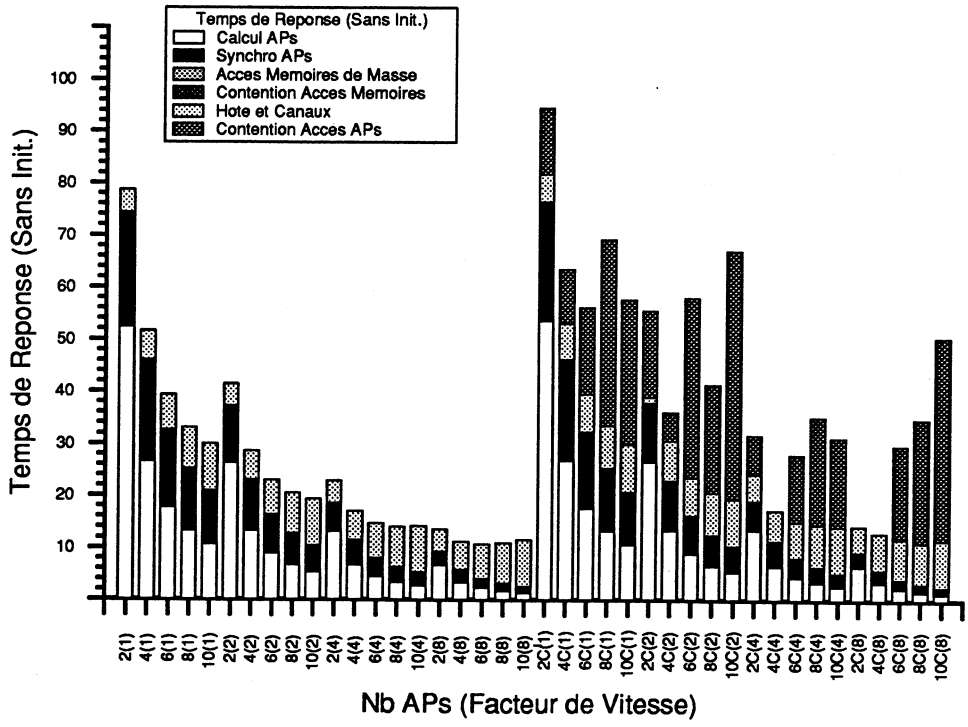

Figure 6.40: Influence de la Vitesse des APs: Temps de Réponse (Sans Initialisation)

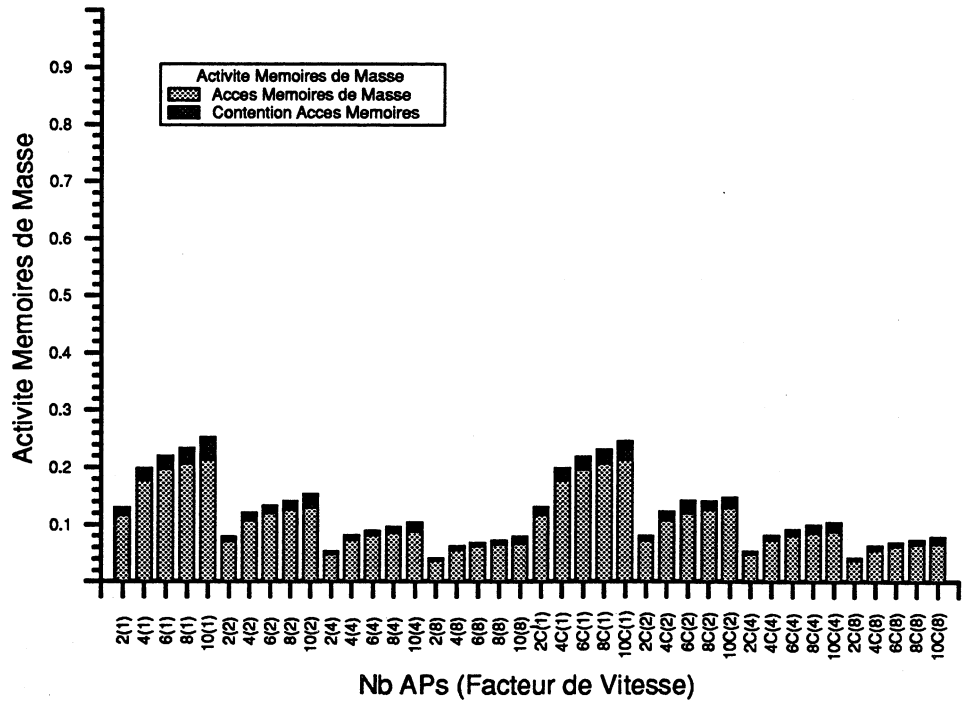

Figure 6.41: Influence de la Vitesse des APs: Activité Mémoires de Masse

Accroissement de la Vitesse d'Accès aux Mémoires de Masse

L'accroissement de la vitesse d'accès aux mémoires de masse est, comme dans le cas sans charge, guère sensible, compte tenu des petits messages échangés entre APs (cf figure 6.42).

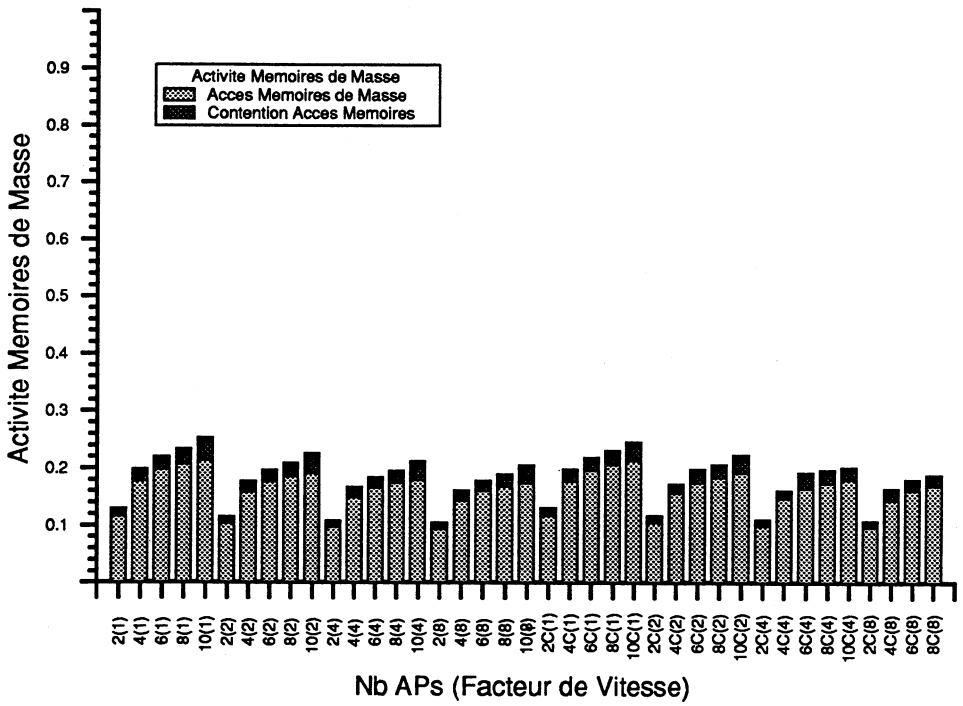

Figure 6.42: Influence de la Vitesse d'Accès aux Mémoires de Masse: Activité Mémoires de Masse

Modifications Système

Réduction et Accroissement de la Durée de la Tranche de Temps de l'Hôte

Une modification de la durée de la tranche de temps de l'hôte ne semble provoquer aucun changement notable dans le temps d'exécution du job principal (cf figure 6.43).

Aucune tendance nette ne se décèle non plus sur les taux d'occupation (cf figure 6.44).

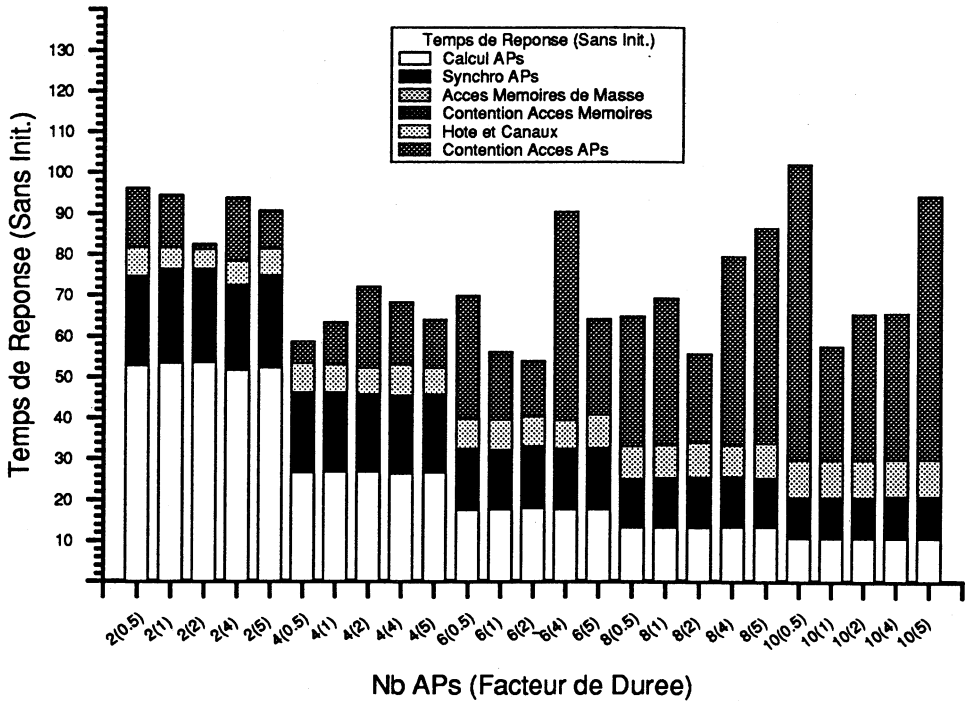

Figure 6.43: Influence de la Durée de la Tranche de Temps de l'Hôte: Temps de Réponse (Sans Initialisation)

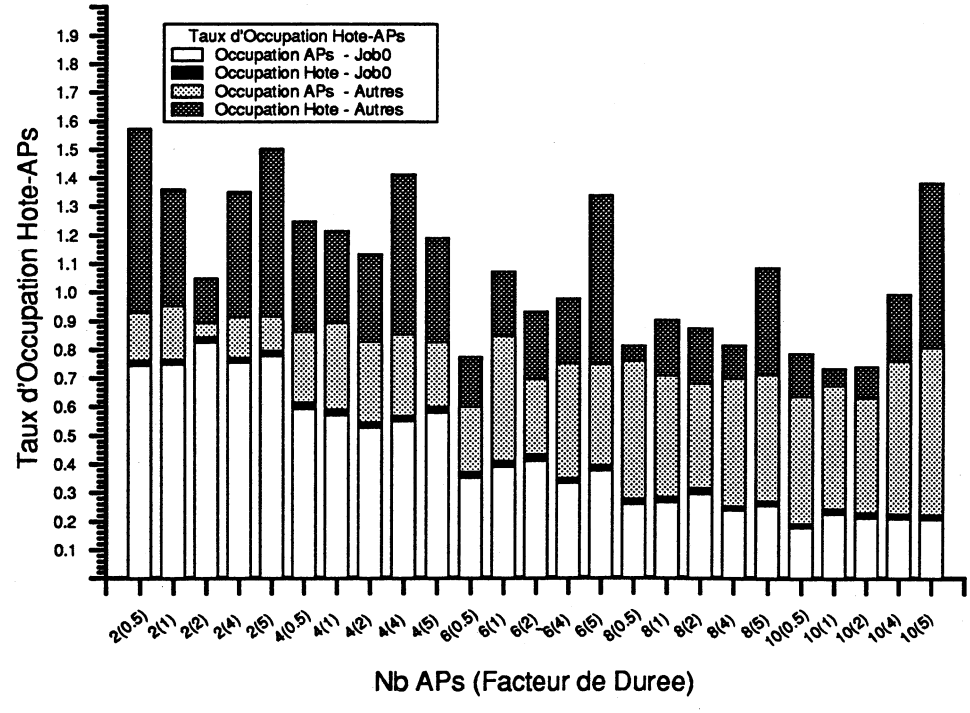

Figure 6.44: Influence de la Durée de la Tranche de Temps de l'Hôte: Taux d'Occupation Hôte-APs

Réduction et Accroissement de la Durée de la Tranche de Temps des APs

Une augmentation de la durée de la tranche de temps des APs a un effet néfaste sur le job principal (cf figure 6.45), qui doit attendre en moyenne plus longtemps pour accéder aux APs, de par le mécanisme de "rollin/rollout" qui ne peut s'effectuer qu'en fin de tranche de temps (cf paragraphe 6.2.1).

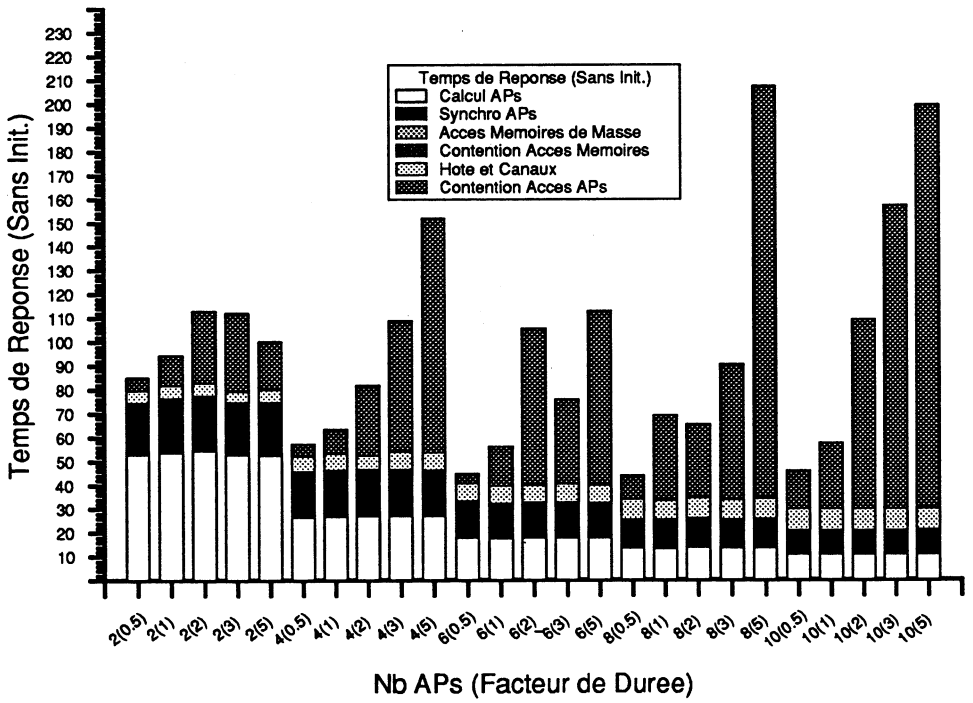

Figure 6.45: Influence de la Durée de la Tranche de Temps des APs: Temps de Réponse (Sans Initialisation)

Cela se traduit sur l'histogramme de type D (cf figure 6.46) par une diminution du taux moyen d'occupation des APs par le job principal, au profit des jobs d'arrière-plan, dont le taux moyen d'occupation sur ces mêmes APs augmente.

On ne peut donc pas dire que l'augmentation de la durée de la tranche de temps des APs est une mauvaise chose en soi, car elle maintient le taux moyen d'occupation des APs globalement (c'est-à-dire pour l'ensemble des jobs présents dans le système) constant. Certes, elle pénalise quelque peu les jobs de forte priorité, donc de courte durée de calcul, tels que le job principal étudié ici.

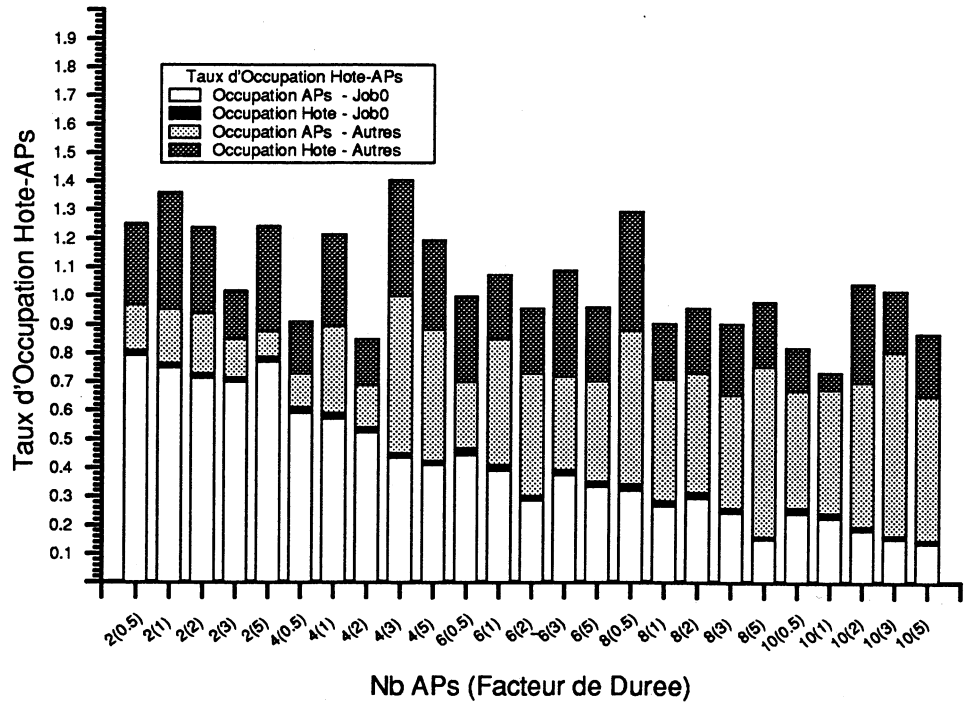

Figure 6.46: Influence de la Durée de la Tranche de Temps des APs: Taux d'Occupation Hôte-APs

Rappelons néanmoins le coût très important (de l'ordre d'une dizaine de secondes) associé à chaque "rollin/rollout", qui va à l'encontre d'une bonne utilisation des ressources que sont les APs. Il est par conséquent souhaitable de limiter le nombre des "rollin/rollouts" grâce à une tranche de temps d'une durée suffisante.

Tout le dilemme pour le système parallèle ICAP est de garantir une bonne efficacité du système, par une utilisation satisfaisante des ressources, tout en maintenant des temps de réponse acceptables pour l'utilisateur, principalement pour des jobs de courte durée de calcul.

6.7.5 Conclusion

Rappelons maintenant les points les plus importants exhibés lors de cette analyse de prédiction de performances.

Si l'on s'intéresse au job principal étudié, on a mis en évidence que 6 (respectivement 8) APs semblent être le nombre optimal d'APs à utiliser pour exécuter ce job avec (respectivement sans) charge, afin d'enregistrer un temps d'exécution minimal, en faisant abstraction du temps d'initialisation du job. Il est clair que cette conclusion s'adresse à un job identique au job principal étudié, mais réalisant un nombre d'itérations suffisant pour que le temps d'initialisation soit négligeable.

Si l'on prend en compte le temps d'initialisation, le nombre optimal est ramené à 4 APs seulement.

On définira donc la granularité des tâches de calcul parallèles sur les APs en fonction de ce nombre optimal.

D'une manière plus générale, on a vu qu'un meilleur équilibrage de charge entre les tâches parallèles de calcul sur les APs a un effet bénéfique non négligeable sur les temps d'attente dûs aux synchronisations, que ce soit sur les APs comme sur l'hôte. Cependant, pour des programmes qui réaliseraient des communications entre APs constituées de gros messages ou qui effectueraient de gros transferts de données AP-mémoire de masse partagée, un équilibrage de charge parfait peut accroître les contentions pour l'accès aux mémoires de masse, ayant pour conséquence la perte de l'avantage procuré par l'équilibrage de charge.

Parmi les extensions système étudiées, celle apportant l'amélioration la plus sensible sur le temps d'exécution du job principal est l'augmentation de la vitesse de calcul des APs. De même, une augmentation du nombre d'APs connectés à l'hôte peut diminuer notablement le temps d'attente des jobs pour l'accès aux APs qui leur sont attachés. Toutefois, cette augmentation accroît la concurrence des machines virtuelles esclaves pour l'accès au CPU de l'hôte. Il s'agit donc d'adapter le nombre d'APs du système à la puissance de traitement de l'hôte, surtout si l'hôte est par ailleurs utilisé pour des programmes de calcul séquentiels (non considérés lors de cette étude).

D'un point de vue méthodologique, cette étude de prédiction de performances a mis en évidence tout l'intérêt apporté par une représentation graphique bien adaptée des différents critères de performance. Elle a permis aussi de souligner la difficulté rencontrée quelquefois dans l'interprétation des résultats avec charge, et dans le cas présent, la nécessité d'un nombre plus important de simulations avec les mêmes paramètres, afin de réduire les variations, induites par les jobs d'arrière-plan sur le temps d'attente du job principal pour l'accès aux APs, d'un jeu de paramètres à l'autre.

Insistons également sur le fait qu'il est toujours préférable d'effectuer en premier une étude sans charge [**T**], ce qui permet de bien caractériser le job principal et d'exhiber les points importants de son comportement. Ensuite, l'analyse des résultats avec charge ne pourra qu'en être facilitée, par une comparaison avec les résultats sans charge.

Enfin, on restera très prudent dans l'interprétation des résultats avec charge: on essaiera plutôt de dégager des tendances générales que de formuler des conclusions très précises, qui pourraient s'avérer un peu hâtives. Il est sans doute préférable d'en dire peu, mais d'être quasiment assuré de ce que l'on dit.

6.8 Prolongements Possibles

Nous consacrons le dernier paragraphe de ce chapitre aux prolongements que nous envisageons pour cette modélisation.

Nous prévoyons de travailler sur deux plans:

1. tout d'abord, nous nous attacherons à réaliser d'autres analyses de prédiction de performances, afin de généraliser les résultats déjà obtenus,
2. puis, nous nous intéresserons à améliorer l'interface du programme de simulation avec l'utilisateur.

6.8.1 D'autres Analyses de Prédiction de Performances

L'analyse de prédiction de performances présentée au paragraphe 6.7 n'est évidemment pas une fin en soi. D'autres alternatives quant aux paramètres d'entrée du modèle peuvent être prises en compte.

Parmi elles, citons:

- pour le modèle de l'architecture, la modification de la politique d'allocation des APs (par une action sur la fonction de poids dynamique des jobs), la modification de la topologie du réseau d'interconnexion AP-mémoires de masse,

- pour le modèle des programmes, la variation de la classe de priorité du job principal (qui, rappelons-le, était fixe dans la prédiction de performances que nous avons réalisée, procurant au job principal une priorité maximale), la considération de jobs principaux appartenant aux autres catégories de programmes (telles qu'elles ont été définies au paragraphe 6.7.2), la prise en compte d'une charge supplémentaire sur l'hôte représentant les jobs exécutant des programmes séquentiels ou les sessions interactives des utilisateurs.

Pour les simulations avec charge, il sera nécessaire de simuler une centaine de fois le job principal pour le même jeu de paramètres d'entrée afin de mieux caractériser l'influence moyenne des jobs d'arrière-plan sur l'exécution du job principal. Cependant, une migration du code Turbo Pascal du programme de simulation devra être effectuée afin de permettre la réalisation des simulations sur un système plus puissant que l'IBM PS/2 de la série 60, que nous avons utilisé.

6.8.2 Interface du Programme de Simulation Avec l'Utilisateur

L'amélioration de l'interface utilisateur du programme de simulation s'opérera par l'utilisation omniprésente d'un interface graphique.

Tout d'abord, l'interface graphique développé pour le simulateur du gestionnaire de ressources (présenté au paragraphe 6.5) devra être complété afin de permettre, en cours de simulation, une visualisation sélective des jobs présents dans le système.

L'idée de base est une représentation graphique du système ICAP semblable à celle de la figure 6.47, correspondant au modèle de l'architecture (gestionnaire de ressources excepté). Sur cette représentation graphique, l'activité de chaque job sera symbolisée par des points d'une couleur différente. Ainsi, chaque client du modèle des programmes, associé à ce job, sera représenté par un point dont la taille sera proportionnelle à la demande de service qu'il effectue à la station où il se trouve actuellement.

En temps réel simulé, l'utilisateur aura une bonne visualisation de la charge induite sur le système par chacun des jobs. Il pourra, par action sur une souris, sélectionner un job particulier (à partir d'une palette de couleurs) et obtenir les critères de performances courants pour ce job (temps d'exécution, temps de service à chaque station de service, temps d'attente dû aux synchronisations et aux contentions d'accès aux ressources partagées, taux d'occupation, ...). De même, en désignant,

grâce à la souris, un composant particulier du système (hôte, canal hôte-AP, AP, mémoire de masse), il pourra obtenir les critères de performances courants propres à cette station de service (nombre de clients en train d'être servis ou en attente, temps de service moyen des clients, temps d'attente moyen des clients, ...), globalement et pour chaque job.

Parmi les nombreux avantages apportés par cet interface graphique (convivialité, meilleure analyse du comportement des jobs à l'intérieur du système), il est à souligner que cet interface procurera à l'utilisateur une excellente visualisation des phénomènes transitoires, que l'analyse seule des critères de performance en fin de simulation peut masquer.

