UNIVERSITÉ DU QUÉBEC

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INGÉNIERIE

PAR

ERIC CHABOT

B.ING.

PARALLÉLISME ET COMMUNICATIONS DANS LES APPLICATIONS SCIENTIFIQUES (FORTRAN)

Décembre 1993

Bibliothèque
Paul-Émile-Boulet

UQAC
Université du Québec
à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

Remerciements

Il est utopique de penser que l'atteinte de la qualité globale d'un travail de recherche peut être assumé par l'intervention d'un seul individu. Ainsi, je profite de cette opportunité pour remercier toutes les personnes qui ont collaboré à l'élaboration de ce mémoire. Je remercie particulièrement M. Daniel Audet Ph. D., mon directeur de recherche, pour son étroite collaboration, ses conseils éclairés et sa très grande disponibilité.

Je ne pourrais passer sous silence le support sans faille que m'a consacré ma conjointe Mme France Gauthier. Aussi, j'aimerais remercier mes parents pour m'avoir transmis le goût pour l'avancement des limites de la connaissance et ce, depuis ma plus tendre enfance.

Enfin, je tiens à remercier tout spécialement la ligne de code A = B + C pour son entière dévotion et son altruisme tout au long de ce travail de recherche.

Résumé

Ce mémoire de recherche présente une méthode d'évaluation du potentiel de concurrence contenu dans des applications scientifiques programmées en FORTRAN et ayant été écrites en vue d'une exécution séquentielle sur un ordinateur conventionnel. Cette évaluation cherche à déterminer l'impact des coûts de communication ainsi que le nombre maximal de processeurs à utiliser afin d'exécuter le plus rapidement possible une application scientifique sur un système parallèle ayant une architecture donnée. Trois classes d'architectures multi-ordinateurs ont fait l'objet d'une étude particulière, à savoir: les architectures hypercubes, en grille et à bus simple.

Un logiciel s'inspirant des techniques d'insertion de moniteurs utilisées pour obtenir des statistiques concernant les programmes exécutés sur des ordinateurs conventionnels, a été élaboré. Celui-ci permet d'établir, durant l'exécution de ces programmes, le parallélisme maximal pouvant être obtenu avec ou sans contraintes, telles que le nombre de processeurs et le coût d'une communication, et ce, en fonction de l'architecture choisie. Cet outil s'appelle OSASMO, qui est l'acronyme de: **O**util de **S**imulation **A**rchitecturale pour **S**ystèmes **M**ulti-**O**rdinateurs permet donc l'analyse automatique de n'importe quelle application écrite en FORTRAN standard.

Dix applications scientifiques connues ont été analysées à l'aide de l'outil logiciel. Plus de 700 simulations ont été exécutées afin d'obtenir une vue d'ensemble de tous les facteurs intervenant dans l'exécution d'un programme parallèle. Plusieurs simulations confirment certains résultats expérimentaux observés par d'autres chercheurs. De plus, plusieurs autres résultats d'importance pour les utilisateurs et les concepteurs de systèmes parallèles sont présentés. La conclusion la plus intéressante de ce travail est que, généralement, peu de processeurs sont nécessaires pour réaliser l'exécution parallèle optimale d'applications scientifiques.

Table des matières

Liste des figures

Liste des tableaux

INTRODUCTION

Les systèmes d'ordinateurs ont subi, depuis la venue des puces de silicium, une course incessante en vue de leur miniaturisation. Actuellement, la technologie des circuits d'intégration à très grande échelle nous permet de faire des manipulations en-deça du micron voire même au niveau de l'atome [19]. L'utilisation de ces technologies nous amène à un constat: la diminution des temps de calcul des ordinateurs conventionnels obtenue par l'augmentation de la vitesse des transistors ainsi que de leur nombre par unité de surface a des limites pour une technologie donnée. D'autres moyens doivent donc être mis à la disposition des concepteurs de systèmes informatiques. Une des méthodes les plus utilisées consiste à faire travailler en parallèle, c'est-à-dire simultanément, plusieurs unités de calcul sur une même tâche. Ces systèmes sont dits à architectures parallèles.

Les systèmes parallèles actuels sont, en général, complexes tant au niveau de leur structure que de leur programmation. C'est pourquoi, pour optimiser leur efficacité, certaines techniques utilisant des moniteurs [12] sont utilisées. Un moniteur permet, habituellement, d'extraire d'une application scientifique les caractéristiques reliées au traitement parallèle. Par exemple, il peut être intéressant de connaître combien de processeurs une application scientifique peut utiliser efficacement [12]. Cette information peut permettre d'utiliser certaines portions d'un système de grande dimension à d'autres fins. Cependant, ce nombre ne permet pas, à lui seul, de juger du degré de performance réel

qu'un programme aurait s'il était exécuté sur une machine parallèle. Plusieurs autres facteurs influencent, à des degrés différents, les performances, tels l'agencement de la mémoire, l'architecture du réseau de communication de l'ordinateur, les coûts de la gestion interne des ressources ("overhead"), etc. Ces paramètres doivent être considérés dans l'expression des critères de performances.

Le coût des communications, qui est fonction du nombre de processeurs et du réseau de communication, est un des facteurs qui influencent le plus la topologie et les performances des systèmes parallèles. À la lumière de la recherche bibliographique effectuée, peu de travaux ont été réalisés sur l'évaluation quantitative des coûts des communications. La présente recherche propose une méthode de quantification de ces coûts et met en évidence la relation qui existe entre le nombre de processeurs et la communication inter-processeur pour différentes architectures du réseau de communication.

Un logiciel complexe a été développé pour implanter de façon automatique, dans des applications scientifiques séquentielles, un moniteur qui quantifie les caractéristiques mentionnées ci-haut. Un des avantages de ce logiciel est que l'utilisateur n'a pas à connaître les détails d'un système parallèle particulier ni même les applications scientifiques étudiées, ce qui lui confère, notamment, une très grande facilité d'utilisation.

Chaque programme source analysé est décomposé en code à trois adresses, ce qui a pour effet de réaliser une étude plus réaliste du comportement des processeurs. En effet, la majorité des processeurs actuels ne peuvent exécuter qu'une instruction à la fois. Les lignes de code à trois adresses reflètent bien ce fait. De plus, une table d'occupation des processeurs est construite à même l'exécution du programme source augmenté (incluant le moniteur). Cette table permet de s'assurer que chaque processeur n'utilise chaque unité de

temps relatif que pour l'exécution d'une seule instruction. Ceci permet de se rapprocher suffisamment de la structure physique sans pour autant être dépendant d'une machine précise. La trop grande fidélité à la structure physique des simulateurs ou encore les études sur des machines spécifiques empêchent, justement, de connaître le degré réel de concurrence et le coût des communications d'une application scientifique.

CHAPITRE I

Problématique

1.1 Introduction

Plusieurs méthodes peuvent être utilisées afin de tirer des informations concernant le parallélisme d'une application. En effet, on peut tout d'abord mesurer directement ce parallélisme en effectuant des essais sur un ensemble de systèmes. Cependant cette méthode est coûteuse et est souvent biaisée par les détails architecturaux propres à chacune [4]. D'autres méthodes souvent plus efficaces proposent l'utilisation de moniteurs. Parmi ces dernières, celle de Kumar [12] présente des caractéristiques intéressantes. Cependant celle-ci ne tient pas compte des coûts temporels engendrés par les communications inter-processeur, ni des stratégies d'allocation des ressources de calcul. De plus, elle considère qu'un processeur peut exécuter plusieurs instructions de code durant un même cycle machine en durée relative. Ces dernières caractéristiques n'étant pas estimées, il est difficile d'évaluer de façon réaliste les performances d'applications scientifiques sur des systèmes parallèles en ne jugeant que la mesure de parallélisme maximale. D'ailleurs Stone [17] révèle que l'utilisation de tous les processeurs n'est pas toujours la solution optimale. En effet, le coût des communications augmente généralement avec le nombre de processeurs

utilisés. Stone conclut en spécifiant que la mesure maximale de parallélisme n'est pas synonyme de vitesse maximum. D'après ce dernier, la communication doit être prise en considération dans l'évaluation des performances. Il énonce que les performances dépendent largement du ratio R/C où R est le temps d'exécution d'un certain nombre d'instructions regroupées en tâche, et C est le coût des communications produites par ces mêmes instructions.

1.2 Gain de vitesse des systèmes parallèles

Un des facteurs de quantification largement utilisé est le gain de vitesse [3, 8, 9, 10, 18], défini comme étant le temps d'exécution d'un certain nombre d'instructions de façon séquentielle (T_1) sur le temps d'exécution de ces mêmes instructions avec N processeurs en parallèle (T_N).

$$S = T_1 / T_N \qquad (1)$$

Dans le meilleur des cas, une application (tâche) pouvant être fractionnée en N sous-tâches indépendantes, chacune étant exécutée par un processeur différent aurait un gain de vitesse égal à N. En général, cette valeur idéale est difficilement atteignable dû, en majeure partie, aux coûts de gestions des ressources internes (initialisation, synchronisation, ...) et des coûts de communication. La figure 1 montre un exemple de gain de vitesse idéal et réel. D'après Herzog [8], dans certains types d'applications, la courbe réelle est pratiquement confondue à la courbe idéale pour des architectures vectorielles. C'est le cas, notamment, de certaines applications scientifiques, d'automatisation des outils VLSI, d'intelligence artificielle, d'opérations sur des bases de données, etc.

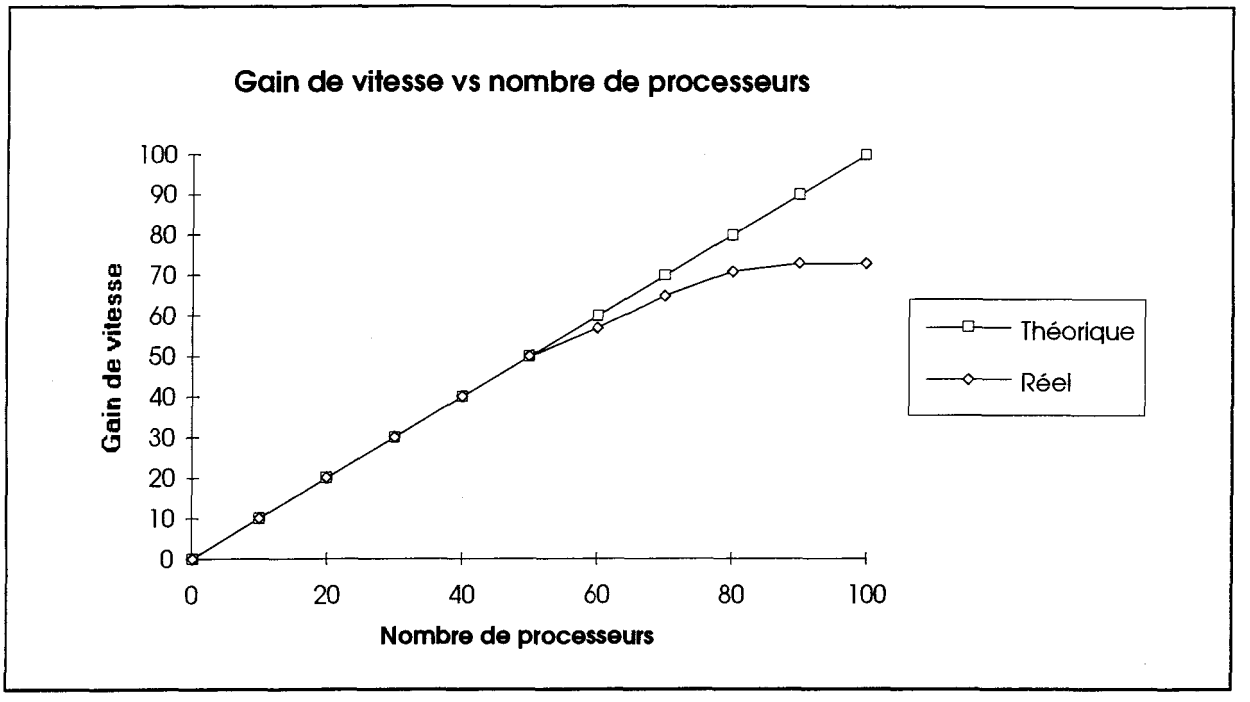

Figure 1 Gain de vitesse idéal et réel.

Dans le même ordre d'idées, Fromm *et al.* [6] dans leurs recherches, ont montré par des modèles théoriques, ainsi que de façon expérimentale sur des architectures vectorielles, que la diminution du gain de vitesse devient généralement prohibitive pour des coûts de communications excédant les dix (10) pourcent du temps moyen d'exécution. Par exemple, considérons un système parallèle à quatre (4) processeurs avec un temps d'exécution moyen par sous-tâche de 10 ms et ayant un coût de communication de base de 1 ms (10% du temps d'exécution moyen). Démarrer les quatre (4) sous-tâches dans les processeurs requiert 4 opérations de communications, donc 4 ms. Le calcul du gain de performance est le suivant:

$$\text{Gain de vitesse} = 40 \text{ ms} / (10 \text{ ms} + 4 \text{ ms}) = 2{,}86 \qquad (2)$$

alors qu'avec un coût de communication de 0,1 ms (1%), le gain de performance donne comme résultat

$$\text{Gain de vitesse} = 40 \text{ ms} / (10 \text{ ms} + 0{,}4 \text{ ms}) = 3{,}84 \qquad (3)$$

Le résultat de l'équation (2) montre bien la dégradation du gain de vitesse causé par le coût prohibitif des opérations de communication. En contre partie, le résultat de l'équation (3) illustre que, pour un coût de communication de 1% des sous-tâches, la dégradation du gain de vitesse reste acceptable. Ceci confirme que les communications doivent être prises en considération dans l'évaluation des performances des systèmes parallèles.

La figure 2 présente l'allure de ce type de courbe. Le seuil acceptable de dégradation du gain de vitesse est à 10% comme introduit précédemment et correspond au point ayant la valeur, en ordonnée, de 2,86.

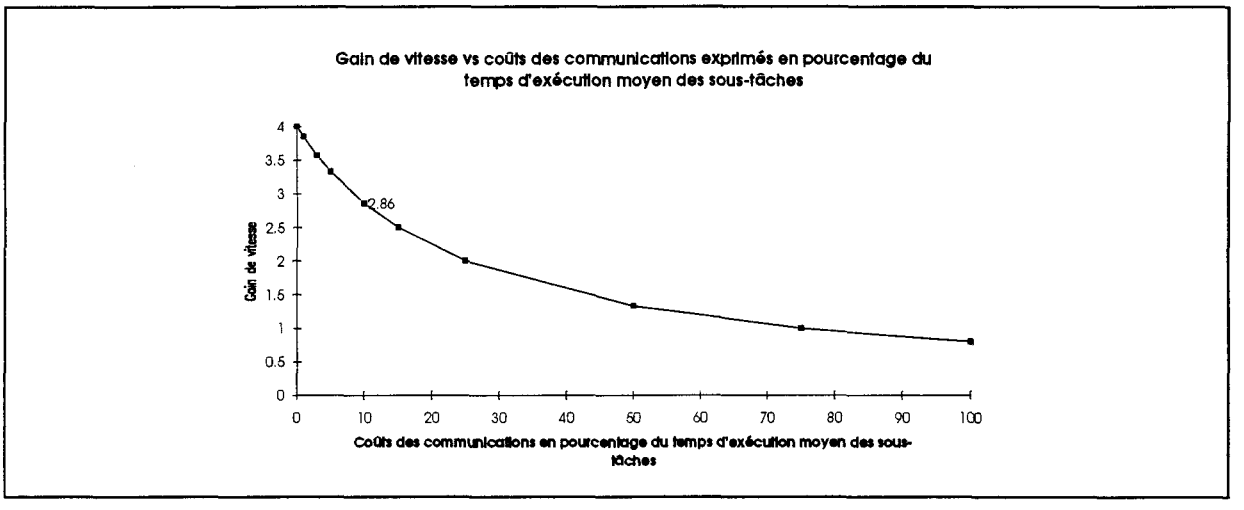

Figure 2 Courbe caractéristique du gain de vitesse en fonction du coût d'une communication

1.3 Critères d'évaluation des systèmes parallèles

Un des problèmes relié à l'évaluation des performances du réseau de communication des systèmes parallèles est la façon de les exprimer. Trop souvent, des travaux équivalents ne peuvent être comparés en raison de la façon dont ces performances sont présentées. Par

ailleurs, Dongarra *et al.* [4], rapportent que la marge d'erreur dans l'évaluation des performances augmente proportionnellement au niveau de performance de la machine étudiée. Soulignons que, lors des experimentations, quelques bancs d'essai écrits en FORTRAN sont utilisés. Ceci permettra de constater le degré d'adaptation de chacun d'eux sur des architectures parallèles. Pour la présentation des résultats, Levitan [14] propose des critères d'évaluation de la structure de communication appelés "métriques".

1- diamètre: Le diamètre est défini comme étant la plus longue distance que doit parcourir un message lorsqu'un processeur communique avec un autre.

2- Largeur de bande: La largeur de bande est le nombre total de messages qui peut être reçu ou envoyé par un processeur dans le système durant un cycle d'exécution d'une instruction.

3- Calcul de la trajectoire: Le calcul de la trajectoire consiste en la détermination du nombre de trajectoires dont chaque processeur dans le réseau est responsable. Une trajectoire est définie dans ce cas particulier comme l'envoi d'un message à chacun des autres processeurs du système.

4- Étroitesse: L'étroitesse se mesure en divisant les processeurs d'un système en deux groupes inégaux (le premier étant plus grand ou égal au deuxième). Le nombre d'interconnexions entre les deux (2) groupes devient ainsi l'étroitesse du réseau.

5- Épaisseur: L'épaisseur se mesure en divisant en deux groupes égaux les processeurs du système. Chaque processeur d'un groupe envoie un message à un processeur de l'autre groupe. En calculant le temps normalisé pour réaliser ce travail, on trouve l'épaisseur du réseau.

Certains de ces critères d'évaluation seront utilisés pour définir les hypothèses de départ qui fixeront la profondeur de l'étude.

D'après Qin *et al.* [16], les facteurs qui influencent le temps d'exécution sont:

1- Le nombre des données d'entrées.

2- Les algorithmes utilisés dans le calcul.

3- Le type de structure de données.

4- La vitesse des processeurs.

5- Le nombre de processeurs disponibles.

6- La stratégie d'allocation des ressources de calcul.

7- La communication entre les unités de calcul (communication inter-processeur).

8- Délai de gestion interne ("overhead").

9- L'état de la charge de travail de l'environnement des processeurs.

Ce travail considère principalement les facteurs 5, 6, et 7, car ils varient en fonction du type d'architecture. Un des objectifs de ce travail est de recueillir des données propres à

certaines architectures et de considérer ces dernières comme idéales. De ce fait, on évite de considérer les facteurs limitatifs particuliers à une architecture. Par exemple, on constate que les facteurs 4, 8, et 9 sont dépendants d'un système particulier. Les facteurs 1, 2, et 3, quant à eux, seront les mêmes ou du moins ne donneront pas d'informations pertinentes sur les performances d'un type d'architecture.

1.4 Méthode d'analyse développée

Le présent travail s'inspirera de l'expérimentation de Kumar [12], en incorporant le coût des communications de façon à quantifier à la fois le degré de parallélisme et le coût des communications présentes dans des applications scientifiques écrites en langage de haut niveau (FORTRAN). Il permettra, entre autres, de vérifier l'effet des coûts des communications sur les performances que Stone [17] définit à l'aide de modèles théoriques généraux.

Le niveau de la présente étude se situe entre la macro et la micro analyse des coûts temporels d'exécution, selon la définition qu'en donne Qin *et al.* [16]. Effectivement, la macro analyse utilise une instruction typique et l'exécute un certain nombre de fois et il en résulte un indice des performances. La micro analyse étudie le coût temporel d'exécution de chaque instruction d'un programme, en considérant que tous les facteurs énoncés précédemment sont considérés.

Les méthodes suivantes de fragmentation de programme aussi appelées stratégies d'allocation des ressources de calcul seront plus particulièrement étudiées:

1- Allocation aléatoire ligne-à-ligne: Les ressources de calculs sont allouées de façon aléatoire à chaque ligne de code.

2- Allocation aléatoire par bloc de base: Les ressources de calculs sont allouées suivant le fractionnement d'un programme en bloc de base (tel que définis dans [1]) avec une profondeur de retrait variable. Les processeurs de chaque bloc de base sont alloués de façon aléatoire.

L'utilisation d'applications scientifiques classiques déjà écrites en FORTRAN, nous évite de programmer des versions dédiées à des systèmes parallèles spécifiques. Cela impliquerait, en effet, une connaissance et une compréhension, à la fois, des méthodes numériques parallèles et des langages de programmation parallèles propres à chaque système. Ceci constituerait une somme de travail colossale pour chaque système étudié. De plus, une perte potentielle dans la finesse du parallélisme serait à craindre puisque la recherche de l'utilisation optimale des caractéristiques de la machine serait confiée au programmeur.

Les applications scientifiques étudiées seront décomposées en lignes de code à trois adresses ("three-address code" [1]), ce qui se traduira par une quantification plus exacte du parallélisme et des coûts de communications contenus dans celles-ci. Il est important de mentionner que cette approche est plus près de la réalité sans être reliée à un système particulier, du point de vue de l'exécution des instructions, car la majorité des processeurs actuels ne peuvent exécuter plus d'une opération à la fois.

Le chapitre II étudie en profondeur les aspects pratiques de l'outil logiciel servant à implanter le moniteur dans les applications scientifiques. Le logiciel a été baptisé OSASMO, qui est l'acronyme d'**O**util de **S**imulation **A**rchitecturale pour **S**ystèmes **M**ulti-**O**rdinateur. Les méthodes d'expérimentation, les hypothèses de départ ainsi que la validation des instructions synthétiques sont détaillées au chapitre III. Les résultats obtenus

ainsi que les discussions et le développement d'une métrique sont présentés, au chapitre IV. Finalement, la conclusion de ce travail met en évidence les enseignements que l'on peut tirer des expérimentations et de leurs résultats. De plus, des perspectives pour des recherches ultérieures sont aussi présentées.

CHAPITRE II

OSASMO — Un outil de simulation architecturale

OSASMO est un logiciel qui implante un moniteur dans des programmes écrits en FORTRAN. Ce moniteur recueille les données statistiques nécessaires à la réalisation de ce projet. Les principes, les techniques ainsi que les particularités du moniteur font l'objet de ce chapitre.

2.1 Aspects pratiques de l'insertion du moniteur

La méthode utilisée pour recueillir les statistiques de parallélisme et de communication consiste à insérer un moniteur à l'intérieur même d'un fichier source d'une application scientifique écrite en langage de haut niveau, le FORTRAN. La figure 3 illustre le principe d'insertion du moniteur dans le fichier source d'une application scientifique.

Pour insérer un moniteur qui correspond à l'application étudiée, il est impératif de réaliser, au préalable, une analyse lexicale et syntaxique de celle-ci. Chaque ligne du code source est analysée et la ou les lignes de moniteur correspondant à cette dernière sont écrites à même le fichier source **P**, comme le montre la figure 3. Ce sont les lignes de code du moniteur **M** qui créent la trace de la simulation parallèle de l'exécution du programme scientifique et qui s'occupent de recueillir les résultats traitant du parallélisme et de la communication. Le résultat de l'insertion est le programme augmenté **P'**.

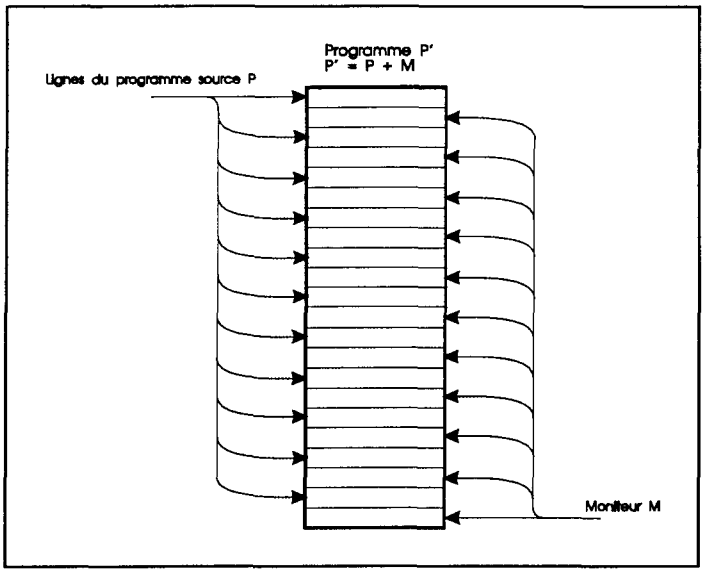

Figure 3 Insertion du moniteur dans un fichier source d'une application scientifique.

L'insertion du moniteur s'effectue de façon automatique à l'aide d'OSASMO qui est constitué d'un analyseur lexical et d'un analyseur syntaxique (ou grammatical). En fonction des règles lexicales et grammaticales reconnues, OSASMO insère certaines lignes de code.

L'analyse lexicale et syntaxique de l'application scientifique est réalisée à l'aide d'outils dédiés à cette tâche. Ces outils sont incorporés dans le système d'exploitation UNIX et se nomment respectivement LEX et YACC (**Y**et **A**nother **C**ompiler-**C**ompiler) [1, 2].

Ces deux outils sont utilisés de façon interactive comme l'illustre la figure 4. Cette figure montre que Lex procède à l'analyse lexicale du fichier source et envoie à YACC les jetons reconnus. YACC, une fois le jeton analysé, envoie à LEX un signal pour obtenir le prochain jeton. LEX construit une table des symboles pour tirer parti du contexte d'utilisation des divers jetons, et partage cette table des symboles avec YACC si besoin est.

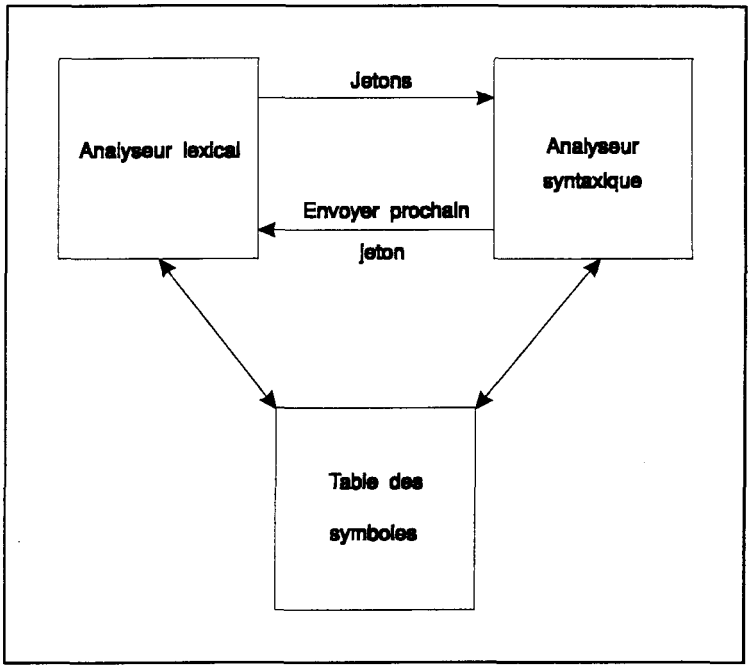

Figure 4 Interaction de LEX et de YACC.

L'analyseur de langage formel mis en oeuvre dans ce projet est en fait la première moitié d'un compilateur complet. Cette partie dépend du langage source et s'occupe de l'analyse lexicale, syntaxique, sémantique, de la création de la table des symboles et de la génération du code intermédiaire et notons qu'une certaine dose d'optimisation peut être réalisée à ce niveau [1]. Cette phase s'occupe aussi de la manipulation des erreurs. Pour faciliter la compréhension et alléger le texte, le terme compilateur sera employé dans le reste de l'ouvrage pour désigner la première moitié d'un compilateur. L'analyseur de langage formel tient compte des hypothèses suivantes:

- Les fichiers sources écrits en FORTRAN ont déjà été compilés et sont prêts à être utilisés. La phase de manipulation des erreurs peut donc être omise.

- Puisque cet analyseur n'ajoute que du code FORTRAN aux fichiers sources déjà existants, la phase de génération du code intermédiaire peut être omise, car un compilateur classique de FORTRAN est utilisé pour générer la version exécutable.

- Du fait de l'hypothèse précédente, la phase d'optimisation peut aussi être omise.

2.1.1 L'outil de construction d'analyseur lexical — LEX

LEX a comme vocation (dans le cadre de ce projet) de réaliser l'analyse lexicale de fichiers sources écrits en FORTRAN. Il s'agit donc de lire les caractères de ces fichiers, en entrée. LEX divise ces caractères lus, en groupes nommés jetons. Les caractères constituant le jeton se nomment lexèmes. De même, on nomme patron un ensemble de chaînes de caractères reconnues par le même jeton. Ainsi, jeton est le nom donné au patron. LEX envoie ensuite chaque jeton vers l'analyseur syntaxique (dans notre cas YACC) dans le but de reconnaître une règle. Le contexte d'utilisation de LEX est illustré à la figure 5.

Le langage LEX est un langage dit compilé. Lors de la compilation, LEX transforme les règles écrites dans son langage contenu dans un fichier *lex.l* en un automate fini déterministe exprimé en langage C. Le fichier qui en résulte doit être compilé à l'aide d'un compilateur C classique (voir figure 5).

Les avantages d'utiliser un compilateur d'analyseur lexical (LEX) pour faire la reconnaissance syntaxique de langage formel sont les suivants:

- Réduit de beaucoup la complexité de la phase d'analyse grammaticale.

- Permet la conception d'outil plus structuré, plus rapide à modifier et ayant une modularité accrue.

- Portabilité améliorée, puisque l'analyseur syntaxique est indépendant de l'alphabet reconnu par l'analyseur lexical. Ce qui implique qu'un seul analyseur syntaxique peut avoir plusieurs analyseurs lexicaux.

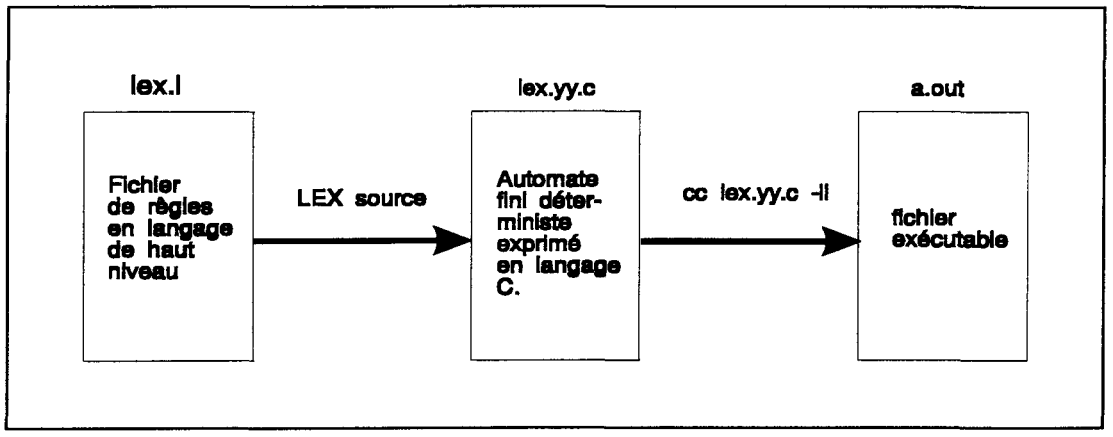

Figure 5 Contexte d'utilisation de LEX.

LEX est un automate fini déterministe qui construit une table des transitions lors de la compilation. Cette table des transitions est créée à partir des règles écrites en langage LEX. Elle peut s'exprimer sous forme schématique comme le montre la figure 6. Cette représentation est appelée graphe de transition.

Il existe aussi des automates finis non-déterministes. La différence fondamentale entre les automates non-déterministes et déterministes est que le dernier ne peut avoir qu'une seule transition d'un état vers un autre pour le même symbole grammatical. Du côté performance l'automate fini déterministe est plus rapide, mais génère, en contre partie, une table des transitions beaucoup plus volumineuse que l'automate fini non-déterministe.

Pour identifier les lexèmes, LEX utilise deux pointeurs dans un tampon d'entrée comme l'illustre la figure 7. De gauche à droite, le premier est le pointeur de début du lexème et le second est le pointeur sentinelle ("lookahead"). Le lexème étant défini comme

étant l'entité de base reconnue par les règles écrites dans le fichier LEX et envoyée à YACC.

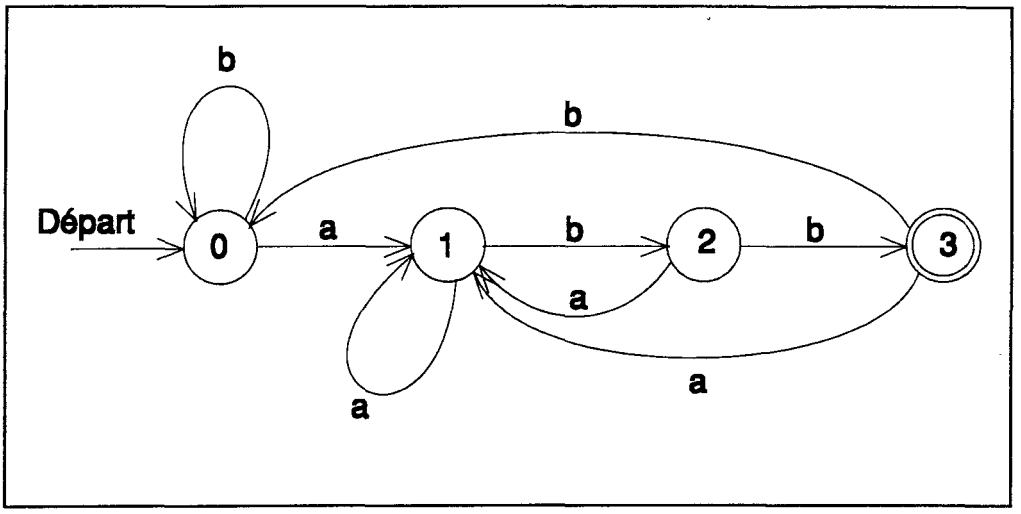

Figure 6 Exemple de graphe de transition pour un automate fini déterministe.

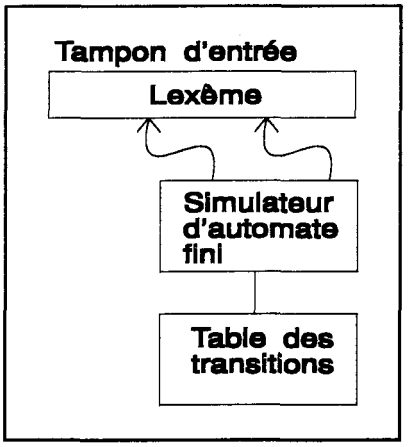

Figure 7 Diagramme en bloc de l'analyseur lexical.

2.1.2 L'outil de construction d'analyse syntaxique — YACC

YACC doit son nom au fait qu'à l'époque de son élaboration, plusieurs outils de ce genre ont vu le jour, c'est pourquoi ses créateurs l'ont appelé "encore un autre compilateur de compilateurs" (YET ANOTHER COMPILER-COMPILER).

YACC recueille les jetons envoyés par LEX et exécute l'analyse grammaticale de ceux-ci dans le but de reconnaître des séquences qui rendent une production effective.

YACC est un "LALR(1) Parser", c'est-à-dire un analyseur grammatical de type sentinelle, lecture de gauche à droite, réduction la plus à droite avec un jeton de sentinelle("LookAhead, Left-to-right scanning, Rightmost derivation with one lookahead token"). Ce genre d'analyseur peut être construit pour des langages pouvant être exprimés par une grammaire sans-contexte (Backus-Naur). De plus, cette méthode est la plus utilisée de celles appelées "non-backtracking shift/reduce methods", et peut être implantée aussi efficacement que n'importe quel autre méthode dite "shift/reduce". L'analyseur de type LR permet de détecter une erreur de syntaxe aussi rapidement que cela puisse l'être en scrutant de gauche à droite.

Comme pour LEX, YACC possède aussi son langage de programmation de haut niveau. Ce langage porte le nom de grammaire sans contexte et est constitué de quatre éléments essentiels.

1- Un ensemble de jetons connus sous le nom de symboles terminaux.

2- Un ensemble de symboles non-terminaux.

3- Un ensemble de règles de production où chacune est constitué d'un symbole non-terminal appelé côté gauche de la production et d'une séquence de symboles terminaux et/ou de symboles non-terminaux appelée côté droit de la production.

4- La désignation d'un symbole non-terminal comme symbole de départ.

Le langage YACC est lui aussi un langage compilé. Lors de la phase de compilation, YACC transforme les règles écrites dans son langage contenu dans un fichier *yacc.y* en un automate fini déterministe exprimé en langage C (voir figure 8).

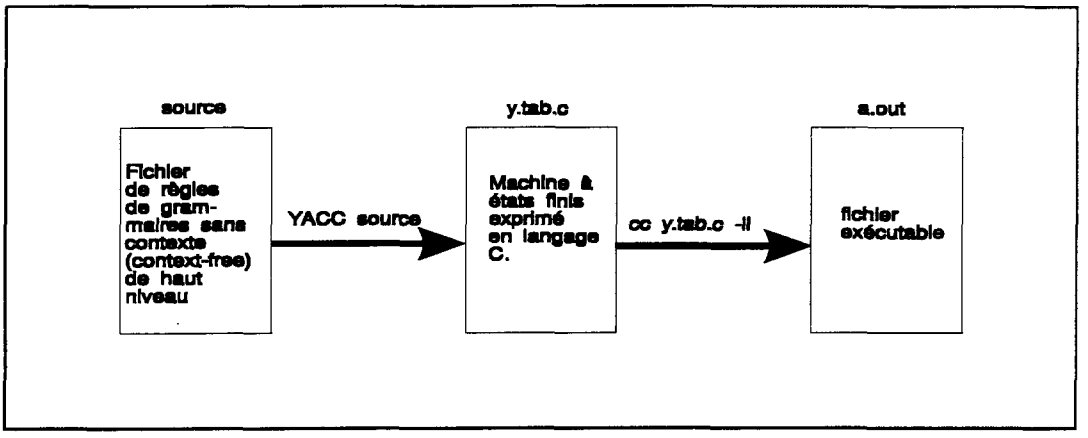

Figure 8 Contexte d'utilisation de YACC.

Malgré l'emploi d'outils spécialisés, l'analyse du langage formel de programmation FORTRAN est rendu difficile en raison de certaines caractéristiques de sa définition. Notons, entre autres, l'absence de caractère de fin de ligne, les six premières colonnes réservées et les parenthèses servant à la fois pour les matrices, vecteurs et fonctions.

2.2 Fonctionnement de l'insertion automatique du moniteur

OSASMO, d'un point de vue global, analyse le contexte des fichiers sources écrits en FORTRAN et prend des actions en fonction de ce contexte. YACC possède les règles qui conduisent au choix des actions. Ces règles sont directement dépendantes du contexte. Ce sont les actions qui réalisent l'insertion de lignes de code FORTRAN. Par conséquent, ce sont ces lignes qui constituent le moniteur qui recueille les données statistiques de parallélisme et de communication.

Les règles de YACC sont en fait des fragments de programmes FORTRAN hiérarchisés selon un ordre prédéterminé et écrits en grammaire sans contexte. Ces fragments composent un graphe de dépendance à chaque ligne du programme source analysé. Il faut donc donner un ordre de préséance aux symboles terminaux et non-terminaux. Il est évident que la multiplication doit être plus prioritaire que l'addition dans l'arbre de flôt de contrôle de la décomposition de chaque ligne en code à trois adresses, ce qui a pour effet de réduire la ligne d'analyse de l'application source dans l'ordre de préséance préalablement défini. Par exemple, voici l'ensemble suivant de règles écrites en grammaire sans contexte de YACC:

expr	:	expr '+' expr	(4)
	|	expr '-' expr	(4)
	|	expr '*' expr	(3)
	|	expr '/' expr	(3)
	|	'(' expr ')'	(2)
	|	**CHIFFRE**	(1)
	|	**VARIABLE**	(1)

où **CHIFFRE**, **VARIABLE**, '(', ')', '+', '-', '*', et '/' sont envoyés par LEX durant l'analyse. Les chiffres entre parenthèses illustrent le niveau de priorité dans lequel la reconnaissance des règles doit être faite. Le (1) est le plus prioritaire. Cette expression constitue un ensemble de règles nécessaires et suffisantes à la réduction de ligne de code comme:

(1+var1)*(3*var2)/var3

Dans l'exemple précédent, aucune action n'est prise en fonction de la règle reconnue. Voici un exemple qui en tient compte. Soit l'ensemble suivant de règles écrites en

grammaire sans contexte et ayant pour chacune l'action correspondante écrite en pseudo-code:

```
expr :    expr '+' expr      { Écrire ligne(s) de moniteur
                               correspondante(s) à l'addition }
      |   expr '-' expr      { Écrire ligne(s) de moniteur
                                   correspondante(s) à la soustraction }
      |   expr '*' expr      { Écrire ligne(s) de moniteur
                                   correspondante(s) à la multiplication }
      |   expr '/' expr      { Écrire ligne(s) de moniteur
                                   correspondante(s) à la division }
      |   '(' expr ')'       { Écrire ligne(s) de moniteur
                                   correspondante(s) aux parenthèses }
      |   CHIFFRE            { Écrire ligne(s) de moniteur
                                   correspondante(s) aux chiffres }
      |   VARIABLE           { Écrire ligne(s) de moniteur
                                   correspondante(s) aux variables }
```

où **CHIFFRE**, **VARIABLE**, '(', ')', '+', '-', '*', et '/' sont envoyés par LEX durant l'analyse. Lorsqu'une règle est reconnue, l'action entre accolade est exécutée. Dans notre cas, les actions insèrent des lignes de code FORTRAN en format ASCII dans le fichier source analysé.

Un fichier YACC est donc constitué, entre autre, de deux parties (gauche et droite, voir figure 9). À cela s'ajoute la zone des définitions.

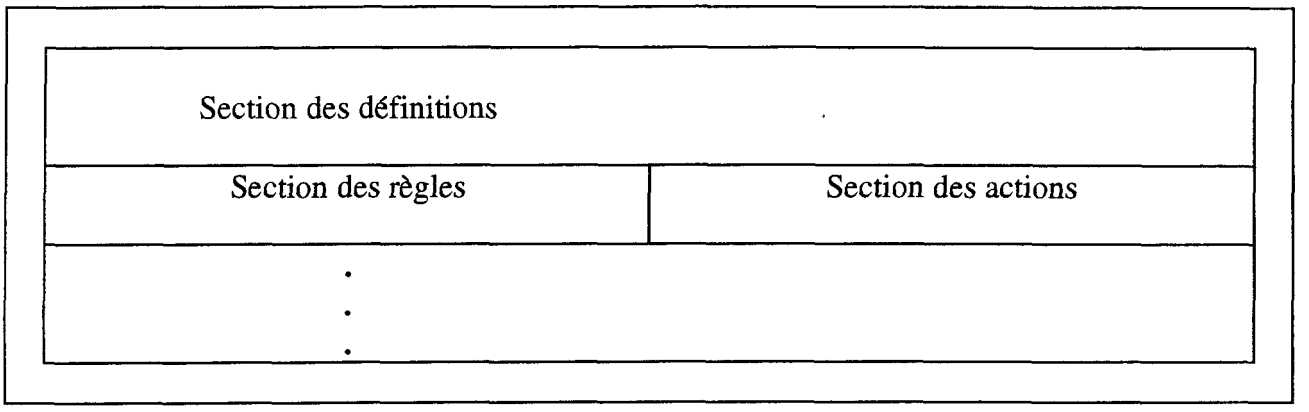

Figure 9 Description d'un fichier YACC.

