Développement de techniques de lancer de rayon dans des géométries 3D adaptées aux machines massivement parallèles

Jean-Christophe Nebel

Année 1997

THESE

présentée devant

L'Université de Saint-Etienne et l'Ecole Des Mines de Saint-Etienne

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Jean-Christophe NEBEL

Développement de techniques de lancer de rayon dans des géométries 3D adaptées aux machines massivement parallèles.

Soutenue le 1er Décembre 1997 devant la commission d'examen :

Président du jury : M. Peroche, Professeur à l'école des Mines de St-Etienne

Rapporteur : M. Guitton, Professeur à l'université Bordeaux I

Rapporteur : M. Priol, Directeur de recherche à l'INRIA Rennes

Membre du jury : M. Brochard, Ingénieur au CEA

Membre du jury : M. Nominé, Ingénieur au CEA

Membre du jury : M. Ubeda, Maître de conférences, Université Jean Monnet

Année 1997

THESE

présentée devant

L'Université de Saint-Etienne et
l'Ecole Des Mines de Saint-Etienne

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Jean-Christophe NEBEL

Développement de techniques de lancer de rayon dans des géométries 3D adaptées aux machines massivement parallèles.

Soutenue le 1er Décembre 1997 devant la commission d'examen :

Président du jury : M. Peroche, Professeur à l'école des Mines de St-Etienne

Rapporteur : M. Guitton, Professeur à l'université Bordeaux I

Rapporteur : M. Priol, Directeur de recherche à l'INRIA Rennes

Membre du jury : M. Brochard, Ingénieur au CEA

Membre du jury : M. Nominé, Ingénieur au CEA

Membre du jury : M. Ubeda, Maître de conférences, Université Jean Monnet

Remerciements

Je remercie Bernard PEROCHE, professeur à l'Université de Saint-Etienne, de m'avoir fait l'honneur de présider le jury de cette thèse.

Je remercie Pascal GUITTON, professeur à l'Université de Bordeaux, et Thierry PRIOL, professeur à l'Université de Rennes, d'avoir accepté la charge de rapporteur.

Je remercie Monsieur UBEDA, maître de conférences à l'université Jean Monnet de Saint Etienne, et Messieurs NOMINE et BROCHARD, ingénieurs au CEA, d'avoir accepté de juger mon travail.

Je remercie mon directeur de thèse Bernard PEROCHE, professeur à l'Ecole des Mines de Saint-Etienne, et mon responsable Pierre BROCHARD, Ingénieur au CEA, qui ont veillé au bon déroulement de cette thèse et qui ont su être présents tout en me laissant une large autonomie.

Je tiens à remercier également Nicolas MALLEJAC, compagnon de la première heure, qui m'apporta une aide précieuse tout au long de cette thèse.

Mes remerciements vont aussi à tous les membres du laboratoire GDD, qui m'ont aidés et soutenus. Plus particulièrement, j'adresse ma sincère reconnaissance à Dominique CHAIGNE, Michel CHINEAUD, Guillaume COLIN de VERDIERE, Gérard DEJONGHE et Christèle DOMALAIN pour leurs relectures et corrections de cette thèse.

Et pour finir, je TE remercie.

Table des matières

Liste des figures

Liste des tableaux

Introduction générale

Le lancer de rayon est un algorithme permettant de représenter des scènes 3D de façon très réaliste. Son principe, simple et puissant, repose sur les lois de l'optique géométrique : l'image que voit un observateur est le fruit des interactions entre des photons émis par des sources lumineuses et des objets possédant des propriétés géométriques et optiques. L'utilisation d'un modèle basé sur la physique permet une bonne approche des phénomènes naturels suivants : l'éclairement par des sources lumineuses, la réflexion de la lumière sur des objets et la transmission de la lumière à travers des objets transparents.

L'algorithme dérivant de ce modèle possède pourtant un inconvénient majeur : le temps de traitement d'une image est très important. Bien que de nombreuses méthodes d'accélération aient été proposées pour tenter d'améliorer sa vitesse d'exécution, ce problème demeure bien présent.

Toutefois, avec le développement actuel des machines parallèles à mémoire distribuée et des réseaux d'ordinateurs, une voie d'accélération très prometteuse se développe : la parallélisation. De plus, la mise en commun des capacités "mémoire" de chaque composante d'une architecture parallèle rend possible le traitement de scènes comportant un plus grand nombre d'objets aux propriétés de plus en plus complexes.

Le chapitre 1 présente l'algorithme de lancer de rayon ainsi que les accélérations séquentielles proposées dans la littérature. Il apparaît que les techniques les plus efficaces sont basées soit sur l'utilisation de volumes englobants soit sur des subdivisions de l'espace. Malgré ces optimisations, la parallélisation demeure la source d'accélération au plus fort potentiel.

Le chapitre 2 recense les différents types d'architectures parallèles et expose les familles d'algorithmes utilisées pour la parallélisation du lancer de rayon sur des machines parallèles à mémoire distribuée. Deux stratégies permettent une distribution de la base de données : l'envoi de rayons et l'envoi d'objets.

Dans le chapitre 3, après avoir présenté nos contraintes, nous comparons les algorithmes parallèles pouvant y répondre grâce à leur modélisation. Les résultats obtenus vont nous amener à proposer deux nouveaux algorithmes de parallélisation basés sur un nouveau type de flot.

Enfin, dans le chapitre 4 nous présentons nos résultats expérimentaux et leur analyse. Nos tests sont réalisés sur des machines massivement parallèles et sur un réseau de stations de travail.

Chapitre 1
L'algorithme de lancer de rayon

Dans ce chapitre nous commençons par retracer l'historique de l'algorithme du lancer de rayon. Ensuite nous décrivons plus précisément son principe. Enfin nous nous intéressons aux différentes méthodes qui ont été proposées pour accélérer cet algorithme dans le cadre de l'utilisation de machines séquentielles.

1.1 Présentation générale

Les débuts de l'infographie datent des années 50 avec la visualisation d'objets tridimensionnels en utilisant une représentation dîte en "fils de fer". Depuis lors, un effort constant a été entrepris pour produire des images de plus en plus réalistes dans des délais de plus en plus courts. Un ensemble de techniques de représentation et de manipulation de scènes ont ainsi été proposées tels les modèles filaires, les algorithmes de surfaces et de lignes cachées, la radiosité et le lancer de rayon.

La première référence sur le lancer de rayon dans le monde de l'infographie vient d'Appel en 1968 [App68] qui résolvait uniquement le problème des surfaces cachées, c'est à dire qu'il rendait compte de l'éloignement des objets par rapport à l'observateur. Cet algorithme de conception simple est extrêmement puissant puisqu'il peut être facilement enrichi d'une grande variété de nouveaux effets. Ainsi en 1971, Goldstein [GN71] poursuit les travaux d'Appel en intégrant l'illumination des objets d'une scène par des sources lumineuses ponctuelles ("illumination directe"). En 1980, Whitted [Whi80] étend encore le domaine d'application de cet algorithme en y ajoutant l'illumination par réflexion et réfraction ("illumination indirecte").

Aujourd'hui, le lancer de rayon demeure le modèle de simulation d'illumination-réflexion le plus complet en infographie et permet l'obtention d'images très réalistes. Sa philosophie est basée sur le fait qu'un observateur, voyant un point sur la surface d'un objet, regarde en fait le résultat des interactions de l'objet en ce point avec des rayons provenant d'autres points de la scène. Dans un modèle simple de réflexion de surface, seule l'interaction des points de la surface avec les rayons provenant des sources lumineuses est considérée. En réalité, d'autres rayons lumineux atteignent la surface indirectement à travers des réflexions sur d'autres surfaces, des transmissions à travers des objets transparents ou des combinaisons des deux. Lorsque tous ces phénomènes sont pris en compte, on parle alors d'illumination globale par opposition à l'illumination locale provenant uniquement de l'illumination directe des sources de lumière.

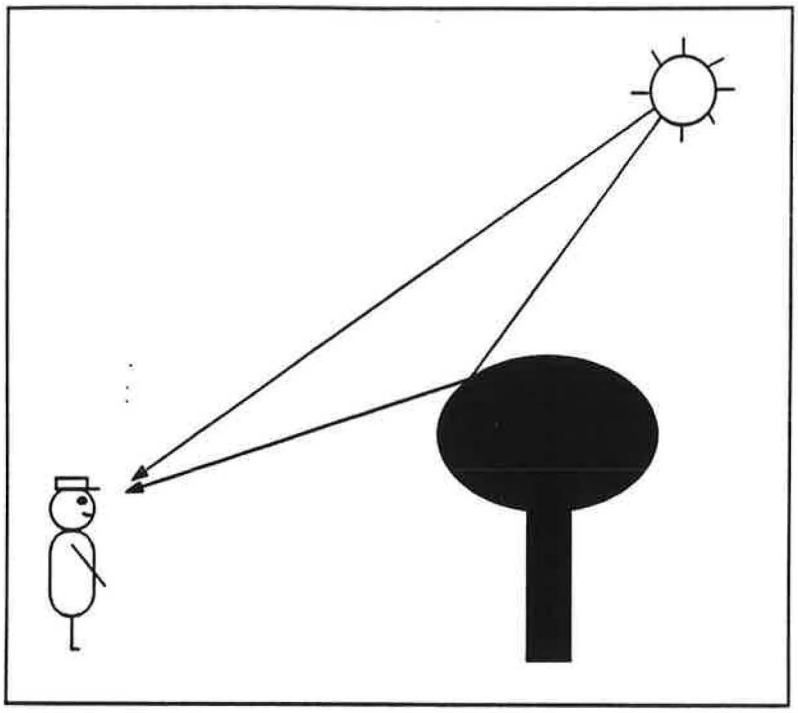

Figure 1 Parcours des photons permettant la formation d'une image pour un observateur

Pour résumer, on peut dire que le lancer de rayon est un modèle de calcul d'éclairement permettant de prendre en compte simultanément de nombreux effets comme :

1. L'éloignement des objets par rapport à l'observateur.
2. L'éclairement dû aux illuminations directes.
3. L'éclairement dû aux illuminations globales ou indirectes.
4. Les effets d'ombrage.

Par contre, il dépend de la position de l'observateur et ne prend pas en compte les interactions diffuses entre les objets.

L'inconvénient majeur de ce modèle est le coût très important en calculs induit par les recherches d'intersections entre les rayons et les objets de la scène. L'obtention d'une image unique pouvant nécessiter des heures, l'utilisation pratique de cette technique reste limitée. Aussi l'enjeu des nouveaux algorithmes de lancer de rayons n'est pas d'en changer la philosophie, mais de permettre l'accélération du procédé.

1.2 L'algorithme

La vision d'un objet provient des rayons lumineux issus des sources de lumière, cependant cet algorithme prend le problème à rebours car seul un faible pourcentage des rayons provenant des sources atteint l'oeil de l'observateur. On utilise des rayons fictifs qui sont lancés depuis l'oeil de l'observateur et on étudie leur parcours dans la scène afin de savoir s'ils peuvent atteindre une source lumineuse.

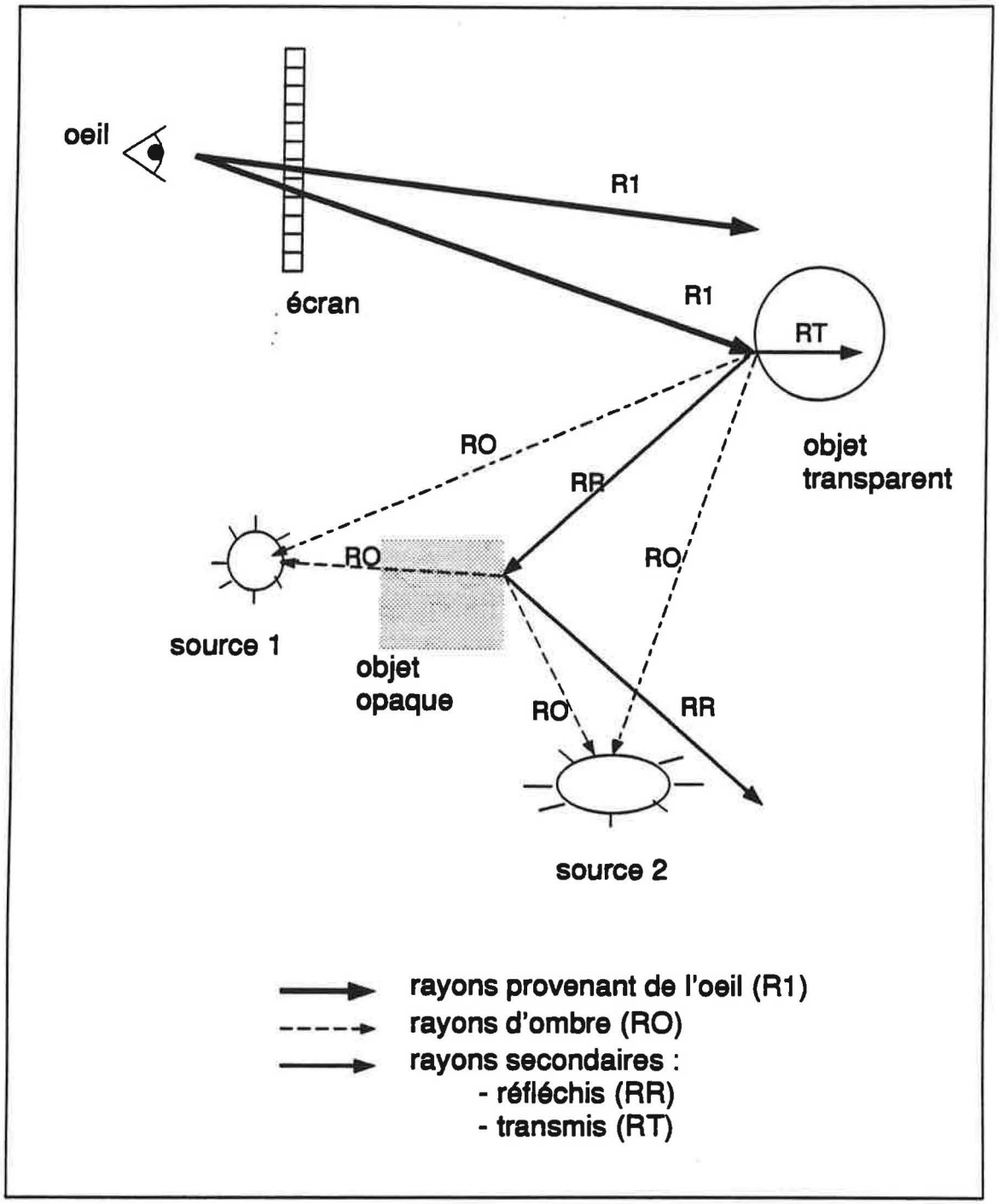

Figure 2 Parcours des rayons lumineux pendant l'algorithme de lancer de rayon

Dans cet algorithme, l'observateur regarde une scène à travers un écran. Afin de déterminer les caractéristiques des pixels de cet écran, on lance depuis l'oeil de l'observateur des rayons traversant chacun des pixels de l'écran; ces rayons sont appelés rayons primaires (R1). Le parcours de chaque rayon va permettre de déterminer les couleurs et intensités des différents pixels en fonction de la visibilité des objets et de la façon dont ils sont éclairés.

Quand un rayon rencontre un objet, on va chercher si cet objet est directement illuminé par une source lumineuse, que l'on suppose ponctuelle. Pour le savoir, on lance depuis le point d'intersection, un rayon en direction de chacune des sources lumineuses présentes dans la scène. Ces rayons sont appelés rayons d'ombre (RO) et vont permettre de déterminer si le point d'intersection est illuminé directement par une ou plusieurs sources lumineuses ou si au contraire il est à l'ombre.

Un calcul à partir des propriétés du rayon initial, de l'objet et de l'illumination directe en ce point va permettre de déterminer la couleur de la lumière directement réfléchie depuis cette surface.

Si l'objet rencontré est partiellement réfléchissant, transparent ou les deux, la couleur du point d'intersection avec le rayon primaire dépendra également de la contribution des rayons réfléchis (RR) et transmis (RT) appelés aussi rayons secondaires. Le parcours de ces rayons dans la scène va donc être étudié.

Ce processus se répète récursivement puisque ces rayons secondaires vont rencontrer des objets qui vont générer l'envoi de rayons d'ombre ainsi que la création de nouveaux rayons secondaires. L'algorithme se termine quand tous les rayons sont sortis de la scène ou quand leurs contributions aux pixels initiaux deviennent négligeables.

Le modèle photométrique de référence pour l'algorithme de lancer de rayon est celui de Whitted [Whi80]. S'appuyant sur des sources lumineuses ponctuelles, il introduit une "intensité ambiante", approximation de la réalité exprimant une luminosité moyenne présente au sein d'une scène et pouvant correspondre à la lumière du jour par exemple. Whitted exprime l'intensité lumineuse en un point comme étant la somme de l'intensité ambiante de la scène, des intensités réfléchies (spéculaire et diffuse) et de l'intensité transmise.

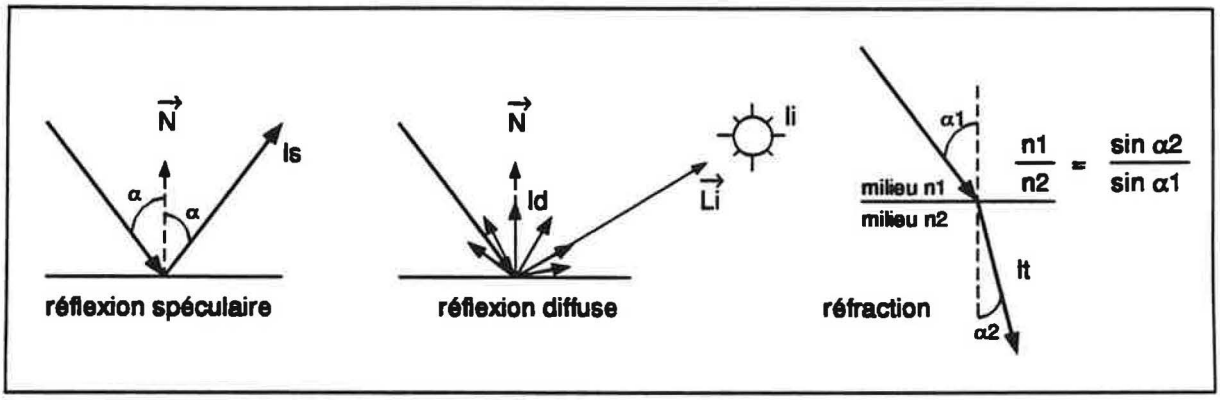

Figure 3 Réflexions et réfraction

Ainsi l'intensité lumineuse d'un pixel dans une scène comprenant n sources lumineuses s'exprime par les formules suivantes :

$$I_{pixel} = I_{ambiante} + I_{diffuse} + I_{speculaire} + I_{transmise}$$
$$I = I_a + K_d C_{obj} \sum_{i=1}^{n} \left(\vec{N}.\vec{L_i}\right) I_i + K_s I_s + K_t I_t \qquad (1)$$

où les différentes intensités lumineuses sont :

1. I l'intensité du pixel,
2. I_a l'intensité ambiante,
3. I_i l'intensité de la ième source lumineuse,
4. I_s l'intensité reçue dans la direction de réflexion,
5. I_t l'intensité reçue dans la direction de réfraction,

et les caractéristiques optiques (ou texture) de l'objet sont :

1. K_d le coefficient de réflexion diffuse $(0 \leq K_d \leq 1)$,
2. K_s le coefficient de réflexion spéculaire $(0 \leq K_s \leq 1)$,
3. K_t le coefficient de réfraction $(0 \leq K_t \leq 1)$,

4. C_{obj} la couleur de l'objet,

enfin :

1. $\vec{N}$ la direction normale au point d'impact,
2. $\vec{L_i}$ la direction vers la ième source lumineuse.

L'algorithme de base pour le lancer de rayon est de la forme suivante :

Pour *chaque pixel de l'écran*

Etape 1 : *Créer un rayon primaire (R) partant de l'oeil de l'observateur et traversant le pixel considéré.*

Etape 2 : *Rechercher l'intersection la plus proche entre ce rayon (R) et les objets de la scène.*

Si *il y a intersection*
Alors

Pour *chaque source lumineuse*

Lancer un rayon d'ombre du point d'intersection vers la source
Chercher s'il y a une intersection entre ce rayon d'ombre et un objet afin d'apporter la contribution éventuelle de la source à la couleur du pixel

Si *la contribution du rayon (R) n'est pas négligeable*
Alors

Lancer du rayon réfléchi (R) et retour à l'étape 2
Lancer d'un éventuel rayon réfracté (R) et retour à l'étape 2

Sinon

Calcul de la contribution du rayon (R) aux propriétés du pixel

Sinon

Calcul de la contribution du rayon (R) aux propriétés du pixel

Etape 3 : *Calcul des propriétés du pixel*

1.3 Méthodes d'accélération

L'inconvénient majeur de l'algorithme du lancer de rayon est sa vitesse d'exécution qui en limite l'utilisation pratique. Dans notre travail, nous nous intéressons donc particulièrement aux différentes méthodes d'accélération de cet algorithme.

Ris [Ris96] propose une classification originale de ces techniques d'accélération en se basant sur trois types de graphes :

1. Le graphe des objets de la scène (les objets réels).
2. Le graphe des rayons (les objets virtuels).
3. Le graphe de la structure du programme (les processus).

Toutefois, afin d'éviter d'introduire de nouveaux concepts qui ne nous seront pas utiles dans la suite de notre étude, nous nous inspirerons de la classification plus classique proposée par Arvo et Kirk [AK89].

Les méthodes d'accélération peuvent être regroupées en deux familles complémentaires :

1. La réduction du temps de calcul des intersections.
2. La parallélisation des calculs.

Le premier groupe de méthodes est présenté dans ce chapitre, le chapitre suivant étant consacré à la parallélisation des calculs.

1.3.1 Introduction

Dans l'algorithme du lancer de rayon classique, à chaque étape du trajet d'un rayon, il faut calculer les intersections entre le rayon concerné et tous les objets de la scène afin de déterminer l'intersection la plus proche, si elle existe. Aussi, dès que le nombre d'objets devient important, le nombre de calculs à effectuer devient énorme. De plus, ces calculs peuvent être compliqués et longs si les objets sont complexes. Whitted [Whi80] a montré que les calculs d'intersection pouvaient constituer jusqu'à 95% du temps de calcul total. C'est pourquoi des techniques permettant de limiter ou de simplifier les calculs d'intersection ont été mises au point.

Nous avons répertorié cinq groupes de méthodes d'accélération séquentielle pour l'algorithme de lancer de rayon que nous allons détailler :

1. L'utilisation de la cohérence entre les rayons.
2. Le contrôle de profondeur adaptable.
3. Les volumes englobants et les hiérarchies de volumes englobants.
4. La subdivision de l'espace.
5. Les techniques directionnelles.

Bien entendu toutes ces techniques peuvent être combinées entre elles pour une meilleure optimisation de l'algorithme.

1.3.2 Utilisation de la cohérence entre les rayons

Le premier type d'accélération qui peut être envisagé repose sur la cohérence des rayons. L'idée générale est que deux rayons issus de deux pixels proches ont de grandes chances de rencontrer les mêmes objets lors de leur parcours. Il existe plusieurs façons d'exploiter cette cohérence. Nous présentons d'abord l'utilisation de l'arbre d'un pixel voisin, puis le traitement des rayons par classe.

1.3.2.1 Utilisation de l'arbre d'un pixel voisin

Cette première méthode permet de bénéficier des calculs d'un pixel voisin. En effet, l'ensemble des rayons générés par l'envoi d'un rayon primaire conduit à la construction d'un arbre des intersections.

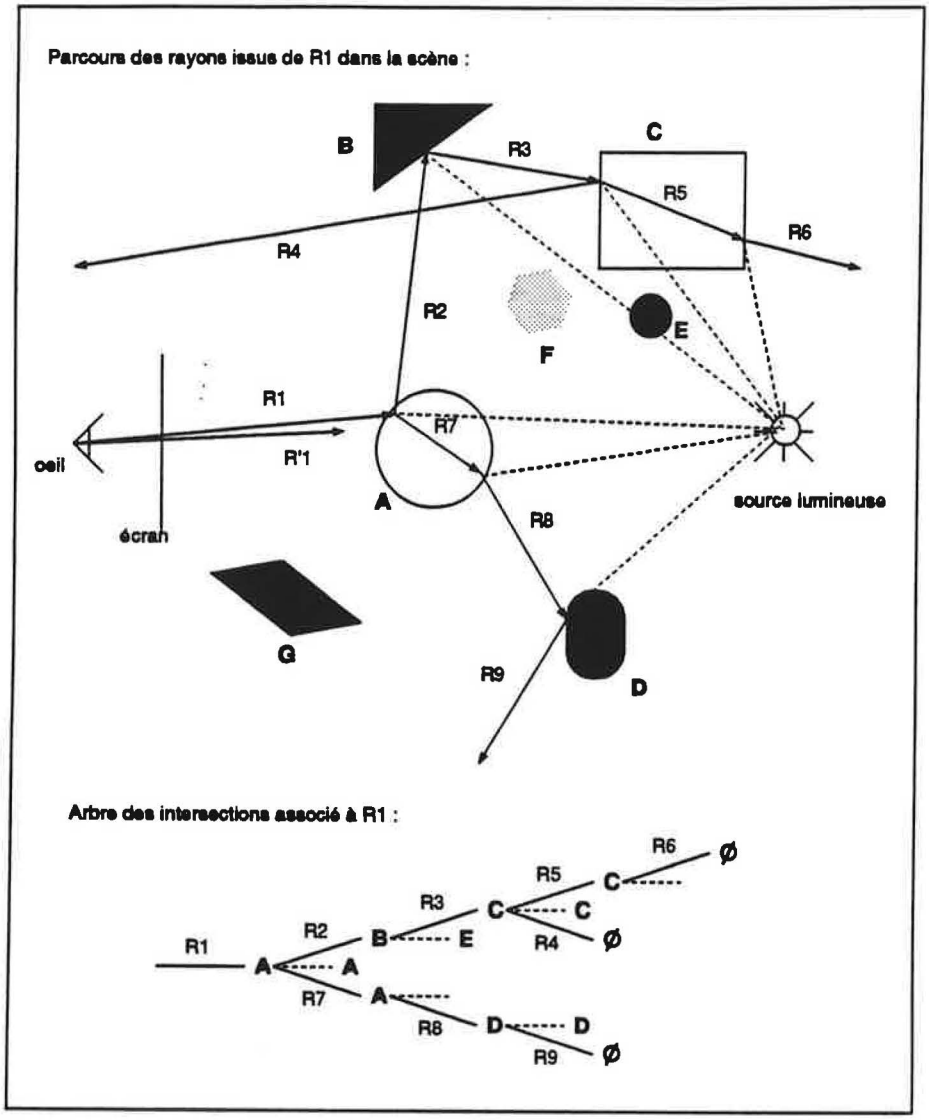

Figure 4 Arbre des intersections d'un rayon

Aussi, une fois qu'un tel arbre a été établi pour un rayon (R1), Arvo et Kirk [AK89] proposent de l'utiliser pour construire celui d'un rayon proche (R'1).

Ceci se fait en deux étapes :

1. On commence par tester l'intersection de ce deuxième rayon (R'1) avec l'objet rencontré par le premier rayon (R1),
2. ensuite on cherche s'il n'y a pas eu d'intersections sur le trajet séparant l'origine du rayon de cette intersection.

La deuxième phase étant assez longue, on peut construitre à partir de "l'arbre modèle" une zone de sécurité formée par des cylindres ne coupant que les objets appartenant à l'arbre du rayon modèle. Ainsi tous les rayons voisins voyageant dans ces cylindres sont assurés de ne pas rencontrer d'objets différents de ceux rencontrés par le rayon à l'origine de ces cylindres (R1).

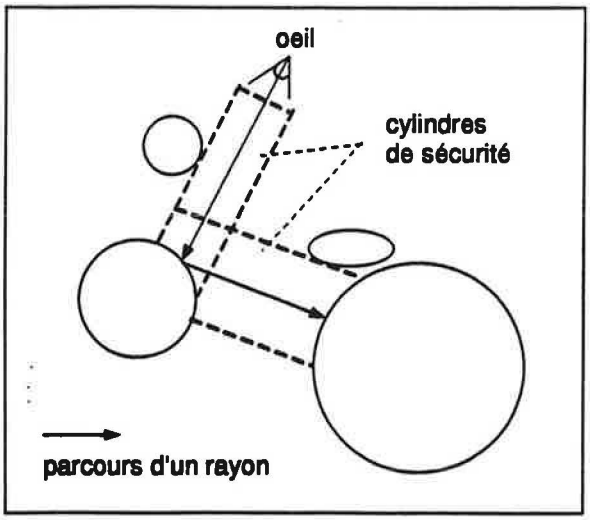

Figure 5 Cylindres de sécurité

Bien que la recherche d'intersection soit accélérée, les coûts de calcul engendrés pour la création des cylindres de sécurité semblent plus élevés que les bénéfices apportés par la prise en compte de cette cohérence.

Une autre approche est de réaliser des cônes de sécurité, où chaque cône comprendra la liste des objets qu'il coupe. Malheureusement leurs coûts de construction sont également prohibitifs.

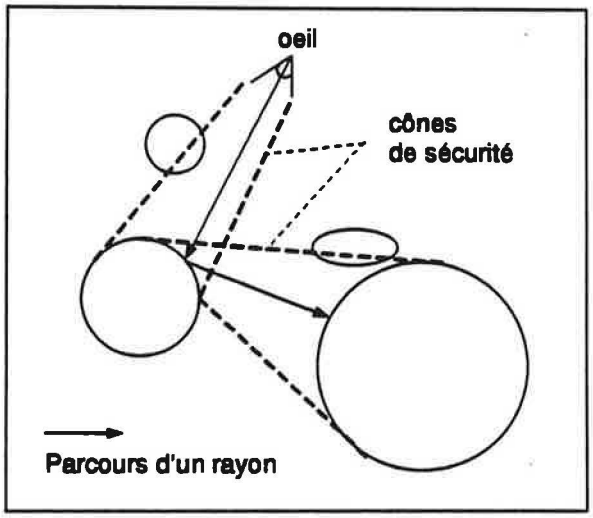

Figure 6 Cônes de sécurité

1.3.2.2 Traitement des rayons par classe

Une autre façon de profiter de la cohérence des rayons est de traiter séparément les différents types de rayons. Ainsi, au lieu de calculer chaque rayon sur tout son parcours, Green et Paddon [GP90] proposent de traiter d'abord tous les rayons primaires, puis les rayons d'ombrage et enfin les rayons secondaires.

Contrairement à la méthode précédente, il n'y a pas de calculs supplémentaires à effectuer. Par contre, cela nécessite un accroissement significatif de la taille de la mémoire utilisée. En effet, un grand nombre de rayons doit être mémorisé en même temps.

1.3.3 Le contrôle de profondeur adaptable

La construction de l'arbre des rayons issus d'un pixel se termine quand tous les rayons secondaires ont quitté la scène. Cependant, afin de limiter ces calculs qui peuvent éventuellement être infinis, on fixe souvent une profondeur limite fixant le nombre de récursions maximales pour le calcul d'un pixel.

Toutefois, le nombre de récursions souhaitable pour rendre une scène réaliste dépend beaucoup du type d'objets la composant. Ainsi, une scène possédant de nombreux objets transparents aura besoin d'une profondeur d'un ordre supérieur à celle d'une scène composée d'objets opaques. De plus si une scène est très hétérogène, certains rayons peuvent nécessiter une profondeur très supérieure à d'autres. Aussi, plutôt que d'imposer une profondeur à l'ensemble des rayons, il semble naturel de proposer une profondeur adaptable.

A chaque réflexion ou réfraction, la contribution d'un rayon est atténuée par un coefficient. Watt [Wat93] propose de fixer un seuil en dessous duquel la contribution devient négligeable, ainsi les récursions peuvent cesser. Par cette méthode, il y a donc diminution de la profondeur moyenne tout en conservant un rendu de même qualité. Watt [Wat93] annonce pour certaines scènes une profondeur moyenne divisée par un facteur 10.

1.3.4 Les volumes englobants et les hiérarchies de volumes englobants

1.3.4.1 Les volumes englobants

L'utilisation de volumes englobants, volumes de géométrie simple englobants les objets réels d'une scène, a été très tôt proposée par Whitted [Whi80]. En effet, les calculs d'intersection entre une droite et les objets d'une scène peuvent être très complexes. Aussi, pour les limiter, on commence par effectuer un calcul d'intersection avec leur objet englobant. S'il y a intersection, le calcul avec l'objet lui-même sera réalisé, sinon aucune intersection n'est possible entre le rayon et l'objet.

De nombreux types de volumes englobants sont proposés. Les plus utilisés sont des sphères, des cubes ou des parallélogrammes.

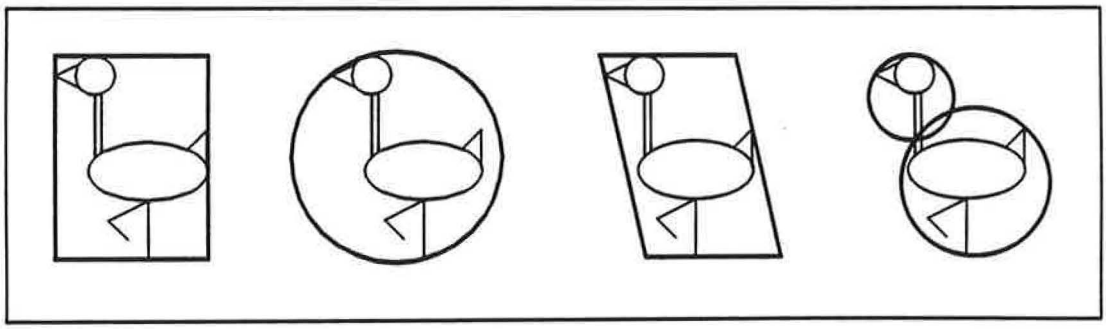

Figure 7 Exemples de volumes englobants

Le choix du type de volume englobant à utiliser se pose. Pour cela Arvo et Kirk [AK89] proposent une expression du coût des calculs d'intersection :

$$cout = n * B + m * I \qquad (2)$$

où :

n est le nombre de rayons dont on calcule l'intersection avec un volume englobant,

B est le coût du calcul d'une intersection rayon-volume englobant,
m est le nombre de rayons ayant intersecté un volume englobant,
I est le coût du calcul d'une intersection rayon-objet.

n et I étant fixés, il faut donc chercher à réduire B et m. La difficulté est que ces deux paramètres évoluent de façons opposées. En effet, la réduction de m passe par l'utilisation de volumes englobants plus proches des objets. Cela implique que ces volumes englobants seront de nature plus compliquée, ce qui va provoquer une augmentation de B. Il va falloir trouver un bon compromis dans les choix de B et de m.

Pour réduire m il est intéressant de connaître la probabilité qu'un rayon touche un objet sachant que le rayon a touché son volume englobant :

$$P(rayon\ touche\ l'objet / rayon\ touche\ l'englobant) \qquad (3)$$

Les travaux de Arvo et Kirk aboutissent à l'utilisation de parallélépipèdes comme volumes englobants permettant le meilleur arbitrage entre B et m.

Forgue et al. [FGBA87] préconisent un traitement différent selon que le rayon est primaire ou non. Pour ce faire, ils proposent d'exploiter le concept de volumes englobants en utilisant deux types de volumes : des sphères et des parallélépipèdes.

Ainsi, ils commencent par englober les objets dans des parallélépipèdes qui sont projetés sur le plan écran. Seuls sont lancés les rayons dont le pixel origine est contenu dans au moins une surface projetée. Pour chacun de ces rayons, une liste d'objets candidats à une intersection est créée. Cette liste comprend les objets contenus dans les boîtes objets associées aux projections.

Une fois les rayons primaires calculés, Forque et al. utilisent des sphères englobantes pour déterminer les intersections entre les rayons générés et les objets de la scène.

1.3.4.2 Hiérarchie de volumes englobants

L'utilisation de volumes englobants simplifie les calculs, mais le nombre de calculs d'intersection reste le même. Aussi Rubin [RW80] propose l'utilisation d'une hiérarchie de volumes englobants. On obtient alors une structure arborescente dont les noeuds sont des volumes englobants et les feuilles les objets. Le nombre de calculs d'intersection passe donc de *O(obj)* à *O(log obj)* pour un arbre binaire équilibré.

Goldsmith et Salmon [GS87] proposent une méthode permettant de générer automatiquement une hiérarchie de volumes englobants. D'abord, ils définissent une heuristique permettant d'estimer le coût d'une hiérarchie. Ce coût s'exprime par le nombre moyen d'intersections avec des volumes englobants qu'un rayon devra faire avant de rencontrer un objet. Il dépend donc de la profondeur de la hiérarchie et de la "qualité" des volumes englobants.

La hiérarchie est construite par ajouts successifs des objets de la scène. Pour chaque objet, on recherche le volume englobant de la hiérarchie qui nécessitera la plus faible augmentation de volume permettant l'addition de l'objet à l'intérieur.

On dispose de deux possibilités :

Choix 1 : On ajoute l'objet au noeud trouvé, ce qui fait augmenter le nombre d'objets associés au noeud.

Choix 2 : On crée un nouveau noeud à la hiérarchie contenant le volume englobant et l'objet, ce qui fait augmenter le nombre de volumes englobants.

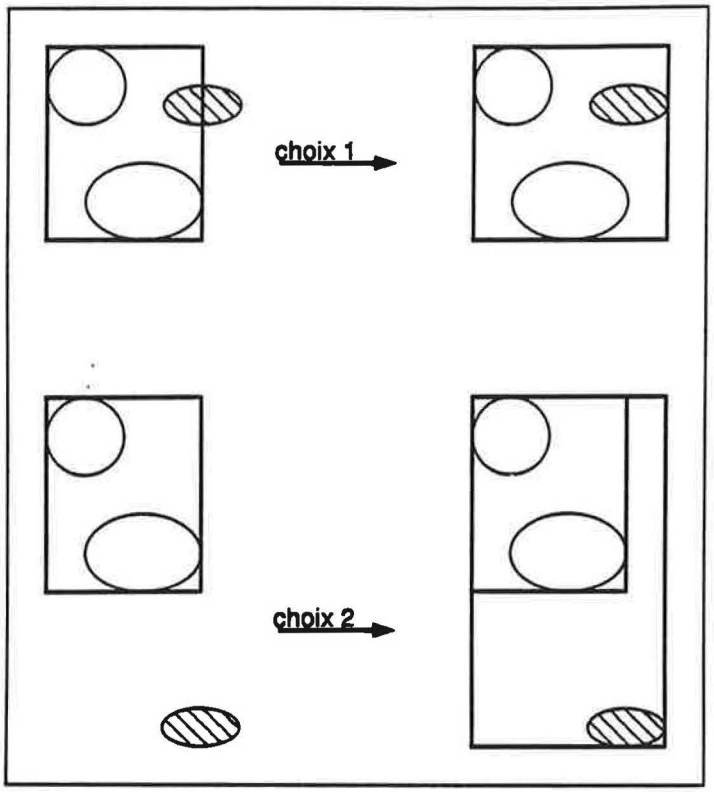

Figure 8 Construction de la hiérarchie de volumes englobants

Une fois que la hiérarchie est construite, il est encore possible d'accélérer le parcours de l'arbre obtenu. En effet au lieu d'un parcours récursif classique, Kay [KK86] propose une méthode favorisant les volumes englobants les plus proches l'un de l'autre, permettant ainsi de trouver plus rapidement l'intersection avec l'objet le plus proche. Ainsi, plutôt que de faire un parcours en profondeur, on obtient un parcours plus en largeur. Le coût de cet algorithme correspond au maintien d'une queue de noeuds prioritaires, ce qui est faible comparé aux gains apportés.

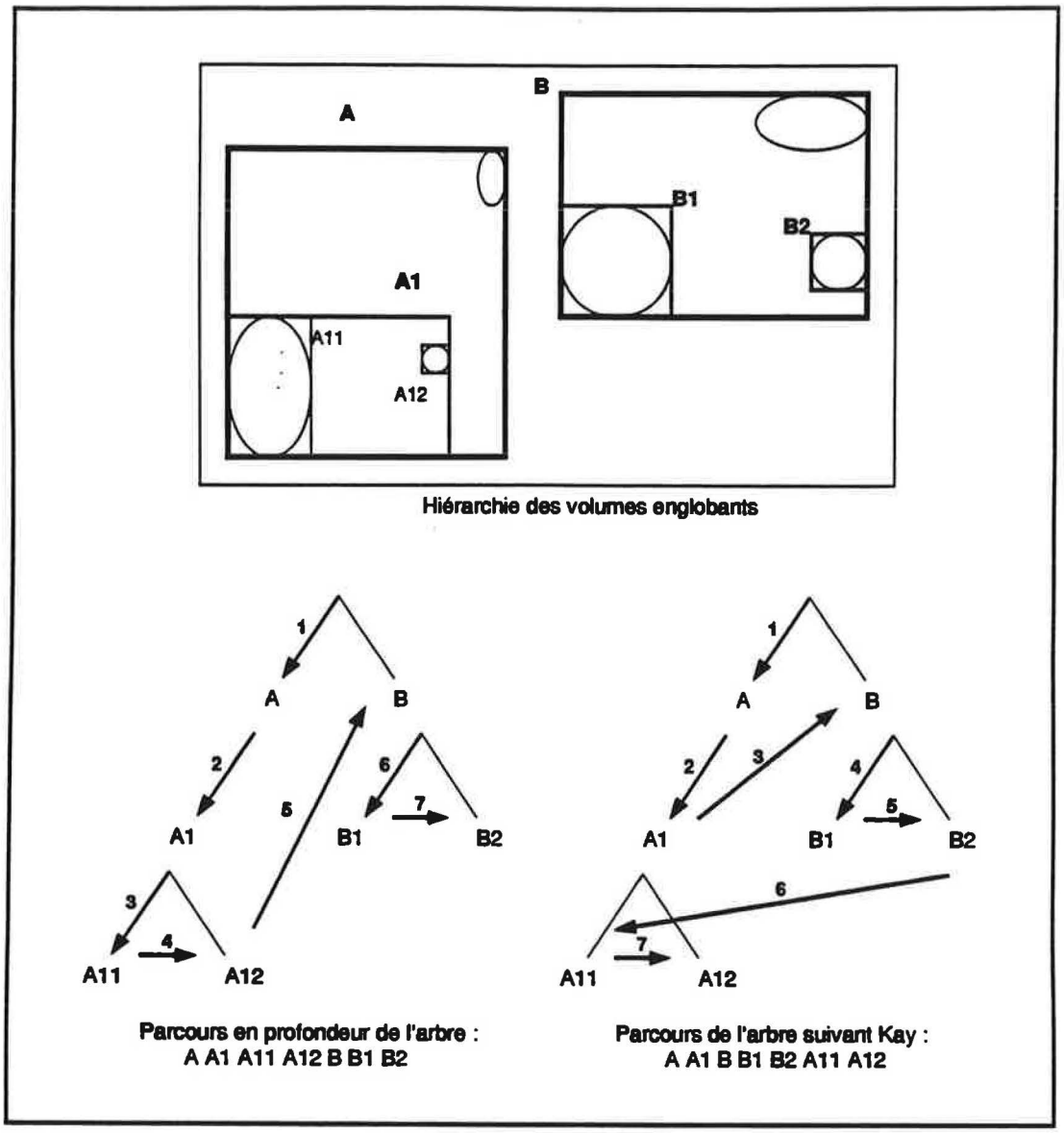

Figure 9 Accélération du parcours d'un arbre

1.3.5 Subdivision de l'espace

Afin de limiter le nombre d'intersections, des méthodes de subdivision de l'espace sont proposées. Elles permettent d'exploiter la notion de "voisinage" et de limiter le nombre d'objets candidats à une intersection.

Ces algorithmes sont basés sur une observation simple. Si on divise une scène en petits compartiments, chacun d'eux ayant la liste des objets qui y résident, on peut facilement accélérer la recherche d'intersection. En effet, on part du compartiment dans lequel le rayon a son origine, puis on suit le rayon et à chaque changement de compartiment on calcule les intersections entre le rayon et les objets contenus dans ce nouveau compartiment. Quand l'intersection la plus proche au sein d'un compartiment a été trouvée, le calcul est terminé car celle-ci met en cause forcément l'objet le plus proche de l'origine.

Ainsi si chaque compartiment possède peu d'objets, le nombre d'intersections calculées afin de trouver l'intersection la plus proche est diminuée fortement. Le gain en temps peut être conséquent si le coût nécessaire pour passer d'un compartiment à un autre est faible.

A partir de cette idée générale, deux types de découpage de l'espace sont proposés :

1. Un découpage adaptatif.
2. Un découpage régulier ou en voxels (volumes élémentaires).

1.3.5.1 Découpage adaptatif de la scène

Les premières techniques proposées par Murakami [MM83] et Glassner [Gla84] sont des découpages en octree. On découpe l'espace tridimensionnel suivant chaque axe en utilisant les plans médians : on obtient alors huit sous-espaces. Chacun de ces sous-espaces est également découpé en huit s'il contient plus d'un certain nombre d'objets. Ce procédé est alors itéré tant que le critère précédent n'est pas satisfait. Une arborescence est obtenue dont chaque noeud contient huit branches, chacune d'elles correspondant à un sous-espace de la scène. Chaque cube possède la liste des objets qu'il contient.

Sur la figure suivante est présenté le découpage adaptatif d'une scène 2D (le découpage se fait donc en "quadtree" au lieu d'"octree"). Chaque sous-espace est divisé tant qu'il possède plus d'un objet.

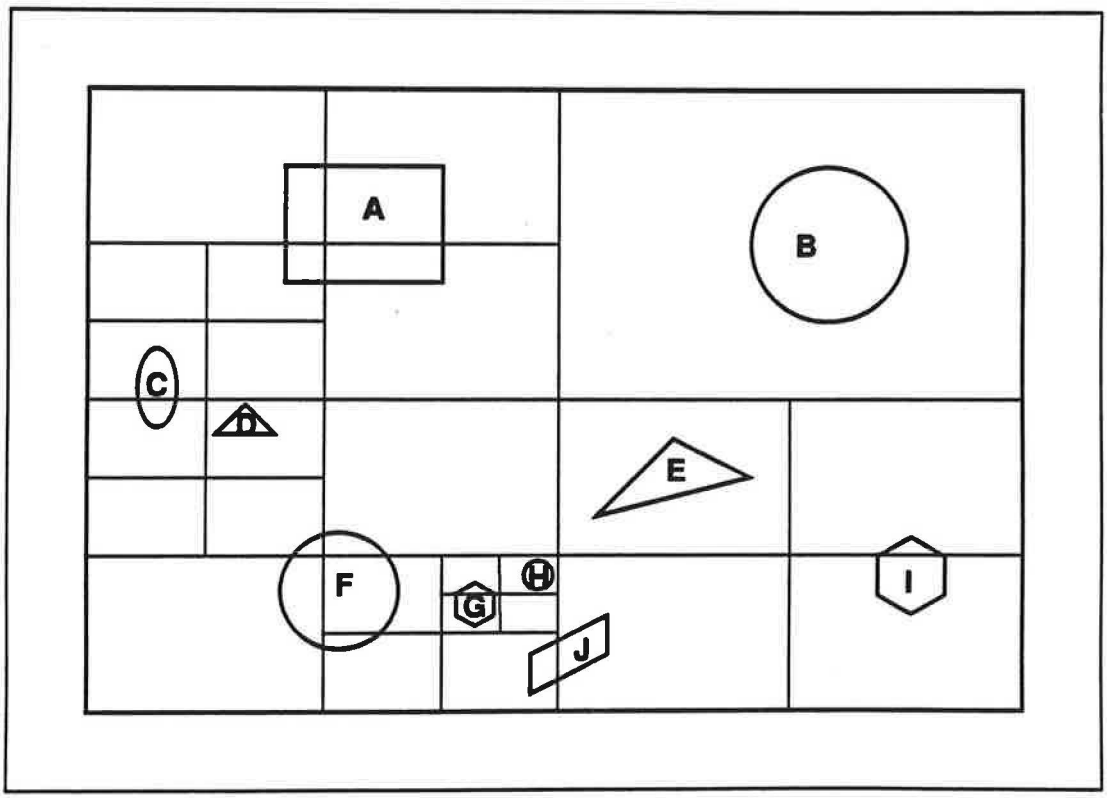

Figure 10 Découpage en quadtree d'une scène 2D

Le découpage précédent induit la construction de l'arbre suivant :

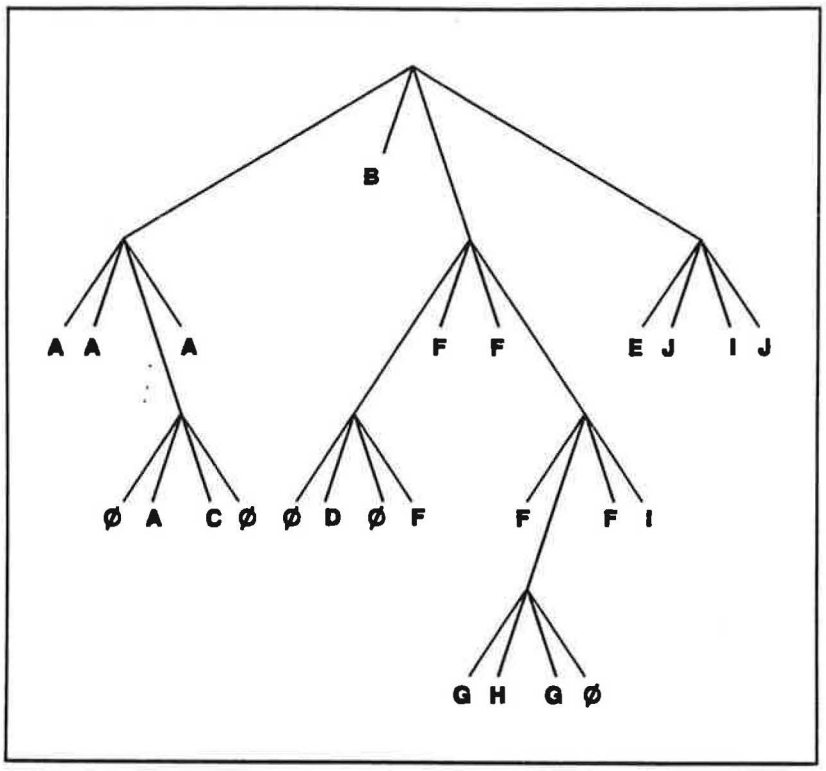

Figure 11 Arbre de découpage en quadtree d'une scène 2D

Afin d'assurer un passage rapide d'un cube vers un autre cube adjacent, Agate [AGL89] et Spackman [SW91] proposent des algorithmes performants, nommés respectivement HERO et SMART. La comparaison de ces algorithmes de parcours de subdivision de l'espace est présentée ultérieurement (§ 1.3.5.3).

Hebert et al. [HMS+92] ont étudié le taux d'utilisation des feuilles de bas niveau pour la décomposition en octree de la base de données "teapot" (voir annexe B). Il s'avère que les quatre derniers niveaux correspondent à 56,6% des noeuds, mais seulement à 2,2% des accès. Il pourrait être tentant de ne pas faire une décomposition si fine. Toutefois supprimer ces niveaux conduit à des temps de calcul d'intersections beaucoup plus longs. Une solution serait de construire dynamiquement les faibles niveaux, quand ils sont nécessaires, pendant l'exécution du lancer de rayon. Cette proposition permet ainsi de réduire la taille de la base de données pour un faible coût en temps.

D'autres découpages adaptatifs ont été proposés. On peut citer le BSP ("Binary Space Partition") de Kaplan [Kap85], méthode qui donne des résultats similaires à ceux de l'octree bien que basée sur une partition binaire.

Enfin Whang et al. [WSC+95] proposent une forme dérivée de découpage en octrees : l'octree-R. Le découpage de la scène en huit se fait également itérativement, mais au lieu de partager les sous-espaces par des plans médians, Whang et al. font des découpes par des plans qui minimisent le nombre d'objets contenu dans chaque sous-espace. De ce fait, il y a réduction du nombre de calculs d'intersection rayon-objet.

Les résultats de l'octree-R présentés montrent que le gain en performance par rapport à l'octree classique est de 4 à 47% suivant les scènes considérées. Bien entendu, le temps de construction de l'arbre est augmenté d'un facteur compris entre 6 et 8, mais il reste négligeable par rapport au temps de calcul total.

1.3.5.2 Découpage régulier

Fujimoto et al. [FI85], [FTI86] ont été les premiers à proposer une méthode appelée SEADS ("Spatially Enumerated Data Structure"). L'espace est découpé en une grille 3D de cubes de même taille (voxels) , pouvant être stockée sous la forme d'un tableau tridimensionnel indépendant de la scène à représenter.

Sur la figure suivante est présenté le découpage régulier d'une scène 2D.

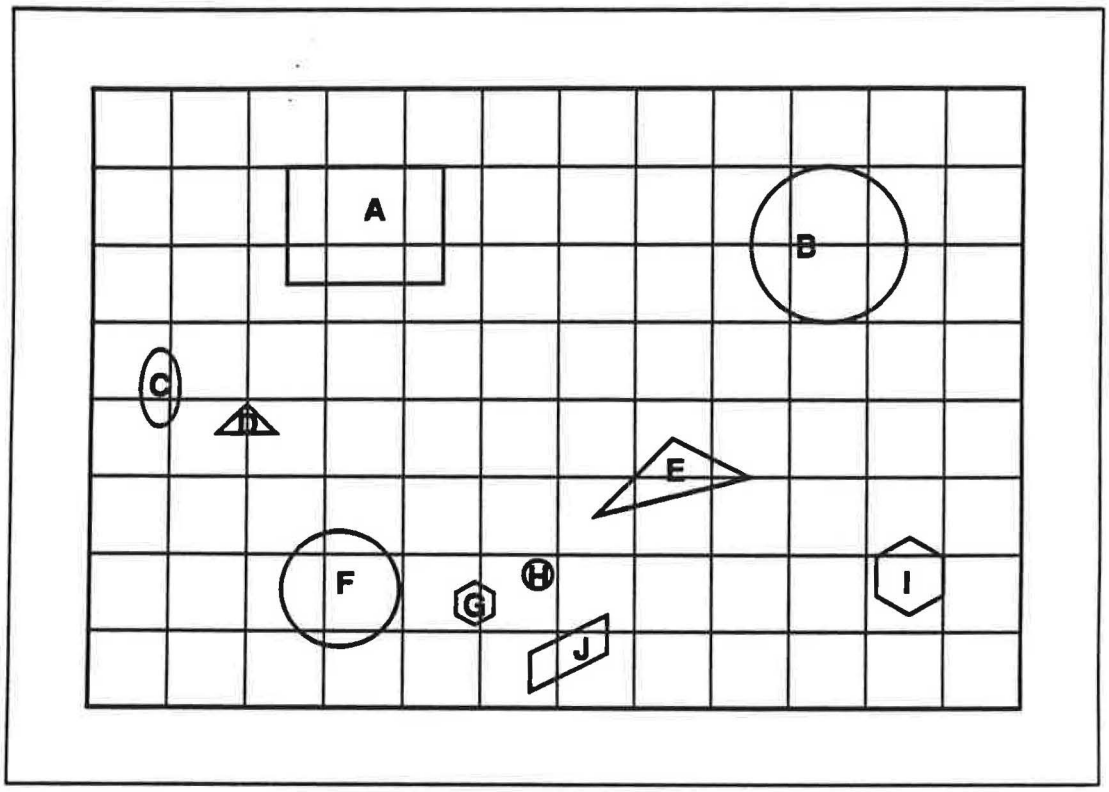

Figure 12 Découpage régulier d'une scène 2D

L'intérêt de ce type de découpage est la rapidité de parcours de la structure. Ainsi Murakami et al. [MHI88], bien qu'ayant commencé à travailler sur des découpages adaptatifs [MM83], préconisent de faire une subdivision de l'espace objets par une structure en cubes réguliers. Le coût de parcours d'une structure en octree lui apparaît beaucoup trop lourd comparé à celui d'une structure en voxels réguliers.

Les avantages d'un tel découpage sont la simplicité de mise en oeuvre et l'accès rapide d'une cellule à une autre. Par contre, quand la répartition des objets dans la scène est hétérogène, il y a une nette augmentation du nombre de calculs d'intersections rayon-objet par rapport à l'octree. De plus, les besoins en mémoire sont beaucoup plus importants que pour un découpage adaptatif.

1.3.5.3 Comparaisons

Les deux types de découpages que nous venons de présenter apportent incontestablement une accélération de l'algorithme de lancer de rayon. Le problème semble être le choix entre l'une ou l'autre technique. Ce choix doit être guidé par la comparaison des coûts de calcul et des coûts mémoire.

Endl et Sommer [ES94] ont réalisé une étude très complète afin de comparer les différents algorithmes de parcours de structure en octree et en voxel. Neuf algorithmes ont été implémentés

et testés sur des scènes identiques. Ces algorithmes ont été regroupés en quatre catégories qui dépendent du type de structure utilisée pour le découpage et de la méthode employée pour déterminer la cellule suivante :

1. "Octree depuis la racine" :
   la cellule suivante est cherchée en redescendant l'arbre depuis la racine de l'octree (algorithme de Glassner et al. [GHH+89]).
2. "Octree par parent" :
   la cellule suivante est trouvée en remontant jusqu'à la première cellule parente à la cellule actuelle et à la cellule recherchée, puis en redescendant (algorithme de Samet [Sam89], méthodes Samet_Corner et Endl_Cebit [ES92]).
3. "Octree par réseau" :
   les cellules de même tailles sont trouvées directement, les plus grosses sont soit trouvées directement, soit en remontant au-dessus d'un voisin virtuel de même taille. Les cellules plus petites sont trouvées en descendant à partir d'une cellule voisine existante (algorithme Samet_Net, Endl_Net et Endl_FastNet).
4. "Subdivision uniforme" (algorithmes de Müller [Mül85] et de Amanatides et Woo [AW87]

Les résultats des tests réalisés peuvent être résumés par le graphique suivant :

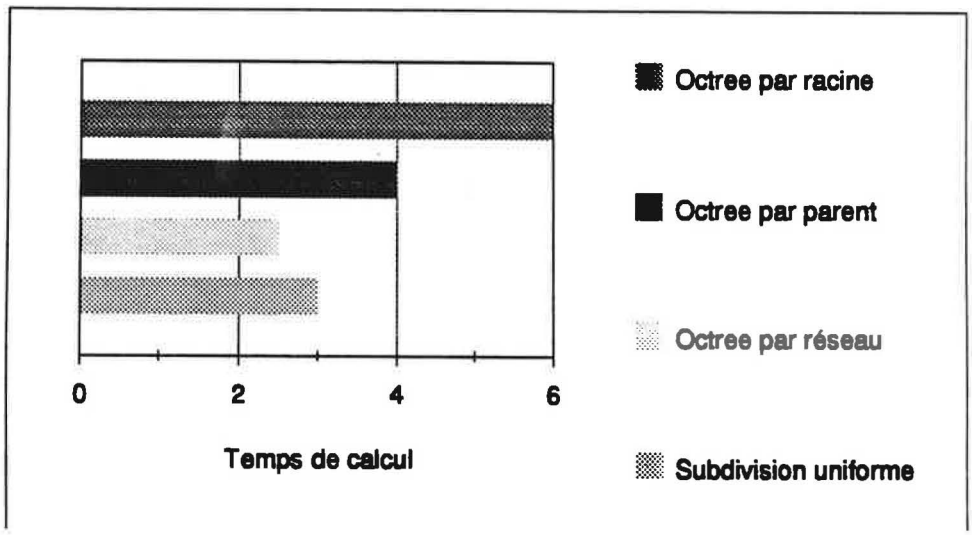

Figure 13 Comparaison des algorithmes de parcours de structure

Pour les scènes testées, il apparaît que les méthodes "octree par réseau" sont environ deux fois plus rapides que celles "depuis la racine", les méthodes "octree par parent" donnant des résultats intermédiaires. Les méthodes de "subdivision uniforme" sont également plus lentes que les méthodes "octree par réseau", mais restent néanmoins intéressantes.

En effet, l'auteur précise que l'on ne peut pas généraliser en annonçant la supériorité en temps de calcul des méthodes "octree par réseau" par rapport aux méthodes de "subdivision uniforme". En effet, cela dépend de la scène traitée : les algorithmes utilisant une subdivision uniforme sont plus performants pour les scènes relativement homogènes. Par contre, pour des scènes hétérogènes, un découpage en octree utilisant une méthode "par réseau" doit donner les meilleurs résultats.

On peut ajouter que les découpages en octree ont un coût mémoire bien inférieur aux découpages uniformes qui contiennent généralement de nombreux voxels vides. Avec ces nouvelles méthodes "par réseau", le rapport coût mémoire efficacité devient particulièrement intéressant pour un découpage en octree.

1.3.5.4 Utilisation de volumes englobants et de découpages de l'espace

Afin d'obtenir une accélération maximale de l'algorithme, Priol [Pri89] propose une méthode de découpage combinant volumes englobants et voxels pyramidaux

Ainsi pour chaque objet de la scène, le volume englobant (parallélépipède) minimal est recherché. Puis, ces volumes englobants sont projetés en perspective sur l'écran, on obtient ainsi un ensemble de rectangles.

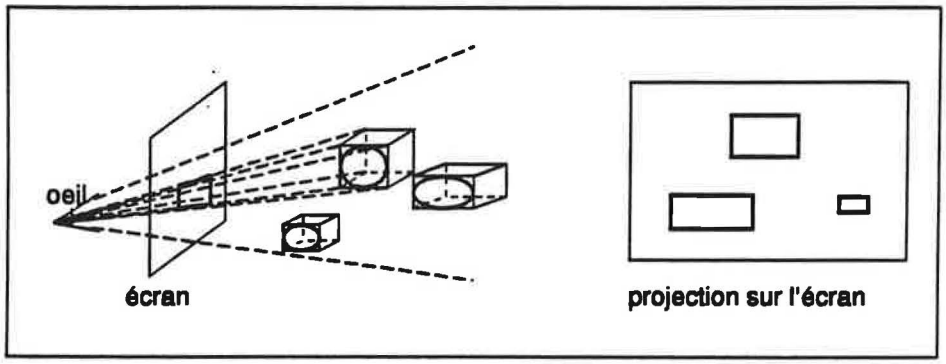

Figure 14 Projection des volumes englobants sur l'écran

A l'aide des segments composant ces rectangles, les sous-espaces de l'écran sont coupés tour à tour en deux. On obtient alors une partition du plan d'écran comprenant des cellules vides ou non vides.

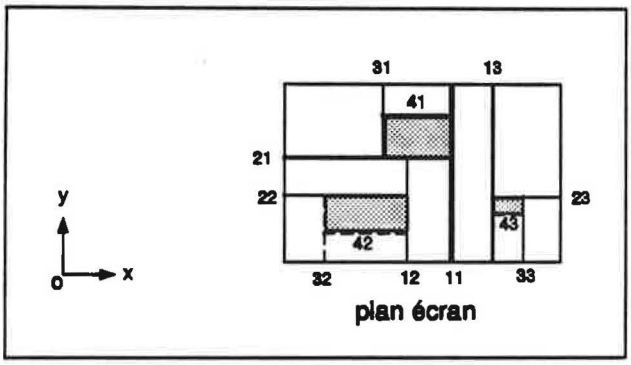

Figure 15 Partition du plan écran

On prolonge en profondeur au travers de la scène ces rectangles pour obtenir des cellules 3D.

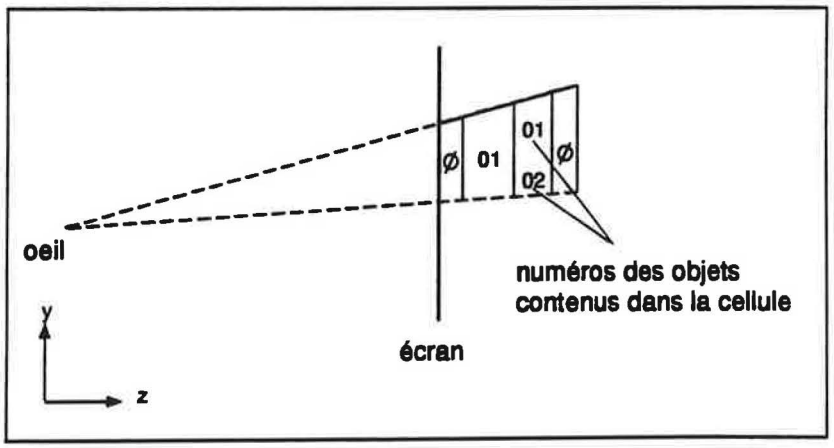

Figure 16 Obtention d'une cellule 3D à partir d'un rectangle de projection

Ensuite on effectue un découpage en profondeur en projetant les volumes englobants sur Oy,Oz et on associe à chacune des cellules 3D la liste des objets contenus.

Une des difficultés induite par ce découpage est la connexion entre les régions qui nécessite l'utilisation de dix pointeurs par cellule. Une autre complication provient du changement de repère à opérer dans la scène afin de permettre des calculs simples dans ces cellules pyramidales asymétriques.

Les résultats obtenus avec ce découpage sont excellents d'un point de vue temps de calcul. Dans les exemples présentés ce temps a été divisé par 30 par rapport à ceux où seuls les volumes englobants sont utilisés. Par contre, les besoins en mémoire deviennent exorbitants puisqu'ils sont multipliés par 10.

Malgré les qualités de cette méthode, sa complexité et sa gourmandise en ressources "mémoire" la rendent rarement utilisable en pratique.

Arquès et Ris [AR94], [Ris96] se servent également de la combinaison de voxels et de volumes englobants pour optimiser l'algorithme du lancer de rayon.

Lors d'une phase de précalcul, l'espace est découpé en voxels réguliers qui seront utilisés pour définir des boîtes englobantes grossières pour les objets de la scène. Ces volumes englobant seront affinés dynamiquement pendant le calcul de l'image.

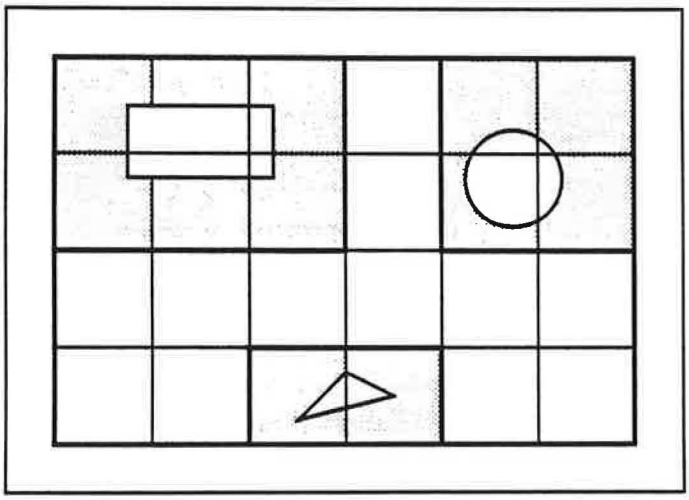

Figure 17 Construction de boîtes englobantes grossières à partir de voxels

Les rayons primaires sont envoyés dans un ordre précis; alternativement sont traitées la ligne la plus basse de gauche à droite et la colonne la plus à gauche du bas vers le haut. Lorsqu'une ligne ou une colonne coupe une boîte englobante sans intersecter l'objet associé, la taille de la boîte englobante est diminuée. Ainsi une fois que tous les rayons primaires ont été traités, on obtient une scène disposant de volumes englobants très fins.

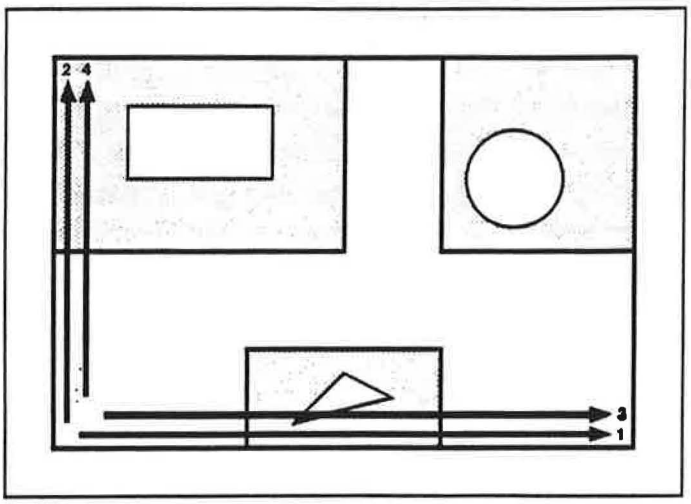

Figure 18 Ordre d'envoi des rayons primaires

Cette méthode permettant l'obtention de volumes englobants n'apporte pas d'amélioration lors d'un calcul sur une machine séquentielle, puisque l'obtention de boîtes englobantes fines est réalisée pendant la phase de calcul au lieu de la phase de précalcul. Par contre, sur des architectures multi-processeurs, l'algorithme proposé est particulièrement intéressant puisque cette construction de volumes englobants peut être réalisée en parallèle.

Chaque processeur va envoyer des rayons primaires à travers la zone de l'écran qui lui est associée. Les différents processeurs vont donc découvrir de façon précise une partie de la topologie de la scène. Les informations obtenues seront transmises régulièrement aux processeurs voisins afin qu'ils puissent raffiner leur connaissance des boîtes englobantes de la scène.

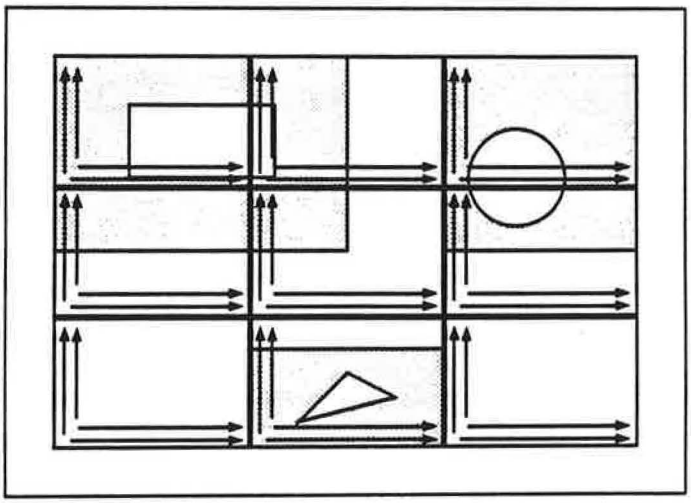

Figure 19 Parallélisation du raffinement des boîtes englobantes

1.3.6 Les techniques directionnelles

Les techniques directionnelles reposent sur le classement des différents rayons qui vont parcourir la scène dans des catégories bien définies. Cette organisation va permettre de ne proposer à chaque rayon qu'un nombre restreint d'objets à intersecter. Nous allons présenter les principales techniques directionnelles : le cube de direction, le "light buffer" et la classification de rayons dans un espace 5D.

1.3.6.1 Le cube de direction

Cette technique est proposée afin d'accélérer les calculs d'intersection des rayons primaires. Elle est inspirée de la méthode de l'hémicube utilisée en radiosité [CG85].

L'oeil de l'observateur est placé au centre d'un cube. A partir de cet oeil partent les rayons primaires qui vont couper les différentes faces du cube. A chaque rayon sont associées des coordonnées 2D (u,v) correspondant aux coordonnées de l'intersection entre le rayon et la face du cube coupée.