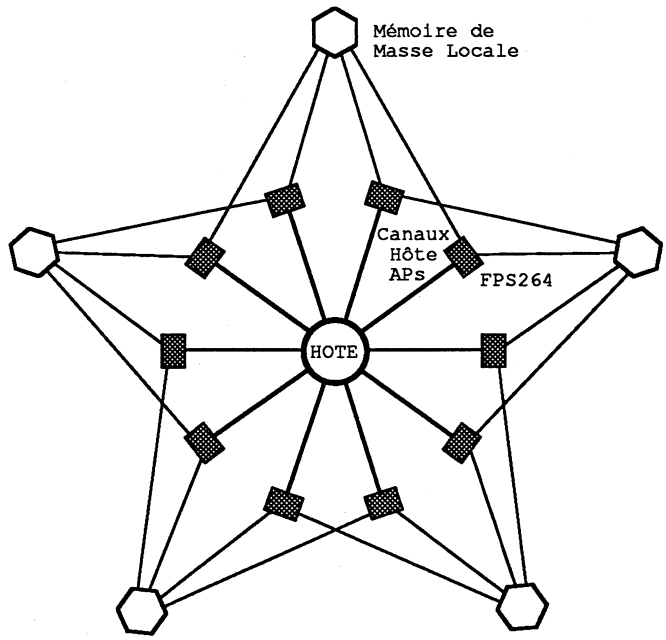

Figure 6.47: Représentation Graphique du Système ICAP Utilisée pour le Futur Interface Graphique du Programme de Simulation

Dans un deuxième temps, un interface graphique viendra enrichir le module de calcul et d'édition des critères de performance en fin de simulation. L'utilisation de cet interface devra permettre à l'utilisateur de sélectionner interactivement les critères de performances qu'il veut voir figurer sur le même histogramme, de visualiser sur l'écran cet histogramme, d'en modifier à loisir l'apparence et, s'il le désire, d'en obtenir une copie sur imprimante (si possible couleur). L'analyse des résultats de chaque simulation est ainsi grandement facilitée et la représentation de toute combinaison de résultats issus de simulations différentes est offerte à l'utilisateur, augmentant les possibilités de comparaison d'alternatives multiples.

Certes, le travail nécessaire au développement de cet interface utilisateur est énorme et nous ne sommes pas au bout de nos peines, mais nous considérons que le développement très en vogue actuellement des applications graphiques, dans tous les domaines de l'informatique, doit être mis à profit en évaluation des performances. En effet, seul le développement d'interfaces graphiques pour les outils d'évaluation de performances pourra réellement faire prendre conscience aux non spécialistes (et notamment aux décideurs des entreprises) de l'intérêt essentiel de cette discipline, en les incitant à utiliser ses outils par l'attrait et la convivialité de leur interface utilisateur.

CHAPITRE 7

Conclusion

Au travers de quatre études de cas, nous avons présenté des modèles et des mesures de performances d'architectures parallèles particulièrement détaillés. Une **bonne concordance entre résultats issus de la résolution des modèles et mesures** a pu être constatée.

Ces exemples ont montré tout le travail nécessaire pour mener à bien une modélisation précise de systèmes réels. Cela requiert beaucoup de temps et de minutie de la part du modéliseur, dont le souci principal est la **recherche du meilleur compromis entre la précision** qu'il aimerait bien introduire dans le modèle **et les coûts d'obtention des paramètres d'entrée associés**.

A partir de ces études de cas, nous avons proposé une méthodologie de modélisation adaptée à l'évaluation des performances des architectures parallèles. Les principaux aspects de cette méthodologie ont été largement illustrés par des exemples tirés des études de cas.

Ainsi, en phase de conception du modèle, la **distinction entre modèle de l'architecture et modèle des programmes** permet une bonne décomposition du processus de modélisation, et on a vu (cf modèle du système ICAP) comment cette distinction peut être mise à profit pour permettre, sans modification du modèle de l'architecture, une utilisation déterministe, puis probabiliste, du modèle global.

Pour modéliser les programmes, on distingue d'une part **les tâches de calcul**, dont on détermine une **finesse de modélisation** et dont on génère le **degré de parallélisme** (cf le modèle de contrôle du calculateur scientifique FPS264), en ayant recours à la notion de **processeur virtuel** si les processeurs de l'architecture parallèle sont hétérogènes (comme dans le cas du réseau local d'IBM PC), d'autre part **les communications et les synchronisations entre tâches parallèles**, dont les deux composantes principales sont le temps de latence et le temps effectif de transmission. En outre, afin de bien refléter les phénomènes d'attente dûs au déséquilibre de charge entre tâches parallèles, on veille à modéliser fidèlement les mécanismes de synchronisation entre ces tâches (cf modèle du système ICAP).

Un autre aspect très important de la méthodologie est la **constitution de classes de programmes** par l'application de techniques empruntées à l'analyse des données, permettant une réduction appréciable du nombre des résolutions du modèle (cf modèle du FPS264).

En phase d'estimation et de mesure des paramètres d'entrée nécessaires à la résolution du modèle, on peut utiliser un **moniteur hybride distribué** s'il est disponible, ou avoir recours à de simples appels à l'horloge système (attention au problème des horloges asynchrones des divers éléments du système parallèle).

Dans les deux cas, on s'efforce de reproduire les mesures un grand nombre de fois afin de mieux **apprécier leur variabilité**, et on peut avoir recours à des méthodes d'ajustement aux moindres carrés pour la détermination de formules simples de calcul de coût (cf modèle du système ICAP).

Pour résoudre le modèle, on utilise si possible des techniques de réduction de l'espace des états par l'application de **méthodes d'agrégation** (cf modèles des réseaux point-à-point de systèmes HP3000 et du calculateur scientifique FPS264) ou des techniques plus spécifiques aux réseaux de Petri stochastiques, telles que le **tir simultané des transitions synchrones tirables**, l'élimination des transitions à taux de tir nul (cf modèle du FPS264).

Pour valider le modèle, on procède à une comparaison entre les résultats fournis par la résolution du modèle et les valeurs correspondantes mesurées sur le système réel.

Pour ce faire, on prend en compte un **nombre maximum de critères de performances** que l'on mesure sur le **système dédié**. Une démarche progressive est souhaitable (cf modèles du FPS264 et du système ICAP) et les mesures doivent, si possible, être répétées plusieurs fois, afin d'appréhender leur reproductibilité. On veille à limiter l'impact de l'outil de mesures sur l'exécution des programmes mesurés (cf modèle du système ICAP).

La dernière phase concerne l'utilisation du modèle à des fins de prédiction de performances du système réel. On a recours de préférence à une **modélisation probabiliste** qui permet une étude exploratoire beaucoup plus générale.

L'approche, que nous préconisons pour un système parallèle conçu pour une exploitation en environnement non dédié, est, dans un premier temps, **l'étude précise** (relativement déterministe) **d'un job** (ou de plusieurs jobs) **en environnement dédié**, et dans un deuxième temps, **l'étude du même job** (ou des mêmes jobs) **en compétition avec d'autres jobs**, modélisant de manière aléatoire la charge moyenne du système étudié (cf modèle du système ICAP).

Cette approche permet une **meilleure caractérisation des résultats avec charge par leur comparaison avec les résultats sans charge** et réalise un compromis entre une utilisation du modèle très déterministe, requérant un nombre prohibitif de paramètres d'entrée, et une utilisation purement probabiliste, dont les résultats s'avèrent parfois assez éloignés de la réalité. On réservera le mode d'utilisation purement probabiliste à des études prospectives réalisées en phase de conception des systèmes.

Parmi les prolongements possibles à ce travail, le principal à mes yeux est une étude plus générale des caractéristiques essentielles des programmes parallèles, qui pourrait permettre une réelle **classification de ces programmes**. Les différentes classes pourraient dépendre de quelques caractéristiques bien spécifiques à l'architecture parallèle cible.

D'après mon expérience en matière de modélisation, il s'avère que le modèle des programmes est généralement plus difficile à construire que le modèle de l'architecture. Une telle classification, outre l'avantage de produire des classes de programmes dont l'étude pourrait se réduire à celle d'un (voire de quelques) représentant(s), faciliterait la conception du modèle des programmes grâce à la caractérisation des programmes qu'elle procurerait, en fonction de l'architecture parallèle cible.

Le dernier point que j'aborderai dans ce rapport de thèse me tient particulièrement à coeur. Il concerne la nécessité d'**enrichir chaque programme, ou outil, d'évaluation de performances, d'interfaces graphiques**, si l'on veut sensibiliser les chefs d'entreprise et les responsables de services informatiques aux bénéfices que peuvent apporter des études, prédictives ou non, sur les performances de leurs systèmes.

En effet, j'ai pu me rendre compte, lors de colloques ou congrès sur les performances (notamment "Computer Measurement Group '88" et "Computer Measurement Group Francophone '89"), que la plupart des responsables de grandes entreprises sont confrontés au dur problème d'adapter les besoins en traitement informatique de leur entreprise, ou service informatique, à la capacité de traitement de leurs systèmes informatiques. Bien souvent, leur démarche est empirique. Bien sûr, des raisons historiques ont encouragé cette tendance, dans la mesure où la technologie des systèmes évoluait aussi vite que la croissance des besoins en traitement informatique. Ainsi, le remplacement d'un système par celui de la nouvelle génération suffisait à soutenir les nouveaux besoins de l'entreprise. Or, il semble que l'évolution de la technologie marque actuellement le pas. Le problème de la gestion des performances, tel qu'il a été défini dans l'introduction du présent rapport, devient de ce fait très complexe.

Malgré cet état de fait, rares sont les entreprises qui disposent à ce jour d'une équipe d'évaluation de performances, voire d'un simple spécialiste en ce domaine. La principale raison en est, à mon avis, un manque d'information de la part des chefs d'entreprise du domaine de l'évaluation des performances, causé indirectement par des programmes et outils d'évaluation des performances fort peu conviviaux, ne facilitant pas le dialogue entre spécialistes et non-spécialistes. Une méconnaissance presque totale des bénéfices que peut procurer l'évaluation des performances et une réticence à l'égard des outils disponibles aujourd'hui sur le marché en résultent.

Afin de promouvoir le domaine de l'évaluation des performances et lui faire franchir facilement la porte des entreprises, un atout majeur est, selon moi, l'introduction d'interfaces graphiques dans les programmes et outils d'évaluation des performances, aussi bien pour une **interprétation aisée des résultats**, par l'édition de courbes et d'histogrammes, que pour une **simulation animée des systèmes**, lors de la résolution des modèles sous-jacents.

Gageons que cette évolution, déjà amorcée dans des logiciels tels que QNAP [Véran84] ou RESQ [Sauer86], se confirmera dans un proche avenir, conjointement à celle des systèmes experts d'aide à la modélisation.

Références Bibliographiques

[Auber87] **J. Auber, M. Becker, J.-P. Prost**
"NETPERF, A Modeling Tool for HP3000 Point-to-Point Networks"
Proceedings of the 3rd International Conference on Data Communication Systems and their Performance, Rio de Janeiro, 22-25 Juin 1987, p.437-452.

[Autom] **Hewlett-Packard**
"Automan User's Guide"

[Bacce86] **F. Baccelli, A.M. Makowski, A. Shwartz**
"Simple Computable Bounds and Approximations for the Fork-Join Queue"
Journal of the A.C.M., 1986.

[Balbo84] **G.-F. Balbo, M.A. Marsan, G. Conte**
"A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems"
ACM Transactions on Computer Systems, Vol. 2, No. 2, Mai 1984.

[Baske75] **F. Baskett, K.M. Chandy, R.R. Muntz, F.G. Palacios**
"Open, Closed and Mixed Networks of Queues with Different Classes of Customers"
J.A.C.M., No. 22, Avril 1975, p.248-260.

[Becke87] **M. Becker**
"Review of Performance Evaluation of ICAP Supercomputers"
International Workshop on Modeling Techniques and Performance Evaluation, North Holland, S. Fdida, G. Pujolle, eds., 9-11 Mars 1987.

[Becke88] **M. Becker, J.-P. Prost**
"An Aggregated Stochastic Petri Net for the FPS/264 Model"
Poster in 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Santa Fé, 24-27 Mai 1988
A paraître dans "Computer Systems Science and Engineering".

[Bough88] **E. Boughter, W. Alexander, T. Keller**
"*A Tool for Performance-Driven Design of Parallel Systems*"
Proceedings of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Palma de Mallorca, 14-16 Septembre 1988, p.171-191.

[Bouha88] **J.P. Bouhana**
"*Visual Characterization of Transaction Processing Workloads*"
Computer Measurement Group 1988 International Conference on Management and Performance Evaluation of Computer Systems, Dallas, 12-16 Décembre 1988, p.463-475.

[Brine88] **J.V. Briner, Jr.**
"*A Framework for Analyzing Parallel Discrete Event Simulation*"
Computer Measurement Group 1988 International Conference on Management and Performance Evaluation of Computer Systems, Dallas, 12-16 Décembre 1988, p.180-185.

[Broch85] **L. Brochard**
"*Parallelization of Transonic Flow on ICAP*"
IBM Kingston Research Report, KGN-37, 26 Septembre 1985.

[Broch87] **L. Brochard**
"*Communication and Control Costs on a Loosely Coupled Parallel Multiprocessor*"
IBM Kingston Research Report, KGN-71, 20 Mai 1987.

[Broch88] **L. Brochard, J.-P. Prost, F. Faurie**
"*Synchronization and Load Unbalance Effects of Parallel Iterative Algorithms*"
IBM Palo Alto Scientific Center Research Report, G320-3514, Octobre 1988
Proceedings of the 18th International Conference on Parallel Processing, St Charles, IL, 8-12 Août 1989, Vol. III, p.153-160.

[Cao85] **W.-L. Cao, W.J. Stewart**
"Iterative Aggregation / Disaggregation Techniques for Nearly Uncoupled Markov Chains"
Journal of the ACM, Vol. 32, No. 3, Juillet 85, p.702-719.

[Capla] **Hewlett-Packard**
"HPCAPLAN User's Manual and Report"

[Calta87] **R. Caltabiano, A. Carnevali, J. Detrich**
"Directives for the Use of the Shared Bulk Memories: A Precompiler Extension"
IBM Kingston Research Report, KGN-110, 3 Mars 1987.

[Capot86] **A. Capotondi, V. Sonnad, S. Chin**
"Parallel Resolution of the Shallow Water Equations Using an Explicit Finite Difference Algorithm"
IBM Kingston Research Report, KGN-57, 8 Septembre 1986.

[Caube88] **S. Cauberghs, J. Detrich**
"Computation on the Loosely Coupled Array of Processors System: Parallel Scheduler for the lCAP System: Operator and User Guide"
IBM Kingston Research Report, KGN-168, 13 Janvier 1988.

[Chand88] **K.M. Chandy**
"Performance Management of Parallel Computers"
Computer Measurement Group 1988 International Conference on Management and Performance Evaluation of Computer Systems, Dallas, 12-16 Décembre 1988, p.1076-1078.

[Chate84] **F. Chatelin**
"Iterative Aggregation/Disaggregation Methods"
International Workshop in Applied Mathematics and Performance / Reliability Models of Computer / Communication Systems, North Holland, G. Iazeolla, S. Tucci, eds., 1984.

[Chin85] **S. Chin, L. Domingo, R. Caltabiano, A. Carnevali, J. Detrich**
"sl Parallel Computation on the Loosely Coupled Array of Processors: A Guide to the Precompiler"
IBM Kingston Research Report, KGN-42, 25 Novembre 1985.

[Chiol87] **G. Chiola**
"*A Graphical Petri Net Tool for Performance Analysis*"
Proceedings of the 3rd International Workshop on Modeling Techniques and Performance Evaluation, AFCET, Paris, Mars 1987

[Chiol88] **G. Chiola**
"*Compiling Techniques for the Analysis of Stochastic Petri Nets*"
Proceedings of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Palma de Mallorca, 14-16 Septembre 1988, p.13-27.

[Cleme86] **E. Clementi, J. Detrich, S. Chin, G. Corongiu, D. Folsom, D. Logan and R. Caltabiano, A. Carnevali, J. Helin, M. Russo, A. Gnudi, P. Palamidese**
"*Large Scale Computations on a Scalar, Vector and Parallel Supercomputer*"
IBM Kingston Research Report, KGN-77, 15 Août 1986.

[Cleme88] **E. Clementi, D. Logan, J. Saarinen**
"*ICAP/3090: Parallel Processing for Large-Scale Scientific and Engineering Applications*"
IBM Systems Journal, Vol. 27, No. 4, 1988, p.475-509.

[Comfo87] **J.C. Comfort**
"*Simulation Model Testing*"
Proceedings of the 20th Annual Simulation Symposium, Tampa, 11-13 Mars 1987, p.185-196.

[Coron88] **G. Corongiu, J. Detrich, E. Clementi**
"*Study of Communication and Synchronization Overhead on the Loosely Coupled Array of Processors Systems*"
IBM Kingston Research Report, KGN-170, 29 Mars 1988.

[Court77] **P.-J. Courtois**
"Decomposability: Queueing and Computer System Applications"
ACM Monograph Series, 1977.

[Covin88] **R.C. Covington, S. Madala, V. Mehta, J.R. Jump, J.B. Sinclair**
"The Rice Parallel Processing Testbed"
Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Santa Fé, 24-27 Mai 1988, p.4-11.

[Darem88] **F. Darema, D.A. George, V.A. Norton, G.F. Pfister**
"A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN"
Parallel Computing, Vol. 7, No. 1, Avril 1988, p.11-24.

[David70] **H.A. David**
"Order Statistics"
John Wiley, New York, 1970.

[Deese88] **D.R. Deese**
"Designing an Expert System for Computer Performance Evaluation"
Computer Measurement Group 1988 International Conference on Management and Performance Evaluation of Computer Systems, Dallas, 12-16 Décembre 1988, p.75-80.

[Denni87] **P.J. Denning, G.B. Adams III**
"Research Questions for Performance Analysis of Supercomputers"
Proceedings of the International Seminar on Scientific Supercomputers, Paris, 2-6 Février 1987, p.149-162.

[Detri88] **J. Detrich, D. Folsom, L. Rosenzweig**
"ICAP/3090 at IBM Kingston: Evolution of Software to Support Parallel Execution"
IBM Kingston Research Report, KGN-173, 18 Février 1988
Third International Conference on Supercomputing, Boston, Mai 1988.

[Doman88] **B. Domanski**
"System Performance Management and Capacity Planning"
Computer Measurement Group 1988 Int. Conf. on Management and Performance Evaluation of Computer Systems, Dallas, 12-16 Décembre 1988, p.846-853.

[Duda86a] **A. Duda, T. Czachorski**
"Performance Evaluation of the Fork and Join Synchronisation Primitives"
Rapport de Recherche, Laboratoire d'Informatique des Systèmes Expérimentaux et leur Modélisation, Janvier 86.

[Duda86b] **A. Duda, G. Harrus, Y. Haddad, G. Bernard**
"Monitoring of Distributed Systems"
Rapport de Recherche No. 52, Laboratoire d'Informatique des Systèmes Expérimentaux et leur Modélisation, Décembre 86.

[Duda88] **A. Duda**
"On the Trade-Off Between Parallelism and Communication"
Proceedings of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Palma de Mallorca, 14-16 Septembre 1988, p.379-391.

[Fang88] **C. Fang, W. ZhengZhong, T. Renshou**
"The Application of Expert System in Automatic Programming of Discrete Event Simulation System"
IMACS 1988 12th World Congress on Scientific Computation, Paris, 18-22 Juillet 1988, Vol. 2, p.605-607.

[Feath84] **F. Feather**
"Validation of a Fault-Tolerant Multiprocessor: Baseline Experiments and Workload Implementation"
Thèse de Master, Department of Electric Engineering and Computer Science, Université de Carnegie-Mellon, Pittsburgh, 1984.

[Feuga88] **M. Feuga, Y. Raynaud**
"Distributed Systems Design: An Integrated Software Tool for Performance Analysis"
Proceedings of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Palma de Mallorca, 14-16 Septembre 1988, p.51-68.

[Flori78] **G. Florin, S. Natkin**
"Evaluation des performances d'un protocole de communication à l'aide des réseaux de Petri et des processus stochastiques"
Journées AFCET Multi-ordinateurs et multi-processeurs en temps réel, C.N.R.S., Paris, 1978.

[Flori85] **G. Florin, S. Natkin**
"Les réseaux de Petri stochastiques"
Technique et Science Informatiques, Vol. 4, No. 1, 1985.

[Fouca84] **T. Foucart**
"Analyse factorielle - Programmation sur micro-ordinateurs - Analyse factorielle de tableaux multiples"
Masson, 1984.

[FPS264] **Floating Point Systems**
"FPS-264 APAL64 Programmer's Guide"

[Frabo87] **C. Fraboul**
"MIMD Parallelism Expression, Exploitation and Evaluation"
Proceedings of the International Seminar on Scientific Supercomputers, Paris, 2-6 Février 1987, p.133-148.

[Gelen77] **E. Gelenbe, G. Pujolle**
"A Diffusion Model for Multiple Class Queueing Networks"
Rapport de Recherche I.N.R.I.A., No. 242, Août 1977.

[Gelen82] **E. Gelenbe, G. Pujolle**
"Introduction aux réseaux de files d'attente"
Eyrolles, 1982.

[Gentz88] **W. Gentzsch, F. Szelényi, V. Zecca**
"Use of Parallel FORTRAN for Engineering Problems on the IBM 3090 Vector Multiprocessor"
Parallel Computing, Vol. 9, No. 1, Décembre 1988, p.107-115.

[Goldb83] **A. Goldberg, G. Popek, S. Lavenberg**
"A Validated Distributed System Performance Model"
Performance '83, North Holland, A.K. Agrawala, S.K. Tripathi, eds., 1983, p.251-268.

[Graph85] **International Business Machines Corporation**
"Programmer's Reference for graPHIGS"
IBM Program Product, Décembre 1985.

[Harms88] **U. Harms, H. Luttermann**
"Experiences in Benchmarking the Three Supercomputers CRAY 1M, CRAY X/MP, FUJITSU VP-200 Compared with the CYBER 76"
Parallel Computing, Vol. 6, No. 3, Mars 1988, p.373-382.

[Herbi87] **R. Herbin, S. Gerbi, V. Sonnad**
"Parallel Implementation of a Multigrid Method on the ICAP Supercomputer: Part I & Part II"
IBM Kingston Research Report, KGN-144, 24 Juillet 1987
IBM Kingston Research Report, KGN-155, 14 Septembre 1987.

[Herzo87] **U. Herzog**
"Performance Evaluation Principles for Vector- and Multiprocessor Systems"
Proceedings of the 2nd International SUPRENUM Colloquium, Bonn, 30 Septembre - 2 Octobre 1987, Parallel Computing, Vol. 7, No. 3, Septembre 1988, p.425-438.

[Hills86] **P.R. Hills**
"A Portable Personal Computer Based Mimic System for Simulations"
International AMSE Conference, Sorrento, Italie, 29 Septembre - 1er Octobre 1986.

[Hockn85] **R.W. Hockney**
"MIMD Computing in the USA"
Parallel Computing, Vol. 2, 1985, p.119-136.

[Hockn89] **R.W. Hockney**
"Synchronization and Communication Overheads on the LCAP Multiple FPS-164 Computer System"
Parallel Computing, Vol. 9, No. 3, Février 1989, p.279-290.

[Holid85] **M.A. Holiday, M.K. Vernon**
"A Generalized Timed Petri Net Model for Performance Analysis"
International Workshop on Timed Petri Nets, Turin, Italie, Juillet 1985.

[Houei85] **P. Houeix, C. Mazel, N. Smaili**
"Application de méthodes d'agrégation à la résolution d'un modèle du système radio téléphone automatique"
Rapport de Recherche IMAG No. 494, Janvier 1985.

[Houei88] **P. Houeix**
"Evaluation de performances d'une architecture parallèle pour le traitement d'images"
Thèse de Docteur-Ingénieur, Institut National Polytechnique de Grenoble, France, 19 Septembre 1988.

[IBMPC1] **International Business Machines Corporation**
"IBM PC Network Technical Reference Manual"

[IBMPC2] **International Business Machines Corporation**
"IBM PC Network Application Program User's Guide"

[Jalby] **W. Jalby, J.M. Frailong, J. Lenfant**
"Diamonds Schemes: An Organization of Parallel Memories for Efficient Array Processing"
Rapport de Recherche No. 342, I.N.R.I.A.

[Johns88] **E. Johnson, J.P. Poorte**
"A Hierarchical Approach to Computer Animation in Simulation Modeling"
Simulation, Vol. 50, No. 1, Janvier 1988, p.30-36.

[Kasan] **Hewlett-Packard**
"Kasandra Reference Manual"

[Kienz79] **M.G. Kienzle, K.C. Sevcik**
"A Systematical Approach to the Performance Modelling of Computer Systems"
Proceedings of the 4th International Symposium on Modelling and Performance Evaluation of Computer Systems, Vienne, Autriche, Février 1979, Vol. 1

[Klar87] **R. Klar, M. Knaack, N. Luttenberger**
"Zählmonitor 4: A Monitor for Hardware and Hybrid Monitoring of Multicomputer Systems"
Proceedings of the International Seminar on Scientific Supercomputers, Paris, 2-6 Février 1987, p.183-199.

[Klein76] **L. Kleinrock**
"Queueing Systems Vol. 2: Computer Applications"
Wiley, 1976.

[Koury84] **R. Koury, D.F. McAllister, W.J. Stewart**
"Methods for Computing Stationary Distributions of Nearly Completely Decomposable Markov Chains"
SIAM J. Alg. Disc. Math., Vol. 5, No. 2, 1984, p.164-186.

