Thèse de doctorat

Présentée en vue de l'obtention du titre de

Docteur de SUPAERO

Spécialité Informatique

Une algorithmique adaptée à la distribution pour la résolution de problèmes irréguliers

---

Par Olivier Poitou

Soutenue publiquement le 2 juillet 2003, devant le jury composé de :

Pr. Jacques Chassin de Kergommeaux	Rapporteur
Pr. Bernard Toursel	Rapporteur
Pr. Pierre Manneback	Examinateur
Pr. Jean-Marie Garcia	Examinateur
Pr. Patrice Cros	Examinateur
Pr. Bernard Lecussan	Directeur de thèse

à Virginie, à Tom

à mes parents

Remerciements

La réalisation d'une thèse est un travail qui ne se fait pas seul et les personnes qui m'ont aidé à accomplir la mienne sont nombreuses. Certains noms n'apparaîtront certainement pas ici, mais que ceux qui les portent ne s'en offusquent pas, je leur suis tout de même très reconnaissant.

D'un point de vue personnel, je tiens d'abord à remercier Virginie, ma compagne, pour sa présence à mes côtés. Je lui dédie cette thèse ainsi qu'à notre fils Tom et à mes Parents. Je tiens particulièrement à ce qu'ils sachent que ce travail est en partie le leur. Je n'oublierai pas non plus d'adresser un très grand merci à mon frère et à mes amis, toujours là eux-aussi, Sylvain, Guillaume, Thierry, Jérôme, Sabine, Amandine, Sophie et tous les autres.

D'un point de vue professionnel, mes premiers remerciements reviendront, bien entendu, à Bernard Lecussan qui a dirigé ma thèse et m'a accordé sa confiance en me laissant une grande liberté dans mon travail de recherche. Je dois aussi beaucoup à Sébastien Bermes qui m'a permis d'utiliser son travail sur le lancer de rayon paresseux. Je n'oublierai pas non plus Christophe Coustet, pour ses interventions toujours à propos.

Je remercie aussi Jacques Chassin de Kergommeaux et Bernard Toursel, rapporteurs de ma thèse, pour la qualité de leur relecture et de leur remarques. J'adresse un remerciement supplémentaire à Jacques Chassin de Kergommeaux pour la gentillesse avec laquelle il a répondu à mes nombreuses questions. Je suis reconnaissant à l'ensemble des membres du jury de me faire l'honneur d'assurer ce rôle.

Dire que mes collègues du CERT m'ont bien accueilli est un euphémisme, je les remercie tous. J'ai une pensée particulière pour Nourrédine Hifdi dont la gentillesse et l'intelligence ont éclairé de nombreuses journées, ainsi que Patrice Cros, Paul Bourret, Martin Adelantado. Quant à Béatriz, Eric et Xavier, je ne sais pas comment les remercier de tous les bons moments passés avec eux, j'espère que ce n'est qu'un début.

Ces remerciements ne seraient pas complets si je ne remerciais pas les personnes que j'ai souvent inquiétées, parfois avec raison, en tous les cas que j'ai beaucoup sollicitées et qui m'ont beaucoup apportées : les administrateurs des différentes machines que j'ai utilisées pour mes tests. Un grand merci donc à Jean-Yves Rousselot, Christian Réau, Régine Leconte et David Gauchard.

Résumé

Cette étude concerne la distribution d'applications sur des machines de type réseau d'ordinateurs ou grappe de processeurs. Ce type de machine est observé de manière théorique puis de manière pratique, à travers différentes mesures. Le contexte des applications irrégulières est privilégié, leur distribution posant encore souvent des problèmes d'efficacité. La distribution des applications régulières est abordée plus succinctement. Une démarche de développement d'application distribuée ainsi que des outils de programmation sont proposés. Il est notamment proposé d'utiliser la paresse comme outils de distribution des données ainsi que d'effectuer un découpage récursif et dynamique des calculs au fur et à mesures des assignations de tâches. Ces propositions sont ensuite validées par l'étude d'un cas concret, le lancer de rayon distribué. Enfin l'applicabilité des ces propositions à des cas plus généraux est étudiée et des critères sont dégagés, permettant d'évaluer l'opportunité d'utiliser ces outils originaux en fonction des caractéristiques de l'application à distribuer.

Mots clés : grappe de processeurs, réseaux d'ordinateurs, applications irrégulières, distribution, paresse, lancer de rayon distribué.

Abstract

This study introduces the main characteristics of the architecture of the network of workstations (NOW) or cluster machines and their consequences on program coding. A first theoretical presentation of this architecture is made and then a more practical one enforces known results by concrete experimentations. This survey mainly focuses on the distribution of irregular applications though the distribution of regular applications is also discussed. A program coding strategy along with original tools is proposed and then validated with a practical example: a distributed ray tracing application. An automatic data distribution mechanism, based on the use of lazy algorithms, and a dynamic recursive task splitting are the two main tools introduced in this work. Finally, an extent to more general cases is made and some criteria are given to evaluate the opportunity to use these original tools depending on the characteristics of the application to distribute.

Keywords: cluster computing, network of workstations, irregular applications, algorithm distribution, laziness, distributed ray tracing.

Table des matières

REMERCIEMENTS **5**

RESUME **6**

ABSTRACT **6**

TABLE DES MATIERES **7**

TABLE DES EQUATIONS **11**

TABLE DES EXEMPLES **12**

TABLE DES FIGURES **13**

CHAPITRE I : INTRODUCTION **16**

1. ENVIRONNEMENT DE RECHERCHE 17
2. PRESENTATION DE L'ETUDE 18
3. INTRODUCTION AUX RESEAUX D'ORDINATEURS 19
   3.1. Bref historique *19*
   3.2. Définitions *21*
4. DEFINITION DE LA REGULARITE 22
   4.1. Les applications régulières *22*
   4.2. Les applications irrégulières *23*

CHAPITRE II : REFERENCES DU MONDE DU PARALLELISME **26**

1. GENERALITES 27
   1.1. Machines de références *27*
      1.1.1. IBM SP2 28
      1.1.2. CRAY T3E 29
      1.1.3. Points communs 30
   1.2. Modèles d'exécution *30*
2. MEMOIRE PARTAGEE VIRTUELLE 32
   2.1. Mémoire physiquement partagée *33*
      2.1.1. Cohérence des données dans un environnement parallèle 33
      2.1.2. Cohérence par désactivation des mémoires caches 34
      2.1.3. Cohérence par espionnage du bus mémoire 34
   2.2. Mémoire physiquement distribuée *35*
      2.2.1. Principes de la mémoire virtuellement partagée 36
      2.2.2. Les modèles de mémoire virtuellement partagée 37
      2.2.3. Conclusion 42
3. DISTRIBUTION DES DONNEES ET ACCES DISTANT 43
4. CHOIX DE L'ALGORITHME SEQUENTIEL 45
5. CHOIX DE L'ORDONNANCEMENT 47

CHAPITRE III : L'ARCHITECTURE « RESEAU D'ORDINATEURS » **50**

1. CARACTERISTIQUES GENERALES DES RESEAUX D'ORDINATEURS 51
2. LIMITES POTENTIELLES DES PERFORMANCES D'UN RESEAU DE MACHINES 52

2.1. Le matériel ... *52*
2.1.1. Généralités ... 52
2.1.2. Matériel réseau ... 53
2.2. Les pilotes de périphériques ... *55*
2.3. Le système d'exploitation ... *55*
2.4. Le logiciel ... *56*
3. LIMITES REELLES OBSERVEES SUR UN RESEAU DE MACHINES ... 57
3.1. Mesures de performances générales ... *57*
3.1.1. Les messages de taille inférieure au kilo-octet ... 59
3.1.2. Les messages de taille supérieure au kilo-octet ... 59
3.2. Coût mesuré du débit ... *60*
3.3. Coût mesuré de la latence ... *63*
3.3.1. Mesures de la latence ... 63
3.3.2. Proportion d'information utile dans un message ... 64
3.3.3. Coûts additionnels ... 64
3.4. Conclusion sur les performances des communications ... *65*
3.5. Impact de ces deux critères sur un cas concret : la multiplication de matrices ... *66*
3.5.1. Distribution selon une granularité minimum ... 67
3.5.2. Distribution selon une granularité moyenne ... 68
3.5.3. Distribution selon une granularité épaisse ... 70
3.5.4. Conclusion ... 73

CHAPITRE IV : UNE APPROCHE EFFICACE DE LA DISTRIBUTION ... 76
1. PRINCIPES ... 77
1.1. Mieux exploiter le débit ... *77*
1.2. Réduire l'impact des latences ... *78*
2. OUTILS CLASSIQUES ... 78
2.1. Le recouvrement ... *78*
2.2. L'anticipation ... *80*
2.3. Conclusion ... *82*
3. NOUVEAUX OUTILS ... 83
3.1. La Paresse ... *83*
3.1.1. Définition de la paresse ... 83
3.1.1.1. Première propriété : non-évaluation des paramètres inutiles ... 84
3.1.1.2. Seconde propriété : manipulation de structures infinies possible ... 85
3.1.1.3. Source de paresse ... 86
3.1.2. Paresse et distribution ... 87
3.2. La granularité adaptable ... *88*
3.2.1. Généralités ... 88
3.2.2. Les algorithmes irréguliers ... 89
3.2.2.1. Pourquoi il faut une granularité la plus fine possible ... 89
3.2.2.2. Pourquoi il faut une granularité la plus épaisse possible ... 89
3.2.3. Solution proposée ... 89
3.3. Le regroupement de messages ... *90*
4. CHOIX DE L'ALGORITHME A DISTRIBUER ... 91
4.1. Un calcul plus adapté de la complexité temporelle ... *91*
4.1.1. Intégration de la complexité des communications ... 91
4.1.2. Seuil de rentabilité d'un algorithme de moindre complexité ... 92
4.2. Cas concret : La Transformation de Fourier d'ordre deux ... *93*
4.2.1. La transformée de Fourier séquentielle ... 93
4.2.1.1. Transformée d'ordre un ... 93
4.2.1.2. Transformée d'ordre deux ... 94
4.2.2. La transformée de Fourier parallèle ... 95

4.2.3. Conclusion ... 97
5. DEMARCHE DE DISTRIBUTION ... 98

CHAPITRE V : LANCER DE RAYON ... 102
1. PRESENTATION DE L'ETUDE DE CAS ... 103
2. PREMIERE ETAPE : CHOIX DE L'ALGORITHME SEQUENTIEL ... 103
2.1. Principes de base de l'algorithme de lancer de rayon ... 103
2.2. Optimisations classiques de l'algorithme précédent ... 107
2.3. Algorithme paresseux et justification de ce choix ... 109
3. SECONDE ETAPE : CHOIX DU MODE DE DISTRIBUTION ... 115
3.1. Deux caractéristiques essentielles du lancer de rayon ... 115
3.2. Différentes manières de distribuer le lancer de rayons ... 115
3.3. Méthode choisie et raisons de ce choix ... 117
4. RESULTATS ET MESURES ... 117
4.1. Description des indicateurs utilisés ... 117
4.2. Mesures commentées ... 121
4.2.1. Conditions expérimentales ... 121
4.2.2. Algorithmes de test ... 122
4.2.3. Mesures ... 123
4.2.3.1. Proportion de code séquentiel ... 123
4.2.3.2. Efficacité ... 125
4.2.3.3. Déséquilibre de charge ... 127
4.2.3.4. Mémoire utilisée ... 130
4.2.3.5. Pourcentage de communication ... 132
4.2.4. Conclusion ... 135
4.2.4.1. En séquentiel ... 135
4.2.4.2. En distribué ... 135
5. TESTS ET OPTIMISATIONS ... 136
5.1. Influence de la taille des blocs ... 136
5.2. Troisième étape : Le regroupement de messages ... 141
5.3. Quatrième étape : Le mode de communication ... 142
6. CONCLUSION ET COMPARAISON AVEC DES RESULTATS EXTERIEURS ... 143

CHAPITRE VI : BILAN ET GENERALISATION ... 146
1. PARESSE ... 147
1.1. Condition d'efficacité de la paresse ... 147
1.2. Applicabilité de la paresse ... 148
1.2.1. L'AMR distribué ... 148
1.2.2. Barnes-Hut ... 149
1.2.3. Conclusion ... 151
2. GRANULARITE ADAPTABLE ... 152
2.1. Avantage de la granularité adaptable ... 152
2.2. Condition d'utilisation de la granularité adaptable ... 152
3. REGROUPEMENT DE MESSAGES ... 156
4. MODE DE COMMUNICATION ... 157
5. SYSTEME ET PILOTES DE PERIPHERIQUES ... 157

CHAPITRE VII : CONCLUSION ET PERSPECTIVES ... 160
1. CONCLUSION ... 161
2. PERSPECTIVES : VERS PLUS DE FLEXIBILITE ... 161
2.1. Amnésie ... 161

2.1.1. Ce qui est exprimé 161
2.1.2. Comment reconnaître un algorithme susceptible de bénéficier de l'amnésie ? 162
2.1.3. Intérêt de l'amnésie 162
2.2. Interruptibilité 163
2.2.1. Ce qui est exprimé 163
2.2.2. Comment reconnaître un algorithme susceptible de bénéficier de l'interruptibilité ? 163
2.2.3. Intérêts de l'interruptibilité 164

CHAPITRE VIII : REFERENCES 166

BIBLIOGRAPHIE 174
1. ARTICLES ET JOURNAUX 174
2. OUVRAGES COMPLETS 180
3. SITES INTERNET 180

ANNEXE I. MACHINES UTILISEES POUR LES MESURES 182
1. LE RESEAU DE STATIONS SUN 183
2. LA GRAPPE DU DTIM 184
3. LA GRAPPE DE SUPAERO 185
4. LA GRAPPE GRAPHIQUE 186

ANNEXE II. FICHES DE MESURES 188
1. UNE SCENE DE PETITE TAILLE : THE TEAPOT 190
2. UNE SCENE DE TAILLE MOYENNE : THE MOUNT 192
3. UNE SCENE VOLUMINEUSE : THE TREE 194
4. UNE SCENE DEPASSANT LA CAPACITE MEMOIRE : THE GEARS 196
5. VERS LE LANCER DE RAYON EN TEMPS REEL : THE TETRA 198

Table des équations

Équation 1 : Temps d'accès moyen aux données 34
Équation 2 : Notation usuelle de la complexité algorithmique 46
Équation 3 : Complexité parallèle idéale 46
Équation 4 : Transformée de Fourier Discrète d'ordre un 93
Équation 5 : Transformée de Fourier Discrète d'ordre deux 94
Équation 6 : Expression matricielle compacte d'une 2D DFT 96
Équation 7 : Complexité minimale de la 2D FFT pour une matrice NxN 97
Équation 8 : Complexité du nouvel algorithme de 2D DFT pour une matrice NxN 97
Equation 9 : Temps d'exécution parallèle 118
Equation 10 : Efficacité parallèle 118
Equation 11 : Temps minimum théorique 118
Equation 12 : Déséquilibre de charge entre les nœuds de calcul 119
Équation 13 : Consommation mémoire d'un programme parallèle 119
Équation 14 : Indicateur de pourcentage de communication 120
Equation 15 : Décomposition du temps global de calcul de la version stricte 133

Table des exemples

Exemple 1 : Champ d'action de la cohérence 33
Exemple 2 : Débits et latences annoncés de quelques réseaux 53
Exemple 3 : Comparaison entre les valeurs annoncées et mesurées de quelques réseaux .. 57
Exemple 4 : Principe du recouvrement 79
Exemple 5 : Principe de l'anticipation 81
Exemple 6 : Une fonction n'utilisant pas tous ces arguments 84
Exemple 7 : Définition d'une liste infinie 85
Exemple 8 : Exemples d'images produites par lancer de rayons 104
Exemple 9 : Conséquence de l'estimation de la granularité de départ 155

Table des figures

Figure 1 : Aspect extérieur de l'IBM SP 28
Figure 2 : Aspect extérieur du CRAY T3E 29
Figure 3 : Protocole MESI simplifié 35
Figure 4 : Principe de la mémoire virtuellement partagée 36
Figure 5 : Temps de communication mesuré sur la grappe du DTIM (Réseau Myrinet) 58
Figure 6 : Temps de communication mesuré sur la grappe du DTIM (réseau Fast Ethernet) 58
Figure 7 : Temps de communication mesuré sur la grappe Graphique (Fast Ethernet) 58
Figure 8 : Temps de communication mesuré sur la grappe de Supaero (Fast Ethernet) 58
Figure 9 : Temps de communication mesuré sur le réseau de stations Sun (Ethernet) 58
Figure 10 : Courbes de mesure du temps de communication sur des machines réelles 58
Figure 11 : Débit mesuré sur réseau de type Fast Ethernet 60
Figure 12 : Débit de la grappe du DTIM (réseau Myrinet) 62
Figure 13 : Débit de la grappe graphique (Fast Ethernet) 62
Figure 14 : Débit de la grappe Supaero (Fast Ethernet) 62
Figure 15 : Débit du réseau de stations Sun du DTIM (Ethernet) 62
Figure 16 : Courbes de mesure du débit réseau sur quatre machines réelles 62
Figure 17 : Estimation de la latence par des mesures 63
Figure 18 : Détails d'une distribution de granularité trop fine 67
Figure 19 : Détails d'une distribution de granularité moyenne 68
Figure 20 : Algorithme distribué de multiplication de matrices 2 69
Figure 21 : Chronogramme d'un algorithme à granularité moyenne 70
Figure 22 : Détails d'une distribution de granularité épaisse 70
Figure 23 : Algorithme distribué de multiplication de matrices 3 71
Figure 24 : Chronogramme d'un algorithme à découpage grossier 72
Figure 25 : Chronogramme d'un algorithme utilisant la fonction diffusion de MPICH 73
Figure 26 : flot d'exécution sans recouvrement 79
Figure 27 : flot d'exécution avec recouvrement 79
Figure 28 : flot d'exécution sans anticipation 81

Figure 29 : flot d'exécution avec anticipation 81
Figure 30 :Constructeur de liste paresseux et non paresseux 83
Figure 31 : Evaluation stricte et paresseuse d'une même expression 84
Figure 32 : Comparaison algorithme statique – algorithme dynamique 86
Figure 33 : Parallélisation classique de l'algorithme de transformée de Fourier d'ordre deux 95
Figure 34 : Principe intuitif du lancer de rayon 105
Figure 35 : Principe retenu du lancer de rayon 106
Figure 36 : voxélisation régulière (2D) 108
Figure 37 : voxélisation arborescente (2D) 108
Figure 38 : Comparaison des deux types de voxélisation 108
Figure 39 : Comparaison entre la taille du modèle et celle de son arbre octal 110
Figure 40 : Arbre octal statique 111
Figure 41 : Arbre octal paresseux 111
Figure 42 : Deux méthodes de construction de l'arbre octal 111
Figure 43 : Comparaison arbre octal statique - arbre octal paresseux 113
Figure 44 : Algorithme paresseux de lancer de rayon 114
Figure 45 : Proportion fonctionnelle partie séquentielle – partie parallèle 123
Figure 46 : Proportion quantitative partie séquentielle – partie parallèle 124
Figure 47 : Efficacité (Equation 10) 125
Figure 48 : Déséquilibre de charge (Equation 12) 127
Figure 49 : Répartition de la charge au sein d'une scène 129
Figure 50 : Consommation moyenne en mémoire locale par nœud de calcul 130
Figure 51 : Taux de communication des différents algorithmes 132
Figure 52 : Temps d'exécution algorithmes strict et paresseux 133
Figure 53 : Temps de construction de l'arbre octal strict 134
Figure 54 : Déséquilibre de charge sur huit nœuds (découpage statique) 137
Figure 55 : Taux de communication sur huit nœuds (découpage statique) 138
Figure 56 : Comparaison des taux de communication sur huit nœuds 139
Figure 57 : Echange de messages de l'algorithme à découpage statique 140
Figure 58 : Echanges de messages de l'algorithme à découpage dynamique 140
Figure 59 : Envoi des premières tâches avec les fonctions classiques 143
Figure 60 : Envoi des premières tâches avec les fonctions non bloquantes 143

Figure 61 : Une station Sun Ultra10 183
Figure 62 : La grappe du DTIM 184
Figure 63 : La grappe de Supaero 185
Figure 64 : La grappe graphique 186
Figure 65 : Fiche de test de la scène Teapot 12 190
Figure 66 : Fiche de test de la scène Mount 8 192
Figure 67 : Fiche de test de la scène Tree 10 194
Figure 68 : Fiche de test de la scène Gears 15 196
Figure 69 : Fiche de test de la scène Tetra 6 198

Chapitre I : Introduction

Sommaire

1. ENVIRONNEMENT DE RECHERCHE 17
2. PRESENTATION DE L'ETUDE 18
3. INTRODUCTION AUX RESEAUX D'ORDINATEURS 19
3.1. Bref historique 19
3.2. Définitions 21
4. DEFINITION DE LA REGULARITE 22
4.1. Les applications régulières 22
4.2. Les applications irrégulières 23

Ce chapitre va présenter le contexte de mes travaux, la forme qu'ils ont prise et la problématique générale de ceux-ci. La définition de certains termes sera rappelée ainsi que quelques notions qui revêtent une importance particulière pour cette étude.

1. Environnement de recherche

J'ai effectué mes recherches à l'ONERA Centre de Toulouse, le principal domaine d'activité de ce laboratoire est l'aéronautique et les applications qui en sont dérivées. L'aéronautique nécessite des recherches dans plusieurs domaines de la physique comme la mécanique des fluides ou le traitement du signal. Ces disciplines produisent en permanence des problèmes dont la résolution nécessite une puissance de calcul très importante, le département informatique est donc chargé d'apporter des solutions, matérielles et logicielles, permettant de résoudre ces problèmes dans les meilleures conditions possibles.

L'équipe MaRT à laquelle j'ai été rattaché avait déjà produit de très bons résultats par le passé à travers la librairie de calcul MaRT destinée à la simulation de parcours d'onde dans un environnement en trois dimensions. Cette librairie a été intégrée, entre autres, à un logiciel de synthèse d'images réalistes par lancer de rayons et à un logiciel d'électromagnétisme dont le thème principal est la recherche par calcul du meilleur émetteur GPS dans un environnement urbain.

Des besoins en puissance de calcul sans cesse grandissant ont naturellement amené l'ONERA à s'intéresser au calcul distribué sur réseaux de machines. Plusieurs projets en ce sens ont donc vu le jour dans le laboratoire.

Lors de mes recherches j'ai été amené à travailler en collaboration avec le laboratoire d'informatique et d'automatisme de SUPAERO, dont les orientations sont très proches de celles de l'ONERA ainsi qu'avec le CNRS-LAAS pour les aspects plutôt système du calcul distribué. Ces collaborations ont été fructueuses et ont fait l'oeuvre d'une présentation commune de nos outils de calcul lors du salon du SITEF en 2001.

2. Présentation de l'étude

L'étude présentée dans ce document concerne la distribution efficace d'applications irrégulières sur un réseau d'ordinateurs. Pour mener cette étude, plusieurs points devront être abordés.

Dans un tout premier temps les particularités des applications irrégulières devront être mises à jour afin de mieux comprendre les difficultés actuellement rencontrées pour les distribuer efficacement. Leur comportement n'est pas aussi prévisible que celui des applications régulières ce qui a tendance à rendre peu efficace les outils traditionnellement utilisés sur ces dernières. Une caractérisation de ces deux types d'applications figure dans la suite de ce chapitre, ainsi que quelques définitions et un bref historique des réseaux d'ordinateurs (visant, entre autre, à justifier le choix de cette architecture comme cible privilégiée des travaux rapportés dans ce document).

Il conviendra de définir les caractéristiques d'une machine de type « réseau d'ordinateurs ». La parenté de ces machines avec les supercalculateurs des grandes enseignes du monde du calcul intensif ne fait aucun doute mais leurs particularités suffisent à rendre différente leur exploitation efficace. L'exploration de cet aspect constitue les chapitres II et III de ce document. Je profiterai, en outre, du chapitre III pour vérifier quelques résultats connus par des mesures réelles effectuées sur les différentes machines auxquelles j'ai pu avoir accès.

L'efficacité de la distribution d'une application peut être mesurée selon différents critères, le premier d'entre eux étant évidemment le temps de calcul. Des outils existent permettant d'améliorer l'efficacité du programme distribué, une rapide présentation de ceux ci sera faite au début du chapitre IV. Je proposerai ensuite deux nouveaux outils s'adressant plus particulièrement au traitement distribué des applications irrégulières, il s'agit de la paresse et de la granularité adaptable. Le sujet principal de cette étude étant la distribution d'algorithmes (irréguliers) séquentiels (même si certains aspects sont aussi valables dans le cadre d'un travail sur des algorithmes purement parallèles), je me pencherai aussi sur les critères de choix de l'algorithme de départ. Je proposerai finalement une démarche de distribution hiérarchisant les diverses étapes et outils associés.

Une mise en application suivra immédiatement les considérations théoriques des chapitres précédents. Un logiciel de lancer de rayon (sujet très présent dans le domaine) sera produit, testé et optimisé en utilisant les bases théoriques retenues. Ces résultats pratiques valideront, dans le chapitre V, l'approche plus abstraite développée précédemment.

Le bilan, l'étude du champ d'application des méthodes et les perspectives offertes par cette étude viennent clôturer ce document dans les chapitres VI et VII .

3. Introduction aux réseaux d'ordinateurs

3.1. Bref historique

Depuis quelques années nous assistons à une montée en puissance des ordinateurs individuels, ces derniers offrent à présent des puissances de calcul faramineuses, plusieurs centaines de mégaflops, à un prix dérisoire. La mémoire, autre facteur critique pour l'efficacité d'une machine ainsi que pour son aptitude à traiter des problèmes de grande taille, a suivi une évolution similaire du point de vue de la capacité et ne coûte maintenant pratiquement plus rien.

Cette constatation a fait naître l'idée d'utiliser ces ordinateurs « grand public » dans le cadre du calcul scientifique intensif. Un exemplaire unique de ces machines étant tout de même insuffisant pour traiter des cas significatifs et une interconnexion entre plusieurs de ces machines étant souvent déjà en place pour des raisons de partage d'accès à l'Internet ou des espaces de travail, la distribution des calculs sur plusieurs machines est rapidement apparue comme une solution séduisante.

C'est au début des années 1990 que l'utilisation de réseaux d'ordinateurs communs (par opposition aux matériels spécialisés) a réellement pris son essor. Les différentes machines organisées en réseau homogène ou hétérogène vont des PC jusqu'au SMP en passant par les stations de travail. Les principaux avantages de cette nouvelle approche sont l'adaptabilité, l'extensibilité mais avant tout le prix. Ces machines parallèles « nouvelle génération » sont en effet construites avec des composants très ordinaires et donc de prix modique : les ordinateurs sont ceux que l'on retrouve chez n'importe quel particulier, les réseaux sont très communs aussi puisqu'une grande majorité des réseaux d'ordinateurs sont interconnectés via Ethernet, Fast Ethernet ou encore Myrinet et les composants logiciels sont Linux, Windows ou UNIX pour le système d'exploitation auquel on adjoint traditionnellement des outils de communication comme PVM ou MPI.

Un projet va bénéficier d'un engouement particulier : c'est le projet Beowulf [BEO]. Au départ ce nom est celui d'un réseau d'ordinateurs (de type grappe de processeurs) construit en 1994 au CESDIS, un laboratoire de recherche de la NASA, dans le cadre du projet Earth and Space Science [ESS] du programme High Performance Computing Project [HPCC]. La performance de cette machine originale a rapidement fait l'objet de publications vantant les avantages de cet outil de calcul [STE95]. L'équipe de recherche a finalement publié un guide expliquant la construction de A à Z d'une machine de type Beowulf à partir de machines que tout un chacun possédait d'ores et déjà. Ce guide était assorti d'un logiciel d'évaluation de performance. Le classement de l'ensemble des réseaux d'ordinateurs de

type Beowulf selon le logiciel d'évaluation était disponible en permanence sur le site Internet du projet, chacun pouvant envoyer le résultat de sa propre machine. Le succès fut important et sa participation à l'extension des réseaux d'ordinateurs ne fait aucun doute.

Le premier obstacle qui va s'opposer à la démocratisation du réseau de machines est la complexité de programmation d'une telle architecture. Cet obstacle est, aujourd'hui encore, relativement présent même si de nombreux outils ont été écrits pour faciliter l'utilisation de ce type de machine. On trouvera par exemple :

- Les bibliothèques de communications [MPI95] amenant une interface plus conviviale à cet aspect, parfois rébarbatif, de la programmation distribuée.
- Des outils d'exploitation du réseau de machines tel que le suivi de la charge de chaque machine ou du réseau [GAR97].
- Des solutions « clé en main », apparues plus récemment et regroupant l'ensemble des outils nécessaires [ALINKA].

Il n'est à mon avis pas possible de parler de réseau de machines sans parler de Linux [LDP], le système d'exploitation libre le plus utilisé actuellement. Linux a fédéré toute une communauté d'informaticiens passionnés qui ont amené une dynamique formidable dans le milieu des systèmes d'exploitation. C'est en partie grâce à cette dynamique et au principe du logiciel libre [FSF] que les outils performants que j'évoquais dans le paragraphe précédent ont pu voir le jour et être diffusés aussi rapidement. Le logiciel libre a, en outre, fait tomber un autre frein sérieux à la généralisation des architectures de type réseaux d'ordinateurs en apportant une certaine indépendance par rapport à la machine cible (Les mêmes outils sont souvent disponibles pour de nombreuses architectures).

Les réseaux de machines sont devenus une alternative sérieuse aux supercalculateurs de grandes enseignes (actuellement le cinquième calculateur le plus puissant du monde est un réseau d'ordinateurs à base de processeurs Intel [TOP02]). Le monde industriel ne s'y trompe d'ailleurs pas et des géants comme IBM ou Hewlett Packard, après avoir été quelque peu bousculés par l'arrivée des réseaux de machines (en 1999, la Amerada Hess Corporation avait préféré acheter un réseau de machines sous Linux plutôt que de continuer la location de son IBM SP2 [COM99]), sont de plus en plus impliqués dans cette voie de recherche (IBM propose actuellement des serveurs de type réseau d'ordinateurs IBM équipés du système d'exploitation Linux [IBM]).

En résumé, les réseaux d'ordinateurs forment une nouvelle approche de ce que l'on nomme parfois le supercomputing, c'est à dire le monde du calcul intensif parallèle. Ces nouvelles machines ont l'avantage d'être peu chères tout en offrant un niveau de performance élevé. Elles sont de plus évolutives et adaptables au budget comme au problème. Leurs

principaux défauts résident dans la complexité de leur programmation et la relative jeunesse des connaissances de leur fonctionnement comme des outils d'exploitation.

3.2. Définitions

Avant tout il me semble important de donner les deux définitions suivantes extraites de [TAN99] :

Un **réseau d'ordinateurs** est un ensemble interconnecté d'ordinateurs autonomes. Deux ordinateurs étant considérés comme interconnectés s'ils sont capables d'échanger de l'information.

Un **système distribué** rend transparente, invisible à l'utilisateur, la répartition en ordinateurs autonomes.

Une autre distinction à faire réside entre le terme **Grappe (ou Cluster en anglais)** et le terme **réseau de stations de travail (NOW : Network Of Workstations en anglais, les termes réseau de machines ou encore réseau d'ordinateurs étant aussi utilisés)**. La grappe de processeurs est une machine dédiée à son fonctionnement de calculateur « haute performance » (qui ne se distingue donc d'un supercalculateur que par le choix de composants plus standard), tandis qu'un réseau de stations de travail « détourne » un certain nombre de machines de leur utilisation quotidienne pour les utiliser à des fins de calcul haute performance. La différence entre ces deux types de machines peut sembler subtile sur le principe, dans la pratique elle revêt trois aspects essentiels :

- L'équilibre de charge est plus facile à réaliser sur une grappe de processeurs (comparé à un réseau d'ordinateurs) puisque moins d'événements parasites ne viennent perturber le calcul contrairement aux réseaux de machines où l'interaction de chaque machine avec un éventuel utilisateur entraîne un ralentissement du calcul sur la machine concernée.

- La performance du réseau subit les mêmes aléas que ceux décrits précédemment pour le processeur : la latence en particulier peut devenir très grande dans le cadre de réseaux d'ordinateurs.

- L'optimisation de la machine à travers divers paramètres système n'est pas la même pour une utilisation interactive que pour une utilisation de calcul haute performance.

4. Définition de la régularité

Dans le cadre des applications distribuées, il convient de distinguer nettement deux types d'applications aux comportements très éloignés : les applications **régulières** et les applications **irrégulières**. Les premières sont relativement faciles à paralléliser et sont aujourd'hui bien maîtrisées dans le monde de l'informatique distribuée. Les secondes sont plus complexes et leur distribution est réputée difficile et peu efficace.

4.1. Les applications régulières

Les applications régulières sont totalement prévisibles, que cela soit du point de vue de la charge de calcul ou de l'utilisation des données.

Une application régulière peut être vue comme un ensemble fini de tâches dont le nombre d'opérations ainsi que les données manipulées sont parfaitement connus.

Les structures de données utilisées par les applications régulières sont, elles aussi, qualifiées de régulières. Ce sont principalement des structures à base de vecteurs ou de matrices. Ce type de structures de données apporte de bonnes localités spatiales et temporelles.

La localité spatiale permet de définir des schémas d'accès aux données efficaces que cela soit pour une exécution séquentielle ou distribuée.

La localité temporelle offre quant à elle la possibilité de travailler en prédiction au niveau de l'accès aux données et donc d'utiliser des techniques d'optimisations avancées comme le chargement anticipé des données et le traitement pipeline. Les calculateurs vectoriels ainsi que nos ordinateurs actuels sont parfaitement adaptés à ce type de traitement et l'effort à fournir pour optimiser une application régulière séquentielle ou distribuée est bien connu.

Des exemples d'applications régulières sont les opérations de l'algèbre linéaire pleine comme la multiplication de matrices classique.

Les applications régulières peuvent, malgré leurs bonnes propriétés initiales, adopter un comportement contrariant l'exécution idéale de leur calcul. Ce mauvais comportement est moins à reprocher aux applications elles-mêmes qu'aux faiblesses des optimisations mises en place au niveau matériel, les systèmes de cache en particulier. En effet, lorsque l'implantation en mémoire des données à traiter est mal adaptée, de part sa taille ou son organisation, au système de cache de la machine cible, ce dernier peut se révéler inefficace

réduisant considérablement la performance du calcul. Cette difficulté peut se rencontrer dans le cadre d'exécution séquentielle comme distribuée. Les solutions à y apporter sont un fractionnement par blocs du traitement (solution que l'on retrouve dans des outils mathématiques comme BLAS [LAW79, DON90]) ou un arrangement différent des données [CAR99].

En bref, malgré la remarque exprimée ci-dessus, les applications régulières sont de bonnes candidates à une exécution sur une machine de type réseau d'ordinateurs. Elles sont d'ailleurs présentes dans de nombreux articles ayant pour objet la performance de réseau d'ordinateurs comme outil d'évaluation ou comme finalité des travaux présentés. Preuve ultime de la bonne maîtrise que l'on possède de cette classe d'application, l'existence de bibliothèques telles que la bibliothèque mathématique ScaLapack [BLA97] permettant d'effectuer (entre autres) du calcul matriciel à un très haut niveau de performance sur un réseau de machines.

4.2. Les applications irrégulières

Contrairement aux applications que nous venons d'évoquer, les applications irrégulières ne sont pas totalement prévisibles. Leur irrégularité peut résider dans un ou plusieurs facteurs tels : l'utilisation de structures de données irrégulières ou dynamiques, une charge de calcul dépendant des données elles-mêmes et pas seulement de leur volume, un accès aux données imprévisible avant exécution.

Les structures de données irrégulières sont principalement les structures hiérarchiques permettant d'étudier un phénomène à plusieurs échelles suivant les « zones d'intérêt » s'adaptant ainsi à un ensemble de données dont la pertinence n'est pas homogène. Ces structures peuvent être dynamiques si les zones d'intérêt sont mouvantes ou fluctuantes. Ces structures n'offrent d'intérêt que si l'étude du phénomène à l'échelle la plus fine sur l'ensemble du domaine ne peut être traitée efficacement, en effet leur manipulation est plus complexe du point de vue informatique, mais aussi parfois du point de vue du domaine d'application. On peut déjà noter que ce type de structures ne conserve pas automatiquement la localité des informations, ce qui est de nature à perturber le bon fonctionnement de toute la chaîne de calcul : le matériel, le système et l'application.

Une charge de calcul qui dépend des données elles-mêmes et pas uniquement de leur volume est un sérieux obstacle à une bonne distribution du travail. Il ne suffit plus de découper ce dernier en n blocs de volumes égaux pour répartir équitablement la charge sur n processeurs. Si l'on recherche l'efficacité, il devient nécessaire de mettre en place des moyens de distribution plus élaborés pour prendre en compte l'irrégularité de la charge de calcul.

L'accès aux données ne suit pas un schéma préétabli mais dépend lui aussi de l'exécution. Il n'est donc pas possible d'effectuer statiquement une distribution des données (différente d'une recopie globale de l'ensemble des données) capable d'assurer que : toute donnée nécessaire au traitement qui sera effectué par chaque nœud de calcul sera disponible localement. Il est fréquent de traiter ce problème selon l'une des deux alternatives suivantes :

- Considérer que l'ensemble des données est nécessaire à toute partie du calcul et recourir par conséquent à une recopie, sur chaque nœud de calcul, de l'ensemble des données.

- Utiliser un système de mémoire partagée virtuelle et déléguer la gestion des données à ce dernier.

Des exemples d'applications irrégulières sont souvent issus du domaine de la physique, on y trouve le lancer de rayons et les maillages à raffinement auto-adaptatif.

En conclusion de ce paragraphe je dirais que bien que les applications irrégulières revêtent différents aspects (et parfois les cumulent), leur dénominateur commun est un comportement très dépendant du jeu de données. Un corollaire à cette propriété est que la distribution d'applications irrégulières selon les méthodes courantes (statiques) n'est pas efficace puisqu'elles nécessitent une adaptation suivant le jeu de données que ne leur offre pas le comportement strict des méthodes statiques. Tout le monde s'accorde généralement sur le fait que la distribution efficace d'application irrégulière est un véritable défi en particulier dans le domaine des réseaux de machines (L'article [MUK95] fait un exposé précis des différentes difficultés posées par la distribution d'application irrégulière sur machines à mémoire distribuées).

Chapitre II : Références du monde du parallélisme

Sommaire

1. GENERALITES 27
1.1. Machines de références 27
1.2. Modèles d'exécution 30
2. MEMOIRE PARTAGEE VIRTUELLE 32
2.1. Mémoire physiquement partagée 33
2.2. Mémoire physiquement distribuée 35
3. DISTRIBUTION DES DONNEES ET ACCES DISTANT 43
4. CHOIX DE L'ALGORITHME SEQUENTIEL 45
5. CHOIX DE L'ORDONNANCEMENT 47

L'informatique parallèle est loin d'être une nouveauté, elle dispose de nombreuses références théoriques et pratiques tant du point de vue matériel que logiciel. Ce chapitre est destiné à en donner un très bref aperçu et à mettre en valeur certains points qui nous servirons ensuite de référence lors du travail sur les réseaux de machines. L'aspect gestion de la mémoire dans le cadre du parallélisme sera particulièrement développé puisqu'une partie importante de notre travail concerne justement ce domaine. Il m'a semblé opportun de faire l'inventaire des différentes méthodes, de montrer l'évolution de celles-ci afin de pouvoir ensuite donner les raisons d'un choix différent et faire apparaître cette différence le plus clairement possible.

1. Généralités

1.1. Machines de références

Les supercalculateurs ne sont pas la cible retenue pour mon étude aussi ne vais-je pas en donner une description très détaillée, cependant ces machines sont des références incontournables du monde du parallélisme et il m'a semblé important de ne pas les passer complètement sous silence. Ce type de machines sert donc couramment de référence que cela soit d'un point de vue théorique : à travers les matériels utilisés et leur mode de fonctionnement, que d'un point de vue pratique : de part leur représentation dans les documents traitant de parallélisme.

N'ayant pas de meilleur critère de choix, les deux machines que j'ai retenues pour illustrer ce chapitre sont simplement celles que j'ai le plus souvent rencontrées lors de mon travail de documentation : l'IBM-SP2 et le CRAY T3E. Notons toutefois que ces machines sont déjà relativement anciennes, une description de machines plus récentes peut être trouvé dans le TOP 500 [TOP02].

1.1.1. IBM SP2

IBM qualifie son SP de système parallèle adaptable (« scalable » est le mot anglais utilisé), généraliste, basé sur une mémoire distribuée et une communication par passage de messages [AGE95].

Figure 1 : Aspect extérieur de l'IBM SP

Les machines SP habituellement rencontrées vont de 2 à 128 nœuds (IBM indique cependant l'existence de quelques systèmes plus important, jusqu'à 512 nœuds de calcul). Les nœuds formant le réseau d'un IBM SP2 sont de deux types : les nœuds de calcul (processeurs, entrée/sortie, passerelle de communication) et les nœuds de communication (composés de ports d'entrée/sortie).

Les nœuds de calcul sont des groupes de 16 processeurs connectés à un côté d'un commutateur, les processeurs utilisés sont des RISC System/6000 POWER2. Le réseau, haute performance, est multi-niveau, bidirectionnel et à commutation par paquet. Chaque lien de communication comprend deux canaux transportant les données dans des directions opposées pour augmenter la tolérance aux pannes. Dans la même optique de sûreté de fonctionnement chaque commutateur est doublé (« shadowed »). Chaque nœud possède sa propre copie du système AIX et des logiciels systèmes RISC System/6000 standard plus un jeu de logiciels propriétaires, assurent l'exploitation parallèle de la machine. Ces logiciels propriétaires sont en partie formés par des versions optimisées de logiciels disponibles en version grand public : PVMe par exemple est la version optimisée pour l'IBM SP2 de la librairie de communication PVM. La librairie de communication MPI est aussi disponible.

La version 2 de la gamme de machines IBM SP est maintenant ancienne, IBM propose, à ce jour, une version 4 à base de processeurs POWER4.

1.1.2. CRAY T3E

Les systèmes CRAY sont certainement les supercalculateurs les plus connus, précurseurs en la matière ils continuent aujourd'hui encore à produire des systèmes très haut de gamme. Le Cray T3E-1350 [CRAY] est constitué de quelques centaines à plusieurs milliers de processeurs. Les processeurs utilisés sont cadencés à 675 MHz et interconnectés par « l'interconnexion CRAY » c'est à dire un tore 3D bidirectionnel, faible latence haut débit capable d'assurer une performance crête de 500 Mo/s/interface.

Figure 2 : Aspect extérieur du CRAY T3E

Le système est extensible par l'ajout d'**éléments de calcul** composés chacun de :

- 8 processeurs alpha 21164A, disposant chacun d'un accès mémoire d'un débit de 1200Mo/s ;
- 256 ou 512 Mo de mémoire locale ;

La mémoire est par conséquent physiquement distribuée ; cependant, l'adressage global de la mémoire ainsi que la cohérence des caches[1] est assurée par le matériel. Comme à son habitude CRAY a apporté un grand soin au refroidissement des unités de calcul puisque chaque module de huit processeurs dispose d'un système de refroidissement liquide. Le matériel est pris en charge par le système opératoire UNICOS/mk de CRAY. La programmation distribuée peut s'effectuer de manière explicite via les outils habituels (MPI, PVM …) ou de manière implicite via HPF et le système propriétaire de répartition de travail CRAFT. Pour optimiser l'utilisation de cette machine, CRAY a ajouté deux composants à son système d'exploitation : STREAMS, qui permet d'augmenter le débit entre le processeur et sa mémoire locale pour des traitements de type vectoriel et E-Registers qui offre des opérations efficaces de diffusion ou de rapatriement de données sur les mémoires locales et distantes. CRAY met l'accent sur l'équilibre de performance entre les différents composants de son système : processeurs, réseau et outils logiciels sont tous

[1] La modification de données utilisées par un processeur peut ne s'effectuer que dans sa mémoire cache. Deux processeurs travaillant chacun sur une copie locale d'une même donnée (contenue dans leur mémoire cache) pourraient amener à un résultat aberrant. Un système, dit de cohérence de caches, est donc chargé de gérer ce type de situation.

très haut de gamme. Les puissances de crête annoncées par CRAY vont de 54 GFLOPS à 3 TFLOPS.

1.1.3. Points communs

Ce qu'il faut retenir de cette rapide présentation des deux supercalculateurs que nous venons de voir est l'attention portée par le fabricant aux choix des composants en général et à l'interconnexion en particulier. Les processeurs de calcul sont finalement assez similaires à ceux que nous trouvons dans les stations de travail classiques (PC, Station de travail Sun, etc.), c'est l'interconnexion de ces derniers qui, elle, est très différente. Des réseaux de communication de topologies plus complexes, plus robustes et plus rapides ainsi que des librairies de communication optimisées sont les points forts de ces machines. Le prix exorbitant est leur point faible puisqu'il faut débourser plusieurs millions de dollars pour acheter une de ces machines.

Malgré cela beaucoup d'articles basés sur l'utilisation de ces machines révèlent que les communications restent le point faible des applications testées.

1.2. Modèles d'exécution

Un modèle d'exécution permet d'étudier le déroulement d'un algorithme, d'en estimer la performance et, dans une certaine mesure, de vérifier sa validité en se souciant le moins possible de la machine cible. Un modèle permet habituellement d'alléger les coûts de développement et d'en automatiser certaines phases. Si le monde séquentiel dispose de tout ce qu'il faut en matière de modèle d'exécution il n'en va malheureusement pas de même du monde parallèle. Les modèles que l'on trouve alors sont soit trop proches d'une machine particulière (que les concepteurs du modèle avaient en tête ou entre les mains lors de la rédaction de celui-ci), ou au contraire sont très généraux et divergent excessivement de la réalité. Dans un cas comme dans l'autre ils sont peu utilisables, certains efforts dans ce domaine méritent pourtant d'être cités.

Le modèle de programmation le plus reconnu (et un des plus anciens) est le modèle PRAM (Parallel Random Access Machine) introduit par Fortune et Wyllie en 1978 [FOR78]. Ecrit dans le but d'obtenir un modèle le plus proche possible d'un modèle séquentiel il modélise une machine à mémoire partagée. Son principe est assez simple, les paramètres concernent principalement la contention au niveau des accès à la mémoire par un ensemble infini de processeurs.

Les différentes variantes sont :

- EREW PRAM : Exclusive Read Exclusive Write PRAM. Le modèle le plus restrictif de tous, il considère que les accès à la mémoire sont tous séquentiels.

- CREW PRAM : Concurrent Read Exclusive Write PRAM. Ce modèle autorise l'accès concurrent aux données en lecture tout en conservant la séquentialité des accès en écriture.

- CRCW PRAM : Concurrent Read Concurrent Write PRAM. Ce modèle est le plus permissif, les lectures comme les écritures se font simultanément pour tous les nœuds sans problèmes de contention.

- ERCW PRAM : Exclusive Read Concurrent Write PRAM. Cette variante ne me semble correspondre à aucune machine concrète et n'est pas utilisée à ma connaissance.

Le choix entre exclusif ou concurrent permet une certaine souplesse de modélisation. Malheureusement cette dernière n'est pas assez précise pour refléter la réalité, le modèle EREW est souvent considéré comme trop pessimiste et le modèle CRCW trop optimiste. En outre, les processeurs sont sensés fonctionner de manière synchrone, le débit de l'interconnexion est illimité et sa latence est nulle amenant à une communication entre processeurs de coût nul. Ces hypothèses éloignent significativement la machine (quelle qu'elle soit) de son abstraction et si des algorithmes efficaces développés à partir d'un modèle PRAM existent bel et bien, beaucoup d'autres obtiennent des performances très éloignées de l'estimation qui en avait été faites et sont peu ou pas applicables.