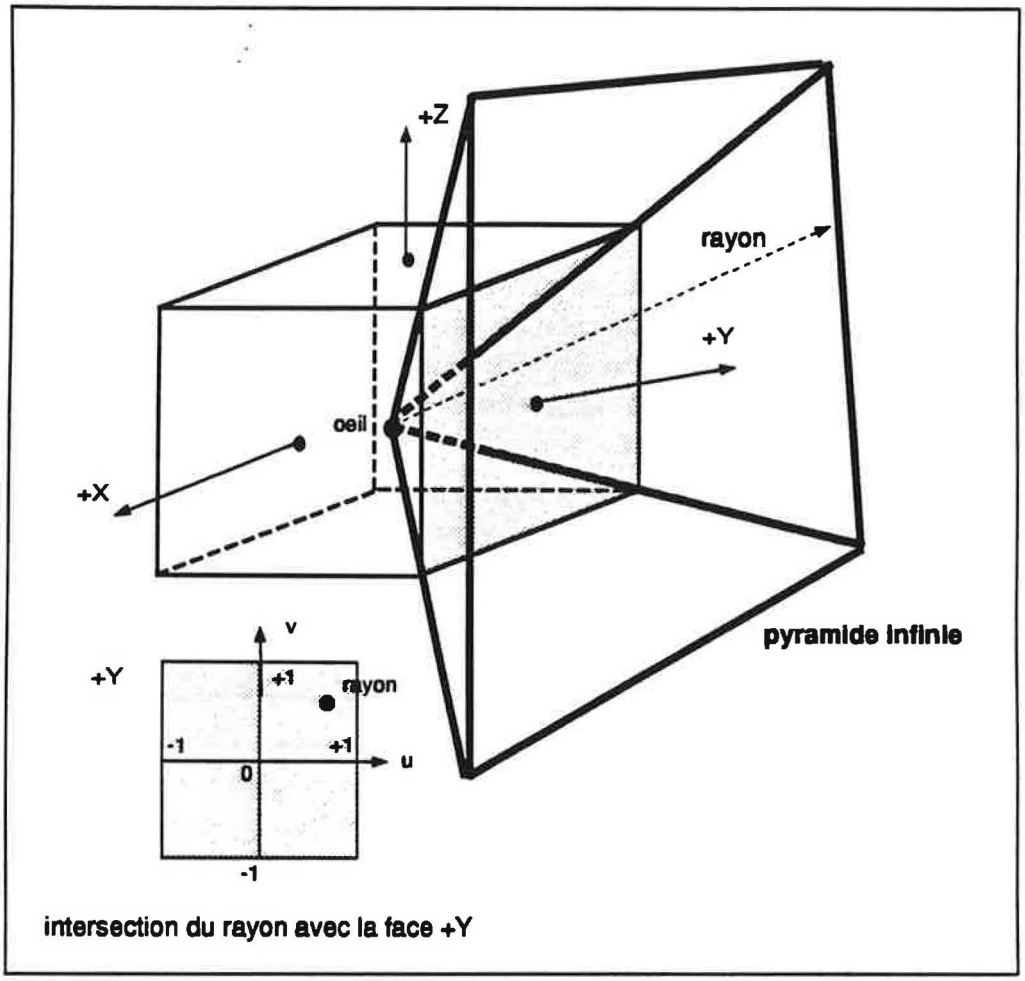

Figure 20 Le cube de direction

Ensuite à partir de l'oeil, on construit six pyramides infinies comprenant chacune une des faces du cube. Ces pyramides pourront également être subdivisées pour obtenir autant de cellules pyramidales infinies que souhaitées. Enfin, à chacune de ces cellules on va associer la liste des objets qui l'intersectent.

Lors du calcul des rayons primaires, on commence par déterminer la cellule à laquelle appartient chacun des rayons, ensuite il n'y a plus qu'à calculer les intersections entre les rayons et la liste des objets associés à leur cellule.

Suivant l'endroit, où l'écran est placé, certaines pyramides vont être privilégiées. Il apparaît donc que la construction de certaines pyramides aurait pu être évitée. Toutefois, la réalisation d'un pré-calcul indépendant de la position de l'écran permet sa réutilisation si l'écran est déplacé. C'est intéressant pour certaines applications pouvant demander des effets de zoom ou de vision à 360 degrés par exemple.

1.3.6.2 Le light buffer

Haines [HG86] reprend les idées du cube de direction, en s'intéressant au calcul des rayons d'ombrage provenant des sources.

Il propose de construire pour chaque source ponctuelle un cube de direction dont la source serait le centre. A chacune des cellules pyramidales obtenue, il associe la liste des objets pouvant cacher la source associée. Ainsi lors du calcul d'un rayon d'ombre, il suffit de déterminer à quelle cellule pyramidale le rayon appartient pour obtenir la liste des objets qu'il pourrait intersecter. Ensuite on cherche s'il y a intersection entre le rayon et un de ces objets. Comme on ne s'intéresse pas à la plus proche intersection, mais à une réponse binaire (la source est visible ou invisible), dès qu'une intersection est obtenue, le calcul est fini.

1.3.6.3 Classification de rayons

Arvo [AK87] propose également une méthode de subdivision de l'espace permettant de réduire le nombre d'intersections entre les rayons et les objets. L'algorithme s'appuie sur une division d'un espace en cinq dimensions.

En effet, bien que généralement un rayon soit caractérisé par six coordonnées (trois pour le point à l'origine et trois pour le vecteur de la direction), cinq coordonnées suffisent. Un rayon a cinq degrés de liberté : trois pour l'origine et deux pour la direction (deux angles en coordonnées sphériques). L'idée de Arvo est donc de découper cet espace 5D en hyper-voxels 5D, chacun d'eux comprenant une courte liste d'objets pouvant être intersectés par tous les hyper-points (ou rayons) qu'il comprend.

Une fois cette classification des rayons terminée, le nombre de calculs d'intersections entre les rayons et les objets qui devaient être effectués s'est considérablement réduit.

L'algorithme du lancer de rayon peut alors se décomposer en cinq étapes :

1. Trouver un volume englobant 5D de la scène et de tous les rayons possibles.
2. Subdiviser ce volume englobant en hyper-voxels disjoints.
3. Déterminer pour chacun de ces hyper-voxels la liste des objets qui peuvent être intersectés par n'importe lequel de ses rayons.
4. Associer à chaque rayon à calculer l'hyper-voxel auquel il appartient.
5. Déterminer l'intersection la plus proche entre le rayon et les objets candidats si elle existe. Alors de nouveaux rayons peuvent être générés, et il y a retour à l'étape précédente.

Arvo précise également que pour réduire davantage le nombre d'objets candidats dans chaque hyper-voxel, il faut opérer une division de l'espace 5D de manière récursive afin d'obtenir une hiérarchie d'hyper-voxels.

Les résultats en performance pure semblent plutôt encourageants puisqu'ils sont comparables à ceux de Glassner [Gla84] et Kay [KK86]. Cependant il n'est pas donné d'information sur les besoins en mémoire de cette méthode qui semble être particulièrement gourmande en raison du stockage nécessaire des hyper-voxels et de leur liste de rayons associée.

On peut également citer Simiakakis [Sim95] qui a travaillé sur cet algorithme. Il propose diverses accélérations, notamment une adaptation permettant sa parallélisation.

1.3.6.4 Bilan des techniques directionnelles

Ces méthodes directionnelles, bien qu'étant très efficaces d'un point de vue temps de calcul, sont très coûteuses en mémoire. En effet elles nécessitent la construction de structures assez lourdes.

La mémoire étant une ressource très souvent critique, ces techniques ne sont pas très utilisées en pratique.

1.4 Conclusion

Dans ce chapitre, nous avons défini les principes de l'algorithme du lancer de rayon. Ensuite nous avons présenté de nombreuses méthodes d'accélération séquentielles.

Parmi toutes ces techniques, les hiérarchies de volumes englobants et les découpages en octree et en voxels réguliers sont les plus utilisées. En effet, elles apportent des accélérations importantes sans induire des coûts "mémoire" prohibitifs.

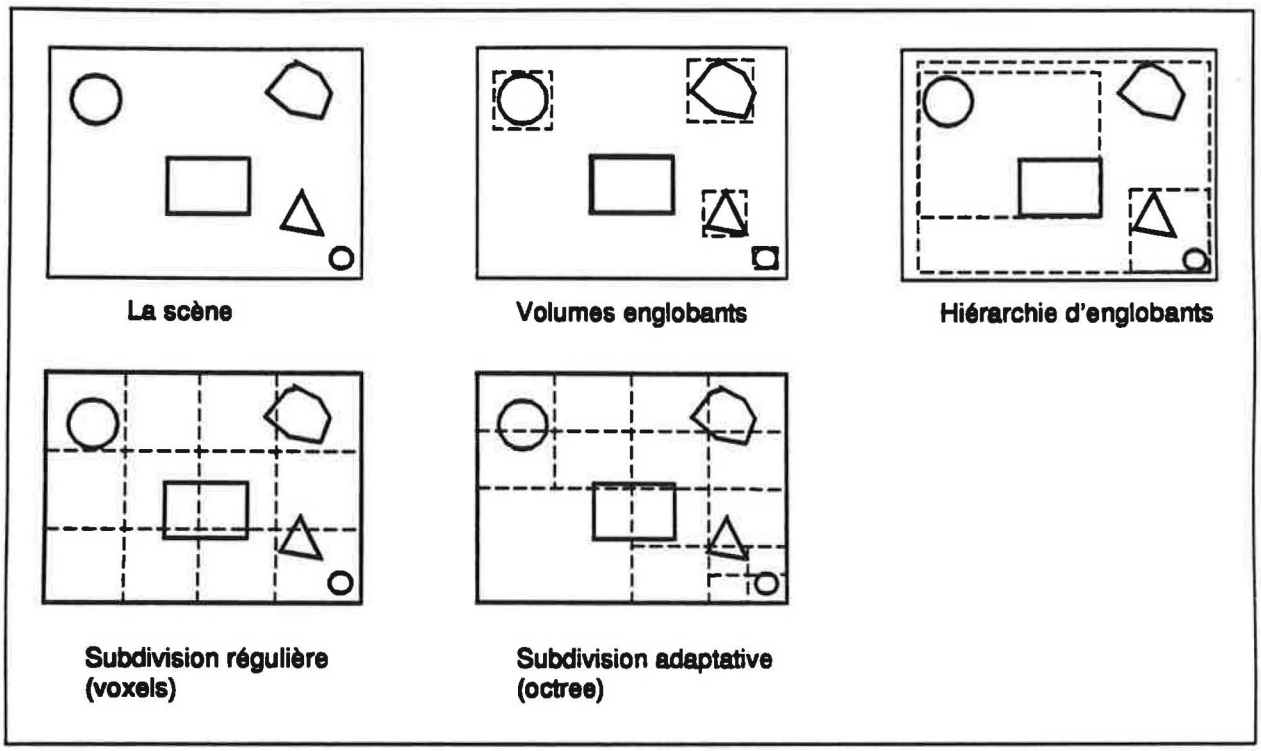

Figure 21 Les types d'accélération séquentielle les plus performants

Ces optimisations sont souvent complétées par un contrôle adaptable de l'arbre des rayons qui permet également de réduire le nombre de calculs.

Malgré l'utilisation de toutes ces méthodes, l'algorithme de lancer de rayon reste très coûteux en temps de calcul. Aussi avec le développement des environnements multi-processeurs, il semble naturel d'utiliser ces machines pour accélérer la production d'images de synthèse. Dans le chapitre suivant, nous allons donc présenter les différents algorithmes parallèles qui ont été proposés pour le lancer de rayon.

Chapitre 2
Les algorithmes parallèles de lancer de rayon

Dans notre travail, nous nous intéressons essentiellement aux algorithmes parallèles pour **machines à mémoire distribuée**, puisque c'est pour ce type de machines que nous souhaitons proposer un algorithme parallèle de lancer de rayon. Nous commençons par spécifier les différents types de machines parallèles. Puis nous présentons les algorithmes parallèles de lancer de rayon qui ont été proposés pour les machines parallèles MIMD à mémoire distribuée.

2.1 Les machines parallèles

Dans cette section nous allons présenter les principales caractéristiques des machines parallèles. D'abord nous présentons des classifications possibles, ensuite les différentes métriques utilisées pour évaluer leurs performances et enfin les modèles de programmation parallèle proposés.

2.1.1 Classifications

Avec le développement des machines parallèles, de nombreuses classifications sont proposées, mais elles recouvrent des concepts très différents. Nous présentons d'abord celle de Flynn [Fly72] qui est la plus utilisée et ensuite celle de MIMESI [DPT94] qui nous semble être la classification la plus précise et la plus complète.

2.1.1.1 La classification de Flynn

Cette classification se base sur le comportement du flot d'instructions par rapport au flot de données [Fly72].

En définissant,

1. le flot d'instruction (Instruction stream) comme la séquence d'instructions réalisées par la machine,
2. le flot de données (Data stream) comme la séquence de données appelée par le flot d'instructions,
3. et en utilisant S pour Single (unique) et M pour Multiple

Quatre architectures de base sont ainsi définies, les initiales anglaises sont conservées car elles sont consacrées par l'usage :

1. SISD : un seul flot d'instructions, un seul flot de données.
2. SIMD : un seul flot d'instructions, plusieurs flots de données.
3. MISD : plusieurs flots d'instructions, un seul flot de données.
4. MIMD : plusieurs flots d'instructions, plusieurs flots de données.

On retrouve ces architectures sur la figure suivante :

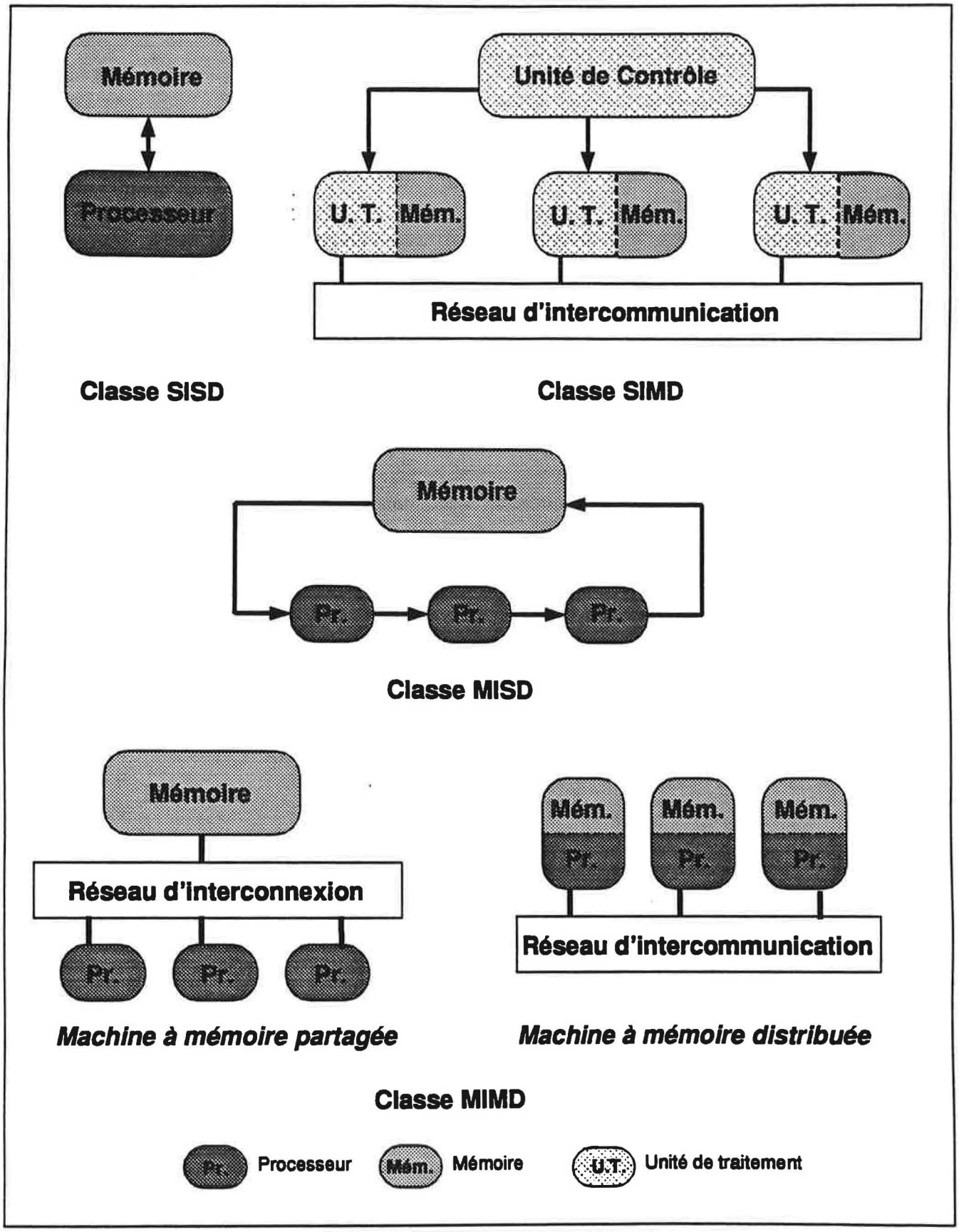

Figure 22 La classification de Flynn

La classe SISD Elle regroupe toutes les machines monoprocesseur sans considération quant à l'architecture interne du processeur.

La classe SIMD L'architecture de cette classe est composée de plusieurs unités de traitement, avec leur propre unité de mémoire pour stocker les opérandes. Ces unités de traitement sont supervisées par une seule unité de contrôle. Toutes les unités de traitement reçoivent la même instruction, mais l'exécutent sur des données distinctes.

Chaque unité de traitement exécute la même instruction au même moment. Une telle machine est donc synchrone, l'organe de contrôle unique donnant la cadence.

La classe MISD Cette architecture impose un cheminement aux opérandes. Ainsi, le flot de données va être successivement traité par les différentes unités. Aussi, seule la première unité voit le flot de données initiales, alors que les autres traitent des dérivées de ce flot initial.

Ceci correspond à la description habituelle d'une architecture en pipeline.

La classe MIMD Cette classe comprend deux architectures bien distinctes qui ont une caractéristique commune : la possibilité de fonctionnement asynchrone des processeurs.

Machine à mémoire partagée

Cette architecture comprend un certain nombre de processeurs reliés à une mémoire commune, où chaque processeur peut exécuter un code indépendamment des autres. Les échanges d'informations entre les processeurs se font à l'aide de la mémoire (en général, lecture simultanée et écriture exclusive).

Les machines appartenant à cette classe possèdent un faible nombre de processeurs relativement puissants. En effet, les accès à la mémoire deviennent très peu performants, en particulier à cause de la gestion de la cohérence des caches.

Machine à mémoire distribuée

Afin de réaliser des machines massivement parallèles, il faut choisir des architectures où la mémoire est décentralisée. Ainsi, une machine à mémoire distribuée est une machine où chaque processeur possède sa propre mémoire et où les échanges d'information entre les processeurs se font par l'intermédiaire de messages qui transitent dans un réseau d'interconnexion.

Sur une telle machine, le temps d'exécution d'un code parallèle dépend donc également du temps de latence nécessaire à l'envoi ou à la réception d'un message, du débit et de la géométrie du réseau utilisé.

Les machines SPMD Les machines de type SPMD (Single Program Multiple Data) sont des machines dérivées du concept de machines MIMD. Ce type d'architecture impose deux restrictions par rapport au modèle MIMD :

1. Une gestion statique des processus : tous les processus de l'application sont lancés simultanément.
2. Tous les processus exécutent le même programme sans qu'il y ait synchronisation entre ces exécutions par le système.

Toutefois il est généralement aisé d'exécuter un code MIMD sur une machine SPMD. En effet, il suffit souvent de concaténer tous les programmes d'un code MIMD en un unique exécutable. Chaque processus de la machine SPMD va pouvoir exécuter dans cet unique code une portion différente grâce au test par exemple de son numéro de processus. Le prix à payer n'est donc que celui d'un accroissement de la taille mémoire de l'exécutable.

Voici par exemple, la transformation d'un code MIMD de type maître-esclaves en un code adapté à une machine SPMD.

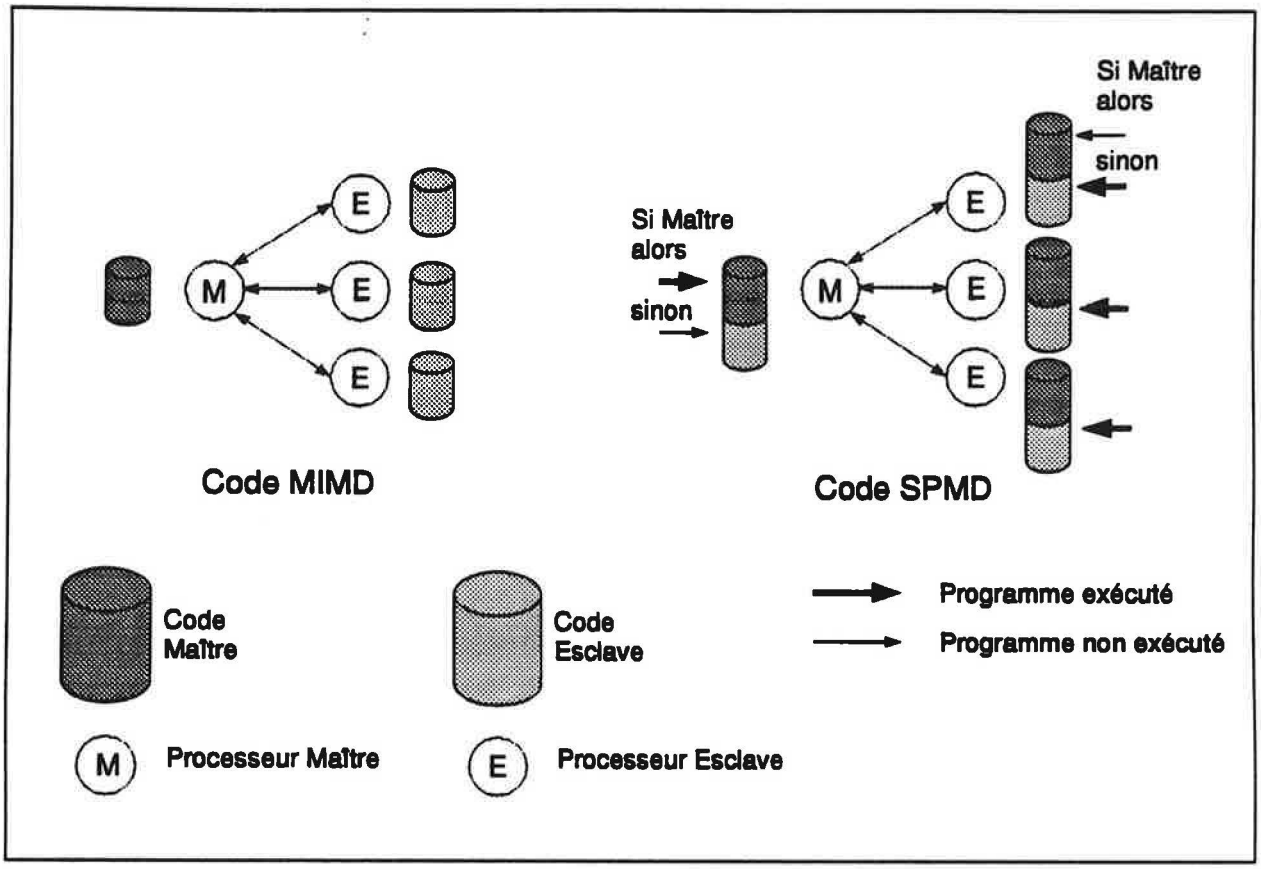

Figure 23 Différences entre codes MIMD et SPMD

Par contre si le code MIMD est basé sur une création non bornée de processus en cours d'exécution, son implémentation sur machine SPMD est alors impossible. Cependant, la plupart du temps, il est possible d'évaluer une limite supérieure du nombre de processus. En les créant tous à l'initialisation, la machine SPMD pourra exécuter un tel code.

2.1.1.2 La classification de MIMESIS

MIMESIS est un atelier de modélisation des différents types de calculateurs [DPT94]. Ce travail a conduit à la création d'une classification regroupant de nombreux concepts. En effet, elle représente une synthèse des classifications passées et emploie les notations les plus répandues.

Cette classification est présentée dans la figure suivante :

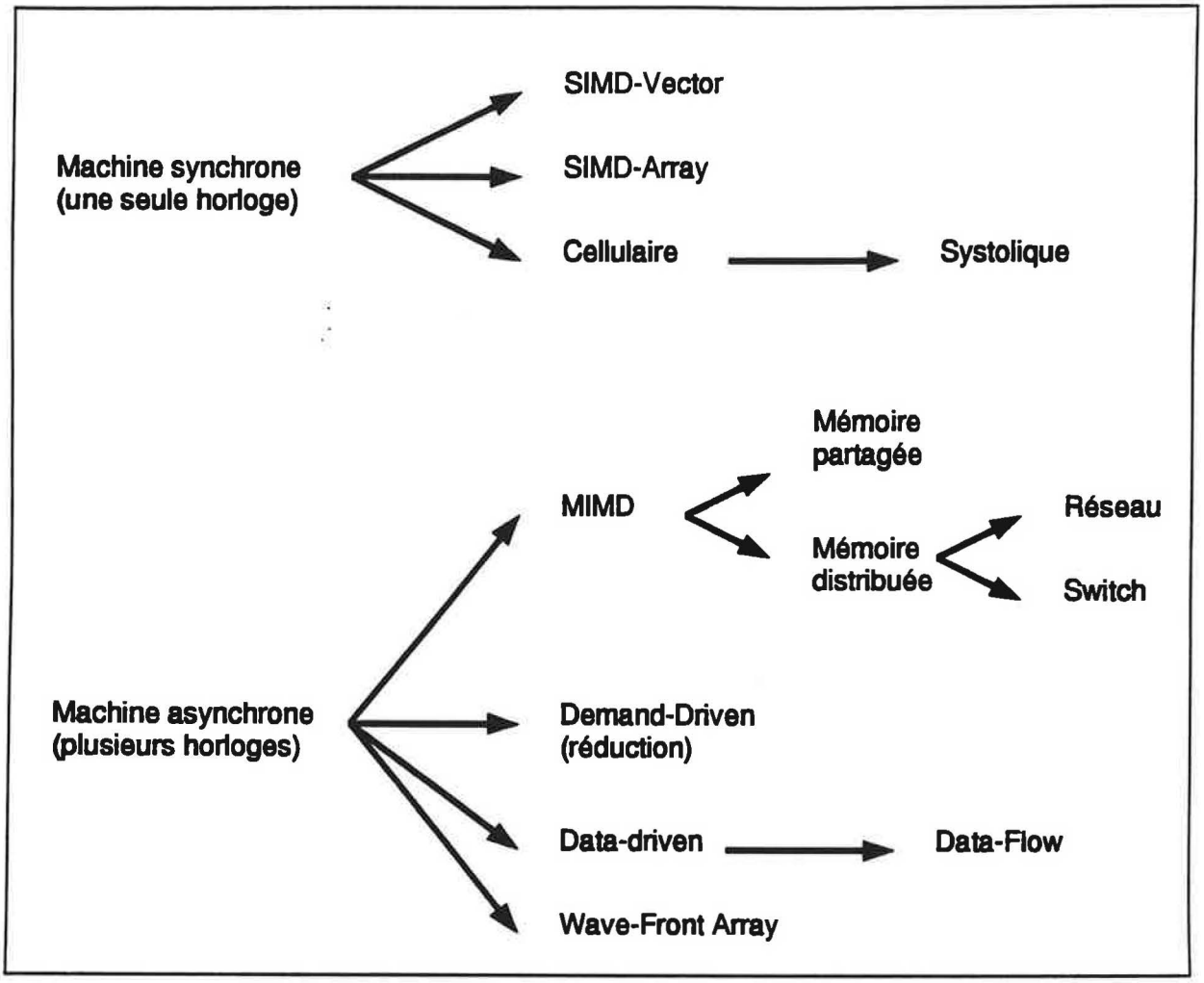

Figure 24 La classification de MIMESIS

Dans cette classification, on retrouve les principales classes définies par Flynn. Nous nous contentons ici de préciser celles qui apportent une nouveauté.

La classe SIMD-Vector Un processeur traite des vecteurs en pipeline.

La classe SIMD-Array Un processeur de contrôle charge les instructions et les transmet à un ensemble de processeurs esclaves qui les exécutent en même temps sur des données différentes. On parle également d'approche "data parallel".

La classe machine systolique Les processeurs travaillent en pipeline multidimensionnel sur un flux de données arrivant à cadence constante.

Les classes MIMD à mémoires distribuées Dans cette classification, le moyen d'échanger des messages entre les processeurs est spécifié. Il peut être de deux types :

1. "Switch" : par l'intermédiaire d'un commutateur.
2. "Network" : par l'intermédiaire d'un réseau, dont les processeurs sont les sommets, en suivant un certain protocole.

La classe "demand-driven" Les processeurs se partagent les instructions nécessaires à l'obtention du résultat final. Les instructions (noeuds de la machine) sont activées seulement lorsque leurs résultats sont requis par d'autres instructions.

La classe "data-flow" Les processeurs se partagent les sommets d'un graphe de tâches. Ils traitent et transmettent les données au fur et à mesure qu'elles sont disponibles.

La classe "wavefront array" Cette classe reprend l'idée des machines systoliques mais avec un flot de données asynchrone et une communication interprocesseurs asynchrone. Le réseau de processeurs constitue ainsi le support de propagation d'"ondes de données" : plusieurs fronts d'ondes successifs franchissent à leur rythme le réseau.

Bien que cette classification nous apparaisse comme la plus complète et la plus précise, la classification de Flynn étant la plus connue, il nous semble difficile de ne pas l'utiliser. Dans la suite de ce travail, nous utilisons donc la classification de Flynn pour définir les différentes architectures rencontrées. Cette définition est éventuellement complétée par celle de MIMESIS si cela nous semble nécessaire.

2.1.2 Critères de performance

Afin d'évaluer les performances d'un algorithme parallèle sur une machine parallèle, deux mesures sont couramment utilisées : l'accélération et l'efficacité. Il nous semble indispensable d'en introduire une troisième, **la surcapacité**, afin d'évaluer la taille maximale des problèmes que l'algorithme parallèle peut résoudre. Enfin, il existe également des critères d'évaluation s'appuyant sur des caractéristiques locales à chacun des processeurs.

2.1.2.1 Accélération

Ce critère répond à une des premières préoccupations de la parallélisation, l'accélération des performances de traitement d'une application.

L'accélération consiste donc à calculer le rapport du temps d'exécution sur un processeur (T_{seq}) - exécution séquentielle - avec le temps d'exécution en parallèle (T_{par}) sur p processeurs identiques. Ainsi, on la définit par :

$$Acc_p = \frac{T_{seq}}{T_{par_p}} \tag{4}$$

L'accélération est bornée de telle sorte que :

$$\forall p,\ 1 \leq Acc_p \leq p \tag{5}$$

Un problème est la définition du "temps d'exécution en séquentiel". En effet, généralement l'élaboration d'un algorithme dépend du choix du mode d'exécution. Le temps d'exécution en séquentiel, obtenu par exécution d'un code parallèle sur un processeur, est souvent surévalué car l'algorithme a été "pensé" de manière parallèle. Afin d'être plus rigoureux, il serait souhaitable d'utiliser, pour mesurer le temps séquentiel, l'algorithme le plus performant en séquentiel. Hélas, pour le cas du lancer de rayon, il n'existe pas d'algorithme de référence, aussi dans ce travail nous nous contentons de l'algorithme parallèle utilisé sur un unique processeur.

2.1.2.2 Efficacité

L'accélération d'un traitement ne doit pas être le seul critère pour mesurer les performances d'un algorithme parallèle. En effet, toute augmentation du nombre de processeurs a comme conséquence une augmentation de la complexité du réseau. Aussi, il est intéressant de mesurer le rendement du réseau et voir s'il est optimal ou pas. Pour cela, on définit l'efficacité d'un réseau de p processeurs identiques par :

$$Eff_p = \frac{T_{seq}}{p \times T_{par_p}} = \frac{Acc_p}{p} \tag{6}$$

Avec :

$$\forall p,\ Eff_p \leq 1 \tag{7}$$

2.1.2.3 Surcapacité

Les critères précédents sont pertinents sur des machines parallèles à mémoire partagée. Cependant sur des machines parallèles à mémoire distribuée, ils sont insuffisants. Ces métriques ne rendent pas compte du gain en taille des problèmes pouvant être traités grâce à la mise en commun des mémoires des différents processeurs de la machine. Ce critère peut être crucial pour de nombreuses applications, car l'aptitude à pouvoir traiter un problème donné est prépondérante sur d'éventuelles performances sur les temps de calcul.

Aussi, nous introduisons une nouvelle métrique : la "surcapacité" ($Scap_p$) pour p processeurs. La "surcapacité" permet d'évaluer le gain dans la taille maximale des problèmes que l'on peut aborder avec p processeurs par rapport aux problèmes abordables en séquentiel.

En posant

$CapaciteMax_{seq}$ la mémoire maximale disponible pour un problème pouvant être résolu en séquentiel,

$CapaciteMax_{par_p}$ la mémoire maximale disponible pour un problème pouvant être résolu par p processeurs.

On définit la surcapacité d'un algorithme parallèle par :

$$Scap_p = \frac{CapaciteMax_{par_p}}{CapaciteMax_{seq}} \tag{8}$$

Avec :

$$\forall p,\ 1 \leq Scap_p \leq p \tag{9}$$

2.1.2.4 Métriques locales

En dehors des métriques permettant d'évaluer les performances d'un algorithme parallèle dans sa globalité, il existe des métriques permettant d'étudier le comportement des processeurs d'une machine parallèle pris individuellement. Celles-ci permettent souvent d'expliquer les comportements d'une application parallèle.

Nous nous limitons à la définition du déséquilibre de charge noté *Deq*. Pour une architecture comprenant p processeurs où chaque processeur i nécessite un temps T_i pour effectuer ses calculs, le déséquilibrage de charge *Deq* s'exprime par :

$$Deq = \frac{\left(Max_{0\leq i<p}(T_i) - Min_{0\leq j<p}\left(T_j\right)\right)}{Max_{0\leq i<p}(T_i)} \qquad (10)$$

Plus le déséquilibrage de charge *Deq* est proche de 0, plus l'équilibrage des tâches est bon.

2.1.3 Les modèles de programmation parallèle

Le but d'un modèle de calculs parallèles est de donner un cadre dans lequel il est possible de décrire et d'analyser les algorithmes parallèles, permettant ensuite de les implémenter sur des machines parallèles. Nous allons présenter les quatre modèles principaux.

2.1.3.1 Les modèles PRAM et "data parallel"

Le Parallel Random Access Machine (PRAM) est un modèle de programmation dans lequel un ensemble de processeurs communiquent entre eux en utilisant une mémoire partagée qui est accédée de façon synchrone. Chaque processeur possède une mémoire locale et exécute son propre programme. C'est le modèle le plus utilisé en raison de sa mise en oeuvre aisée. Son inconvénient majeur est la possibilité de goulots d'étranglement lors de l'accès à la mémoire partagée.

Un modèle très proche est le data parallel. Un algorithme développé sur ce modèle est une succession d'étapes parallèles, pendant lesquelles des opérations parallèles sont exécutées sur un grand nombre de données. Pour cela il faut faire deux hypothèses : l'utilisation d'une mémoire partagée et l'exécution synchrone de chaque étape parallèle.

On peut citer comme langage data parallel HPF (High Performance Fortran) [For92] qui est une extension du Fortran 90.

2.1.3.2 Le modèle à mémoire partagée

La caractéristique principale des modèles précédents est l'exécution synchrone des instructions qui rend la programmation facile, mais entraîne une perte d'efficacité. Le modèle à mémoire partagée permet par contre un accès totalement asynchrone à la mémoire, mais c'est au programmeur de gérer la protection des données partagées et les éventuels points de synchronisation.

2.1.3.3 Le modèle par échange de messages

Pour ce modèle, on suppose que chaque processeur dispose d'une mémoire locale et qu'aucune forme de mémoire partagée n'est disponible. Les processeurs communiquent entre eux explicitement en envoyant et en recevant des messages, ce sont donc des communications point à point. Le coût de communication dépend de la longueur du chemin emprunté par le message pour parvenir à sa destination. La topologie du réseau de communication joue donc un rôle important.

Le principal avantage de ce modèle est la maîtrise par le code des échanges de messages, puisqu'il y participe de façon active. En revanche il est possible d'obtenir des "blocages mortels" (deadlocks) si les messages sont incorrectement gérés.

Actuellement PVM (Parallel Virtual Machine) est la bibliothèque d'échanges de messages la plus utilisée, mais MPI (Message Passing Interface) devrait rapidement la détrôner et devenir le nouveau standard (voir §3.1.2.3).

2.1.3.4 Le modèle BSP

Le Bulk Synchronous Parallelism [Val90] est un modèle où un code parallèle est représenté comme un code séquentiel dans lequel interviennent de temps en temps des communications. Ce modèle regroupe toutes les communications dans un unique appel. Les phases de calcul séquentiel et de communications sont nettement séparées :

1. Lors de la phase séquentielle ("superstep"), le programme ne fait que des calculs locaux et demande à faire des communications lors de la prochaine phase de communication. Celles-ci peuvent être de type échange de messages ou PRAM.
2. Quand le code arrive à la phase de communication, il y a une barrière de synchronisation et les communications demandées sont réalisées.

L'avantage est donc une gestion aisée des communications et la garantie de la cohérences des données globales qui ne peuvent pas être modifiées lors d'un "superstep". Toutefois l'usage des barrières de synchronisation nuit à l'optimisation du code.

2.2 Algorithmes pour machines parallèles à mémoire distribuée

Comme le but de notre étude est de réaliser un noyau parallèle pour l'algorithme du lancer de rayon sur des machines à mémoire distribuée, nous nous intéressons surtout aux propositions qui ont été faites pour ce type de machines.

La parallélisation de l'algorithme du lancer de rayon sur une machine parallèle à mémoire distribuée dépend d'abord de la nécessité de distribuer ou non la scène sur l'ensemble des processeurs.

Dans le cas où toute la scène peut résider dans la mémoire de chacun des processeurs, la parallélisation s'opère par distribution de l'image à calculer sur l'ensemble des noeuds de la machine.

Par contre, si la scène doit être distribuée sur l'ensemble des processeurs, plusieurs stratégies peuvent être envisagées :

1. Une coopération entre processeurs exécutant des tâches complémentaires.
2. Un traitement avec échanges de rayons.
3. Un traitement avec échanges de données.

Ensuite nous étudions les algorithmes implémentés sur les réseaux de stations de travail. Et enfin nous présentons d'autres algorithmes de parallélisation pour des applications dont la problématique est proche de celle du lancer de rayon.

2.2.1 Traitement avec base de données dupliquée

Quand l'ensemble de la scène peut être dupliqué sur chaque processeur, la parallélisation se fait très naturellement par partage des pixels à calculer sur l'ensemble des processeurs. On se trouve dans une situation comparable à la parallélisation de l'algorithme sur machine parallèle MIMD à mémoire partagée. Chaque processeur calcule de façon indépendante une partie de l'image.

La qualité de l'algorithme dépendra de l'équilibrage des tâches entre les processeurs. Afin que la charge de travail soit également répartie sur la machine parallèle, il est important de choisir une stratégie efficace de répartition des pixels.

Deux voies ont été explorées : la distribution statique et la distribution dynamique.

2.2.1.1 Distribution statique

Les premières partitions d'une image sur des processeurs étaient des partitions par blocs, qui avaient un équilibrage de charge très médiocre.

Murakami et al. [MHI88] ont étudié l'utilisation de distributions statiques allouant une partie de l'écran à chacun des p processeurs d'une machine. Ils ont ainsi comparé le partage par point, où chaque processeur prend un pixel tous les $\sqrt{p}$ pixels en horizontal et en vertical, et celui par ligne, où chaque processeur prend une ligne toutes les p lignes.

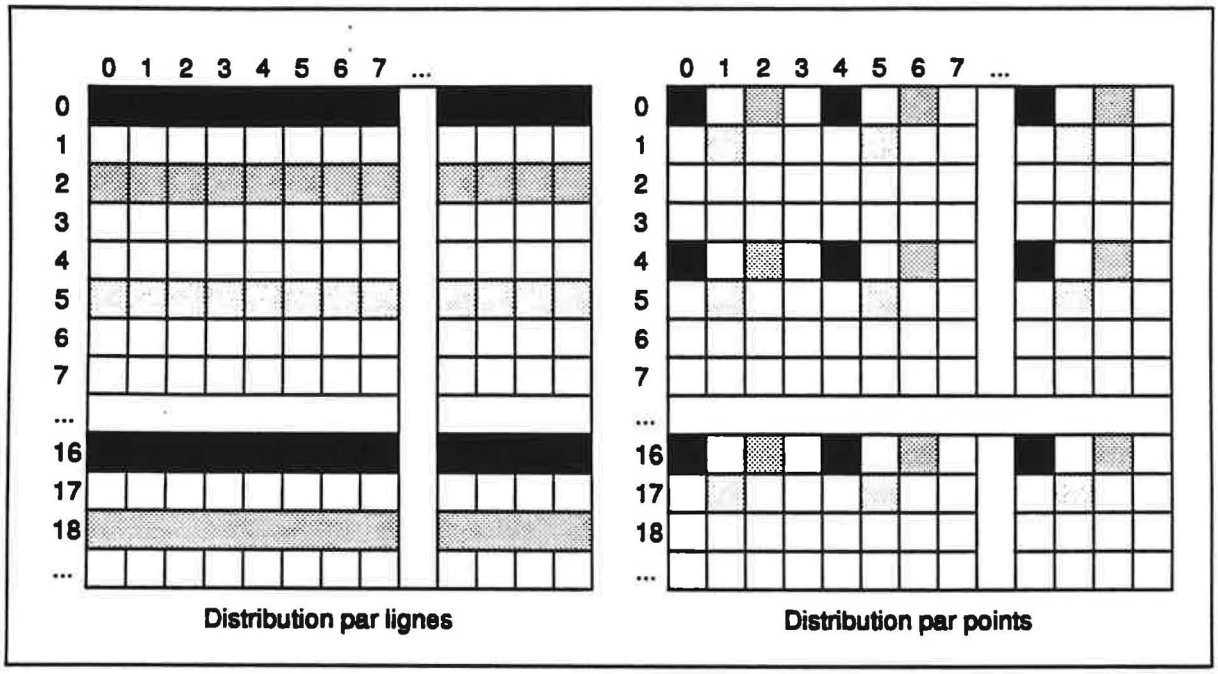

Figure 25 Distributions statiques de pixels sur 16 PEs

Avec leur scène de test, ils obtiennent un déséquilibre de charge atteignant 20 % pour la distribution en lignes et seulement 5% pour celle par points. Ainsi en passant de 16 à 64 processeurs, l'accélération est de 3.96 pour les points, ce qui est proche de l'accélération optimale, alors qu'elle est seulement de 3.65 pour les lignes. Cela laisse présager d'une baisse importante des performances pour la distribution en lignes lors de l'utilisation d'un plus grand nombre de processeurs.

Bien que les performances obtenues par le partage par point semblent idéales, il faut noter que cette distribution perd la cohérence des pixels, ce qui empêche d'en tirer parti pour accélérer l'algorithme de lancer de rayon.

Une solution, qui garderait la cohérence des pixels et un bon équilibrage statique, serait de procéder à un sous-échantillonage de l'image, et ainsi déterminer les coûts a priori du calcul des différents pixels. Ainsi à partir de ces informations, on pourrait réaliser une distribution équilibrée et cohérente des pixels. Toutefois, il faudrait vérifier que le coût de ce précalcul n'est pas prohibitif par rapport à l'accélération obtenue.

2.2.1.2 Distribution dynamique

Afin d'améliorer encore l'équilibrage de charge et de permettre la conservation de la cohérence des pixels, des équilibrages dynamiques ont été proposés. Les architectures les plus utilisées sont de type maître-esclaves, mais d'autres solutions semblent permettre un meilleur comportement quand le nombre de processeurs augmente.

Architecture maître-esclaves

Reeth et al. [RLF92] proposent l'utilisation d'une architecture maître-esclaves. L'image est subdivisée en régions de $n \times n$ pixels et un processeur maître possède ces blocs d'images.

Le maître commence par distribuer un bloc à chaque processeur esclave. Une fois qu'un esclave a terminé de calculer son bloc, il envoie ses résultats au maître et lui demande une nouvelle région à calculer. Il y a donc des échanges entre le maître et ses esclaves tant que le maître possède des pixels à distribuer. L'algorithme se termine quand le maître possède l'image calculée en totalité. Il peut alors l'afficher. Par ce mécanisme, il y a donc un équilibre des tâches : plus un processeur a des calculs simples à effectuer, plus le nombre de calculs traités sera important.

Afin d'augmenter encore la cohérence dans le traitement des pixels, Green et Paddon [GP90] proposent que la charge allouée soit la plus cohérente possible avec la charge qui vient d'être calculée.

Prenons le cas où on dispose de deux processeurs (*PE0* et *PE1*) et un partage en 3×3 blocs de pixels. On commence par distribuer le *bloc 1* sur le PE0 et le *bloc 2* sur le *PE1*. Mais quand le *PE0* demandera une nouvelle charge, on va plutôt lui proposer le *bloc 4* au lieu du *bloc 3* comme dans les algorithmes classiques. En effet, on peut supposer que les pixels des *blocs 1* et *4* ont une plus grande cohérence que ceux des *blocs 1* et *3*.

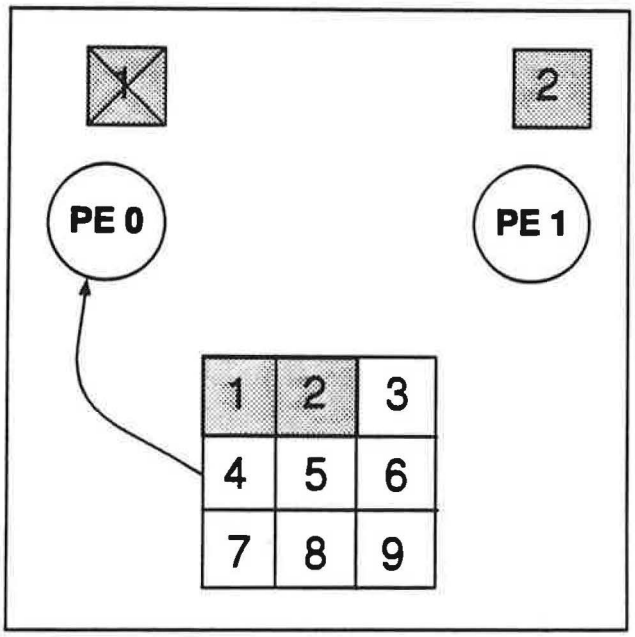

Figure 26 Distribution dynamique cohérente

Pandzic [PMR95] souhaite résoudre le dilemme suivant : d'une part pour profiter au maximum de la cohérence des rayons, les blocs alloués par le processeur maître doivent être de la plus grande taille possible; d'autre part, pour assurer un équilibrage fin, les portions allouées doivent être petites. Il propose d'allouer des tâches dont la taille diminue dynamiquement au cours du temps. Ainsi il commence par allouer des blocs de grandes tailles et quand l'exécution se termine, les blocs sont plus petits afin d'avoir un équilibrage plus fin. C'est un maître qui distribue ces tâches, celles-ci étant proportionnelles au travail restant et inversement proportionnelles au nombre de processeurs. De plus, Pandzic note que le processeur maître a une tâche relativement légère, aussi afin d'utiliser ce processeur au maximum, il va également lui attribuer des tâches de calcul. Grâce à l'implémentation d'un mécanisme d'interruption, le processeur maître va pouvoir avoir un rôle d'esclave et à chaque interruption reprendre son rôle de maître.

Les résultats obtenus sont plutôt bons : une efficacité de l'ordre de 80% pour 16 PEs. Il est intéressant de noter que des résultats sont donnés pour des configurations massivement parallèles et qu'ils restent corrects : l'efficacité est encore supérieure à 30% pour 256 PEs.

Il apparaît toutefois que, sur une machine massivement parallèle, l'utilisation d'un unique maître à la réception de l'ensemble des requêtes des esclaves provoque un goulot d'étranglement qui réduit de façon notable l'efficacité. Il semble qu'une architecture maître-esclaves ne soit envisageable pour des environnement massivement parallèles que s'il y a plusieurs maîtres.

On peut se rapporter aux travaux de Kobayashi et al. [KNK+88] (§2.2.3) qui utilisent une architecture comprenant des grappes, chacune d'elles ayant un maître avec une partie de l'image et des esclaves.

Architecture sans maître

Badouel [Bad90] souhaite également profiter de la cohérence des rayons, mais aussi permettre une implémentation efficace sur machines massivement parallèles. Il commence par réaliser une distribution statique par bloc. Ensuite il propose un équilibrage dynamique de charge. Quand un noeud a fini ses calculs, il demande un nouveau groupe de pixels à un noeud voisin. La demande est transmise de voisin en voisin en parcourant un anneau tant qu'elle n'est pas satisfaite. Si la demande revient insatisfaite, le noeud sait que le calcul de l'image est fini.

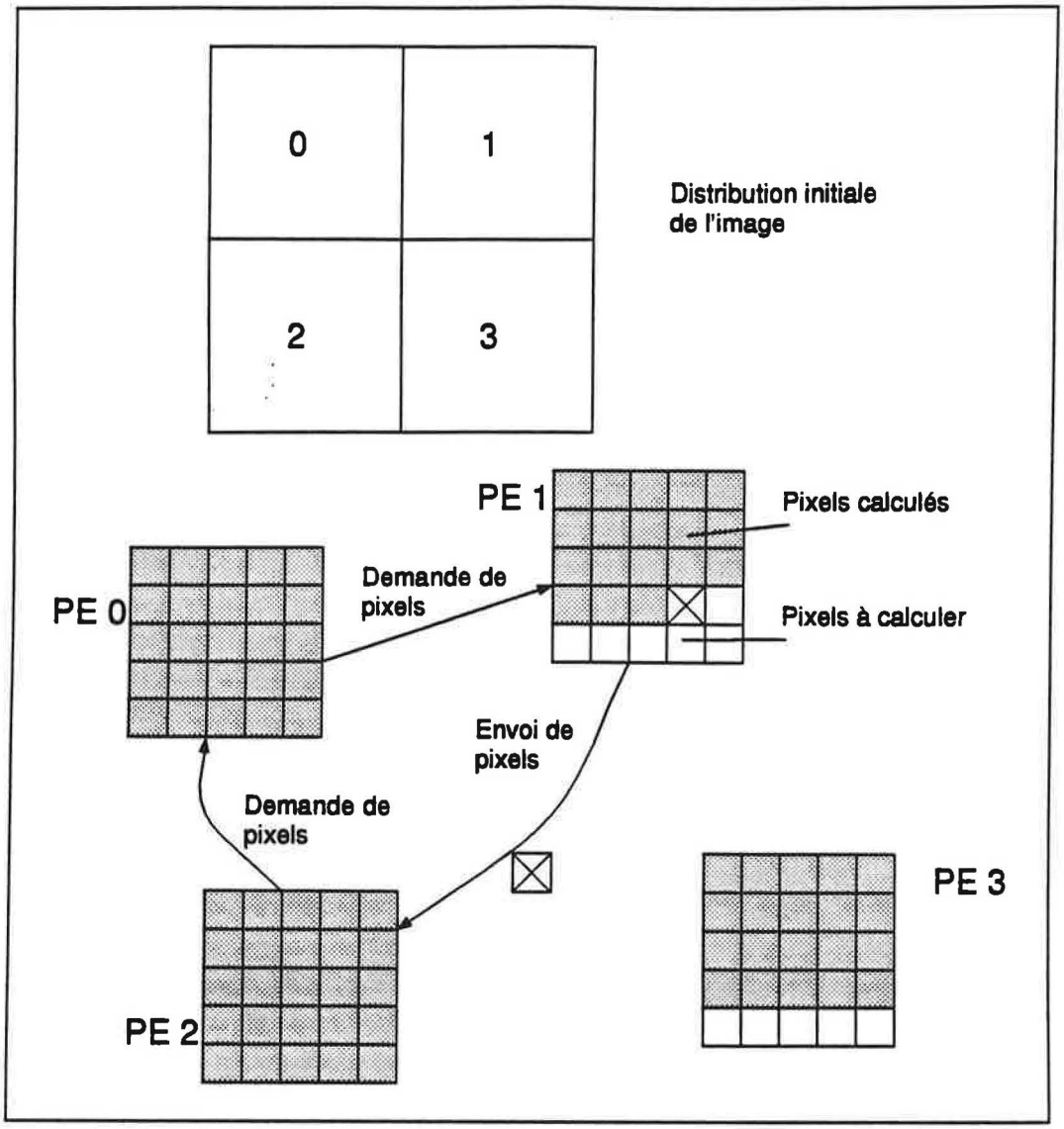

Figure 27 Distribution statique par blocs des pixels et équilibrage dynamique en anneau

La taille optimale du petit groupe de pixels envoyé est un compromis entre le moindre coût de communication (gros groupe) et le meilleur équilibre (petit groupe). Expérimentalement, la valeur obtenue est de 3×3 pixels. Les résultats obtenus permettent d'envisager un équilibrage satisfaisant dans un environnement massivement parallèle.

2.2.1.3 Conclusion

Il apparaît que, pour ce type d'algorithme parallèle, il faut tout d'abord déterminer si on a la possibilité d'exploiter la cohérence des pixels. Dans la négative, une distribution statique par points semble suffisante. Par contre, si l'exploitation de la cohérence des rayons est envisageable, l'implémentation d'un équilibrage dynamique semble indispensable. Un équilibrage basé sur une architecture maître-esclave peut suffire pour une machine possédant peu de processeurs. Par contre, pour des machines massivement parallèles, d'autres stratégies doivent être mises en oeuvre, comme la suppression du maître ou l'utilisation de hiérarchies de maîtres.

L'inconvénient majeur de cette approche demeure toutefois dans le fait qu'elle ne permet pas de traiter des scènes de grande taille.

2.2.2 Traitement coopératif

Au lieu d'avoir des processeurs qui exécutent tous le même algorithme, on peut partager l'algorithme en tâches distinctes qui seront exécutées sur des processeurs différents.

Barett [Bar90] s'est attaché à décomposer le calcul de construction de l'arbre d'un rayon. Il l'a partagé en quatre composantes :

1. La tâche des rayons : initialisation des rayons et calculs de luminosité des pixels.
2. La tâche d'octree : intégration de l'octree pour la recherche du prochain voxel.
3. La tâche d'intersections : calculs d'intersections rayon-objet.
4. La tâche d'arbre : coordination des travaux des autres tâches et calculs des rayons de réfraction et de réflexion.

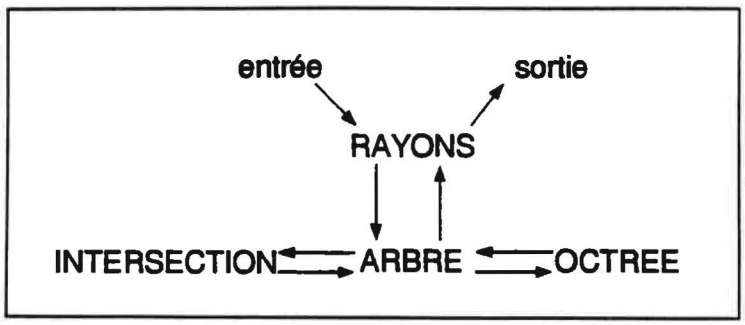

Figure 28 Traitement coopératif de 4 tâches

Les rayons primaires et leurs arbres de construction sont initialisés par la tâche RAYONS. Les structures des arbres sont ensuite transmises à la tâche ARBRE.

Pour chaque rayon, cette tâche demande à la tâche OCTREE de lui présenter le prochain voxel non nul traversé par le rayon. Après réception de ce voxel, le rayon et le voxel sont transmis à la tâche INTERSECTION, qui va calculer l'intersection éventuelle entre le rayon et l'objet contenu dans le voxel.

Ce mécanisme de demande de voxel et de calcul d'intersection s'effectue sur chaque rayon tant qu'il ne sort pas de la scène. La tâche ARBRE crée au passage les rayons de réfraction et de réflexion nécessaires.

Quand un rayon quitte la scène, son arbre du rayon est transmis à la tâche RAYONS qui va calculer les caractéristiques du pixel associé.

Un point crucial pour obtenir de bonnes performances est la gestion de l'équilibre des tâches. Trois types d'équilibrage ont été testés :

Un équilibrage statique.
Un équilibrage dynamique périodique.
Un équilibrage dynamique événementiel.

L'équilibrage statique consiste à affecter à chaque processeur une des quatres tâches en essayant d'assurer le meilleur équilibrage posssible. Les résultats présentés par l'auteur font apparaître qu'en moyenne la distribution des tâches doit être la suivante :

1 % des tâches pour RAYONS.
6 % des tâches pour ARBRE.
35 % des tâches pour OCTREE.
51 % des tâches pour INTERSECTION.

L'équilibrage dynamique repose sur la réaffectation de processeurs à d'autres tâches au cours de l'exécution. Les processeurs à cours de calcul vont se voir allouer les tâches qui créent des ralentissements au sein de l'exécution du code.

Une première forme d'équilibrage dynamique est l'utilisation d'un équilibrage dynamique périodique. Régulièrement, une inspection de la charge de chacun des processeurs est réalisée. Sa mise en place n'a permis qu'un gain en temps de 2,5% par rapport à l'algorithme n'utilisant qu'un équilibrage statique.

Un équilibrage dynamique événementiel est le deuxième type d'équilibrage dynamique proposé. Chaque processeur dispose d'une liste des calculs en attente d'être effectués. Aussi une fois que cette liste dépasse une certaine limite, le processeur est surchargé, un mécanisme de rééquilibrage est mis en route. Cette méthode, beaucoup plus souple que la précédente, donne de meilleurs résultats : une accélération de 12,5%.

Afin de déterminer le bien fondé de son approche coopérative, Barett a comparé les résultats obtenus avec ceux d'une parallélisation classique où l'image est subdivisée sur le réseau de processeurs. Cette dernière donne des temps de calcul deux fois plus courts.

Cela peut s'expliquer d'une part par le nombre élevé de messages échangés dans l'approche coopérative. D'autre part, l'équilibrage de charge est beaucoup plus facile à assurer avec l'algorithme à partition d'image. En effet, une répartition équilibrée des tâches pour l'algorithme coopératif nécessite un nombre de processeurs supérieur à une centaine : la tâche la plus légère (RAYON) ne nécessite que 1% des temps de calcul totaux.

Cette étude a été réalisée sur une machine à mémoire partagée. Elle est réalisable sur machine à mémoire distribuée, où il serait facile de distribuer la scène sur les processeurs effectuant les tâches INTERSECTION. Toutefois, les résultats obtenus n'encouragent pas une telle tentative qui serait encore plus coûteuse en communications.

Afin de pouvoir traiter des scènes de grande taille, il apparaît indispensable de distribuer les objets sur l'ensemble des processeurs. Une fois que ce partage est effectué et que chaque processeur dispose de pixels à calculer, l'algorithme de lancer de rayon peut commencer à s'effectuer de façon classique. Cependant, quand un rayon est amené à rencontrer un objet qui n'est pas présent dans la mémoire du processeur sur lequel il est, deux stratégies sont possibles : envoyer ce rayon sur le processeur possédant l'objet à rencontrer ou aller chercher l'objet afin de permettre le calcul du rayon sur place. Cette première méthode nommée "flots de rayons" est présentée ci-dessous, la seconde à "flots de données" est explicitée par la suite.

2.2.3 Traitement avec flots de rayons

Au fur et à mesure que les rayons "voyagent" dans la scène et rencontrent des objets non présents sur leur processeur, les rayons circulent sur les processeurs de la machine parallèle.

Goldsmith et Salmon [GS88] ont étudié, par le biais de la hiérarchie de volumes englobants qui composent sa scène, la fréquence des intersections entre les rayons et les volumes englobants en fonction de leur niveau dans la hiérarchie. Il apparaît que la plupart des intersections ont lieu dans les premiers niveaux. Il semble donc intéressant de dupliquer les premiers niveaux de la hiérarchie d'englobants sur l'ensemble des processeurs. De plus, les auteurs estiment que l'obtention d'un code efficace repose sur un compromis entre trois critères :

1. La mémoire disponible sur chaque processeur.
2. L'équilibre des charges sur les différents processeurs.

3. Le nombre de communications entre processeurs.

Afin que les calculs de rayons soient équitablement répartis sur l'ensemble des processeurs, il est important d'utiliser des stratégies d'équilibrage de charges.

D'abord, nous présentons des mécanismes reposant sur une distribution statique de la scène permettant d'assurer cet équilibre, puis des mécanismes d'équilibrage dynamique.

2.2.3.1 Equilibrages statiques

Kobayashi [KNK+88] a étudié de nombreuses stratégies de répartition d'une scène subdivisée régulièrement.

Ainsi il a effectué des découpages réguliers de la scène en 'tranches' (1D), en 'baguettes' (2D) et en 'cubes' (3D).

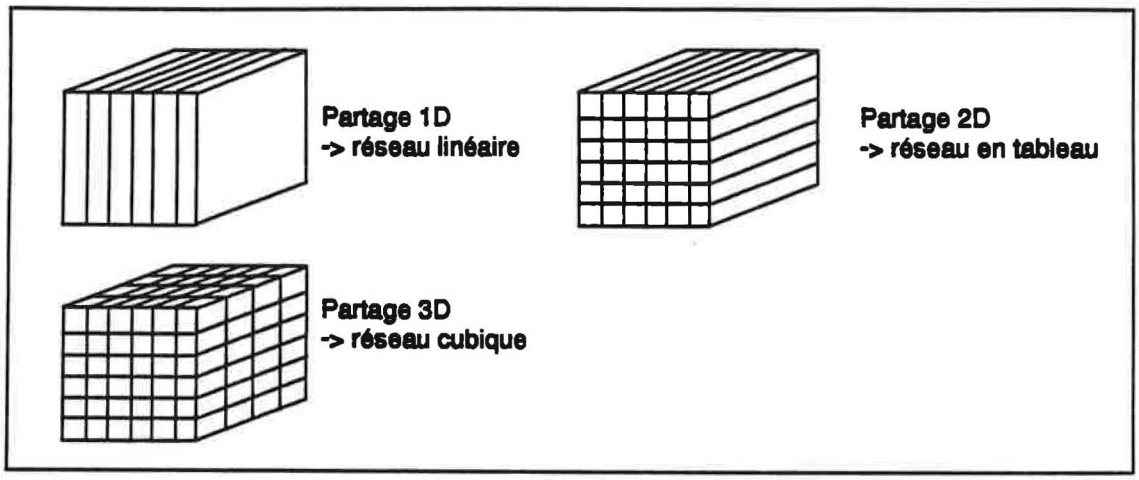

Figure 29 Découpage de l'espace

Il apparaît que le partage 2D est plus efficace que les 3D et 1D. En effet, peu de rayons atteignent le bout opposé de l'espace, aussi les sous-espaces associés ont peu de calculs à faire. Il y a mauvaise répartition des tâches entre le premier et l'arrière plan de la scène.

Une fois le choix d'un découpage bi-dimensionnel réalisé, Kobayashi a cherché comment allouer de façon équilibrée les 'baguettes' ainsi obtenues. Il a comparé deux types d'allocation :

1. Une allocation distribuée, où chacun des p processeurs est responsable d'un sous-espace comprenant une 'baguette' toutes les $\sqrt{p}$ 'baguettes' horizontalement et verticalement.
2. Une allocation par blocs, où chaque processeur est responsable d'une partie connexe de l'espace.

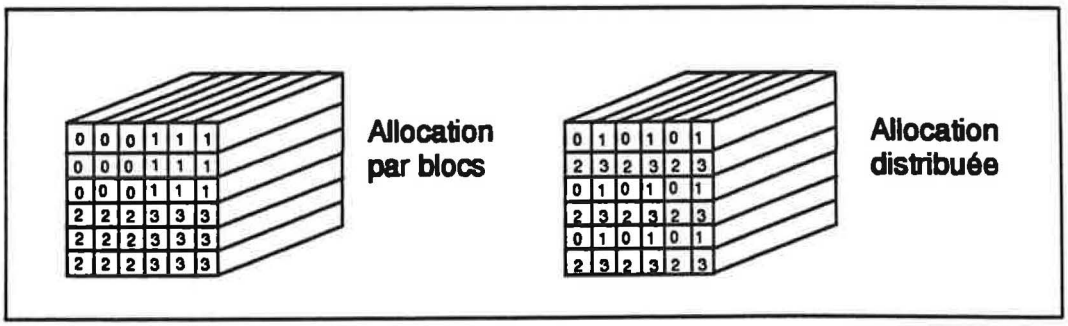

Figure 30 Allocation distribuée et allocation par blocs

L'allocation distribuée apparaît comme étant la plus efficace, car les tâches sont réparties de façon plus homogène. Toutefois, quand le nombre de processeurs augmente, le nombre de "baguettes" dont chacun est responsable diminue : l'allocation distribuée perd de son homogénéité et les résultats obtenus tendent vers ceux de l'allocation par blocs. Aussi l'efficacité de ce découpage baisse considérablement quand le nombre de processeurs devient important.

Afin de lutter contre ce mauvais comportement induit par l'augmentation du nombre de processeurs, un autre type de partage statique a été proposé par Priol et al. : un partage après pré-échantillonage [Pri89], [BBP94].

Ainsi, le partage de l'espace se décompose en trois étapes :

1. Partage de la scène en cellules avec une grille 2D.
2. Sous-échantillonage de l'image, qui conduit au calcul de quelques pixels. On associe un compteur à chaque cellule et on estime le temps nécessaire pour traiter tous les rayons qui iront dans la cellule (A).
3. Regroupement, à l'aide d'une partition dichotomique, des cellules en fonction des valeurs des compteurs, afin d'obtenir des régions 3D équitables (B, C et D).

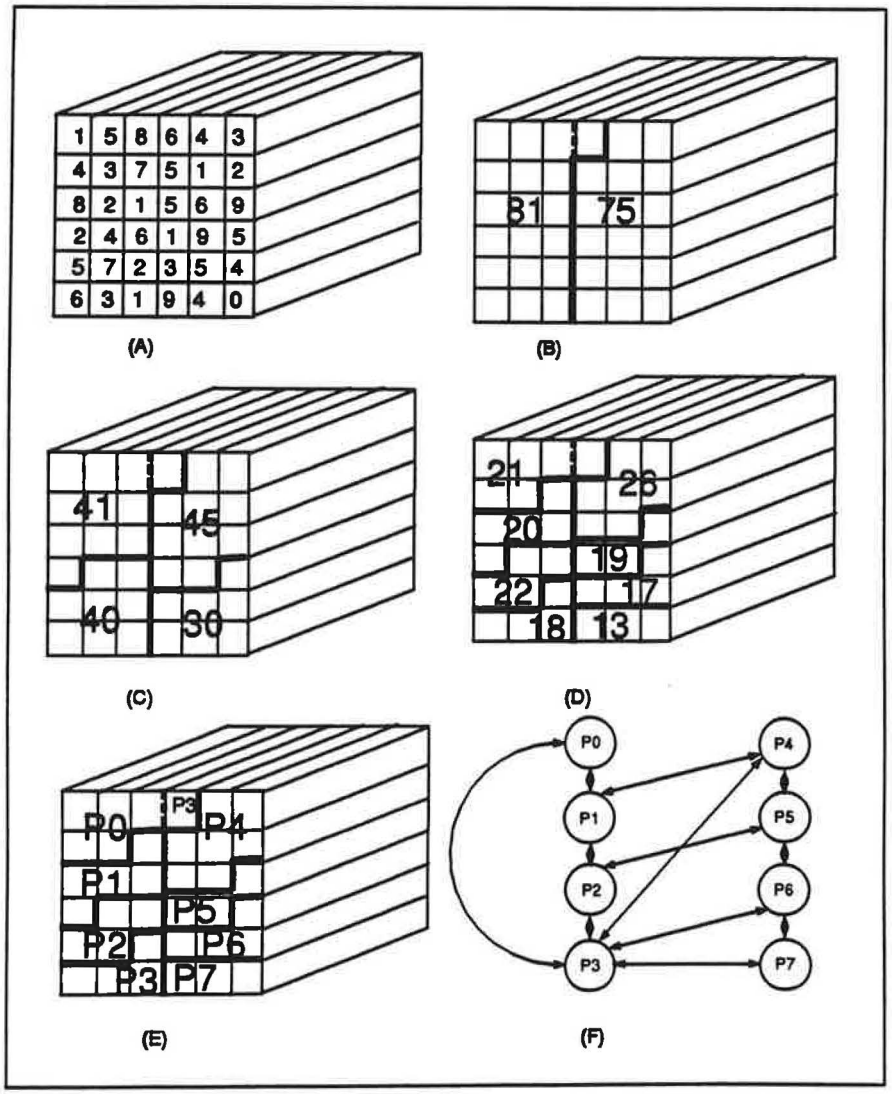

Figure 31 Partage après échantillonage de la scène

On obtient un ensemble des régions adjacentes (E) que l'on peut modéliser à l'aide d'un graphe (F). Ce graphe de processus permet de modéliser les flux de communications interprocesseurs

nécessaires au calcul de l'image concernée. Il est alors possible d'utiliser ce graphe en le plaquant directement sur une architecture réelle.

A l'aide d'un sous-échantillonage par des grilles 8×8 (1,5% de l'image), une charge moyenne supérieure à 80% a été obtenue contre 20% avec une répartition classique.

Bien que le résultat soit positif, cette technique de répartition possède certains inconvénients :

1. Les choix lors du découpage restent très empiriques.
2. Deux régions adjacentes peuvent partager un même objet, ce qui peut conduire à la répétition de calculs d'intersection.

Isler et al. [IAO91] proposent également un équilibrage reposant sur un échantillonnage. La méthode comprend trois phases :

1. Le problème est converti en un problème de partitionnement de graphe. La scène 3D est partagée en sous-volumes rectangulaires, puis un rayon primaire est lancé vers chaque sous-volume. Pendant ce sous-échantillonage on compte le nombre de rayons entrant, sortant et étant calculés dans chaque sous-volume. Les sous-volumes et les relations entre sous-volumes adjacents correspondent respectivement aux noeuds et aux arcs du graphe. Le poids de chaque noeud est obtenu à partir du nombre de rayons calculés dans le sous-volume associé et la mémoire nécessaire pour stocker les objets contenus par le sous-volume. Le poids de chaque arc provient du nombre de calculs échangés entre deux sous-volumes adjacents. Le poids d'un arc peut indiquer un nombre de communications inter-processeurs si les deux sous-volumes concernés sont sur deux processeurs distincts, alors que le poids d'un noeud donne plutôt une idée de la charge.
2. Ensuite, on partage ce graphe en P portions, où P représente le nombre de processeurs disponibles. Et on cherche à minimiser le poids des arcs inter-portions, tout en conservant un certain équilibre au niveau du poids de la somme des noeuds de chaque portion.
3. Enfin, une fois que les P portions définitives ont été obtenues, on associe un processeur à chacune d'elles en cherchant à minimiser les distances des communications inter-processeurs.

Hélas, aucun résultat n'est présenté et le coût de la phase de précalculs par rapport à celui de la phase de rendu n'a pas été évalué.

Silva [SK94] propose une nouvelle méthode d'évaluation de la charge de travail permettant un découpage équilibré de la base de données.

