Conclusions et perspectives

Deux nouvelles représentations

Cette thèse, située dans le cadre de la synthèse d'images de paysages, a été consacrée plus spécifiquement à la représentation des arbres pour le rendu. Les techniques de modélisation d'arbres donnent à ce jour de bons résultats en termes de diversité d'espèces et de formes représentables, mais au prix de la génération de milliers de polygones par arbre, et par conséquent, de milliards pour une forêt. En plus du coût exorbitant de traitement, cette multitude de polygones représente des détails très fins qui, une fois projetés à l'écran, ont une taille souvent inférieure à celle d'un pixel, et posent de gros problèmes d'aliassage. Pour résoudre ces difficultés, l'approche couramment envisagée est l'utilisation de niveaux de détails. Les techniques de simplification de maillage visent à remplacer un ensemble de polygones par un polygone unique. Cependant, l'aspect disparate et non continu de la répartition des feuilles d'un arbre ne permet pas d'appliquer ces méthodes sur un arbre sans en modifier l'opacité et l'illumination globale.

L'idée directrice développée au cours de cette thèse a été de représenter un ensemble de primitives (les feuilles ou les branches) par paquet, tout comme le peintre les factorise par des taches de pinceau (*cf.* Étude de cas). Ceci conduit à des représentations conservant l'aspect visuel, *i.e.* ayant le même comportement vis à vis de la lumière que l'ensemble des primitives remplacées.

En suivant cette idée, nous avons proposé deux approches hiérarchiques. La première, destinée au rendu haute-qualité, est basée sur le calcul analytique du modèle d'illumination d'une géométrie représentant un rameau de conifère. La seconde, destinée au rendu temps réel, est basée sur des ensembles hiérarchiques d'images, pré-calculant toutes les configurations d'illumination et d'auto-ombrage.

Un modèle hiérarchique de shaders analytiques

Nous avons introduit (*cf.* partie II) une hiérarchie de trois *shaders* capables de représenter à une échelle donnée les effets cumulés des niveaux plus fins, sans avoir à les échantillonner, et en prenant en compte l'auto-ombrage et la visibilité. Nous nous sommes servis de la connaissance disponible a priori concernant la distribution géométrique des aiguilles pour calculer analytique-

ment ces caractéristiques visuelles :

- Le premier *shader* est basée sur l'intégration du modèle d'illumination (classique) de Phong sur un cylindre représentant une aiguille.
- Le deuxième *shader* correspond à l'illumination d'un cône d'aiguilles que nous avons modélisé par une distribution continue semi-transparente d'aiguilles du type précédent.
- Le troisième *shader* correspond à l'illumination d'un rameau d'aiguilles que nous avons modélisé par une série de cônes d'aiguilles empilés.

L'aspect analytique de nos *shaders* permet dans le même temps d'accélérer les calculs (notre implémentation, facilement optimisable, est environ 8 fois plus rapide que le système de lancer de rayons *Rayshade* utilisant le sur-échantillonnage pour diminuer l'aliassage) et d'obtenir des images de qualité (en particulier avec peu d'aliassage).

Un modèle hiérarchique à base d'images pour la visualisation temps réel d'arbres

Nous avons introduit (*cf.* partie III) une représentation à base d'images dédiées aux arbres, permettant un rendu de qualité avec des effets complexes comme l'illumination, l'ombrage, la prise en compte de l'illumination du ciel et le mouvement de la source de lumière. Notre implémentation, largement optimisable (notamment avec la nouvelle génération de cartes graphiques grand public), tourne de 7 à 20 images par seconde sur une Onyx^2 Infinite Reality avec une scène comptant 1000 arbres.

Notre représentation se compose d'une hiérarchie de *BTF* (une fonction qui associe une image à chaque couple direction de vue-direction de lumière), que nous affichons à l'aide de *billboards* en interpolant les images. En association à cette hiérarchie, nous construisons une structure précalculée, basée sur des *cubes de visibilité* (*VCM*) qui nous permet de traiter l'auto-ombrage et l'ombrage en temps interactif. Du fait de la hiérarchie et de l'instanciation des données, la consommation mémoire reste raisonnable : quelques dizaines de Mo en mémoire texture pour plusieurs espèces d'arbres et quelques dizaines de Mo en mémoire centrale pour les *VCM* pré-calculés. Il y a un compromis à faire entre quantité de mémoire occupée et qualité visuelle : comme les *BTF* et les *VCM* sont des structures discrètes, plus l'échantillonnage des directions est fin, et plus la qualité du rendu sera élevée, mais plus la quantité de mémoire sera importante. Néanmoins, avec quelques instances d'arbres différents et une quantité mémoire raisonnable nous avons pu produire une scène de 1000 arbres avec illumination, auto-ombrage, ombrage, et avec possibilité de déplacer interactivement la caméra et la source de lumière.