[Leca87] **P. Leca**
"Conception d'algorithmes parallèles pour des systèmes multiprocesseurs à mémoires distribuées: application à la programmation du système multi-FPS164 lCAP1"
International Seminar on Scientific Supercomputers, Paris, 1987.

[Lenfa] **J. Lenfant**
"Comparaison des ensembles de travail et des intervalles bornés de localité"
RAIRO Informatique, Vol. 12, No. 1, p.15-35.

[Logan88] **D. Logan, J. Saarinen, E. Clementi**
"lCAP/3090: Genesis and Evolution of a Parallel Processing System"
IBM Kingston Research Report, KGN-173, 18 Février 1988
Third International Conference on Supercomputing, Boston, Mai 1988.

[Lubac84] **B.D. Lubachevsky, K.G. Ramakrishnan**
"Parallel Time-Driven Simulation of a Network on a Shared Memory MIMD Computer"
Proceedings of the International Conference on Modelling Techniques and Tools for Performance Analysis, Paris, 16-18 Mai 1984.

[Lubec88] **O.M. Lubeck, V. Faber**

"Modeling the Performance of Hypercubes: A Case Study Using the Particle-In-Cell Application"

Parallel Computing, Vol. 9, No. 1, Décembre 1988, p.37-52.

[Marie79] **R. Marie**

"An Approximative Analytical Method for General Queueing Networks"

I.E.E.E. Transactions on Software Engineering, Vol. 5, Mai 1979, p.503-538.

[Mazel88] **C. Mazel**

"Evaluation des performances par simulations. Application aux canaux de signalisation de systèmes radiotéléphoniques"

Thèse de Docteur-Ingénieur, Institut National Polytechnique de Grenoble, France, 30 Juin 1988.

[McBry87] **O. McBryan**

"New Architectures: Performance Highlights and New Algorithms"

Proceedings of the 2nd International SUPRENUM Colloquium, Bonn, 30 Septembre - 2 Octobre 1987, Parallel Computing, Vol. 7, No. 3, Septembre 1988, p.477-499.

[Mink88] **A. Mink, J.M. Draper, J.W. Roberts, R.J. Carpenter**

"Hardware-Assisted Multiprocessor Performance Measurement"

Performance '87, North Holland, P.-J. Courtois, G. Latouche, eds., 1988, p.151-168.

[Nacht88] **G. Nacht, A. Mink**

"A Hardware Instrumentation Approach for Performance Measurement of a Shared Memory Multiprocessor"

Proceedings of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Palma de Mallorca, 14-16 Septembre 1988, p.321-339.

[PenCh87] **Pen Chung Yen, Nian Feng Tzenq, D.H. Lawrie**

"Distributing Hot Spot Addressing in Large Scale Multiprocessors"

I.E.E.E. Transactions on Computers, Vol. 36, No. 4, Avril 87.

[Plate84] **B. Plateau**
"De l'évaluation du parallélisme et de la synchronisation"
Thèse d'Etat, Université de Paris Sud, Orsay, 28 Novembre 1984.

[Popek81] **G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel**
"LOCUS: A Network Transparent, High Reliability Distributed System"
Proceedings of the 8th Symposium on Operating System Principles, Pacific Grove, California, Décembre 1981.

[Press88] **L. Press**
"Benchmarks for LAN Performance Evaluation"
Communications of the ACM, Vol. 31, No. 8, Août 1988, p.1014-1017.

[Prost86] **J.-P. Prost**
"Modélisation et évaluation des performances de réseaux point-à-point entre systèmes HP3000"
Mémoire de D.E.A. en Informatique, Institut National Polytechnique de Grenoble, Septembre 1986.

[Prost88a] **J.-P. Prost, J. Detrich, M. Becker**
"A Simulator of the Parallel Scheduler for the ICAP System"
IBM Kingston Research Report, KGN-54, 16 Mai 1988
Deuxième Conférence sur l'Ingénierie de Performance et la Technologie de l'Information, Computer Measurement Group Francophone, Nice, 10-12 Mai 1989.

[Prost88b] **J.-P. Prost, J. Detrich, M. Becker**
"Cost Characterization of the Precompiler Communication and Synchronization Directives on the ICAP System"
IBM Kingston Research Report, KGN-73, 19 Mai 1988.

[Prost88c] **J.-P. Prost, M. Becker**
"Implementation of Parallel Algorithms on an IBM PC Local Area Network"
Invited paper in Franco-American Workshop on New Research Directions in Integrated Networks, I.N.R.I.A. Sophia Antipolis, 22-24 Juin 1988.

[Prost89a] **J.-P. Prost, J. Detrich, D. Folsom, M. Becker**
"Modeling the ICAP/FPS264 Parallel Architecture for Evaluating its Performance"
IBM Kingston Research Report, KGN-178, 24 Janvier 1989.

[Prost89b] **J.-P. Prost, J. Detrich, D. Folsom, M. Becker**
"A Performance Evaluation Model for the ICAP/FPS264 Parallel System"
20th Annual Modeling and Simulation Conference, Pittsburgh, 4-5 Mai 1989.

[Prost89c] **J.-P. Prost, J. Detrich, D. Folsom, M. Becker**
"Modeling and Performance Evaluation of Parallel Architectures"
Invited Paper in ACM 1989 International Conference on Supercomputing, Crète, Grèce, 5-9 Juin 1989.

[Prost89d] **J.-P. Prost, M. Becker**
"Modeling Methodology for Performance Evaluation of Parallel Architectures: A Case Study, ICAP"
A paraître dans un numéro spécial de "International Journal of High Speed Computing", D. Gannon, A. Lichnewsky, Y. Muraoka, A. Sameh, eds., Janvier 1990.

[Reise80] **M. Reiser, S. Lavenberg**
"Mean Value Analysis of Closed Multichain Queueing Networks"
J.A.C.M, No. 27, Avril 1980, p.313-322.

[Rusch82] **M. Ruschitzka**
"Performance Evaluation of Computer Systems and Networks and its Techniques"
Proceedings of the 10th IMACS World Congress on System Simulation and Scientific Computation, Montréal, 8-13 Août 1982, Vol. 4, p.231-232.

[Russo87a] **M. Russo, A. Perez-Ambite, R. Caltabiano, J. Detrich, D. Folsom**
"An Approach to Parallel Scheduling for the ICAP System"
IBM Kingston Research Report, KGN-135, 17 Juin 1987.

[Russo87b] **M. Russo, A. Perez-Ambite, S. Cauberghs, J. Detrich, D. Folsom**
"Simulation of Parallel Scheduling for the ICAP System"
IBM Kingston Research Report, KGN-136, 24 Juin 1987.

[Salsb88] **M.A. Salsburg**
"Simulation is Not a Four Letter Word"
Computer Measurement Group 1988 International Conference on Management and Performance Evaluation of Computer Systems, Dallas, 12-16 Décembre 1988, p.128-139.

[Sarge82] **R.G. Sargent**
"Model Validation: The Current State"
Proceedings of the 10th IMACS World Congress on System Simulation and Scientific Computation, Montréal, 8-13 Août 1982, Vol. 2, p.112-114.

[Sauer86] **C.H. Sauer, A.M. Blum, P.G. Loewner, E.A. McNair, J.F. Kurose**
"The Research Queueing Package - Version 2: CMS Reference Manual"
IBM Research Report, Yorktown Heights, Novembre 1986.

[Schru87] **L. Schruben, V.J. Cogliano**
"An Experimental Procedure for Simulation Response Surface Model Identification"
Communications of the ACM, Vol. 30, No. 8, 1987, p.716-730.

[Schwe79] **P. Schweitzer**
"Approximate Analysis of Multiclass Closed Networks of Queues"
International Conference on Stochastic Control and Optimization, Amsterdam, 1979.

[Shapi79] **S.D. Shapiro**
"A Stochastic Petri Net with Application to Modeling Occupancy Times for Concurrent Task Systems"
Networks, Vol. 9, 1979.

[Simon61] **H.A. Simon, A. Ando**
"Aggregation of Variables in Dynamic Systems"
Econometrica 29, 1961, p.111-138.

[Smith82] **C.U. Smith, D.D. Loendorf**
"Performance Analysis of Software for an MIMD Computer"
Performance Evaluation Review, Vol. 11, No. 4, ACM SIGMETRICS, Septembre 1982, p.151-162.

[Som88] **T.K. Som, R.G. Sargent**
"*Application of Frequency Domain Experiments to Simulations of Computer Systems*"
Performance '87, North Holland, P.-J. Courtois, G. Latouche, eds., 1988, p.115-130.

[Spep] **Hewlett-Packard**
"*System Performance Evaluation Project Reference Specification*"

[Stoer83] J. Stoer, R. Bulirsch
"*Introduction to Numerical Analysis*"
Springer-Verlag, 1983, p.545-554.

[Takah75] **Y. Takahashi**
"*A Lumping Method for Numerical Calculations of Stationary Distributions of Markov Chains*"
Research Report B-18, Department of Information Sciences, Tokyo Institute of Technology, Juin 1975.

[Tchue85] **B. Tchuente Kemdjo**
"*Application des méthodes de classification et d'agrégation à l'évaluation des performances des systèmes informatiques*"
Mémoire de D.E.A. en Informatique, Institut National Polytechnique de Grenoble, Septembre 1985.

[Tepe] **Hewlett-Packard**
"*TEPE/3000 User's Guide*"

[Trele88] **P.C. Treleaven**
"*Parallel Architecture Overview*"
Parallel Computing, Vol. 8, Nos. 1-3, Octobre 1988, p.59-70.

[Trico84] **M. Tricot, J.F. Ballif, J.R. Barra, M. Becker**
"*Application d'un algorithme de classification à un problème d'agrégation pour la résolution d'un modèle de systèmes informatiques*"
Rapport de Recherche IMAG No. 451, Juin 1984.

[Vanti81] **H. Vantilborgh**

"The Error of Aggregation. A Contribution to the Theory of Decomposable Systems and Applications"

Thèse de Docteur en Sciences Appliquées, Faculté des Sciences Appliquées, Université Catholique de Louvain, Belgique, 1981.

[Véran84] **M. Véran, D. Potier**

"A Portable Environment for Queueing Networks Modelling"

International Conference on Modelling Techniques and Tools for Performance Analysis, Paris, Mai 1984.

[Wang81] **R.T. Wang, J.C. Browne**

"Virtual Machine-Based Simulation of Distributed Computing and Network Computing"

ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Las Vegas, 14-16 Septembre 1981, p.154-156.

[Wasse88] **H.J. Wasserman, M.L. Simmons, O.M. Lubeck**

"The Performance of Minisupercomputers: Alliant FX/8, Convex C-1, and SCS-40"

Parallel Computing, Vol. 8, Nos. 1-3, Octobre 1988, p.285-293.

[Woodb88] **M.H. Woodbury, K.G. Shin**

"Performance Modeling and Measurement of Real-Time Multiprocessors with Time-Shared Buses"

IEEE Transactions on Computers, Vol. 37, No. 2, Février 1988, p.214-224.

[Wybra88] **D. Wybranietz, D. Haban**

"Monitoring and Performance Measuring Distributed Systems During Operation"

Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Santa Fé, 24-27 Mai 1988, p.197-206.

[Zahor81] **J. Zahorjan, E. Wong**

"The Solution of Separable Queuing Network Models Using Mean Value Analysis"

ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Las Vegas, 14-16 Septembre 1981.

ANNEXES

ANNEXE A

Structure Générale du Simulateur du Système lCAP

PROGRAMME PRINCIPAL LCAPSIM:

Le programme principal du simulateur du système lCAP gère la liste des événements. Pour chaque événement de la liste, pris chronologiquement, il interprète l'événement en effectuant le traitement adéquat, il met à jour l'interface graphique si nécessaire, il passe à l'événement suivant et réitère ce processus jusqu'à la fin de la simulation, caractérisée par la fin de la simulation du job principal (appelé *job0*) pour la *job0_nb_simul* ième fois.

Le programme principal a accès à l'ensemble des procédures et variables exportées par les unités, dont la structure générale suit celle du programme principal.

```
program lcapsim;

{Liste des unites utilisees par le programme principal}
uses  crt,global,apsched,host,channel,aps,bulk,graphics;

procedure main_interpret (ident : event_type);

{Cette procedure interprete l'evenement courant
 en lancant la procedure qui correspond a son type;
 l'ensemble de ces procedures est defini dans les
 unites modelisant les stations de service du modele;
 l'interpretation de l'evenement activera si necessaire
 l'interface graphique desactive a priori}
```

```
begin
      case ident of
         ht_get_event    : host_get_event;
         ht_computation  : host_computation;
         ht_external     : host_external;
         ht_exec         : host_exec;
         ht_go_data      : host_go_data;
         ht_wait         : host_wait;
         ht_results      : host_results;
         sc_clenup       : apsch_clenup;
         sc_start        : apsch_start;
         sc_new_user     : apsch_new_user;
         sc_wait_master  : apsch_wait_master;
         sc_master_go    : apsch_master_go;
         sc_rolled_in    : apsch_rolled_in;
         sc_inquiry      : apsch_inquiry;
         sc_rolled_out   : apsch_rolled_out;
         sc_finish       : apsch_finish;
         sc_detach_user  : apsch_detach_user;
         ch_data         : chan_data;
         ch_results      : chan_results;
         ap_get_event    : aps_get_event;
         ap_rollin       : aps_rollin;
         ap_executing    : aps_executing;
         ap_computing    : aps_computing;
         ap_rollout      : aps_rollout;
         ap_go_results   : aps_go_results;
         ap_barrier      : aps_barrier;
         ap_first_move   : aps_first_move;
         ap_next_move    : aps_next_move;
         ap_send         : aps_send;
         ap_receive      : aps_receive;
         bm_barrier      : bulk_barrier;
```

```
        bm_move         : bulk_move;
        bm_send         : bulk_send;
        bm_receive      : bulk_receive
    end
end;

begin
    while not end_simulation do
        {Ce n'est pas la fin de la simulation}
        begin
            {Desactive l'interface graphique a priori}
            view_it := false;
            {Determine le temps simule courant, nul au debut de la simulation}
            cur_time := cur_event^.date;
            {Calcule la date courante}
            glob_date_add (init_date,cur_date,cur_time);
            {Si le mode de debugging a ete active par l'utilisateur,
             enregistre une trace de toutes les caracteristiques
             de l'evenement courant}
            if debug then glob_debug;
            {Interprete l'evenement en fonction de son type}
            main_interpret (cur_event^.ident);
            {Met a jour l'interface graphique, si le mode graphique
             est active et si l'interpretation de l'evenement requiert
             cette mise a jour}
            if graphic and view_it then gr_update;
            {Passe a l'evenement suivant}
            glob_delete
        end;
    {C'est la fin de la simulation: ferme les
     fichiers de resultats et l'interface graphique}
    close (screen);
    close (outputfile);
```

```
        close (bulkfile);
        close (resultfile);
        if trace then close (tracefile);
        if debug then close (debugfile);
        if graphic then gr_close
end.
```

UNITE GLOBAL:

Cette unité, accessible de toutes les autres unités, contient toutes les structures de données globales, dont notamment celles représentant l'architecture du système ICAP et celles représentant les programmes d'application. Cette unité réalise également l'initialisation du simulateur en appelant la procédure locale *glob_init_simul*.

```
unit global;

interface

{La partie 'interface' contient d'une part la liste des unites utilisees par
 cette unite, d'autre part la declaration de toutes les constantes, types,
 variables, procedures et fonctions exportes par cette unite et accessibles
 par l'ensemble des unites, qui utilisent cette unite, et par le programme
 principal, s'il utilise cette unite (ce qui est le cas presentement)}

uses  crt;

{Structures de donnees globales,
 accessibles depuis toutes les autres unites;
 seules les plus importantes sont commentees}

const
   {Nombre maximal de jobs sequentiels
    et de sessions interactives sur l'hote simulees}
      maxi_external_jobs              = 15;
   {Nombre maximal de jobs paralleles simules}
      maxi_studied_jobs               = 15;
   {Nombre maximal de classes de priorite de jobs}
      maxi_system_queues              =  5;
   {Nombre maximal de jobs par classe de priorite}
      maxi_queue_jobs                 = 10;
```

```
{Facteur maximal de vitesse de calcul de l'hote}
   maxi_host_processor_speed_ratio = 10;
{Facteur maximal de vitesse de transfert des canaux hote-APs}
   maxi_channel_speed_ratio        = 10;
{Nombre maximal d'APs attaches a un meme job;
 c'est egalement le nombre maximal d'APs presents sur le systeme}
   maxi_job_aps                    = 20;
{Nombre maximal d'utilisateurs que chaque AP peut prendre en compte
 successivement par le mecanisme de "rollin/rollout"}
   maxi_ap_users                   =  7;
{Facteur maximal de vitesse de calcul des APs}
   maxi_ap_speed_ratio             = 10;
{Nombre maximal de memoires de masse presentes sur le systeme}
   maxi_system_bulks               = 12;
{Nombre maximal d'utilisateurs,
 soit de partitions memoire actives, par memoire de masse}
   maxi_bulk_users                 = 20;
{Nombre maximal de reseaux que l'on
 peut definir pour un meme job en mode message}
   maxi_job_networks               =  4;
{Facteur maximal de vitesse d'acces des memoires de masse}
   maxi_bulk_access_ratio          = 20;
{Nombre maximal de programmes d'application}
   maxi_program_structures         = 20;
{Longueur maximale d'une chaine de programme, maitre ou esclave}
   maxi_program_length             = 40;
{Nombre maximal de categories de programmes d'application}
   maxi_basic_prog_str             =  4;
{Niveau maximal d'imbrication pour les boucles}
   maxi_nesting_levels             =  3;
```

```
    {Constantes utilisees par le generateur de nombres aleatoires}
       seed : longint      = -157787;
       m    : longint      =  714025;
       a    : longint      =    1366;
       c    : longint      =  150889;
       maxi_discrete_range =      20;

    {Temoin de mise a jour de l'interface graphique}
       view_it : boolean = false;

var
    {Variables utilisees par le generateur de nombres aleatoires}
       rand   : array [1..97] of longint;
       interm : longint;
       rm     : double;

const
       int0 =   0;
       db10 = 0.0;

type
    {Definition d'intervalles de variation pour les variables de boucles}
       job_range0  = 0..maxi_studied_jobs;
       job_range1  = 1..maxi_studied_jobs;
       job_range2  = -maxi_external_jobs..maxi_studied_jobs;
       ap_range0   = 0..maxi_job_aps;
       ap_range1   = 1..maxi_job_aps;
       bulk_range0 = 0..maxi_system_bulks;
       bulk_range1 = 1..maxi_system_bulks;

    {Codification d'une date: MM/JJ/AAhh:mm:ss.ssssss}
       date_type = string[23];
```

```
{Differents types d'evenements}
   event_type = (ht_get_event,    {lecture du prochain caractere ou groupe
                                   de caracteres dans la chaine de programme
                                   du maitre}
                 ht_computation, {calcul hote}
                 ht_external,    {traitement d'un job non parallele}
                 ht_exec,        {directive EXECUTE ON}
                 ht_go_data,     {initialisation d'un transfert de donnees
                                   hote -> AP}
                 ht_wait,        {directive WAIT}
                 ht_results,     {reception de resultats en provenance d'un
                                   AP}
                 ht_end,         {fin du job parallele}
                 sc_clenup,      {operation de "clean up" (cf unite APSCHED)}
                 sc_start,       {directive START}
                 sc_new_user,    {fin de l'initialisation d'un job parallele}
                 sc_wait_master, {AP pret a executer une nouvelle AProutine}
                 sc_master_go,   {maitre desirant lancer sur les APs une
                                   nouvelle AProutine}
                 sc_rolled_in,   {fin d'une operation de "rollin"}
                 sc_inquiry,     {fin de tranche de temps AP}
                 sc_rolled_out,  {fin d'une operation de "rollout"}
                 sc_finish,      {directive FINISH}
                 sc_detach_user, {liberation des APs}
                 ch_data,        {transfert de donnees hote -> AP}
                 ch_results,     {transfert de donnees AP -> hote}
                 ap_get_event,   {lecture du prochain caractere ou groupe
                                   de caracteres dans la chaine de programme
                                   de l'esclave courant}
                 ap_rollin,      {debut d'une operation de "rollin"}
                 ap_executing,   {lancement de l'execution d'une nouvelle
                                   AProutine sur un AP}
                 ap_computing,   {calcul sur un AP}
```

```
                ap_rollout,     {debut d'une operation de "rollout"}
                ap_go_results,  {initialisation d'un transfert de donnees
                                 AP -> hote}
                ap_detach,      {liberation d'un AP}
                ap_barrier,     {directive BARRIER}
                ap_first_move,  {directive MOVE: premier bloc de donnees
                                                 contigues}
                ap_next_move,   {directive MOVE: blocs suivants}
                ap_send,        {directive SEND}
                ap_receive,     {directive RECEIVE}
                bm_barrier,     {synchronisation de type BARRIER}
                bm_move,        {acces a une memoire de masse partagee}
                bm_send,        {envoi d'un message}
                bm_receive);    {reception d'un message}

const
      event_string : array [event_type] of string[14] =
                     ('ht_get_event',
                      'ht_computation',
                      'ht_external',
                      'ht_exec',
                      'ht_go_data',
                      'ht_wait',
                      'ht_results',
                      'ht_end',
                      'sc_clenup',
                      'sc_start',
                      'sc_new_user',
                      'sc_wait_master',
                      'sc_master_go',
                      'sc_rolled_in',
                      'sc_inquiry',
                      'sc_rolled_out',
```

```
                    'sc_finish',
                    'sc_detach_user',
                    'ch_data',
                    'ch_results',
                    'ap_get_event',
                    'ap_rollin',
                    'ap_executing',
                    'ap_computing',
                    'ap_rollout',
                    'ap_go_results',
                    'ap_detach',
                    'ap_barrier',
                    'ap_first_move',
                    'ap_next_move',
                    'ap_send',
                    'ap_receive',
                    'bm_barrier',
                    'bm_move',
                    'bm_send',
                    'bm_receive');

type
   {Definition d'un evenement;
    seules les trois premieres caracteristiques
    ainsi que la derniere ont toujours une signification}
      event_ptr = ^event;
      event     = record
                      ident      : event_type;  {type}
                      date       : double;      {instant d'occurrence}
                      job_nb     : job_range2;  {numero du job courant}
                      ap1_nb     : ap_range0;   {numero du 1er AP associe}
                      ap2_nb     : ap_range0;   {numero du 2eme AP associe}
                      time       : double;      {demande de service}
```

```
                length      : double;       {longueur des donnees associees}
                next_event  : event_ptr     {pointeur sur l'evenement
                                             suivant dans la liste}
            end;

   result_type = (current,
                  mean);

{Raisons pouvant engendrer la fin du job courant}
   info_reason = (kill,      {destruction du job en cours de traitement}
                  clean,     {destruction du job des sa soumission}
                  finish);   {terminaison normale du job}

{Differents types de distributions de probabilite supportees}
   distribution_type = (standard,  {constantes, exponentielles,
                                    hyperexponentielles, hypoexponentielles}
                        uniform,   {uniformes, reelles ou entieres}
                        discrete); {discretes (tableau de probabilites)}

{Type d'un tableau de probabilites}
   proba_array_type = array [1..maxi_discrete_range] of double;

{Definition des distributions de probabilite supportees}
   distribution =
      record
         case distribution_nature : distribution_type of
            standard : (mean       : double;     {moyenne}
                        coeff_var  : double);    {coefficient de variation}
            uniform  : (min_unif     : double;   {borne inferieure}
                        max_unif     : double;   {borne superieure}
                        is_discrete  : boolean); {intervalle reel ou entier}
            discrete : (min_discr : integer;     {borne inferieure}
                        max_discr : integer;     {borne superieure}
```

```
                        proba      : proba_array_type)
                                                {tableau de probabilites}
      end;

{Definition de l'hote}
   host_type =
      record
      {facteur de vitesse de calcul}
         processor_speed  : 1..maxi_host_processor_speed_ratio;
      {duree de la tranche de temps}
         time_slice       : double;
      {taux d'occupation par les jobs non paralleles, le job principal
       et les jobs d'arriere-plan respectivement}
         cur_utilization  : array [-1..1] of double;
            {lors de la simulation courante du job principal}
         mean_utilization : array [-1..1] of double
            {moyens sur l'ensemble des simulations du job principal}
      end;

{Definition d'un canal hote-AP}
   channel_type =
      record
      {facteur de vitesse de transfert}
         rate             : 1..maxi_channel_speed_ratio;
      {taux d'occupation par le job principal
       et les jobs d'arriere-plan respectivement}
         cur_utilization  : array [0..1] of double;
         mean_utilization : array [0..1] of double
      end;
```

```
{Definition d'un AP}
   ap_type =
      record
      {facteur de vitesse de calcul}
         speed             : 1..maxi_ap_speed_ratio;
      {duree de la tranche de temps}
         time_slice        : double;
      {nombre courant d'utilisateurs}
         nb_active_users   : 0..maxi_ap_users;
      {numero courant du job qu'il execute}
         running_job_nb    : job_range0;
      {AP sur le point d'etre arrete par l'operateur}
         is_not_closing    : boolean;
      {etat logique de l'AP: libre, occupe ou non disponible}
         logical_status    : (logical_free,
                              logical_executing,
                              non_available);
      {etat effectif de l'AP: libre ou occupe}
         actual_status     : (actual_free,
                              actual_executing);
      {taux d'occupation par le job principal
       et les jobs d'arriere-plan respectivement}
         cur_utilization   : array [0..1] of double;
         mean_utilization  : array [0..1] of double
      end;

{Definition d'une memoire de masse}
   bulk_type =
      record
      {facteur de vitesse d'acces}
         access_ratio      : 1..maxi_bulk_access_ratio;
      {capacite}
         size              : longint;
```

```
      {nombre courant de partitions memoire actives}
         nb_users          : 0..maxi_bulk_users;
      {numero du job associe a chaque partition active}
         user_nb           : array [1..maxi_bulk_users] of job_range0;
      {etat de chaque partition active: reservee ou en cours d'utilisation}
         user_status       : array [1..maxi_bulk_users] of (not_using,
                                                            using);
      {adresse de base de chaque partition active}
         starting_address  : array [1..maxi_bulk_users] of longint;
      {taille de chaque partition active}
         length            : array [1..maxi_bulk_users] of longint;
      {temps d'occupation par le job principal
       lors de la simulation courante du job principal}
         cur_service_time  : double;
      {temps d'attente du job principal, du aux contentions d'acces,
       lors de la simulation courante du job principal}
         cur_waiting_time  : double;
      {temps de service moyen par le job principal,
       sur l'ensemble des simulations du job principal}
         mean_service_time : double;
      {temps d'attente moyen du job principal, du aux contentions d'acces,
       sur l'ensemble des simulations du job principal}
         mean_waiting_time : double;
      {taux d'occupation par le job principal
       et les jobs d'arriere-plan respectivement}
         cur_utilization   : array [0..1] of double;
         mean_utilization  : array [0..1] of double
      end;

{Differents modes d'utilisation des memoires de masse}
   bulk_mode_type = (none,      {non utilisees}
                     shared,    {mode partage}
                     message);  {mode message}
```

```
const
      bulk_mode_string : array [bulk_mode_type] of string[7] =
                         ('none',
                          'shared',
                          'message');

type
   {Definition du systeme lCAP/FPS264}
      system_type =
         record
         {hote}
            host                    : host_type;
         {canaux hote-APs}
            channel                 : array [ap_range1] of channel_type;
         {APs}
            ap                      : array [ap_range1] of ap_type;
         {nombre courant d'APs presents sur le systeme}
            cur_aps                 : ap_range1;
         {nombre maximal courant d'utilisateurs par AP}
            cur_maxi_ap_users       : 1..maxi_ap_users;
         {nombre courant d'APs disponibles}
            cur_available_aps       : ap_range0;
         {nombre courant d'APs qui ne sont pas
          sur le point d'etre arretes par l'operateur}
            cur_non_closing_aps     : ap_range0;
         {memoires de masse}
            bulk                    : array [bulk_range1] of bulk_type;
         {nombre courant de memoires de masse presentes sur le systeme}
            cur_bulks               : bulk_range1;
         {reseau d'interconnexion entre les APs et les memoires de masse}
            configuration           : array [bulk_range1,
                                             ap_range1] of bulk_range0;
```

```
      {nombre courant de classes de priorite de jobs}
         cur_system_queues      : 1..maxi_system_queues;
      {nombre maximal courant de jobs par classe de priorite}
         cur_maxi_queue_jobs    : 1..maxi_queue_jobs;
      {nombre courant de jobs non paralleles presents dans le systeme}
         cur_external_jobs      : 0..maxi_external_jobs;
      {nombre courant de jobs paralleles presents dans le systeme}
         cur_studied_jobs       : job_range0;
      {nombre courant de programmes d'application definis}
         cur_program_structures : 0..maxi_program_structures;
      {nombre courant de categories de programmes d'application}
         cur_basic_prog_str     : 1..maxi_basic_prog_str;
      {niveau maximal d'imbrication courant pour les boucles}
         cur_nesting_levels     : 1..maxi_nesting_levels;
      {simulation avec charge ou sans charge}
         background_jobs        : boolean;
      {nombre courant de jobs d'arriere-plan}
         cur_background_jobs    : byte
      end;

{Definition d'un job non parallele:
 execution d'un programme sequentiel ou session interactive}
   external_job_ptr  = ^external_job_type;
   external_job_type =
      record
      {loi de distribution des temps de calcul sur l'hote}
         computation : distribution;
      {loi de distribution des temps de reflexion utilisateur
       pour les sessions interactives}
         think_time  : distribution
      end;
```

```
   {Differents etats possibles d'un job parallele}
      job_status_type = (enqueued,            {en cours de soumission}
                         ready,               {pret a s'executer}
                         non_dispatchable,    {ne pouvant acceder a un AP ayant
                                               atteint son nombre d'utilisateurs
                                               maximal}
                         ordered_rolled_in,   {en attente de "rollin"}
                         rolling_in,          {en cours de "rollin"}
                         executing,           {en train de s'executer}
                         ordered_rolled_out,  {en attente de "rollout"}
                         rolled_out,          {"rolloute"}
                         finishing);          {en cours de terminaison}

const
      job_status_string : array [job_status_type] of string[10] =
                           ('Enqueued',
                            'Ready',
                            'Non Dispat',
                            'O.Roll In',
                            'Rolling In',
                            'Executing',
                            'O.Roll Out',
                            'Rolled Out',
                            'Finishing');

type
   {Types de donnees necessaires a la generation aleatoire
    de directives SEND et RECEIVE correlees}
      posted_send_type =
         record
            length : double;
            delay  : byte
         end;
```

```
  posted_receive_ptr  = ^posted_receive_type;
  posted_receive_type =
     record
        length : double;
        delay  : byte;
        next   : posted_receive_ptr
     end;

{Definition d'un job parallele}
   studied_job_ptr  = ^studied_job_type;
   studied_job_type =
      record
      {job d'arriere-plan ou non}
         background                          : boolean;
      {identification de l'utilisateur ayant soumis le job}
         user_id                             : string[6];
      {numero de la classe de priorite du job}
         class_nb                            : 1..maxi_system_queues;
      {nombre d'APs attaches}
         nb_aps                              : ap_range1;
      {numero de chaque AP attache}
         ap_nb                               : array [ap_range1] of ap_range0;
      {requete specifique ou non}
         specific_request                    : boolean;
      {date et heure de soumission}
         submitting_time                     : date_type;
      {temps d'execution estime par l'utilisateur}
         expected_run_time                   : double;
         initial_expected_run_time           : double;
      {valeur courante du poids dynamique du job}
         weight                              : double;
```

```
{etat du job}
   status                              : job_status_type;
{job eligible ou non: propre a la politique d'allocation des APs}
   is_eligible                         : boolean;
{job penalise ou non}
   is_penalized                        : boolean;
{numero du programme d'application que le job execute}
   program_structure_nb                : 0..maxi_program_structures;
{chaines des programmes maitre et esclaves respectivement}
   program_string                      : array [ap_range0]
                                         of string[maxi_program_length];
{pointeur courant dans chaque chaine de programme maitre ou esclave}
   program_ptr                         : array [ap_range0]
                                         of 0..maxi_program_length;
{nombre courant d'iterations restant a effectuer a chaque niveau
 d'imbrication de boucle par le maitre et chaque esclave}
   nb_iter                             : array [ap_range0,
                                                1..maxi_nesting_levels]
                                         of word;
{niveau d'imbrication de la boucle courante
 du maitre et de chaque esclave}
   cur_nesting_level                   : array [ap_range0]
                                         of 0..maxi_nesting_levels;
{taille du code et des donnees des APRoutines}
   code_and_data_size                  : double;
   cur_aproutine_arg_send_init_time    : double;
{temps d'execution courant du job}
   cur_run_time                        : double;
{temps d'attente courant du job}
   cur_waiting_time                    : double;
   accumulated_run_time                : double;
   accumulated_waiting_time            : double;
```

```
{premier "rollin" ou non}
   is_first_rollin                    : boolean;
{instant d'occurrence du dernier "rollin"}
   last_rollin_time                   : double;
{instant d'occurrence du dernier "rollout"}
   last_rollout_time                  : double;
   is_waiting_and_actual_rollin       : array [ap_range1] of boolean;
   remaining_computation_time         : array [ap_range1] of double;
{nombre d'esclaves non encore arrives a un point de synchronisation
 correspondant a une directive WAIT}
   nb_wait_slaves                     : word;
{nombre d'esclaves non encore arrives a un point de synchronisation
 correspondant a une directive BARRIER}
   nb_barrier_slaves                  : ap_range0;
{esclave en attente a un point de synchronisation ou non}
   is_waiting                         : array [ap_range1] of boolean;
{prochain message de type INQUIRY deja enregistre ou non}
   is_next_inquiry_ordered            : array [ap_range1] of boolean;
{instant du debut de la tranche de temps courante pour chaque AP}
   last_inquiry_time                  : array [ap_range1] of double;
   nb_char_to_be_transferred          : array [ap_range1] of double;
{temps de reponse du systeme pour l'execution du job}
   response_time                      : double;
{execution de l'AProutine terminee ou non}
   slave_program_finished             : array [ap_range1] of boolean;
   case bulk_mode                     : bulk_mode_type of
   {mode partage: numero de la memoire de masse partagee}
      shared  : (bulk_shared_nb             : bulk_range0);
   {mode message}
      message :
               {numero du reseau courant defini par l'utilisateur}
                 (cur_network                : 0..maxi_job_networks;
```

```
                {chemins de donnees en emission de message}
                   send_path                : array [ap_range1,
                                                    1..maxi_job_networks,
                                                    ap_range1]
                                              of bulk_range0;
                {chemins de donnees en reception de message}
                   receive_path             : array [ap_range1,
                                                    1..maxi_job_networks,
                                                    ap_range1]
                                              of bulk_range0;
                {variables necessaires a la generation aleatoire de
                 directives SEND et RECEIVE correlees}
                   posted_send_full         : array [ap_range1,ap_range1]
                                              of boolean;
                   posted_send              : array [ap_range1,ap_range1]
                                              of posted_send_type;
                   posted_receive           : array [ap_range1,ap_range1]
                                              of posted_receive_ptr;
                   wait                     : array [ap_range1]
                                              of ap_range0;
                   nb_sent_messages         : array [ap_range1,ap_range1]
                                              of byte;
                   expected_receive_length : array [ap_range1,ap_range1]
                                              of double)
      end;

{Definition d'une classe de priorite de jobs}
   class_type =
      record
      {priorite de la classe}
         priority             : single;
      {temps maximal d'execution pour un job de la classe}
         maxi_run_time        : double;
```

```
      {seuil de "non-rollout" pour un job de la classe}
         no_rollout_run_time  : double;
      {loi de distribution des interarrivees des jobs de la classe}
         job_inter_arrival    : distribution;
      {loi de repartition des nombres d'APs}
         nb_aps_proba         : array [ap_range1] of double;
      {loi de probabilite regissant la categorie d'appartenance
       du programme d'application execute par un job de la classe}
         basic_prog_str_proba : array [1..maxi_basic_prog_str] of double;
      {nombre courant de jobs de la classe}
         nb_jobs              : job_range0;
      {variables utilisees par l'interface graphique pour visualiser
       l'occupation de la file d'attente associee a la classe}
         last_update_time     : double;
         mean_occupation      : double;
         maxi_occupation      : double
      end;

{Definition d'un esclave}
   slave_type =
      record
      {etat courant de l'esclave}
         status          : (psim_not_used,
                            psim_ready,
                            psim_ready_to_rollin,
                            psim_executing,
                            psim_waiting_for_master,
                            psim_rolling_out,
                            psim_rolled_out,
                            job_rolled_out,
                            psim_free,
                            psim_waiting_ready_to_rollin,
                            psim_waiting_rolling_out,
```

```
                        psim_waiting_rolled_out,
                        psim_waiting_job_rolled_out);
      {le maitre a-t-il deja envoye le message MASTGO
       au gestionnaire de ressources}
         go_received     : (init,        {en debut d'execution: non}
                        yes,         {oui}
                        no);         {non}
      {l'AP est-il susceptible de "rollouter" le job
       a la fin de la tranche de temps courante}
         rollout_ordered : boolean;
         rqn_status      : (rqn_in,
                        rqn_out)
      end;

{Definition d'un programme d'application}
   program_structure_type =
      record
      {temps d'execution estime par l'utilisateur}
         expected_time : double;
      {programme maitre}
         master        : record
                     {chaine du programme maitre}
                        program_string : string[maxi_program_length];
                     {nombre d'iterations a effectuer a chaque niveau
                      d'imbrication pour les boucles}
                        nb_iterations  : array [1..maxi_nesting_levels]
                                         of distribution;
                     {distribution des temps de calcul de type 1}
                        computation_1  : distribution;
                     {distribution des temps de calcul de type 2}
                        computation_2  : distribution;
                     {nombre de parametres formels de l'AProutine esclave}
                        nb_arguments   : distribution;
```

```
                {taille du code et des donnees de l'AProutine esclave}
                   code_size       : distribution
                   end;
{AProutine esclave}
   slave           : record
                {chaine du programme esclave}
                   program_string : string[maxi_program_length];
                {nombre d'iterations a effectuer a chaque niveau
                 d'imbrication pour les boucles}
                   nb_iterations  : array [1..maxi_nesting_levels]
                                    of distribution;
                {distribution des temps de calcul}
                   computation    : distribution;
                   case bulk_mode : bulk_mode_type of
                   {mode partage}
                      shared  :
                              {probabilite pour un acces d'etre
                               une lecture en memoire partagee}
                                (read_percent : 1..100;
                              {longueur de chaque bloc de donnees
                               contigues}
                                 block_length : distribution;
                              {nombre de blocs non contigus de donnees}
                                 nb_blocks    : distribution);
                   {mode message}
                      message :
                              {probabilite pour une communication
                               d'etre un envoi de message}
                                (send_percent           : 1..100;
                              {longueur d'un message}
                                 message_length         : distribution;
```

```
                                        {delai, exprime en nombre de
                                         communications intermediaires,
                                         necessaire a la generation aleatoire de
                                         communications correlees entre esclaves}
                                           intermediate_transfers : distribution)
                              end
        end;

var
   {Date et heure, exprimees en temps simule, du debut de la simulation}
      init_date : date_type;
   {Date et heure courantes}
      cur_date  : date_type;
   {Temps simule courant; nul en debut de simulation}
      cur_time  : double;
   {Evenement en cours de traitement (tete de la liste des evenements)}
      cur_event : event_ptr;

   {Simulation terminee ou non;
    le critere de terminaison est donne dans le programme principal}
      end_simulation : boolean;

   {Systeme lCAP/FPS264}
      lcap : system_type;

   {Variables necessaires a la determination
    des temps de service a la station HT;
    les valeurs des variables reperees par un asterisque
    ont ete obtenues lors de la caracterisation des couts
    de communication et de synchronisation entre taches paralleles}
      host_free                  : double;
      incremental_new_user       : double;
      bulk_config_send_init_time : double;
```

```
      bulk_config_send_ap_time     : double;
      last_detach_user             : double;
      inter_clenup_time            : double;
{*}   conversion_host_cpu_time     : double;
{*}   argument_transfer_init_time : double;
      execute_init_time            : double;
{*}   data_transfer_init_time      : double;
      result_reception_cpu_time    : double;

   {Vitesse de transfert de reference des canaux hote-APs}
      base_channel_rate : double;

   {Duree maximale d'une operation de "rollin"}
      maxi_rollin_time        : double;
   {Duree maximale d'une operation de "rollout"}
      maxi_rollout_time       : double;
   {Temps necessaire a la conversion des donnees
    du format AP au format de l'hote}
      conversion_ap_cpu_time : double;

   {Variables necessaires a la determination
    des temps de service aux stations APi et BMj;
    les valeurs des variables reperees par un asterisque
    ont ete obtenues lors de la caracterisation des couts
    de communication et de synchronisation entre taches paralleles}
      bulk_free                        : array [bulk_range1] of double;
{*}   local_barrier_init_time          : double;
{*}   global_barrier_init_time         : double;
{*}   local_barrier_ap_time            : double;
{*}   global_barrier_ap_time           : double;
{*}   local_move_init_time             : double;
{*}   global_move_init_time            : double;
{*}   local_physical_move_init_time    : double;
```

```
{*}   global_physical_move_init_time : double;
{*}   local_send_init_time           : double;
{*}   global_send_init_time          : double;
{*}   send_word_time                 : double;
{*}   local_receive_init_time        : double;
{*}   global_receive_init_time       : double;
{*}   receive_word_time              : double;

   {Jobs non paralleles}
      external_job      : array [1..maxi_external_jobs] of external_job_ptr;
   {Jobs paralleles}
      studied_job       : array [job_range1] of studied_job_ptr;

   {Caracteristiques du job principal 'job0'}
      {numero de sa classe de priorite}
      job0_class_nb             : 1..maxi_system_queues;
      {nombre d'APs qui lui sont attaches}
      job0_nb_aps               : ap_range1;
      {numeros des APs qui lui sont attaches}
      job0_ap_nb                : array [ap_range1] of ap_range0;
      {nombre de simulations successives du job principal}
      job0_nb_simul             : 0..99;
      {nombre de simulations deja effectuees, incluant la simulation courante}
      job0_cur_nb_simul         : 0..99;
      {instant de la derniere soumission du job principal}
      job0_last_start_time      : double;

    {Variables intermediaires necessaires
     au calcul des criteres de performances du job principal}
      is_next_job_foreground    : boolean;
      first_slave_arriving      : boolean;
      first_slave_arriving_time : double;
      first_barrier_slave_time  : double;
```

```
   receiving_slave_time      : array [ap_range1] of double;
   bulk_nb_simul             : array [bulk_range1] of 0..99;

{Numeros des jobs presents dans la file d'attente
 associee a chaque classe de priorite}
   queue : array [1..maxi_queue_jobs,1..maxi_system_queues]
           of job_range0;
{Classes de priorite}
   class : array [1..maxi_system_queues] of class_type;

{Pourcentage du temps maximal d'execution d'un job
 au dela duquel il ne doit plus etre "rolloute"}
   no_rollout_run_time_percent : 1..100;
{Temps d'execution minimal
 pour qu'un job puisse etre "rolloute"}
   rollout_minimum_run_time    : double;
{Coefficients de la fonction de poids dynamique des jobs}
   param_a                     : single;
   param_b                     : single;
   param_c                     : single;

{Esclaves}
   slave : array [ap_range1,job_range1] of slave_type;

{Programmes d'application}
   program_structure : array [0..maxi_program_structures]
                       of program_structure_type;

{Criteres de performances calcules en fin de simulation;
 les variables reperees par un asteristique sont incluses
 dans la representation des resultats par histogramme}
   cur_response_time         : double;
   cur_waiting_time          : double;
```

```
      cur_computation_time      : array [ap_range0] of double;
      cur_channel_time          : array [ap_range0] of double;
      cur_bulk_time             : array [ap_range0] of double;
      cur_contention_time       : array [ap_range0] of double;
      cur_synchronization_time  : array [ap_range0] of double;
{*}   mean_response_time        : double;
{*}   mean_waiting_time         : double;
{*}   mean_computation_time     : array [ap_range0] of double;
{*}   mean_channel_time         : array [ap_range0] of double;
{*}   mean_bulk_time            : array [ap_range0] of double;
{*}   mean_contention_time      : array [ap_range0] of double;
{*}   mean_synchronization_time : array [ap_range0] of double;

   {Fichiers d'entree-sortie}
      screen     : text;
      inputfile  : text;
      configfile : text;
      jobfile    : text;
      outputfile : text;
      bulkfile   : text;
      resultfile : text;

      debug      : boolean;
      debugfile  : text;

      trace      : boolean;
      tracefile  : text;
```

```
{Liste des procedures exportees,
 auxquelles les autres unites,
 ainsi que le programme principal, ont acces}

procedure glob_date_diff (date1     : date_type;
                          date2     : date_type;
                          var time  : double);
   {Calcule le temps 'time' separant les deux dates 'date1' et 'date2'}

procedure glob_date_add (date1      : date_type;
                         var date2  : date_type;
                         time       : double);
   {Calcule la nouvelle date 'date2',
    obtenue en ajoutant le temps 'time' a la date 'date1'}

procedure glob_insert (ident    : event_type;
                       date     : double;
                       job_nb   : job_range2;
                       ap1_nb   : ap_range0;
                       ap2_nb   : ap_range0;
                       time     : double;
                       length   : double);
   {Insere dans la liste des evenements
    l'evenement dont les caracteristiques sont donnees par
    'ident', 'date', 'job_nb', 'ap1_nb', 'ap2_nb', 'time' et 'length'}

procedure glob_delete;
   {Supprime de la liste des evenements
    l'evenement courant, qui est la tete de liste}
```

```
procedure glob_read_bulk_mode (var inputfile : text;
                               var bulk_mode : bulk_mode_type);
   {Lit dans le fichier 'inputfile' le mode d'utilisation 'bulk_mode'
    des memoires de masse pour le job courant}

procedure glob_read_distribution (var inputfile : text;
                                  var distr     : distribution);
   {Lit dans le fichier 'inputfile'
    la distribution de probabilite 'distr'}

procedure glob_debug;
   {Si le mode de debugging est actif (i.e. si la variable 'debug'
    a la valeur 'true'), ecrit dans le fichier 'debugfile' une trace
    de toutes les caracteristiques de l'evenement courant}

function glob_uniform_0_1 : double;
   {Renvoie un nombre aleatoire compris dans l'intervalle [0,1[}

function glob_random (distr : distribution) : double;
   {Renvoie un nombre aleatoire qui satisfait a la
    distribution de probabilite definie par 'distr'}

procedure glob_print_simulation_results (result : result_type);
   {Calcule et edite dans le fichier 'resultfile'
    l'ensemble des criteres de performances obtenus
    lors de chaque simulation du job principal
    (la variable 'result' a alors la valeur 'current');
    a l'issue de la 'job0_nb_simul' ieme simulation
    du job principal (la variable 'result' a alors la
    valeur 'mean'), calcule la moyenne des criteres
    de performances sur l'ensemble des simulations
    du job principal}
```

```
{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}
implementation

{La partie 'implementation' contient la definition de toutes les procedures
 exportees, declarees precedemment, ainsi que celle des procedures locales,
 appelees par les procedures exportees ou utilisees dans la partie initiali-
 sation de l'unite (voir infra); elle comprend egalement la declaration des
 constantes, des types et des variables non exportes de niveau 0}

................................................................................
................................................................................
................................................................................

{==============================================================================}
procedure glob_init_simul;

{Procedure locale utilisee dans la partie initialisation de l'unite,
 dont la tache est d'initialiser la plupart des variables du simulateur,
 exportees par cette unite}

................................................................................
................................................................................
................................................................................

{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
begin

{Cette partie constitue la partie initialisation de l'unite; elle est
 executee, de meme que la partie initialisation de chaque unite utilisee
 par le programme principal, avant la premiere instruction du programme
 principal; l'ordre d'execution des parties initialisation des differentes
 unites utilisees par le programme principal est celui dans lequel ces
 unites apparaissent dans la clause 'uses' du programme principal}
```

```
        {Initialise le simulateur}
        glob_init_simul
end.
```

UNITE APSCHED:

Cette unité réalise la simulation du gestionnaire d'APs ("AP SCHEDuler"), l'une des deux machines virtuelles du gestionnaire de ressources (station de service *RM*). Chaque client déclenche, suivant sa classe, le lancement de la procédure appropriée.

```
unit apsched;

interface

uses  crt,global,bmmanag;

procedure apsch_clenup;
   {Operation de "clean up":
    realise a intervalles reguliers (dont la duree est specifiee par la
    variable 'inter_clenup_time') le calcul des temps d'execution courants
    des jobs paralleles presents dans le systeme pour appliquer le cas
    echeant la regle de penalisation;
    met a jour le temps d'attente des jobs presents dans le systeme et
    calcule leur fonction de poids dynamique;
    lance la politique d'allocation des APs pour determiner quels sont les
    jobs susceptibles d'etre "rollines" et quels sont ceux susceptibles
    d'etre "rolloutes"}

procedure apsch_start;
   {Traite chaque client de classe JOBSTR,
    representant l'arrivee d'un nouveau job
    parallele dans le systeme;
    appelle si necessaire la fonction
    'bmmanag_bulk_config_init' pour ce job;
    lance la politique d'allocation des APs}
```

```
procedure apsch_new_user;
   {Signifie la fin de l'initialisation du job courant;
    le job est maintenant pret a s'executer}

procedure apsch_wait_master;
   {Traite chaque client de classe WMASTR}

procedure apsch_master_go;
   {Traite chaque client de classe MASTGO}

procedure apsch_rolled_in;
   {Traite chaque client de classe INCOMP}

procedure apsch_inquiry;
   {Traite chaque client de classe USRINQ}

procedure apsch_rolled_out;
   {Traite chaque client de classe ROLUSR}

procedure apsch_finish;
   {Traite chaque client de classe JOBFIN}

procedure apsch_detach_user;
   {Traite chaque client de classe DETACH}

{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}
implementation
................................................................................
................................................................................
................................................................................
{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
begin
end.
```

UNITE BMMANAG:

Cette unité simule le gestionnaire de mémoires de masse ("Bulk Memory MANAGer"), deuxième machine virtuelle du gestionnaire de ressources.

```
unit bmmanag;

interface

uses  crt,global;

function bmmanag_bulk_config_init (job_nb : job_range1) : boolean;
   {Effectue l'allocation de l'espace memoire requis par
    le job de numero 'job_nb' dans les memoires de masse;
    renvoie la valeur 'true' si cette allocation s'est
    deroulee sans erreur}

procedure bmmanag_release_space (job_nb : job_range1);
   {Libere l'espace memoire occupe par le job de numero 'job_nb'
    dans les memoires de masse}

{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}
implementation
................................................................................
................................................................................
................................................................................
{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
begin
end.
```