Il commence par effectuer un découpage en voxels non vides de ses volumes englobants. Afin de réaliser un équilibre statique performant, il utilise ces cubes comme base pour l'équilibrage. Il estime que les cubes sont une bonne approximation de la charge de travail à réaliser durant le rendu d'une scène, aussi il utilise le nombre de cubes par processeur comme mesure du travail. Pour qu'il y ait un bon équilibrage, il faut essayer de donner le même nombre de cubes non vides à chacun des processeurs. Pour cela, Silva propose un regroupement par tranches d'espace disjointes. Il compare les résultats obtenus par cette technique et ceux provenant d'un simple découpage régulier en tranches de l'espace.

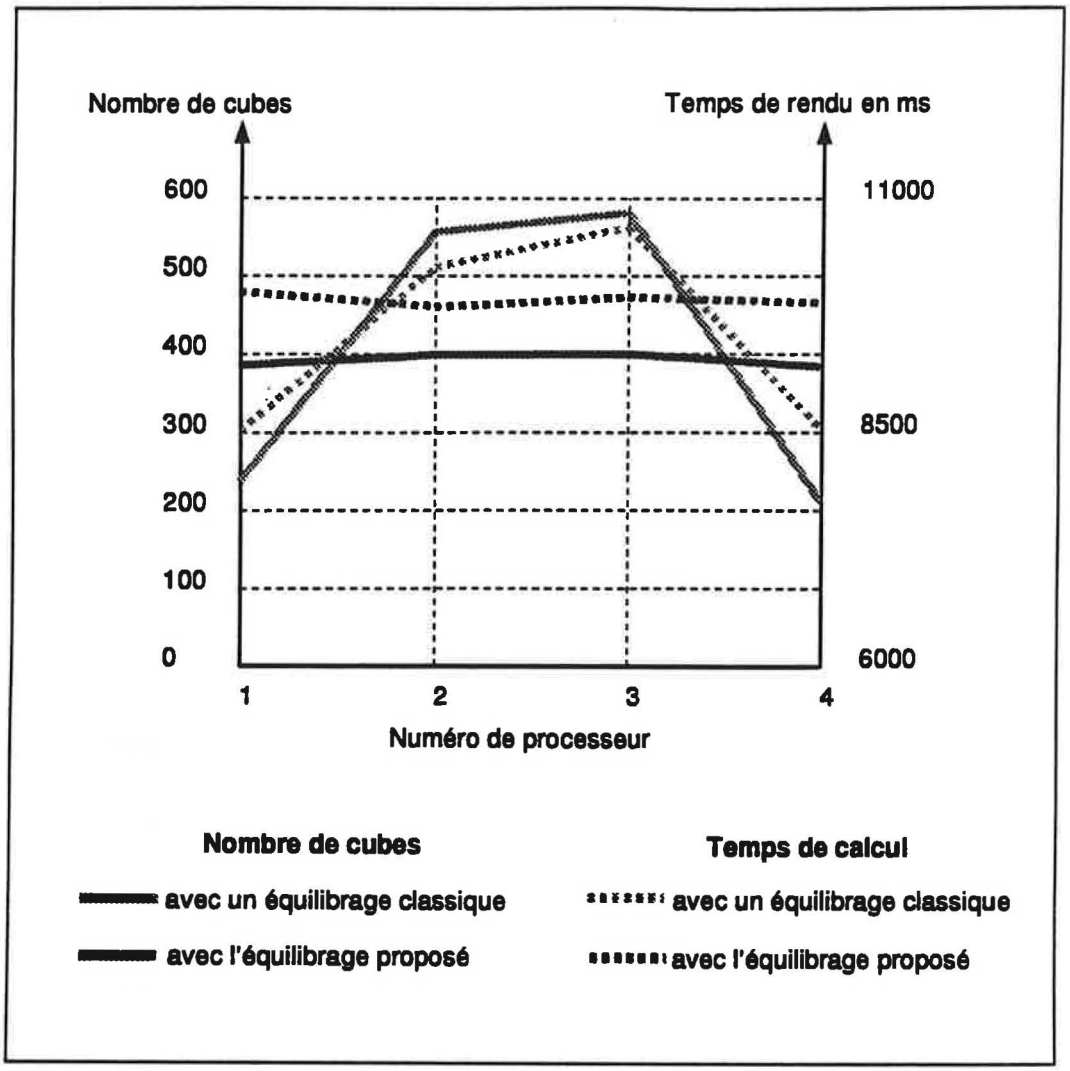

Figure 32 Comparaison de l'équilibrage des charges entre un découpage en tranches et celui proposé par Silva

Le temps de calcul paraît bien corrélé avec le nombre de cubes possédés : la méthode proposée permet un excellent équilibrage de charge. Toutefois, les tests ne semblent avoir été réalisés que sur un nombre de processeurs limité (moins de 10), aussi on peut être sceptique sur l'utilisation de cet équilibrage sur une machine massivement parallèle, où l'obtention de tranches 'égales' paraît plus difficile.

Ces méthodes d'équilibrage statique n'étant pas complètement satisfaisantes, il a été également proposé des équilibrages dynamiques.

2.2.3.2 Equilibrages dynamiques

Une amélioration importante, pour lutter contre la baisse de l'efficacité, a été étudiée par Kobayashi [KNK+88]. Il propose un mécanisme mixte d'équilibrage de tâches, statique et dynamique, à l'aide d'un système multi-processeurs hiérarchique. Ce système comprend une machine hôte qui supervise un ensemble de grappes (k), chacune d'elles étant composée de m processeurs gérés par un contrôleur.

Il peut y avoir deux niveaux d'équilibrage :

1. Au niveau des grappes : les sous-espaces sont alloués à chaque grappe en suivant l'allocation distribuée 2D présentée précédemment.

2. Au niveau de l'élément de calcul : les processus alloués à une grappe sont exécutés en parallèle par les différents processeurs de la grappe. C'est le contrôleur qui alloue dynamiquement les tâches à ces processeurs.

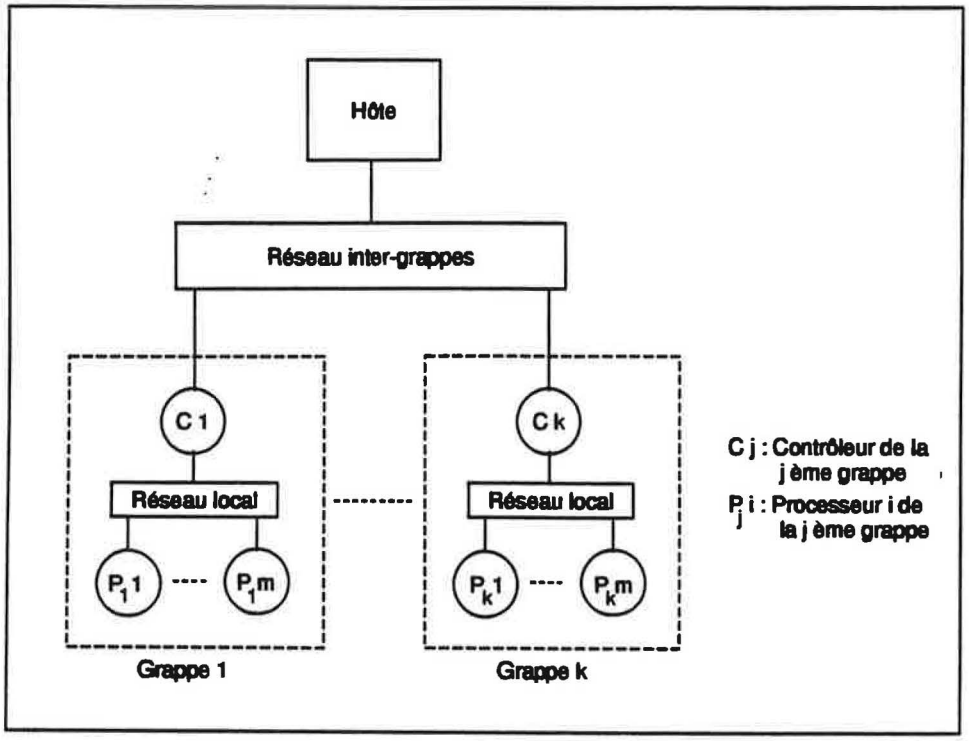

Figure 33 Architecture hiérarchique

La vitesse est alors quasiment linéaire avec le nombre de processeurs. Elle est entre 2 à 4 fois plus rapide que pour le 2D statique. L'efficacité atteint 85%.

Priol [Pri89] propose une autre forme de partage dynamique avec des charges à deux niveaux. Lorsque qu'une parallélisation par flots de rayons ainsi qu'une partition de l'arbre des primitives sur les processeurs sont réalisées, il peut être intéressant de répliquer les premiers niveaux de l'arbre sur tous les processeurs. Ainsi on peut produire un équilibrage dynamique en mettant deux processus sur chaque processeur :

1. Un processus de calcul d'intersections sur la racine.
2. Un processus de calcul d'intersections sur le sous-arbre associé au processeur.

L'équilibrage se met en place en faisant faire plus ou moins de calculs au premier processus. Cette approche est particulièrement intéressante car chaque processeur est actif en permanence et est capable de choisir l'activité la plus appropriée.

Lefer [Lef92], [Lef93] présente une autre forme d'équilibrage dynamique de tâches en divisant également le processus de calcul d'intersection en deux phases :

1. Le parcours des subdivisions de l'espace, qui permet de fournir à un rayon les primitives qu'il est susceptible d'intersecter.
2. La détermination de la primitive effectivement rencontrée et les calculs photométriques.

Il apparaît que, pour la subdivision de l'espace choisie par Lefer, la phase de parcours représente 80% du temps de calcul. Avec le parallélisme à flots de rayons, Lefer va associer un parallélisme de tâches permettant l'obtention d'un équilibrage de tâches performant.

En fait, ces deux tâches n'utilisent pas la même base de données : la première nécessite la structure de décomposition de l'espace alors que la seconde a besoin de l'ensemble des objets. Comme la base de décomposition est de taille réduite par rapport à la base objet, il est possible de la dupliquer sur l'ensemble des processeurs, alors que la seconde sera distribuée. La première tâche pourra donc être exécutée par n'importe quel processeur, sur n'importe quel rayon.

Chaque processeur possédera ces deux tâches concurrentes, effectuant la deuxième de façon prioritaire et la première si sa charge de travail est faible.

Cet algorithme a été testé sur une machine possédant 8 processeurs. Quelle que soit la scène traitée, l'efficacité demeure supérieure à 70%. L'auteur ne dispose pas d'informations sur le comportement de sa solution avec du parallélisme massif.

Toutefois, il prédit que les performances de son algorithme diminueront s'il y a accroissement du nombre de processeurs. En effet, l'augmentation de la distribution de la scène a pour effet d'accentuer le déséquilibrage entre les processeurs.

2.2.3.3 Conclusion

Malgré toutes les méthodes d'équilibrage présentées, les performances du traitement avec flots de rayons se dégradent fortement quand le nombre de processeurs augmente, l'accélération semble suivre une loi logarithmique en fonction du nombre de noeuds. En effet, plus l'architecture comprend de processeurs, plus le nombre de communications augmente, ce qui a pour effet de faire baisser l'efficacité.

Un autre inconvénient majeur de cet algorithme est que certaines images peuvent induire des déséquilibrages de charge très difficiles à gérer qui pénalisent très fortement les résultats obtenus. Cette mauvaise répartition des charges peut provenir par exemple de la présence de sources lumineuses dans une région de la scène ou du déplacement des rayons dans une direction privilégiée.

Cet algorithme n'apparaît donc pas comme une solution permettant une parallélisation satisfaisante de l'algorithme de lancer de rayon.

2.2.4 Parallélisation hybride

Montani [MPS92] a commencé par développer un algorithme parallèle à flots de rayons. Pour des raisons de simplicité d'implémentation, de gestion et d'équilibrage de tâches, un partage de la scène en tranches a été adopté :

- Le partage en tranches est très facile.
- La communication entre les processeurs est simple et rapide, car il n'y a communications qu'entre voisins.
- Cette partition permet la recherche d'un meilleur équilibrage en décalant les plans de coupe réalisés au travers de la scène.
- L'équilibrage dynamique ne remet pas en cause les relations de voisinage entre processeurs.

Cependant dans des environnements massivement parallèles, si la taille de la scène n'augmente pas dans les mêmes proportions que le nombre de processeurs, cette découpe en tranches devient très coûteuse en raison du faible nombre de voxels par processeur. Une solution à ce problème est une combinaison des approches avec "flots de rayons" et avec "partition de l'image".

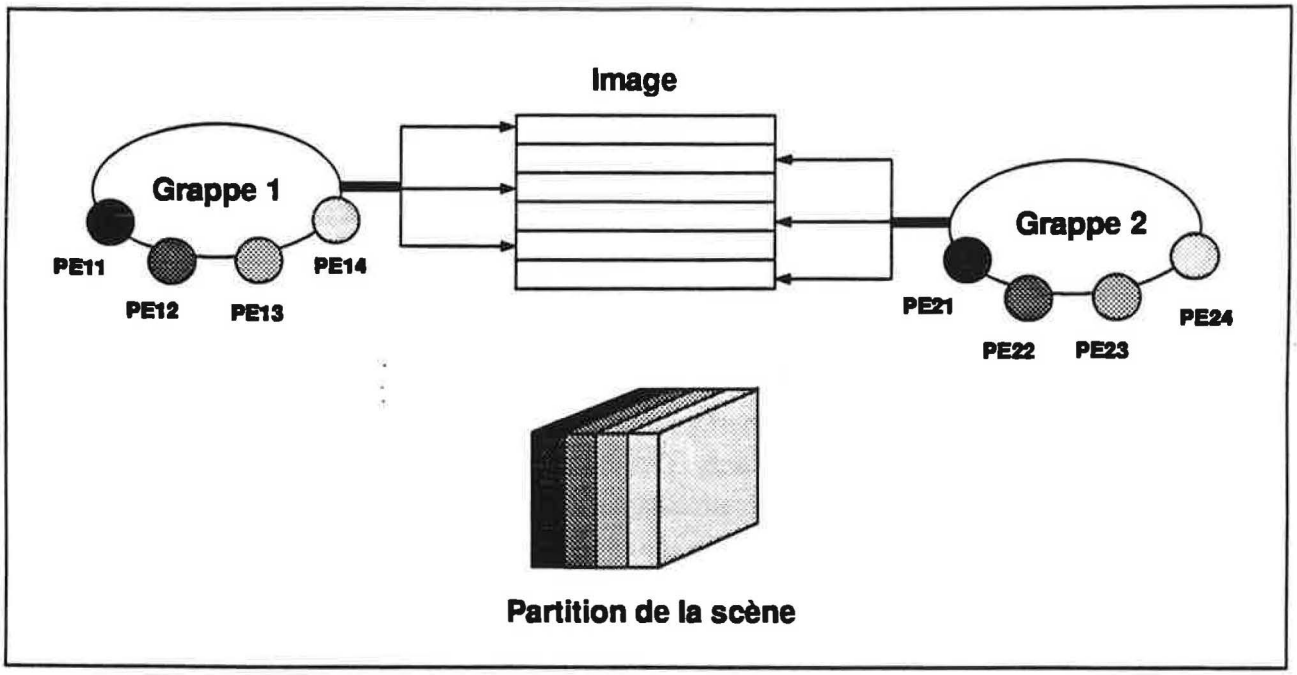

Figure 34 Architecture de la 'parallélisation hybride'

Montani propose un partage de l'image sur des grappes. Chaque grappe va calculer les pixels qui lui ont été alloués de façon indépendante. De plus, ces grappes sont composées de processeurs possédant chacun une tranche de la scène. Aussi les pixels d'une grappe seront calculés au sein de celle-ci par coopération des différents processeurs par le biais de flots de rayons.

L'efficacité obtenue avec 2 grappes est de 74% sur 128 processeurs, alors qu'elle n'était que de 59% avec 1 grappe. Cet algorithme permet des gains intéressants en performance, mais c'est au détriment de la surcapacité puisque celle-ci est divisée par le nombre de grappes utilisées.

2.2.5 Traitement avec flots de données

Chaque processeur a en charge un certain nombre de rayons, qu'il va calculer intégralement. Aussi quand l'absence d'une donnée ne permet plus à un calcul de s'effectuer, la donnée manquante est recherchée pour être copiée en mémoire locale. Le calcul peut alors se poursuivre. Contrairement au traitement avec flots de calculs, ce sont les objets, plutôt que les rayons, qui sont transmis à travers les noeuds de la machine parallèle.

La plupart des stratégies qui sont employées pour le traitement sans flot peuvent être facilement adaptées pour le traitement avec flots de données. L'architecture la plus employée est de type maître-esclaves, mais nous verrons également une architecture en forme d'arbre et l'utilisation d'une mémoire partagée virtuelle.

2.2.5.1 Architecture maître-esclaves

Green et Paddon [GP90] ont proposé une architecture maître-esclaves sur laquelle la base de données est distribuée. Cette architecture comprend un maître relié à une machine hôte et des esclaves reliés au maître grâce à un réseau d'interconnexion.

Le maître distribue les tâches, c'est à dire des portions d'images à calculer (cohérence des pixels), et assure les échanges de données entre esclaves.

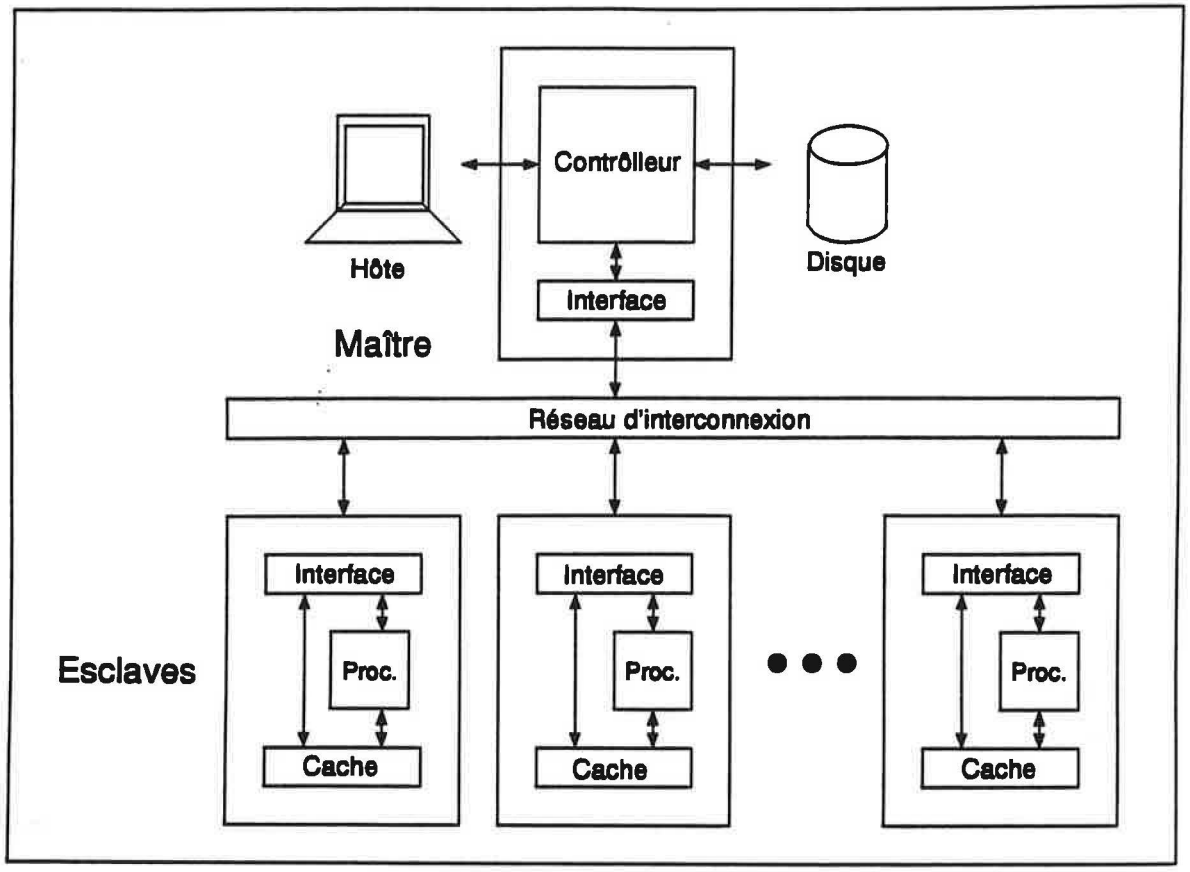

Figure 35 Architecture maître-esclaves

Initialement, le maître distribue la base de données sur les noeuds esclaves. Chaque esclave possède des données permanentes, mais aussi une mémoire cache où sont stockées les données demandées. Ensuite le maître donne un bloc de pixels à calculer à chaque esclave.

Quand un noeud a besoin d'une donnée qui n'est pas présente en mémoire locale, il y a demande de la donnée au maître. Celui-ci répondra à la requête en allant chercher la donnée sur son noeud de rattachement.

Quand un esclave a terminé de calculer sa portion d'image, il la renvoie au maître qui en échange lui donnera de nouveaux calculs à effectuer. C'est ainsi que l'équilibrage des tâches est assuré.

En raison de la cohérence des pixels à calculer par chaque esclave, il y a peu de demandes de données. Green et Paddon ont étudié la fréquence d'usage des différentes données de la base. Il apparaît qu'un petit groupe de données représente une large fraction des demandes de données (10% de la base de données représenterait 70% des utilisations de données).

Sur la figure suivante, nous avons représenté les efficacités annoncées par Green et Paddon pour 22 processeurs sur différentes images en fonction du pourcentage de la base présent sur chaque processeur.

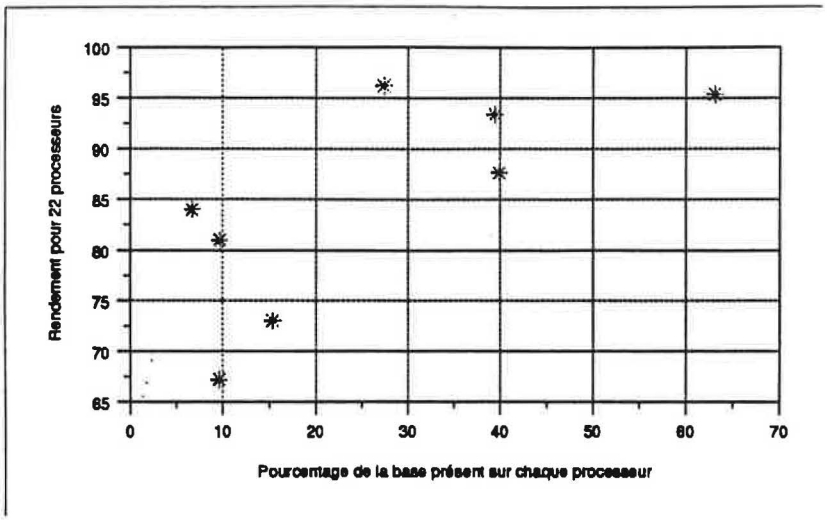

Figure 36 Rendement pour 22 processeurs sur différentes images
en fonction du pourcentage de la base présent sur chaque processeur

Il apparaît que lorsque chaque esclave possède en mémoire locale plus de 25% de la base de données, la distribution de la base n'influe que très peu sur les performances.

Green et Paddon proposent pour améliorer les performances de cet algorithme de copier l'arborescence de la base de données sur tous les processeurs. En effet, ayant évalué que l'octree ne prend que 1/6 à 1/4 de l'espace nécessaire pour stocker les voxels, ce faible coût en mémoire permettrait à chaque processeur de pouvoir effectuer la traversée de l'arbre sur place et ne chercher les données que pour les voxels non nuls.

Bien que cet algorithme donne de très bons résultats, il faut noter que ces performances ont été obtenues sur une architecture ne comportant que 22 processeurs esclaves et que déjà l'efficacité baisse de façon régulière avec l'augmentation du nombre de processeurs (87% pour 10, 78% pour 18 et 73% pour 22). Le choix d'une architecture maître-esclaves ne permet pas d'envisager un bon comportement avec un grand nombre de processeurs.

Reeth [RLF92] a également travaillé sur une architecture maître-esclaves, mais à partir d'un découpage de la scène en voxels réguliers.

Tout d'abord, il a expérimenté deux méthodes de communications des données entre les processeurs :

1. La transmission systolique : les données sont envoyées suivant un rythme régulier sur le réseau de processeurs.
2. La transmission de voxels : une donnée est envoyée quand un processeur en a besoin.

La transmission systolique a été finalement rejetée en raison de la distribution non uniforme du travail qui produit des déséquilibres de charge importants. La transmission de voxels a donc été retenue.

Ensuite, il a étudié le nombre de voxels non vides traversés par des rayons : il apparaît que seulement un très petit nombre de voxels non vides sont traversés pour le calcul d'un rayon.

Enfin, Reeth [RLF92] a proposé quelques optimisations permettant de réduire le coût des communications :

1. Ne pas chercher les voxels uniquement chez le maître, mais aussi sur les processeurs qui peuvent être rencontrés lors de cette communication vers le maître.

2. Optimisation des rayons d'ombrage : la réponse attendue au lancer d'un rayon d'ombrage est binaire (la source est visible ou pas). Il n'est donc pas nécessaire d'accéder aux voxels dans l'ordre dans lequel ils sont traversés par le rayon d'ombre. Il est possible de commencer les calculs avec les voxels qui sont déjà en mémoire. Si le rayon rencontre un objet, le calcul est fini. Sinon, les demandes de voxels seront effectuées.
3. Donner la priorité aux communications sur les calculs : les processeurs doivent attendre le moins possible et de toute façon la communication des données doit être faite, aussi il faut le faire le plus vite possible.
4. Utilisation de techniques de bufferisation pour que les processeurs ne soient pas en attente : la demande d'un nouveau bloc de pixels est faite avant que la tâche en cours ne soit finie.
5. Ne pas transmettre les voxels vides.

La base de données utilisée par Carter [CT90] est organisée sous la forme d'une hiérarchie de volumes englobants. Quand elle est distribuée sur les différents processeurs, le haut de l'arbre est dupliqué sur l'ensemble de la machine et chaque sous-arbre possède son noeud de rattachement qui gardera en permanence cet ensemble de données. Aussi, quand une donnée nécessaire n'est pas en mémoire locale, il y a recherche de la donnée sur son noeud de rattachement. Cette donnée est ensuite copiée sur le noeud demandeur suivant la méthode LRU (Least Recently Used) : la nouvelle donnée remplace la donnée qui est la moins utilisée.

Plutôt que de chercher uniquement la donnée nécessaire, il préconise de prendre également une partie du sous-arbre associé. Ceci pour deux raisons :

1. Le coût d'envoi d'un message de petite taille est proportionnellement supérieur à celui d'un message de grande taille en raison de la phase de lancement qui est fixe.
2. La cohérence des rayons traités par un processeur conduit à favoriser les intersections avec les mêmes objets.

De plus, en raison du temps de communication important, les arrivées des données demandées ne sont pas attendues. Le rayon demandeur est mis dans une file d'attente (sauvegarde de son environnement) et le rayon suivant est traité. Les communications sont donc asynchrones.

Expérimentalement, il apparaît que quand la mémoire cache atteint 20 % de la taille de la base de données des objets, sa taille n'influe plus sur le temps de calcul, qui est très proche de celui obtenu avec duplication des données. On remarquera toutefois que le nombre maximal de processeurs testés est relativement faible puisqu'il ne dépasse pas 32.

2.2.5.2 Architecture en forme d'arbre

Green et al. [GPL88] proposent de ranger les processeurs dans une structure d'arbre dont la racine est le processeur de contrôle. On obtient un système de n processeurs avec k sous-arbres par noeud, le nombre de noeuds à traverser pour une communication entre deux processeurs quelconques est au maximum en $log_k n$.

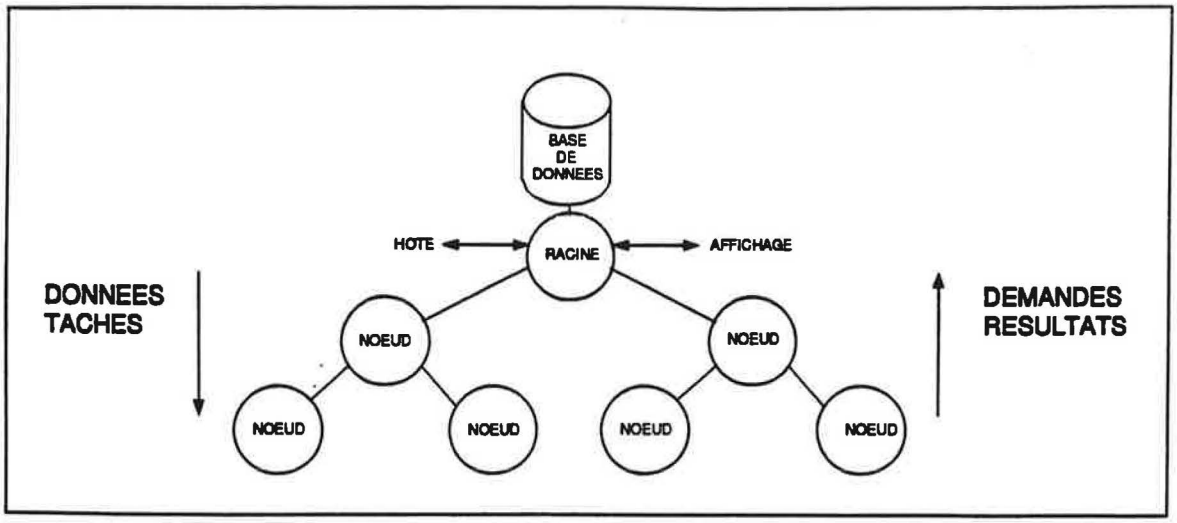

Figure 37 Architecture en forme d'arbre

Les communications sont simplifiées :

1. Un processeur en attente fait une demande (de tâche ou de données) qui est propagée en remontant jusqu'à la racine.
2. Les tâches et les données redescendent l'arbre jusqu'au noeud demandeur.
3. La racine fait aussi la collecte des résultats.

Cette architecture voit deux processus concurrents sur l'arbre :

1. Un processus racine qui a un rôle de superviseur :
   a. Contrôle du réseau.
   b. Interface avec l'hôte.
   c. Affichage des résultats.
   d. Génération des rayons primaires.
   e. Gestion de la base de données.
2. Un processus noeud qui s'occupe d'actions locales :
   a. Calcul des rayons.
   b. Renvoi des tâches terminées à la racine.
   c. Réception des messages du parent et des fils en les relayant si nécessaire.
   d. Maintenance de la base de données locale.

Cette architecture permet une gestion originale de la charge. En effet, quand des rayons secondaires sont créés, on ne sait pas si des processeurs sont libres pour les calculer. Aussi on utilise des piles distribuées des tâches à exécuter. Cela limite les communications puisque les demandes peuvent souvent être satisfaites localement et assure une attente minimale. Ainsi pour chaque demande non satisfaite on remonte à la pile du parent. On peut noter qu'en général, les rayons calculés par un processeur sont adjacents, il y a donc des chances qu'ils rencontrent les mêmes objets.

La gestion des données est relativement classique. La base de données est distribuée, une partie de la base de données (ici une structure d'octree) est associée à chaque processeur. Quand des données sont nécessaires, les processeurs font des demandes à leurs parents et les données sont stockées en mode LRU.

Les résultats présentés montrent une accélération quasi-linéaire quand la base de données est dupliquée. Cependant quand la taille de la mémoire cache devient faible (moins de 10%), les performances se dégradent. De plus, on peut noter que ces résultats sont obtenus sur une architecture ne dépassant pas 8 processeurs, ce qui ne permet pas d'imaginer un bon comportement sur des configurations plus importantes.

L'architecture en forme d'arbre, proposée par Green et al. [GPL88], ne semble donc pas une voie prometteuse.

2.2.5.3 Mémoire partagée virtuelle

L'algorithme de lancer de rayon à flots de données étant basé sur les échanges des données, Badouel [BP90], [Bad90] s'est intéressé plus particulièrement aux travaux existant dans le domaine du partage des données dans un environnement distribué. En effet, une mémoire partagée simplifie l'implémentation de l'algorithme de lancer de rayon et peut améliorer les performances par rapport à l'échange de messages.

Ne disposant pas de mécanisme système permettant le partage de données, il a opté pour la mise en oeuvre d'une mémoire partagée au niveau de l'application. Cette mémoire est virtuelle puisque l'espace d'adressage dépasse les capacités mémoires d'un processeur.

Pour créer une mémoire partagée virtuelle, Badouel fait de la 'pagination d'objets' :

1. L'objet est un élément d'une page.
2. La page est l'unité pour l'échange de données entre mémoires locales.
3. Un objet n'appartient qu'à une et une seule page.

Chaque mémoire locale est distribuée en 3 parties :

1. Le code du processus.
2. Une partie de la base de données.
3. Une mémoire cache.

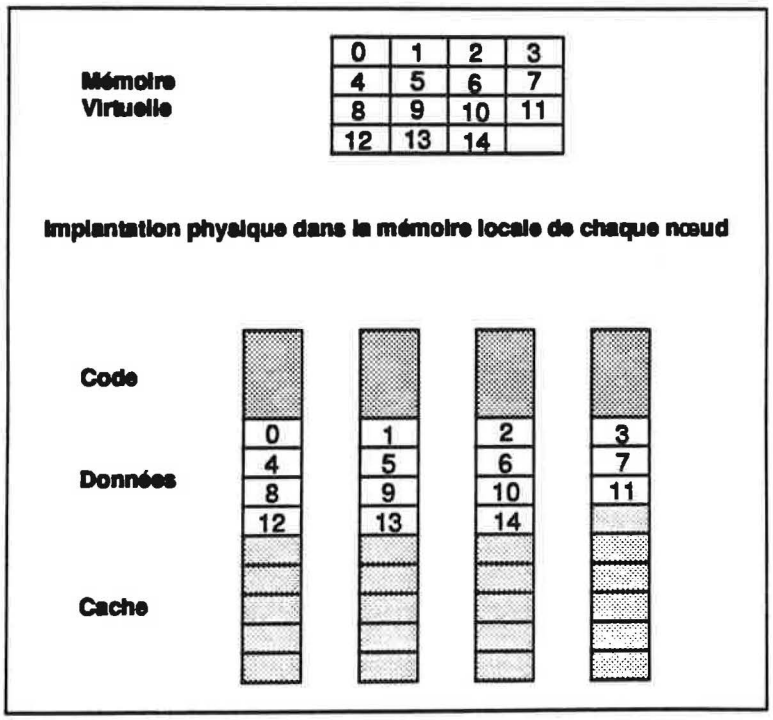

Figure 38 Implantation d'un mémoire partagée virtuelle

Si dans un noeud il manque des données, une demande est envoyée pour avoir la page les contenant. A la réception, elle est stockée en mémoire cache en mode LRU (least recently used). Ainsi tout noeud peut accéder à la base entière.

La gestion de l'équilibrage des tâches se fait comme présenté en §2.2.1, c'est à dire une première distribution statique par blocs, puis un équilibrage dynamique en anneau. L'abandon d'une architecture maître-esclaves va permettre d'envisager l'utilisation de machines massivement parallèles.

La figure suivante présente l'efficacité pour 64 processeurs de l'algorithme en fonction de la taille mémoire disponible pour chaque processeur :

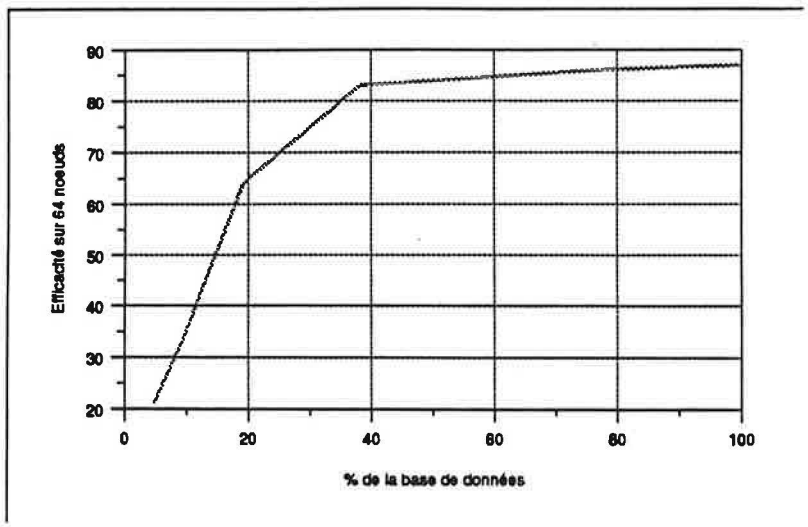

Figure 39 Mesures d'efficacité pour 64 processeurs données par Badouel pour l'image Mountain en fonction de la taille mémoire disponible pour chaque processeur

Les résultats obtenus pour les différentes scènes étudiées donnent une efficacité autour de 78% pour 64 processeurs, quand au moins 40% de la base de données peut résider dans la mémoire locale à un processeur. Il est à noter que le surcoût engendré par la gestion logicielle de l'adressage de la mémoire virtuelle est assez conséquent, celui-ci étant évalué à 20% du temps de calcul total. Par contre, celui de l'équilibrage est tout à fait négligeable, puisqu'il n'est que de 0,5%.

Enfin, lorsque la fraction de la base de données présente en mémoire locale passe sous la barre des 20%, l'efficacité s'effondre.

On peut noter que Keates et Hubbold [KH95] ont également travaillé à l'aide d'une mémoire virtuellement partagée sur une machine comportant 230 processeurs. Les auteurs ne donnent des résultats que sur des configurations où la mémoire locale contient au moins 50% de la base de données. Les efficacités obtenues sur cette machine massivement parallèle sont excellentes (autour de 70%), toutefois les données sont largement redondantes puisqu'en regardant la totalité de l'architecture, il apparaît que la base de données est dupliquée 115 fois.

2.2.5.4 Conclusion

Dans la plupart des études, il apparaît qu'il faut que la mémoire cache atteigne au moins 20% de la taille de la base de données pour que sa taille n'influe plus sur le temps de calcul. Enfin, il semble de nouveau qu'il faille réserver les architectures maître-esclaves aux environnements ne possédant qu'un nombre de processeurs limité.

Cet algorithme de parallélisation ne semble pas avoir réellement de limitation au niveau de la taille de l'architecture utilisée, du moment que la taille de la mémoire cache est suffisante.

2.2.6 Cas des réseaux de stations de travail

Pour terminer ce panorama des algorithmes de parallélisation du lancer de rayon sur machines parallèles à mémoire distribuée, il faut noter que les réseaux de stations de travail munis de liaisons à haut débit peuvent être assimilés à ces machines. Cependant, très peu de travaux ont été publiés sur ce type d'architecture. Nous ne présenterons donc que ceux de Sung [SLSA96] et Hamdi [HL97].

Sung fait apparaître les différences majeures qu'il y a entre l'utilisation d'une machine parallèle et un réseau de stations de travail :

1. Le coût des communications est particulièrement élevé :
   a. Il faut éviter la parallélisation avec un grain trop fin.
   b. Un équilibrage de tâche dynamique ne devra pas distribuer des tâches trop petites.
   c. Il faudra essayer de privilégier les communications asynchrones.
2. Une station dans un réseau peut-être à tout moment redémarrée. La mise au point de mécanismes de tolérance aux pannes semble encore plus souhaitable que sur machines parallèles.
3. L'environnement utilisé est hétérogène : les stations peuvent être de puissances variables, avoir des systèmes d'exploitation différents ou avoir des formats de données distincts.
   a. Utilisation d'un environnement de communication standardisé afin qu'il soit disponible sur l'ensemble des stations et que les données aient la même représentation.
   b. On peut envisager une distribution de tâches dont la taille serait proportionnelle à la puissance de la machine concernée.

A partir de cette analyse, Sung décide de réaliser une architecture maître-esclaves où le maître distribue dynamiquement les tâches à des processeurs qui possèdent la totalité de la scène. Il choisit des tâches suffisament petites pour que l'hétérogénéité des puissances des machines ne produise pas de déséquilibrage; il abandonne donc l'idée d'une distribution de tâches adaptée à la machine esclave. Enfin, toutes les communications sont asynchrones.

Les résultats sur un réseau de stations homogènes sont comparables à ceux obtenus sur une machine parallèle. De plus, pour un réseau hétérogène, en normalisant la puissance de chacune des machines, les résultats obtenus sont très proches des précédents. Cela s'explique par la taille suffisamment faible des tâches allouées qui assure un bon équilibrage des charges. Les tests réalisés ne dépassent pas 50 machines, mais l'auteur pense qu'au delà d'une centaine, l'utilisation d'une architecture possédant plusieurs maîtres serait indispensable.

L'auteur ne décrit pas de mécanismes de tolérances aux pannes. Toutefois, vue l'architecture utilisée, il est possible de proposer un tel mécanisme. D'abord, il faut éliminer le cas où le maître est sujet à une panne; en effet, il n'y a pas de reprise possible. Par contre, comme le maître distribue les tâches aux esclaves qui lui renvoient les pixels calculés, il peut connaître la liste des pixels en cours de calcul. Cette connaissance centralisée va simplifier la méthode de tolérance aux pannes, puisqu'il n'y aura pas besoin de dater les pixels, ce qui permettra d'éviter les problèmes de synchronisation des différentes horloges.

Quand le maître a fini de distribuer toutes les tâches et qu'un processeur (A) demande de nouveaux calculs, le maître peut lui attribuer le calcul de pixels qui sont encore en cours de calcul. Le maître choisira les pixels distribués le plus anciennement. Ainsi si le processeur (B), qui était

responsable de ces pixels, les calcule avant le processeur (A), les calculs envoyés par le processeur (A) ne seront pas pris en compte. Par contre, si ce n'est pas le cas - le processeur (B) est soit trop lent, soit en panne -, les résultats attendus proviendront du processeur (A). Par un tel mécanisme, on peut assurer le calcul d'une image tant que le processeur maître et au moins un esclave restent en état de marche.

Hamdi [HL97] s'est plus intéressé au cas de réseaux de stations homogènes dont la charge est très variable au cours du temps en raison de leur utilisation par plusieurs utilisateurs simultanément. Aussi l'implémentation de mécanismes d'équilibrage dynamique de tâches apparaît comme incontournable. L'auteur propose une architecture de type maître-esclaves. Le rôle du maître se décompose de la façon suivante :

1. Envoi de messages demandant la charge de chacun des esclaves.
2. Réception des informations concernant la charge des esclaves.
3. Partition de l'image sur les esclaves à partir des informations collectées.
4. Pendant que les esclaves calculent leurs pixels, les étapes 1 et 2 sont répétées régulièrement pour éventuellement rééquilibrer la charge en ordonnant une redistribution de l'image.
5. Collecte des résultats.

Afin de profiter de la cohérence des pixels consécutifs et de la proximité physique des stations (réseau en anneau), la redistribution de l'image se fait par échange de blocs de pixels entre processeurs voisins.

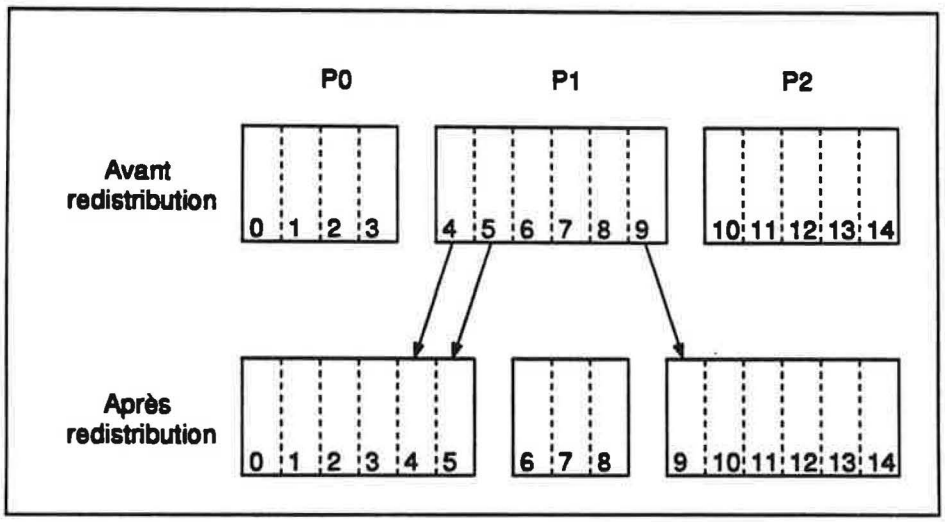

Figure 40 Redistribution de l'image

Les tests réalisés soit par simulation, soit par expérimentatiom, montrent que ce mécanisme d'équilibrage de tâches est efficace quelles que soient les configurations de charge. Plus le nombre de processeurs est élevé ou plus la charge du système est faible, plus cet équilibrage dynamique est performant comparé à un algorithme sans équilibrage de tâches. Bien que la configuration maximale testée ne comportait que 16 stations, le comportement encourageant du mécanisme d'équilibrage laisse espérer de bons résultats sur des configurations plus importantes.

Finalement, on peut dire que les algorithmes de parallélisation du lancer de rayon implémentés sur des machines parallèles peuvent être facilement adaptés aux réseaux de stations de travail à condition de privilégier des algorithmes peu coûteux en communications. L'hétérogénéité des machines du réseau peut être gommée en adaptant la taille de la charge à la puissance de la machine. Par contre, l'utilisation de machines dont la charge varie très fortement en raison d'applications concurrentes impose un équilibrage de charge dynamique.

2.2.7 Algorithmes parallèles pour d'autres applications dont la problématique est proche

L'algorithme du lancer de rayon peut être schématisé comme étant le calcul de rayons indépendants, ce calcul nécessitant la connaissance de données invariantes au cours du temps. En remplaçant le vocable "rayon" par calcul, la problématique de la parallélisation de l'algorithme du lancer de rayon peut être étendue. Elle devient la parallélisation d'un algorithme comportant un grand nombre de **calculs indépendants** nécessitant des données **invariantes** au cours d'un pas de temps donné, ces données pouvant être distantes.

Nous ne détaillerons dans notre étude que deux autres types d'applications, mais de nombreuses autres peuvent être envisagées comme l'accès parallèle par des codes de calcul à des bases de données numériques [Mal97].

Nous commencerons par présenter l'accélération de l'algorithme de radiosité "progressive", qui a beaucoup bénéficié des expériences de parallélisation pour le lancer de rayon, puis celle des applications de type suivi particulaire.

2.2.7.1 Application à la radiosité progressive

Introduction à la radiosité

L'algorithme de radiosité a été proposé par Goral et al. en 1984 [GTGBce] et Nishita en 1985 [NNce] afin de modéliser les interactions lumineuses entre surfaces diffuses. Cette méthode est en fait une adaptation d'une technique utilisée dans l'étude des transferts radiatifs de chaleur.

Le principe de la radiosité est d'établir un équilibre énergétique entre tous les objets composant une scène (conservation de l'énergie). Ces objets sont composés de facettes émettant ou réfléchissant de l'énergie lumineuse, constante sur la facette, de manière parfaitement diffuse. Ce modèle dans sa version initiale ne permet donc pas de traiter les réflexions spéculaires.

Afin de calculer une scène à visualiser par la méthode de la radiosité, il faut tout d'abord déterminer pour chaque facette la fraction d'énergie qu'elle peut recevoir de chacune des autres facettes de la scène. Ces fractions, appelées facteurs de forme, ne tiennent compte que de la géométrie de la scène et traduisent comment deux facettes se perçoivent.

La radiosité d'une surface correspond à son énergie propre augmentée de la fraction réémise des énergies reçues, c'est à dire la somme des radiosités qui parviennent à atteindre cette surface. La radiosité d'une surface j s'obtient par la résolution du système suivant :

$$B_j = E_j + \rho_j \sum_i B_i F_{ij} \qquad (11)$$

1. B est la radiosité d'une surface,
2. E est l'énergie propre émise par une facette,
3. F est le facteur de forme liant deux facettes,
4. ρ est la fraction de l'énergie reçue, réémise par une facette.

Le point faible de cet algorithme est qu'il faut calculer et stocker la totalité de la matrice des facteurs de forme, afin de pouvoir résoudre le système. Ceci impose des ressources "mémoire" importantes et ne permet l'obtention d'une image qu'à la fin des calculs.

Afin d'y remédier, un modèle un peu différent peut être utilisé : la radiosité progressive.

Principe de la radiosité progressive

Dans l'algorithme précédent, chaque facette collecte l'énergie diffusée par les autres facettes. Cohen et al. proposent d'inverser le procédé en calculant l'énergie reçue par chacune des facettes suite à la diffusion d'une facette [CCWG88].

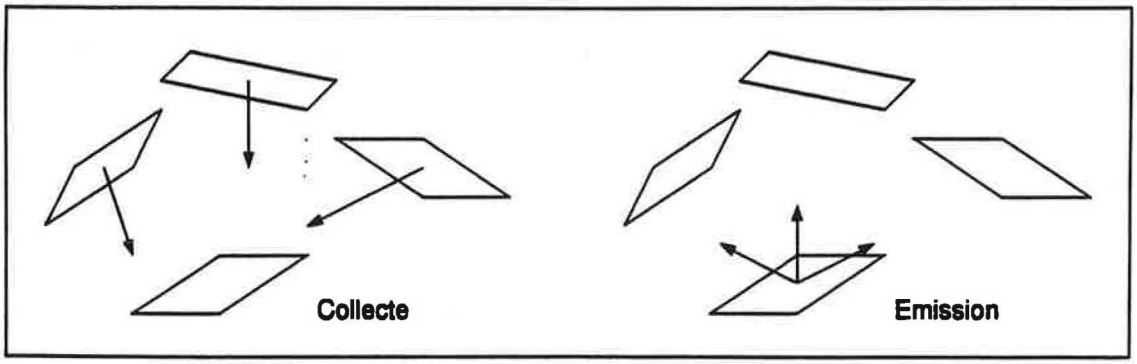

Figure 41 Radiosité par collecte et par émission

Ce calcul d'énergie ne demande que la connaissance d'une ligne de la matrice des facteurs de forme, qui pourra être ensuite effacée, et permet l'obtention d'une première image qui se raffinera au fur et à mesure de la diffusion des autres facettes.

Toutefois, les facettes émettrices pouvant recevoir à leur tour de l'énergie, il faudra qu'elles réémettent ce surplus d'énergie. Ainsi, une facette pourra être sélectionnée plusieurs fois et la ligne de la matrice des facteurs de forme qui lui est associée devra être recalculée. Ce procédé se réitère jusqu'à ce que les critères de convergence fixés soient atteints.

Cet algorithme a des besoins en mémoire limités à la taille d'une ligne de la matrice des facteurs de forme et permet l'affichage progressif de l'image. Toutefois, il a pour inconvénient de ralentir l'obtention de l'image finale.

Parallélisation de l'algorithme de radiosité progressive

L'algorithme de radiosité progressive calculant chaque facette séparément, la parallélisation peut se faire en effectuant le calcul de plusieurs facettes en même temps. De plus, la radiosité est un modèle d'éclairement beaucoup plus récent que le lancer de rayon. Aussi, elle peut bénéficier des expériences de parallélisation du lancer de rayon pour son accélération.

Guitton et al. [GRS95] proposent le partage de la scène sur l'ensemble des noeuds de la machine. Chaque processeur est alors responsable d'une partie de la scène et de tous les calculs associés.

Chaque processeur choisit la facette la plus énergétique de sa région et lance ses rayons de transport de lumière. Quand l'intersection entre une facette et un rayon ne peut être effectuée, le calcul est exporté sur un processeur voisin.

Ce schéma a quelques similitudes avec les techniques de parallélisation du lancer de rayon (flots de rayons) et devrait pouvoir être implémenté sur notre noyau parallèle.

On peut noter également que l'aspect flot de données est proposé également pour cet algorithme comme le montrent les travaux de Bouatouch et al. sur l'utilisation d'une mémoire partagée virtuelle [BMP93].

2.2.7.2 Dynamique particulaire

La dynamique particulaire est une technique de physique qui permet de calculer les trajectoires de particules dans un milieu. Celles-ci obéissent individuellement aux lois classiques de la mécanique. La consommation importante en calculs et le caractère local des phénomènes physiques étudiés font que ce problème semble bien se prêter à la parallélisation.

Pour chaque pas de temps, le champ créé par l'ensemble des particules est évalué, puis on calcule le déplacement des particules dans ce champ.

Ainsi, Norton et al. [NSD95] proposent une parallélisation d'un code de simulation de plasma, où des électrons interagissent au travers du champ électrique qu'ils produisent. Les forces mises en jeu par chaque particule étant à courte distance, la distribution du maillage, représentant le champ électrique, se fait de manière régulière sur l'ensemble des processeurs. Ainsi, les électrons "voyagent" d'un processeur à l'autre en fonction de la région parcourue. La mise à jour du maillage peut se faire de façon locale avec seulement des communications avec les processeurs voisins. Cette méthode de parallélisation peut être assimilée à celle du flot de rayons pour l'algorithme de lancer de rayon.

Souffez et Cuoq [SC95] lors de l'étude de la parallélisation d'un code de dynamique moléculaire, "simulation d'un système monoatomique d'argon", passent en revue les méthodes de parallélisation possibles. La méthode utilisée par Norton et al. est étudiée, mais les difficultés dues à l'équilibrage des tâches font qu'elle a été rejetée.

Souffez et Cuoq proposent une méthode de partition du nombre de molécules sur l'ensemble des processeurs afin d'assurer un bon équilibrage de charge. Toutefois cela impose un grand nombre de communications afin de mettre à jour l'espace de simulation ainsi que le déplacement des particules. En effet, il n'est plus possible de tirer parti de l'aspect local des forces induites. Cette méthode de parallélisation peut être assimilée à celle du flot de données pour l'algorithme de lancer de rayon.

Comme nous venons de le voir de nombreux problèmes de dynamique particulaire aboutissent aux mêmes méthodes de parallélisation que l'algorithme de lancer de rayon.

Nous venons de présenter diverses applications dont les algorithmes utilisés pour leur parallélisation sont très proches de ceux mis en oeuvre pour la parallélisation du lancer de rayon. Les algorithmes étudiés n'apportent pas de voies nouvelles, mais vont nous inciter à développer un noyau parallèle aussi générique que possible afin qu'il puisse être utilisé par les applications précédemment étudiées.

2.2.8 Bilan

Avant la comparaison des différents algorithmes de parallélisation du lancer de rayon, nous allons évaluer leur surcapacité.

2.2.8.1 Surcapacité des algorithmes

Rappellons que ce nouveau critère permet d'évaluer le gain dans la taille maximale des problèmes que l'on peut aborder avec n processeurs par rapport aux problèmes abordables en séquentiel. La surcapacité s'exprime donc par la formule suivante : $Scap_n = \frac{CapaciteMax_{par\ n}}{CapaciteMax_{seq}}$.

Nous allons calculer la surcapacité pour les différentes classes d'algorithmes que nous venons de présenter. Ce classement repose sur l'utilisation faite de la mémoire locale :

1. Duplication de la base sur l'ensemble des processeurs :

$$Scap_n = 1 \tag{12}$$

2. Distribution de la base de façon disjointe sur l'ensemble des processeurs alors

$$Scap_n = n \ , \ avec \ n \in I\!N^* \tag{13}$$

3. Architectures en grappes (soit g le nombre de grappes) :

   a. soit une partie différente de la base est dupliquée sur l'ensemble des processeurs de chaque grappe :

$$Scap_n = g \ , \ avec \, g \in I\!N^* \tag{14}$$

   b. soit la base est dupliquée sur chaque grappe et distribuée sur chaque processeur de la grappe :
   Le nombre de processeurs par grappe est de $\frac{n}{g}$ alors

$$Scap_n = \frac{n}{g} \ , \ avec \, (n, g) \in (I\!N^* * I\!N^*) \tag{15}$$

4. Distribution de la base sur l'ensemble des processeurs avec une duplication partielle (structure de la base, racines d'arbres...) :
   Soit F la fraction de base dupliquée, B la taille de la base de données et M la mémoire de chaque PE.

   Alors $M = \frac{B*(1-F)}{n} + F * B$

   soit $B = \frac{M}{\frac{(1-F)}{n}+F}$
   donc

$$Scap_n = \frac{1}{\frac{(1-F)}{n} + F} = \frac{n}{1 + F * (n-1)} \ , \ avec \ n \in I\!N^* \ et \ F \in]0, 1] \tag{16}$$

5. Distribution de la base sur l'ensemble des processeurs et utilisation d'une mémoire cache. Soit C la fraction que représente la taille de la mémoire cache par rapport à la mémoire disponible.

   Alors $M = \frac{B}{n} + CB$

   soit $B = \frac{M}{\frac{1}{n}+C}$
   donc

$$Scap_n = \frac{1}{\frac{1}{n} + C} = \frac{n}{1 + nC} \ , \ avec \ n \in \ I\!N^* \ et \ C \in]0, 1] \tag{17}$$

6. Distribution de la base sur l'ensemble des processeurs, utilisation d'une mémoire cache et duplication partielle des données.

   Alors $M = \frac{B*(1-F)}{n} + F * B + C * B$

   soit $B = \frac{M}{\frac{(1-F)}{n}+F+C}$

   donc

$$Scap_n = \frac{1}{\frac{(1-F)}{n} + F + C} = \frac{n}{1 + F * (n-1) + C * n} \text{ , } avec\ n \in I\!N^*\ et\ (F, C) \in]0, 1]^2 \tag{18}$$

2.2.8.2 Comparaisons des algorithmes

Avant de comparer les différentes techniques de parallélisation, nous proposons d'en avoir une vision plus globale. Le tableau suivant est une synthèse des différents algorithmes parallèles, sur machine à mémoire distribuée, présentés dans notre étude.

Algorithme	Référence	Nombre de PEs	Accélération	Efficacité	Surcapacité (expression) Base par PE	Divers
Sans flot	Murakami88	64	3.96 (16->64)	99%	(12)	Partition statique par points
	Reeth92	-	-	-	(12)	Partage dynamique (maitre-esclaves)
	Green90	-	-	-	(12)	Partage dynamique (maitre-esclaves)
	Pandzic95	256	76.8	30%	(12)	Partage dynamique (maitre-esclaves)
	Kobayashi88	-	-	-	(12)	Partage dynamique (grappes)
	Badouel90b	-	-	-	(12)	Partage dynamique (anneau)
Coopératif	Barett90	-	-	-	(12)	4 tâches
Flots de rayons	Goldsmith88	-	-	-	(16)	Duplication des racines
	Kobayashi88	-	-	40%	(13)	Découpage statique 2D de la scène
	Priol89	32	14	45%	(13)	Partage par échantillonage
	Isler91	-	-	-	(13)	Partitionnement de graphe
	Silva94	10	-	-	(13)	Partage des voxels non vide
	Kobayashi88	-	-	85%	(16)	Equ. dyn. (2 proc.: grappe et calcul)
	Priol89	-	-	-	(16)	Equ. dyn. (2 proc.: racine et sous-arbre)
	Lefer92	8	5,6	70%	(16)	Equ. dyn. (2 proc.: intersection et parcours)
Hybride	Montani92	128	95	74%	(15)	Flots de rayons et partition de l'image
Flots de données	Green90	22	-	-	(18)	Equilibrage dynamique (maitre-esclaves)
			20	91%	B=40%	
			16	73%	B=10%	
	Reeth92	-	-	-	(17)	Equilibrage dynamique (maitre-esclaves)
	Carter90b	32	-	-	(18)	Equilibrage dynamique (maitre-esclaves)
	Green88	-	-	-	(18)	Architecture arborecente
	Badouel90a,b	64	-	-	(17)	Mémoire partagée virtuelle
			50	78%	B=40%	
			41	64%	B=20%	
			22	34%	B=10%	
			14	21%	B=5%	
	Keates95				(17)	Mémoire partagée virtuelle
Réseaux de stations, sans flots	Sung96	50	-	-	(12)	Partage dynamique (m.-escl.) et asynchronisme
	Hamdi97	16	-	-	(12)	Redistribution dynamique entre voisins
	Arques94	-	-	-	(12)	Acquisition dynamique de la topologie

Table 1 Récapitulatif des algorithmes parallèles présentés

A partir de ce panorama des méthodes de parallélisation de l'algorithme de lancer de rayon, il apparaît clairement, afin de pouvoir en choisir une, il faut déterminer si les bases de données à traiter doivent être dupliquées ou distribuées sur les processeurs disponibles.

Si les bases peuvent être dupliquées, une solution s'impose : il faut utiliser un algorithme sans flot permettant un équilibrage de charges. Une architecture maître-esclaves peut être suffisante sur les machines ne comprenant que quelques processeurs; par contre pour une utilisation dans un cadre massivement parallèle, une architecture décentralisée s'impose.

Si les bases doivent être distribuées, il nous faut comparer les algorithmes à flots de données et à flots de rayons. Pour cela, on s'intéresse tout d'abord aux surcapacités de chacun de ces types d'algorithme.

Il apparaît qu'en général les algorithmes à flots de données demandent plus de mémoire que ceux à flots de rayons en raison de la nécessité de disposer d'une mémoire cache. Bien entendu

la taille de cette mémoire cache peut être réduite assez facilement, mais c'est au détriment des performances. Les comparaisons doivent être faites pour une même surcapacité.

On peut également essayer de comparer les accélérations et le rendement de ces algorithmes. Toutefois, les comparaisons entre ces algorithmes sont particulièrement difficiles en raison des nombreux facteurs qui influencent leurs performances :

1. La nature de l'algorithme séquentiel ayant servi de base à la parallélisation.
2. Le type de machine utilisé.
3. Le nombre de processeurs.
4. Le réseau de communication.
5. La description de la base de données (volumes englobants, voxels...).
6. La scène à représenter.
7. Les caractéristiques de l'image demandée (taille, profondeur...).
8. Les résultats disponibles.

Priol et Badouel dans [BBP94] font une comparaison de leur implémentation parallèle de l'algorithme de lancer de rayon, respectivement avec flots de rayons et flots de données. Ils arrivent à la conclusion que l'algorithme à flots de rayons a un coût élevé et est peu efficace pour assurer un bon équilibrage de tâches — la présence de sources dans la scène étant catastrophique. De plus, quand le nombre de processeurs augmente, le partage d'objets conduit à des répétitions de calculs d'intersection qui nuisent à l'efficacité de l'algorithme.

Par contre, avec l'algorithme à flots de données, l'équilibrage est assuré par l'échange de pixels et le système de mémoire cache utilisé permet de limiter les communications. Toutefois, quand la mémoire cache est trop petite, les performances s'effondrent.

Bien que cette étude soit intéressante, rarement les deux types d'algorithme avec flots ont été comparés de façon précise. Elle ne permet pas de conclure à la supériorité, quelle que soit la configuration, de l'algorithme à flots de données sur celui à flots de rayons. En effet, quand la taille de la mémoire cache disponible est faible, on ne sait pas quel algorithme est le plus performant.

Cet état de l'art nous permet d'extraire des voies prometteuses de parallélisation sans toutefois autoriser un choix définitif entre celles-ci.

Par ailleurs, au sujet de la parallélisation du lancer de rayons sur réseaux de stations, très peu de tentatives sont présentées dans la littérature et elles sont toutes avec duplication des données. Cependant, il apparaît que les contraintes particulières engendrées par ce type d'architectures - machines hétérogènes et communications lentes - sont relativement mineures par rapport à celles des machines parallèles. De plus elles peuvent être assurées par un équilibrage dynamique de tâches et des communications asynchrones. Nous ne distinguerons donc plus ces deux types d'architectures très voisines dans le reste de notre travail.

2.3 Conclusion

Ce chapitre nous a permis de passer en revue les différentes architectures existantes et les types de programmations parallèles possibles. Une étude bibliographique nous a conduit à confronter les différentes voies utilisées pour accélérer l'algorithme de lancer de rayon par le biais de la parallélisation. Certains critères qualitatifs ont pu être dégagés, mais nous n'avons pas pu conclure à la supériorité d'une méthode sur les autres. Il nous faut donc poursuivre notre étude afin de déterminer la voie à retenir pour l'élaboration d'un noyau parallèle pour le lancer de rayon.

Pour cela, nous allons commencer par préciser nos contraintes et proposer une comparaison des méthodes de parallélisation envisageables. Ensuite, nous définirons une stratégie adaptée à nos besoins spécifiques.

Chapitre 3
Algorithmes parallèles à flots mixtes

La finalité de notre projet est la réalisation d'un noyau parallèle pour un algorithme de lancer de rayon. Ce travail dépend de certaines contraintes que nous commençons par définir. Ensuite, nous cherchons à déterminer quels algorithmes présentés dans l'étude bibliographique peuvent y répondre. Pour cela, nous les comparons grâce à une modélisation de leur comportement afin d'évaluer lequel est le plus performant dans notre contexte. Enfin, nous proposons de nouveaux algorithmes parallèles de lancer de rayon reposant sur l'utilisant conjointe de flots de données et de calculs.

3.1 Contraintes

Nos contraintes dans la réalisation d'un noyau parallèle pour un moteur de lancer de rayon sont de deux types :

1. Les premières proviennent du cahier des charges qui nous est imposé.
2. Les secondes dépendent de notre environnement aussi bien matériel que logiciel.

Après les avoir détaillées, nous définirons les caractéristiques du noyau parallèle que nous allons développer.

3.1.1 Objectif de notre travail

Le but de nos travaux est de réaliser un noyau parallèle pour un moteur de lancer de rayon. Ce logiciel doit permettre de traiter des scènes de toutes tailles sur des architectures parallèles aussi variées que possible. Ceci impose :

1. Une distribution de la base de données.
2. La réalisation d'un code qui ne soit pas dédié à une machine ou à un environnement donné.

De plus, il serait souhaitable de pouvoir généraliser l'utilisation de notre noyau parallèle à d'autres applications que le lancer de rayon destiné à la synthèse d'images. Nous avons vu dans le chapitre précédent que la problématique de la parallélisation du lancer de rayon pouvait être assez facilement étendue : on peut la généraliser comme étant la parallélisation d'algorithmes comportant un grand nombre de calculs indépendants nécessitant des données distantes et invariantes pendant une durée donnée.

Afin que de nombreuses applications puissent être parallélisées grâce à un noyau parallèle, celui-ci doit être développé de manière suffisament générique. Cette généricité devra résider d'une part dans sa structure "informatique" et d'autre part dans les choix stratégiques qui devront éviter d'être guidés par la seule application du rendu d'images.

3.1.2 Environnement

3.1.2.1 Environnement matériel

L'organisation de notre centre de calcul peut être résumée par la figure suivante :

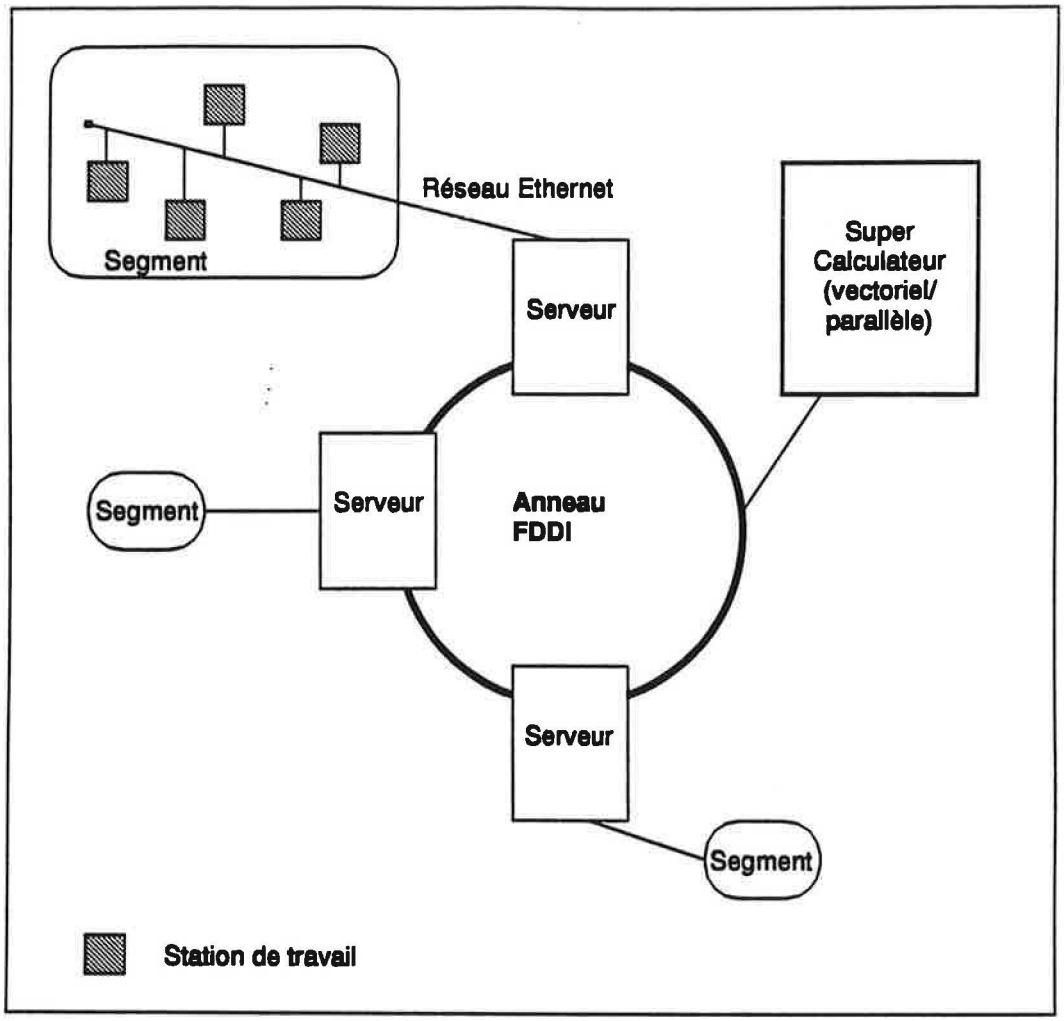

Figure 42 Organisation du centre de calcul

Dans la conception de ce centre de calcul, il existe deux types de réseaux :

1. Un réseau local Ethernet permettant la connexion entre les différents serveurs (serveurs NFS, serveurs de produits, serveurs dédiés à des applications spécifiques...) et les stations de travail (SUN, Silicon Graphics, HP et IBM).
2. Un réseau rapide sur fibre optique (anneau FDDI) reliant les surpercalculateurs aux serveurs.

Toutes les machines de ces réseaux disposent d'un système d'exploitation Unix.

Les "supercalculateurs" sont des machines qui se caractérisent par des performances particulièrement élevées essentiellement dans un environnement scientifique. Ces systèmes sont extrêmement puissants dans des applications impliquant des calculs flottants et utilisant des procédés de parallélisation et/ou de vectorisation. Ces calculateurs, d'autre part, sont caractérisés par des entrées-sorties très rapides.

Dans un environnement réseau, les supercalculateurs fournissent des services complémentaires par rapport à ceux offerts par les stations de travail.

Les machines dont nous disposons sont des calculateurs CRAY :

- Vectoriel : CRAY Y-MP

Ce calculateur est constitué de 8 processeurs (vectoriels) à mémoire partagée.

- Vectoriel : CRAY T90

Cette machine comprend 24 processeurs (vectoriels) à mémoire partagée.

- Massivement parallèle : CRAY T3D

Il est composé de 128 processeurs (DEC Alpha, processeurs scalaires) à mémoire distribuée. Les informations peuvent être échangées entre les noeuds à travers un réseau de topologie toroïdale. Cette machine ne possède aucun disque, le CRAY Y-MP sert de frontal pour lancer les exécutions.