Bilan

Nous avons introduit deux nouvelles représentations traitant la complexité des arbres, en remplaçant un ensemble de primitives (*i.e.* feuilles et branches) par une représentation plus efficace. Ces deux représentations permettent un rendu d'arbres efficace (*i.e.* réaliste et sans aliassage) et rapide : l'une dans le cadre du rendu de qualité et l'autre dans le cadre du temps réel. Ces techniques améliorent fortement la complexité et la qualité visuelle présentes dans les scènes complexes de forêts (*e.g.* auparavant aucun moteur interactif de rendu ne permettait de modifier dynamiquement la lumière).

Perspectives

Augmenter le réalisme des arbres

Il serait intéressant de voir comment l'hypothèse de forte instanciation présente dans notre hiérarchie de *BTF* (nécessaire à la faible consommation mémoire) se comporte avec une grande diversité d'arbres. Il faudrait étudier comment traiter l'instanciation de branches avec des logiciels de modélisation d'arbres plus élaborés [Bio, LD] que notre implémentation des *L-systems*, ce qui permettrait de tester les capacités de notre modèle à grande échelle.

Augmenter le réalisme de nos modèles

Dans notre modèle hiérarchique à base d'images (*cf.* chapitre 6) l'apparence de toute la hiérarchie est basée sur celle de la *BTF* initiale : une *BTF* d'un niveau donné est calculée à partir des *BTF* des niveaux inférieurs. Pour améliorer le réalisme, il serait possible d'utiliser des images réelles de branches à la manière de Max (*cf.* chapitre 2 section 2.3.4). Debevec [DHT+00] utilise un appareillage sophistiqué pour capturer des images de visages humains avec différentes directions de lumière. Cette technique pourrait également être utilisée pour construire la *BTF* initiale à partir de photos, ce qui enrichirait toute la hiérarchie.

En allant plus loin, il serait intéressant d'essayer de reconstruire un *shader* analytique en mettant en correspondance les données d'illumination extraites de ces images et des courbes paramétriques, à la manière de Stam [Sta01] pour son modèle de rendu de peau (*cf.* chapitre 2 section 1.2.1).

Shaders temps réel

Avec l'arrivée des techniques de calcul d'illumination par pixel (*per pixel shading*) sur les nouvelles générations de cartes, il devient envisageable de transférer l'évaluation des *shaders* complexes au matériel de la carte graphique. Dans ce but, l'arrivée des micro-langages de *shader* [PMTH01] (inspirés de celui de *Renderman*) nous aiderait certainement.

L'ombrage de nos *billboards* est calculé aux quatre sommets du polygone, puis interpolé par le matériel graphique lors de la phase de *rasterisation.* Toujours grâce au calcul par pixel, il est sûrement possible d'améliorer cet ombrage, en calculant pour chaque pixel du *billboard* son ombrage exact à partir des cartes de visibilité chargées sur la carte sous forme de texture.

Animation d'arbres

La structure de visibilité que nous avons introduite au chapitre 6 pour l'ombrage est relativement générale, et pourrait être utilisée pour d'autres types de scènes. En revanche, elle ne permet pas l'animation d'objets sans avoir à recalculer les cartes de visibilité. Certains types d'animations doivent être réalisés à moindre coût. Par exemple, il serait intéressant d'essayer d'animer avec peu d'amplitude un arbre au vent sans recalculer les cartes : le résultat est alors inexact, mais il est probable que, visuellement, cette approximation soit acceptable.

Une approche plus correcte serait de ne reconstruire que les cartes qui ont été modifiées par le mouvement de l'objet. Pour ceci, on pourrait s'inspirer de techniques d'animation en radiosité hiérarchique capables de recalculer uniquement les liens modifiés entre les différentes parties d'une scène (*cluster*) lors du mouvement d'une scène.

Familles de shaders dédiées

La nature offre de nombreuses familles d'objets présentant une structure relativement régulière et comportant des similarités. Il devrait par conséquent être possible pour chacune d'entre elles de calculer analytiquement leur modèle d'illumination, comme nous l'avons fait pour les rameaux de conifères.