2.3 Fonctionnement du moniteur

OSASMO est un outil très complexe, qui a la possibilité de simuler, de façon expérimentale, plusieurs architectures parallèles. Cet état de fait rend cet outil hautement versatile puisqu'avec des modifications mineures, il est possible de comparer diverses architectures parallèles ou encore de faire l'optimisation de certains types d'instructions qui peuvent provoquer des engorgements. Ces performances sont rendues possibles grâce à l'implantation d'un moniteur dans l'application scientifique à étudier.

2.3.1 Principes de base

Le moniteur est constitué de plusieurs éléments qui seront décrits dans cette section. Certaines de ces composantes sont communes à toutes les instructions typiques tandis que d'autres se rapportent soit à un type particulier d'instruction ou encore à une architecture donnée.

Le but du moniteur est de tracer le comportement parallèle, avec ou sans contraintes, d'un programme écrit en FORTRAN pour une architecture séquentielle. Cette étape est

réalisée durant l'exécution même du programme source étudié. Il est évident que le moniteur doit être implanté à même l'application source (en FORTRAN) avant le début de l'exécution.

L'exécution parallèle d'une application scientifique écrite en FORTRAN pour une utilisation séquentielle, est illustrée par l'exemple de la figure 10. Un moniteur peut donc recueillir dynamiquement les paramètres de concurrence et de communication durant l'exécution de cette application. Une des principales statistiques de concurrence qui peut être extraite de cette exécution est le nombre moyen d'instructions exécutées simultanément. De plus, il est possible de caractériser l'utilisation du réseau, comme nous le verrons plus loin.

Soit le fragment de programme suivant:

```
...
A = 1
B = 3
C = 3
D = 4
E = A * SIN(B)
F = C * D
G = F + E
...
```

Voici le fragment de programme exécuté sur une machine parallèle idéale:

(1e ligne //)	A = 1; B = 2; C = 3; D = 4; ...
(2e ligne //)	E = A * SIN(B); F = C * D
(3e ligne //)	G = F + E; ...

Et enfin le même fragment de programme sur une machine parallèle n'ayant que deux modules processeur-mémoire:

(1e ligne //)	A = 1; B = 2
(2e ligne //)	C = 3; D = 4
(3e ligne //)	E = A * SIN(B); F = C * D
(4e ligne //)	G = F + E

Figure 10 Exemple de parallélisation d'une application FORTRAN écrite pour une exécution séquentielle.

2.3.2 Les variables synthétiques

Pour extraire le nombre moyen d'instructions pouvant être exécutées en parallèle, le moniteur utilise des entités donnant une information temporelle sur le déroulement du programme. Ces entités sont les homologues de toutes les variables se trouvant dans l'application source étudiée et se nomment *variables synthétiques.* Les variables synthétiques peuvent avoir comme valeur, par exemple, le temps minimum auquel une variable (leur homologue) est disponible pour être utilisée par un processeur ou par une opération de communication. Le temps de disponibilité est exprimé en unité de temps sur une échelle relative ayant des divisions égales. Dans le reste du texte, lorsque la forme "unité de temps" est utilisée, ceci implique qu'il s'agit d'unités relatives de temps. L'exemple de la figure 11, expose sommairement comment les variables synthétiques sont utilisées pour recueillir des informations pertinentes. Ces variables synthétiques permettent de suivre l'évolution de l'exécution du programme et de conclure sur le niveau de concurrence contenu dans l'application étudiée. En se référant au même exemple, on conclut que les lignes (1),(2), et (3) peuvent être exécutées au même instant pour une machine idéale puisque les variables synthétiques ont toutes les trois la même valeur. Pour chacune des instructions suivantes, la variable dont la variable synthétique correspondante est la plus élevée indiquera le moment où ces instructions pourront être exécutées. À l'aide de la fonction MAX, qui calcule la valeur maximale parmi les variables synthétiques en

argument, on concluera donc que la ligne (4) ne peut être exécutée avant un temps égal à deux et la ligne (5) ne peut être exécutée, pour les mêmes raisons, avant le temps relatif égal à trois. La valeur un qui est ajoutée pour le calcul de $D et de $E représente le temps d'exécution de la ligne qui est considéré ici comme étant égal à une unité supplémentaire de temps. Il faut donc ajouter une unité de temps à chaque exécution d'une instruction. On remarquera que l'on pose ici l'hypothèse que toutes les instructions requièrent le même temps d'exécution, ce qui n'est évidemment qu'une approximation de la réalité. Cependant, de par la manière dont on calcule les nouvelles valeurs des variables synthétiques, il est très facile de raffiner les analyses pour étudier l'influence de ce facteur. Dans le présent travail, cet aspect n'a pas été considéré.

Soit le fragment de programme fortran suivant:

```
                ...
(1)             A = 1
(2)             B = 2
(3)             C = 3
(4)             D = A + B
(5)             E = D * A
                ...
```

Voici le même fragment avec les variables synthétiques ($):

```
$A = 1
A = 1
$B = 1
B = 2
$C = 1
C = 3
```

$D = 1 + MAX($A, $B)	($D = 2)
D = A + B	
$E = 1 + MAX($A, $D)	($E = 3)
E = D * A	
l'exécution d'une ligne = 1 cycle de temps relatif (1)	

Figure 11 Exemple d'utilisation des variables synthétiques.

2.3.3 Coût des communications

Il est possible d'élargir le principe des variables synthétiques pour être en mesure d'évaluer les coûts de communication inhérents aux différents types de réseaux de communication des architectures parallèles. En créant un autre type de variable synthétique associée à chacune des variables se trouvant dans l'application source étudiée, il est possible d'assigner à chacune un numéro de processeur. Ce numéro correspondra à la *localité* de la variable homologue, définie comme étant le numéro de processeur dans lequel la variable homologue occupe un espace mémoire dans lequel se trouve sa valeur. On retrouve la même chose pour les vecteurs et matrices. Donc, avec les deux types précédents de variables synthétiques, il est possible de déterminer certains paramètres spatio-temporels pour chaque variable de l'application étudiée. L'exemple de la figure 12, met en évidence l'utilisation des deux types de variables synthétiques. Le calcul des temps de disponibilité des variables s'effectue à l'aide, cette fois, d'une fonction MAX_COM qui tient compte de la localité de chaque variable et du numéro de processeur pour lequel la ligne est exécutée sans oublier les temps de disponibilité de chacune de ces variables. Cette fonction donne les coûts de communication qui forme en fait une pénalité. Cette pénalité est additionnée à l'intérieur même de la fonction MAX_COM. La valeur qu'elle retourne est donc calculée en fonction des coûts de communication engendrés par l'exécution de la ligne étudiée. Le coût d'une opération de communication dépend de certains paramètres et est exprimée en

fonction des unités de temps. Dans l'exemple de la figure 12, on a réparti la localité initiale des variables de façon aléatoire. Les coûts de communication pour le calcul de D et de E sont respectivement exprimés par com_D et com_E. Les deux paramètres com_D et com_E varient en fonction de l'architecture simulée et du coût d'une opération de communication. Les lignes (1), (2), et (3) étant des assignations de valeur, aucune pénalité dûe à la communication n'est engendrée.

En remplaçant la variable de localité ($P) par un vecteur de localité, vecteur ayant la dimension du nombre de processeurs de l'architecture simulée, il est possible de connaître les processeurs ayant utilisé la variable homologue dans leurs calculs. Si une nouvelle valeur est assignée à cette variable, les copies de cette dernière dans les autres processeurs doivent être purgées. Ainsi ce vecteur doit être réinitialisé à chaque fois qu'une variable homologue se trouve à la gauche d'une égalité.

Soit le fragment de programme FORTRAN suivant:

```
                ...
(1)             A = 1
(2)             B = 2
(3)             C = 3
(4)             D = A + B
(5)             E = D * A
                ...
```

Voici le même fragment avec les deux types de variables synthétiques ($):

```
$PA = ALEA()        (allocation initiale de la localité...)
$A = 1              (...est aléatoire.)
A = 1
$PB = ALEA()
$B = 1
B = 2
```

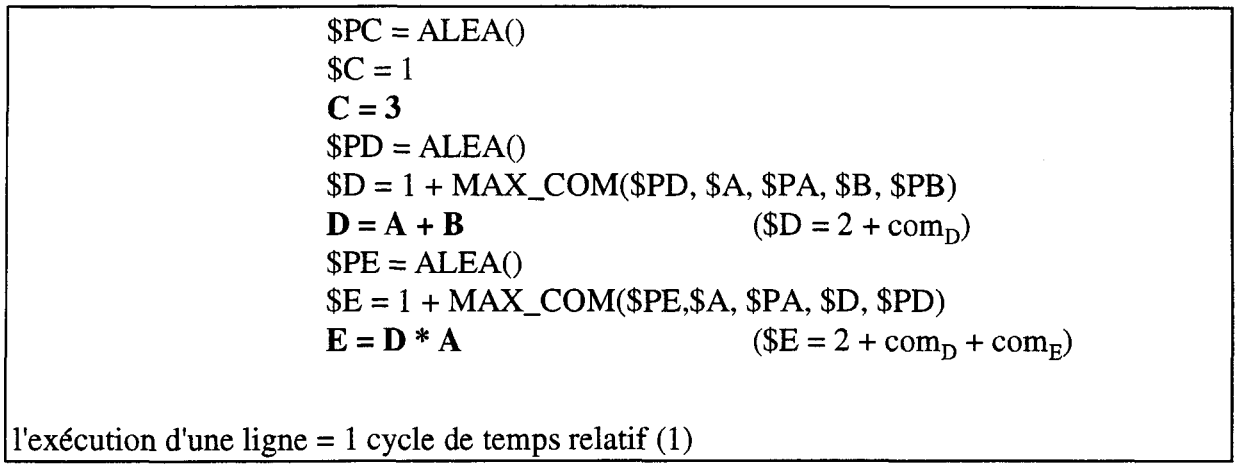

Figure 12 Exemple d'utilisation des deux types de variables synthétiques.

2.3.4 Compilation des statistiques

Les notions présentées jusqu'à présent permettent de situer plus précisément l'utilisation d'un moniteur dans l'élaboration d'un outil de simulation architecturale. La présente sous-section sera consacrée à la description détaillée des différentes parties du moniteur implantées dans les applications sources par OSASMO. Une certaine uniformité dans la présentation de ces attributs doit être respectée. La nomenclature utilisée est celle utilisée dans la littérature contemporaine traitant du sujet. Il s'agit d'un système de règles dérivé de la forme Backus-Naur. Ce système fût utilisé pour la première fois en 1960 pour décrire ALGOL 60 (tiré de Kelley et Pohl [11]). Les symboles utilisés sont les suivants:

Symboles	Définitions
Italique	Indique une catégorie syntaxique
::=	Peut-être réécrit comme/avec ce/ces symboles
|	Implique l'alternative entre les choix
$\{\}_1$	Choisir un des items entre les accolades
$\{\}_{0+}$	Répéter zéro ou plusieurs items entre les accolades
$\{\}_{1+}$	Répéter un ou plusieurs items entre les accolades
$\{\}_{opt}$	Item optionnel

Voici quelques définitions qui seront utilisées dans la description approfondie du moniteur.

lettre ::= a | b | c | ... | z | A | B | C | ... | Z
chiffre ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
alphanum ::= lettre{*lettre* | *chiffre*}$_{0+}$
K_Wtype ::= K_W*alphanum*
K_Rtype ::= K_R*alphanum*
K_Ptype ::= K_P*alphanum*
K_Ltype ::= K_L*alphanum*({*chiffre*}$_{1+}$)
TMPtype ::= TMP{*chiffre*}$_{1+}$
IFtype ::= KEC_IF{T| }$_1${*chiffre*}$_{1+}$
DOWtype ::= KEC_DOW{*chiffre*}$_{1+}$
nbproc ::= {*chiffre*}$_{1+}$ (nombre de processeur(s) du système)
profondeur ::= {*chiffre*}$_{1+}$ (durée de la simulation en cycles relatifs)

Le moniteur complet possède, pour sa gestion, une panoplie de variables et de fonctions, qui sont toutes détaillées dans la suite de cette section. Ces caractéristiques seront présentées selon leur influence. Trois (3) types d'influence sont répertoriés. On a d'abord les entités se rapportant à l'histogramme, ou en d'autre mots, à la sortie des résultats. Ensuite, on retrouve celles définissant la structure de l'architecture et, pour finir, celles se rapportant au calcul des temps de disponibilité et aux pénalités dues à la communication.

Le moniteur recueille des données qui sont emmagasinées sous forme d'histogramme. L'axe des abscisses représente le temps d'exécution de l'application scientifique en unité de temps relative. L'axe des ordonnées donne, premièrement, le degré de parallélisme par unité temporelle relative ou, en d'autres termes, le nombre de lignes de code séquentiel qui peuvent être exécutées par unité de temps relative et, deuxièmenent, le nombre moyen d'opérations de communication qui sont réalisées pour chaque unité de

temps. Le nombre de lignes séquentielles qui peuvent être exécutées par unité de temps relative est la mesure du gain de vitesse. La figure 13 illustre, sous forme graphique, l'histogramme tel que compilé par OSASMO après l'exécution d'une application scientifique incluant le moniteur.

Pour compiler l'histogramme, le moniteur doit avoir la possibilité de garder en mémoire les statistiques accumulées lors du déroulement du programme. Pour ce faire, une matrice garde en mémoire les statistiques de concurrence et de communication. Elle est définie par:

integer K_STAT(3, *profondeur*)

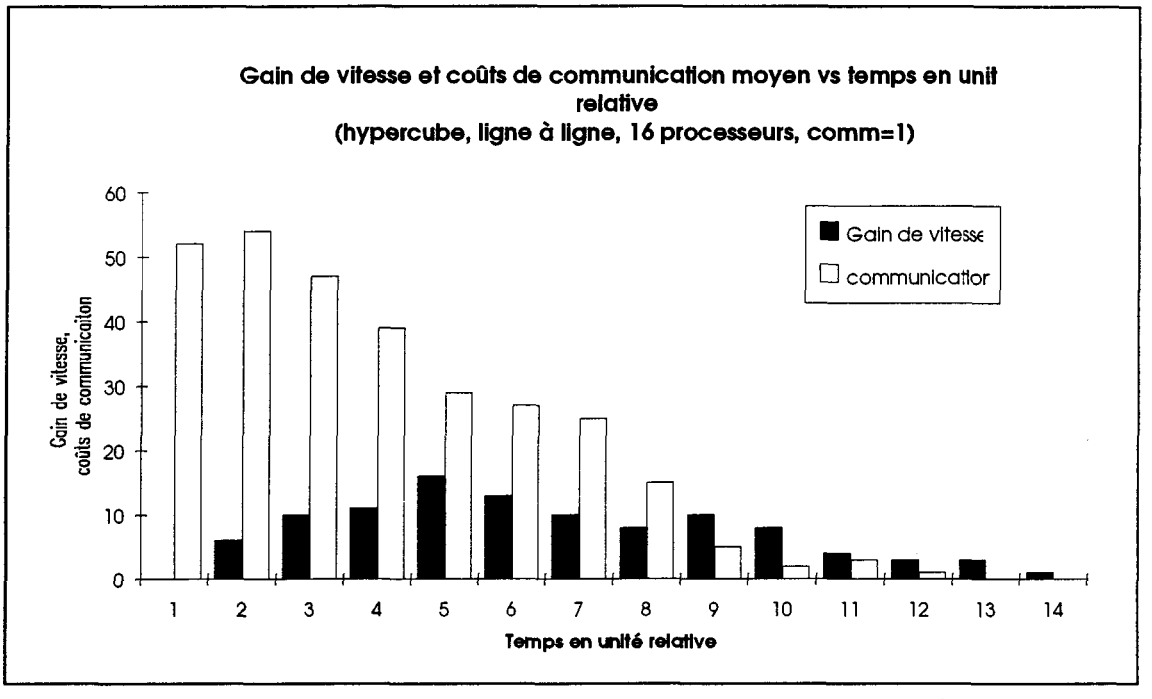

Figure 13 Exemple d'histogramme révélant les résultats recueillis par le moniteur.

Cette matrice conserve, dans la première ligne, les valeurs de temps, dans la deuxième, le nombre d'opérations simultanées pour la valeurs de temps de la première colonne et, dans la troisième, le nombre d'opérations de communication pour chaque valeur de temps relatif.

Avant d'insérer les valeurs à l'intérieur de cette matrice, une fonction vérifie si une autre instruction est exécutée au même instant. En d'autres termes, elle vérifie si, à la valeur calculée du temps de l'exécution de l'instruction dans un processeur donné, il n'y a pas déjà une donnée existante indiquant que le processeur est déjà employé à une autre tâche. Si le processeur est occupé, on incrémente le temps de disponibilité de un (1) et on recommence le processus de vérification. La fonction réalisant cette tâche est définie par:

function integer KEC_SEARCH()

Les fonctions qui insèrent les valeurs de parallélisme (nombre d'opérations simultanées) et de communication dans la matrice de l'histogramme K_STAT sont définies comme suit:

subroutine IN_PARA()

subroutine IN_COM()

Chacune servant respectivement à incrémenter l'histogramme du parallélisme contenu dans la colonne 2 de la matrice K_STAT et à incrémenter l'histogramme de la communication contenu dans la colonne 3 de la même matrice.

La matrice de l'histogramme tire ses valeurs, tout au long de l'exécution du programme augmenté, des variables synthétiques décrites au début de la section, et d'une matrice qui donne toutes les opérations de communication de la ligne étudiée. Chaque

variable du programme source possède trois (3) variables synthétiques dans le programme augmenté. Ces dernières ont la forme suivante:

K_Wtype : exprime le dernier temps où la variable homologue a été écrite.

K_Rtype : exprime le dernier temps où la variable homologue a été lue.

K_Ltype : exprime chaque endroit (numéro de processeur) où la variable a été lue et est disponible (depuis la dernière fois où la variable a été écrite).

Le type *K_Rtype* est rendu nécessaire pour simuler fidèlement le comportement de la disponibilité des variables. En effet, une variable ne peut être écrite, en général, avant que les instructions précédentes utilisant cette dernière n'aient terminé leur exécution. Cette caractéristique se nomme l'effet de la synchronisation des écritures après lecture/écriture. L'exemple de la figure 14 illustre les répercussions qu'engendrerait le fait d'ignorer la dépendance aux écritures après lecture/écriture. En ne tenant pas compte des écritures après lecture/écriture, le temps de disponibilité de A serait égal à 2. Lorsque l'effet des écritures après lecture/écriture est incorporé dans le calcul de la disponibilité de A, le temps de disponibilité de A est de 12, ce qui est plus fidèle à la réalité. C'est pourquoi deux variables synthétiques de temps sont utilisées plutôt qu'une seule, comme introduit en début de section. En effet, une seule variable synthétique de temps ne tient pas compte de l'effet ci-haut mentionné.

(La technique de monitoring expliquée en début de section est ici sous-entendue)

Soit le fragment de programme FORTRAN suivant:

```
...
A = 5                    (posons t = 1)
```

```
B = Y          (posons t = 10)
C = A + B      (t = 11)
A = -A         (t = 12) ou (t = 2)
...
```

Figure 14 Répercussion qu'engendrerait le fait d'ignorer l'effet des écritures après lecture/écriture.

Le type de variable synthétique *K_Ltype* est utilisé de la façon suivante: si, par exemple, un système que l'on veut tester comporte seize processeurs, la dimension des vecteurs de localité sera définie: K_L*alphanum*(16). Chaque valeur d'indice de ce vecteur (de localité) correspond à un numéro de processeur. On considère ainsi qu'une variable puisse être stockée dans plusieurs processeurs. Ainsi, dans le vecteur K_LA de dimension 16, l'élément 6 de ce vecteur est associé à la case mémoire du processeur 6 occupée par la variable A.

Le calcul du coût des opérations de communication s'effectue à l'aide d'une matrice qui garde en mémoire toutes les opérations de communication requises pour une ligne d'exécution du code source. C'est le vecteur KCOM qui contient les résultats de toutes les opérations de communications associés à une ligne de code de l'application scientifique analysée. Il est defini comme suit:

```
integer KCOM(2, 31)
```

Chaque élément de cette matrice contient les données requises pour déterminer le temps requis pour amener une variable dans le processeur où s'effectue les calculs. Ces temps sont obtenus à l'aide de fonctions qui sont définies dans le paragraphe suivant. La première ligne donne le début de la communication en unité temporelle et la seconde, la durée de la communication (aussi en unité de temps). La taille de la deuxième dimension de la matrice KCOM a été fixé à trente et un (31) puisque le nombre maximal d'arguments

pouvant être envoyés aux fonctions de calculs du temps pour une ligne de code est de trente (30).

Les variables synthétiques ainsi que le vecteur KCOM sont rafraîchis, à chaque ligne de code source exécutée, et ce, à l'aide de fonctions et de sous-routines pré-définies. Elles sont au nombre de quatre.

1- fonction max: Cette fonction rafraîchit la variable ou le vecteur synthétique de type *K_Wtype* se trouvant à gauche de l'égalité (pour les assignations).

2- fonction maxfct: Cette fonction rafraîchit la variable ou le vecteur synthétique pour les fonctions définies par l'usager ainsi que les fonctions intrinsèques du FORTRAN. Cette fonction est aussi utilisée pour les lignes n'ayant pas d'égalité ni de variable à gauche de l'égalité (ex.: IF, DOWHILE).

3- fonction maxtmp1: Cette fonction rafraîchit la variable synthétique de type *K_Wtype* où *type* a la forme *TMPtype* se trouvant à gauche de l'égalité dans la situation où l'on a qu'une seule variable à droite de l'égalité pour une ligne de code à trois adresses (Ex.: TMP1 = A + 2).

4- fonction maxtmp2: Cette fonction rafraîchit la variable synthétique de type *K_Wtype* où *type* a la forme *TMPtype,* se trouvant à gauche de l'égalité dans la situation où l'on a deux variables à

droite de l'égalité pour une ligne de code à trois adresses (Ex.: TMP1 = A + B).

La décomposition en ligne de code à trois adresses implique que des lignes additionnelles sont ajoutées. Puisque chaque ligne de code arithmétique doit avoir une égalité avec une variable à sa gauche, de nouveaux noms de variables doivent être créés. Ces variables sont des variables temporaires qui n'ajoutent pas d'informations au code source analysé, aussi elles sont définies selon le format *TMPtype*. Un exemple de leur utilisation est montré à la figure 15. Le moniteur utilise aussi les types *K_Wtype*, *K_Rtype*, *K_Ltype* pour ce type de variable, au même titre que les variables d'origine de l'application étudiée.

Soit le fragment de programme FORTRAN suivant:

A = B + C - D

devient en code à trois adresses (une des deux possibilités):

TMP1 = B + C

A = TMP1 - D

Figure 15 Exemple d'utilisation des variables de type *TMPtype*.

2.3.5 Considération des contraintes physiques d'un système

Le moniteur doit possèder des entités donnant des informations sur la structure physique simulée. On réfère ici au nombre de processeurs de l'architecture, à l'utilisation des processeurs (en tout temps), au numéro de processeur dans lequel la ligne de code est exécutée, au coût de base d'une opération de communication et à la durée relative de la

simulation. Ces informations sont disponibles tout au long de la simulation et sont gardées à l'aide de variables et de matrices.

Chaque processeur, en général, ne peut exécuter qu'une seule instruction à un instant donné. Il est donc essentiel de connaître son occupation en tout temps. OSASMO conserve cette information tout au long de l'exécution du programme en stockant cette dernière dans une matrice définie par:

integer KTPROC(*nb_processeur, profondeur*)

où *nb_processeur* représente le nombre de processeurs et *profondeur* la durée (nombre de cycle relatif) de l'exécution du programme. Grâce à cette matrice, on évite que plusieurs lignes de code soient exécutées au même moment.

OSASMO considère que chaque ligne ou ensemble de lignes appartenant à un même bloc de base peut potentiellement être exécutée dans un processeur différent. La variable KPROC contient le numéro de processeur dans lequel la ligne de code source est exécutée. Cette variable est rafraîchie à chaque ligne ou bloc de base par la fonction Kec_proc(). La fonction renvoie le numéro de processeur dans lequel la ligne de code sera exécutée.

La simulation de l'architecture pour un programme donné doit avoir une durée finie. OSASMO doit connaître cette valeur qui est exprimée en unité de temps relatif pour donner les bonnes dimensions aux différentes entités de gestions (matrices). La variable PROF contient cette information.

L'architecture du système simulé peut varier d'une simulation à une autre. Il est donc impératif de connaître les informations de base de l'architecture testée qui sont, pour cette recherche, le nombre de processeurs ainsi que le coût de base d'une opération de

communication. Chacune des deux informations est donnée en unité de temps relatif. La variable NB_PROC contient le nombre de processeurs de l'architecture testée et la variable MULT_COM contient le coût d'une opération de communication de base.

2.3.6 Considérations des contraintes logicielles

La *nécessité* d'avoir les temps de disponibilité de chaque variable de l'application source pose des problèmes particuliers lors de la rencontre de blocs d'influence délimités par des lignes de contrôles. Les lignes de contrôles sont celles qui provoquent les branchements conditionnels (ex.: IF). Un *bloc d'influence* est défini comme un ensemble de lignes qui sont exécutées si et seulement si un certain branchement est préféré à un autre, comme pour les instructions de type d'instruction IF-THEN-ELSE. Le *bloc d'influence* doit être monolithique. Il ne peut être scindé en plusieurs entités à travers le programme. Les lignes de code constituant un certain bloc d'influence ne peuvent pas être exécutées avant que la ligne de contrôle ne le soit elle-même. Ceci implique qu'il faut connaître, pour toutes les lignes de code du bloc d'influence, le temps minimum auquel on peut conclure que la ligne de contrôle du bloc est exécutée. KTIME$ est la variable qui donne le temps minimum auquel on peut conclure qu'une ligne de code du bloc d'influence peut-être exécutée. Le cas de la figure 16 démontre clairement que les blocs 1 et 2 ne peuvent être exécutés avant que la décision à la ligne "contrôle" ne soit prise. En prenant soin de garder ce temps dans KTIME$, le moniteur peut déterminer le moment où l'exécution d'un des deux blocs d'instructions peut démarrer. Les blocs d'influence imbriqués sont décrits à la section traitant de l'analyse détaillée du moniteur (section 2.4). Notons que les arguments ainsi que les retours de toutes les fonctions et sous-routines de gestion sont entièrement définis à l'annexe 1.

```
KTIME$ = temps minimum d'exécution de (contrôle)
IF(A.GT.1)THEN      (contrôle)
      A = A + 1     (bloc 1)
ELSE
      A = A - 1     (bloc 2)
ENDIF
```

Figure 16 Exemple d'utilisation de la variable KTIME$.

La figure 17 montre la transformation que subit une ligne de code après insertion du moniteur par OSASMO. Sur cette figure, la variable synthétique ayant été rafraîchie est K_WA, puisque son homologue, la variable A, a reçu une nouvelle valeur dans la cellule de mémoire du processeur dont le numéro est KPROC. Les sous-routines IN_PARA et IN_COM sont utilisées, après le rafraîchissement de la variable synthétique K_WA et de la matrice des coûts de communication KCOM, pour recueillir les données de la ligne dans l'histogramme des résultats.

Soit le fragment de programme

```
...
A = B + C
...
```

Voici le programme augmenté tel que généré par OSASMO:

```
*       Localité de l'exécution de la ligne
        KPROC = Kec_proc()
*       Réinitialisation du vecteur de localité de la variable à
*       gauche de l'égalité (A)
        DO I = 1, NB_PROC
              K_LA(I) = 0
        ENDDO
*       A sera présente dans le processeur KPROC
        K_LA(KPROC) = 1
```

```
*       Calcul du temps de disponibilité après l'exécution de la ligne
        K_WA = 1 + max(KTIME$, KCOM, K_WA,
     1  K_RA, KPROC, K_WB, K_LB, K_WC, K_LC)
*       Rafraîchissement des variables synthétiques de lecture
        K_RB = 1 + MAX0(K_RB, K_WA)
        K_RC = 1 + MAX0(K_RC, K_WA)
*       Insertion des valeurs de parallélisme et de communication
*       dans la matrice d'histogramme K_STAT
        CALL IN_PARA(K_STAT, K_WA)
        CALL IN_COM(K_STAT, KCOM)
*       Ligne analysée finalement exécutée
        A = B + C
```

Figure 17 Exemple d'insertion du moniteur.

2.4 Insertion du moniteur selon les classes d'instructions

Comme on a pu le voir à la section précédente, l'insertion du moniteur dans l'application scientifique se fait en fonction des lignes de codes analysées dans le fichier source. Il est donc impératif d'identifier des types d'instructions typiques afin de caractériser un peu plus l'algorithme d'analyse d'OSASMO. Quatre (4) grands types d'instructions ont été identifiés:

- les lignes ordinaires en code à trois adresses,
- les boucles,
- les lignes de contrôles
- les fonctions et sous-routines

2.4.1 Pré-traitement des applications scientifiques

Les applications scientifiques ne sont généralement pas écrites en code à trois adresses. Par conséquent, il est nécessaire de les traiter de façon à permettre l'extraction des statistiques désirées. Cette étape, qui pourrait être réalisée dans un pré-traitement indépendant, est exécutée en même temps que l'analyse de langage formel. Ceci élimine une manipulation pour l'obtention du programme final, augmenté du moniteur.

Tel que mentionné précédemment, la réduction en code à trois adresses permet au moniteur de simuler plus fidèlement le comportement d'un processeur. Ceci permet aussi de ne pas sous-évaluer les coûts de communication. Finalement, une analyse plus fine du parallélisme et des coûts de communicaton présents dans une application scientifique est obtenue.

2.4.2 Lignes ordinaires en code à trois adresses

Les lignes de code à trois adresses sont caractérisées par la structure suivante: une destination, une égalité, deux opérandes et un opérateur (figure 18). La décomposition en code à trois adresses est utilisée ici pour augmenter la résolution lors de la quantification du parallélisme et des coûts de communication. Tous les compilateurs, de façon générale, compilent le code source en le transformant en code à trois adresses. En effet, un processeur n'exécute, en général, que des lignes de code à trois adresses pour exécuter des opérations arithmétiques.

destination = opérande opérateur opérande	(a)
ex.: **A = B + C**	(b)

Figure 18 (a) Définition d'une ligne de code à trois adresses. (b) Exemple de ligne de code à trois adresses.

2.4.3 Lignes de contrôle

Les lignes de contrôle sont des lignes de code ayant un IF et un ou des opérateurs relationnels et/ou conditionnels. Trois types différents ont été considérés:

1- IF-THEN-ELSE
2- Logical-IF
3- Autres (GOTO et étiquettes)

Le premier type est structuré pour les mêmes raisons que celles énoncées à la sous-section précédente. Le IF-THEN-ELSE comprend toutes les combinaisons de IF, incluant les ELSEIF. Comme pour la boucle DOWHILE, le IF ne possède pas de variable à gauche de l'égalité, l'égalité étant absente de ce type de ligne. Pour conserver la valeur du temps où

l'on peut conclure que la ligne peut-être exécutée, il faut créer une variable synthétique temporaire de type *IFtype*. C'est cette variable qui, dans tous les blocs d'influence du IF, donne le temps minimum auquel les lignes du bloc d'influence peuvent commencer à être exécutées. L'intérêt d'utiliser des variables synthétiques temporaires devient plus évident lors de l'imbrication des blocs d'influence. L'exemple de la figure 19 met en relief un cas typique. KEC_IFT1 donne le temps minimum auquel on peut conclure que les lignes des deux blocs d'influence du premier IF (1) (voir 1 à la figure 19) peuvent être exécutées, alors que KEC_IFT2 donne la même information, mais pour les blocs d'influence du deuxième IF (2) (voir 2 à la figure 19). À la fin des blocs d'influence du deuxième IF (3) (voir 3 à la figure 19), puisque l'exécution du programme quitte le deuxième bloc d'influence pour le premier, la variable KTIME$ est réinitialisée à la valeur de la variable synthétique temporaire du premier bloc d'influence, KEC_IFT1.

Les lignes de contrôle de type Logical-IF (Ex.: IF(A.GT.B)GOTO 10) sont structurées et réalisent la même fonction que le IF-THEN-ELSE, mais sont exprimées avec des instructions "goto" et des étiquettes. L'implantation des variables synthétiques temporaires est effectuée de la même façon que pour le IF-THEN-ELSE.

Les IF-calculés (Ex.: IF(A)10, 20, 30) sont transformés en IF-THEN-ELSEIF avec des GOTO dans chaque bloc d'influence pour donner la destination.

Soit le fragment de programme suivant:

```
IF(A.GT.B)THEN                    (1)
    IF(C.GT.A)THEN                (2)
        C = C + 1
    ELSE
        C = C - 5
    ENDIF                         (3)
```

```
            A = B + C
        ELSE
            A = B - C
        ENDIF
```

Ajoutons les variables temporaires synthétiques de lignes de contrôles (les autres variables synthétiques ont été omises pour des raisons de clarté):

```
        KEC_IFT1 = 1 + maxfct(KTIME$, KCOM, K_WA,
    1   K_LA, K_WB, K_LB)
        KTIME$ = KEC_IFT1
        IF(A.GT.B)THEN                    (1)
            KEC_IFT2 = 1 + maxfct(KTIME$, KCOM, K_WC,
    1       K_LC, K_WA, K_LA)
            KTIME$ = KEC_IFT2
            IF(C.GT.A)THEN                (2)
                C = C + 1
            ELSE
                C = C - 5
            ENDIF                         (3)
            KTIME$ = KEC_IFT1
            A = B + C
        ELSE
            A = B - C
        ENDIF
        KTIME$ = 1
```

Figure 19 Exemple d'utilisation des variables synthétiques temporaires pour lignes de contrôle.

Le type Autres est constitué de toutes les autres façons que le langage FORTRAN possède pour contrôler l'exécution d'un programme avec des GOTO et des étiquettes. Cette manière de programmer est qualifiée de non-structurée. Ceci implique que les blocs d'influence que ces lignes engendrent peuvent être segmentés en plusieurs parties et dispersés à travers le programme sans logique structurée. Cette partie complique énormément l'insertion du moniteur. En effet, l'absence de logique structurée rend quasi-

irréalisable l'analyse exhaustive des blocs d'influence par l'analyseur de langage formel. Cette particularité est l'objet d'une section au chapitre III de ce travail.

2.4.4 Boucles

On divise les boucles en deux catégories différentes. Premièrement, les boucles DO et deuxièmement les boucles DOWHILE qui sont une version structurée des if-goto. Les boucles DOWHILE ne sont pas des instructions standards dans la version 77 du FORTRAN, mais elles ont quand même été considérées dans la version de l'analyseur de langage formel étant donnée la facilité avec laquelle elles pouvaient être implantées dans le moniteur.

Les deux types de boucle terminent leur bloc d'influence par un ENDDO. Il est à noter, contrairement aux travaux de Kumar [12], que la présente implantation ne considère pas les variables d'index des boucles DO dans le calcul des temps d'exécution des lignes de code. Ces types de variables sont traitées comme des constantes. Ceci permet de capturer le potentiel parallèle présent dans les boucles. Les boucles DO se retrouvant en grand nombre dans les applications scientifiques, il est important d'en extraire le potentiel de concurrence. Ce choix a pour effet "d'horizontaliser" l'exécution de la boucle. En d'autres termes, chaque itération de la boucle peut potentiellement être exécutée dans un processeur différent. "L'horizontalité" de la boucle est directement fonction du type d'allocation des ressources de calcul. Effectivement, l'allocation ligne-à-ligne donne la possibilité à chaque ligne de la boucle d'être exécutée sur un processeur différent. Pour l'allocation par bloc de base, ce sont les blocs entiers qui sont exécutés sur des processeurs différents. Notons que ces deux types d'allocation ont été définis à la section 1.4.

Soit le fragment de programme suivant:

```
      DO I=1,10
            A(I) = B(I) + C(I)
      ENDDO
```

Voici le programme modifié incluant le moniteur (partiel)

```
      DO I=1,10
            KPROC = Kec_proc()
            K_WA(I) = 1 + max(KTIME$, KCOM, K_WA(I), K_RA(I),
     1      KPROC, K_WB(I), K_LB, K_WC(I), K_LC)
            K_RB(I) = 1 + MAX0(K_RB(I), K_WA(I))
            K_RC(I) = 1 + MAX0(K_RC(I), K_RA(I))
            A(I) = B(I) + C(I)
      ENDDO
```

Figure 20 Exemple de boucle DO incluant le moniteur.

On notera, à la figure 20, l'absence de moniteur pour les lignes de la boucle DO. C'est cet état de fait qui permet de considérer les variables d'index comme des constantes disponibles en tout temps.

Ces deux types de boucles peuvent être qualifiées de structurées. On entend par structuré, que l'influence de la boucle est délimitée par un bloc qui comporte un début et une fin.

Les boucles DO peuvent aussi être écrites avec des étiquettes (figure 21). L'insertion des lignes de moniteur avant la ligne de code étudiée oblige OSASMO à faire des accrobaties pour que les lignes de moniteur puissent être exécutées tout au long de chaque itération de la boucle. En effet, en se référant à la figure 21, OSASMO doit créer un nouveau numéro d'étiquette qui prend la place de l'ancien à la première ligne de la boucle et ensuite placer l'ancienne étiquette à la première ligne de moniteur placée à la dernière ligne de la boucle DO. On garde ainsi l'ancien numéro d'étiquette pour le reste de la logique du

programme qui peut potentiellement faire référence à celui-ci. Ainsi les lignes de moniteur associée à la dernière ligne de code de la boucle DO seront exécutées si cette étiquette devient la destination d'un saut (GOTO). OSASMO s'assure que la nouvelle étiquette choisie (Ex.: # 11 à la figure 21) n'existe pas ailleurs dans l'application source.

Les boucles de type DOWHILE, ayant une ligne de contrôle, ne possèdent pas de variable à gauche de l'égalité puisque l'égalité est tout simplement absente de celle-ci. Afin de garder la valeur du temps pour laquelle on peut conclure que la ligne peut-être exécutée, il faut créer une variable synthétique temporaire. Cette variable synthétique est de type *DOWtype*. Elle a le même rôle que la variable synthétique KTIME$, mais uniquement pour le bloc d'influence de la boucle DOWHILE. Cette variable permet d'avoir plusieurs blocs d'influence DOWHILE, imbriqués les uns dans les autres en changeant la valeur du chiffre final de la définition *DOWtype*.

Soit le fragment de programme suivant:

```
        DO 10 I=1,FIN
        ...
10      A = B + C
        ...
        GOTO 10
```

(suite autre page)

Le même fragment avec le moniteur inclu:

```
        lignes de moniteur
        DO 11 I=1,FIN
        ...
10      lignes de moniteur
11      A = B +C
        ...
        GOTO 10
```

Figure 21 Exemple de traitement des boucle DO avec étiquettes.

2.4.5 Fonctions et sous-routines

Dans cette catégorie, on distingue deux types différents d'instruction:

1- Définition des fonctions, sous-routines et appel de sous-routines,
2- Appel de fonction.

Dans un premier temps, toutes les fonctions et sous-routines, appels ou définitions, sont traités de la même façon. On insère, parmi leurs arguments, les homologues synthétiques, et pour les fonctions, on inclut également le temps d'achèvement de l'appel de la fonction afin d'obtenir la valeur de la variable synthétique correspondant à la variable à gauche de l'égalité. Les fonctions et sous-routines sont considérées comme étant une continuation du déroulement du programme, mais à une localisation spatiale (dans le programme) qui est différente. Les fonctions intrinsèques (Ex.: sinus()) sont jugées comme des lignes ordinaires de code à trois adresses.

2.4.6 Détermination des lignes de code du moniteur à insérer pour chaque instruction typique

Pour chaque catégorie d'instructions typiques, des lignes de moniteur leurs sont associées. L'énumération de celles-ci fera l'objet de la présente section.

La première zone du fichier programme (incluant le moniteur) est celle des définitions, des déclarations et des initialisations. Dans cette zone, OSASMO insère la déclaration des vecteurs synthétiques ainsi que des variables et des vecteurs de gestion du moniteur.

Plusieurs lignes, implantées par le moniteur, sont communes à toutes les instructions typiques. En d'autres termes, elles se retrouvent dans les insertions du moniteur de chaque ligne de code source analysée:

1- Allocation du processeur pour la ligne d'analyse à exécuter (allocation ligne-à-ligne). Cette tâche est confiée à la fonction Kec_proc($\{nbproc\}_{opt}$). Le nombre de processeurs est optionnel et dépend de l'architecture étudiée.

2- Mise à jour des variables synthétiques de type *K_Rtype* ayant leurs homologues dans la ligne d'analyse.

3- Recherche dans le vecteur d'utilisation des processeurs, KTPROC, pour vérifier l'occupation de ceux-ci à l'aide de la fonction de recherche définie comme étant Kec_search(KPROC, *K_Wtype*, KTPROC, LAST).

4- L'appel des sous-routines d'insertion des valeurs de parallélisme (nombre d'instructions exécutées simultanément) et des coûts de communication pour

mettre à jour l'histogramme des résultats. Les deux sous-routines sont définies comme étant IN_PARA(K_STAT, *K_Wtype*, PROF) et IN_COM(K_STAT, KCOM, *K_Wtype*, PROF)

5- L'initialisation, à la valeur un, des variables synthétiques utilisées pour la première fois dans le programme (incluant le moniteur). De façon à être certain que les variables synthétiques ne sont initialisées qu'une seule fois dans l'exécution du programme, un vecteur, KCLE, s'occupe de gérer celles-ci, comme l'illustre la figure 22. Une clé (KCLE) existe pour chaque variable à initialiser, c'est pour cette raison qu'il s'agit d'un vecteur. Chaque nouvelle variable rencontrée implique un changement d'indice pour KCLE, selon un principe itératif. Ces clés, qui sont essentiellement des blocs IF-THEN, ne doivent être utilisées qu'une seule fois durant l'exécution du programme. La valeur 33 de l'exemple de la figure 22 est arbitraire puisqu'il s'agit d'un fragment de programme.

Soit le fragment de programme suivant:

```
      KPROC = Kec_proc()
      K_PA = KPROC
      IF(KCLE(33).EQ.0)THEN          (La valeur 33 est arbitraire)
            K_WB = 1
            K_RB = 1
            K_PB = Kec_proc()
            KCLE(33) = 1
      ENDIF
      K_WA = 1 + max(KTIME$, KCOM, K_WA, K_RA, K_PA
     1 K_WB, K_PB, K_WC, K_PC, KPROC)
      A = B + C
```

Dans le cas où la variable B est utilisée pour le première fois dans le programme.