- Massivement parallèle : CRAY T3E

Il dispose de 152 processeurs élémentaires, appelés PEs, répartis en 150 PEs utilisateur (128 pour l'application, 22 pour les commandes dont 2 redondants) et 2 PEs supports (pour le système). Les PEs "application" sont destinés aux applications parallèles uniquement, les PEs "commande" permettent les commandes, compilations et exécutions de codes mono-processeur. Les PEs "système" sont des serveurs. Les PEs redondants sont utilisés en cas de problème sur les autres PEs.

Notre travail devant permettre l'exécution en parallèle d'un code de lancer de rayon, nous utiliserons plus particulièrement les calculateurs CRAY massivement parallèles (T3D et T3E) et le réseau de stations de travail. L'exploitation de deux architectures matérielles très différentes nous imposera le développement d'un code utilisant un langage et des bibliothèques standardisés.

3.1.2.2 Environnement de programmation

Nous souhaitions utiliser un langage objet afin de pouvoir écrire un code structuré et potentiellement réutilisable. En effet, la programmation orientée objet devrait permettre de généraliser sans trop de difficultés l'utilisation d'un noyau parallèle. Il sera ainsi possible de sortir du cadre strict du lancer de rayon pour d'autres applications ayant des caractéristiques proches : une base de données dépassant la taille de la mémoire d'un processeur et un grand nombre de calculs "indépendants".

Nous disposons de compilateurs C++ standard sur l'ensemble de nos machines cibles. Ce langage ne comportant aucune fonctionnalité pour le parallélisme, il faut lancer un exécutable par tâche. Les tâches communiquent entre elles à l'aide d'une bibliothèque de communications.

Le seul protocole de communication inter-processus disponible sur l'ensemble de ces machines est celui de l'envoi de messages : ainsi notre application parallèle sera composée d'un ensemble de processeurs communiquant entre eux par envoi de messages. Toutefois, nous savons que ce choix est loin d'utiliser au mieux les capacités des CRAY. Par exemple, le débit assuré par ce protocole n'est de l'ordre que de 10% de la vitesse du réseau de communication accessible via shmem, la librairie de communication par mémoire partagée spécifique aux machines CRAY.

Dans la section suivante, nous allons détailler le protocole de communication choisi.

3.1.2.3 Protocole de communication : envoi de messages

Introduction

Alors que la construction de machines parallèles se développait, une approche plus économique est apparue : la ferme de stations. Plutôt que de concevoir des machines spécifiques, il a été proposé d'utiliser les stations de travail qui sont naturellement reliées par un réseau et de les faire communiquer entre elles comme le feraient les processeurs d'une machine parallèle à mémoire distribuée. Il y a donc création d'une machine parallèle virtuelle communiquant par envoi de messages.

Cette communication est donc utilisée par deux types d'architectures : des machines reliées en réseau et des machines parallèles.

Les primitives de communication proposées par une bibliothèque d'envoi de messages permettent les fonctionnalités suivantes :

1. La création et la destruction de processus.
2. L'obtention d'informations sur la configuration de la machine.
3. La configuration dynamique de la machine.
4. L'envoi de signaux.
5. La création de groupes de processus.
6. L'envoi de messages à d'autres processus.
7. La réception bloquante ou non bloquante de messages.
8. La création de buffers de communication.
9. L'empaquetage et le dépaquetage de données.
10. Des opérations collectives de communication.
11. Des opérations de synchronisation de processus.

Actuellement, il existe deux bibliothèques "standard" proposant des mécanismes d'envoi de messages aussi bien sur réseaux de machines que sur machines parallèles :

1. PVM (Parallel Virtual Machine), [GBD+94].
2. MPI (Message Passing Interface), [GLS94].

Bien que ces bibliothèques proposent des services proches, elles sont basées sur des philosophies différentes.

PVM (Parallel Virtual Machine)

PVM fut la première bibliothèque destinée à l'émulation d'une machine parallèle à mémoire distribuée. Son développement a commencé en 1989 au Oak Ridge National Laboratory (ORNL). PVM, qui a été longtemps la seule librairie portable d'échange de messages, propose une librairie de fonctions de communication utilisables en FORTRAN et en C.

PVM a été conçu pour créer une machine parallèle virtuelle à partir d'un réseau de machines hétérogènes. PVM permet donc une programmation par échanges de messages sur une machine de type MIMD. En plus des fonctionnalités classiques, PVM assure aussi une gestion des tâches dynamique et propose de nombreux outils permettant d'avoir des informations sur les machines connectées.

Les CRAY parallèles étant des machines SPMD dont les processeurs sont monotâches, l'implémentation de PVM sur ces machines comporte quelques restrictions par rapport au PVM standard :

1. Il n'y a qu'un seul processus par processeur.
2. Tous les processus de l'application sont lancés simultanément et exécutent le même programme (machine SPMD).

Enfin il faut positionner de nombreuses variables d'environnement. Ces variables déterminent le nombre de messages autorisés à la réception par chacun des processeurs (PVM_SM_POOL) ainsi que la taille des messages (PVM_DATA_MAX) et permettent d'optimiser l'envoi de messages en précisant par exemple la taille minimale pour l'allocation de buffers (PVM_MAX_PACK).

L'ensemble de ces variables d'environnement est décrit dans l'annexe G.

MPI (Message Passing Interface)

Suite à l'engouement pour PVM, il a semblé nécessaire de donner un contexte plus formel au paradigme que constitue l'envoi de messages et plus particulièrement de définir un standard pouvant

servir de cadre aux implémentations futures. Ainsi MPI a été mis en place en 1993 suite aux réunions des représentants de 40 organismes et laboratoires américains et européens, comprenant la majorité des constructeurs de machines parallèles et des laboratoires concernés (dont l'ORNL).

Ce standard est donc le fruit d'un consensus des principaux acteurs du parallélisme. Dans sa version initiale, MPI a été développé pour des machines parallèles de type SPMD : la gestion des tâches était purement statique. Toutefois, la version suivante, MPI-2, reprend la plupart des fonctionnalités propres à PVM (gestion dynamique des tâches, implémentation sur réseaux hétérogènes...).

MPI propose une interface de programmation de plus de 130 fonctions (PVM en propose moins de 60) et autorise :

1. Des communications bloquantes ou non-bloquantes avec des modes d'émission différents.
2. De très nombreuses communications collectives.
3. La mise en place d'une topologie virtuelle des processus.
4. Des outils de gestion de groupes de processus.

Conclusion

Le tableau suivant permet une comparaison de ces deux librairies :

PVM (Parallel Virtual Machine)	MPI (Message Passing Interface)
Standard né sur des fermes de stations	Standard mis en place pour des machines parallèles génériques
Gestion dynamique des tâches	Gestion statique dans MPI-1, dynamique avec MPI-2
Bonne portabilité	Bonne portabilité
Pas de routine de topologie de réseau	Cartographie du réseau possible
Format de représentation externe des données (XDR)	Aucune représentation externe spécifiée
Les numéros identifiant les tâches ne sont pas contiguës	Les numéros identifiant les tâches sont contiguës
Gestion des groupes limitée	Facilités de gestion des groupes
Communication inter-application difficile Assez peu de modes de communication différents	Communication inter-application facilitée De nombreux modes de communication différents
Envois non bloquants	Envois bloquants et non bloquants
Communication globale restreinte	De nombreuses routines de communication globale
Prototypes très différents entre le C et le Fortran	Prototypes peu différents entre le C et le Fortran, prototype C++ en MPI-2

Table 2 Tableau comparatif et récapitulatif de PVM et MPI.

Finalement, il apparaît qu'avec l'arrivée de MPI-2, MPI reprend toutes les fonctionnalités de PVM et en propose de nouvelles. Toutefois, ces deux bibliothèques restent très proches, aussi, afin d'assurer une portabilité maximale, nous avons créé une bibliothèque d'échanges de messages encapsulant MPI et PVM qui propose l'ensemble des fonctionnalités communes et qui permet donc de choisir indifféremment l'emploi de l'une ou de l'autre lors de l'exécution.

3.1.3 Caractéristiques du noyau parallèle

Le noyau parallèle à réaliser doit autoriser le traitement de scènes ne pouvant être contenues dans la mémoire d'un unique processeur. De plus, il doit être portable afin de pouvoir être utilisé sur des machines parallèles très diverses (réseaux de stations de travail, CRAY T3D, CRAY T3E...). Enfin, son utilisation ne doit pas se limiter à l'application du lancer de rayon destiné à la synthèse d'images. Ces contraintes vont permettre d'orienter nos choix en matière de parallélisation :

1. La base de données doit pouvoir être distribuée sur l'ensemble des processeurs disponibles.
2. Notre code ne doit pas chercher à tirer parti de l'architecture de l'un ou l'autre des réseaux.
3. L'utilisation des CRAY impose la présence d'un processus unique par processeur.
4. Le code doit rester performant dans un environnement massivement parallèle.
5. L'utilisation de plusieurs architectures parallèles impose des communications utilisant des bibliothèques standardisées.
6. Le noyau parallèle doit avoir une certaine généricité : les applications comportant un grand nombre de calculs indépendants nécessitant des données invariantes au cours d'un pas de temps donné doivent pouvoir l'utiliser pour leur parallélisation.

A partir de ces contraintes et de l'étude des précédents travaux, nous pouvons déjà orienter notre réflexion.

La contrainte (1) impose le choix d'une méthode à flots de rayons ou à flots de données. Les méthodes de partage d'image sans flots et coopératives sont donc à proscrire. De plus, la méthode à flots de rayons utilisée seule n'est pas envisageable, car certaines scènes peuvent conduire à des déséquilibres ingérables.

L'aspect multi-architectures ne permet pas de profiter de la topologie d'un réseau : on ne devra pas chercher à faire des communications par "le plus court chemin" (contrainte (2)).

La contrainte (3) interdit l'exécution de plusieurs tâches concurrentes sur un même processeur et donc toutes les méthodes d'équilibrage de charges basées sur cette concurrence.

La contrainte (4) est une contrainte forte, qui n'a été que rarement posée dans les travaux publiés. Elle n'autorise pas les architectures de type maître-esclaves et impose des mécanismes d'équilibre de charges dynamiques.

La contrainte (5) induit des communications par échange de messages (PVM ou MPI). Le coût de ce type de communication est très élevé. Il ne permet pas d'envisager un noyau synchrone efficace et ne rend pas non plus raisonnable l'ajout d'une surcouche de type mémoire partagée virtuelle.

Enfin, la contrainte (6) va nous conduire à prendre du recul par rapport aux choix guidés par l'application synthèse d'images. De plus, elle a contribué au choix d'un langage objet qui facilite le développement de code générique.

Nos contraintes étant posées, nous allons pouvoir chercher à déterminer précisément la stratégie de parallélisation qui nous convient.

3.2 Choix d'une méthode de parallélisation par le biais de modélisations

Maintenant que nous avons explicité nos contraintes, nous allons définir l'algorithme parallèle à réaliser.

L'étude bibliographique a montré que deux méthodes performantes ont été utilisées pour résoudre des problèmes proches des nôtres : l'échange de rayons ou l'échange d'objets entre les processeurs. Toutefois, aucune comparaison convaincante entre ces deux algorithmes n'a pu être trouvée dans la littérature. En effet, les recherches sont menées sur des machines parallèles très différentes, les débits et les architectures des réseaux de communication sont très variables, les scènes représentées sont souvent distinctes malgré les efforts de standardisation de Haines [Hai87] et les algorithmes séquentiels servant de base aux parallélisations sont rarement les mêmes. A celà, il faut ajouter que les résultats fournis par ces travaux sont souvent très partiels et les métriques utilisées ne sont pas toujours clairement définies.

Ne pouvant choisir entre les algorithmes à flots de données et à flots de rayons a priori, nous allons réaliser leur comparaison. Le développement de ces deux algorithmes étant un travail très lourd, il est plus raisonnable et suffisament informatif de comparer uniquement deux modélisations de ces algorithmes. Ces modélisations numériques seront réalisées à partir d'une étude de complexité des deux algorithmes.

Les résultats de ces comparaisons nous serviront alors de base de réflexion pour l'élaboration de notre noyau parallèle.

3.2.1 Hypothèses de travail

Avant de présenter nos modélisations, il nous faut préciser les hypothèses que nous allons formuler afin de simplifier les algorithmes à modéliser.

Nous posons donc les hypothèses suivantes :

1. Le calcul d'une image se fait par étapes. A chaque étape, chaque rayon va intersecter un objet. Si des communications sont nécessaires afin de réaliser un calcul d'intersection, alors elles auront lieu pendant l'étape et les calculs d'intersection associés également.
2. Une étape se termine quand tous les processeurs ont terminé toutes leurs communications et tous leurs calculs.
3. La distribution des objets sur les processeurs se fait de manière équilibrée.
4. Le nombre de rayons est très supérieur au nombre d'objets.
5. Tous les objets ont la même probabilité d'être intersectés, quelle que soit l'étape de calcul.
6. Chaque objet est sous la responsabilité d'un unique processeur. Toutefois, des copies peuvent résider dans les mémoires cache des autres processeurs.
7. Chaque processeur est capable de savoir quel processeur possède un objet manquant donné.
8. Aucun algorithme d'équilibrage de tâches n'est implémenté.
9. On suppose que la taille d'un objet est supérieure à celle d'un rayon et que tous les objets ont la même taille.
10. On assigne un temps de calcul fixe pour chaque opération.
11. On ne traite ni les rayons d'ombre ni les rayons réfractés.

Nous sommes conscients que, sous ces hypothèses, il n'est pas possible de représenter le comportement réel d'un algorithme parallèle pour le lancer de rayon. Toutefois, elles vont permettre de réaliser une étude qualitative du comportement général des algorithmes à flots de données et de rayons.

3.2.2 Algorithmes retenus

Les hypothèses de travail étant posées, nous pouvons décrire les deux algorithmes que nous modéliserons par la suite. Nous pouvons les présenter par le pseudo-code suivant :

1. D'abord les objets sont distribués sur les N processeurs, chacun recevant un pourcentage fixé et noté X des objets (chaque processeur n'est responsable que de $100/N$ % des objets, la mémoire cache comprend d'autres objets distribués au hasard).
2. Les rayons primaires sont distribués sur les N processeurs, chacun d'eux recevant $100/N$ % des rayons.
3. A chaque étape, $(100-X)$% des rayons ont besoin d'objets ne figurant pas sur leur processeur.
4. Alors les stratégies suivantes de flots de rayons ou flots de données sont utilisées :
   a. Algorithme à flots de données :
      - Chaque fois qu'un rayon nécessite un objet, une requête est envoyée au processeur qui possède cet objet.
      - Le processeur renvoie une copie de l'objet et le calcul d'intersection est réalisé.
   b. Algorithme à flots de rayons :
      - Les rayons nécessitant un objet pour être calculés sont envoyés au processeur qui possède cet objet (les envois de rayons sont bufferisés, chaque processeur n'envoie qu'un packet de rayons à chacun des autres processeurs).
      - Quand tous les rayons sont arrivés sur le processeur qui possède l'objet demandé, les calculs d'intersection sont réalisés.
5. Quand un calcul d'un rayon est terminé, le rayon est supprimé et un nouveau rayon est généré. A partir d'une certaine étape, des rayons commencent à disparaître car on suppose qu'ils ont terminé leur parcours dans la scène. Certains rayons supprimés ne sont donc pas remplacés; on quitte ce que nous appellons le régime permanent.
6. L'étape suivante commence quand tous les processeurs ont terminé de calculer leurs rayons, l'algorithme reprend en 3.
7. L'algorithme est terminé quand tous les rayons ont quitté la scène.

3.2.3 Modélisation

Afin de comparer les techniques de parallélisation par flots de données et flots de rayons, nous modélisons les algorithmes présentés précédemment. Nos modèles seront basés sur des séries récurrentes exprimant un temps de calcul d'image pour chacun des deux algorithmes en fonction du nombre de processeurs et de la fraction de la base de données présente dans la mémoire de chacun des processeurs.

Afin d'exprimer ces modèles, nous posons les notations suivantes :

$\mathbf{n_0}$ le nombre total de rayons à calculer au début de l'algorithme.
nb le nombre de processeurs utilisés, $nb \geq 2$.
x la fraction de la base de données présente sur chaque processeur, $0<x\leq 1$.

$\mathbf{e_{P_i->P_j,E_k}}$ qui représente à l'étape k (E_k) :

soit le nombre d'envois de rayons du PE_i vers le PE_j, pour l'algorithme à flots de rayons,

soit le nombre de rayons du PE_i ayant besoin d'un objet du PE_j, pour l'algorithme à flots de données.

$\mathbf{e_{P_i->P_j,E_k}} \geq 0$

$\mathbf{n_{P_i,E_k}}$ le nombre de rayons présents sur le PE_i à l'étape k, $\mathbf{n_{P_i,E_k}} \geq 0$.

On retrouve ces notations sur la figure suivante :

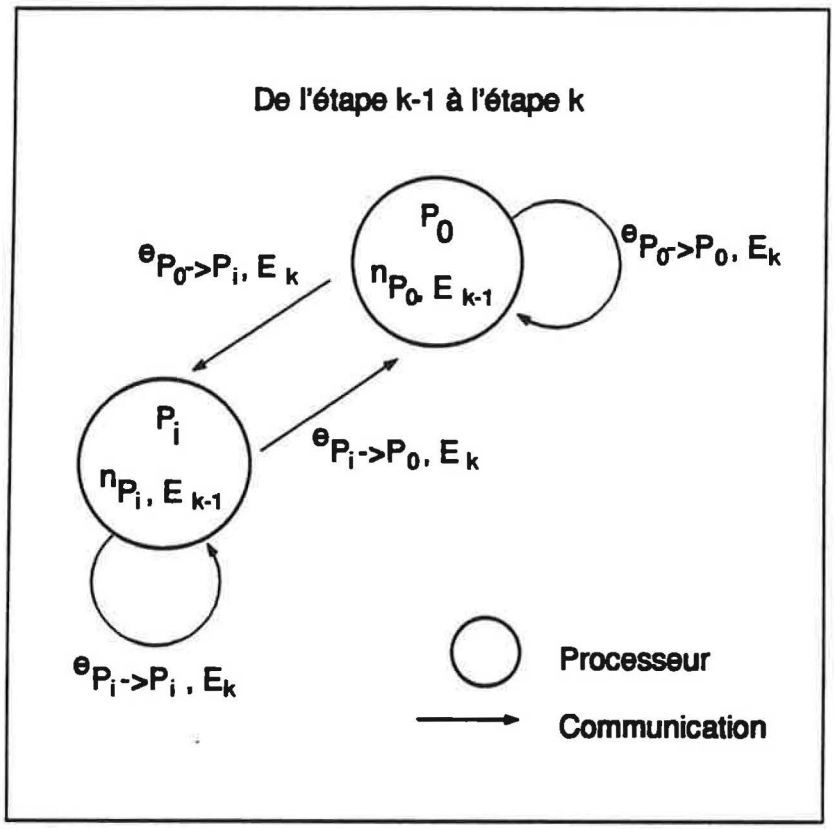

Figure 43 Schéma du modèle

Dans la suite, les équations de nos modèles sont exprimées uniquement pour le régime "permanent", c'est à dire le régime pendant lequel le nombre total de rayons sur la machine parallèle simulée reste constant.

Cherchons à exprimer $\mathbf{e_{P_i->P_j,E_k}}$.

Chaque processeur possédant la fraction x de la base de données, une proportion de x rayons présents va être calculée sur le processeur i sans qu'il y ait de communications :

$$\forall (i,k) \in (\mathbb{N} \times \mathbb{N}^*),\ 0 \leq i < nb,$$
$$\mathbf{e}_{P_i->P_i,E_k} = x * \mathbf{n}_{P_i,E_{k-1}} \tag{19}$$

Il y aura donc une proportion de $1-x$ des calculs présents à l'étape $k-1$ qui ne pourra pas être effectuée directement sur le processeur i. Comme il y a $nb-1$ processeurs, une fraction de $\frac{1-x}{nb-1}$ des

calculs présents à l'étape *k-1* peut être traitée grâce à des communications (envois de calculs ou demandes de données) avec un autre processeur j :

$$\forall (i,j,k) \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N}^*),\ 0 \leq i,j < nb \text{ et } i \neq j$$
$$\mathbf{e}_{P_i->P_j,E_k} = \frac{1-x}{nb-1} * \mathbf{n}_{P_i,E_{k-1}} \tag{20}$$

Les équations (19) et (20) n'induisent aucun déséquilibre de tâche entre les différents processeurs, puisque ils vont tous posséder le même nombre de rayons ou demander le même nombre de données. Aussi, afin que la comparaison de nos modèles puisse tenir compte de l'effet d'éventuels déséquilibrages, nous avons ajouté une variable aléatoire $\xi_{\mathbf{P_i->P_j,E_k}}$ permettant d'introduire un déséquilibrage de tâche à chaque itération k pour chaque processeur i en rapport avec un processeur j ($| \xi_{\mathbf{P_i->P_j,E_k}} | \leq 1$).

En introduisant cette variable aléatoire $\xi_{\mathbf{P_i->P_j,E_k}}$ dans les équations (19) et (20), nous obtenons l'expression suivante pour $\mathbf{e}_{\mathbf{P_i->P_j,E_k}}$:

$$\forall (i,j,k) \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N}^*),\ 0 \leq i,j < nb,$$
$$si\ i = j,\ \mathbf{e}_{P_i->P_j,E_k} = x * \mathbf{n}_{P_i,E_{k-1}} * \left(1 + \xi_{P_i->P_j,E_k}\right)$$
$$si\ i \neq j,\ \mathbf{e}_{P_i->P_j,E_k} = \frac{1-x}{nb-1} * \mathbf{n}_{P_i,E_{k-1}} * \left(1 + \xi_{P_i->P_j,E_k}\right) \tag{21}$$

Exprimons maintenant le nombre de rayons qu'un processeur doit traiter à une étape donnée k. Pour l'algorithme à flots de données, le nombre de rayons est constant en régime permanent :

$$\forall i \in \mathbb{N},\ 0 \leq i < nb,$$
$$\mathbf{n}_{P_i,E_k} = \frac{n_0}{nb} \tag{22}$$

Par contre pour l'algorithme à flots de rayons, le nombre de rayons d'un processeur i à l'étape k est la somme des rayons qui ont besoin du processeur i pour être traités :

$$\forall (i,j) \in (\mathbb{N} \times \mathbb{N}),\ 0 \leq i,j < nb,$$
$$\mathbf{n}_{P_i,E_k} = \sum_j \mathbf{e}_{P_j->P_i,E_k} \tag{23}$$

L'équation (21) est vraie à la condition que l'utilisation de la variable $\xi_{\mathbf{P_i->P_j,E_k}}$ n'induise pas la création ou la disparition de rayons en régime permanent. Aussi il faut que :

$$\forall i \in \mathbb{N},\ 0 \leq i < nb,$$
$$\sum_i \mathbf{n}_{P_i,E_k} = n_0 \tag{24}$$

En remplaçant $\mathbf{n}_{\mathbf{P_i,E_k}}$ dans l'équation (24) par l'expression (23), on obtient :

$$\sum_i \sum_j \mathbf{e}_{P_j->P_i,E_k} = n_0 \tag{25}$$

Finalement, en remplaçant $\mathbf{e}_{\mathbf{P_i->P_j,E_k}}$ par l'expression (21), on obtient une équation reliant les $\xi_{\mathbf{P_i->P_j,E_k}}$:

$$\forall (i,j,k) \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N}^*),\ 0 \le i,j < nb,$$

$$\sum_j \left(x * {}^{n}_{P_i,E_{k-1}} * \left(1 + \xi_{P_i -> P_j,E_k}\right) + \sum_{i \neq j} \frac{1-x}{nb-1} * {}^{n}_{P_i,E_{k-1}} * \left(1 + \xi_{P_i -> P_j,E_k}\right)\right) = {}^{n}_{0} \qquad (26)$$

Utilisons les notations suivantes pour exprimer les temps de calcul de chacun des deux types d'algorithme :

soit T_{calcul} le temps pour calculer un rayon,
soit T_{envoi} le temps pour faire un envoi,
soit $T_{reception}$ le temps pour recevoir un envoi,
soit $T_{empaq\text{-}rayon}$ le temps pour empaqueter un rayon,
soit $T_{empaq\text{-}donnee}$ le temps pour empaqueter une donnée,
soit $T_{empaq\text{-}demande}$ le temps pour empaqueter une demande de donnée,
soit $T_{depaq\text{-}rayon}$ le temps pour dépaqueter un rayon,
soit $T_{depaq\text{-}donnee}$ le temps pour dépaqueter une donnée,
soit $T_{depaq\text{-}demande}$ le temps pour dépaqueter une demande de donnée.

Le temps de calcul nécessaire à un processeur P_i pour accomplir l'étape E_j à l'aide de l'algorithme à flots de rayons est noté TR_{P_i,E_j}. Il s'exprime de la façon suivante :

$$TR_{P_i,E_j} = \overbrace{{}^{n}_{P_i,E_j} * T_{calcul}}^{\text{rayons presents}} + \overbrace{T_{envoi} * (nb-1) + T_{empaq-rayon} * \sum_{k \neq i} {}^{e}_{P_{i->k},E_j}}^{\text{envoi de flots aux autres PEs}} + \underbrace{T_{reception} * (nb-1) + T_{depaq-rayon} * \sum_{k \neq i} {}^{e}_{P_{k->i},E_j}}_{\text{reception de flots des autres PEs}} \qquad (27)$$

On en déduit que le temps nécessaire à l'aide de l'algorithme à flots de rayons pour que tous les processeurs aient terminé d'accomplir l'étape E_j est le temps réalisé par le processeur le plus lent.

$$\mathbf{TR}_{\mathbf{E_j}} = Max_i \left(TR_{P_i,E_j} \right) \qquad (28)$$

Aussi le temps pour l'obtention du résultat final à l'aide de l'algorithme à flots de rayons peut s'exprimer par :

$$\mathbf{TR}_{\mathbf{total}} = \sum_j TR_{E_j} = \sum_j Max_i \left(TR_{P_i,E_j} \right) \qquad (29)$$

Effectuons le même travail pour pour l'algorithme à flots de données. Le temps de calcul nécessaire à un processeur P_i pour accomplir l'étape E_j à l'aide de cet algorithme est noté TD_{P_i,E_j}. Il s'exprime de la façon suivante :

$$TD_{P_i,E_j} = \overbrace{{}^{n}_{P_i,E_j} * T_{calcul}}^{\text{calculs presents}} + \overbrace{\left(T_{envoi} + T_{empaq-demande} + T_{reception} + T_{depaq-donnee}\right) * \sum_{k \neq i} {}^{e}_{P_{i->k},E_j}}^{\text{Envoi des demandes et reception des donnees}} + \underbrace{\left(T_{envoi} + T_{empaq-donnee} + T_{reception} + T_{depaq-demande}\right) * \sum_{k \neq i} {}^{e}_{P_{k->i},E_j}}_{\text{Envoi des donnees et reception des demandes}} \qquad (30)$$

On en déduit comme précédemment que le temps pour l'obtention du résultat final à l'aide de l'algorithme à flots de données peut s'exprimer par :

$$\mathrm{TD}_{\mathrm{total}} = \sum_{j} Max_{i}\left(TD_{P_i,E_j}\right) \tag{31}$$

3.2.4 Résultats

A partir de ces équations, nous implémentons notre modèle en fixant des temps de calcul pour les différentes opérations primitives.

Nos modèles étant très simplifiés, le résultat attendu est bien entendu qualitatif. Ce ne sont donc pas les valeurs de temps fixées qui sont importantes, mais leur rapport entre elles.

Nous exécutons le code issu de nos modélisations en faisant varier le nombre de processeurs utilisés *nb* et la fraction de la base de données présente sur chaque processeur x pour chacun des modèles. Nous réitérons plusieurs fois cette exécution en changeant les valeurs des temps de calcul afin de tester différents rapports pouvant représenter des cas réels.

Un exemple de valeurs données aux opérations primitives est présenté dans le tableau suivant :

Temps des opérations primitives	En unité de temps
T_{calcul}	1000
T_{envoi}, $T_{reception}$	10
$T_{empaq\text{-}rayon}$, $T_{depaq\text{-}rayon}$	50
$T_{empaq\text{-}donnee}$, $T_{depaq\text{-}donnee}$	100
$T_{empaq\text{-}demande}$, $T_{depaq\text{-}demande}$	10

Table 3 Exemple de valeurs données aux opérations primitives

En fait, les modifications de ces valeurs ont eu peu d'effet sur le résultat final. Les graphiques représentant l'algorithme le plus efficace pour une répartition de la base et pour un nombre de processeurs donné sont toujours de la même forme :

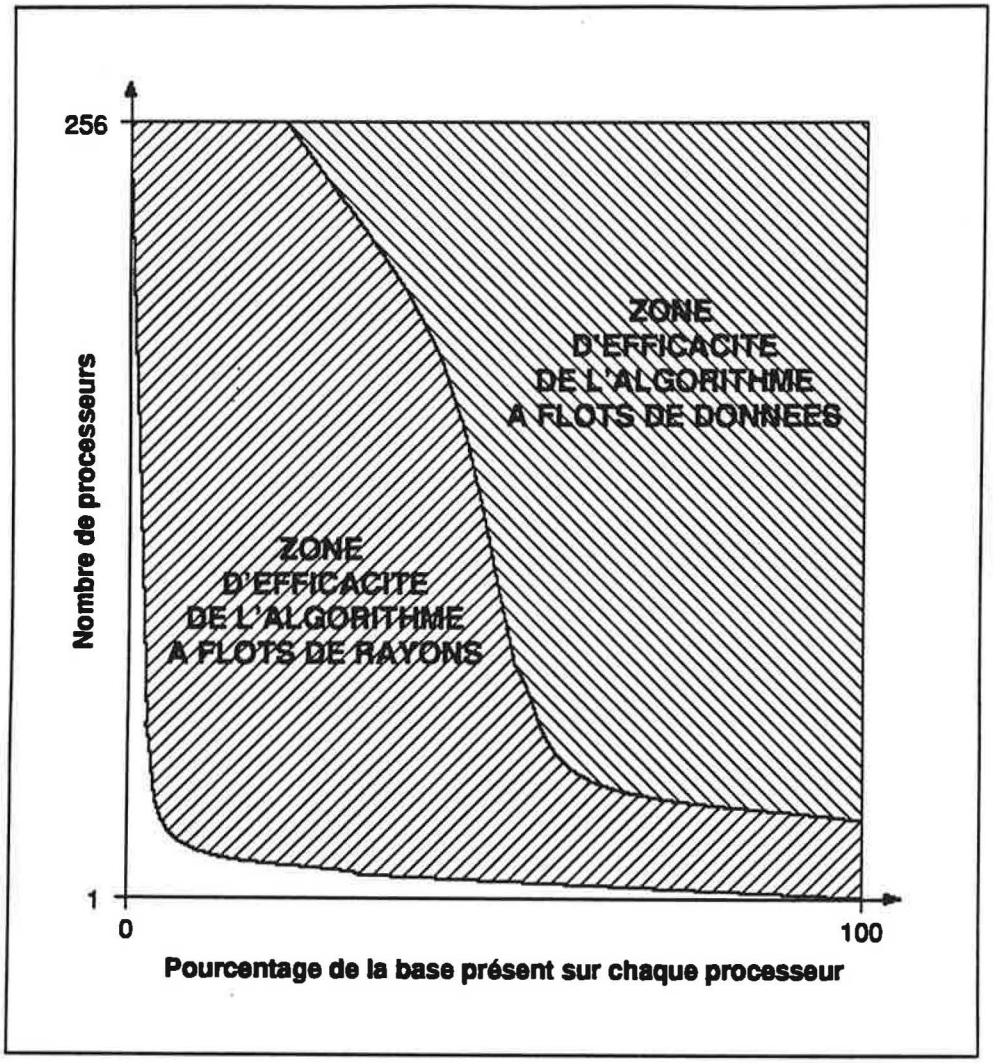

Figure 44 Zones d'efficacité des deux modèles

Il apparaît tout d'abord que chaque algorithme a son domaine d'efficacité. L'algorithme à flots de données a un meilleur comportement que celui à flots de rayons quand chaque processeur possède une grande partie de la base. Le résultat est par contre tout à fait inverse quand les processeurs ont une faible partie de la base.

Ensuite, on peut noter que l'augmentation du nombre de processeurs semble plutôt profitable à l'algorithme à flots de données.

Bien que la modélisation choisie simplifie beaucoup les deux algorithmes (pas de cohérence, ni d'équilibrage et un comportement synchrone...), elle laisse entrevoir une nouvelle voie de parallélisation pour l'algorithme de lancer de rayon : le choix entre l'un ou l'autre des algorithmes en fonction du nombre de processeurs et du taux de distribution de la scène.

3.2.5 Conclusion

Notre contrainte de distribution de la base de données nous conduit au choix d'un algorithme parallèle avec flots. Les algorithmes à flots de données et à flots de calculs ont été implémentés sur différents types d'architectures logiques :

1. Architecture maître-esclaves.
2. Architecture hiérarchique.

3. Architecture en anneau.

Nous avons montré qu'une architecture maître-esclaves ne pouvait être retenue en raison du caractère massivement parallèle de notre futur noyau. D'autre part, une structure hiérarchique ou en anneau est surtout souhaitable si elle s'appuie sur une architecture matérielle en accord avec elle. Sinon elle a tendance à provoquer une augmentation du nombre de communications et à figer les "degrés de liberté" pour l'équilibre des tâches.

L'utilisation du protocole de communication par envoi de messages nous dicte la réalisation d'un noyau parallèle asynchrone. Ainsi les différences de temps de communication, entre deux processeurs différents, liées à l'architecture physique de la machine employée apparaissent comme négligeables.

La représentation logique de notre architecture sera celle d'une machine parallèle dont le réseau est complètement connecté, mais dont le coût de communication est important. Chaque processeur exécutera un code identique et pourra communiquer indifféremment avec n'importe quel autre processeur de la machine parallèle.

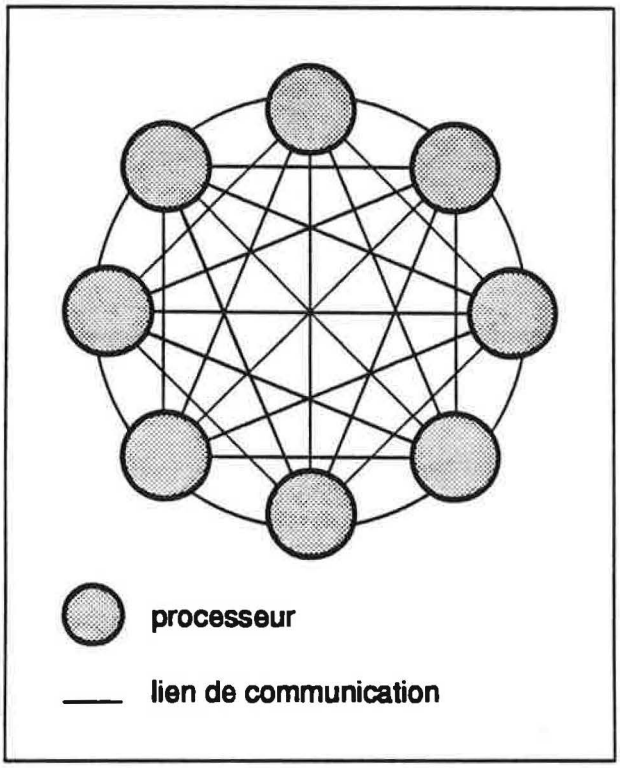

Figure 45 Réseau complètement connecté

Notre modélisation a montré que chacun des deux algorithmes étudiés, flots de rayons et flots de données, avait son domaine d'efficacité. Un algorithme de parallélisation, pour être performant quels que soient le nombre de processeurs et la distribution de la scène, doit permettre de choisir entre une stratégie à flots de données et une stratégie à flots de rayons.

Une première solution serait de déterminer en fonction du nombre de processeurs et du taux de distribution de la scène lequel des deux types de flots doit être utilisé pour le calcul d'une image donnée. Il y aurait donc un choix statique entre les deux algorithmes lors de la phase de précalcul.

Toutefois les résultats de notre modélisation peuvent être également analysés comme des tendances moyennes sur le temps de calcul total. Des changements de stratégie en cours d'exécution pourraient apporter une plus grande efficacité à l'algorithme.

Aussi un deuxième type de parallélisation peut être envisagé : chaque processeur pourra dynamiquement opter pour l'une ou l'autre des stratégies avec flots.

C'est cette voie, qui nous semble plus prometteuse que la précédente à choix statique, que nous allons nous efforcer d'implémenter.

Nous nous proposons donc de réaliser un algorithme de parallélisation asynchrone nommé à **flots mixtes** : les processeurs pouvent échanger aussi bien des données que des rayons. Chaque processeur va au cours du temps choisir, en fonction des informations qu'il possède sur son état et celui des autres processeurs, la stratégie la plus efficace en temps de calcul et en équilibrage de charge.

3.3 Présentation des algorithmes parallèles proposés

Le noyau parallèle que nous souhaitons réaliser ne se limitant pas au champ du lancer de rayon, nous ne considèrerons le lancer de rayon que comme un cas particulier d'utilisation de ce noyau. Nous ne parlerons donc plus de "rayons" et d' "objets de la scène", mais de calculs et de données.

Le principe de parallélisation que nous allons définir permet dynamiquement le choix entre l'envoi de données et l'envoi de calculs (algorithmes à flots mixtes). L'utilisation conjointe des deux types de flots autorise l'élaboration de nouvelles stratégies. Nous en présentons deux : un algorithme à **flots multi-informatifs** et un algorithme à **flots concurrents**.

3.3.1 Principe des algorithmes à flots mixtes

Les algorithmes à flots **mixtes** que nous proposons peuvent être décrits de la façon suivante :

Lors de la phase d'initialisation, tous les pixels à calculer sont distribués sur l'ensemble des processeurs de la machine parallèle mise à disposition. On fait de même avec les différents objets de la scène, seule la structure comprenant les volumes englobants est dupliquée.

La phase de calcul peut alors commencer : chaque processeur va chercher à calculer l'ensemble des pixels mis à sa disposition. La base de donnée étant répartie, des communications inter-processeurs vont être nécessaires afin que tous les calculs puissent être effectués (ces communications seront asynchrones afin de recouvrir le coût élevé en temps nécessaire à une communication). Aussi quand un processeur ne peut continuer le calcul d'un rayon en raison du manque d'un objet, il a deux possibilités :

1. Soit le processeur estime qu'il est préférable de disposer de cet objet, alors une demande est transmise au processeur la possédant. Celui-ci enverra en retour la donnée au demandeur.
2. Soit le processeur estime préférable d'envoyer le calcul à un processeur pouvant le calculer, alors le calcul est transmis à ce processeur.

Chaque processeur va donc au cours du temps choisir, en fonction des informations qu'il possède sur son état et celui des autres processeurs, la stratégie la plus efficace en temps de calcul et en équilibrage de charge. Nos algorithmes possèdent donc de façon intrinsèque un mécanisme dynamique d'équilibrage de charge.

Enfin quand tous les processeurs ont fini d'effectuer tous leurs calculs, le programme se termine.

Le fondement de nos deux algorithmes est le choix de l'envoi de calculs ou de données par chaque processeur en fonction de la connaissance de l'état des différents acteurs de la machine parallèle.

La transmission d'information relative à la charge de chacun des processeurs est donc essentielle. Nous présentons dans la section suivante notre méthode de diffusion de cette connaissance.

3.3.2 Diffusion de la connaissance de la charge de chacun des processeurs

Nos algorithmes à flots mixtes reposent sur la connaissance par chaque processeur de l'état de charge des autres processeurs de la machine. L'architecture logique de notre noyau parallèle n'étant pas centralisée, cette connaissance devra être diffusée entre les processeurs. Cette diffusion pourrait être réalisée en suivant des schémas de communication de type *échange total,* où chaque processeur d'un réseau qui en contient p doit communiquer son propre message à tous les autres - l'analyse du problème de la diffusion est traitée dans [Rum92] - . Cependant ces méthodes ne conviennent pas à la résolution de notre problème à cause des coûts de communication excessifs (p^2 messages à chaque diffusion) qu'elles entraînent par rapport à la nature des informations dont nous souhaitons disposer. En effet, nous n'avons pas besoin d'états de charge exacts, des tendances suffisent pour la bonne marche de nos critères de choix. Aussi nous pensons que la distillation d'informations partielles en continu permet l'obtention d'une connaissance satisfaisante pour chacun des processeurs.

Comme les algorithmes proposés entraînent des communications inter-processeurs, nous souhaitons profiter de celles-ci pour transmettre à moindre coût les états de charge des processeurs composant la machine parallèle.

Ainsi, chaque fois qu'un processeur (P) émet un message, ce message comprend en plus de ses propres informations, une entête comportant des informations sur l'état de charge de (P). Au bout d'un certain nombre de communications, chaque processeur connait l'état de charge de n'importe quel autre processeur si ces communications sont suffisamment bien réparties sur le réseau. Bien entendu, plus le nombre de messages est important, plus la connaissance sur les états de charge des autres processeurs est précise et à jour.

Par contre, si le nombre de communications inter-processeurs est faible, l'information possédée par chaque processeur risque d'être obsolète. Il faudrait alors mettre en place des mécanismes permettant l'envoi régulier de messages à tous les processeurs afin que les stratégies de choix entre les différents flots restent efficaces. Toutefois, le but de notre travail étant essentiellement de manipuler des bases de données qui devront être fortement distribuées, il y a très peu de risques qu'il y ait un manque de flux de communication. Cet envoi régulier de messages ne sera donc pas implémenté.

Le mécanisme de diffusion de la charge des processeurs ayant été explicité, nous allons pouvoir décrire précisément les deux algorithmes à flots mixtes que nous proposons : les algorithmes à flots multi-informatifs et à flots concurrents.

3.3.3 Algorithme à flots multi-informatifs

Notre première implémentation d'un algorithme à flots mixtes est celle d'un algorithme à flots multi-informatifs. Notre noyau parallèle permet l'envoi aussi bien de données que de calculs. De plus, l'augmentation de la taille d'un message induit une réduction de son coût par unité d'information. Il est alors naturel de proposer l'envoi de messages comportant en leur sein les deux types d'information.

Cet algorithme repose sur l'algorithme classique à flots de données, celui-ci ayant été optimisé grâce à une bufferisation des demandes de données (voir § 3.4.4.2). L'idée de base pour ce type de flots est que, lors de l'envoi d'une donnée à un processeur (A) par un processeur (B), on peut profiter

de cet envoi pour ajouter dans le message des calculs. Ces calculs sont ceux que le processeur (B) ne peut calculer sans demandes de données et que le processeur (A) pourrait calculer car il possède les données nécessaires.

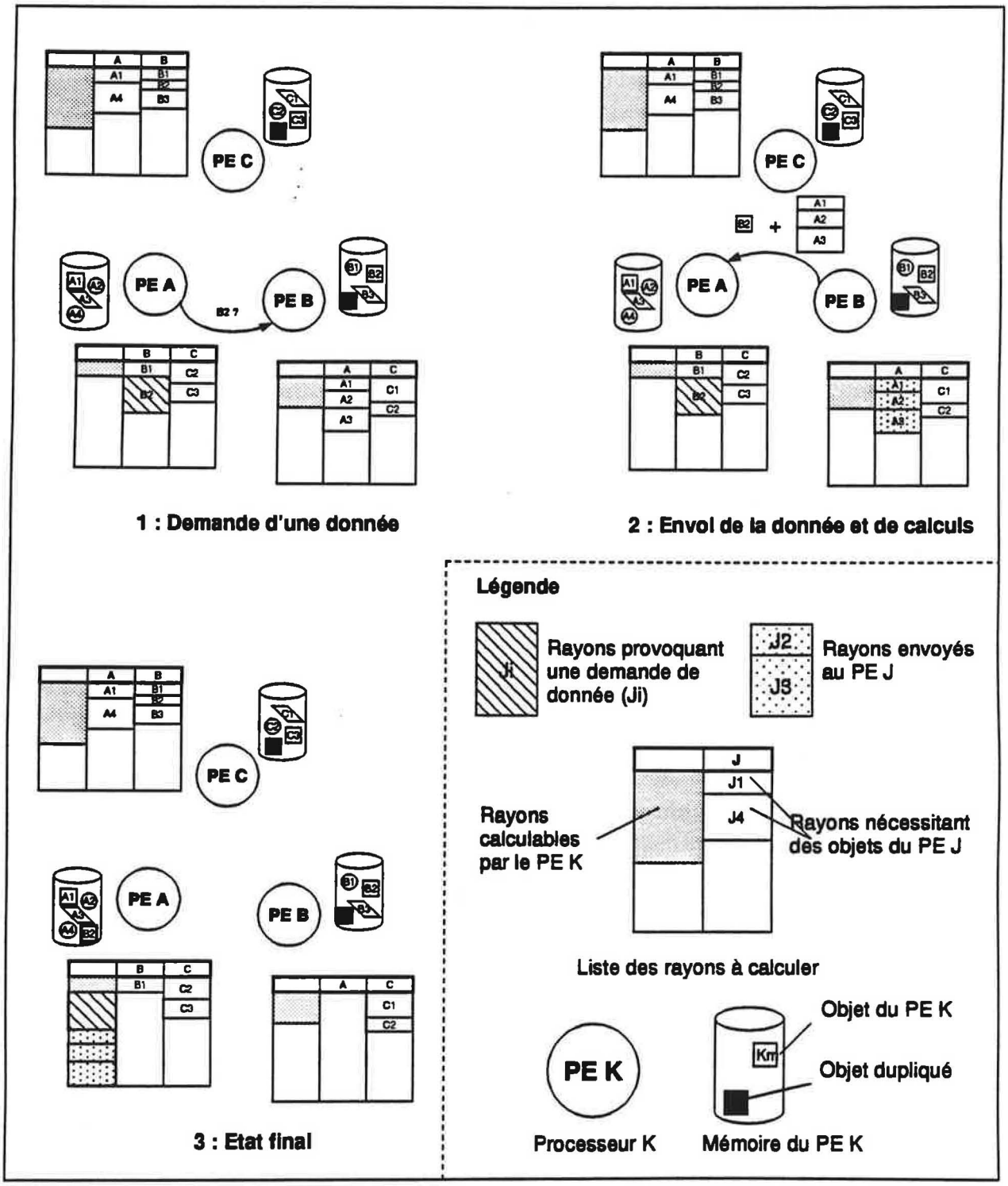

Figure 46 Schéma représentant l'envoi d'un message multi-informatif

Un processeur (A) ne pouvant pas effectuer une série de calculs va donc demander la donnée nécessaire à sa poursuite au processeur (B) qui la possède. Les demandes de données étant bufferisées (voir § 3.4.4.2), ce processeur (B) peut posséder des calculs dont la réalisation nécessite des données possédées par le processeur (A). Aussi le processeur (B), au lieu de renvoyer

simplement la donnée demandée par le processeur (A), va pouvoir profiter de cet envoi de messages en adressant, en plus de la donnée demandée, les calculs qui ont besoin des données du processeur (A). Cet envoi de calculs sera réalisé uniquement s'il ne contribue pas à provoquer un manque de calculs sur le processeur (B).

Grâce à cet algorithme, le nombre de messages échangés sera réduit au profit de messages de plus grande taille dont le coût par unité d'information est plus faible (des calculs ne pourront être envoyés que si leur nombre est suffisant). De plus, une forme d'équilibrage dynamique est assurée : pour qu'un processeur puisse envoyer des calculs avec la donnée demandée, il faut que le processeur émetteur n'ait pas une charge en calculs trop faible.

Plus précisément l'algorithme est de la forme suivante :

SI réception d'une demande de donnée du processeur p ALORS

SI ma charge est supérieure à une charge minimale ALORS

SI suffisament de rayons demandent des objets du processeur p ALORS

Envoi des rayons au processeur p.

Envoi de la donnée au processeur p.

3.3.4 Algorithme à flots concurrents

Le second type d'algorithme à flots mixtes proposé est un algorithme à flots concurrents. Notre étude a montré que chacun des algorithmes à flots de données et de calculs avait son domaine d'efficacité.

Il dépend de nombreux facteurs. Aussi nous souhaitons permettre un choix dynamique entre ces deux flots. Ainsi chaque processeur, grâce à sa connaissance de la charge des autres processeurs, pourra déterminer à tout moment en fonction de sa charge, de la charge des autres processeurs et du nombre de ses calculs nécessitant une donnée non présente, s'il est préférable d'envoyer à tel ou tel processeur une demande de données ou un ensemble de calculs.

Ainsi, un processeur ne pourra envoyer des calculs que si sa charge n'est pas trop faible. Ensuite, le choix se fera essentiellement en comparant les charges des processeurs émetteurs et destinataires. Si l'émetteur a une charge supérieure, il y aura envoi de calculs sinon demande de données. Ainsi, on assure en plus d'une optimisation du temps de calcul un équilibrage dynamique des charges. Celui-ci pourra éventuellement être complété par l'équilibrage dynamique qui est présenté ultérieurement.

Plus précisément l'algorithme est de la forme suivante :

SI ma charge est supérieure à une charge minimale ALORS

POUR chaque processeur

SI ma charge est supérieure à celle du processeur concerné ET le nombre de rayons à lui envoyer est suffisant ALORS

Envoi des rayons au processeur.

POUR chaque objet manquant

SI le nombre de rayons demandant cet objet est suffisant ALORS

Demande de l'objet au processeur qui le possède.

3.3.5 Algorithme d'équilibrage dynamique de charge

Assurer une bonne répartition des calculs sur les différents processeurs est fondamental pour tout code parallèle. Pour ce faire nous utilisons un équilibrage de charge à trois niveaux.

Tout d'abord, nous utilisons un équilibrage statique qui répartit de façon homogène tous les calculs sur l'ensemble des processeurs à l'initialisation.

Ensuite les deux algorithmes à flots mixtes disposent intrinsèquement d'un équilibrage dynamique de charge tendant à assurer en permanence une bonne répartition des calculs. Toutefois, cet équilibrage ne permet de réguler que des déséquilibres sur de petites fluctuations. Un processeur ayant un brusque et important déséquilibrage risquerait de ne pas pouvoir retourner à une position d'équilibre.

C'est pourquoi il nous paraît nécessaire de compléter notre dispositif par un équilibrage dynamique de charges plus direct. Nous offrons un équilibrage original où le processeur qui est à court de calculs s'adresse directement au meilleur processeur pouvant potentiellement lui en fournir. Cet équilibrage pourra être utilisé aussi bien par les algorithmes à flots mixtes que par l'algorithme à flots de données.

Grâce à notre mécanisme de diffusion des charges des processeurs, chaque processeur possède une certaine connaissance de la charge des autres processeurs de la machine. Aussi quand un processeur est en déficit de charge - un processeur n'attend pas de ne plus avoir de calculs pour en faire une demande, elle est faite à partir d'un seuil critique -, il peut s'adresser directement au processeur qui est potentiellement le meilleur donneur en consultant sa table de charges. Ce processeur donneur, s'il est effectivement capable de fournir des calculs, va lui envoyer un paquet de calculs. La taille de ce paquet sera inférieure à une taille maximale afin d'éviter des effets de "ping-pong" : si le nombre de calculs envoyé est trop important, le processeur donneur risque à son tour d'être à court de calculs.

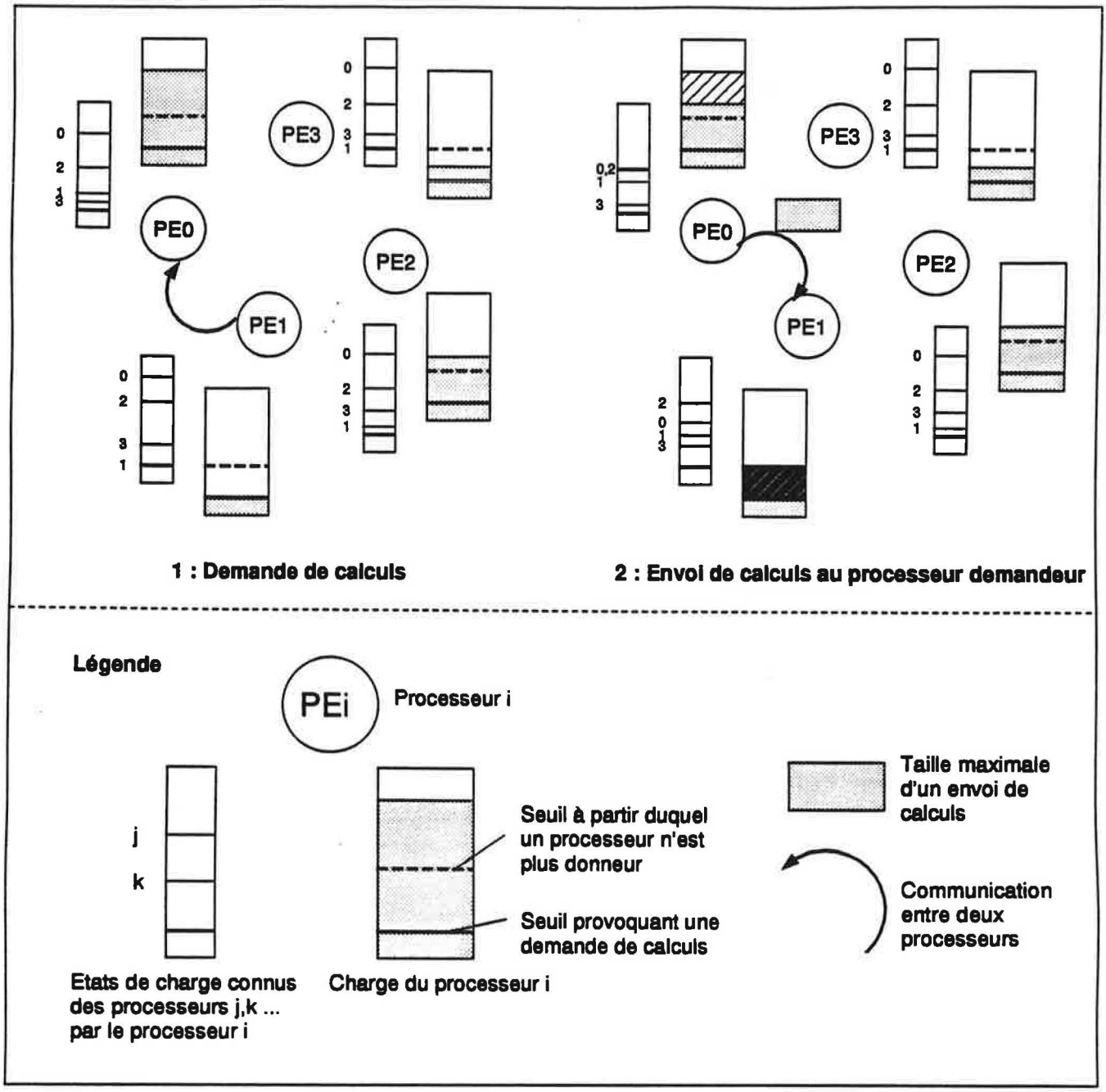

Figure 47 Nouvel équilibrage dynamique

Par contre, si le processeur n'est plus en mesure d'envoyer des calculs - un seuil, à partir duquel les processeurs cessent d'être donneurs, a également été fixé - , il va se contenter d'envoyer au processeur demandeur un message qui aura comme effet de réactualiser sa connaissance des charges et donc le processeur en attente de calculs pourra choisir un meilleur donneur.

L'envoi de calculs est réalisé suivant une stratégie particulière. En effet, grâce aux mécanismes de bufferisation mis en place pour la gestion de flots de données et de rayons, le processeur donneur va pouvoir "choisir" les calculs qu'il va envoyer. Il va commencer par privilégier l'envoi des calculs qui nécessitent une donnée présente sur le processeur demandeur, puis il enverra des calculs nécessitant une donnée d'un autre processeur et enfin, éventuellement, des calculs qu'il aurait pu traiter directement.

3.4 Mise en oeuvre

Nos algorithmes ayant été expliqués dans leurs grandes lignes, nous allons détailler leurs caractéristiques d'implémentation.

Avant de présenter les particularités de nos algorithmes parallèles, nous allons décrire la méthodologie de développement de notre noyau et l'algorithme séquentiel de lancer de rayon utilisé comme support pour notre travail. Ensuite, nous présentons les spécificités liées aux algorithmes parallèles asynchrones à mémoire distribuée, puis celles provenant de l'aspect flots de données et enfin celles résultant de l'utilisation de flots de calculs.

3.4.1 Méthodologie de développement

Le noyau parallèle que nous voulons réaliser nécessite l'utilisation de messages variés et de communications asynchrones. Il est périlleux de se lancer directement dans une telle entreprise car on risquerait d'aboutir à un programme comportant de nombreux blocages mortels (deadlock).

Nous avons donc décidé d'aborder ce projet par étapes de développement :

1. Algorithme de lancer de rayon séquentiel.
2. Parallélisation avec duplication de la base de données.
3. Parallélisation par échanges de données.
4. Parallélisation par échanges de données avec équilibrage dynamique.
5. Parallélisation par échanges de calculs.
   (Comparaison des parallélisations par échanges de calculs et de données et validation de la modélisation)
6. Parallélisation par échanges de flots mixtes.

Chacune de ces six étapes seront conclues par une validation.

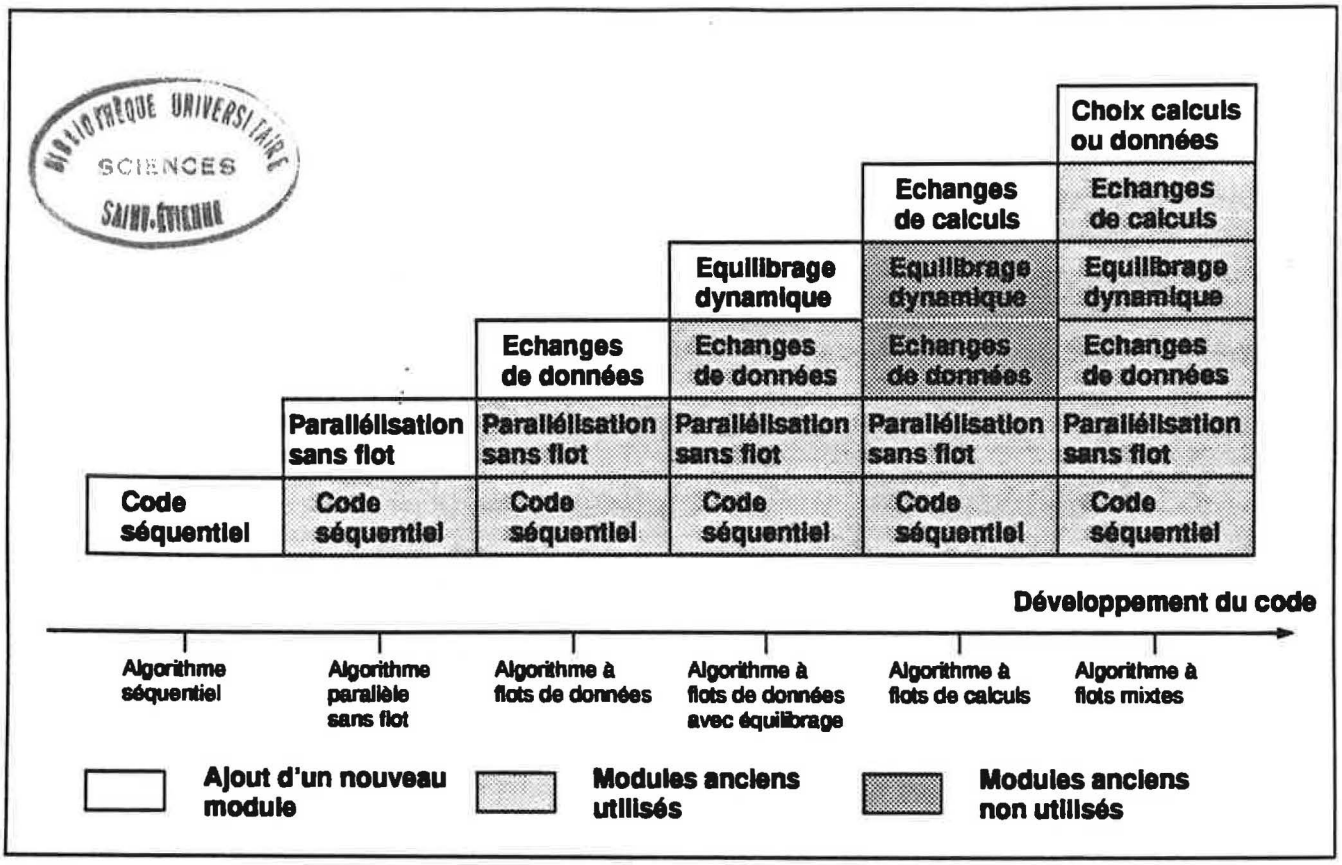

Figure 48 Etapes de développement de l'algorithme parallèle

L'algorithme parallèle implémenté permet deux types de parallélisme :

1. Un parallélisme sur les données : distribution des objets de la scène et échanges d'objets entre les processeurs.
2. Un parallélisme sur les calculs : distribution des rayons et échanges de rayons entre les processeurs.

Pour ce faire, nous avons réalisé un noyau parallèle pour l'algorithme de lancer de rayon avec flots de données. Une fois qu'il a été validé, avec en particulier sa méthode originale d'équilibrage dynamique de charge, nous l'avons complété afin d'obtenir un noyau parallèle à flots de calculs. Enfin, nous avons réalisé à partir de ces deux stratégies un noyau à flots mixtes, celui-ci comportant deux modes d'utilisation : à flots multi-informatifs et à flots concurrents.

Cette construction modulaire autorise l'utilisation de chacune de ses fonctionnalités indépendamment : flots de données, flots de calculs, flots multi-informatifs et à flots concurrents. Une comparaison expérimentale de ces différents algorithmes est réalisée dans le chapitre suivant.

3.4.2 Algorithme séquentiel de lancer de rayon

Nous souhaitions utiliser comme base de notre noyau parallèle un logiciel de lancer de rayon pouvant être adapté à d'autres applications. Nous avons choisi d'utiliser l'algorithme séquentiel "Object-Oriented Ray Tracing" (OORT) conçu par Wilt [Wil94]. En effet, OORT apporte un logiciel de lancer de rayon entièrement orienté-objet dont les classes sont particulièrement bien documentées. Il a donc été possible de le transformer assez facilement en un code de lancer de rayon autorisant également le suivi de particules.

3.4.2.1 Caractéristiques

OORT est un logiciel de lancer de rayon permettant de traiter des objets d'une grande variété :

1. Plans,
2. Anneaux,
3. Sphères,
4. Parallélépipèdes,
5. Polygones,
6. Quadriques (Sphères, cônes, cylindres, ellipsoïdes, hyperboloïdes, paraboloïdes...), d'équation :
   $Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + 2Gx + 2Hy + 2Jz + K = 0$

A partir de ces objets simples, il est possible de créer des objets complexes à l'aide d'un arbre de construction (appelé en anglais Constructive Solid Geometry, CSG) [Rot82]. Cette construction est obtenue en réalisant des opérations booléennes (l'union, la différence et l'intersection) sur les objets. Chaque objet CSG peut donc s'exprimer par un arbre dont les opérations sont placées sur les noeuds et les objets simples sur les feuilles. La figure suivante présente un exemple d'objet CSG :

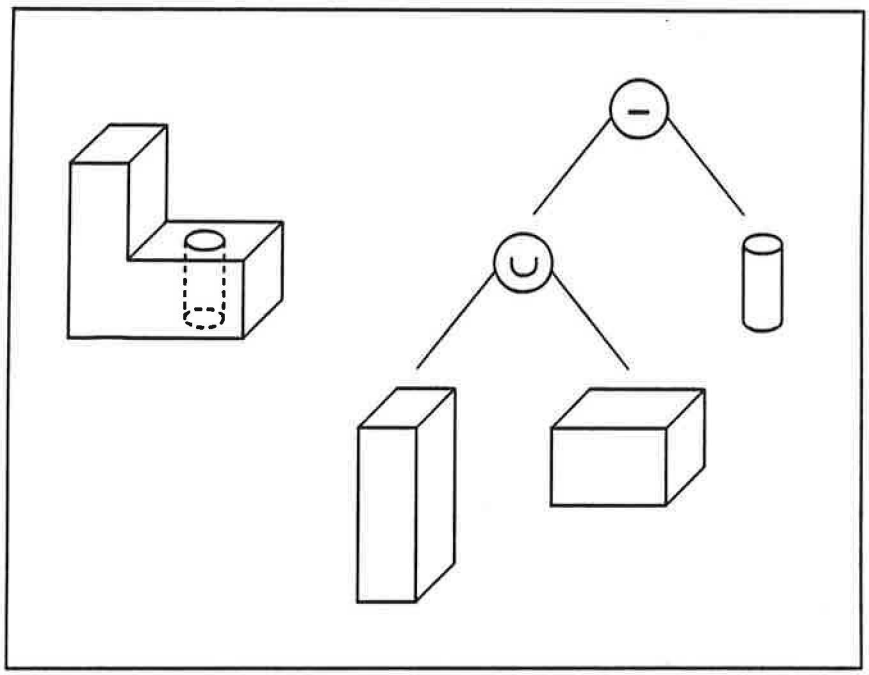

Figure 49 Construction d'un objet complexe

Afin d'accélérer la recherche d'intersections avec les objets de la scène, chaque objet possède une boîte englobante. De plus à partir de ces englobants, une hiérarchie de volumes englobants est générée automatiquement en utilisant les travaux proposés par Goldsmith-Salmon [GS87]. Enfin, OORT propose une grande variété de propriétés qui peuvent être appliquées aux différents objets de la scène.

L'architecture générale du code est présentée en annexe E.

3.4.2.2 Codage des couleurs

OORT utilise le système de codage RVB (ou RGB en anglais) pour représenter la couleur des pixels. Il repose sur trois couleurs fondamentales (Rouge, Vert et Bleu) dont toute couleur

existante est une combinaison linéaire. L'espace des couleurs appartient donc à un cube unité comme indiqué sur la figure suivante :

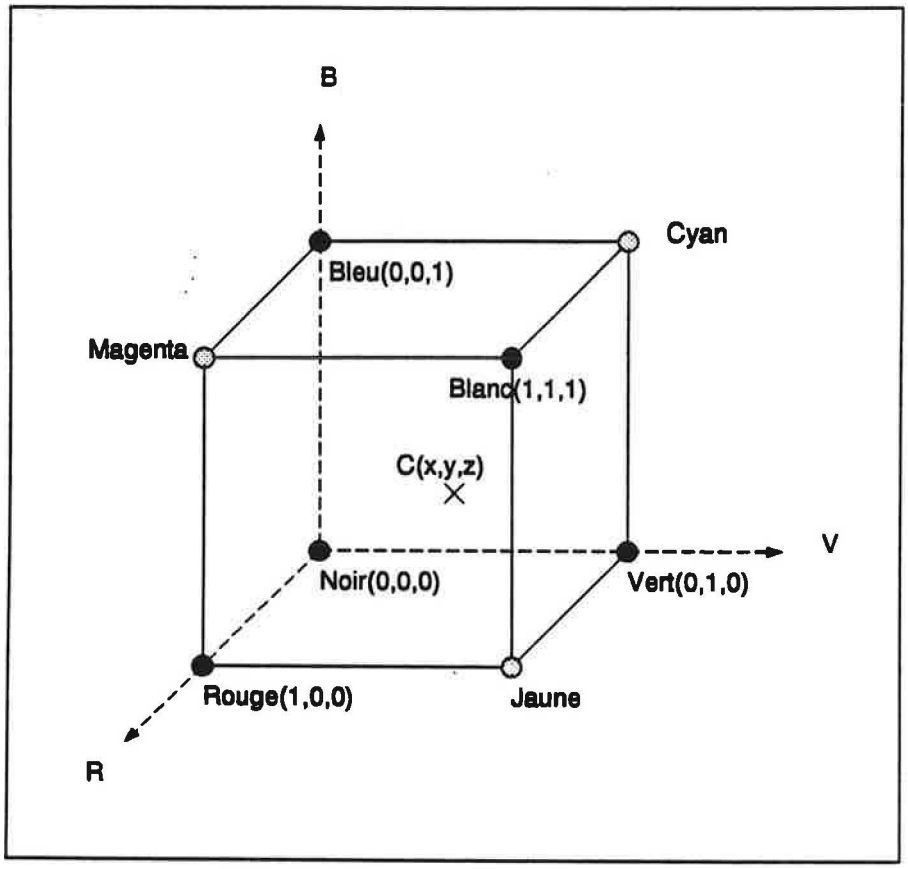

Figure 50 Modèle de couleur RVB

Toute couleur C peut s'exprimer sous la forme d'un triplet *(x,y,z)* tel que :
$C = xR + yV + zB \quad avec \quad (x,y,z) \in [0,1]^3$

Le format de sortie de l'image est de type "ppm" (portable pixmap file format). C'est un format standard lisible par la plupart des afficheurs, où chaque pixel est codé par un triplet d'entier RVB. Les résultats de nos calculs donnant des triplets de nombres réels inférieurs à 1 pour chaque pixel, nous les convertissons au format "ppm" sur un intervalle d'entiers [0,255].

3.4.2.3 Description de la scène

La plupart des logiciels de lancer de rayon disposent d'un langage particulier permettant de décrire la scène à afficher. Cependant la création d'un langage "propriétaire" induit le développement d'un ensemble d'outils (compilateur, debogueur...).

Wilt estime qu'il est plus intéressant de décrire la scène directement en C++. Ce langage permet de faire une description de scène très lisible, dispose de nombreux outils de mise au point et autorise une description de scène beaucoup plus puissante que dans la plupart des langages "propriétaire".

La description d'une scène comporte les éléments suivants :

1. La taille de l'image à représenter (en pixels).
2. La position de l'observateur dans l'espace, *o*.
3. La distance entre l'observateur et l'écran, *d*.
4. La taille de l'écran, *(x,y)*.

5. La position du centre de l'écran dans l'espace, *c*.
6. La direction par laquelle l'observateur regarde le haut de l'écran, *h*.
7. La lumière ambiante
8. La couleur donnée à l'extérieur de la scène.
9. La description des objets de la scène :
    a. Le type d'objet (sphère, parallélépipède...).
    b. Les caractéristiques géométriques (centre, diamètre, largeur...).
    c. Les caractéristiques optiques ou texture :
        - La couleur.
        - La nature de la réflexion (spéculaire ou diffuse).
        - Les indices de réflexion et de réfraction.
10. La description des sources lumineuses :
    a. La position dans l'espace.
    b. La couleur émise.
    c. L'atténuation en fonction de la distance.