Pour les feuillus, dont la structure est plus stochastique (concernant la distribution et l'orientation des feuilles), il faudrait envisager un modèle statistique. Dans ce cas, la connaissance a priori prend une forme plus probabiliste, dont l'intégrale est similaire (il faudrait pour cela, obtenir ce type de données auprès de botanistes). Avec ces données, il doit être possible de construire un modèle paramétrable, convenant donc à beaucoup de familles de feuillus.

Représentation temps réel basée sur le point

Au chapitre consacré à l'étude de cas, nous avons vu une technique présente chez les peintres "réalistes", puis développée par les "impressionnistes", consistant à représenter un paquet de feuilles par un "point", en réalité une tache de pinceau.

Durant l'état de l'art, nous avons évoqué une technique temps réel de rendu par points, les *surfels* (*cf.* chapitre 2 section 2.2) qu'il serait intéressant d'appliquer au rendu d'arbres en s'inspirant de la technique des peintres. Un paquet de feuilles peut être représenté par un "point" (un disque à l'écran), que le matériel graphique sait traiter rapidement. Il faut alors trouver une manière de calculer la couleur de ce disque. Il est raisonnablement envisageable d'utiliser un *shader* similaire à celui que nous avons développé au cours de cette thèse pour les rameaux d'aiguilles. Pour être plus générique, il doit être possible de construire un *shader* dans l'esprit de la fonction de réflectance introduite par Neyret [Ney96] pour les textures volumiques (*cf.* chapitre 2 section 2.4) ou celle introduite par Fournier pour les surfaces complexes [Fou92], et d'essayer de l'enrichir en ajoutant une information (éventuellement statistique) d'auto-ombrage.

Peinture évolutive

Avec notre modèle de *shaders* nous disposons d'un outil capable de représenter le comportement photométrique d'un rameau de conifère, et ceci de manière indépendante de l'objet sous-jacent. Il serait alors possible d'utiliser ces *shaders* comme outil de peinture en semi-relief : un coup de pinceau ordinaire, avec comme matériau "aiguilles", permettrait de faire "vivre" le tableau, en modifiant, soient les paramètres locaux (orientation, longueur, densité), soient les paramètres globaux (orientation de la lumière). En outre, il serait envisageable de s'appuyer sur une peinture existante : en recouvrant virtuellement les différentes zones de la toile par un *shader* approprié, il deviendrait possible d'en modifier l'aspect général en changeant les conditions d'illumination. Par exemple, on pourrait simuler l'aspect d'une toile de Waldmüller (*cf.* Étude de cas) au couché du soleil.

ANNEXE A

Moyenne pondérée d'images avec le matériel graphique

Nous voulons effectuer une moyenne pondérée de n images en profitant de l'accélération qu'offre le matériel graphique par rapport aux traitement de cette opération par logiciel. La composition des transparences (*alpha blending*), opération classique du matériel graphique ne donne pas le résultat attendu. De plus notre but est de composer le résultat de cette moyenne avec les données qui sont déjà stocké dans le tampon de couleur et de profondeur.

Nous verrons en 2 que la composition des transparences ne permet pas de calculer la moyenne d'images, puis en 3 et en 4 nous proposerons deux solutions à ce problème. Enfin, en 5 nous conclurons par une considération à propos du coût.

1 Moyenne pondérée d'images

L'objet de cette annexe est de calculer la moyenne d'images $I1, I2, etc.$ dont les pondérations respectives sont $P1, P2, etc.$:

$$M_{rgba} = P1.I1_{rgba} + P2.I2_{rgba} + ... \tag{A.1}$$

Puis d'appliquer le résultat de cette moyenne M_{rgba} au tampon T_{rgba} :

$$T^{final}_{rgba} = (P1 + P2 + ...).M_{alpha}.M_{rgba} + (1 - (P1 + P2 + ...).M_{alpha}).T^{0}_{rgba}$$

La somme des pondérations vaut 1 *i.e.* $P1 + P2 + ... = 1$, donc :

$$T^{final}_{rgba} = M_{alpha}.M_{rgba} + (1 - M_{alpha}).T^{0}_{rgba} \tag{A.2}$$