Figure 22 Exemple d'initialisation des variables synthétiques.

La figure 23 montre les lignes de moniteur pour une ligne de code à trois (3) adresses typique. La ligne de code considérée est KTMP1 = 2 * NN. Il est à noter que les fonctions intrinsèques du FORTRAN sont traitées de la même façon que les lignes de code à trois adresses. On remarque qu'aux lignes marquées d'un (X), la variable KPROC est augmentée de 1 pour indexer les matrices où KPROC se trouve. La raison de cette addition est purement technique. En FORTRAN le premier espace mémoire des vecteurs et matrices débute à l'indice un. Dans la langage C, il débute à l'indice zéro. Or, le module (fonction) qui calcule le numéro de processeur est programmé en langage C. Les numéros de processeur commencent donc à zéro dans cette fonction. L'addition du 1 n'est donc requis que pour rendre compatibles ces numéros avec les indices de matrices de gestion.

Soit la ligne de code FORTRAN suivante:

```
      KTMP1 = 2 * NN
```

La même ligne après l'insertion du moniteur par OSASMO:

```
                    KPROC=Kec_proc(NB_PROC)
                    K_WKTMP1 = 1+maxtmp1(MULT_COM, NB_PROC,
                $   KTIME$, KCOM, KPROC, K_WNN
                $   ,K_LNN,0,0,0,0,0,0,0,0,0,0)
                    DO WHILE(Kec_search(KPROC, K_WKTMP1, KTPROC,
                $   LAST, NB_PROC, PROF).EQ.1)
                        K_WKTMP1=K_WKTMP1+1
                    ENDDO
(X)                 KTPROC(KPROC + 1,LAST)=K_WKTMP1
                    K_RNN = MAX0(K_WKTMP1,K_RNN)
(X)                 K_LNN(KPROC+1) = 1
                    DO KEC=1,NB_PROC
                        K_LKTMP1(KEC)=0
                    ENDDO
(X)                 K_LKTMP1(KPROC+1)=1
                    KTMP1 = 2*NN
```

Figure 23 Exemple typique d'une ligne de code à 3 adresses avec le moniteur inséré par OSASMO.

Le cas de la boucle DO est illustré à la figure 24. On constate que le moniteur n'ajoute aucune ligne de code supplémentaire. De fait, puisque les variables d'index sont utilisées comme des constantes, elles sont considérées comme étant disponibles en tout temps, et ce, dans toutes les unités processeur-mémoire du système simulé. Aucune pénalité n'est donc comptabilisée.

Soit la boucle DO suivante:

```
                    DO I = 1, NN
                *
                *   Corps de la boucle ici
                *
                    ENDDO
```

Voici la même boucle après traitement par OSASMO:

```
      DO I=1, NN
*
*Corps de la boucle ici
*
      ENDDO
```

Figure 24 Exemple de boucle DO avec le moniteur inséré par OSASMO.

La figure 25 montre un arrangement IF-THEN-ELSE typique avec le moniteur inséré par OSASMO. La même méthode est utilisée pour tous les branchements conditionnels.

Soit l'arrangement IF-THEN-ELSE suivant:

```
      IF(J.GT.I)THEN
*     Bloc d'influence 1
      ELSE
*     Bloc d'influence 2
      ENDIF
```

Voici le même arrangement avec le moniteur inséré par OSASMO:

```
      KPROC=Kec_proc(NB_PROC)
      K_EC_IFT1=1+maxfct(MULT_COM, NB_PROC, KTIME$,
$     KCOM, KPROC, K_WJ, K_LJ, K_WI, K_LI, 0,0,0,0,0
$     ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
$     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
$     ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
      DOWHILE(Kec_search(KPROC,K_WI,KTPROC,LAST
1     ,NB_PROC,PROF).EQ.1)
            K_WI=K_WI+1
      ENDDO
      KTPROC(KPROC+1,LAST)=MAX0(K_WJ,K_WI)
      K_RJ=K_WI
      K_LJ(KPROC+1) = 1
```

```
      K_RI=K_WI
      K_LI(KPROC+1) = 1
      KTIME$=K_EC_IFT1
      CALL IN_PARA(K_STAT,K_EC_IFT1,PROF)
      CALL IN_COM(K_STAT,KCOM,K_EC_IFT1,PROF)
      IF(J.GT.I)THEN
*         Bloc d'influence 1
      ELSE
*         Bloc d'influence 2
      ENDIF
      KTIME$=1
```

Figure 25 Exemple d'une structure IF-THEN-ELSE typique avec le moniteur inséré par OSASMO.

La figure 26 identifie les lignes supplémentaires insérées par OSASMO pour une fonction typique définie par l'usager. À la ligne 4 de cette figure, on rencontre la variable KFUNCTN qui permet d'obtenir le moment (en temps relatif) auquel on peut conclure que la fonction est terminée. C'est donc cette valeur qui est utilisée pour déterminer le temps relatif de disponibilité de la variable temporaire TMP1.

Soit la fonction définie par l'usager suivante:

```
      TMP1=FUNCTN(X, I, A)
```

Voici la même fonction avec le moniteur inséré par OSASMO:

```
1        KPROC=Kecproc(NB_PROC)
2        TMP1 = FUNCTN(X, I, A, K_WX, K_RX, K_LX, K_WI,
3     $  K_RI, K_LI, K_WA, K_RA, K_LA, KTIME$, KFUNCTN)
4        K_WTMP1=1+maxfct(MULT_COM, NB_PROC,
5     $  KFUNCTN, KCOM, KPROC, K_WX, K_LX, K_WI,
6     $  K_LI, K_WA, K_LA, K_WJ, K_LJ, 0,0,0,0,0,0,0,0,0,0,0,0
7     $  ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
8     $  ,0,0,0,0,0,0,0,0)
9        DOWHILE(Kec_search(KPROC,K_WTMP1,KTPROC,LAST
```

```
10          1  ,NB_PROC,PROF).EQ.1)
11                 K_WTMP1=K_WTMP1+1
12             ENDDO
13             KTPROC(KPROC+1,LAST)=K_WTMP1
14             DO KEC=1,NB_PROC
15                 K_LTMP1(KEC)=0
16             ENDDO
17             K_LTMP1(KPROC+1)=1
18             CALL IN_COM(K_STAT,KCOM,K_WTMP1,PROF)
```

Figure 26 Exemple typique de fonction définie par l'usager incluant le moniteur inséré par OSASMO.

Mentionnons que les sous-routines sont considérées comme étant un prolongement du programme qui n'est tout simplement pas placé à la suite du code principal. Il s'agit donc d'un simple saut avec un retour au code principal à la même ligne lorsque les instructions de la sous-routine ont été exécutées. Aucune mesure spéciale n'est donc prise pour ces instructions.

2.4.7 Extraction des résultats

Deux types d'algorithmes d'extraction des résultats sont présents dans le moniteur. Tout d'abord, les algorithmes se rapportant au parallélisme et, deuxièmement, ceux se référant aux coûts de communication. Les deux types sont directement influencés par le genre d'architecture étudiée. Ce projet se concentrant sur trois architectures typiques, les algorithmes requis pour ces architectures seront décrits dans les prochaines sous-sections.

2.4.7.1 Niveau de parallélisme

Le degré de parallélisme d'une application scientifique est défini par le nombre moyen de lignes pouvant être exécutées en parallèle (simultanément). La méthode générale pour recueillir ces données étant largement documentée dans le début de ce chapitre, des remarques plus spécifiques seront énoncées ici.

La décomposition en code à trois adresses constitue un avantage véritable pour la recherche du degré de parallélisme. Sans cette décomposition, les lignes de codes ayant plus de deux opérandes seraient évaluées comme étant une seule ligne à exécuter. Puisqu'en général, les processeurs n'exécutent qu'une seule ligne de code à trois adresses à un instant donné, cela induirait des erreurs dans l'évaluation du degré de parallélisme qui ne serait alors qu'approximatif. Cette approximation indiquerait une compression dans la durée en temps (relatif) de l'exécution alors qu'en réalité cette durée serait plus longue, d'où des résultats trop optimistes.

Les fonctions intrinsèques sont traitées comme ayant le même nombre de cycles d'exécution qu'une ligne de code à trois (3) adresses. Ce paramètre est variable, mais dans le cadre de cette recherche, le temps d'exécution d'une fonction intrinsèque a été fixé à 1 cycle de temps relatif.

Les lignes temporaires engendrées par la décomposition en code à trois adresses, sont considérées comme étant exécutées par le même processeur qui aurait été utilisé pour exécuter la ligne d'origine (sans décomposition). OSASMO considère donc que si une ligne du code source est décomposée en x lignes temporaires, ces x lignes temporaires seront exécutées par le même processeur. En effet, puisque l'étape de la décomposition en code à trois adresses est normalement réalisée par un compilateur, et que l'allocation des

processeurs, dans un système parallèle, est généralement confiée au programmeur, ce dernier peut allouer les différents processeurs à des lignes de code qui ne sont pas encore décomposées en code à trois adresses. Puisque le compilateur ne peut ajouter d'information au programme que le programmeur a créé, toutes les lignes qui sont décomposées en code à trois adresses par le compilateur ne peuvent être exécutées que par le processeur qui a été déterminé au préalable. Ceci a pour conséquence que toute les lignes de code à trois adresses correspondant à une ligne du code source sont exécutées par le même processeur.

Lorsque les lignes du code source d'origine ont plus d'une ligne temporaire (après décomposition en code à trois adresses), OSASMO s'assure que ces lignes sont exécutées à des instants différents. En effet les processeurs ne peuvent, en général, exécuter qu'une seule ligne de code à trois adresses à un temps donné.

2.4.7.2 Coûts de communication

L'évaluation des coûts de communication sont une des contributions importantes de cette recherche. Les simulations ont été réalisées sur trois différents types d'architecture du réseau de communication. La méthode d'évaluation des coûts de communication est propre à chacune. Ainsi, chaque architecture est caractérisée par son propre algorithme d'évaluation des communications. Les équations, présentées ci-dessous, expriment les coûts de communication engendrés par l'exécution d'une ligne complète de code complète.

Coûts des communications pour une architecture en hypercube.

$$\text{Coûts} = \sum_{i=1}^{N} \left[(P_{req} \oplus P_{rép}) * \text{coût d'une opération de communication} \right]_i \qquad (3)$$

où $\oplus$ est le symbole du "OU exclusif" et où P_{req} est le numéro du processeur requérant une communication, et $P_{rép}$ est le numéro du processeur répondant à la *i*ème requête. *N* est le nombre de requêtes de communication de la ligne de code considérée. Une opération de communication de base, pour l'hypercube, est une communication entre deux processeurs dont la distance de Hamming entre les adresses qui leur sont associées est égale à 1 [10]. L'hypercube peut avoir jusqu'à M opérations de communications pour une seule requête lorsque la distance de Hamming est de M entre le requérant et le répondant, où M est le diamètre d'un hypercube comptant 2^M processeurs.

Coûts des communications pour une architecture en bus.

$$\text{Coûts} = \sum_{i=1}^{N} (\text{coût d'une opération de communication})_i \qquad (5)$$

où N est le nombre de requêtes de communication de la ligne de code considérée.

Coûts des communications pour une architecture en grille.

$$\text{Coûts} = \sum_{m=1}^{N} \left[(|i-k| + |j-l|) * \text{coût d'une opération de communication} \right]_m \qquad (6)$$

où N est le nombre de requêtes de communication de la ligne de code source étudiée. L'adresse de tous les processeurs de la grille est exprimée à l'aide de deux indices (i et j pour le requérant, k et l pour le répondant) puisque le réseau est bidimensionnel.

2.5 Remarques

OSASMO étant un outil très complexe, il est difficile dans le cadre de ce mémoire d'en faire une explication exhaustive. De plus, cet outil étant un prototype, certaines caractéristiques peuvent changer ou évoluer. Cependant, les informations contenues dans

ce chapitre sont les plus récentes spécifications concernant le logiciel et les résultats qui sont présentés dans les chapitres suivants ont été obtenus avec la version d'OSASMO précédemment décrite.

CHAPITRE III

Validation expérimentale d'OSASMO

Tout outil d'analyse doit nécessairement être vérifié afin de démontrer sa validité, son efficacité et ses limites. Ce chapitre présente la méthode utilisée pour valider l'outil de simulation architecturale OSASMO. Les sections qui suivent décrivent les hypothèses de départ concernant l'outil et exposent le format des instructions synthétiques standards d'un programme. De plus, un exemple détaillé est fourni et les limites d'OSASMO sont discutées.

3.1 Hypothèses de départ concernant le fonctionnement d'OSASMO

Les hypothèses de départ nous serviront comme exercice d'introduction aux capacités et aux limites de l'outil. En effet, à la lumière des hypothèses qui seront énumérées dans cette section, il sera plus aisé de discerner les caractéristiques d'OSASMO. Les hypothèses de départ seront présentées sous forme de tableau. Elles sont divisées en deux catégories. La première est composée des hypothèses de départ communes à chaque architecture choisie et la seconde présente les hypothèses qui sont propres à une ou plusieurs architectures. Le tableau 1 présente ces hypothèses. Comme proposé dans le chapitre I, certains critères énoncés par Levitan [14] seront utilisés pour définir des hypothèses de départ.

Tableau 1 Hypothèses de départ et leur type.

#	Type	Hypothèse	Hyp.	Bus	Grille
1a	C	Largeur de bande infinie.	X		X
1b	C	Largeur de bande de un.		X	
2a	C	Le diamètre est de un.		X	
2b	C	Le diamètre est de n, où n est la dimension d'un hypercube.	X		
2c	C	Le diamètre est de M + N, où M et N sont les dimensions d'une grille.			X
3a	C	L'architecture de base est de type flot de donnée, seul le réseau est de type hypercube.	X		
3b	C	L'architecture de base est de type flot de donnée, seul le réseau est de type bus.		X	
3c	C	L'architecture de base est de type flot de donnée, seul le réseau est de type grille.			X
4	C	Les liens du réseau de communication sont bi-directionnels.	X	X	X
5	P	Un processeur peut recevoir plusieurs réponses au même instant.	X	X	X
6	P	Un processeur ne peut exécuter qu'une seule instruction à un instant donné.	X	X	X
7	P	Les processeurs ne sont pas ralentis par les opérations de communications.	X	X	X
8	L	Les variables d'index sont considérées comme des constantes.	X	X	X
9	L	Une ligne de code est exécutée en une unité de temps relatif.	X	X	X
10	L	OSASMO tient compte des différentes localités de la variable après lecture.	X	X	X
11	L	Les définitions, les déclarations, les instructions DATA et PARAMETER, ainsi que toutes les entrées/sorties ne sont pas comptabilisées dans les résultats de l'exécution des applications scientifiques étudiées.	X	X	X
12	L	Les données sont introduites dans la mémoire des processeurs simulés de façon aléatoire au début de l'exécution de l'application scientifique.	X	X	X

Légende	
C	Hypothèses se rapportant au réseau de communication
L	Hypothèses se rapportant à l'application-source étudiée
P	Hypothèses se rapportant aux processeurs
hyp	Hypercube

OSASMO offre la possibilité de changer la majorité de ces hypothèses. Ceci confirme bien le côté versatile de cet outil.

La largeur de bande (#1a et b) est le nombre de messages que peut recevoir un noeud par cycle relatif d'exécution. On se rend compte que le bus ne peut envoyer qu'un seul message à la fois. On notera que cette hypothèse a une grande influence sur les résultats.

Le diamètre définit (#2 a, b et c) la plus longue distance que doit parcourir un message dans le système simulé. Cette valeur varie avec le type d'architecture.

L'architecture de base de type "à flot de données" (#3 a,b et c) signifie qu'aussitôt qu'une donnée est prête à être utilisée, le système peut en faire usage. On notera cependant que le temps s'écoule de façon discrete, c'est-à-dire par incrément constant.

Les hypothèses qui suivent (#4 et plus) sont les mêmes pour toutes les architectures simulées. Chaque lien de communication peut à la fois recevoir et envoyer des messages et le faire au même instant. L'hypothèse #8 (les variables d'index sont considérées comme des constantes) implique que les boucles peuvent être exécutées le plus parallèlement possible en fonction du contexte.

On notera que les entrées/sorties sont beaucoup trop spécifiques à chaque système pour tenter de les inclures dans les calculs de coûts.

Initialement les données sont dispersées aléatoirement dans le système simulé. Cette hypothèse tend à augmenter le coût des communications au début de l'exécution du programme.

3.2 Validation du traitement des instructions typiques par OSASMO

La méthode utilisée pour valider le traitement des instructions typiques, par le moniteur, est de capturer la trace des dites instructions lorsqu'un programme-source est exécuté. Six différents programmes ayant chacun leurs particularités ont été soumis à cette validation. Les traces ainsi que les programmes sources sont répertoriés dans l'annexe 2. Le format de sortie des traces est celui illustré par la figure 27. Chaque programme a été testé avec une architecture hypercube ayant un diamètre de 8, (dimension égale à 3) et une allocation aléatoire des ressources. Ces programmes testent les différentes instructions typiques mentionnées au chapitre II.

Paramètres spatio-temporels associés aux variables de l'instruction FORTRAN analysée.	Instruction FORTRAN analysée

Figure 27 Format de sortie des traces de l'exécution des programmes tests.

Plusieurs formes d'expression des paramètres spatio-temporels sont possibles pour la section gauche du format de sortie montrée à la figure 27. Ces diverses formes sont dépendantes du type d'instruction de la section de droite du format de la même figure. Plusieurs abréviations sont utilisées pour alléger le texte de la trace. Ces abréviations sont présentées au tableau 2. La première abréviation COC donne le temps maximum que requiert une ligne de code pour amener localement les variables qui sont dans d'autres cellules processeur-mémoire, et ce, en tenant compte des localités de chaque variable. Le symbole P correspond au numéro du processeur dans lequel la ligne de code est exécutée.

C'est grâce à ce paramètre que l'on peut déduire les coûts de communication. La valeur de T, quant à elle, donne la valeur de temps minimum de disponibilité de la variable correspondante, et ce, en tenant compte des différents processeurs (localités) où l'on peut retrouver la variable. L'abréviation TBI, nous informe sur le temps minimum auquel on peut conclure qu'un bloc d'influence (délimité par des lignes de contrôle) peut être exécuté.

Tableau 2 Tableau des abréviations utilisées dans les traces des programmes tests.

Abréviation	Signification
COC	Nombre de **C**ycles d'**O**pération de **C**ommunication maximum requis pour les communications d'une ligne de code.
P	Numéro du **P**rocesseur, où la ligne de code est exécutée.
TBI	Valeur de **T**emps auquel on peut conclure que le **B**loc d'**I**nfluence d'une ligne de contrôle peut être exécutée.
T	Valeur de **T**emps donnant la disponibilité de la variable correspondante.

Chaque type de format est expliqué en détail dans le présent paragraphe. Le format de sortie du code à trois adresses est défini comme suit:

$$\text{P: \{chiffre\}}_{1+}\text{; T: \{chiffre\}}_{1+} = \{\text{P: \{chiffre\}}_{1+}\text{; T: \{chiffre\}}_{1+}\}_{1+} \{\text{TBI: \{chiffre\}}_{1+}\}_{opt} (\text{COC \{chiffre\}}_{1+})$$

La partie à gauche de l'égalité donne les coordonnées spatio-temporelles après l'exécution de la ligne de la variable occupant la même position dans la ligne de code. Il en est de même pour les données occupant la partie droite de l'égalité. Ces dernières donnent cependant les valeurs avant l'exécution de la ligne. Le temps TBI n'est nécessaire que lorsque la ligne est à l'intérieur d'un bloc d'influence. Ce temps est donc optionnel. Le dernier membre de la partie de droite, donne le nombre maximal de cycles utilisés pour les opérations de communication.

Le format d'une ligne d'instruction IF est, quant à lui, défini comme étant:

$$\text{IF:T} = \{\text{chiffre}\}_{1+}\text{; P: } \{\text{chiffre}\}_{1+}\,(\text{COC } \{\text{chiffre}\}_{1+})$$

La valeur de T donne le temps auquel on peut conclure que la ligne peut être exécutée, le P indique le numéro du processeur dans lequel le test conditionnel a été exécuté et la dernière partie a la même signification que pour la ligne de code à trois adresses.

Le DOWHILE a le même format de sortie que le IF, sauf pour ce qui est des premières lettres donnant le type d'instruction (IF:). Elles sont remplacées par "DOW:".

La boucle DO ne possède pas de format de sortie puisque la variable d'index est considérée comme une constante.

3.3 Exemple typique

Ce chapitre n'a pas pour but de donner une description détaillée des traces présentées à l'annexe 2. En effet, l'explication exhaustive serait excessivement longue et n'apporterait aucun élément additionnel. Cependant, pour aider la consultation et la compréhension de l'annexe 2, l'exemple 5 sera brièvement décrit.

La figure 28 montre le code FORTRAN de l'exemple 5 de l'annexe 2. La trace d'exécution de cet exemple sera présentée à l'aide des formats de sorties présentés précédemment. L'explication de la trace d'exécution sera développée sous forme de tableau pour plus de clarté.

Code source de l'exemple # 5

```
      PROGRAM ERIC_C5
C
      INTEGER A,B
C
      A=1
      B=2
C
      IF(A*7.LT.6+B)THEN
          A=B*9
      ELSE
          B=B-A
      ENDIF
      END
```

Figure 28 Code FORTRAN de l'exemple 5 de l'annexe 2

Tableau 3 Trace de l'exécution de l'exemple # 5 avec le moniteur inséré

Lignes de code et le format de sortie correspondant	Explication du format de sortie
A=1 ; [P: 0, T: 2] B=2 ; [P: 6, T: 2]	Assignation de la valeur 1 dans la variable A au temps 2 et dans le processeur # 0. Assignation de la valeur 2 dans la variable B au temps 2 dans le processeur # 6.
TMP1=A*7 P: 1; T: 13 = P: 0; T: 2 (COC 10)	Cette ligne est exécutée dans le processeur # 1. La variable A est présentement dans la mémoire du processeur # 0. Donc, on doit amener cette variable dans la mémoire du processeur # 1. D'après la distance de Hamming (hypercube), entre les processeurs # 0 et # 1, il n'y a qu'une étape de communication entre les deux processeurs. Le COC est alors de 10 puisqu'on considère que le coût d'une opération de base est de 10 unités relatives. La valeur 7 étant une constante, aucun coût de communication n'est engendré. Le bilan temporel est donc le suivant: T = 10 cycles de communication + 1 cycle d'exécution de la ligne + 2 (temps de disponibilité de A) = 13.

Tableau 3 Trace de l'exécution de l'exemple # 5 avec le moniteur inséré (suite)

TMP2=6+B P: 1; T: 33 = P: 6; T: 2 (COC 30)	Cette ligne est exécutée dans le processeur # 1. La variable B est présentement dans la mémoire du processeur # 6. Donc, on doit amener cette variable dans la mémoire du processeur # 1. D'après la distance de Hamming (hypercube), entre les processeurs # 6 et # 1, il y a trois étapes de communication entre les deux processeurs. Le COC est alors de 30 puisque le coût d'une opération de base est de 10 unités relatives. La valeur 6 étant une constante, aucun coût de communication n'est engendré. Le bilan temporel est donc le suivant: T = 30 cycles de communication + 1 cycle d'exécution de la ligne + 2 (temps de disponibilité de B) = 33.
IF(TMP1.LT.TMP2)THEN IF:T = 34; P: 1 | P: 1; T: 13, P: 1; T: 33 (COC 0)	La vérification du test du IF est exécutée dans le processeur # 1. Puisque les lignes de variables temporaires sont exécutées dans le même processeur que celui du IF, aucun coût de communication n'est engendré. En effet, les deux valeurs (TMP1 et TMP2) sont déjà placées dans la mémoire du processeur # 1. Le bilan temporel est donc le suivant: T = max(13, 33) + 1 cycle d'exécution de la ligne = 34. Les valeurs de temps 13 et 33 sont les temps de disponibilité des variables temporaires TMP1 et TMP2 respectivement.

Tableau 3 Trace de l'exécution de l'exemple # 5 avec le moniteur inséré (suite)

A=B*9 P: 6; T: 35 = P: 6; T: 2 TBI: 34 (COC 0)	Cette ligne est exécutée dans le processeur # 6. La variable B est présentement dans la mémoire du processeur # 6 et disponible au temps 2. Donc, il n'est pas nécessaire d'amener cette variable dans la mémoire du processeur # 6, puisqu'elle s'y trouve d'où un COC de 0. La valeur 9 étant une constante, aucun coût de communication n'est engendré. Par contre, cette ligne ne peut être exécutée qu'à la condition où la branche THEN du IF soit prise. Il faut donc attendre que le test de la ligne du IF soit exécutée. On ne peut donc conclure avant le temps TBI: 34 que cette ligne peut ou non être exécutée. Il faut donc tenir compte du temps TBI dans le calcul du temps de disponibilité de A Le bilan temporel est donc le suivant: T = max(34, 2) + 1 cycle d'exécution de la ligne = 35, où 34 est le temps TBI et 2 est le temps de disponibilité de la variable B.
ENDIF	

3.4 Limitations de l'outil

OSASMO possède certaines limites et contraintes d'utilisation. Il est important de les relever pour permettre à l'utilisateur d'évaluer dans une plus juste mesure les résultats extraits et, à un autre niveau, pour permettre à des recherches ultérieures d'améliorer le domaine d'application.

3.4.1 Traitement des étiquettes

L'utilisation des étiquettes dans des applications scientifiques implique généralement que cette dernière possède une logique de programmation non-structurée. Ce fait a pour conséquence de rendre l'application d'OSASMO très difficile. Dans certaines circonstances, OSASMO doit traiter certaines parties de l'application de façon différente. En d'autres mots, OSASMO ne peut insérer le moniteur qui réussirait à extraire de façon optimale le niveau de parallélisme et les communications. Illustrons par un exemple typique (figure 30) les difficultés que présente une section de programme écrit en FORTRAN non-structuré. La principale difficulté découle de la nécessité pour KTIME$ d'obtenir le temps minimum auquel on peut conclure qu'un bloc d'influence peut être exécuté. Dans une structure comme celle de l'exemple de la figure 30, il est impossible de tirer le maximum de la situation. Il est extrêmement difficile de déterminer de façon systématique le début et la fin des blocs d'influence. Ceci complique les initialisations de la variable de gestion du moniteur KTIME$.

Normalement, les initialisations de la variable KTIME$ sont effectuées au début et à la fin d'un bloc d'influence. Pour un IF-THEN-ELSE ou une boucle DOWHILE, il est facile de déterminer le début et la fin, mais pour des programmes comme celui de l'exemple

de la figure 30, OSASMO ne peut les déterminer sans erreur. La figure 29 montre un extrait de programme incluant une condition sous forme de IF-LOGIQUE (LOGICAL-IF). On peut conclure que la ligne de l'étiquette 20 peut être exécutée peu importe la branche du IF qui est exécutée. OSASMO analyse de façon structurée les étiquettes des programmes sources. Il y a, par contre, des situations où l'analyse structurée des étiquettes pose des problèmes.

Voici un fragment de programme incluant une condition de la forme "IF-LOGIQUE" (LOGICAL-IF):

```
          ...
          IF(A)GOTO 10
               ...
          GOTO 20
10
               ...
20
          ...
```

Figure 29 Fragment de programme incluant une condition sous la forme d'un "IF-LOGIQUE".

L'étiquette 20 de l'exemple présenté à la figure 30 est un cas typique où l'analyse pose un problème. En effet, cette étiquette ne peut être atteinte qu'en passant par un ou l'autre des IF du fragment de programme. Il y a donc une condition minimale à respecter. La condition requise est la ligne de contrôle de l'un ou l'autre des IF. Ceci implique que le temps où l'on peut conclure que la ligne (6) peut être exécutée ne sera jamais le temps minimum (i.e. =1), puisque dans toutes les situations où cette ligne est atteinte, la trace de l'exécution du programme doit nécessairement croiser un des IF. OSASMO change le temps de la variable KTIME$ au début et à la fin d'un bloc de base. La ligne (3) est la fin

du bloc de base débutant à la ligne (1). Ainsi, KTIME$ doit être réinitialisé à une certaine valeur à la fin de ce bloc. Cependant, puisque la ligne (6) ne peut être atteinte qu'en passant par la séquence de lignes (1-4-5), le temps de KTIME$ ne devrait pas être réinitialisé au temps minimum (i.e. =1), lors du passage par les lignes (1-3), puisque l'influence du bloc continue hors des limites normalement considérées par OSASMO. Le traitement qu'en fait OSASMO est de remettre à un la valeur de KTIME$ après la fin des blocs d'influence des IF des lignes (1) et (3). Cette procédure donne une valeur de temps minimum KTIME$ trop optimiste.

Soit le fragment de programme suivant:

```
(1)                 IF(A)GOTO 10
(2)                 ...
(3)                 GOTO 20
(4)           10    ...
(5)                 IF(B)GOTO 30
(6)           20    ...
(7)                 GOTO 40
(8)           30    ...
(10)          40    CONTINUE
(11)                ...
```

Les ... indiquent des parties de programmes qui ont été omises afin d'améliorer la compréhension de l'exemple.

Figure 30 Exemple de fragment de programme contenant des étiquettes et ne possédant pas de logique structurée.

3.4.2 Traitement des boucles DO

Le traitement des boucles DO doit être étudié avec circonspection, puisqu'un énorme potentiel de parallèlisme peut en être extrait. Afin d'extraire ce potentiel, OSASMO, traite les variables d'index des boucles DO comme des constantes. En d'autres

mots, les valeurs de ces variables sont considérées comme étant disponibles à tous les processeurs, et ce, en tout temps. Aucun coût de communication et aucun parallélisme n'est donc comptabilisé pour les variables d'index. Ceci permet d'extraire une partie du potentiel parallèle des boucles DO en fonction du type d'allocation des ressources et de l'architecture simulée. Cette façon de procéder ne limite pas OSASMO à un traitement particulier que pourrait imposer un système parallèle donné.

3.4.3 Limite du nombre de processeurs

Le nombre de processeurs constituant les différents types de systèmes d'ordinateurs parallèles peut varier sur une très large plage. La méthode utilisée par OSASMO pour gèrer l'information de la simulation implique la création de matrices ayant pour dimensions le nombre de processeurs et la durée (en unité de temps relative) de la simulation. Cela signifie que, pour un grand nombre de processeurs, un très grand espace mémoire doit être disponible pour conserver l'information. La limite du nombre de processeurs est, en fait, directement reliée à la capacité d'emmagasinage et à la mémoire volatile disponible sur les ordinateurs utilisés pour exécuter les applications scientifiques augmentées du moniteur.

Des problèmes peuvent être rencontrés lors de la compilation des programmes augmentés incluant les matrices de grandes dimensions. En effet, le compilateur demande beaucoup de mémoire volatile et d'espace-disque pour compiler ces applications. Ainsi, si le nombre de processeurs et de cycles relatifs de durée de la simulation sont trop élevés, l'ordinateur effectuant la compilation ne pourra arriver à créer la version exécutable.

L'avènement d'ordinateurs plus puissants ne peut qu'améliorer les performances d'OSASMO. On peut espérer que dans un proche avenir l'importance des simulations ne

pourra qu'augmenter permettant, par le fait même, la possibilité de réaliser des travaux mis de côté pour des raisons de durée de simulation.

CHAPITRE IV

Analyse et discussion des résultats expérimentaux

Ce chapitre expose les résultats d'expérimentation réalisés à l'aide d'OSASMO sur dix applications scientifiques différentes. Ces applications scientifiques sont tirées de sources connues, notamment du livre "Numerical recipes" [15] et d'une banque de logiciels développées par M. Jack Dongarra et M. Eric Grosse [5]. On y retrouve des algorithmes classiques tel le calcul d'une FFT, d'une inversion de matrices, de lissage de courbes, d'interpolation par le polynôme de Lagrange, etc. On y rencontre également deux bancs d'essais ("benchmark"), soit une décomposition LU de matrices et le banc d'essai de Whetstone. Comme nous pourrons le constater, ce dernier (Whetstone) nous amène à des conclusions intéressantes sur le potentiel de concurrence de certains types d'applications. Nous débuterons ce chapitre par une étude quantitative des données d'un fichier de test comprenant deux boucles DO imbriquées. Cet exemple permettra au lecteur d'avoir une meilleure vue d'ensemble du comportement dynamique de l'exécution des applications scientifiques.

4.1 Étude du comportement dynamique de l'exécution d'un programme simple

La quantification des performances de ce code permettra de montrer qu'OSASMO possède bien la capacité de tirer le potentiel de concurrence d'une application scientifique comportant des boucles DO. Cet exemple comporte deux boucles DO imbriquées, comme le présente la figure 31. Cet exemple simple familiarisera le lecteur avec la procédure d'extraction dynamique des statistiques de concurrence et de communication.

```
      INTEGER A(10, 10), I, J
C
      A=1
C
      DO I=1,10
            DO J=1,10
                  A(I, J)=A(I, J)+1
            ENDDO
      ENDDO
      END
```

Figure 31 Code FORTRAN d'un programme exemple.

Tout d'abord, deux traces d'exécution seront étudiées. La première provient de la simulation d'un système avec une architecture hypercube ayant quatre noeuds dans laquelle le coût d'une communication est égal au temps d'exécution d'une ligne de code à trois adresses (coût de communication = 1). Dans la seconde simulation, l'architecture considérée, possède seize noeuds et renferme les mêmes caractéristiques de communication.

Les résultats de la simulation de l'hypercube à quatre noeuds sont illustrés à la figure 32.

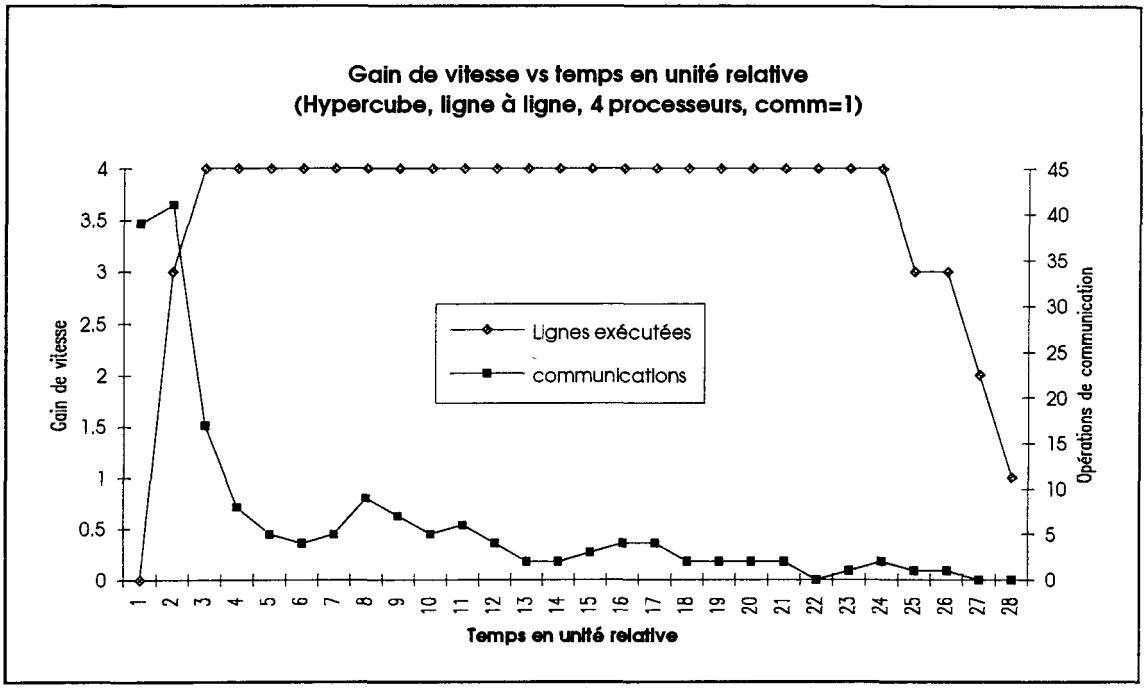

Figure 32 Trace d'exécution du programme de la figure 31 sur une architecture hypercube ayant quatre noeuds et dont le coût d'une communication est d'une unité relative de temps.

L'allocation des ressources de calculs (processeurs) utilisée pour obtenir les résultats de la figure 32 est aléatoire. Le temps total de la trace d'exécution parallèle est de 28 cycles relatifs pour un programme requérant un total de 100 instructions séquentielles. On remarque que les quatre processeurs sont utilisés la majeure partie du temps, car le gain de vitesse est égal à 4 pour les cycles compris entre 3 et 24. Cet exemple permet donc une utilisation efficace de la puissance de calcul disponible. Augmentons le nombre de processeurs disponibles afin de diminuer le nombre de cycles relatifs d'exécution. La figure 33, présente les résultats obtenus pour un hypercube de seize noeuds.

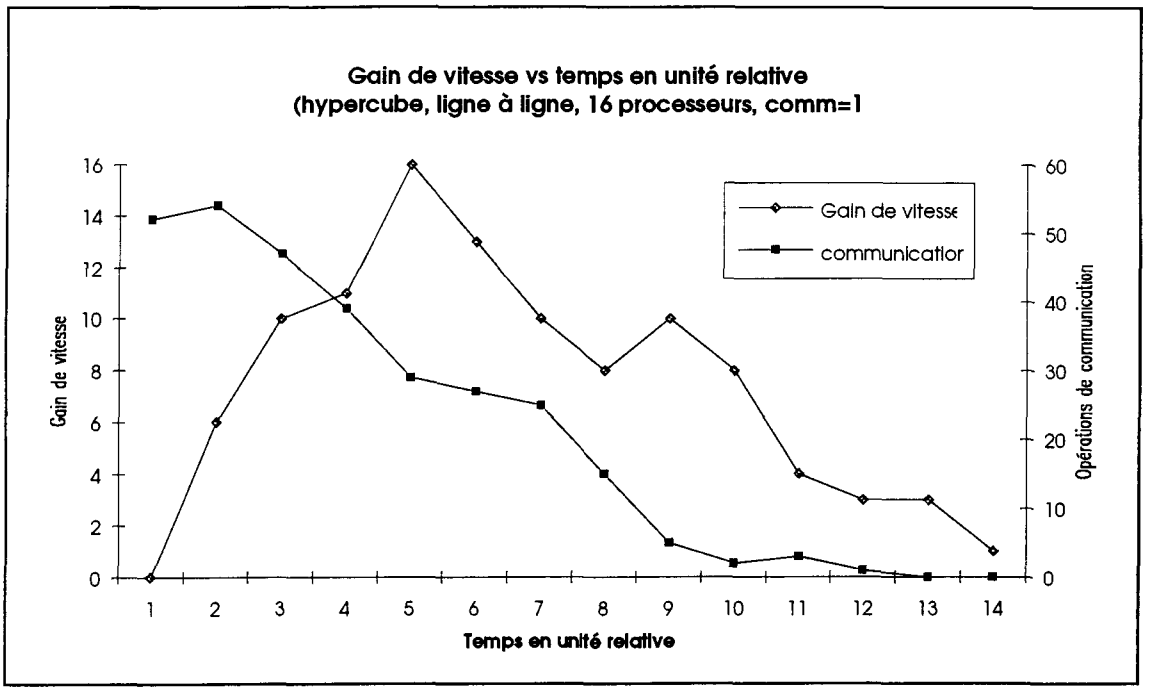

Figure 33 Trace d'exécution du fichier de la figure 31 sur une architecture hypercube ayant seize noeuds et un coût de communication d'une unité relative de temps.

Le temps d'exécution de cette simulation est de 14 cycles relatifs pour un programme requérant un total de 100 instructions séquentielles, ce qui constitue une bonne amélioration du temps d'exécution. Par contre, il a été nécessaire d'avoir quatre fois plus de processeurs pour diminuer d'un facteur deux le temps d'exécution. De plus, les processeurs sont exploités moins efficacement que ceux de la figure 32, puisque le seul moment où les seize processeurs sont tous utilisés est au cycle 5. Une tendance peut dès lors être mise en compte. La puissance de calcul augmente moins rapidement que le nombre de processeurs utilisés dans un système d'ordinateurs parallèles. Ce comportement montre bien que les communications sont une composante essentielle devant être considérée lors de l'évaluation des performances.

Étudions maintenant le comportement de l'exécution du programme de la figure 31 pour différentes valeurs du nombre de processeurs avec le coût d'une communication fixé à un cycle relatif. La figure 34 présente les résultats obtenus. La légende montre deux courbes: la première courbe est celle obtenue expérimentalement et la deuxième est une courbe logarithmique présentée pour fins de discussion.

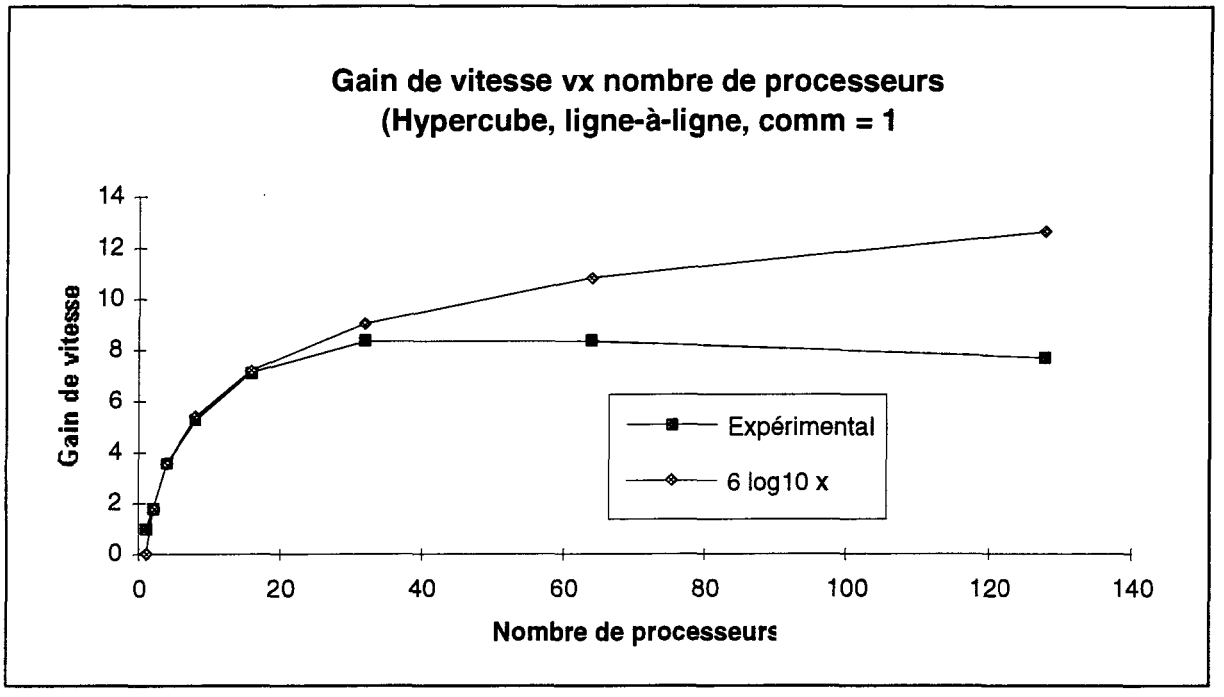

Figure 34 Gain de vitesse en fonction du nombre de processeurs pour le programme de la figure 31.