Cette constatation a amené à développer de nouvelles variantes essayant de se situer entre les deux extrêmes que sont EREW et CRCW comme par exemple le modèle d'exécution QRQW (Queue Read Queue Write) PRAM [GIB96]. Ce dernier fonctionne comme un CRCW PRAM à la différence près qu'il attribue un coût à l'accès concurrent aux données. Ce coût est proportionnel au nombre de processeurs demandant l'accès à la donnée, les auteurs démontrent par ailleurs que ce modèle est plus puissant qu'un EREW PRAM.

Le BSP (Bulk Synchronous Parallel model) [VAL90] est une tentative de pont entre l'abstraction du modèle et la réalité des machines modélisées : il autorise les processeurs à fonctionner de manière asynchrone et modélise la latence et un débit limité du réseau. LogP [CUL93] utilise BSP comme base, il modélise une machine selon quatre paramètres : le nombre de processeurs (P), le temps minimum entre deux communications (g : gap, son inverse étant le débit par processeur), un majorant de la latence (L) et le surcoût dû à la communication (o : overhead) (la portion non recouvrable d'une communication). BSP et LogP n'ont ni l'un ni l'autre retenu des informations comme la topologie du réseau ou les

différents caches jugeant ces informations trop attachées aux machines cibles. Les deux modèles permettent potentiellement de modéliser une machine à mémoire distribuée (i.e. la communication y est apparente).

Les modèles d'exécution parallèle ont beaucoup progressé en quelques années. Pourtant leur utilisation ne s'est pas généralisée. La raison en est certainement la complexité d'utilisation de ces derniers par rapport à la qualité et surtout la fiabilité des informations que l'on peut en obtenir. Le rapport entre ces deux aspects (coût d'utilisation/qualité) ne permet pas encore de retirer les bénéfices justifiant l'emploi de modèle en remplacement de tests et d'études pratiques. Trop peu de technologie, topologie ou méthode n'a suffisamment affirmé sa suprématie dans le monde de l'informatique parallèle pour être standardisée. Il en résulte une grande diversité parmi les machines parallèles qui est en grande partie responsable de la difficulté à créer un modèle à la fois fiable et d'un bon niveau d'abstraction.

2. Mémoire partagée virtuelle

Comme j'y ai déjà fait allusion lorsque j'ai introduit le modèle PRAM, une volonté très nette des programmeurs est de ramener la machine parallèle à ce qu'ils connaissent mieux : la machine séquentielle. Deux types de machines parallèles s'opposent : les machines à mémoire partagée et les machines à mémoire distribuée, les premières se rapprochant plus des machines séquentielles que les secondes. Cependant la construction des machines à mémoire distribuée étant moins onéreuse et offrant de plus grandes possibilités d'extension, ces dernières obtiendront souvent la préférence des constructeurs. Les machines à mémoire partagée n'ont pas disparu pour autant puisqu'elles sont encore largement représentées en particulier dans de petits systèmes comme les SMP (Symetric MultiProcessor) qui comportent de deux à huit ou 16 nœuds. La mise en place de mémoire distribuée virtuellement partagée fit donc son apparition afin de tenter de retrouver un espace d'adressage global dans des machines n'en disposant pas de par leur construction

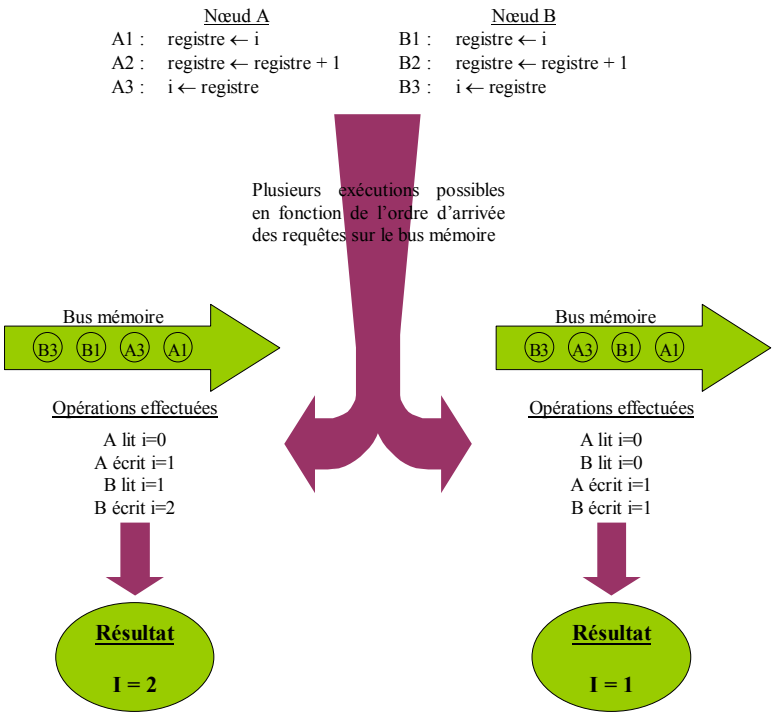
De nombreuses solutions existent à l'heure actuelle [RAI92, IFT99] afin de partager virtuellement de la mémoire, je vais en expliquer le principe général puis en présenter un éventail représentatif pour avoir des éléments de réponses à la question suivante : Avoir un espace d'adressage global semble très pratique, cela nous rapproche du monde séquentiel et nous n'avons plus de problème de gestion de la répartition des données mais assurer cette facilité lorsque la machine cible en est dépourvue est-il viable du point de vue de la performance ?

2.1. Mémoire physiquement partagée

Observons tout d'abord le fonctionnement d'une machine parallèle à mémoire partagée. Ce n'est pas parce que l'adressage est commun que tous les problèmes d'accès mémoire sont résolus : les requêtes des différents processeurs transitent par le même bus d'où une séquentialité des accès, l'accès à certaines données doit être protégé et un mécanisme doit assurer la cohérence des mémoires caches des processeurs. Si la protection de données (accès exclusif, lecture concurrente/écriture exclusive, …) est laissée à la charge du programmeur, le matériel se charge quant à lui d'assurer la cohérence des caches et tente souvent aussi de pallier la séquentialité des accès par des optimisations comme la satisfaction de requête par des caches voisins ou le regroupement de requêtes.

2.1.1. Cohérence des données dans un environnement parallèle

La cohérence est indispensable à l'obtention d'un fonctionnement équivalent à celui des machines séquentielles. Notons toutefois que le fait que la cohérence soit assurée ne délivre pas le programmeur de certaines synchronisations : une séquence lecture écriture par exemple (cas typique d'une incrémentation) effectuée en parallèle par deux nœuds peut amener à différents résultats (Exemple 1).



Exemple 1 : Champ d'action de la cohérence

Tout ce que garantit le respect de la cohérence de données est qu'une fois une opération d'écriture effectuée, toute opération de lecture en verra le résultat. Dans l'exemple précédent, solution de la colonne de gauche, la cohérence garantit que l'opération B1 ayant lieu après l'opération A3 elle renverra la valeur écrite lors de cette dernière c'est à dire 1.

2.1.2. Cohérence par désactivation des mémoires caches

Le moyen le plus radical d'obtenir cette cohérence dans une machine à mémoire partagée est la désactivation des mémoires caches. Lorsque les mémoires caches sont désactivées chaque opération sur une donnée est réalisée directement en mémoire principale, chaque nœud « voit » donc la même information. Désactiver les mémoires caches a cependant l'inconvénient majeur de réduire les performances globales de la machine de manière très spectaculaire puisque le temps d'accès moyen à une donnée (Tam) est ramené au temps d'accès à la mémoire principale (ta_{mp}). En effet, le temps d'accès du cache (ta_c) n'intervient dans le calcul du temps d'accès moyen à une donnée (Équation 1) puisque le taux de présence dans le cache (Tp) est de zéro.

$$Tam = (ta_c \times Tp) + (ta_{mp} \times (1 - Tp))$$

Équation 1 : Temps d'accès moyen aux données

En outre cette méthode séquentialise des opérations qui auraient pu s'exécuter en parallèle comme la lecture d'une même donnée sans modification.

La désactivation des mémoires caches présente l'avantage d'être facile à mettre en œuvre mais son impact négatif sur les temps d'accès au données est tel que cette méthode ne saurait être retenue comme une solution performante pour assurer la cohérence des données.

2.1.3. Cohérence par espionnage du bus mémoire

La solution habituellement choisie est l'espionnage du bus mémoire exploitant l'unicité de ce dernier. Toutes les requêtes transitant par le bus sont analysées et les données contenues dans les mémoires caches locales sont gérées en fonction de ces requêtes. Un protocole de maintien de la cohérence très répandu est le protocole MESI, il est basé sur l'étiquetage des données contenues dans le cache selon quatre états : Modified (Modifié), Exclusive (Exclusif), Shared (Partagé) et Invalid (Invalide). Au départ les données sont toutes considérées comme invalides (elles sont absentes du cache) ensuite elles passent d'un état à l'autre en fonction des requêtes transitant sur le bus en respectant le schéma de la Figure 3.

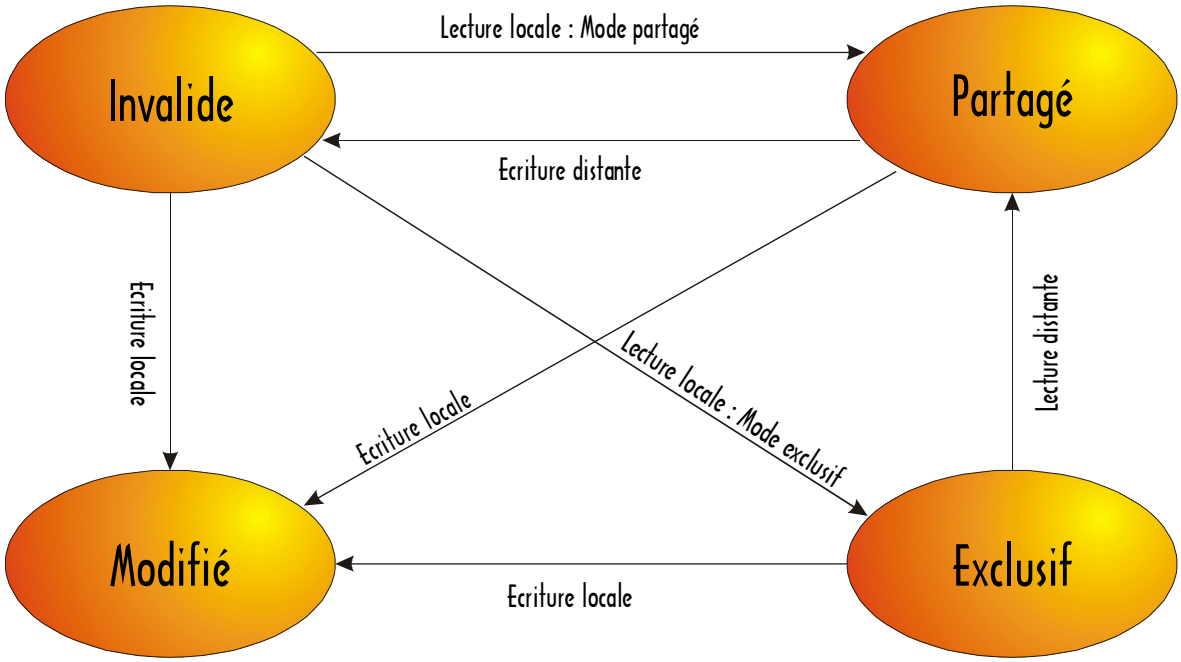


Figure 3 : Protocole MESI simplifié

Le système d'espionnage du bus mémoire permet de réduire les accès à la mémoire principale par rapport à la désactivation des mémoires caches en réintroduisant l'utilisation de ces dernières. Dit autrement : l'utilisation de l'espionnage de bus amène à des taux de présence non nuls des données dans les mémoires caches. Le calcul du taux de présence d'une donnée en cache lors de l'utilisation du protocole MESI est plus complexe que son équivalent séquentiel puisqu'une donnée est considérée comme présente lorsqu'elle n'est pas dans l'état Invalide. Le taux de présence dépendra donc non seulement des caractéristiques physiques de la machine (taille des mémoires caches et des mémoires principales) mais aussi du schéma d'utilisation des données par le programme parallèle.

2.2. Mémoire physiquement distribuée

L'idée de la mémoire virtuellement partagée consiste à émuler un système multiprocesseur à mémoire physiquement partagée sur un système multiprocesseur à mémoire physiquement distribué. Le but est, comme nous en avons l'habitude depuis quelques paragraphes, d'offrir au programmeur un contexte le plus intuitif possible. Les principes de cette mémoire virtuellement partagée ou SVM (Shared Virtual Memory) furent posés par Kaï Li en 1986 [LIH86, LI86].

2.2.1. Principes de la mémoire virtuellement partagée

Le principe de la mémoire virtuellement partagée est d'utiliser un espace de mémoire virtuel commun découpé en pages. La mémoire locale de chaque nœud se comportant comme un cache de cet espace.

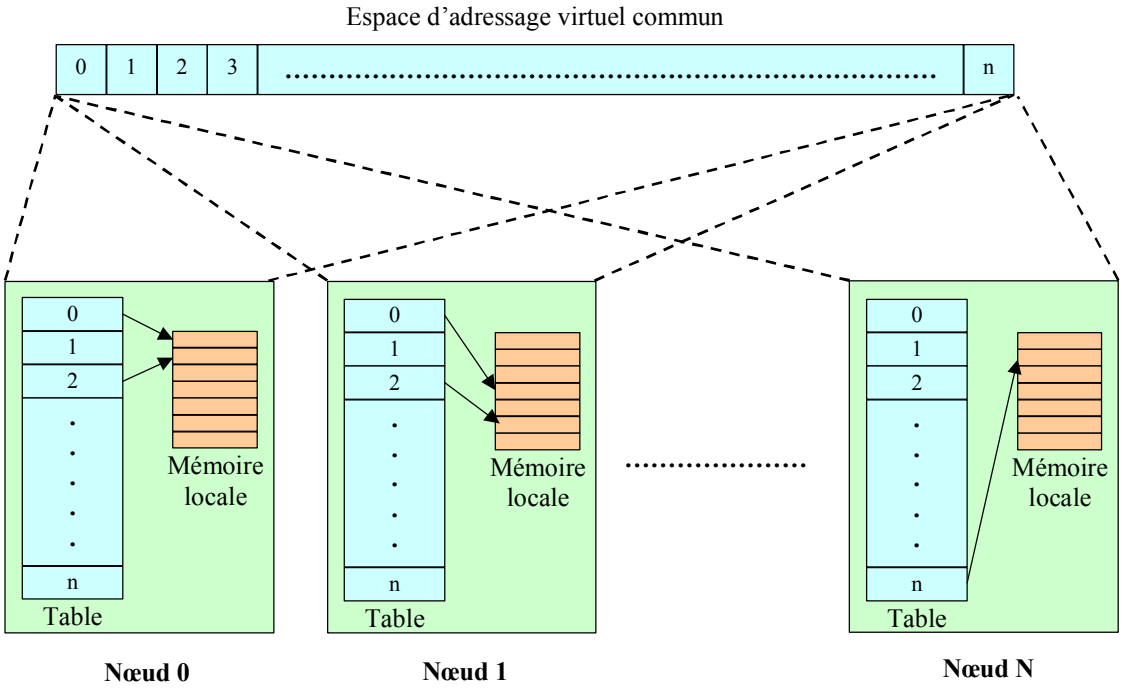


Figure 4 : Principe de la mémoire virtuellement partagée

Afin d'assurer la cohérence, une table des pages est mise en place au sein de chaque nœud, indiquant la validité (ou l'invalidité) de la représentation locale de chaque page virtuelle.

En cas de référence invalide le système de défaut de page mis en place est chargé de récupérer la page référencée au sein d'un nœud possédant une instance valide de celle-ci.

Si le principe est au point, les performances d'un tel système sont limitées par les facteurs suivants :

- Assurer une cohérence par pages entraîne un phénomène de faux partage et de fragmentation. Le faux partage est le fait que plusieurs accès dont un au moins en écriture ait lieu simultanément sur des adresses différentes mais localisées sur la même page amenant des échanges d'informations inutiles. La fragmentation est le fait de devoir récupérer une page entière alors qu'une partie seulement de celle-ci a été modifiée par rapport à l'instance locale. Ces deux effets parasites augmentent inutilement le trafic réseau.
- Aucun matériel n'étant mis en place pour assurer le travail de maintien de la cohérence, les échanges d'information se font par logiciel, par l'intermédiaire de messages. Cela entraîne un coût d'action unitaire important (résolution d'un défaut de page par exemple) et vient aggraver le défaut précédent.
- Le coût additionnel du protocole est important puisqu'il amène fréquemment à de lourdes opérations de synchronisation : interruption des processeurs, demandeur et serveur, jusqu'à la récupération complète d'une page invalide dans le cas d'un défaut de page, interruption de tous les processeurs lors d'une demande d'accès à une page en écriture etc.
- La pollution importante des mémoires tampon de chaque processeur par des routines de protocole n'est pas non plus négligeable.

Pour implémenter la mémoire virtuellement partagée, de nombreux modèles sont employés [ADV95], les chapitres qui suivent en présentent quelques-uns.

2.2.2. Les modèles de mémoire virtuellement partagée

• Le modèle SC : « Sequential Consistency »

Dans le modèle « Sequential Consistency » les opérations de cohérence sont propagées immédiatement et les processus doivent à chaque fois attendre la terminaison des opérations de protocole avant de pouvoir accéder à une page mémoire.

Un tel modèle est qualifié de modèle de cohérence stricte, c'est à dire qu'il vérifie en permanence le fait que la lecture d'une variable renvoie la valeur qui lui a été donnée lors de la dernière opération d'écriture la concernant.

Pour obtenir ce résultat, une stratégie lecteurs multiples / rédacteur unique est appliquée. Une page valide ne peut appartenir qu'à un et un seul nœud en accès lecture/écriture ou

bien plusieurs copies d'une même page peuvent coexister dans le système, elles sont alors toutes en accès lecture uniquement.

Une écriture sur une variable de la page provoque l'invalidation des éventuelles copies en lecture seule et le nœud ayant effectué la modification devient l'unique propriétaire de la page, ce dernier peut alors effectuer d'autres lectures/écritures sur cette page jusqu'à ce qu'un autre nœud en réclame l'accès.

Une demande d'accès en lecture par un nœud non propriétaire restreint l'éventuel unique accès lecture/écriture vers un accès lecture seule (dans le cas où un accès en écriture préexistait). Elle provoque la récupération d'une copie valide de la page par le demandeur auprès de l'ancien possesseur de l'accès lecture/écriture ou d'un propriétaire quelconque si un tel accès n'existait pas.

Les avantages de ce modèle sont l'aspect intuitif de la programmation dans le contexte offert par celui-ci et une implémentation des opérations de cohérences simple et rapide.

Le principal défaut de ce modèle est le faux partage : au pire, ce modèle peut amener à une invalidation par écriture et un défaut de page par lecture sans que les nœuds se disputant des pages communes ne travaillent sur les mêmes adresses.

La propagation immédiate des opérations de cohérence ainsi que le fonctionnement par pages entraînent un important trafic réseau inutile. Pour limiter ce type de comportement, un regroupement des opérations de cohérences avant envoi est souvent mis en place : on parle alors de « Relaxed Consistency ». La « Relaxed consistency » autorise le report de la propagation des opérations de cohérences à des points de synchronisation. Une relative transparence pour le programmeur est conservée à condition que ce dernier utilise une synchronisation explicite.

- Le modèle RC : « Release Consistency »

Le modèle « Release Consistency », basé sur le principe de la « Relaxed Consistency », amène un réel bond en avant des performances par rapport au modèle SC.

Dans ce modèle les synchronisations sont divisées en *acquire* et *release* qui sont utilisées pour entrer et sortir de section critique. Les invalidations sont cumulées et ne sont propagées qu'au moment du *release*. Une page ne sera mise à jour de manière fiable que pour un nœud ayant effectué un *acquire*.

Le fait de reporter la propagation des opérations de cohérence, et éventuellement d'en propager plusieurs en une seule fois, réduit la fréquence des opérations de protocole et par conséquent le trafic réseau.

La programmation (en utilisant ce modèle) ainsi que la mise en œuvre des opérations de protocole sont plus compliquées qu'avec le modèle SC.

Les modèles EC : « Entry Consistency » [BER91] et ScC : « Scope Consistency » [SIN96] sont des variantes de RC et reposent, eux aussi, sur le principe de la « Relaxed Consistency ».

Les modèles basés sur RC réduisent le trafic réseau par rapport aux modèles SC. Le volume de communication et la fréquence des opérations de cohérence restent malgré tout importants et d'autres modèles vont s'attacher à les réduire, il s'agit des modèles de cohérence paresseux. Une séparation s'opère cependant ici selon le type de mise en oeuvre puisque les modèles paresseux LRC : « Lazy Release Consistency » vont nécessiter une gestion élaborée de nombreuses informations d'état les réservant à des implémentations logicielles tandis que les mises en œuvre matérielles seront basées sur de l'ERC : « Eager Release Consistency » [CAR91] utilisant des opérations plus simples mais plus fréquentes.

- Le modèle LRC : « Lazy Release Consistency »

Dans le modèle LRC [KEL92] les mises à jour correspondant aux écritures de A ne seront pas diffusées lors de *release* de A mais envoyées à B lors du prochain *acquire* de B (si celui-ci se produit après le *release* de A). A devra alors envoyer à B toutes les informations dont il dispose et que B n'aurait pas vues, ce qui nécessite la conservation de l'ordre de survenue des évènements. A cette fin, les intervalles délimités par les enchaînements *acquire* puis *release* sont estampillés de manière locale (nous disposons donc d'une fonction de datage), ce modèle est multirédacteurs, l'ordre obtenu n'est que partiel.

Lors de la première modification locale d'une page, une erreur de protection est levée et la routine correspondante crée une copie de la page avant modification, cette copie est nommée : le *twin*, et ajoute la page à une liste locale des pages modifiées depuis le dernier *acquire* du nœud courant.

A la fin de l'intervalle, le processeur enregistre la date et la liste des pages modifiées dans une structure appelée le *write-notice*.

Pour chaque page modifiée, les modifications effectives sont retrouvées par comparaison de la page actuelle (*dirty copy*) et du *twin* puis stockées dans un *diff*. Cette opération peut être effectuée soit systématiquement à la fin de chaque intervalle (*eagerly*) soit à la demande (*lazily*).

Lors d'un *acquire*, le nœud demandeur envoie son vecteur de datage, il reçoit en réponse les *write-notices* correspondant provenant des derniers *releasers*. Il invalide alors ses pages locales en fonction des *write-notices*. Lorsque par la suite il tentera d'accéder à une des pages invalidées, la routine de défaut de page récoltera les *diffs* concernant cette page et les

appliquera sur la copie locale de la page dans l'ordre défini par les datages, reconstruisant ainsi une page cohérente.

L'utilisation de ce modèle permet de réduire sensiblement le trafic réseau puisque les pages ne transitent plus sur le réseau mais que seuls les *diffs* sont envoyés.

La complexité du modèle lui même est importante : les opérations de cohérences sont lourdes et les informations d'état dont il a besoin sont nombreuses et leur gestion complexe. Les *diffs* et les *write-notices* doivent être conservés par un nœud jusqu'à ce qu'aucun autre ne soit susceptible de les lui demander : l'évaluation de cette condition n'est pas toujours triviale. LRC est donc un grand consommateur de ressources mémoire, la mise en place d'un ramasse miette est souhaitable, voir nécessaire, mais ajoute encore à la lourdeur des opérations de protocole. L'ensemble de ces défauts réserve LRC à des tailles de problème relativement réduites.

Les modèles suivants introduisent l'idée d'un propriétaire nommé d'une page afin de corriger les défauts repérés ci-dessus [IFT98b].

- Le modèle HLRC : « Home-based Lazy Release Consistency »

L'idée de ce modèle est d'attribuer à chaque page un nœud *home* pour lequel cette page est toujours à jour [ZHO96, SAM97,IFT98b].

Par rapport à LRC :

- Le mécanisme d'invalidation est le même.
- Lors d'un défaut de page, c'est au nœud *home* que l'on s'adresse : celui-ci vérifie le datage joint à la requête et si le sien est postérieur, envoie la page complète au nœud en défaut ; si au contraire c'est le datage de la requête qui est postérieur, cela signifie que les releases sont en cours et la requête est mise en file d'attente jusqu'à résolution de ceux-ci.
- A la fin de chaque intervalle (*acquire* - *release*), chaque nœud non-home calcule les *diffs* puis les envoie au nœud *home*. Il peut ensuite les détruire (quelques variantes existent sur le type d'acquittement attendu du nœud *home*, explicite ou implicite lors d'une communication ultérieure, et retardent parfois la destruction des *diffs*). Le nœud *home* applique les *diffs* dès réception pour ensuite les détruire à son tour.

Le choix du nœud *home* pour une page donnée n'est pas neutre puisque le nœud *home* n'a pas à produire de *diffs* concernant les modifications qu'il fait sur la page dont il est propriétaire.

La destruction au fur et à mesures des informations de cohérence résout en grande partie le problème de l'espace mémoire évoqué lors de la présentation du modèle LRC. Cela permet en outre de ne plus recourir à un ramasse miette, ce qui allège d'autant les opérations de protocole second point faible de LRC.

Le défaut le plus important subsistant dans ce modèle est la consommation de ressources CPU pour la production des *diffs* et des *write-notices* et la pollution des mémoires caches qui en découle. La limite des implémentations « tout logiciel » n'est certainement pas loin d'être atteinte par ce modèle.

Les modèles que je vais présenter maintenant ne sont plus « tout logiciel », ils utilisent du matériel dédié afin d'optimiser les opérations de protocole de maintien de la cohérence.

- Le modèle AURC : « Automatic Update Release Consistency »

Ce modèle utilise lui aussi la notion de nœud *home* et son mécanisme d'invalidation est celui de LRC. La nouveauté réside dans l'utilisation de la mise à jour automatique du système SHRIMP, c'est à dire un système matériel appelé *Automatic Update* qui ponctionne les demandes d'écriture locales directement sur le bus mémoire de la machine et les propage via le réseau [IFT98a].

Dès qu'un nœud, hormis le nœud *home*, fait une opération d'écriture, l'*Automatic Update* la lit sur le bus de la machine et envoie directement au nœud *home* les informations permettant à celui-ci de maintenir à jour sa copie.

En cas d'erreur de page, la routine de gestion de défaut de page récupère l'intégralité de la page auprès du nœud *home*.

Ce modèle est libéré des overheads dus aux opérations de *diffs* puisque leur fonction est entièrement assurée par le matériel. Ceci ne signifie pourtant pas que ces opérations sont instantanées mais uniquement qu'elles ne pénalisent plus le nœud utilisateur de la page.

Le principal défaut de AURC est le nombre important de messages qu'il génère, un par écriture, et donc sa forte dépendance vis à vis des caractéristiques, des performances et de la taille du réseau.

AURC nécessitant du matériel dédié, les cartes réseaux intégrant l'*Automatic Update*, sa mise en place est plus chère que les implémentations purement logicielles.

Les modèles dit *Overlapped* contournent quant à eux le problème d'une autre manière : ils utilisent un coprocesseur pour effectuer une partie des opérations de protocole de maintien de la cohérence.

Deux de ces modèles sont OLRC [ZHO96] et OHLRC, implémentant respectivement LRC et HLRC. Ces modèles souffrent globalement des même faiblesses que leurs homologues « tout logiciel » à l'exception des surcoûts dus aux opérations de protocoles. L'utilisation d'un coprocesseur dédié sur la carte réseau améliore les performances de ces modèles mais elle se révèle financièrement plus coûteuse.

Un effet sournois de l'utilisation du coprocesseur est que l'application d'opérations de protocole volumineuses (rares dans une utilisation classique) par celui-ci peut se révéler plus lente que leur application par le processeur d'exécution et provoquer une baisse de performance. Ceci est du à la piètre puissance de calcul du coprocesseur comparée à celle du processeur d'exécution. Cet inconvénient peut certainement être limité par une anticipation des demandes d'accès dans les programmes utilisant ce modèle.

Dans la pratique un modèle comme OHLRC montre des performances honorables, quoique reposant encore beaucoup sur les performances du réseau de communication. Son inconvénient majeur est le coût supérieur de la machine dû à l'achat de matériel dédié.

2.2.3. Conclusion

Nous venons de faire un tour d'horizon des modèles de cohérence permettant d'utiliser la mémoire virtuellement partagée de manière fiable et il en ressort une dépendance très forte vis à vis du réseau. Même les modèles comme LRC sensés réduire à leur minimum les échanges restent encore de forts consommateurs de trafic réseau.

Il est intéressant de noter le changement d'orientation des optimisations opéré lors du passage d'LRC à HLRC ainsi que dans l'intégration de matériel dédié. Jusqu'à LRC le principal souci des développeurs de modèle était la réduction du trafic réseau, ce qui les a menés à un modèle fort consommateur de ressources mémoire et processeur (les informations de protocoles de LRC sont nombreuses et les opérations assurant la gestion de celles-ci sont lourdes, la fréquence des messages liés à LRC est réduite autant que possible et les informations contenues dans ceux-ci sont optimisées) puis le coût des opérations de protocoles au sein des machines est devenu la priorité. HLRC envoie régulièrement des pages entières afin de réduire les informations que chaque nœud doit conserver en local, AURC envoie des messages sur le réseau à chaque opération d'écriture et les modèles OLRC et OHLRC n'ont pour but que d'assister le processeur d'exécution lors des opérations de protocoles sans chercher à réduire les échanges à travers le réseau. Cette orientation des recherches a même été jusqu'à réintroduire les principes de SC, non plus sur un système de pages, mais directement sur des mots mémoire dans Fine-Grain SC [SCA96], rendant ainsi les opérations de cohérence triviales aux prix d'un nombre de messages très important. Durant cette période, une tendance très nette consistait en l'assurance que les performances réseaux avaient fait ou allaient faire de tels progrès que le goulot d'étranglement était devenu le temps de calcul utilisé par les opérations de

protocoles et non plus par les communications elles-mêmes (des illustrations de cette conviction peuvent être trouvées dans les articles IFT96a et IFT96b).

Ce point de vue est encore fortement ancré dans le monde de l'informatique distribuée : se dire que le manque de performance réseau sera résolu par l'évolution du matériel est très pratique. Malheureusement si les performances des réseaux ont en effet subit une très forte amélioration depuis quelques années, l'évolution des processeurs d'exécution a été parallèlement encore plus importante et l'écart entre les deux s'est donc agrandi. Les problèmes de manque de performance des réseaux qui se posaient quelques années en arrière se posent encore plus de nos jours.

Ces convictions n'étaient pas pour autant dénuées de raisons, une période pendant laquelle les réseaux n'étaient plus le point faible d'une architecture parallèle a d'ailleurs existée. Un équilibre relatif entre les performances de calcul et de communication se perpétue encore de nos jours dans le monde des supercalculateurs décrits paragraphe 1.1.

De plus, si l'on différencie dans les performances du réseau l'aspect logiciel (librairie de communication, système, pilote de périphérique) de l'aspect purement matériel du réseau, la constatation reprend ici toute sa valeur, le matériel réseau n'est plus le point faible de l'ensemble. C'est en effet le côté logiciel, par des temps de latences importants, qui réduit les performances globales.

3. Distribution des données et accès distant

Nous avons vu dans le chapitre précédent que la mémoire partagée virtuelle offrait un certain confort de programmation à l'utilisateur d'un système à mémoire distribuée au prix d'un certain nombre d'opérations de protocole. Plusieurs mécanismes différents assurant un fonctionnement cohérent du système de mémoire partagée virtuelle ont été introduits mais ils ont tous en commun de ne laisser à l'utilisateur que peu ou pas de contrôle sur leur fonctionnement. L'optimisation d'applications, irrégulières en particulier, est donc rendue complexe et hasardeuse. Une autre approche est donc de se passer d'espace d'adressage commun et d'opérer soi-même une gestion des données au sein du système distribué afin de conserver un contrôle le plus complet possible des échanges entre les nœuds du système.

De manière générale, les systèmes à mémoire virtuellement partagée obtiennent des performances correctes sur des applications régulières manipulant un volume de données important. Sur des jeux de données plus modestes le rapport entre les communications et les calculs est trop important et les performances ne sont pas bonnes. Les applications irrégulières ne bénéficient pas de la même façon de l'augmentation du volume de données

traitées : le rapport communication/calculs n'est pas forcément réduit par une augmentation de la taille du problème et la localité des données n'est pas forcément améliorée par un accroissement de leur volume. Il en résulte que l'implémentation efficace d'applications irrégulières sur des systèmes à mémoire virtuellement partagée nécessite des modifications relativement profondes de celles-ci afin d'exploiter au mieux le mécanisme de partage de la mémoire sous-jacent et n'aboutissent malheureusement souvent qu'à des résultats médiocres [IFT96b, IFT99, BIL98]. De plus, l'intérêt principal du partage virtuel de mémoire résidant dans la simplicité de programmation, sa justification est mise à mal si cette dernière devient complexe et lourde.

Dans le domaine des applications régulières, distribuer les données au sein d'une machine à mémoire distribuée ne pose généralement pas de problème particulier. Lors d'une multiplication de matrices par exemple, il est aisé de déterminer les données dont chaque nœud aura besoin en fonction de la distribution du calcul. On distingue alors deux types de données : celles à utilisation locale uniquement et celles nécessaires à plusieurs nœuds de calcul. Le placement des données de la première catégorie est trivial, en ce qui concerne celui des données de la seconde catégorie une solution de copies multiples peut-être envisagée si les données ne sont accédées qu'en lecture seule. Si des données de la deuxième catégorie nécessitent un accès en lecture/écriture, plusieurs solutions s'offrent au programmeur qui se rapprochent des mécanismes employés par les systèmes à mémoire virtuellement partagée : nœud propriétaire, accès par synchronisation explicite etc. Les données de ce type sont peu nombreuses dans le domaine des applications régulières distribuées « manuellement » et se résument souvent aux variables de synchronisation.

Dans la grande majorité des cas, une solution optimale ou sous-optimale de placement des données peut être déterminée, de manière statique, lors de la programmation d'applications régulières distribuées. La granularité des blocs de données transitant sur le réseau pouvant être parfaitement adaptée à l'application, les problèmes de faux partage et de fragmentation n'existent pas dans le cadre d'une distribution « manuelle » des données. La distribution des données par le programmeur au lieu du système de mémoire virtuelle semble donc une solution plus performante et ne présentant finalement qu'un surplus de travail relativement réduit pour le programmeur. Il n'en est malheureusement pas de même dans les contextes des applications irrégulières.

Dans une application irrégulière, l'utilisation des données est imprevisible : la distribution des calculs et la distribution des données sont décorrélées. Il est alors impossible de déterminer statiquement un placement des données optimal ni même sous-optimal en un temps raisonnable ; deux choix sont donc possibles pour traiter ce problème : l'accès distant ou la recopie totale. Lorsque la recopie totale des données est possible (i.e. lorsque le jeu de données n'est pas trop volumineux et que ce dernier n'est pas souvent modifié) elle est souvent la meilleure solution, elle présente en outre l'avantage d'être très simple à mettre en œuvre. L'accès distant permet de traiter des jeux de données plus volumineux et/ou souvent modifiés mais il complique la programmation et se rapproche des

mécanismes de mémoire virtuellement partagée, il pèse lourd sur le rapport communication/calculs. Un placement initial des données selon leur utilisation probable peut parfois améliorer les performances de manière significative mais reste une solution hasardeuse à la vue de l'irrégularité de l'application cible.

Il n'y a pas, à ma connaissance, de solution parfaite et générique à la gestion des données dans des applications irrégulières distribuées : quelques schémas existent, mais il convient de les adapter afin d'établir une solution spécifique à une application donnée. De nombreux exemples de solutions de gestion des données distribuées, souvent compliquées, se retrouvent dans les articles traitant de la distribution d'applications irrégulières comme le lancer de rayon ou les maillages autoadaptatifs [REI95, CHA93, PAR95, PAR98]. Je reviendrai un peu plus loin sur la place que peut occuper la paresse en tant qu'outil de distribution des données dans les différents contextes évoqués ci-dessus.

4. Choix de l'algorithme séquentiel

Lors du travail de distribution d'une application, une des premières étapes est le choix de l'algorithme séquentiel qui servira de base à cette parallélisation. La règle qui est traditionnellement observée est de choisir l'algorithme séquentiel le plus efficace : celui dont la complexité algorithmique est la plus faible. Ce choix amène parfois à des parallélisations difficiles, il arrive même que celles-ci soient impossibles et qu'il faille recommencer la distribution en choisissant un autre algorithme séquentiel de base. Ce paragraphe tente de comprendre les raisons de ce choix de la complexité algorithmique comme seul critère de sélection et d'en évaluer le bien fondé.

Tout d'abord rappelons le principe théorique de la complexité algorithmique : il s'agit d'estimer le temps d'exécution d'un calcul en fonction de la taille du jeu de données en entrée, l'unité de mesure du temps étant l'opération élémentaire. Le résultat d'un calcul de complexité algorithmique doit être considéré comme un taux de croissance (du temps d'exécution en fonction de la taille du problème) plutôt que comme une estimation du temps de calcul. Un algorithme ayant une plus faible complexité algorithmique qu'un autre amènera au résultat en un nombre d'opérations moindre à partir d'un certain volume de données en entrée. En deçà de ce volume seuil, l'algorithme de plus faible complexité est moins efficace, ceci s'explique par le fait que les algorithmes peu complexes sont souvent plus compliqués (il est par exemple courant de compliquer les itérations d'une boucle afin d'en réduire le nombre). En pratique le seuil de rentabilité de l'algorithme de plus faible complexité est souvent ignoré, puisque le calcul des complexités algorithmiques n'est employé que pour sélectionner au mieux des algorithmes destinés à traiter des volumes de données importants.

La notation suivante est habituellement utilisée (Équation 2) :

$$T(n) = f(n)$$

Équation 2 : Notation usuelle de la complexité algorithmique

Ce qui pourrait traduire que l'algorithme de complexité T(n) traitera un problème de taille n en f(n) opérations élémentaires mais signifie plutôt que si la taille du problème varie selon n, alors le temps d'exécution de l'algorithme variera selon f(n).

Le calcul de complexité algorithmique est souvent utilisé pour comparer les performances théoriques de deux algorithmes. Cependant la comparaison obtenue n'est totalement fiable que dans des conditions d'exécution identiques et certains paramètres opérationnels tels que la localité sont susceptibles d'entraîner des écarts de performances difficilement détectables par les calculs de complexité. La multiplication de matrices avec un stockage par ligne ou par colonne entraînant une utilisation des mémoires tampons très différente et des performances éloignées illustre ce propos.

Il est courant de considérer qu'une parallélisation est (virtuellement) parfaite si elle répartit équitablement l'intégralité de la charge de calcul entre les nœuds sans introduire de surcharge pour effectuer cette répartition. Cet idéal amène à l'estimation de la complexité parallèle suivante (Équation 3) :

$$T_{parallèle}(n,p) = \frac{T_{séquentiel}(n)}{p}$$

Équation 3 : Complexité parallèle idéale

Cela signifie que si un algorithme séquentiel à une complexité $T_{séquentiel}(n) = f(n)$ et peut donc traiter un problème de taille n par f(n) opérations et que l'on parallèlise cet algorithme sur p machines, alors la complexité de l'algorithme parallèle obtenu devrait idéalement être de $\frac{f(n)}{p}$. Bien sûr, ce résultat ne peut être atteint que si la parallélisation est « idéale » mais il est courant de considérer cette valeur comme un but à approcher au plus près par un soin toujours plus important apporté à la parallélisation.

Pour en revenir au choix de l'algorithme séquentiel qui servira de base à la parallélisation il découle directement des deux équations données ci-dessus (Équation 2 et Équation 3). En effet, si un algorithme A_1 est de complexité $T_1(n) = f(n)$ tandis qu'un algorithme A_2 est de complexité $T_2(n) = f'(n)$ et que pour tout $n>n_0$, $f(n)<f'(n)$ alors :

$$\text{pour tout p et pour tout } n>n_0, \frac{f(n)}{p}<\frac{f'(n)}{p} \text{ donc,}$$

$$\text{pour tout p et pour tout } n>n_0, \frac{T_1(n)}{p}<\frac{T_2(n)}{p} \text{ donc,}$$

$$\text{pour tout p et pour tout } n>n_0, T_{\text{parallèle 1}}(n,p) < T_{\text{parallèle 2}}(n,p)$$

Ce qui signifie que la version parallèle de l'algorithme A_1 sera meilleure que la version parallèle de l'algorithme A_2.

Il existe des manières plus élaborées de calculer la complexité algorithmique mais les différentes méthodes convergent vers la même conclusion : l'algorithme séquentiel devant servir de base à une parallélisation optimale est celui de complexité la plus faible à condition d'effectuer parfaitement le travail de répartition sans interférer sur le traitement initial. La clé de l'efficacité parallèle semble ensuite résider uniquement dans le soin apporté à la répartition de la charge. Dans la suite de ce document, je remettrai en question les affirmations faites dans ce chapitre en tentant de faire ressortir le lien qui existe entre l'algorithme séquentiel servant de base et la qualité de la répartition.

5. Choix de l'ordonnancement

Les problèmes d'ordonnancement ne sont pas l'objet de cette thèse, notons tout de même qu'ils se heurtent aux même difficultés que les problèmes de répartition de charge. Traités de manière statique dans le cadre des applications régulières par une analyse des dépendances ils ont la plupart du temps une solution analytique. Le recours à des heuristiques reste parfois nécessaire, lorsque l'ordonnancement optimal n'est pas décidable, on se contente alors d'obtenir un ordonnancement sous-optimal dans un temps raisonnable. Dans le cadre des applications irrégulières, les dépendances ne sont pas obligatoirement connues, pas plus que la charge de chaque tâche, il devient alors pratiquement impossible d'anticiper un ordonnancement même sous-optimal de manière statique. Il faut recourir à des méthodes dynamiques exploitant au mieux les bribes d'informations dont dispose l'ordonnanceur sans avoir réellement de garantie quant au résultat final ; la réplication de tâche, l'anticipation avec possibilité de retour sont des

exemples de stratégies pouvant être mises en œuvre dans ces applications irrégulières distribuées.

Chapitre III : L'architecture « réseau d'ordinateurs »

Sommaire

1. CARACTERISTIQUES GENERALES DES RESEAUX D'ORDINATEURS 51
2. LIMITES POTENTIELLES DES PERFORMANCES D'UN RESEAU DE MACHINES 52
2.1. Le matériel *52*
2.1.1. Généralités 52
2.1.2. Matériel réseau 53
2.2. Les pilotes de périphériques *55*
2.3. Le système d'exploitation *55*
2.4. Le logiciel *56*
3. LIMITES REELLES OBSERVEES SUR UN RESEAU DE MACHINES 57
3.1. Mesures de performances générales *57*
3.1.1. Les messages de taille inférieure au kilo-octet 59
3.1.2. Les messages de taille supérieure au kilo-octet 59
3.2. Coût mesuré du débit *60*
3.3. Coût mesuré de la latence *63*
3.3.1. Mesures de la latence 63
3.3.2. Proportion d'information utile dans un message 64
3.3.3. Coûts additionnels 64
3.4. Conclusion sur les performances des communications *65*
3.5. Impact de ces deux critères sur un cas concret : la multiplication de matrices *66*
3.5.1. Distribution selon une granularité minimum 67
3.5.2. Distribution selon une granularité moyenne 68
3.5.3. Distribution selon une granularité épaisse 70
3.5.4. Conclusion 73

Les paragraphes qui suivent portent encore sur des considérations architecturales, toutefois les réseaux de machines étant la cible privilégiée de notre étude, nous avons jugé opportun de leur consacrer un nouveau chapitre. Comme pour les supercalculateurs, une rapide présentation de l'architecture des réseaux d'ordinateurs va être donnée. Nous étudierons ensuite les points clés de ces derniers et enfin nous tenterons de trouver quelques pistes permettant de gommer les principaux défauts de ce type de machine. L'aspect des communications sera particulièrement développé afin de bien comprendre les coûts inhérents à celles-ci et de pouvoir ensuite en tirer le meilleur parti. Plusieurs mesures concrètes seront rapportées et analysées dans ce chapitre.

1. Caractéristiques générales des réseaux d'ordinateurs

Nous l'avons vu : un réseau de machines est une interconnexion de machines généralistes à travers un réseau généraliste. Un réseau de machines peut éventuellement se présenter sous la forme d'une grappe, les composants (nœuds, serveurs, commutateur réseau …) sont alors regroupés dans un seul boîtier ou dans une armoire, la grappe de processeurs est un réseau de machines dédié à son utilisation de calculateur haute performance.

Actuellement, le matériel généraliste est constitué d'une part de machines offrant de bonnes performances et d'autre part de réseaux aux performances souvent médiocres. Le stéréotype actuel du réseau de machines est donc un ensemble de machines performantes interconnectées par un réseau peu performant. Cette description bien que très simpliste voire caricaturale, n'en est pas moins une base incontournable qu'il faut sans arrêt garder à l'esprit lors de quelque travail que ce soit sur ce type de machine.

L'équilibre relatif observé sur les supercalculateurs entre la puissance de calcul des nœuds et les performances du réseau qui les relie n'est donc plus de mise avec les réseaux d'ordinateurs. Cette différence entre les deux architectures est en grande partie la source de l'inefficacité trop souvent constatée lors de l'application directe d'algorithmes issus des supercalculateurs sur des réseaux d'ordinateurs. Il conviendra par conséquent de porter une grande attention à ce déséquilibre entre les coûts des calculs et les coûts des communications pour exploiter correctement une machine parallèle de type réseau d'ordinateurs.

Une autre caractéristique importante des réseaux de machines est l'absence de mémoire physiquement partagée. Notons que cette caractéristique implique que le partage d'information au sein d'un réseau de machines s'obtienne par des échanges de messages. Ces échanges peuvent être explicites : la programmation est alors qualifiée de

programmation par passage de messages ; ou implicites : dans le cadre de l'utilisation de mémoire distribuée virtuellement partagée (voir Chapitre II : 2.2).

Nous pouvons aussi remarquer l'absence d'horloge globale, les nœuds du réseau de machines fonctionnent de manière asynchrone. Toute synchronisation sera donc explicite et obtenue grâce à des échanges de messages.

Comme le matériel, les logiciels utilisés dans le cadre des calculs sur les réseaux d'ordinateurs sont plus souvent généralistes que spécialisés. Il faut noter que les outils disponibles pour les réseaux d'ordinateurs sont assez récents et que si certains sont très aboutis, d'autres manquent à l'appel ou sont encore au stade de prototype. Le mouvement GNU de logiciels libres est très représenté dans la communauté du calcul sur réseaux d'ordinateurs, ces logiciels sont l'objet d'un développement rapide et le réseau de passionnés qui les entourent est un gage de circulation rapide d'information et d'entraide le cas échéant.

2. Limites potentielles des performances d'un réseau de machines

Lorsque l'on s'intéresse aux performances des réseaux de machines il est opportun de les considérer comme un ensemble de couches. Ceci est souvent fait sur la base des trois couches suivantes : la couche physique, c'est à dire le matériel, la couche système et la couche application. Je découpe pour ma part la couche système en deux niveaux correspondant aux pilotes de périphérique d'une part et au système d'exploitation de l'autre. Ce chapitre présente donc les quatre couches que je viens d'évoquer et leur impact sur les performances globales.

2.1. Le matériel

2.1.1. Généralités

Les limites imposées par le matériel sont les plus connues, elles servent souvent d'indicateurs de la performance des machines. On parlera par exemple de machines 2,2 GHz possédant une mémoire de 256 Mo à 20 ns et interconnectées par un réseau 1 Gbit/s. Ce que nous indiquons ainsi est que les nœuds de calcul sont basés sur des processeurs dont la fréquence d'horloge est de 2,2 GHz dont la mémoire (hors caches) a

une capacité de 256 Mo et un temps d'accès moyen de 20 ns et le réseau qui relie ces nœuds possède un débit maximal théorique de 1 Gbit par seconde.