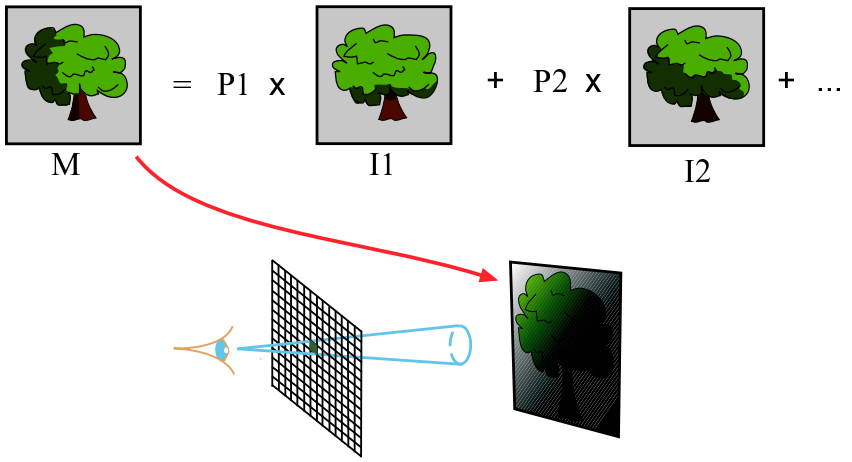

FIG. A.1 – Nous voulons effectuer une moyenne pondérée d'images en profitant de l'accélération qu'offre le matériel graphique.

2 Composition des transparences

Dans cette section, nous calculerons le résultat de la composition des transparences, au sens classique, et nous verrons que ce n'est pas le résultat escompté. A des fins pédagogiques, nous considérons uniquement deux images I1 et I2, de poids respectifs P1 et P2. Le raisonnement effectué ici se généralise facilement. Nous noterons T^k_{rgba} et T^k_{alpha} les composantes RGBA et la composante *alpha* du pixel du tampon après la kème passe.

La composition des transparences classique s'utilise en *OpenGL* avec pour équation de mélange *ADD* et pour coefficients `(SRC_ALPHA,1-SRC_ALPHA)`. Le poids d'une image (P1 ou P2) est transmis à *OpenGL* comme étant la couleur du polygone texturé (*glcolor3f(P1,P1,P1)*). *OpenGL* permet d'effectuer une opération de mélange entre la couleur du polygone et sa texture, grâce à la fonction *glTexEnvf* : nous la configurons à *GL_MODULATE*, ce qui correspond à une multiplication des composantes entre elles.

Après le rendu de la première image les pixels du tampon valent :

$$T^1_{rgba} = (P1.I1_{alpha}).I1_{rgba} + (1 - P1.I1_{alpha}).T^0_{rgba}$$

Puis après le rendu de la deuxième image, nous obtenons :

$$\begin{aligned} T^2_{rgba} &= (P2.I2_{alpha}).I2_{rgba} + (1 - P2.I2_{alpha}).T^1_{rgba} \\ &= (P2.I2_{alpha}).I2_{rgba} + (1 - P2.I2_{alpha}).(P1.I1_{alpha}).I1_{rgba} + \\ &\quad (1 - P2.I2_{alpha}).(1 - P1.I1_{alpha}).T^0_{rgba} \end{aligned} \tag{A.3}$$

En utilisant la composition des transparence (*cf.* équation A.3), nous n'obtenons pas le résultat recherché (*cf.* équation A.2). Nous venons d'effectuer le calcul pour deux images, il est trivial que pour n images le résultat ne sera toujours pas celui recherché. Il faut alors trouver une autre solution.

3 Utilisation d'un tampon annexe

Pour moyenner des images de manière pondérées à l'aide du matériel graphique nous proposons d'effectuer le rendu des images dans un tampon annexe TA (*P-Buffer*) en utilisant une formule

de composition que nous allons décrire maintenant, différente de la composition des transparences. Nous utilisons ensuite ce tampon comme texture d'un polygone représentant le *billboard* . Dans le tampon annexe TA nous calculons $TA_{rgba} = P1.I1_{rgba} + P2.I2_{rgba}$, en utilisant comme équation de mélange *ADD* avec pour coefficients `(SRC_ALPHA,0)`.

Puis nous convertissons l'image résultat de ce calcul en texture, ce qui demande un traitement relativement coûteux au matériel graphique. Cette solution nous offre le résultat attendu, mais est coûteuse, à cause de la conversion du tampon en texture. Dans la section suivante, nous proposons une solution qui n'utilise pas de tampon annexe.