La figure 34 montre que le du gain de vitesse se dégrade pour un grand nombre de processeurs. De plus l'augmentation pour un nombre réduit de processeurs n'est pas linéaire. En fait, elle est approximativement logarithmique. En effet, la courbe **6 log x** tracée à la figure 34 montre une grande similarité avec la courbe expérimentale pour $x < 16$. Par la suite, le gain de vitesse expérimental ne suit pas la même augmentation asymptotique que la courbe **6 log x**. Lorsqu'un coût de communication égal à un cycle relatif est considéré, la valeur de gain de vitesse décroît pour les situations où l'on retrouve trente

processeurs et plus. Effectivement, les coûts de communications, pour ces valeurs du nombre de processeurs, deviennent prohibitifs et provoquent une dégradation importante des performances représentées par le gain de vitesse. Ces résultats sont en accord avec ceux obtenus par Frömm *et al.* [6] à l'aide d'un modèle mathématique ainsi que de façon expérimentale.

On constate donc qu'il existe un nombre optimum de processeurs pour l'exécution parallèle de programmes FORTRAN. Les résultats obtenus sur une dizaine d'applications sont présentés plus loin dans ce chapitre. Le développement d'une métrique serait intéressant afin de pouvoir juger de façon appropriée la valeur optimale (moyenne) du nombre de processeurs requis pour l'exécution d'applications scientifiques sur les systèmes parallèles considérés. Celle-ci devrait tenir compte des facteurs d'optimisation suivant: le gain de vitesse et le pourcentage d'utilisation des processeurs. La communication étant un facteur qui influence directement les valeurs de ces facteurs, elle serait implicitement considérée dans cette métrique. Une telle analyse a été réalisée et sera présentée dans une section ultérieure. Mais auparavant décrivons les applications choisies pour étude.

4.2 Présentation des applications scientifiques choisies

Des analyses ont été réalisées sur dix applications scientifiques connues et différentes. Le tableau 3 présente les applications et leur provenance. Certaines de ces applications sont des versions réduites des programmes originaux, ceci afin de diminuer les temps d'exécution des simulations. Les temps d'exécutions varient en effet de 15 minutes à 36 heures environ. C'est la raison pour laquelle plusieurs applications ont vu leur nombre d'itérations diminué pour rendre le temps d'exécution raisonnable. Ces applications sont identifiées par un astérisque (*) dans le tableau 3.

Tableau 3 Applications scientifiques utilisées pour les simulations et leur provenance.

Applications scientifiques	Sources
Décomposition en matrice LU (LU)*	Banque de logiciel netlib [5]
Banc d'essai de Whetstone (BENCH)*	Banque de logiciel netlib [5]
Lissage par la méthode d'expansion des gradient (LIS1)*	Local, (Voir source à l'annexe 3)
Lissage semi-paramétrique par la transformée de Fourier (LIS2)	Numerical recipes [15]
Algorithme de FFT (FFT)	Numerical recipes [15]
Inversion de matrice (INVMAT)	Numerical recipes [15]
Interpolation polynomiale (INTERPO.)	Numerical recipes [15]
Algorithme de réduction de Gauss (GAUSS)	Numerical recipes [15]
Optimisation par la méthode du Simplex (SIMPLEX)	Numerical recipes [15]
Multiplication de matrices carré (MULMAT)	Local, (Voir source à l'annexe 3)

4.3 Reproduction des résultats de Manoj Kumar

Comme mentionné au chapitre I, Kumar [12], dans ses expérimentations, a négligé la communication qui est pourtant un des facteurs pouvant influencer le plus les performances de programmes exécutés sur des machines parallèles [17]. Il est donc nécessaire de les inclure dans la simulation d'architectures parallèles. C'est ce qui a été réalisé avec l'outil de simulation architecturale OSASMO.

Cette section présentera l'évaluation du parallélisme dans des applications scientifiques en utilisant les critères de Kumar. Les hypothèses de départ de Kumar ne tiennent cependant pas compte des communications ou de la localisation spatiale des variables contenues dans les programmes testés.

Une mise au point est à faire à ce point de l'analyse, concernant la décomposition en code à trois adresses. Kumar ne spécifie pas qu'il décompose les applications scientifiques en code à trois adresses. Cette caractéristique est absolument nécessaire puisqu'il est généralement impossible à un processeur d'exécuter plusieurs opérations simultanément. Ceci a pour effet de rendre ses résultats plus optimistes comparativement à ceux obtenus lorsque le programme est décomposé en code à trois adresses. En effet, puisque dans son analyse, chaque ligne de code peut être exécutée en un seul cycle relatif, si ces lignes de code source avaient été décomposées en code à trois adresses nécessitant en moyenne 6 lignes de code à trois adresses, six cycles relatifs auraient dû être comptabilisés alors qu'un seul ne l'a été dans l'analyse de Kumar.

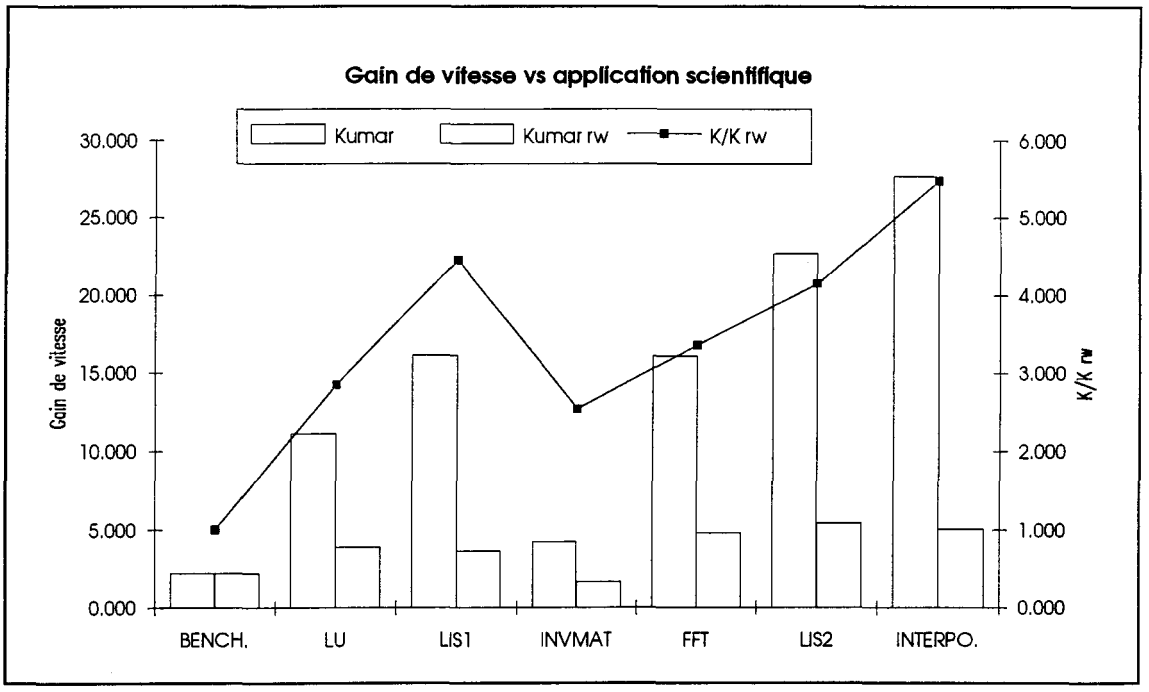

Figure 35 Parallélisme moyen en fonction du programme étudié.

Kumar différencie deux catégories de simulations, l'une tenant compte des écritures après lecture/écriture et l'autre non. Les résultats présentés dans [12] n'en tiennent pas compte. Évidemment, la première catégorie est plus restrictive que la deuxième pour les raisons énoncées dans la section 2.3.4 du chapitre II. OSASMO tient compte des écritures après lecture/écriture dans l'exécution des simulations et décompose l'application scientifique étudiée en code à trois adresses. Ceci nous amène donc à penser, intuitivement, que les résultats d'OSASMO seront moins optimistes et donc plus réalistes que ceux de Kumar [12].

La figure 35 représente les résultats obtenus des simulations selon les critères de Kumar. Chaque point de ces courbes est le résultat d'une simulation. Le temps requis pour réaliser une simulation varie de 15 minutes à 2 heures sur des stations de travail de type SPARCstation 1+™ et SPARCstation 2™ de SUN Microsystems™. La légende

accompagnant le graphe montre les différents types de simulations. Le premier type (Kumar) est celui ne tenant pas compte des coûts de communications, le second (Kumar rw) correspond aux simulations tenant compte des écritures après lecture/écriture. Le troisième type de la légende est simplement le ratio du premier type sur le second, donnant ainsi une comparaison des deux méthodes utilisées.

Une différence notable peut être observée concernant les résultats obtenus pour chacune des deux méthodes (figure 35). En effet, pour le programme d'interpolation (voir figure 35, INTERPO.), le rapport du potentiel parallèle d'une des méthodes par l'autre (K/K rw) donne environ 5,5. La première méthode donne des résultats trop optimistes. Ceux-ci reflétent peu les phénomènes réels, l'hypothèse des écritures après lecture/écriture s'avère donc indispensable à une modélisation fidèle d'un système parallèle.

La figure 35 présente quelques autres caractéristiques. On remarque que le banc d'essai (BENCH) ne possède qu'un faible potentiel parallèle dans les deux catégories. Le banc d'essai utilisé est celui de Whetstone. Après étude du code de Whetstone, il est apparu que toutes les lignes de code des boucles DO étaient constituées de variables scalaires. Ceci a pour effet de créer un engorgement dû aux dépendances des lignes d'une itération i à une itération i+1 lors d'une exécution parallèle et rend, par le fait même, difficile l'exécution horizontale (parallèle) de la boucle. Peu de potentiel de concurrence peut être extrait de ces boucles, ce qui explique les faibles performances parallèles de ce code. Cette remarque s'applique aussi à toutes les expérimentations utilisant ce code.

4.4 Simulation de l'exécution parallèle de dix applications scientifiques classiques

Les simulations réalisées dans cette section seront présentées sous forme de graphiques. Les essais sont réalisées avec trois architectures. Il s'agit de l'hypercube, de la grille et du bus. Chaque architecture a été simulée avec les dix applications scientifiques présentées au tableau 3. De plus, chaque application a été testée avec, premièrement, le coût d'une communication variable et un nombre de processeurs fixé à soixante-quatre et, deuxièmement, avec un nombre variable de processeurs et le coût d'une communication fixé à un cycle relatif. Plus loin dans la section, le pourcentage d'utilisation des processeurs sera présenté comme indice de performance.

4.4.1 Analyse de l'effet du coût d'une opération de communication

Cette sous-section met en évidence l'effet des communications sur les performances générales des systèmes parallèles. Les simulations ont été réalisées sur les dix applications scientifiques présentées plus tôt avec le coût d'une communication variant de zéro à dix cycles relatifs. Rappelons qu'un cycle relatif est défini comme étant égal au temps d'exécution d'une ligne de code à trois adresses. L'allocation des ressources a été réalisée de deux façons soit: aléatoire ligne-à-ligne et aléatoire par bloc de base, comme présenté à la section 1.4 du chapitre I. On a fixé à soixante-quatre le nombre de processeurs disponibles pour l'exécution parallèle. Les résultats d'un des dix programmes sont présentés dans cette section. Ceux des autres programmes sont fournis à l'annexe 4. En tout, chaque programme possède deux graphiques présentant des résultats liés au coût des communications. Ainsi, un total de vingt graphiques ont été obtenus pour les dix

programmes. L'inclusion de tous les graphiques dans le présent chapitre alourdirait inutilement ce dernier.

Les deux graphiques (figure 36 et figure 37) associés à un programme donné expriment, respectivement, le gain de vitesse moyen par cycle relatif en fonction du coût d'une opération de communication et le nombre moyen de communications par cycle relatif (trafic) en fonction du coût d'une opération de communication. Les coûts (pour une communication) considérés sont les suivants: 0, 1, 3, 5 et 10 cycles relatifs. Chaque graphique illustre les résultats obtenus en utilisant l'allocation aléatoire ligne-à-ligne et l'allocation aléatoire par bloc de base (voir Aho et Ullman [1]). Les figures 36 et 37 montrent ces deux graphiques pour le programme de décomposition en matrice LU (LU).

Notons que chaque point des graphiques des figures 36 et 37 représente le résultat d'une simulation. La légende de ce graphique donne tous les types d'architectures simulés et représentés sur le graphique. Les symboles sont définis comme suit: hyp=hypercube; bus=bus; gr=grille; r=allocation ligne-à-ligne; b= allocation par bloc. Les résultats de chaque simulation sont fournis sous forme d'histogrammes par OSASMO et doivent être analysés individuellement. L'obtention de chaque histogramme requiert en moyenne de 1 à 15 heures (en temps réel) d'exécution sur des stations de travail de type SPARCstation 1+™ et SPARCstation 2™ de SUN Microsystems™, selon la valeur du coût d'une opération de communication.

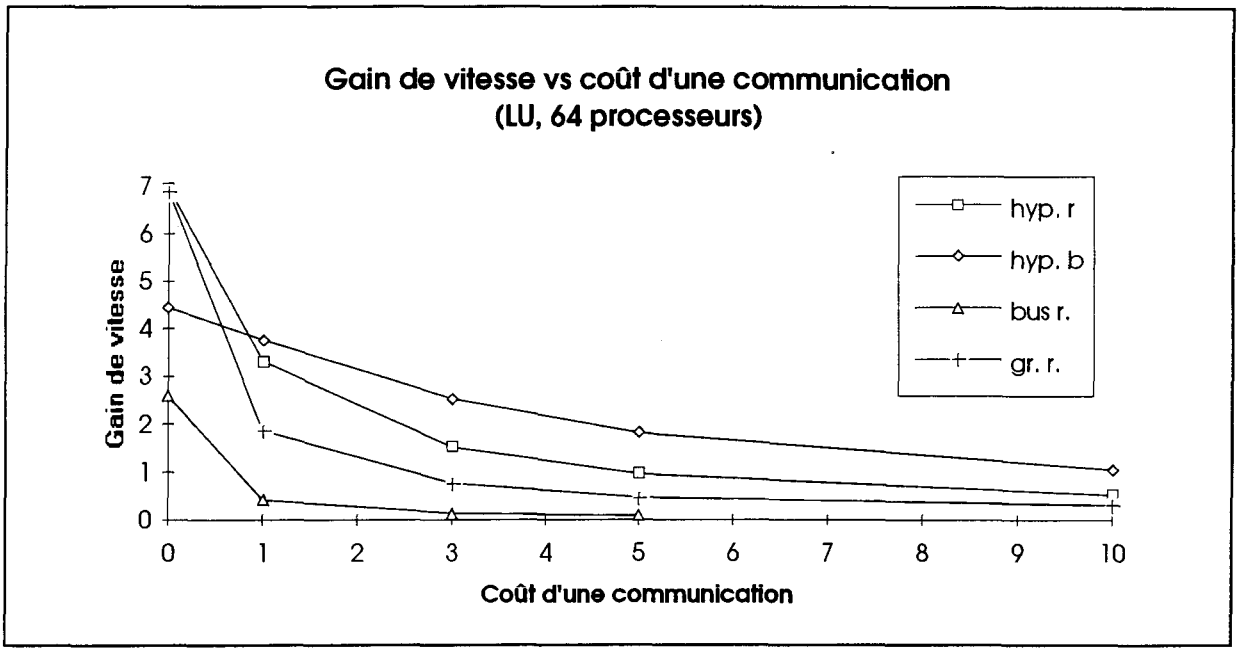

Figure 36 Exemple de graphique du gain de vitesse moyen par cycle relatif en fonction du coût d'une communication.

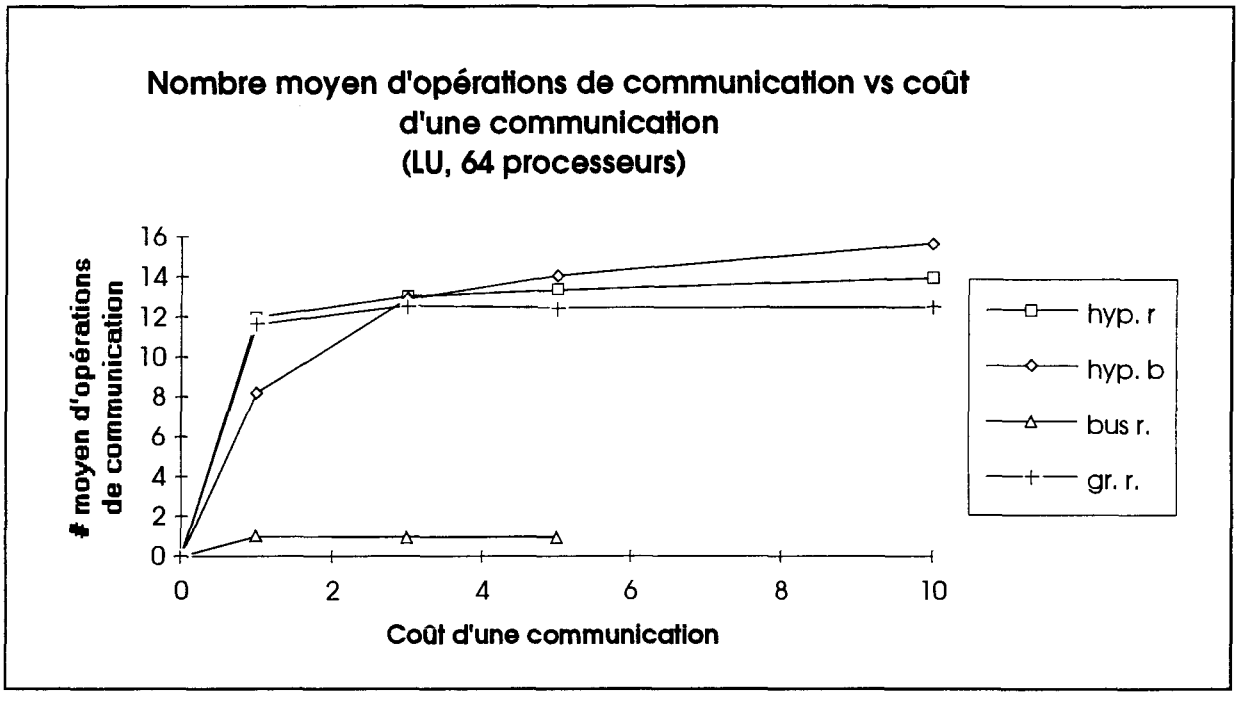

Figure 37 Exemple de graphique du nombre moyen de communications par cycle relatif en fonction du coût d'une communication.

Le graphique de la figure 36 illustre l'effet des communications sur le gain de vitesse. On se rend compte, sur cette figure, que la dégradation du gain de vitesse est quasi-exponentielle pour l'allocation aléatoire ligne-à-ligne des ressources de calculs. L'allocation par bloc de base présente une dégradation plus linéaire et moins rapide du gain de vitesse. Cette particularité se retrouve à travers toutes les applications scientifiques analysées dans ce travail. Elle est imputable au fait que l'allocation ligne-à-ligne peut potentiellement exécuter chaque ligne de code dans un processeur différent. Ceci demande une utilisation considérable du réseau de communication, ce qui a pour effet de donner les meilleures performances lorsque le coût d'une communication est nul mais, en contre-partie, la dégradation des performances est plus rapide que l'autre méthode d'allocation lorsque le coût d'une communication est non-nul.

L'hypercube et la grille sont les deux architectures ayant les performances les plus remarquables. Ceci est dû, en partie, à l'hypothèse #1b du chapitre III, qui stipule que la largeur de bande est de 1 pour une architecture en bus. Ceci implique qu'un seul message peut être communiqué à un instant donné sur le bus. Cet état provoque un engorgement lors des opérations de communication et détériore plus rapidement les performances de l'architecture en bus. Il serait certainement intéressant d'étudier une architecture incluant plusieurs bus pour diminuer l'effet des engorgements créés par la largeur de bande de 1.

Le graphique de la figure 37 présente le nombre moyen de communications par cycle relatif en fonction du coût d'une communication. L'architecture en bus fait cavalier seul avec un nombre moyen d'opérations de communication situé entre 0 et 1. Encore une fois, l'hypothèse #1b est la cause de ce comportement puisqu'un seul message ne peut êre communiqué à un instant donné sur le bus. Cette valeur doit donc être plus petite ou égale à un.

On remarque que l'allocation ligne-à-ligne surcharge le réseau pour des coûts de communication faibles (1 et 3) et que la tendance tend à diminuer pour des valeurs plus grandes (5 et 10). La situation contraire se produit pour l'allocation par bloc de base. L'explication de ce phénomène est simple. L'allocation par bloc de base découpant les programmes en blocs pouvant potentiellement être exécutés sur différent processeurs implique que les boucles DO sont exécutées de façon séquentielle. Ceci entraîne qu'une boucle DO est exécutée au complet dans le même processeur. Ceci génère des valeurs de gain de vitesse plus petites et par le fait même un nombre moyen de communications par cycle relatif plus faible pour l'allocation par bloc de base et inversement pour l'allocation ligne-à-ligne. Lorsque le coût d'une opération de communication augmente, l'allocation par bloc avec son nombre réduit d'opérations de communications n'est pas défavorisée puisque cela lui permet d'exploiter plus efficacement chaque processeur en ayant un ratio R/C plus grand que pour l'autre méthode d'allocation. Comme l'allocation ligne-à-ligne peut potentiellement exécuter une ligne par processeur, un grand nombre d'opérations de communication sont générés et puisque chaque communication est coûteuse, le potentiel parallèle de cette méthode d'allocation devient difficile à exploiter. Il en résulte une exécution échelonnée sur un plus large intervalle de temps créant ainsi moins de communications par cycle relatif. En se rapportant au graphique de la figure 37, pour des valeurs élevées de coût d'une communication (5 et 10), on remarque que l'allocation par bloc de base donne un meilleur gain de vitesse (figure 36) que l'autre méthode, car davantage de lignes de code sont exécutées par cycle relatif. Bon nombre de lignes de code ayant un contenu dépendant d'autres lignes sont exécutées sur un même processeur.

Afin de donner une vue globale en comparant les 10 applications scientifiques, chaque type d'architecture a été étudié séparément. Les résultats présentés dans les

graphiques 38, 39, 40 et 41 présentent les gains de vitesse en fonction du coût d'une communication, et ce, pour toutes les applications scientifiques testées.

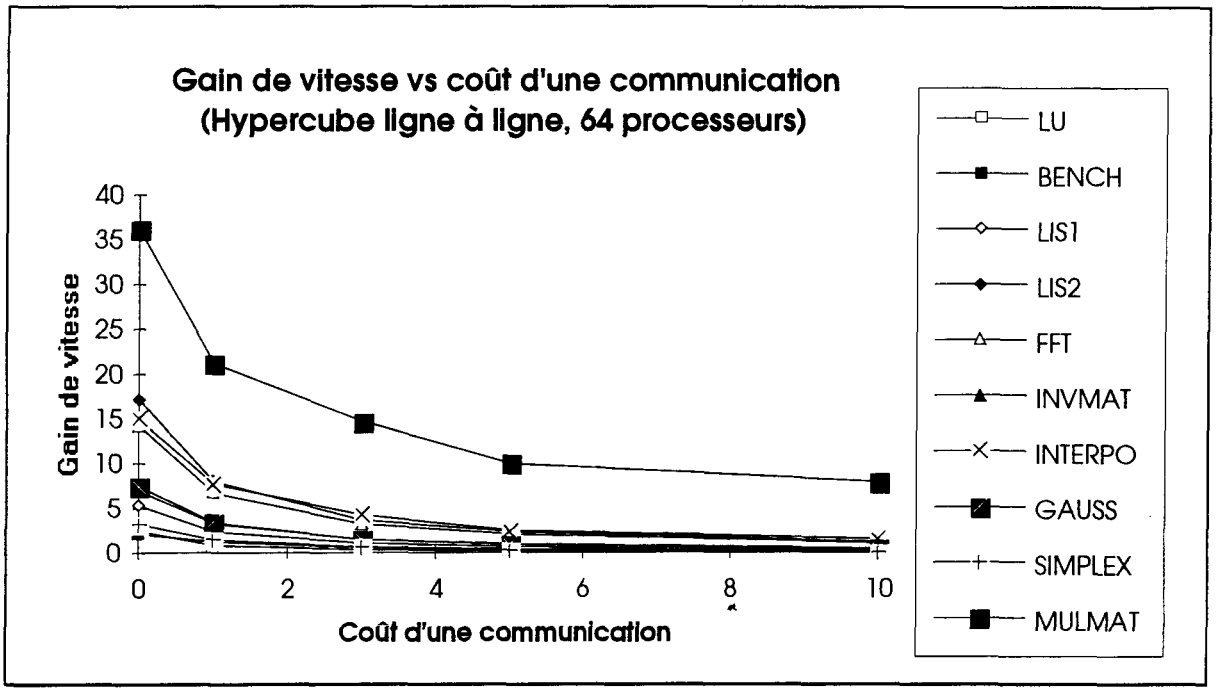

Figure 38 Graphique du gain de vitesse en fonction du coût d'une communication pour une architecture hypercube avec une allocation ligne-à-ligne et pour les 10 applications scientifiques testées.

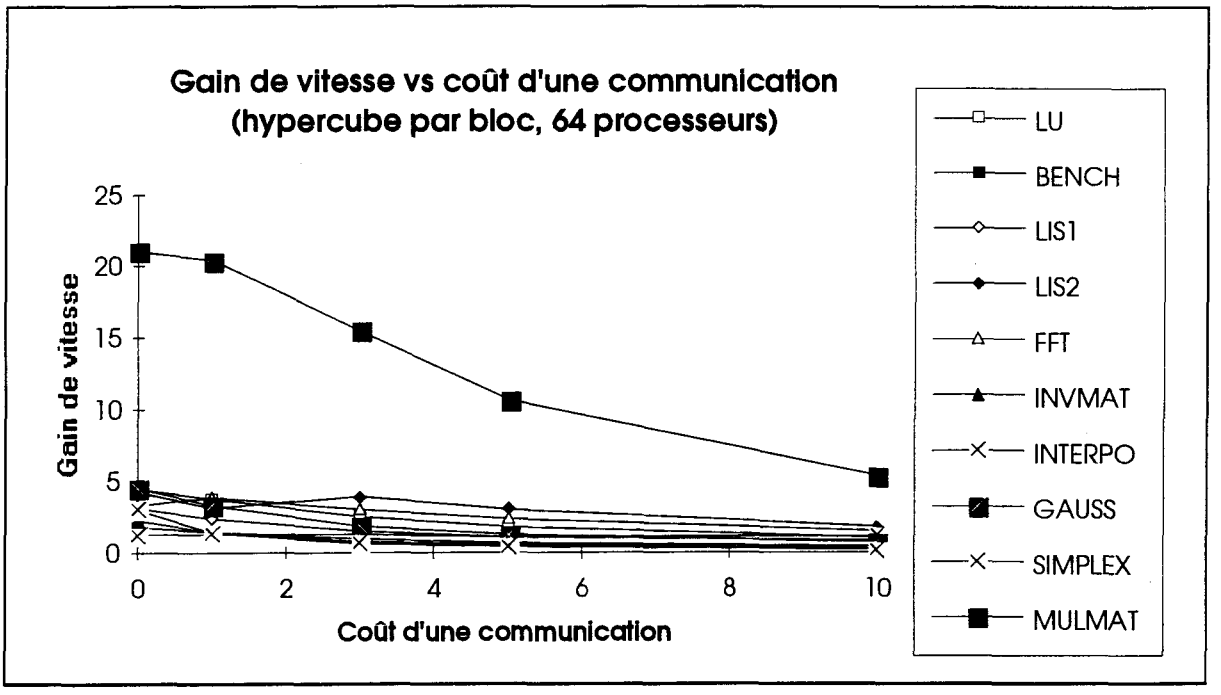

Figure 39 Graphique du gain de vitesse en fonction du coût d'une communication pour une architecture hypercube avec une allocation par bloc et pour les 10 applications scientifiques testées.

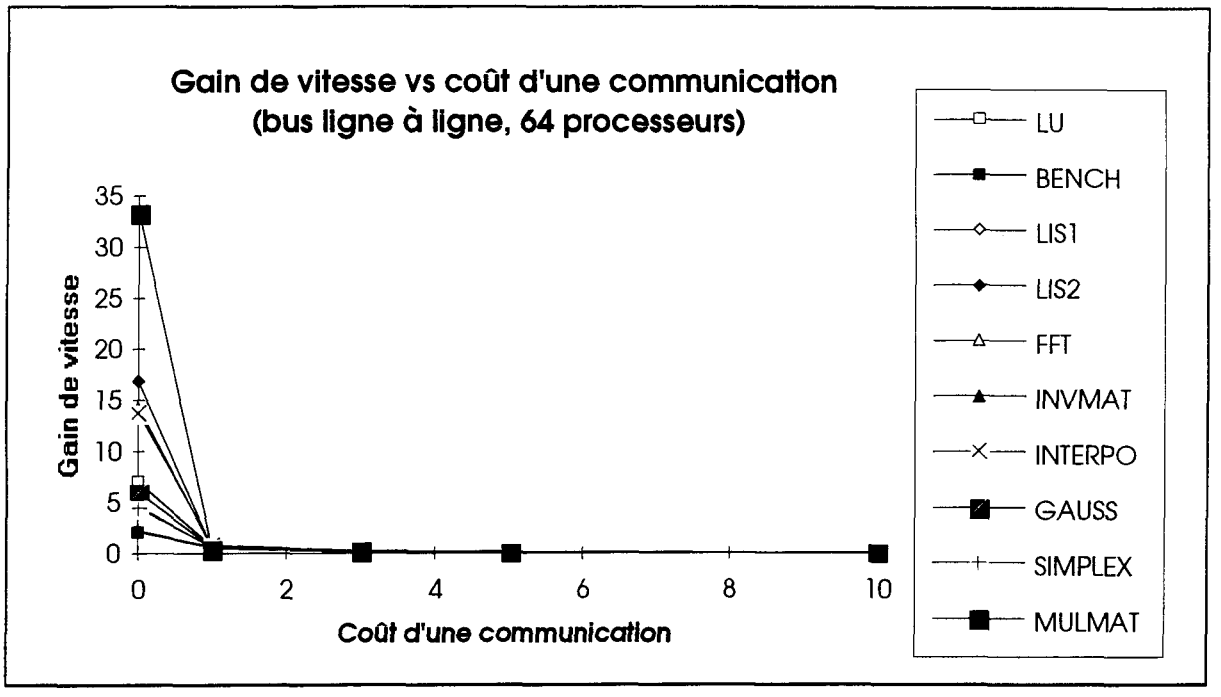

Figure 40 Graphique du gain de vitesse en fonction du coût d'une communication pour une architecture en bus simple avec une allocation ligne-à-ligne et pour les 10 applications scientifiques testées.

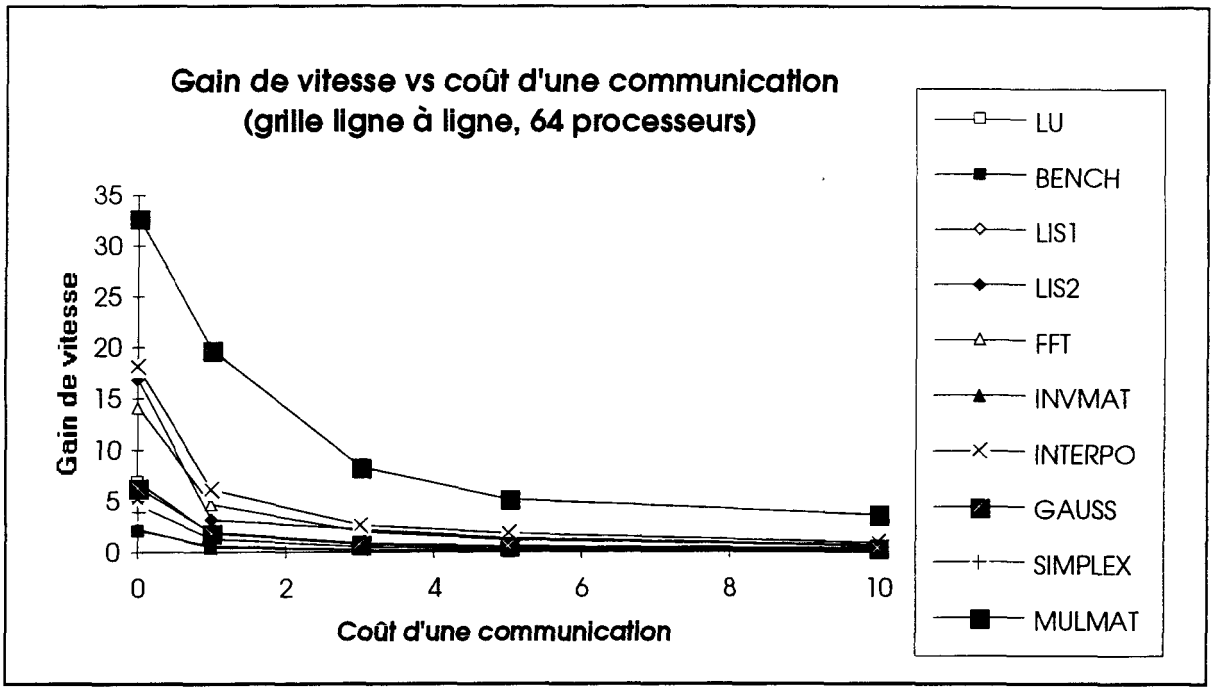

Figure 41 Graphique du gain de vitesse en fonction du coût d'une communication pour une architecture en grille avec une allocation ligne-à-ligne et pour les 10 applications scientifiques testées.

Les figures 38, 39, 40 et 41 présentent une vue incluant toutes les applications testées en fonction d'un type particulier d'architecture. Comme il est plus spécifiquement illustré pour l'application LU aux figures 36 et 37, le gain de vitesse des figures 38, 39, 40 et 41 se dégrade en fonction du coût des communications, et ce pour l'ensemble des applications testées. À noter, la dégradation particulière de l'architecture en bus. Cette dernière se dégrade de façon très accentuée lorsqu'un temps de communication plus grand que zéro est utilisé. Ce phénomène est normal compte tenu de la largeur de bande du canal de communication considéré pour ce type d'architecture. Le canal agit, en fait, comme un goulot d'étranglement.

4.4.2 Analyse de l'effet du nombre de processeurs

Cette sous-section met en évidence le fait qu'un très grand nombre de processeurs pour un système parallèle n'est pas nécessairement le meilleur choix à faire. De fait, les communications augmentent en rapport étroit avec le nombre de processeurs, comme mis en évidence dans l'exemple de la section 4.1. Les simulations ont ici été orientées sur l'analyse d'un nombre variable de processeurs. Les tests ont été effectués avec 2^n processeurs où $0 \leq n \leq 7$ processeurs. Les dix applications scientifiques présentées précédemment ont servi de base d'analyse. Le coût d'une opération de communication a été fixé à un cycle relatif.

La figure 42 illustre les résultats des simulations réalisées avec différents nombres de processeurs pour une architecture hypercube et une allocation aléatoire ligne-à-ligne des ressources de calcul. Les graphiques des autres architectures testées sont présentés à l'annexe 4 pour les mêmes raisons qu'évoquées précédemment.

Chaque point du graphique de la figure 42 a été constitué à partir d'une simulation correspondant aux paramètres des deux axes. Le temps d'exécution de chaque simulation variait encore une fois selon une large gamme de valeurs, soit de 30 minutes à 24 heures. Les calculs ont été éffectués sur des stations de travail SPARCstation 1+™ et SPARCstation 2™ de SUN Microsystems™.

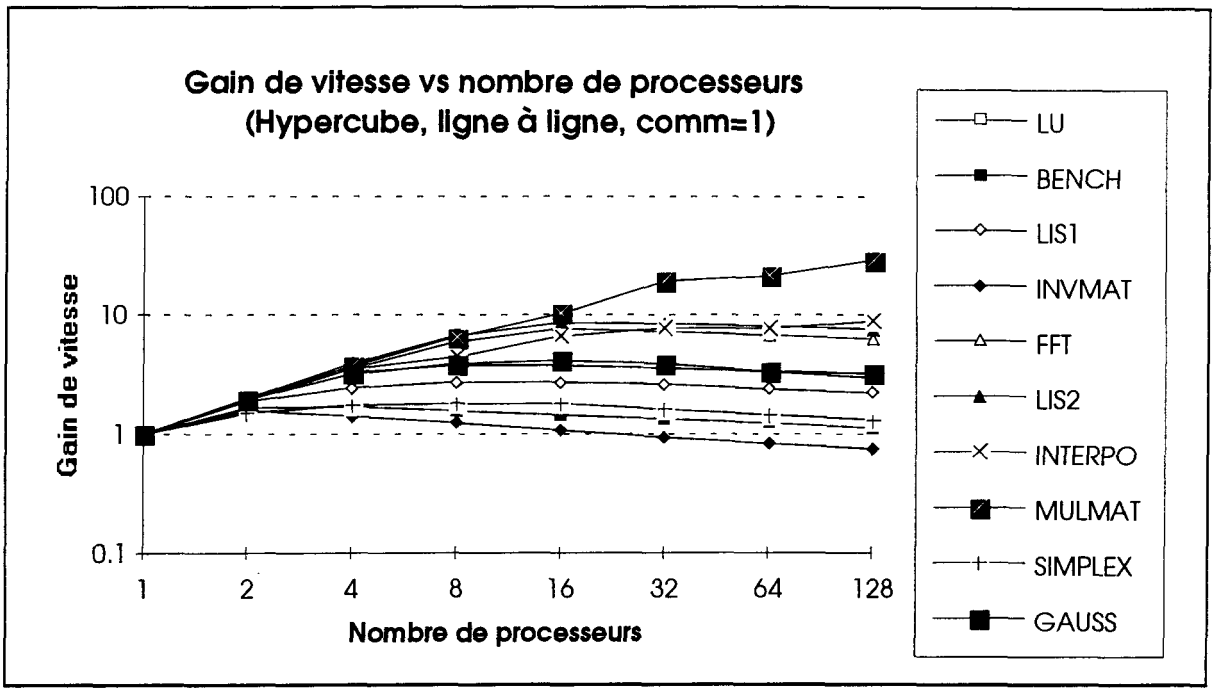

Figure 42 Graphique du gain de vitesse en fonction du nombre de processeurs.

On constate, dans la figure 42, que la majorité des programmes obtiennent leur gain de vitesse optimum à une valeur autre que la valeur maximale considérée du nombre de processeurs (i.e. 128). Fait à remarquer, le comportement du programme de banc d'esssai de Whetstone (BENCH) est en accord avec les commentaires formulés en début de chapitre, car deux processeurs constituent le nombre optimum pour l'exécution la plus efficace de cette application. D'après ce même graphique, l'algorithme d'inversion de matrices est particulièrement difficile à paralléliser ayant un optimum à deux processeurs et par la suite, toutes les autres valeurs du nombre de processeurs ne font que dégrader les performances. Ceci tend à confirmer que le parallélisme contenu dans les applications scientifiques est intimement lié aux arbres de dépendance du flot des données.

On relève, au graphique de la figure 42, un comportement qui semble être partagé par la majorité des applications scientifiques simulées: le nombre de processeurs requis

pour atteindre une exécution optimale du programme se situe en deça de 10 lorsque le coût d'une communication de 1 cycle relatif est comptabilisé. Ceci est une conclusion très importante, compte tenu du fait que plusieurs personnes s'orientent actuellement vers des systèmes parallèles possédant des quantités massives de processeurs. Pour des applications scientifiques typiques, il semble que peu de processeurs suffisent à les exécuter de façon optimale. Seuls les programmes de FFT (FFT, LISSAGE semi-paramétrique), l'interpolation polynômiale (INTERPO) et la multiplication de matrices obtiennent leur exécution optimale avec le nombre maximal de processeurs (128). L'analyse d'un code de FFT nous apprend, que cet algorithme est constitué de plusieurs boucles ayant un contenu peu dépendant. Il convient donc de dire que ce type d'algorithme possède un énorme potentiel de concurrence. Ce fait est connu et largement commenté dans la littérature spécialisée [9]. Hormis ces exemples, le peu de concurrence des applications scientifiques est explicable en partie en réalisant une simple étude du graphe de dépendance des données de certaines de ces applications. En effet, les lignes de code, entre elles, possèdent une très grande dépendance au niveau des données. Il est donc difficile d'effectuer un découpage du programme en bloc d'exécution, où chaque bloc peut être exécuté sur un processeur différent.

Le nombre optimal de processeurs utilisés est toujours plus grand ou égal à la valeur du gain de vitesse moyen. Ceci implique que le pourcentage moyen d'utilisation des processeurs est plus petit que 100% pour une exécution optimale de l'application scientifique. Difficilement prévisible à priori, le pourcentage moyen d'utilisation des processeurs est le sujet de la prochaine sous-section

4.4.3 Le pourcentage moyen d'utilisation des processeurs comme indice de performance

Les deux dernières sous-sections ont exprimé les résultats sous forme graphique, avec, comme indicateur de performance, le gain de vitesse moyen par cycle relatif. Les mêmes graphiques sont réalisés dans cette sous-section, mais en utilisant cette fois, comme indice de performance, le pourcentage moyen d'utilisation des processeurs par cycle relatif. Deux graphiques d'exemples sont donnés aux figures 43 et 44. Il s'agit, pour celui de la figure 43, du pourcentage moyen d'utilisation des processeurs en fonction du coût d'une communication. Ce graphe présente l'exécution du fichier de décomposition en matrice LU avec, comme indice de performance, le pourcentage d'utilisation des processeurs. La légende accompagnant ce graphe représente tous les types d'architectures simulées ainsi que le type d'allocation des processeurs (hyp = hypercube; bus=bus; gr = grille; l = allocation ligne-à-ligne; b = allocation par bloc). Le graphe de la figure 44 présente le pourcentage moyen d'utilisation des processeurs en fonction du nombre de processeurs, et ce, pour toutes les applications testées comme le montre la légende. Chaque symbole de la légende réfère à une application définie dans le tableau 3. Les autres résultats sont disponibles à l'annexe 4.

On calcule le pourcentage moyen d'utilisation des processeurs de la manière suivante:

$$\% \text{ d'utilisation moyen} = (\overline{G} \ / \ P) \ * \ 100 \qquad (7)$$

où $\overline{G}$ est le gain de vitesse

P est le nombre de processeurs

Le gain de vitesse, en fait, donne le nombre moyen de processeurs utilisés durant la simulation. En divisant ce nombre par le nombre total de processeurs dans le système simulé, on obtient l'utilisation moyenne des ressources disponibles.