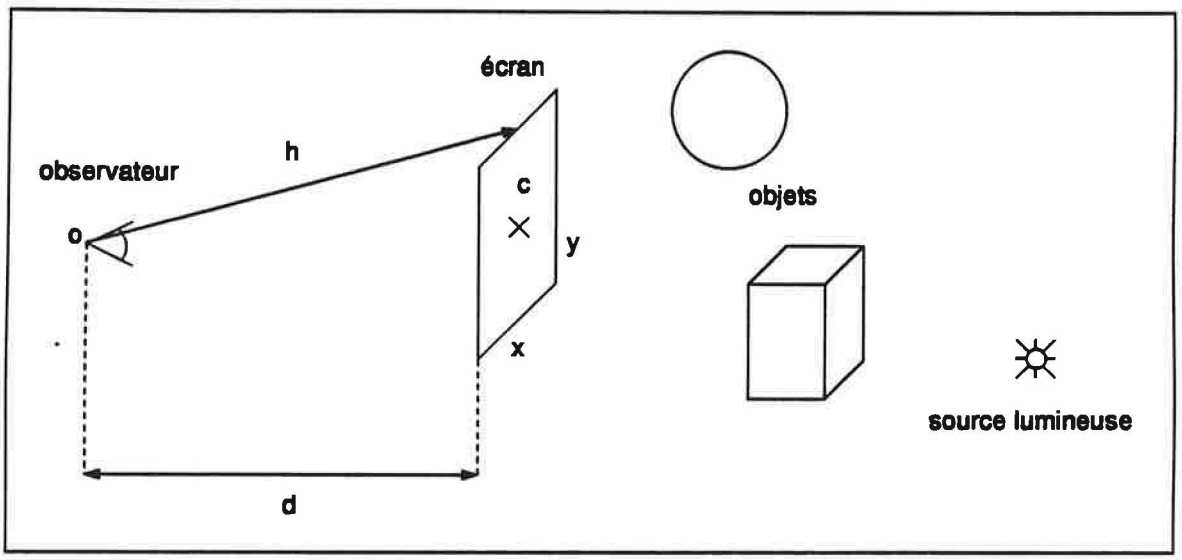

Figure 51 Description d'une scène.

On peut trouver un exemple de description de scène en annexe F.

3.4.3 Spécificités générales dues à la parallélisation

3.4.3.1 Communications asynchrones

Le choix de réaliser un noyau totalement asynchrone pendant sa phase de traitement s'est imposé en raison du coût élevé des communications.

L'envoi de messages par un processeur se fait donc sans accusé de réception, il peut poursuivre ses calculs dès qu'il a terminé cette opération. De plus, un processeur n'est jamais en train d'attendre un message spécifique; en effet, périodiquement il "écoute" si des messages sont arrivés, et dans l'affirmative choisit de les recevoir ou pas.

Ainsi les processeurs sont occupés en permanence (sauf si il n'y a plus de calculs à effectuer), le temps entre une demande et sa réponse est totalement recouvert par des calculs. Le coût d'une communication se réduit donc à celui de la composition du message (empaquetage) et à celui de sa lecture (dépaquetage).

3.4.3.2 Distribution de la base de données

Notre noyau parallèle devant permettre l'augmentation de la taille des problèmes traités par la distribution des données, nous allons définir comment est réalisé ce partage.

Le choix d'une stratégie de partage de la base de données est important pour assurer un bon équilibrage aussi bien en mode flots de données qu'en mode flots de calculs. Mais notre noyau parallèle se voulant indépendant du problème traité, aucune stratégie reposant sur des propriétés de la base de données ne peut être utilisée lors du partage initial.

Aussi nous avons décidé de nous contenter d'un découpage régulier des données sur les processeurs, l'équilibrage étant assuré dynamiquement au cours de l'exécution. Ce partage simple permet à chaque processeur de connaître aisément l'emplacement de chacune des données dans une architecture donnée.

Bien entendu pour qu'un problème puisse être soluble d'un point de vue des besoins mémoire, il faut que la mémoire globale de l'architecture soit suffisante pour contenir la base de données.

1. soit B la taille de la base de données,
2. soit n le nombre de processeurs dans l'architecture,
3. soit m la taille mémoire disponible pour un processeur i.

Il faut donc que :

$$\sum_{i=0}^{n-1} m_i > B \tag{32}$$

Pour l'application du lancer de rayon, la hiérarchie de volumes englobants est dupliquée afin d'accélérer la recherche d'intersections. La surcapacité de nos algorithmes est donc de la forme :

$$Scap_n = \frac{n}{1 + F * (n-1) + C * n}, \; avec \; n \in \mathbb{N}^* \; et \; (F, C) \in]0,1]^2 \tag{33}$$

où F est la fraction de base dupliquée et C la fraction que représente la taille de la mémoire cache par rapport à la mémoire disponible

3.4.3.3 Distribution des calculs

La parallélisation d'un programme demande une distribution des calculs sur les processeurs disponibles. On souhaite que ce partage des tâches soit équilibré afin que chaque processeur ait un temps de calcul identique. Par contre, il est également souhaitable de profiter d'une éventuelle cohérence entre les calculs qui permet de diminuer le temps nécessaire à la réalisation de la tâche impartie.

En raison de l'équilibrage dynamique qui est mis en place, il semble logique de privilégier l'aspect cohérence des calculs, en réalisant un partage en blocs qui est le plus adapté à l'algorithme pour le calcul d'images (voir §2.2.5.3). Toutefois, en raison de l'aspect générique de notre projet, nous ne voulons profiter d'aucune cohérence spécifique à un problème donné; aussi, nous ne recherchons qu'à réaliser la distribution la plus équilibrée possible, c'est à dire en implémentant un partage par point.

3.4.3.4 Stockage des résultats

Généralement, lors de la parallélisation d'un code, chaque processeur a en charge un certain nombre de tâches dont les résultats sont envoyés, au fur et à mesure de leur obtention, dans un fichier, à une machine hôte ou à un éventuel processeur maître. Cependant une de nos architectures cible est le CRAY T3D, dont la gestion des fichiers est réalisée par une machine hôte (CRAY Y-MP) vers laquelle les entrées-sorties sont réalisées au travers d'un unique lien qui peut devenir un goulot d'étranglement. Aussi un processeur ne peut envoyer ses résultats à l'hôte ou dans un fichier qu'au risque d'un coût prohibitif.

Comme notre architecture ne dispose pas de processeur maître, une solution est le stockage des résultats de chaque processeur dans sa propre mémoire. La phase de mise en commun des résultats est réalisée une fois toutes les tâches accomplies.

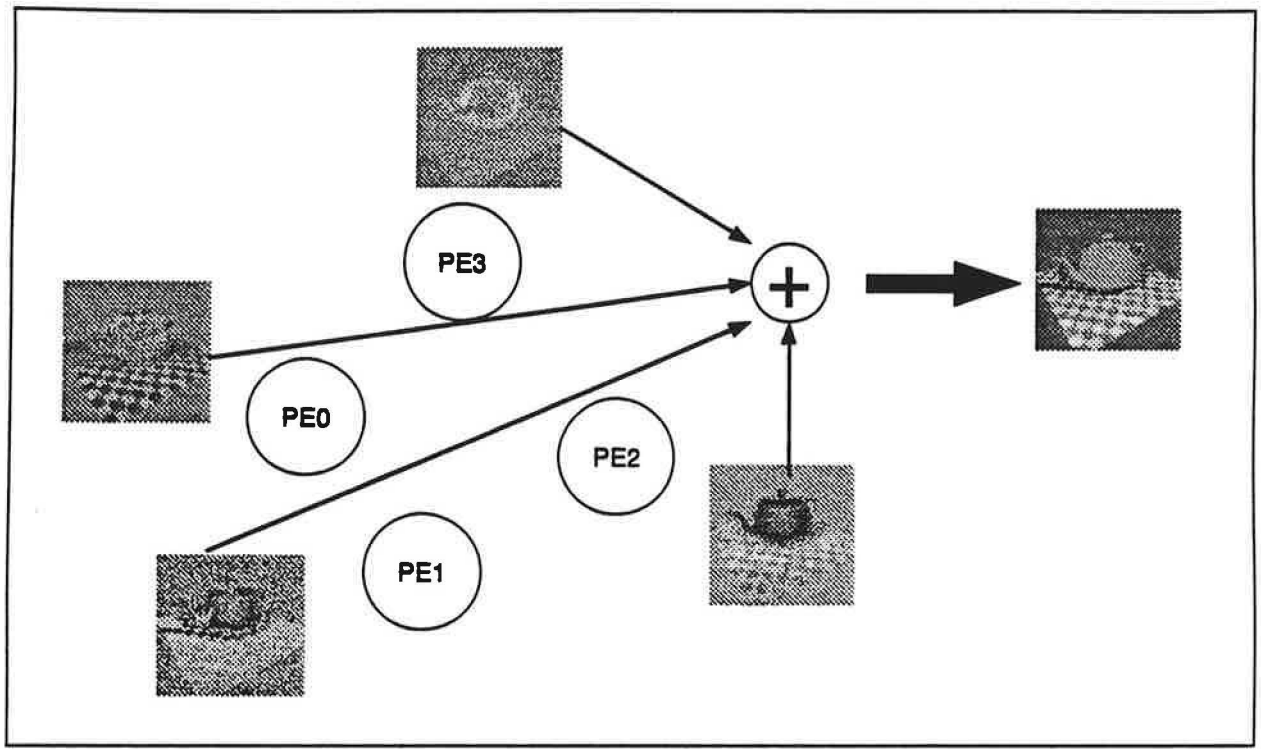

Figure 52 Mise en commun des images résultats

Certes le surcoût mémoire de cette solution peut être important puisque chaque processeur doit posséder un tableau pouvant contenir la totalité de l'image :

$$\begin{aligned} Besoinsmemoire &= Tailledel'image \times Tailled'untripletRGB \\ &= Tailledel'image \times (Tailled'un\ entier \times 3) \end{aligned}$$

Par exemple, si l'image à calculer doit comporter 512 x 512 pixels, et si la taille nécessaire pour représenter un nombre entier est de 8 octets, les besoins mémoire pour stocker l'image sont de 6 Mo.

C'est à ce prix que nous pouvons assurer l'obtention de résultats performants en temps sur l'ensemble des machines dont nous disposons. Une solution permettant d'alléger le coût mémoire de l'image serait de la subdiviser, ainsi on effectuerait plusieurs fois le calcul de plus petites images.

On peut noter toutefois que notre contrainte est très particulière à l'architecture dont nous disposons et qu'il serait facile de réaliser un noyau parallèle, qui ne stocke pas ses résultats en mémoire locale. Les rayons calculés seraient alors envoyés régulièrement par chaque processeur à une machine hôte ou dans un fichier.

Récemment le CRAY T3D de notre centre de calcul a été remplacé par un CRAY T3E qui assure lui même sa gestion de fichiers. La contrainte sur le stockage des résultats est donc levée.

3.4.4 Spécificités dues à l'aspect flots de données

L'algorithme à flots de données induit une certaine organisation de la mémoire. Le choix de communications asynchrones impose une gestion des demandes de données. Enfin, l'utilisation de machines massivement parallèles nous contraint à définir un équilibrage dynamique non centralisé.

3.4.4.1 Organisation de la mémoire

Quand un rayon est amené à rencontrer un objet qui n'est pas présent dans la mémoire de son processeur, une copie de cet objet doit être demandée au processeur qui le possède. Cette copie sera placée dans la mémoire cache du processeur.

La mémoire servant à contenir les objets de la base de données est donc divisée en 2 parties :

1. Une mémoire permanente dans laquelle chaque processeur possède $\frac{1}{n}$ des objets de telle sorte que l'ensemble des objets contenus dans ces mémoires locales corresponde à exactement la description de la scène $\left(\sum_{i=0}^{n-1} \frac{1}{n} = 1\right)$.
2. Une mémoire cache comprenant des copies temporaires d'objets possédés dans la mémoire résidente d'autres processeurs.

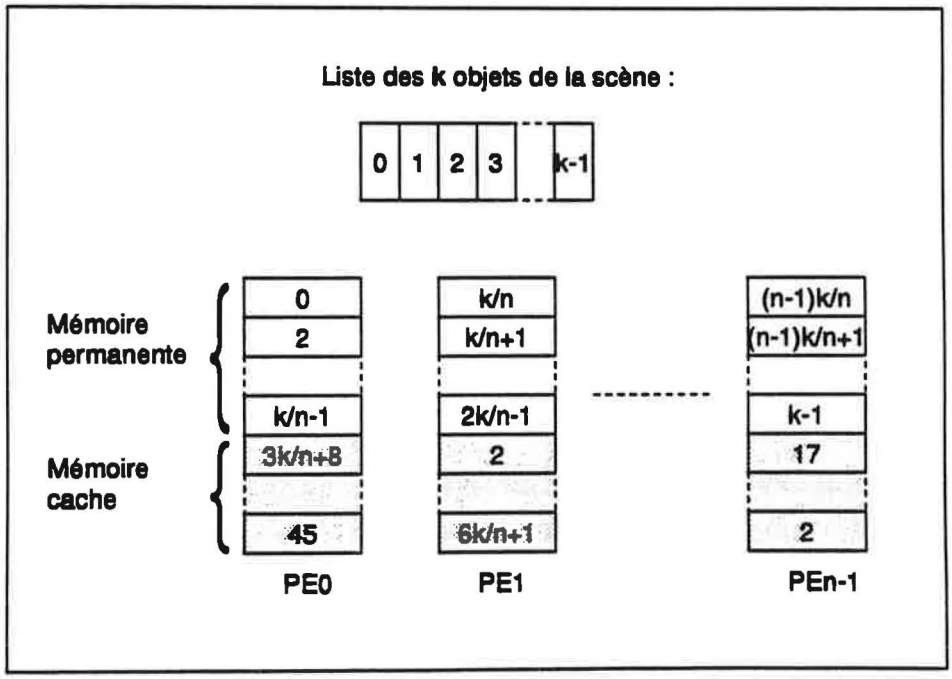

Figure 53 Organisation de la mémoire

La gestion de cette mémoire cache suit une méthode où les objets qui n'ont pas été utilisés depuis la plus longue période sont les objets susceptibles d'être supprimés en premier (méthode

LRU). Il faut toutefois noter que les objets qui ont été demandés explicitement par des rayons ne pourront disparaître avant leur utilisation par les rayons demandeurs.

3.4.4.2 Gestion des demandes de données

Dans un algorithme classique à flots de données, les communications se font de manière synchrone. Ainsi quand un calcul nécessite une donnée, il y a recherche de la donnée sur le processeur la possédant. Une fois que celle-ci a été transférée sur le processeur demandeur, alors le calcul se poursuit.

Puisque nous avons choisi un mode de communication asynchrone, cela nous permet de pouvoir bufferiser les demandes et ainsi d'envoyer les requêtes au processeur concerné uniquement quand elles sont demandées par *suffisamment* de calculs, ce critère sera déterminé expérimentalement. On peut ainsi limiter le nombre de messages de demandes de données circulant sur le réseau de processeurs en fixant une taille limite pour le buffer contenant les demandes de données.

Chaque fois qu'un calcul doit être réalisé, on teste si la donnée nécessaire est présente dans la mémoire du processeur. Dans l'affirmative, on effectue le calcul; sinon, on traite la demande de donnée et on passe au calcul suivant. Une demande de donnée peut être mise dans un buffer, envoyée si le buffer est plein ou alors ne pas être prise en compte s'il n'y a plus de buffers libres. Dans ce dernier cas, quand on cherchera à effectuer de nouveau ce même calcul, sa demande de données sera réexaminée. Pour effectuer une demande de donnée, des données sont supprimées de la mémoire cache si besoin est, afin que la donnée demandée puisse être reçue, et un message est adressé au processeur possédant la donnée recherchée. Ce processeur, dès réception du message, enverra une copie de la donnée au processeur demandeur.

3.4.5 Spécificités dues à l'aspect flots de calculs

Quand un calcul d'intersection entre un rayon et un objet est impossible, car l'objet est absent de la mémoire du processeur, le rayon doit être envoyé sur le processeur possédant l'objet manquant. Le calcul d'un pixel pouvant être réalisé sur plusieurs processeurs, l'utilisation d'un algorithme récursif est à proscrire comme on le détaillera plus loin. Ceci va entraîner la nécessité d'une gestion rigoureuse du nombre de rayons présents sur chaque processeur afin qu'il n'y ait pas de dépassement de la taille mémoire. L'envoi de rayons est bufferisé afin de limiter le nombre de communications. Toutefois, le nombre de messages pouvant transiter simultanément sur un réseau étant également une ressource critique, il nous faudra limiter strictement le nombre de communications.

Enfin, nous allons présenter un algorithme de terminaison du code, celle-ci étant difficile à détecter en flots de calculs.

Les équilibrages de charge dynamiques reposent sur l'utilisation de processus concurrents. Ceci ne pouvant être réalisé sur l'ensemble de nos machines "cible", aucun mécanisme d'équilibrage ne sera implémenté lors de la réalisation de cet algorithme à flots de calculs. Celui-ci n'étant qu'une étape dans la conception de nos stratégies à flots mixtes qui utilisent d'autres formes d'équilibrage, cette absence d'équilibrage pour l'algorithme à flots de calculs ne sera pas pénalisante pour le résultat final.

3.4.5.1 Abandon de la récursivité de l'algorithme

La figure suivante présente les premiers niveaux de l'arbre d'intersection permettant le calcul de l'intensité d'un pixel de l'écran I_0. Afin de ne pas surcharger le dessin, nous nous sommes limités à une scène ne comportant que 3 sources lumineuses.

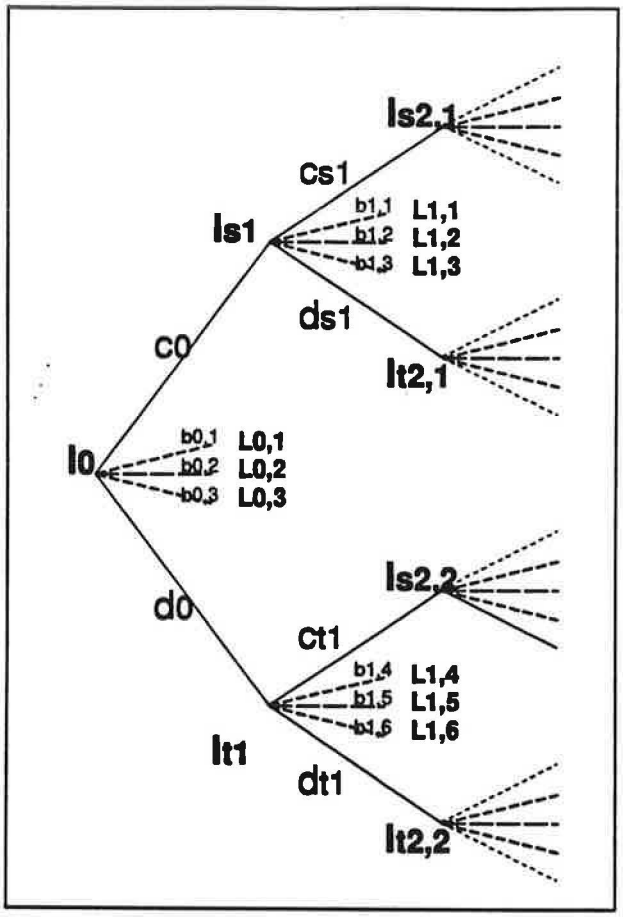

Figure 54 Arbre des rayons issus d'un pixel

L'expression mathématique permettant de calculer la couleur du pixel I_0 est de la forme suivante :

$$I_0 = a + \sum_{j=1}^{l} b_{0,j} L_{0,j} + c_0 Is_1 + d_0 It_1 \tag{35}$$

Avec

a une constante représentant l'intensité ambiante
l le nombre de sources lumineuses (ici l=3)
$b_{n,j}$ des constantes dépendantes des objets rencontrés et des sources lumineuses
$L_{n,j}$ les intensités reçues à la profondeur n de la jème source lumineuse
c_0, cs_n, ct_n, d_0, ds_n et dt_n des constantes dépendantes des objets rencontrés
Is_n et $Is_{n,k}$ les intensités reçues à la profondeur n dans la direction de réflexion
It_n et $It_{n,k}$ les intensités reçues à la profondeur n dans la direction de réfraction

Exprimons de la même manière Is_1 et It_1 :

$$Is_1 = a + \sum_{j=1}^{3} b_{1,j} L_{1,j} + c_{s1} Is_{2,1} + d_{s1} It_{2,1} \tag{36}$$

$$It_1 = a + \sum_{j=4}^{6} b_{1,j} L_{1,j} + c_{t1} Is_{2,2} + d_{t1} It_{2,2} \tag{37}$$

La méthode la plus naturelle pour calculer l'intensité d'un pixel est l'utilisation d'une fonction récursive. Il y a alors construction de l'arbre du rayon dont la somme des valeurs aux noeuds permet d'obtenir la couleur du pixel désiré. Cette algorithme classique, récursif, provoque un parcours en profondeur de l'arbre : Is_1 ne peut être calculé qu'après l'évaluation de $Is_{2,1}$ dans un premier temps et de $It_{2,1}$ dans un deuxième temps. La parallélisation d'un tel algorithme aboutit à ce que seuls des rayons provenant de pixels différents peuvent être calculés en parallèle. De plus, au calcul de chaque pixel est associée une pile de rayons secondaires, ce qui rend coûteux le transfert d'un calcul de rayon d'un processeur à un autre.

Dans l'algorithme à flots de calculs, plusieurs processeurs peuvent être amenés à calculer la couleur d'un même pixel, ce qui nécessite l'envoi de rayons entre les processeurs. L'utilisation d'un mode de calcul récursif n'est donc pas du tout adaptée. C'est pourquoi nous allons chercher à abandonner cette récursivité.

Remplaçons dans l'équation (35) Is_1 et It_1 par leurs expressions (36) et (37) :

$$I_0 = a + \sum_{j=1}^{3} b_{0,j} L_{0,j} + c_0 \left(a + \sum_{j=1}^{3} b_{1,j} L_{1,j} + c_{s1} Is_{2,1} + d_{s1} It_{2,1} \right) + d_0 \left(a + \sum_{j=4}^{6} b_{1,j} L_{1,j} + c_{t1} Is_{2,2} + d_{t1} It_{2,2} \right) \quad (38)$$

En développant l'équation (38) et en regroupant les constantes, on obtient l'expression suivante de I_0 :

$$I_0 = a(1 + c_0 + d_0) + \sum_{j=1}^{3} (b_{0,j} L_{0,j} + c_0 b_{1,j} L_{1,j} + d_0 b_{1,j+3} L_{1,j+3}) + c_0 c_{s1} Is_{2,1} + c_0 d_{s1} It_{2,1} + d_0 c_{t1} Is_{2,2} + d_0 d_{t1} It_{2,2} \quad (39)$$

En posant

$A = a(1+c_0+d_0)$
$B_{1,j} = c_0 b_{1,j}$
$B_{2,j} = d_0 b_{1,j}$
$Cs_{2,1} = c_0 cs_1$
$Ct_{2,1} = c_0 ds_1$
$Cs_{2,2} = d_0 ct_1$
$Ct_{2,2} = d_0 dt_1$

L'équation (39) devient :

$$I_0 = A + \sum_{j=1}^{3} (b_{0,j} L_{0,j} + B_{1,j} L_{1,j} + B_{2,j+3} L_{1,j+3}) + Cs_{2,1} Is_{2,1} + Ct_{2,1} It_{2,1} + Cs_{2,2} Is_{2,2} + Ct_{2,2} It_{2,2} \quad (40)$$

Il apparaît clairement dans l'équation (40) que les calculs des rayons secondaires et des rayons d'ombre peuvent être réalisés indépendamment, I_0 étant la somme de l'ensemble de ces rayons. Ainsi, à chaque création d'un rayon secondaire ou d'ombrage, en affectant la constante d'atténuation du rayon père multiplié par celles précédemment rencontrées dans l'arbre et le nom du pixel d'origine, les rayons peuvent être calculés séparément. Leur contribution est ajoutée à la couleur du pixel d'origine une fois leur calcul terminé.

Par exemple, le rayon Is_1 va générer un rayon $Is_{2,1}$ comprenant les deux champs suivants :

1. Pixel d'origine : I_0.
2. Atténuation : $c_0 cs_1$

Dans notre algorithme, nous utilisons donc cette méthode de calcul non récursive, qui permet un envoi de rayons "légers" entre les processeurs et le calcul en parallèle de rayons issus d'un même pixel. Toutefois, ceci est au prix d'un surcoût mémoire puisque à chaque intersection est généré l'ensemble des rayons secondaires et d'ombrage munis de leur coefficient d'atténuation et du nom du pixel d'origine. Afin que cette génération de rayons ne provoque pas de dépassement

mémoire, il va falloir mettre au point un mécanisme permettant de limiter le nombre de rayons présents à un instant donné sur un processeur donné. Ceci est présenté dans le paragraphe suivant.

3.4.5.2 Gestion du nombre de rayons

Avant de présenter la gestion du nombre de rayons pour notre algorithme parallèle, nous allons commencer par la traiter dans le cas de l'algorithme séquentiel.

Cas séquentiel

En raison de l'abandon de la récursivité de l'algorithme, le nombre de rayons présents à un moment sur un processeur peut augmenter de façon incontrôlée, ce qui peut conduire à des risques de dépassement de la taille de la mémoire disponible. Il nous faut donc gérer cette ressource critique.

Tout d'abord, nous allons chercher le nombre de rayons que peut engendrer au maximum un rayon primaire pour une profondeur limite donnée p, pour une scène comportant l sources lumineuses. A l'étape p, un rayon cesse de générer des rayons secondaires puisque l'on considère que ceux-ci ont une contribution négligeable.

Un rayon primaire ou secondaire va donner naissance à une étape e inférieure à p au maximum à $2+l$ rayons :

1. un rayon réfléchi
2. un rayon réfracté
3. l rayons d'ombre

Par contre, les rayons d'ombre ne donnent naissance à aucun autre rayon.

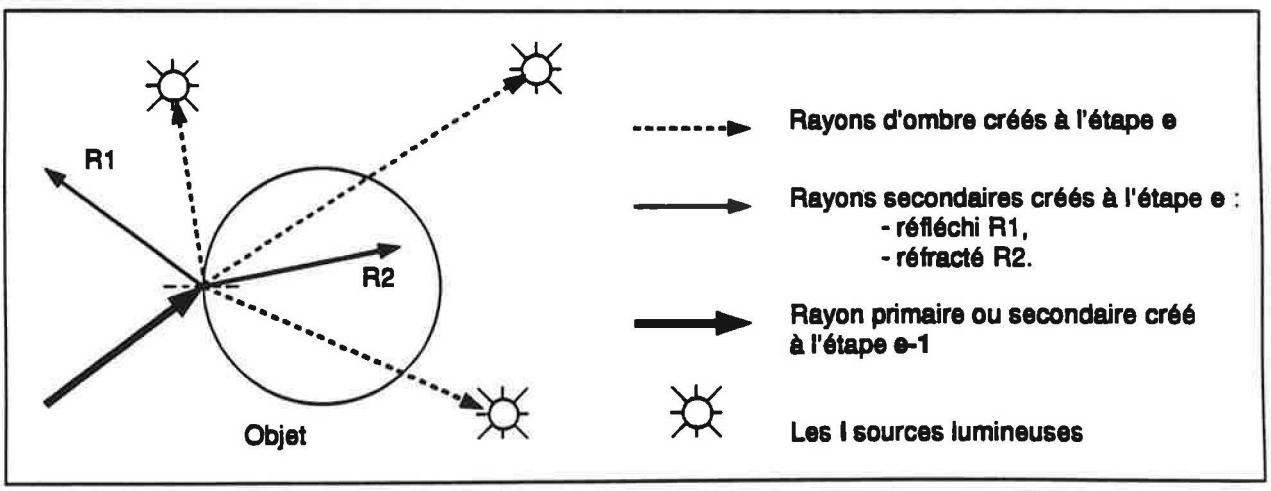

Figure 55 Création de rayons à chaque étape

A la dernière étape $e=p$, chaque rayon primaire ou secondaire donne naissance à l rayons d'ombre, mais ne va pas générer de rayons secondaires.

Etudions l'évolution du nombre de rayons n au cours des p étapes :

$$\begin{aligned}
Etape\ 0\ :\ & n_0 = 1 \\
Etape\ 1\ :\ & n_1 = 2 + l \\
Etape\ e < p\ :\ & n_e = 2(n_{e-1}) + l \\
& n_e = 2^e + l\sum_{i=0}^{e-1} 2^i \\
& n_e = 2^e + l\left(2^{e+1} - 1\right) \\
Etape\ p-1\ :\ & n_{p-1} = 2^{p-1} + l\left(2^p - 1\right) \\
Etape\ p\ :\ & n_p = l\left(2^{p+1} - 1\right)
\end{aligned} \tag{41}$$

Si on suppose qu'il n'est lancé qu'un seul rayon par pixel, chaque pixel peut entraîner la génération de $l(2^{p+1} - 1)$ rayons au maximum.

Ainsi si on fixe une taille limite pour la pile des rayons $MAX.TRACERS$, à l'initialisation de la pile on ne peut y mettre que $\frac{MAX.TRACERS}{l(2^{p+1}-1)}$ rayons primaires. Si d'autres rayons primaires doivent être calculés, il faudra attendre que les premiers rayons soient calculés pour pouvoir les incorporer dans la pile.

Toutefois, nous pouvons remarquer que la pile n'augmente que si un rayon primaire ou secondaire est en cours de traitement. Par contre, le traitement d'un rayon d'ombre permet de diminuer la taille de la pile. En effet le calcul d'un rayon d'ombre libère une place dans la pile et aucun autre rayon n'est généré.

Aussi, une condition suffisante pour que le nombre de rayons ne soit pas supérieur à la taille de la pile est qu'elle ne puisse pas être pleine uniquement de rayons primaires ou secondaires. En effet si elle est pleine, il est possible de la vider en calculant les rayons d'ombre. Il faut donc assurer une taille minimale de 2^{p-1} places par rayon primaire introduit.

On va donc exprimer le potentiel de création de rayons secondaires d'un rayon primaire ou secondaire : $2^{p-1-depth_i}$ où $depth_i$ est la profondeur à laquelle le rayon a été créé.

On peut donc introduire initialement $\frac{MAX.TRACERS}{2^{p-1}}$ rayons primaires. Le potentiel de l'ensemble étant de $MAX.TRACERS$. Chaque fois qu'un rayon quitte la scène, le potentiel est diminué de $2^{p-1-depth_i}$. Il en est de même si un des rayons secondaires n'est pas généré. Enfin, quand un rayon atteint la profondeur maximale, le potentiel est réduit de 1.

Ainsi tant qu'un processeur possède un potentiel inférieur à $MAX.TRACERS$, il est susceptible de pouvoir augmenter son nombre de rayons primaires ou secondaires.

Cas parallèle

Dans le cas de l'utilisation de l'algorithme sur une machine parallèle, tout ce qui a été vu précédemment reste vrai. Mais il faut compléter cette analyse par la prise en compte de la réception et de l'envoi de rayons.

L'envoi de rayons entraîne pour le processeur émetteur une diminution de son potentiel de $2^{p-1-depth_i}$ par rayon et l'augmentation du potentiel du processeur receveur de la même valeur. Toutefois, il faut s'assurer que, quand un rayon est envoyé, il pourra être incorporé à la pile du processeur le recevant sans qu'il y ait dépassement de son potentiel. Pour ce faire, les rayons qui sont reçus par un processeur sont mis dans une première pile qui sera utilisée pour alimenter la pile

principale en rayons quand celle-ci sera à cours de rayons. Cependant, si cette pile de réception de rayons ne possède pas suffisamment de place pour recevoir DOSE_MAX, qui est le nombre de rayons maximum qu'un processeur peut envoyer, alors le message ne pourra pas être reçu. Afin d'éviter que des messages comprenant des rayons à calculer ne soient perdus, il nous faut réguler l'envoi de rayons.

Tout d'abord, à l'initialisation de l'algorithme, nous allouons deux listes de longueur fixe à chaque processeur (MAX_TRACERS et MAX_TRACERS_RECUS) qui contiendront respectivement l'ensemble des rayons à calculer et les rayons reçus d'autres processeurs qui n'ont pas encore été incorporés faute de place dans la liste principale.

Ne disposant pas de mécanisme centralisateur, nous imposons à chaque processeur de n'envoyer vers un processeur donné qu'un nombre limité de rayons - qui correspond à plusieurs envois de DOSE_MAX - sans qu'il n'y ait eu confirmation des réceptions. Le processeur destinataire ne recevra des messages comprenant des rayons que s'il reste de la place dans la pile MAX_TRACERS_RECUS, sinon la réception se fera ultérieurement, le message reste en attente. Une fois les rayons reçus, le processeur mémorise le nombre de rayons reçus et le numéro du processeur émetteur. Lors d'un envoi de messages vers ce processeur-ci, il y aura envoi également de cette information. Ce processeur pourra alors remettre à jour le nombre de rayons qu'il peut envoyer.

3.4.5.3 Terminaison de l'algorithme

Contrairement à l'algorithme à flots de données, où la terminaison du programme est triviale (en effet chaque processeur s'arrête quand il a fini les calculs dont il est responsable ou quand il a reçu et traité les calculs qu'il a demandés), la terminaison de l'algorithme à flots de calculs est beaucoup plus délicate.

Si on ne fait pas l'hypothèse d'un temps de communication nul entre deux processeurs, il est particulièrement difficile d'implémenter une terminaison de l'algorithme efficace et assurant la fin effective de tous les calculs sur tous les processeurs quel que soit le cas de figure. Ainsi Priol [Pri89] fait cette hypothèse et utilise l'algorithme du jeton proposé par Dijkstra et al. [DFG83], tout en mettant bien en évidence les risques que cela représente.

D'autre part, on peut remarquer que si quelques calculs ne sont pas effectués, cela peut n'avoir qu'une incidence minime sur le résultat final. Par exemple, pour l'application du lancer de rayon, les rayons qui risquent de ne pas être calculés sont généralement des rayons de profondeur importante. Leur contribution étant faible pour le calcul des pixels, leur absence peut ne pas être gênante. Par contre pour d'autres applications, l'absence de certains calculs peut conduire à des résultats erronés.

Ce que nous proposons est également l'utilisation d'une forme d'algorithme à jeton parcourant un anneau afin d'assurer une terminaison rapide. Toutefois, nous souhaitons signaler à l'utilisateur le nombre éventuel de calculs non effectués afin que celui-ci puisse le cas échéant intervenir en relançant l'application avec une finesse de détection de terminaison plus grande et ainsi obtenir un résultat final complet.

Aussi, nous allons proposer une terminaison efficace et signalant le cas échéant le nombre de calculs qui n'ont pas été pris en compte pour le résultat final.

Comme dans l'algorithme à flots de données, tout échange de message comprend une information sur l'état de charge du processeur émetteur. De plus, quand un processeur a terminé tous ses calculs, il en informe le *processeur 0*. Lorsque le *processeur 0* n'a plus de calculs à faire et que tous les autres processeurs lui ont signalé qu'ils étaient dans ce même état, l'algorithme semble terminé. Cependant celui-ci étant à flots de calculs, un processeur peut signaler au *processeur 0* que

sa charge est nulle et recevoir quelques instants après de nouveaux calculs à effectuer. Aussi pour s'assurer que l'algorithme est bien terminé, le *processeur 0* lance une procédure de terminaison.

Le *processeur 0* émet un jeton blanc qui va circuler de processeur en processeur suivant une topologie en anneau. Le processeur qui reçoit un jeton a deux possibilités :

1. L'envoyer au processeur suivant s'il a effectivement terminé ses calculs.
2. Retourner un jeton noir au processeur 0 pour lui signaler qu'il a de nouveau des calculs à effectuer.

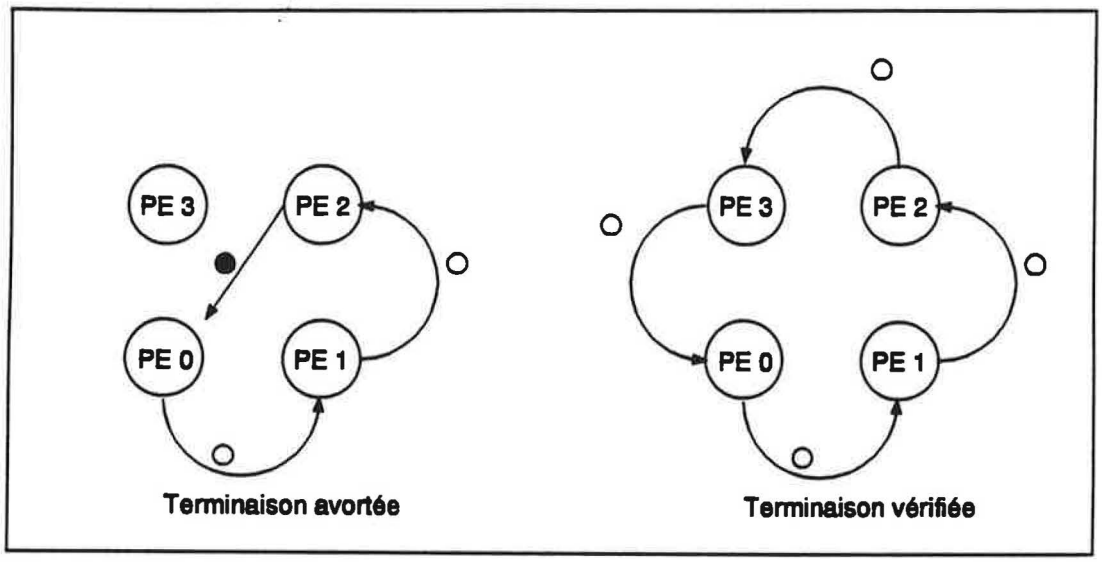

Figure 56 Algorithme de terminaison

Pour le processeur 0, il y a trois cas possibles :

1. Il reçoit un jeton noir. La procédure de terminaison est alors annulée. Elle ne reprendra que quand le processeur 0 aura l'information que tous les processeurs ont terminé leurs calculs.
2. Il reçoit un jeton blanc.
   a. Si le processeur 0 n'a toujours pas de calculs à effectuer, la terminaison du programme est assurée.
   b. Si le processeur 0 possède de nouveaux calculs, la procédure de terminaison est alors annulée.

Cette procédure de terminaison est relativement efficace car un jeton ne circule que quand tous les processeurs ont potentiellement fini leurs calculs. Toutefois, il est possible dans de rares cas qu'un processeur émette un jeton blanc juste avant de recevoir de nouveaux calculs. Aussi pour détecter cela, en phase de post-traitement les processeurs garderont la possibilité de recevoir d'éventuels calculs. Bien sûr, ils ne pourront pas être traités, mais leur présence et leur nombre seront signalés à l'utilisateur.

Si celui-ci estime que leur absence nuit au résultat final, il aura la possibilité de relancer l'application avec un paramètre demandant un processus de détection de terminaison beaucoup plus fin. C'est à dire qu'au lieu de faire circuler un jeton blanc une seule fois sur l'anneau reliant tous les processeurs, l'utilisateur pourra préciser le nombre de parcours d'anneaux souhaités pour valider la terminaison de l'algorithme.

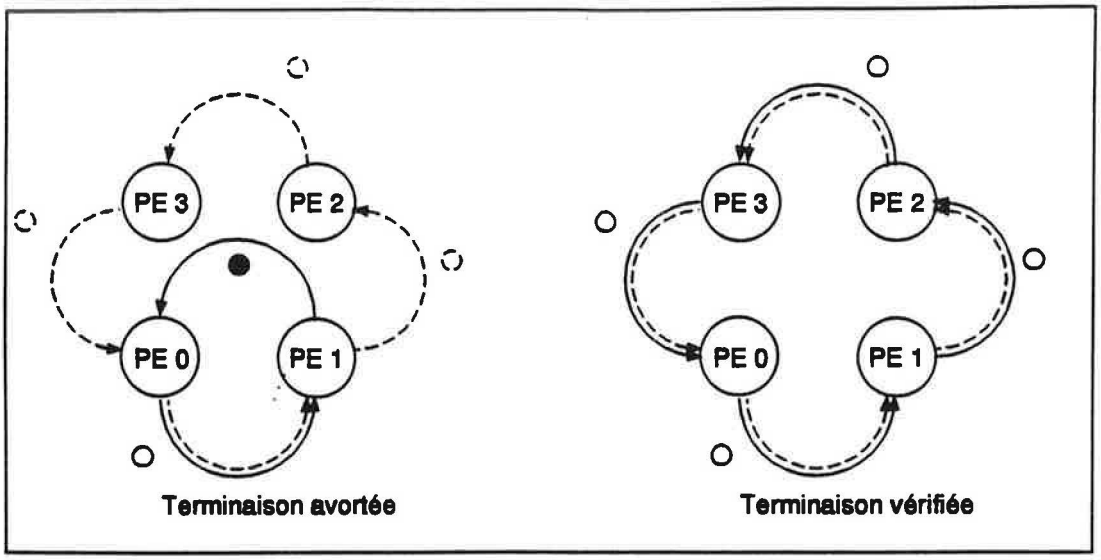

Figure 57 Trajets du jeton sur plusieurs parcours

Pour certaines applications, la relance de l'exécution peut être partielle. Pour le lancer de rayon, si des rayons ne sont pas calculés, il est possible de ne demander que le recalcul des pixels dont ils sont issus.

Expérimentalement, il apparaît que pour notre application, avec une finesse de deux parcours, moins de 0,1 % des exécutions comportent une absence de calculs.

3.4.5.4 Gestion du nombre de messages

Le fait d'avoir choisi d'implémenter un noyau parallèle asynchrone induit la nécessité de limiter le nombre de messages envoyés sur la machine parallèle. En particulier l'utilisation de PVM sur CRAY impose de fixer un nombre maximum de messages pouvant être envoyés par un processeur sur cette machine simultanément (*PVM_SM_POOL*, annexe G). De toute façon, même si la limitation du nombre de messages n'est pas aussi stricte sur réseau de stations ou sur d'autres machines, il est important de pouvoir gérer ce nombre et ainsi éviter une saturation pouvant bloquer le réseau ou dégrader très fortement les performances. Pour ce faire, nous allons analyser les différentes caractéristiques des messages employés dans notre noyau parallèle pendant la phase de calcul. On retrouvera la liste des messages utilisés en annexe D.

Les messages envoyés peuvent être classés en plusieurs catégories :

1. Les messages prioritaires et uniques : leur circulation doit toujours être possible pour le bon fonctionnement de l'algorithme et chacun d'eux ne peut être envoyé plus d'une fois vers un processeur donné sans qu'il y ait eu confirmation de leur réception. Ils sont de quatre types :
   a. Envoi par un processeur à un autre d'informations sur son état (*C_INFORMATIF*).
   b. Demande de calculs à un processeur susceptible d'en fournir (*C_DEMANDE_CALCULS*).
   c. Recherche si tous les processeurs ont bien fini d'effectuer leurs calculs (*C_TERMINAISON*).
   d. Annonce par le processeur *0* de la fin de la phase de calcul (*C_END*).
      Ce message ne sera pas comptabilisé par la suite pour la gestion du nombre de messages circulant à un instant donné. Il est envoyé uniquement quand l'activité de la machine est nulle.

2. Les messages uniques non prioritaires : leur envoi peut-être différé dans le temps.
   a. Envoi des calculs à un processeur qui les a demandés (*C_FOURNITURE_CALCULS_DEMANDES*).
   b. Envoi d'un message signalant l'impossibilité de fournir des calculs à un processeur qui en a demandés (*C_FOURNITURE_CALCULS_IMPOSSIBLE*).
3. Les messages multiples dont le nombre est fixé à une valeur n : le nombre de messages "simultanés" de demande de données est limité par l'utilisateur.
   a. Demande de donnée (*C_DEMANDE_DONNEE*), cette demande est "prioritaire" pour notre algorithme.
   b. Envoi d'une donnée demandée (*C_ENVOI_DONNEE* ou *C_ENVOI_MIXTE*), cette envoi peut-être différé.
4. Les messages multiples non prioritaires : ce sont les messages d'envoi de flots de calculs (*C_FLOT_CALCULS*).

A partir de ce classement, nous allons proposer une stratégie permettant de s'assurer qu'un processeur ne dépasse pas l'envoi de *PVM.SM.POOL* messages sur la machine. Une condition suffisante est qu'un processeur n'envoie pas plus de $\frac{PVM.SM.POOL}{(p-1)}$ messages vers chacun des *p-1* autres processeurs.

Calculons à l'aide de notre classification une borne supérieure au nombre de messages prioritaires que peut envoyer "simultanément" sur le réseau d'interconnexion un processeur donné à un autre :

1. Le nombre de messages prioritaires et uniques : *3*.
2. Le nombre de messages prioritaires et multiples : n.

Chaque processeur peut donc envoyer au moins $\frac{PVM.SM.POOL}{(p-1)} - 3 - n$ messages non prioritaires à chacun des autres processeurs du réseau.

Aussi nous proposons que chaque processeur stocke les informations suivantes :

- Le nombre de messages non prioritaires en cours d'envoi vers chacun des autres processeurs.
- Le nombre de messages non prioritaires reçus de chacun des autres processeurs.

Ces informations sont mises à jour après chaque envoi ou réception d'un message.

Ainsi, avant qu'un processeur envoie un message non prioritaire vers un processeur donné, il faut qu'il vérifie que le nombre de messages en envoi vers ce processeur n'a pas atteint $\frac{PVM.SM.POOL}{(p-1)} - 3 - n$. Sinon l'envoi du message est annulé ou différé.

De plus, chaque fois qu'un message est envoyé, il est précisé dans son entête le nombre de messages reçus provenant du processeur destinataire. Le processeur émetteur mettra alors à zéro cette variable. Le processeur destinataire, lui, réactualisera le nombre de messages et de rayons en cours d'envoi.

3.5 Conclusion

Dans ce chapitre, nous avons présenté les différentes exigences inhérentes à notre projet et comparé les algorithmes y répondant grâce à leur modélisation. Finalement, la particularité de notre problème, fortement influencé par des contraintes techniques, une architecture indéterminée et une approche massivement parallèle, nous conduit à proposer des stratégies originales **à flots mixtes**. Ces algorithmes reposent sur l'emploi de deux types de flots mis en concurrence. Leur mise en oeuvre a également été développée.

Dans le chapitre suivant, sont exposés les résultats obtenus grâce à nos algorithmes. Leur analyse permettra de vérifier leur bien-fondé.

Chapitre 4
Résultats

Avant de présenter les performances de nos algorithmes à flots mixtes, nous commençons par décrire nos conditions expérimentales. Ensuite nous nous intéressons aux résultats des algorithmes à flots classiques, notre noyau parallèle permettant l'exécution dans ces modes. Puis nous exposons les performances assurées par nos stratégies mixtes. Enfin nous montrons les avantages que procure la réalisation d'algorithmes de parallélisation utilisant des communications asynchrones, ainsi que quelques résultats obtenus sur des machines parallèles hétérogènes.

4.1 Conditions expérimentales

Le noyau parallèle que nous proposons comprend de nombreux paramètres. Certains définissent les caractéristiques de l'image à représenter. D'autres permettent de configurer le code par le choix du mode dans lequel l'exécution sera effectuée (algorithme classique, à flots mixtes...). Enfin les derniers servent à optimiser ces algorithmes.

Nous commençons donc par présenter ces paramètres. Puis, nous spécifions l'environnement utilisé pour l'obtention des résultats.

4.1.1 Paramètrage de l'image

Nous avons défini 5 paramètres permettant de préciser pour une scène donnée l'image attendue :

1. La résolution verticale : la hauteur de l'image en pixel (par défaut *512*).
2. La résolution horizontale : la largeur de l'image en pixel (par défaut *512*).
3. La profondeur maximale : le nombre de réflexions ou/et de réfractions à partir duquel la contribution d'un rayon au pixel est considérée comme négligeable (par défaut *5*).
4. Le nombre de récursions : chacune des images testées est construite à partir d'une fonction récursive permettant d'augmenter le nombre d'objets (pour chaque résultat présenté, ce nombre sera précisé)
5. La taille de la texture : afin de pouvoir tester des objets de grosse taille, nous nous autorisons à compliquer la texture associée à chacun des objets (par défaut *0*, texture définie dans SPD (annexe B))

4.1.2 Configuration du code

Nous utilisons un fichier de paramètres qui est lu par l'application et qui permet de fixer des options d'optimisation du noyau parallèle.

Tout d'abord, l'utilisateur doit préciser la ou les fonctionnalités qui seront utilisées lors de l'exécution :

1. Flots de calculs.
2. Flots de données.
3. Utilisation de l'équilibrage dynamique.
4. Flots multi-informatifs.
5. Flots concurrents.

Pour chacune de ces fonctionnalités des paramètres doivent être fixés :

1. Pour l'option flots de calculs :
   a. Le nombre minimal de calculs à envoyer pour autoriser un envoi de calculs (*Seuil_Flots_Calculs*).
   b. Le nombre minimal de calculs à envoyer pour autoriser un envoi de calculs lors d'un message multi-informatifs (*Seuil_Flots_Mixtes*).
   c. Le nombre maximum de calculs qui peuvent être envoyés dans un même message (*Dose_Max*).
2. Pour l'option flots de données :
   a. Le nombre maximum de demandes de données simultanées que peut effectuer un processeur (*Nb_canaux_donnees*).
   b. Le rapport entre le nombre de données bufferisables et la valeur précédente (*Max_Demandes_Donnees_Coeff*).
3. Pour l'option équilibrage dynamique :
   a. Le nombre de calculs restants à partir duquel un processeur doit faire une demande de calculs (*Famine*).
   b. Le nombre maximum de calculs qui peuvent être envoyés à un processeur lors d'une demande de calculs (*Dose*).
   c. Le nombre minimal de calculs que doit posséder un processeur afin d'accepter de faire un transfert de calculs (*Seuil_Donneur*).
4. Pour les options à flots mixtes, il faut fixer l'ensemble des paramètres nécessaires aux options flots de calculs et flots de données.

Un exemple de fichier de configuration est donné en annexe C.

4.1.3 Mode opératoire

Les machines parallèles utilisées sont de trois types (voir détails en annexe A) :

1. Un calculateur parallèle CRAY T3D comprenant 128 processeurs.
2. Un calculateur parallèle CRAY T3E comprenant 128 processeurs.
3. Un réseau de stations de travail SUN.

Le placement physique des processus sur les processeurs ne peut pas être géré par l'application.

Les communications se font soit par PVM, soit par MPI. La taille des images est de *512x512* pixels, la profondeur maximale des rayons est de *5*.

Les temps de calcul donnés correspondent à la durée comprise entre le moment où tous les processeurs ont terminé leurs précalculs (initialisation et chargement de leur portion de la base de données) - ce temps de départ est assuré par une barrière de synchronisation - et l'instant où le dernier processeur a terminé de calculer son dernier rayon (la phase de concentration des rayons calculés sur un unique processeur n'est donc pas prise en compte). Les temps nécessaires à l'exécution de la phase de précalcul sont donnés en annexe B pour les différentes scènes présentées dans notre travail.

Les temps de calcul "séquentiel" sont obtenus sur les différentes machines par utilisation du noyau parallèle sur un unique processeur. Ces temps sont également présentés en annexe B.

Le but de notre noyau étant de permettre une certaine généricité, nous n'exploitons pas la cohérence de rayons pour une scène donnée. Au contraire, nous cherchons à distribuer de façon

homogène les rayons sur les différents processeurs. Toutefois, l'algorithme de lancer de rayon étant très sensible au type de scène traitée, il apparaîtra malgré tout, des différences significatives dans le traitement des scènes.

Le lancement du code s'opère de deux façons distinctes suivant la nature de la machine utilisée : SPMD (CRAY) ou MIMD (réseau de stations).

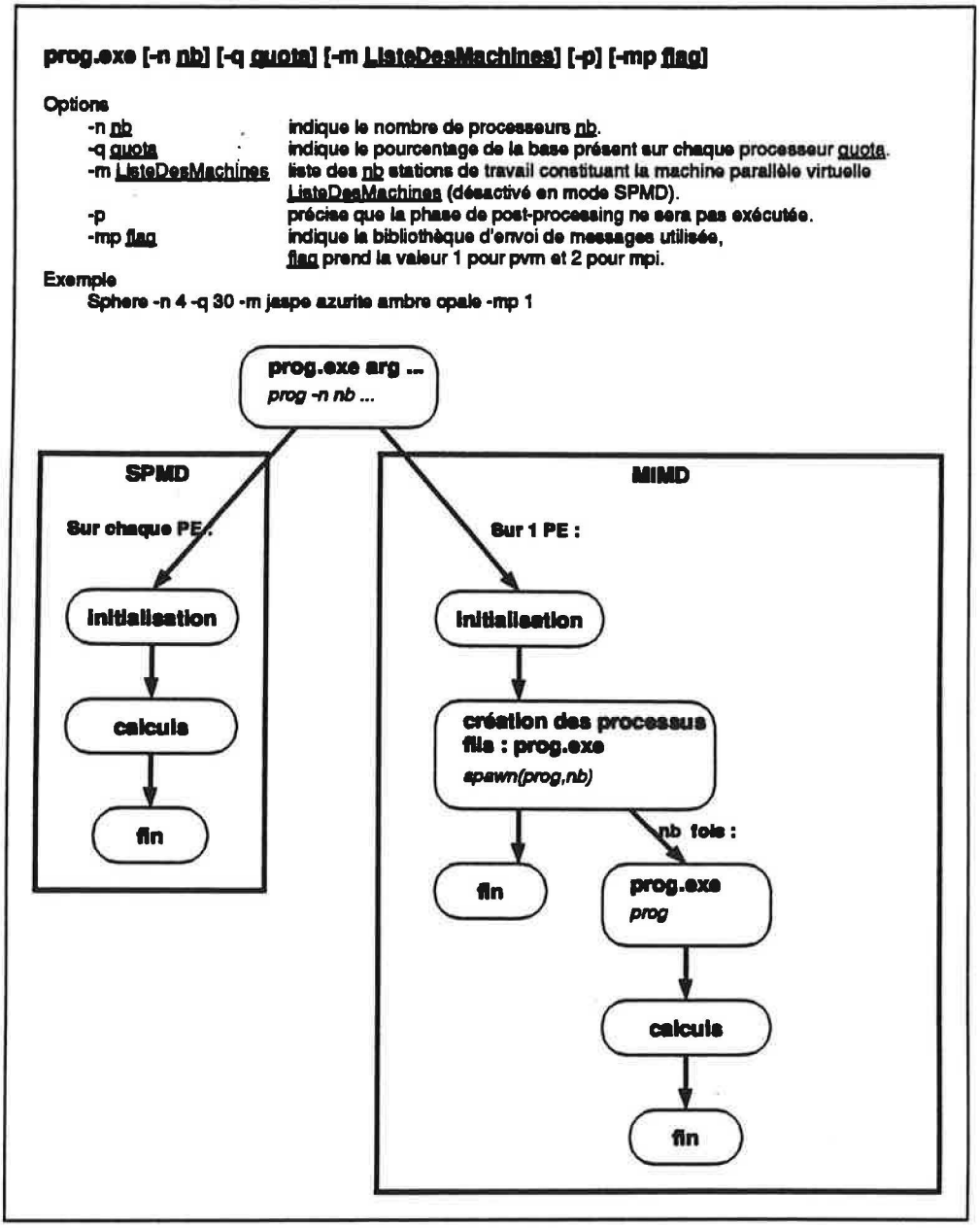

Figure 58 Procédure de lancement du logiciel

Le temps obtenu est le temps utilisateur, c'est à dire le temps d'attente de l'utilisateur derrière son écran avant d'avoir de nouveau la main. Ce temps peut comporter des surcoûts parasites provenant de l'engorgement du réseau dû à d'autres utilisateurs ou de l'utilisation partielle d'un processeur par un processus étranger à notre application. On peut noter toutefois que ce dernier cas ne peut pas se produire pour notre application sur les CRAY, car les processeurs sont dédiés à un unique utilisateur. Par contre, sur station, ils sont fort probables, nous avons cherché à les atténuer au maximum en utilisant ces machines aux horaires de faible activité et en réalisant un plus grand nombre de mesures.

4.1.4 Scènes utilisées

Les scènes utilisées sont issues de la base de données SPD [Hai87] (voir en annexe B les représentations et caractéristiques des scènes). Elle sont sensées constituer un échantillon représentatif des scènes les plus courantes.

4.1.5 Avertissements

En raison du mode d'exploitation des calculateurs au centre de Limeil, les possibilités d'utilisation des 128 processeurs sont réduites. C'est pourquoi la plupart des résultats se limiteront à une configuration comportant 64 processeurs, le passage à 128 processeurs étant réservé à la mise en évidence de phénomènes particuliers. De plus, le mode interactif ne permettant pas l'utilisation des machines pour des durées supérieures à 10 minutes, la plupart de nos résultats auront une durée inférieure à cette limite.

Enfin, notre travail s'intéressant plus particulièrement aux environnements massivement parallèles, nous privilégions la présentation de résultats sur 64 processeurs. Sauf indication contraire, les expérimentations sont réalisées sur le CRAY T3E et en utilisant la librairie d'échanges de messages PVM.

4.2 Algorithmes classiques

Afin de pouvoir évaluer le bien-fondé de nos algorithmes mixtes, nous présentons nos résultats également pour les algorithmes à flots de données et flots de calculs. Puis nous comparons les résultats obtenus afin de déterminer s'ils vérifient ceux prédits par notre modélisation.

Avant cela, nous commençons par montrer les performances de l'algorithme sans flot.

4.2.1 Sans flot

Cet algorithme ne comprend qu'un équilibrage statique (répartition par points). En effet, l'équilibrage dynamique que nous proposons ne peut être utilisé qu'avec un code basé sur le flot de données.

La figure suivante présente l'accélération obtenue pour des images variées :

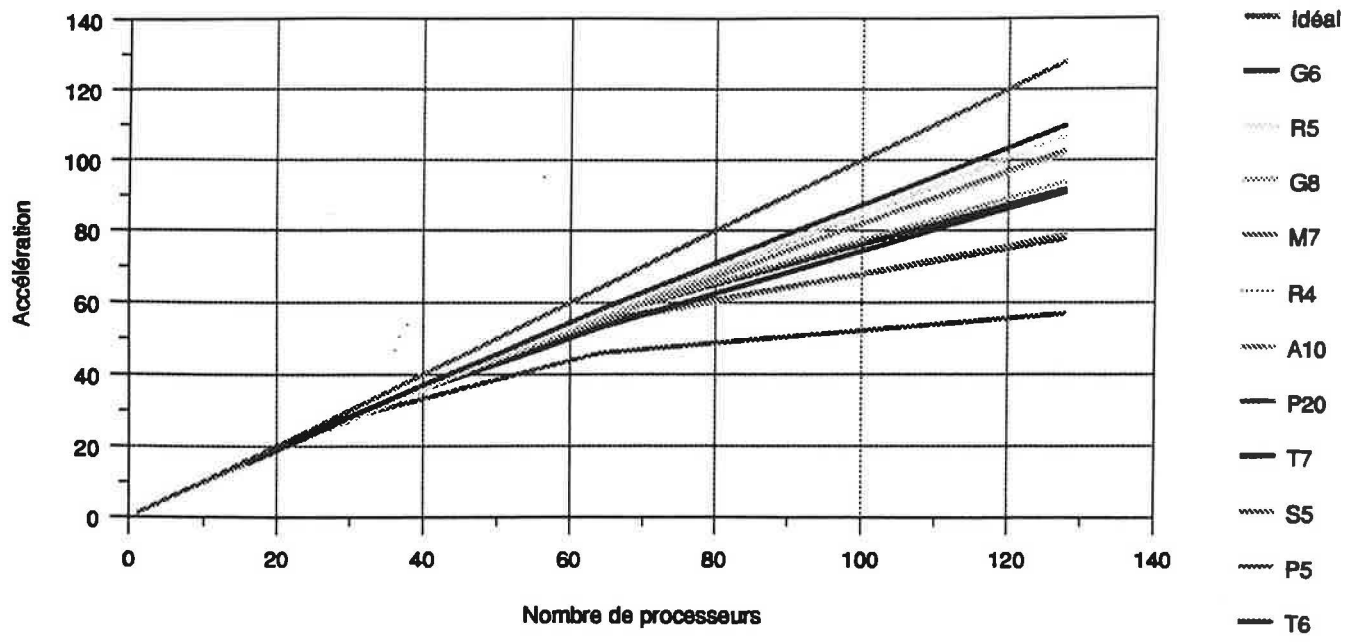

Figure 59 Accélération avec base dupliquée

Il apparaît que pour des configurations comprenant jusqu'à 32 processeurs, l'accélération reste très proche de l'accélération idéale. Au-delà, l'accélération continue à progresser mais moins rapidement que l'augmentation du nombre de processeurs. Afin d'analyser plus finement ce ralentissement des gains en temps, nous allons étudier l'efficacité obtenue pour ces images :

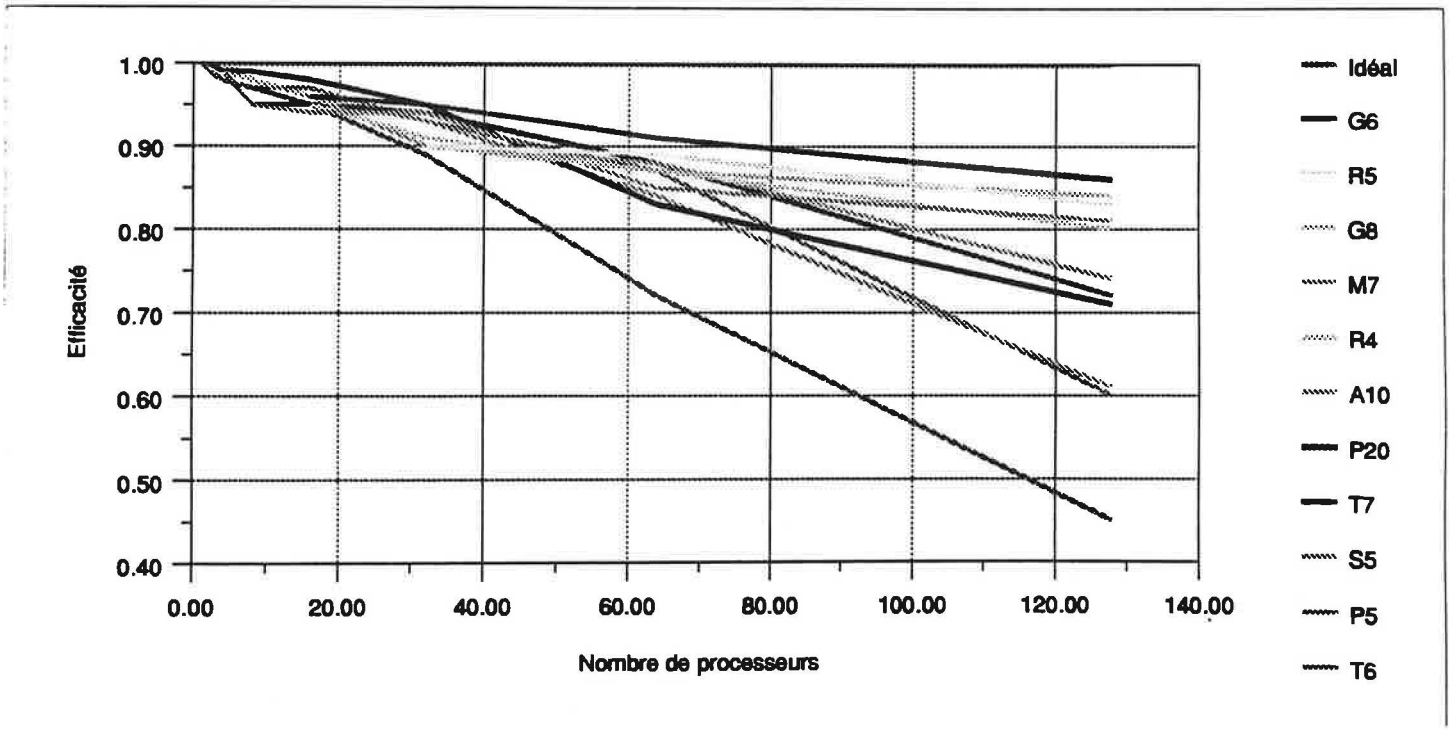

Figure 60 Efficacité avec base dupliquée

L'efficacité reste très bonne jusqu'à 32 PEs (autour de 90 %). Au-delà l'efficacité diminue régulièrement. En effet, il apparaît que le nombre de processeurs est trop important pour qu'une

distribution statique suffise à assurer un bon équilibrage de charges. Toutefois pour 128 processeurs l'efficacité reste encore correcte (supérieure à 45 %).

Il nous faut également préciser que les efficacités les plus basses sont obtenues à partir de temps de calculs très faibles (t<15 s) où les erreurs de mesures et l'algorithme de terminaison pèse un poids important sur le temps de calcul. Par contre, quand ces critères deviennent négligeables (t>30 s), l'efficacité est bien meilleure (entre 72 et 86 %).

4.2.2 Flots de données

Le développement de notre noyau parallèle commence par la réalisation d'un algorithme à flots de données. D'abord, nous étudions les résultats obtenus pour un algorithme sans équilibrage dynamique de charges, ensuite l'amélioration obtenue grâce à notre algorithme d'équilibrage.

4.2.2.1 Flots de données sans équilibrage dynamique de tâches

Nous allons étudier les résultats donnés par cet algorithme et à partir de ceux-ci déterminer d'éventuelles voies permettant de l'améliorer.

Sur les graphiques suivants sont présentés les temps de calculs de différentes images pour des nombres de processeurs différents en fonction du pourcentage de la scène possédée par chacun des processeurs.

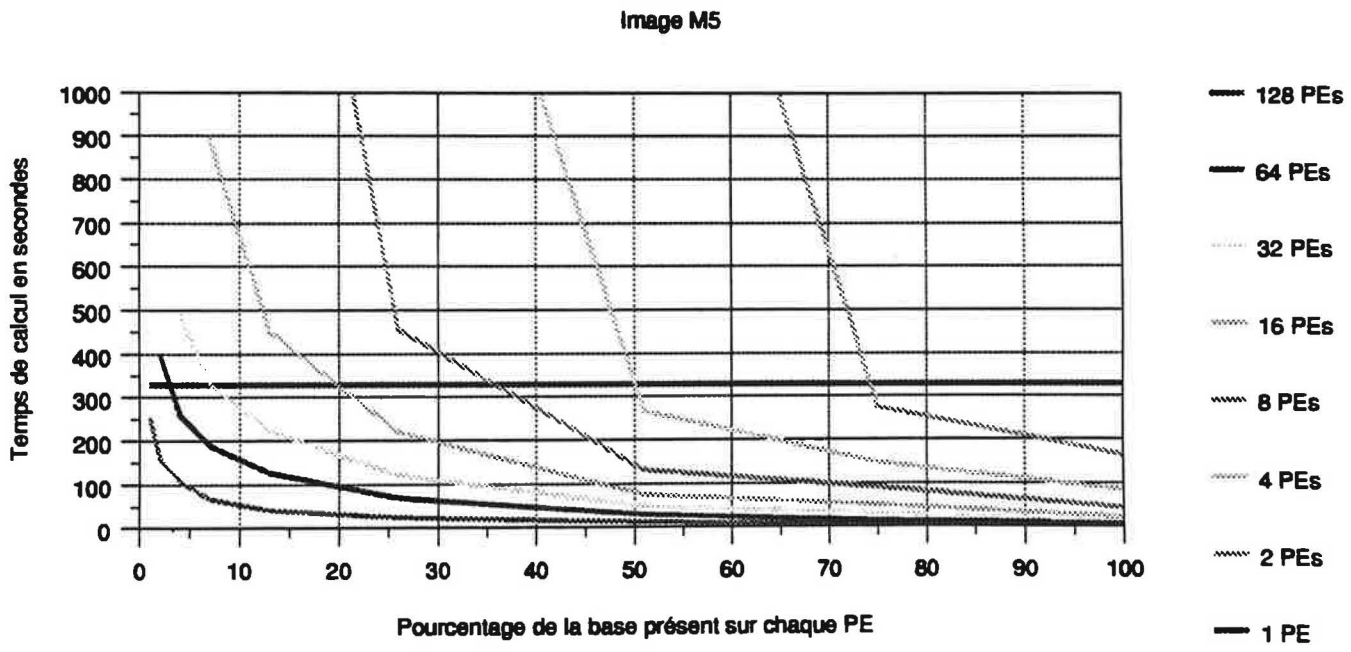

Figure 61 Temps de calcul en fonction de la distribution de la scène pour des images différentes pour des nombres de processeurs différents (Suite) . . .

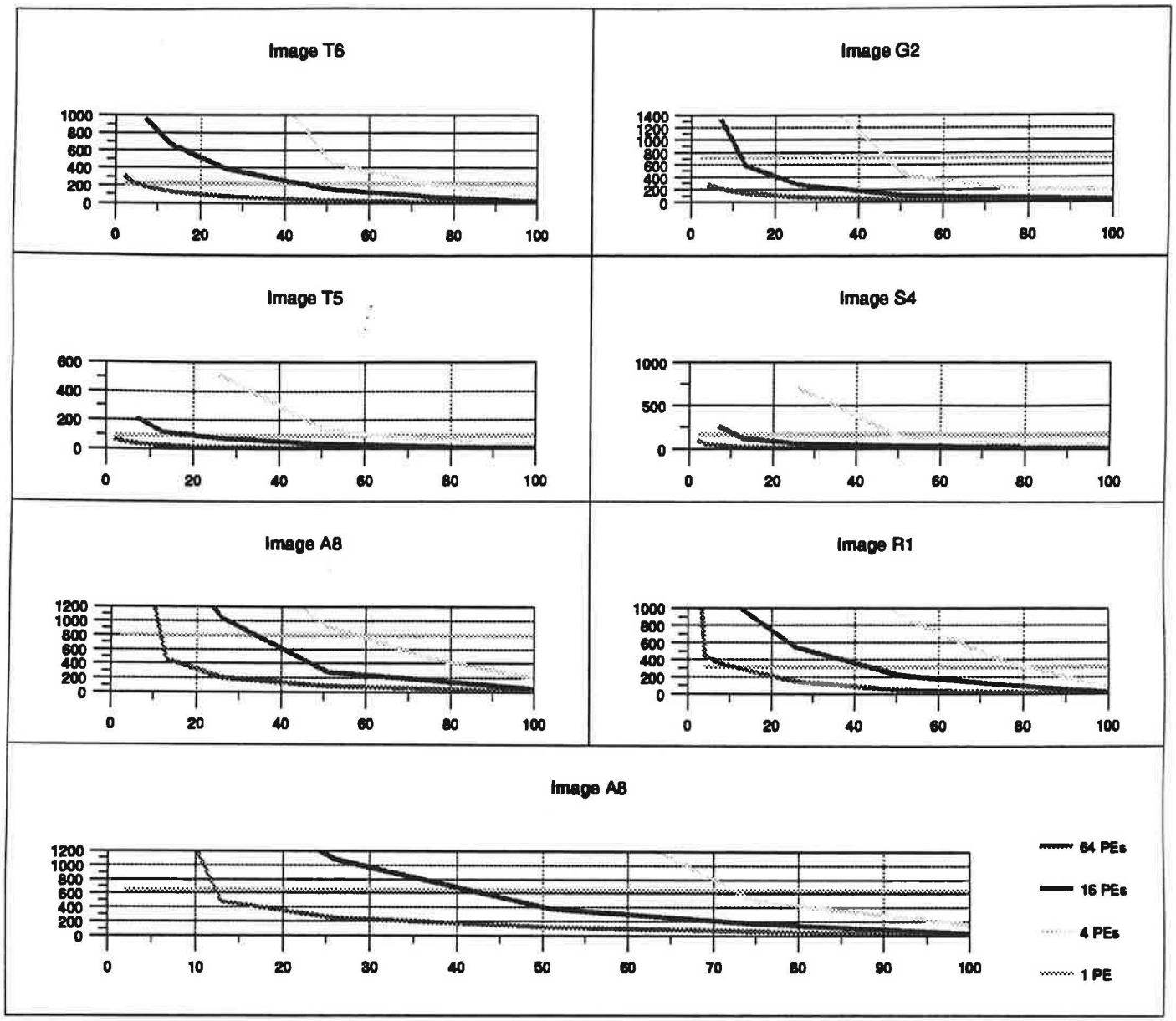

Figure 61 Temps de calcul en fonction de la distribution de la scène pour des images différentes pour des nombres de processeurs différents

Une première remarque d'importance est que le comportement de l'algorithme est relativement indépendant de la scène traitée, les graphiques ont des allures très proches.