UNITE HOST:

Cette unité simule la station de service *HT*.

```
unit host;

interface

uses  crt,global;

procedure host_get_event;
   {Lit et interprete le caractere ou groupe de caracteres
    pointe par le pointeur courant de la chaine du programme
    maitre du job courant et fait avancer ou reculer ce
    pointeur du nombre de caracteres adequat}

procedure host_computation;
   {Traite chaque client de classe HTC}

procedure host_external;
   {Traite chaque client modelisant un job non parallele}

procedure host_exec;
   {Traite chaque client de classe EXC ou de classe HTT}

procedure host_go_data;
   {Traite chaque client de classe GODATA}

procedure host_wait;
   {Traite chaque client de classe WAI}

procedure host_results;
   {Traite chaque client de classe RESULT}
```

```
{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}
implementation
................................................................................
................................................................................
................................................................................
{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
begin
end.
```

UNITE CHANNEL:

Cette unité simule les stations de service CH_i, $i = 1, lcap.cur\_aps$.

```
unit channel;

interface

uses  crt,global;

procedure chan_data;
    {Traite chaque client de classe DATA}

procedure chan_results;
    {Traite chaque client de classe RESULT}

{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}
implementation
..............................................................................
..............................................................................
..............................................................................
{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
begin
end.
```

UNITE APS:

Cette unité simule les stations de service AP_i, $i = 1, lcap.cur\_aps$.

```
unit aps;

interface

uses  crt,global;

procedure aps_get_event;
   {Lit et interprete le caractere ou groupe de caracteres
    pointe par le pointeur courant de la chaine du programme
    esclave du job et de l'esclave courants, et fait avancer
    ou reculer ce pointeur du nombre de caracteres adequat}

procedure aps_rollin;
   {Traite chaque client de classe ROLLIN}

procedure aps_executing;
   {Traite chaque client de classe XING}

procedure aps_computing;
   {Traite chaque client de classe APC}

procedure aps_rollout;
   {Traite chaque client de classe ROLOUT}

procedure aps_go_results;
   {Traite chaque client de classe APT}

procedure aps_barrier;
   {Traite chaque client de classe APB}
```

```
procedure aps_first_move;
   {Traite chaque client de classe APM}

procedure aps_next_move;
   {Traite chaque client de classe NEXT}

procedure aps_send;
   {Traite chaque client de classe APS}

procedure aps_receive;
   {Traite chaque client de classe APR}

{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}
implementation
...............................................................................
...............................................................................
...............................................................................
{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
begin
end.
```

UNITE BULK:

Cette unité simule les stations de service BM_j, $j = 1, lcap.cur\_bulks$.

```
unit bulk;

interface

uses  crt,global;

procedure bulk_barrier;
   {Traite chaque client de classe BARRIER}

procedure bulk_move;
   {Traite chaque client de classe MOVE ou de classe NEXT}

procedure bulk_send;
   {Traite chaque client de classe SEND}

procedure bulk_receive;
   {Traite chaque client de classe RECEIVE}

{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}
implementation
................................................................................
................................................................................
................................................................................
{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
begin
end.
```

UNITE GRAPHICS:

Cette unité gère l'interface graphique du simulateur. Au début de la simulation, l'interface graphique est initialisé par l'appel de la procédure locale *gr_init*.

```
unit graphics;

interface

uses  crt,global,graph;

var
   {Mode graphique active ou non}
      graphic : boolean;



procedure gr_update;
   {Met a jour l'interface graphique, a partir des options
    specifiees par l'utilisateur, lors de l'initialisation
    de l'interface graphique ou a l'issue d'une modification
    interactive}

procedure gr_close;
   {Arrete l'interface graphique}

{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}
implementation
.................................................................................
.................................................................................
.................................................................................
{===============================================================================}
```

```
procedure gr_init;
   {Initialise et lance l'interface graphique,
    si l'utilisateur en fait la demande;
    saisit alors les options de frequence de mise
    a jour, le mode d'utilisation (pas a pas ou continu)}

.....
.....
.....
{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
begin
      {Initialise l'interface graphique}
      gr_init
end.
```

ANNEXE B

Description des Références Bibliographiques

[Bough88]

Un outil permettant de prédire les performances d'un système distribué de gestion de base de données (appelé Bubba) dès la phase de sa conception est présenté dans ce papier. Ce système sera constitué de 50 à 1000 noeuds (composés chacun d'un processeur, d'un disque, d'une mémoire locale et d'un processeur de communication), interconnectés par un réseau. Aucune mémoire partagée n'est disponible et toute interaction entre noeuds se fait au travers du réseau d'interconnexion par échange de messages.

Les principaux problèmes posés par la conception d'un tel système sont la détermination optimale (du point de vue des performances du système, exprimées en débit maximal atteint, taux d'utilisation des ressources, et du coût associé) du nombre de noeuds à considérer, d'un schéma de placement des relations de la base de données dans les différents noeuds, de la taille de la mémoire cache à utiliser sur chaque noeud.

L'outil d'aide à la conception doit être rapide, convivial (interface graphique pour l'analyse des résultats), capable de prendre en compte des modifications hardware et software de l'architecture initiale et procurer différents niveaux de détail dans les résultats (permettant dans un premier temps de rejeter rapidement les alternatives très défavorables, puis de faire des analyses beaucoup plus fines au fur et à mesure de l'avancement de la conception).

L'outil (FIRM) est composé de trois modules constituant un pipeline.

1. L'assesseur détermine les demandes de service de chaque composant d'une transaction aux différentes ressources et réalise le placement des relations de la base dans les noeuds en fonction du nombre de noeuds, des caractéristiques de la base de données, de règles générales de

placement (contraintes liées à une stratégie) et de la charge du système composée de cinq types de transaction. Les transactions sont caractérisées par des graphes de flot de données où chaque opération associée à la transaction (appelée composant) est instanciée par un ou plusieurs processus sur un ou plusieurs noeuds.

2. Le réfracteur calcule à partir des résultats issus de l'assesseur la fréquence relative d'occurrence de chaque composant par processeur (ou noeud) et le profil d'utilisation des ressources par composant (incluant les échanges de messages sur le réseau d'interconnexion).

3. Le troisième module réalise la détermination des performances du système, telles que le débit de chaque type de transaction, le taux d'utilisation des ressources - processeurs, disques, réseau - (calculés à l'aide d'un réseau de files d'attente fermé, constituant le premier niveau de détail de l'outil), le temps de réponse moyen des transactions (calculé par un simulateur, qui analyse beaucoup plus finement le graphe des composants, en tenant compte des dépendances de données et des conflits de verrous).

Un système expérimental (de 40 noeuds seulement), appelé *experimental vehicle*, a été implémenté afin de permettre une analyse beaucoup plus détaillée des alternatives software pour la conception du système. L'EV est un élément complémentaire de FIRM qui supporte l'interface utilisateur de FIRM et notamment la caractérisation des transactions par graphe de flots de composants. L'EV dispose d'un système très précis d'enregistrement de traces d'événements, permettant une analyse très fine des mesures et ainsi de déceler des erreurs conceptuelles (dégradant les performances) aussi bien que des erreurs de codage. Enfin, l'EV pourra être utilisé pour valider l'outil FIRM par comparaison des résultats expérimentaux avec ceux du modèle.

[Bouha88]

Une méthode originale de classification de transactions soumises à un système informatique est l'objet de ce papier. Les transactions sont caractérisées par les requêtes d'entrées-sorties qu'elles adressent au système. L'avantage d'une telle classification permet le regroupement des transactions en classes. Chaque classe peut alors être représentée par une transaction lui appartenant, d'où une réduction non négligeable des données nécessaires à la modélisation des transactions pour l'évaluation des performances du système transactionnel. Lors de la conception d'une nouvelle ap-

plication transactionnelle, on essaiera dans un premier temps de déterminer pour chaque transaction sa classe d'appartenance, et dans un deuxième temps, si aucune classe ne convenait, on définira une nouvelle classe et on procèdera à une reclassification de toutes les transactions.

On distingue 4 types d'opérations élémentaires: la lecture simple, l'insertion, la lecture en vue d'une mise à jour et l'écriture (pour destruction ou mise à jour). On définit 3 profils pour caractériser les transactions: le profil des opérations (pourcentages relatifs des 4 types d'opérations), le profil de mise à jour (pourcentages respectifs des opérations de lecture et d'écriture pour mise à jour) et le profil de lecture/écriture (pourcentages relatifs des opérations de lecture - lecture simple et lecture en vue d'une mise à jour - et des opérations d'écriture - insertion et mise à jour). On définit également l'intensité d'une transaction comme le nombre total d'entrées-sorties qu'elle génère.

A partir de ces définitions, une représentation graphique de la caractérisation de chaque transaction est réalisée sous la forme de deux diagrammes ombrés (shadowgrams). Ces shadowgrams représentent les trois profils précédemment définis. L'utilisateur peut alors regrouper les différentes transactions en classes de par la similarité de leurs shadowgrams. Un exemple est décrit dans le papier, donnant naissance ainsi à 11 catégories. Ensuite, chaque catégorie est caractérisée numériquement afin de permettre l'incorporation de nouvelles transactions dans les classes de manière automatisée. Cela peut nécessiter la création de nouvelles classes. On procèdera alors à une reclassification visuelle de toutes les transactions.

[Brine88]

Ce papier présente des algorithmes de parallélisation de simulations à événements discrets, suffisamment généraux pour pouvoir s'implémenter sur diverses architectures parallèles ou distribuées.

On définit une tâche comme l'évaluation du modèle à l'occurrence d'un nouvel événement (extraction de la queue des événements). Les algorithmes sont basés sur la connaissance a posteriori (après exécution d'une simulation séquentielle test) de la durée de chaque tâche et des interdépendances entre tâches (données sous la forme de couples). La parallélisation consiste à allouer aux processeurs les différentes tâches, en tenant compte des interdépendances, afin de minimiser la durée totale de la simulation.

Trois algorithmes sont proposés:

1. considération d'un nombre infini de processeurs
2. contrainte sur le nombre limité de processeurs
3. prise en compte d'un partitionnement statique des données entre les mémoires locales des processeurs, entraînant un surcoût lors d'accès à distance (au travers d'une mémoire globale ou d'un réseau d'interconnexion).

Dans ces algorithmes, les surcoûts associés aux synchronisations entre processeurs, à la contention pour l'accès aux mémoires partagées ou au réseau d'interconnexion, ainsi que le temps nécessaire aux insertions dans et aux extractions de la queue des événements sont négligés.

Les accélérations obtenues lors de la simulation parallèle de trois circuits électriques sont présentées en fonction du nombre de processeurs de traitement et de l'algorithme utilisé. On remarque que les contraintes liées au partitionnement statique de données ont pour effet de dégrader considérablement l'accélération et que le recours au partitionnement dynamique ou à la réplication de données pourrait être très bénéfique dans certains cas.

[Chand88]

Le papier présente tout d'abord les arguments en faveur des systèmes parallèles (réduction de taille des composants au silicium pour une même puissance de calcul, coût peu élevé par association de composants standard, accroissement incrémental facilitant la prévision de configuration) et ceux qui vont à l'encontre de leur essor (difficulté de programmation, performance effective souvent bien inférieure à la performance de crête du fait du surcoût de communication, réticence des entreprises à abandonner ou réécrire leurs programmes d'application écrits pour la plupart en FORTRAN séquentiel).

Trois approches de programmation parallèle sont généralement utilisées:

1. adaptation de compilateurs de langages fonctionnels (FORTRAN, Pl/1) afin qu'ils supportent la parallélisation, quitte à négliger quelque peu les performances

2. utilisation d'un langage permettant au programmeur d'exprimer explicitement le contrôle de la machine cible par des instructions spécifiques (send, receive)

3. utilisation d'un langage de haut niveau aux étapes initiales de conception des programmes parallèles et d'un langage spécifique à l'architecture pour les phases terminales de conception, où l'efficacité est un souci primordial (approche hybride des deux précédentes).

Concernant les architectures parallèles, K.M. Chandy distingue quatre classes principales:

- les pipelines (ex: CRAY-1), où la première approche de programmation parallèle fait légion
- les architectures SIMD (ex: Connection Machine), où la première approche est également employée
- les architectures MIMD, qui sont constituées d'un certain nombre de processeurs coopérant par échange d'information au travers de mémoires partagées (ex: BBN Butterfly Parallel Processor) ou par transmission de messages (Intel iPSC), et où la seconde approche est la plus fréquente
- les autres architectures, de type SPMD (ex: VM/EPEX-FORTRAN) ou à flot de données (dataflow).

Les principaux problèmes liés aux performances des programmes parallèles sont le temps de latence associé à chaque transfert de message entre processus pour les architectures MIMD de type transmission de messages, le temps de commutation de processus, la contention pour l'accès aux voies de communication. Les entrées-sorties de même que la mémoire peuvent s'avérer être des goulots d'étranglement, les entrées-sorties par la masse des données transmises d'un processeur hôte vers les autres processeurs à l'initialisation des processus, la mémoire par une puissance de calcul globale accrue plus en rapport avec le système de gestion de mémoire (cache, mémoire virtuelle).

Enfin, on relève un manque certain d'outils et de méthodologies pour une utilisation efficace des architectures parallèles, notamment en matière d'analyse de programmes (contention due aux communications entre processus, rapport calcul/communication ou calcul/accès de mémoires distantes - importance des outils d'analyse de performances), de debugging, d'équilibrage de charge entre processus statique ou dynamique (utilisation possible de méthodes appliquées à la répartition des entrées-sorties sur un ensemble d'unités), de prévision de configuration (rendue très ardue de par la difficulté de prédire comment une machine parallèle répondra à une charge de production).

[Chiol88]

Des techniques de compilation sont présentées pour réduire d'un ordre de magnitude l'espace mémoire et le temps nécessaires à la génération du graphe des marquages des réseaux de Petri stochastiques. Une méthode de classement des marquages lors de leur génération est détaillée afin de produire une matrice du processus markovien associée au graphe des marquages la plus diagonale par bandes possible, d'où une réduction non négligeable du temps de résolution du réseau de Petri correspondant.

Une implémentation de ces techniques a été réalisée dans le package GreatSPN [Chiol87] et le gain en espace mémoire et temps de calcul de la nouvelle version sur la précédente est donné. Un autre avantage non négligeable de ces techniques est la possibilité de prendre en compte des réseaux plus complexes et de les résoudre sur les mêmes minicomputers que ceux où la version précédente aurait échoué.

Afin de réduire l'espace mémoire nécessaire à la génération du graphe des marquages, un codage compact des marquages est tout d'abord opéré, tenant compte des invariants de places (linéaires ou non), des bornes des places, éliminant les places implicites. Pour réduire et l'espace mémoire et le temps de génération du graphe des marquages, le stockage des marquages est réalisé sous forme d'arbre binaire à plusieurs niveaux, afin d'accélérer la recherche d'un marquage et l'insertion d'un nouveau marquage. Chaque niveau de l'arbre binaire correspond au classement de chaque octet du code des marquages. Enfin, afin d'activer le tir des transitions, une procédure spécialisée est écrite pour chaque transition, réalisant le tir directement sur le code compact des marquages. De plus, une liste des transitions tirables est maintenue et mise à jour à chaque tir de transition en tenant compte des propriétés structurelles du réseau de Petri étudié.

L'application de ces techniques nécessite un pré-traitement pour faire l'analyse structurelle du réseau de Petri, pour générer une portion de code dépendant des propriétés structurelles du réseau étudié, la compiler et en faire l'édition de liens avec la partie du package commune à tous les types de réseaux. Cette phase préliminaire n'est bien sûr justifiée que pour des réseaux conséquents où le temps gagné lors de l'exécution compense largement le temps d'analyse et de compilation initial.

[Cleme88]

Cet article fait une présentation très complète du projet ICAP. Il détaille tout d'abord les raisons qui ont donné naissance au projet et les motivations qui ont concouru à l'avénement du système ICAP/3090 (puissance intrinsèque du système IBM 3090 avec des possibilités vectorielles sur chaque processeur, existence du langage parallèle Parallel FORTRAN permettant d'exécuter un même programme sur plusieurs processeurs d'un même système, augmentation du degré de parallélisme offert par un seul système par extension du langage Parallel FORTRAN afin de supporter le parallélisme inter-système). Deux architectures possibles du système ICAP/3090 sont proposées et le software permettant d'implémenter le parallélisme est décrit.

Des considérations de performances sont ensuite abordées, concernant les paramètres qui limitent l'accélération obtenue lors de l'exécution parallèle d'un programme d'application. La loi d'Amdahl montre tout d'abord qu'en fonction du pourcentage de code qui peut être parallélisé, il existe une valeur asymptotique de l'accélération en fonction du nombre de processeurs qui est d'autant plus vite atteinte que la portion de code parallèle est faible. Il ne présente alors que peu d'intérêt à augmenter indéfiniment le nombre de processeurs. Ce phénomène est encore plus dramatique si le pourcentage de code parallélisé est une fonction décroissante du nombre de processeurs. Dans ce cas, l'accélération passe par un maximum qui détermine le nombre optimal de processeurs à utiliser. Deux autres paramètres influent directement sur l'accélération: les communications inter-processus (voire inter-processeurs) et l'équilibrage de charge entre les processeurs. Ce dernier paramètre peut n'influer que statiquement (et ne dépendre que de l'application) dans le cas de systèmes dédiés ou à la fois statiquement et dynamiquement dans le cas de systèmes multi-programmés. L'effet statique peut être amoindri, voire éliminé, par modification de l'application, mais l'effet dynamique qui dépend de la charge extérieure au programme imposée au système est imparable. Tous ces paramètres déterminent donc, pour une application donnée, un nombre de processeurs optimal à utiliser en parallèle pour l'exécution du programme.

L'évolution chronologique du projet ICAP est proposée, expliquant notamment l'extension des systèmes initiaux ICAP1 et ICAP2 par l'adjonction de mémoires partagées et d'un bus nécessaires pour accélérer les communications entre processeurs, critiques dans le cas d'applications à granularité moyenne (applications d'ingénierie), voire lentes. L'étude détaillée du débit de communication obtenu pour réaliser une diffusion de message (de taille variable) vers un nombre variable de processeurs fait l'objet d'une comparaison des efficacités respectives des divers chemins de données offerts

par le système (canaux IBM-AP, mémoires partagées, bus). Il ressort de cette analyse que le bus est le médium à privilégier pour des transferts de données massifs, mais que les mémoires partagées produisent de bien meilleures performances pour des messages de quelques mots ou kilo-mots (de par leur faible temps de latence) et doivent être utilisées dans la plupart des applications. Cependant, le rapport asymptotique (pour de très gros messages) entre le débit obtenu avec les canaux et celui obtenu avec les mémoires partagées est de l'ordre de 5 à 10 seulement. Pour tous les média de transfert, on a relevé une croissance quasi-linéaire du débit avec le nombre de processeurs récepteurs.

Deux autres points sont rapidement abordés, le gestionnaire de ressources parallèles et l'extensibilité du système.

Le gestionnaire de ressources s'occupe de l'attribution des processeurs aux différents jobs parallèles s'exécutant sur le système, suivant un mécanisme de priorités et de préemption de processeurs. Deux contraintes sont imposées sur les systèmes ICAP1 et ICAP2: le temps de commutation de contextes très important (tendant à limiter les préemptions de processeurs et à augmenter la tranche de temps d'allocation des processeurs), la nécessité pour conserver le caractère parallèle de l'exécution des programmes de préempter en même temps tous les processeurs exécutant le programme de plus faible priorité.

L'extensibilité du système s'opère sur deux plans: remplacer chaque noeud par un noeud plus puissant (en respectant cependant l'homogénéité des noeuds dans un souci de conserver un meilleur équilibrage de charge et en prenant en considération l'effet dégradant que le gain en puissance de calcul aura sur les communications), ou augmenter le nombre de noeuds du système par un couplage de systèmes (ex: ICAP1 et ICAP2 couplés pour former ICAP3) ou l'adjonction de noeuds supplémentaires.

[Comfo87]

Le papier s'intéresse au test des modèles de simulation de systèmes. Le test s'opère aussi bien au niveau de la vérification du modèle (qui s'assure de la correction du programme de simulation) qu'au niveau de sa validation (qui vérifie que le comportement du modèle est fidèle à celui du système réel).

Les erreurs peuvent provenir d'une incompréhension d'aspects importants du système réel de la part du modéliseur, d'hypothèses de simplification outrancières, d'un mauvais codage du modèle.

Des techniques similaires à celles employées dans le test de logiciels peuvent être utilisées pour le test en phase de vérification du modèle. La technique du partitionnement en classes d'équivalence des données d'entrée du programme permet de construire des jeux de tests qui recouvrent les cas valides de données ainsi que les cas non valides. L'analyse des valeurs limites doit également être réalisée. On vérifiera que les jeux de tests font intervenir l'ensemble des instructions du programme, satisfaisant successivement à toutes les alternatives de chacun des tests de branchement. Si le programme met en jeu plusieurs processus, les différents problèmes de concurrence des processus pour l'accès à des sections critiques doivent être abordés. Enfin, tout jeu de données dont le testeur a l'intuition qu'il puisse conduire à une erreur doit également être pris en compte.

La simulation d'un système met également en jeu des phénomènes stochastiques. Afin de conduire les tests, on s'attachera à supprimer le caractère aléatoire des événements. On considérera tout d'abord des intervalles d'inter-arrivée et des temps de service constants dont la valeur sera égale à la moyenne de la loi de distribution effective, ou proche de valeurs limites. Puis, on s'intéressera à des situations où les arrivées se font par vagues de clients bien espacées.

Quant aux tests de validation du modèle, on s'efforcera de comparer le comportement du modèle à celui du système réel dans des circonstances artificielles ou fortuites de fonctionnement du système (retrait d'une ressource pour cause de panne, arrivée de clients en grand nombre tendant à saturer le système, du moins momentanément).

[**Covin88**]

RPPT (Rice Parallel Processing Testbed) est un simulateur d'exécution de programmes parallèles sur diverses architectures parallèles. Cet outil permet aussi bien d'étudier différentes alternatives d'algorithmes que différents choix de systèmes pour l'implémentation d'un programme parallèle. Le simulateur permet d'obtenir des évaluations de performances sur l'exécution des programmes, tels que le taux d'utilisation des processeurs, des média de communication, les overheads dûs respectivement aux communications entre processus, aux synchronisations entre processus, à la gestion des processus (création et activation ou commutation de contexte).

Les principaux constituants de RPPT sont:

1. le langage C concurrent (extension du langage C), permettant la création et le contrôle de processus, leur synchronisation et les transferts d'information entre eux
2. le package de simulation (CSIM) écrit en C concurrent, qui permet la gestion de listes d'événements et la production de statistiques
3. le préprocesseur de simulation d'architecture (ASIMP), qui modifie le code source C concurrent afin de simuler les délais dûs aux interactions entre processus sur processeurs distants (communications, synchronisations) par appel de routines modélisant l'overhead dû à l'initialisation du réseau d'interconnexion, la contention d'accès au réseau et les temps de transmission de messages
4. l'estimateur de temps de traitement (timing profiler TPROF), qui détermine, à partir du code source C concurrent assemblé, le temps de calcul de chaque bloc de traitement entre deux interactions de processus (ce temps dépend bien sûr du processeur de calcul du système simulé), et qui insère dans le code assemblé des appels à des routines de simulation de délais correspondant au temps précédemment estimé
5. l'interface de simulation (SIMTOOL), qui offre à l'utilisateur la possibilité de combiner par utilisation de fenêtres le programme à simuler (écrit en C concurrent), le modèle de l'architecture et la répartition des processus sur les processeurs et qui génère le simulateur prêt à être exécuté
6. le traceur et debugger parallèle (TRAPP), qui propose une interface à base de fenêtres permettant de conduire et de contrôler l'avancement de la simulation ainsi que d'en produire une trace.

La simulation est conduite par l'exécution, c'est-à-dire que, contrairement à la simulation conduite par instruction - où non seulement la durée de chaque instruction du programme est estimée, mais où également l'instruction elle-même est simulée (ce qui a pour effet d'augmenter considérablement la durée de la simulation) -, chaque instruction est véritablement exécutée sur l'ordinateur réalisant la simulation (les processus concurrents s'exécutant alors en pseudo-concurrence). L'ordinateur doit donc avoir le même jeu d'instructions que les processeurs du système simulé. L'estimateur de temps a la charge d'estimer la durée, non pas de chaque instruction, mais de chaque bloc séparant deux interactions entre processus.

Un exemple d'utilisation de RTTP est présenté pour la comparaison de deux algorithmes parallèles de calcul de transformée de Fourier rapide, sur la même architecture multiprocesseur à bus partagé.

Le dernier point du papier traite de l'overhead induit par la simulation de l'exécution de divers algorithmes sur une architecture multiprocesseur sur le temps de calcul de ces algorithmes en pseudo-concurrence sur un seul processeur (ce temps de calcul ne prenant pas en compte les délais dûs aux interactions entre processus). Rappelons que la simulation se fait également en pseudo-concurrence sur un seul processeur. Le facteur de ralentissement dû à la simulation est étroitement lié à la granularité des blocs de calcul séparant deux interactions entre processus. Plus la granularité est fine, plus l'overhead pour le calcul du temps d'exécution des blocs (généré par TPROF) est grand et plus le nombre d'interactions entre processus est important (et, associé à ce nombre, l'overhead pour modéliser les délais correspondants).