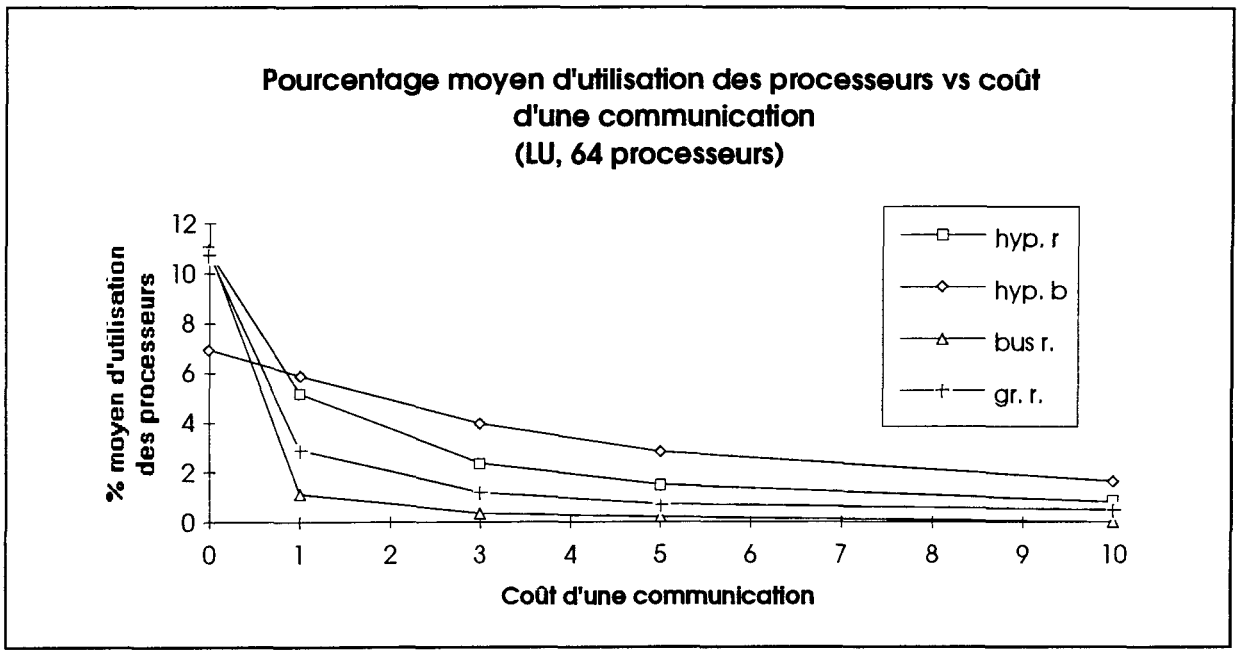

Figure 43 Pourcentage d'utilisation moyen par cycle relatif des processeurs en fonction du coût d'une communication.

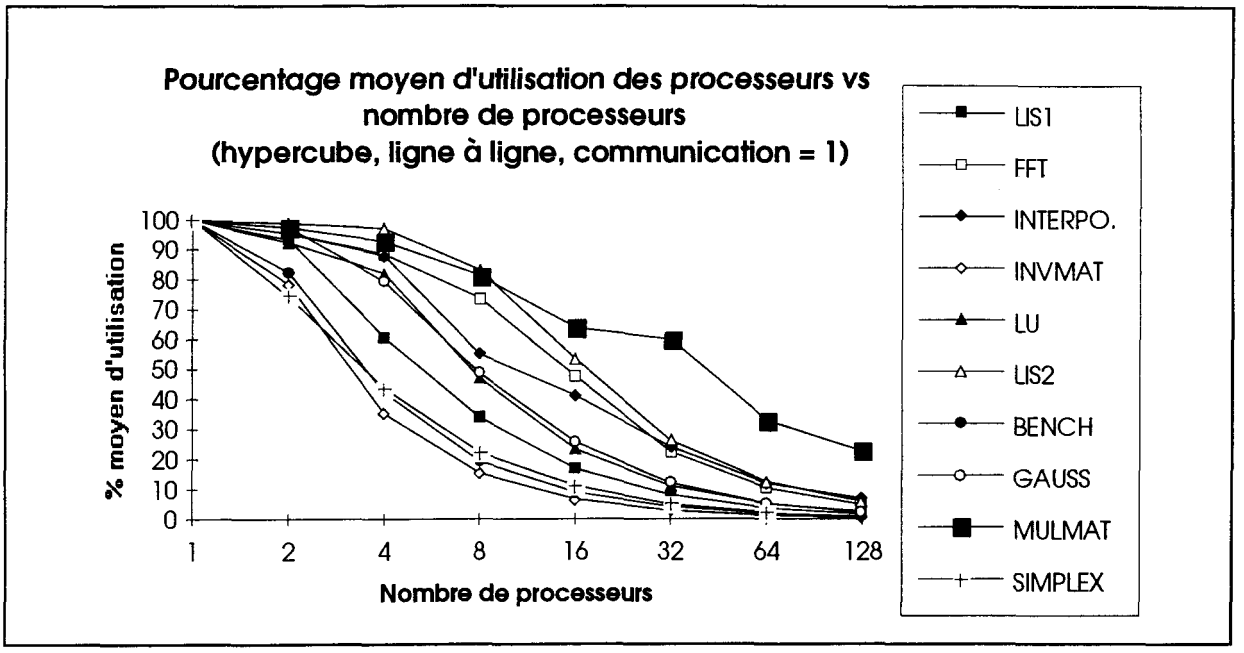

Figure 44 Pourcentage d'utilisation moyen par cycle relatif des processeurs en fonction du nombre de processeurs.

On note au graphique de la figure 43 que le pourcentage d'utilisation des processeurs en fonction du coût d'une communication, tel que déjà vu au paragraphe 4.4.2 est faible diminue rapidement en regard de l'augmentation de ce coût. En outre, on retrouve les mêmes patrons à travers toutes les courbes du pourcentage moyen d'utilisation. Les patrons sont similaires selon le type d'allocation. On remarque, notamment, que l'allocation par bloc possède un pourcentage moyen d'utilisation des processeurs plus uniforme, malgré l'acroissement des coûts de communication. Par contre, les résultats associés à l'allocation ligne-à-ligne des ressources décroîssent quasi-exponentiellement avec l'augmentation du coût d'une communication. Ce comportement est dû, en majeure partie, au fait que l'allocation aléatoire ligne-à-ligne peut potentiellement créer plus d'opérations de communication à chaque ligne de code exécutée, comme discuté précédemment. Les

variations des coûts de communication affectent donc dans une plus grande mesure ce type d'allocation des ressources.

On peut conclure en avançant que le pourcentage moyen d'utilisation des processeurs subit l'influence directe des variations des coûts de communication pour des allocations aléatoires ligne-à-ligne des ressources.

La figure 44 représente le pourcentage moyen d'utilisation des processeurs en fonction du nombre des processeurs, pour l'architecture hypercube avec une allocation des ressources par ligne. Le pourcentage moyen d'utilisation des processeurs dans le cas de cette figure soutient l'analyse de la figure précédente en démontrant qu'il décroît avec les communications et le nombre de processeurs. En effet, l'importance des communications augmente avec le nombre de processeurs. Et puisque l'augmentation du gain de vitesse en fonction du nombre de processeurs suit un comportement quasi-logarithmique, il est prévisible que le pourcentage d'utilisation des processeurs doit décroître en fonction des mêmes paramètres.

4.5 Détermination d'une métrique

L'analyse globale des résultats présentées dans ce chapitre n'est pas simple s'il faut tenir compte de tous les facteurs associés à l'exécution de programme en parallèle. En conséquence, il serait intéressant de définir un paramètre qui permettrait de pondérer les effets de ces facteurs. Pour ce faire, nous définirons donc une métrique.

Les systèmes parallèles actuels, qui sont pour la plupart multi-usagers, posent un problème de partage des ressources utilisables. Effectivement, aucun paramètre donnant l'utilisation optimale des ressources n'est défini. Un tel paramètre devrait tenir compte à la

fois de la performance profitable pour l'usager, soit le gain de vitesse, et celle profitable pour le propriétaire du système, soit le pourcentage maximal d'utilisation de chaque processeur.

Dans le but de trouver la valeur optimale du nombre de processeurs à utiliser pour l'exécution d'une application donnée, une métrique a été développée. Cette métrique est fonction du pourcentage d'utilisation (P) et du gain de vitesse (G):

$$\text{Métrique PG} = P * G \tag{8}$$

et en substituant (7) dans (8) on a:

$$\text{Métrique PG} = \frac{G^2}{\text{Nombre de processeurs}} \tag{9}$$

Le nom donné à cette métrique est *métrique PG*, en raison des paramètres dont elle est fonction. Dans le but d'en montrer l'utilité, un graphe de cette métrique en fonction du nombre de processeurs contenus dans un système parallèle est présenté à la figure 45. La légende présente les différentes applications scientifiques simulées à l'aide des noms de code du tableau 3. Comme pour les sous-sections précédentes, les graphiques des autres architectures se retrouvent à l'annexe 4.

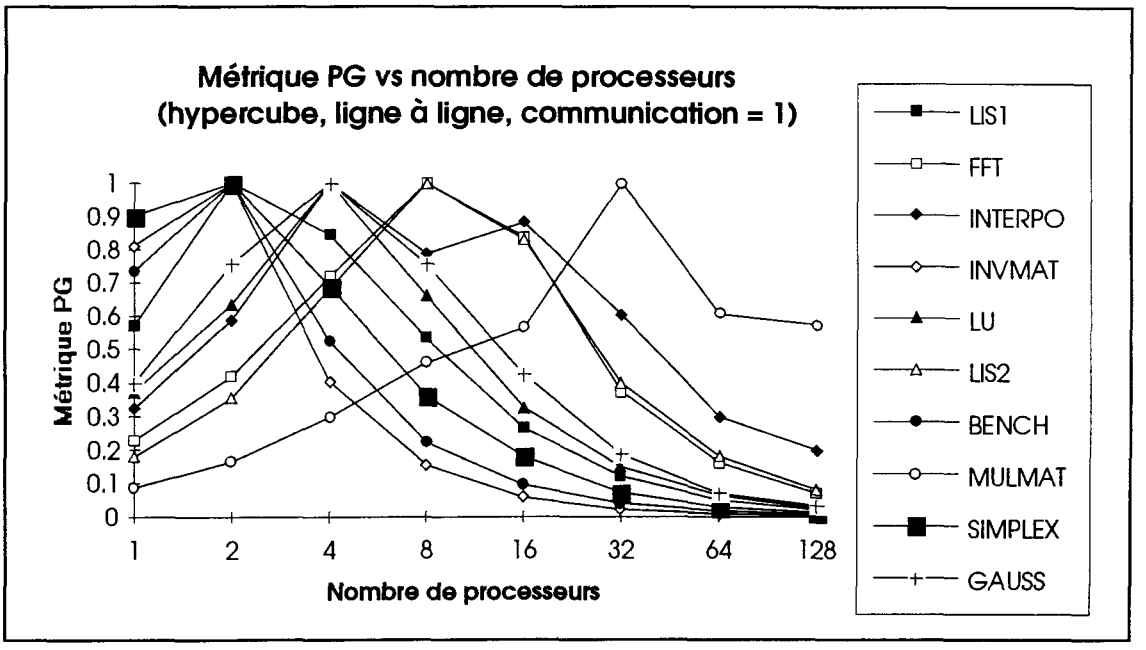

Figure 45 Graphique de la métrique PG en fonction du nombre de processeurs.

En comparant les trois graphes des figures 42, 44 et 45, on remarque, premièrement, que le gain de vitesse (figure 42) vise à utiliser une grande quantité de processeurs pour l'exécution la plus rapide d'une application scientifique, tandis que le pourcentage d'utilisation (figure 44), pour la même progression du nombre de processeurs, voit sa valeur diminuer quasi-exponentiellement. En bref, le pourcentage d'utilisation tend à diminuer la valeur de la métrique, alors que le gain de vitesse amène le phénomène inverse. Le comportement du produit de ces deux paramètres est présenté à la figure 45, où la métrique PG est exprimée en fonction du nombre de processeurs. Le maximum de chaque courbe exprime le nombre de processeurs à exploiter pour une utilisation optimale des ressources, eut égard aux usagers et au propriétaire du système, tout en ayant un gain de vitesse respectable pour l'exécution de l'application scientifique. Il est à noter que les valeurs de la figure 45 ont été normalisées pour améliorer la présentation.

4.6 Discussion

L'analyse globale des résultats, grâce à la métrique de la section précédente, est beaucoup plus simple à réaliser. En effet, cette métrique permet d'avoir une idée générale des performances optimales quant à l'exécution d'un programme scientifique sur un système parallèle, et amène une prise de décision rapide et efficace.

La quantification du potentiel de concurrence des applications scientifiques, en tenant compte des communications nous apprend, principalement, que les applications scientifiques ne possèdent généralement pas un grand potentiel parallèle. En fait les gains de vitesse enregistrés dans les simulations présentées dans ce chapitre ne dépassent pas le seuil de 36, et ce, dans des conditions idéales de fonctionnement (coût d'une communication nul).

Comme le chapitre IV le démontre, la communication entre les processeurs influence grandement les performances d'exécution parallèle des applications scientifiques. Bien que toutes les architectures simulées soient affectées par les coûts de communication, il n'en demeure pas moins qu'elles ne possèdent pas toutes la même vulnérabilité à ces coûts. D'après les résultats obtenus, en effet, l'architecture en hypercube semble être celle étant la moins affectée par les coûts de communication.

Le type d'allocation des ressources est un des facteurs qui, coordonné avec les coûts de communication, permet de contrôler dans une certaine mesure les pertes de performance. De fait, une allocation des ressources ligne-à-ligne permet un parallélisme très fin occasionnant un grand nombre d'échanges avec le réseau de communication. Pour un coût d'une communication qui est faible (1-3 cycles relatifs), l'exécution d'un progamme n'est pas gênée par ce type d'allocation, par contre, pour un coût d'une communication qui est

élevé (5-10 cycles relatifs), l'allocation ligne-à-ligne devient rapidement un handicap. L'allocation par bloc de base donne les résultats inverses. On a donc intérêt, pour des coûts de communication de base élevés, à utiliser une allocation par bloc de base.

Les applications contenant un grande quantité de boucle DO et ne contenant pas de scalaire dans la partie gauche des opérations d'assignation (ex.: A[I] = B[I] + C[I]), possèdent intrinsèquement un plus grand potentiel de concurrence. On pense notamment à la multiplication des matrices. Ce type d'application, grâce aux boucles DO, est plus commode à modifier pour une exécution parallèle et laisse une plus grande lattitude au programmeur pour mettre en valeur le potentiel parallèle.

Notons finalement que l'ensemble des résultats de ce chapitre, a demandé la simulation de plus de 700 différents fichiers comportant le moniteur inséré par OSASMO. Chacun des points des graphiques du chapitre IV et de l'annexe 4 est obtenu à la suite d'une procédure en quatre étapes suivantes:

1- Insérer le moniteur dans une application scientifique programmée en FORTRAN à l'aide d'OSASMO.
2- Compiler cette nouvelle application à l'aide d'un compilateur FORTRAN classique.
3- Exécuter cette application.
4- Traiter les résultats émanant de la simulation qui sont emmagasinés sous forme d'histogramme.

Conclusions

Tout au long de ce mémoire, nous avons démontré l'utilité d'un outil de simulation architecturale pour système multi-ordinateurs ayant la possibilité de simuler de façon simple des architectures parallèles et d'obtenir un ensemble de statistiques qu'il serait difficile d'obtenir autrement. L'outil développé, nommé OSASMO qui est l'acronyme d'Outil de Simulation Architecturale pour Système Multi-Ordinateurs, propose une méthode de simulation basée sur l'utilisation d'applications scientifiques déjà existantes et dédiées à une utilisation séquentielle. Les avantages de cette technique de simulation sont nombreux. On considère principalement le fait que l'utilisateur n'a besoin que de connaissances de base en architectures parallèles. Effectivement, il n'est pas nécessaire de connaître les particularités propres à des systèmes parallèles existants, correspondants aux architectures simulées. OSASMO n'est donc pas dépendant des caractéristiques intrinsèques de certains systèmes réels, ce qui évite les contraintes particulières imposées par certaines architectures. De plus, les entrées/sorties n'entrant pas en ligne de compte pour les simulations, il est possible d'extraire le potentiel parallèle réel d'une application scientifique. Par ailleurs, l'originalité du travail a déjà été reconnue. En effet, un poster [20] traitant de l'outil OSASMO a été présenté lors de la conférence SIAM (Society of Industrial and Applied Mathematics) ayant comme thème le traitement parallèle en mars 1993 à Norfolk, Virginie. Ce poster présentait les simulations concernant les machines parallèles idéales.

L'extraction des résultats expérimentaux est aussi assurée par OSASMO. La méthode utilisée par cet outil est d'implanter de façon automatique un moniteur directement dans des applications scientifiques FORTRAN. Ces nouvelles applications incluant le moniteur sont recompilées et exécutées. Le moniteur extrait des données lors de l'exécution. Ces données fournissent des statistiques de concurrence quant au parallélisme et à la communication résultant de l'exécution parallèle simulée de l'application.

Les applications utilisées pour réaliser les simulations sont des algorithmes scientifiques classiques largement utilisés par la communauté scientifique. On y trouve notamment une FFT, une décomposition en matrice LU, une résolution de matrice par la méthode de GAUSS, etc. Stone [17] a démontré théoriquement que le ratio R/C, où R est le temps pour exécuter une sous-tâche et C le temps de communication relié à cette sous-tâche, doit être relativement grand pour éviter la dégradation des performances causée par des pertes de temps significatives au niveau des communications. Les communications n'apportant rien de profitable, au niveau temporel, à l'exécution d'une application, il est important d'en minimiser les impacts. Par ailleurs, Geist et Sunderam [7] apportent la preuve expérimentale que, dans certaines conditions, l'exécution parallèle d'application scientifiques peut dégrader les performances (gain de vitesse < 1) en comparaison avec un système séquentiel (gain de vitesse $= 1$). Les résultats des simulations obtenus avec OSASMO montrent, elles aussi, qu'il n'est pas toujours profitable d'exécuter de façon parallèle les applications scientifiques. La communication en est la principale responsable. En effet, le gain de vitesse diminue quasi-exponentiellement en fonction du coût des communications. Des coûts de communications trop élevés peuvent, à eux seuls, amener à la conclusion qu'il n'est pas profitable de paralléliser l'algorithme.

Les résultats obtenus au chapitre IV montrent clairement que des coûts de communication très élevés dégradent le gain de vitesse des systèmes parallèles quasi-exponentiellement . Il convient donc de dire qu'il existe une plage d'utilisation du rapport R/C que l'on doit respecter. Effectivement, le rapport R/C doit être suffisament grand pour ne pas que la communication handicape l'exécution du programme. Aussi le rapport R/C doit être suffisament petit pour éviter que le programme soit exécuté de façon séquentielle. Donc, la segmentation de l'application scientifique en sous-tâches est une étape cruciale de la parallélisation d'un algorithme. Il s'agit, en fait, de trouver l'équilibre entre utiliser le maximum de processeurs et générer le minimum de communications. Jusqu'à maintenant, aucune méthode formelle n'existe pour quantifier cet équilibre. De plus, les limites d'une méthode d'analyse statique des caractéristiques R/C d'une application sont rapidement atteintes. Ceci nous mène aux méthodes dynamiques d'extraction des paramètres parallèles, qu'OSASMO utilise. Ce type de méthode est une des plus efficaces pour extraire les paramètres qui mène à une utilisation optimum d'un type d'architecture parallèle.

Il a été montré que le gain de vitesse en fonction du nombre de processeurs n'a pas une croissance linéaire. D'autre part, le pourcentage d'utilisation des processeurs diminue exponentiellement en fonction du nombre de processeurs. Ceci implique qu'un compromis entre le gain de vitesse et le pourcentage d'utilisation peut être obtenu afin d'utiliser efficacement les ressources parallèles disponibles. Les communications sont considérées par l'entremise du pourcentage d'utilisation des processeurs. Pour obtenir un certain optimum, une métrique a été développée. Cette métrique tient compte du pourcentage d'utilisation des processeurs (P) et du gain de vitesse (G) et se nomme *métrique PG*. La courbe de la *métrique PG* en fonction du nombre de processeurs, pour chaque application scientifique analysée, présente un maximum. Le nombre de processeurs associé à ce

maximum donne l'information qui permet l'utilisation optimale des ressources parallèles disponibles.

Après l'analyse globale des résultats des simulations, il est apparu que les applications scientifiques ne possèdent généralement pas un grand potentiel de concurrence. En effet, la majorité des gains de vitesses ne dépassent pas 18, et ce, dans des conditions idéales d'exécution (coût de communication nul). Il faut donc analyser avec circonspection les avantages d'exécuter en parallèle des applications scientifiques. Au préalable, il est avantageux de programmer le plus modulairement possible ces applications. De plus, si tous les modules sont indépendants entre eux, il sera plus simple d'analyser l'algorithme et de le diviser en sous-tâches. La parallélisation d'un algorithme n'est pas un processus qui débute après la programmation séquentielle du dit algorithme, mais bien dès sa conception.

Un regard sur la possibilité de faire la détection automatique du parallélisme contenu à l'intérieur d'un programme serait intéressant pour des travaux futurs. Cet exercice permettrait de prendre des décisions rapides quant aux avantages ou aux inconvénients de paralléliser une application. Sur la base d'une utilisation multi-ordinateurs, un outil semblable pourrait même détecter, à l'intérieur du programme, où il est avantageux de paralléliser. Dans un premier temps, l'outil devrait faire une étude statique, et non dynamique, pour diminuer le temps de décision. Puisque le parallélisme se retrouve majoritairement dans les boucles DO, cette étude statique pourrait être une analyse des dépendances à l'intérieur de ce type de boucle, ce qui donnerait une information suffisante pour prendre une décision de premier ordre sur le potentiel parallèle d'une application.

Références bibliographiques

[1] A. V. Aho, *et al.*, Compilers Principles, Techniques, and Tools, Addison Wesley, Reading, Ma., 1986.

[2] A. V. Aho, S. C. Johnson, LR Parsing, *Computing Surveys*, vol. 6, no. 2, June 1974, p.99-124.

[3] T.S. Axelrod, Comparing the Performance of Parallel Computers Extended Abstract, *Spring COMPCON 85': Thirtieth IEEE computer society international conference*, 1985, p.398-401.

[4] J. Dongarra, *et al.*, Computer benchmarking: paths and pitfalls, *IEEE Spectrum*, Juillet 1987, p. 38-43.

[5] J. Dongarra, et E. Grosse, Distribution of mathematical software via electronic mail, *ACM*, 1987, vol. 30, p. 403-407

[6] H. Fromm, *et al.*, Experiences with Performance Measurement and Modeling of a Processor Array, *IEEE Trans. on Computers*, Jan 1983, vol. C-32, no 1, p.15-31.

[7] G. A. Geist et V. S. Sunderam, Network based de concurrence computing on the PVM system, rapport technique no ORNL/TM-11826, Oak Ridge 1991.

[8] U. Herzog, Performance Evaluation Principles for Vector- and Multiprocessor Systems, *Parallel computing*, 7, 1988, p.425-438.

[9] R. W. Hockney et C. R. Jesshope, Parallel computers : architectures, programming and algorithms, A. Hilger, Bristol, England, 1983.

[10] K. Hwang, et F. A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, N. Y., 1984.

[11] A. Kelly, et I. Pohl, A book in C, Programming in C, Second Edition, Benjamin Cummings Publishing Company, inc., Redwood City, Ca., 1990.

[12] M. Kumar, Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications, *IEEE Trans. on Computers*, Sept. 1988, vol. 37, no. 9, p.1088-1098.

[13] M. Kumar, Effect of Storage Allocation/Reclamation Methods on Parallelism and Storage Requirements, *14th Annual International Symposium on Computer Architecture*, 1987, p.197-205.

[14] S. P. Levitan, Evaluation Criteria for Communication Structures in Parallel Architectures, *Proceedings of the 1985 International Conference on Parallel Processing*, 1985, p.147-154.

[15] W. H. Press *et al.*, Numerical recipes in C, the art of scientific computing, Cambridge University press, Cambridge, NY, 1988.

[16] B. Qin, *et al.*, Micro Time Cost of Parallel Computations, *IEEE Trans. on Computers,* May 5th, 1991, vol. 40, p.613-628.

[17] H. S. Stone, High-performance computer architecture, Addison-Wesley, Reading, Ma., 1990.

[18] C. J. Wang, V. P. Nelson, C. H. Wu, Performance modeling of the modified mesh-connected parallel computer, *The 9th International Conference on Distributed Computing Systems*, 1989, p.490-497.

[19] P. Zeppenfeld, *et al.*, On manipule même les atomes, *LA RECHERCHE*, vol. 23, 241, mars 1992, p.360-362.

[20] E. Chabot et D. Audet, Estimating Optimum Parallelism in Standard Scientific Applications, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, vol. 2, 1993, p.1115-1118.

Annexe 1

Définitions des variables, fonctions et sous-routines du moniteur inséré par OSASMO

Définition des variables, fonctions et sous-routines du moniteur inséré par OSASMO

La légende explicant chaque catégorie de ce tableau est donnée à la fin de ce dernier.

catégories	Entité	Définition
M-I-H	KSTAT(3, *profondeur*)	Regroupe les données de concurrence et de communication sous forme d'histogramme.
F-I-H	KEC_SEARCH(KPROC, *K_Wtype*, KTPROC, LAST, NB_PROC, PROF)	Vérifie si le processeur KPROC est occupé pour la valeur de temps *K_Wtype*.
S-H	IN_PARA(K_STAT, *K_Wtype*, PROF)	Insère la valeur de temps K_Wtype, de parallélisme, dans KSTAT.
S-H	IN_COM(K_STAT, KCOM, *K_Wtype*, PROF)	Insère les valeurs de temps de communication contenu dans KCOM dans KSTAT.
V-I-H	*K_Wtype*	Contient la dernière valeur de temps où la variable homologue a été écrite.
V-I-H	*K_Rtype*	Contient la dernière valeur où la variable homologue a été lue.
V-I-H	*K_Ltype*	Contient tous les processeurs où la variable a été lue depuis la dernière écriture.
M-I-H	KCOM(2, 31)	Contient toutes les opérations de communication de l'exécution d'une ligne de code source étudiée.
M-I-DA	KTPROC(*nb_processeur*, *profondeur*)	Contient l'occupation de tous les processeurs à tous les instants de la simulation en unité de temps relatif.
V-I-DA	KPROC	Contient le numéro de processeur dans lequel la ligne de code étudié est exécutée.

F-I-DA	Kec_proc(arguments variables) Les arguments dépendent du type d'allocation de processeur.	Donne le numéro de processeur dans lequel la ligne de code sera exécutée. Peut faire différents types d'allocation.
V-I-DA	PROF	Contient la durée de la simulation en unité de temps relatif.
V-I-DA	NB_PROC	Contient le nombre de processeurs que contient l'architecture.
V-I-DA	MULT_COM	Contient le nombre d'unités de temps relatif que prend une opération de communication de base.
V-I-CTM	KTIME$	Contient le temps minimum auquel on peut conclure qu'une partie de programme peut être exécutée.
F-I-CTM	max(MULT_COM, NB_PROC, KTIME$, KCOM, *K_WtypeG*, *K_RtypeG*, KPROC, *K_Wtype1*, *K_Ltype1*, ..., *K_Wtype30*, *K_Ltype30*)	Rafraîchit la variable ou le vecteur synthétique de type *K_WtypeG* se trouvant à gauche de l'égalité. Sert pour les assignations et les boucles DO.
F-I-CTM	maxfct(MULT_COM, NB_PROC, KTIME$, KCOM, KPROC, *K_Wtype1*, *K_Ltype1*, ..., *K_Wtype30*, *K_Ltype30*)	Rafraîchit la variable ou le vecteur synthétique pour les fonctions défini par l'usager ainsi que les fonctions intrinsèques du fortran. Cette fonction est aussi utilisée pour les lignes n'ayant pas d'égalité et de variable à gauche de l'égalité (ex.: IF, DOWHILE).
F-I-CTM	maxtmp1(MULT_COM, NB_PROC, KTIME$, KCOM, KPROC, *K_Wtype1*, *K_Ltype1*, ..., *K_Wtype5*, *K_Ltype5*)	Rafraîchit la variable synthétique de type *K_Wtype,* où *type* est *TMPtype,* se trouvant à gauche de l'égalité dans la situation où une variables se trouve à droite de l'égalité.
F-I-CTM	maxtmp2(MULT_COM, NB_PROC, KTIME$, KCOM, KPROC, *K_Wtype1*, *K_Ltype1*, ..., *K_Wtype5*, *K_Ltype5*)	Rafraîchit la variable synthétique de type *K_Wtype,* où *type* est *TMPtype,* se trouvant à gauche de l'égalité dans la situation où deux variables se trouvent à droite de l'égalité.

Légende des catégories

Catégorie	Définition
CTM	Entités se rapportant au calcul du temps minimum.
DA	Définition de l'architecture simulée.
F	Fonction
H	Entités se rapportant à l'histogramme.
I	Integer (entier)
M	Matrice
S	Sous-routine
V	Variable

Annexe 2

Exemples de validation de l'outil de simulation architecturale

Exemple de validation # 1

Voici le code source de l'exemple # 1

```
      PROGRAM ERIC_C1
C
      INTEGER A,B
C
      A=1
      B=2
C
      DO I=1,5
          IF(A.LT.B)THEN
              B=A
          ELSE
              A=B
          ENDIF
      ENDDO
      END
```

La trace d'exécution de l'exemple #1 avec le moniteur inséré

```
      A=1 ; [P: 0, T: 2]
      B=2 ; [P: 6, T: 2]
                                          DO I =1,5
                                              IF(A.LT.B)THEN
IF:T = 23; P: 0 | P: 0; T: 2, P: 6; T: 2 (COC 20)
                                                  B=A
P: 2; T: 34 = P: 0; T: 2; TBI: 23 (COC 10)
                                              ENDIF
                                          DO I =1,5
                                              ELSE
ELSE:T = 55; P: 7 | P: 0; T: 2, P: 2; T:34 (COC 30)
                                                  A=B
P: 0; T: 66 = P: 2; T: 34; TBI: 55 (COC 10)
                                              ENDIF
                                          DO I =1,5
                                              ELSE
```

```
ELSE:T = 87; P: 3 | P:0; T: 66, P: 2; T: 34 (COC 20)
                                                    A=B
P: 6; T: 98 = P: 2; T: 34 TBI: 87 (COC 10)
                                                  ENDIF
                                              DO I =1,5
                                                  ELSE
ELSE:T = 119; P: 5 | P: 6; T: 98, P: 2; T: 34 (COC 30)
                                                    A=B
P: 3; T: 120 = P: 2 ;T: 34 TBI: 119 (COC 0)
                                                  ENDIF
                                              DO I =1,5
                                                  ELSE
ELSE:T = 141, P: 5 | P: 3; T: 120, P: 2; T: 34 (COC 20)
                                                    A=B
P: 5;T: 142 = P: 2 ;T: 34 TBI: 141(COC 0)
                                                  ENDIF
                                              ENDDO
```

Temps d'exécution du fichier de test #1 réalisé par OSASMO: 142 cycles relatifs

Exemple de validation # 2

Voici le code source de l'exemple # 2

```
      PROGRAM ERIC_C2
C
      INTEGER A,B,I
C
      A=1
      B=3
C
      IF(A.LT.B)THEN
            A = A + 1
            DO I=1,3
                  B = ERIC(B)
            ENDDO
      ENDIF
      END
C
      FUNCTION ERIC(B)
      INTEGER B,ERIC
C
      ERIC = B + 1
C
      RETURN
      END
```

La trace de l'exécution de l'exemple # 2 avec le moniteur inséré

```
      A=1 ; [P: 0, T: 2]
      B=3 ; [P: 4, T: 2]
                                          IF(A.LT.B)THEN
IF:T = 23, P: 6 | P: 0; T: 2, P: 4; T:2 (COC 20)
                                                A=A+1
P: 5; T: 44 = P: 0; T: 2 TBI: 23 (COC 20)
                                                DO I =1,3
                                                      ERIC=B+1
P: 1; T: 44 = P: 4; T: 2 TBI: 23 (COC 20)
                                                      B=ERIC
P: 0; T: 55 = P: 1; T: 44 TBI: 23(COC 10)
```

```
                                                    DO I =1,3
                                                          ERIC=B+1
P: 7;T: 86 = P: 0; T: 55 TBI: 23 (COC 30)
                                                          B=ERIC
P: 3; T: 97 = P: 7; T: 86 TBI: 23 (COC 20)
                                                    DO I =1,3
                                                          ERIC=B+1
P: 5; T: 118 = P: 3; T: 97 TBI: 23 (COC 20)
                                                          B=ERIC
P: 4;T: 129 = P: 5 ;T: 118 TBI: 118 (COC 30)
                                                    ENDDO
                                              ENDIF
```

Temps d'exécution du fichier de test #2 réalisé par OSASMO: 129 cycles relatifs

Exemple de validation # 3

Voici le code source de l'exemple # 3

```
      PROGRAM ERIC_C3
C
      INTEGER A, B
C
      A = 1
      B = 3
C
      DO WHILE(A.LT.B)
            A = A+1
            IF(A.GE.B)THEN
                  B = B-1
            ENDIF
      ENDDO
      END
```

La trace d'exécution de l'exemple # 3 avec le moniteur inséré

```
      A = 1 ; [P: 3, T: 2]
      B = 3 ; [P: 6, T: 2]
                                                DO WHILE(A.LT.B)
DOW:T = 13; P: 2 I P: 3; T: 2, P: 6; T: 2 (COC 10)
                                                      A=A+1
P: 7; T: 24 = P: 3; T: 2 TBI: 13 (COC 10)
                                                      IF(A.GE.B)THEN
IF:T = 35; P: 3 I P: 7; T: 24, P: 6; T: 2 TBI: 13 (COC 20)
                                                      ENDIF
                                                DO WHILE(A.LT.B)
DOW:T = 35; P: 5 I P: 7; T: 24, P: 6; T: 2 (COC 20)
                                                            A=A+1
P: 4; T: 56 = P: 7; T: 24 TBI: 35 (COC 20)
                                                      IF(A.GE.B)THEN
IF:T = 67; P: 6 I P: 4; T: 56, P: 6; T: 2 TBI: 35 (COC 10)
                                                            B=B-1
P: 2;T: 78 = P: 6 ;T: 2 TBI: 67 (COC 10)
                                                      ENDIF
```

ENDDO

Temps d'exécution du fichier de test #3 réalisé par OSASMO: 78 cycles relatifs

Exemple de validation # 4

Voici le code source de l'exemple # 4

```
      PROGRAM ERIC_C4
C
      INTEGER A,I,J
C
      A=1
C
      DO I=1,3
            DO J=1,3
                  A=A+1
            ENDDO
      ENDDO
      END
```

La trace d'exécution de l'exemple # 4 avec le moniteur inséré

```
      A=1 ; [P: 0, T: 2]
                                          DO I =1,3
                                                DO J =1,3
                                                      A=A+1
P: 4; T: 13 = P: 0; T: 2 (COC 10)
                                                DO J =1,3
                                                      A=A+1
P: 7; T: 34 = P: 4; T: 13 (COC 20)
                                                DO J =1,3
                                                      A=A+1
P: 7; T: 35 = P: 7; T: 34 (COC 0)
                                                ENDDO
                                          DO I =1,3
                                                DO J =1,3
                                                      A=A+1
P: 2; T: 56 = P: 7; T: 35 (COC 20)
                                                DO J =1,3
                                                      A=A+1
P: 0; T: 67 = P: 2; T: 56 (COC 10)
                                                DO J =1,3
                                                      A=A+1
```

P: 3; T: 88 = P: 0; T: 67 (COC 20)

```
                ENDDO
            DO I =1,3
                DO J =1,3
                    A=A+1
```

P: 4; T: 119 = P: 3; T: 88 (COC 30)

```
                DO J =1,3
                    A=A+1
```

P: 7; T: 140 = P: 4; T: 119 (COC 20)

```
                DO J =1,3
                    A=A+1
```

P: 3; T: 151 = P: 7; T: 140 (COC 10)

```
                ENDDO
            ENDDO
```

Temps d'exécution du fichier de test #4 réalisé par OSASMO: 151 cycles relatifs

Exemple de validation # 5

Voici le code source de l'exemple # 5

```
      PROGRAM ERIC_C5
C
      INTEGER A,B
C
      A=1
      B=2
C
      IF(A*7.LT.6+B)THEN
            A=B*9
      ELSE
            B=B-A
      ENDIF
      END
```

La trace de l'exécution de l'exemple # 5 avec le moniteur inséré

```
      A=1 ; [P: 0, T: 2]
      B=2 ; [P: 6, T: 2]
                                          TMP1=A*7
P: 1; T: 13 = P: 0; T: 2 (COC 10)
                                          TMP2=6+B
P: 1; T: 33 = P: 6; T: 2 (COC 30)
                                          IF(TMP1.LT.TMP2)THEN
IF:T = 34; P: 1 | P: 1; T: 13, P: 1; T: 33 (COC 0)
                                                A=B*9
P: 6; T: 35 = P: 6; T: 2 TBI: 34 (COC 0)
                                          ENDIF
```

Temps d'exécution du fichier de test #5 réalisé par OSASMO: 35 cycles relatifs

Exemple de validation # 6

Voici le code source de l'exemple # 6

```
      PROGRAM ERIC_C6
C
      INTEGER A,B
C
      A=1
      B=2
C
      DO I=1,3
           IF(A.GT.B)GOTO 10
                A = A+1
           GOTO 20
 10              B = B+1
 20    ENDDO
      END
```

La trace d'exécution de l'exemple # 6 avec le moniteur inséré

```
      A=1 ; [P: 0, T: 2]
      B=2 ; [P: 3, T: 2]
                                          DO I =1,3
                                               IF(A.GT.B) GOTO 10
IF:T = 23; P: 0 | P: 0; T: 2, P: 3; T:2 (COC 20)
                                                    A=A+1
P: 7; T: 54 = P: 0; T: 2 TBI: 23 (COC 30)
                                          DO I =1,3
                                               IF(A.GT.B) GOTO 10
IF:T = 85; P: 0 | P: 7; T:54, P: 3; T: 2 (COC 30)
                                                    A=A+1
P: 6;T: 96 = P: 7; T: 54 TBI: 85 (COC 10)
                                          DO I =1,3
                                               IF(A.GT.B) GOTO 10
IF:T = 117; P: 5 | P: 6; T:96, P: 3; T: 2 (COC 20)
                                                    B=B+1
P: 5; T: 118 = P: 5; T: 2 TBI: 117(COC 0)
                                          ENDDO
```