Sur chaque courbe, on peut dissocier deux zones : quand la plus grande partie de la base de donnée est présente sur chaque processeur, le surcoût dû à une diminution du cache est faible. Par contre, à partir d'un certain seuil dépendant du nombre de processeurs, le temps de calcul est très sensible à une diminution du cache, en effet $\lim_{cache \to 0} t = +\infty$.

On peut noter également que cet algorithme permet une augmentation significative de la taille de la scène traitée sans que le surcoût soit prohibitif. Avec 128 processeurs ne possédant que 1% de la base M5, on obtient un temps de calcul inférieur à celui pour cette même image en séquentiel.

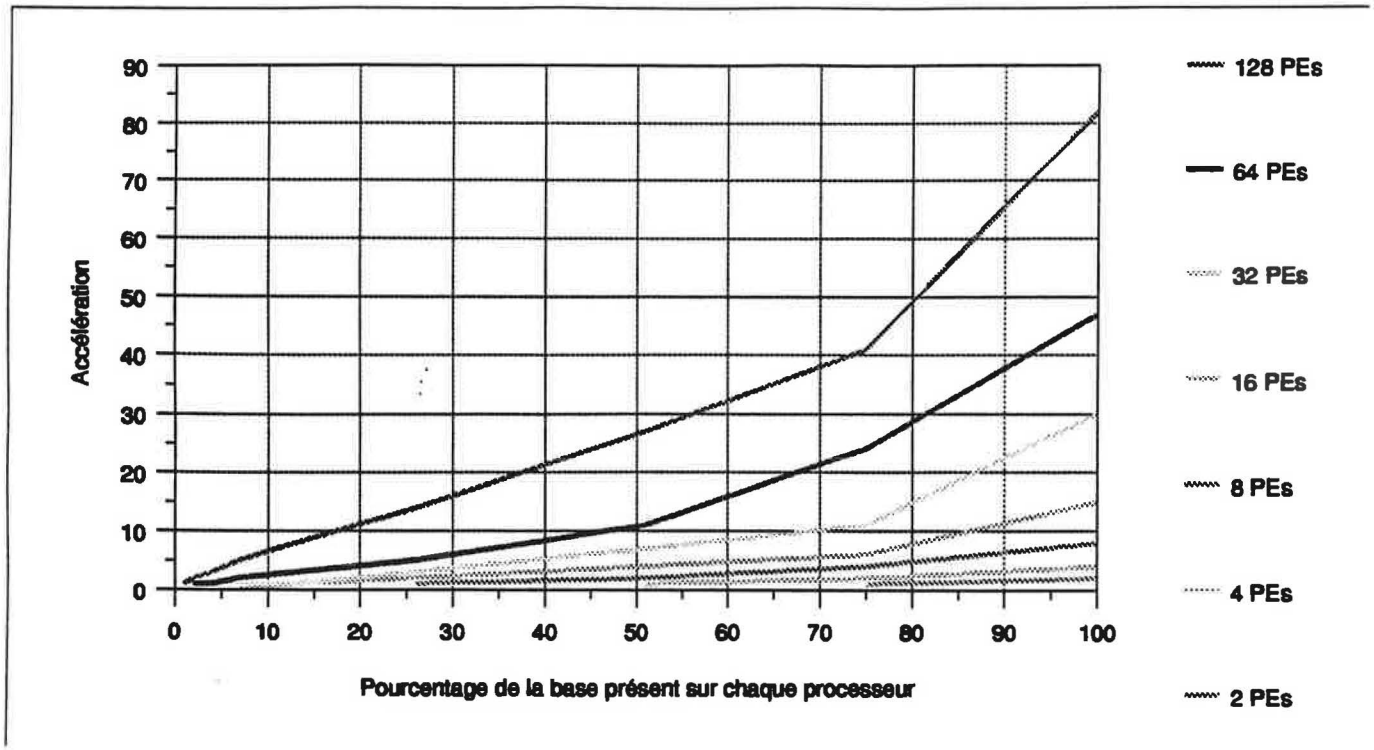

Figure 62 Accélération en fonction de la distribution de la scène pour l'image M5.

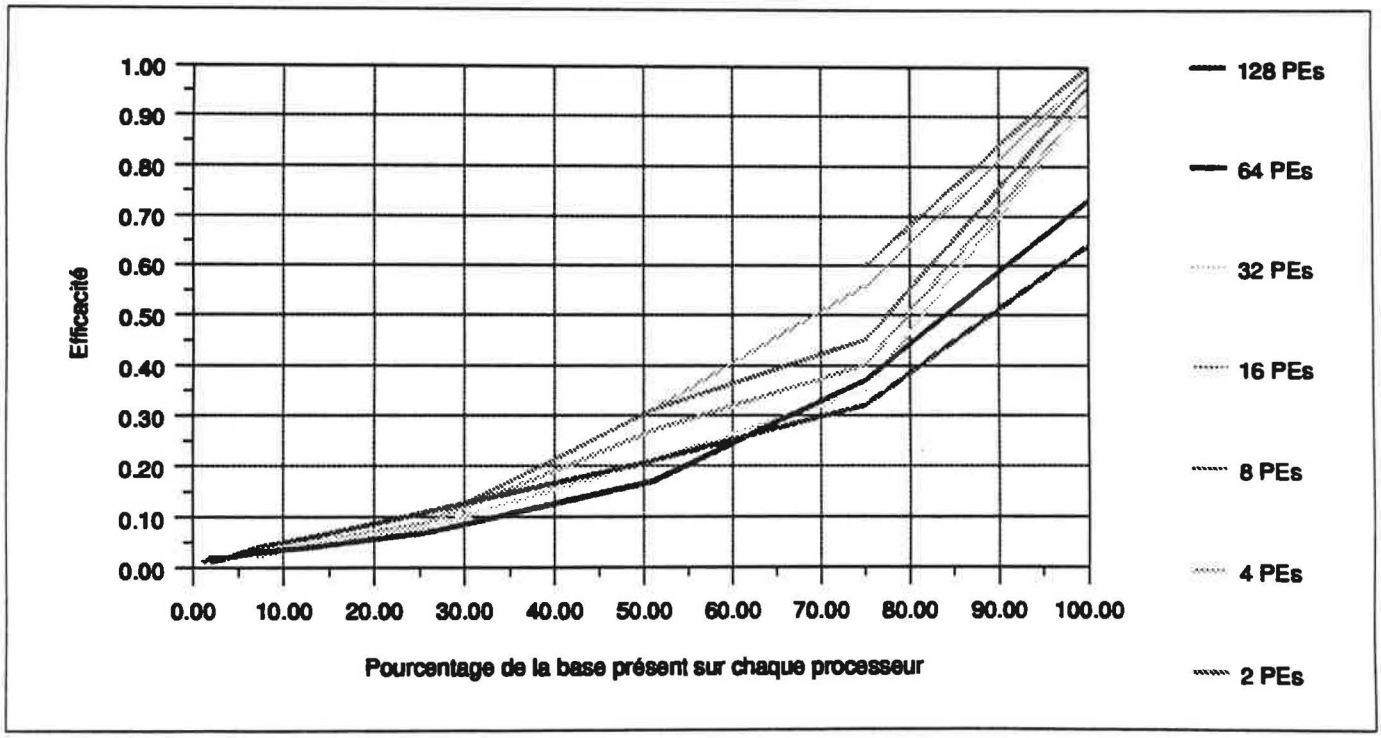

Figure 63 Efficacité en fonction de la distribution de la scène pour l'image M5.

La figure précédente représentant l'efficacité montre que celle-ci baisse avec l'augmentation du nombre de processeurs. Ce qui justifie la nécessité de compléter cet algorithme par un mécanisme d'équilibrage de charge dynamique. Toutefois, il apparaît qu'avec la réduction de la taille du cache, les efficacités des différentes configurations ont tendance à se rapprocher. Ceci peut s'expliquer

par le fait que l'augmentation du nombre de calculs sur chacun des processeurs induit une moins grande disparité des temps moyens de calcul des différents processeurs.

Afin d'étudier l'accélération pour une répartition donnée de la base de données en fonction du nombre de processeurs, nous présentons la figure suivante :

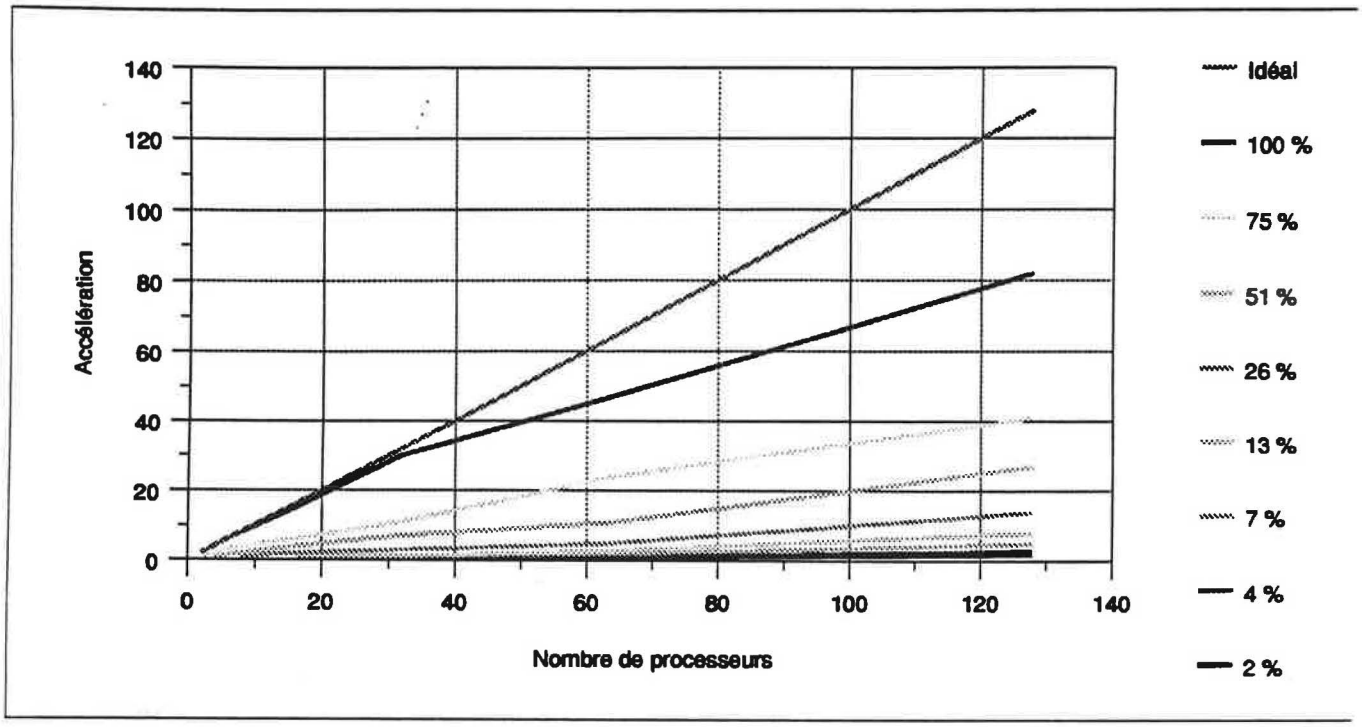

Figure 64 Accélération pour une distribution donnée en fonction du nombre de processeurs pour l'image M5.

L'accélération semble rectiligne quelle que soit la distribution de la base de données. La pente est donnée par la taille du cache, l'accélération restant proportionnelle au nombre de processeurs.

Maintenant, nous allons nous intéresser au nombre de messages échangés pour le calcul d'une image. Nous allons présenter le nombre de messages moyen par processeur par seconde pour une image donnée, c'est à dire la division du nombre de messages émis en moyenne par un processeur par le temps de calcul total de l'image.

Nous présentons ci-dessous les résultats obtenus pour l'image M5.

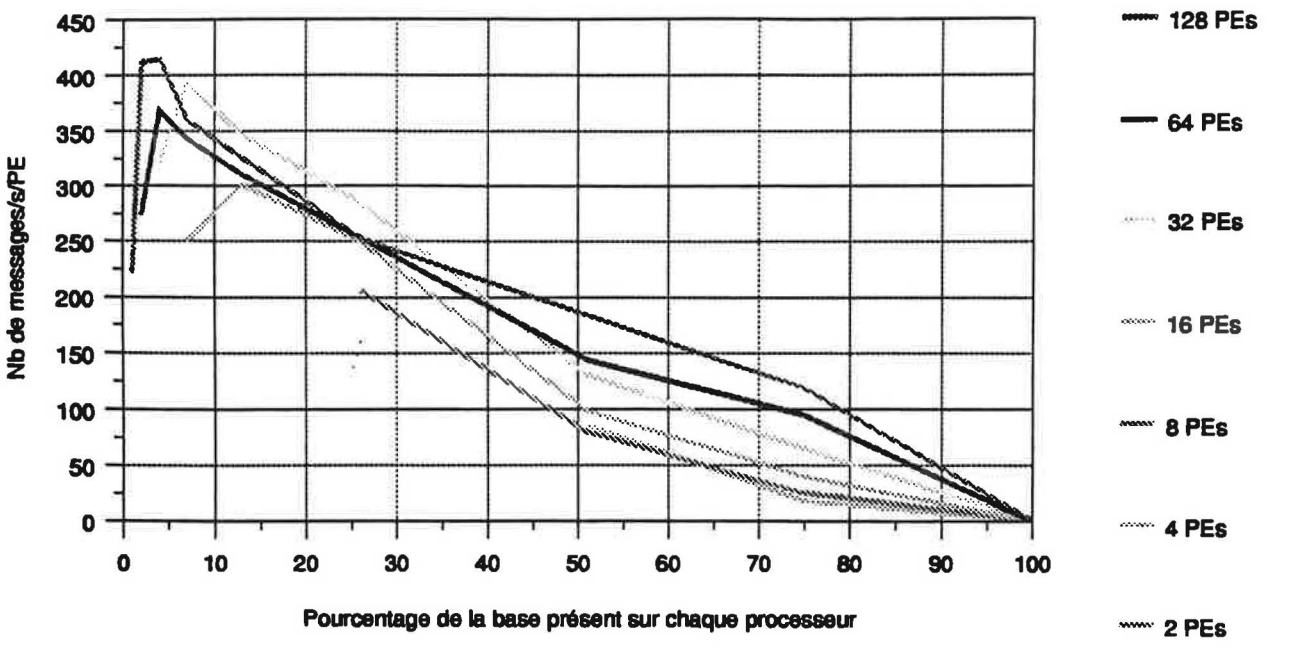

Figure 65 Nombre moyen de messages émis par PE par seconde pour M5

Ces courbes sont composées de deux phases :

1. Une première phase, où la diminution de la taille du cache provoque un accroissement du taux moyen de messages par seconde. Ce taux est relativement stable pour une distribution donnée quel que soit le nombre de processeurs utilisés.
2. Une deuxième phase, où après avoir atteint un taux seuil (autour de 350 msg/s/PE) le taux se met à chuter. Cette chute peut s'expliquer par le fait qu'à partir d'un certain taux le réseau et/ou l'algorithme se mettent à saturer. Ainsi le taux de réception des messages se réduit. Cela conduit à :

   a. Un temps d'attente passif de messages qui augmente
   b. Un taux d'envoi de demandes de données plus faible, ce qui conduit au regroupement de calculs qui nécessitent une même donnée. Aussi certaines demandes de données peuvent être évitées.

   Le temps de calcul total de l'image est augmenté considérablement et le nombre de demandes de données cesse d'augmenter de façon aussi importante que précédemment (on peut même voir une diminution du nombre de demandes dans certains cas). Celà provoque une baisse du taux moyen de messages par seconde.

Il apparaît donc que cet algorithme peut induire pour des tailles de cache faibles (moins de 15 %) une saturation du réseau et/ou de l'algorithme, ce qui provoque une dégradation des performances. La recherche d'un algorithme plus performant va conduire à une réduction importante du nombre de communications.

Enfin, nous nous sommes intéressés aux liens entre l'efficacité et la taille de la base de données. La figure suivante donne pour une même base de données, mais pour un nombre d'objets différent (nombre de récursion différent), l'efficacité pour 64 processeurs en fonction de la taille du cache.

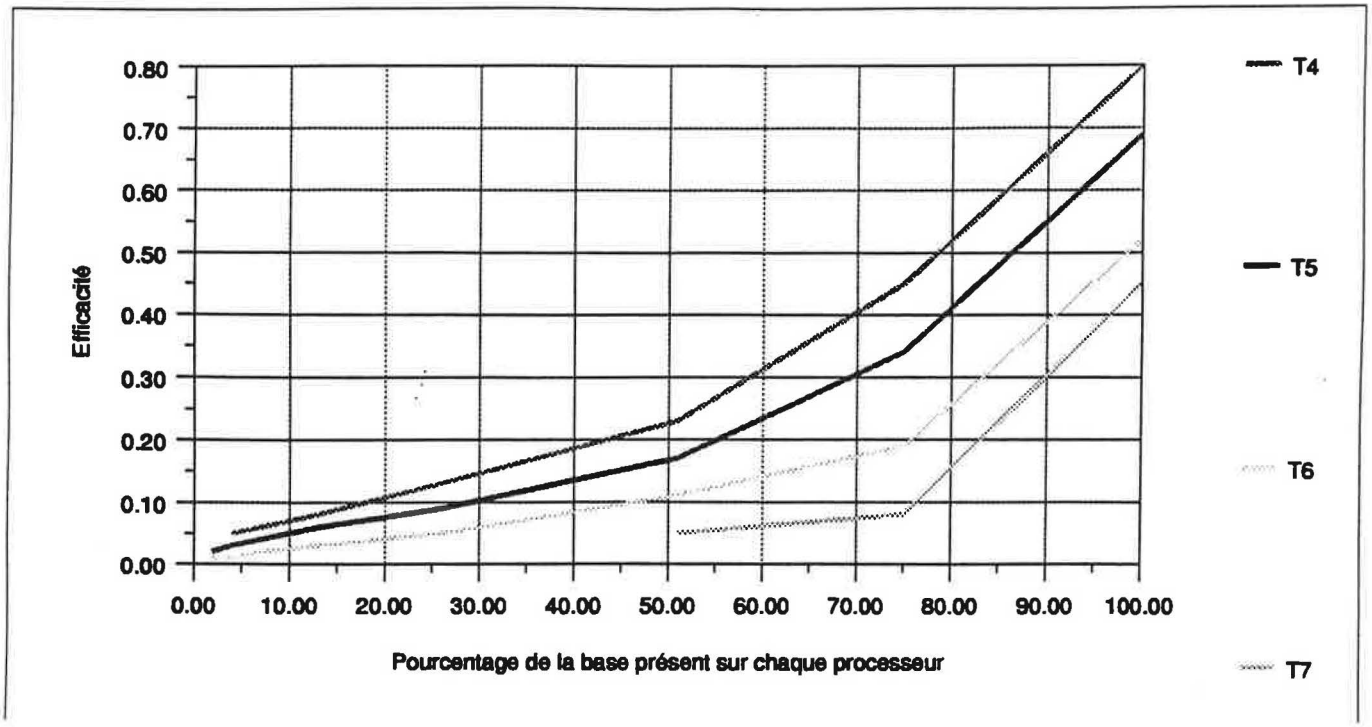

Figure 66 Efficacité pour 64 processeurs en fonction de la taille du cache pour l'image Tetrahedron avec des nombres de récursion différents.

Image	Nombre d'objets
T4	256
T5	1024
T6	4096
T7	16384

Table 4 Nombre d'objets pour différentes récurtion de l'image Tetrahedron

Il apparaît que l'augmentation du nombre d'objets induit une baisse de l'efficacité. Cela peut s'expliquer assez facilement. En effet, le temps de calcul d'un rayon dépend du nombre d'objets candidats à une intersection, une baisse du nombre d'objets induit donc un raccourcissement du temps de calcul des rayons. Aussi l'écart entre le temps de calcul d'un rayon qui donnera naissance à des rayons secondaires et le temps de calcul d'un rayon quittant la scène se resserre. Ce rapprochement des temps de calcul entre les rayons va donc entraîner un meilleur équilibrage des charges de travail entre les processeurs.

Le tableau suivant résume l'ensemble des remarques provenant de l'analyse de cet algorithme :

Avantages	Inconvénients
Comportement général ne dépendant pas de la scène traitée	A partir d'un certain seuil de cache, le surcoût en temps de calcul devient très important
Permet de traiter des scènes de grande taille	Equilibrage insuffisant
Accélération proportionnelle au nombre de processeurs	Saturation quand la taille du cache est faible

Table 5 Récapitulatif pour l'algorithme à flots de données

4.2.2.2 Flots de données avec équilibrage dynamique de tâches

Une amélioration classique de l'algorithme à flots de données est la mise en place d'un mécanisme d'équilibrage de tâches dynamique. La méthode originale que nous proposons donne de bons résultats comme le montrent les résultats suivants.

Afin de comparer cet algorithme avec l'algorithme précédent, nous avons divisé le temps de calcul de l'algorithme sans équilibrage par le temps de calcul de l'algorithme avec équilibrage. Les valeurs obtenues, que nous appelons les gains en temps, G_T, sont présentées sur la figure suivante.

$$G_T = \frac{T_{sans\ equilibre}}{T_{avec\ equilibre}}$$

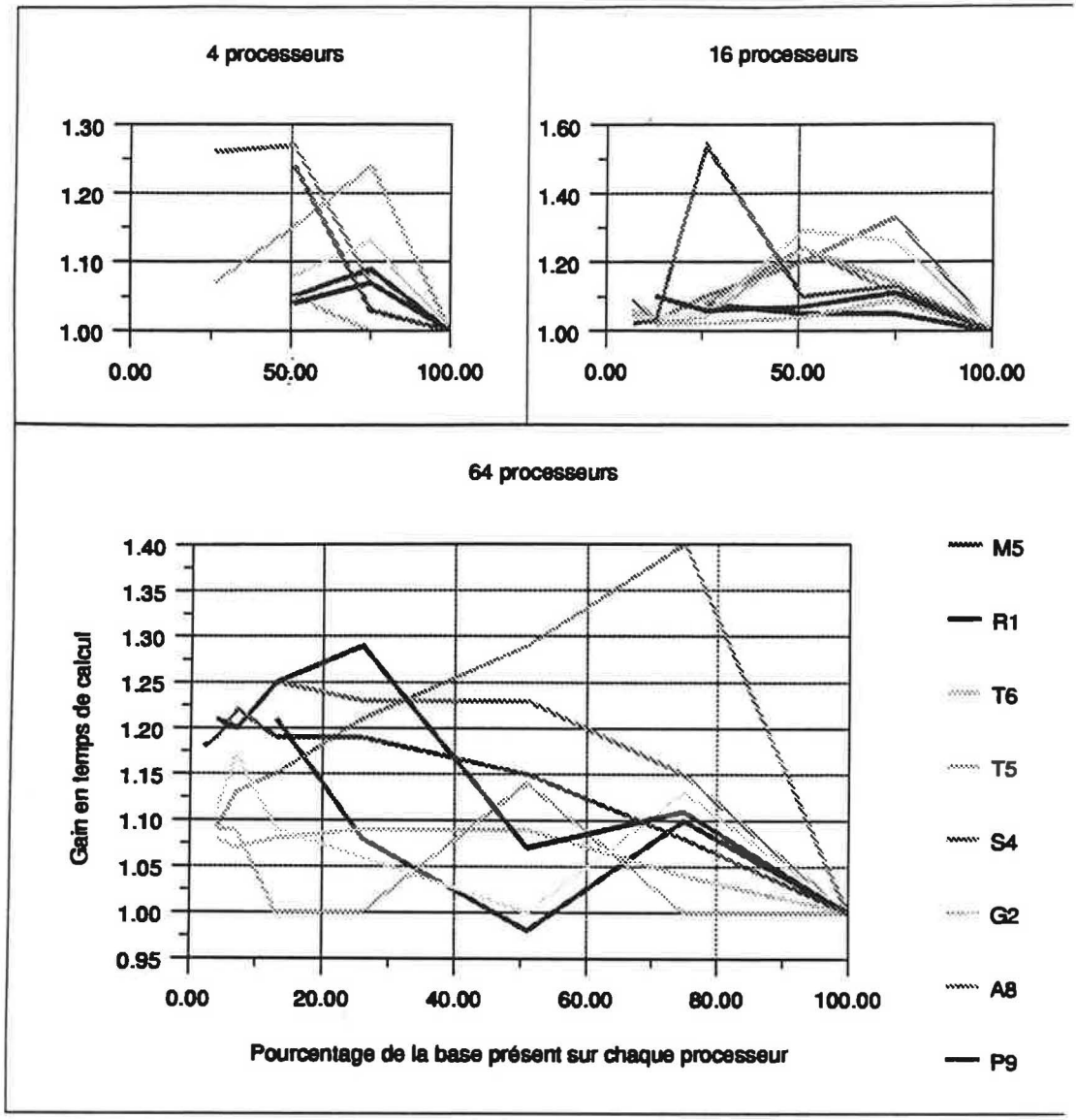

Figure 67 Gains en temps de l'algorithme à flots de données avec équilibrage par rapport à l'algorithme à flots de données sans équilibrage

Les gains en temps provenant d'un mécanisme d'équilibrage de tâches sont fortement liés à la nature des images testées. Cependant les résultats obtenus permettent de déterminer certaines constantes. Nous commenterons le graphique pour 64 processeurs car l'équilibrage est particulièrement intéressant pour des architectures comprenant de nombreux processeurs. Toutefois, les résultats obtenus pour d'autres configurations restent cohérents avec ces derniers.

Les gains obtenus sont majoritairement compris entre 1 et 1,4 et ces gains ont tendance à être meilleurs quand le cache est de petite taille. En effet, notre algorithme d'équilibrage de tâches est basé sur la connaissance de l'état de charge des autres processeurs. Cette connaissance étant transmise lors des demandes et des réceptions de données, il faut un certain flux de communications pour qu'elle soit suffisamment précise. A partir du moment où chaque processeur dispose d'au moins 13% de la base, le gain est compris entre 1,08 et 1,3 quelle que soit l'image considérée.

Etudions maintenant le surcoût en messages qu'induit cette forme d'équilibrage :

$$S_{Msg} = \frac{Msg_{sans\ equilibre}}{Msg_{avec\ equilibre}}$$

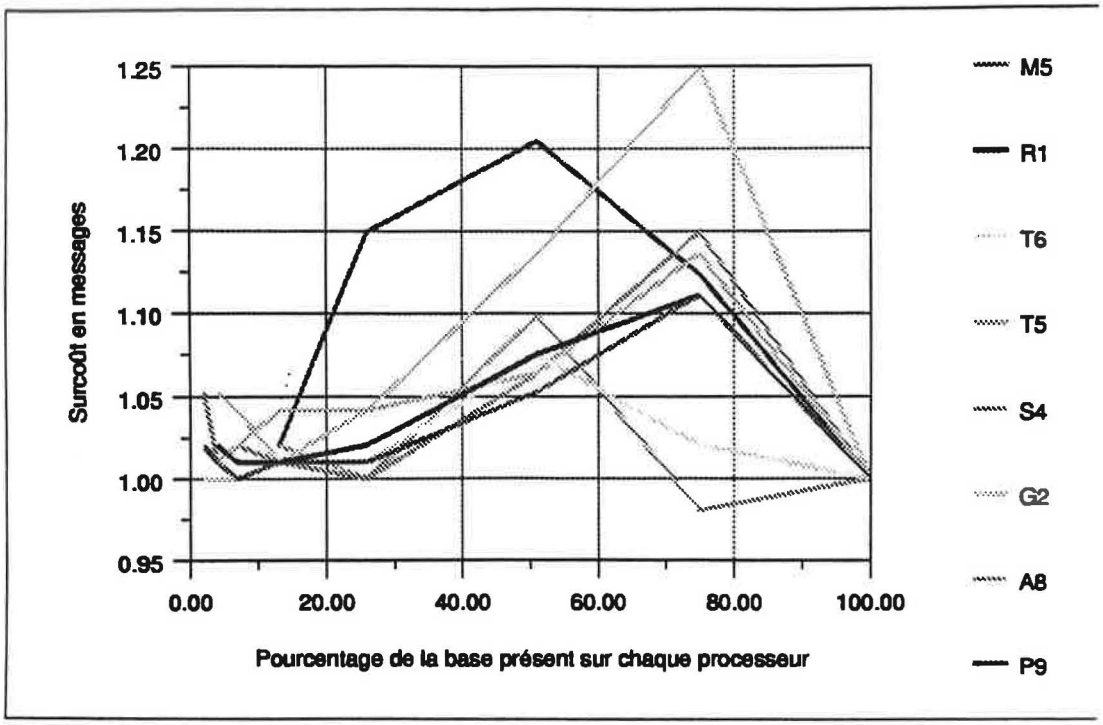

Figure 68 Surcoût en messages induit par l'équilibrage dynamique pour 64 processeurs

Le surcoût en messages engendré par l'équilibrage de charge décroit avec la diminution de la taille du cache. En fait, une étude plus fine permettant de comptabiliser les différents types de messages fait apparaître que le nombre de messages assurant l'équilibrage dynamique est de l'ordre d'une centaine par processeur quelle que soit l'image ou la taille du cache. En effet, le mécanisme d'équilibrage ne se déclenche qu'à la fin de l'exécution et a un coût quasi-constant.

Cela explique bien que la diminution de la taille du cache entraine un surcoût en messages de plus en plus faible (moins de 1,05).

Comparons maintenant le déséquilibrage de chacun des processeurs dans le cas de l'algorithme avec flots de données avec et sans équilibrage dynamique sur quelques exemples :

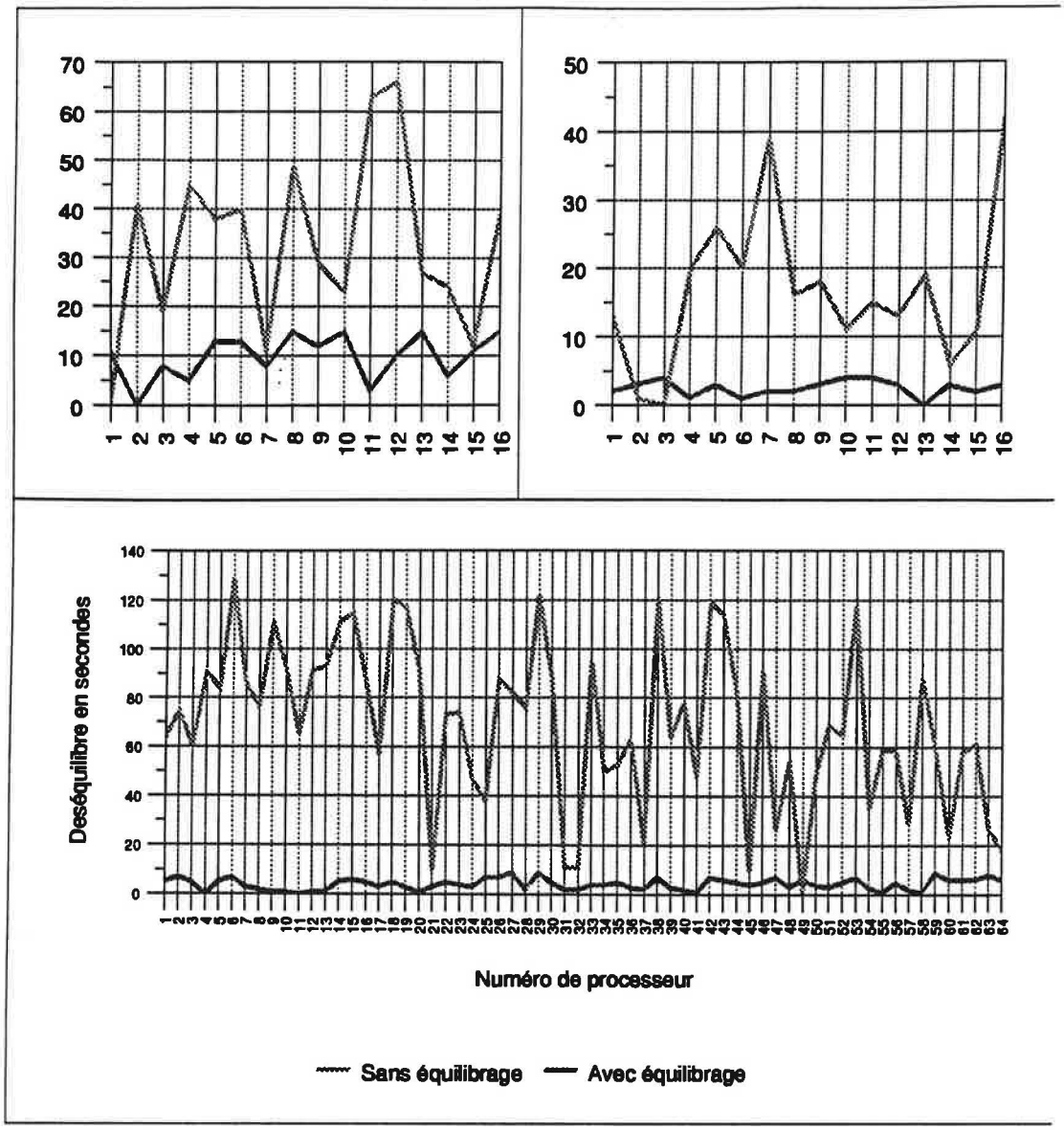

Figure 69 Ré-équilibrage de la charge de chaque processeur

L'effet rééquilibrant de notre méthode d'équilibrage apparaît clairement sur ces graphiques.

Afin d'essayer d'évaluer la qualité de l'équilibrage obtenu, nous allons chercher à extrapoler à partir du temps obtenu sans équilibrage et des temps de déséquilibre de chacun des processeurs un temps de calcul optimal qui serait celui de l'algorithme sans mécanisme d'équilibrage dynamique et dont l'équilibrage statique serait parfait.

Posons $T_{ideal} = T_{calcul} - deseq_{max} + \frac{\sum deseq}{nb_{PEs}}$.

La valeur de T_{ideal} n'est bien sûr qu'indicative. En effet, l'équation précédente suppose qu'un algorithme idéal ne nécessite qu'un équilibrage final en temps, alors que d'autres facteurs peuvent contribuer également à l'amélioration de l'algorithme comme une bonne répartition des communications sur l'ensemble des processeurs et sur l'ensemble du temps de calcul. Toutefois, cette valeur T_{ideal} reste un indicateur intéressant.

Calculons T_{ideal} sur les exemples présentés précédemment ainsi que les rendements obtenus avec et sans notre algorithme d'équilibrage dynamique.

	$deseq_{max}$	$\sum deseq$	T_{calcul}	T_{ideal}	T_{equ}	$\frac{T_{ideal}}{T_{cal}}$	$\frac{T_{ideal}}{T_{equ}}$
M5-16	42 s	270 s	452 s	427 s	440 s	94,5 %	97 %
T6-16	66 s	526 s	665 s	632 s	642 s	95 %	98,4 %
M5-64	129 s	4503 s	401 s	342 s	341 s	85,3 %	100,3 %

Table 6 Amélioration de l'équilibrage de la charge avec notre équilibrage dynamique

Les résultats précédents montrent que l'algorithme proposé permet bien de se rapprocher de T_{ideal}.

Cela indique également que les gains encore susceptibles d'être obtenus par un meilleur équilibrage dynamique de l'algorithme à flots de données sont relativement faibles. Aussi, il semble bien qu'il faille chercher une autre voie pour améliorer la parallélisation de l'algorithme de lancer de rayon.

4.2.3 Flots de calculs

Comme nous l'avons vu dans le chapitre précédent, la performance de l'algorithme à flots de calculs dépend fortement du problème traité. De plus, nous n'avons pas souhaité nous investir dans l'implémentation d'un mécanisme d'équilibrage de tâches qui ne serait pas réutilisé dans nos stratégies à flots mixtes. Il n'est donc pas étonnant que notre approche générique ne puisse pas donner des résultats concluants quelle que soit l'image considérée. Aussi les résultats présentés seront partiels et ne concerneront que les images T5, T6 et S4.

Les courbes que nous présentons sont quelque peu originales puisque le plus souvent les résultats avec flots de calculs sont donnés pour une répartition totalement disjointe de la base de données sur les processeurs. Toutefois, il est également possible de dupliquer une partie de la base comme dans le cas du flot de donnée. On obtient les courbes suivantes :

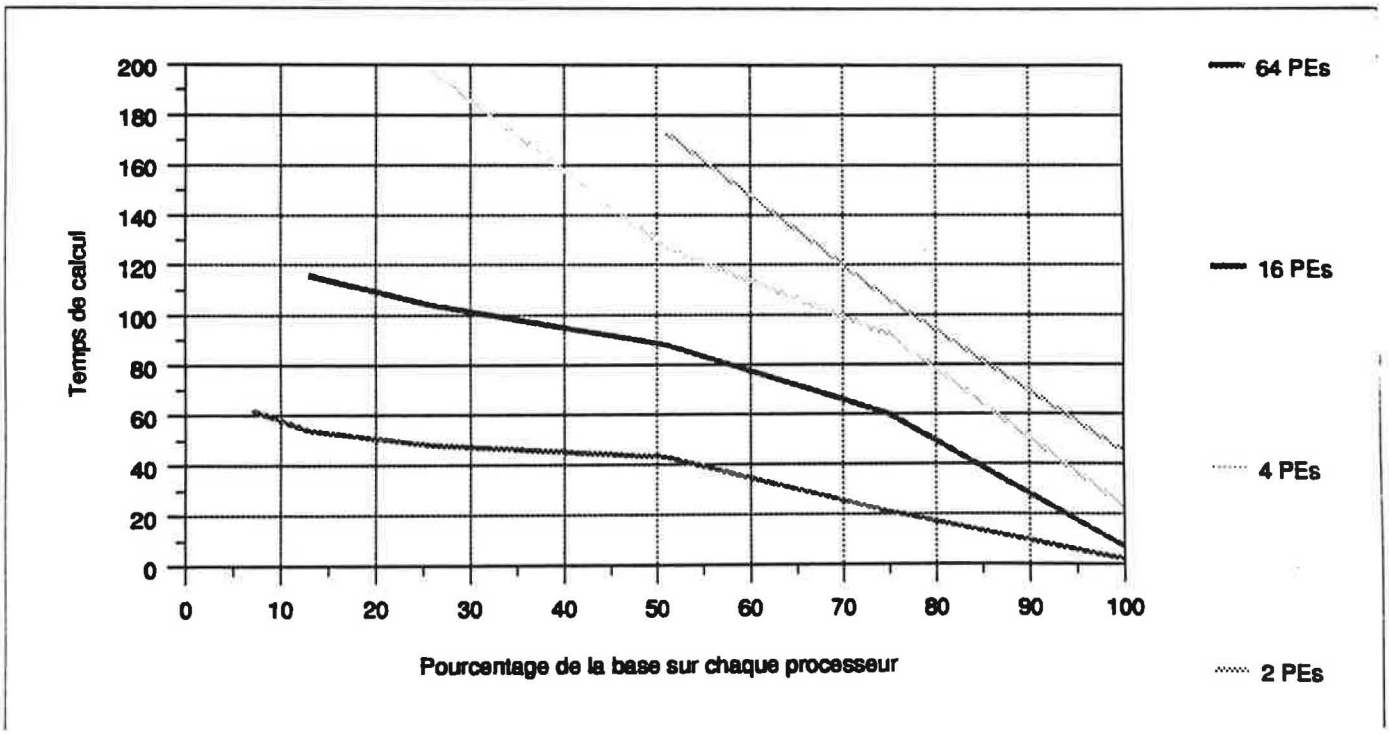

Figure 70 Temps de calcul pour l'image T5

Contrairement à l'algorithme à flots de données, les courbes précédentes ne possèdent pas d'asymptote verticale.

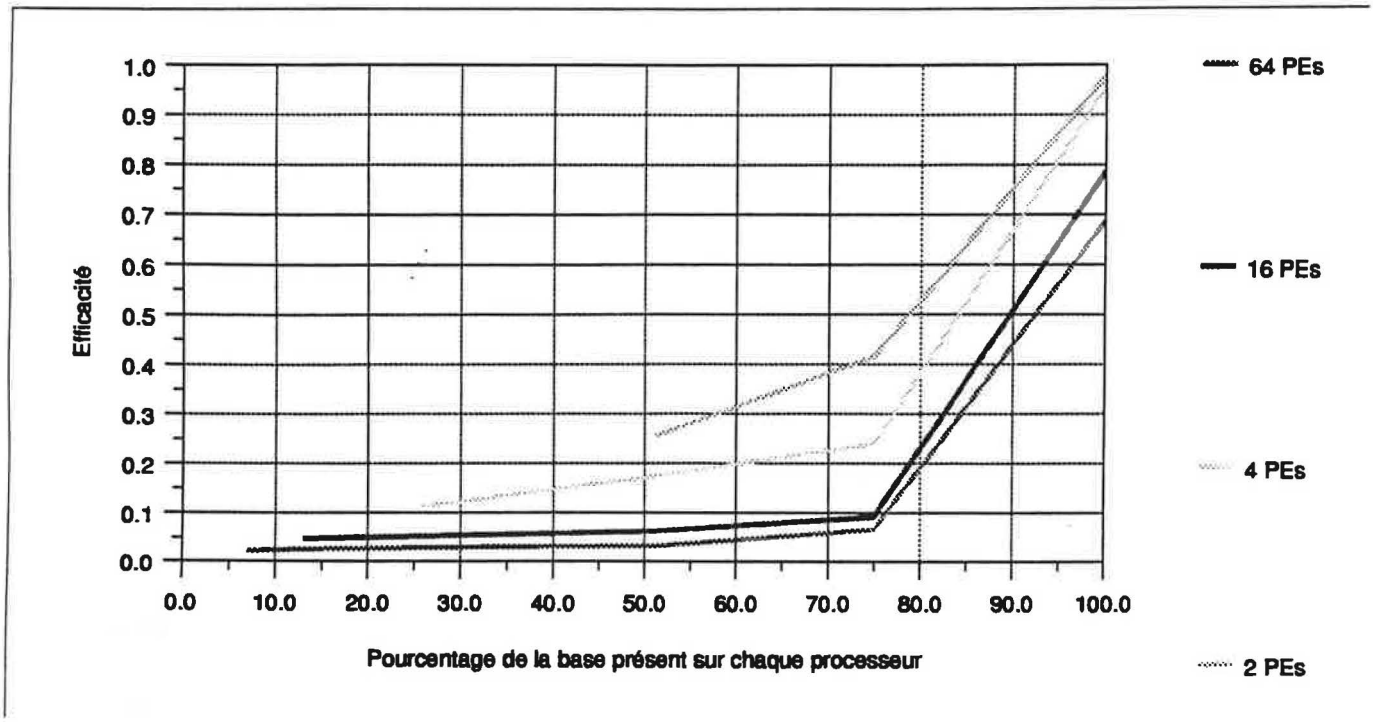

Figure 71 Efficacité pour l'image T5

L'efficacité de l'algorithme baisse de façon très significative quand le nombre de processeurs augmente.

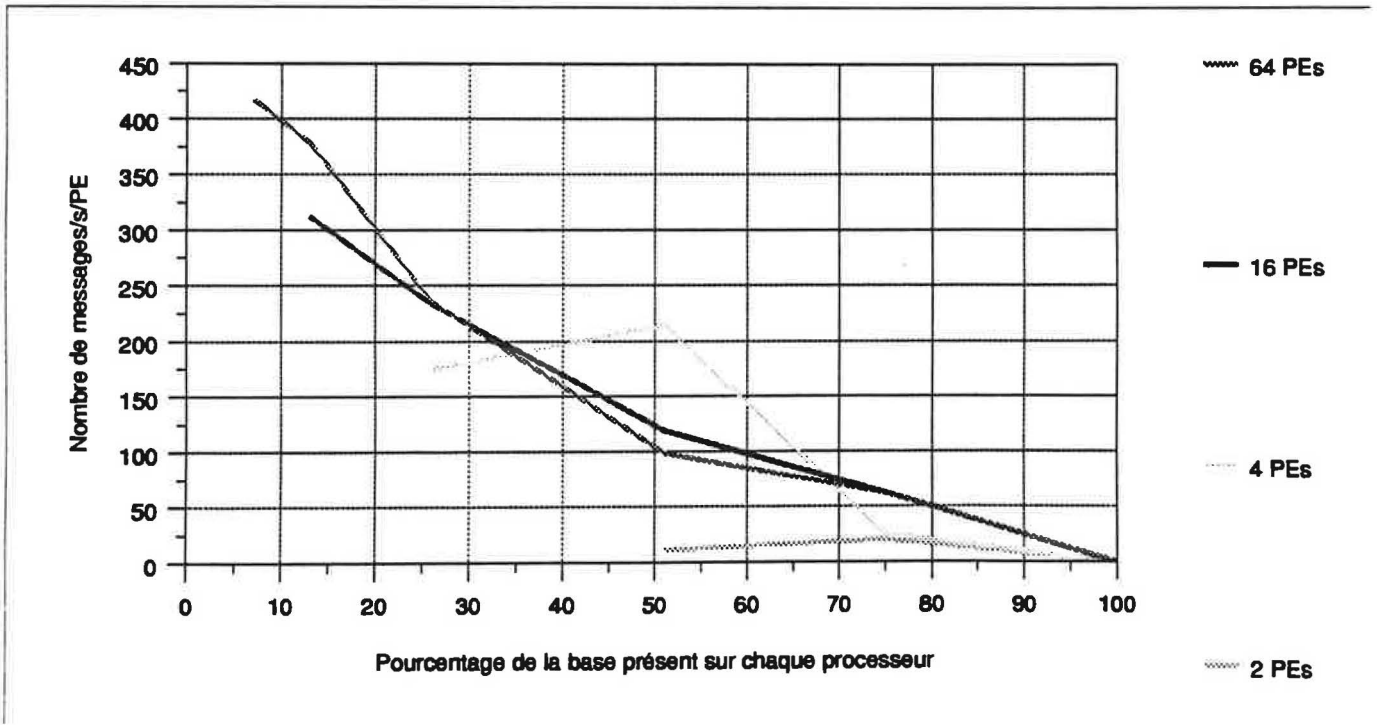

Figure 72 Nombre de messages

La réduction de la taille de la taille de la base conduit à une augmentation linéaire du nombre de messages émis par seconde.

Avantages	Inconvénients
Permet de traiter des scènes de grande taille	Comportement dépendant fortement de la scène traitée
La diminution de la taille du cache conduit a un temps de calcul ne tendant pas vers une asymptote verticale	Baisse importante de l'efficacité avec l'augmentation du nombre de processeurs

Table 7 Récapitulatif pour l'algorithme à flots de calculs

4.2.4 Comparaison entre les algorithmes à flots de données et à flots de calculs

Comme nous l'avons précisé précédemment, les résultats obtenus pour l'algorithme à flots de calculs ne sont que partiels. Leurs comparaisons avec ceux de l'algorithme à flots de données ne pourront donc avoir valeur de preuve, mais contribueront à la comparaison entre ces deux algorithmes.

La figure suivante permet de comparer les temps de calcul obtenus pour les deux algorithmes classiques en fonction du pourcentage de la base présent sur chaque processeur.

Nous définissons le gain en temps G_T de l'algorithme à flots de calculs sur l'algorithme à flots de données par l'inverse du rapport de leurs temps de calcul :

$$G_T = \frac{T_{flots\ de\ donnees}}{T_{flots\ de\ calculs}}$$

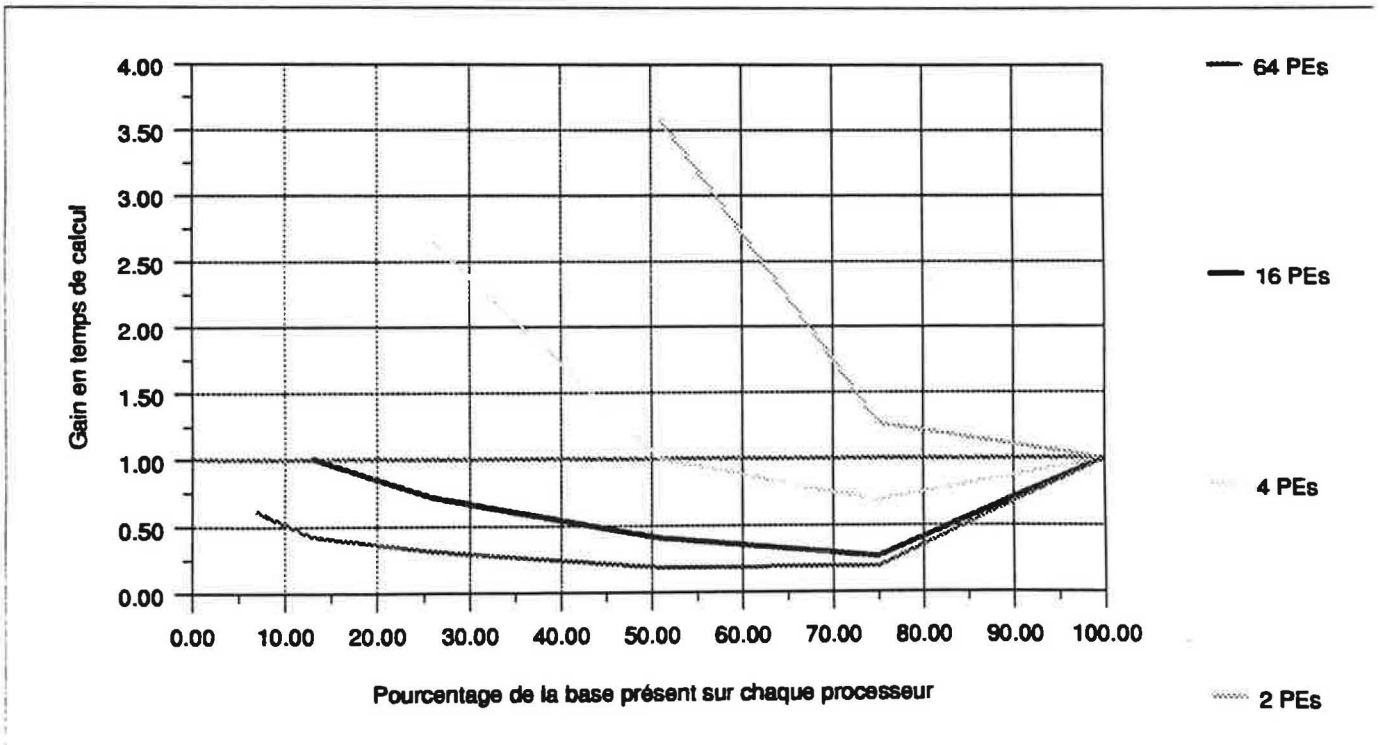

Figure 73 Gains en temps de l'algorithme à flots de calculs par rapport à l'algorithme à flots de données

Les résultats sont conformes avec ceux provenant de notre modélisation :

1. L'algorithme à flots de données est le meilleur quand chaque processeur possède une partie importante de la base (partie droite de la figure).
2. Par contre l'algorithme à flots de calculs ne peut être le meilleur que quand chaque processeur ne possède qu'une faible partie de la base (partie gauche de la figure).
3. Enfin, plus le nombre de processeurs est important, plus l'algorithme à flots de données tend à supplanter celui à flots de calculs.

Cela confirme bien que le choix d'un algorithme efficace doit dépendre au moins de la taille de la mémoire de chaque processeur, de la taille de la scène et du nombre de processeurs utilisés.

Flots de données	**Flots de calculs**
Performant quand chaque processeur possède une partie importante de la base de données	Plutôt performant quand chaque processeur possède une faible partie de la base de données
Performant quand le nombre de processeurs est important	
Relativement indépendant de l'image traitée	Très dépendant de l'image traitée
Equilibrage dynamique aisé	Equilibrage dynamique difficile

Table 8 Récapitulatif de la comparaison flots de données et flots de calculs

La voie d'un choix statique entre ces deux méthodes en fonction de ces paramètres ayant été écartée, nous présentons maintenant les résultats de nos algorithmes à flots mixtes.

4.3 Algorithmes à flots mixtes

Afin d'améliorer les résultats obtenus par les algorithmes classiques, nous avons proposé deux méthodes originales : les algorithmes à flots multi-informatifs et à flots concurrents. Nous présentons maintenant les résultats obtenus en les comparant avec ceux de l'algorithme à flots de données.

4.3.1 Flots multi-informatifs

Tout d'abord, étudions les gains en temps obtenus par cette méthode par rapport à l'algorithme à flots de données.

$$G_T = \frac{T_{flots\ de\ donnees}}{T_{flots\ multi-informatifs}}$$

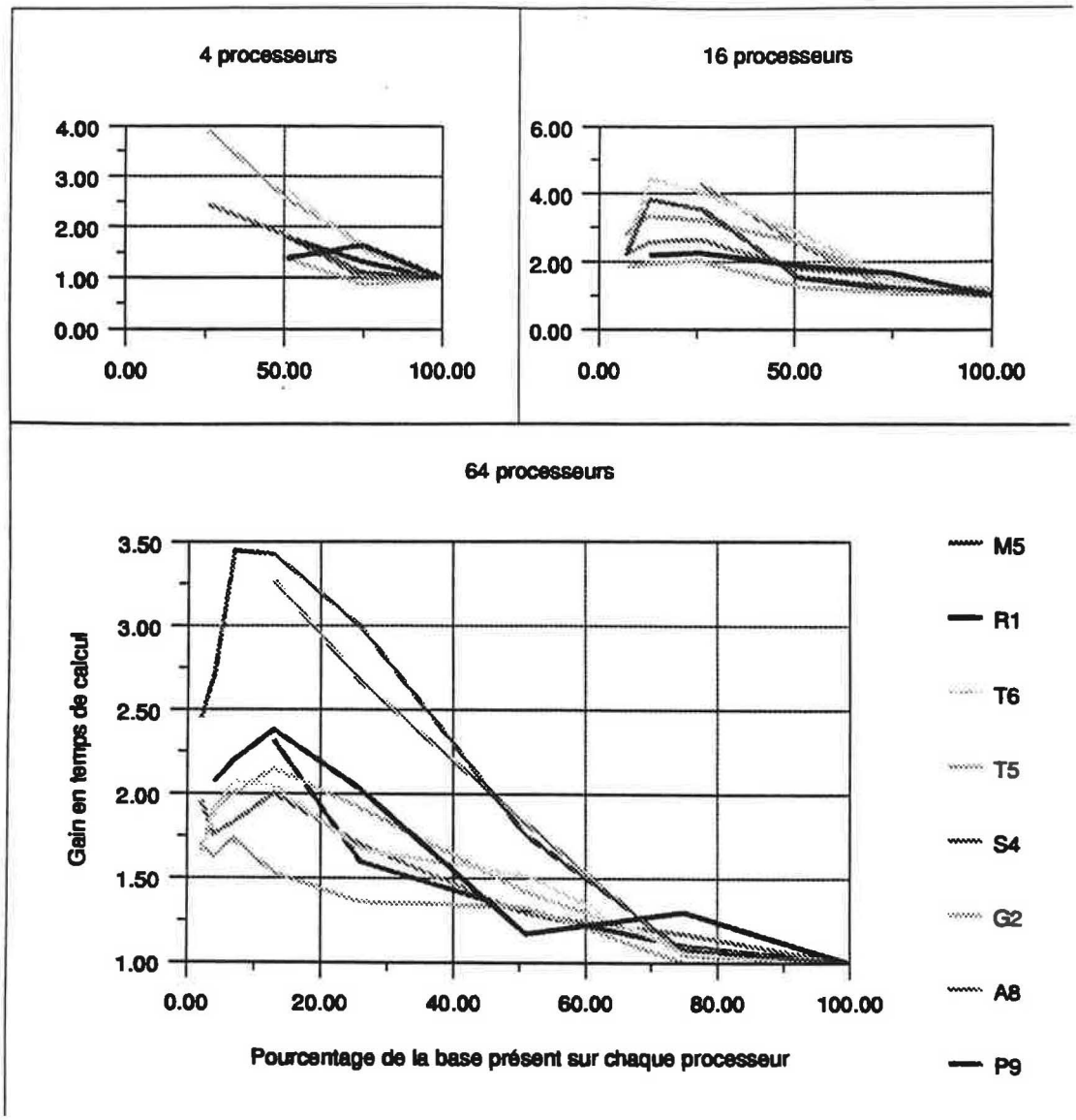

Figure 74 Gains en temps de l'algorithme à flots multi-informatifs par rapport à l'algorithme à flots de données

De nouveau, nous mettrons l'accent sur les résultats obtenus pour 64 processeurs. Il apparaît tout d'abord que ces gains en temps sont moins dépendants de l'image considérée que pour l'équilibrage dynamique présenté précédemment.

Nous obtenons un gain de 1,5 à 3,5 quand la base de données est suffisamment distribuée. Quand elle est peu distribuée, les résultats sont variables. En fait, nous pouvons expliquer ces résultats par le fait que quand la base est insuffisamment distribuée, le flot de communications ne permet pas de rafraichir de façon satisfaisante les informations au sujet de l'état de charge des différents processeurs, ce qui amène parfois l'algorithme à effectuer des choix inefficaces. On peut noter que les gains sont en augmentation quand le nombre de communications augmente (état de charge plus précis).

Etudions maintenant le gain en nombre de messages qu'induit cette méthode :

$$G_{Msg} = \frac{Msg_{flots\ de\ donnees}}{Msg_{flots\ multi-informatifs}}$$

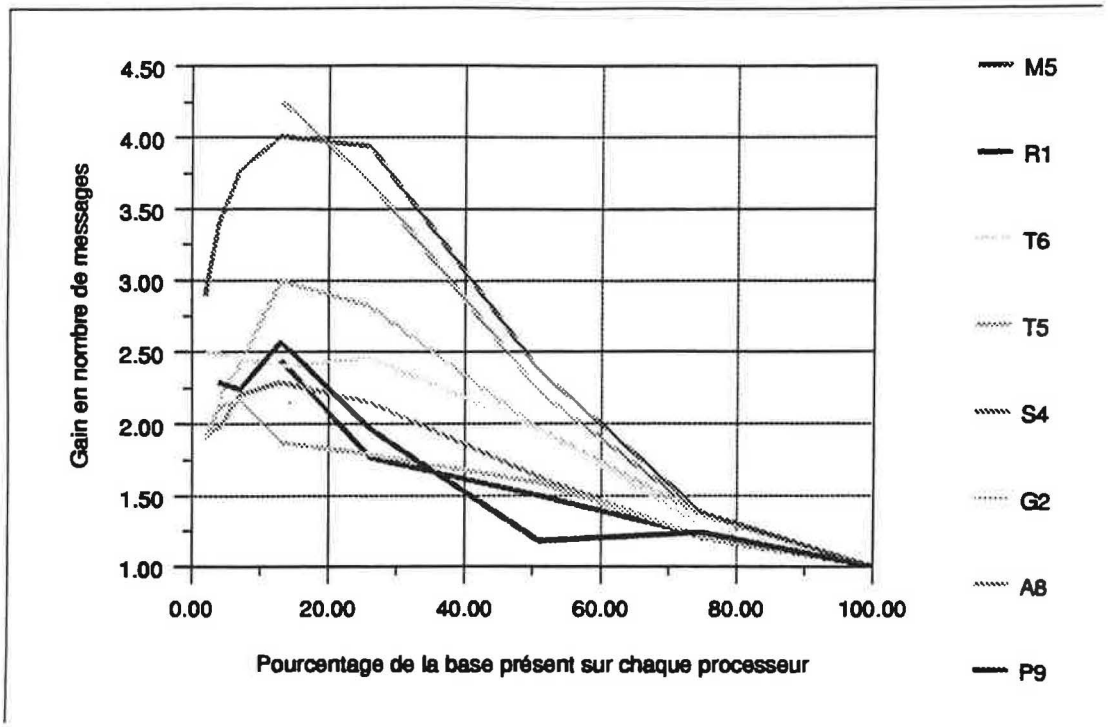

Figure 75 Gains en nombre de messages de l'algorithme à flots multi-informatifs par rapport à l'algorithme à flots de données

Il apparaît que ce nouvel algorithme, en plus d'accélérer les temps de calcul, assure une réduction notable du nombre de messages comprise entre 2 et 4, une fois qu'il y a suffisamment de communications pour permettre un bon rafraichissement des connaissances de l'état de charge du réseau. D'ailleurs, on peut noter que le gain en temps de calcul est fortement lié au gain en nombre de messages.

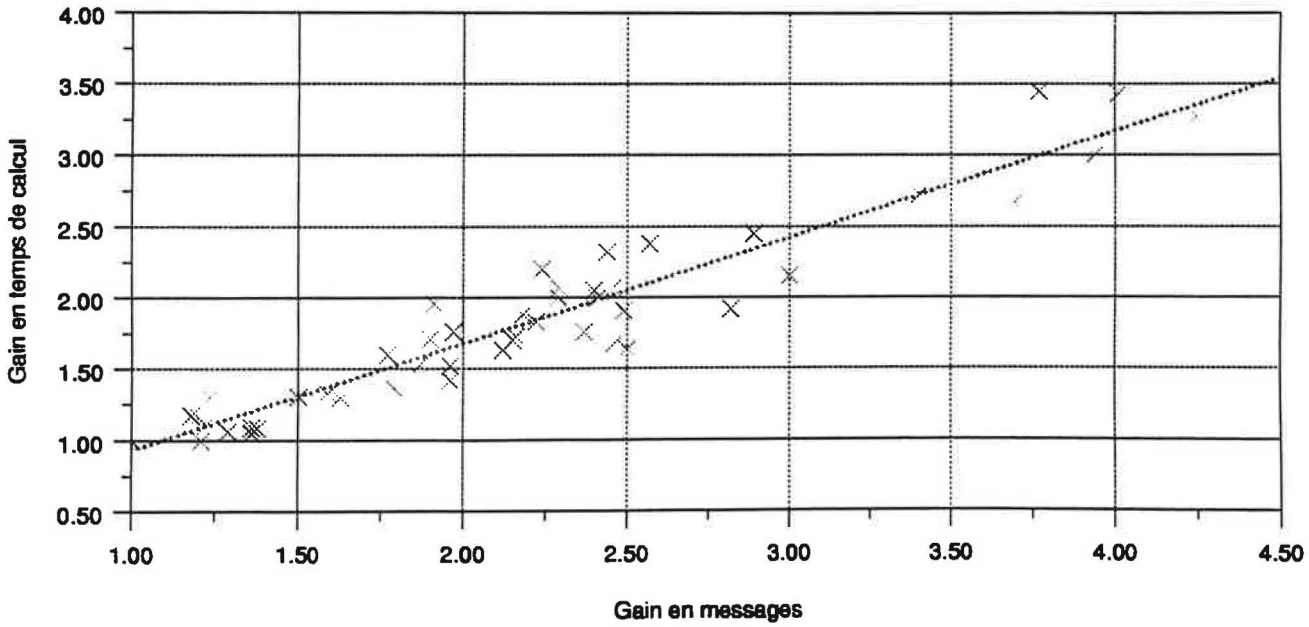

Figure 76 Relation entre gain en temps et gain en messages

La figure précédente représente pour l'ensemble des images le gain en messages en fonction du gain en temps pour une configuration de 64 processeurs. Une direction privilégiée apparaît. Cette interdépendance entre les gains en temps et en messages est très forte puisque nous obtenons un coefficient de régression de 89 %.

Intéressons nous au nombre de messages par seconde émis par chaque processeur pour l'algorithme à flots multi-informatifs. Pour cela, nous allons comparer ces résultats avec ceux de l'algorithme à flots de données.

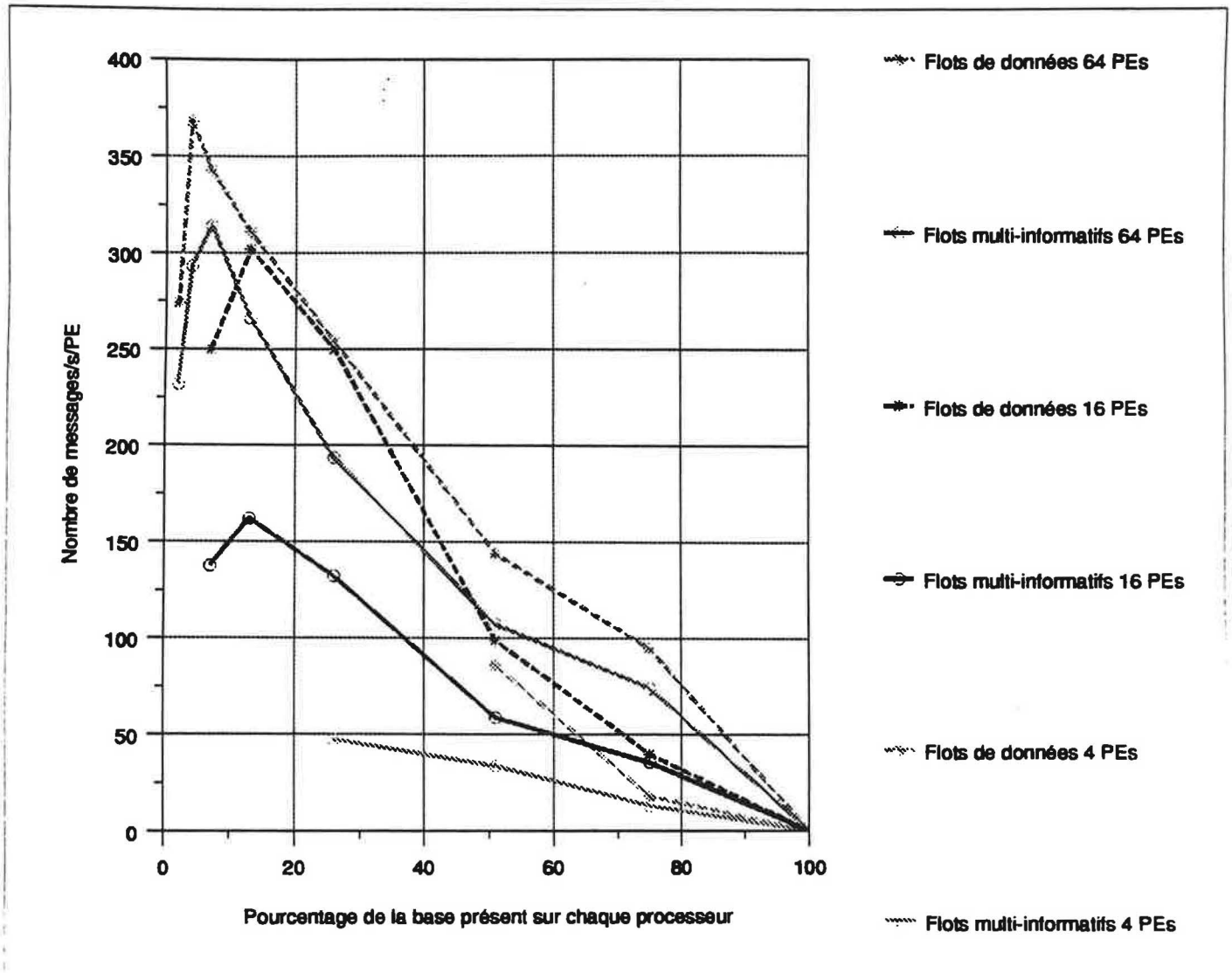

Figure 77 Comparaison du nombre de messages par seconde de l'algorithme à flots multi-informatifs et de l'algorithme à flots de données pour l'image M5

Il apparaît clairement que l'algorithme à flots multi-informatifs permet une réduction du flux de messages sur le réseau. La durée de saturation du réseau est donc réduite.

Avantages	Inconvénients
Comportement général ne dépendant pas de la scène traitée	A partir d'un certain seuil de cache le surcoût en temps de calcul devient très important
Permet de traiter des scènes de grande taille	Saturation quand la taille du cache est faible
Réduction importante des temps de calcul (-> facteur 3,5 de gain)	
Réduction importante du nombre de messages par seconde (-> 4,5 de gain)	

Table 9 Récapitulatif pour l'algorithme multi-informatif

4.3.2 Flots concurrents

Nous commençons par étudier les gains en temps obtenus par cet algorithme par rapport à celui basé sur les flots de données :

$$G_T = \frac{T_{flots\ de\ donnees}}{T_{flots\ concurrents}}$$

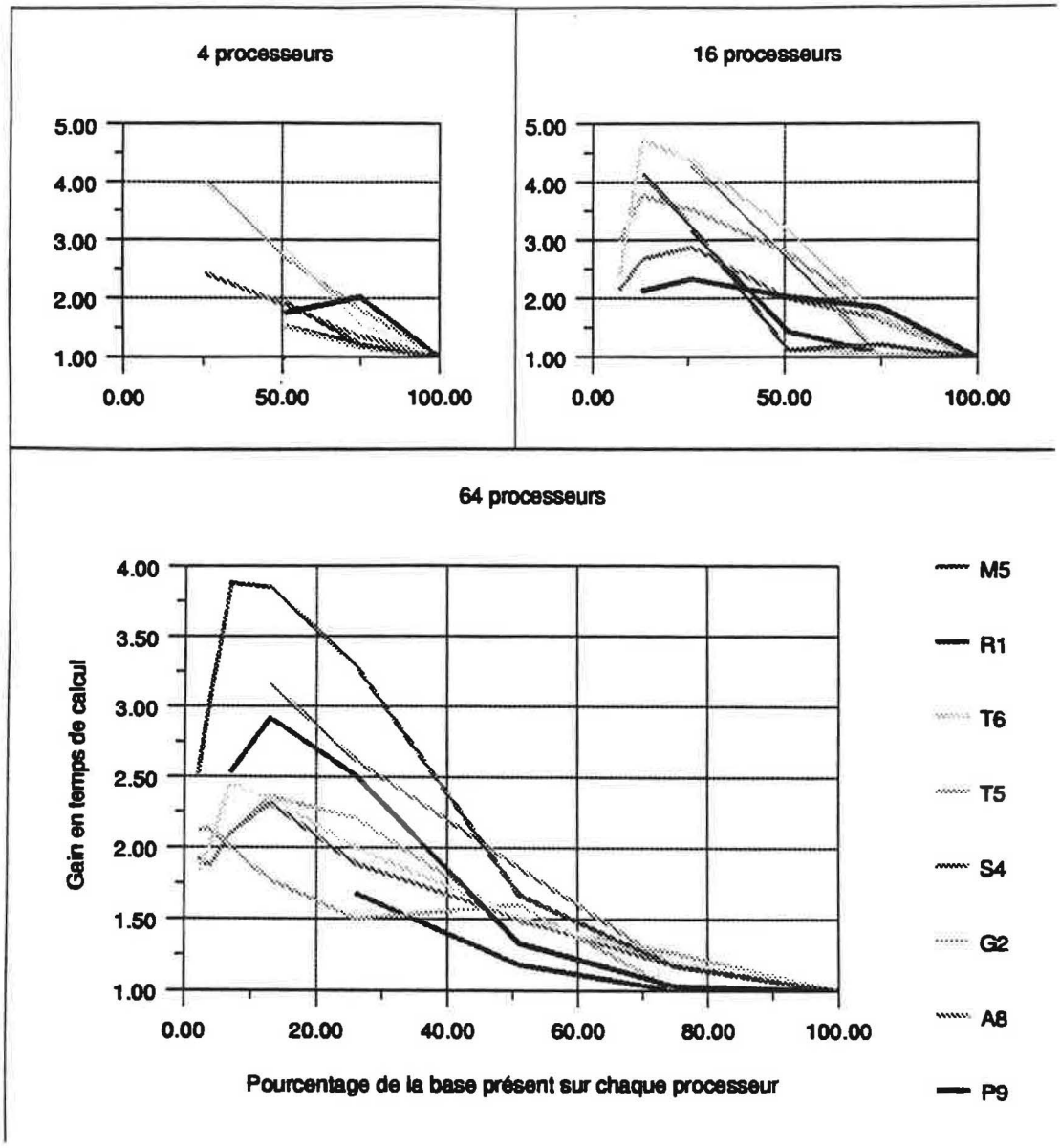

Figure 78 Gains en temps de l'algorithme à flots concurrents par rapport à l'algorithme à flots de données

Globalement, on retrouve des résultats proche de ceux vus précédemment. Toutefois les gains sont plus importants que pour l'algorithme à flots multi-informatifs : 2 à 4 par rapport à l'algorithme à flots de données. Le taux de distribution de la base de données influe de la même façon que dans l'algorithme multi-informatif.