Je pense que la plupart de ces valeurs devraient être vues comme des limites et en aucun cas comme un indicateur fiable de la performance du matériel en question.

2.1.2. Matériel réseau

Les limites matérielles qui nous concernent en priorité sont les limites du réseau, elles sont principalement celles de l'interface réseau utilisée dans chaque nœud de la machine distribuée. Deux valeurs sont habituellement retenues :

- le **débit** : nombre maximal théorique de bits qui peuvent transiter à travers l'interface par secondes
- la **latence** : temps de traversée d'un bit de bout en bout c'est à dire jusqu'à ce que l'interface réseau du destinataire l'ai rendu disponible à celui-ci. Une autre définition de la latence (lue dans [KRA97]) est le temps d'accès à une information vide (et distante dans le cas d'une latence réseau)

Le débit et la latence sont les deux valeurs généralement fournies par les constructeurs de matériel réseau (cf Exemple 2)

	Ethernet	**Fast-Ethernet**	**Gigabit Ethernet**	**Giganet**	**Myrinet**	**SCI**	**Earth Simulator (NEC)**
Latence				8µs	5µs	2,4µs	
Débit	10Mb/s	100Mb/s	1Gb/s	1,25Gb/s	1,28Gb/s	8Gb/s	16Gb/s

Données relevées dans [AUG00]

Exemple 2 : Débits et latences annoncés de quelques réseaux

Ces deux valeurs sont capitales, bien entendu, mais d'autres caractéristiques de l'interface réseau sont essentielles, je pense en particulier à la topologie du réseau, la présence d'un accès direct à la mémoire (DMA : Direct Memory Access) qui permet d'accéder aux données en mémoire sans interrompre le processeur d'exécution et donc le système. La rencontre entre ce mécanisme d'accès direct à la mémoire et les réseaux, a donné naissance à ce que l'on appelle l'accès direct à la mémoire distante (RDMA : Remote Direct Memory Access) dont on peut trouver une étude détaillée et appliquée au domaine des réseaux de

machines dans [GLU02]. Le RDMA modifie sensiblement le comportement et les performances du réseau.

Un autre composant qui a son importance dans les performances d'un réseau est le commutateur. Chaque interface réseau sera reliée directement à celui-ci et il assurera la transmission vers l'interface destinataire soit par diffusion, on parle alors de concentrateur ou « hub », soit uniquement au destinataire réel par lecture de l'adresse du paquet on parle alors de commutateur ou « switch ». On trouve plusieurs qualités de commutateur en fonction de leur niveau de connectivité, ceux assurant une interconnexion totale sont qualifié de « crossbar » (à l'intérieur de cette dernière catégorie il existe des sous catégories, la plus performante étant le « full-crossbar » ou une interconnexion directe entre chaque nœud est assurée), ils permettent à deux nœuds quelconques du réseau de communiquer au même moment que deux autres (pas de séquentialisation des messages). Le cas des « crossbar » mis à part, deux messages arrivant au même moment à un commutateur sont généralement traités l'un après l'autre tandis que deux messages arrivant au même moment à un concentrateur produisent ce que l'on nomme une collision qui va nécessiter la réexpédition des deux messages par leurs émetteurs au bout d'un délai aléatoire.

Nous pouvons noter au passage que, sous des aspects extérieurs très similaires, l'utilisation des différents types de matériels ci-dessus modifie la topologie du réseau :

- un concentrateur mènera à une topologie en bus (topologie d'origine des réseaux Ethernet)
- un commutateur non totalement interconnecté produira une topologie en étoile (Ethernet commuté)
- un « crossbar » correspondra à une topologie dite d'interconnexion totale ou chaque nœud est physiquement connecté à tous les autres.

Outre son type, une valeur importante concernant le matériel d'interconnexion est son temps de traversée. Les temps de traversée des commutateurs varient en fonction de la qualité des composants utilisés et du type de connexion, dans les « crossbar » toutes les routes sont câblées, le temps de traversée est donc relativement court dans les autres commutateurs, la route est établie dynamiquement et le temps de traversée est donc plus important. Le temps de traversé d'un concentrateur est très court puisque celui-ci se contente de diffuser le signal reçu sur l'ensemble des câbles.

2.2. Les pilotes de périphériques

Les pilotes de périphériques ont un rôle de jonction entre le matériel et le système. La plupart des matériels physiques dispose d'au moins un pilote de périphérique qui permet à leur utilisateur d'y accéder avec plus ou moins de facilité.

Les pilotes de périphériques influent sur les performances globales de deux manières : en offrant à l'utilisateur (généralement le système) l'accès à toutes les fonctions du matériel et par leurs performances intrinsèques.

- L'adéquation d'un pilote de périphérique avec le matériel précis dont il a la charge est primordiale : les pilotes généralistes négligent trop souvent certains raffinements de tel ou tel matériel privant alors l'utilisateur de fonctions parfois précieuses. Ces pilotes sont fréquemment soit fournis avec le matériel, ils sont alors généralement issus du constructeur qui fournit par défaut un pilote unique capable de gérer toute la gamme à laquelle le matériel appartient ou bien fourni avec le système d'exploitation, il s'agit alors souvent de version allégée des pilotes (la grande quantité de pilotes à regrouper en étant la principale raison).

- Les performances intrinsèques du pilote de périphérique proviennent quant à elles de la qualité du codage et de son adaptation au système cible (utilisation des instructions avancées du processeur par exemple). L'idéal sur ce point étant souvent de disposer des sources du pilote pour pouvoir en optimiser la compilation pour le système particulier sur lequel nous travaillons.

L'importance des pilotes de périphérique ne doit pas être sous-estimées, leur fréquence d'utilisation est souvent la même que celle du matériel et les différences entre les versions sont importantes (Nous avons constaté lors de nos expérimentations des différences de 20 à 30% suivant les pilotes utilisés pour les interfaces réseaux).

2.3. Le système d'exploitation

Le système d'exploitation doit assurer la sécurité et le bon fonctionnement de la machine mais il est difficile de vérifier chaque application indépendamment. La solution utilisée par les systèmes d'exploitation est l'interfaçage des espaces réels à travers des espaces virtuels (HAL : Hardware Abstract Layer de Windows NT par exemple) ainsi en cas de problème celui-ci va se situer dans l'espace virtuel et les conséquences seront plus réduites. C'est pour des raisons similaires que la plupart des systèmes disposent d'un espace noyau et d'un espace utilisateur n'ayant pas les mêmes accès. Les fonctions traditionnelles de

communication doivent traverser ces différentes couches pour s'exécuter, leurs performances sont fortement influencées par ce parcours.

Le système d'exploitation a d'autre priorité que la performance et le coût de celui-ci est lourd sur les opérations telles que les communications.

2.4. Le logiciel

Le logiciel est la couche de plus haut niveau, il doit composer avec les services offerts en principe par le système d'exploitation qui cumule donc les défauts des matériels, de leurs pilotes et du système lui-même.

Les applications proprement dites ont la plupart du temps recours à des librairies pour accéder à des ressources complexes telles que le réseau. L'utilisation d'une librairie de communications permet de bénéficier d'une interface simplifiée par rapport au primitives offertes par le système qui restent assez fastidieuses à manipuler.

Les librairies sont donc une couche supplémentaire dont les performances influeront directement sur celles qui nous intéressent le plus c'est à dire celles dont nous disposerons réellement pour notre application. Ces performances dépendent finalement de l'algorithme codé, des librairies utilisées du (ou des) compilateur(s) utilisé(s) et de la manière dont celui-ci (ceux-ci) a(ont) été(s) utilisé(s).

L'impact des différentes couches évoquées ci-dessus sur les performances réellement disponibles est important et c'est pour cette raison que toute la « chaîne » communication doit être optimisée et qu'un matériel très performant ne suffit pas [ONG00]. Malheureusement même en optimisant avec soin chaque maillon de cette chaîne les performances dont bénéficie l'application restent assez éloignées des maxima théoriques (cf Exemple 3).

	Giganet	Myrinet	SCI
Latence mesurée (annoncée)	**20µs** (8µs)	**12µs** (8µs)	**6µs** (2,4µs)
Débit mesuré (annoncé)	**96Mb/s** (1,25Gb/s)	**100Mb/s** (1,28Gb/s)	**70Mb/s** (8Gb/s)

Mesures de performances en utilisant une librairie de communication MPI optimisée [AUG00]

Exemple 3 : Comparaison entre les valeurs annoncées et mesurées de quelques réseaux

3. Limites réelles observées sur un réseau de machines

Les observations en amont nous avaient amenés à penser que les communications seraient la principale source de limitations aux performances des machines de type « réseau d'ordinateurs ». Quelques mesures permettent de très vite se rendre compte de la validité de cette hypothèse et montrent que la différence entre les performances idéales et les performances réelles provient effectivement des communications, nous allons essayer d'analyser plus précisément la ou les sources de ce phénomène.

3.1. Mesures de performances générales

Dans un premier temps je vais présenter des performances globales sans chercher à isoler tel ou tel autre aspect des communications, mon premier test sera l'envoi de message de différentes tailles à travers le réseau.

Les résultats de ces tests sont donnés pour les différents environnements sur lesquels j'ai été amené à travailler et dont une description détaillée figure en annexe.

Mesures du temps de transmission sur les différentes machines

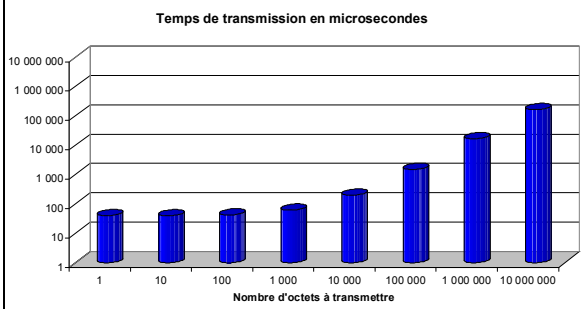


Figure 5 : Temps de communication mesuré sur la grappe du DTIM (Réseau Myrinet)

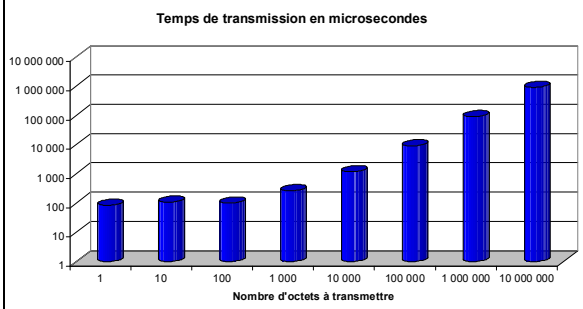


Figure 6 : Temps de communication mesuré sur la grappe du DTIM (réseau Fast Ethernet)

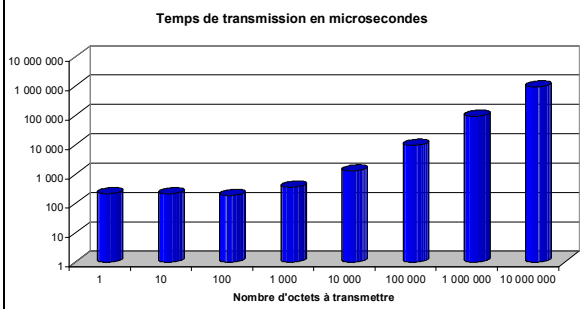


Figure 7 : Temps de communication mesuré sur la grappe Graphique (Fast Ethernet)

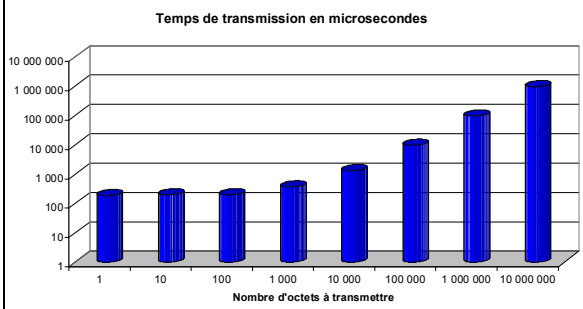


Figure 8 : Temps de communication mesuré sur la grappe de Supaero (Fast Ethernet)

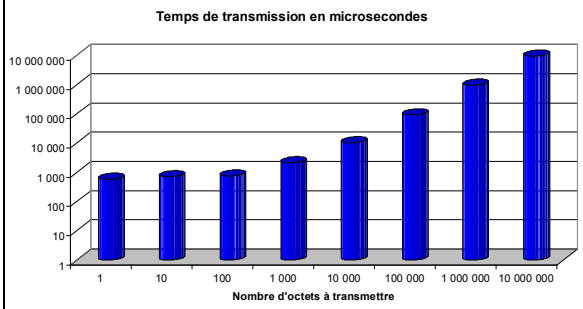


Figure 9 : Temps de communication mesuré sur le réseau de stations Sun (Ethernet)

Figure 10 : Courbes de mesure du temps de communication sur des machines réelles

Sur les différentes courbes (Figure 10) on remarque deux zones distinctes, une première concerne les messages de tailles inférieures au Kilo-octet, elle est caractérisée par un temps de transmission constant quel que soit le volume de données à transmettre, au-delà du Kilo-octet le temps de transmission croît de façon linéaire par rapport au volume de données à transmettre.

Il est intéressant de constater que ce seuil est le même quel que soit le réseau utilisé. Une seconde caractéristique remarquable sur ces courbes est que le temps de transmission d'un message de petite taille (i.e. de taille inférieure au seuil précédemment évoqué : 1Ko) est moins influencé par le passage d'un réseau à l'autre que les messages plus importants.

3.1.1. Les messages de taille inférieure au kilo-octet

La première zone citée précédemment correspond à des messages dont le temps de transmission est largement dominé par la latence de communication. Le débit n'intervient pratiquement pas dans le temps de transmission total. Ceci explique la constance de ce dernier pour les différentes tailles de messages en deçà du kilo-octet.

Le temps de préparation du message : temps de traversée des différentes couches évoquées au Chapitre III : paragraphe 2, temps d'encapsulation des données etc. définit un seuil minimum en dessous duquel il n'est pas possible de descendre. Ce temps de préparation n'est pas totalement constant en fonction de la taille du message ; il évolue en revanche bien moins rapidement qu'elle, puisque seules quelques étapes de ce processus de préparation de la communication sont concernées par la taille du message : les recopies éventuelles des données de l'espace utilisateur vers l'espace système puis vers l'interface réseau, le découpage éventuels des données en plusieurs messages suivant les paramètres du réseau.

3.1.2. Les messages de taille supérieure au kilo-octet

La seconde zone évoquée précédemment correspond aux messages de taille suffisante pour « digérer » la latence des communications. Ces messages montrent donc un temps de communication influencé à la fois par la latence et par le débit du réseau (Ce qui est bien entendu le cas pour tout message mais pas de manière aussi visible). Les temps de communication mesurés ne sont plus constants en fonction de la taille des messages ; ils évoluent de manière quasi linéaire par rapport à celle-ci ce qui correspond principalement à l'impact du débit du réseau sur le temps final de communication. Lorsque l'évolution du temps de communication final devient totalement linéaire par rapport à la taille du message c'est que le temps de latence est devenu négligeable par rapport au temps de « transport » des données.

3.2. Coût mesuré du débit

Ce que nous cherchons à mesurer ici est le débit réel offert par l'interface réseau accompagnée de son driver et d'une librairie de communication standard. Pour cela reprenons les résultats des chronométrages du 3.1 et ré-examinons les sous forme de débit.

Si l'on observe par exemple la grappe du DTIM (Fast Ethernet), on obtient le graphe suivant :

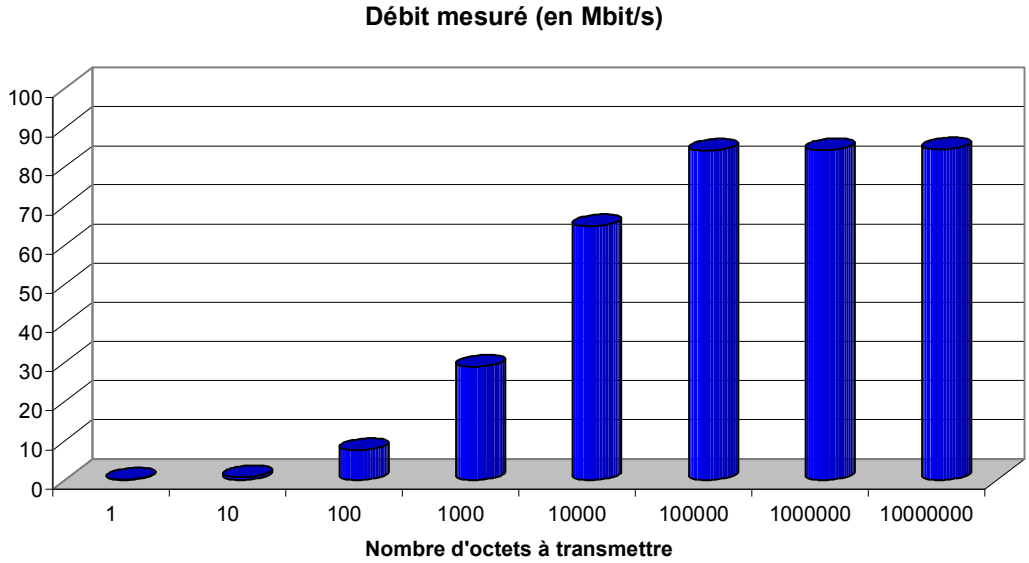


Figure 11 : Débit mesuré sur réseau de type Fast Ethernet

Ce graphe met en valeur plusieurs points importants :

- Le premier point est aujourd'hui admis par tous : les messages de petite taille sont défavorables à une exploitation efficace des performances d'un réseau actuel. Les résultats sur les temps de communication du chapitre précédent nous l'avaient fait pressentir mais nous pouvons mieux ici en apprécier la proportion puisque l'on peut lire sur le graphe que pour des messages d'une dizaine d'octets le débit réel d'un réseau supposé 100Mbit/s est d'environ 1Mbit/s soit une dégradation d'un facteur de 100 de ses performances.

- Le second point important est qu'il faut atteindre une taille d'environ 100 Ko pour que le débit soit très proche du maximum, l'atténuation rapide de la pente de la courbe indiquant l'approche d'un extremum. Ce seuil fait apparaître la troisième

zone suggérée dans le 3.1.2, celle ou le volume de données à transmettre est suffisant pour rendre négligeable le temps de communication provenant de la latence.

En conclusion, notre réseau 100Mbit/s annoncé a donc un débit réel maximum mesuré de 85Mbit/s ce qui semble très correct mais ce niveau de performance n'est réellement atteint que pour des messages dont la taille dépasse 100 Kilo-octet. Cette dernière remarque fixe un seuil qui peut s'avérer gênant pour certaines utilisations d'un tel réseau puisque, mis à part pour d'importants échanges de données, les informations à transmettre durant l'exécution d'un programme distribué sont loin d'atteindre un tel volume.

Les mesures sur les autres systèmes montrent des profils très similaires avec, en particulier, le même seuil de 100 Ko (Figure 16). On peut tout de même remarquer que le réseau de station Sun atteindrait son débit crête légèrement plus tôt que les autres (Figure 15) et que le réseau Myrinet affiche quant à lui un débit crête beaucoup plus éloigné de son maximum théorique avec seulement 400 à 500 Mbit/s réels contre 1 Gbit/s annoncés (Figure 12).

Courbes de mesures du débit sur les différentes machines

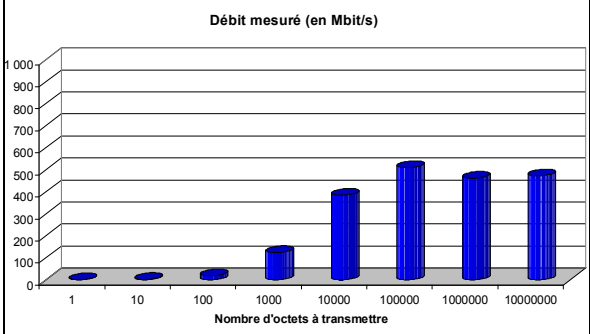


Figure 12 : Débit de la grappe du DTIM (réseau Myrinet)

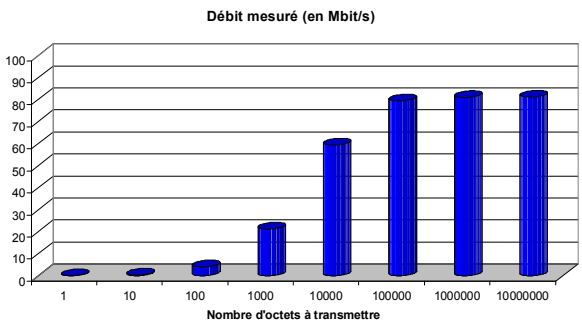


Figure 13 : Débit de la grappe graphique (Fast Ethernet)

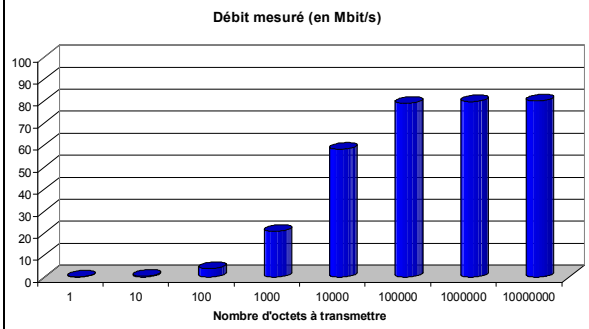


Figure 14 : Débit de la grappe Supaero (Fast Ethernet)

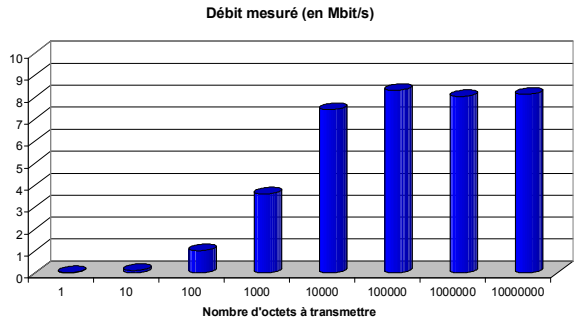


Figure 15 : Débit du réseau de stations Sun du DTIM (Ethernet)

Figure 16 : Courbes de mesure du débit réseau sur quatre machines réelles

L'ensemble de ces mesures nous amène à la conclusion suivante :

- Pour de petits messages les différents réseaux sont assez proches les uns des autres et montrent tous de mauvais débits
- Pour les messages volumineux par contre les rapports de performances entre les différents réseaux sont bien plus fidèles à ceux qui sont annoncés. Nous avons un rapport très proche de dix entre le débit de l'Ethernet et celui des Fast Ethernet.

Dans une moindre mesure nous pouvons observer le même phénomène pour Myrinet offrant un débit environ 7 fois supérieur à celui du Fast Ethernet (contre 10 à 12 fois annoncé)

3.3. Coût mesuré de la latence

Concrètement le coût de la latence est le temps minimum, qu'il va falloir payer pour transférer une information aussi petite soit elle. Une estimation de celle-ci peut donc être obtenue en observant les temps mesurés dans ce que j'ai qualifié de première zone dans les courbes de la Figure 10.

3.3.1. Mesures de la latence

Les valeurs relevées ont été reportées sur le graphique suivant (Figure 17) :

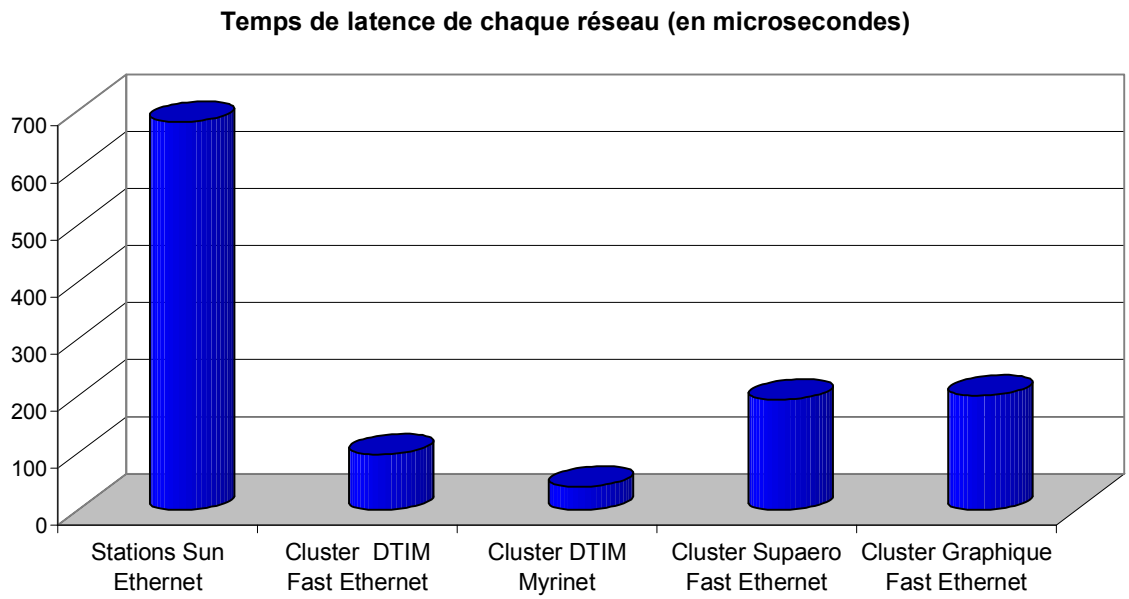


Figure 17 : Estimation de la latence par des mesures

Les différences observées entre les réseaux semblent importantes au premier regard mais si nous les comparons avec les rapports entre les débits des mêmes réseaux nous nous rendons compte que les progrès faits sur le débit des réseaux ne sont malheureusement pas les mêmes pour la latence. Nous observons un rapport entre la latence d'un Ethernet

original et celle d'un Fast Ethernet de seulement 3,5 (contre un rapport de 10 en ce qui concerne le débit) et un rapport de seulement 3 entre les même Fast Ethernet et le Myrinet (contre 7 pour le débit).

L'inefficacité des réseaux pour la transmission de petit volume d'information est en train de diminuer dans l'absolu mais si nous la relativisons par rapport à l'efficacité du réseau à transmettre de gros volume de données alors nous ne pouvons que constater l'aggravation de cette inefficacité.

Les causes de la latence proviennent de plusieurs facteurs, le tout premier étant la proportion d'information utile dans ce qui est réellement transmis et le second les latences matérielles et logicielles dues aux protocoles utilisés (le temps de traversés des câbles et interfaces physiques pourraient être ajouté à cette liste).

3.3.2. Proportion d'information utile dans un message

Chaque message n'est pas constitué uniquement des données à transmettre, à celles-ci viennent s'adjoindre des informations de protocole nécessaire au bon fonctionnement du réseau. Les protocoles utilisés sont parfois assez anciens et lourds, ils ont par contre le mérite d'être universels.

Au niveau logiciel, il n'est pas rare d'utiliser un autre protocole réseau, souvent TCP/IP, c'est le cas de nos réseaux test. Avant de parvenir à l'interface réseau le message a donc déjà été encapsulé dans une trame TCP et reçu au passage un en-tête de 24 octets puis dans une trame IP et reçu là aussi une en-tête de 24 octets soit au total 48 octets d'en-tête quel que soit le volume de données initial.

Une trame Ethernet (valable pour Ethernet comme pour Fast Ethernet) comporte 26 octets de contrôle qui seront associés aux données avant de les transmettre ; cet ajout est effectué au niveau matériel par l'interface réseau elle-même.

Ce volume d'informations supplémentaires (74 octets dans la configuration prise en exemple) n'est pas négligeable lorsque l'on ne transfère que quelques octets, il le devient bien sûr dès que le volume de données utile à transmettre devient plus important.

3.3.3. Coûts additionnels

Parmi les coûts additionnels on recense les manipulations de l'interface réseau mais aussi le temps de traversée du commutateur (dans le cas d'Ethernet commuté) et parfois il est même tenu compte des coûts des erreurs sur la ligne, pondérés par la fréquence moyenne de leur survenue. Les coûts logiciels existent à deux niveaux, au niveau du système

d'abord, qui transmet la demande d'émission de l'utilisateur puis au driver qui la reçoit et la fait suivre à l'interface matérielle, ces deux opérations s'effectuant souvent avec copie des données à transmettre.

Un comportement du réseau est important à signaler ici : une baisse importante des performances se produit lorsque le taux de communication atteint un certain seuil. Dans les réseaux de types Ethernet non commuté, le taux de collisions augmente alors de façon exponentielle, il entretient de plus un certain nombre de messages supplémentaires issus des échecs dus aux collisions elles-mêmes. Dans un contexte commuté, le commutateur doit stocker plus de messages et ses performances seront mises à rude épreuve (l'impact sur les performances dépendra directement des capacités du commutateur à gérer de telles situations).

3.4. Conclusion sur les performances des communications

Les réseaux ont fait des progrès considérables depuis quelques années bouleversant les références classiques. En effet, contrairement à ce que nous pouvions observer il y a quelques années la bande passante n'est plus le point faible des réseaux mais au contraire son point fort. La latence par contre est devenue le coût le plus important des communications sur les systèmes actuels. Comme nous l'avons exposé dans le chapitre précédent, le matériel n'est pas le seul responsable de cet état de fait, le système aggrave grandement le coût de la latence matérielle d'une communication en y ajoutant souvent un protocole lourd induisant un coût logiciel important.

Nous avons pu observer que les messages pouvaient être classés selon trois catégories :

- les petits messages, inférieurs au Kilo-octet, dont le temps de transmission est principalement lié à la latence,
- les messages de taille moyenne, dont le temps de communication est le résultat à la fois de la latence et du débit
- les gros messages dont la taille permet de ne plus percevoir l'impact de la latence.

Parmi ces trois catégories de messages, les plus pénalisants pour le système sont bien entendu les petits messages, ils sont malheureusement aussi les plus fréquents dans les applications distribuées puisque, entre autres, tous les messages de contrôle et de synchronisation rentre dans cette catégorie.

Les messages de taille moyenne forment le reste des communications traditionnellement observées dans une application distribuée, les gros messages restant relativement exceptionnels.

L'utilisation actuelle des réseaux semble donc mal adaptée aux caractéristiques de ceux-ci. L'utilisation efficace des outils de communication étant très liée à la taille des messages, nous pouvons envisager d'**utiliser la taille moyenne des messages comme un indicateur de la « bonne exploitation » du réseau.** Si toutefois l'écart-type correspondant était trop important, d'autres indicateurs pourraient compléter l'analyse d'un algorithme distribué du point de vue des communications, comme des informations statistiques sur la proportion des différentes tailles de messages ainsi que les fréquences différenciées pour chacune de ces catégories de messages (courts, moyens ou longs).

3.5. Impact de ces deux critères sur un cas concret : la multiplication de matrices

La multiplication de matrice distribuée efficace est un sujet de recherche pour de nombreux scientifiques [CHA98], je tiens donc à préciser ici qu'il ne s'agit en rien de trouver un algorithme efficace mais seulement d'observer l'impact des communications sur un cas réel mais simple à appréhender.

L'algorithme le plus simple possible de multiplication de matrice (triple boucle) sera donc utilisé et distribué selon trois granularités différentes :

- Granularité fine : distribution au niveau de la multiplication,
- Granularité intermédiaire : distribution au niveau du calcul d'un élément de la matrice résultat,
- Granularité épaisse : distribution au niveau du calcul d'une partie de la matrice résultat (plusieurs colonnes) représentant une charge de travail identique pour chaque nœud (le nombre de colonnes de la matrice résultat divisé par le nombre de processeurs).

Un prétraitement de la matrice B afin de ranger ses éléments par colonne est tout de même effectué pour chaque version de l'algorithme de multiplication de matrices, qu'il soit séquentiel ou distribué, afin d'optimiser sensiblement les performances de l'algorithme brut (des gains de 72% on été observés grâce à cette manipulation sur un Sun ultra 10 et d'environ 30% sur un des nœuds de la grappe du DTIM).

A des fins de comparaison, le temps de calcul selon l'algorithme séquentiel sur un nœud de la machine cible sera systématiquement rappelé.

Cette expérience a été réalisée sur la grappe du DTIM en utilisant le réseau Myrinet avec des matrices de flottants double précision de taille 1024 x 1024.

Pour chacun des algorithmes distribués, une analyse de la distribution du point de vue de la granularité sera donnée sous la forme d'un tableau commenté (Pour que cette analyse reste générale, le cas de la multiplication d'une matrice m x n par une matrice n x p servira de référence). Une analyse correcte de la granularité de distribution d'un algorithme met en place quatre facteurs qui sont : la complexité d'une opération distante, le nombre de ces opérations, la taille des messages échangés ainsi que leur nombre. On peut évidemment envisager de restreindre le nombre de ces facteurs mais, à mon avis, seul le nombre d'opérations distantes peut sans trop de perte être supprimé de l'analyse, les trois autres facteurs semblent primordiaux.

3.5.1. Distribution selon une granularité minimum

L'algorithme de multiplication de matrices devrait ici être parallélisé au niveau de la multiplication selon un modèle se rapprochant du SIMD. Ces conditions correspondent à l'application directe d'un algorithme prévu pour un calculateur parallèle d'une autre nature à un réseau de machines.

Le résultat n'offre aucune surprise : le temps d'exécution est faramineux car très nettement dominé par les communications (99,02 % du temps total).

L'analyse de la distribution effectuée par cet algorithme de multiplication de matrices parallèle donne les valeurs suivantes :

Analyse de la granularité			
Calcul		**Communication**	
Complexité d'une opération distante	Nombre d'opérations distantes	Taille des données dans les messages	Nombre de messages
1	m.n.p	2, 1(*)	m.n.p, m.n.p (*)

(*) les requêtes d'opérations distantes et les envois de résultats sont de tailles différentes, les deux valeurs indiquées correspondent respectivement aux requêtes d'abord et aux envois de résultats ensuite

Figure 18 : Détails d'une distribution de granularité trop fine

Une granularité similaire à celle qui serait utilisée sur un calculateur SIMD est évidemment trop fine pour un calculateur de type réseau de machines. Le choix d'une granularité trop fine mène à des temps calcul absolument rédhibitoires dans un contexte de haute-performance. L'utilisation du réseau ne porte ici que sur des messages très brefs de type messages de contrôle, la taille moyenne des messages (hors informations de protocole) est ici de 1,5 fois la taille d'un flottant double précision. L'impact négatif de la latence est multiplié par un très grand nombre de messages, les performances globales sont mauvaises.

Rappelons une règle évidente mais primordiale : **il n'est pas efficace de déporter une tâche dont le temps de traitement est inférieur au temps de communication nécessaire à ce déport**.

3.5.2. Distribution selon une granularité moyenne

L'algorithme de distribution de multiplication de matrices sera ici parallélisé selon le calcul d'un coefficient de la matrice résultat ce qui nous amène à une granularité moyenne, détaillée dans la Figure 19.

Analyse de la granularité			
Calcul		**Communication**	
Complexité d'une opération distante	Nombre d'opérations distantes	Taille des données dans les messages	Nombre de messages
2n	m.p	2n, 1 (*)	m.p, m.p (*)

(*) les requêtes d'opérations distantes et les envois de résultats sont de tailles différentes, les deux valeurs indiquées correspondent respectivement aux requêtes d'abord et aux envois de résultats ensuite

Figure 19 : Détails d'une distribution de granularité moyenne

Chaque nœud est maintenant chargé d'effectuer la somme des produits d'une ligne de A avec une colonne de B. Le réarrangement de la matrice B selon un stockage par colonne a maintenant un double effet : comme dans les cas précédents il permet d'accélérer l'accès aux données de B mais il permet également de ne pas avoir à sérialiser ces données avant de les transmettre et les remettre dans une structure adéquate après. Ceci étant un choix important, j'ai préféré faire apparaître cette étape dans l'algorithme. Ensuite pour chaque coefficient de la matrice résultat, une requête sera envoyée vers un nœud. Enfin, toutes les requêtes seront récupérées et les résultats structurés en matrice.

L'algorithme de multiplication de matrices selon ce schéma de distribution est le suivant (Figure 20) :

Algorithme de multiplication de matrices distribué sur P nœuds selon le calcul d'un coefficient C(m,p) ← A(m,n) x B(n,p)

```
Début
  // Préparation de la matrice B
  Classement de B selon les colonnes
  // Envoi des requêtes
  Pour k=1 à p faire
    Pour j=1 à m faire
      Demande de calcul au nœud ((p*m) mod P)(A[m,*],B[k,*])
    Fin Pour
  Fin Pour
  // Récupération des résultats
  Pour k=1 à m*p faire
    Récupération d'un coefficient
    C[coef.ligne, coef.colonne] ← coef.valeur
  Fin Pour
Fin
```

Figure 20 : Algorithme distribué de multiplication de matrices 2

Cette granularité va s'avérer encore trop fine pour l'architecture cible, le temps de calcul sur quatre nœuds d'une multiplication de matrices 1024x1024 est cette fois d'environ 26 minutes et 40 secondes. Ce temps est bien sûr beaucoup moins important que celui de l'algorithme à fine granularité (plus d'une journée de calcul) mais reste malheureusement très mauvais comparé au temps séquentiel (18 secondes). Le chronogramme de suivi d'exécution (Figure 21) nous montre que le trafic réseau reste très dense et qu'il est très certainement la cause des mauvaises performances évoquées ci-dessus.

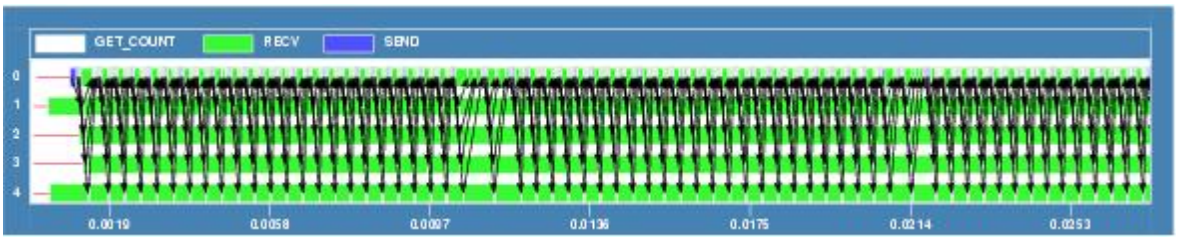


Figure 21 : Chronogramme d'un algorithme à granularité moyenne

La granularité choisie ici semble encore trop fine, une remarque importante à faire est que contrairement à ce que nous observions dans l'algorithme précédent cette dernière dépend de la taille des matrices à multiplier. La complexité d'une opération distante était fixe et valait 1 dans le premier algorithme alors que cette dernière vaut maintenant 2n et dépend donc linéairement de la taille des matrices de départ. Cela signifie qu'il est possible de déterminer (expérimentalement au moins) une taille de matrice minimum à partir de laquelle cette solution devient rentable.

3.5.3. Distribution selon une granularité épaisse

L'algorithme de multiplication de matrice sera maintenant distribué selon des blocs de matrices résultats, soit une granularité épaisse.

Dans ce cas précis la distribution sera obtenue en découpant la matrice finale en autant de part que de processeurs P participant au calcul. Chaque part est un ensemble de colonnes comprenant autant d'éléments pour chaque processeur. C'est le découpage le plus grossier que nous puissions faire sur ce calcul, la granularité obtenue est détaillée dans la Figure 22.

Analyse de la granularité			
Calcul		**Communication**	
Complexité d'une opération distante	Nombre d'opérations distantes	Taille des données dans les messages	Nombre de messages
m.2n.p/P	P	n.m, m.p/P, m.p/P (*)	P, P, P (*)

(*) Trois tailles de message différentes : envoi/réception de A en entier, envoi/réception d'un bloc de B, envoi/réception d'un bloc de C

Figure 22 : Détails d'une distribution de granularité épaisse

Concrètement, la matrice B est, comme dans l'exercice précédent, réarrangée suivant ses colonnes. Une boucle sur P processeurs envoie ensuite une requête à chaque processeur pour qu'il calcule un ensemble de colonnes de C, il est ici nécessaire de transmettre à chaque processeur la matrice A en entier et le bloc de colonnes de B correspondant au calcul demandé. Une seconde boucle sur P processeurs se charge de récupérer et d'arranger les résultats pour construire C.

L'algorithme de distribution prend alors la forme suivante (Figure 23) :

Algorithme de multiplication de matrices distribué sur P nœuds selon le calcul de blocs de coefficients
C(m,p) ← A(m,n) x B(n,p)

```
Début
  // Préparation de la matrice B
  Classement de B selon les colonnes
  // Envoi des requêtes
  Pour i=1 à P faire
    Demande de calcul au nœud i)(A[*,*],B[(i-1)*p/P ... i*p/P,*])
  Fin Pour
  // Récupération des résultats
  Pour i=1 à P faire
    Récupération d'un bloc de coefficients
    Mise en place du bloc dans C
  Fin Pour
Fin
```

Figure 23 : Algorithme distribué de multiplication de matrices 3

Pour la première fois depuis le début de cette expérience le temps de calcul est meilleur que le temps séquentiel avec un chronométrage à 13,5 secondes pour le calcul sur quatre nœuds d'une multiplication de matrices 1024x1024 contre un temps séquentiel de 18 secondes. Le chronogramme de suivi d'exécution (Figure 24) fait clairement apparaître le déroulement de cet algorithme :

- Au niveau du maître : la toute première étape est la création (ou la lecture) des matrices ; puis la matrice A dans son intégralité suivi d'un bloc de colonnes de la matrice B sont envoyés à chaque nœud de calcul l'un après l'autre (partie bleue de l'axe numéroté 0) enfin le maître tente de recevoir les résultats partiels et entre dans une première réception bloquante relativement importante avant la réception effective du premier résultat (partie verte de l'axe 0) et des trois suivants.

- Au niveau des nœuds : Le travail des nœuds commence par une réception (en vert) qui représente l'attente d'une tâche à effectuer. Sitôt les données de calcul reçues le calcul commence. Notons que l'attente des données prendra un temps différent pour chaque nœud puisque le maître effectue cet envoi « nœud par nœud ». La seconde phase est le calcul proprement dit (trait rouge sur le chronogramme). Contrairement à la phase d'attente, la phase de calcul à la même longueur pour chaque nœud (ceci est obtenu grâce à la régularité de la multiplication de matrice : le même volume de calcul entraîne le même temps de calcul). Enfin une dernière phase (rectangle bleu) est l'envoi au maître des données calculées localement.

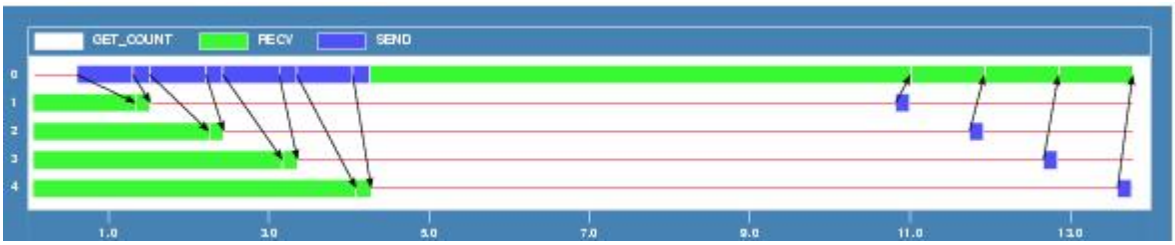


Figure 24 : Chronogramme d'un algorithme à découpage grossier

La granularité de cette solution est la plus adaptée à l'architecture cible, elle utilise de gros messages en petite quantité, réduisant l'impact de la latence sur les communications au profit du débit. Le temps de calcul n'est tout de même pas bon puisque l'utilisation de

quatre machines ne nous permet d'aller que 1,33 fois plus vite qu'en séquentiel. L'envoi séquentiel des données à chaque nœud de calcul (au départ) est en grande partie responsable de ce résultat. En travaillant sur des matrices de plus grande taille, le speed up est un peu amélioré ce qui montre une atténuation de l'impact de cette phase initiale. Mais pour obtenir un réel gain il faudrait modifier la manière de communiquer.

A défaut de réelle diffusion, MPICH propose une fonction optimisant l'envoi de donnée pour un contexte de diffusion (utilisation d'une mémoire tampon unique, envoi non bloquant …). L'utilisation de cette fonction pour envoyer la matrice A à tous le nœuds de calcul permet un gain important sur le temps de calcul : 8,5 secondes sont maintenant nécessaires au même calcul (contre 13,5 secondes en utilisation des envois de messages classiques).

Le chronogramme de suivi d'exécution (Figure 25) fait apparaître un envoi de A plus confus (l'ordre de « fin de réception de A » n'est pas celui dans lequel le maître distribue ensuite les blocs de B) mais le temps de transmission global de A est plus court que lors de l'envoi organisé de la solution précédente.

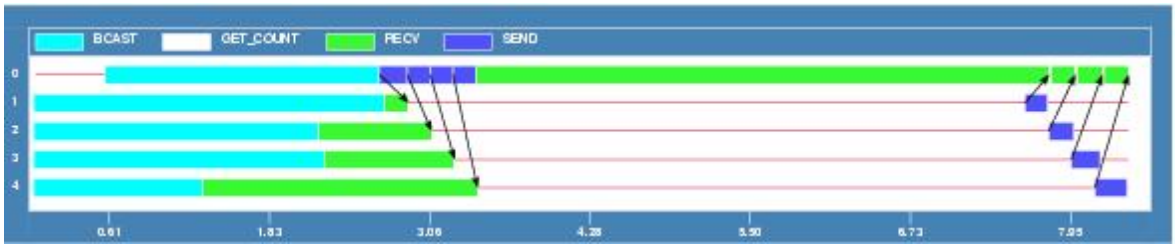


Figure 25 : Chronogramme d'un algorithme utilisant la fonction diffusion de MPICH

3.5.4. Conclusion

Les différentes versions de la multiplication de matrices mises en place dans ce paragraphe ont montré l'**impact très important de la latence sur le coût des communications**. Le premier algorithme, dont la taille des messages est la plus réduite possible pour que l'impact du débit soit le plus réduit possible (au profit de la latence), amène à des résultats catastrophiques en terme de performance.

Le **débit** a quant à lui un **impact moins négatif sur les performances** des communications, il est le point fort des réseaux utilisés sur les réseau de machines. Il présente en outre l'avantage de voir son impact réduit par des modes de communication différents.

Une autre information importante est apparue lors de cette expérience, il s'agit de l'échelle de granularité. Ce qui est habituellement considéré comme une granularité moyenne (celle que j'ai employée serait même facilement qualifiée de grossière dans le monde des supercalculateurs) se révèle une granularité fine dans le cadre des réseau de machines. **L'échelle de granularité est donc assez différente dans le monde du calcul en réseau de celle des supercalculateurs**. Il semble par ailleurs que **l'approche la plus adaptée à cette nouvelle architecture** en terme de performance ne soit plus « quelle est la granularité la plus fine à laquelle je puisse être efficace ? » mais plutôt **« quelle est la granularité la plus épaisse que je puisse utiliser pour effectuer ce traitement ? »**

Chapitre IV : Une approche efficace de la distribution

Sommaire

1. PRINCIPES 77
1.1. Mieux exploiter le débit 77
1.2. Réduire l'impact des latences 78
2. OUTILS CLASSIQUES 78
2.1. Le recouvrement 78
2.2. L'anticipation 80
2.3. Conclusion 82
3. NOUVEAUX OUTILS 83
3.1. La Paresse 83
3.1.1. Définition de la paresse 83
3.1.2. Paresse et distribution 87
3.2. La granularité adaptable 88
3.2.1. Généralités 88
3.2.2. Les algorithmes irréguliers 89
3.2.3. Solution proposée 89
3.3. Le regroupement de messages 90
4. CHOIX DE L'ALGORITHME A DISTRIBUER 91
4.1. Un calcul plus adapté de la complexité temporelle 91
4.1.1. Intégration de la complexité des communications 91
4.1.2. Seuil de rentabilité d'un algorithme de moindre complexité 92
4.2. Cas concret : La Transformation de Fourier d'ordre deux 93
4.2.1. La transformée de Fourier séquentielle 93
4.2.2. La transformée de Fourier parallèle 95
4.2.3. Conclusion 97
5. DEMARCHE DE DISTRIBUTION 98

1. Principes

Comme nous l'avons montré dans le chapitre précédent, le coût d'une communication est toujours formé de deux composantes : la latence et le débit. Le débit étant le point fort des interconnexions utilisées dans les réseaux de machines, il convient de lui donner le dessus par rapport à la latence.