4 Rendu direct

Afin d'éviter l'utilisation d'un tampon annexe, nous proposons une solution adaptée au cas particulier de nos *billboards* et basée sur la séparation de la somme A.2 en deux parties. Nous supposons que toutes les images à moyenner représentent la même géométrie (*i.e.* les valeurs de la composantes *alpha* est identiques sur toutes les images). Ceci est bien le cas de nos *billboards* (*cf.* chapitre 5 section 1.2) lorsque nous composons les images d'illuminations (prises depuis le même point de vue mais avec différentes directions de lumière). Les images prises depuis des points de vue différents que nous composons sont toujours assez similaires (car par hypothèse les trois points de vue sont choisis pour être les plus proches possibles du point de vue de la caméra), leurs valeurs d'*alpha* peuvent donc être considérées comme équivalentes. Ceci se traduit par $M_{alpha} = I1_{alpha} = I2_{alpha} = ...$

Nous séparons donc la somme A.2 en deux parties :

- la partie traitant de ce qui se trouve dans le tampon : $(1 - M_{alpha}).T^0_{rgba}$
  Comme nous avons supposez que toutes les images représentent la même géométrie (*i.e.* les valeurs de la composantes *alpha* est identiques sur toutes les images), cette partie peut être effectuée en une passe en utilisant n'importe laquelle des images à moyenner car $M_{alpha} = I1_{alpha} = I2_{alpha} = ...$
  Pour réaliser cette opération, nous utilisons l'équation de mélange *ADD* en *OpenGL* avec pour coefficients `(0,1-SRC_ALPHA)` et effectuons le rendu d'une image des images (par exemple la première) avec pour coefficient de pondération 1.
- La partie ajoutant les images au tampon :

$$\begin{aligned} M_{alpha}.M_{rgba} &= M_{alpha}.(P1.I1_{rgba} + P2.I2_{rgba} + ...) \\ &= M_{alpha}.P1.I1_{rgba} + M_{alpha}.P2.I2_{rgba} + ... \end{aligned}$$

  Comme $M_{alpha} = I1_{alpha} = I2_{alpha} = ...$, cette partie consiste simplement à ajouter les n images au tampon :

$$M_{alpha}.M_{rgba} = I1_{alpha}.P1.I1_{rgba} + I2_{alpha}.P2.I2_{rgba} + ...$$

  Pour ceci nous utilisons *(SRC_ALPHA,1)* comme coefficients de mélange et nous rendons les n images comme texture d'un polygone ayant pour valeur d'*alpha* la pondération de l'image, comme dans le cas de l'utilisation d'un tampon annexe.

Pour éliminer les interactions dues au test de profondeur, nous le désactivons pour toutes les passes de rendu sauf pour la dernière.

Algorithme A.1 Algorithme

```
Dessine_scène( arrière_vers_avant)
Désactive( écriture_Z)
BlendCoeff( 0 , 1-SRC_ALPHA)
AlphaPolygone( 1 )
Dessine( images[1] )
BlendCoeff( SRC_ALPHA , 1 )
 pour  i ∈ 1..Nimages faire
       | si ( i==Nimages ) alors
       |      | Active(écriture_Z)
       | fin si
       | AlphaPolygone( poids[i])
       | Dessine( images[i])
 fin pour
```

5 Considération de coût et conclusions

La solution utilisant un tampon annexe (*P-buffer*) requière n passes pour construire l'image, la conversion de ce tampon en texture, plus une passe de rendu du *billboard* dans la scène en le texturant de l'image précédemment calculée. Ce qui totalise $n + 1$ rendus, plus surtout une conversion du tampon en texture, ce qui est relativement coûteux.

En revanche, notre solution alternative ne coûte que $n + 1$ passes : une passe pour traiter la composante *alpha* et n passes pour le traitement classique des images. Nous évitons donc la conversion du tampon en texture.

Remarque : L'objet de cette annexe n'est pas nécessaire dans le cas de l'utilisation d'une carte capable de traiter plusieurs textures en une passe (*multi-texturing*[1]).

[1]La fonctionnalité de *multi-texturing* n'est pas présente sur $Onyx^2$ Infinite Reality, mais est disponible sur *Nvidia GeForce* 2 et 3.