[**Darem88**]

Ce papier décrit un modèle de mise en oeuvre du parallélisme pour des programmes d'applications scientifiques écrits en FORTRAN. Son implémentation sur le système VM/EPEX est détaillée et illustrée au travers de deux exemples. Quelques données de performance (overhead induit par les routines de synchronisation, accélération) sont également présentées.

Le modèle repose sur le traitement parallèle du programme par plusieurs processus, exécutant le même code sur des données différentes (Single Program Multiple Data). Les processus ont accès à une mémoire partagée.

On distingue dans le code commun aux processus des sections séquentielles (exécutées par un seul processus), des sections parallèles (découpage des boucles du programme en plusieurs portions et répartition dynamique des portions aux processus) et des sections répliquées (portions de code exécutées par chaque processus sur les mêmes données). Les sections séquentielles et parallèles sont délimitées par des points de synchronisation. Le mécanisme de barrière est également défini pour permettre la mise à jour de variables partagées qui conditionnent le contrôle global du flot des processus. L'attente des processus aux points de synchronisation se fait par attente active intermittente (intermittent busy-wait), qui consiste à mettre le processus dans un état d'attente

passive et à le réveiller périodiquement afin qu'il vérifie s'il peut continuer son exécution. On définit des données partagées, qui dépendent de l'application ou alors sont nécessaires pour l'implémentation des mécanismes de synchronisation, ainsi que des données privées accessibles par chaque processus seulement. En fin de section séquentielle ou de section parallèle, les processus sont soit forcés d'attendre que les autres processus ait terminé leur traitement dans la section, soit autorisés à poursuivre leur exécution. Ainsi, un recouvrement de l'exécution de sections indépendantes peut être réalisé, afin d'augmenter l'efficacité de la parallélisation.

Une implémentation du modèle a été réalisée sous VM/EPEX qui permet de faire coopérer plusieurs machines virtuelles pour exécuter un programme en parallèle. A chaque processus est associé une machine virtuelle. Les processus ont accès à des variables partagées déclarées par une commande spécifique. La parallélisation du programme reste entièrement à la charge de l'utilisateur. Celui-ci doit insérer dans son code FORTRAN des commandes spécifiques qui seront traduites ensuite en des appels à des routines de synchronisation FORTRAN par un pré-processeur. Le pseudo-code pour l'algorithme de synchronisation des boucles parallèles illustre l'utilisation d'horloges locales (une par processus) et globale (partagée par l'ensemble des processus) pour permettre le traitement répétitif de boucles parallèles et de sections séquentielles, sans avoir recours à des synchronisations de type barrière, induisant une sérialisation des traitements, pour réinitialiser les variables partagées de contrôle de ces sections. L'implémentation actuelle ne permet pas cependant l'imbrication de boucles parallèles.

Concernant les performances d'un tel modèle de parallélisation et de son implémentation sur un ordinateur IBM S/3081, des accélérations de 1.95 (avec 2 processeurs) et de 3.70 (avec 4 processeurs) ont été mesurées pour les deux programmes d'application étudiés, ce qui permet d'apprécier le faible overhead induit par la synchronisation des processus et par le gestionnaire de machines virtuelles.

[Deese88]

Un système expert d'aide à la détection de facteurs limitant les performances de systèmes hôtes IBM, opérant sous MVS/370 ou MVS/XA, est décrit dans ce papier. Il fonctionne sur IBM/PC et utilise le produit VP/EXPERT.

Trois modes de fonctionnement sont possibles:

- en mode consultatif, l'utilisateur doit fournir toutes les données que le système expert demande à l'aide de questions précises
- en mode automatique, un logiciel de support du système expert récupère automatiquement sur le système hôte les informations relatives au comportement de l'environnement (pour l'intervalle de temps étudié) dont le système expert a besoin et les transfère automatiquement au PC (par le logiciel SAS); ces informations sont des données issues de rapports RMF, SMF, ...
- en mode de compte-rendu de gestion du système, extension du mode automatique, les conclusions et recommandations du système expert sont stockées sur le PC et périodiquement transférées sur le système hôte pour l'édition de rapports pour le gestionnaire du système.

Le moteur d'inférence du système expert applique les règles de type SI/ALORS, contenues dans la base de connaissances, pour produire des conclusions quant aux facteurs limitant les performances du système enregistrées et formuler des recommandations tendant à lever ou à repousser ces limites. Il accompagne chaque recommandation d'un coefficient de confiance et donne la liste des recommandations par coefficients de confiance décroissants.

L'utilisateur a la possibilité de connaître le raisonnement que le système expert a suivi pour aboutir à ces conclusions et recommandations, en enregistrant la trace de toutes les règles que le système expert a appliquées. La clause BECAUSE associée à chaque règle permet d'adjoindre un message expliquant les raisons qui ont pu conduire à appliquer cette règle.

Les informations à l'intérieur de la base de connaissances sont regroupées en plusieurs sous-ensembles, correspondant chacun à un thème spécifique.

- La base de règles contient toutes les règles actionnant le moteur d'inférence. Elle est constituée par un spécialiste en évaluation de performances des systèmes IBM opérant sous MVS.
- La base de description de l'environnement regroupe toutes les informations propres au système hôte étudié (hardware, paramètres du système d'exploitation, localisation des fichiers).
- La base de guidage contient des informations permettant d'émettre des jugements de valeur sur le comportement du système, par rapport aux objectifs que s'était fixés le gestionnaire du

système et à des critères d'évaluation plus généraux (insérés dans cette base par un spécialiste), accompagnés de valeurs d'incertitude.

- La base des diagnostics contient le libellé en langue naturelle de toutes les conclusions que peut produire le système expert.
- La base des conseils regroupe le libellé en langue naturelle de toutes les recommandations que peut émettre le système expert.
- La base de référence fournit, pour chaque règle, conclusion ou recommandation, le document source qui s'y rapporte.
- La base de sources contient l'ensemble des informations nécessaires à l'obtention de chaque donnée relative au comportement de l'environnement.
- La base de comportement de l'environnement stocke l'ensemble des données décrivant comment le système hôte se comporte pendant l'intervalle de temps étudié. Elle est constituée automatiquement en modes automatique et de compte-rendu de gestion du système, ou en mode consultatif par l'utilisateur.

[Denni87]

S'appuyant sur le rapport de Novembre 1986 du Conseil National de la Recherche américain, le papier souligne les insuffisances de l'état de l'art en matière d'évaluation de performances des super-ordinateurs.

Deux axes principaux de recherche sont à développer pour pallier ces insuffisances, la prédiction de la vitesse d'exécution d'un programme sur un super-ordinateur d'une part, et la détermination de l'utilisation d'un super-ordinateur pour les différents domaines de la recherche d'autre part. Le deuxième point recouvre le développement de méthodes permettant d'optimiser soit l'adéquation d'un algorithme à une architecture existante, soit l'adéquation d'une architecture en phase de conception à une classe (ou plusieurs classes) d'algorithmes ayant trait à différents domaines.

Pour ce qui est de la prédiction de la vitesse d'exécution d'un programme sur un super-ordinateur, les principaux besoins ressentis sont les suivants:

- les réseaux de files d'attente actuels ne permettent pas une bonne expression des phénomènes inhérents au parallélisme (création de processus, synchronisations, sections critiques, délais de communication) et surtout ne sont plus résolubles analytiquement; d'autres formalismes, tels que les réseaux de Petri stochastiques, permettent certes une meilleure expression de ces phénomènes mais ne bénéficient pas actuellement d'approximations analogues à celles des réseaux de files d'attente classiques afin de réduire l'espace d'étude et permettre une résolution rapide; un nouveau formalisme de réseau de files d'attente basé sur l'espace d'état des réseaux de Petri devrait être élaboré,
- un nouveau formalisme est nécessaire pour représenter la charge d'un système parallèle indépendamment de son architecture propre,
- une méthodologie de mesures est à mettre au point afin de pouvoir comparer des architectures de classes différentes sans avoir recours à l'utilisation d'algorithmes test dont l'implémentation peut avoir une influence non négligeable sur leur temps d'exécution,
- des outils de mesures adaptés aux architectures parallèles sont souhaitables, permettant la saisie de traces, le calcul automatique de métriques et de paramètres d'entrée de modèles, l'insertion de sondes dans les programmes; associée à ce besoin d'outils, une méthodologie, tendant à incorporer dès la phase de conception de nouveaux systèmes les outils de mesures utiles à en estimer les performances, devrait être développée,
- le recours aux systèmes experts devrait être encouragé dans l'utilisation de modèles (analytiques ou autres) pour estimer l'influence d'une modification de configuration ou de charge du système sur les performances des programmes d'application,
- une méthodologie de validation des modèles est à développer afin de permettre une reproductibilité des expérimentations, surtout pour les architectures nouvelles.

[Doman88]

Le papier s'intéresse au modèle du cycle de vie d'un système et décrit brièvement les différentes techniques qui concourent à assurer une bonne gestion du système, tout au long de son existence.

Le modèle du cycle de vie d'un système informatique se compose de trois phases, qui se succèdent de manière itérative utilisant des techniques bien précises.

1. Les besoins auxquels devra pouvoir répondre le système sont tout d'abord analysés. Parmi ces besoins, on citera notamment la charge (ou demande de travail), exprimée en nombre d'unités de travail, et le niveau de service requis, exprimé en débit atteint ou temps de réponse soutenu. Il faut bien sûr prendre en compte le facteur temps et prévoir la croissance de la charge à court ou moyen terme.

2. L'acquisition d'un système est alors à l'étude. Pour ce faire, on peut se référer aux données des constructeurs, réaliser des benchmarks ou avoir recours à la modélisation dans le but de définir la meilleure configuration de système qui puisse répondre à nos besoins, présents et futurs. Cette étude est habituellement désignée par le terme de prévision de configuration (capacity planning). Cette analyse permet même d'élaborer tout un plan de gestion du système (capacity management plan) et notamment de définir quelles valeurs limites de service ou de charge seront susceptibles de nécessiter l'accroissement de tel ou tel élément du système. La modélisation peut se faire analytiquement (analyse opérationnelle, réseaux de files d'attente) ou par simulation. Dans tous les cas, elle requiert une forte connaissance du système étudié. Son but est d'obtenir des relations quantitatives entre charge soumise au système et service rendu par le système.

3. La dernière phase concerne la gestion de performances (performance management) qui consiste à surveiller le bon fonctionnement du système maintenant installé, du point de vue de ses performances. On réalisera des mesures fréquentes afin de vérifier que les prévisions de la phase d'acquisition sont toujours satisfaites. Ces mesures seront faites à l'aide de moniteurs (software ou hardware) ou d'enregistrements de traces d'événements. Si tel ou tel seuil de charge ou de service est dépassé, on appliquera le plan de gestion du système et on effectuera l'accroissement adéquat. Toutefois, si le franchissement de ce seuil semble être prématuré, on cherchera tout d'abord à en analyser la cause et à voir s'il ne serait pas dû à une mauvaise gestion des ressources (mauvaise répartition de la charge par exemple), et si un réajustement du système ne pourrait pas pallier le problème. Le plan de gestion du système est alors révisé en conséquence. Il est possible également que la charge estimée lors de l'expression des besoins ait été sous-évaluée et un nouveau cycle est alors enclenché.

[Duda88]

Le papier se propose de déterminer le nombre optimal de tâches parallèles à considérer pour effectuer un calcul, compte tenu de l'overhead en communication résultant de la création dynamique des tâches (sur des processeurs différents) et du transfert des données associé.

Pour ce faire, deux critères de performance sont étudiés, l'accélération S_n d'une part et la qualité Q_n d'autre part, définies comme suit:

$$S_n = \frac{t_1}{t_n} \quad , \quad Q_n = \frac{S_n E_n}{R_n}$$

où E_n et R_n, représentant respectivement l'efficacité et la redondance, sont données par:

$$E_n = \frac{t_1}{n t_n} \quad , \quad R_n = \frac{\sum_{i=1}^{n} \tau_i}{t_1}$$

où τ_i représente le temps de calcul et de communication de la tâche i.

Le temps de communication étant fonction du schéma d'interconnexion entre les processeurs, différentes structures sont considérées: l'anneau, l'arbre binaire, l'hypercube et le réseau complet.

Deux modèles sont successivement étudiés: le modèle déterministe et un modèle stochastique où le temps de calcul et le temps de communication suivent tous deux une distribution Gamma. Une approximation est alors faite afin que le temps global (calcul + communications) de chaque tâche satisfasse à une loi exponentielle. Ainsi on est ramené au calcul de l'espérance mathématique du maximum de n lois exponentielles [David70].

Pour chaque modèle et chaque schéma d'interconnexion des processeurs, les nombres de tâches parallèles, qui optimisent respectivement l'accélération et la qualité, sont calculés ou résolus numériquement. Ces nombres sont supérieurs dans le cas déterministe à ceux du cas stochastique. De plus, les nombres optimaux, pour un même rapport temps moyen de calcul d'une tâche sur temps moyen de communication, sont supérieurs pour le critère d'accélération à ceux du critère de qualité. En effet, la courbe de la qualité en fonction du nombre de tâches passe par un maximum, dans tous les cas, pour un nombre de tâches inférieur à 500, ce qui n'est pas le cas pour l'accélération.

[Fang88]

Le papier présente un système expert, développé sur IBM PC/XT, utilisé pour la génération automatique de programmes de simulation à événements discrets.

Une interface homme-machine permet à l'utilisateur par un dialogue en langue naturelle de décrire le modèle du système qu'il veut simuler. Les différents paramètres nécessaires à la définition des éléments du modèle sont saisis interactivement.

Par l'utilisation d'une base de connaissances (créée au préalable par un expert en simulation sous la forme de règles "SI (condition) ALORS (action)"), le moteur d'inférence du système expert (écrit en Turbo Prolog) génère le programme de simulation en langage GPSS-C. Outre la description des stations de service, où la politique d'allocation des serveurs peut être FCFS, FCFS avec priorités, SJF (Shortest Job First) ou priorités dynamiques, des critères de fin de simulation ainsi que des sorties spécifiques de résultats peuvent être précisés.

Le système expert actuel requiert une amélioration de l'interface de dialogue par une extension graphique. Il bénéficierait beaucoup d'un module d'apprentissage automatique. La possibilité d'une modélisation dynamique, où les résultats de simulations antérieures seraient utilisés pour modifier certains paramètres, voire la structure, du modèle en vue de son optimisation, est même envisagée.

[Feuga88]

Un outil d'aide à la conception de systèmes informatiques distribués est présenté. Cet outil permet aux ingénieurs système de prévoir les performances des systèmes dès la phase de conception, en fonction de choix de configuration, d'architecture et de la charge en processus.

Cet outil est composé de quatre modules principaux.

1. Le module de définition permet à l'utilisateur de saisir les caractéristiques du système, aussi bien sur le plan hardware (avec une représentation graphique de la configuration) que sur le plan des processus qui s'exécuteront sur le système. Les liens entre composants peuvent être établis graphiquement à l'aide d'une souris. Les données numériques de chaque élément sont saisies à partir de menus contenus dans des fenêtres spécifiques.

2. Le module de génération du modèle sous forme de réseau de files d'attente consiste à créer à partir des définitions du précédent module des sous-modèles et de les interfacer afin de constituer un modèle qui pourra être interprété par le logiciel QNAP2 et résolu par simulation. Quatre familles de composants hardware sont considérées (processeurs, unités de stockage, unités annexes telles qu'imprimantes ou terminaux, unités de communication telles que bus, liaisons série). A chaque famille correspond un sous-modèle générique représenté par une macro QNAP2 et qui sera instancié par les données numériques données par l'utilisateur. Les processus correspondent aux clients du réseau de files d'attente dont les caractéristiques propres sont stockées par le mécanisme d'attribut de QNAP2.

3. Le module de simulation lance la simulation du modèle QNAP2 en saisissant préalablement les paramètres propres de la simulation, tels que le temps de la simulation, les unités à observer (dont on veut les résultats à l'issue de la simulation), le pas d'échantillonnage pour l'enregistrement de résultats intermédiaires. La simulation peut se faire soit en mode batch, soit en mode interactif. Dans ce dernier cas, il est possible d'avoir une animation graphique en temps réel du taux d'utilisation de différentes unités du système (jusqu'à 5), d'activer un nouveau processus en précisant alors ses paramètres propres et de stopper la simulation.

4. Le module d'interprétation des résultats présente sous forme de courbes et d'histogrammes l'ensemble des résultats proposés par le logiciel QNAP2 à l'issue de la simulation. L'utilisateur peut ainsi interpréter les résultats avec un bon support visuel et mieux comprendre le comportement du futur système sous une charge donnée. Dès la conception du système, des améliorations peuvent être apportées et des erreurs évitées.

Le principal atout de cet outil intégré d'aide à la conception de systèmes distribués est sa facilité d'utilisation pour tout ingénieur système sans la connaissance préalable du formalisme des réseaux de files d'attente ni même l'aide d'un expert en évaluation de performances.

[Frabo87]

Un langage d'expression du parallélisme adapté aux architectures MIMD, XANADU, est décrit. Un programme d'application est composé de trois parties:

- la description des données contient l'ensemble des déclarations de données usuelles ainsi que

des directives d'implémentation des données permettant d'optimiser l'exécution du programme en réduisant les conflits d'accès aux mémoires partagées,

- la description des tâches, qui peuvent être des tâches de calcul (C-tâches) ou des tâches de mouvements de données (M-tâches) - utiles pour une réorganisation des données en mémoire partagée -, simples ou génériques,
- la description du séquencement des tâches, appelée tâche de synchronisation (S-tâche), définit les interdépendances entre tâches de calcul ou de mouvements de données (séquentialité, condition, itération, exécution parallèle, indépendance ou synchronisation).

Le papier s'intéresse aux architectures MIMD à couplage lâche, constituées de plusieurs processeurs élémentaires sans mémoire principale commune, mais ayant chacun une mémoire locale et pouvant accéder à une mémoire secondaire commune. Les tâches de calcul s'exécutent sur les processeurs élémentaires alors que le séquencement des tâches et leur attribution aux processeurs est géré par un superviseur résidant sur un ordinateur hôte. Les tâches de mouvements de données sont effectuées par des processeurs spécialisés appelés processeurs de mémoire capables de transférer du code et des données entre mémoire secondaire et mémoire locale.

Un simulateur de programmes d'application s'exécutant sur de telles architectures a été développé afin d'estimer le temps d'exécution des programmes. Ce simulateur a été utilisé pour l'étude de trois programmes et de trois architectures différentes. De plus, diverses implémentations du même programme ont été considérées en faisant varier la granularité des tâches parallèles. Différentes stratégies de supervision ont été prises en compte, telles que l'anticipation des transferts de données (lorsque le recouvrement entre calculs et transferts de données est possible) et le maintien de données en mémoire locale d'une tâche sur la suivante, afin d'estimer leur impact sur les performances des programmes d'application.

Le papier présente enfin l'architecture MIMD expérimentale de l'ONERA Châtillon, composée d'un ordinateur hôte GOULD SEL 32/77 relié à 4 AP 120B de Floating Point Systems. Les programmes d'application sont exprimés à l'aide du langage parallèle LESTAP, sous-ensemble du langage XANADU. Les différentes fonctions du superviseur sont détaillées. L'implémentation de programmes d'application de taille industrielle permettra de mesurer les performances effectives de cette architecture.

[Gentz88]

Ce papier présente le nouveau langage Parallel FORTRAN, commercialisé par IBM pour permettre l'exécution parallèle de programmes sur ses systèmes vectoriels multiprocesseurs 3090. L'optique retenue est une adaptation à moindre coût de code source VS FORTRAN par l'addition de commandes supplémentaires et l'extension d'autres commandes existantes (DO et CASE). Une parallélisation de boucle automatique est proposée.

Deux niveaux de parallélisme sont ainsi offerts, à faible granularité par la parallélisation de boucle, et à forte granularité par la définition et l'activation de tâches parallèles (une tâche étant l'exécution d'une subroutine). Des mécanismes de synchronisation par événement et par verrou sont également fournis.

Parallel FORTRAN est supporté par les deux systèmes d'exploitation d'IBM VM/XA et MVS/XA. Quelques détails d'implémentation sont présentés dans le papier.

La parallélisation d'un code de magnéto-hydrodynamique illustre les différentes alternatives possibles d'implémentation parallèle d'un programme: la parallélisation automatique de boucle, la gestion et la synchronisation de tâches, la synchronisation par événement ou la parallélisation de boucle et de commande CASE. Les efficacités relatives de ces alternatives sont comparées et des règles générales d'utilisation de Parallel FORTRAN en sont déduites:

- préférer la forte granularité (gestion de tâches, synchronisation par événement) à la faible granularité (parallélisation de boucle), même si sa mise en oeuvre est moins aisée
- limiter le nombre de lancements de tâches asynchrones (par la commande DISPATCH)
- utiliser si possible la synchronisation par événement ou par verrou (plutôt que la commande WAIT).

[Goldb83]

Le papier présente le modèle sous forme de réseau mixte de files d'attente en forme produit d'un système distribué de mini-computers (VAX 11/750) reliés à un réseau local à jeton. Le système d'exploitation distribué, Locus [Popek81] développé à l'Université de Californie de Los Angeles,

intègre entièrement le support du réseau local.

La démarche de modélisation est la suivante:

1. conception du modèle que l'on exige de forme produit
2. adaptation du système distribué au modèle (avec approximations)
3. résolution du modèle par la méthode MVA pour les réseaux mixtes (développée par Zahorjan [Zahor81]) avec approximation de Schweitzer [Schwe79]; on distingue la charge foreground des utilisateurs (une chaîne fermée pour chaque site) de la charge background du processus de mise à jour des fichiers répliqués (une chaîne ouverte pour chaque site)
4. caractérisation de la charge du système par création d'un programme de simulation d'utilisateurs avec distinction entre requêtes locales et requêtes distantes (exprimées comme un pourcentage du nombre de requêtes locales)
5. détermination par mesure sur le système des temps de service des stations de service du modèle (processeur et disque de chaque site)
6. validation du modèle par comparaison de mesures effectuées sur système réel et des prédictions correspondantes du modèle (variation du nombre d'utilisateurs par site, variation de la charge background exprimée comme un pourcentage de la charge foreground, variation de la localité des accès fichiers entraînant une modification de l'activité du réseau).

Les résultats montrent une bonne adéquation du modèle au système distribué avec quelques réserves lorsque les accès fichiers à distance tendent à croître.

[Harms88]

Dans le but de remplacer l'ordinateur central du centre de calcul de l'Université de Hannovre, une campagne de benchmarks a été réalisée afin de comparer les performances (temps CPU) de trois super-ordinateurs vectoriels, le CRAY-1M, le CRAY-X/MP et le FUJITSU VP-200, à celles du CYBER 76. Seul le FUJITSU VP-200 dispose d'une unité vectorielle (pipeline) masquable. Un seul processeur du CRAY-X/MP a été considéré. L'étude ne prend en compte que la vectorisation automatique, c'est à dire celle réalisée par le compilateur.

Au lieu d'utiliser des programmes étalons, qui permettent de déterminer les vitesses de traitement pour des opérations élémentaires ne faisant appel qu'à une ou deux unités arithmétiques, 28 programmes d'application couramment employés par les utilisateurs du centre de calcul furent testés. L'avantage d'une telle approche permettait en outre de juger de la facilité de conversion des programmes et de leur rapidité d'implémentation sur les trois super-ordinateurs.

Les conclusions de la campagne de mesures vont en faveur du FUJITSU VP-200, qui possède, rappelons-le, une unité vectorielle masquable. Cependant, l'accent est mis sur l'importance tant du hardware que du software (le compilateur notamment) sur la vitesse d'exécution d'un programme d'application. Il semble à ce propos que le compilateur FORTRAN des CRAY vectorise mieux que celui du CYBER 205, mais moins bien que celui du FUJITSU du fait de l'absence de masque pour son unité vectorielle.

Un autre point important du papier est la mise en garde sur la comparaison des vitesses moyennes relatives de plusieurs ordinateurs. La taille des problèmes (liée à des longueurs de boucles plus ou moins grandes) influe directement sur le temps CPU nécessaire à l'exécution du programme. L'importance relative de certains programmes dans le calcul de la moyenne peut accroître ou réduire le rapport des vitesses moyennes de deux ordinateurs. Pour pallier ce problème, il semble plus correct de calculer la moyenne des rapports des temps CPU plutôt que le rapport des moyennes des temps CPU.

Enfin, lorsque les ordinateurs disposent d'unités vectorielles, on s'attachera à considérer des programmes dont les longueurs de vecteurs sont différentes pour évaluer l'importance de l'initialisation des unités vectorielles (ou des pipelines) dans le temps de calcul du programme. Il est possible qu'un ordinateur ait un temps d'initialisation plus court et soit cependant moins performant pour des longueurs de vecteurs importantes qu'un autre ordinateur dont le temps d'initialisation serait plus long.

[Herzo87]

Un tour d'horizon des techniques de mesures et de modélisation pour l'évaluation des performances des systèmes vectoriels et multiprocesseurs est présenté.

Les notions classiques de Mips, de Mflops, de taux d'occupation, de débit et de temps de réponse sont définies. Des benchmarks et des scripts (représentant synthétiquement la charge réelle d'un système) sont souvent employés pour comparer les performances respectives de différents systèmes.

Pour les systèmes vectoriels, le critère d'accélération, défini par l'approximation d'Amdahl, peut également être utilisé. Il est donné par la formule:

$$S = \frac{1}{(1-f) + f/k}$$

où f est la fraction de code vectorisé et k la vitesse relative de l'unité vectorielle par rapport à l'unité scalaire du système considéré.