1.1. Mieux exploiter le débit

L'exploitation du débit réseau est effective lorsque les messages transitant par le réseau sont de grande taille. Plus le volume de données est important, plus le rapport entre le coût de la latence et celui du transfert proprement dit sera favorable et donc plus l'efficacité de la communication sera grande.

L'utilisation d'une granularité importante, nécessitant souvent d'importants transferts de données, est souhaitable afin de tirer parti au mieux des hauts débits offerts par les réseaux actuels.

L'analyse de l'algorithme du point de vue de la taille des messages fournit des indications sur la qualité de l'exploitation du débit. Si les messages sont à peu près tous identiques, la taille moyenne peut suffire. S'il existe plusieurs catégories de messages il est possible de les caractériser par leur taille et le nombre de messages correspondants, cela permettra éventuellement d'orienter le travail d'optimisation si une anomalie est constatée.

Deux principaux moyens d'améliorer l'exploitation du débit sont donc : le choix de l'algorithme (méthode, granularité, …), qui permettra de favoriser des messages volumineux mais moins fréquents plutôt que de petits messages plus nombreux, et, dans un deuxième temps, le regroupement de messages qui peut permettre d'améliorer sensiblement encore la proportion de gros messages. Les tableaux d'analyse de la granularité, décrits dans le chapitre précédent, étendus par une différentiation des petits et des gros messages forment un outil d'analyse et de contrôle simple mais relativement efficace.

1.2. Réduire l'impact des latences

La latence est le point faible des interconnexions utilisées dans les architectures « réseaux de machines » courantes ; réduire son impact est nécessaire à l'obtention d'applications distribuées performantes.

Les mesures de choix d'algorithme et de regroupement de messages abordées dans le paragraphe précédent ont aussi pour effet de réduire l'impact de latence. Elles font partie des moyens algorithmiques.

Nous l'avons vu précédemment, le coût logiciel de la latence est très important. La première mesure à prendre, pour obtenir une amélioration globale de la latence et donc des communications, est le choix d'un protocole de communication performant. Certains protocoles font jusqu'à trois copies des données avant l'envoi effectif des données lorsque d'autres n'en font aucune (protocole zero-copie généralement à base d'accès direct à la mémoire distante) [SKE01, THA95].

Les moyens de lutter contre le coût des latences ne sont donc pas uniquement algorithmiques. Des choix adaptés au niveau de l'algorithme permettent de réduire le nombre de latences dont il faudra subir le coût mais la diminution de ce coût provient quant à lui du matériel (choix de cartes intelligentes, capable d'utiliser le DMA) et de l'environnement logiciel de communication (driver exploitant l'intelligence des interfaces réseaux et protocole efficace de communication).

2. Outils classiques

Des outils de programmation distribuée fréquemment rencontrés dans la recherche de réduction des coûts des communications sont le recouvrement, le travail par anticipation le remplacement de communications par des calculs, la redondance d'information et les envois de données volumineuses par blocs. Je ne vais pas ici décrire tous ces mécanismes, voici une rapide présentation de deux d'entre eux suivie d'une réflexion sur leur utilisation. Certains autres de ces mécanismes apparaîtront dans le reste de ce document.

2.1. Le recouvrement

Le recouvrement est une des techniques de programmation les plus connues, elle consiste à exécuter du code utile pendant les temps d'attentes générés par les communications au sens

large (les accès à un périphérique lent comme un disque dur peuvent ici être considérés comme des communications).

Le recouvrement consiste donc en l'utilisation de fonctions de communication non bloquantes permettant de continuer l'exécution d'instructions utiles pendant que la communication se déroule. Le principe du recouvrement est illustré dans l'exemple suivant :

Illustration du recouvrement

Soit le code suivant :

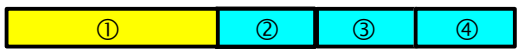
① Lire D

② C ← 2 x A

③ A ← B + C

④ C ← C + D

Les flots d'exécution ont donc l'aspect suivant :

Sans recouvrement	Avec recouvrement
 **Figure 26 : flot d'exécution sans recouvrement**	 **Figure 27 : flot d'exécution avec recouvrement**
L'opération de lecture est entièrement effectuée avant de continuer l'exécution	La fin de l'opération de lecture 1 n'est pas nécessaire à l'exécution des instructions 2 et 3, le temps d'attente est recouvert par celles-ci. La lecture est terminée au moment de l'exécution de l'instruction 4 qui en utilise le résultat, le temps de traitement global n'est plus influencé par la lenteur de l'opération de lecture.

Exemple 4 : Principe du recouvrement

Concrètement le recouvrement est une opération délicate :

- les instructions utilisées pour recouvrir la latence d'une opération de communication doivent être indépendantes de cette opération ;
- Il convient de s'assurer de la terminaison effective de l'opération de communication avant l'exécution de toute opération ne répondant pas au critère précédent.

Ce mode de fonctionnement nécessite une analyse de dépendance similaire à celle des pipelines à l'intérieur des processeurs mais devant être effectuée par le programmeur afin de décider de la portion de code qu'il peut exécuter pendant que la communication s'effectue. La limite de cette technique est facile à identifier : elle réside dans la différence de durée entre une communication et une instruction, sur les machines actuelles on observe généralement la proportion suivante : **une communication équivaut à plusieurs milliers d'instructions**. Certains réseaux performants permettent de rendre ce chiffre un peu plus raisonnable en le ramenant à environ un millier d'instructions seulement.

Nous venons de le voir l'utilisation du recouvrement peut être une solution efficace mais cette efficacité nécessite un nombre très important d'instructions indépendantes de la communication et ne s'applique donc pas à la majorité des cas courants. L'écart de performance entre la puissance de calcul des nœuds et le réseau qui les relie est responsable de cette restriction à l'utilisation plus générale du recouvrement. Malheureusement la puissance de calcul des machines évolue plus rapidement que la performance des réseaux, ce qui restreint de plus en plus le domaine d'efficacité de technique comme le recouvrement.

2.2. L'anticipation

Le recouvrement est déjà un moyen de travailler par anticipation mais ce cas particulier est suffisamment reconnu pour être traité à part. Le travail par anticipation est un point de vue différent de celui présenté lors du chapitre sur le recouvrement. On ne cherche plus à combler le temps que coûte une communication par du travail utile mais à faire cette communication le plus tôt possible afin que son contenu soit disponible au moment ou le destinataire en aura besoin.

Une illustration relativement évidente de cette approche réside dans un mode de fonctionnement client-serveur, le serveur attend traditionnellement une requête du client, la traite puis lui renvoie le résultat, le client quant à lui envoie une requête, attend le résultat puis recommence cet enchaînement jusqu'à ce que l'ensemble des requêtes qu'il désirait formuler soient traitées. Si on travaille par anticipation la séquence sera la suivante : le

client enverra sa requête, le serveur commencera à la traiter, au lieu d'en attendre le résultat le client formulera sa seconde requête et l'enverra au serveur. Le serveur ayant terminé le traitement de la première requête, il enverra le résultat et lira immédiatement la requête suivante reçue pendant le traitement de la première.

Le principe de l'anticipation est illustré dans l'exemple suivant :

<table>
<tr><th colspan="2">Illustration de l'anticipation</th></tr>
<tr><th>Sans anticipation</th><th>Avec anticipation</th></tr>
<tr><td>① A ← B+C
② C ← 2 x A
③ Lire D
④ C ← C + D</td><td>③ Lancer la lecture de D par anticipation
① A ← B+C
② C ← 2 x A
④ C ← C + D</td></tr>
<tr><td colspan="2">Les flots d'exécution ont donc l'aspect suivant :</td></tr>
<tr><td> 

Figure 28 : flot d'exécution sans anticipation</td><td> 

Figure 29 : flot d'exécution avec anticipation</td></tr>
<tr><td>Les opérations sont effectuées dans l'ordre. L'opération de lecture est entièrement effectuée avant de continuer l'exécution et ralentit donc l'ensemble du traitement.</td><td>L'ordre des opérations est modifié dans le but d'initier l'opération de lecture 3 plus tôt et de recouvrir celle-ci par l'exécution des instructions 1 et 2. La lecture est terminée au moment de l'exécution de l'instruction 4 qui en utilise le résultat mais une partie de son temps d'exécution n'affecte plus le temps de traitement global.</td></tr>
</table>

Exemple 5 : Principe de l'anticipation

L'anticipation étant basée sur le recouvrement, elle en conserve le principal inconvénient de n'être réellement efficace que dans des cas où de très nombreuses instructions sont disponibles. Cependant le réarrangement des instructions donne une liberté importante et permet de s'adapter à un nombre de cas plus importants.

2.3. Conclusion

Le recouvrement et l'anticipation permettent de faire disparaître une partie du coût des communications. Dans un cas idéal il ne reste de ce coût que celui de la requête effectuée par le CPU émetteur pour demander l'émission et par le CPU destinataire pour effectuer la lecture des données (dans sa mémoire locale).

Ces techniques souffrent de deux inconvénients majeurs :

- Leur domaine assez restreint d'efficacité dû au nombre trop important d'instructions indépendantes de la communication dont il faut disposer pour recouvrir la totalité de celle-ci. Ce nombre d'instructions augmente avec **l'évolution du matériel** ce **qui provoque une réduction progressive de l'efficacité de ces techniques**.

- L'utilisation de ces techniques nécessite l'emploi de fonctions de communication non bloquantes amenant à un code plus compliqué, plus difficile à déboguer et à mettre au point. La gestion de la synchronisation ainsi que celle des tampons d'entrées/sorties sont à la charge directe du programmeur. Un code inadapté peut être correct mais rendre totalement inefficace la communication.

Le recouvrement et l'anticipation sont de moins en moins adaptés au matériel au fur et à mesure de l'évolution de celui-ci. Ces techniques montrent encore une certaine efficacité lorsqu'elles sont bien utilisées et que l'algorithme s'y prête. **Le recouvrement et l'anticipation sont des techniques d'appoint, elles peuvent apporter un plus indéniable mais ne doivent venir qu'en complément d'une restructuration adaptée de l'algorithme**.

3. Nouveaux outils

3.1. La Paresse

3.1.1. Définition de la paresse

La paresse est un outil issu des langages fonctionnels, elle exprime le fait qu'une expression non réduite puisse être passée en paramètre à une fonction sans que l'évaluation de cette fonction ne provoque automatiquement la réduction de l'expression [LAU93]. L'illustration la plus courante de ce comportement est le constructeur de liste des langages fonctionnels (Figure 30).

La notation en lambda calcul, usuelle dans le domaine des langages fonctionnels, étant parfois hermétique, un commentaire sera systématiquement ajouté aux figures l'utilisant, dans une notation plus libre.

Non paresseux	**Paresseux**
$cons((\lambda x.e_1)y,(\lambda x.e_2)y) \rightarrow ([e_1]_x^y ; [e_2]_x^y)$ La liste construite contient les expressions réduites.	$cons((\lambda x.e_1)y,(\lambda x.e_2)y) \rightarrow (((\lambda x.e_1)y;(\lambda x.e_2)y)$ La liste construite contient les expressions non réduites.
Une fonction paresseuse recevant en paramètres des appels de fonction utilisera ses appels de fonctions au sein de son code plutôt que d'en provoquer l'évaluation préalable. Par exemple : construire_une_liste(3!, 5^2) renverra la liste (3!, 5^2) et non (6, 25).	

Figure 30 :Constructeur de liste paresseux et non paresseux

Deux propriétés de la paresse sont :

1. La « non-évaluation » des paramètres inutiles à un calcul donné
2. La possibilité de travailler avec des structures de données infinies.

3.1.1.1. Première propriété : non-évaluation des paramètres inutiles

En ce qui concerne la propriété de non-évaluation des paramètres inutiles, considérons l'exemple suivant :

$$(\lambda xy\ x)(2\ 10!)$$

Cette fonction reçoit deux paramètres en entrée et en renvoie le premier

Exemple 6 : Une fonction n'utilisant pas tous ces arguments

Le résultat de l'évaluation de l'expression ci-dessus est « 2 » mais les étapes nécessaires pour aboutir à ce résultat diffèrent selon les caractéristiques du langage utilisé.

La traduction de la lambda-expression de l'Exemple 6 dans un langage fonctionnel serait la suivante : car(cons(2 fact(10))), son évaluation peut prendre les deux formes suivantes (Figure 31) :

Langage strict	Langage paresseux
car(cons(2 fact(10))) →car(cons(2 3628800)) →car(2 3628800) →2 L'évaluation de 10! a lieu avant de pouvoir fournir le résultat.	car(cons(2 fact(10))) →car(2 fact(10)) →2 L'évaluation de 10! n'a jamais lieu puisqu'il n'est pas fait appel à ce paramètre.
Une traduction de ces expressions est la suivante : Quel-est-le-premier-élément-du-couple(2,10!) Un langage fonctionnel renverra directement 2 alors qu'un langage strict, nécessitera le passage par l'étape intermédiaire de traduction du premier appel : Quel-est-le-premier-élément-du-couple(2,3628800)	

Figure 31 : Evaluation stricte et paresseuse d'une même expression

Si l'on s'intéresse maintenant à la complexité temporelle de cette évaluation (Figure 31) on remarque que le calcul de celle-ci avec un langage strict est en O(n) (c'est à dire l'ordre de complexité du calcul de factorielle) alors que la complexité temporelle de cette même évaluation avec un langage paresseux est en O(1) (c'est à dire que la complexité temporelle

de l'évaluation de l'expression est totalement indépendante de la complexité temporelle du calcul de la factorielle).

3.1.1.2. Seconde propriété : manipulation de structures infinies possible

La possibilité de travailler avec des structures de données infinies peut être vue comme une conséquence de la propriété précédente, si l'on désire une liste infinie de zéro par exemple, il suffit dans un langage paresseux de définir cette liste comme ceci :

```
(define list0 (cons 0 list0))
```

Ce qui traduit directement le fait qu'une liste infinie de zéro équivaut à un zéro suivi d'une liste infinie de zéro (le deuxième argument étant effectivement une liste infinie de zéro et non un pointeur vers l'élément suivant)

Exemple 7 : Définition d'une liste infinie

Un langage non paresseux refusera cette définition puisqu'il cherchera à évaluer le second argument de *cons* dès la définition de list0 ce qui l'entraînera dans une boucle infinie. Un langage paresseux au contraire ne cherchera à évaluer ce second argument que lorsqu'il en aura réellement besoin, la définition sera donc acceptée et chaque appel à un élément quelconque de la liste provoquera une évaluation minimale qui renverra zéro systématiquement.

3.1.1.3. Source de paresse

Nous venons de le voir la paresse peut provenir du langage utilisé mais son principe peut aussi être étendu à un cadre plus général pour la création de structures de données (principalement arborescente et infinie) ainsi que pour la lecture des données (nous nous rapprochons alors de l'évaluation par nécessité des langages fonctionnels) uniquement en modifiant la manière de programmer. Un algorithme devant traiter un ensemble de données volumineux peut par exemple être écrit de deux manières (Figure 32) :

Cas n°1 : Algorithme classique	**Cas n°2 : Algorithme paresseux**
Début Lecture des données *Pour* chaque donnée Calcul(x) *fin Pour* *Fin* Avec : Calcul(x) *Début* Traitement(x) *Fin*	*Début* *Tant que* traitement non terminé Calcul(donnée_suiv) *Fin Tant que* *Fin* Avec : Calcul(donnée_suiv) *Début* x = Lecture(donnée_suiv) Traitement(x) *Fin*

Figure 32 : Comparaison algorithme statique – algorithme dynamique

Dans le premier cas de la Figure 32 l'ensemble des données est lu en une seule instruction placée en tête de traitement laissant au système d'exploitation le loisir de gérer comme il l'entend la lecture réelle des données avec le risque que la gestion effectuée par celui-ci ne corresponde pas à l'utilisation réelle des données telle qu'elle va être entreprise.

Dans le second cas au contraire, c'est le calcul qui dirige l'exécution et la lecture des données au moment précis où elles sont nécessaires. En fonction de l'utilisation réelle des données, la seconde manière d'opérer peut éviter une fastidieuse étape de réarrangement des données, elle permet de libérer les données au fur et à mesure qu'elles ne sont plus

nécessaires apportant donc la garantie d'une bonne localité et un besoin mémoire inférieur à condition que toutes les données ne soient pas nécessaires au même moment. La seconde méthode permet aussi d'éviter de lire des données qui ne seront finalement pas utilisées, ceci dépendant à nouveau de l'utilisation réelle des données par le calcul.

3.1.2. Paresse et distribution

La distribution d'un algorithme du premier cas (Figure 32) nécessitera une opération préalable de répartition des données. Une recopie totale du jeu de données est une possibilité simple à mettre en œuvre, elle est généralement efficace puisque les données sont toutes locales après la première opération de diffusion mais très coûteuse en espace mémoire et plutôt réservée aux algorithmes manipulant de petits jeux de données. Si l'ensemble des données nécessaires à chaque nœud en fonction du travail qui lui sera alloué est prévisible, une répartition plus fine des données pourra être effectuée, rappelons que cette prévisibilité n'existe pas dans le cadre des applications irrégulières.

La distribution d'un algorithme du second cas (Figure 32) ne nécessitera, au contraire, aucune opération explicite de répartition des données. Les données ne seront envoyées aux nœuds qu'au fur et à mesure des calculs et en fonction de ceux-ci. Cette méthode s'expose à coût supérieur de transmission des données (par rapport aux techniques du paragraphe précédent) puisqu'elle utilise généralement plusieurs communications, en cours de traitement, au lieu d'une transmission unique au début de celui-ci. Par contre, seules les données réellement nécessaires au traitement local de chaque nœud seront transmises amenant à une bonne répartition des données en fonction du travail. Si une certaine correspondance, même approximative, entre les tâches et les données utilisées par celles-ci peut-être exploitée lors de l'attribution des tâches, la répartition des données devrait en être améliorée ainsi que les performances. En effet, si l'on parvient à attribuer deux tâches utilisant des données communes à un même nœud de calcul, ces données ne transiteront par le réseau qu'une seule et unique fois (elles ne sont pas détruites à la fin du traitement de la première tâche les nécessitant), l'efficacité est donc améliorée : le deuxième accès est local, la répartition est améliorée : ces données ne seront peut-être pas dupliquées sur un autre nœud alors qu'elle l'aurait forcément été si les deux tâches n'avaient pas été attribuées au même nœud de calcul.

La paresse formerait donc un nouvel outil de distribution des données dont le principal intérêt serait d'éviter une recopie totale des données dans les situations où l'attribution des tâches aux différents nœuds n'est pas statiquement déterminé en prétraitement, ou encore dans les situations où le lien exact entre les données et les calculs n'est pas connu. Ses principaux atouts sont : une grande simplicité de codage (elle simplifie particulièrement l'opération de distribution de l'algorithme) et la production d'une répartition des données bien adaptée aux calculs réels.

3.2. La granularité adaptable

Nous l'avons vu précédemment une granularité adaptée est une granularité qui permet de réduire le nombre de messages au profit de leur taille. Mais d'autres contraintes importantes pour le bon déroulement de l'algorithme distribué peuvent venir modifier le choix de la granularité la plus adaptée, l'espace mémoire et l'équilibre de charge font partie de ceux-ci.

3.2.1. Généralités

Une base de travail peut être la classification suivante :

- Dans le cas des algorithmes réguliers ne nécessitant pas de volume de données supérieur à la mémoire locale de chaque nœud : la meilleure granularité est la plus épaisse : un découpage statique du problème en P sous-problèmes de même complexité pour P processeurs.

- Dans le cas des algorithmes réguliers manipulant des volumes de données supérieur à la capacité des mémoires locales des nœuds, la meilleure granularité est la plus épaisse permettant de ne travailler que sur des jeux de données ne provoquant pas de recours à la mémoire secondaire. Les données d'entrées, de sortie et éventuellement intermédiaires ainsi que le code du programme et le système opératoire doivent représenter un volume mémoire inférieur (ou égal) à la capacité mémoire de chaque nœud.

- Dans le cas des algorithmes irréguliers le choix de la granularité devient beaucoup plus complexe puisque la granularité la plus adaptée sera la plus épaisse permettant d'obtenir un bon équilibre de charge sans avoir recours à la mémoire secondaire. Cependant la répartition de la complexité au sein du problème à traiter n'est pas connue à priori, pas plus que l'utilisation de la mémoire. Une répartition dynamique des tâches va donc être mise en place, il faudra alors que le nombre de tâches soit beaucoup plus important que le nombre de processeurs (nécessaire à l'obtention d'un équilibre de charge correct) ce qui va dans le sens d'une granularité fine. Le choix d'une granularité adaptée est donc ici soumis à des contraintes contradictoires : la plus épaisse possible pour l'efficacité (réflexion issue du chapitre précédent) et la plus fine possible pour l'équilibre de charge et par conséquent l'efficacité aussi. Le choix de la granularité la plus adaptée est donc souvent un compromis entre ces deux paramètres.

3.2.2. Les algorithmes irréguliers

Les algorithmes irréguliers forment le troisième cas identifié dans le paragraphe précédent (3.2.1). Il conviendra donc de surveiller l'équilibre correct de la charge entre les nœuds de calcul tout en s'assurant que le pourcentage de communication reste faible.

3.2.2.1. Pourquoi il faut une granularité la plus fine possible

Assurer un bon équilibre de charge entre les différents nœuds correspond très concrètement à les faire terminer leur travail en même temps. La charge de travail que va engendrer une tâche n'est pas prévisible dans le cadre des algorithmes irréguliers et le seul moyen d'assurer l'équilibre de charge est de découper le travail en un nombre de tâches le plus grand possible par rapport au nombre de processeurs et de distribuer dynamiquement ce travail au fur et à mesure de l'avancement des calculs. Il va de soit que plus les tâches sont de petite taille moins elles sont susceptibles de provoquer une lourde charge de travail. Une justification un peu plus formelle de l'intérêt de sous-parties les plus petites possible peut être énoncé ainsi : **La différence de complexité de calcul entre deux tâches est strictement supérieure à la différence maximale de complexité entre n'importe quelle sous-partie de l'une et n'importe quelles sous-partie de l'autre**.

3.2.2.2. Pourquoi il faut une granularité la plus épaisse possible

Nous avons vu précédemment que l'efficacité d'un algorithme distribué était très fortement influencée par le ratio entre le nombre et la taille des messages utilisés par rapport à la complexité des calculs distants. La distribution des tâche étant effectuée dynamiquement, l'exécution distante d'une tâche nécessite au moins deux messages, le premier définissant la tâche à accomplir et le second indiquant la terminaison de cette dernière et la disponibilité du processeur pour une nouvelle tâche (ainsi que le résultat éventuellement). Donc plus les tâches de bases seront petites et donc nombreuses, moins leur complexité sera grande et plus le nombre de messages augmentera (certains d'entre eux verront en plus leur taille diminuer) ce qui correspond à une implémentation inefficace d'un algorithme distribué. En conséquence la granularité doit être la plus épaisse possible.

3.2.3. Solution proposée

Le paragraphe précédent indique que l'adaptation de la granularité est soumise à des contraintes opposées dans le cas d'algorithmes irréguliers. L'approche que nous proposons n'est pas un compromis qui nous amènerait à un résultat moyen au niveau de l'équilibre de charge et une utilisation du réseau juste correcte. Nous avons remarqué que la contrainte

sur l'utilisation de petits blocs n'était réellement présente qu'à la fin du calcul. L'utilisation de blocs d'abord de grandes tailles puis de taille de plus en plus réduite au fur et à mesure de l'avancement des calculs permet d'obtenir un équilibre de charge équivalent à celui d'une exécution avec de petits blocs durant toute la durée du calcul. L'avantage de la méthode de découpage récursif et dynamique des blocs est de nécessiter un nombre de messages beaucoup moins important que son concurrent statique (à blocs de petite taille).

Le découpage récursif dynamique présente un risque lié au choix d'une taille de blocs initiale trop importante qui pourrait faire aboutir à la situation suivante : un des blocs initiaux contient une charge de calcul plus importante que celle de tous les autres blocs réunis divisée par le nombre de nœuds chargés du reste de l'image (nombre de nœuds de calcul – 1). Il convient donc de ne pas être trop « gourmand » dans le choix de la taille de blocs initiale et de réduire celle-ci suffisamment rapidement.

Tomas Plachetka reprend ce principe et tente de formaliser les choix que je viens d'évoquer mais le choix de la taille de départ reste pragmatique [PLA02]. L'hypothèse de départ est qu'un coefficient K a été (arbitrairement) choisi tel que la charge de calcul engendrée par n'importe lequel des blocs de la taille la plus grande est au plus égale à la charge du reste de l'image divisée par K. Si cette présomption est vérifiée alors Tomas Plachetka estime que l'équilibre de la charge de calcul utilisant le découpage récursif dynamique de l'image sera optimal. Malheureusement aucune méthode absolue de choix de la taille de départ ou du coefficient K ne permet d'affirmer que le critère exigé sera vérifié puisque nous n'avons aucune connaissance à priori sur le poids des différentes tâches.

3.3. Le regroupement de messages

Le regroupement de messages n'est pas exactement un nouvel outil, il a pourtant sa place dans ce chapitre puisqu'il est un moyen efficace de réduction de coût des communications.

La mise en place d'une granularité adaptable devrait permettre d'obtenir un taux de communication faible, l'utilisation de techniques complémentaires, comme le regroupement de messages, sont des optimisations de moindre envergure mais devraient pouvoir amener un léger gain en performance ainsi qu'assurer une meilleure « scalabilité ».

Le regroupement de message est basé sur le principe de n'utiliser qu'une seule « enveloppe » de message pour transmettre des données qui ne sont pas forcément liées.

4. Choix de l'algorithme à distribuer

Le choix de l'algorithme séquentiel qui servira de base est un point clé de la démarche de distribution. La complexité algorithmique est souvent le critère de sélection retenu, les algorithmes ont pourtant d'autres aspects qui peuvent radicalement influer sur l'application distribuée qui en découlera. En effet, certains algorithmes vont réclamer plus de synchronisations, de communications que d'autres lors de leur distribution. Les algorithmes peuvent aussi posséder des mécanismes qui influenceront la gestion des données (c'est le cas des algorithmes paresseux) ou tout autre aspect de la future application distribuée.

Même du seul point de vue de la performance, le choix de l'algorithme séquentiel de moindre complexité n'est pas toujours préférable, deux raisons principales à cela : la plage d'utilisation de l'algorithme n'est pas adaptée ou la distribution de cet algorithme amène à des communications trop importantes.

4.1. Un calcul plus adapté de la complexité temporelle

Le calcul de la complexité temporelle d'un algorithme à distribuer (ou déjà distribué) ne prend habituellement pas en compte les communications, se concentrant plutôt sur les opérations de calcul. De plus, s'il est admis qu'une moindre complexité algorithmique ne présage d'une meilleure performance qu'à partir d'un certain seuil, celui-ci est souvent considéré facilement atteint (sauf dans des contextes spécifiques comme par exemple la création de circuits intégrés). Ce seuil peut pourtant se révéler important lors de la comparaison entre complexité des calculs et complexité des communications. Sans définir de nouvel outil formel de calcul de complexité, nous proposons ici de porter une attention sur ces deux points et donnons quelques pistes sur la manière de les prendre en compte.

4.1.1. Intégration de la complexité des communications

Une hypothèse fréquente (rappelée page 46, Équation 3) est que la distribution optimale d'un algorithme de complexité temporelle T(n) sur p processeurs doit amener à une complexité de T(n)/p. Celle-ci peut pourtant être remise en question si l'on considère que « l'outillage de distribution », sous entendu nul dans le terme « distribution optimale » précédent, n'est pas décorrélé du choix de l'algorithme de base.

La complexité temporelle effective d'un algorithme distribué est décomposable en complexité des calculs et complexité des communications. Le calcul analytique précis de la composition de ces deux valeurs est loin d'être trivial, une partie des communications

pouvant être recouverte par du calcul utile (Son ordre de grandeur est par contre le maximum des ordres de grandeur des complexités des calculs et des communications). Il semble donc raisonnable de conserver ces deux valeurs sans tenter de les agréger.

La comparaison de deux algorithmes s'effectuant alors selon deux critères, deux cas de figure se présentent :

- La complexité des calculs et celle des communications sont en faveur du même algorithme, le choix de ce dernier semble donc indiscutable.
- Un des algorithmes présente une meilleure complexité de calcul mais sa complexité de communication est moins bonne. Il convient alors de vérifier la plage d'utilisation du programme cible et de la mettre en balance avec le seuil de rentabilité de l'algorithme de plus faible complexité de calcul par rapport à son concurrent.

4.1.2. Seuil de rentabilité d'un algorithme de moindre complexité

Certes le fait que la complexité indique une meilleure efficacité uniquement à partir d'un certain seuil n'est pas une nouveauté en soit. Mais le seuil en question peut devenir suffisamment important pour qu'il faille y porter une attention particulière lors des comparaisons entre les complexités des calculs et des communications.

En effet le nombre d'opérations équivalentes (i.e. correspondant au même laps de temps) à une communication de cesse d'augmenter. Actuellement il est courant de rencontrer des équivalences d'environ 50 à 100 0000 opérations pour un message vide.

Si l'évolution technologique continue à creuser l'écart entre le coût des calculs et celui des communications, alors, le seuil de rentabilité d'un algorithme exigeant du point de vue des communications va continuer de s'élever (même si cet algorithme est efficace en ce qui concerne les calculs).

Le calcul de la complexité d'un algorithme à distribuer devrait donc prendre en compte les diverses possibilités de distribution que celui-ci offre du point de vue des méthodes de distribution et les complexités de communications qui en découleront. Le choix entre deux algorithmes séquentiels devant servir de base à une distribution pourra alors être différent de celui obtenu par l'utilisation unique de la complexité temporelle basée sur les calculs. Par exemple si l'algorithme de moindre complexité des calculs présente la plus importante complexité des communications. L'exemple de la FFT (dans le paragraphe suivant) illustre le cas ou le seuil de rentabilité de l'algorithme de moindre complexité (des calculs) est suffisamment important pour remettre en question le choix de celui-ci.

4.2. Cas concret : La Transformation de Fourier d'ordre deux

L'algorithme de transformation de Fourier est très fréquemment utilisé dans le monde du calcul scientifique et les optimisations de son algorithme sont nombreuses. L'article [MAR99] introduit un nouvel algorithme de moindre efficacité séquentielle mais qui semble mieux adapté au contexte d'utilisation choisi grâce à une réduction des communications inhérentes à la distribution de l'algorithme de base.

4.2.1. La transformée de Fourier séquentielle

4.2.1.1. Transformée d'ordre un

La transformée de Fourier est un outil provenant du traitement du signal, sa version discrète (DFT) est utilisée pour le traitement numérique. Pour un signal x(t) à N échantillons, la transformée de Fourier discrète (d'ordre un) est un vecteur y obtenu par la formule suivante :

$$y(k) = \frac{1}{N} \sum_{t=1}^{N} x(t).e^{-2\pi j \frac{kt}{N}}$$

Équation 4 : Transformée de Fourier Discrète d'ordre un

Une variante optimisée de l'algorithme de transformée de Fourier discrète est la transformation de Fourier rapide (ou FFT : Fast Fourier Transform). Cet algorithme célèbre a été inventé par Cooley et Tukey, ingénieurs dans le centre de recherche d'IBM au début des années 1960 [COO65]. Il a eu, du fait de son efficacité, un impact considérable sur le développement des applications en traitement numérique des signaux. Un calcul de transformée de Fourier discrète est un calcul de produit d'une matrice par un vecteur. Il nécessite donc T^2 multiplications/additions de nombres complexes. L'algorithme de transformée de Fourier rapide sépare le calcul des éléments pairs et impairs. Un calcul sur N échantillons sera donc la combinaison du calcul de deux sous-ensembles de N/2 échantillons (les pairs d'un côté les impairs de l'autres). Le calcul de ces sous-ensembles comptant un nombre d'opérations de l'ordre de N, la complexité résultante est alors en nlog(n).

4.2.1.2. Transformée d'ordre deux

La transformée de Fourier discrète d'ordre deux, dont le domaine d'application est plutôt le traitement d'images, s'applique sur une matrice d'ordre N. Son résultat est une matrice Y obtenue grâce à la formule suivante :

$$y(p,q) = \sum_{r=1}^{N} \sum_{s=1}^{N} x(r,s).w_N^{pr}.w_N^{qs} \text{ avec } W_N^k = e^{-2\pi k}$$

Équation 5 : Transformée de Fourier Discrète d'ordre deux

Le calcul d'une transformée de Fourier discrète d'ordre deux peut être vu comme l'enchaînement de deux transformée de Fourier discrète d'ordre un. Observé d'un point de vue matriciel, une première multiplication de matrice est effectuée, puis une transposition de la matrice résultat suivi d'une seconde multiplication de matrice. C'est en particulier la nécessité de la transposition de la matrice intermédiaire qui est remise en question dans l'article [MAR99]

4.2.2. La transformée de Fourier parallèle

L'approche classique de parallélisation de l'algorithme de transformée de Fourier discrète d'ordre deux peut être schématisée de la manière suivante (Figure 33) :

Algorithme parallèle de 2D DFT

1. Emission vers les nœuds de calcul des données de X (par lignes ou blocs de lignes)
2. Effectuer une transformée de Fourier discrète (rapide) d'ordre un sur chaque ligne
3. Regrouper l'ensemble des résultats au sein de chaque nœud de calcul afin de pouvoir transposer la matrice
4. Effectuer une transformée de Fourier discrète (rapide) d'ordre un sur chaque colonne
5. Regrouper les résultats finaux

Figure 33 : Parallélisation classique de l'algorithme de transformée de Fourier d'ordre deux

Notons l'utilisation du remplacement de communication par des calcul évoqué lors de la présentation des outils classiques de distribution (Chapitre I :2). En effet la matrice W étant calculable elle n'est pas transmise mais calculée localement ce qui amène à de meilleures performances. Il reste à faire le choix d'un calcul préalable de l'ensemble de cette matrice ou du calcul de ses éléments un à un au moment où ils sont nécessaires. Ce choix dépend de plusieurs critères :

- Les éléments étant utilisés plusieurs fois, un calcul préalable évitera des calculs redondants et pourra favoriser l'exploitation de la mémoire cache ceci à condition que l'espace mémoire soit suffisant à ce type de traitement.
- Un calcul à la volée ne posera aucun problème de stockage de l'information

Le choix entre ces deux options dépend certainement de la taille des matrices traitées, lors de nos tests, le calcul préalable de l'ensemble de la matrice W s'est révélé le plus efficace.

L'étape 3 (Figure 33) demande un échange important d'information réduisant la performance de cet algorithme parallèle. Cependant l'utilisation de l'algorithme rapide de transformée de Fourier discrète, algorithme le plus efficace, impose une transposition de la matrice (elle est traitée selon les lignes dans un premier temps et selon les colonnes dans un deuxième).

Une approche plus matricielle de la transformée de Fourier peut nous amener à la formulation (compacte) suivante :

$$y_p^R = (w_{N_p}^R \times X) \times W_N$$

avec y_p^R et $w_{N_p}^R$ respectivement les p-ième lignes de Y et de W_N

Équation 6 : Expression matricielle compacte d'une 2D DFT

Si l'on s'attache à paralléliser cette expression plutôt que l'expression classique plusieurs conséquences vont apparaître :

1. L'utilisation de l'algorithme rapide de transformée discrète de Fourier n'est pas applicable lors de la première transformation (il le redevient pour la multiplication externe).

2. L'intégralité de la matrice X est nécessaire à chaque nœud de calcul (au lieu d'une ou plusieurs lignes)

3. Il n'y a plus d'échange d'information pendant le calcul (seulement au début pour distribuer les données et à la fin pour récupérer les résultats)

Parmi ces conséquences, la troisième est clairement un avantage à l'utilisation de la nouvelle approche alors que la première et la seconde sont plutôt des inconvénients.

La deuxième conséquence, la nécessité de distribuer l'intégralité de X, permet d'utiliser une diffusion au lieu d'un ensemble de messages (pour les architectures disposant de cette facilité). De plus les nœuds peuvent commencer leur travail dès réception des premières valeurs sans attendre la réception de la totalité de la matrice. Cette conséquence de l'utilisation est donc bien adaptée à l'architecture réseau de machines et ne doit donc plus être considérée comme un inconvénient de cette approche.

Reste la non utilisation de l'algorithme rapide pour la première partie de l'algorithme. Il est intéressant de noter que, dans le cas d'une transformée quelconque (même une transformée monodimensionnelle peut être ramenée à une transformée bidimensionnelle), le problème ne se pose pas. Il est spécifique à la transformée de Fourier discrète pour laquelle un algorithme rapide exploitant les particularités de la matrice W existe. Francescomaria

Marino explique qu'il est tout de même possible de faire certaines optimisations et fournit les complexités suivantes :

$$O(N^4 \log^2(N))$$

Équation 7 : Complexité minimale de la 2D FFT pour une matrice NxN

$$O(N^4 \log^3(N))$$

Équation 8 : Complexité du nouvel algorithme de 2D DFT pour une matrice NxN

Soit un écart de l'ordre de log(N) entre la nouvelle approche et l'utilisation exclusive de l'algorithme FFT.

L'article [MAR99] conclut en affirmant la supériorité des performances de l'algorithme sans communication sur l'algorithme à base de FFT, bien que ce dernier affiche une complexité algorithmique inférieure. L'écart sur la complexité algorithmique serait amplement compensé par le gain sur les communications.

4.2.3. Conclusion

Les résultats annoncés sont sans aucun doute dépendant de l'architecture cible ainsi que de la tailles des données utilisées. Les expérimentations effectuées par nos soins n'ont pas affiché de résultats aussi tranchés et demanderaient à être approfondies. Toutefois une estimation même très grossière du coût des communications nous amène à ceci :

- Algorithme à base de FFT : Pour P processeurs,
  - Au départ : P messages (ligne ou bloc de lignes),
  - Au niveau intermédiaire : la méthode la moins coûteuse du point de vue de la communication c'est à dire l'envoi des résultats au serveur, la transposition de la matrice par celui-ci et enfin l'envoi des nouvelles lignes aux esclaves amènent à 2P messages de la même taille que les précédents
  - A la fin : P messages (toujours de la même taille) pour récupérer l'ensemble des données et construire la matrice résultat.

- Algorithme sans communication intermédiaire : Pour P processeurs,
    - Au départ : au pire, c'est à dire sans distribution possible, P messages (de taille plus importante, mais nous avons vu que la taille d'un message avait un impact moins négatif sur les performances que la quantité de messages et que le travail pouvait commencer bien avant la réception complète des données).
    - Pas de messages intermédiaires
    - A la fin : P messages (de même taille que ceux de l'algorithme à base de FFT).

La complexité des communications en fonction du nombre de processeurs est donc en faveur du nouvel algorithme (2P contre 4P) tout en restant du même ordre de grandeur. Cette complexité varie lentement par rapport à N (seule la taille des messages évolue).

Cette analyse est trop grossière pour réellement prédire le comportement exact des deux algorithmes lorsque N est grand (en particulier les coûts des latences et du débit devraient être séparés pour obtenir une estimation de la complexité dépendant à la fois de P et de N) mais même en s'en tenant à celle-ci notons que l'algorithme de plus faible complexité (à base de FFT) ne devance le nouvel algorithme que de log(N) alors que la différence de coût de communication est de 2P messages. En prenant une valeur réaliste du coût d'un message estimé en nombre d'opérations, soit 100 000, on obtient une différence de performance de 200 000 x P opérations en faveur du nouvel algorithme. L'algorithme de meilleure complexité ne pourra espérer compenser cet écart (en considérant de plus que le coût des messages soit constant vis à vis de N) que pour une valeur de N supérieure à $2^{200\,000 \times P}$ soit plus de $10^{20\,000 \times P}$ ce qui correspond à une taille de matrice absolument faramineuse et qui poserait des problèmes bien différents lors de son calcul.

Il semble donc confirmer que l'algorithme réduisant les communications au prix d'une moins bonne complexité algorithmique soit finalement plus performant que son concurrent à base de FFT. Ce résultat est très important et devrait remettre en question la confiance aveugle généralement accordée à la complexité algorithmique pour estimer la performance de telle ou telle autre solution (Dans le cadre des réseaux de machines).

5. Démarche de distribution

En s'appuyant sur tout ce qui a été évoqué dans les paragraphes et chapitres précédents, il semble qu'une certaine démarche de développement d'une application distribuée efficace puisse être ébauchée. Les différentes étapes sont présentées ici par ordre chronologique

mais ce même ordre reflète aussi l'importance de chaque étape sur l'efficacité finale de l'application distribuée de la plus grande à la plus faible.

0. L'environnement

Une étape, que l'on peut qualifier de préliminaire car elle n'est pas directement incluse dans le processus de distribution d'une application, est de s'assurer que l'environnement est correctement exploité :

- Le système est « à niveau » et il utilise des protocoles de communication rapides s'il en existe de disponibles pour le matériel utilisé.
- Les pilotes de périphériques sont adaptés au matériel.
- Le compilateur est récent et, si possible, reconnaît l'architecture cible (ce dernier point indique généralement des possibilités d'optimisations étendues).

1. Le choix de l'algorithme séquentiel

La première étape de la démarche de distribution proposée est le choix de l'algorithme séquentiel en fonction de ses diverses caractéristiques. En particulier :

- La portion de l'algorithme parallélisable comparée à celle qui restera séquentielle.
- Les synchronisations et communications qui seront imposées par la distribution de l'algorithme choisi.
- Les éventuels mécanismes qui modifieront le processus de distribution ou le comportement de l'algorithme distribué.
- La complexité algorithmique de l'algorithme de départ.

La stratégie de distribution ainsi que la difficulté de mise en œuvre de celle-ci dépend grandement de l'algorithme séquentiel choisi.

Notons aussi que c'est souvent de ce choix que va découler l'ordre de grandeur du taux de communication, les étapes qui vont suivrent n'auront généralement pas un impact suffisant pour « rattraper » un mauvais choix de l'algorithme séquentiel de départ.

2. Le choix du mode de distribution

Certains algorithmes offrent plusieurs possibilités de distribution différentes, dans ce cas, il faudra préférer le mode de distribution le moins exigeant du point de vue des communications (en particulier sur le nombre de petits messages).

Notons particulièrement que le mode de distribution selon les données (data driven) fréquemment employé sur les supercalculateurs est souvent délicat à adapter à une architecture de type réseau de machines. Ce mode de distribution fonctionne sur le principe d'une répartition préalable du jeu de données sur les nœuds de calcul, puis d'une assignation des tâches sur les nœuds disposant les données nécessaires, suivie d'éventuelles migrations des tâches au fur et à mesure de l'évolution des ressources employées. Ces migrations produisent un taux de communication mal adapté aux interconnexions habituellement utilisées dans les réseaux de machines, phénomène souvent amplifié par l'utilisation d'une topologie en bus (la plus fréquente sur les réseaux de machines) à la place de réseaux dont la topologie permet des communications point à point simultanées entre des couples de nœuds disjoints.

3. Le regroupement de message

Nous avons présenté cet outil dans le paragraphe 3.3, il peut offrir un impact significatif, assez directement parfois ou au prix d'un agencement différent des communications dans certains cas plus complexes.

4. Le mode de communication

Dans le mode de communication, je place en particulier :

- Les communications asynchrones : utilisées à bon escient, elles peuvent amener des gains non négligeables (rappelons qu'utilisées maladroitement elle peuvent aussi engendrer des coûts plus importants).

- Le travail sur l'aspect des informations à transmettre, comme la compression des données.

Dans ce chapitre une démarche de création d'une application distribuée efficace a été proposée et certains nouveaux outils ont été introduits.

Le chapitre suivant présente un cas concret d'utilisation de cette démarche et de ces outils dans le but de produire une application distribuée efficace de lancer de rayon. Le lancer de rayon est un algorithme irrégulier, réputé difficile à distribuer efficacement. Il nous permettra de juger de l'efficacité réelle des outils introduits dans ce chapitre et de la validité de la démarche choisie.

Chapitre V : Lancer de rayon

Sommaire

1. PRESENTATION DE L'ETUDE DE CAS 103
2. PREMIERE ETAPE : CHOIX DE L'ALGORITHME SEQUENTIEL 103
2.1. Principes de base de l'algorithme de lancer de rayon 103
2.2. Optimisations classiques de l'algorithme précédent 107
2.3. Algorithme paresseux et justification de ce choix 109
3. SECONDE ETAPE : CHOIX DU MODE DE DISTRIBUTION 115
3.1. Deux caractéristiques essentielles du lancer de rayon 115
3.2. Différentes manières de distribuer le lancer de rayons 115
3.3. Méthode choisie et raisons de ce choix 117
4. RESULTATS ET MESURES 117
4.1. Description des indicateurs utilisés 117
4.2. Mesures commentées 121
5. TESTS ET OPTIMISATIONS 136
5.1. Influence de la taille des blocs 136
5.2. Troisième étape : Le regroupement de messages 141
5.3. Quatrième étape : Le mode de communication 142
6. CONCLUSION ET COMPARAISON AVEC DES RESULTATS EXTERIEURS 143

L'étude d'un cas concret de distribution d'une application irrégulière sur une machine de type réseau de machines permettra de valider l'analyse théorique effectuée dans la première partie de ce document. La mise en application aura l'avantage de rapidement montrer les limites des méthodes classiques du point de vue de leur efficacité réelle et de focaliser notre attention sur les points effectivement importants. Les étapes correspondant à celles décrites dans la démarche de distribution (Chapitre IV : 5) seront repérées suivant leur rang (de la première à la quatrième)

1. Présentation de l'étude de cas

Une application de lancer de rayon a été choisie comme cas concret de distribution d'application irrégulière. C'est un grand classique des applications irrégulières et de leur distribution ce qui permet d'avoir accès à une bibliographie importante.