Etudions maintenant le gain en nombre de messages qu'induit cette méthode :

$$G_{Msg} = \frac{Msg_{flots\ de\ donnees}}{Msg_{flots\ concurrents}}$$

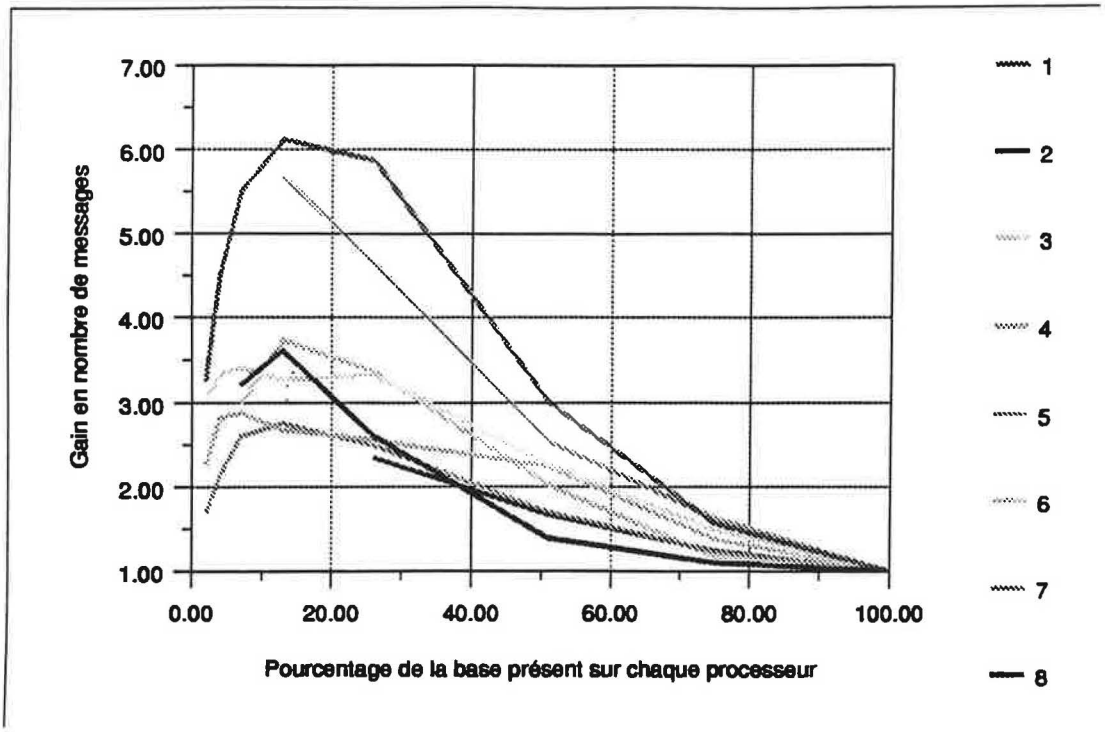

Figure 79 Gains en nombre de messages de l'algorithme à flots concurrents par rapport à l'algorithme à flots de données

On retrouve une diminution du nombre de communications d'un facteur compris entre 2 et 6.

L'interdépendance entre les gains en temps et en messages est également très forte puisque nous obtenons un coefficient de régression de 87 %.

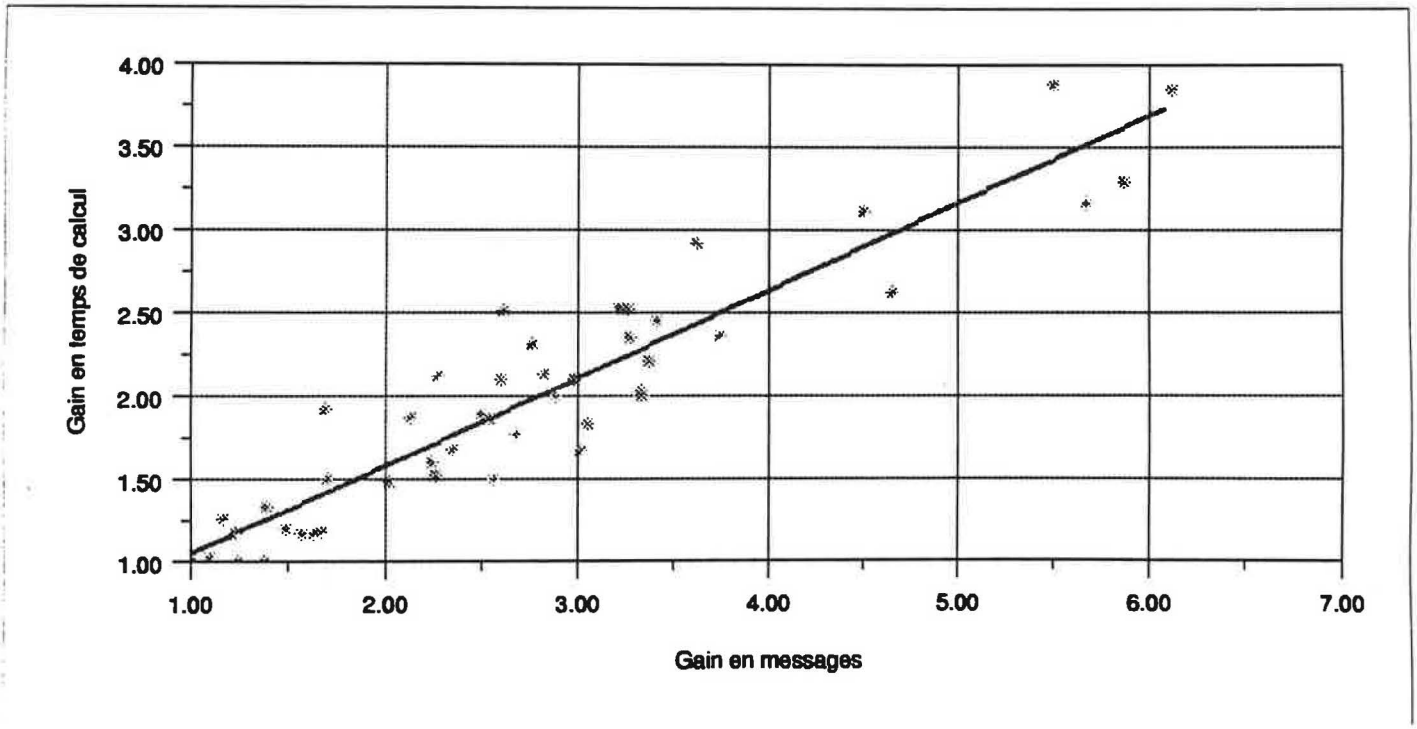

Figure 80 Relation entre gain en temps et gain en messages

Intéressons-nous au nombre de messages par seconde émis par chaque processeur pour l'algorithme à flots concurrents. Pour cela, nous allons comparer ces résultats avec ceux de l'algorithme à flots de données.

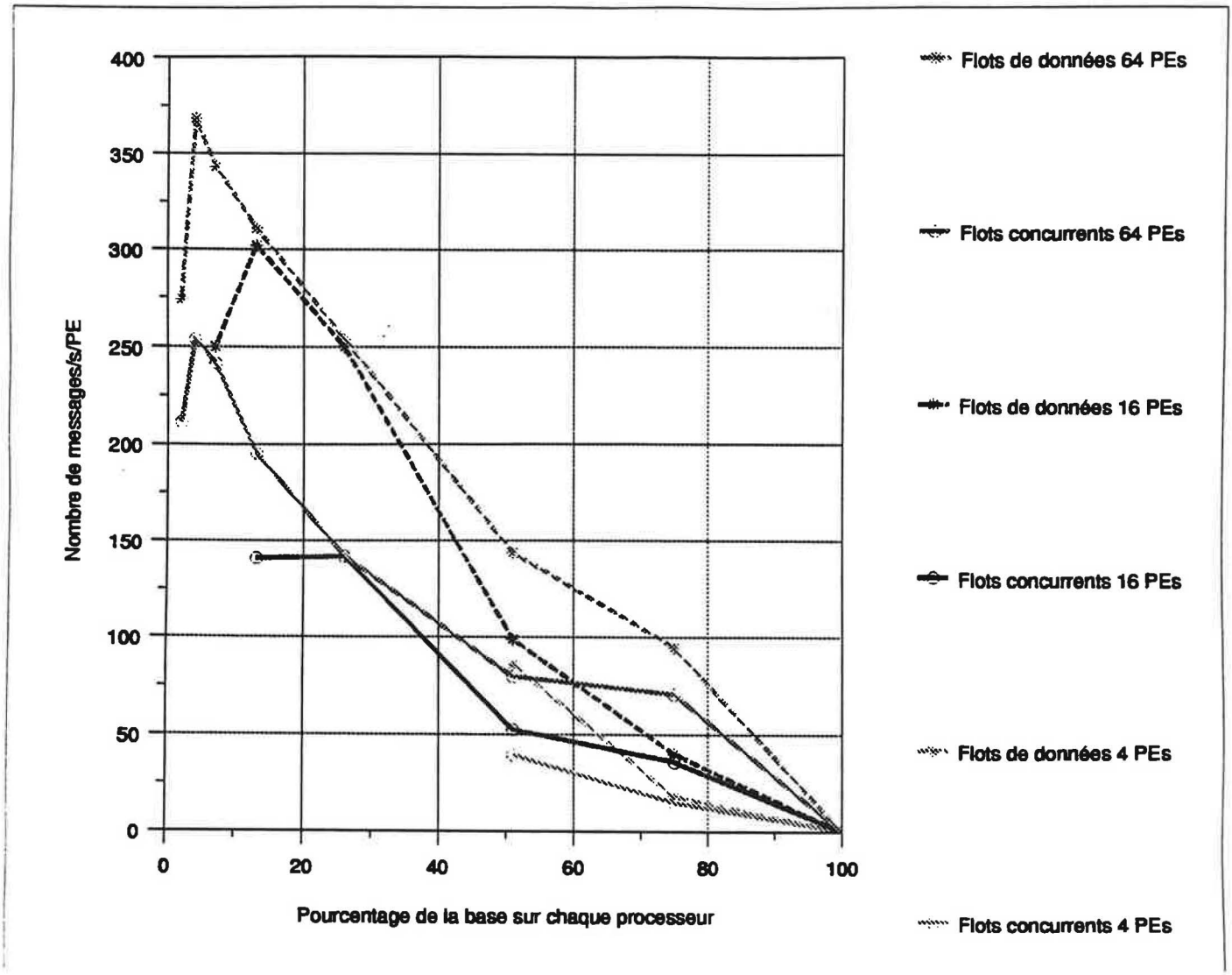

Figure 81 Comparaison du nombre de messages par seconde de l'algorithme à flots concurrents et de l'algorithme à flots de données pour l'image M5

Il apparaît clairement que l'algorithme à flots concurrents permet une réduction du flux de messages sur le réseau. La durée de saturation du réseau est donc réduite.

Nous allons étudier maintenant comment se décomposent les communications émises par chaque processeur. La figure suivante permet de comparer pour une architecture de 64 processeurs le pourcentage de messages demandant une donnée et le pourcentage de messages envoyant des calculs.

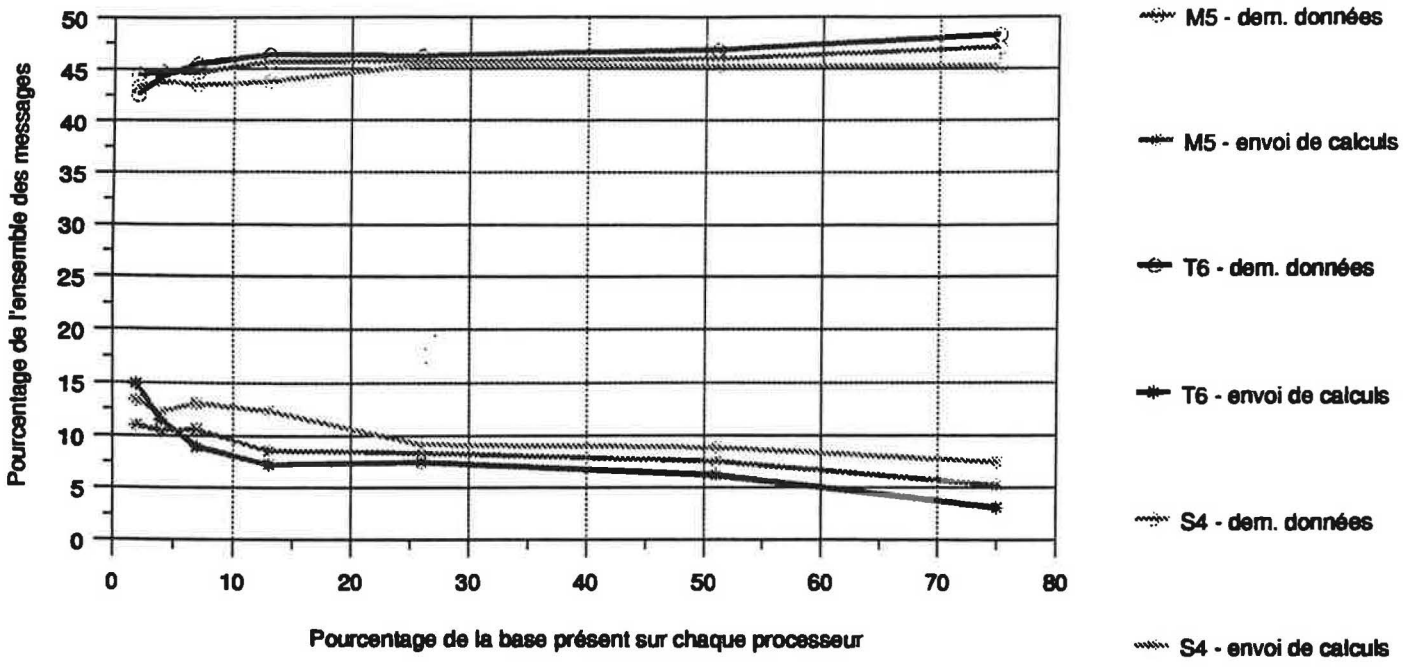

Figure 82 Comparaison du nombre de messages de chaque type pour l'algorithme à flots concurrents

Il apparaît qu'environ 45 % des messages sont des demandes de données (il y a donc également 45 % des messages qui sont des réponses à ces demandes de données) et 10 % des messages sont des envois de calcul, le pourcentage des autres types de messages est négligeable. Ces taux sont relativement stables quelles que soient l'image et la taille du cache.

On peut cependant noter que lorsque la base est très distribuée et qu'on arrive en zone de saturation de l'envoi de messages, la proportion de messages d'envoi de calculs augmente. Cela peut s'expliquer par le fait que le ralentissement de l'envoi de messages engendre un accroissement de la bufferisation des calculs non traités qui est favorable à l'envoi de calculs.

Le ratio entre les deux taux de messages $\alpha = \frac{flots\ de\ donnees}{flots\ de\ calculs}$ (ici α=4,5) dépend essentiellement des heuristiques de choix entre les deux types de flots, il pourrait donc être intéressant à l'avenir de modifier ces heuristiques afin de déterminer le ratio le plus performant.

Avantages	Inconvénients
Comportement général ne dépendant pas de la scène traitée	A partir d'un certain seuil de cache le surcoût en temps de calcul devient très important
Permet de traiter des scènes de grande taille	Saturation quand la taille du cache est faible
Réduction importante des temps de calcul (-> facteur 4 de gain)	
Réduction importante du nombre de messages par seconde (-> 6 de gain)	

Table 10 Récapitulatif pour l'algorithme concurrent

4.3.3 Comparaison entre les algorithmes à flots multi-informatifs et à flots concurrents

Maintenant que nous avons montré les avantages apportés par nos deux méthodes à flots mixtes, nous allons les comparer afin d'essayer de déterminer laquelle des deux est la plus performante.

La figure suivante exprime les gains de temps de l'algorithme à flots concurrentiels par rapport à l'algorithme multi-informatif.

$$G_T = \frac{T_{flots\ multi-informatifs}}{T_{flots\ concurrents}}$$

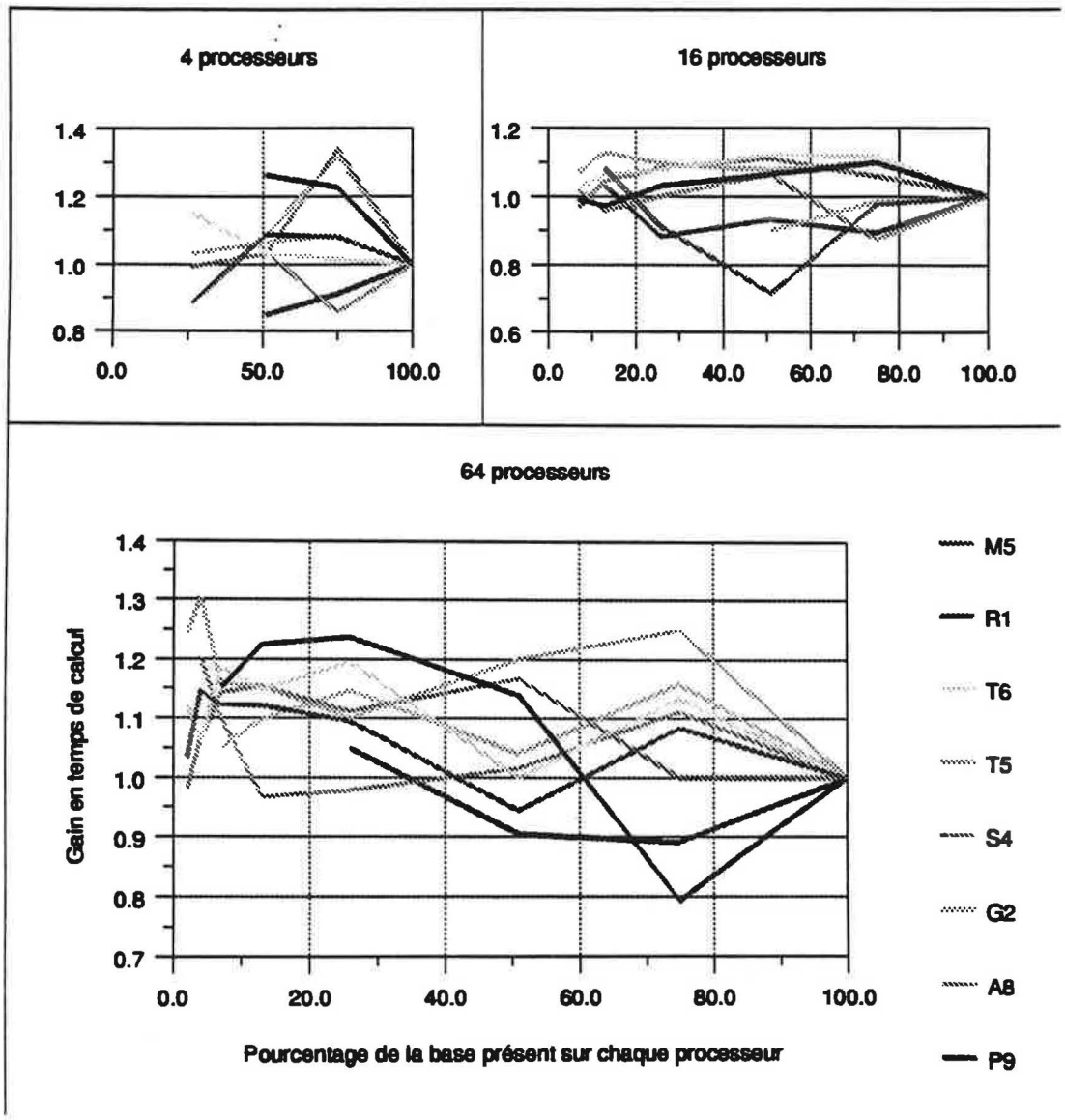

Figure 83 Gain en temps de l'algorithme à flots concurrents par rapport à l'algorithme à flots multi-informatifs

A partir de ces courbes, il est difficile de déterminer lequel des deux algorithmes est le meilleur. On peut dire toutefois que ces deux algorithmes sont de performance proche vu que les gains varient entre 0,8 et 1,3. Pour la configuration à 64 processeurs, il semble que l'algorithme à flots concurrentiels soit légèrement plus performant.

Les gains en temps de calcul n'étant pas suffisants pour apporter un verdict concluant à la supériorité d'un algorithme sur l'autre, étudions maintenant les gains en messages :

$$G_{Msg} = \frac{Msg_{flots\ multi-informatifs}}{Msg_{flots\ concurrents}}$$

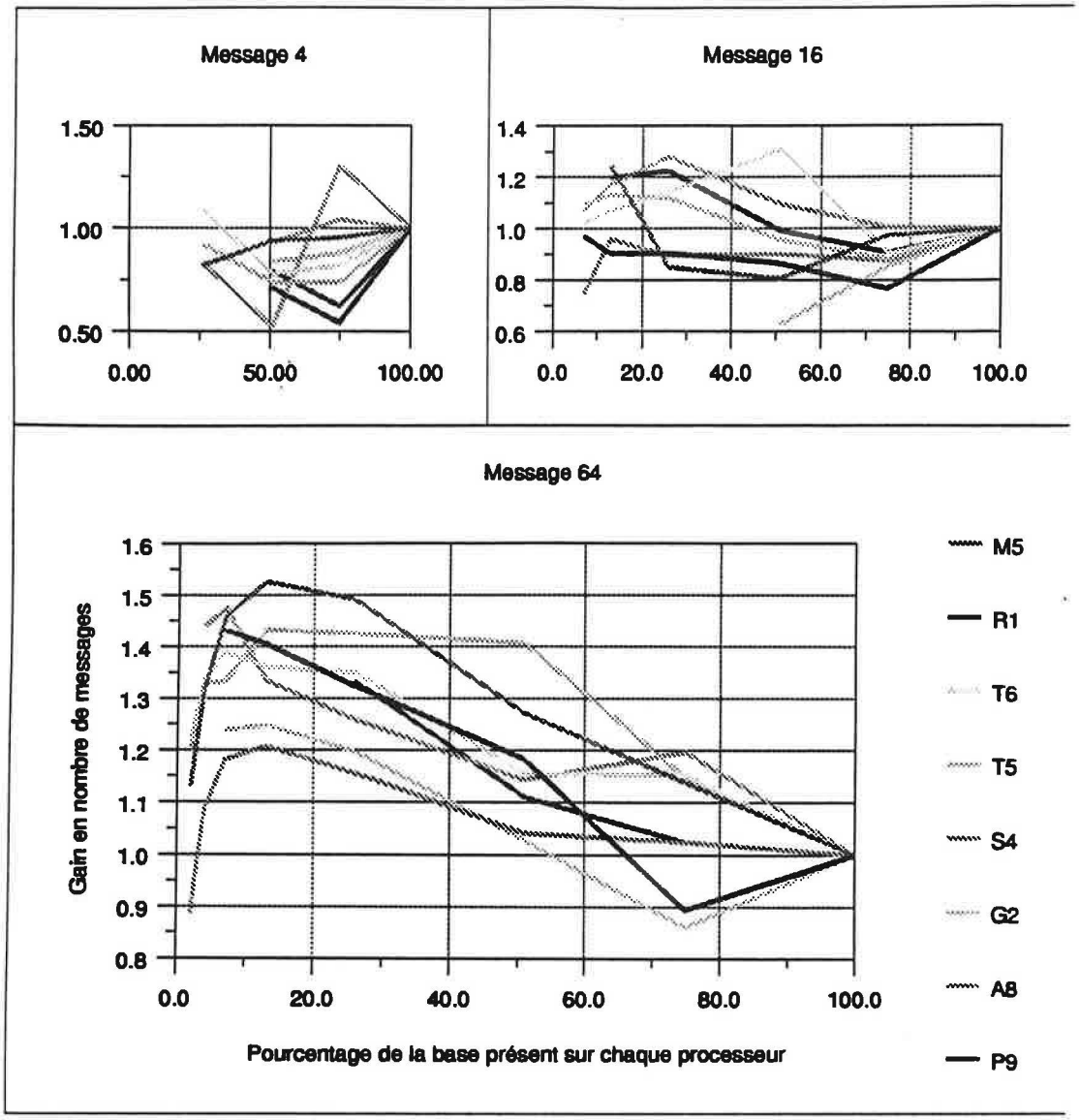

Figure 84 Gain en messages

Les courbes précédentes permettent de dégager une tendance : l'algorithme à flots concurrentiels est moins coûteux en messages quand le nombre de processeurs est important; par contre, l'algorithme à flots multi-informatifs semble plus économe quand le nombre de processeurs est faible.

Finalement, il apparaît que bien que les deux algorithmes à flots mixtes aient des comportements assez proches, l'algorithme à flots concurrents semble avoir l'avantage pour les architectures comportant de nombreux processeurs, en particulier grâce à la réduction du nombre de messages émis qu'il apporte. De plus, cet algorithme semble pouvoir être assez facilement perfectible par une étude précise du ratio α.

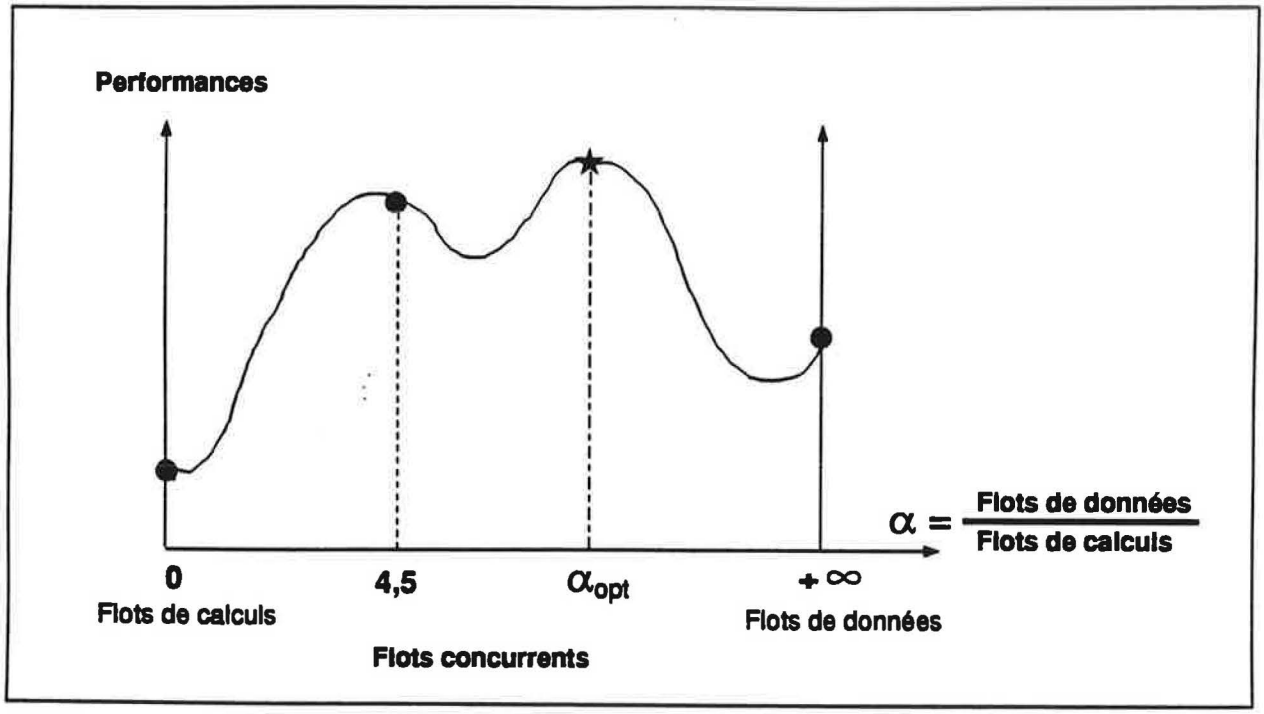

Figure 85 Etude des performances de l'algorithme à flots concurrents en fonction du rapport entre le nombre de messages d'envoi de données et le nombre de messages d'envoi de calculs.

4.3.4 Autres combinaisons envisageables

Notre logiciel propose un certain nombre de fonctionnalités :

1. Flots de calculs.
2. Flots de données.
3. Equilibrage dynamique par envoi de calculs.
4. Flots multi-informatifs.
5. Flots concurrents.

Ces possibilités peuvent se combiner entre elles. Ainsi, en plus des flots multi-informatifs et concurrents, on peut imaginer de nouvelles associations :

1. Flots multi-informatifs avec équilibrage dynamique par envoi de calculs.
2. Flots concurrents avec équilibrage dynamique par envoi de calculs.
3. Flots concurrents multi-informatifs.
4. Flots concurrents multi-informatifs avec équilibrage dynamique par envoi de calculs.

Si toutes ces combinaisons ont été testées, on peut noter que les résultats obtenus sont tous inférieurs à ceux des algorithmes à flots concurrents et à flots multi-informatifs utilisés seuls.

On peut noter qu'en effet, ces deux algorithmes possédant de façon intrinsèque un équilibrage de charges dynamique, l'ajout de l'équilibrage dynamique par envoi de calculs apparaît comme redondant et son coût de mise en oeuvre a pour effet de ralentir l'exécution du programme.

Par contre, l'utilisation de flots concurrents multi-informatifs pourrait être une voie prometteuse si les heuristiques de choix étaient déterminées de façon très fine, ce que nous n'avons pas cherché à faire. Cependant on peut imaginer que la précision demandée induise une grande dépendance à la scène ou à l'application traitée.

4.4 Avantages de l'asynchronisme

Nous avons choisi de réaliser un noyau parallèle asynchrone afin de limiter les coûts engendrés par les communications inter-processeurs. Afin de déterminer l'efficacité de nos communications asynchrones, nous allons tester si elles sont réellement indépendantes de la bibliothèque d'envoi de messages, du type de réseau utilisé et enfin de la taille des messages échangés.

4.4.1 Indépendance de la bibliothèque d'envoi de messages

Les machines CRAY que nous utilisons disposent de deux bibliothèques d'envoi de messages : PVM et MPI. Nous allons comparer les résultats obtenus avec ces deux bibliothèques sur notre algorithme concurrent afin d'étudier leur influence sur le déroulement de notre code.

Sur la figure suivante, nous exprimons les écarts en temps entre les résultats obtenus avec les bibliothèques PVM et MPI en fonction de la distribution de la base sur 64 processeurs.

$$E_T = \frac{T_{PVM} - T_{MPI}}{\left(\frac{T_{PVM} + T_{MPI}}{2}\right)}$$

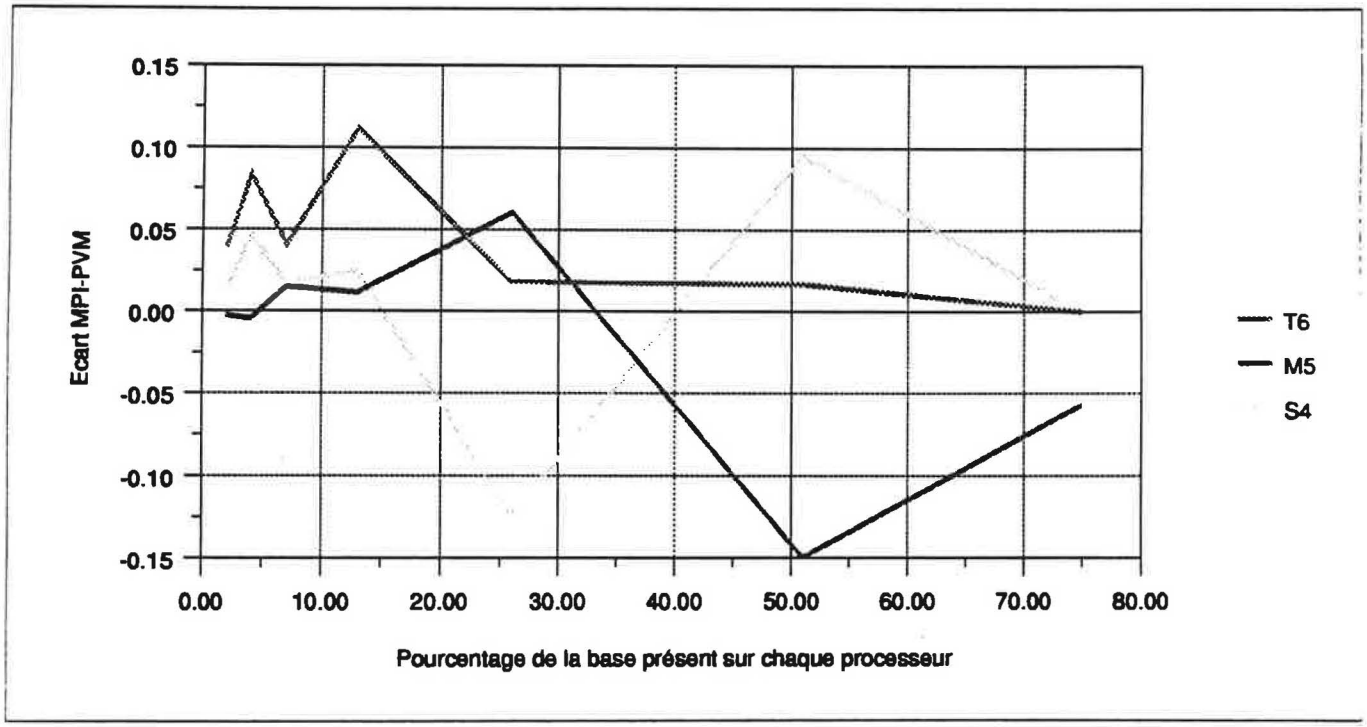

Figure 86 Ecart entre les temps de calculs avec MPI et PVM pour 64 processeurs

Les résultats sur les différentes images permettent d'abord de voir que l'écart ne dépasse pas (± 15 %). Ensuite aucune tendance ne peut être dégagée pouvant précisément montrer la plus grande efficacité d'une bibliothèque ou d'une autre. De plus, l'étude du nombre de messages envoyés ne révèle aucune différence significative lors de l'exécution de notre code avec MPI ou PVM.

Finalement, on peut en déduire que ces deux bibliothèques donnent des résultats très comparables. Le comportement de notre algorithme asynchrone est donc indépendant de la bibliothèque d'envoi de messages utilisée.

4.4.2 Indépendance du type de réseau

Nous allons dans cette section comparer les résultats obtenus sur différentes architectures matérielles. Nous commençons par confronter les performances sur CRAY, puis celles obtenues sur réseau de stations. Les caractéristiques des machines utilisées sont présentées en annexe A.

Pendant notre travail, nous avons pu utiliser deux machines massivement parallèles : les CRAY T3D et T3E. Celles-ci disposent de processeurs et de réseaux de communication de conception et de performances différentes :

	Microprocesseur	Réseau de communication	Puissance par PE
CRAY T3D	Alpha (21064) 150 MHz 2 instructions/cycle	Grille 3D torique (2PEs par noeud) Débit max. : 300 Mo/s	150 MFlops crête
CRAY T3E	Alpha (21164) 300 MHz 4 instructions/cycle	Grille 3D torique (1PE par noeud) Débit max. : 600 Mo/s	600 MFlops crête

Table 11 Caractéristiques des CRAY T3D et T3E

Comparons les résultats obtenus sur ces deux types de machines pour le calcul de différentes images. Nous présentons des résultats sur des architectures comportant 64 processeurs.

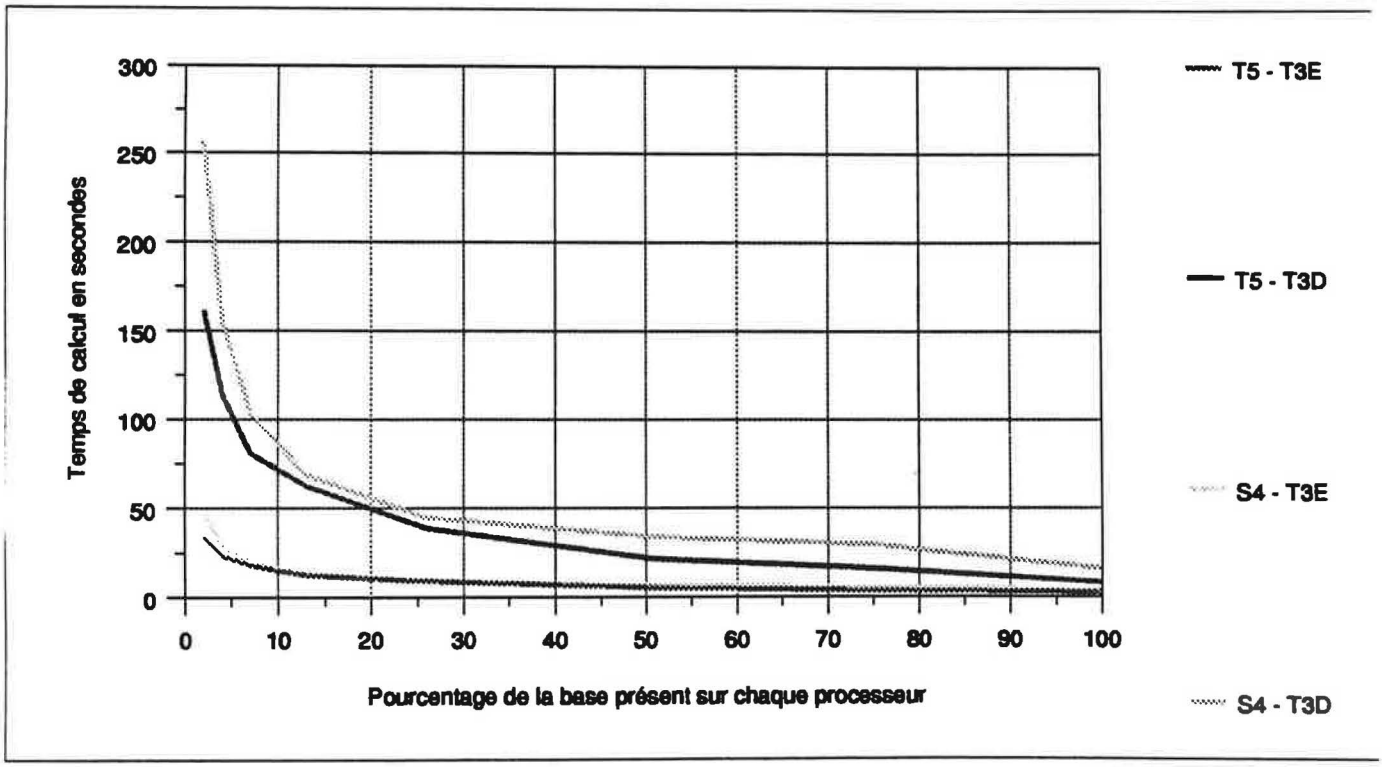

Figure 87 Comparaison des temps de calcul obtenus sur CRAY T3D et T3E avec 64 processeurs

Il apparaît que la puissance du CRAY T3E est bien supérieure à celle du CRAY T3D pour un même nombre de processeurs. On trouve un rapport proche de 5 entre les temps de calcul de ces machines. Ceci provient d'une part de l'augmentation de la puissance de la machine et d'autre part de l'amélioration des compilateurs proposés.

Afin de pouvoir comparer plus finement le comportement de notre algorithme asynchrone sur ces deux architectures, nous allons représenter les accélérations obtenues.

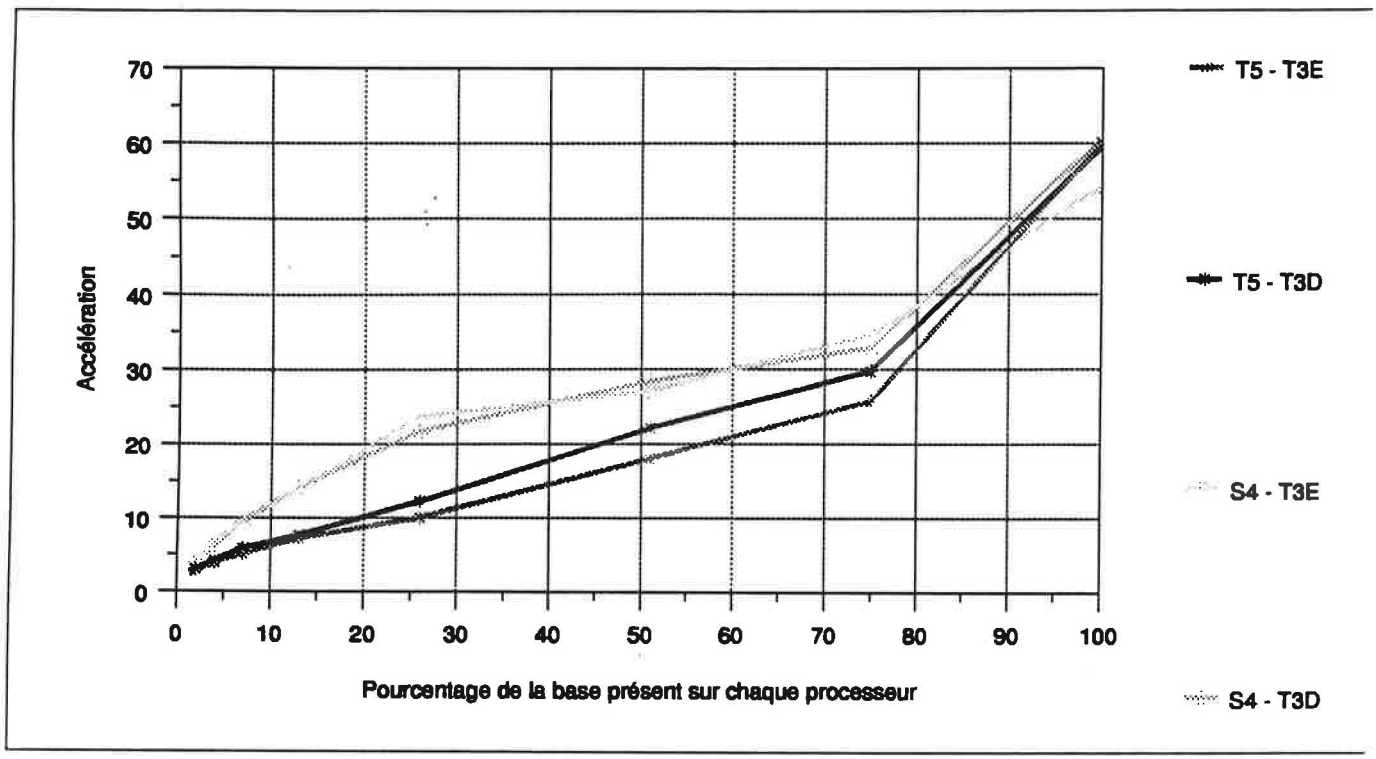

Figure 88 Accélération sur CRAY T3D et T3E avec 64 processeurs

Les accélérations obtenues sur T3D et T3E sont très proches quelle que soit l'image. Les écarts ne dépassent pas 20 % et aucune règle ne se dégage quant à un comportement différent de notre algorithme sur ces deux machines.

Comparons maintenant le nombre de messages envoyés par chacune des machines pour vérifier que l'algorithme se déroule bien de la même manière.

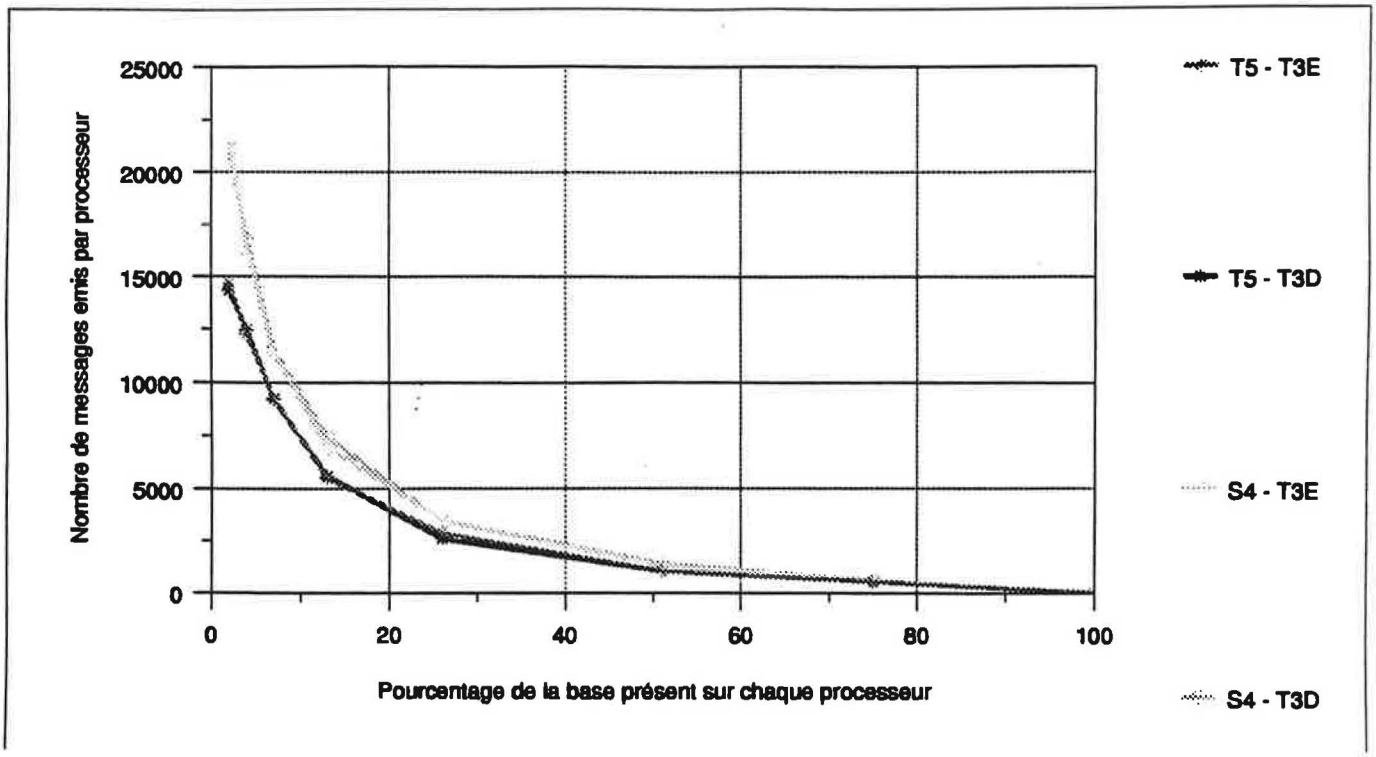

Figure 89 Nombre de messages émis sur CRAY T3D et T3E avec 64 processeurs

Les écarts sont également très faibles (inférieur à 10 %).

Il semble bien que le comportement de notre algorithme asynchrone soit bien identique quelle que soit la machine CRAY utilisée.

Comparons maintenant les résultats obtenus sur réseau de stations de travail homogènes avec ceux provenant de l'utilisation des CRAY T3D et T3E.

Nous étudions les résultats obtenus sur des architectures comprenant 4 et 8 processeurs car nous ne pouvons pas obtenir de configuration homogène comportant plus de machines.

Sur la figure suivante, nous comparons les accélérations obtenues avec des Sparc 5, Ultra Sparc, T3E et T3D pour le calcul de la scène S4.

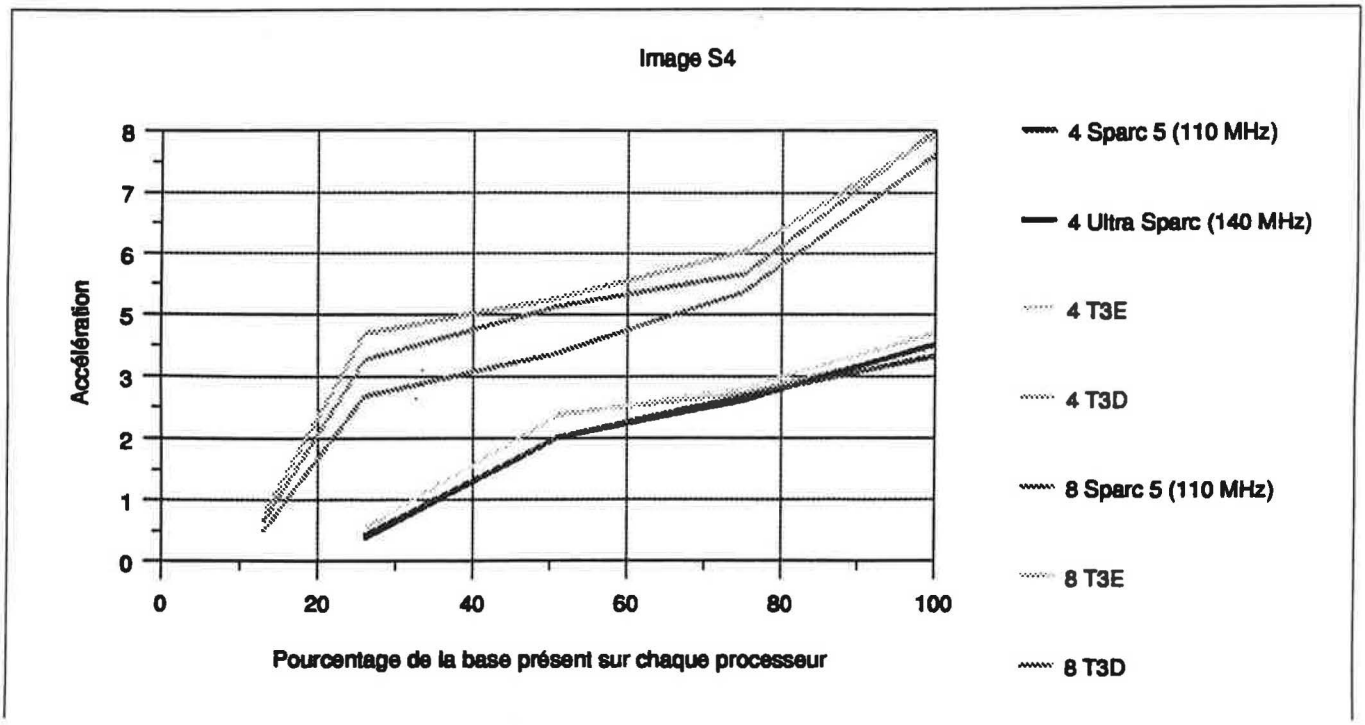

Figure 90 Comparaison de l'efficacité avec 4 et 8 processeurs sur des architectures variées (image S4)

Les résultats sont très proches. Il n'y a pas de différences significatives entre les performances provenant de l'utilisation de réseaux rapides (T3D et T3E) et celle issue de réseaux lents (Sparc 5 et Ultra Sparc). Cette conclusion se retrouve sur l'étude d'autres images.

Afin de vérifier que le comportement de l'algorithme est bien le même sur ces deux types de réseaux, nous présentons le nombre de messages émis lors de l'exécution par chacune des architectures précédentes.

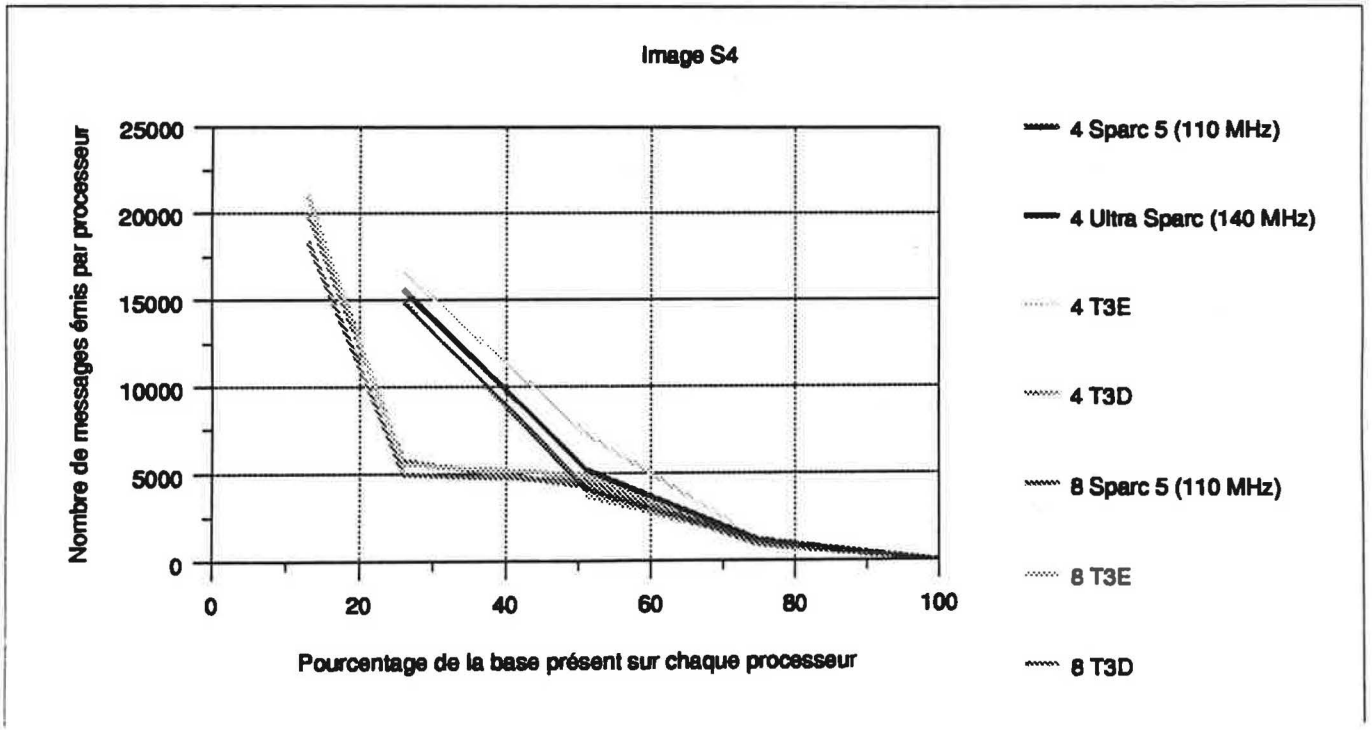

Figure 91 Comparaison du nombre de messages émis avec 4 et 8 processeurs sur des architectures variées (image S4)

Le nombre de messages émis est bien indépendant de l'architecture utilisée. Le déroulement de l'algorithme est donc identique sur l'ensemble des configurations testées.

Les résultats obtenus montrent bien l'intérêt de l'utilisation de communications asynchrones, puisque les performances sont indépendantes du réseau d'interconnexion utilisé.

Architecture	Réseau de communication
CRAY T3D	Grille 3D torique (2PEs par noeud) Débit max. : 300 Mo/s
CRAY T3E	Grille 3D torique (1PE par noeud) Débit max. : 600 Mo/s
Réseau de stations	Brin éthernet Débit max. : 10 Mo/s

Table 12 Réseaux d'interconnexion utilisés

Les tests réalisés sur réseau de stations ne comportaient qu'un nombre de machines peu élevé. Aussi nous ne pouvons pas en déduire de résultats sur des architectures beaucoup plus importantes. Toutefois on peut penser que la saturation du réseau risque d'apparaître plus rapidement sur ce type d'architectures que sur CRAY.

4.4.3 Indépendance de la taille des objets

L'intérêt de réaliser un noyau parallèle asynchrone est d'assurer également une certaine indépendance du temps de calcul vis-à-vis de la taille des messages échangés.

Afin d'étudier le comportement de l'algorithme en fonction de la taille des messages, nous avons augmenté la taille des objets de la base de données en compliquant les textures qui leur sont associées.

Nous présentons, dans la figure suivante, le temps de calcul en utilisant notre algorithme concurrent pour différentes images en fonction de la taille de chaque objet de la base de données. L'architecture comprend 64 processeurs et chacun d'eux possède 4% de la base de données.

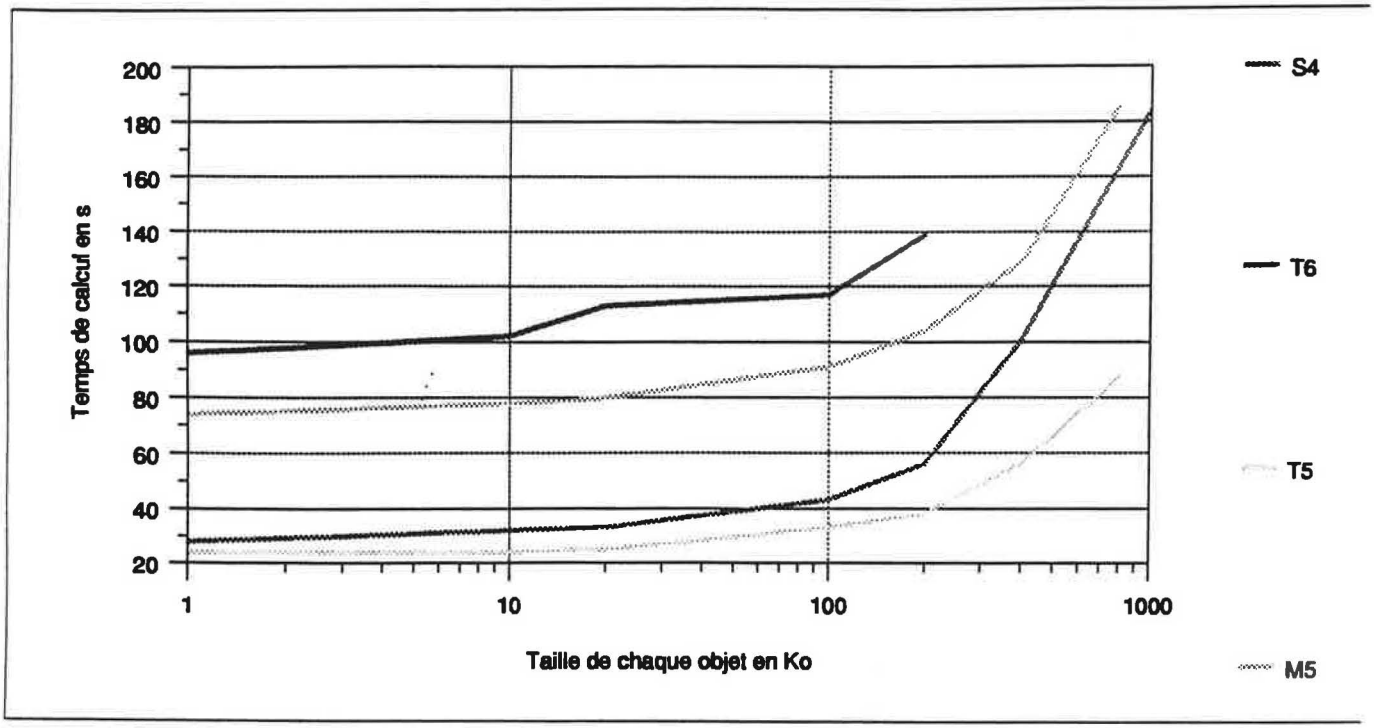

Figure 92 Temps de calcul pour 64 processeurs en fonction de l'augmentation de la taille des objets

Il apparaît que l'augmentation de la taille des objets a peu d'influence sur les résultats tant que ces objets ne dépassent pas une taille de 200 Ko chacun. Au-delà, les performances se dégradent fortement, la taille excessive des messages engendrant une saturation du réseau.

Dans le tableau suivant, nous donnons les tailles maximum des scènes testées.

Nom de la scène	Taille maximale des objets	Taille de la scène
Mountain_5	800 Ko	1600 Mo
SphereFlake_4	1000 Ko	821 Mo
Tetrahedron_5	800 Ko	800 Mo
Tetrahedron_6	200 Ko	800 Mo

Table 13 Tailles de scènes comportant des objets de grande taille

On peut remarquer que grâce à cette distribution de la base de données (4 %) sur 64 processeurs, une scène qui nécessite 1,6 Go pour la décrire a pu être calculée (la mémoire disponible est de 8 Go avec 64 processeurs du CRAY T3E).

4.4.4 Bilan

L'ensemble des résultats présentés montre bien l'intérêt de la réalisation d'un noyau parallèle asynchrone. L'asynchronisme permettant le recouvrement des temps de communications, il est possible d'implémenter une parallélisation relativement portable. En effet ses performances ne sont liées ni à la bibliothèque d'envoi de messages utilisée ni au type de réseau reliant employé. Elles dépendent uniquement de la puissance des processeurs disponibles et de l'application à traiter.

4.5 Comportement sur réseau de machines hétérogènes

Dans la section précédente, nous avons montré que notre algorithme avait un comportement équivalent quelque soit le réseau de machines homogènes. Bien que notre étude ne se soit pas attardée sur l'utilisation du noyau parallèle sur réseau de machines hétérogènes, nous allons maintenant nous intéresser à ce type d'exécution.

Pour cela nous allons tout d'abord présenter, pour chaque type de machine (voir annexe A) qui sera utilisée dans les architectures hétérogènes testées, le temps de calcul nécessaire en séquentiel pour obtenir l'image T5. Nous utilisons deux type de stations Sparc 5 et deux types de stations UltraSparc, les performances sur CRAY T3E nous servent de référence.

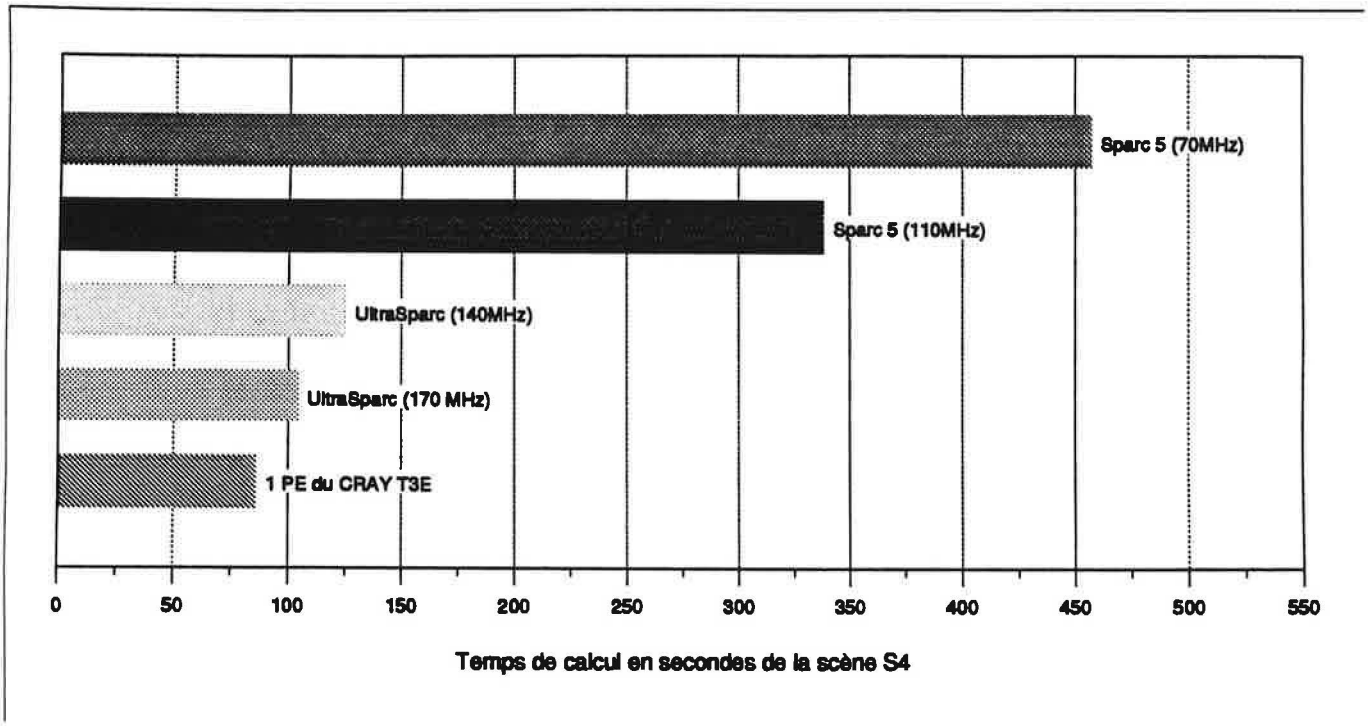

Figure 93 Temps de calcul de la scène S4 pour différents types de processeur

Deux groupes de machines relativement homogènes apparaissent :

1. Les Sparc 5, leur différence de puissance est de 30 %.
2. Les UltraSparc, leur différence de puissance est de 18 %.

Par contre, les puissances des Sparc 5 et des UltraSparc sont très distinctes.

A présent nous allons définir différentes architectures comportant toutes 8 processeurs, pour chacune d'elles nous calculons la puissance cpu théorique qu'elle représente en fonction de la puissance d'un processeur du T3E.

Une machine comprenant n_A processeurs de type A, n_B processeurs de type B ... possède une puissance *puissance* pouvant s'exprimer par :

$$puissance = T_{SeqT3E} * \left(\frac{n_A}{T_{SeqA}} + \frac{n_B}{T_{SeqB}} + \ldots\right)$$

où

1. T_{SeqT3E} est le temps nécessaire à un processeur du T3E pour calculer l'image.

2. T_x est le temps nécessaire à un processeur de type x pour calculer l'image.
3. n_x est le nombre de processeurs de type x dans la machine.

Architecture	Puissance cpu en nombre de processeurs CRAY T3E
a : CRAY T3E 8 processeurs	8
b : 8 Sparc 5 (110 MHz)	2
c : 4 Sparc 5 (110MHz) + 4 Sparc 5 (70 MHz)	1,8
d : 4 Sparc 5 (110MHz) + 4 UltraSparc (140 MHz)	3,4

Table 14 Définitions des architectures à 8 processeurs testées

Les architectures (a) et (b) sont composées de processeurs homogènes et nous servent de témoin. L'architecture (c) comprend deux types de processeurs aux caractéristiques proches. Enfin, l'architecture (d) est très hétérogène puisqu'elle est formée de Sparc 5 et d'UltraSparc.

La figure suivante présente les résultats obtenus par ces différentes architectures. Elle représente l'efficacité en fonction de la répartition de la base de données. L'efficacité *Eff* pour les architectures hétérogènes est calculée de la façon suivante :

$$Eff = \frac{T_{SeqT3E}}{T * puissance}$$

où

1. T est le temps nécessaire à la machine pour calculer l'image.
2. *puissance* est la puissance de la machine en cpu T3E.

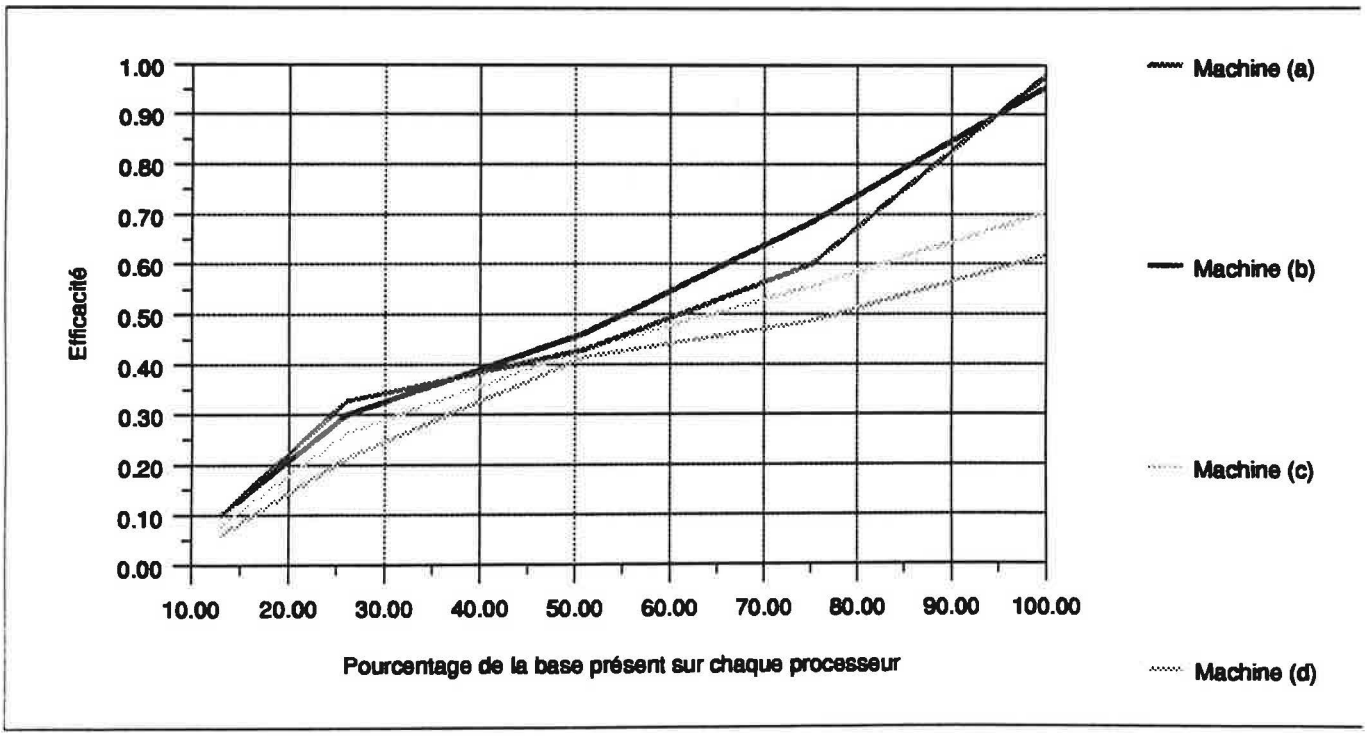

Figure 94 Efficacité des différentes machines

Les machines (a) er (b), qui sont homogènes, ont des efficacités très proches. Par contre les efficacités des machines hétérogènes (c) et (d) sont beaucoup moins bonnes. On peut noter que c'est pour la machine la plus hétérogène, (d), que les performances sont les plus faibles.

Sur la figure suivante, nous comparons le nombre de messages émis par processeur.