Temps d'exécution du fichier de test #6 réalisé par OSASMO: 118cycles relatifs

Annexe 3

Fichiers sources d'applications FORTRAN non-disponibles dans la littérature

Fichier source du programme de lissage 1

```
      REAL X(9),Y(9),SY(9),A(4),DELTAA(4)
      REAL YFIT(9),CHISQR,SIGMAA(4)
      REAL FCHI,SCHI
      INTEGER I,NPTS,NCOEF,NITER
C
C  Ne jamais changer ces trois valeurs
C
      FCHI=0.
      SCHI=0.
      NITER=0
C
C  Parametres du lissage (a modifier selon le cas)
C
      NPTS=8
      NCOEF=2
C
      X(1)=2
      X(2)=3
      X(3)=5
      X(4)=7
      X(5)=10
      X(6)=15
      X(7)=30
      X(8)=50
C
      Y(1)=1.
      Y(2)=0.645
      Y(3)=0.345
      Y(4)=0.245
      Y(5)=0.175
      Y(6)=0.125
      Y(7)=0.075
      Y(8)=0.05
C
      DO 10 I=1,4
         DELTAA(I)=.01
 10   SY(I)=0.
```

```
C
C  Valeurs intiales des coefficients
C   (attention car on peut tomber sur un minimum local
C     du chi2 - essayer avec quelques valeurs)
C
     A(1)=2
     A(2)=-0.3
C
C  Appel de la sous-routine
C
          II = 0
 15                                                                    CALL
CURFIT(X,Y,SY,NPTS,NCOEF,0,A,DELTAA,SIGMAA,0.001,YFIT,CHISQR)
          II = II + 1
          WRITE(6,*)'ITERATION # ', II
     SCHI=CHISQR
     NITER=NITER+1
     DIFF=ABS (SCHI-FCHI)
     IF( (DIFF.LE. SCHI/100. .AND. NITER .GT. 10)
    1  .OR. CHISQR .LE. 1E-8) GO TO 19
     FCHI=SCHI
     GO TO 15
C
C  Fin du calcul (affichage des resultats)
C
 19   WRITE(*,20) (A(I),I=1,2),(Y(I),YFIT(I),I=1,NPTS),CHISQR,NITER
 20   FORMAT(/,1X,'Coefficients:',/,2(1X,F9.4),//,1X,'YFIT:',/,
     1 8(1X,F9.4,5X,F9.4,/),//,1X,'chi2 final:',/,1X,F9.7,//,
     1 1X,'Nombre d''iterations:',I9,/)
     STOP
     END
C
C
C
     FUNCTION FUNCTN4(X,I,A)
     REAL X(4),A(2)
     FUNCTN=A(1)/(1+A(2)*X(I))
     RETURN
     END

     FUNCTION FUNCTN3(X,I,A)
     REAL X(4),A(2)
```

```
      FUNCTN=A(1)*EXP (A(2)*X(I)**0.5)
      RETURN
      END

      FUNCTION FUNCTN2(X,I,A)
      REAL X(4),A(2)
      FUNCTN=A(1)*EXP(A(2)*X(I)*X(I))
      RETURN
      END

      FUNCTION FUNCTN(X,I,A)
      REAL X(4),A(2)
      FUNCTN=A(1)*EXP(A(2)*X(I))
      RETURN
      END
*
*
* Generateur de nombres aleatoires
*
*
      FUNCTION RAN1(IDUM)
      DIMENSION R(97)
      PARAMETER(M1=259200,IA1=7141,IC1=54773)
      PARAMETER(M2=134456,IA2=8121,IC2=28411)
      PARAMETER(M3=243000,IA3=4561,IC3=51349)
      DATA IFF, iff1 /0, 0/
      RM1=1./M1
      RM2=1./M2
      IF(IDUM .LT. 0 .OR. IFF .EQ. 0) THEN
            IFF=1
            IX1=MOD(IC1-IDUM,M1)
            IX1=MOD(IA1*IX1+IC1,M1)
            IX2=MOD(IX1,M2)
            IX2=MOD(IA1*IX1+IC1,M1)
            IX3=MOD(IX1,M3)
            DO 20 J=1,97
                  IX1=MOD(IA1*IX1+IC1,M1)
                  IX2=MOD(IA2*IX2+IC2,M2)
                  R(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1
20          CONTINUE
            IDUM=1
      ENDIF
```

```
      IX1=MOD(IA1*IX1+IC1,M1)
      IX2=MOD(IA2*IX2+IC2,M2)
      IX3=MOD(IA3*IX3+IC3,M3)
      J=1+(97*IX3)/M3
      IF(J.GT.97 .OR. J.LT.1) PAUSE
      RAN1=R(J)
      R(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1
      RETURN
      END
*
*
*  SOUS-ROUTINES POUR LE LISSAGE
*
      SUBROUTINE CURFIT(X,Y,SIGMAY,NPTS,NTERMS,MODE,A,DELTAA,
     1                  SIGMAA,FLAMDA,YFIT,CHISQR)
      DOUBLE PRECISION ARRAY
      DIMENSION X(NPTS),Y(NPTS),SIGMAY(NPTS),A(NPTS),DELTAA(NPTS),
     1          SIGMAA(NPTS),YFIT(NPTS),WEIGHT(100),ALPHA(10,10),
     1          BETA(10),DERIV(10),ARRAY(10,10),B(10)
 11   NFREE=NPTS-NTERMS
      IF(NFREE) 13, 13, 20
 13   CHISQR=0.
      GO TO 110
C
C     EVALUATE WEIGHTS
C
 20   DO 30 I=1,NPTS
 21   IF(MODE) 22, 27, 29
 22   IF(Y(I)) 25, 27, 23
 23   WEIGHT(I)=1./Y(I)
      GO TO 30
 25   WEIGHT(I)=1./(-Y(I))
      GO TO 30
 27   WEIGHT(I)=1.
      GO TO 30
 29   WEIGHT(I)=1./SIGMAY(I)**2
 30   CONTINUE
C
C     EVALUATE ALPHA AND BETA MATRICES
C
 31   DO 35 J=1,NTERMS
      BETA(J)=0.
```

```
      DO 34 K=1,J
 34   ALPHA(J,K)=0.
 35   CONTINUE
 41   DO 50 I=1,NPTS
      CALL FDERIV(X,I,A,DELTAA,NTERMS,DERIV)
      DO 47 J=1,NTERMS
      BETA(J)=BETA(J)+WEIGHT(I)*(Y(I)-FUNCTN(X,I,A))*DERIV(J)
      DO 46 K=1,J
 46   ALPHA(J,K)=ALPHA(J,K)+WEIGHT(I)*DERIV(J)*DERIV(K)
 47   CONTINUE
 50   CONTINUE
 51   DO 54 J=1,NTERMS
      DO 53 K=1,J
 53   ALPHA(K,J)=ALPHA(J,K)
 54   CONTINUE
C
C     EVALUATE CI SQUARE AT STARTING POINT
C
 61   DO 62 I=1,NPTS
 62   YFIT(I)=FUNCTN(X,I,A)
 63   CHISQ1=FCHISQ(Y,SIGMAY,NPTS,NFREE,MODE,YFIT)
CSIGMAY,NPTS,NFREE,MODE,YFIT
C     INVERT MODIFIED CURVATURE MATRIX TO FIND NEW PARAMETERS
C
 71   DO 74 J=1,NTERMS
      DO 73 K=1,NTERMS
 73   ARRAY(J,K)=ALPHA(J,K)/SQRT(ALPHA(J,J)*ALPHA(K,K))
 74   ARRAY(J,J)=1.+FLAMDA
 80   CALL MATINV(ARRAY,NTERMS,DET)
 81   DO 85 J=1,NTERMS
      B(J)=A(J)
      DO 84 K=1,NTERMS
 84   B(J)=B(J)+BETA(K)*ARRAY(J,K)/SQRT(ALPHA(J,J)*ALPHA(K,K))
 85   CONTINUE
C
C     IF CHI SQUARE INCREASED, INCREASE FLAMDA AND TRY AGAIN
C
 91   DO 92 I=1,NPTS
 92   YFIT(I)=FUNCTN(X,I,B)
 93   CHISQR=FCHISQ(Y,SIGMAY,NPTS,NFREE,MODE,YFIT)
      IF(CHISQ1-CHISQR) 95, 101, 101
 95   FLAMDA=10.*FLAMDA
```

```
      GO TO 71
C
C     EVALUATE PARAMETERS AND UNCERTAINTIES
C
 101  DO 103 J=1,NTERMS
      A(J)=B(J)
 103  SIGMAA(J)=SQRT(ARRAY(J,J)/ALPHA(J,J))
      FLAMDA=FLAMDA/10.
 110  RETURN
      END
*
*
*
*
      SUBROUTINE FDERIV(X,I,A,DELTAA,NTERMS,DERIV)
      DIMENSION X(NTERMS),A(NTERMS),DELTAA(NTERMS),DERIV(NTERMS)
 11   DO 18 J =1,NTERMS
      AJ=A(J)
      DELTA=DELTAA(J)
      A(J)=AJ+DELTA
      YFIT=FUNCTN(X,I,A)
      A(J)=AJ-DELTA
      DERIV(J)=(YFIT-FUNCTN(X,I,A))/(2.*DELTA)
 18   A(J)=AJ
      RETURN
      END
*
*
*
*
      SUBROUTINE MATINV(ARRAY,NORDER,DET)
      DOUBLE PRECISION ARRAY,AMAX,SAVE
      DIMENSION ARRAY(10,10),IK(10),JK(10)
 10   DET=1.
 11   DO 100 K=1,NORDER
C
C     FIND LARGEST ELEMENT ARRAY(I,J) IN REST OF MATRIX
C
      AMAX=0.
 21   DO 30 I=K,NORDER
      DO 29 J=K,NORDER
 23   IF(DABS(AMAX)-DABS(ARRAY(I,J))) 24, 24, 30
```

```
 24   AMAX=ARRAY(I,J)
      IK(K)=I
      JK(K)=J
 29   CONTINUE
 30   CONTINUE
C
C     INTERCANGE ROWS AND COLUMS TO PUT AMAX IN ARRAY(K,K)
C
 31   IF(AMAX) 41, 32, 41
 32   DET=0.
      GO TO 140
 41   I=IK(K)
      IF(I-K) 21, 51, 43
 43   DO 50 J=1,NORDER
      SAVE=ARRAY(K,J)
      ARRAY(K,J)=ARRAY(I,J)
 50   ARRAY(I,J)=-SAVE
 51   J=JK(K)
      IF (J-K) 21, 61, 53
 53   DO 60 I=1,NORDER
      SAVE=ARRAY(I,K)
      ARRAY(I,K)=ARRAY(I,J)
 60   ARRAY(I,J)=-SAVE
C
C     ACCUMULATE ELEMENTS OF INVERSE MATRIX
C
 61   DO 70 I=1,NORDER
      IF(I-K) 63, 70, 63
 63   ARRAY(I,K)=-ARRAY(I,K)/AMAX
 70   CONTINUE
 71   DO 80 I=1,NORDER
      DO 79 J=1,NORDER
      IF(I-K) 74, 80, 74
 74   IF(J-K) 75, 80, 75
 75   ARRAY(I,J)=ARRAY(I,J)+ARRAY(I,K)*ARRAY(K,J)
 79   CONTINUE
 80   CONTINUE
 81   DO 90 J=1,NORDER
      IF(J-K) 83, 90, 83
 83   ARRAY(K,J)=ARRAY(K,J)/AMAX
 90   CONTINUE
      ARRAY(K,K)=1./AMAX
```

```
 100  DET=DET*AMAX
C
C    RESTORE ORDERING OF MATRIX
C
 101  DO 130 L=1,NORDER
      K=NORDER-L+1
      J=IK(K)
      IF(J-K) 111, 111, 105
 105  DO 110 I=1,NORDER
      SAVE=ARRAY(I,K)
      ARRAY(I,K)=-ARRAY(I,J)
 110  ARRAY(I,J)=SAVE
 111  I=JK(K)
      IF(I-K) 130, 130, 113
 113  DO 120 J=1,NORDER
      SAVE=ARRAY(K,J)
      ARRAY(K,J)=-ARRAY(I,J)
 120  ARRAY(I,J)=SAVE
 130  CONTINUE
 140  RETURN
      END
*
*
*
*
      FUNCTION FCHISQ(Y,SIGMAY,NPTS,NFREE,MODE,YFIT)
      DOUBLE PRECISION CHISQ,WEIGHT
      DIMENSION Y(NPTS),SIGMAY(NPTS),YFIT(NPTS)
 11   CHISQ=0.
 12   IF(NFREE) 13, 13, 20
 13   FCHISQ=0.
      GO TO 40
C
C    ACCUUMULATE CHI SQUARE
C
 20   DO 30 I=1,NPTS
 21   IF(MODE) 22, 27, 29
 22   IF(Y(I)) 25, 27, 23
 23   WEIGHT=1./Y(I)
      GO TO 30
 25   WEIGHT=1./(-Y(I))
      GO TO 30
```

```
 27   WEIGHT=1.
      GO TO 30
 29   WEIGHT=1./SIGMAY(I)**2
 30   CHISQ=CHISQ+WEIGHT*(Y(I)-YFIT(I))**2
C
C     DIVIDE BY NUMBER DEGREES OF FREEDOM
C
 31   FREE=NFREE
 32   FCHISQ=CHISQ/FREE
 40   RETURN
      END
```

Fichier source du programme de multiplication de matrice (MULMAT)

```
      PROGRAM MULMAT
c
c     multiplication d'une matrice A de NxN par une matrice B de NxN
c
      integer I, J, K, N
      real    A(10, 10), B(10, 10), R(10, 10)
c
      data    N       / 10 /
c
      open(UNIT=10, NAME='MULMAT_DATAA', STATUS='UNKNOWN')
      read(10,*)((A(I, J), I=1,N), J=1,N)
      close(10)
c
      open(UNIT=10, NAME='MULMAT_DATAB', STATUS='UNKNOWN')
      read(10,*)((B(I, J), I=1,N), J=1,N)
      close(10)
c
c     algorithme de multiplication de matrice carre
c
      do I = 1, N
            do J = 1, N
                  R(J,I) = 0
                  do K = 1, N
                        R(J,I) = R(J,I) + A(J,K) * B(K,I)
                  enddo
            write(6,*)'j= ', J
            enddo
      enddo
      write(6,*)'fin de la multiplication'
      end
```

Annexe 4

Graphiques des résultats du chapitre IV

Liste des graphiques du gain de vitesse en fonction des coûts d'une communication pour chaque application testée

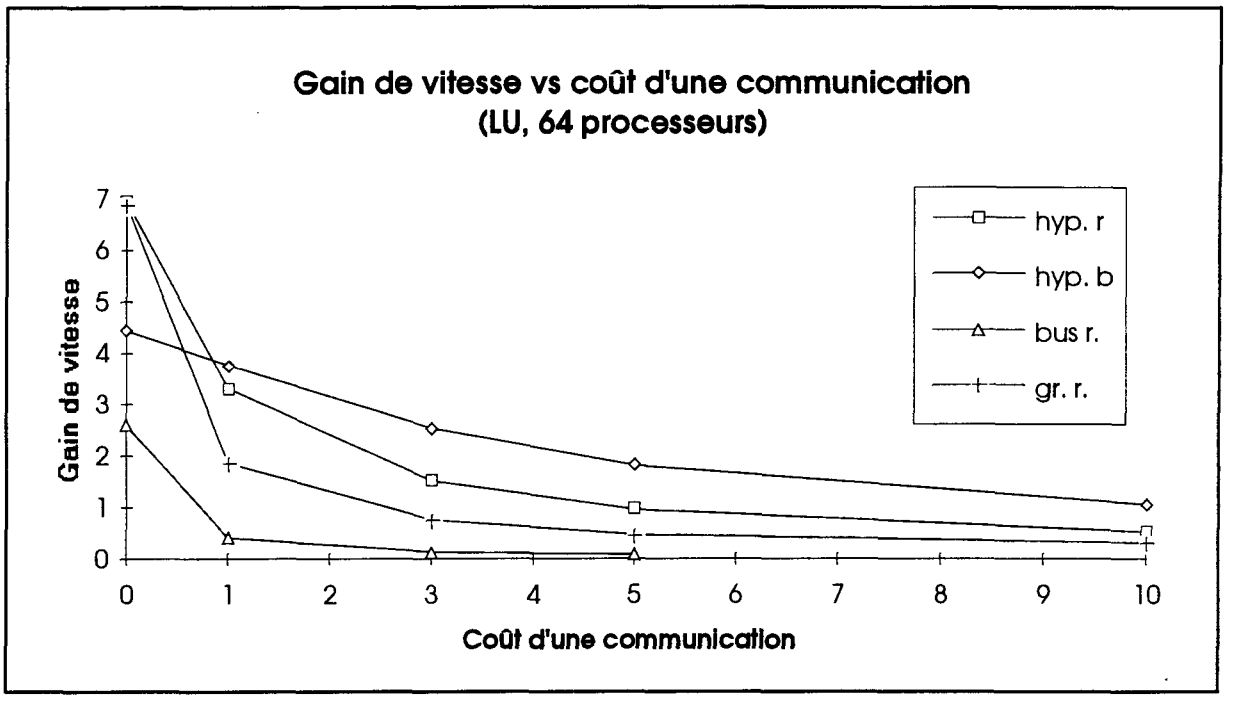

Figure A4.1 Gain de vitesse vs coûts de communication pour l'application LU.

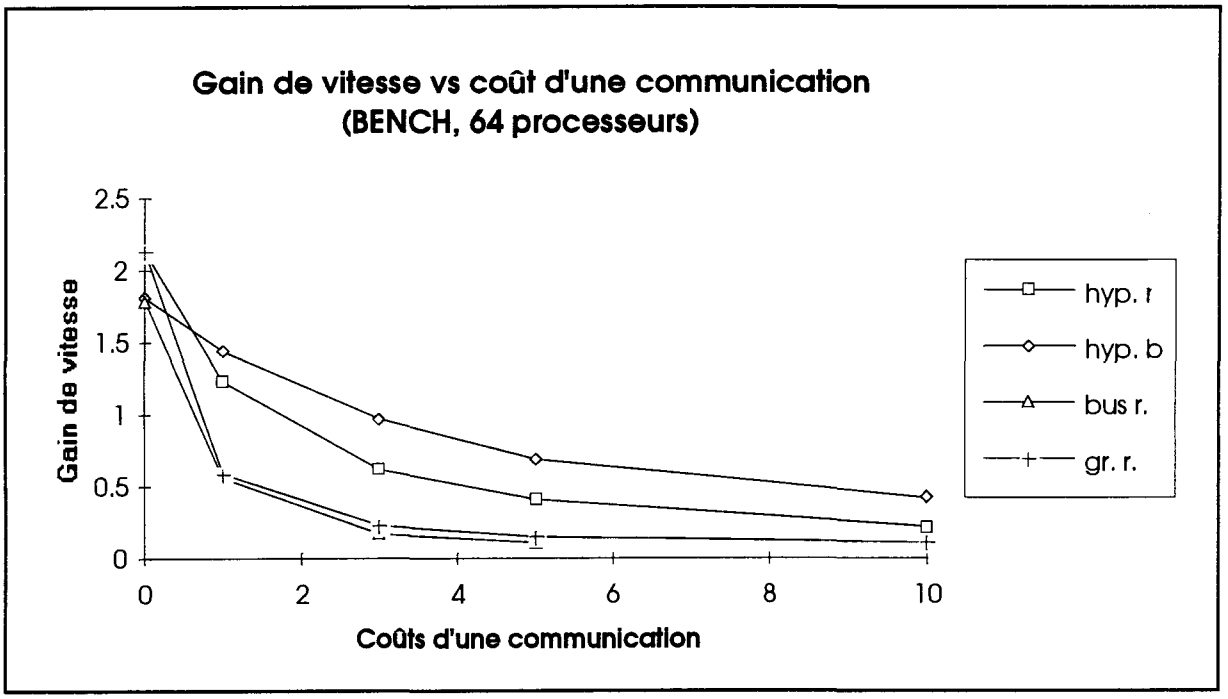

Figure A4.2 Gain de vitesse vs coûts de communication pour l'application BENCH.

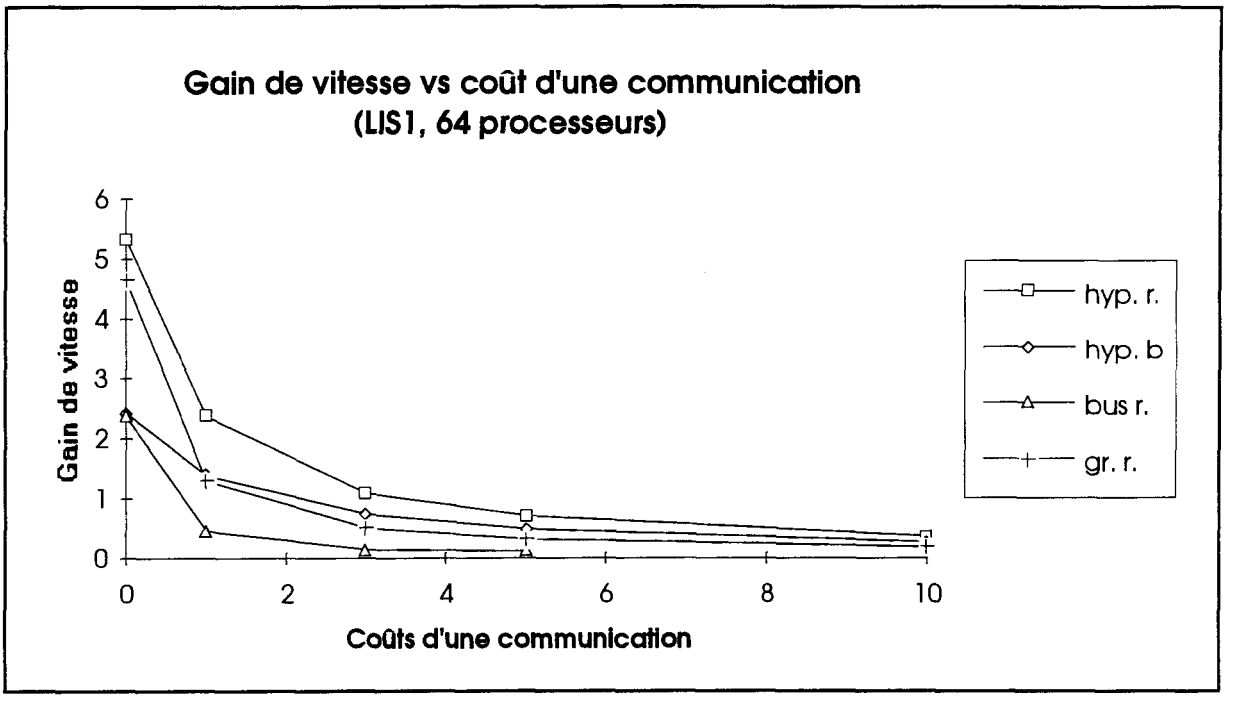

Figure A4.3 Gain de vitesse vs coûts de communication pour l'application LIS1.

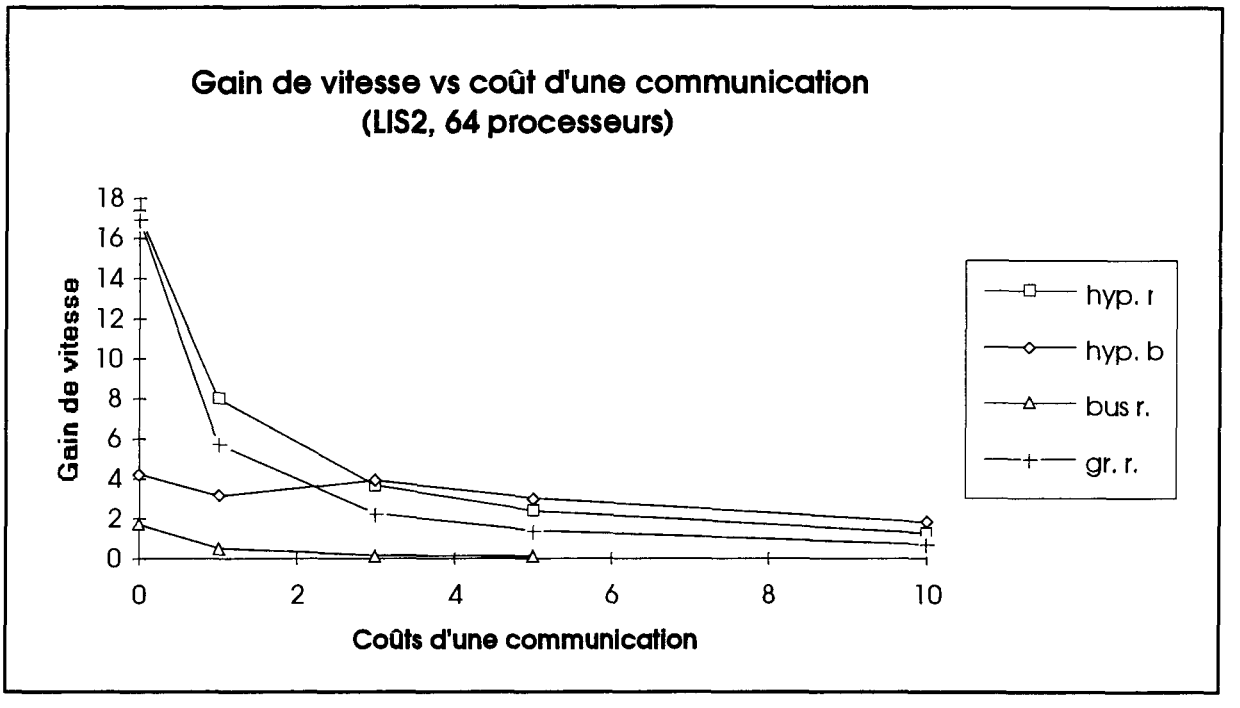

Figure A4.4 Gain de vitesse vs coûts de communication pour l'application LIS2.

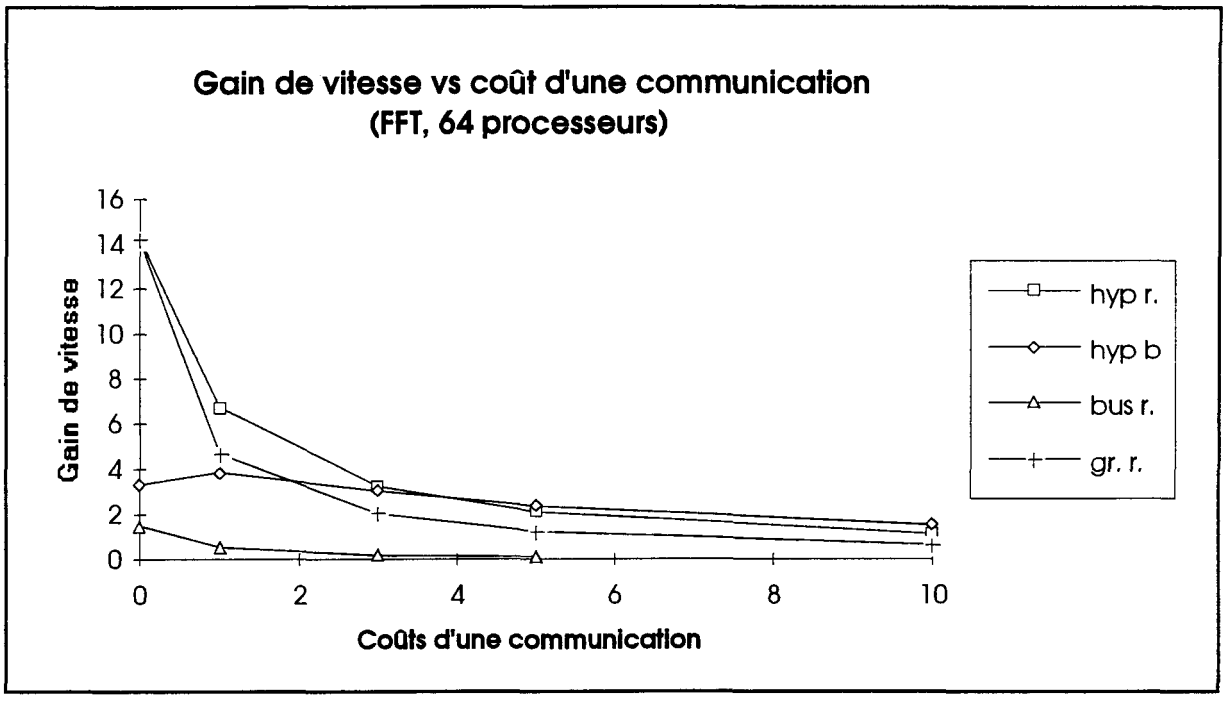

Figure A4.5 Gain de vitesse vs coûts de communication pour l'application FFT.

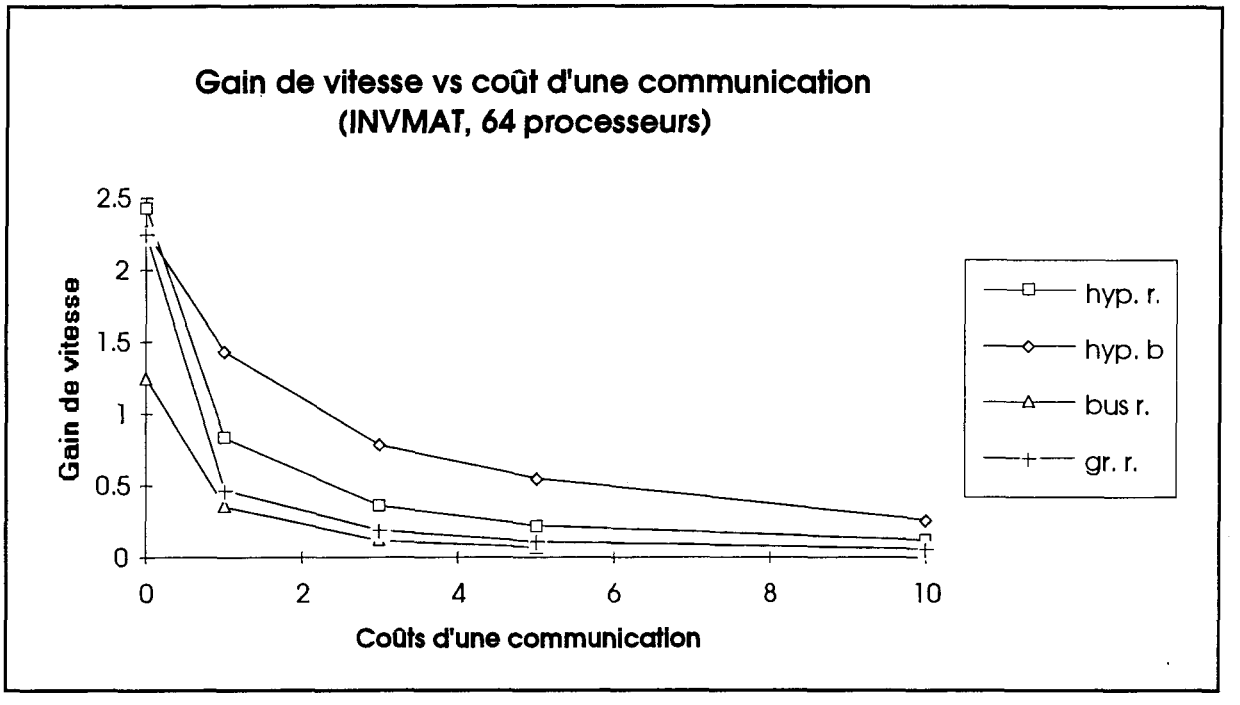

Figure A4.6 Gain de vitesse vs coûts de communication pour l'application INVMAT.

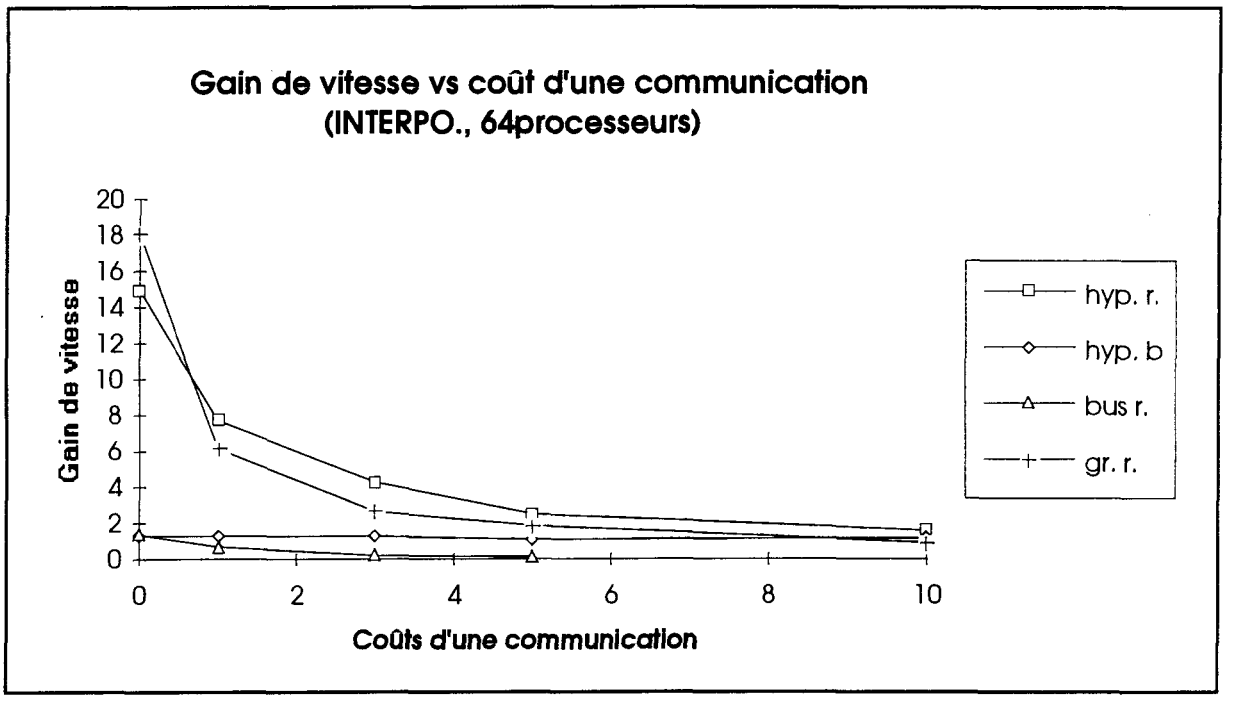

Figure A4.7 Gain de vitesse vs coûts de communication pour l'application INTERPO.

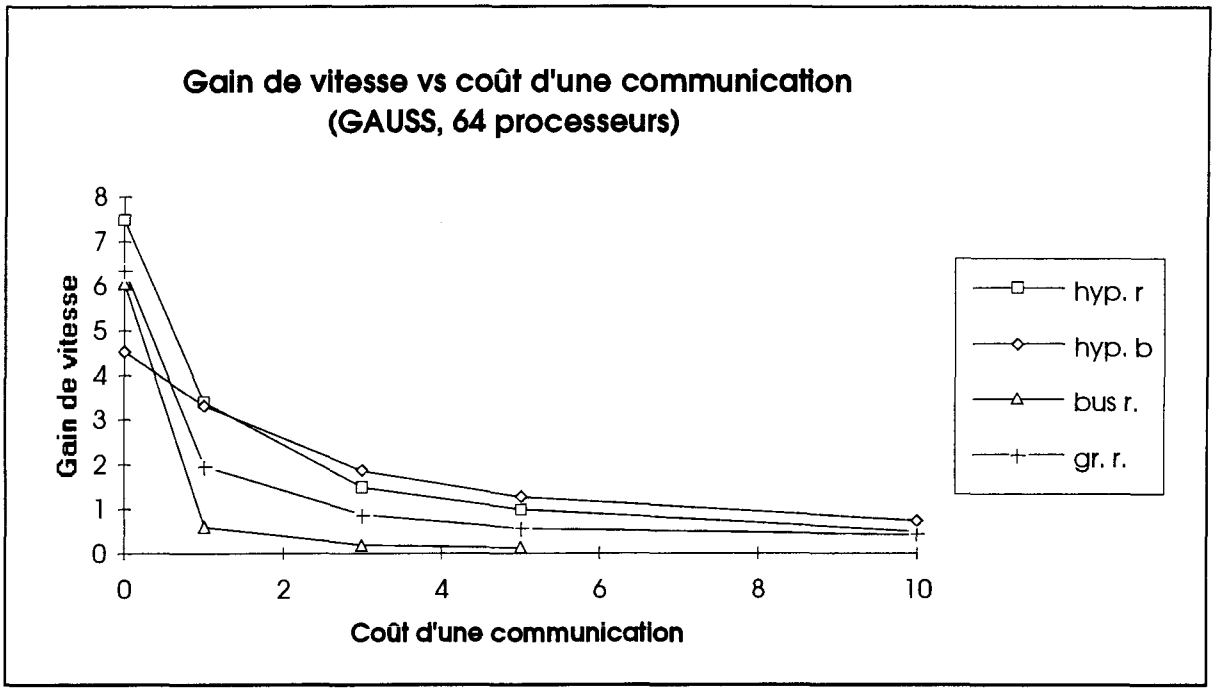

Figure A4.8 Gain de vitesse vs coûts de communication pour l'application GAUSS.

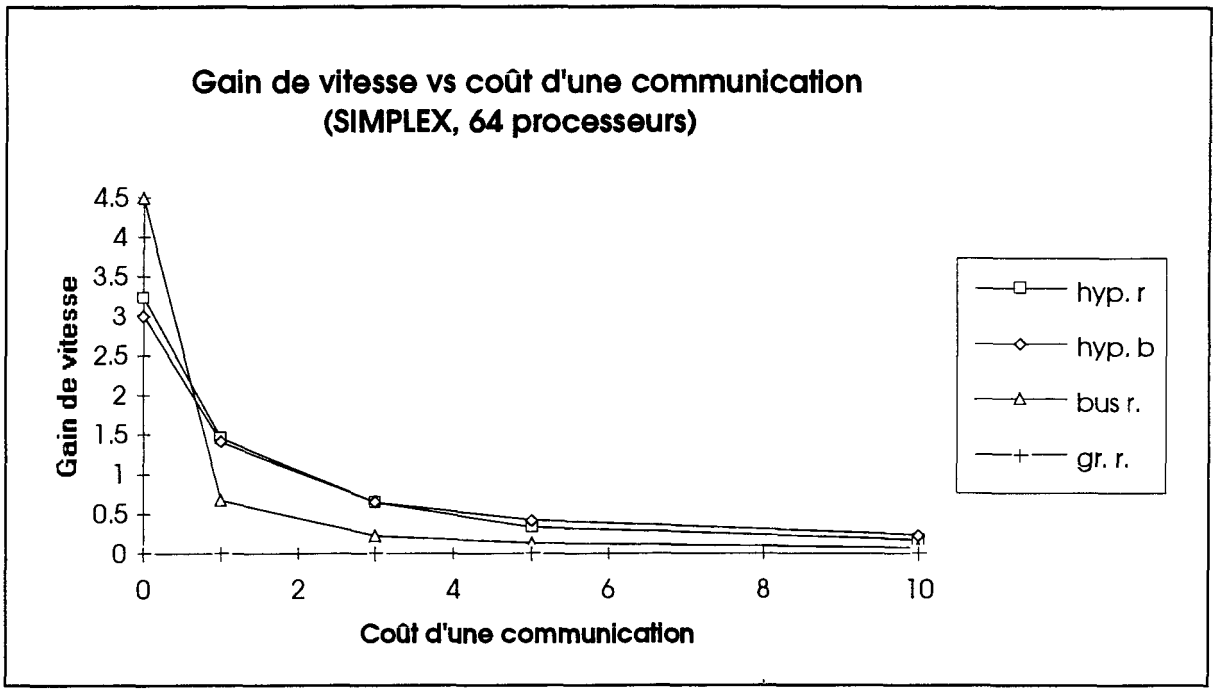

Figure A4.9 Gain de vitesse vs coûts de communication pour l'application SIMPLEX.

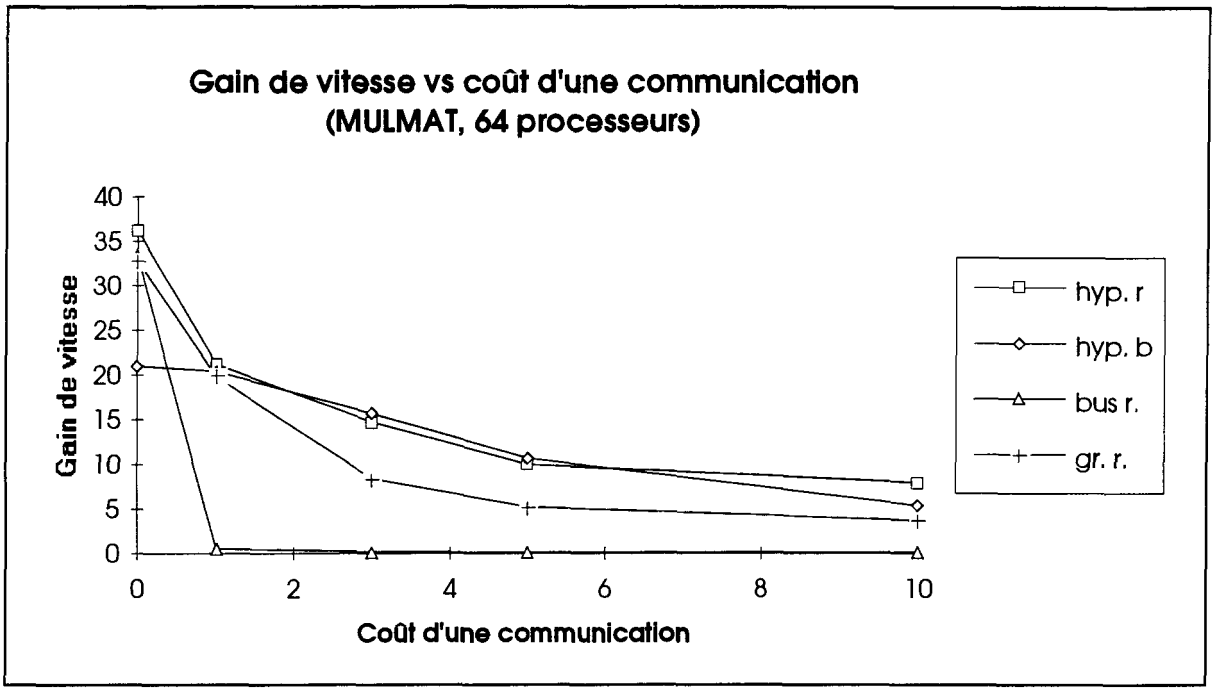

Figure A4.10 Gain de vitesse vs coûts de communication pour l'application MULMAT.

Liste des graphiques du nombre moyen d'opérations de communications vs coûts d'une communication pour chaque application testée

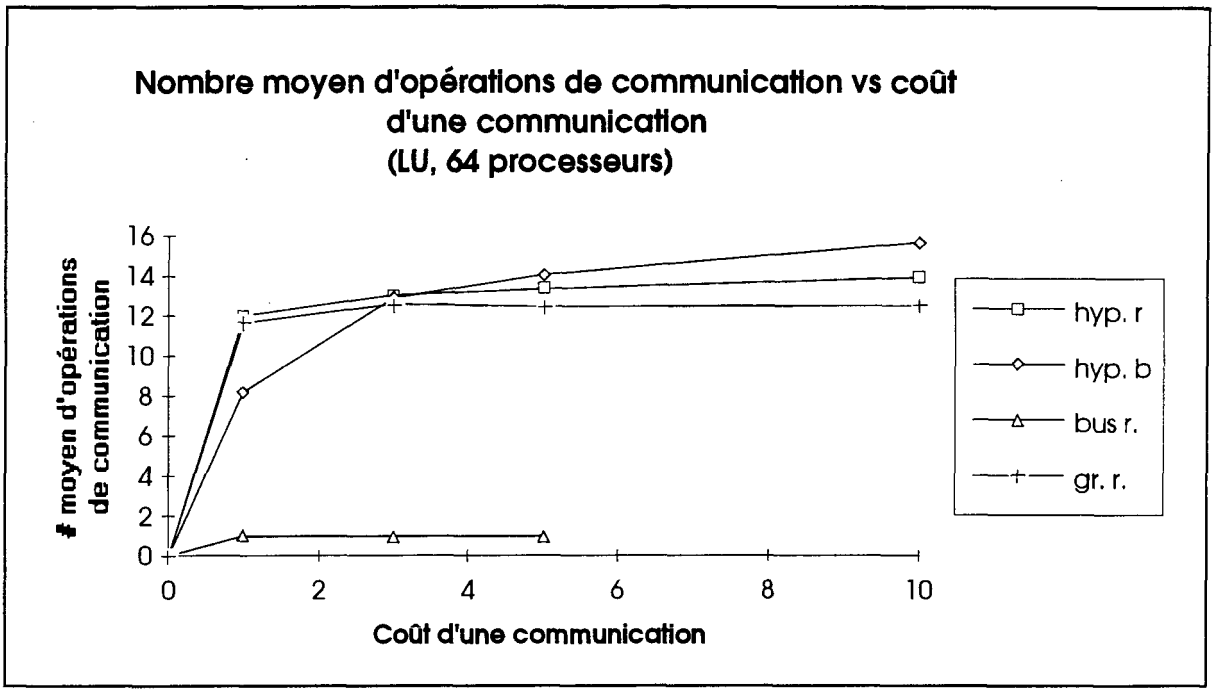

Figure A4.11 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application LU.

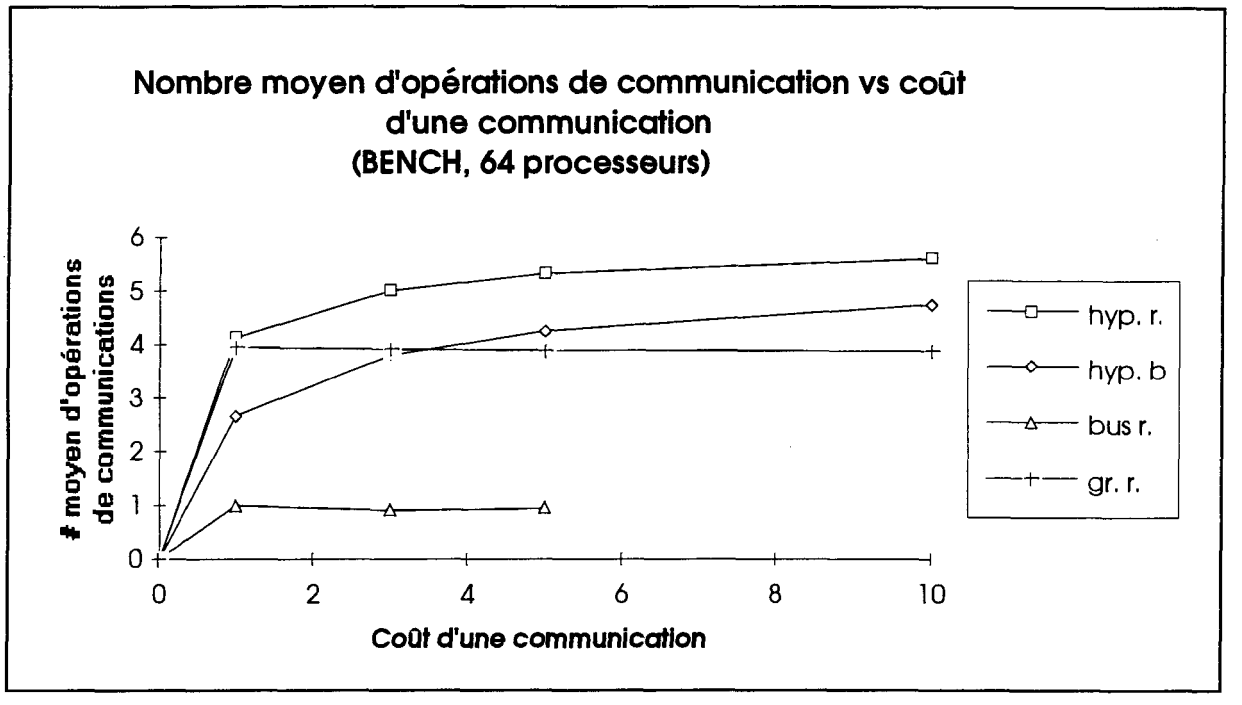

Figure A4.12 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application BENCH.