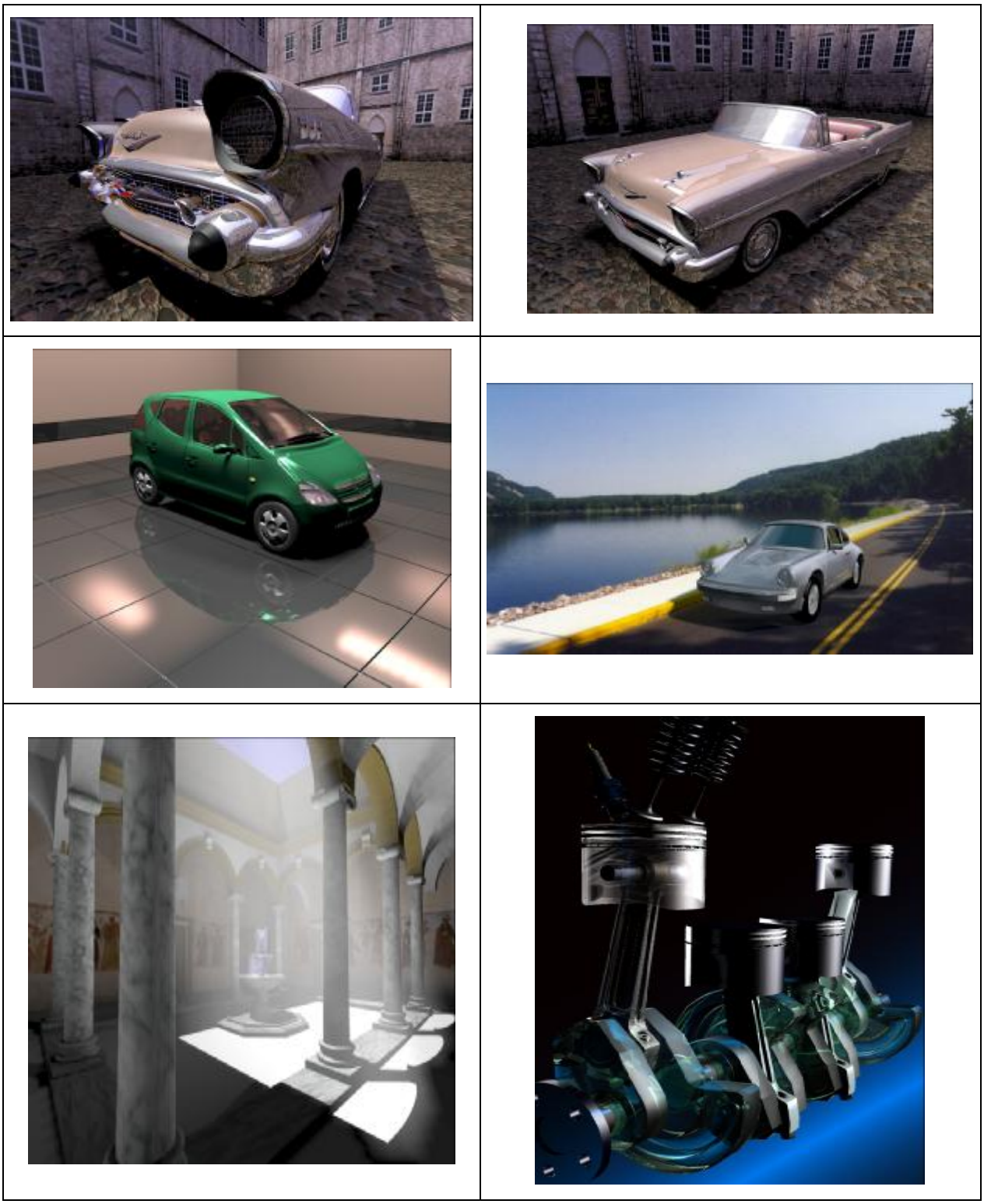
Un algorithme paresseux de lancer de rayon, développé à l'ONERA, fait l'objet de la thèse de Sébastien Bermes [BER98]. Cet algorithme formera une base solide de travail et, le développement de l'aspect lancer de rayon proprement dit étant déjà validé, il sera possible de se concentrer davantage sur la distribution elle-même.

2. Première étape : Choix de l'algorithme séquentiel

Je ne vais pas présenter en détail l'algorithme de lancer de rayon, ce n'est pas le sujet de ce document, mais en montrer les principes de bases ainsi que les principaux points faibles de l'algorithme direct et les solutions qui y sont apportées quand elles existent.

2.1. Principes de base de l'algorithme de lancer de rayon

Le lancer de rayon est une méthode réaliste de simulation d'onde pouvant s'appliquer à toute étude concernant des ondes dont la fréquence est suffisamment élevée pour que leur trajectoire puisse être assimilée à une droite. Bien que cette méthode, générale, soit utilisée dans des domaines aussi variés que la simulation sismique, l'électromagnétisme [DEP98] etc. je ne vais l'aborder que dans le cadre de l'imagerie de synthèse puisque c'est celui dans lequel je l'ai découverte et explorée.



Exemple 8 : Exemples d'images produites par lancer de rayons

Le lancer de rayon est une méthode réaliste de synthèse d'image basée sur la simulation des ondes lumineuses et dont le but est de reproduire le plus fidèlement possible la vue qu'aurait un œil placé en un endroit précis d'une scène définie en 3 dimensions réelles [GLA89, SCH97] (Exemple 8). Les données de départ du lancer de rayon sont donc la position de l'œil, la direction dans laquelle il regarde ainsi que son champ de vision et la scène 3D. La scène est une liste d'objets dont la position est donnée en coordonnées réelles, la forme est donnée soit sous forme équationnelle soit de manière universelle dans le cas d'objets discrétisés selon une forme simple (généralement un triangle), objets dont la couleur est exprimée selon les trois couleurs primaires et enfin objets dont on donne un certain nombre de caractéristiques supplémentaires comme la transparence, la diffusion, la réflexivité, l'absorption... on adjoint à cette liste une énumération des sources lumineuses elles aussi définies par leurs coordonnées dans l'espace ainsi que leur couleur, leur type (lumière directionnelle ou omnidirectionnelle par exemple) et quelques caractéristiques supplémentaires comme l'intensité.

A partir de ces données, la vision qu'aurait l'œil de la scène définie va être construite, pour cela chaque rayon de lumière reçu par celui-ci va être exploré. Les rayons qui seront étudiés seront ceux qui partent de l'œil vers la scène et non l'ensemble des rayons émis par l'ensemble des sources lumineuses ce qui nous amènerait à beaucoup de calculs inutiles (Figure 34).

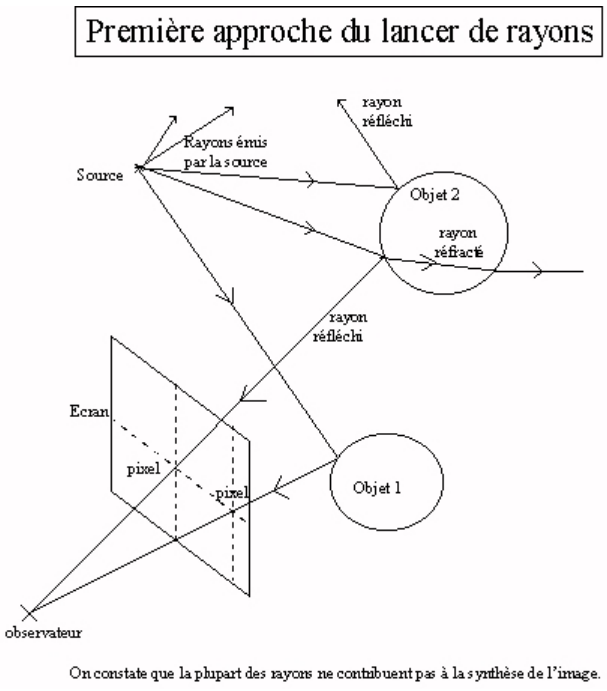


Figure 34 : Principe intuitif du lancer de rayon

L'ensemble des rayons est donc défini par la position de l'œil, qui leur sert d'origine, et le champ de vision généralement défini par un écran de taille et de résolution données. Les rayons primaires seront dans ce cas l'ensemble des demi-droites d'origine l'œil et passant par un point de l'écran. Chacun de ces rayons est « lancé » vers la scène, c'est à dire que l'on étudie le point d'intersection entre ce rayon et l'ensemble des objets de la scène. S'il y a intersection avec plusieurs objets on ne conserve que le plus proche du point d'origine et on applique en ce point les lois de Snell-Descarte afin de déterminer les différents rayons émanant de l'impact entre le rayon de départ et la scène. Ces nouveaux rayons sont le résultat de la réflexion, la réfraction, la diffraction du rayon initial sur le premier objet qu'il a rencontré, chacun des rayons engendrés va subir récursivement le même traitement que son « ancêtre » (Figure 35).

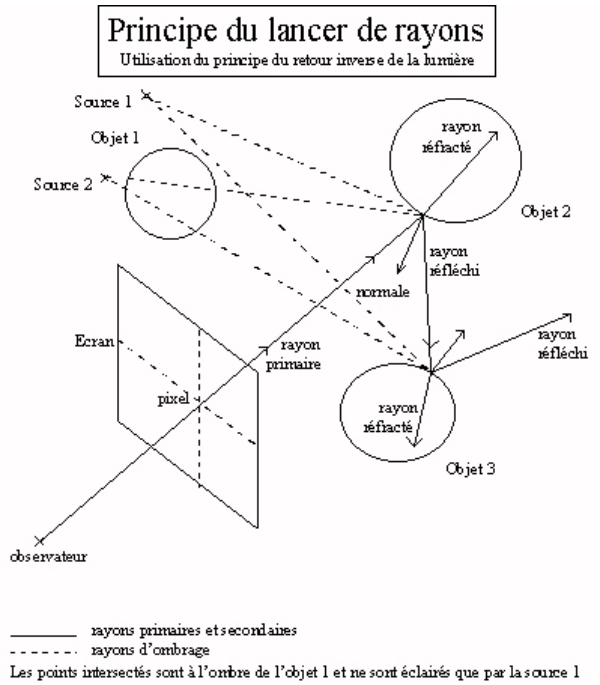


Figure 35 : Principe retenu du lancer de rayon

Ce processus est virtuellement sans fin si l'on ne rencontre pas de sources lumineuses, mais comme il est tenu compte de l'atténuation due à la distance parcourue ainsi qu'aux diverses péripéties du rayon on arrête le processus de calcul lorsque l'impact qu'aurait la découverte d'une source lumineuse devient négligeable (par exemple lorsque aucune nuance de couleur ne permet de distinguer cette éventuelle découverte d'une autre). Lorsque la récursion est terminée les caractéristiques précises du rayon qui sera reçu par l'œil sont connus et il ne reste plus qu'à les reporter sur l'écran. Nous venons de calculer le premier point de l'image, ce traitement devra être effectué pour tous les points de celle-ci.

L'algorithme de lancer de rayon direct répète donc un processus identique et présentant une charge de calcul importante sur un certain nombre de rayons indépendants, il semble en cela un candidat parfait à une exécution distribuée. Malheureusement il présente déjà une première forme d'irrégularité, en effet, seuls les rayons primaires sont connus dès le départ et chacun d'entre eux engendre un nombre de rayons secondaires différents en fonction de l'objet qu'il va rencontrer et ces rayons secondaires font de même et ainsi de suite. Il est donc tout à fait envisageable que le calcul d'un rayon primaire soit court (aucun objet rencontré) ou beaucoup plus long (nombreux rayons secondaires créés) cela ne se révélant que lors du calcul effectif du rayon. Cette irrégularité de la charge de calcul place dès maintenant le lancer de rayon dans les algorithmes dont la distribution est réputée délicate : les algorithmes irréguliers.

2.2. Optimisations classiques de l'algorithme précédent

La charge de calcul nécessaire au rendu d'une scène de complexité moyenne (quelques milliers de surfaces) par l'algorithme direct, tel qu'il a été présenté dans le paragraphe précédent, est très importante. Cette charge de calcul réside principalement dans les calculs d'intersection rayon-objet de l'algorithme. En effet, trouver le premier point d'intersection entre un rayon et la scène implique de tester **tous les objets de la scène** pour déterminer celui qui interceptera réellement le rayon (autrement dit le premier objet « sur le chemin »). Ce calcul relativement lourd sera répété pour **tous les rayons traités, primaires comme secondaires.**

Un moyen classique pour réduire cette charge de calcul est un découpage de l'espace contenant la scène en sous-espaces nommés voxels, le calcul d'intersection ne se faisant alors qu'avec les objets contenus dans les voxels traversés. Le calcul d'intersection utilisé pour ce nouvel algorithme est itératif, chaque itération étant constituée de deux phases : la première détermine le prochain voxel traversé, la seconde détermine s'il y a intersection entre le rayon étudié et les objets contenus par le voxel. Si aucune intersection n'est trouvée le calcul passe à l'itération suivante [GLA84].

Le découpage de l'espace contenant la scène peut être effectué par une grille régulière ou bien par une structure hiérarchique. L'avantage de la grille régulière réside principalement dans la première phase de l'algorithme, la détermination des voxels traversés est extrêmement rapide puisqu'elle est entièrement analytique. Une grille régulière présente toutefois l'inconvénient de ne pouvoir être adaptée à l'ensemble de la scène, certaines partie de la scène seront étudiées de manière trop précise tandis que d'autres contiendront trop d'objets par voxel rendant le calcul d'intersection moins efficace (Figure 36). Une structure hiérarchique est plus à même de s'adapter à l'hétérogénéité de la complexité à l'intérieur de la scène mais elle détruit la notion de localité et rend le calcul du prochain voxel traversé moins efficace (Figure 37).

Deux types de voxélisation

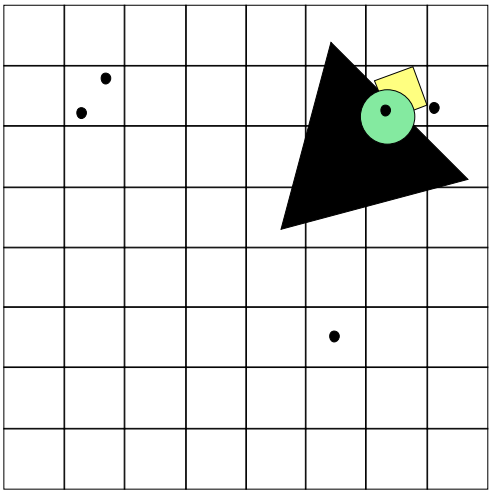


Figure 36 : voxélisation régulière (2D)

Représentation en deux dimensions d'une voxélisation régulière. On perçoit clairement l'inadéquation du découpage régulier à la complexité hétérogène de la scène.

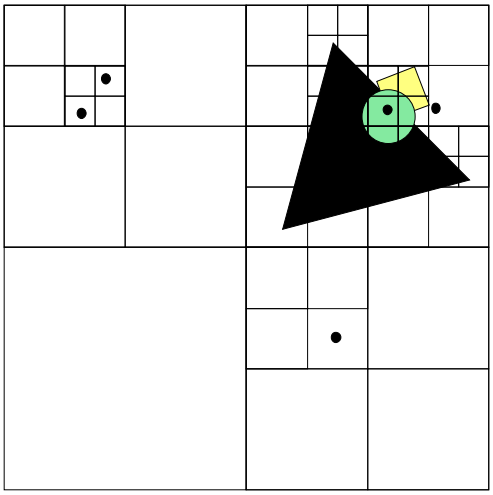


Figure 37 : voxélisation arborescente (2D)

Représentation en 2 dimensions montrant l'adaptation de la structure arborescente de voxels à la complexité hétérogène de la même scène que précédemment.

Figure 38 : Comparaison des deux types de voxélisation

Les exigences des utilisateurs du calcul ont été déterminantes dans le choix entre ces deux méthodes de découpage puisque le traitement à une précision suffisante d'une scène complexe n'est pas possible à l'aide de grilles régulières. La principale raison étant la taille mémoire qui serait nécessaire au traitement selon cette méthode. Le choix s'est alors porté sur les structures hiérarchiques ou bien sur des structures hybrides. La lenteur du calcul du prochain voxel rencontré a donné lieu à des études consacrées à l'écriture d'algorithme de traversée plus efficaces que je ne détaillerai pas ici mais dont un exemple peut être lu dans [AMA87].

La structure hiérarchique la plus fréquemment utilisée dans le contexte du lancer de rayons est l'arbre octal. Celui-ci est un arbre dont la racine correspond au plus petit volume englobant la totalité de la scène. Chaque nœud est décomposé en huit fils correspondant au découpage régulier du volume correspondant au nœud en deux parties égales selon chaque direction. Les feuilles correspondent à des volumes contenant un nombre de surfaces

inférieur à un seuil donné et à l'intérieur duquel on considère qu'un calcul analytique du premier objet traversé est efficace.

L'arbre octal, chargé de contenir le modèle selon une subdivision spatiale adaptée, représente un volume de données non négligeable si l'on désire rendre le calcul efficace. En effet, certaines informations concernant le voxel sont stockées afin d'accélérer les calculs à venir. La construction de cette structure représente un coût non négligeable en temps d'exécution. Mais il est communément admis que cela soit le prix à payer pour obtenir un traitement efficace. Nous noterons aussi que l'arbre octal est une structure de données irrégulière et que l'algorithme de lancer de rayons optimisé verra cet aspect de l'irrégularité s'ajouter à celui observé sur l'algorithme direct.

2.3. Algorithme paresseux et justification de ce choix

Le paragraphe 2.2 nous a montré l'intérêt d'utiliser un arbre octal pour optimiser le calcul du lancer de rayon. Malheureusement construire l'arbre octal d'une scène représente une opération non négligeable en temps CPU et produit un volume de données important (Figure 39). Il semble donc raisonnable de chercher à améliorer cet aspect de l'algorithme de lancer de rayons optimisé du paragraphe précédent.

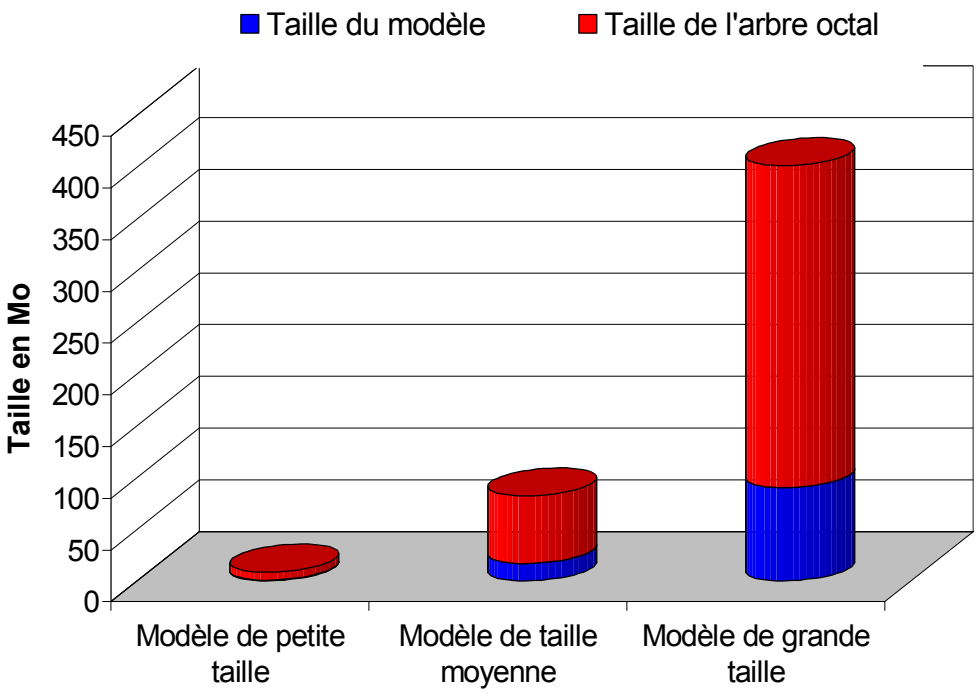


Figure 39 : Comparaison entre la taille du modèle et celle de son arbre octal

Nous avons vu, toujours dans le paragraphe 2.2, que l'utilisation d'une structure hiérarchique permet d'adapter le grain de discrétisation aux différentes zones de la scène selon un critère de complexité (complexité estimée à travers le nombre de surfaces contenues dans celles-ci).

Un second critère susceptible d'amener un gain de performance notable est la pertinence des informations contenues dans chaque zone de la scène. En effet, le principe du lancer de rayon est que le premier objet touché « arrête » le rayon (pour en produire de nouveaux). On peut dès lors imaginer que des zones de la scène soient « cachées » par un objet et ne soient donc jamais atteintes par aucun rayon. Le développement de l'arbre octal pour ces zones « cachées » de la scène est par conséquent inutile au calcul de l'image (en rouge sur la Figure 40).

Deux méthodes de construction de l'arbre octal

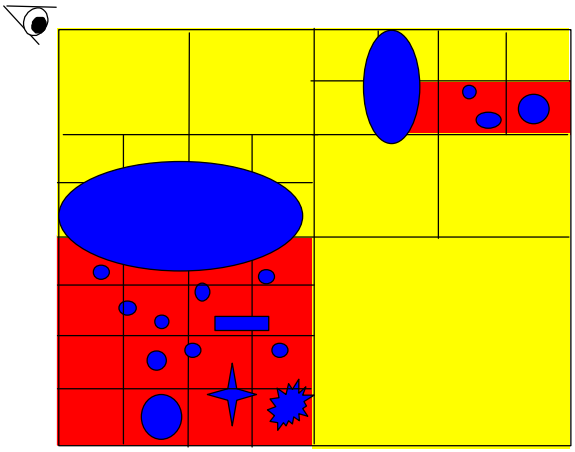


Figure 40 : Arbre octal statique

Le maillage arborescent est détaillé dans certaines zones (en rouge) contenant de nombreux objets mais que l'œil ne distinguera pas, autrement dit qu'aucun rayon ne viendra « toucher », et qui se révéleront donc inutiles au calcul.

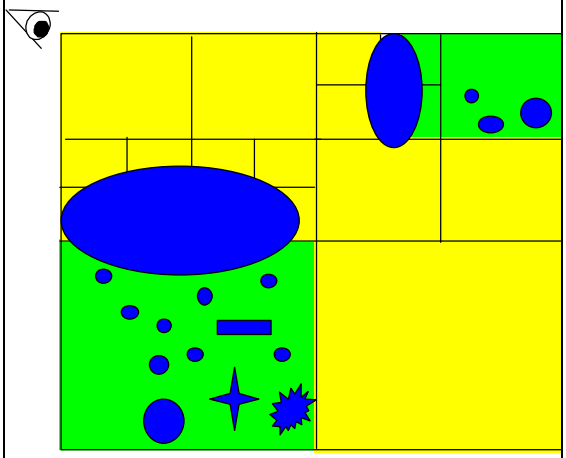


Figure 41 : Arbre octal paresseux

Le critère de pertinence est pris en compte et l'on peut remarquer que grâce à lui certaines zones (en vert) ne sont pas développées bien qu'elles contiennent de nombreux objets. Elles n'ont été atteintes par aucun rayon et un maillage plus fin n'a donc pas été nécessaire.

Figure 42 : Deux méthodes de construction de l'arbre octal

Malheureusement le critère de pertinence n'est pas évaluable à priori puisque le trajet réel des rayons n'est connu que lors du calcul. La construction classique de l'arbre octal ne tient donc compte que du premier critère et construit par conséquent un arbre octal complet, développant ainsi certaines branches qui s'avéreront inutiles au rendu.

L'alternative proposée dans [BER98] est l'utilisation d'arbres octaux paresseux, l'arbre octal paresseux est un arbre potentiellement infini et dans lequel les voxels passent par trois états différents :

- Vide : le voxel ne contient aucune surface
- Nœud : le voxel n'est pas vide, ses huit fils sont déjà construits
- Feuille : le voxel n'est pas vide, ses fils ne sont pas construits

Au début de la simulation, i.e. quand le premier rayon est lancé, l'arbre octal est réduit à un unique voxel feuille : l'évaluation paresseuse va autoriser la transformation de ce voxel feuille en un voxel nœud : ce processus se nomme l'évaluation du voxel ; il est effectué dynamiquement au fur et à mesure de la simulation. Chaque fois qu'un rayon atteint un voxel, il faut décider si son contenu peut être traité de manière analytique ou si l'on continue le processus d'évaluation du voxel. La limite fixée pour arrêter le processus d'évaluation d'un voxel est le nombre de surfaces contenues dans celui-ci, ou l'angle avec lequel le rayon atteint le voxel en question. Si cette condition limite n'est pas atteinte le voxel est évalué et la démarche est réitérée sur le voxel fils atteint par le rayon.

Contrairement aux méthodes conventionnelles utilisant des arbres octaux statiques, tous les voxels construits ont été atteints par au moins un rayon et aucun voxel inutile n'a été construit (Figure 41). Ceci peut aboutir à un gain sensible en terme de consommation mémoire si le modèle contient une grande proportion de parties cachées, en effet ces parties n'étant atteintes par aucun rayon, elles ne seront pas construites par la solution paresseuse alors qu'elles l'auraient été par une solution statique (Figure 43).

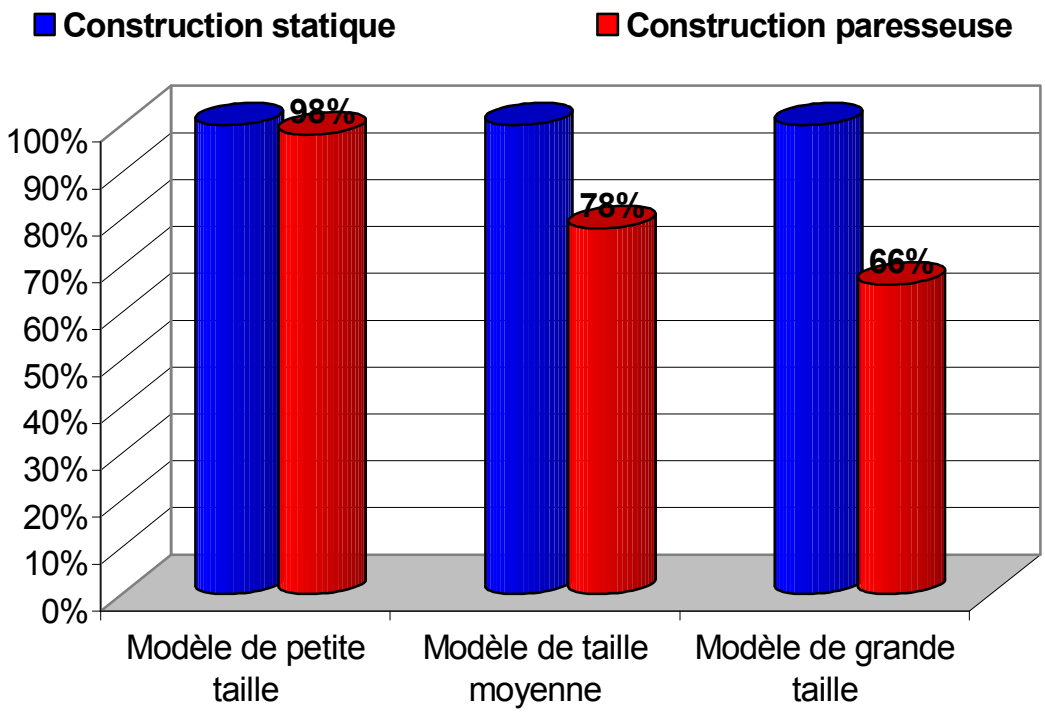


Figure 43 : Comparaison arbre octal statique - arbre octal paresseux

Cet algorithme (Figure 44) a donc comme propriété que seule la partie nécessaire de l'arbre octal est effectivement construite ce qui offre un gain sur le temps de construction de la structure ainsi que sur l'espace mémoire requis pour stocker celle-ci. De plus, une structure de donnée moins volumineuse amène souvent à une meilleure exploitation de la hiérarchie mémoire de la machine cible.

Algorithme paresseux de lancer de rayon

L'algorithme principal de calcul est le suivant :

```
Propage(rayons)
  Pour chaque rayon
    Eval(rayon, nœud_racine)          // détermine le premier point d'intersection avec un objet
    Appliquer les lois de Snell Descartes          // détermine les rayons secondaires
    Propage(rayons_secondaires)                    // appel récursif
  Fin Pour
Fin Propage.
```

L'algorithme de la fonction paresseuse Eval étant :

```
Eval(rayon, nœud)
  Si les conditions limites sont satisfaites par nœud
    résultat ← Calcule (rayon, liste_de_surface_du_noeud)        // Calcul analytique
    Si aucune intersection trouvée
      nœud_suivant ← Détermine prochain nœud traversé
      Si pas de nœud suivant                                     // le rayon sort de la scène
        Renvoie Null                  // aucune intersection entre le rayon et la scène
      Sinon
        Renvoie(Eval (rayon, nœud_suivant)) // Appel récursif pour chercher une intersection
                                            // dans le prochain nœud traversé par le rayon
      Fin si
    Sinon
      Renvoie(résultat)        // La première intersection entre le rayon et la scène a été trouvée
    Fin si
  Sinon
    Creer_fils(nœud)                                  // Création des huit fils du nœud
    premier_fils ← Détermine premier fils touché par le rayon
    Renvoie(Eval (rayon, premier_fils))                          // Appel récursif
  Fin si
Fin Eval.
```

Figure 44 : Algorithme paresseux de lancer de rayon

L'arbre octal paresseux présente plusieurs caractéristiques : un nœud fils n'est évalué que s'il contient des données qui sont nécessaires à la poursuite du calcul ; d'autre part, dès qu'un nœud est évalué, il est stocké définitivement dans l'arbre octal et donc, pour de nouveaux traitements nécessitant ce nœud, il ne sera pas recalculé : la cohérence spatiale des traitements est ainsi exploitée.

Bien qu'il soit habituel de choisir le meilleur algorithme séquentiel (meilleur étant ici compris comme le plus rapide) lors de la sélection de l'algorithme à paralléliser (voir Chapitre IV : 4), le choix fait ici n'est pas uniquement lié à la performance mais aussi à la caractéristique paresseuse de l'algorithme élu. Nous venons de le voir : seules les données nécessaires au calcul sont construites par cet algorithme. Par conséquent, si chaque nœud de la machine cible n'effectue qu'une partie du calcul, nous espérons qu'il n'aura pas systématiquement besoin de construire l'intégralité de l'arbre octal mais que ses besoins se limiteront à une partie de celui-ci (voir Chapitre IV : 3.1).

3. Seconde étape : choix du mode de distribution

3.1. Deux caractéristiques essentielles du lancer de rayon

Avant de se lancer dans le choix du mode de distribution, deux caractéristiques du lancer de rayons influençant particulièrement la distribution devraient être mises en valeur.

En tout premier lieu, **le calcul de chaque rayon est indépendant** des autres, aucune communication ou synchronisation ne sera nécessaire de ce point de vue.

Le **jeu de données est statique**, au sens ou il n'est pas modifié durant les calculs. Seules les données concernant l'image calculée sont modifiées pendant le traitement.

3.2. Différentes manières de distribuer le lancer de rayons

Il existe plusieurs manières de distribuer l'algorithme de lancer de rayon, les plus fréquentes sont les suivantes :

- La **distribution selon les données :** commence par répartir les données de manière équitable entre les différents nœuds de calcul, ensuite chaque nœud commence le calcul d'un rayon primaire correspondant à la zone de données dont il dispose, dès

qu'un des rayons traités (primaires ou secondaires) sort de la zone de données disponibles en local son calcul est transféré au nœud possédant les données adaptées et le calcul d'un nouveau rayon (demandé par un autre nœud ou provenant de la liste de rayons primaires à traiter) est engagé [DIP84, KOB88].

Cette méthode de distribution a plutôt pour cible un supercalculateur disposant d'un réseau de communication efficace, elle engendre un taux de communication très important.

- La **distribution selon l'image à calculer** : définit comme tâches à accomplir des portions de l'image. Des ensembles de rayons sont donnés à calculer à chaque nœud et chaque nœud effectue le calcul de chacun des rayons qui lui ont été attribués intégralement. Cela implique que chaque nœud dispose de l'intégralité de la scène ce qui est par ailleurs le principal inconvénient de cette méthode. Deux types d'implémentation se rencontrent pour cette méthode de distribution.

  - La première est une distribution du modèle au sein des nœuds de calcul impliquant des accès à distance pour les données non locales, la distribution et les accès pouvant être implémentés de manière explicite dans le logiciel de lancer de rayon ou bien de manière implicite par l'utilisation de mémoire partagée virtuelle [PAN98]. Cette solution réduit la consommation mémoire de chaque nœud mais engendre un taux de communication important.

  - La seconde est une recopie de l'intégralité du modèle et de la structure de données contenant la voxelisation au sein de chaque nœud. Cette solution est beaucoup plus consommatrice de mémoire mais semble plus conforme à l'optique de la distribution selon l'image à calculer puisque chaque nœud est capable de calculer chaque rayon dans son intégralité sans aucun accès distant. La communication est beaucoup plus réduite que dans la première solution.

  Certaines implémentations enfin sont des hybrides des deux précédentes, différenciant généralement la gestion des données du modèle et celles de la structure de voxelisation [BAD94].

La distribution sur réseau de machines d'un algorithme de lancer de rayon est aujourd'hui encore l'objet de nombreuses recherches, une référence en la matière est le projet POV (Persistence Of Vision) [POV91]. Comme nous venons de le voir il n'existe pas une méthode que l'on pourrait qualifier de meilleure mais simplement plusieurs approches ayant chacune ses avantages et ses inconvénients. Il ressort tout de même de ce rapide tour d'horizon qu'un compromis doit être fait entre un taux de communication important ou une consommation mémoire importante.

3.3. Méthode choisie et raisons de ce choix

Le système de communication est le point faible des réseaux de d'ordinateurs, cible de nos travaux, le taux de communication sera donc le critère déterminant dans le choix d'une méthode de distribution.

Parmi les méthodes présentées dans le paragraphe 3.1 deux engendrent un important trafic réseau et sont par conséquent mal adaptées à une utilisation sur réseau de machines, c'est la troisième qui retiendra notre attention. L'algorithme de lancer de rayon sera donc découpé selon l'image à calculer et les données recopiées sur chaque machine.

Le choix de l'algorithme paresseux devrait permettre de limiter la recopie totale de données au modèle brut uniquement, l'arbre octal étant construit localement selon les besoins de chaque nœud de calcul. Notons que cet algorithme parallélise par la même occasion la construction habituellement séquentielle de l'arbre octal, il entre donc parfaitement dans le cadre des optimisations décrites précédemment.

4. Résultats et mesures

4.1. Description des indicateurs utilisés

Avant de commencer les expérimentations proprement dites il me semble important de définir ce qui sera considéré comme la performance d'une application sur une machine parallèle. La première caractéristique qui vient à l'esprit lorsque nous entendons parler de performance est la performance en temps de calcul : il faut en effet obtenir un temps de calcul minimum pour que nous puissions considérer notre application comme performante. Un second critère de performance réside dans une consommation mémoire réduite qui offre plusieurs avantages : elle est souvent un moyen d'améliorer la performance en temps de calcul, elle permet une moins grande exigence quant à la configuration matérielle minimum, elle offre la possibilité de travailler sur des jeux de données de départ plus important et enfin, dans un contexte multitâche, elle laisse « plus de place » aux applications concurrentes. Ces deux critères peuvent être exprimés sous différentes formes et niveaux de détail et leur obtention nécessite parfois de s'intéresser à des caractéristiques plus précises de l'application. Les indicateurs définitifs retenus pour les mesures de performances à venir sont détaillés dans la suite de ce paragraphe.

Le temps de calcul de l'algorithme distribué est celui du nœud qui termine son traitement en dernier, il est donné par la formule suivante :

$$T_p = \max_i t_i$$

Equation 9 : Temps d'exécution parallèle

dans laquelle *i* indice l'ensemble des nœuds de calcul et t_i représente le temps de calcul, mesuré, du i-ème nœud.

Le temps d'exécution brut a pour principal défaut de ne pas être assez général, en effet, il ne concerne qu'une implémentation donnée appliquée sur un jeu de données fixé et exécutée sur une machine précise. Nous ajouterons donc des indications telles que le Speed-Up ou l'Efficacité qui offre une information un peu moins ciblée.

L'efficacité d'une machine parallèle à p processeurs est donnée par la formule suivante :

$$E = \frac{T_s}{p \times T_p}$$

Equation 10 : Efficacité parallèle

dans laquelle T_s est le temps de calcul du meilleur algorithme séquentiel. L'efficacité sera un indicateur intéressant de performance de l'application distribuée.

Le temps de calcul optimal théorique, pour un algorithme distribué donné, survient lorsque tous les nœuds de calcul terminent leur travail en même temps il est alors de :

$$T_{min} = T_{seq} + \frac{T_{par}}{p}$$

Equation 11 : Temps minimum théorique

où T_{seq} et T_{par} sont respectivement les temps de calcul de la partie séquentielle et de la partie parallèle de l'algorithme (Loi de Amdahl : « l'amélioration de performance qui peut être obtenue en utilisant un mode d'exécution plus rapide est limité par la fraction de temps pendant laquelle ce mode peut être utilisé » [HEN96]).

L'Equation 11 montre que la proportion de code non parallélisable d'un algorithme (i.e. qui formera la partie séquentielle du code parallèle) est certainement le premier critère à prendre en compte lors du choix de l'algorithme séquentiel qui servira de base au travail de distribution. Il importe en effet d'être conscient de l'impact immense de la portion

résiduelle de code séquentiel sur la performance finale. Pour s'en convaincre il suffit d'un rapide calcul : si 1% du code reste séquentiel dans un algorithme parallélisé sur 100 machines, le gain maximal théorique sera de 50,25 au lieu de 100 soit presque divisé par deux.

Il est précisé précédemment que le temps minimum survient lorsque tous les nœuds de calcul terminent leur traitement en même temps, ce qui revient à dire pour des nœuds identiques que la charge de travail qui leur a été attribuée a été équivalente (et qu'elle leur a été attribuée au même moment). Un autre indicateur à surveiller sera donc l'équilibre de la charge de travail sur les différents nœuds de calcul de la machine cible.

Une estimation du déséquilibre de charge est obtenue par la formule [HSU96]:

$$\text{Déséquilibre} = \frac{Tp - \frac{T_s}{p}}{\frac{T_s}{p}}$$

Equation 12 : Déséquilibre de charge entre les nœuds de calcul

La performance en terme de consommation mémoire sera exprimée par la consommation maximale qui est plus à même de rendre compte de la capacité de l'application à traiter efficacement des jeux de données volumineux. La quantité de mémoire maximale nécessaire à l'exécution de l'application, M, sera celle du nœud de calcul qui en aura consommé le plus :

$$M = \max_i m_i$$

Équation 13 : Consommation mémoire d'un programme parallèle

où m_i est la consommation mémoire maximale du nœud i.

La distribution d'un algorithme offre souvent un coût intrinsèque à celle-ci, le travail des nœuds de calcul n'est pas continu mais entrecoupé de communications avec le maître ou avec les autres esclaves. Dans le cas de notre application de lancer de rayons distribuée aucune communication entre esclaves ne sera nécessaire, les communications qui auront lieu seront de trois types : du maître vers un esclave pour attribuer une tâche ou d'un esclave vers le maître pour lui indiquer la terminaison de la tâche en cours ou pour lui envoyer des résultats locaux. Dans la pratique les échanges de messages vérifieront toujours le schéma suivant vu par le côté esclave : envoi de l'indication de fin de tâche, envoi des résultats associés puis réception de la nouvelle tâche à accomplir. Ce mode de fonctionnement permet d'effectuer une mesure de « temps de service » qui correspond ici

au temps qui s'écoule entre la fin de la tâche courante et le début de la tâche suivante. Le temps passé par l'esclave en temps de service, autrement dit à envoyer les résultats et attendre une nouvelle tâche, par rapport au temps total de calcul de l'esclave représente ce que je qualifie de « Pourcentage de Communication », il est obtenu par la formule suivante :

$$PC_i = \frac{\sum \text{temps de services}_i}{\frac{\text{durée du calcul}}{t_i}} \times 100$$

$$PC = \frac{\sum_{\text{nœuds de calcul}} PC_i}{\text{nombre de nœuds de calcul}}$$

Équation 14 : Indicateur de pourcentage de communication

t_i étant le temps de calcul mesuré du i-ème nœud. Le pourcentage de communication PC sera un bon indicateur de la performance de l'algorithme de distribution lui-même.

En conclusion, les cinq indicateurs finalement retenus pour estimer la qualité de l'application distribuée de lancer de rayon seront :

- La proportion de la partie séquentielle du traitement (T_{seq} dans l'Equation 11).
- L'efficacité E (Equation 10).
- La qualité de l'équilibre de charge obtenu grâce à cette solution, mesurée par le pourcentage de déséquilibre (Equation 12).
- La quantité de mémoire nécessaire à l'exécution de cette solution dont la réduction garantit, entre autre, la capacité à traiter efficacement des problèmes de grande taille (Équation 13).
- La proportion de communication qui permettra d'estimer le coût de la parallélisation elle-même (Équation 14).

Les indicateurs étant maintenant choisis nous allons pouvoir commencer les expérimentations en nous basant sur les mesures de ceux-ci. Il est bien évident qu'il sera parfois souhaitable de compléter les indications relevées ici par des mesures ponctuelles d'autres critères de performance pour faire apparaître ou simplement préciser les raisons de phénomènes ponctuels.

4.2. Mesures commentées

Des mesures absolues à mettre en regard de résultats extérieurs m'ont semblé difficiles à utiliser. En effet, il existe une multitude « d'options » au niveau de l'algorithme de lancer de rayon proprement dit dont les valeurs apparaissent rarement dans les documents traitant du sujet ce qui rend délicate toute comparaison de cet ordre.

La méthode de mesure qui m'est apparue comme la plus raisonnable est la comparaison d'algorithmes différents au sein d'un squelette identique, le squelette contenant les parties communes à toutes les versions : l'initialisation des esclaves, la lecture du modèle, l'écriture des résultats et les outils de mesures.

4.2.1. Conditions expérimentales

Sauf indication contraire, les valeurs consignées dans les graphes qui vont suivrent correspondent à la moyenne effectuée parmi les résultats de nombreuses scènes. Ces scènes ont été obtenues de la manière suivante :

- Cinq scènes de la bibliothèque standard de test des applications de lancer de rayon de Eric Haines (« Standard Procedural Database » [SPD]) ont été utilisées : gears, mount, teapot, tetra et tree.
- Chacune de ces scènes a ensuite été traitée selon plusieurs complexités afin d'obtenir un panel représentatif des différentes tailles de scènes. Ces différentes tailles correspondant à :
  - de petites scènes (quelques milliers de facettes)
  - des scènes de taille moyenne (jusqu'à quelques centaines de milliers de facettes)
  - des scènes de grande taille (entre plusieurs centaines de milliers de facettes et environ un million de facettes).

Bien que ces tests ait été effectués sur différentes machines, les résultats présentés dans ces graphes proviennent des exécutions effectuées sur la grappe de calcul de Supaero (8 machines biprocesseurs 550 MHz disposant chacune de 256Mo de mémoire, réseau Fast Ethernet) (voir description complète Annexe I-3).

Des résultats isolés de certaines de ces scènes sont présentés Annexe II (calculés sur la grappe de calcul du DTIM).

4.2.2. Algorithmes de test

Trois algorithmes seront utilisés pour les tests :

Algorithme 1 : Un algorithme séquentiel sans paresse est choisi et un découpage du travail en autant de sous-ensembles que de nœuds de calcul lui est appliqué. Chaque sous-ensemble correspond à une partie de l'image à calculer et chacune de ces parties a la même taille. En pratique, l'image est découpée en N bandes verticales de même largeur, N étant aussi le nombre de nœuds de calcul. Cet algorithme est un exemple trivial visant à mettre en valeur les difficultés posées par la distribution du lancer de rayon.

Algorithme 2 : L'algorithme séquentiel sans paresse de l'algorithme 1 est conservé mais le découpage du travail s'effectue maintenant en beaucoup plus de sous-ensembles que de nœuds de calcul. En pratique, l'image est maintenant découpée selon une grille régulière. La finesse de ce découpage (i.e. la taille des mailles élémentaires) fera l'objet d'une discussion plus étendue mais, pour ces premiers tests, une valeur de 32x32 sera utilisée. Quelques essais préalables ont en effet montré que cette valeur apportait d'assez bons résultats. Les tâches sont attribuées dynamiquement au fur et à mesure que les nœuds de calcul terminent leur tâche précédente. L'algorithme 2 est un exemple très conforme à ce que l'on retrouve habituellement dans la littérature et il servira de point de repère.

Algorithme 3 : L'algorithme de distribution de l'algorithme 2 est conservé et c'est, cette fois, l'algorithme séquentiel classique qui est abandonné en faveur de l'algorithme paresseux.

Ces algorithmes ont chacun leur rôle à jouer dans la compréhension de la problématique « lancer de rayon distribué » : le premier représente une solution triviale mettant en valeur l'inapplicabilité des solutions statiques employées habituellement pour les applications régulières ; le second étant très conforme à ce qui se fait à ce jour va servir de référence, il utilise les solutions classiques employées pour les applications irrégulières et enfin le troisième contient notre approche et vise à mettre en valeur ses qualités et ses défauts afin de mieux la juger et éventuellement l'améliorer.

4.2.3. Mesures

4.2.3.1. Proportion de code séquentiel

Afin de déterminer la partie séquentielle des deux algorithmes voici un schéma de fonctionnement de chacun d'eux annoté de façon à faire apparaître clairement les parties séquentielles (en rouge) et parallèles (en vert) du traitement effectué par chacun d'eux (Figure 45)

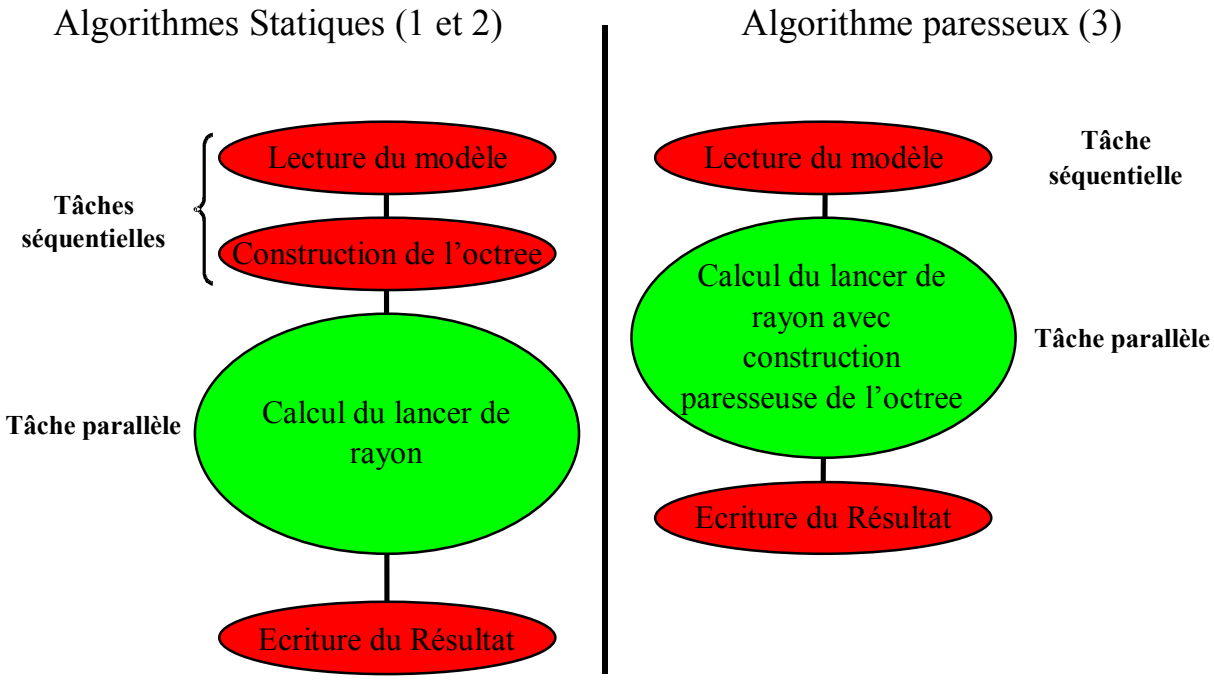


Figure 45 : Proportion fonctionnelle partie séquentielle – partie parallèle

L'utilisation de l'algorithme de lancer de rayon dans une application indépendante n'est pas réaliste. Cet algorithme est le plus souvent intégré dans un module de calcul d'un logiciel d'imagerie ou d'animation en trois dimensions réalistes. Dans ces conditions la phase de lecture du modèle n'a plus lieu en tant qu'étape préliminaire mais est à la charge du logiciel hôte. Elle s'effectue vraisemblablement au fur et à mesure de la création du modèle par le logiciel 3D. La phase finale d'écriture du résultat n'a pas lieu d'être non plus, les résultats partiels sont directement intégrés à l'image attendue par le logiciel hôte. Ces deux étapes ne seront par conséquent plus prises en compte dans la suite de ce document.

Le schéma (Figure 45) montre que la version paresseuse dispose d'une étape séquentielle de moins que la version statique d'un point de vue fonctionnel : La construction de l'arbre octal n'est plus une étape préliminaire mais est intégrée au calcul distribué. La Figure 46, quant à elle, fait la même analyse entre la proportion de code parallélisable et non parallélisable mais d'un point de vue quantitatif. On y découvre qu'en moyenne, sur un ensemble varié de cas, l'étape de construction de l'arbre octal représente 11% du temps de calcul total (lors d'un calcul séquentiel).