ANNEXE B

Lancer de cônes parallèle utilisant une mémoire partagée distribuée sur grappe de PC SCI [MC01]

Comme nous l'avons vu dans l'état de l'art, le lancer de rayons (et donc le lancer de cônes) reste un algorithme coûteux. Les nombreuses optimisations existantes sont poussées à bout, pour diminuer encore les temps de calcul il est intéressant de paralléliser. D'autant plus, que l'algorithme de lancer de rayons se parallélise très simplement.

Le but de cette annexe n'est pas de présenter un nouveau schéma de parallélisation du lancer de rayons, de nombreux travaux portent sur ce domaine et de nombreux algorithmes efficaces existent [WDP99, Hai]. Notre but est de comparer deux types d'architectures, en utilisant l'algorithme de lancer de rayons comme repère : les architectures parallèles et les grappes d'ordinateur.

La machine cible sur laquelle nous avons initialement développé notre lancer de cônes (variante du lancer de rayons) est une machine parallèle : Onyx^2 Infinite Reality disposant de 6 processeurs et 4 Go de mémoire. Cependant, les machines parallèles coûtent cher (l'*Onyx* n'échappe pas à cette règle), plus qu'un réseau de machines disposant au total du même nombre de CPU et de la même quantité de mémoire, donc théoriquement de la même puissance de calcul. L'avantage des machines parallèles est de partager directement les ressources (mémoires, disques, etc.), alors que les réseaux de machines partagent leurs ressources en utilisant le réseau. Cependant, avec l'augmentation des débits des réseaux et la création de nouvelles technologies accélérant les communications [SIR, HEB+01], la question de l'utilisation de grappes de machines en remplacement des machines parallèles se pose.

Afin, d'effectuer cette comparaison entre machine parallèle et grappe de machine, l'équipe *SIRAC* a mis à notre disposition une grappe expérimentale de 12 machines fonctionnant sous le système d'exploitation *Linux*. Le portage de notre lancer de cônes parallèle sur cette grappe a été,

du même coup, l'occasion pour eux de tester en grandeur nature la pertinence de leur choix de réseau et de leur implémentation des outils le partage des ressources.

Nous verrons rapidement en 1 le schéma de parallélisation classique que nous utilisons, nous décrirons en 2 les particularités de cette grappe expérimentale de PC/Linux, puis nous donnerons en 3 les détails utiles (issus de notre expérience) au développement d'applications sur ce type de grappe, et nous conclurons en 4 par des résultats comparatifs entre les deux architectures.

1 Lancer de cônes parallèle

Notre parallélisation de l'algorithme de lancer de rayons suit un schéma classique [CPC98], qui consiste à découper l'image ou l'animation en sous-parties qui sont distribuées entre les processeurs ou les machines. Nous disposons de deux schémas de parallélisation :

- Nous décomposons une animation en images, le calcul de chaque image est distribué sur les différents processeurs. Les images résultats de chaque processus sont sauvées sur un disque partagé par *NFS* (*cf.* figure B.1 à gauche).
- Nous décomposons une image en sous-images, le calcul de chaque partie est distribuée sur les différent processeurs. Un processus particulier s'occupe de collecter les sous-parties de l'image, puis sauve le résultat final sur le disque (*cf.* figure B.1 à droite).

Dans les deux cas tous les processus partagent les données de la scène, seuls les objets se déplaçant sont dupliqués (*i.e.* la caméra).

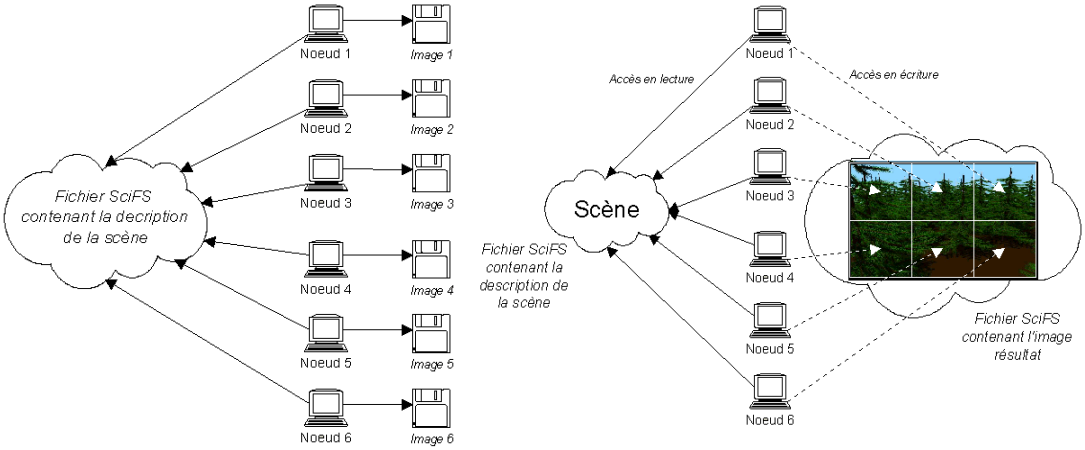

FIG. B.1 – Les deux schémas de parallélisation.