Pour les systèmes multiprocesseurs, on considère également l'accélération, qui, loin d'être linéaire avec le nombre de processeurs, est limitée par l'overhead dû aux communications et synchronisations entre processeurs.

Parmi les outils de mesure, permettant d'observer le comportement dynamique des systèmes sous une charge donnée, les moniteurs sont divisés en deux catégories, les moniteurs hardware et les moniteurs software. Les moniteurs hybrides sont un bon compromis pour allier les avantages des uns et des autres. Leurs objectifs sont l'enregistrement d'événements hardware et software intra et inter processeurs, l'analyse de ces événements pour la détection de possibles goulots d'étranglement, la visualisation de résultats clé pour les responsables du système ou les programmeurs d'application et la validation de modèles d'évaluation de performances. Ce genre d'outils nécessite, pour être efficace, des utilitaires de compactage de données et d'analyse de données très performants.

Les moniteurs hardware utilisent des sondes électroniques capables de saisir de l'information binaire et de la transférer dans des compteurs. Ils présentent le grand avantage de n'avoir aucune influence sur les performances du système sous test, mais l'interprétation des traces d'événements requiert souvent l'assistance d'un spécialiste.

Les moniteurs software sont des programmes de saisie de données, souvent résidant dans le système d'exploitation. L'avantage principal est l'interprétation possible des traces au niveau application. Cependant, l'impact de l'overhead créé par ces programmes sur les performances des applications que l'on veut observer est loin d'être négligeable.

La modélisation est un autre moyen d'évaluer les performances d'un système multiprocesseur. Elle peut se faire à l'aide de modèles probabilistes, résolus par des outils mathématiques, si les modèles sont simples, ou par la simulation. L'approche mathématique doit être préférée à la simulation, si elle est applicable, car elle est beaucoup moins coûteuse en temps.

Les modèles doivent être suffisamment simples, pour permettre une évaluation efficace, et très précis, pour bien décrire le flot réel de l'information au travers du système. Une approche modulaire et hiérarchique est recommandée.

Deux modélisations, traitant deux problèmes inhérents aux systèmes multiprocesseurs, sont très brièvement décrites: le conflit d'accès à des mémoires partagées, l'impact des synchronisations entre processeurs sur le temps de traitement d'un programme parallèle de structure itérative.

[Hills86]

Ce papier présente un programme de visualisation sur écran graphique des résultats de simulations.

Les simulations, nécessitant souvent une grande puissance de calcul, sont réalisées sur un gros ordinateur central et tous les résultats intermédiaires sont stockés dans un fichier en cours de simulation. En fin de simulation, le fichier est transféré sur un IBM PC (ou compatible), connecté à l'ordinateur central.

Le programme de visualisation interprète chaque enregistrement du fichier issu de la simulation pour représenter graphiquement l'évolution de l'état du système simulé. Chaque composant du système est représenté par une icône que l'utilisateur peut définir lui-même, stocker sur disque et réutiliser à sa guise. L'association adéquate des icônes représentant les constituants du système définit la toile de fond de l'animation qui est conduite à partir des résultats du fichier de la simulation.

L'avantage principal de l'utilisation d'un fichier stockant tous les résultats intermédiaires de la simulation sur une animation en temps réel simulé est la simplicité de réalisation d'un backtracking, permettant de "remonter le temps".

[**Johns88**]

Les avantages apportés par l'utilisation de l'animation dans la modélisation par simulation sont présentés. Une bibliographie fouillée des publications relatives à l'animation est également fournie. Une étude de cas illustrant l'approche hiérarchique de l'animation à trois niveaux de détail différents (avec des degrés d'abstraction décroissants) est décrite.

L'animation peut compléter très profitablement l'approche classique de la modélisation par simulation, et ce à toutes les phases d'un projet de simulation (vérification, validation, analyse, présentation).

En phase de vérification du modèle de traitement (debugging du codage du modèle conceptuel), l'animation apporte une aide non négligeable au debugging du modèle. Plus conviviale et aisée que l'analyse de traces ou de sorties de simulation, elle facilite l'observation d'événements simultanés (le niveau un, assez abstrait, est utilisé par le modéliseur).

En phase de validation, l'animation tisse le lien entre le modèle et l'expert du système réel, permettant ainsi une meilleure compréhension entre le modéliseur et l'expert et un oeil plus critique de la part de l'expert sur le modèle (les niveaux un et surtout deux sont utilisés conjointement par le modéliseur et l'expert du système).

L'animation est également un outil d'analyse très intéressant. Elle permet une visualisation des problèmes et ainsi une meilleure analyse de leurs causes. Elle autorise en outre l'utilisateur à voir et comprendre la simultanéité de certains événements (le niveau deux, plus imagé que le niveau un, est utilisé).

Enfin, l'animation est un excellent support pour la communication et la présentation d'un modèle. Par une représentation dynamique très réaliste du système modélisé, le modéliseur peut expliquer très simplement ses choix de modélisation et le comportement du système dans tel ou tel cas de figure. L'animation peut ainsi être un outil d'enseignement et d'apprentissage (le niveau trois, représentant le système de manière très détaillée, est utilisé).

[Kienz79]

Une méthodologie de modélisation des systèmes informatiques est présentée dans ce papier, illustrée par une étude de cas (IBM S/370-165 II sous VS2).

Les différentes phases d'une modélisation, conception du modèle, mesures sur système réel, estimation des paramètres du modèle, analyse du modèle, validation du modèle et prédiction de performances, sont organisées selon une procédure constituée de différentes étapes:

1. mesures (→ modèle des mesures)
2. estimation des paramètres (→ modèle du système)
3. représentation des paramètres (→ modèle à résoudre)
4. résolution du modèle (par méthode analytique, simulation ou méthode hybride) permettant d'obtenir l'évaluation des performances du système.

La démarche suivie pour la conception des trois modèles consiste d'abord à définir les buts de la modélisation, soit les critères de performances du système réel que l'on veut évaluer. Partant de ces objectifs, un modèle sous forme de réseau de files d'attente est déterminé afin que sa résolution puisse produire les résultats désirés. Le modèle du système est alors défini en fonction des paramètres d'entrée nécessaires au modèle à résoudre. Un ou plusieurs modèles de mesures sont alors conçus pour fournir les données indispensables à la définition du modèle du système.

Le modèle du système est la représentation d'un point de vue logique de tous les composants et fonctions essentielles du système réel (configuration du système, fonctions du système d'exploitation, charge du système, caractérisation des programmes utilisateur, ...). Ce modèle n'est pas habituellement résoluble car il comporte nombre de détails qui devront être "approximés" pour permettre un temps de résolution non prohibitif du modèle. Il est composé de plusieurs constituants:

- modèle du comportement des programmes utilisateur: demandes de service de chaque classe de programmes aux différentes ressources du système, sans prise en compte des interactions avec les fonctions du système d'exploitation
- schéma d'interférence: interactions dynamiques ou statiques entre les programmes utilisateur qui voient le système comme une machine abstraite et la machine physique (hardware)

contrôlée par le système d'exploitation (prise en compte de l'overhead système nécessaire à l'exécution des programmes utilisateur) - composant du modèle le plus difficile à définir avec précision -

- niveau de multiprogrammation pour chaque classe de programmes pouvant évoluer dans le temps
- modèle de la charge: détermination de la *matrice de charge* obtenue à partir du modèle du comportement des programmes utilisateur et du schéma d'interférence
- caractéristiques des ressources: discipline de service, taux de service qui peut être fonction de la classe de programmes et de la charge du système.

Une modélisation doit cependant reposer sur une large gamme d'expérimentations prenant en compte un large éventail des conditions de charge du système. Plus les mesures sur système réel seront nombreuses et détaillées, plus le modèle du système pourra être précis et permettre des études de performances raffinées.

[Klar87]

Afin d'obtenir des éléments de performance sur les systèmes multiprocesseurs autres que le débit et le temps de réponse, la saisie de traces d'événements pour chaque processus impliqué dans l'exécution d'un programme est une solution. Au monitoring software qui peut induire un overhead non négligeable pouvant fausser les résultats de l'analyse par interférence avec le programme étudié, il est préférable d'utiliser un moniteur hardware qui gère un ensemble de sondes disposées aux endroits stratégiques du système cible.

Pour les systèmes multiprocesseurs, le moniteur hardware doit respecter la structure distribuée de l'architecture du système et doit donc lui aussi être distribué. Zählmonitor 4 est un moniteur hardware distribué à architecture maître-esclaves. Le maître contrôle les mesures et réalise l'ordonnancement des traces issues des esclaves, chaque esclave gérant les sondes propres au processeur qui lui est associé. Le maître a également pour tâche l'analyse des traces et l'évaluation des performances du programme. Ce type de moniteur peut aider l'utilisateur à déterminer quelle peut être la meilleure configuration système (surtout si celui-ci est reconfigurable) pour résoudre un problème donné

et, associé à cette configuration, quel partitionnement du problème entre les différents processeurs adopter.

Afin de faciliter l'interprétation des résultats de l'analyse, Zählmonitor 4 permet de réaliser du monitoring hybride par adjonction dans le code source du programme de marques qui génèreront, lors de l'exécution, des signaux détectés par des sondes appropriées. Ainsi, à chaque événement correspondant à de tels signaux, le maître sera en mesure de situer l'événement dans le programme source et donc de mieux l'interpréter. Notamment, le processus ayant engendré l'événement sera ainsi identifié.

Un exemple de monitoring hybride est présenté pour un programme parallèle résolvant les équations différentielles partielles de Laplace par relaxation. La grille de discrétisation est partitionnée sur les différents processeurs. L'insertion de marques dans le code source est réalisée en langage de haut niveau (en l'occurrence Modula 2) par génération d'entrées-sorties immédiates (sans appel au superviseur) qui ne provoquent qu'un très faible overhead (de l'ordre de 5 microsecondes).

[Lubac84]

Un simulateur parallèle d'un réseau d'ordinateurs est décrit. Il est prévu pour être utilisé sur une architecture MIMD similaire au NYU Ultracomputer.

Un simulateur séquentiel a d'abord été réalisé utilisant une simulation temporelle dirigée à partir de tables. Puis, une parallélisation de ce simulateur a été opérée afin de pouvoir l'implanter sur l'ultra-ordinateur NYU. Un simulateur de cette architecture MIMD a été écrit afin de pouvoir déterminer, avant l'existence réelle du NYU Ultracomputer, l'efficacité de la parallélisation.

L'algorithme général du simulateur parallèle est donné et montre son utilisation possible pour tout réseau d'ordinateurs synchrones.

L'originalité de ce papier, outre la parallélisation de l'algorithme du simulateur séquentiel (mettant l'accent sur la reproductibilité des simulations par stockage des semences des générateurs de nombres aléatoires pour chaque noeud du réseau), réside véritablement dans la présentation de la méthode de simulation temporelle conduite par tables.

Cette méthode permet l'écriture rapide d'un simulateur pour tout réseau d'ordinateurs synchrones. En effet, l'algorithme général du simulateur reste identique lorsqu'on passe d'un réseau à un autre, seules les tables ont besoin d'être mises à jour.

Quatre types de tables sont utilisés:

- une table d'interconnexion globale qui donne pour chaque noeud la liste de ses voisins directs
- pour chaque terminal connecté sur le réseau (ou chaque utilisateur) une table de script décrivant la suite de commandes exécutées itérativement avec pour chacune d'entre elles la durée moyenne de réflexion de l'utilisateur (suivant une loi exponentielle); possibilité de prendre en compte l'ensemble des commandes supportées par l'ordinateur auquel le terminal est rattaché et de définir des boucles ou de simples délais (NOP)
- pour chaque commande pouvant apparaître dans une table de script, une table de scénario qui donne l'ensemble des processus mis en jeu par la commande, leur temps moyen de service (suivant une loi exponentielle) et leur localisation géographique sur le réseau; des branchements probabilistes sont utilisés pour exprimer des ruptures de séquence dans la succession des processus dues à des erreurs de paramétrage de la commande (Note: les erreurs de transmission sur les lignes de communication sont ignorées)
- pour chaque système d'exploitation utilisé sur le réseau, une table propre en décrivant les paramètres principaux et permettant ainsi de simuler l'activité système.

[Lubec88]

Une modélisation analytique de l'exécution d'un algorithme de Particle-In-Cell (PIC) sur hypercube iPSC est présentée.

La méthode PIC implémentée permet de simuler le mouvement de particules chargées sous l'influence d'un champ électrostatique. Elle est caractérisée par une faible granularité.

L'algorithme est décrit et trois implémentations parallèles possibles sur l'hypercube sont présentées. La troisième alternative sera la seule à faire l'objet de la modélisation. Elle consiste d'une part à répartir la grille de discrétisation sur l'ensemble des processeurs pour le calcul des

potentiels, d'autre part à répliquer la grille des potentiels sur chaque processeur pour le calcul des déplacements et assurer ainsi un équilibrage de charge entre les processeurs par équi-répartition des particules. Ces choix d'implémentation imposent une diffusion universelle des potentiels calculés sur chaque processeur en fin de première phase (calcul des potentiels) et une accumulation universelle des positions des particules en fin de deuxième phase (calcul des déplacements). Ces deux restructurations de données entraînent de nombreux échanges de messages entre processeurs et constituent une bonne part des communications entre processeurs. Toutes deux donnent lieu à la même complexité, qui dépend directement du degré de parallélisme possible pour les communications à partir d'un même noeud.

Un modèle permettant de calculer le temps d'exécution de la méthode PIC ainsi implémentée sur un hypercube est proposé. Les principaux paramètres du modèle sont d'une part des paramètres propres à la machine, tels que la dimension du cube (donnant le nombre de noeuds), le degré de parallélisme pour les communications, les vitesses de calcul des noeuds en modes scalaire et vectoriel, le temps d'initialisation des communications, la vitesse de transfert sur les canaux, d'autre part des paramètres propres à l'application, tels que le nombre de cellules (taille du problème), le pourcentage de code vectorisé, le nombre d'opérations en virgule flottante réalisées respectivement en séquentiel et en parallèle (ces nombres sont des fonctions linéaires du nombre de cellules).

Les paramètres propres à l'application sont estimés par approximation aux moindres carrés des temps de calcul d'une itération sur un noeud. De même, les paramètres liés aux communications entre voisins ont été "approximés" en mesurant les temps de communication entre voisins sur l'hypercube.

Le temps de calcul de chaque phase (calcul des charges des points de grille, calcul des potentiels, calcul des déplacements) et les temps nécessaires aux communications (calcul des potentiels par la méthode multi-grilles, diffusions pour la réorganisation des données) sont exprimés en fonction des paramètres d'entrée du modèle.

Une validation du modèle est réalisée en mesurant les temps de calcul et de communication de l'application PIC sur un hypercube réel, pour différentes valeurs du nombre de processeurs et du nombre de cellules, et en comparant ces temps aux temps obtenus par application des formules du modèle. Une utilisation du modèle est fournie où l'on considère cette fois-ci un hypercube prototype où chaque noeud serait un CRAY X-MP/1. Les accélérations, que l'on pourrait enregistrer lors de

l'exécution de l'application PIC pour différentes valeurs du nombre de processeurs et pour une taille de problème conséquente, sont dérivées grâce au modèle. Une accélération d'un facteur 10 serait possible avec un hypercube de dimension 4.

[McBry87]

Le papier présente tout d'abord une tentative de classification des architectures parallèles (distinction entre machines SIMD et machines MIMD, distinction entre systèmes à mémoire globale partagée et systèmes à mémoires locales, distinction possible suivant la structure du réseau d'interconnexion des processeurs).

Le second point abordé est une description sommaire de plusieurs systèmes multiprocesseurs, précisant notamment le nombre de processeurs, la structure du réseau d'interconnexion, la performance de crête exprimée en Mflops et la capacité mémoire. Parmi ces systèmes, citons:

1. TERA (Denelcor): 256 processeurs, 256 Gflops, mémoire partagée
2. iPSC/2 (Intel): 64 processeurs 80386 avec 3 co-processeurs et 16 Moctets de mémoire, 424 Mflops en double précision et 1280 Mflops en simple précision, hypercube
3. Ametek 2010: 1024 processeurs M68020 avec co-processeur M68882 et 8 Moctets de mémoire, 20 Gflops, système de routage de messages avec communication possible entre noeuds distants au travers de noeuds intermédiaires sans en interrompre le calcul (technique du "trou de ver" - worm-hole)
4. DAP AMT 510: tableau de 32 par 32 processeurs disposés sur une grille à deux dimensions, un bus par ligne et par colonne, 1 Mbit de mémoire par processeur, 60 Mflops, SIMD
5. Gemini (Paralex Research Inc.): 1000 processeurs, 500 Mflops, hypercube
6. Pegasus (Paralex Research Inc.): 512 processeurs, 8 Goctets de mémoire, 15 Gflops (1989)
7. Genesis (Paralex Research Inc.): 2 Teraflops (1991)
8. Connection Machine 2 (Thinking Machines Inc.): 65536 processeurs, 2000 processeurs à virgule flottante Weitek, 512 Moctets de mémoire supplémentaire, SIMD, hypercube avec système de routage de type "trou de ver" permettant des communications à 3 Goctets par seconde, possibilité de définir des processeurs virtuels (de l'ordre du million)

9. Myrias (Myrias Research Corp.): 65536 processeurs, 8 Goctets de mémoire, 1600 Mflops
10. RP3 (IBM): 512 processeurs (actuellement 64), 500 Mflops, mémoire partagée avec autant de bancs que de processeurs
11. Ultracomputer (NYU): 16 processeurs, identique à RP3 (utilisé pour le développement software de RP3)
12. SUPRENUM: grille de 16 par 16 processeurs, bus de communication pour chaque ligne et chaque colonne à 200 Moctets par seconde, 16 Mflops et 8 Moctets de mémoire par processeur (1989)
13. GF-11 (IBM): 576 processeurs (dont 64 de sauvegarde), réseau de Benes à trois étages reconfigurable à chaque cycle (1024 possibilités), 11 Gflops, SIMD
14. FPS T-Series: de 16 à 16386 INMOS T400 Transputer, 1 Moctets de mémoire, 16 Mflops et 4 voies de communication externes par noeud, hypercube
15. CCI Navier-Stokes Machine (Université de Princeton): 64 processeurs, 32 unités de calcul reconfigurables (fonctionnalité - additionneur, multiplieur - et inter-connectivité) et 640 Mflops par noeud, hypercube.

Ensuite, le papier s'intéresse à certains algorithmes parallèles, tels que la résolution d'équations différentielles partielles par la méthode du gradiant conjugué pré-conditionné, la résolution des équations des eaux peu profondes, les méthodes multi-grilles en deux et trois dimensions. Pour chaque algorithme, une implémentation sur la Connection Machine 2 (CM2) à parallélisme massif est présentée.

Pour le premier algorithme, une performance de 3,8 Gflops est obtenue avec 65536 processeurs sur une grille de discrétisation de 4096 par 4096 points.

Pour le second algorithme, une performance de 1714 Mflops est réalisée avec 65536 processeurs sur une grille de 2048 par 2048 points, à comparer aux 560 Mflops obtenus avec un CRAY X-MP4/8 à 4 processeurs sur une grille de 512 par 512 points, ce qui tend à dire qu'il est préférable d'utiliser du parallélisme massif que d'avoir recours à une vectorisation très performante.

Pour les méthodes multi-grilles, on montre que le parallélisme massif n'est pas toujours la meilleure solution, surtout pour des algorithmes à structure hiérarchique. En effet, dans ce cas précis,

il est très difficile de maintenir chaque processeur actif lorsque le maillage de la grille s'élargit. On aura dans de tels cas recours à des systèmes comportant quelques dizaines de processeurs tout au plus. Le problème est lié au fait que les méthodes multi-grilles ont été implémentées à partir d'un code séquentiel, respectant la structure hiérarchique de l'algorithme initial. Une nouvelle méthode, appelée méthode multi-grilles parallèle super-convergente (PSMG: Parallel Superconvergent Multigrid), est alors détaillée. Cette méthode rentre dans la catégorie des nouveaux algorithmes multi-échelles.

[Mink88]

L'avantage d'utiliser un outil hybride (hardware-software) pour mesurer le temps d'exécution de petites portions de code est présenté ici.

Après une large introduction qui dresse un historique des différents types d'instruments de mesures (seulement hardware, seulement software ou hybrides) et de leurs avantages et inconvénients respectifs, l'outil TRAMS (hardware assisted TRAce Measurement System) est décrit comme un outil de mesures adapté à de multiples architectures multiprocesseurs. Il est constitué d'une carte EDC (Event Data Card) connectée directement au bus reliant les différents processeurs. L'utilisateur doit insérer dans son code des instructions EDC spécifiques précisant l'événement correspondant ainsi que le numéro du processus. Par hardware, l'instant d'occurrence de chaque événement est enregistré ainsi que le numéro du processeur qui a généré cet événement. De plus, le mode d'utilisation du processeur (utilisateur ou superviseur) est récupéré. Par logiciel, l'information utilisateur (événement et numéro de processus) est ajoutée à l'enregistrement qui est transmis au buffer d'un analyseur logique.

Principal avantage de ce système: très faible overhead généré par la trace des événements qui correspond à des écritures en mémoire partagée (diminution de l'overhead d'un facteur de 100 par rapport à l'utilisation des procédures système standard).

Inconvénients: limitations dues à la capacité du buffer de l'analyseur logique (512 enregistrements), à la possible contention pour l'accès des processeurs parallèles à la carte EDC (vue par les processeurs comme une mémoire partagée).

Le papier montre l'utilité de l'outil pour mesurer l'overhead généré par l'exécution parallèle d'un programme d'application: création de processus, synchronisations entre processus (utilisation de verrous pour l'accès à des sections critiques), coûts d'initialisation et d'allocation dynamique de la mémoire partagée. Il conclut à la nécessité d'implémenter des programmes parallèles à forte granularité sur le système testé.

[Nacht88]

Un outil de mesure hardware pour systèmes multiprocesseurs à mémoire partagée est décrit. REMS (REsource Measurement System) permet, sans interférence avec le programme d'application étudié, d'obtenir des traces d'événements (correspondant à des enregistrements de trames de 32 bits, qui représentent principalement des adresses virtuelles) et des taux d'occupation de resources par mise à jour de compteurs, tels que le taux d'accès cache, le temps de latence moyen du bus interne de chaque processeur, le taux de service des différents composants (CPU, mémoire, ...) des processeurs.

Aucune illustration de l'utilisation pratique de l'outil n'est donnée dans le papier.

L'outil est composé d'un ordinateur d'analyse, relié à des unités d'échantillonnage (Sample Units). Une SU est connectée à chaque processeur du système étudié. Son rôle est d'échantillonner certains événements se produisant dans l'activité du processeur et de les stocker dans une mémoire locale. L'ordinateur d'analyse récupère en fin (voire en cours) de mesure toutes les données que les SU ont mémorisées dans leur mémoire locale afin de faire l'analyse des résultats des mesures.

Chaque SU est composée d'un comparateur de trames de bits (Pattern Matcher), relié à un processeur du système par des sondes. Son rôle est de comparer les signaux (de 32 bits), récupérés par les sondes, à des adresses spécifiques, que l'utilisateur lui a données par l'intermédiaire de l'ordinateur d'analyse avant le début des mesures. Si l'occurrence d'une adresse (qui pour l'utilisateur a une signification précise) se produit, chaque comparateur stocke alors son signal courant dans la mémoire locale de sa SU. L'occurrence d'un tel événement peut en outre déclencher une remise à zéro sélective de tous les compteurs de certains préprocesseurs de SU. En effet, chaque SU possède un préprocesseur dont le rôle est soit de maintenir un compteur de ticks représentant une estampille adressée à la mémoire locale de la SU à chaque occurrence d'un événement (correspondance entre

adresse utilisateur et signal sonde), soit de mettre à jour des compteurs de ratios à la fréquence de 10MHz, permettant en fin de mesure le calcul de taux d'occupation par l'ordinateur d'analyse. Un préprocesseur peut gérer jusqu'à 2 paires de compteurs, par un formatage très compact des données numériques (l'ensemble devant être représenté sur 32 bits), permettant ainsi le calcul de deux ratios.

Les limitations d'un tel outil de mesure sont énumérées.

1. Le premier problème quant à l'utilisation possible d'un tel outil est l'accessibilité des signaux. En effet, il peut s'avérer très difficile de placer les sondes dans les nouveaux microprocesseurs VLSI (Very Large Scale Integration), intégrant bien souvent la mémoire cache ainsi que le système de gestion de mémoire et rendant ainsi impossible l'accès aux adresses virtuelles et aux données concernant le cache (taux d'accès cache, taux de défaut de cache). Une solution au problème des adresses virtuelles pourrait être le recours à un outil hybride, où les traces d'événements seraient assurées, du moins partiellement, par des insertions de code dans le programme utilisateur [Mink88].

2. Un autre problème est la connectivité des sondes. Il est souhaitable pour une utilisation grand public d'un tel outil que des connecteurs standard puissent être employés pour connecter les sondes au système sous test. Cela peut demander de la part des fabricants la conception de composants avec option de connecteurs de sondes, quitte à un supplément de prix.

3. D'autres problèmes inhérents à l'architecture des processeurs peuvent apparaître, tels que la détection erronée d'instructions par anticipation inutile de chargement de code (instruction prefetch) ou le biais induit sur les instants d'occurrence de certaines instructions ou données dû également à des anticipations de chargement. Seul le recours à un outil hybride peut pallier ce problème, la contrepartie étant une interférence de l'outil avec le code utilisateur et donc une influence sur les mesures.