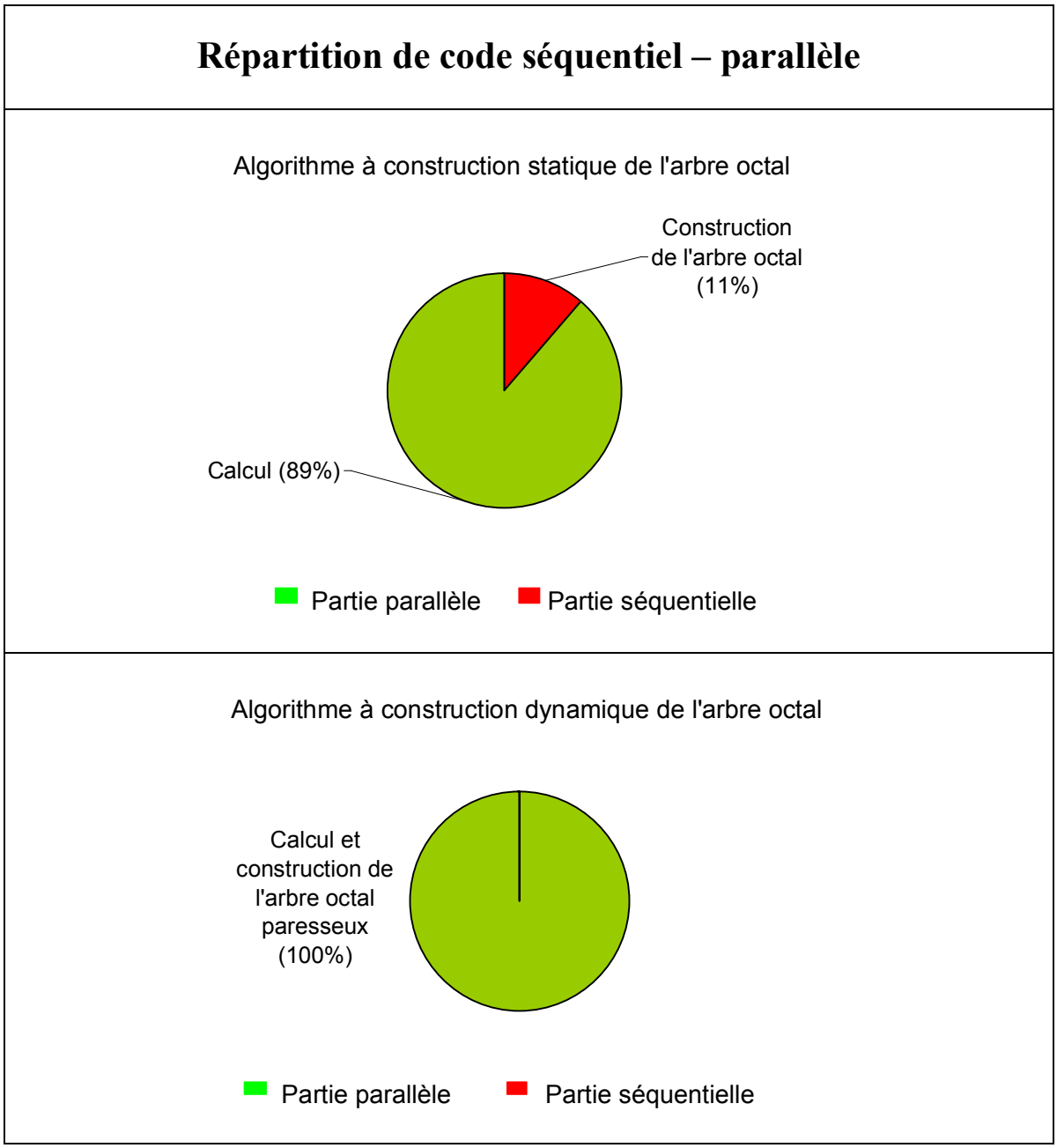


Figure 46 : Proportion quantitative partie séquentielle – partie parallèle

L'algorithme paresseux n'effectue pas une construction totale de l'arbre octal préalablement au calcul mais une construction partielle durant celui-ci. La construction de l'arbre octal, habituellement une partie séquentielle du lancer de rayon distribué, est maintenant parallélisée ce qui améliore la performance potentielle de l'algorithme 3 sur ces concurrents selon le critère de l'Equation 11.

4.2.3.2. Efficacité

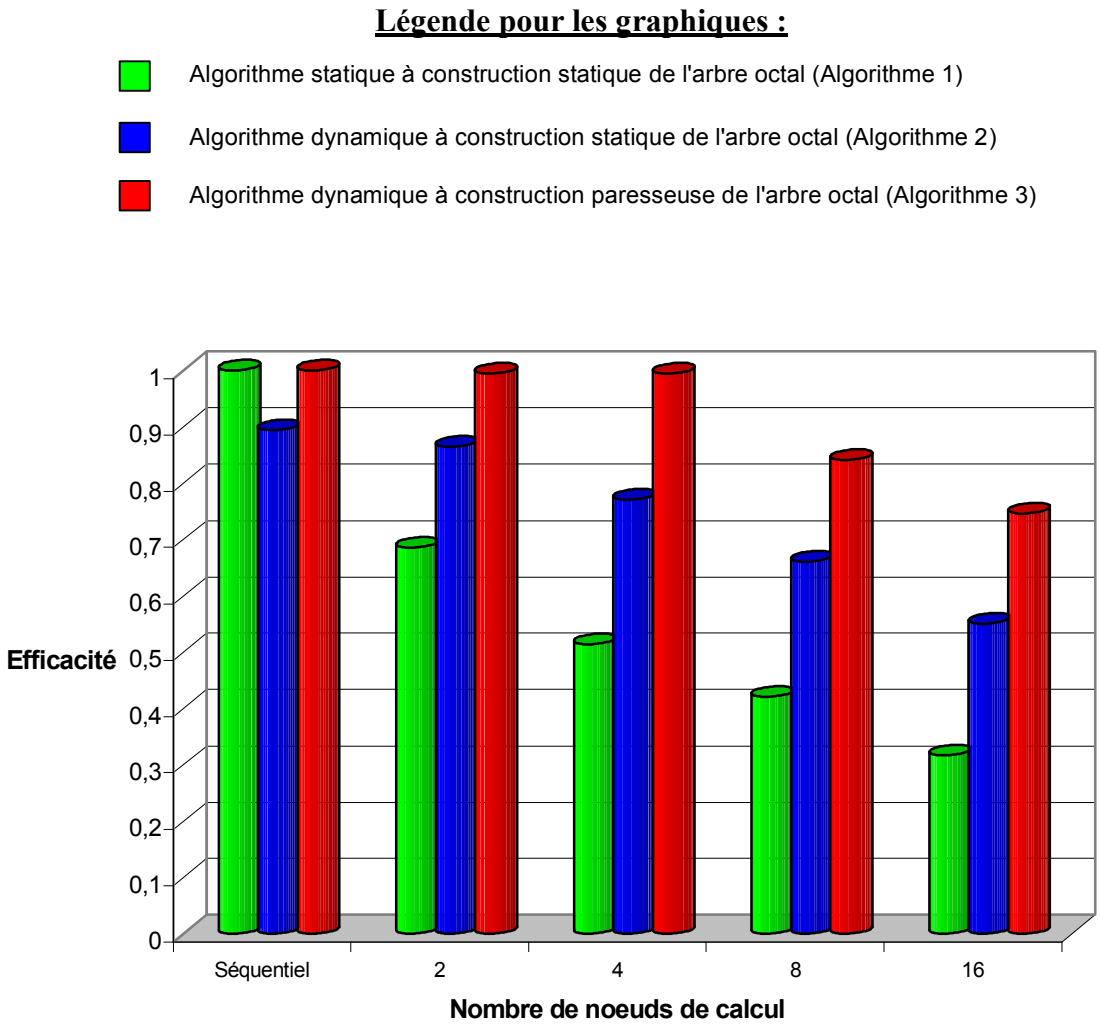


Figure 47 : Efficacité (Equation 10)

Du point de vue de la performance séquentielle (un seul esclave) :

On peut remarquer en comparant les algorithmes 1 et 2 que la communication introduite par la distribution du travail par un maître à un esclave entraîne un surcoût significatif sur le temps d'exécution par rapport à une exécution complètement séquentielle.

Cette observation n'est pas surprenante en elle-même, mais elle trouve tout son intérêt une fois mise en regard avec les résultats de l'algorithme 3. Ce dernier construit l'arbre octal de manière paresseuse ce qui amène à des allocations dynamiques multiples qui représentent, elles aussi, un surcoût de temps d'exécution. Pourtant les performances de ce dernier algorithme sont du même ordre que celle de la version stricte à découpage statique (Algorithme 1). Les apports de la paresse, la localité et la non construction de parties de la l'arbre octal correspondant à des zones cachées de la scène, compensent donc le coût des allocations dynamiques ainsi que celui de l'algorithme de distribution. En effet, les performances qui étaient dégradées par la distribution du travail (observé pour l'algorithme 2) sont maintenant ramenées à leur niveau initial. **Le coût de la paresse ne doit donc pas être simplement considéré comme le coût des allocations dynamiques multiples par rapport à celui d'une allocation statique unique**.

En ce qui concerne les performances distribuées (Figure 47) :

L'efficacité de l'algorithme 1 décroît rapidement lorsque le nombre de nœuds augmente : l'algorithme statique souffre certainement d'un déséquilibre de charge important (qui sera confirmé Figure 48). En effet le temps de calcul global correspond au temps de calcul local le plus long et un algorithme déséquilibré contient une période finale pendant laquelle seule une partie des nœuds de calcul sont actifs d'où une efficacité globale basse.

L'algorithme numéro 2 offre des performances bien meilleures que le premier (nous vérifierons dans le paragraphe suivant qu'un meilleur équilibre de charge est à la source de cette amélioration).

Comme dans le cas de l'exécution séquentielle l'algorithme paresseux offre les meilleures performances, son speed-up est linéaire jusqu'à 4 nœuds de calcul et son rendement est toujours supérieur à celui des deux autres solutions.

4.2.3.3. Déséquilibre de charge

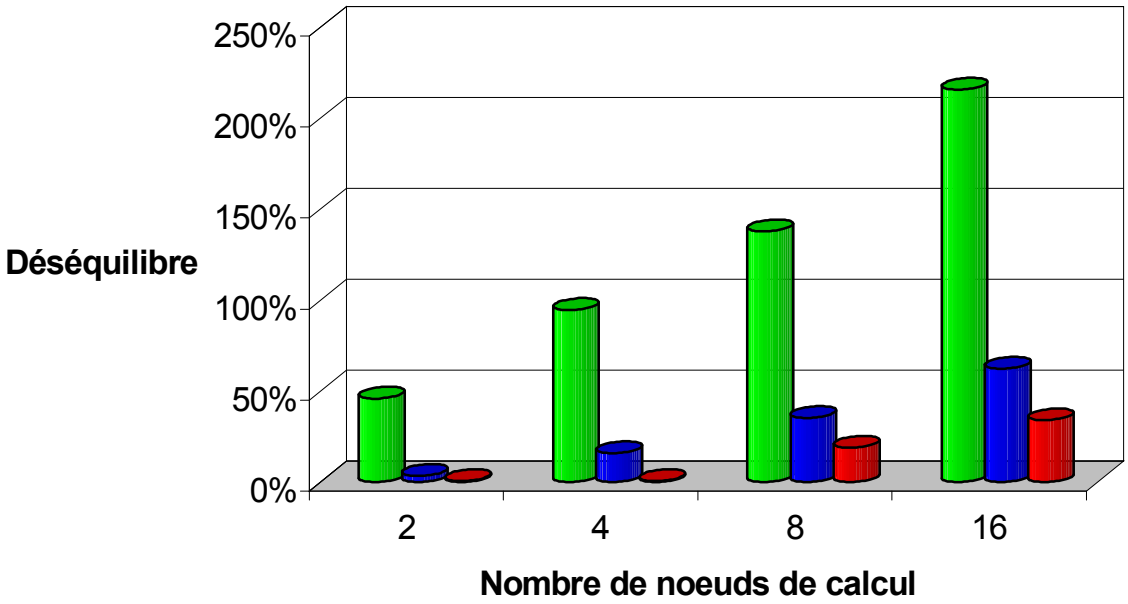


Figure 48 : Déséquilibre de charge (Equation 12)

Lors des exécutions du premier algorithme, chaque nœud doit calculer une portion équivalente de l'image (équivalente s'entend ici du point de vue de la surface des portions d'image) et pourtant le temps de calcul de chacun de ces nœuds est très différent (Figure 48). Le déséquilibre de charge observé fait très clairement apparaître l'irrégularité du lancer de rayon.

Le déséquilibre de charge observé pour les algorithmes 2 et 3 est beaucoup moins important, il montre la bonne adéquation d'une distribution dynamique du travail. L'algorithme 2 contient une partie séquentielle correspondant à la construction statique et totale de l'arbre octal. La charge correspondant à cette étape n'est pas distribuée et éloigne par conséquent l'algorithme 2 de la répartition idéale de la charge totale de calcul (Equation 12). L'algorithme 3, ne possédant pas de partie séquentielle, obtient une meilleure répartition de charge.

La non uniformité de la complexité de calcul des différentes parties de l'image à calculer, observée grâce aux résultats de l'algorithme 1, se traduit par des différences de surfaces entre les portions d'images calculées par chacun des nœuds.

La Figure 49 illustre ce propos en mettant en regard deux versions d'une même scène :

- Dans la première scène la complexité est à peu près uniformément répartie par rapport à l'image à calculer (représentée en haut à gauche). L'algorithme de répartition du calcul, tentant d'équilibrer la charge, aura finalement attribué une surface à peu près équivalente de l'image à calculer à chaque nœud lorsque le traitement se terminera. C'est ce que nous pouvons constater dans l'image en haut à droite qui représente par un niveau de gris différent le travail effectué par chaque nœud (nœuds qui sont au nombre de quatre dans le cas présent).
- La seconde scène reprend la même description que la première, les mêmes objets, les mêmes sources lumineuses, mais le point de vue a été décalé vers la gauche. Cette modification a eu pour conséquence de concentrer la charge de travail vers la droite de l'image à calculer (représentée en bas à gauche). Une fois encore l'image de la répartition du travail sur les esclaves (en bas à droite) est conforme à ce que nous espérions : les deux nœuds de calcul (blanc et noir) ayant commencé leur travail dans des zones vides de la scène ont calculé de plus grandes surfaces de celle-ci que les deux nœuds (gris clair et foncé) ayant été tout de suite confrontés à des objets plus complexes.

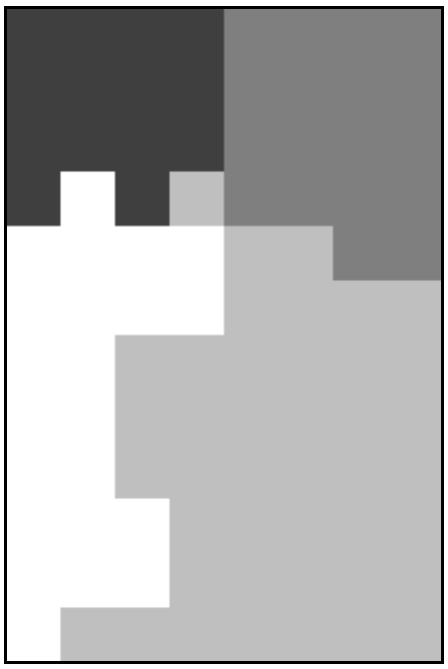
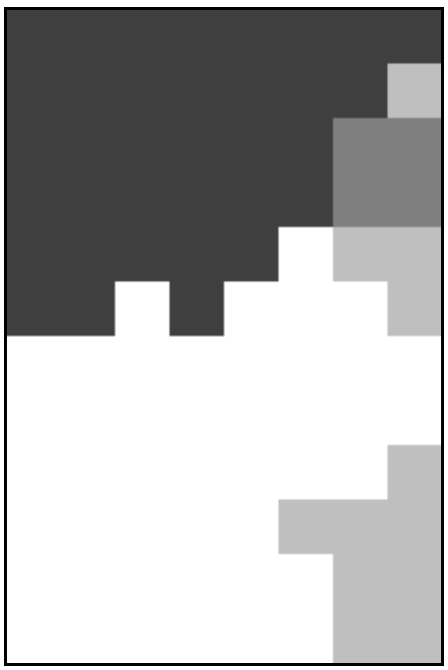
Image	Répartition du travail
	
	

Figure 49 : Répartition de la charge au sein d'une scène

4.2.3.4. Mémoire utilisée

Pour les tests de consommation mémoire seules les scènes de grande taille ont été utilisées.

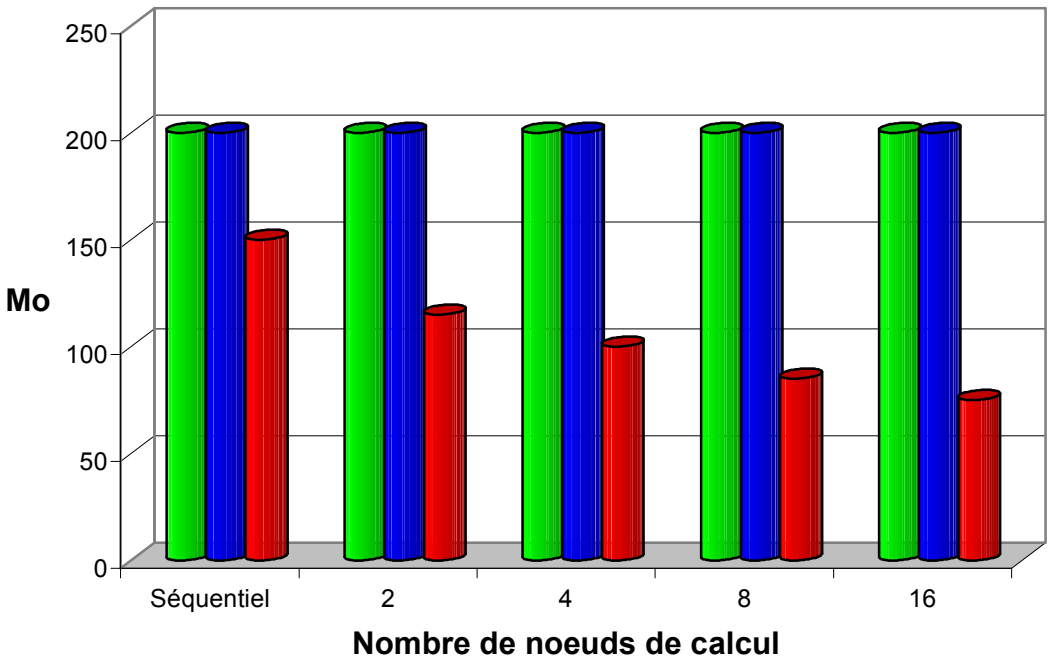


Figure 50 : Consommation moyenne en mémoire locale par nœud de calcul

En séquentiel :

Les deux premiers algorithmes étant stricts ils amènent au même résultat, correspondant à une construction complète de l'arbre octal pour la scène concernée et donc à une consommation mémoire maximale.

L'algorithme paresseux quant à lui amène un gain de 25% sur la consommation mémoire grâce à l'économie réalisée en ne construisant pas les zones de la scène inutiles au calcul.

En parallèle :

Dans les deux algorithmes stricts la consommation mémoire est maximale. Chaque nœud de calcul contient l'arbre octal complet, la consommation de chaque nœud est donc la même que celle que l'on observe lors de l'exécution séquentielle.

La consommation mémoire de la version paresseuse décroît lorsque le nombre de nœuds augmente. **Les données sont donc distribuées au sein de la machine parallèle sans qu'aucune disposition n'ait été explicitement prise en ce sens au niveau de la distribution du calcul.** Cette réduction n'est toutefois pas linéaire, ce qui signifie que certaines données sont présentes dans les mémoires locales de plusieurs nœuds. Ceci est lié à l'application : le calcul de deux rayons différents, parfois très éloignés l'un de l'autre, peut nécessiter des données communes. Le principe de la paresse étant d'obtenir localement toute information nécessaire au calcul, ces données seront présentes dans chaque nœud de calcul dont le traitement aura montré cette nécessité. Le gain en consommation mémoire est moins important au fur et à mesure que le nombre de nœuds augmente. Une fois encore ce phénomène est dû à l'application elle-même, en effet certaines données sont nécessaires à chaque nœud quel qu'en soit le nombre. La première de ces données est constituée des données brutes du modèle, en second viennent les premiers nœuds de l'arbre octal ; le nœud racine par exemple est forcément présent dans chaque mémoire locale et le résultat de l'évaluation de ce nœud racine, ses huit fils, sera lui aussi présent au sein de chaque nœud de calcul. Le volume de données représenté par cette catégorie d'information, constant quel que soit le nombre de nœuds de calcul, est moins significatif lorsque la consommation globale est importante. Mais au fur et à mesure que celle-ci diminue, la proportion de données « incontournables » croît et le gain en consommation mémoire s'effectue sur une partie de plus en plus petite du volume d'information.

4.2.3.5. Pourcentage de communication

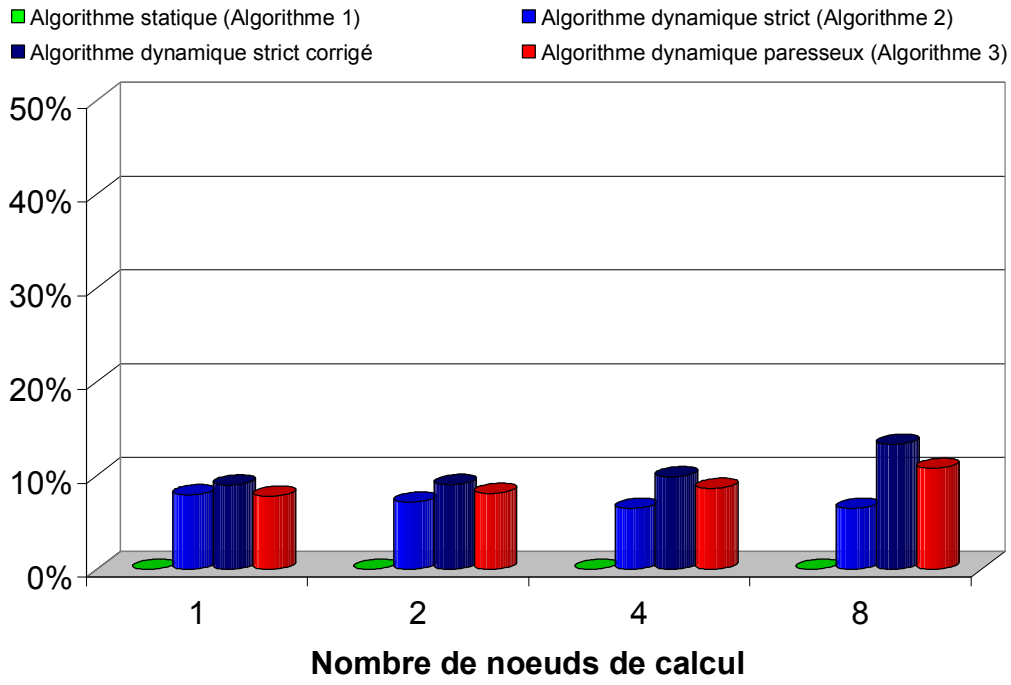


Figure 51 : Taux de communication des différents algorithmes

L'algorithme statique (Algorithme 1) possède un taux de communication négligeable c'est ici son seul avantage. Le découpage étant équitable entre tous les nœuds de calcul, la tâche de chacun d'entre eux est entièrement déterminé par leur rang au sein de la machine parallèle, aucune communication n'est donc nécessaire pour commencer le calcul. Un seul message sera, en réalité, envoyé par chaque nœud à la fin de son calcul pour envoyer la portion d'image qu'il vient d'évaluer.

Les algorithmes dynamiques (Algorithmes 2 et 3) ayant un système de distribution du travail basé sur des échanges de messages leur taux de communication est plus important. Ce taux est assez stable jusqu'à quatre nœuds et on observe une augmentation un peu plus marquée sur huit nœuds de calcul. Il est en pire cas de l'ordre de 10%, indiquant par conséquent qu'au moins 90% du temps est consacré au calcul proprement dit.

On remarque qu'une première mesure du pourcentage de communication de l'algorithme 2 indique une valeur meilleure que celle de l'algorithme 3. Plus surprenant, ce taux semble quasiment constant quel que soit le nombre de nœuds utilisé. Ce comportement trouve son

explication en mettant en regard le taux de communication avec le temps d'exécution global (Figure 52) utilisé pour son calcul.

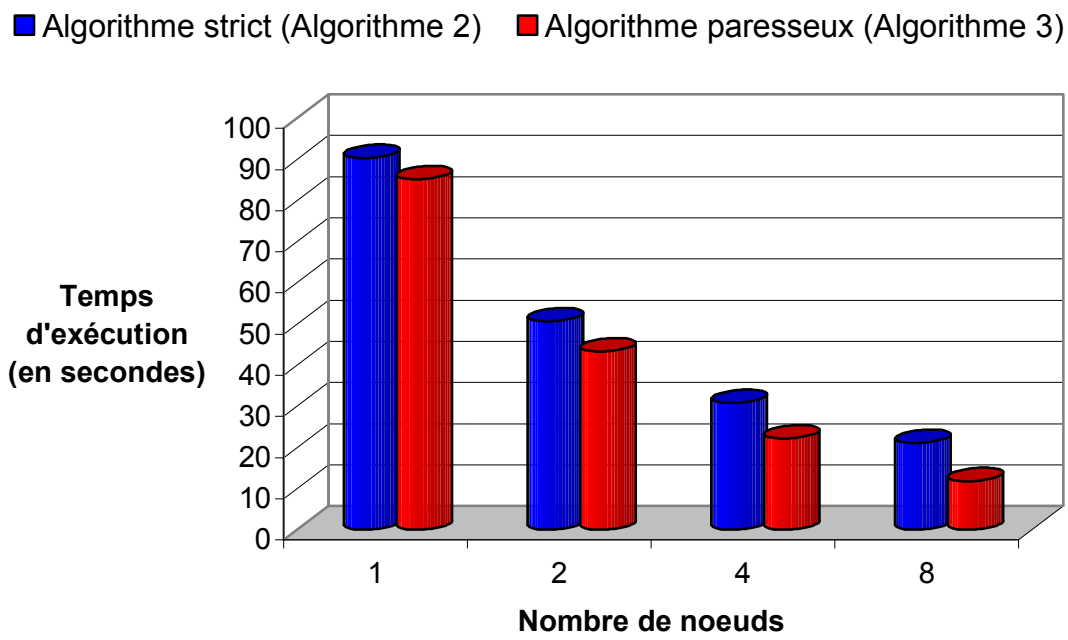


Figure 52 : Temps d'exécution algorithmes strict et paresseux

Le temps global d'exécution de l'algorithme 2 est plus long que celui de l'algorithme 3 et cette différence s'accentue avec l'augmentation du nombre de nœuds. La partie du temps global correspondant à la construction de l'arbre octal amène cette nette différence entre l'algorithme strict et l'algorithme paresseux.

Temps global = Calculs + Construction de l'arbre octal (partielle ou totale)

Equation 15 : Décomposition du temps global de calcul de la version stricte

Les deux versions vérifient l'Equation 15 mais sa signification n'est pas tout à fait la même :

- Dans la version paresseuse la construction de l'arbre octal n'est que partielle et se fait au fur et à mesure des calculs
- Dans la version stricte la construction de l'arbre octal est totale et faite préalablement à tout calcul.

La construction de l'arbre octal est une tâche séquentielle pour la version stricte, tous les nœuds de calcul l'effectuent en même temps et sans communication, seulement ensuite vient le calcul de l'image proprement dit et les communications qui vont avec (Figure 53).

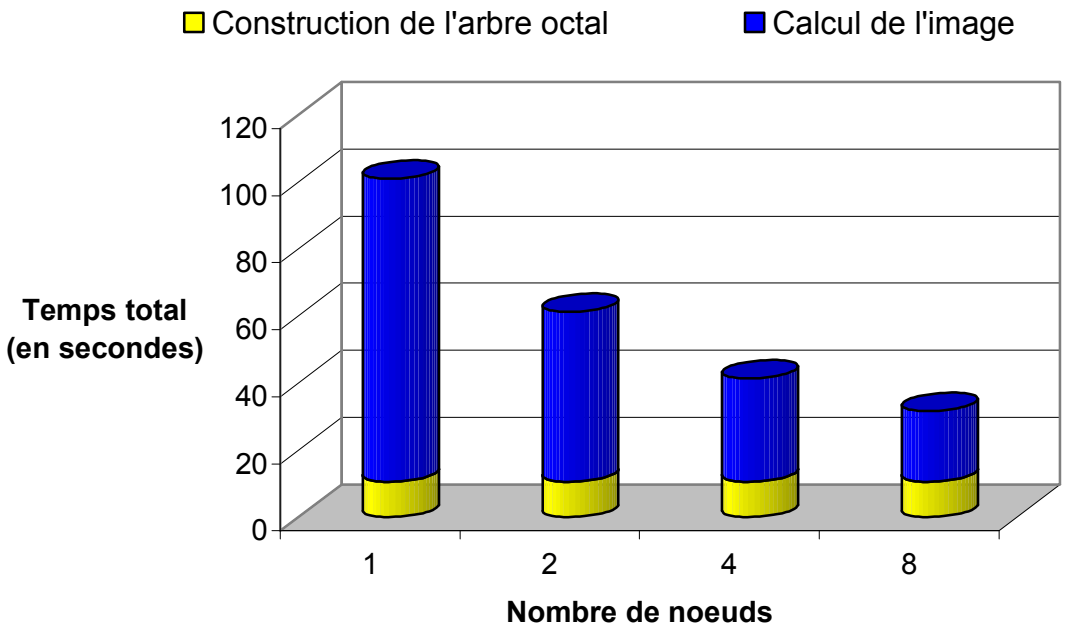


Figure 53 : Temps de construction de l'arbre octal strict

Le temps global d'exécution de l'algorithme strict est donc plus long que celui de la version paresseuse, de plus la partie de celui-ci correspondant à la construction de l'arbre octal s'effectue sans communication : un volume de communication similaire est donc mis en balance avec un temps plus important ce qui produit un taux de communication inférieur.

Le temps de construction de l'arbre octal est le même quel que soit le nombre de nœuds. Le temps de calcul diminuant quant à lui lorsque le nombre de nœuds croît, la proportion de temps sans communication devient plus importante au fur et à mesure qu'augmente le nombre de nœuds de calcul. Ce phénomène compense l'intensification du trafic réseau engendré par l'augmentation du nombre de nœuds gérés par le serveur ainsi le taux de communication semble-t'il stable par rapport au nombre de nœuds.

Ces résultats laissent supposer qu'il y a moins de communication dans la version stricte et que cette communication est indépendante du nombre de nœuds communicants. Une seconde mesure est donc faite en utilisant cette fois le temps du calcul proprement dit (obtenu en utilisant l'Equation 15 « en sens inverse »), elle est référencée comme le taux

de communication de l'algorithme dynamique strict corrigé (Figure 51). Cette fois le taux de communication croît avec le nombre de nœuds ce qui est plus conforme à la réalité.

4.2.4. Conclusion

4.2.4.1. En séquentiel

Globalement le bilan de l'utilisation de l'algorithme paresseux est très positif, son coût en performance est compensé par son apport en localité et, dans la grande majorité des cas, la réduction de la consommation mémoire est notable. Il est à noté qu'il existe une plage de taille de modèle (non prise en compte dans les résultats ci-dessus) à l'intérieur de laquelle la version paresseuse offre des performances très supérieures à la version stricte : cette plage correspond aux cas où la consommation mémoire de la version stricte amène le système à recourir à l'utilisation d'une mémoire secondaire alors que la version paresseuse se contente de la mémoire physiquement disponible. Enfin, il existe des scènes qui ne peuvent être calculées qu'avec la version paresseuse, puisque la consommation mémoire de la version stricte dépasserait les capacités du système (mémoire physique + mémoire secondaire) alors que la version paresseuse permet de ne pas dépasser cette limite.

L'utilisation de la paresse est une alternative intéressante aux solutions actuelles et mérite d'être évaluée lorsqu'un problème de consommation mémoire est découvert. La thèse de Sébastien Bermes [BER98] contient de plus amples informations sur l'utilisation de la paresse dans un cadre séquentiel.

4.2.4.2. En distribué

Nous l'avons vu, l'utilisation de la paresse permet de distribuer une étape habituellement séquentielle : la construction de l'arbre octal. Ceci revêt une grande importance compte tenu de l'impact des parties séquentielles du traitement sur l'efficacité de la machine parallèle.

Les données ne sont plus totalement mais seulement partiellement recopiées, une partie de celles-ci est maintenant distribuée sans aucune intervention explicite du programmeur (la programmation distribuée étant difficile il m'a semblé important de souligner que la paresse, utilisée dans un but de distribution des données, ne complique pas la programmation par rapport à une recopie totale des données sur chaque nœud). Aucune méthode n'ayant actuellement été trouvée pour rapidement déterminer une répartition efficace des données au sein d'une machine parallèle dans le cadre des algorithmes irréguliers, la paresse apparaît comme une solution séduisante. Sa capacité de répartition automatique des données soulage le programmeur d'une tâche parfois très lourde et qui,

dans le cadre des applications irrégulières, amènera la plupart du temps à une moins bonne efficacité. La paresse peut cependant être mise en défaut par une application ou un jeu de données particulier nécessitant un trop grand nombre d'informations incontournables qui amènerait par conséquent à une redondance proche ou même équivalente à une recopie globale des données. Notons que, même dans un tel cas, la paresse n'amène qu'un faible surcoût puisque le coût supplémentaire provenant des allocations dynamiques est en partie compensé par l'amélioration de la localité des données.

Le grain de paresse, que je définirais comme la taille minimale de donnée allouable dynamiquement, est également à considérer avec attention puisque les allocations multiples dues à la dynamicité de cet outil ont un coût supérieur à une allocation unique. Un grain trop fin amène le système à consacrer une trop grande partie de son temps à ces allocations, un grain trop gros limite les apports en localité et en consommation mémoire susceptibles d'être obtenus par l'utilisation de cet outil. Ce grain peut être déterminé en fonction de différents critères suivant le but à atteindre : si le but à atteindre est la performance, il faut trouver un grain de paresse amenant le maximum de localité sans être trop pénalisant pour le système c'est à dire le grain le plus fin qui ne produise pas d'allocations excessives ; si le but est de faire fonctionner l'algorithme dans un espace de mémoire donné, il faudra alors choisir le grain de paresse le plus gros possible qui permette de rester dans les limites fixées de consommation mémoire.

5. Tests et optimisations

Nous venons de voir que l'algorithme de lancer de rayon paresseux associé à une attribution dynamique du travail était préférable aux autres solutions testées. Regardons à présent de plus près la distribution elle-même : les divers paramètres sur lesquels nous pouvons agir et les éventuelles optimisations offertes par les outils dont nous disposons.

5.1. Influence de la taille des blocs

J'ai indiqué précédemment que les algorithmes 2 et 3 bénéficiaient d'un découpage de l'image en un nombre de blocs très supérieur au nombre de nœuds de calcul afin d'obtenir un meilleur équilibre de charge.

Plus les blocs seront petits plus la différence de temps de calcul entre deux blocs aura de chance d'être minime, il semble donc qu'il faille choisir des blocs les plus petits possibles (Figure 54).

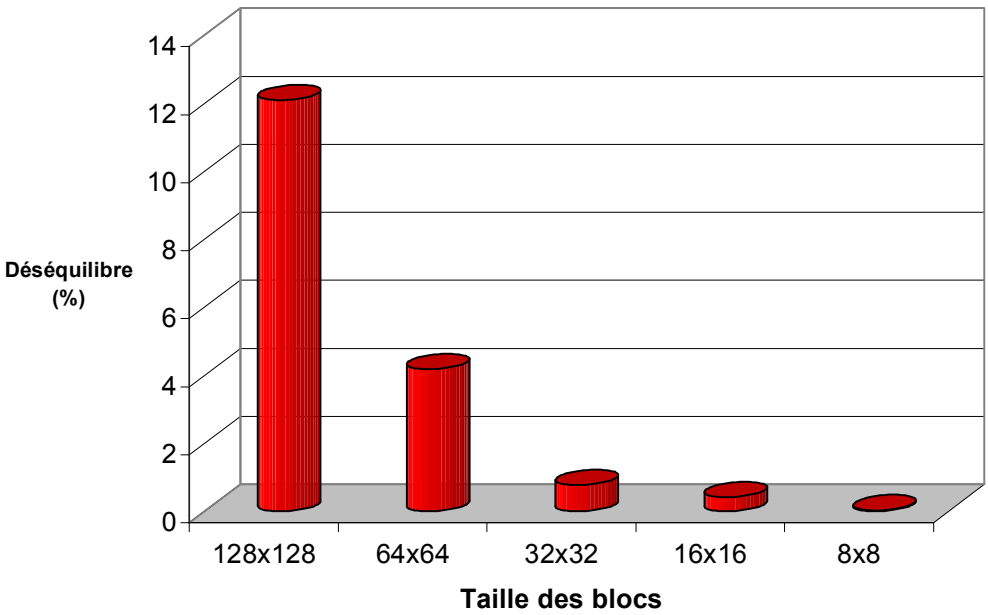


Figure 54 : Déséquilibre de charge sur huit nœuds (découpage statique)

Ce serait certainement le cas si chaque bloc n'engendrait aussi un volume de communication correspondant à l'assignation de ce bloc à un esclave par le maître puis à l'indication au maître par l'esclave que le travail est terminé.

La récupération des résultats n'est pas prise en compte dans cette évaluation. En effet, elle peut faire l'objet d'un traitement différent de celui des calculs, les résultats peuvent être conservés par l'esclave et envoyés régulièrement au maître ou même seulement lorsque tous les blocs ont été calculés dans un message contenant la totalité des résultats de l'esclave.

L'augmentation du nombre de blocs, provoquée par l'utilisation de blocs plus petits, entraîne donc une augmentation du taux de communication lors de l'exécution (Figure 55). Ce taux devant être le plus petit possible il semble à présent qu'il faille utiliser des blocs les plus grands possibles.

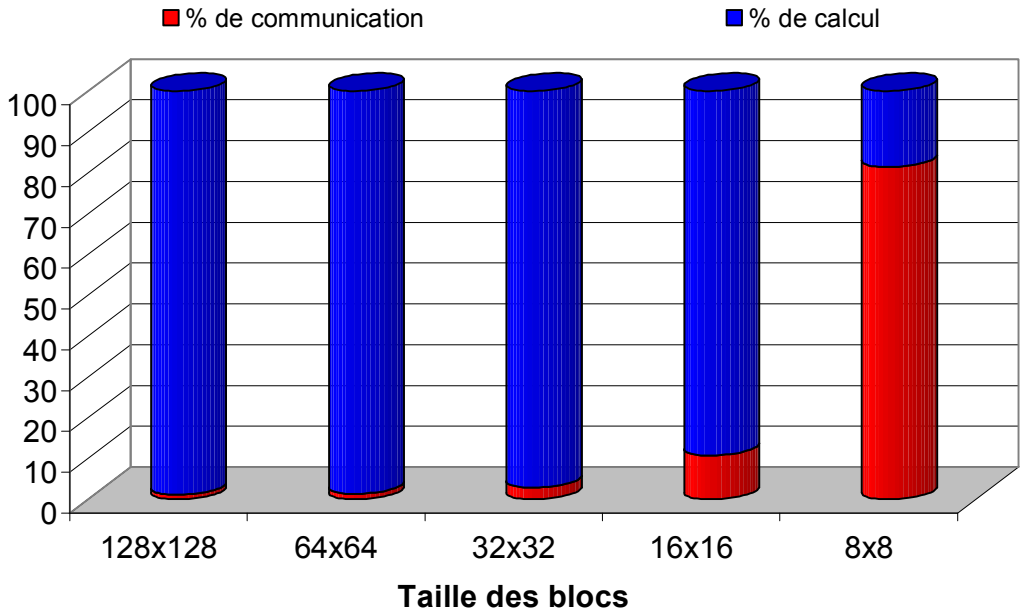


Figure 55 : Taux de communication sur huit nœuds (découpage statique)

Il y a donc un compromis à faire entre l'équilibre de charge le meilleur possible, par l'utilisation de blocs suffisamment petits, et le taux de communication minimum, par l'utilisation de blocs suffisamment grands. Ce choix est d'autant plus difficile à faire qu'il dépend en grande partie de la complexité de l'image calculée par rapport à sa taille (cette complexité n'étant pas évaluable à priori) et des caractéristiques de la machine cible (fréquence d'horloge et propriétés du réseau en particulier).

La solution que je propose est d'utiliser une granularité adaptable (décroissante dans le cas présent). Concrètement cela consiste à effectuer un découpage récursif et dynamique de l'image à calculer selon des grands blocs au début du calcul, pour obtenir un taux de communication correct, puis vers des blocs de plus en plus petits au fur et à mesure que la fin du calcul se rapproche afin de conserver un bon équilibre de charge. Ce processus de découpage s'arrêtant à une taille minimum fixée préalablement.

La détermination de la taille minimum de bloc est aussi complexe que celle d'une taille optimale malheureusement, mais son impact est beaucoup plus limité puisqu'elle n'est utilisée qu'à la fin du calcul. Le déséquilibre observé en cas de surévaluation de la taille minimale de bloc est le même que celui observé avec un découpage statique (Figure 54) par contre le taux de communication engendré en cas de sous-évaluation de la taille minimale de bloc est bien moins pénalisant que celui de l'algorithme à découpage statique (Figure 56).

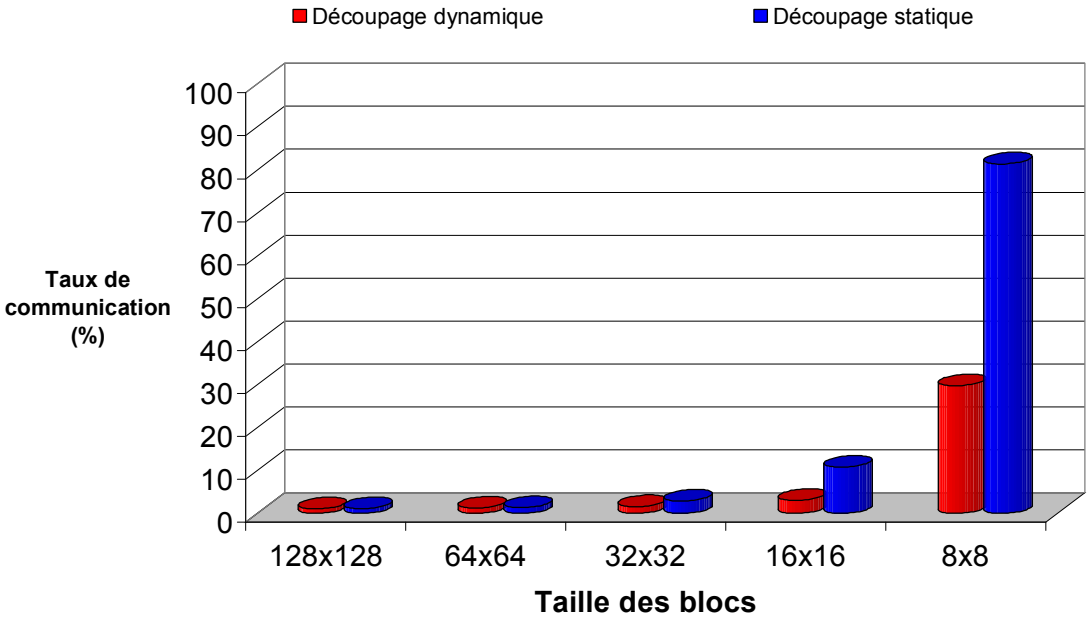


Figure 56 : Comparaison des taux de communication sur huit nœuds

On voit (Figure 56) que pour une taille de blocs de 8x8 pixels le taux de communication observé passe de plus de 80% avec un découpage statique à seulement 28% avec un découpage récursif. Ce dernier taux reste important mais sans aucune mesure avec celui de l'algorithme à découpage statique.

Pour mieux se rendre compte de ce qui se déroule durant le calcul d'une scène comparons maintenant le chronogramme de la version à découpage statique à celui de la version à découpage dynamique (Figure 57 et Figure 58).

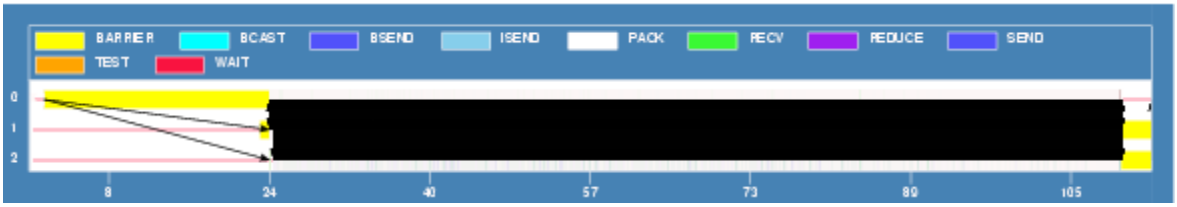


Figure 57 : Echange de messages de l'algorithme à découpage statique

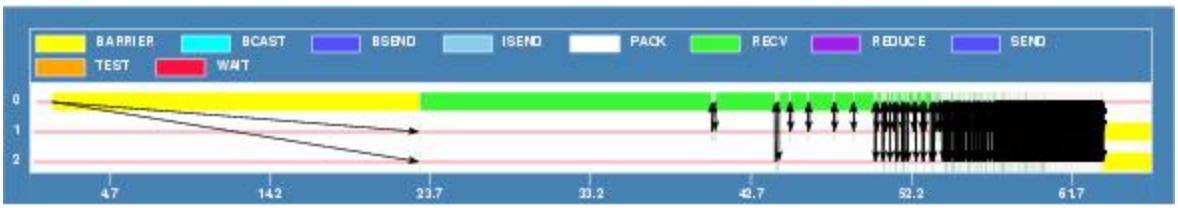


Figure 58 : Echanges de messages de l'algorithme à découpage dynamique

Sur ces chronogrammes (Figure 57 et Figure 58) trois phases apparaissent nettement :

- La première phase où les deux nœuds de calcul (identificateurs 1 et 2 sur les chronogrammes) lisent la scène. Le maître (identificateur 0), se contentant de la taille de l'image à produire, n'effectue pas cette lecture, il a déjà envoyé les premiers messages indiquant à chaque esclave le premier bloc d'image qu'il devra calculer.

- La seconde phase est la phase de calcul proprement dit : Elle commence lorsque les esclaves lisent les messages envoyés par le maître lors de la phase précédente. Elle est caractérisée par un nombre d'échanges de messages plus important correspondant aux assignations de blocs par le maître aux esclaves et aux renvois des résultats partiels au maître par les esclaves.

- La troisième et dernière phase ne fait pas tout à fait partie du calcul, elle correspond à l'écriture par le maître du résultat dans un fichier image ainsi que la production toujours par le maître d'un certain nombre de fichier de statistique concernant le déroulement du calcul.

La première phase est la même quel que soit l'algorithme utilisé, la lecture du fichier contenant la scène n'est pas concernée par le changement de méthode de découpage du

travail. La troisième phase ne concerne pas non plus le calcul proprement dit et le détail de son comportement n'offre aucun intérêt pour la présente analyse. Penchons-nous donc sur la seconde phase qui est celle sur laquelle porte mon travail.