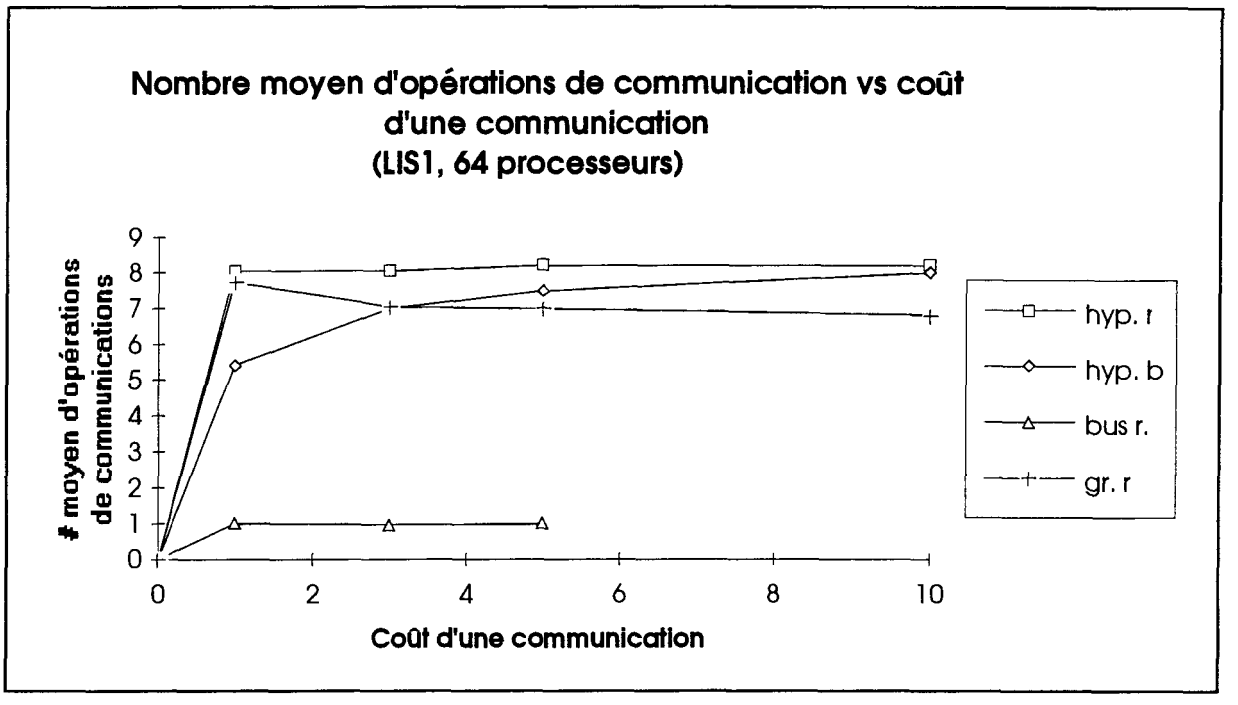

Figure A4.13 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application LIS1.

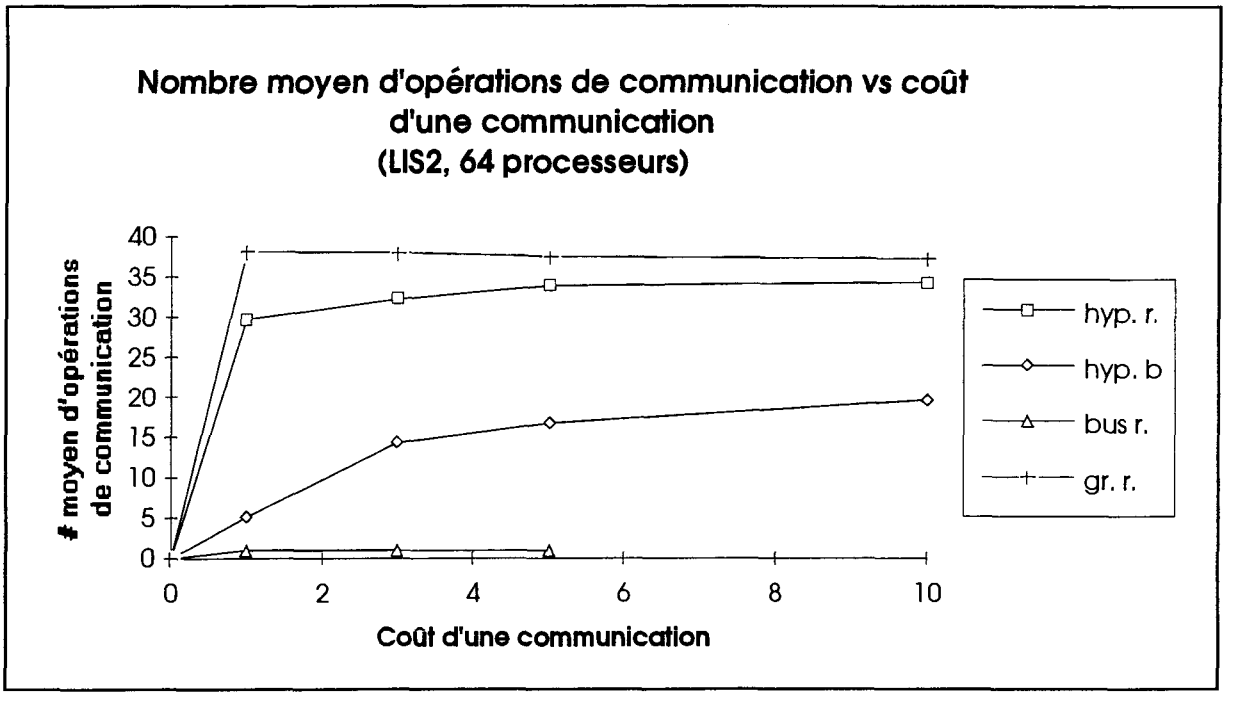

Figure A4.14 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application LIS2.

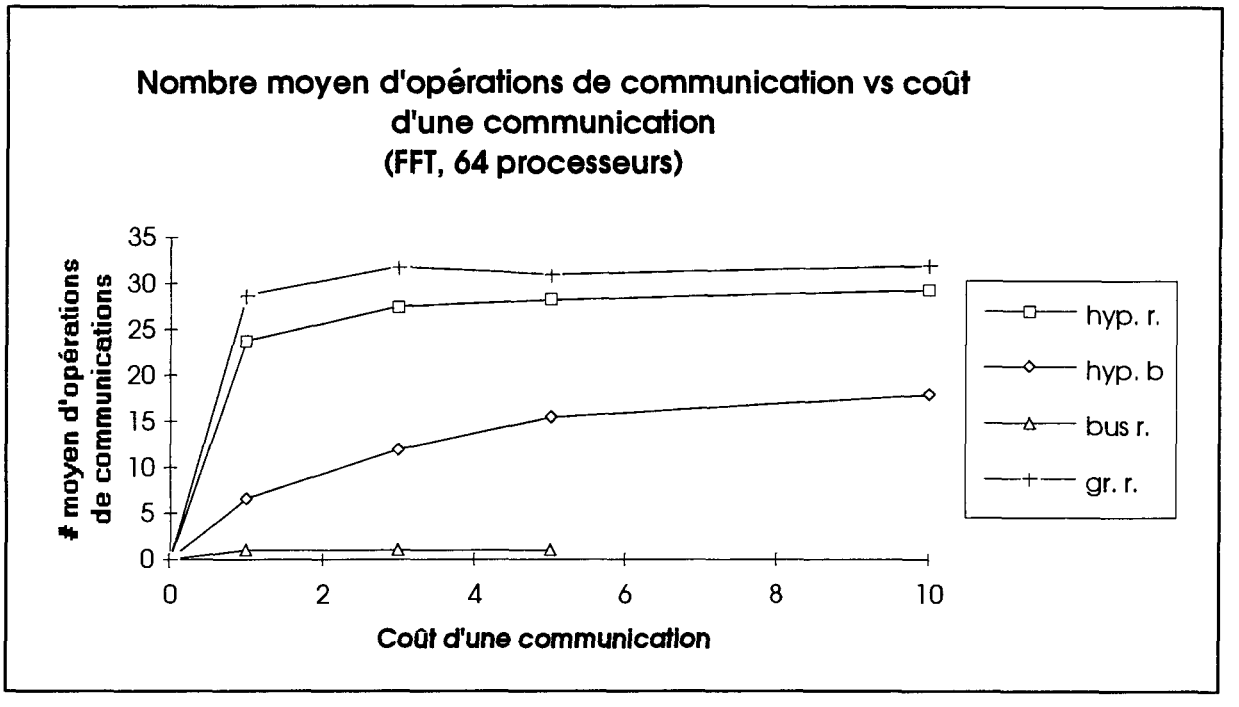

Figure A4.15 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application FFT.

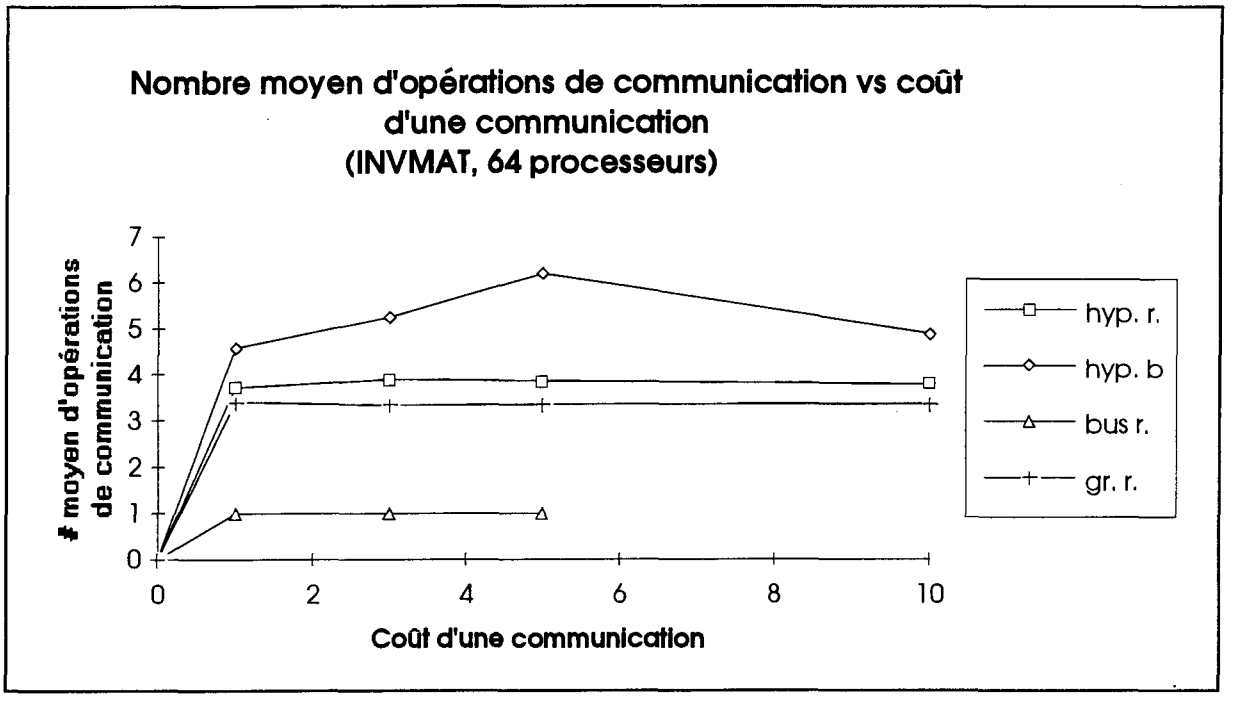

Figure A4.16 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application INVMAT.

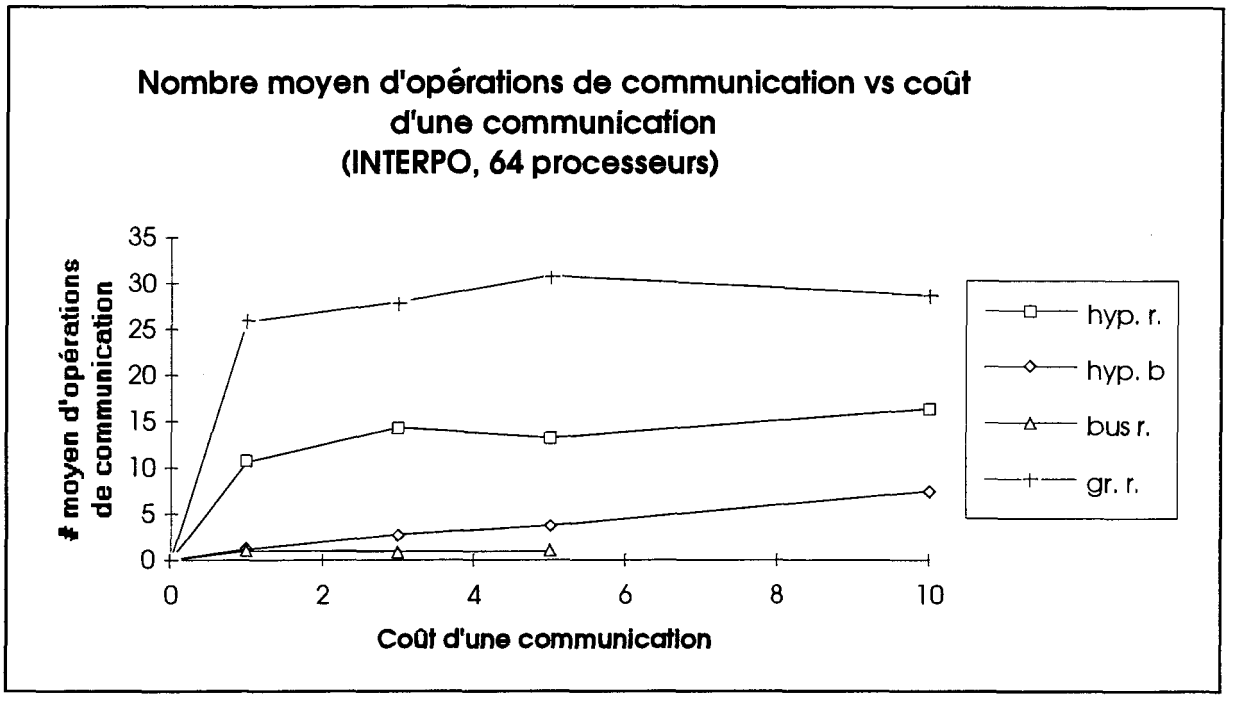

Figure A4.17 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application INTERPO.

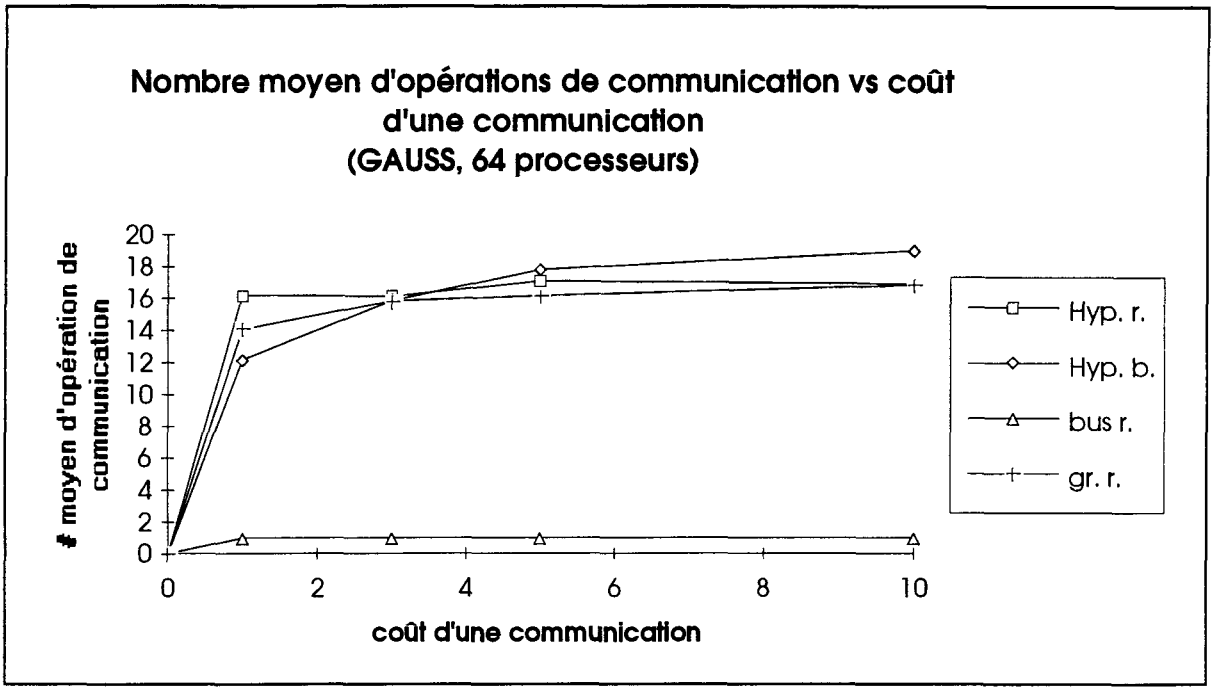

Figure A4.18 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application GAUSS.

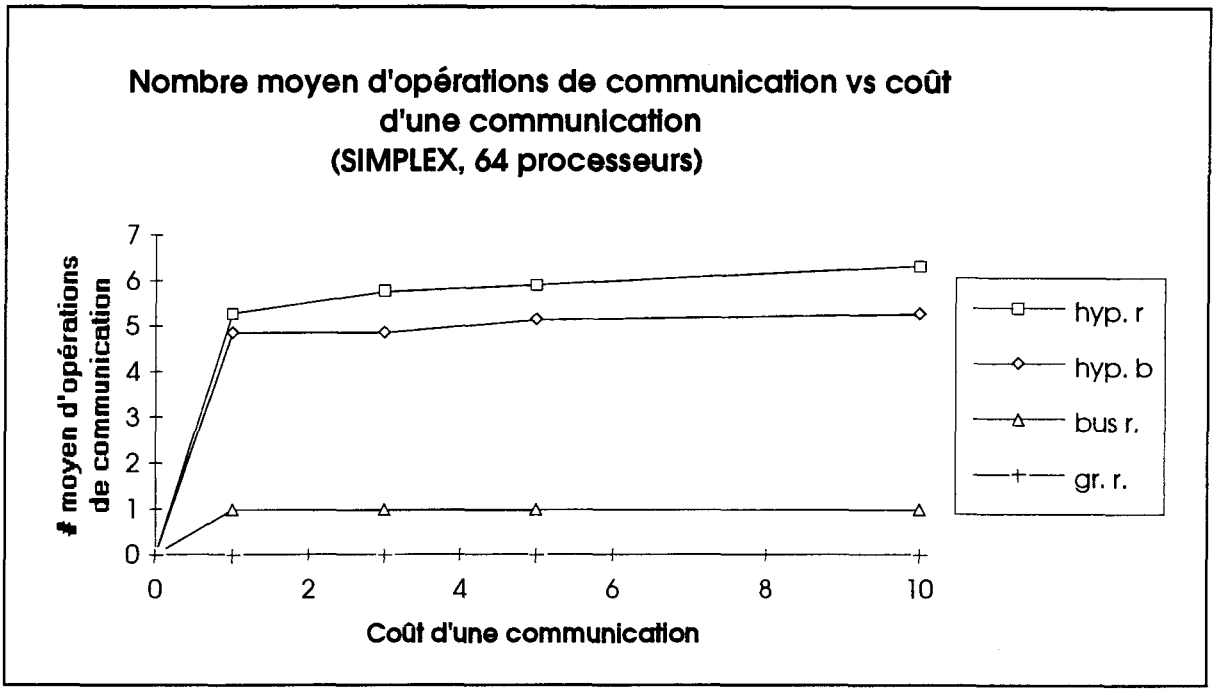

Figure A4.19 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application SIMPLEX.

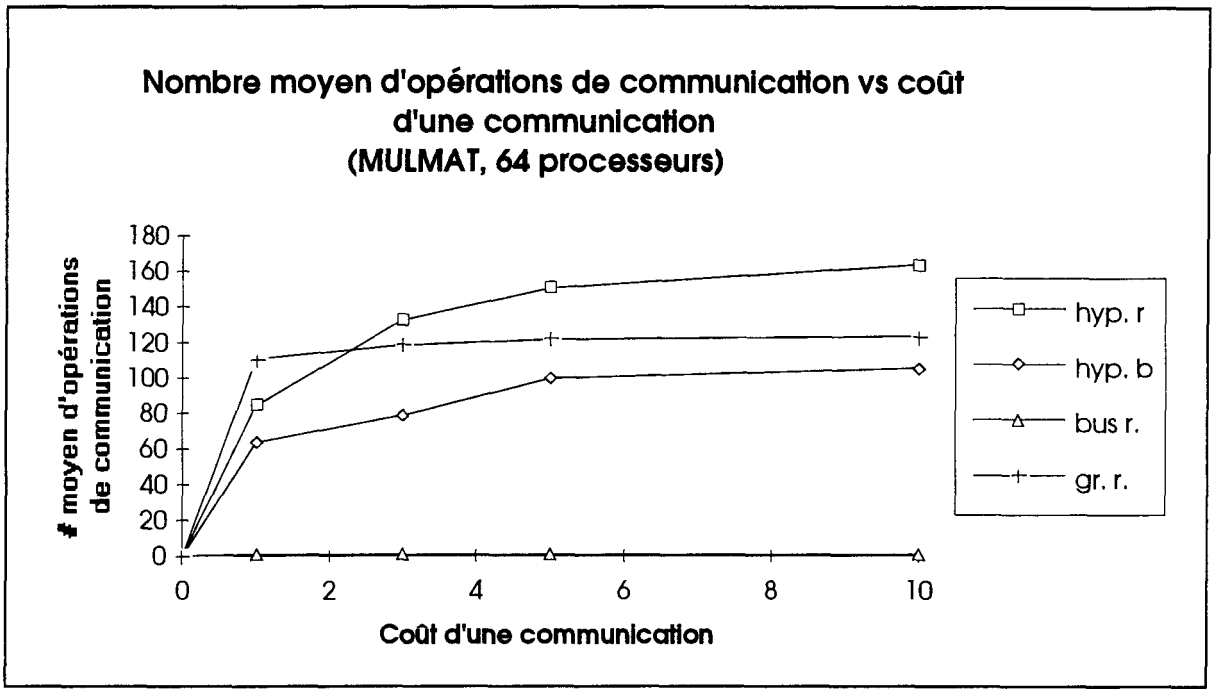

Figure A4.20 Nombre moyen d'opérations de communications vs coûts d'une communication pour l'application MULMAT.

Liste des graphiques du gain de vitesse vs nombre de processeurs pour toutes les architectures testées

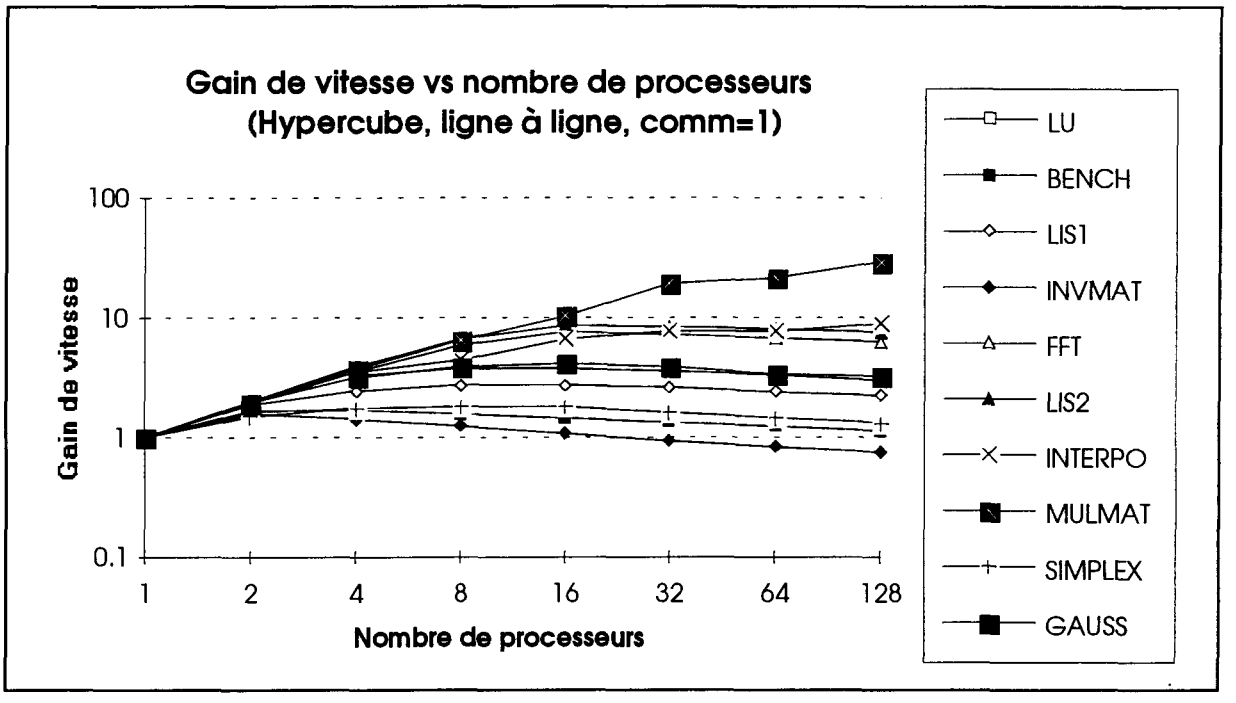

Figure A4.21 Gain de vitesse vs nombre de processeurs pour une architecture hypercube, une allocation ligne à ligne aléatoire et un coût de communication de 1.

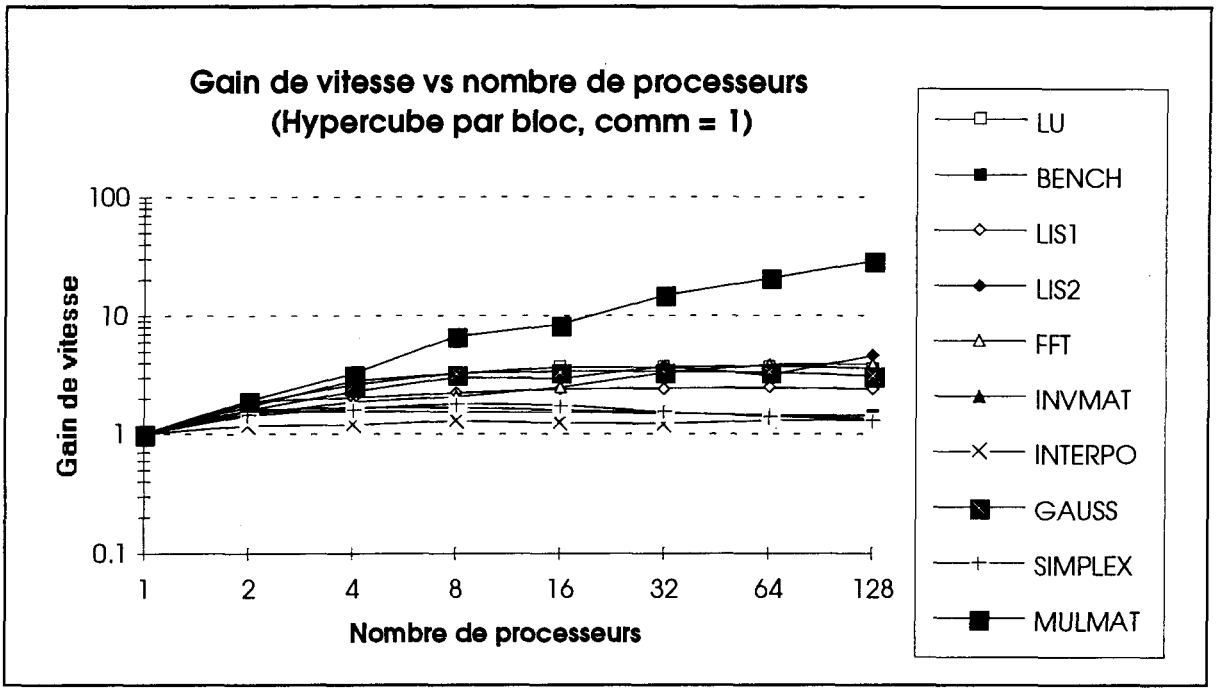

Figure A4.22 Gain de vitesse vs nombre de processeurs pour une architecture hypercube, une allocation par bloc aléatoire et un coût de communication de 1.

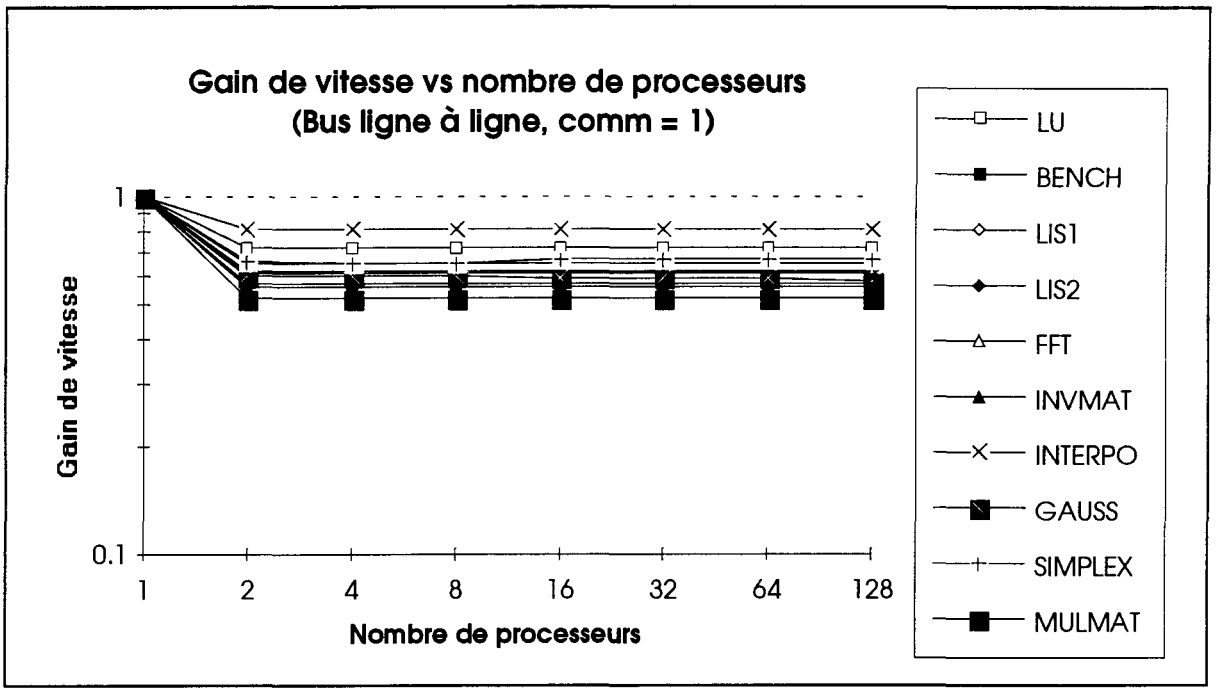

Figure A4.23 Gain de vitesse vs nombre de processeurs pour une architecture en bus, une allocation ligne à ligne aléatoire et un coût de communication de 1.

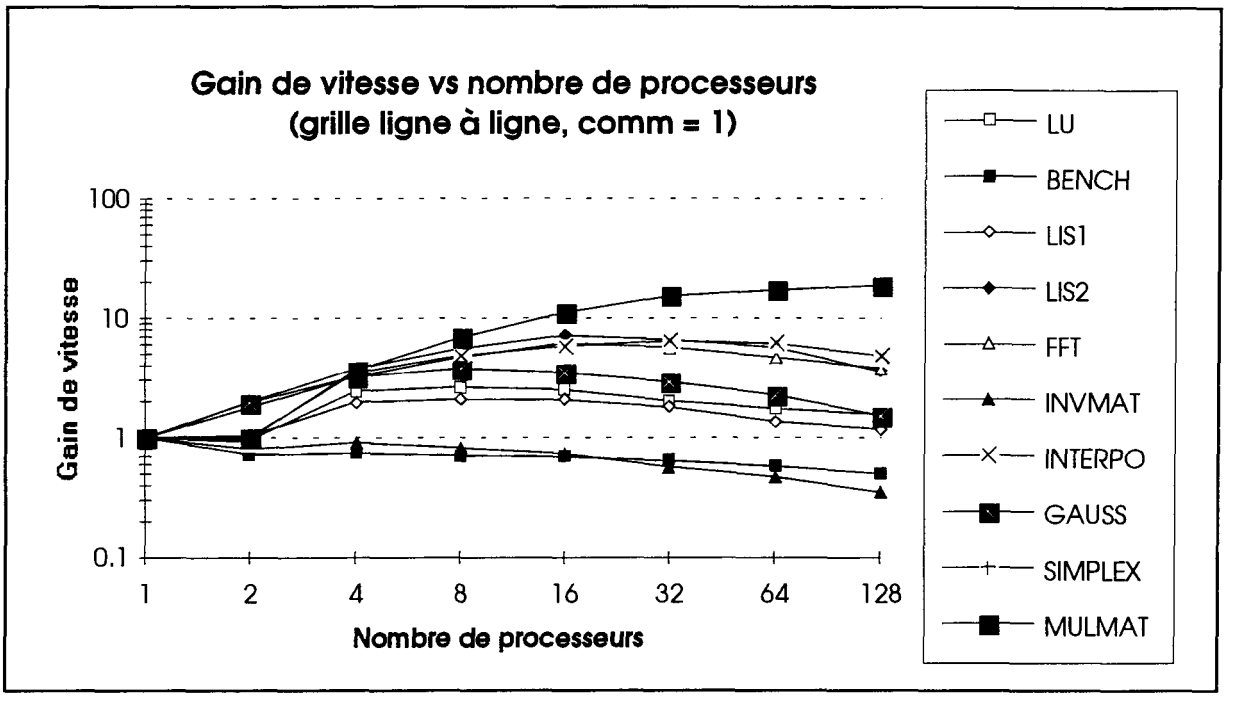

Figure A4.24 Gain de vitesse vs nombre de processeurs pour une architecture en grille, une allocation ligne à ligne aléatoire et un coût de communication de 1.

Liste des graphiques du pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour chaque application testée

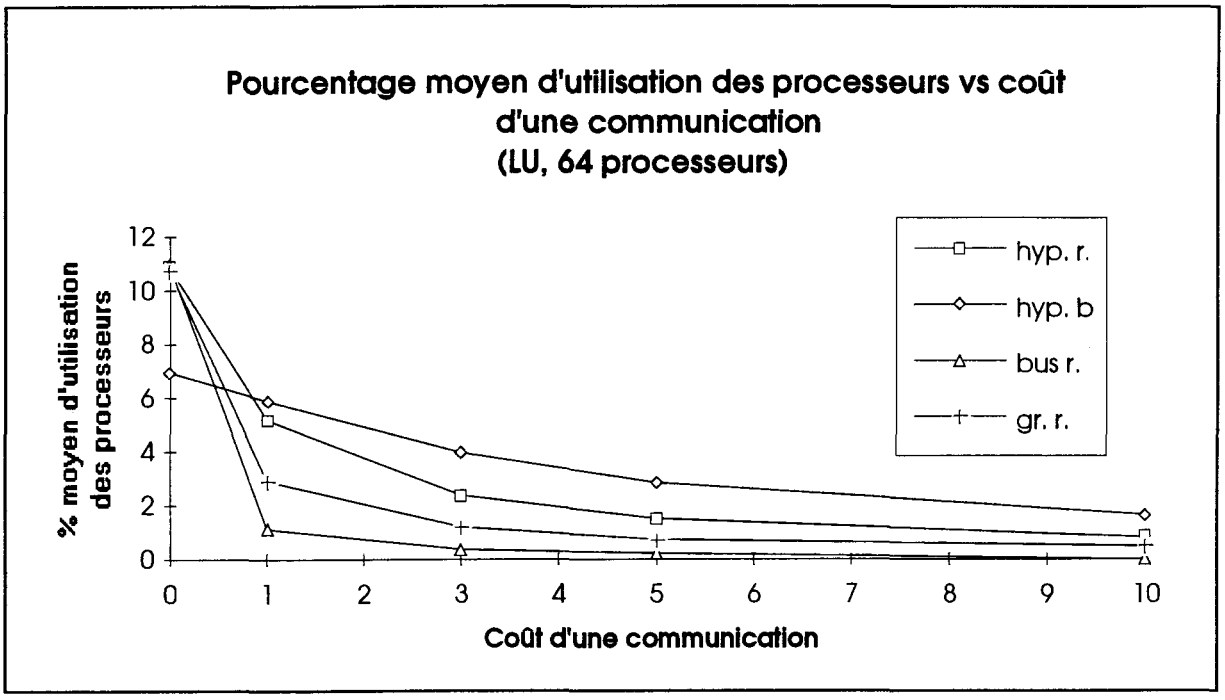

Figure A4.25 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application LU.

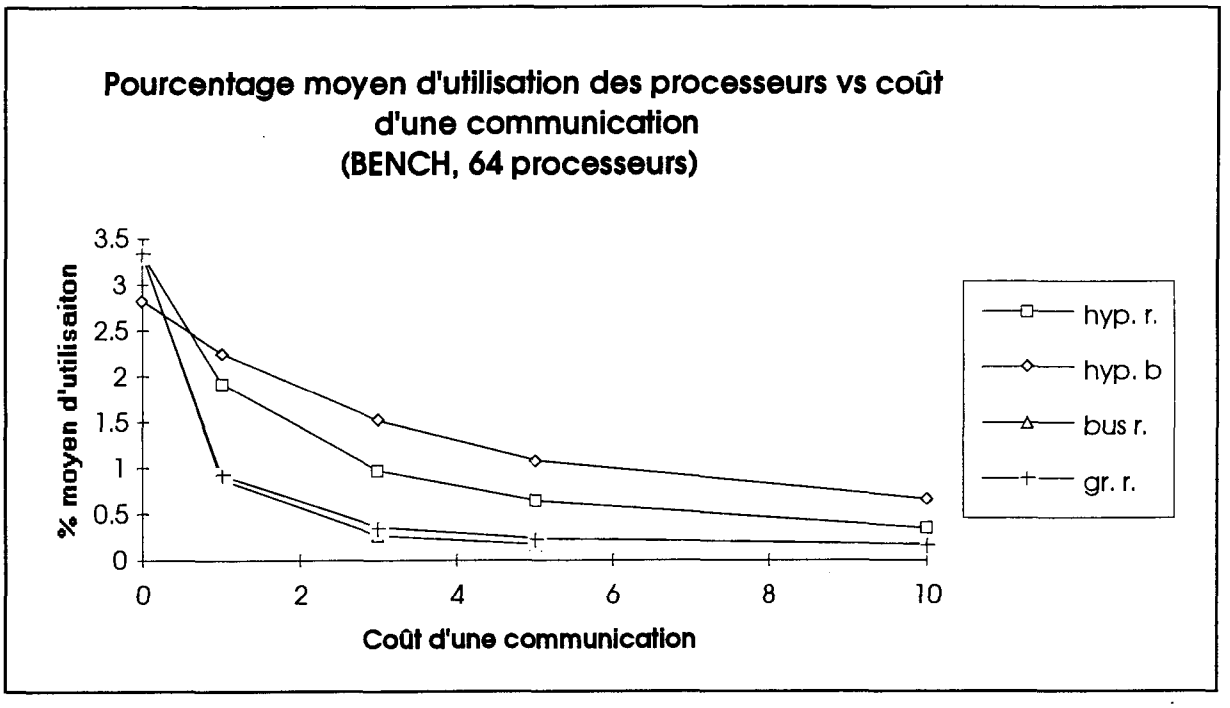

Figure A4.26 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application BENCH.

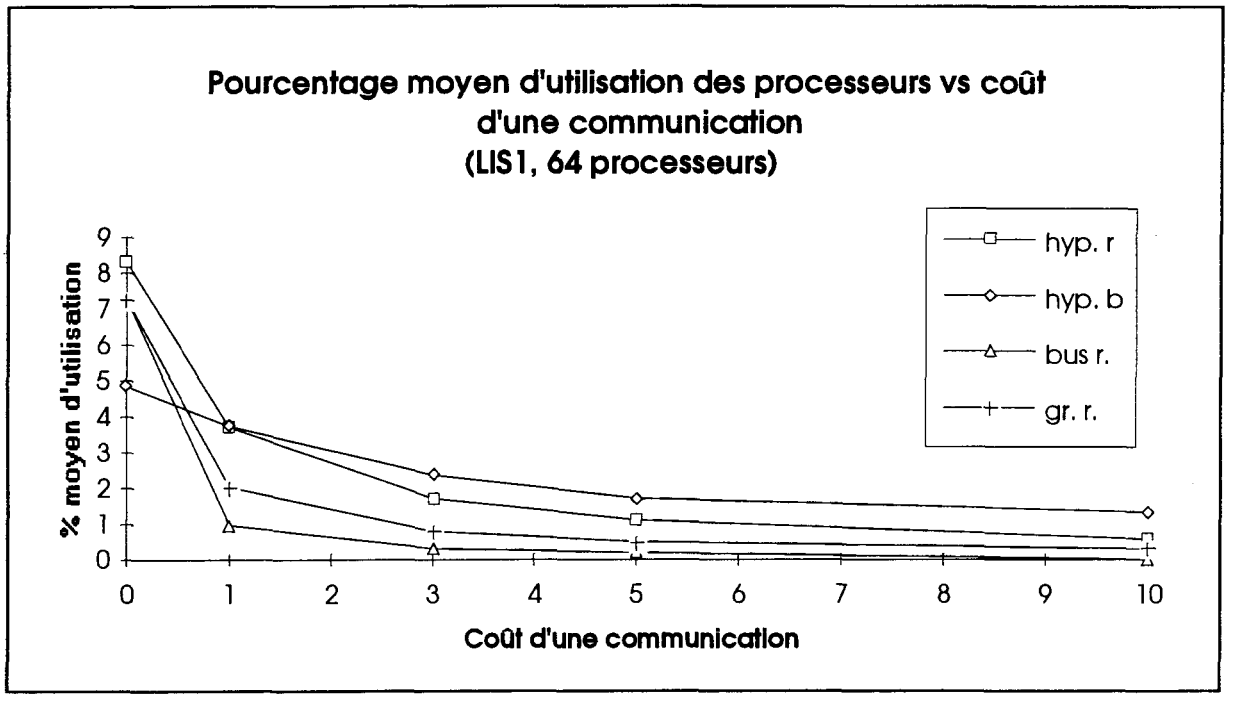

Figure A4.27 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application LIS1.