4. L'outil ne permet pas de distinguer les processus. Pour son utilisation, une assignation statique des processus aux processeurs est requise. Pour permettre une distinction des processus, une modification du scheduleur de processus serait nécessaire afin qu'il stocke dans un emplacement mémoire spécifique le numéro du processus qu'il active. Ce transfert de données pourrait alors être détecté par une sonde appropriée.

5. L'outil est encombrant, dans la mesure où il comporte une SU par processeur du système étudié. L'utilisation de VLSI pourrait permettre la réduction de chaque SU à une seule carte.

[Press88]

Trois techniques peuvent être utilisées pour évaluer les performances des réseaux, la modélisation analytique (principalement employée pour l'étude des couches inférieures des réseaux locaux - couches physique et liaison de données), la simulation (qui requiert une bonne connaissance software et hardware du réseau étudié) et les benchmarks (qui peuvent prendre en compte une grande complexité avec relativement peu d'effort).

Le papier présente l'application de la troisième alternative à la comparaison de 18 configurations de réseaux locaux d'IBM PC. Le serveur de fichiers ainsi que le programme de contrôle du réseau sont les principaux paramètres variables pour chaque configuration. Un PC joue le rôle de station de mesure, où s'exécutent les tâches principales. Les 10 autres PC du système génèrent de l'activité réseau, soit constante (correspondant aux heures de pointe d'utilisation du réseau), soit intermittente (correspondant plutôt à la charge d'un système transactionnel) par application d'une loi de distribution des accès (en l'occurrence une loi normale). L'activité réseau est réalisée par des tâches de fond s'exécutant sur les 10 PC.

Le code source des programmes des tâches de fond est fourni. Deux applications sont étudiées, le nombre de tâches de fond variant de 0 à 10, un traitement de textes ainsi qu'un gestionnaire de fichiers. Il s'avère, au vu des résultats, que la comparaison entre les 18 configurations de réseau aurait pu se faire en limitant le nombre de tâches de fond à 2 ou 3, dans la mesure où les mêmes résultats relatifs ont été obtenus avec les nombres de tâches de fond supérieurs.

[Rusch82]

Les études de performance de systèmes informatiques tentent d'analyser les relations entre configuration du système, charge du système, stratégies d'allocation et mesures de performance.

De telles études sont divisées en six phases, qui se répétent itérativement jusqu'à l'obtention du degré de précision désiré: identification du problème (formulation des objectifs précis de l'étude et des questions à résoudre), rassemblement d'informations, modélisation et sélection conjointe de la technique d'évaluation du modèle, évaluation du modèle, validation, implémentation ou application.

Plusieurs techniques d'évaluation, étroitement corrélées, sont couramment utilisées: la modélisation déterministe et l'optimisation combinatoire (utilisées notamment pour la détermination de stratégies d'allocation optimales, ou sub-optimales par l'emploi d'heuristiques afin de garantir un meilleur compromis entre gain en performance et nécessité de temps de calcul supplémentaire), la modélisation par files d'attente, l'analyse opérationnelle (qui repose essentiellement sur la mesurabilité sur le système réel des paramètres du modèle - facilitant ainsi la phase de validation par une forte correspondance entre modèle et système réel), la simulation (la plus générale et la plus détaillée des méthodes, mais la plus onéreuse aussi), l'expérimentation par utilisation de moniteurs hardware, de sondes microcodées, d'outils softwares (souvent génératrice d'un volume énorme de données difficiles à analyser) et l'analyse de données. Pour cette dernière technique cependant, la conception de l'expérience doit être pensée en fonction de l'analyse des données qui en résultera, afin de pouvoir tirer des conclusions significatives.

[**Sarge82**]

Le papier fait le point sur l'état actuel de la recherche en matière de validation de modèle. Il fait référence à de nombreuses publications.

Il définit tout d'abord la validation d'un modèle comme la justification que le modèle possède, à l'intérieur de son champ d'application, un degré de précision satisfaisant en accord avec les objectifs de son utilisation.

Le problème majeur dans cette définition est la détermination du degré de précision requis pour l'application du modèle.

Associées à la validation du modèle, les notions de crédibilité et d'évaluation du modèle sont définies. Une part de l'évaluation du modèle est l'analyse de la méthode de validation du modèle utilisée.

En général, la validité d'un modèle doit s'étudier à deux niveaux. La validité conceptuelle s'intéresse aux hypothèses qui ont été considérées pour la réalisation du modèle conceptuel (obtenu par analyse du système et modélisation). La validité opérationnelle, quant à elle, atteste que les caractéristiques principales du modèle représentent convenablement le système par rapport à l'utilisation prévue du modèle.

De nombreuses techniques de validation existent, aussi bien au niveau conceptuel qu'au niveau opérationnel, et peuvent être utilisées objectivement ou subjectivement, mais aucune méthodologie ne permet de définir quelle technique utiliser pour tel problème.

Trois phases peuvent cependant être distinguées dans le processus de validation: la validité à première vue (face validity) applicable aux deux niveaux, la vérification des hypothèses de modélisation au niveau conceptuel et le test des transformations entrées-sorties du modèle au niveau opérationnel.

Au niveau opérationnel, des techniques ont été élaborées pour les systèmes observables uniquement (systèmes à partir desquels on peut collecter des données). Si l'approche est objective, on confronte les résultats du modèle avec ceux du système par des tests statistiques (tests d'hypothèses ou intervalles de confiance). Si l'approche est subjective, la comparaison se fait par représentation graphique. Dans ce dernier cas, l'auteur souligne que bien souvent les comparaisons de moyennes peuvent s'avérer insuffisantes et qu'il est préférable de considérer aussi la variance ainsi que les valeurs extrêmes.

[Smith82]

Le modèle de l'exécution du software d'une machine parallèle pour éléments finis (Finite Element Machine) est présenté sous forme de graphe d'exécution.

Ce graphe prend en compte les constituants essentiels de l'overhead induit par la parallélisation que sont les synchronisations et les communications, elles-mêmes fonctions du degré de parallélisme (concurrency).

La méthode des éléments finis parallèle ainsi que l'architecture de la FEM sont décrites.

Le modèle de la méthode est détaillé ainsi que les avantages que peut procurer l'utilisation d'un tel modèle pour la détermination des performances du software (gain induit par un plus grand nombre de processeurs, par d'autres implémentations de la méthode, par d'autres choix d'architecture).

La traduction du modèle sous forme de réseaux de files d'attente peut permettre de prendre

en compte toutes les interactions entre les processeurs parallèles tels que la contention pour l'accès à des ressources partagées (par exemple le bus global de la FEM s'il devait être utilisé lors du calcul). Une traduction automatisée serait souhaitable. Un tel mapping n'a pas été réalisé dans le papier car les activités des processeurs parallèles pouvaient être considérées comme synchrones (mêmes temps de calcul et mêmes transferts de données).

[Som88]

Le problème de la détermination des facteurs prédominants dans le méta-modèle d'un système informatique est abordé. Un méta-modèle est l'expression de l'espérance mathématique du temps de réponse Y du système en fonction de termes Z_j, produits de paramètres d'entrée X_i du modèle, telle que:

$$E(Y) = \beta_0 + \sum_{j=1}^{t} \beta_j Z_j.$$

Une méthode qualitative et une méthode quantitative basées sur la simulation en fréquences introduite par Schruben et Cogliano ([Schru87]) sont décrites pas à pas et appliquées à un exemple de système informatique simple.

Ces deux méthodes permettent de déterminer les facteurs prépondérants intervenant dans le temps de réponse du système et de définir l'expression générale du méta-modèle.

Deux catégories de paramètres d'entrée du modèle sont distinguées:

1. les **paramètres structurels** tels que le nombre d'unités de disques, le nombre de terminaux,
2. les **paramètres non structurels** tels que le temps CPU moyen d'un job.

Enfin, une méthode de substitution des paramètres structurels discrets par des paramètres continus, n'altérant pas la validité du modèle, est décrite.

[Trele88]

Le papier fait un tour d'horizon des diverses tendances actuelles dans le domaine des architectures parallèles. Les différents langages utilisés pour la programmation parallèle ainsi que les

différentes architectures parallèles existantes ou en cours de développement sont répertoriés. Une classification des systèmes parallèles est ensuite proposée, illustrée pour chaque classe par la description d'un système spécifique. De nouvelles technologies (unités de calcul optiques, unités biologiques ou moléculaires), susceptibles à long terme de révolutionner la conception des systèmes parallèles, sont rapidement abordées.

Les principales motivations pour l'essor grandissant des systèmes parallèles sont:

- le gain de puissance considérable avec accroissement notable des performances
- la facilité de croissance des systèmes parallèles (par adjonction de processeurs)
- la tolérance aux pannes par duplication de composants ou détection de panne et fonctionnement en mode dégradé
- l'avénement sur le marché de microprocesseurs 32-bits très puissants
- le développement de langages de programmation parallèle (ex: OCCAM).

Seule ombre au tableau: rareté des logiciels parallèles sur le marché.

Cinq styles de langages sont utilisés pour la programmation parallèle:

- procédural, reposant soit sur le concept de mémoire partagée (ADA, FORTRAN 8X), soit sur celui de transfert de messages (OCCAM)
- orienté-objet (SMALLTALK)
- fonctionnel (LISP)
- logique (PROLOG)
- basé sur la connaissance (OPS5).

R.W. Hockney [Hockn85] considère deux classes principales pour les architectures parallèles MIMD, les architectures commutées et les architectures réseaux. Parmi les architectures commutées on distingue les systèmes à mémoires partagées et ceux à mémoires distribuées. Les systèmes réseaux sont subdivisés selon la topologie du réseau d'interconnexion et on distingue les réseaux maillés (anneaux, carrés), les cubes (hypercubes, cycles connectés en cubes), les réseaux hiérarchiques (arbres, grappes de grappes) et les réseaux reconfigurables.

P.C. Treleaven classifie les systèmes parallèles en cinq catégories.

- Les systèmes transactionnels regroupent les systèmes parallèles UNIX, (SEQUENT Balance 8000), les systèmes parallèles de base de données, les systèmes tolérants aux pannes et les systèmes de communication.
- Les super-ordinateurs numériques comprennent entre autres les hypercubes (INTEL iPSC).
- Les architectures VLSI (Very Large Scale Integration) suivent deux tendances, l'une à base de grilles de processeurs parallèles spécialisés telles que les tableaux systoliques, l'autre à base de microprocesseurs d'usage général à architecture RISC (INMOS Transputer).
- Les ordinateurs à langage de haut niveau, appelés également ordinateurs de la 5ème génération, utilisent les quatre styles de programmation symboliques, orienté-objet, fonctionnel, logique (FUJITSU Kabu-Wake) et basé sur la connaissance.
- Les systèmes neuroniques reposent sur un parallélisme massif et sont constitués d'un très grand nombre (de l'ordre de 100000 à 1000000) de processeurs soit spécialisés (modèle neuronique), soit programmables (modèle connectionniste - TM Connection Machine). Les processeurs sont dans les deux cas très primitifs.

[Wang81]

Les modèles analytiques permettent essentiellement des études de systèmes à un haut degré d'abstraction. Les simulations peuvent prendre en compte le niveau de détail souhaité, mais leur temps d'exécution est alors très long. Un problème commun à ces deux méthodes est le besoin de données empiriques pour alimenter le modèle en paramètres et étendre l'étude.

L'utilisation d'une architecture à machines virtuelles peut être un bon moyen pour modéliser les architectures distribuées et les réseaux informatiques. Une machine virtuelle simule chaque noeud du réseau en utilisant directement le software de l'application étudiée. Un moniteur de machines virtuelles (machine virtuelle lui-aussi) gère les machines virtuelles noeuds et simule toutes les interactions entre noeuds (communications de données notamment). Il a également à sa charge l'enregistrement de traces permettant l'évaluation des performances de l'architecture en fin de simulation. Le moniteur peut prendre en compte des noeuds dont les vitesses de traitement sont

différentes et c'est lui qui maintient l'intégrité des horloges sur les différents noeuds (un message ne peut être reçu sur un noeud avant d'avoir été envoyé depuis un autre noeud).

Un exemple de l'utilisation de ce type de simulation est présenté pour l'analyse des performances du niveau liaison du protocole X.25. La prise en compte d'erreurs de transmission est possible. Les principales données d'entrée de la simulation sont le nombre de noeuds, le nombre de tampons pour le stockage des messages en réception, la vitesse relative des noeuds, la présence de bruit provoquant des erreurs de transmission. Les résultats sont présentés sous forme de tables et représentent l'utilisation des voies de communication, le nombre de messages émis et reçus, le nombre de messages perdus (par saturation des tampons en réception), la queue moyenne des messages en attente à chaque noeud, le nombre d'erreurs de transmission.

[Wasse88]

Les performances de trois mini-super-ordinateurs, l'Alliant FX/8, le Convex C-1 et le SCS-40, sont comparées en leur faisant exécuter la série de benchmarks du Laboratoire National de Los Alamos. Ces benchmarks comportent des routines de base, mettant en oeuvre des opérations élémentaires, ainsi que des applications à structure hiérarchique, plus ou moins vectorisables.

Les trois mini-super-ordinateurs sont vectoriels et seul l'Alliant FX/8 comporte plusieurs processeurs (8). Dans un premier temps, un seul processeur de l'Alliant est considéré. Dans un deuxième temps, des calculs d'accélération sur l'Alliant sont réalisés à partir des programmes d'application de la première phase parallélisés automatiquement par le compilateur FORTRAN, suivant le mode COVI (concurrent outer-vector inner). Ce mode réalise la vectorisation des boucles internes et la répartition des indices de boucles externes sur les différents processeurs.

Malheureusement, du fait de l'absence d'analyse de dépendance de données inter-procédurale, si un appel de fonction ou de sous-programme a lieu à l'intérieur d'une boucle externe, seule la boucle interne est parallélisée, une portion des indices étant assignée à chaque processeur qui vectorise si possible le traitement (mode vector-concurrent).

En mode scalaire, le SCS-40 est 2 fois plus rapide qu'un processeur de l'Alliant et 3 fois plus rapide que le Convex. Le compilateur FORTRAN le plus performant pour la vectorisation est celui du Convex. Cependant, l'efficacité du code vectorisé est la plus importante sur le SCS-40 du fait de

sa fréquence d'horloge plus élevée et d'un débit de bus beaucoup plus important. Viennent ensuite le Convex (de 3 à 4 fois moins rapide) et l'Alliant (environ 5 fois moins rapide si l'on ne considère qu'un processeur). Faisant varier la longueur de vecteur de 10 à 1000, on s'aperçoit que l'Alliant atteint une valeur asymptotique de ses performances aux alentours de 25, alors que cette même valeur est de l'ordre de 100 pour les deux autres systèmes. En ce qui concerne les applications à structure hiérarchique, le classement des trois systèmes (où un seul processeur est considéré pour l'Alliant) s'établit comme suit: 1 SCS-40, 2 Convex, 3 Alliant.

Si l'on considère maintenant les huit processeurs de l'Alliant exécutant en parallèle les programmes d'application, par utilisation directe du code généré par le compilateur (qui a recours dans tous les cas au mode vector-concurrent pour les boucles les plus internes du fait de la présence d'appels à des fonctions dans les boucles externes), une accélération moyenne de l'ordre de 4 est obtenue. On remarque cependant que l'accélération tend à stagner lorsque le nombre de processeurs entraîne une longueur de vecteur inférieure à 25 (valeur asymptotique du gain de la vectorisation) pour chaque portion de boucle. Les performances de l'Alliant s'équilibrent mieux avec celles du SCS-40 (et en tous cas supplantent celles du Convex); elles sont supérieures pour 2 programmes, similaires pour 2 autres programmes et inférieures pour 5 autres (d'un facteur de 0.4 à 2).

[Woodb88]

Le papier présente la modélisation analytique par réseau de files d'attente d'une architecture temps réel multiprocesseur avec bus unique partagé.

La démarche de modélisation est classique:

1. étude du système hardware et software
2. modélisation par réseau de files d'attente
3. résolution analytique en considérant le processus markovien ergodique associé au modèle
4. expérimentation avec mesures hardware et software afin d'obtenir les paramètres d'entrée du modèle et certains critères de performance
5. validation du modèle par comparaison des résultats analytiques et expérimentaux.

Le système est constitué d'un bus partagé par plusieurs processeurs, disposant chacun d'une mémoire locale, par une mémoire externe et par plusieurs ports d'entrée-sortie pour la connexion de capteurs, d'automates, ... Tous les éléments du système sont en fait dupliqués pour des raisons de fiabilité. La charge du système est constituée par le noyau exécutif du système, qui gère les ressources software et hardware, et des tâches utilisateur qui contrôlent les éléments externes du système. Ces tâches sont réparties en classes. Chaque classe est caractérisée par une fréquence d'occurrence des tâches et par une distribution des temps de traitement sur les processeurs. Une tâche s'exécute sur un seul processeur et est constituée de trois phases, l'acquisition des données depuis des éléments externes ou la mémoire externe, le traitement proprement dit et le transfert des résultats sous la forme d'écritures en mémoire externe ou d'actions sur les éléments externes. Différentes priorités entre tâches sont considérées et le bus est alloué selon la politique non préemptive avec priorités. Une tâche particulière possède la plus forte priorité, c'est une boucle d'attente active lorsque le processeur est oisif.

Le réseau de files d'attente modélisant le système est un réseau fermé comportant $n+2$ stations, si n est le nombre de classes de tâches. Une station représente le bus, les ports d'entrée-sortie et la mémoire externe. Le service à cette station comprend chaque transfert d'information entre processeur, mémoire externe et éléments externes au système. Une deuxième station correspond à l'état oisif d'un processeur. Pour chaque classe de tâches enfin, une station représente l'état du processeur actif pour une tâche de cette classe. Les clients sont par conséquent, non pas les tâches, mais les processeurs. Les services correspondent à des temps de séjour des clients dans un certain état: oisif, en cours de transfert, actif pour une classe de tâches donnée.

On définit l'état du système comme le $(n+2)$-uplet formé pour chaque station du nombre de clients dans la station, les clients étant banalisés car tous identiques. La résolution du modèle se fait en considérant le processus markovien ergodique associé au modèle et en calculant les probabilités d'état du système en régime stationnaire (ce qui est justifié par le fait que le système a été conçu pour être fiable).

On dérive ensuite les quantités suivantes:

- la probabilité pour un processeur d'être oisif
- la probabilité de contention du bus

- le temps moyen d'attente d'un processeur pour l'accès au bus
- le temps moyen de réponse du bus.

On considère que toutes les lois de distribution des services sont exponentielles et que la même loi de service est satisfaite par toutes les classes de tâches pour le bus.

Afin de valider le modèle, une expérimentation a été réalisée sur le système FTMP (Fault-Tolerant MultiProcessor) de la NASA. Deux charges de système ont été générées par la construction de tâches utilisateur synthétiques à l'aide de l'outil SWG (Synthetic Workload Generator), développé par l'Université de Carnegie-Mellon [Feath84]. Des mesures hardware (utilisation d'un analyseur logique connecté au bus) et software (calcul de temps par estampilles insérées dans le code des tâches) ont permis d'obtenir d'une part les paramètres d'entrée du modèle, d'autre part les critères de performance qui nous intéressent. Les valeurs expérimentales ont été comparées aux résultats fournis par le modèle. L'erreur, de l'ordre de 10% à 15%, est explicable par la loi de distribution de l'oisiveté des processeurs, dont la variation dans la réalité est bien moindre à celle du cas exponentiel, par un manque de variation des temps de transfert sur le bus, du fait de l'utilisation du générateur SWG, et par l'ajustement qui a été nécessaire pour la détermination des taux de service moyens des stations représentant l'état de traitement des processeurs pour chaque classe de tâches.

[Wybra88]

Ce papier présente un moniteur hybride pour systèmes informatiques distribués. Ce moniteur est composé de processeurs de test et de mesure (TMP) connectés à chaque noeud du système distribué. Un réseau d'interconnexion relie chaque TMP à une station de test centrale, qui affiche graphiquement toutes les mesures de performances que l'utilisateur désire.

Un tour d'horizon des différents moniteurs pour systèmes informatiques est tout d'abord proposé, appuyé par une bibliographie très complète. Les moniteurs software, hardware et hybrides sont considérés successivement.

Les principales qualités d'un moniteur pour système distribué sont ensuite énumérées:

1. possibilité de réaliser le monitoring pendant l'opération du système éventuellement reconfigurable en cours de mesures

2. aucune modification du comportement du système et de ses performances due au moniteur
3. présentation des résultats de mesures en temps réel et au niveau application
4. facilité d'emploi
5. intégration dans le système permettant une utilisation permanente sans préparatif préalable
6. adaptabilité à diverses architectures et applications en cours de fonctionnement
7. feed-back (utilisation des résultats de mesures pour une modification automatique du système sous test, par exemple adaptation d'une application pour optimiser l'équilibrage de charge).

Le monitoring s'effectue par la capture software d'événements sur chaque TMP, relié au bus système du noeud. On distingue les événements standard (associés à la gestion des processus, aux transferts de messages, au fonctionnement propre du TMP, au noyau exécutif du noeud), dont le code nécessaire à la capture est inclus dans le système d'exploitation du noeud, des événements optionnels, dont l'utilisateur a la charge de générer la capture par l'insertion d'instructions assembleur STORE aux endroits adéquats du code de l'application. 256 classes d'événements peuvent être définies.

L'overhead système provoqué par la capture des événements se résume donc au marquage des événements par adjonction d'instructions STORE. Il est de l'ordre de 0.1% pour tous les événements standard. De plus, cet overhead est permanent et constant du fait de l'intégration du marquage (du moins pour les événements standard) dans le système d'exploitation des noeuds.

Chaque TMP a la charge de la gestion de ses événements propres, notamment la constitution de statistiques concernant chaque processus s'exécutant sur le noeud qui lui est associé. Sur la demande de la station de test centrale, il adresse ses résultats locaux que la station centrale pourra incorporer à ceux des autres TMP pour la production de statistiques globales.

Une décomposition hiérarchique du système distribué est considérée. Ainsi, cinq niveaux sont pris en compte: système, grappes d'unités de distribution (éventuellement divisé en sous-niveaux), machines, unités de distribution (constituées de un ou plusieurs processeurs accédant une mémoire partagée), processus. A chaque niveau correspond un certain nombre de mesures de performances, qui seront calculées et stockées à l'échelon local par chaque TMP.

La station de test centrale dispose d'un écran graphique haute résolution pour la présentation graphique de toutes les mesures de performances, et ce à chaque niveau de la hiérarchie.

Afin de permettre une meilleure validation du moniteur, un simulateur de systèmes distribués et de TMP a été développé. Ce simulateur permet de simuler l'activité de certaines applications sur différents systèmes et d'obtenir les mesures de performances correspondantes. Le comportement des divers systèmes vis-à-vis des applications considérées peut ainsi être estimé, sans implémenter les applications sur système réel.

Table des Matières

Liste des Illustrations

Figure

Résumé

Une méthodologie de modélisation adaptée à l'évaluation des performances des architectures parallèles est présentée. Elle repose sur une décomposition du processus de modélisation en sept phases: formulation des objectifs, analyse du système, conception du modèle, estimation et mesure des paramètres d'entrée du modèle, résolution du modèle, validation du modèle et utilisation du modèle pour prédire les performances du système réel.

Les principaux points forts de cette méthodologie sont l'utilisation de méthodes d'agrégation, la nette distinction entre le modèle de l'architecture et le modèle des programmes, le recours à l'analyse des données pour la formation de classes de programmes, le compromis entre un modèle détaillé validé sur quelques exemples précis et un modèle probabiliste permettant une étude exploratoire générale, et une méthode originale de génération aléatoire d'événements corrélés utilisée lors de la modélisation probabiliste des programmes.

La méthodologie est illustrée de manière fort détaillée au travers de quatre études de cas: les réseaux point-à-point de systèmes HP3000, un réseau local d'IBM PC, le calculateur scientifique FPS264 et le système lCAP/FPS264 ("loosely Coupled Array of Processors FPS264") conçu à IBM Kingston. Ces études de cas sont bien différentes de par la nature du système modélisé, le formalisme de description du modèle (réseau de files d'attente ou réseau de Petri stochastique) et la méthode employée pour résoudre le modèle (simulation ou méthode analytique, avec ou sans agrégation).

Mots-Clé

architectures parallèles - évaluation de performances - mesures de performances - méthodes d'agrégation - méthodologie de modélisation - réseaux de files d'attente - réseaux de Petri stochastiques - simulation

Abstract

A modeling methodology for evaluating the performance of parallel architectures is presented. It is based on a decomposition in seven stages of the modeling process: objective expression, system analysis, model design, estimation and measurement of the model input parameters, model solving, model validation, and model utilization for predicting the performance of the real system.

The main key points of this methodology are the application of aggregation methods, the thorough distinction between the model of the architecture and the model of the programs, the use of data analysis techniques to define program classes, the trade-off between a detailed model validated in a few precise cases and a probabilistic model suited for general exploratory analysis, and an original method, that can randomly generate correlated events, used for the probabilistic modeling of the programs.

The methodology is illustrated in much detail through four case studies: HP3000 point-to-point networks, a local area network of IBM PC, the scientific computer FPS264 and the lCAP/FPS264 system (loosely Coupled Array of Processors FPS264 system) designed at IBM Kingston. These case studies present significant differences in the nature of the modeled system, in the formalism adopted for describing the model (queueing network or stochastic Petri net), and in the method used for solving the model (simulation or analytic method, with or without aggregation).

Keywords

aggregation methods - modeling methodology - parallel architectures - performance evaluation - performance measurements - queueing networks - simulation - stochastic Petri nets