Nous pouvons voir que l'algorithme à découpage régulier provoque un nombre de messages tel qu'il est impossible de les distinguer les uns des autres sur un chronogramme de cette dimension (Figure 57), tout ce que nous pouvons voir est un bloc noirci par les flèches schématisant les messages, ce qui correspond tout à fait au taux de communication prohibitif que nous avons constaté précédemment.

L'algorithme à découpage récursif (Figure 58) par contre commence par la manipulation de grands blocs ce qui se traduit sur le chronogramme par des messages très espacés les uns des autres. La taille des blocs diminuant au fur et à mesure du calcul, la fréquence des échanges de messages augmente. Elle n'atteindra la même valeur que celle de l'algorithme à découpage régulier qu'à la fin du calcul, sur le chronogramme on ne peut alors plus distinguer les messages les uns des autres. Nous pouvons aussi noter le décalage entre les premières réponses de chaque nœud de calcul (premiers messages en direction du maître) qui confirme, une fois de plus, l'irrégularité du lancer de rayons vis à vis de la charge de calcul puisque les premiers blocs assignés par le maître sont de même taille et que leurs temps de traitement diffèrent.

Le découpage récursif présente donc un réel avantage par rapport à un découpage régulier :

- Un découpage trop fin sera moins pénalisant avec l'algorithme récursif. L'utilisation de la plus petite taille de bloc (trop fine) n'est effective qu'à la fin du calcul contrairement à l'utilisation globale de celle-ci par l'algorithme à découpage régulier.

- Même lorsque le bloc de base est bien proportionné, le taux de communication obtenu grâce à l'algorithme récursif est meilleur que celui obtenu par l'algorithme régulier.

Ces remarques prennent d'autant plus d'importance qu'à ce jour, il n'a pas été trouvé de méthode permettant d'assurer préalablement au calcul et de manière automatique un dimensionnement optimal du bloc de base.

5.2. Troisième étape : Le regroupement de messages

Dans une première version des algorithmes, les nœuds pouvant être en train de calculer des blocs de tailles différentes et le maître ne sachant pas à l'avance quel esclave sera le prochain à envoyer un résultat, deux messages étaient utilisés pour la réception des blocs

d'images calculés à distance : le premier permettait d'identifier l'expéditeur et d'obtenir la taille du bloc d'image à recevoir par le maître, le second contenait les données elles-mêmes. Or, la fonction de réception de MPICH que nous utilisions ne nécessite en réalité que la connaissance d'une taille maximum de message à recevoir en vue d'allouer un tampon de taille suffisante pour accueillir les données. L'exploitation de cette facilité a permis, en dimensionnant systématiquement le tampon de réception par rapport au plus grand bloc en cours de calcul, de ne plus utiliser qu'un message pour envoyer les indications sur le bloc et les données résultats du calcul. Cette manipulation a eu pour conséquence directe d'augmenter la taille moyenne des messages et d'en réduire la quantité, concrètement le taux de communication par rapport au temps total est passé de 0,5% à environ 0,1%.

5.3. Quatrième étape : Le mode de communication

L'optimisation d'algorithmes distribués grâce à l'utilisation d'un mode de communication asynchrone est très répandue. Bien utilisées, il est vrai que les communications asynchrones peuvent rendre service mais elles possèdent cependant des inconvénients majeurs.

L'envoi de donnée n'étant plus subordonné à une action de réception explicite et synchrone, les modes de communication asynchrone nécessitent l'emploi de drapeaux, indiquant la réception, et de tampons, chargés de stocker les messages en transit. Ces outils additionnels réduisent l'apport potentiel d'un mode de communication asynchrone sur le mode de communication synchrone classique. En choisissant une taille minimale de bloc relativement réaliste (32x32 ou 64x64 par exemple) le pourcentage de communication au sein du traitement est très réduit et le gain à espérer de l'utilisation de communications asynchrones est faible. De plus, les communications asynchrones augmentent de manière significative la complexité de programmation et sont souvent source d'erreurs : la mise en place et la gestion des tampons et des drapeaux permettant d'assurer un fonctionnement correct est souvent fastidieuse.

Dans le cas de l'algorithme de lancer de rayon dont nous nous préoccupons ici, le taux de communication est d'environ 0,1%, l'utilisation de modes de communications plus compliqués d'emploi ne semble donc pas un besoin général de l'algorithme. Une des communications pourtant peut bénéficier d'anticipation, il s'agit de l'envoi des premières tâches. Au début de l'exécution, lorsque les esclaves en sont à lire les données du modèle le maître lui se contente de la connaissance de la taille de l'image à calculer puisque son travail de distribution sera basé uniquement sur le découpage du rectangle contenant l'image à calculer. Le maître est donc prêt à commencer l'étape de calcul avant les esclaves et le laps de temps séparant ceux-ci pourrait être mis à profit pour envoyer les premières tâches de façon non bloquante à tous les esclaves. Tous les messages indiquant

aux esclaves leur première tâche auront donc été envoyés avant que ceux-ci ne soit prêt à les lire, ils seront donc à leur disposition immédiatement après la lecture du modèle et permettront un « départ » synchrone de tous les esclaves contrairement à un envoi classique qui séquentialiserait celui-ci (Figure 59 et Figure 60).

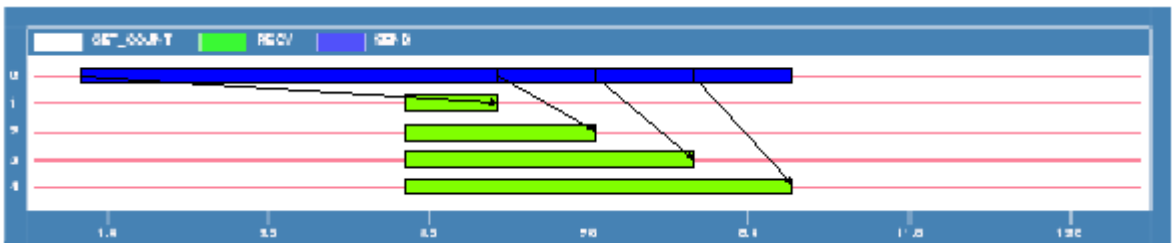


Figure 59 : Envoi des premières tâches avec les fonctions classiques

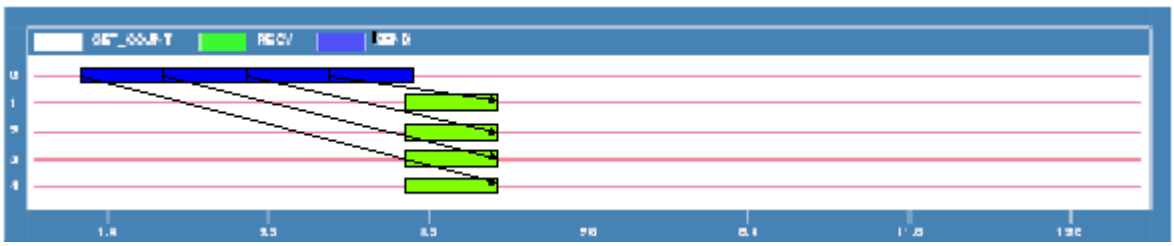


Figure 60 : Envoi des premières tâches avec les fonctions non bloquantes

6. Conclusion et comparaison avec des résultats extérieurs

Le lancer de rayons montre plusieurs caractéristiques compliquant sensiblement sa distribution : son irrégularité pose un vrai problème, le compromis entre volume de données permettant de bonnes optimisations et économie mémoire au prix d'un temps de calcul prohibitif n'est pas toujours évident, les méthodes classiques ne sont donc pas entièrement satisfaisantes.

L'introduction de la paresse dans le code séquentiel avant sa parallélisation dans le but d'obtenir une distribution automatique des données est une vraie nouveauté et son efficacité est très honorable. La distribution obtenue n'est pas exclusive et une certaine redondance de données existe, en particulier lorsque le nombre de nœuds de calcul augmente. Mais le gain est suffisant pour que plusieurs modèles qui ne pouvaient être calculés par un algorithme strict puisse l'être par l'algorithme paresseux (voir Annexe II : 4).

L'introduction d'un découpage récursif pour pallier l'absence d'information fiable quant à la taille idéale de blocs d'image, que l'on peut considérer ici comme la granularité de distribution, obtient de bon résultat aussi. Le gain sur le taux de communication est sensible et la communication est justement le principal point faible de l'architecture réseau d'ordinateurs que nous avons choisie comme cible. En réduisant l'impact d'une sous-évaluation de la taille de bloc (provenant généralement de la volonté d'assurer un bon équilibre de charge) elle permet de ne pas prendre le risque d'une surévaluation qui aboutirait à des performances médiocres (déséquilibre de charge important).

Même s'il est difficile de faire des comparaisons précises avec d'autres logiciels de lancer de rayons (les formats des scènes étant souvent incompatibles et certains paramètres de calcul étant généralement passé sous silence dans les références trouvées) les performances que nous obtenons sont souvent très supérieures à celles dont nous avons eu connaissance à travers les diverses publications lues sur le sujet. L'article [POI00] a été présenté par mes soins à la Winter School of Computer Graphics en 2000 et a suscité un intérêt très encourageant de la part des spécialistes de l'image de synthèse.

Pour finir j'aimerais préciser que la version séquentielle de notre algorithme est déjà intégrée dans plusieurs applications, certaines étant commerciales, et que plusieurs projets en vue d'intégrer la version distribuée de celui-ci sont actuellement à l'étude.

Chapitre VI : Bilan et Généralisation

Sommaire

1. PARESSE 147
1.1. Condition d'efficacité de la paresse *147*
1.2. Applicabilité de la paresse *148*
1.2.1. L'AMR distribué 148
1.2.2. Barnes-Hut 149
1.2.3. Conclusion 151
2. GRANULARITE ADAPTABLE 152
2.1. Avantage de la granularité adaptable *152*
2.2. Condition d'utilisation de la granularité adaptable *152*
3. REGROUPEMENT DE MESSAGES 156
4. MODE DE COMMUNICATION 157
5. SYSTEME ET PILOTES DE PERIPHERIQUES 157

1. Paresse

La paresse est un outil très général et l'impact de son utilisation sur un algorithme devant être distribué est présenté dans ce document.

L'utilisation d'un algorithme paresseux comme base séquentielle d'un algorithme distribué amène les avantages suivants :

- non-construction des structures de données inutiles au calcul
- amélioration de la localité
- distribution d'une partie des données

Mais l'utilisation d'algorithmes paresseux n'est pas une solution adaptée à tout type de problème.

Je vais essayer de décrire des conditions dans lesquelles la paresse peut très probablement améliorer le comportement et les performances d'un calcul de manière relativement théorique. J'essaierai ensuite de faire apparaître la similitude entre le travail effectué sur le lancer de rayon distribué et différents autres travaux dans le but de montrer la non spécificité de l'approche paresseuse.

1.1. Condition d'efficacité de la paresse

Je ne reviendrai pas ici sur les effets de la paresse dans le domaine séquentiel et m'intéresserai donc directement à l'aspect distribué.

Nous avons pu constater que l'utilisation d'un algorithme paresseux comme point de départ a permis de distribuer les données sans aucune directive explicite en ce sens lors de la programmation et a de plus contribué à une meilleure utilisation de ces données en en améliorant la localité. La paresse semble donc préférable à une recopie totale des données (sauf dans le cas où cette recopie est motivée par un besoin effectif de toutes les données par tous les nœuds de calcul et que l'espace mémoire permet celle-ci).

Nous avons aussi pu constater que cette efficacité avait des limites en particulier certaines données devaient être répliquées soit de par leur nature (dans le cas du lancer de rayons, la description brute de la scène, nécessaire à tout calcul, devait être présente dans la mémoire

de chaque nœud) soit de par leur implication dans des calculs différents attribués à des nœuds de calcul différents. La paresse offre donc une distribution des données moins complète que celle obtenue en utilisant la mémoire partagée virtuelle. Elle nécessite, par contre, moins d'accès distants que cette dernière ce qui est un atout majeur dans le cadre d'un traitement sur un réseau de machines.

L'efficacité de la paresse est donc étroitement liée à deux facteurs, la qualité de la séparation entre les différents sous-calculs du problème traité (et des données manipulées par ceux-ci) ainsi que la qualité de l'adaptation de l'algorithme de distribution à ces séparations naturelles.

L'implémentation efficace d'un algorithme paresseux distribué est, plus encore qu'un algorithme distribué classique, sensible à la granularité de découpage du problème (la taille des tâches), à ce découpage lui-même (le choix des tâches) et enfin à l'attribution de ces tâches en fonction des éventuelles connexions qui peuvent parfois subsister entre celles-ci.

Pour conclure, la paresse pourra être bénéfique à une application si celle-ci présente une utilisation des données réellement distribuée. En particulier la paresse montrera tout son potentiel si la distribution de l'utilisation des données est anarchique ou tout au moins non prévisible. La paresse déchargera le concepteur de l'application parallèle d'une distribution explicite des données souvent périlleuse pour les applications cibles au prix d'une attention plus importante encore qu'à l'accoutumé en ce qui concerne la qualité de la distribution du travail au sein des nœuds de calcul.

1.2. Applicabilité de la paresse

Le paragraphe précédent introduit déjà un certain domaine d'application de la paresse sur des caractéristiques particulières des algorithmes. Ce chapitre aborde ce même sujet par un exemple d'application dont nous pensons qu'elle pourrait bénéficier d'une méthode très proche de celle utilisée dans nos travaux, les maillages autoadaptatifs, puis par une comparaison entre l'arbre octal du lancer de rayon sur lequel nous avons travaillé et une construction hiérarchique issue d'un algorithme d'évaluation de performance sensé être représentatif d'un grand nombre d'applications.

1.2.1. L'AMR distribué

Les maillages autoadaptatifs sont un outil principalement utilisé dans des domaines de la physique comme par exemple la mécanique des fluides. Cet outil permet d'étudier un phénomène dans un univers suffisamment grand à un haut niveau de précision. Un tel traitement est souvent rendu impossible par le volume de données et de calculs

correspondant à l'étude totale du domaine au niveau de précision choisi. Pour obtenir une précision suffisante dans un domaine vaste, le niveau de précision n'est pas homogène au sein du domaine d'étude : il est fin dans les zones présentant les activités les plus intenses et grossier dans les zones de moindre intérêt. Ce maillage évoluera avec le phénomène étudié pour toujours être le plus efficace possible.

Le maillage adaptatif est une structure hiérarchique construite dynamiquement par évaluation d'un critère (gradient dans le cas étudié) déterminant si la précision est suffisante ou si un découpage plus fin doit être effectué. Ce mode de fonctionnement est extrêmement proche de celui de l'arbre octal sur lequel nous avons travaillé dans le cadre du lancer de rayon.

Contrairement à l'arbre octal, le maillage adaptatif n'offre pas de parties cachées. L'intérêt d'utiliser la paresse résidera donc uniquement dans la distribution adaptée des données selon les besoins locaux (et donc uniquement dans le cadre d'un algorithme distribué).

Une différence, lourde de conséquence, entre le lancer de rayon distribué et les algorithmes à maillages autoadaptatifs réside dans un jeu de données dynamique et des tâches interdépendantes.

Notre étude, très préliminaire, nous a amenés à la réflexion suivante : il semble qu'au sein d'un seul pas de temps nous puissions envisager que chaque nœud de calcul travaille de manière autonome. Ceci nous permettrait d'utiliser une approche similaire à celle utilisée dans l'étude du lancer de rayon distribué (à l'intérieur d'un pas de temps). Une synchronisation des calculs et une mise à jour des données locales en fonction de résultats distants restent cependant à traiter entre chaque pas de temps.

Une description plus détaillée de l'AMR et de la manière dont les outils proposés dans ce document pourrait lui être appliqués peut être trouvée dans [LEC01].

1.2.2. Barnes-Hut

L'algorithme de manipulation d'une structure de données arborescente de Barnes-Hut [BAR86] présente, lui aussi, de fortes similitudes avec celui de l'arbre octal présent dans le lancer de rayons sur lequel nous avons travaillé et des méthodes similaires pourraient sans doute s'appliquer à cet outil.

L'algorithme de Barnes-Hut propose d'effectuer des calculs d'interactions entre particules sur un grand nombre de particules en répartissant la charge grâce à un arbre octal. La décision de découper une feuille de cet arbre (devenant alors un nœud) est basée sur la proximité des particules qu'elle contient. Le calcul proprement dit sera effectué à un

niveau de précision d'autant plus fin que la hauteur du nœud (distance par rapport à la racine de l'arbre) sera importante.

La construction de l'arbre est habituellement effectuée lors d'une première phase, avant le calcul proprement dit.

Les similitudes entre cet algorithme et celui de l'arbre octal utilisé dans l'application de lancer de rayon sont nombreuses :

- La forme de la structure de données est un arbre dans les deux cas.
- La profondeur de l'arbre est déterminée par les données
- La construction de l'arbre constitue une étape préalable au calcul lui-même.

Les différences sont légères :

- Le critère d'arrêt du développement d'une branche de l'arbre n'est pas le même. C'est le nombre d'objets qui détermine s'il faut ou non développer un nœud dans le cas du lancer de rayon alors qu'il est question de proximité entre les objets pour l'algorithme de Barnes-Hut.
- Le calcul n'est pas exactement le même suivant la profondeur de l'arbre à laquelle les objets ont étés rangés dans l'algorithme de Barnes-Hut alors qu'il est indépendant de la profondeur pour le lancer de rayon. Cette différence est fondamentale mais n'a qu'un impact restreint sur l'implémentation elle même.
- Les données sont toutes utilisées (il n'y a pas d'équivalence aux « parties cachées » des scènes du lancer de rayons)

Comme pour l'AMR, seul l'aspect de distribution des données adaptée aux besoins locaux de la paresse sera exploitable. Dans un contexte distribué l'utilisation d'une construction paresseuse de l'arbre amènera le mécanisme de distribution automatique de la structure arborescente observé sur le lancer de rayon (chaque nœud ne construisant que les branches en rapport avec ses calculs locaux). La nécessité pour chaque nœud de disposer de l'ensemble des données de départ (non structurées) sera identique.

L'algorithme de Barnes-Hut, grand classique du calcul scientifique, semble donc pouvoir bénéficier de méthodes similaires à celles utilisées pour le lancer de rayons, la paresse en particulier. De nombreux algorithmes dérivés de celui de Barnes-Hut existent, il est probable qu'ils présentent eux-aussi des caractéristiques compatibles avec notre approche.

1.2.3. Conclusion

La paresse, dans le monde séquentiel aussi bien que parallèle, est un outil dont la cible principale sont les structures hiérarchiques importantes. Les méthodes dites multi-niveaux fournissent des structures parfaitement adaptées à ce type de traitement et semblent être en pleine expansion [ZUM02]. Les applications cibles ne manqueront donc pas aux méthodes présentées dans ce document.

De manière générale nous pouvons raisonnablement dire que la paresse peut-être utilisée comme outil de distribution des données si l'application cible répond aux trois critères suivants :

- L'application manipule un volume d'information important.
- La relation existant entre les calculs et les données d'entrée n'est pas prévisible ce qui compromet l'utilisation d'une répartition statique des données (dans le cas du lancer de rayon lorsque le calcul de rendu d'une partie de l'image est attribué à une machine, il n'est pas possible de prévoir quelles données du modèle seront nécessaires à celle-ci).
- Une répartition adaptée des données (tout au moins d'une grande partie de celles-ci) est susceptible d'exister même si l'on ne peut la déterminer à priori. Dit autrement, certaines parties du calcul peuvent être réalisées sans que la totalité des données ne soit utilisée.

L'efficacité de la paresse sera la plus grande si au moins l'un des deux critères suivants est vérifié :

- Les sous-tâches utilisent un petit volume de données, cela limite le risque de redondance importante de données au sein des différents nœuds.
- Une certaine cohérence entre certaines tâches existe et est correctement identifiée, favoriser l'attribution de ces tâches à une même machine augmentera alors l'efficacité générale de la paresse. La redondance sera évitée et les accès à une donnée déjà présente en local sont plus rapides. (Dans le cas du lancer de rayon distribué il existe une forte probabilité que le calcul de deux blocs d'image voisins nécessite des données communes. En particulier les rayons primaires de deux blocs d'image voisins ont une forte probabilité d'avoir leur premier point d'intersection avec la scène sur le même objet, nous favorisons donc une attribution des calculs de blocs voisins à un même nœud de calcul.)

Rappelons pour terminer que, même dans les cas où la paresse n'était pas nécessaire, son utilisation ne s'est pas montrée très coûteuse. Lors du calcul d'un modèle créé de manière à

ce que le calcul de chaque bloc de l'image est besoin de l'intégralité des données du modèle, celle-ci a montré un coût très faible par rapport à la solution plus radicale d'une recopie totale des données sur l'ensemble des nœuds. Cette dernière solution s'est avérée la plus performante dans ce cas précis, la solution de l'accès distant aux données offrant alors des performances mauvaises en raison d'un taux de communication très important.

2. Granularité adaptable

2.1. Avantage de la granularité adaptable

L'avantage principal de la granularité adaptable est d'offrir un compromis entre l'utilisation de blocs de petites tailles, menant à un bon équilibre de charge mais un taux de communication élevé, et l'utilisation de blocs de grande taille, permettant un taux de communication bas mais compromettant un équilibre de charge correct.

Nous avons pu vérifier dans le paragraphe Chapitre V : 5.1 que l'utilisation d'une granularité adaptable entraînait les effets suivants :

- Une réduction significative du taux de communication a été observée par rapport à une solution utilisant des petits blocs
- L'équilibre de charge est resté aussi bon que celui de la solution utilisant de petits blocs
- Une réduction de l'impact d'un choix inadapté de taille de bloc a été remarqué (en particulier, lors du choix d'une taille de bloc trop petite : la taille minimale n'étant effective qu'à la fin du traitement son impact sur le taux de communication global est moins important). Ce dernier point a d'autant plus d'importance que le choix de la taille de bloc à utiliser est encore empirique.

2.2. Condition d'utilisation de la granularité adaptable

Une granularité adaptable n'est envisageable que dans le contexte d'une distribution dynamique des tâches, il convient donc de commencer par définir les cas ou une telle technique est préférable à une répartition statique de la charge.

Une répartition statique de la charge est adaptée aux conditions suivantes :

- L'application distribuée est régulière
- La machine sur laquelle s'effectuera le calcul est dédiée à cet usage (au moins lors de l'exécution)
- La machine sur laquelle s'effectuera le calcul est homogène.

Si le dernier critère n'est pas vérifié, c'est à dire si la machine est hétérogène, les deux solutions sont envisageables :

- L'utilisation d'une répartition statique est celle qui offrira les meilleures performances mais elle présente l'inconvénient de devoir connaître précisément les performances de chaque machine afin d'effectuer une répartition de la charge efficace. Le résultat sera donc difficilement portable, à moins qu'une évaluation automatique de ces performances ne soit mise en place ce qui présente une difficulté et un coût non négligeable.
- L'utilisation d'une répartition dynamique sera vraisemblablement moins performante qu'une répartition statique idéale du fait des communications supplémentaires et de l'équilibre de charge qui, même très bon, ne sera pas parfait. Elle présente par contre l'avantage d'être portable et de ne pas nécessiter d'opération d'évaluation préalable des performances de chaque nœud de la machine cible.

Si la machine sur laquelle doit s'effectuer le calcul n'est pas dédiée à celui-ci, la performance disponible pour l'application peut varier au cours du temps au sein de chaque nœud de calcul en fonction des tâches « parasites » concurrentes à l'application. Une répartition dynamique de la charge de l'application est alors souhaitable afin de compenser ces variations.

Si l'application est irrégulière il n'est pas possible de découper celle-ci en tâches de charge connue préalablement au calcul. Une distribution dynamique des tâches est alors inévitable.

Une fois déterminé qu'une répartition dynamique de la charge est la solution la plus adaptée à l'application à distribuer il faut choisir la granularité de découpage du calcul. Le problème du choix de la granularité optimale lors de la distribution d'un algorithme est un problème très général. Lorsque le choix de la granularité dépend de critères divergents, comme c'est le cas dans les algorithmes irréguliers, l'utilisation d'une granularité récursivement décroissante semble tout à fait indiquée.

Cette technique va permettre de respecter l'équilibre de charge en permettant à tous les nœuds de calcul de terminer leur traitement de manière très rapprochée car la granularité utilisée en fin de traitement est fine. Mais cette même technique va permettre de ne pas tomber pour autant dans un rapport communication/calcul trop défavorable puisque la granularité utilisée en début de traitement est importante.

La granularité adaptable nécessite le choix de différentes valeurs qui sont la granularité de début de traitement, de fin de traitement, et le facteur de réduction ou le nombre de niveaux. Le choix de ces valeurs n'est pas trivial et reste souvent empirique. Afin d'obtenir un équilibre de charge correct, il faut que la valeur de la granularité choisie (par estimation, essais successifs, statistique…) pour le début du traitement vérifie la relation suivante : la charge de calcul de chacune des tâches de cette granularité doit être au plus égale à la charge de calcul dégagée par le reste du traitement divisée par le nombre de nœud de calcul restant (tous sauf celui auquel sera affecté la tâche) ce qui équivaut à dire qu'elle ne doit pas dépasser un *n*-ième de la charge de calcul globale (*n* étant le nombre de nœuds de calcul). Dans le cas d'un réseau de machines hétérogènes il faut au minimum pondérer la formule précédente par la puissance de chaque machine, et envisager le cas le pire : tâche la plus lourde attribuée au nœud de calcul le moins puissant.

En résumé, il semble que la granularité adaptable puisse être choisie dès qu'une répartition dynamique de la charge est envisagée, elle présente de nombreux avantages et sa programmation, quoique plus compliquée que celle d'un découpage régulier classique, reste très abordable. Une solution étant d'effectuer un découpage régulier, selon la taille de bloc minimale, et d'attribuer plusieurs blocs à la fois (en nombre décroissant) à chaque nouvelle assignation.

<table>
<tr><td colspan="3">Pour une répartition sur 10 nœuds de calcul identiques, voici les trois cas de figure possibles (en fonction de la granularité choisie pour les premières tâches) :</td></tr>
<tr><td>La tâche la plus importante réprésente moins de 10% de la charge de calcul totale.</td><td>Un nœud traite une tâche représentant 10% de la charge totale de calcul.</td><td>Une tâche représente plus de 10% de la charge de calcul globale.</td></tr>
<tr><td>Le nœud auquel la tâche la plus importante sera attribuée pourra traiter d'autres tâches (de moindre charge) après traitement de celle-ci. En effet, les neuf autres nœuds ne peuvent traiter plus de 90% de la charge dans le même laps de temps nécessaire au dixième pour traiter moins de 10% de celle-ci.</td><td>Il reste donc 90% de la charge de calcul à partager parmi les neuf nœuds restants ce qui permet encore un équilibre de charge correct.

Sur une application régulière, l'équilibre idéal est d'ailleurs obtenu en sélectionnant 10 tâches représentant chacune 10% de la charge totale</td><td>Il reste moins de 90% de la charge de calcul à répartir sur les neuf nœuds restants, il est alors impossible d'obtenir un équilibre de charge correct :
• soit l'équilibre n'est pas respecté au sein des neuf nœuds s'occupant des 90% de la charge il n'est donc pas correct globalement,
• soit l'équilibre est parfait au sein de ces nœuds et le dixième nœud terminera après les neuf autres puisque sa charge de calcul est supérieure dès le départ, l'équilibre de charge global ne sera pas correct non plus.</td></tr>
<tr><td>Cas favorable, le choix de la granularité de départ était correct</td><td>Cas limite, le choix de la granularité de départ était optimal mais risqué</td><td>Cas défavorable, la granularité de départ a été surévaluée</td></tr>
<tr><td colspan="3">L'estimation de la granularité de départ doit donc être faite de manière à ce qu'aucune tâche ne représente plus de 10 % de la charge de calcul.

Pour notre part nous nous sommes basés sur les résultats couramment obtenus lors de nos tests et nous laissant une certaine marge (l'image de départ a été découpée en un nombre de blocs égal au nombre de nœuds de calcul au carré)</td></tr>
</table>

Exemple 9 : Conséquence de l'estimation de la granularité de départ

Le choix du facteur de réduction (ou du nombre de niveaux) doit permettre de continuer à assurer que la proportion entre les tâches reste correcte tout au long du traitement. Une réduction trop rapide de la granularité diminue l'efficacité de la méthode en faisant augmenter le rapport communication/calcul au-delà de ce qui est nécessaire. Une réduction trop lente entraîne le risque qu'à un moment du calcul la proportion entre une tâche et le

reste du traitement à effectuer ne respecte plus le seuil détaillé précédemment et que l'équilibre correct de la charge ne soit plus possible.

La granularité de fin de traitement doit être suffisamment fine pour permettre un équilibre de charge correct. Choisir une granularité excessivement fine permettra certainement d'obtenir une terminaison bien synchrone des nœuds de calcul mais entraînera en même temps un surcoût très important en communication. Un choix de granularité de fin de traitement adapté dépend donc à la fois de la performance des communications (paramètre lié à la machine cible) et de la densité et la répartition de la charge de calcul (paramètre lié au jeu de données). Une manière d'estimer cette valeur est de se fixer un seuil de déséquilibre de charge acceptable (par exemple 1%) et de trouver par un jeu de test la plus grande valeur susceptible de ne pas dépasser celui-ci. Cette méthode ne permet bien sûr pas d'assurer qu'il n'y aura jamais plus de 1% de déséquilibre (cela dépend trop des données dans le cadre des algorithmes irréguliers) mais donne une idée relativement fidèle de ce à quoi il faut s'attendre. Par ailleurs, un mécanisme relativement simple de comparaison entre le temps de service (c'est à dire le temps s'écoulant entre l'envoi du résultat du calcul précédent et l'obtention d'une nouvelle tâche par un esclave) et le temps de traitement d'un bloc permet de déterminer si le bloc était « trop petit » (temps de service supérieur au temps de traitement). Un taux anormalement élevé de blocs trop petits pourrait déclencher l'arrêt du mécanisme de découpage, afin d'éviter d'aggraver encore la situation (action facile mais tardive, visiblement la taille de bloc est déjà trop petite), voire un retour à une taille supérieure de bloc (opération plus compliquée).

Le choix des valeurs définissant le comportement de la granularité dans notre méthode n'est pas très satisfaisant puisqu'il reste très empirique et ne présente pas une fiabilité parfaite cependant des valeurs sensées suffisent pour obtenir de bons résultats car si leur choix est presque aussi problématique que celui d'une granularité optimale unique leur impact sur les performances n'est pas aussi grand (sauf dans le cas de choix très erronés). La validité de la technique d'une granularité récursivement décroissante n'est donc pas remise en cause par cet inconvénient même si celui-ci est réel.

3. Regroupement de messages

Le regroupement de message possède une efficacité remarquable, puisque même dans des conditions de départ offrant une bonne performance il a permis de réduire très sensiblement le taux de communication (de 0,5% à 0,1%). Son utilisation a permis d'améliorer les performances (dans des proportions modestes mais les performances de départ n'offraient pas une marge de progression très importante) et surtout d'améliorer la scalabilité. En effet, si le nombre de machines augmente, si leur puissance de calcul est

plus importante (proportionnellement aux performances du réseau, comme c'est la tendance actuelle) alors le taux de communication augmentera : posséder une marge maximale sur ce taux est donc primordial à la scalabilité de l'algorithme.

Généraliser le regroupement de messages ne signifie certainement pas grand chose, toutefois l'agrégation de messages de valeurs sémantiques différentes ainsi que la sérialisation de structures de données afin d'optimiser leur envoi en un seul message sont des manipulations (classiques) dont l'utilisation est souvent possible.

4. Mode de communication

L'utilisation de modes de communication asynchrones présente souvent l'inconvénient de compliquer la programmation, leur efficacité n'en est pas moins réelle et le choix de leur utilisation constitue la dernière étape d'optimisation de l'aspect communication d'un algorithme distribué (dans la démarche proposée Chapitre IV : 5).

Dans le cas de notre application distribuée de lancer de rayons, l'algorithme de distribution est écrit de façon à fortement limiter les communications. Le recours à un mode de communication plus élaboré, mais plus complexe, ne semble par conséquent pas nécessaire (hormis pour le tout premier envoi). Quelques expériences menées par nos soins ont montré la validité du raisonnement tenu dans ce paragraphe : des gains négligeables ont été observés (et même des surcoûts, certainement dus à une gestion moins bien optimisée des consultations de drapeaux) alors que le code s'alourdissait passablement.

Toutefois il serait bien présomptueux de généraliser ces résultats et nous préférerons donc nous en tenir à ce simple constat.

5. Système et pilotes de périphériques

Aucune investigation approfondie n'a été effectuée sur ce sujet dans le cadre de cette thèse Cependant l'évolution des configurations tout au long des tests a montré un impact significatif sur les performances et, sans présumer d'une portée générale de ces observations, un rapide bilan de celles-ci ne m'a pas semblé totalement inintéressant. Les phénomènes observés sont les suivants :

- Les évolutions du système opératoire lui-même, ou l'utilisation de versions sensiblement différentes, n'ont montré aucun impact mesurable sur les performances.

- L'évolution des pilotes de périphérique, d'un pilote générique (provenant dans notre cas de la distribution Linux) vers un pilote constructeur dédié à la carte réseau, a amené une amélioration significative des temps de calcul (de l'ordre de 15%). Cette amélioration a été observée sur la version de notre algorithme à distribution dynamique et découpage statique de la charge de travail. Sur l'algorithme à découpage dynamique de la charge de travail, l'amélioration a été moins franche (de l'ordre de 5%), ce qui semble cohérent avec la moindre sollicitation du réseau de la part de l'algorithme.

- La mise à jour du compilateur a, elle aussi, eu un impact très important sur les performances. Le compilateur (gcc) fournit avec la distribution Linux utilisée a été remplacé par une version plus récente de ce même compilateur dont les options d'optimisation prenait en charge le processeur précis présent sur la machine cible. Les temps de calcul observés ont été parfois divisé par deux ou trois tout algorithme confondu.

- Plusieurs versions de la bibliothèque de communication (mpich) ont été testées sans qu'un changement de performance n'ait pu être détecté. Un prototype de bibliothèque de communication différent (VIA) a montré une amélioration très nette des performances sur des tests purement réseau, malheureusement nous n'avons pas été en mesure d'effectuer des tests sur l'application distribuée de lancer de rayon pour mesurer l'impact de cette évolution dans un cas d'utilisation réaliste. Il semble probable que le comportement aurait évolué de la même manière qu'avec le changement de pilote de périphérique.

L'optimisation du contexte logiciel de travail permet d'améliorer les performances de manière notable, quoique inégale en fonction des sous-parties de ce contexte. Cependant un algorithme dynamique permet de réduire l'impact d'un environnement moins performant.

Chapitre VII : Conclusion et perspectives

Sommaire

1. CONCLUSION 161
2. PERSPECTIVES : VERS PLUS DE FLEXIBILITE 161
2.1. Amnésie 161
2.2. Interruptibilité 163

1. Conclusion

Lors de cette étude nous avons observé que les architectures de type réseau de machines ou grappe de processeurs souffraient d'un déséquilibre entre les fréquences d'horloge des processeurs et les latences réseau encore plus important que celui observé sur les supercalculateurs des grandes enseignes. De plus, l'évolution, passée et présente, du matériel n'a de cesse d'aggraver cette tendance.

Une démarche de distribution prenant en compte cette caractéristique a été proposée, offrant un travail sur le fond (par le choix d'un algorithme et d'un mode de distribution générant le moins de communication possible) avant de travailler sur la forme (regroupement de messages et mode de communication asynchrones).

Deux outils originaux adaptés à la distribution d'applications irrégulières ont été proposés et étudiés : la paresse, comme moyen de distribuer les données, et la granularité adaptable, permettant de concilier l'équilibrage de la charge de calcul et un taux de communication réduit.

Des travaux sur l'automatisation du choix des paramètres de la granularité adaptable ainsi que l'implantation dans d'autres applications de celle-ci et de la paresse sont souhaitables afin de mieux en cerner le champ d'application, voire d'étendre celui-ci.

2. Perspectives : vers plus de flexibilité

2.1. Amnésie

2.1.1. Ce qui est exprimé

Certaines données ont une « durée de vie » relativement courte, d'autres sont rarement utilisées et d'autres le sont en permanence. L'utilisation de l'amnésie permet de ne conserver sous une forme utilisable que les données appartenant aux dernières catégories voire à la dernière catégorie uniquement.

L'idée d'amnésie vient de l'observation que toute donnée disponible à un moment quelconque du déroulement d'un algorithme est issue d'une instruction au moins. Pour peu que cette instruction réponde au critère d'idempotence, il est à tout moment possible de retrouver la donnée par l'appel de la même instruction. Toute donnée dont la source répond au critère précité peut donc être remplacée en mémoire par un pointeur vers l'instruction susceptible de la reconstruire. L'espace mémoire d'une telle donnée peut donc être libéré pour une autre utilisation sachant qu'en cas de demande de réutilisation de cette donnée nous serons capables de la régénérer. L'exemple le plus évident est une opération de lecture d'une information sur un périphérique de stockage. L'amnésie peut être vue comme un système de swap intelligent car géré par l'application elle-même.

2.1.2. Comment reconnaître un algorithme susceptible de bénéficier de l'amnésie ?

Tout comme la paresse, l'amnésie ne trouve son utilité que dans des algorithmes dont la consommation mémoire est jugée excessive.

De plus, pour que l'amnésie soit pleinement justifiée il faut que l'algorithme cible soit tel que la réutilisation ou non d'un volume important de données soit un problème indécidable. Les algorithmes répondant à ce critère rendent inefficace l'utilisation d'un ramasse miette classique puisque celui-ci ne pourra que conclure que les données sont toutes référençables et imposera donc la conservation de l'ensemble de celles-ci.

D'un point de vue système, une tâche amnésique offre l'avantage de pouvoir à tout moment, sur ordre du système d'exploitation, réduire sa consommation mémoire. Cette aptitude permet de gérer l'arrivée imprévue d'une nouvelle tâche au sein d'un système du point de vue des ressources.

2.1.3. Intérêt de l'amnésie

L'amnésie, comme la paresse, est un outil de réduction de la consommation mémoire qui ne trouve donc un intérêt que lorsque la mémoire est une ressource critique.

Contrairement à un ramasse miette, l'amnésie ne se contente pas de détruire les données jugées inutiles mais conserve un moyen de les reconstruire si elles devenaient à nouveau nécessaires. La mémoire s'en trouve donc soulagée au prix de temps d'exécution lorsqu'une donnée qui avait été supprimée par le système d'amnésie doit être reconstruite. L'efficacité dépend alors de la bonne adéquation de la stratégie de suppression (LRU, Random, LRR …).

La mise en place de l'amnésie est proche de celle d'un système de swap, son avantage est de ne pas nécessiter l'opération de sauvegarde sur disque des données à enlever de la mémoire.

2.2. Interruptibilité

2.2.1. Ce qui est exprimé

Un algorithme interruptible est un algorithme qui maintient un résultat incomplet disponible à tout moment, à partir d'un temps d'initialisation t_0 et jusqu'à l'obtention du résultat définitif. En outre, un algorithme interruptible associe au résultat actuellement disponible une estimation de la qualité de celui-ci.

Pour pouvoir maintenir un résultat incomplet, ce dernier est exprimé sous la forme d'une liste de valeurs. Chaque élément de cette liste voit son domaine de solution étendu par l'ajout d'un élément, généralement noté « $\perp$ », exprimant l'indétermination. L'état initial du résultat est une liste dont tout les éléments ont la valeur $\perp$ (toutes les valeurs sont indéterminées) tandis que l'état final du résultat est une liste dans laquelle aucun des éléments n'a pour valeur $\perp$ (aucune valeur n'est indéterminée). On entrevoit alors une estimation possible de la qualité du résultat par le nombre d'indéterminées subsistant dans la solution, mais cette qualité peut être encore plus finement estimée en pondérant l'intérêt de chaque élément pour la solution ou bien encore en fixant des règles plus restrictives indiquant l'intérêt de tel ou tel élément en fonction de la détermination de tel ou tels autres.

Pour qu'un algorithme interruptible soit utilisable il lui est imposé d'avoir une qualité croissante tout au long de son exécution, une manière d'aboutir à cette propriété est d'imposer une règle d'écriture unique sur les éléments de la solution.

Un autre aspect de l'interruptibilité est la possibilité de reprise d'un calcul à partir du résultat incomplet auquel avait aboutit l'algorithme avant son arrêt et non à partir des données d'entrée initiales.

2.2.2. Comment reconnaître un algorithme susceptible de bénéficier de l'interruptibilité ?

Tout algorithme dont le résultat peut-être représenté comme une liste d'objets peut être exprimé sous la forme d'un algorithme interruptible. En effet le critère de qualité croissante peut être obtenu artificiellement en utilisant comme résultat courant le meilleur résultat obtenu jusqu'ici.

2.2.3. Intérêts de l'interruptibilité

Plusieurs domaines peuvent trouver un intérêt à exprimer un algorithme sous forme interruptible.

Dans le cadre du temps réel, un algorithme interruptible offre la capacité de s'adapter à toute échéance postérieure au temps d'initialisation de l'algorithme. Si un résultat incomplet peut être utilisé par l'environnement alors l'interruptibilité a un intérêt pour cet algorithme.

Dans le cadre des algorithmes infinis, où la qualité du résultat détermine à elle seule la terminaison de l'algorithme, l'interruptibilité est un moyen d'expression extrêmement réaliste. L'échéance n'est plus ici commandée par le temps mais par la qualité du résultat actuel qui est une donnée présente dans la notion même d'interruptibilité.

Dans le cadre système, la possibilité de reprise depuis le dernier résultat offerte par l'interruptibilité permet l'arrêt d'un algorithme pour des raisons de priorité, de manque de ressources ou autre sans perte majeure sur le travail effectué par celui-ci. Cette possibilité va bien au-delà de l'habituel passage dans un état bloqué puisque l'intégralité des ressources attribuées à l'algorithme sont alors récupérables.

Chapitre VIII : Références

Les références figurent ici selon leur ordre d'apparition dans le document, une liste classée par ordre alphabétique de ces mêmes références se trouve dans le chapitre Bibliographie.

[BEO] Projet BEOWULF, http://www.beowulf.org/

[ESS] *NASA's Computational Technologies project*, http://ess.gsfc.nasa.gov/

[STE95] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, Charles V. Packer, *BEOWULF : A Parallel Workstation For Scientific Computation,* International Conference on Parallel Processing, 1995.

[MPI95] *MPI: A Message Passing Interface Standard*. Juin 1995.

[GAR97] Jean-Marie Garcia, Thierry Monteil et P. Guyaux, *Projet LANDA : Local Area Network for Distributed Applications,* Journées d'Etude sur les Logiciels pour le traitement de l'Image, du Signal et l'Automatique (ELISA'97), 14 mai 1997.

[ALINKA] ALINKA *Technologie*, *http://www.alinka.com*

[LDP] *The Linux Documentation Project,* http://sunsite.unc.edu/mdw/linux.html

[FSF] *La Free Software Foundation,* http://okki666.free.fr/newbie/linux056.htm

[TOP02] Top 500 des machines les plus puissantes dans le monde, http://www.top500.org/list/2002/11/

[COM99] *Linux gushes savings for oil giant,* COMPUTERWORLD, 3 mai 1999

[IBM] Site Internet de commercialisation des eServeurs d'IBM, http://www-1.ibm.com/servers/eserver/clusters/

[TAN99] Andrew Tanenbaum, *Réseaux*, Dunod, 3ème édition, 1999 – la 4ème édition de l'ouvrage original *Computer Networks* est en cours de traduction, sa disponibilité est annoncée pour fin mai 2003

[LAW79] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage*, ACM Trans. Math. Soft., 5 (1979), pp. 308-323.

[DON90] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 18-28.

[CAR99] Eddy Caron, Olivier Cozette, Dominique Lazure, Gil Utard : *Virtual Memory Management in Data Parallel Applications*, High Performance Computing and Networking (HPCN'99), avril 1999.

[BLA97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, *ScaLAPACK Users' Guide,* mai 1997.

[MUK95] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers et Joel Salz, *Efficient support for irregular applications on Distributed-memory machines,* 5ème ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP), juillet 1995.

[AGE95] T. Agerwala, J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias et M.Snir, *SP2 system architecture,* IBM System Journal, Volume 34 Numéro 2, 1995.

[CRAY] http://www.cray.com/products/systems/t3e/

[FOR78] S. Fortune et J. Wyllie, *Parallelism in Random Access Machines,* Proceddings of the 10th Annual Symposium on Theory of Computing, pages 114-118, 1978.

[GIB96] Phillip B.Gibbons, Yossi Matias, Vijaya Ramachandran, The Queue Read Queue Write PRAM Model : Accounting for Contention in Parallel Algorithm, SIAM Journal of Computing, Décembre 1996

[VAL90] Leslie G Valiant, *A Bridging Model for Parallel Computation,* Communications of the ACM , 33(8):103, Août 1990.

[CUL93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian et Thorsten von Eicken, *LogP : Towards a Realistic Model of Parallel Computation,* 1993.

[RAI92] Sanjay Raina, *Virtual Shared Memory : A Survey of Techniques and Systems,* Université de Bristol, 1992.

[IFT99] Liviu Iftode et Jaswinder Pal Singh, *Shared Virtual Memory: Progress and Challenges,* IEEE Special Issue on *Distributed* Shared Memory, Vol 87. No. 3, 1999.

[LIH86] K. Li et P. Hudak, *Memory Coherence in Shared Virtual Memory Systems,* Proceedings du 5ème ACM Annual Symposium On Principles of Distributed Computing (PODC'86), pages 229 à 239, 1986.

[LI86] Kaï Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors,* Thèse de doctorat de l'université de Yale département informatique, 1986.

[ADV95] Sarita V. Adve, Kourosh Gharachorloo, *Shared Memory Consistency Models : A tutorial,* Western Research Laboratory, Research Report 95/7, septembre 1995.

[BER91] B.N. Bershad et M.J. Zekauskas, *Midway : Shared Memory Programming with Entry Consistency for Distributed Memory Multiprocessors,* Technical Report CMU-CS-91-170, Carnegie Mellon University, 1991.

[SIN96] Jaswinder Pal Singh, Liviu Iftode et Kaï Li, *Scope Consistency : a bridge between release consistency and entry consistency,* Technical report TR-509-96, Princeton, NJ, 1996.

[CAR91] J.B. Carter, J.K. Bennet et W.Zwaenepoel, *Implementation and performance of Munin*, Proceedings du 13ème Symposium on Operating Systems Principles, pages 106-117, 1994.

[KEL92] Pete Keleher, Alan Cox et Willy Zwaenepoel, *Lazy Release Consistency for Software Distributed Shared Memory, Université Rice, 1992.*

[IFT98b] Liviu Iftode, *Home-based Shared Virtual Memory,* Thèse de doctorat de l'université de Princeton, 1998.

[ZHO96] Yuanyuan Zhou, Liviu Iftode et Kaï Li, *Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems,* Proceedings du second Symposium on Operating Systems Design and Implementation, 1996.

[SAM97] Rudrajit Samanta, Angelos Bilas, Liviu Iftode et Jaswinder Pal Singh, *Home-based SVM Protocols for SMP clusters : Design and Performance,* Universités de Princeton et de Rugers, 1997.

[IFT98a] L. Iftode, M. Blumrich, C. Dubnicki, D.L. Oppenheimer, J.P. Singh et K. Li, *Shared Virtual Memory with Automatic Update support,* Technical Report TR-575-98, Princeton, NJ, 1998.

[SCA96] D.J. Scales, K. Gharachorloo et C.A. Thekkath, *Shasta: A low overhead, software-only* approach *for supporting fine-grain shared memory,* 7th International Conference on Architectural Support for Programming *Languages and Operating systems*, Octobre 1996.

[IFT96a] Liviu Iftode, Jaswinder Pal Singh et Kaï Li, *Understanding Application Performance on Shared Virtual Memory Systems*, Proceedings du 23ème Annual Symposium on Computer Architecture, 1996.