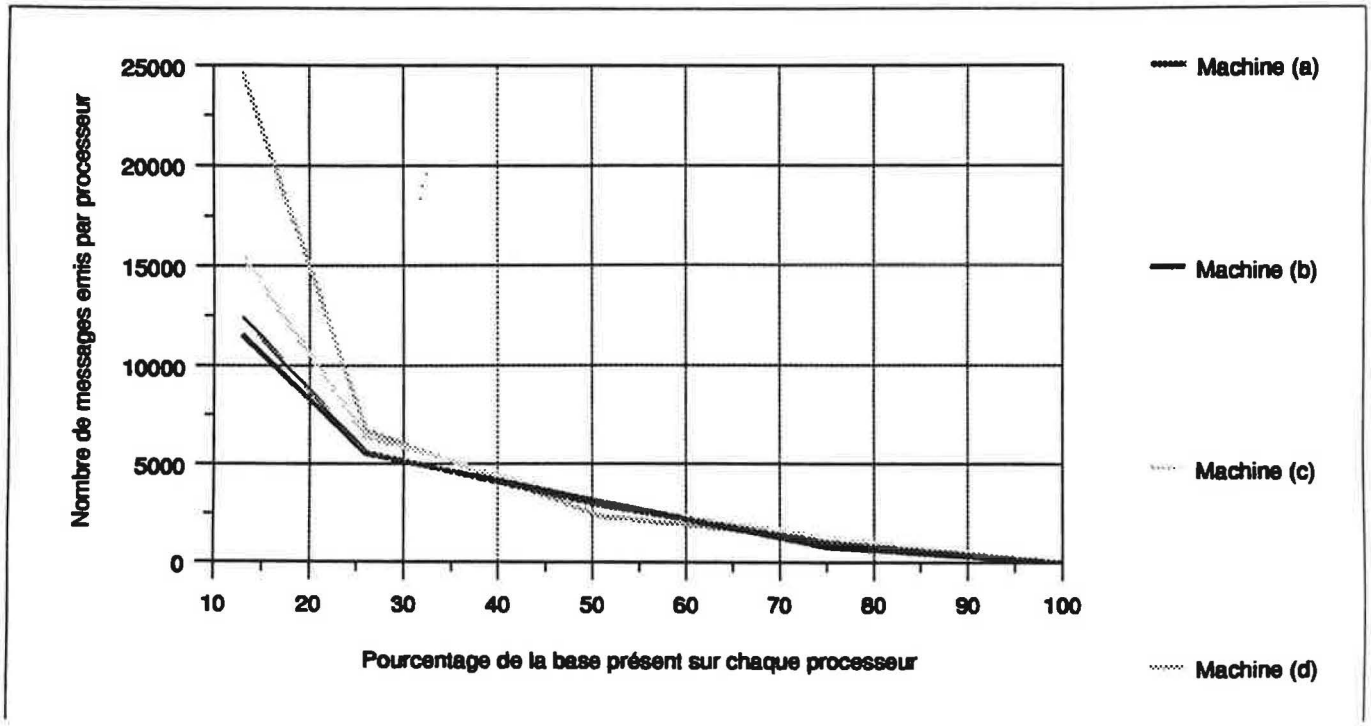

Figure 95 Nombre de messages émis sur les différentes machines

De nouveau, il apparaît que plus la machine est hétérogène, plus le nombre de messages émis est important.

L'analyse de ces deux figures révèle tout d'abord que le déséquilibrage des charges induit une augmentation des communications. Cependant ce rééquilibrage ne semble pas suffisant comme nous le prouve la baisse des performances sur machines hétérogènes.

En fait l'équibrage de nos algorithmes est basé sur la connaissance de la charge des processeurs de la machine. Cette charge est exprimée par le nombre de calculs restant à effectuer. Or sur une machine hétérogène, la charge devrait également dépendre de la puissance cpu des différents processeurs.

Aussi nous pensons qu'un moyen efficace et simple d'améliorer le comportement de notre noyau parallèle sur réseau de stations hétérogènes serait d'exprimer la charge d'un processeur par le ratio du nombre de calculs restant à effectuer par la puissance du processeur.

La charge d'un processeur P peut donc s'exprimer par :

$$\text{charge de P} = \frac{\text{nombre de calculs}}{\text{puissance de P}}$$

La puissance des différents processeurs d'une machine pourrait soit être lue dans un fichier de configuration, soit être calculée lors de la phase de pré-calcul.

4.6 Conclusion

Dans ce chapitre, nous avons présenté les résultats obtenus sur les différents algorithmes parallèles que nous avons proposés. Le choix d'utiliser une parallélisation à flots mixtes s'est avéré particulièrement performant puisque les temps de calcul d'images et le nombre de messages envoyés ont été réduits par des facteurs importants par rapport aux algorithmes classiques. En outre la conception de stratégies sur une architecture logique décentralisée apparait tout à fait adaptée aux machines massivement parallèles comme le prouve le bon comportement obtenu sur 128 processeurs.

De plus l'utilisation de communications asynchrones nous a permis, en plus des gains en temps, d'assurer une grande portabilité de notre noyau parallèle. Nous avons pu aussi tester notre travail sur des architectures composées de stations de travail homogènes. Les résultats exposés pour des configurations allant jusqu'à 8 processeurs montrent que les performances de nos algorithmes ne sont pas liées à l'utilisation de réseaux d'interconnexion à très hauts débits comme ceux des calculateurs CRAY. Il apparaît donc qu'un réseau de stations de travail peut être une excellente machine parallèle pour le type d'applications que nous avons étudiées.

Enfin nous avons expérimenté nos algorithmes sur un réseau de stations de travail hétérogènes. Bien que les résultats obtenus restent très honorables, il est nécessaire de revoir l'expression de la charge de travail sur ce type de configuration afin d'obtenir de meilleures performances. Une solution allant dans ce sens a été proposée.

Conclusion générale

Dans cette thèse nous avons étudié, puis proposé des algorithmes permettant l'accélération via la parallélisation de l'algorithme de lancer de rayon sur des machines parallèles à mémoires distribuées. Nous nous sommes efforcés de réaliser un noyau parallèle pouvant être utilisé également par des applications dont la problématique est proche de celle du lancer de rayon. De plus, nous avons développé un code pouvant être employé aussi bien par des calculateurs massivement parallèles que par des réseaux de stations de travail.

Dans le premier chapitre, nous avons détaillé l'algorithme du lancer de rayon et étudié les accélérations séquentielles proposées dans le domaine. Il est apparu que des techniques de subdivisions de l'espace et de volumes englobants permettent une amélioration notable des performances. Toutefois, le temps de calcul nécessaire pour l'obtention d'une image demeure très important et peut être de plusieurs heures. Seul l'utilisation de machines parallèles semble pouvoir assurer des temps de calcul plus raisonnable.

Au sein de notre deuxième chapitre, nous avons passé en revue les différentes architectures parallèles. Nous nous sommes plus particulièrement intéressés aux machines parallèles à mémoires distribuées qui permettent, en plus de l'accélération de l'algorithme, une augmentation conséquente de la mémoire disponible. Les algorithmes performants pour ce type de machine peuvent être regroupés en trois catégories. La première repose sur une simple partition des calculs de l'image, mais nécessite la duplication de la scène à représenter sur chaque processeur. Les deux autres méthodes autorisent la distribution des données sur l'ensemble des processeurs de la machine. L'une, à flots de rayons, apporte comme réponse à l'absence de certains objets, la réalisation des calculs par plusieurs processeurs; l'autre, à flots de données, va chercher les objets manquants dans la mémoire des processeurs qui les possèdent.

Afin de déterminer quelle méthode répond à nos besoins, nous avons présenté nos contraintes dans le troisième chapitre. Devant traiter des scènes de grande taille, l'algorithme sans flot n'a pas pu être retenu. Par contre, il nous a fallu, à l'aide d'une modélisation, comparer les algorithmes à flots de rayons et de calculs afin de déterminer leurs domaines d'efficacité. Il est apparu que chacun d'eux pouvait, en fonction de la mémoire et du nombre de processeurs disponibles, être le plus performant. Nous avons alors proposé des algorithmes de parallélisation à flots mixtes - pouvant échanger des rayons et des données - afin d'essayer de tirer parti des avantages des deux algorithmes en compétition.

D'autre part, notre cahier des charges nous imposait un bon comportement aussi bien sur des machines massivement parallèles que sur des réseaux de stations de travail. Il nous a donc fallu premièrement concevoir une architecture logique non hiérarchisée, puis établir des communications asynchrones. Enfin, nous avons cherché à réaliser un noyau parallèle aussi générique que possible afin qu'il puisse être utilisé pour d'autres applications comme la radiosité et le suivi de particules.

Dans le quatrième chapitre, nous avons montré, en utilisant l'algorithme du lancer de rayon, que les stratégies que nous avons proposées permettent une amélioration très sensible des temps de calcul ainsi que du nombre de messages échangés par rapport aux algorithmes classiques de parallélisation. De plus, leurs comportements demeurent intéressants sur les machines massivement parallèles (128 processeurs) et sur des configurations composées de stations de travail homogènes.

Ces résultats très encourageants sur l'application "lancer de rayon" demandent à être confirmés pour d'autres utilisations. Une implémentation de l'application "suivi de particules" a été commencée, nous souhaitons qu'elle arrive à son terme. Il sera alors possible d'évaluer la polyvalence de notre noyau parallèle. D'autre part nous avons montré que, grâce à l'emploi de communications asynchrones, un réseau de stations de travail peut constituer une machine parallèle performante. Cependant l'utilisation de stations de travail hétérogènes a permis de mettre en lumière certaines insuffisances de nos algorithmes quant à l'expression de la charge pour ce type de configuration. Une méthode de résolution de ce problème a été proposée, il faudra l'implémenter et la tester afin d'assurer de bonnes performances quelque soit le type de réseau utilisé.

Annexe A
Caractéristiques techniques des machines employées

A.1 CRAY T3D

A.1.1 Configuration matérielle

Modèle : CRAY T3D 128,

Le CRAY T3D est une machine qui possède 128 noeuds et est reliée à un frontal, un CRAY Y-MP.

Architecture :

1. SPMD à mémoire distribuée.
2. Interconnexion par un réseau 3D toroïdal : le débit maximum de chaque lien est de 300 Mo/s.

Caractéristiques d'un noeud :

1. 2 processeurs Alpha (21064) de DEC :
   a. Fréquence d'horloge 150 MHz
   b. 2 instructions par cycle : 1 entière et 1 flottante (150 MFlops crête).
   c. Registres 64 bits.
2. 2 x 64 Mo de RAM.
3. Une interface réseau.

A.1.2 Configuration logicielle

Environnement de programmation :

1. Système d'exploitation : UNICOS MAX Version 1.3.0.2.
2. Compilateur Fortran 77 : CF77 Version 6.2.2.2.
3. Compilateur Fortran 90 : CF90 Version 0.1.3.0.
4. Compilateur C++ : CC Version 1.0.3.4.
5. Compilateur C : cc Version 4.0.4.6.
6. Editeur de liens : mppldr et mppld.
7. Debogueur : Totalview 2.0.0.4.
8. Bibliothèques d'échanges de messages :
   a. PVM
   b. MPI (Edinburgh)
   c. MPI (mpich)
   d. Shmem

A.2 CRAY T3E

A.2.1 Configuration matérielle

Modèle : CRAY T3E LC136/128-128/8-256.

Le CRAY T3E dispose de 152 processeurs répartis en 150 processeurs utilisateur (128 pour les application + 22 pour les commandes dont 2 redondants) et 2 processeurs support (2 pour le système).

Classification :

1. PE application : applications parallèles uniquement.
2. PE commande : commandes, shell, compilation et codes mono-processeur.
3. PE système : serveurs.
4. PE redondant : utilisés en cas de problèmes sur les autres PEs.

Architecture :

1. MIMD à mémoire distribuée.
2. Interconnexion par un réseau 3D toroïdal : le débit maximum de chaque lien est de 600 Mo/s.
3. Présence de mécanismes matériels pour masquer les latences et pour réaliser les synchronisations.
4. Un processeur par noeud.

Caractéristiques d'un processeur élémentaire :

1. DEC alpha 21164 (EV-5)
2. Fréquence d'horloge 300 MHz.
3. 4 instructions par cycle : 2 pipelines entiers et 2 flottants (600 MFlops crête).
4. Registres 64 bits.
5. 128 Mo de RAM.

A.2.2 Configuration logicielle

Environnement de programmation :

1. Système d'exploitation : UNICOS/MK Version 1.4.1.50.
2. Compilateur Fortran 90 : CF90 Version 2.0.3.0.
3. Compilateur C++ : CC Version 2.0.3.0.
4. Compilateur C : cc Version 5.0.3.0.
5. Editeur de liens : cld ("common loader").
6. Debogueur : Totalview 2.1.
7. Analyseur de performances : Apprentice 2.0.
8. "Message passing toolkit" MPT 1.1 :

    a. PVM (équivalent au PVM 3.3.10 de ORNL)
    b. MPI (dérive de l'implémentation de MPI pour CRAY T3D développée à Edinburgh)
    c. Librairie libsma incluant les appels shmem

A.3 Réseau de station SUN

A.3.1 Configuration matérielle

Les stations de travail que nous utilisons sont toutes sur le même brin éthernet. Le débit maximum du réseau est de 10 Mo/s.

Quatre types de stations sont employés :

1. Sparc 5 (70 MHz) :
   a. Microprocesseur Fujitsu MB86904 microSPARC II
   b. Fréquence d'horloge 70 MHz.
   c. Registres 32 bits.
   d. 48 Mo de RAM.
2. Sparc 5 (110 MHz) :
   a. Microprocesseur Fujitsu MB86904 microSPARC II
   b. Fréquence d'horloge 110 MHz.
   c. Registres 32 bits.
   d. 48 Mo de RAM.
3. Ultra Sparc (140 MHz) :
   a. Microprocesseur UltraSparc
   b. Fréquence d'horloge 140 MHz.
   c. Registres 32 bits.
   d. 64 Mo de RAM.
4. Ultra Sparc (170 MHz) :
   a. Microprocesseur UltraSparc
   b. Fréquence d'horloge 170 MHz.
   c. Registres 32 bits.
   d. 64 Mo de RAM.

A.3.2 Configuration logicielle

Environnement de programmation :

1. Système d'exploitation : SunOs Version 5.5.1.
2. Compilateur Fortran 90 : f90 Version 1.1.
3. Compilateur C++ : CC Version 4.1. et gcc Version 2.7
4. Compilateur C : cc Version 4.0. et gcc Version 2.7
5. Debogueur : dbx 3.2. et DDD Version 1.4b
6. PVM 3.3.11 de ORNL

Annexe B
Représentation et caractéristiques des images

B.1 Introduction

Standard Procedural Databases (SPD) est un ensemble de logiciels, mis gracieusement à la disposition de la communauté scientifique par E. Haines [Hai87].

Le but de SPD est de fournir un ensemble d'images de test pour les algorithmes de lancer de rayon. Les différents programmes génèrent des formats de bases de données d'objets qui sont considérés comme standard pour la communauté infographique. Grâce à SPD, les différents algorithmes de lancer de rayon et leurs améliorations peuvent être comparés de façon relativement facile. De plus SPD fournit des statistiques complètes comprenant en plus des temps de calculs le nombre de rayons, d'intersections etc.

Les principales images proposées par SPD sont :

1. "Gears" (Engrenages).
2. "Mount" (Montagne).
3. "Rings" (Anneaux).
4. "Sphereflake" (Flocon de sphères).
5. "Teapot" (Théière).
6. "Tetra" (Tétrahèdre).
7. "Tree" (Arbre).

Ces bases de données sont construites de façon itérative, ce qui permet de pouvoir faire varier le nombre d'objets. Dans l'ensemble de notre travail, les images sont nommées en postfixant leur nom avec le nombre de récursion n.

B.2 Caractéristiques des images utilisées dans notre travail

Nom de l'image	Nombre d'objets	Nombre de sources lumineuses	Taille de la base de données (et de la structure dupliquée)	Temps de pré-calcul et de calcul pour un processeur du CRAY T3E
Gears	$66 * n^3$ polygones	5		
Gears_2 (G2)	528 polygones		1 Mo (0,02 Mo)	1 s - 715 s
Gears_6 (G6)	14256 polygones		26,2 Mo (0,4 Mo)	24 s - 5483 s
Gears_8 (G8)	33792 polygones		62 Mo (0,9 Mo)	39 s - 12873 s
Mountain	$2 * 4^n$ polygones	1		
Mountain_5 (M5)	2048 polygones		3,5 Mo (0,06 Mo)	4 s - 329 s
Mountain_7 (M7)	32768 polygones		56,5 Mo (0,9 Mo)	45 s - 3716 s
Rings	$5 * n * (n+1) * (2 * n + 1)$ sphères et $5 * n * (n+1) * (2 * n + 1) * 16 + 1$ polygones	3		
Rings_1 (R1)	20 sphères + 321 polygones		0,8 Mo (6 Ko)	1 s - 311 s
Rings_4 (R4)	900 sphères + 14401 polygones		24,5 Mo (0,4 Mo)	20 s - 5426 s
Rings_5 (R5)	1650 sphères + 26401 polygones		45 Mo (0,7 Mo)	31 s - 9820 s
SphereFlake	$\sum_{i=0}^{n-1} 9^i$ sphères et 1 polygone	2		
SphereFlake_4 (S4)	821 sphères + 1 polygone		1,2 Mo (0,03 Mo)	1 s - 187 s
SphereFlake_5 (S5)	7381 sphères + 1 polygone		10,4 Mo (0,2 Mo)	37 s - 860 s
Teapot	$n * (65 * n - 8)$ polygones	2		
Teapot_5 (P5)	1585 polygones		2,7 Mo (0,05 Mo)	3 s - 388 s
Teapot_9 (P9)	5193 polygones		8,9 Mo (0,2 Mo)	10 s - 794 s
Teapot_20 (P20)	25840 polygones		44 Mo (0,7 Mo)	33 s - 3210 s
Tetrahedron	4^n polygones	2		
Tetrahedron_4 (T4)	256 polygones		0,4 Mo (6 Ko)	1 s - 58 s
Tetrahedron_5 (T5)	1024 polygones		1,6 Mo (0,03 Mo)	2 s - 88 s
Tetrahedron_6 (T6)	4096 polygones		6,6 Mo (0,1 Mo)	13 s - 221 s
Tetrahedron_7 (T7)	16384 polygones		26,4 Mo (0,5 Mo)	153 s - 618 s
Tree	$2^{(n+1)} - 1$ sphères et $\left(2^{(n+1)} - 1\right) * 16 + 1$ polygones	7		
Tree_8 (A8)	511 sphères + 8177 polygones		14,5 Mo (0,3 Mo)	15 s - 648 s
Tree_10 (A10)	2047 sphères + 32753 polygones		58,5 Mo (1 Mo)	53 s - 1794 s

Table 15 Caractéristiques des images utilisées dans notre travail

B.3 Représentations des différentes images

B.3.1 Gears

Cette image représente un ensemble d'engrenages emboîtés.

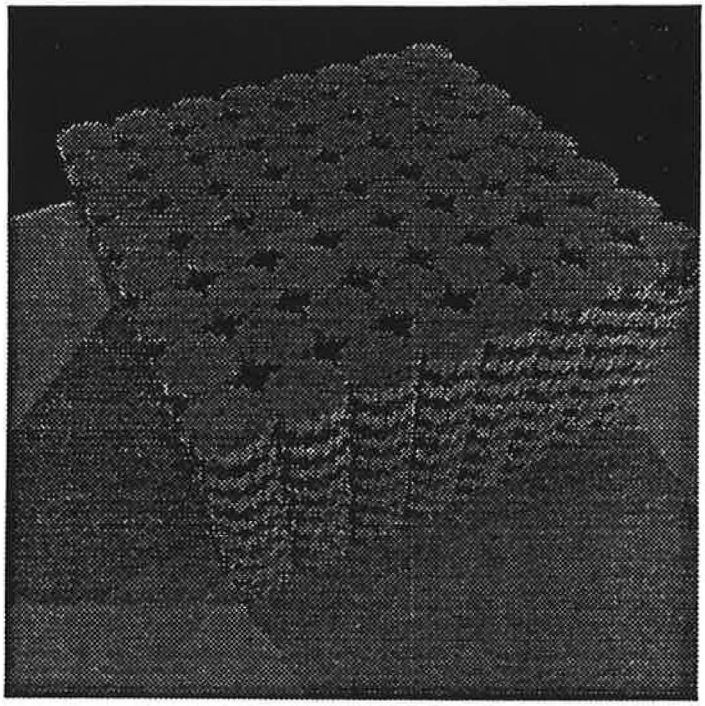

Figure 96 Image Gears_8

B.3.2 Mount

Ce programme permet la génération d'une montagne fractale.

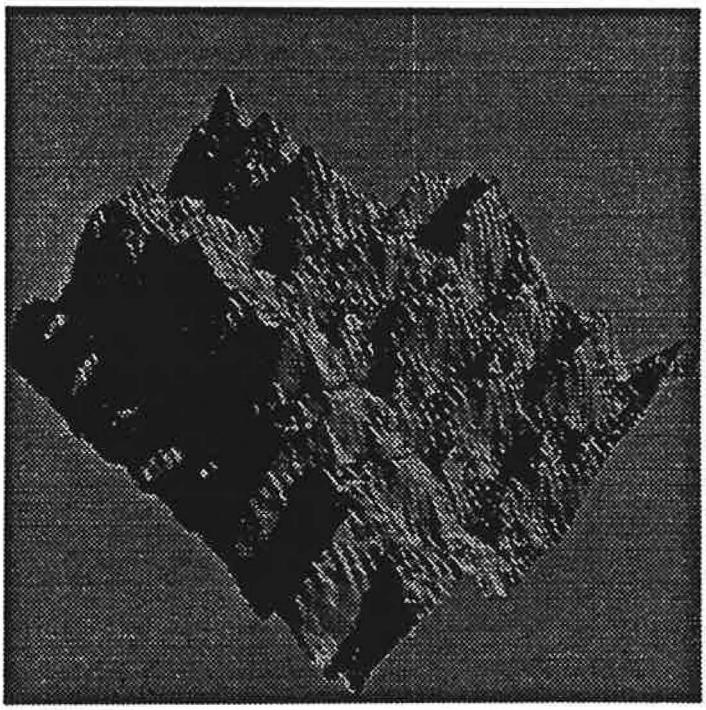

Figure 97 Image Mountain_7

B.3.3 Rings

Cette image représente un enchevêtrement d'anneaux.

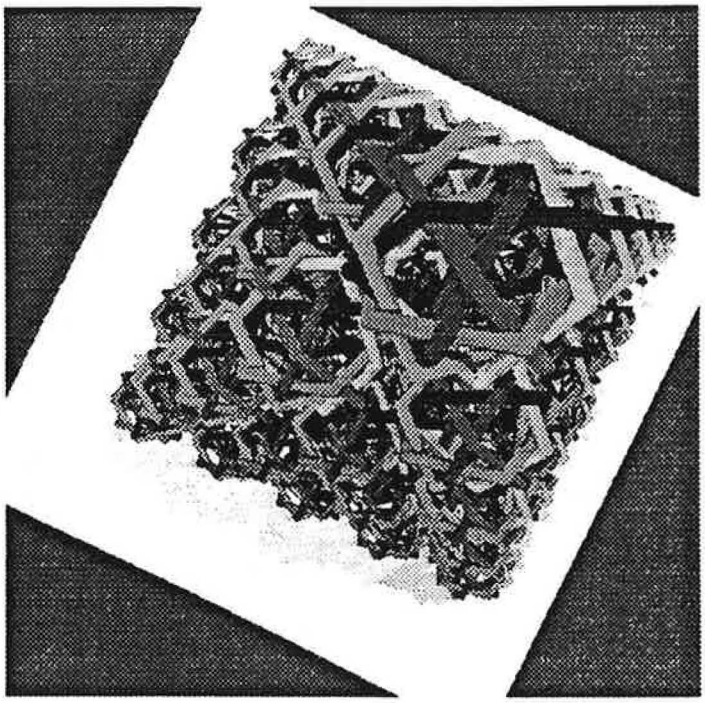

Figure 98 Image Rings_5

B.3.4 SphereFlake

Cette image, représentant un flocon de sphères, consiste en une variété de sphères de tailles différentes. L'utilisation de 3 sources lumineuses engendre un grand nombre de rayon d'ombrages.

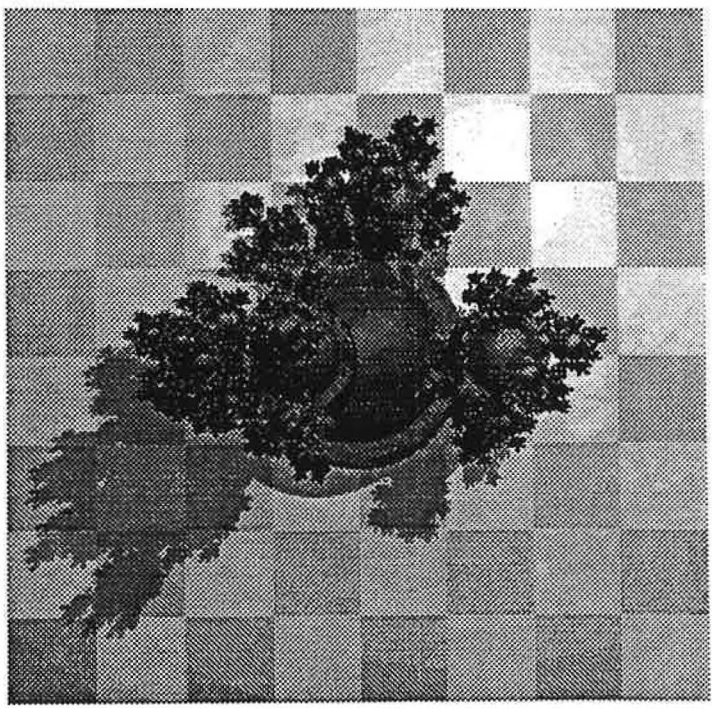

Figure 99 Image SphereFlake_5

B.3.5 Teapot

C'est sans doute l'image la plus connue : une théière sur un échiquier. Ces objets sont représentés par un maillage triangulaire.

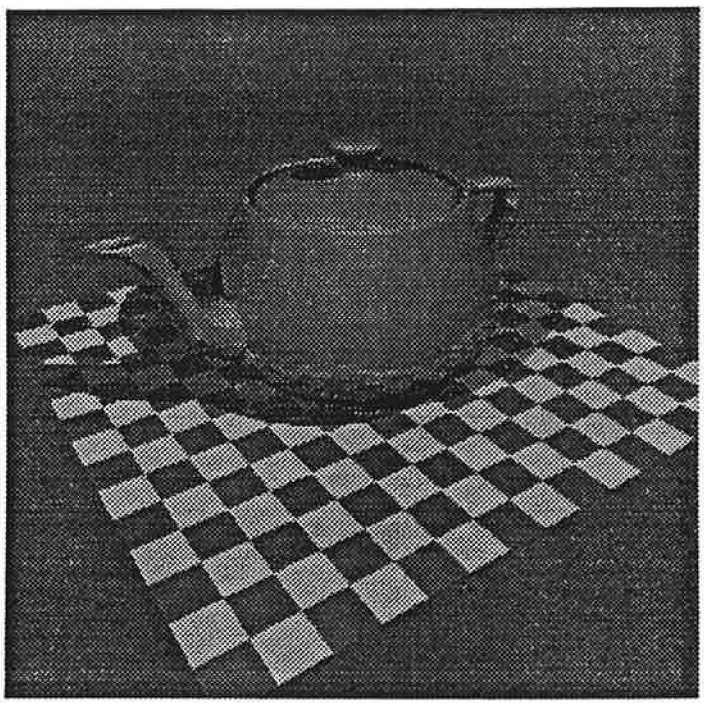

Figure 100 Image Teapot_20

B.3.6 Tetrahedron

Cette image fractale illustre le tétrahèdre de Sierpinski. Elle est composée de quatre tétrahèdres disposés en tétrahèdre, chacun d'eux construit également à partir de quatre tétrahèdres...

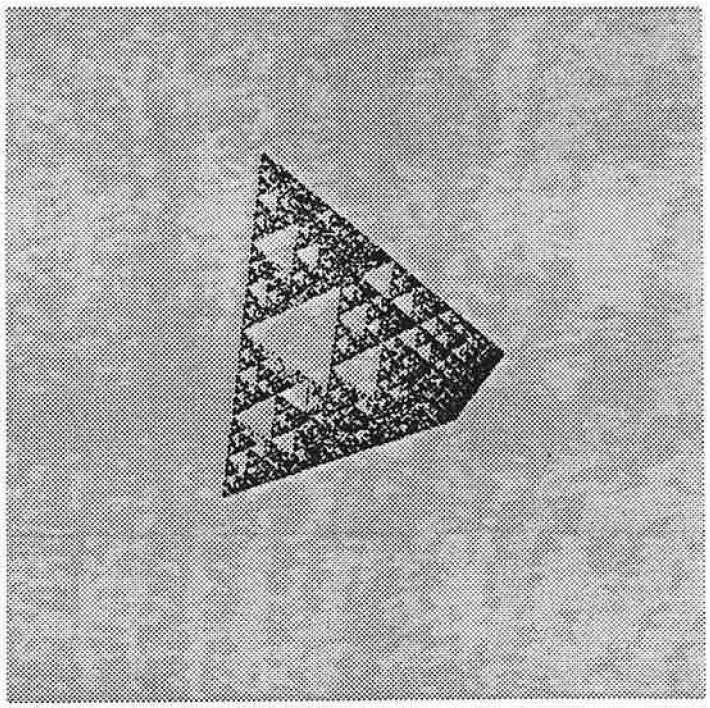

Figure 101 Image Tetrahedron_7

B.3.7 Tree

Cette image représente un arbre éclairé par 7 sources lumineuses.

Figure 102 Image Tree_10

Annexe C
Fichier de paramètres d'optimisation

On peut voir ci-dessous, le contenu du fichier permettant de configurer l'exécution de notre code. Ce fichier comporte quatre rubriques :

1. Les paramètres généraux : ils permettent de spécifier le mode de fonctionnement de l'algorithme (par exemple : flots de données avec équilibrage dynamique) et le seuil signalant un manque de calculs.
2. Les paramètres d'équilibrage : ils précisent le nombre maximum de calculs envoyés dans un message d'équilibrage et la charge en-dessous de laquelle un processeur cesse d'être donneur.
3. Les paramètres pour flots de données : ils fixent le nombre de messages de demandes de données pouvant être adressés simultanément par un processeur, le nombre de demandes de données pouvant être enregistrées et le nombre de calculs à partir duquel une demande peut être effectuée.
4. Les paramètres pour flots de calculs : ils déterminent les nombres de calculs minimum et maximum dans un message.

```
--------------------------
--PARAMETRES_GENERAUX
--------------------------
*Flots_Calculs_Bit 1
*Flots_Donnees_Bit 1
*Equilibrage__Bit 0
.
*Multi_Infos_Bit 0
*Calculs_Locaux_Bit 0
----------
*Seuil_Famine 100
.
--------------------------
--PARAMETRES_EQUILIBRAGES
--------------------------
*Dose 00
*Seuil_Donneur_Coeff_% 00
.
--------------------------
--PARAMETRES_FLOTS_DONNEES
--------------------------
*Nb_canaux_donnees 5
*Max_Demandes_Donnees_Coeff 2
*Dose_Demandes_Donnees 25
.
--------------------------
--PARAMETRES_FLOTS_CALCULS
--------------------------
*Seuil_Flots_Calculs 250
*Seuil_Flots_Mixtes 00
*Dose_Max 250
.
*Choix 10
.
**************************
Bit : OUI(1) ou NON(0)
Coeff : multiplie
% : pourcentage
**************************
```

Annexe D
Types de messages échangés

Pour le fonctionnement de notre noyau parallèle, 11 types de messages différents sont utilisés. Dans le tableau suivant nous indiquons pour chacun d'eux leur nom et leur rôle :

Nom du message	Rôle
C_DEMANDES_DONNEES	Demande une donnée
C_DEMANDE_CALCULS	Demande de calculs
C_END	Signal de fin de la phase de calcul
C_IMAGE	Transmission des tableaux de résultats
C_FOURNITURE_CALCULS_IMPOSSIBLE	Indication de l'absence de calculs disponibles à l'envoi
C_FOURNITURE_CALCULS_DEMANDES	Envoi des calculs demandés
C_FLOT_CALCULS	Envoi de calculs
C_TERMINAISON	Recherche si tous les calculs sont terminés
C_INFORMATIF	Envoi uniquement de l'entête d'information
C_ENVOI_DONNEE_0	Premier des N canaux de transmission des données demandées
C_ENVOI_MIXTE_0	Premier des N canaux de transmission conjointe de données demandées et de calculs

Table 16 Liste des messages

Annexe E
Architecture du code

Le code parallèle à réaliser se devant d'être générique, nous l'avons décomposé en modules indépendants :

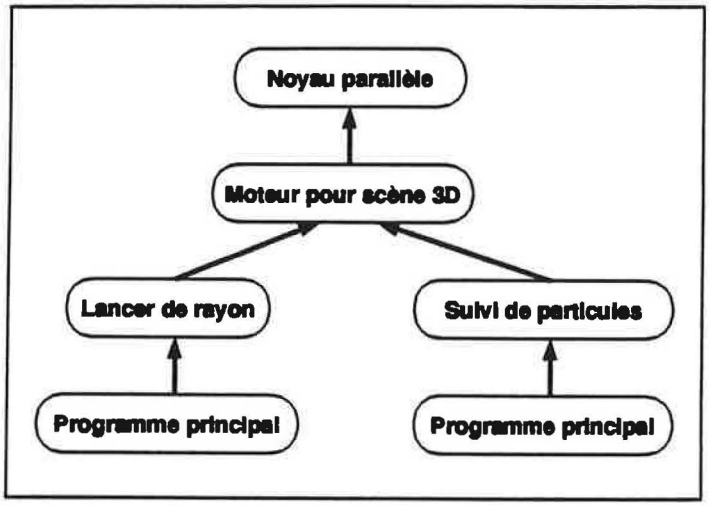

Figure 103 Architecture du code

Le module qui a demandé le plus d'attention est le noyau parallèle. Ce module peut être associé à un moteur d'intersection pour scènes tridimensionnelles, mais également à tout module nécessitant une gestion de l'échange de tâches ou de données sur une architecture parallèle. Le moteur d'intersection peut être utilisé soit pour l'algorithme de lancer de rayon, soit par un algorithme de suivi de particules. Enfin les deux derniers modules sont utilisés par un programme maître qui pourra par exemple lire la topologie de la scène 3D considérée ainsi que les propriétés tant physiques qu'optiques des différents objets la composant.

La présentation de ces différents modules est réalisée par le biais de leur diagramme objet. La figure ci-dessous montre les conventions graphiques que nous utilisons pour représenter nos diagrammes objet.

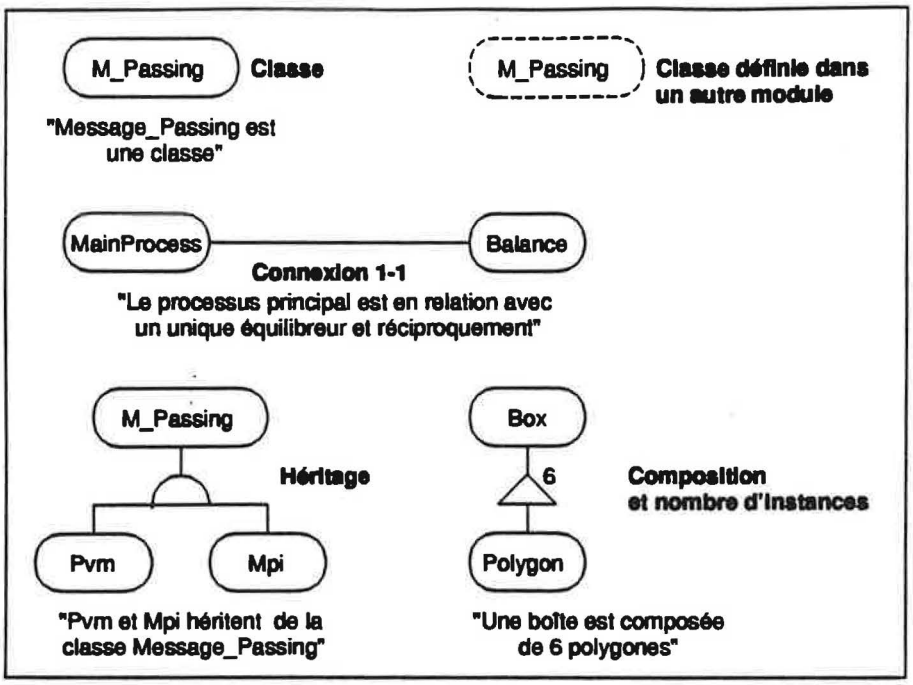

Figure 104 Conventions graphiques de représentation des diagrammes objets

E.1 Module noyau parallèle

Le but de ce module est de permettre à tout utilisateur d'une machine parallèle de pouvoir s'affranchir des problèmes liés à l'utilisation de données distantes et à l'équilibrage de charges.

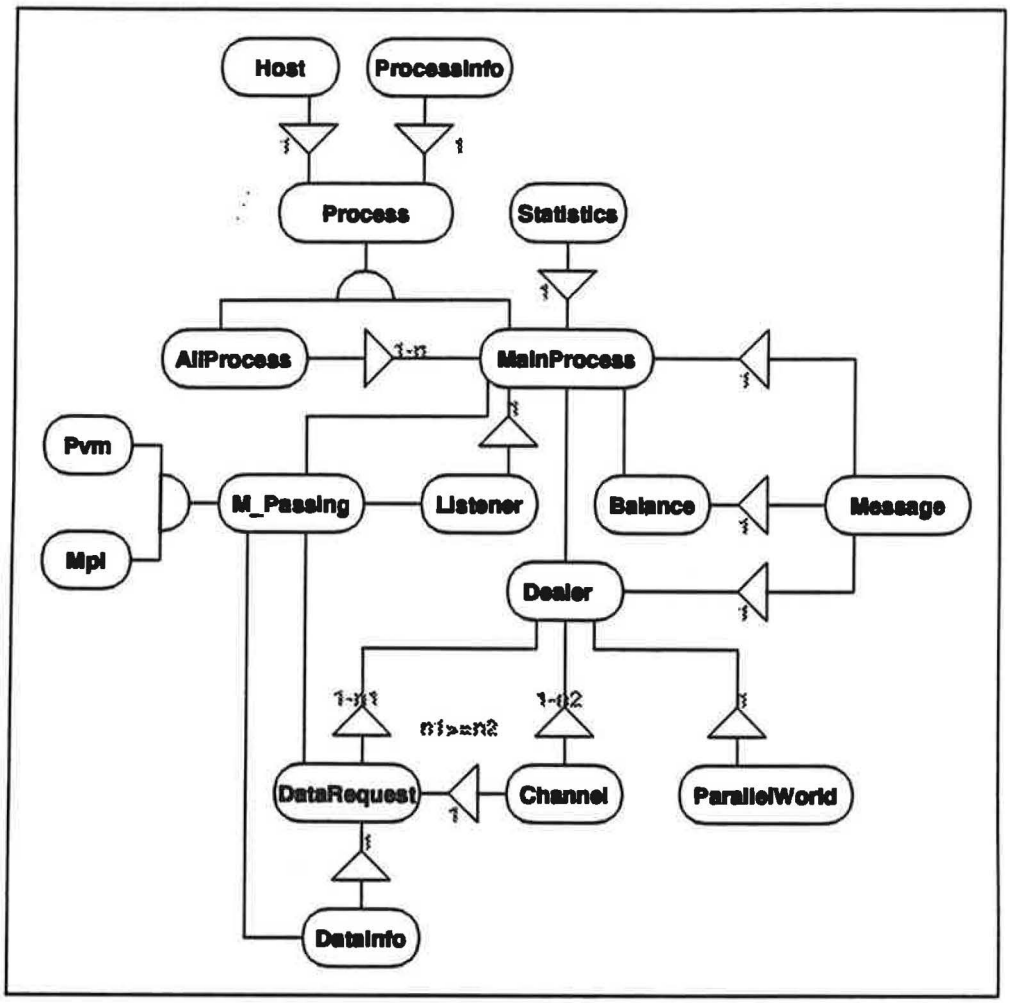

Figure 105 Diagramme objet du noyau parallèle

On peut noter que les communications se font par le biais d'une classe *M-Passing*, qui permet d'utiliser de façon transparente n'importe quelle bibliothèque basée sur l'échange de messages *Message* (actuellement seuls sont implémentés PVM et MPI).

Chaque processus, *MainProcess*, présent sur un processeur dispose d'informations sur les autres processus de la machine, *AllProcess*. Le processus, *MainProcess*, permet la parallélisation d'un code, *ParallelWorld*, en traitant les demandes de données, *DataRequest*, du code par le biais de la classe *Dealer*. Enfin, l'équilibrage dynamique de charge est réalisé par la classe *Balance*. La réception de l'ensemble des messages se fait grâce à la classe *Listener*.

E.2 Module moteur d'intersection

Ce module renferme l'ensemble des définitions permettant la construction d'une scène 3D ainsi que des fonctionnalités permettant l'intersection de droites avec des objets. On retrouve globalement le diagramme proposé par OORT.

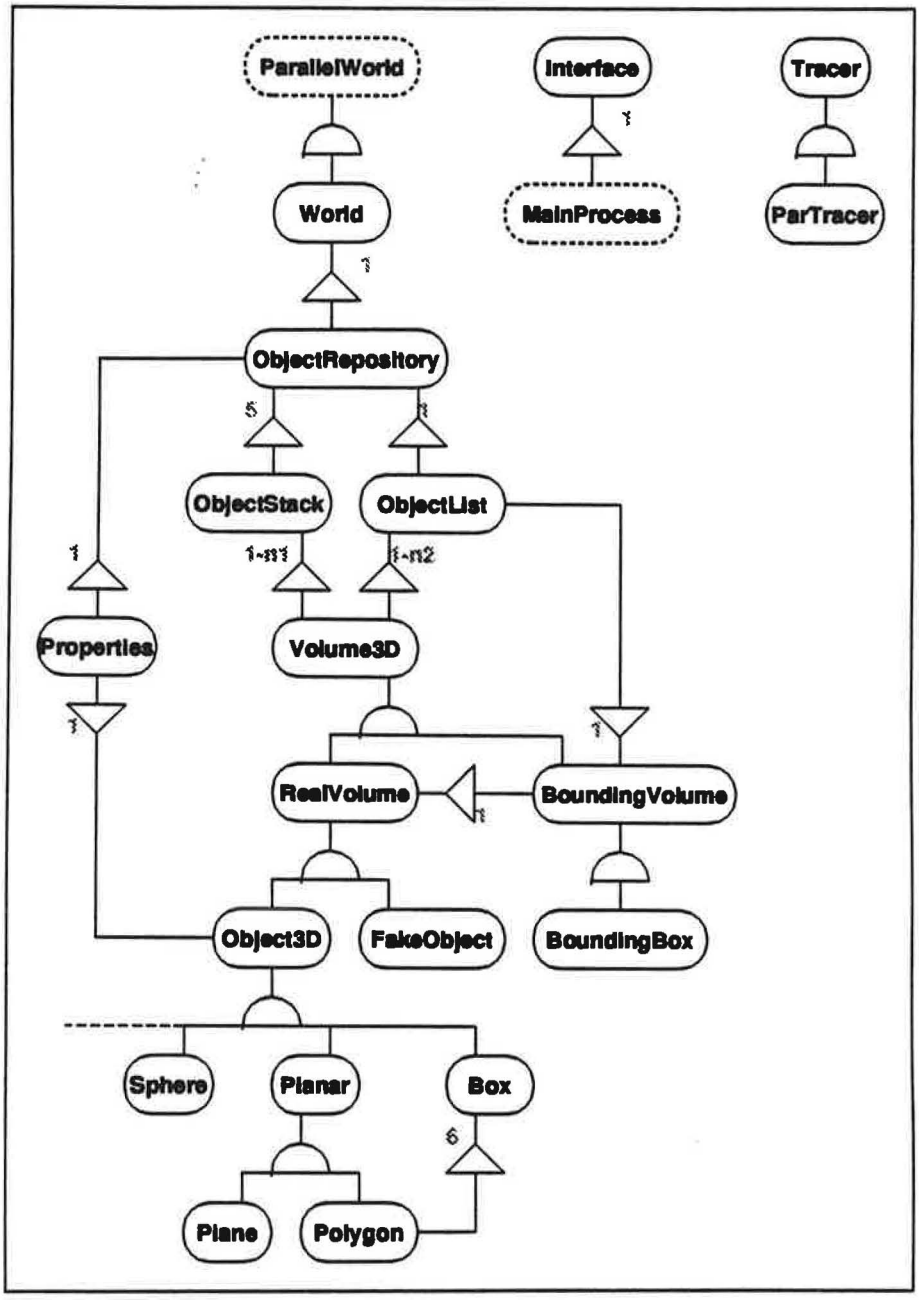

Figure 106 Diagramme objet du moteur d'intersection

La descrition du problème à représenter se fait à l'aide de la classe *World*, qui comprend des objets, *Volume3D*, aux rôles (*BoundingVolume, FakeObject...*) et aux propriétés géométriques (*Sphere, Plane, Polygon...*) variées, et des propriétés non géométriques, *Properties*. Enfin le code dispose de calculs à effectuer, *Tracer*, au sein de cette géométrie tridimensionnelle.

E.3 Module optique

Ce module est totalement lié à l'application lancer de rayon optique. Il comporte les définitions des rayons, des sources de lumière ainsi que des propriétés optiques que peuvent avoir un objet 3D. On retrouve de nouveau un diagramme proche de celui proposé par OORT.

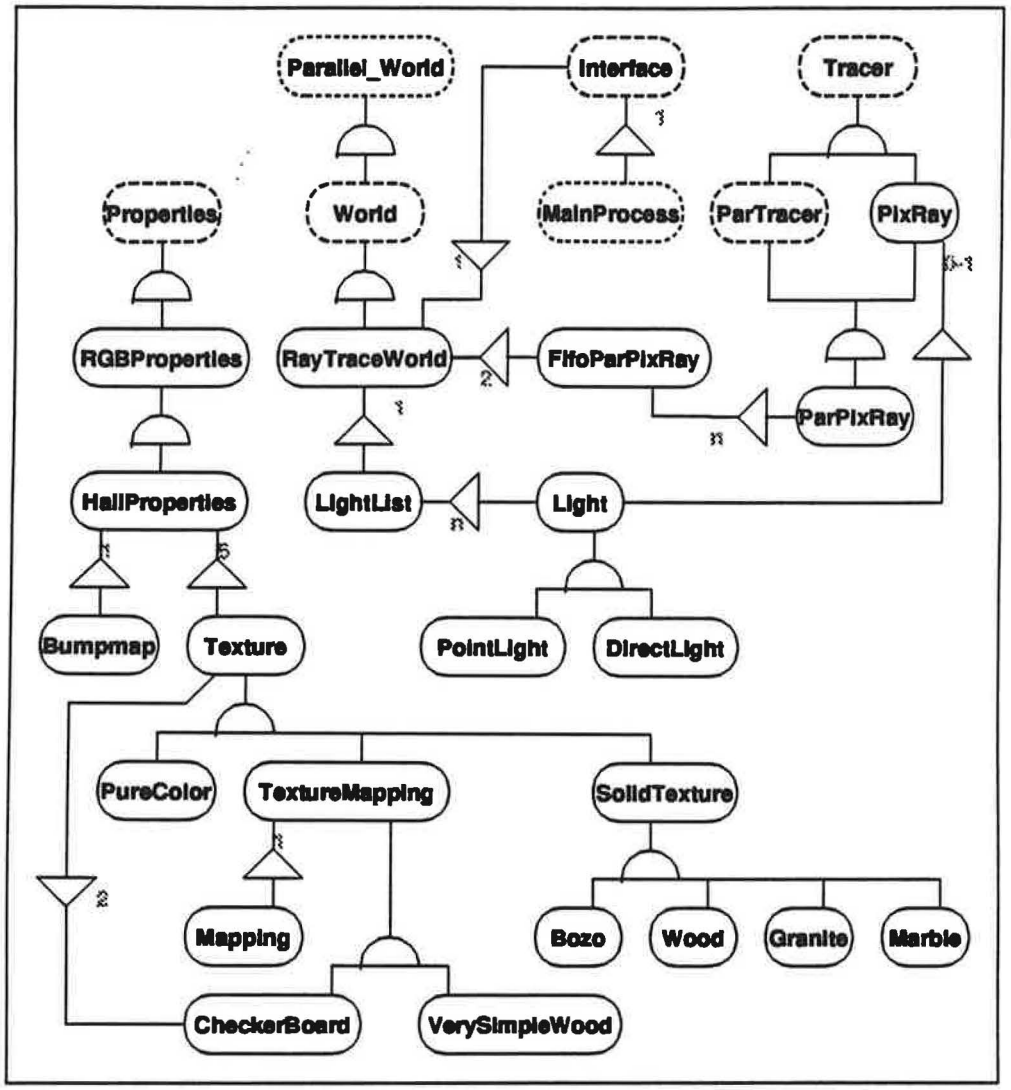

Figure 107 Diagramme objet du module optique

La descrition de la scène à représenter se fait à l'aide de la classe *RaytraceWorld*. Il faut spécifier les sources lumineuses, *Light*, et les propriétés optiques des objets, *RGBProperties (Texture et Bumpmap)*. Enfin il faut préciser les caractéristiques des rayons de la scène, *ParPixRay*.

E.4 Module particulaire

Ce module est totalement lié à l'application suivi de particules.

Il comporte les définitions des particules ainsi que celles des propriétés physiques que peuvent avoir un objet 3D ou une particule.

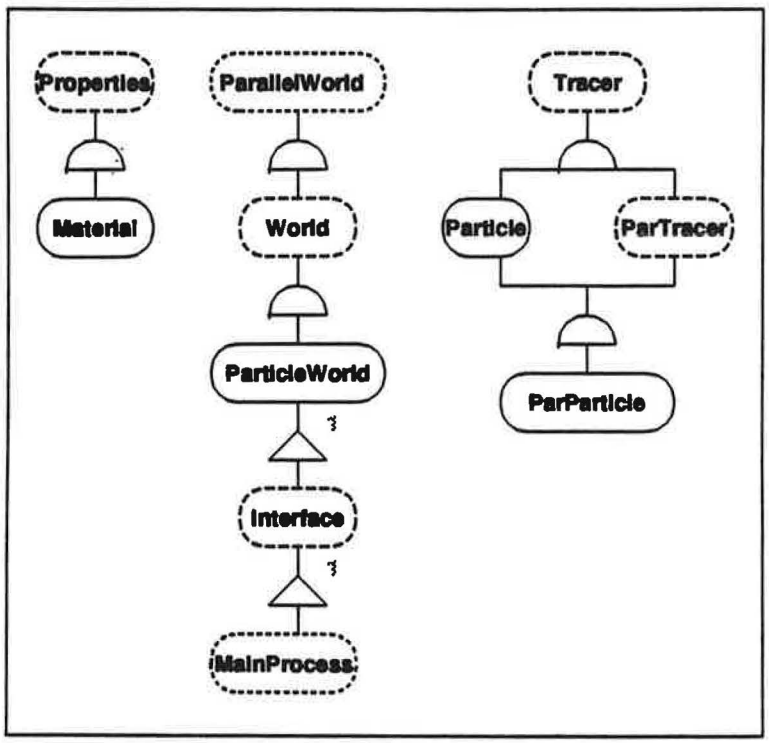

Figure 108 Diagramme objet du module particulaire

La descrition de la scène dans laquelle va évoluer les particules se fait à l'aide de la classe *ParticleWorld*. Il faut préciser les propriétés des différents objets, *Material*, ainsi que celle des particules qui vont les rencontrer, *ParParticle*.

Annexe F
Description d'une scène

Nous présentons ci-dessous un fichier de description de scène. Ce fichier, qui doit être compilé, est écrit en C++ et permet de créer une image comportant une sphère réfléchissante et une source de lumière ponctuelle. Les commentaires intégrés au sein du fichier assure une bonne compréhension des différentes étapes permettant la création de la scène.

```
//*********************************************************************
// DESCRIPTION D'UNE SCENE COMPRENANT UNE SPHERE REFLECHISSANTE ET
// UNE SOURCE DE LUMIERE PONCTUELLE
//*********************************************************************

void PopulateWorld(RayTraceWorld &MaScene)
{
//*********************************************************************
// Positionnement des parameres generaux de la scene
//*********************************************************************

    // Position de l'observateur

    Vector3D PositionObservateur(300,300,300);

    // Direction du centre de l'image

    Vector3D DirectionCentreImage(40,40,0);

    // Direction du haut de l'image

    Vector3D DirectionHautImage(0,1,0);

    // Ajout de l'observateur a la scene

    MaScene.SetViewerParameters(DirectionCentreImage,
                                PositionObservateur,
                                DirectionHautImage);

    // Distance entre l'ecran et l'observateur

    MaScene.SetScreenDistance(100);

    // Taille de l'ecran

    MaScene.SetScreenWidth(50);
    MaScene.SetScreenHeight(50);

    // Taille de l'image en pixels

    MaScene.SetHorzRes(512);
    MaScene.SetVertRes(512);

    // Couleur de la lumiere ambiante

    MaScene.SetAmbientLight(RGBColor(RGBColor(1)));

    // Couleur de l'exterieur de la scene

    MaScene.SetBackgroundColor(RGBColor(0.1, 0.1, 0.3));
```

```
//*********************************************************************
// Creation d'un objet reflechissant (sphere)
//*********************************************************************

    // Proprietes geometriques de l'objet
    //***********************************

     Vector3D PositionCentreSphere(50,0,0);
     float RayonSphere = 50;

    // Proprietes optiques de l'objet
    //*******************************

     // Definition de differentes couleurs

     PureColor* CouleurAmbiante = new PureColor(0.2,0.2,0.2);
     PureColor* CouleurSpeculaire = new PureColor(0.5,0.5,0.5);
     PureColor* CouleurReflexion = new PureColor(0.7,0.7,0.7);

     // Creation d'une texture

     HallProperties* TextureSphere = new HallProperties();

     TextureSphere->SetAmbient(CouleurAmbiante);
     TextureSphere->SetSpecular(CouleurSpeculaire, 10);
     TextureSphere->SetReflect(CouleurReflexion);


    // Creation de la sphere
    //**********************************

    Sphere* MaSphere = new Sphere(PositionCentreSphere,RayonSphere,
                                  TextureSphere);


    // Ajout de la sphere a la scene

    MaScene.AddObject(MaSphere);


//*********************************************************************
// Creation d'une source de lumiere
//*********************************************************************

    // Position de la source de lumiere

    Vector3D PositionLumiere(360,200,200);

    // Couleur de la source de lumiere (blanc)

    RGBColor CouleurLumiere(1,1,1);

    // Attenuation de la source de lumiere avec la distance (d)
    // (l'attenuation suit l'expression : 1/(a+b*d+c*d*d) )

    Vector3D AttenuationLumiere(1,0,0);

    // Creation d'une source de lumiere ponctuelle

    PointLight* Lumiere = new PointLight(PositionLumiere(360,200,200),
                                         CouleurLumiere,
                                         Attenuation);

    // Ajout de la source de lumiere a la scene
    MaScene.AddLight(Lumiere);


}
```

Annexe G
Variables de configuration de PVM sur CRAY

Nous présentons ici les variables d'environnement nécessaire à l'utilisation de PVM sur CRAY. Pour chacune d'elle, nous précisons entre parenthèses la valeur donnée par défaut.

PVM_DATA_MAX <*initial*>,(4096 octets)

A l'envoi d'un message, un header et un morceau du message ('*initial*') sont d'abord envoyés, si le message entier est plus gros que 'initial' alors le reste sera envoyé dans un second transfert plus lent.

PVM_DATA_BUFFER <*nombre*>+<*incrementation*>,(0+1)

A chaque création d'un buffer, il y a appel à une librairie pour créer dynamiquement la mémoire nécessaire. Aussi pour diminuer le coût de cette opération, on peut créer à l'initialisation de l'environnement de PVM '*nombre*' buffers en une fois. De plus, à chaque fois qu'on a besoin d'allouer un buffer supplémentaire, on peut en créer '*incrementation*' en une fois.

PVM_SPOOL <*nombre_messages*>, (Max(nb_processeurs,10))

Cette variable permet de spécifier le nombre de messages maximum *nombre_messages* qu'un processeur peut émettre avant qu'ils soient reçus.

PVM_MAX_PACK <initial>+<incrementation>,(4096 octets)

Quand un buffer est alloué, sa taille minimale est *initial*. S'il a besoin de moins de mémoire, cette mémoire est perdue. S'il a besoin de plus de mémoire, il va réallouer la mémoire nécessaire par bloc de 4096 octets ou va allouer un unique bloc de '*incrementation*' s'il est positionné.

PVM_TOTAL_PACK <limit>, (+infini)

Limite la mémoire utilisable par PVM sur chaque processeur.

L'utilisation de PVM sur CRAY induit la création d'une zone de mémoire réservée à l'échange de message. Sa taille en octet pour chaque processeur peut être exprimée à l'aide des variables d'environnement : PVM_SPOOL*(PVM_DATA_MAX+32).

Annexe H
Glossaire

Blocage mortel (ou dead lock) Interdépendance de communications induisant un blocage irréversible lors de l'exécution d'un code.

Exemple de blocage mortel sur trois processeurs A, B et C :

1. A attend un message de C pour envoyer un message à B,
2. B attend un message de A pour envoyer un message à C,
3. C attend un message de B pour envoyer un message à A.

CSG (Constructive Solid Geometry) Méthode permettant la construction d'objets complexes à partir d'opérations booléennes sur des objets simples.

Flots Nature des communications sur lesquelles repose un algorithme parallèle.

Flots de calculs

Communications par envois de calculs à effectuer ou à poursuivre.

Flots concurrents

Communications par envois de calculs ou de données en fonction de l'envoi le plus adapté.

Flots de données

Communications par envois de données. Pour l'algorithme du lancer de rayon, les données sont les objets de la scène.

Flots mixtes

Communications par envois de calculs et de données.

Flots multi-informatifs

Communications par envois de messages comprenant des calculs et des données.

Flots de rayons

Flots de calculs, ceux-ci étant des calculs de rayons.

Hôte Machine servant d'interface entre l'utilisateur et une architecture matérielle.

Latence Temps nécessaire pour que le plus petit message possible quitte un processeur émetteur et arrive sur un processeur receveur.

LRU (Least Recently Used) Algorithme de gestion de mémoire permettant l'ajout d'une nouvelle donnée par suppression de la donnée qui a été la moins utilisée récemment.

Maître-esclaves Architecture logique centralisée dans laquelle un processeur maître orchestre l'exécution de tâches sur des processeurs esclaves.

Message Passing (Echange de messages) Paradigme de programmation parallèle dans lequel une véritable transaction incluant un processeur receveur et un processeur émetteur est utilisée. Les principales bibliothèques de message passing sont PVM et MPI.

Monotâche Système d'exploitation ne permettant l'exécution d'une unique tâche par processeur.

Multitâches Système d'exploitation permettant l'exécution de plusieurs tâches sur un même processeur.

Noeud Element d'architecture parallèle comprenant des processeurs et des ressources mémoires.

PE (Processing Element) Elément de calcul, souvent utilisé comme synonyme de processeur.

Profondeur d'un rayon Représente dans l'algorithme du lancer de rayon, le nombre de réflexions et/ou de réfractions qu'à subi un rayon primaire.

RVB Modèle de représentation par combinaison des 3 couleurs Rouge, Vert et Bleu (RGB en anglais). Toute couleur peut se présenter sous la forme d'un triplet RVB $(x, y, z) \in [0, 1]^3$.

SPD Standard Procedural Databases est un ensemble de logiciels, proposé par Haines [Hai87], dont le but est de fournir des images de test pour les algorithmes de lancer de rayon. Les formats de bases de données d'objets générés sont considérés comme standard pour la communauté infographique.

Texture Ensemble des propriétés optiques d'un objet.

Voxel Volume élémentaire (Volume Element), par analogie avec le pixel (Picture Element).

Références bibliographiques

[AGL89]M. Agate, R.L. Grimsdale, and P.F. Lister.
The hero algorithm for ray tracing octree.
Advances in computer graphics hardware IV, 1989.

[AK87] J. Arvo and D. Kirk.
Fast ray tracing by ray classification.
Computer graphics, 21(4):55–63, July 1987.

[AK89] J. Arvo and D. Kirk.
An introduction to ray tracing, chapter 5. A survey of ray tracing acceleration techniques, pages 201–262.
Academic press, 1989.

[App68]A. Appel.
Some techniques for shading machine renderings of solids.
Proceedings of AFIPS 1968 Spring joint computer conference, 32:37–45, 1968.

[AR94] D. Arquès and P. Ris.
Méthode de parallélisation du lancer de rayon intégrant une acquisition dynamique de la connaissance topologique de la scène.
Revue internationale de CFAO et d'infographie, 9(1–2):195–217, 1994.

[AW87]J. Amanatides and A. Woo.
A fast voxel traversal algorithm for ray tracing.
EURO-GRAPHICS'87 (Amsterdam) : Proceedings of the European Computer Graphics Conference and Exhibition, Ed. G. Marechal, pages 3–10, 1987.

[Bad90]D. Badouel.
Schémas d'exécution pour les machines parallèles à mémoire distribuée. Une étude de cas : le lancer de rayon.
PhD thesis, Université de Rennes I — IFSIC, Rennes, October 1990, 1990.

[Bar90] M.L. Barett.
Load balancing experiment for parallel ray tracing.
Proceedings of Ausgraph'90, Australasian computer Graphics Association, pages 145–155, 1990.

[BBP94]D. Badouel, K. Bouatouch, and T. Priol.
Distributed data and control for ray tracing in parallel.
IEEE computer graphics and applications, 14(4):69–76, July 1994.

[BMP93]K. Bouatouch, D. Menard, and T. Priol.
Parallel radiosity using a shared virtual memory.
In first Bilkent computer graphics conference on advanced techniques in animation, rendering and visualization, Ankara., 1993.

[BP90] D. Badouel and T. Priol.
An efficient parallel ray tracing scheme for highly parallel architectures.
Advances in computer hardware v. rendering, ray tracing audiovisualisation systems. Lausanne, CH, 2-3 september 1990, pages 93–106, September 1990.

[CCWG88]M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg.
A progressive refinement approach to fast radiosity image generation.
SIGGRAPH'88, 22(4):75–84, August 1988.

[CG85] M. F. Cohen and D. P. Greenberg.
The hemicube: a radiosity solution for complex environments.
SIGGRAPH'85, 19(3):31–40, July 1985.

[CT90] M. Carter and K. Teague.
Distributed object database ray tracing on the intel ipsc/2 hypercube.
Proceedings of the 5th distributed memory computing conference, pages 217–222, April 1990.

[DFG83]E. W. Dijkstra, W. H. J. Feijen, and A. J. M. Van Gasteren.
Derivation of a termination detection algorithm for distributed computation.
Inf. Proc. Letters, (16):217–219, June 1983.

[DPT94]G. Damm, V. Pertuy, and W.Y. Thang.
Classification des architectures paralleles. bibliographie. classes de modèles pour la bibliothèque de mimesis (95nj00067).
Technical report, EDF, Direction des Etudes et Recherches, 1994.

[ES92] R. Endl and M. Sommer.
Ray-tracing mittels adaptiver octree-nachbarsuche.
Internal paper of the university of Marburg, at CeBIT'92, Exposition of the hessische hochschulen, Hannover, 1992.

[ES94] R. Endl and M. Sommer.
Classification of ray-generators in uniform subdivisions and octrees for ray tracing.
computer graphics forum, 13(1):3–19, March 1994.

[FGBA87]M.-C. Forgue, G. Giraudon, F. Boeri, and M. Auguin.
Génération d'images par lancer de rayon sur un calculateur parallèle.
In *11ème colloque GRETSI — Nice du 1 au 5 juin 1987*, volume 2, pages 653–655, June 1987.

[FI85] A. Fujimoto and K. Iwata.
Accelerated ray-tracing.
Proceedings of CGI'85, Tokyo, pages 41–65, 1985.

[Fly72] M.J. Flynn.
Some computer organizations and their effectivness.
IEEE Transactions on computers, 21(9):948–960, 1972.

[For92] High Performance Fortran Forum.
High performance fortran language specification, version 1.0 crpc-tr92225 (revised may. 1993).
Center for Research on Parallel Computation, Rice University, Houston, TX, 1992.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata.
Arts : Accelerated ray-tracing system.
IEEE Computers graphics and applications, 6(4):16–26, April 1986.

[GBD+94]A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam.
PVM : Parallel Virtual Machine a users' guide and tutorial for networked parallel computing.
MIT press, 1994.

[GHH+89]A.S. Glassner, E. Haines, P. Hanrahan, R.L. Cook, J. Arvo, D. Kirk, and P.S. Heckbert.
An introduction to ray tracing.
Academic press, 1989.

[Gla84] A.S. Glassner.
Space subdivision for fast ray tracing.
IEEE computer graphics and applications, 4(10):15–22, 1984.

[GLS94]W. Gropp, E. Lusk, and A. Skjellum.
Using MPI, portable parallel programming with the Message Passing Interface.
MIT press, October 1994.

[GN71] R.A. Goldstein and R. Nagel.
3–d visual simulation.
Simulation, pages 25–31, January 1971.

[GP90] S.A. Green and D.J. Paddon.
A highly flexible multiprocessor solution for ray tracing.
The visual computer, 6(2):62–73, March 1990.

[GPL88]S.A. Green, D.J. Paddon, and E. Lewis.
A parallel algorithm and tree-based computer architecture for ray traced computer graphics.
Parallel processing for computer vision and display. Leeds, UK, 1988, January 1988.

[GRS95]P. Guitton, J. Roman, and G. Subrenat.
Implementation results and analysis of a parallel progressive radiosity.
Proceedings of parallel rendering symposium (1995), Atlanta, Georgia, October 30–31., pages 31–38, October 1995.

[GS87] J. Goldsmith and J. Salmon.
Automatic creation of object hierarchie for raytracing.
IEEE computer graphics and application, 7(5):14–20, May 1987.

[GS88] J. Goldsmith and J. Salmon.
A hypercube ray-tracer.
Proceedings of the third conference on hypercube concurrent computers and applications., 2:1194–1206, January 1988.

[GTGBce]C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile.
Modeling the interaction of light between diffuse surfaces.
SIGGRAPH'84, 18(3):213–222, July reference.

[Hai87] E. Haines.
A proposal for standard graphics environments.
IEEE computer graphics and applications, 7(11):3–5, November 1987.

[HG86] E. Haines and D. P. Greenberg.
The light buffer : a shadow testing accelerator.
IEEE Computers graphics and applications, 6(9):6–16, 1986.

[HL97] M. Hamdi and C. K. Lee.
Dynamic load-balancing of image processing applications on clusters of workstations.
Parallel Computing, 22:1477–1492, 1997.

[HMS+92]Hebert, McNeill, Shah, Grimsdale, and Lister.
Marti — a multiprocessor architecture for ray tracing images.
Advances in computer hardware v. rendering, ray tracing audiovisualisation systems. Lausanne, CH, 2-3 september 1990, pages 69–83, September 1992.

[IAO91]V. Isler, C. Aykanat, and B. Ozguc.
Subdivision of 3d space based on the graph partitioning for parallel ray tracing.
Proc. second eurographics workshop on rendering, univ. of Catalonia, Barcelona, 1991, 1991.

[Kap85]M.R. Kaplan.
Space tracing, a constant time raytracer.
SIGGRAPH'85 tutorial on the uses of spatial coherence in raytracing, San Francisco, CA, July 1985.

[KH95] M. J. Keates and R. J. Hubbold.
Interactive ray tracing on a virtual shared-memory parallel computer.
Computer graphics forum, 14(4):189–202, 1995.

[KK86] T.L. Kay and J.T. Kajiya.
Ray tracing complex scenes.
ACM siggraph, 20(4):269–278, August 1986.

[KNK+88]H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei.
Load balancing strategies for a parallel ray-tracing system based on constant subdivision.
The visual computer, 4(4):197–209, October 1988.

[Lef92] W. Lefer.
Etude de la parallélisation de l'algorithme de lancer de rayon en synthèse d'images.
PhD thesis, Université de Caen, September 1992.

[Lef93] W. Lefer.
An efficient parallel ray tracing scheme for distributed memory parallel computers.
Proceedings of parallel rendering symposium (1993), San Jose, California, October 25–26, pages 77–80, October 1993.

[Mal97]N. Malléjac.
Modèles d'accès massivement parallèles à des tables numériques réparties.
PhD thesis, Université de Rennes, December 1997.

[MHI88]K. Murakami, K. Hirota, and M. Ishii.
Fast ray tracing.
Fujitsu scientific and technical journal, 24(2):150–159, June 1988.

[MM83]H. Matsumoto and K. Murakami.
Fast ray tracing using the octree partitioning.
27th Inform. proc. conf. proc., pages 15–22, 1983.

[MPS92]C. Montani, R. Perego, and R. Scopigno.
Parallel volume visualization on a hypercube architecture.
92 ACM workshop on volume visualization, pages 9–16, October 1992.

[Mül85]H. Müller.
Ray tracing complex scenes by grids.
Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, 22, 1985.

[NNce] T. Nishita and E. Nakamae.
Continuous tone representation of three-dimensional objects takong account of shadows and interreflection.
SIGGRAPH'85, 19(3):23–30, July reference.

[NSD95]C. D. Norton, B. K. Szymanski, and V. K. Decyk.
Object-oriented parallel computation for plasma simulation.
Communication of the ACM, 38(10):88–100, October 1995.

[PMR95]I. Pandzic, N. Magnetat, and M. Roethlisberger.
Parallel raytracing on the ibm sp2 and t3d.
EPFL — Supercomputing review, (7), November 1995.

[Pri89] T. Priol.
Lancer de rayon sur des architectures parallèles : étude et mise en oeuvre.
PhD thesis, IFSIC, Rennes, June 1989, 1989.

[Ris96] P. Ris.
Parallélisation du lancer de rayon par évaluation dynamique de la topologie de la scène.
PhD thesis, Université de Franche-Comté, 1996.

[RLF92]F.V. Reeth, W. Lamotte, and E. Flerackers.
Ray tracing speed-up techniques using mimd architectures.
Programming and computer software, 18(4):173–181, July 1992.

[Rot82] S. D. Roth.
Ray casting for modelling solids.
Computer graphics and image processing, (18):109–144, 1982.

[Rum92]Rumeur.
Communications dans les réseaux de processeurs.
Actes de l'école d'été C3, Cargèse, 1992.

[RW80] S. Rubin and T. Whitted.
A three-dimensional representation for fast rendering of complex scenes.
Computer graphics, 14(3):110–116, 1980.

[Sam89]H. Samet.
Implementing ray tracing with octrees and neighbour finding.
Computers and graphics, 13(4):445–460, 1989.

[SC95] Y. Souffez and D. Cuoq.
Optimisation d'algorithmes en dynamique moléculaire (95nj00051).
Technical report, EDF, Direction des Etudes et Recherches, 1995.

[Sim95]G. Simiakakis.
Accelerating ray tracing with directional subdivision and parallel processing.
PhD thesis, University of East Anglia (UK), 1995.

[SK94] C. Silva and A. Kaufman.
Parallel performance measures for volume ray casting.
Proceedings of visualization '94 conference, pages 196–204, 1994.

[SLSA96]K. Sung, J. Loh, Jen Shiuan, and A. L. Ananda.
Ray tracing in a distributed environment.
Computers & graphics, 20(1):41–49, 1996.

[SW91] J.N. Spackman and P.J. Willis.
The smart navigation of a ray through an octree.
Computers and graphics, 15(2):185–194, 1991.

[Val90] L. G. Valiant.
A bridging model for parallel computation.
Communications of the ACM, 33(8), August 1990.

[Wat93] A. Watt.
3D Computer graphics, chapter 8.10 : 'Making ray tracing efficient'.
Addison-Wesley, 1993.

[Whi80] T. Whitted.
An improved illumination model for shaded display.
Communication of the ACM, 23:343–349, 1980.

[Wil94] N. Wilt.
Object-oriented ray tracing.
Wiley, 1994.

[WSC+95] K-Y. Whang, J-W. Song, J-W. Chang, J-Y. Kim, W-S. Cho., C-M. Park, and I-Y. Song.
Octree-r : An adaptative octree for efficient ray tracing.
IEEE transactions on visualization and computer graphics, 1(4):343–349, December 1995.