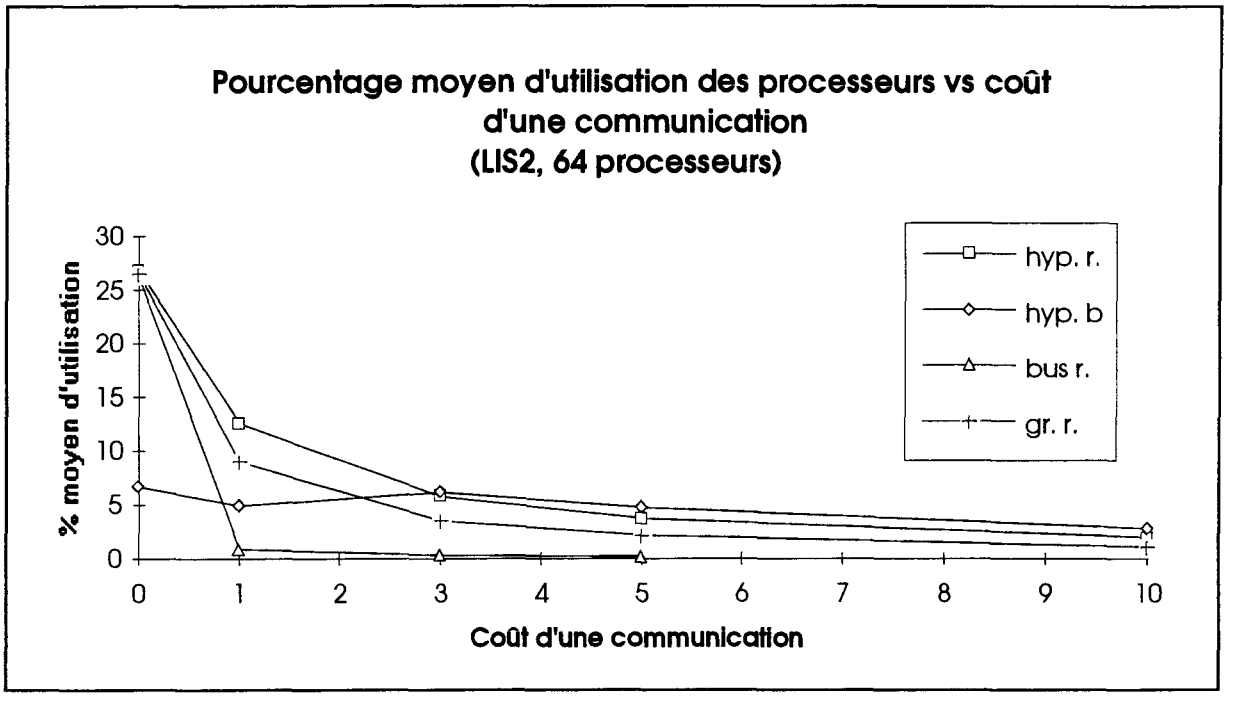

Figure A4.28 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application LIS2.

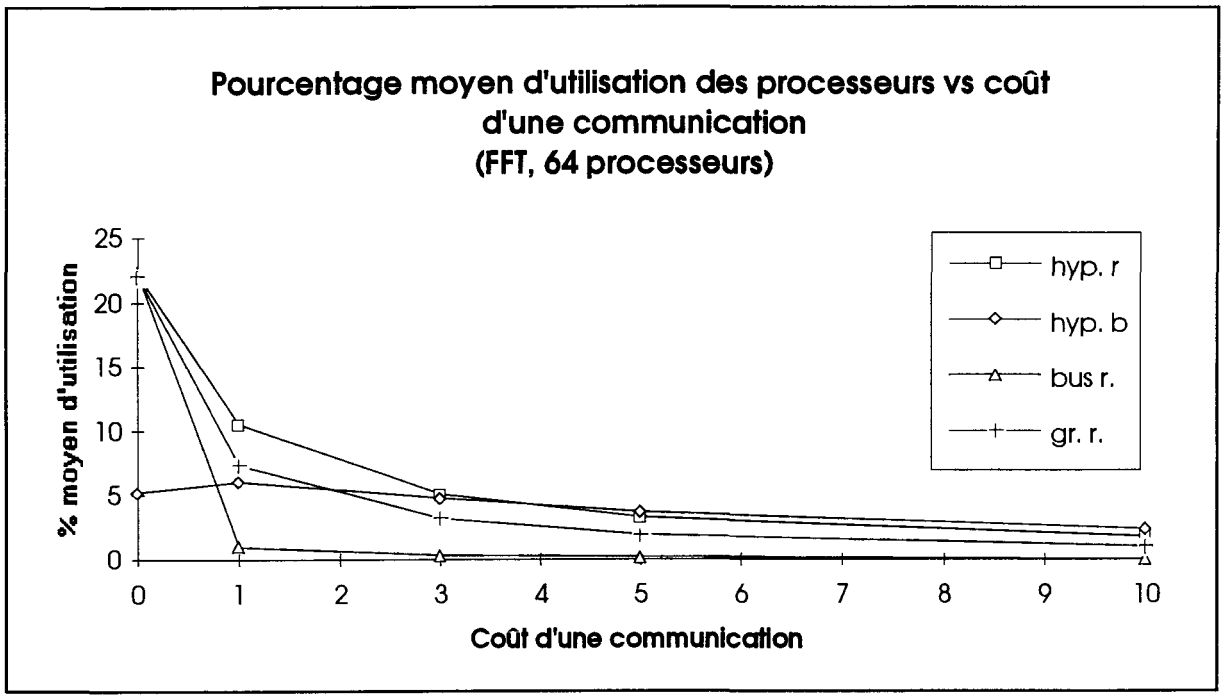

Figure A4.29 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application FFT.

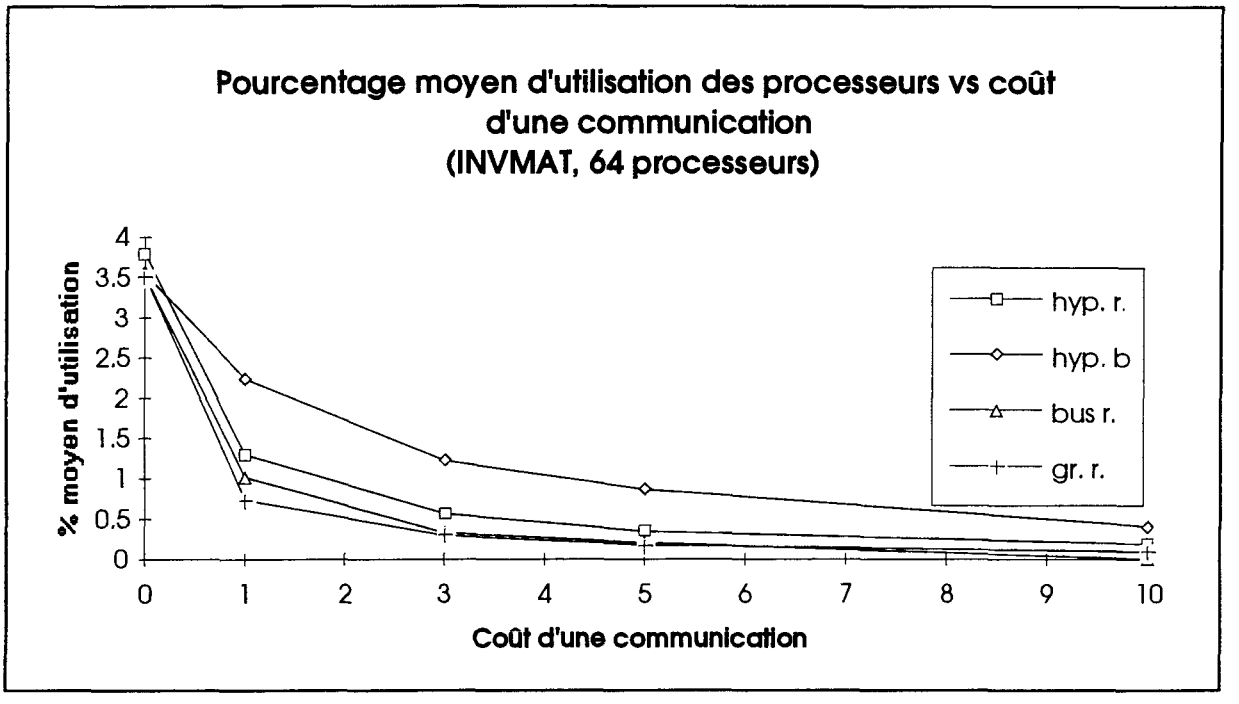

Figure A4.30 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application INVMAT.

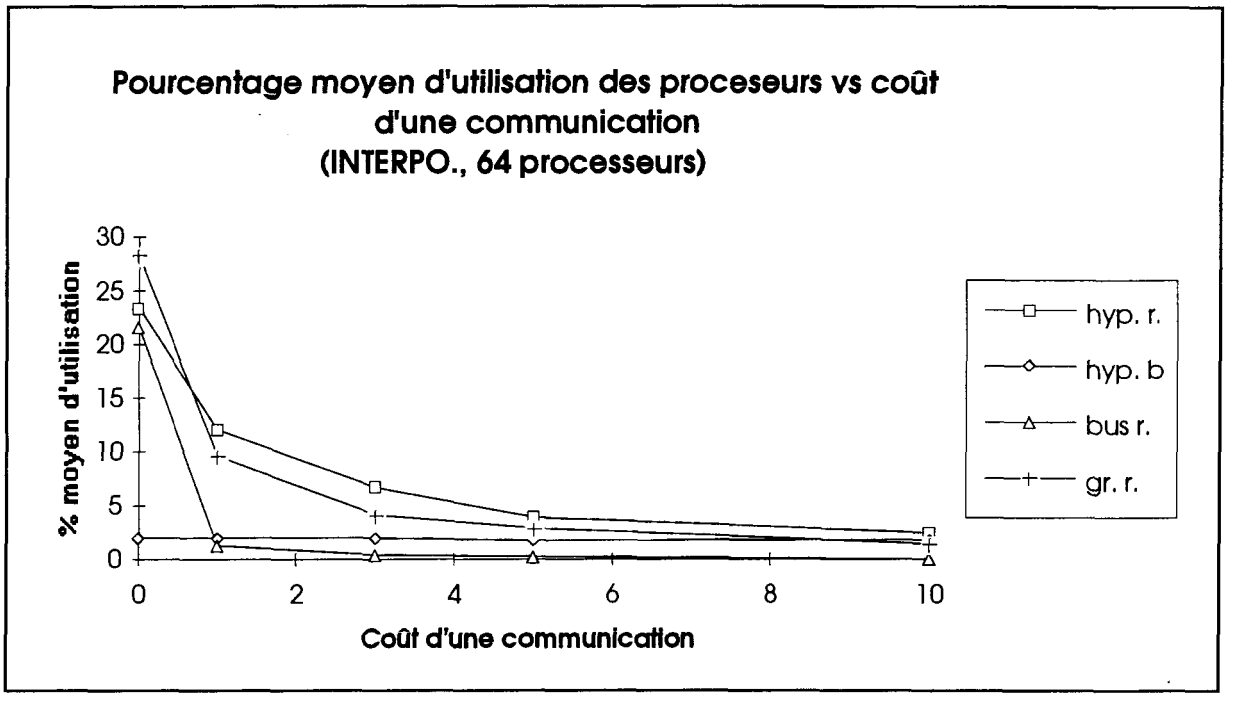

Figure A4.31 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application INTERPO.

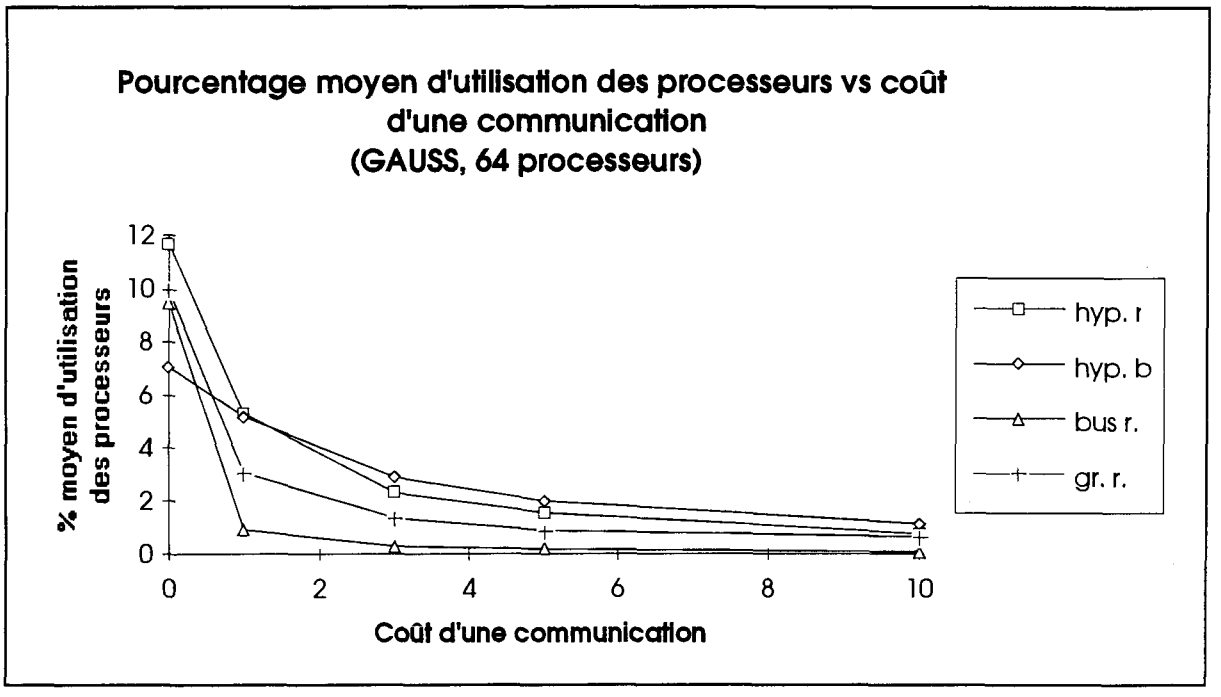

Figure A4.32 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application GAUSS.

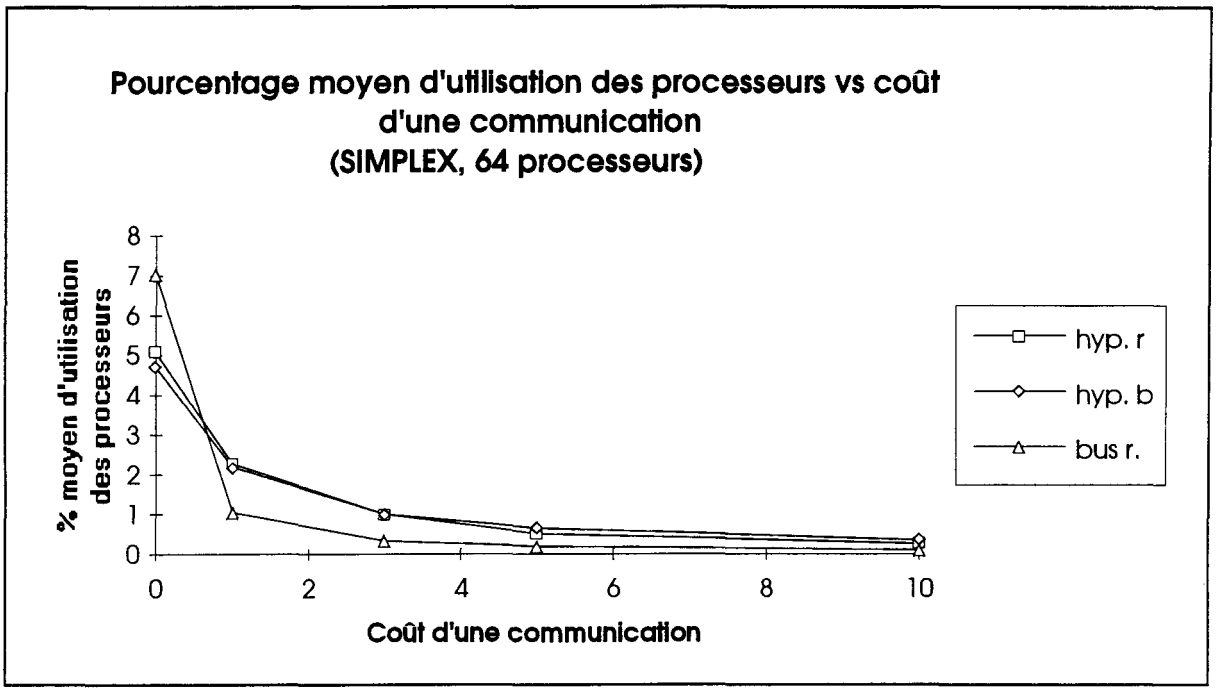

Figure A4.33 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application SIMPLEX.

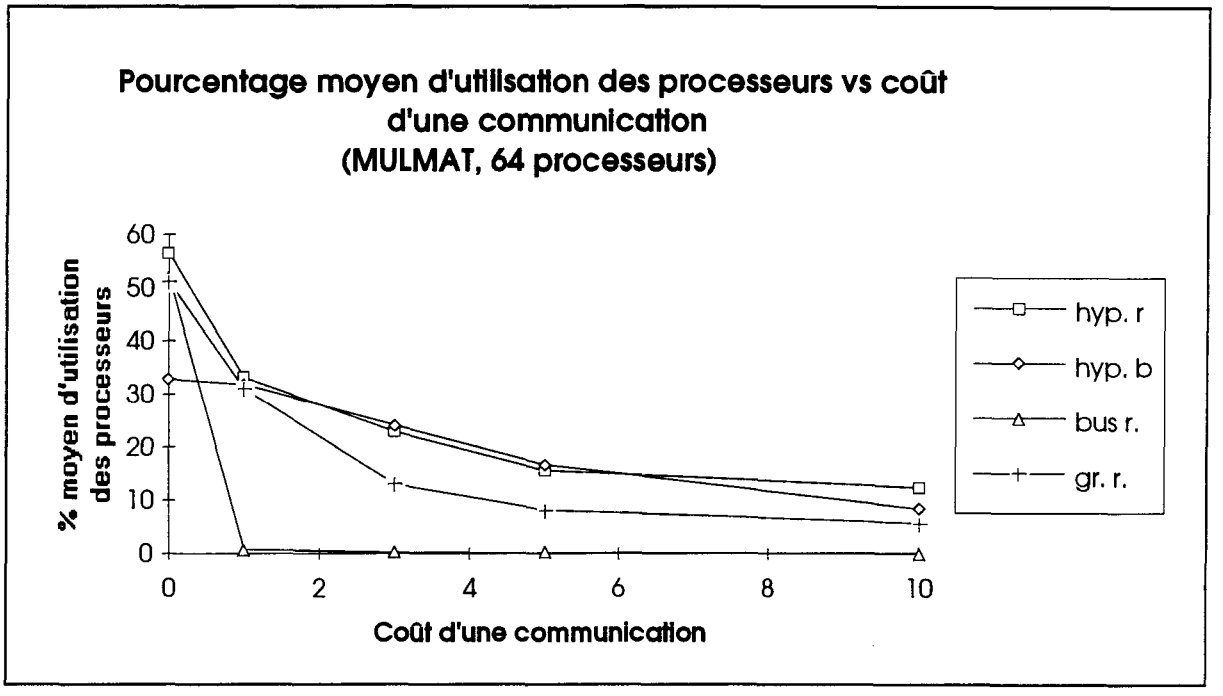

Figure A4.34 Pourcentage moyen d'utilisation des processeurs vs coûts d'une communication pour l'application MULMAT.

Liste des graphiques du pourcentage moyen d'utilisation des processeurs vs nombre de processeurs pour toutes les architectures testées

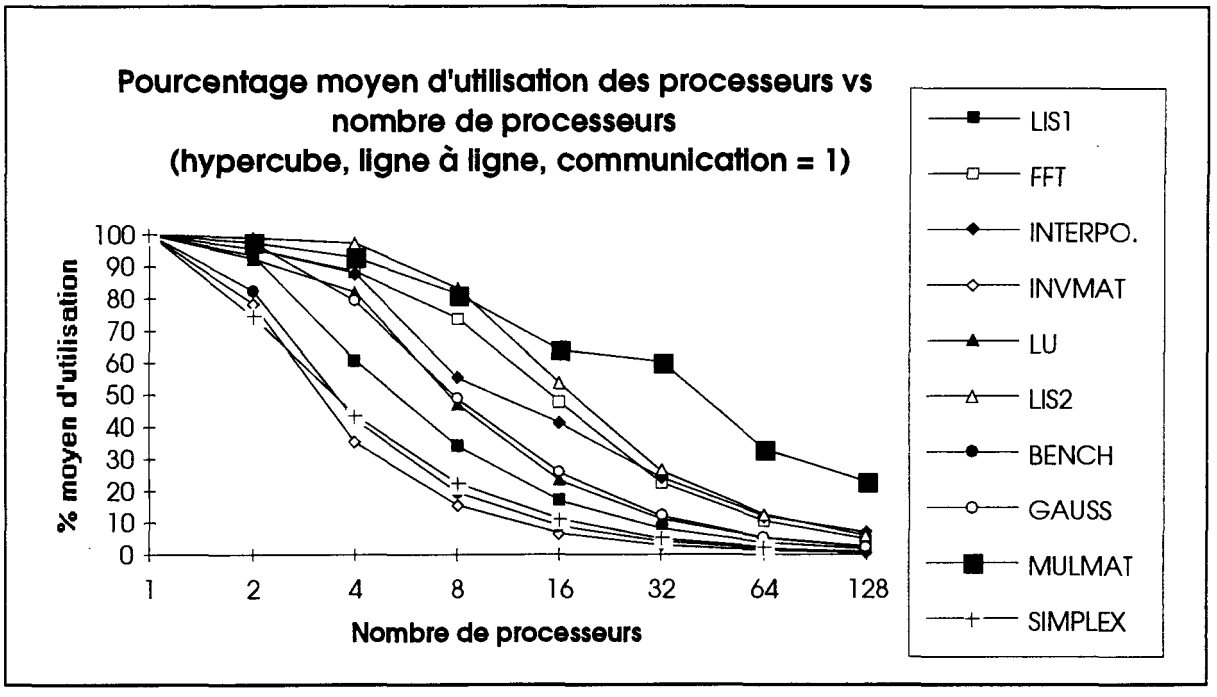

Figure A4.35 Pourcentage moyen d'utilisation des processeurs vs nombre de processeurs pour une architecture hypercube, une allocation ligne à ligne aléatoire et un coût de communication de 1.

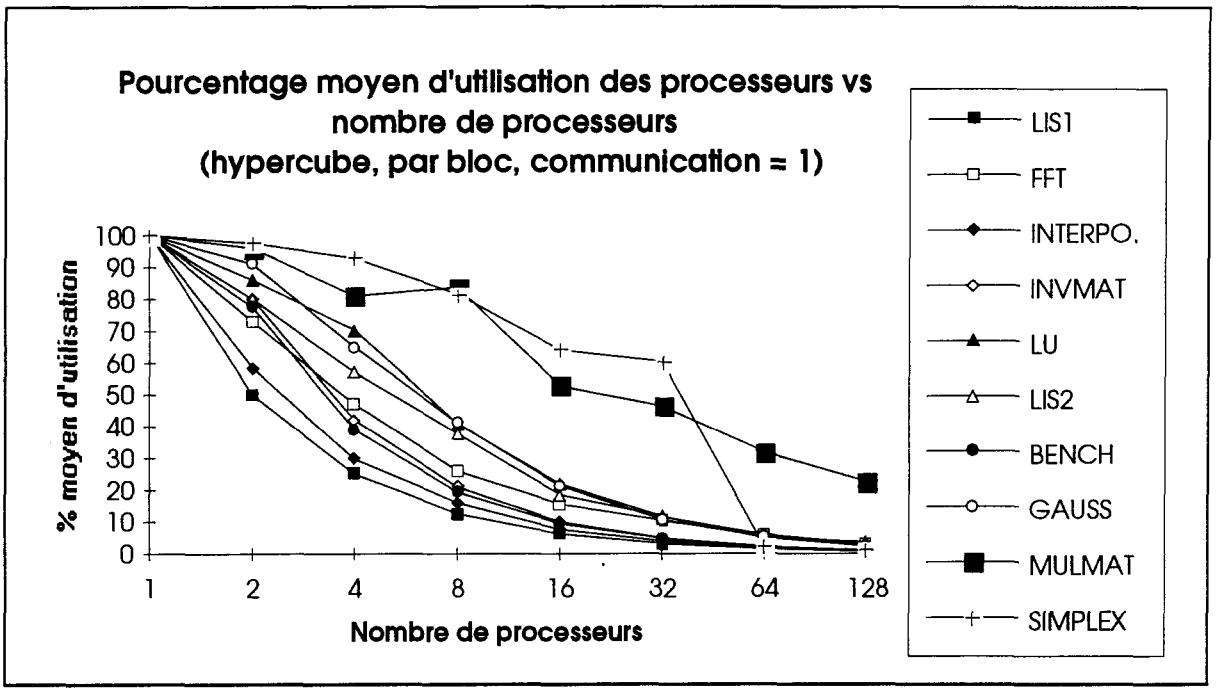

Figure A4.36 Pourcentage moyen d'utilisation des processeurs vs nombre de processeurs pour une architecture hypercube, une allocation par bloc aléatoire et un coût de communication de 1.

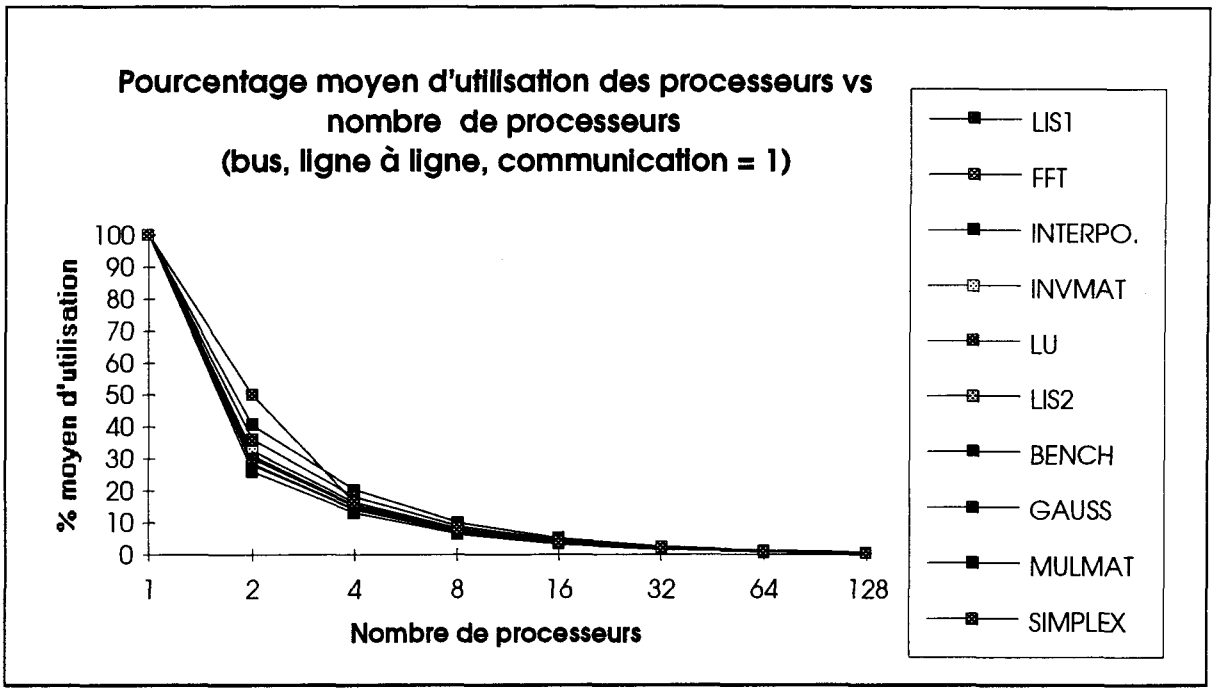

Figure A4.37 Pourcentage moyen d'utilisation des processeurs vs nombre de processeurs pour une architecture en bus, une allocation ligne à ligne aléatoire et un coût de communication de 1.

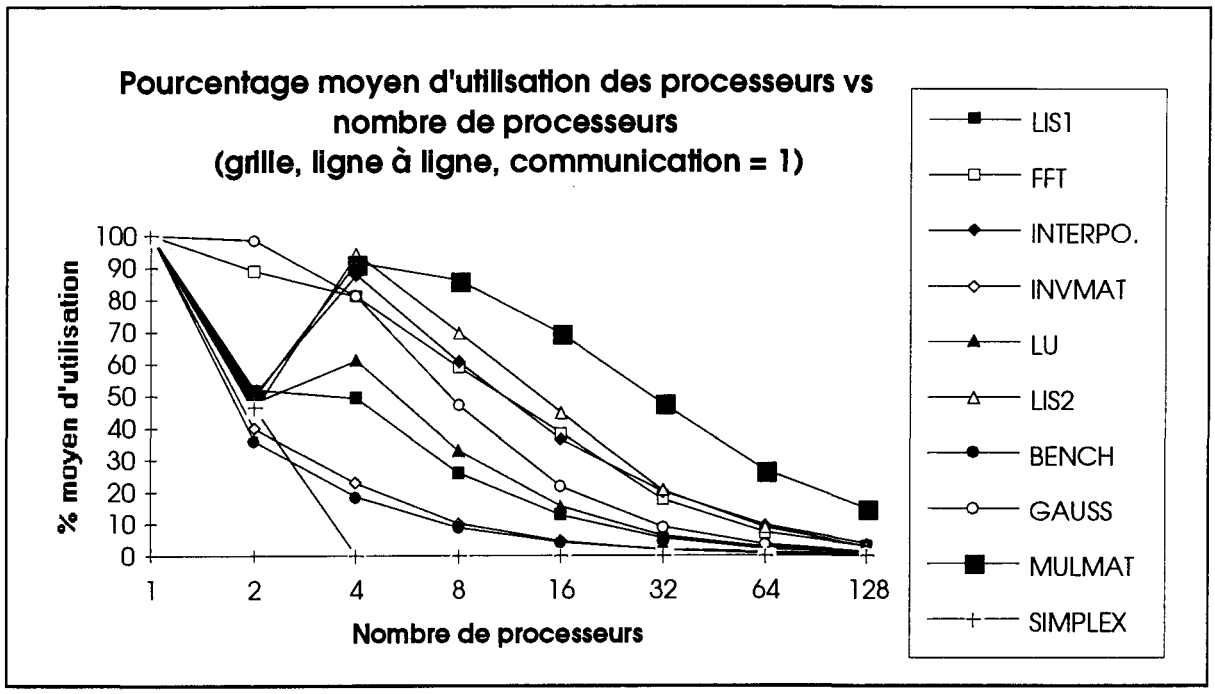

Figure A4.38 Pourcentage moyen d'utilisation des processeurs vs nombre de processeurs pour une architecture en grille, une allocation ligne à ligne aléatoire et un coût de communication de 1.

Liste des graphiques de la métrique PG vs nombre de processeurs pour toutes les architectures testées

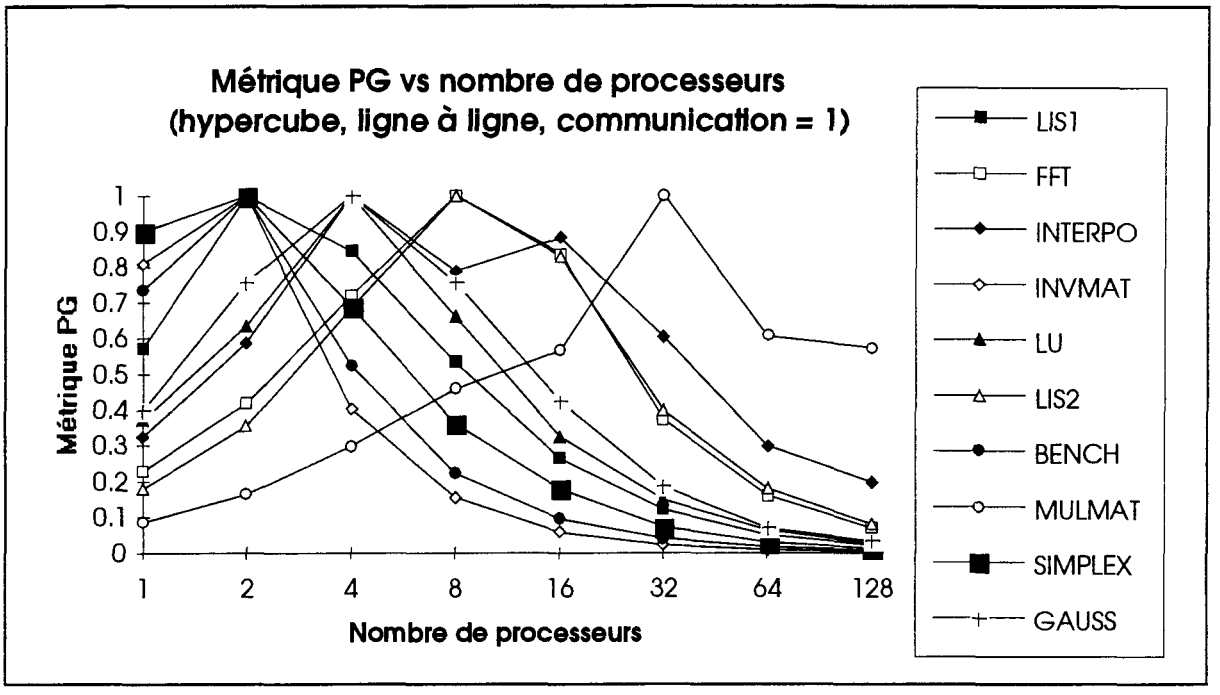

Figure A4.39 Métrique PG vs nombre de processeurs pour une architecture hypercube, une allocation ligne à ligne aléatoire et un coût de communication de 1.

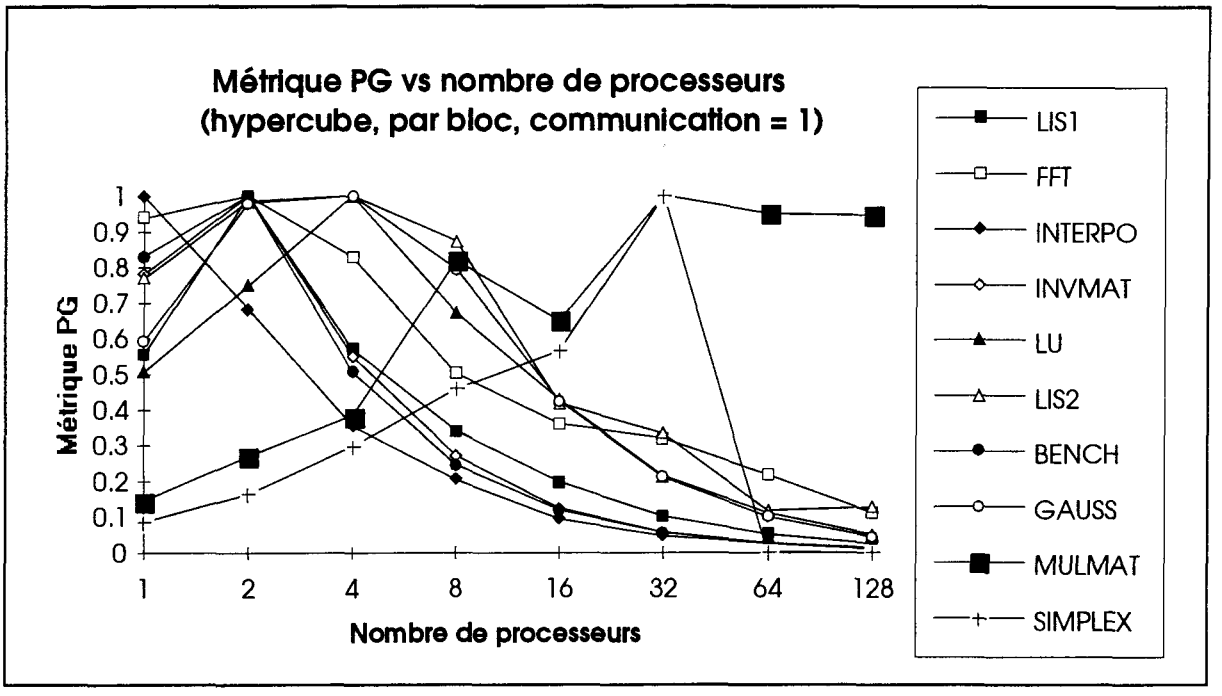

Figure A4.40 Métrique PG vs nombre de processeurs pour une architecture hypercube, une allocation par bloc aléatoire et un coût de communication de 1.

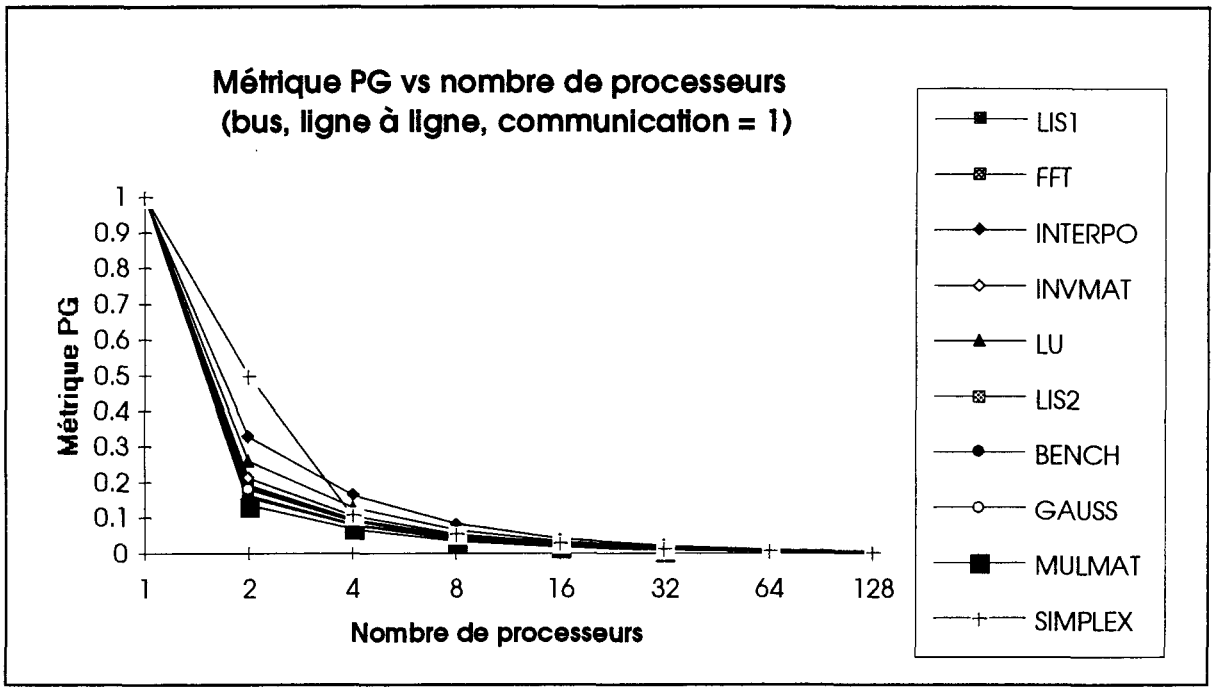

Figure A4.41 Métrique PG vs nombre de processeurs pour une architecture en bus, une allocation ligne à ligne et un coût de communication de 1.

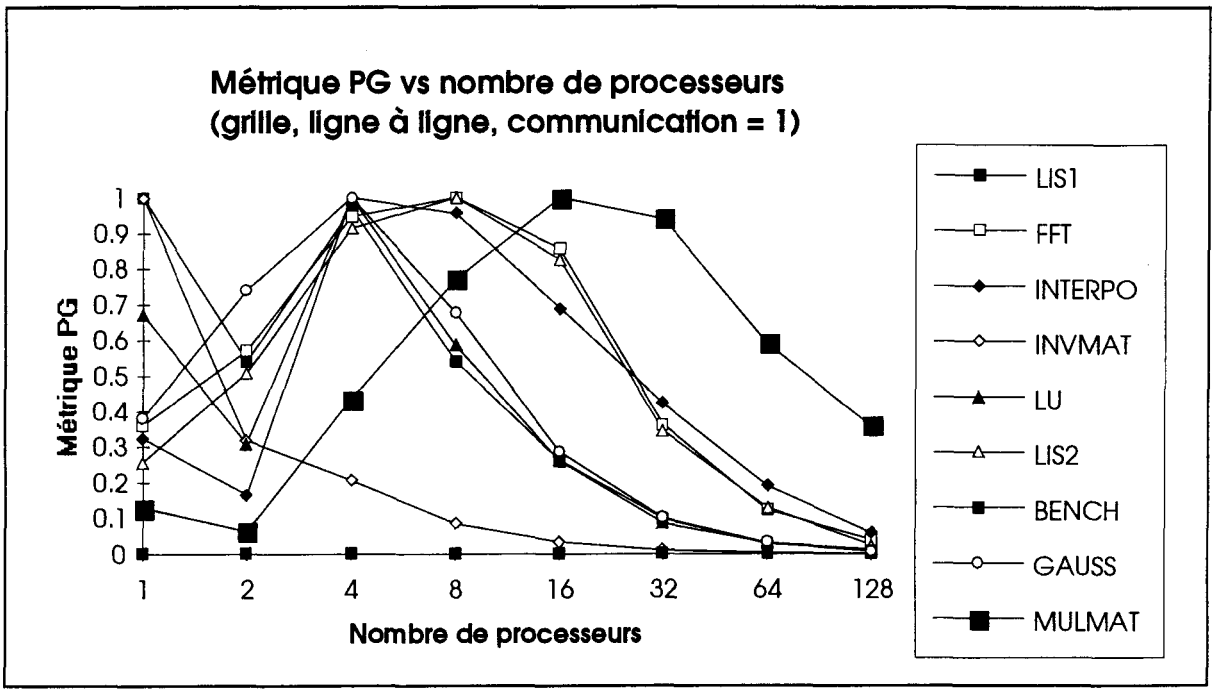

Figure A4.42 Métrique PG vs nombre de processeurs pour une architecture en grille, une allocation ligne à ligne et un coût de communication de 1.