[IFT96b] Liviu Iftode, Jaswinder Pal Singh et Kaï Li, *Irregular Applications under Software Shared Memory, Technical Report TR-514-96, Université de Princeton, 1996.*

[BIL98] Angelos Bilas, Dongming Jiang, Yuanyuan Zhou et Jaswinder Pal Singh, *Limits to the performance of Software Shared Memory : A Layered Approach,* Proceedings du cinquième International Symposium on High Performance Computer Architecture (HPCA), 1998.

[REI95] Erik Reinhard, *Scheduling and Data Management for Parallel Ray Tracing,* Thèse de doctorat, university of East Anglia, 1995.

[CHA93] Alan Chalmers, David Stuttard et Derek J. Paddon, *Data Management for Parallel Ray Tracing of Complex Images, Proceedings of the* International Conference on Computer Graphics (ICCG), pages 149 à 162, 1993.

[PAR95] Manish Parashar et James C.Browne, *Distributed Dynamic Data-Structures for Parallel Adaptive Mesh-Refinement,* Proceedings of the International Conference for High Performance Computing, pages 22 à 27, 1995.

[PAR98] Manish Parashar et James C.Browne, *Integrated Data-Management for Computational Steering, Hawaï International Conference on System Science (HICSS 31), 1998.*

[KRA97] Sacha Krakowiak, *Avancées récentes en systèmes répartis et leur impact sur les SGBD,* Journées « Bases de Données Avancées », 1997.

[AUG00] Philippe Augerat et Yves Denneulin, *Une ferme de 200 PC,* id-imag, CNRS, 2000.

[GLU02] Olivier Glück, *Optimisation de la bibliothèque de communication MPI pour machines parallèles de type „grappe de PCs“ sur une primitive d'écriture distante,* Thèse de doctorat soutenue le 12 juillet 2002 à l'Université Paris VI.

[ONG00] Hong Ong et A. Farell, *Performance Comparison of LAM/MPI, MPICH, end MVICH on a Linux Cluster connected by a Gigabit Ethernet Network,* Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, Octobre 2000.

[CHA98] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, Mithuna Thottethodi, *Recursive Array Layouts and Fast Parallel Matrix Multiplication,* 1998.

[SKE01] K.-A. Skevik, T. Plagemann, V. Goebel, P. Halvorsen, *Evaluation of a Zero-copy Protocol Implementation*, Proceedings of the 27th Euromicro Conference - Multimedia and Telecommunications Track (MTT'2001), Septembre 2001

[THA95] Moti N. Thadani, Yousef A. Khalidi, *An efficient Zero-Copy I/O Framework for Unix®,* Sun Microsystems Laboratories, Inc., Technical Report 95-32, mai 1995

[LAU93] J.Launchbury, *A natural semantics for lazy evaluation,* Principles of Programming Languages 1993 (PoPL'93).

[PLA02] Tomas Plachetka, *Perfect Load Balancing for Demand-Driven Ray Tracing,* Europar 2002, Lecture Notes in Computer Science, B. Monien, R. Feldman (eds.), Springer Verlag, 2400, 2002.

[MAR99] Francescomaria Marino et Earl E. Swartzlander, *Parallel Implementation of Multidimensional Transforms without Interprocessor Communication,* IEEE Transaction on Computers, Vol 48, Numéro 9, septembre 1999.

[COO65] J.W. Cooley et J.W. Tukey, *An Algorithm for the* Machine *Calculation of Complex Fourier Series,* Math. Comput. 19, 297-301, 1965.

[DEP98] Benoît Depont, *Etude d'une nouvelle algorithmique pour le lancer de rayons en électromagnétisme,* SUPAERO/CERT/META-CS, août 1998.

[GLA89] Andrew Glassner, *An Introduction to Ray Tracing*, Palo Alto C.A., Academic Press, 1989.

[SCH97] Robert Schutt, *Ray Tracing 101*, Colgate University Department of Natural Sciences, Ray Tracing on Parallel Microprocessors, 1997.

[GLA84] Andrew Glassner, *Space Subdivision for fast Ray Tracing,* IEEE Computer Graphics and Applications, Volume 4, N.10, pp 15-22, Octobre 1984.

[AMA87] John Amanatides, Andrew Woo, *A Fast Voxel Traversal Algorithm for Ray Tracing*, Proceedings of Eurographics 1987.

[BER98] Sébastien Bermes, *Les arbres octaux paresseux : une méthode dynamique de subdivision spatiale pour le lancer de rayons.* Thèse Supaero, soutenue en 1998.

[DIP84] M.A.Z Dippé et J. Swensen, *An adaptative subdivision algorithm and parallel architecture for realistic image synthesis.* Christiansen ed., Computer Graphics SIGGRAPH'84 proceedings, Vol.18, pp.149-158, 1984.

[KOB88] H. Kobobayashi, S. Nishimura, H. Kubota, T. Nakamura et Y.Shigei, *Load Balancing strategies for a parallel raytracing system based on constant subdivision,* The Visual Computer (4), 1988.

[PAN98] Igor-Sunday Panzic, Michel Roethlisberger, Nadia Magnetat-Thalmann, *Parallel Ray Tracing on the IBM SP2 and CRAY T3D.* Miralab Copyright Information, 1998

[BAD94] D. Badouel, K. Bouatouch, T. Priol, *Distributing data and control for Ray Tracing,* Computer Graphics and Applications, pp.69-77, 1994.

[POV91] Persistence of Vision : http://www.povray.org/ ,1991

[HEN96] J. Hennessy and D Patterson, *Computer Architecture : A quantitative approach* Morgan Kaufman Publishers 1996

[HSU96] Tsan-Sheng Hsu, Joseph C. Lee, Dian Rae Lopez, William A. Royce. *Task Allocation on a Network of Processors*. 1996.

[SPD] http://www.acm.org/tog/resources/SPD/overview.html

[POI00] Olivier Poitou, Bernard Lecussan et Sébastien Bermes, *Laziness, a way to improve distributed computation of the ray tracing algorithm,* WSCG 2000, Prague.

[LEC01] Bernard Lecussan et Olivier Poitou, *Fundamental parallel program issues to improve parallel computations,* Invited talk Ala2001 (Algèbre linéaire et arithmétique : calcul numérique, symbolique et parallèle) 28-31 Mai 2001, Université des Sciences, Rabat Maroc.

[BAR86] J.E. Barnes and P. Hut, *A hierarchical O(n.log(n)) force calculation algorithm*. Nature, 324(4):446-449, December 1986.

[ZUM02] Gerhard Zumberger, *Parallel Adaptive Numerical Multilevel Algorithms,* Tutorial, Europar 2002.

Bibliographie

Ce chapitre reprend les références du chapitre précédent, classées cette fois par ordre alphabétique et selon leur type.

1. Articles et journaux

[ADV95] Sarita V. Adve, Kourosh Gharachorloo, *Shared Memory Consistency Models : A tutorial,* Western Research Laboratory, Research Report 95/7, septembre 1995.

[AGE95] T. Agerwala, J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias et M.Snir, *SP2 system architecture,* IBM System Journal, Volume 34 Numéro 2, 1995.

[AMA87] John Amanatides, Andrew Woo, *A Fast Voxel Traversal Algorithm for Ray Tracing*, Proceedings of Eurographics 1987.

[AUG00] Philippe Augerat et Yves Denneulin, *Une ferme de 200 PC,* id-imag, CNRS, 2000.

[BAD94] D. Badouel, K. Bouatouch, T. Priol, *Distributing data and control for Ray Tracing,* Computer Graphics and Applications, pp.69-77, 1994.

[BAR86] J.E. Barnes and P. Hut, *A hierarchical O(n.log(n)) force calculation algorithm*. Nature, 324(4):446-449, December 1986.

[BER91] B.N. Bershad et M.J. Zekauskas, *Midway : Shared Memory Programming with Entry Consistency for Distributed Memory Multiprocessors,* Technical Report CMU-CS-91-170, Carnegie Mellon University, 1991.

[BER98] Sébastien Bermes, *Les arbres octaux paresseux : une méthode dynamique de subdivision spatiale pour le lancer de rayons.* Thèse Supaero, soutenue en 1998.

[BIL98] Angelos Bilas, Dongming Jiang, Yuanyuan Zhou et Jaswinder Pal Singh, *Limits to the performance of Software Shared Memory : A Layered Approach,* Proceedings du cinquième International Symposium on High Performance Computer Architecture (HPCA), 1998.

[BLA97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, *ScaLAPACK Users' Guide,* 1 May 1997.

[CAR99] Eddy Caron, Olivier Cozette, Dominique Lazure, Gil Utard : *Virtual Memory Management in Data Parallel Applications*, High Performance Computing and Networking (HPCN'99), avril 1999.

[CAR91] J.B. Carter, J.K. Bennet et W.Zwaenepoel, *Implementation and performance of Munin*, Proceedings du 13ème Symposium on Operating Systems Principles, pages 106-117, 1994.

[CHA93] Alan Chalmers, David Stuttard et Derek J. Paddon, *Data Management for Parallel Ray Tracing of Complex Images, Proceedings of the* International Conference on Computer Graphics (ICCG), pages 149 à 162, 1993.

[CHA98] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, Mithuna Thottethodi, *Recursive Array Layouts and Fast Parallel Matrix Multiplication,* 1998.

[COM99] *Linux gushes savings for oil giant,* COMPUTERWORLD, 3 mai 1999

[COO65] J.W. Cooley et J.W. Tukey, *An Algorithm for the* Machine *Calculation of Complex Fourier Series,* Math. Comput. 19, 297-301, 1965.

[CUL93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian et Thorsten von Eicken, *LogP : Towards a Realistic Model of Parallel Computation,* 1993.

[DEP98] Benoît Depont, *Etude d'une nouvelle algorithmique pour le lancer de rayons en électromagnétisme,* SUPAERO/CERT/META-CS, août 1998.

[DIP84] M.A.Z Dippé et J. Swensen, *An adaptative subdivision algorithm and parallel architecture for realistic image synthesis.* Christiansen ed., Computer Graphics SIGGRAPH'84 proceedings, Vol.18, pp.149-158, 1984.

[DON90] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 18-28.

[FOR78] S. Fortune et J. Wyllie, *Parallelism in Random Access Machines,* Proceddings of the 10th Annual Symposium on Theory of Computing, pages 114-118, 1978.

[GAR97] Jean-Marie Garcia, Thierry Monteil et P. Guyaux, *Projet LANDA : Local Area Network for Distributed Applications,* Journées d'Etude sur les Logiciels pour le traitement de l'Image, du Signal et l'Automatique (ELISA'97), 14 mai 1997.

[GIB96] Phillip B.Gibbons, Yossi Matias, Vijaya Ramachandran, The Queue Read Queue Write PRAM Model : Accounting for Contention in Parallel Algorithm, SIAM Journal of Computing, Décembre 1996

[GLA84] Andrew Glassner, *Space Subdivision for fast Ray Tracing,* IEEE Computer Graphics and Applications, Volume 4, N.10, pp 15-22, Octobre 1984.

[GLA89] Andrew Glassner, *An Introduction to Ray Tracing*, Palo Alto C.A., Academic Press, 1989.

[GLU02] Olivier Glück, *Optimisation de la bibliothèque de communication MPI pour machines parallèles de type „grappe de PCs" sur une primitive d'écriture distante,* Thèse de doctorat soutenue le 12 juillet 2002 à l'Université Paris VI.

[HSU96] Tsan-Sheng Hsu, Joseph C. Lee, Dian Rae Lopez, William A. Royce. *Task Allocation on a Network of Processors*. 1996.

[IFT96a] Liviu Iftode, Jaswinder Pal Singh et Kaï Li, *Understanding Application Performance on Shared Virtual Memory Systems*, Proceedings du 23ème Annual Symposium on Computer Architecture, 1996.

[IFT96b] Liviu Iftode, Jaswinder Pal Singh et Kaï Li, *Irregular Applications under Software Shared Memory, Technical Report TR-514-96, Université de Princeton, 1996.*

[IFT98a] L. Iftode, M. Blumrich, C. Dubnicki, D.L. Oppenheimer, J.P. Singh et K. Li, *Shared Virtual Memory with Automatic Update support,* Technical Report TR-575-98, Princeton, NJ, 1998.

[IFT98b] Liviu Iftode, *Home-based Shared Virtual Memory,* Thèse de doctorat de l'université de Princeton, 1998.

[IFT99] Liviu Iftode et Jaswinder Pal Singh, *Shared Virtual Memory: Progress and Challenges,* IEEE Special Issue on *Distributed* Shared Memory, Vol 87. No. 3, 1999.

[KEL92] Pete Keleher, Alan Cox et Willy Zwaenepoel, *Lazy Release Consistency for Software Distributed Shared Memory, Université Rice, 1992.*

[KOB88] H. Kobobayashi, S. Nishimura, H. Kubota, T. Nakamura et Y.Shigei, *Load Balancing strategies for a parallel raytracing system based on constant subdivision,* The Visual Computer (4), 1988.

[KRA97] Sacha Krakowiak, *Avancées récentes en systèmes répartis et leur impact sur les SGBD,* Journées « Bases de Données Avancées », 1997.

[LAU93] J.Launchbury, *A natural semantics for lazy evaluation,* Principles of Programming Languages 1993 (PoPL'93).

[LAW79] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage*, ACM Trans. Math. Soft., 5 (1979), pp. 308-323.

[LEC01] Bernard Lecussan et Olivier Poitou, *Fundamental parallel program issues to improve parallel computations,* Invited talk Ala2001 (Algèbre linéaire et arithmétique : calcul numérique, symbolique et parallèle) 28-31 Mai 2001, Université des Sciences, Rabat Maroc.

[LI86] Kaï Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors,* Thèse de doctorat de l'université de Yale département informatique, 1986.

[LIH86] K. Li et P. Hudak, *Memory Coherence in Shared Virtual Memory Systems,* Proceedings du 5ème ACM Annual Symposium On Principles of Distributed Computing (PODC'86), pages 229 à 239, 1986.

[MAR99] Francescomaria Marino et Earl E. Swartzlander, *Parallel Implementation of Multidimensional Transforms without Interprocessor Communication,* IEEE Transaction on Computers, Vol 48, Numéro 9, septembre 1999.

[MUK95] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers et Joel Salz, *Efficient support for irregular applications on Distributed-memory machines,* 5ème ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP), juillet 1995.

[ONG00] Hong Ong et A. Farell, *Performance Comparison of LAM/MPI, MPICH, end MVICH on a Linux Cluster connected by a Gigabit Ethernet Network,* Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, Octobre 2000.

[PAN98] Igor-Sunday Panzic, Michel Roethlisberger, Nadia Magnetat-Thalmann, *Parallel Ray Tracing on the IBM SP2 and CRAY T3D.* Miralab Copyright Information, 1998

[PAR95] Manish Parashar et James C.Browne, *Distributed Dynamic Data-Structures for Parallel Adaptive Mesh-Refinement,* Proceedings of the International Conference for High Performance Computing, pages 22 à 27, 1995.

[PAR98] Manish Parashar et James C.Browne, *Integrated Data-Management for Computational Steering, Hawaï International Conference on System Science (HICSS 31), 1998.*

[PLA02] Tomas Plachetka, *Perfect Load Balancing for Demand-Driven Ray Tracing,* Europar 2002, Lecture Notes in Computer Science, B. Monien, R. Feldman (eds.), Springer Verlag, 2400, 2002.

[POI00] Olivier Poitou, Bernard Lecussan et Sébastien Bermes, *Laziness, a way to improve distributed computation of the ray tracing algorithm,* WSCG 2000, Prague.

[RAI92] Sanjay Raina, *Virtual Shared Memory : A Survey of Techniques and Systems,* Université de Bristol, 1992.

[REI95] Erik Reinhard, *Scheduling and Data Management for Parallel Ray Tracing,* Thèse de doctorat, university of East Anglia, 1995.

[SAM97] Rudrajit Samanta, Angelos Bilas, Liviu Iftode et Jaswinder Pal Singh, *Home-based SVM Protocols for SMP clusters : Design and Performance,* Universités de Princeton et de Rugers, 1997.

[SCA96] D.J. Scales, K. Gharachorloo et C.A. Thekkath, *Shasta: A low overhead, software-only* approach *for supporting fine-grain shared memory,* 7th International Conference on Architectural Support for Programming *Languages and Operating systems*, Octobre 1996.

[SCH97] Robert Schutt, *Ray Tracing 101*, Colgate University Department of Natural Sciences, Ray Tracing on Parallel Microprocessors, 1997.

[SIN96] Jaswinder Pal Singh, Liviu Iftode et Kaï Li, *Scope Consistency : a bridge between release consistency and entry consistency,* Technical report TR-509-96, Princeton, NJ, 1996.

[SKE01] K.-A. Skevik, T. Plagemann, V. Goebel, P. Halvorsen, *Evaluation of a Zero-copy Protocol Implementation*, Proceedings of the 27th Euromicro Conference - Multimedia and Telecommunications Track (MTT'2001), Septembre 2001

[STE95] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, Charles V. Packer, *BEOWULF : A Parallel Workstation For Scientific Computation,* International Conference on Parallel Processing, 1995.

[THA95] Moti N. Thadani, Yousef A. Khalidi, *An efficient Zero-Copy I/O Framework for Unix®,* Sun Microsystems Laboratories, Inc., Technical Report 95-32, mai 1995

[VAL90] Leslie G Valiant, *A Bridging Model for Parallel Computation,* Communications of the ACM , 33(8):103, Août 1990.

[WOO95] Steven Cameron Woo, Moriyoshi *Ohara*, Evan Torrie, Jaswinder Pal Singh et Anoop Gupka, *The SPLASH-2 Programs : Characterization and Methodological Considerations,* 22ème Annual International Symposium on Computer Architecture, pages 24-36, Juin 1995.

[ZHO96] Yuanyuan Zhou, Liviu Iftode et Kaï Li, *Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems,* Proceedings du second Symposium on Operating Systems Design and Implementation, 1996.

[ZUM02] *Gerhard* Zumberger, *Parallel Adaptive Numerical Multilevel Algorithms,* Tutorial, Europar 2002.

2. Ouvrages complets

[HEN96] J. Hennessy and D Patterson, *Computer Architecture : A quantitative approach* Morgan Kaufman Publishers 1996

[MPI95] *MPI: A Message Passing Interface Standard*. Juin 1995.

[TAN99] Andrew Tanenbaum, *Réseaux*, Dunod, 3ème édition, 1999 – la 4ème édition de l'ouvrage original *Computer Networks* est en cours de traduction, sa disponibilité est annoncée pour fin mai 2003

3. Sites Internet

[ALINKA] ALINKA *Technologie*, *http://www.alinka.com*

[BEO] Projet BEOWULF, http://www.beowulf.org/

[CRAY] http://www.cray.com/products/systems/t3e/

[ESS] *NASA's Computational Technologies project*, http://ess.gsfc.nasa.gov/

[FSF] *La Free Software Foundation,* http://okki666.free.fr/newbie/linux056.htm

[IBM] Site Internet de commercialisation des eServeurs d'IBM, http://www-1.ibm.com/servers/eserver/clusters/

[LDP] *The Linux Documentation Project,* http://sunsite.unc.edu/mdw/linux.html

[POV91] Persistence of Vision : http://www.povray.org/ ,1991

[TOP02] Top 500 des machines les plus puissantes dans le monde, http://www.top500.org/list/2002/11/

Annexe I. MACHINES UTILISEES POUR LES MESURES

Sommaire

1. LE RESEAU DE STATIONS SUN 183
2. LA GRAPPE DU DTIM 184
3. LA GRAPPE DE SUPAERO 185
4. LA GRAPPE GRAPHIQUE 186

1. Le réseau de stations SUN

Figure 61 : Une station Sun Ultra10

Matériel	**Machine parallèle**	**Présentation**		Réseau d'entreprise
		Nombre de nœuds		Machine testée jusqu'à 24
	Nœud	**Type**		Monoprocesseur
		CPU	**Désignation**	Sun Ultra-10
			Fréquence	450 MHz
			Cache	
		Bus	**Largeur**	
			Fréquence	
		Mémoire	**Quantité**	256 Mo
			Fréquence	
		Interface réseau		Ethernet
Logiciel	**Système**			Solaris 8
	Bibliothèque de communication			
	Compilateur			Sun et gcc 2.98
	Pilotes réseau			Propriétaires

2. La grappe du DTIM

Figure 62 : La grappe du DTIM

Matériel	**Machine parallèle**	**Présentation**		Grappe (armoire)
		Nombre de nœuds		8
	Nœud	**Type**		Biprocesseur (smp)
		CPU	**Désignation**	Intel Pentium IV
			Fréquence	933 MHz
			Cache	256 Ko
		Bus	**Largeur**	64 bits
			Fréquence	133 MHz
		Mémoire	**Quantité**	1 Go
			Fréquence	133 MHz
		Interface réseau		Ethernet Intel Pro 10/100 Myrinet SAN 2000
Logiciel	**Système**			Linux RedHat 6.1
	Bibliothèque de communication			Mpich 1.2.2
	Compilateur			Gcc 2.98
	Pilotes réseau			Propriétaires

3. La grappe de Supaero

Figure 63 : La grappe de Supaero

Matériel	**Machine parallèle**	**Présentation**		Grappe
		Nombre de nœuds		8
	Nœud	**Type**		Biprocesseur (smp)
		CPU	**Désignation**	Intel Pentium III (Katmaï)
			Fréquence	550 MHz
			Cache	512 Ko
		Bus	**Largeur**	32 bits
			Fréquence	100 MHz
		Mémoire	**Quantité**	512 Mo pour 4 nœuds 256 Mo pour les 4 autres
			Fréquence	100 MHz
		Interface réseau		Fast Ethernet 3Com (3C905B)
Logiciel	**Système**			Linux RedHat 6.2 (Kernel 2.2.14-5.0 smp)
	Bibliothèques			mpich-1.2.2
	Compilateur			gcc 2.91.66
	Pilotes réseau			m-via v1.1 du NERSC

4. La grappe graphique

Figure 64 : La grappe graphique

Matériel	**Machine parallèle**	**Présentation**		Grappe (armoire)
		Nombre de nœuds		6
	Nœud	**Type**		Biprocesseur (smp)
		CPU	**Désignation**	AMD Athlon
			Fréquence	1,6 GHz
			Cache	256 Ko
		Bus	**Largeur**	32 bits
			Fréquence	266 MHz
		Mémoire	**Quantité**	1 Go
			Fréquence	266 MHz
		Interface réseau		Fast Ethernet 3Com (3C905C-TX) GigaEthernet Broadcom Corporation NetXtreme BCM5701
Logiciel	**Système**			Linux RedHat 7.1 (Kernel 2.4.17 smp)
	Bibliothèques			
	Compilateur			gcc 2.96
	Pilotes réseau			Standard linux pour carte 3Com et propriétaire pour la carte Broadcom

Annexe II. FICHES DE MESURES

Sommaire

1. UNE SCENE DE PETITE TAILLE : THE TEAPOT 190
2. UNE SCENE DE TAILLE MOYENNE : THE MOUNT 192
3. UNE SCENE VOLUMINEUSE : THE TREE 194
4. UNE SCENE DEPASSANT LA CAPACITE MEMOIRE : THE GEARS 196
5. VERS LE LANCER DE RAYON EN TEMPS REEL : THE TETRA 198

Les scènes dont les résultats sont donnés ici proviennent du SPD [REF], une suite très connue dans le monde du lancer de rayon. Ces résultats faciliteront une éventuelle comparaison avec d'autres logiciels de lancer de rayons. Pour information, la ligne de commande utilisée est la suivante :

scene –s *complexité* –t >scene.nff

complexité représentant le facteur de complexité utilisé (il sera indiqué pour chaque scène)

Tous les modèles issus de cette suite ont été créés au format NFF (le plus standard) et les images ont été calculées à une taille de 1024x1024 pixels. Le format NFF offre une grande généricité et c'est là son principal atout, un autre point fort de ce dernier réside dans une syntaxe simple. Les défauts majeurs d'un tel format sont :

- La notation ne dispose d'aucune optimisation, par exemple les sommets de tous les triangles sont explicitement donnés même s'ils sont communs avec les triangles voisins.
- Aucune organisation des triangles dans le fichier n'est prévue ce qui réduit à néant les possibilités de lecture partielle.
- Les données sont stockées au format ASCII ce qui implique un volume plus important et des routines de lectures moins efficaces.

Le fichier modèle NFF est donc très volumineux et sa lecture peu efficace.

L'ensemble des mesures de cette annexe concerne des tests effectués sur la grappe du DTIM (cf Annexe I - 2)

Les trois premières scènes retenues montrent le comportement de notre programme sur une scène de petite, moyenne puis grande taille. Deux scènes supplémentaires ont été ajoutées dans le but de montrer l'intérêt de la paresse pour le calcul d'une scène très volumineuse pour la première et la capacité de distribution d'un calcul très court dans l'optique d'une évolution vers le lancer de rayon en temps réel pour la seconde.

1. Une scène de petite taille : The Teapot

La scène de la cafetière a été crée avec un facteur de complexité de 12 ce qui correspond à un modèle constitué de 9 500 facettes.

Les résultats des mesures sont les suivants :

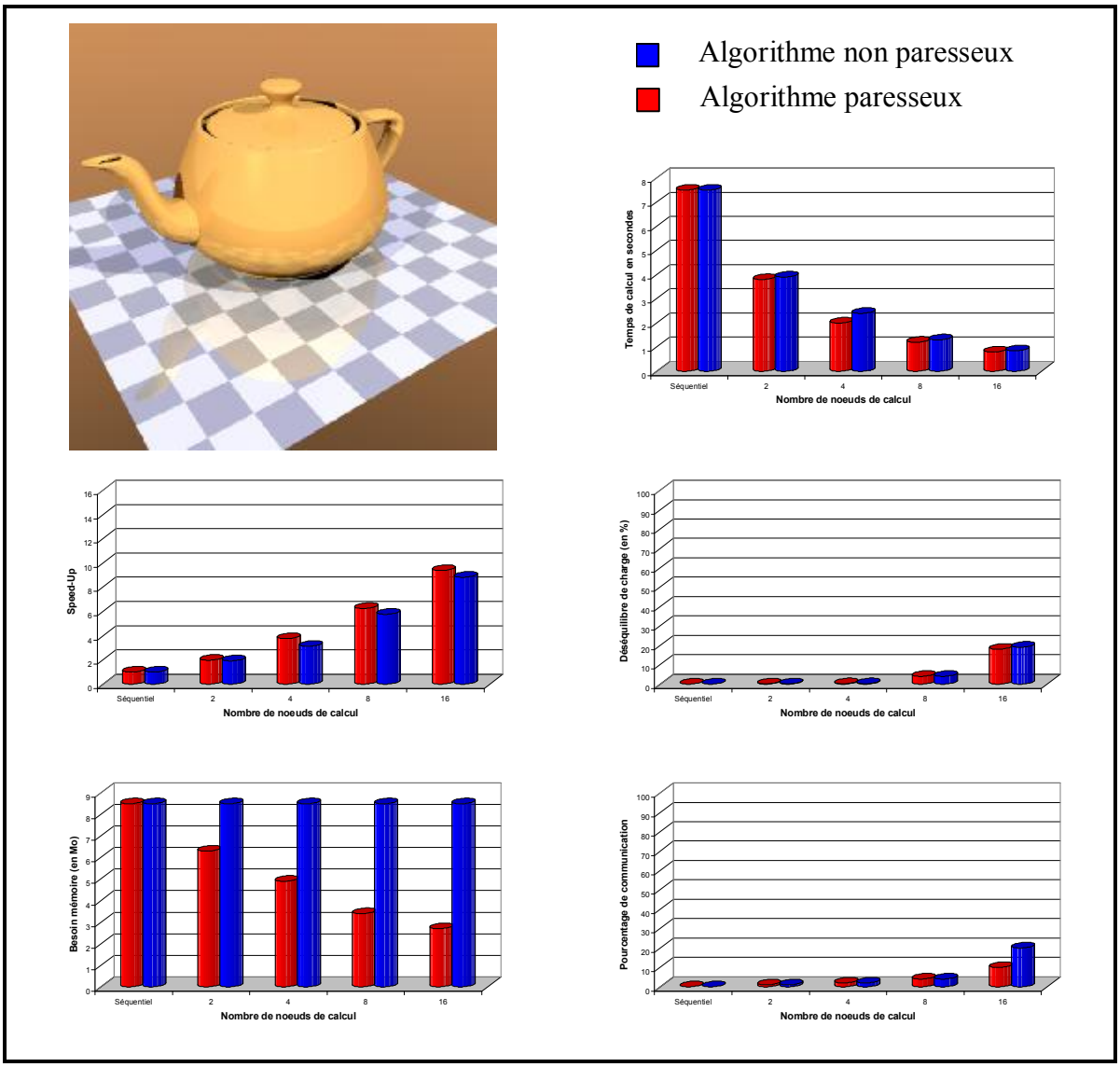


Figure 65 : Fiche de test de la scène Teapot 12

Ces courbes n'appellent pas de commentaires particuliers, le speed-up est acceptable compte tenu du temps de calcul relativement court en séquentiel. On peut remarquer que le déséquilibre de charge ainsi que le pourcentage de communication restent faibles avec une augmentation plus nette sur 16 nœuds. La cause principale de ce pic est l'utilisation de huit machines biprocesseurs et non de 16 machines monoprocesseurs, les processeurs d'une même machine devant donc se partager une interface réseau unique.

2. Une scène de taille moyenne : The Mount

Le modèle de la montagne a été produit avec un facteur de complexité de 8 ce qui correspond à un modèle de 130 000 facettes.

Les résultats des mesures sont les suivants :

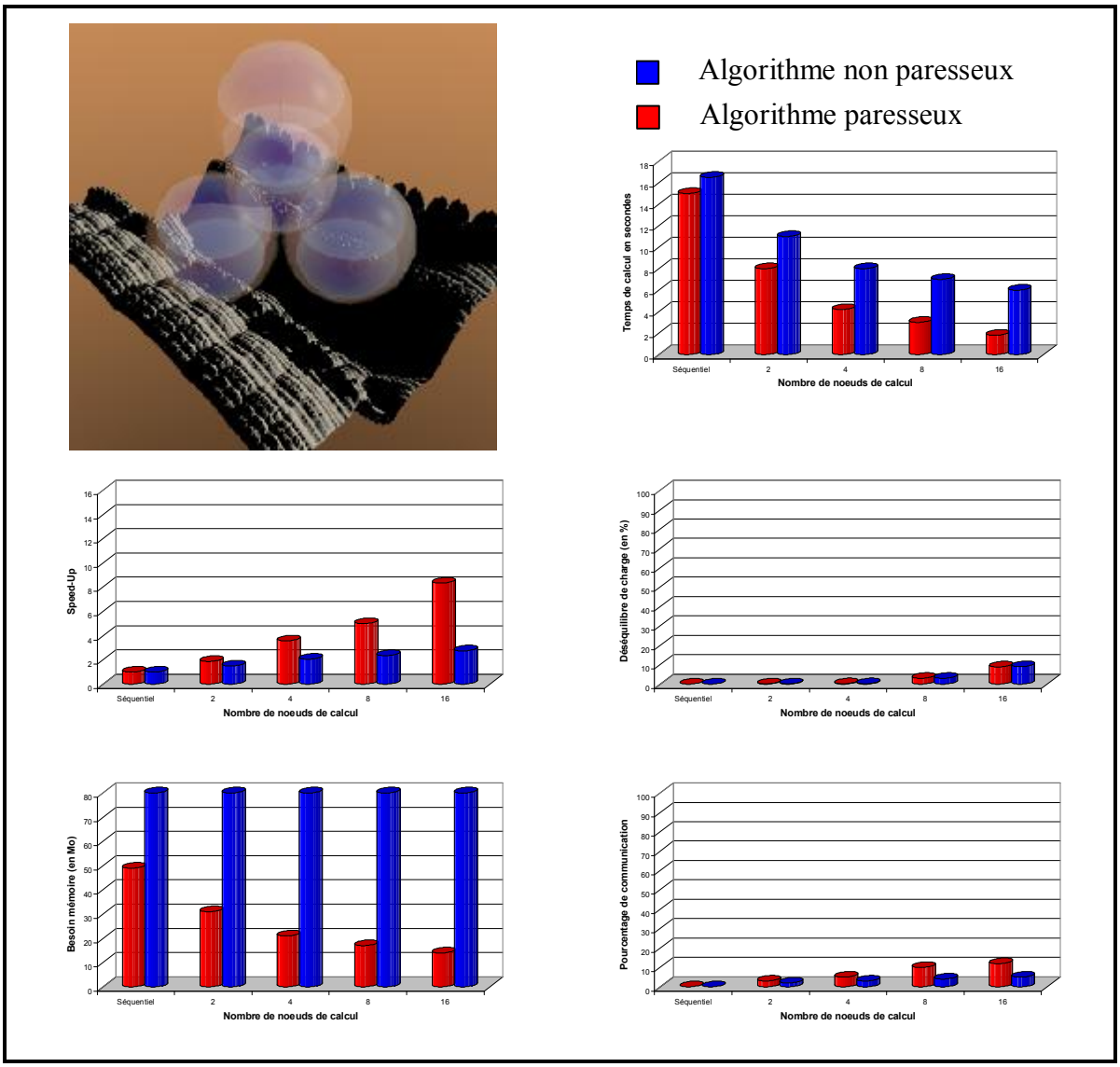


Figure 66 : Fiche de test de la scène Mount 8

L'importante économie mémoire réalisée par la version paresseuse permet d'obtenir un speed-up très supérieur à celui de la version classique. Ce modèle comporte de nombreux objets cachés ce qui permet à la paresse de réduire la consommation mémoire de l'exécution séquentielle également.

3. Une scène volumineuse : The Tree

La scène de l'arbre a été produite avec un facteur de complexité de 10 ce qui correspond à un modèle comprenant 425 000 facettes.

Les résultats des mesures sont les suivants :

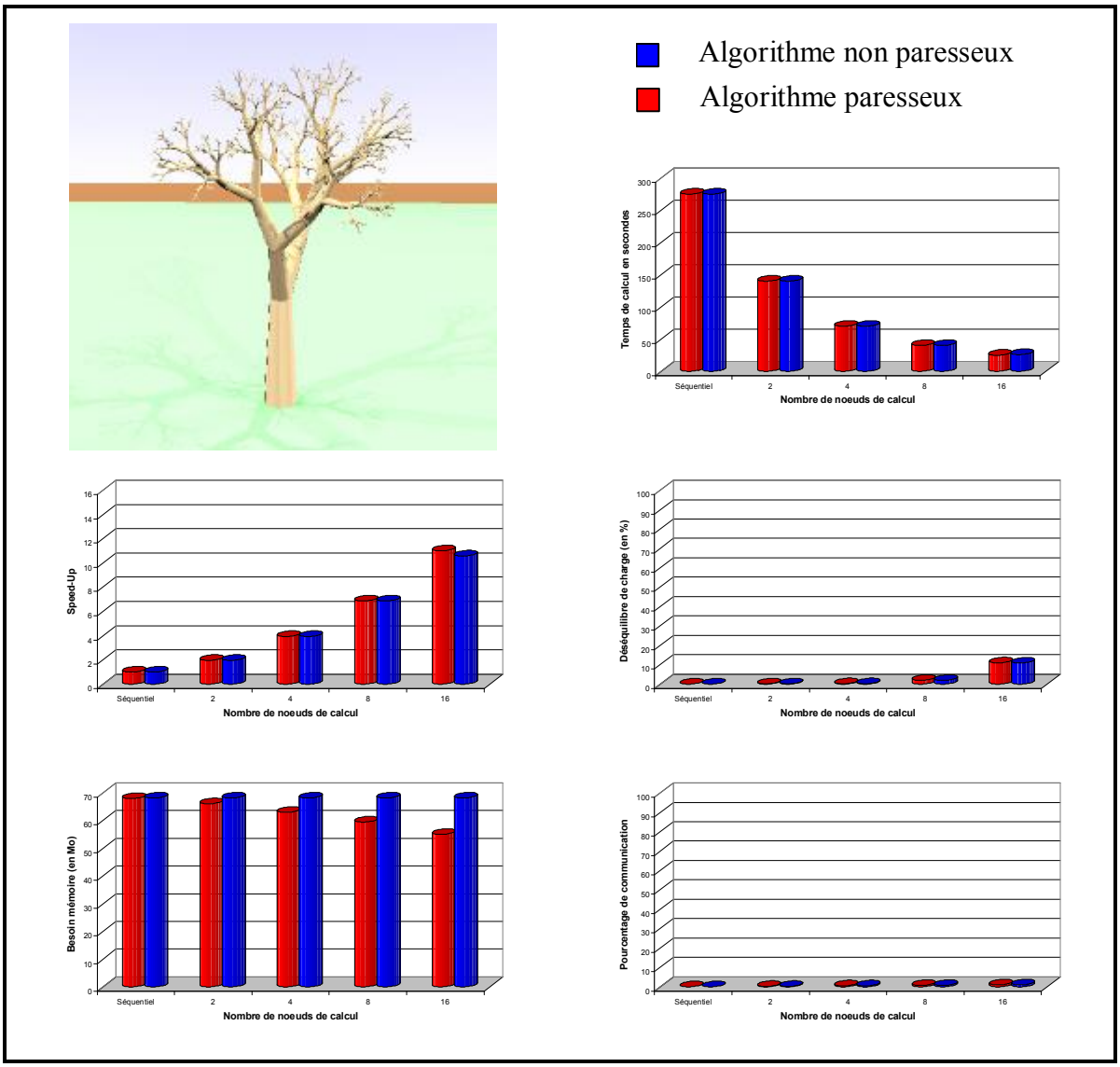


Figure 67 : Fiche de test de la scène Tree 10

On remarque ici le faible taux de communication observé sur toutes les configurations de machines. Le speed-up sur 16 nœuds est malheureusement limité par un déséquilibre de charge un peu trop important.

4. Une scène dépassant la capacité mémoire : The Gears

La scène des engrenages a été produite avec un facteur de complexité de 15 ce qui correspond à un modèle d'environ 2 millions de facettes (1 930 502 facettes dans cet exemple).

Les résultats des mesures sont les suivants :

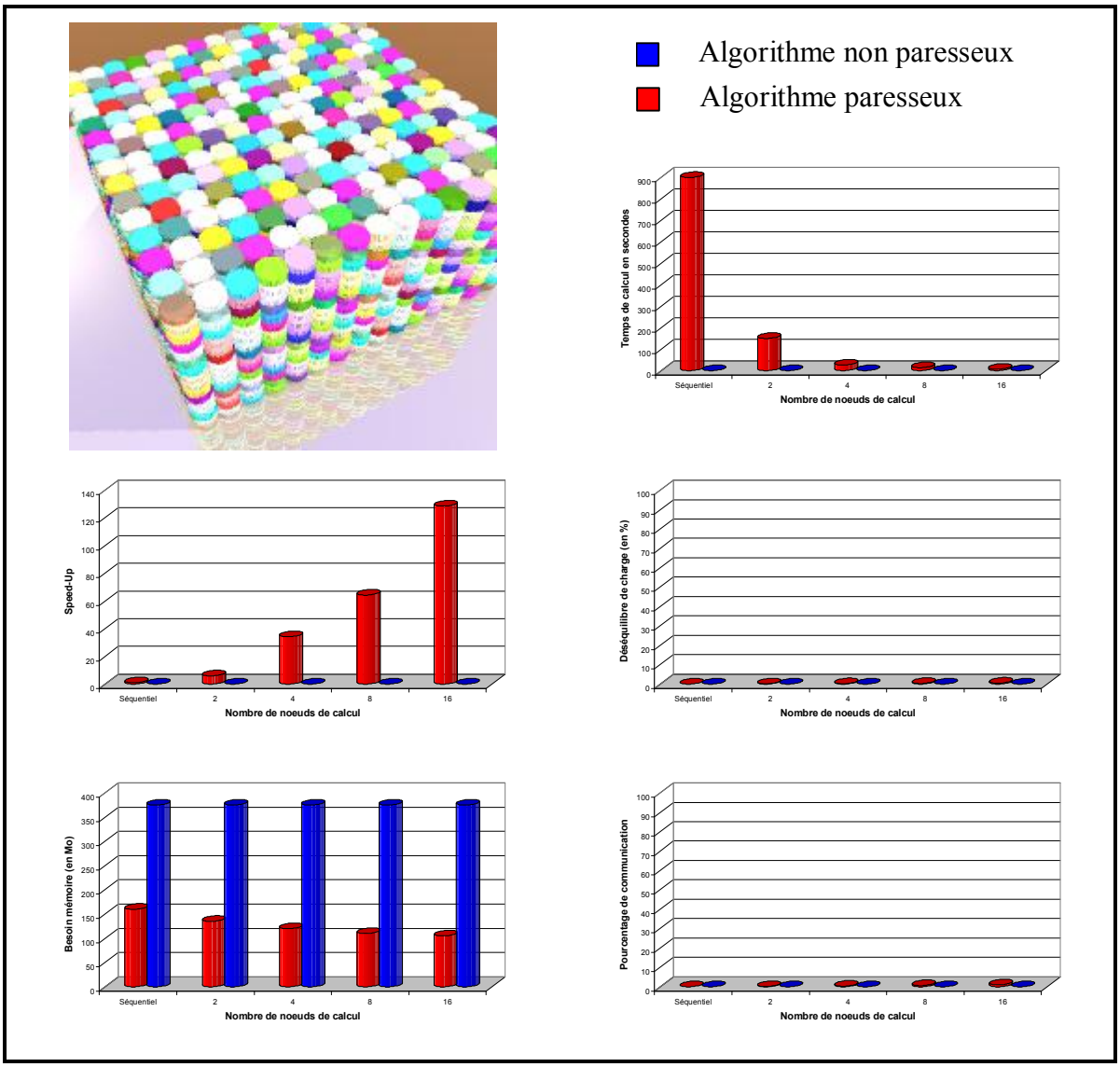


Figure 68 : Fiche de test de la scène Gears 15

Le calcul de cette scène sans paresse nécessiterait plus de 375 Mo de mémoire sur chaque nœud de la machine et la configuration utilisée pour ces tests ne disposait pas d'une telle capacité, le calcul de la scène par l'algorithme sans paresse n'a donc pu aboutir. La version paresseuse de l'algorithme permet une très importante économie de volume mémoire dès l'exécution séquentielle ce qui révèle une quantité importante d'objets cachés (Les engrenages situés à l'intérieur du cube). L'exécution séquentielle de la version paresseuse a été possible mais l'utilisation du système de mémoire auxiliaire (swap) n'a pu être évité d'où un temps de calcul très important. Sur deux nœuds, le système de mémoire auxiliaire est bien moins sollicité et le temps de calcul est grandement réduit. A partir de quatre nœuds de calcul le système de mémoire auxiliaire n'est plus utilisé et les performances augmentent donc de façon très importante. Les valeurs de speed-up sont donc erronées puisqu'un calcul correct nécessite des conditions d'exécution identiques.

Cet exemple montre que la paresse peut permettre de travailler sur des modèles plus volumineux à partir d'une configuration matérielle identique.

5. Vers le lancer de rayon en temps réel : The Tetra

La scène de la pyramide a été produite avec un facteur de complexité de 6 ce qui correspond à une scène de quelques milliers de facettes (4096 facettes dans cet exemple).

Les résultats des mesures sont les suivants :

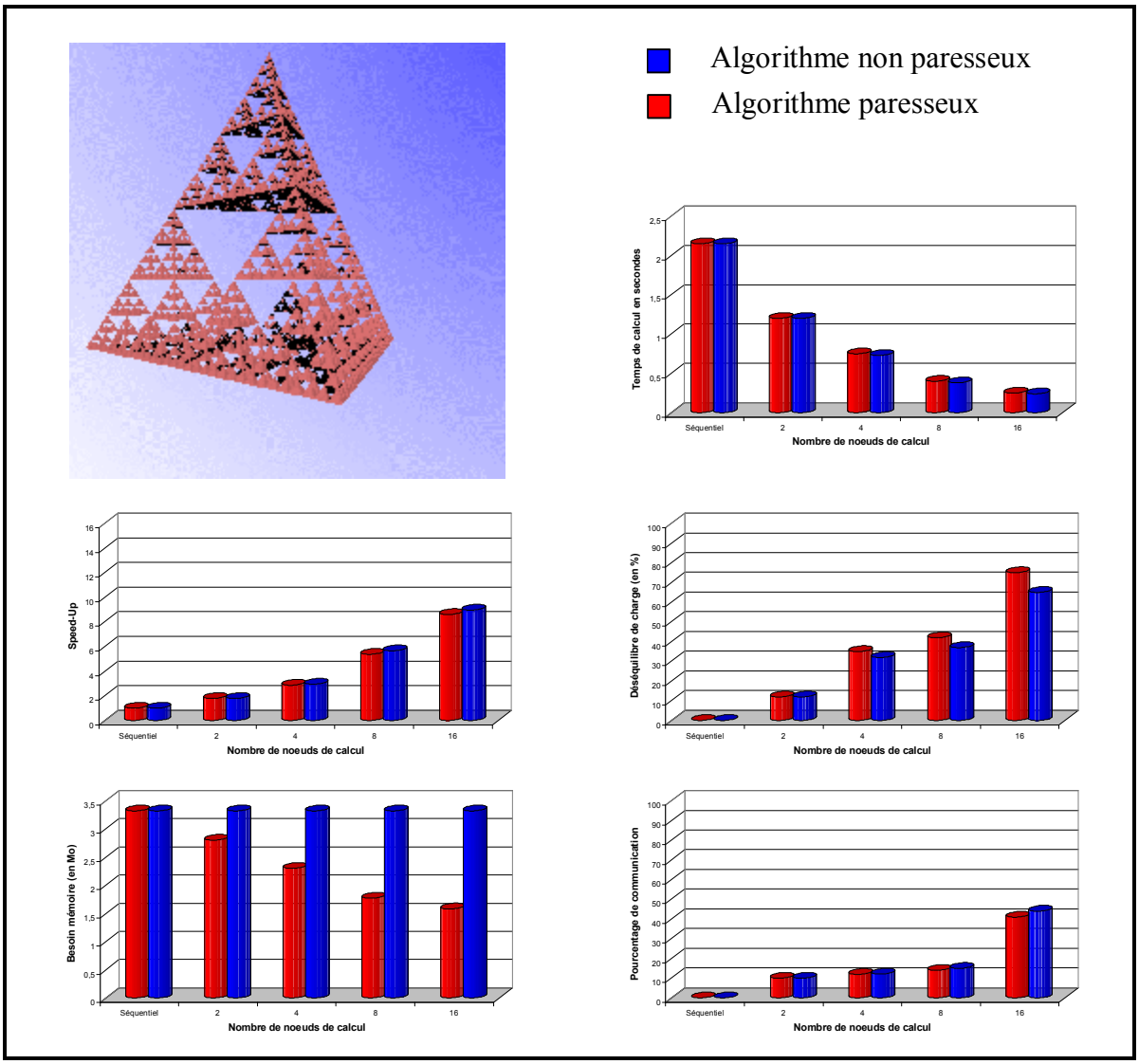


Figure 69 : Fiche de test de la scène Tetra 6

Les valeurs du déséquilibre de charge ainsi que la proportion de communication sont mauvais mais il faut les relativiser en prenant en compte les temps de calcul sur lesquels ils s'appliquent. En effet, le calcul de cette scène ne prenait qu'un peu plus de deux secondes en séquentiel et ce temps est ramené à 0,2 seconde sur 16 nœuds de calcul. L'impact des latences du réseau est donc difficile à compenser.

Ce dernier exemple montre la proximité du lancer de rayon en temps réel puisque, sur un modèle relativement simple, l'algorithme développé par nos soins peut assurer un débit de trois images par seconde (il est courant de commencer à parler de fluidité pour un débit de cinq images par seconde). Il est évident qu'il ne s'agit ici que de tests très préliminaires et qu'il serait imprudent d'affirmer que le lancer de rayon temps réel sera rapidement atteint, mais les performances obtenues ici sont encourageantes.