Ces deux schémas de parallélisation sont efficaces : peu de données sont partagées en écriture, ce qui limite le nombre de verrous[1] à utiliser et donc les attentes potentielles. La scène est partagée par tous les processus et est accédée uniquement en lecture (à part lors du chargement initiale), nous n'utilisons aucun verrou pour cette partie des données. Pour le calcul parallèle d'une animation, notre implémentation ne partage aucun segment de mémoire (les images étant indépendantes), donc aucun interblocage n'est possible : cas idéal. En revanche, pour le calcul distribué d'une image, la zone de données lui correspondant est partagée en écriture, ce qui impose l'utilisation d'un verrou.

[1] Un verrou permet d'autoriser l'accès à une ressource à un nombre limité de tâches. Les tâches demandants une ressource non disponible sont mises en attente.

Dans un but de comparaison coût/performance entre une machine parallèle ($Onyx^2$ Infinite Reality) et une grappe de PC, nous avons testé ces deux schémas de parallélisation sur les deux architectures. Avant d'exposer les résultats de cette comparaison, nous présentons les spécificités de la grappe de PC utilisée.

2 Grappe de PC et SciFS

La grappe utilisée dispose de 12 Pentium II (450 MHz) équipée du réseau *SCI* facilitant le partage de la mémoire. Apres un survol des capacités qu'offre le réseau *SCI* en 2.1, nous verrons en 2.2 le principe de fonctionnement de la mémoire partagée distribuée logicielle *SciFS* développé par les membres de l'équipe *SIRAC*.

2.1 La technologie SCI

La technologie réseau *Scalable Coherent Interface* (*SCI*) permet de lire ou d'écrire dans la mémoire de n'importe quelle autre machine du réseau sans interrompre le processeur de la machine distante et ceci par deux moyens :

- Le processeur demandant effectue sa requête à sa carte *SCI* par entrée/sortie programmées, puis attend la réponse.
- Le processeur demandant effectue la requête à sa carte *SCI* par émission d'une requête DMA, puis reprend la main jusqu'à ce que la carte *SCI* émette une interruption l'informant du résultat.

Dans les deux cas la cartes *SCI* de la machine demandante transmet la requête à la carte *SCI* de la machine distante, qui interroge la mémoire (en passant par le bus) sans interrompre son processeur, puis transmet le résultat.

Cette technologie permet un partage puissant des ressources mémoire au sein d'une grappe de PC. Un accès à la mémoire d'une autre machine à travers la technologie *SCI* est plus lent qu'un accès direct à la mémoire locale, mais considérablement plus rapide que si cette requête était traitée par un système de *socket*. Cette technologie a conduit des membres de l'équipe *SIRAC* à développer une mémoire partagée distribuée (*DSM*) logicielle : *SciFS*.

2.2 Une mémoire partagée distribuée logicielle basée sur un réseau SCI

La mémoire partagée distribuée (*Distributed Shared Memory DSM*) logicielle, baptisé *SciFS*, développé par l'équipe *SIRAC*, et plus particulièrement par Emannuel Cecchet durant sa thèse [Cec01, KHCR99, SIR] se présente sous la forme d'une extension (module) au noyau de *Linux*. Elle permet aux développeurs de "haut niveau" (en l'occurrence nous) de profiter pleinement des capacités de la technologie *SCI* sans avoir à se plonger dans les couches bases du système (*cf.* figure B.2).

Concrètement, cette *DSM* permet de se servir des $n \times 256Mo$ de mémoire des n machines de la grappe comme si il s'agissait d'une mémoire unique, et ce de manière transparente. Le module de *DSM* se charge de répartir les données sur les différentes machines et éventuellement de les déplacer ou de les dupliquer en fonction des statistiques d'accès aux données.

Dans la pratique, cette mémoire partagée s'utilise comme un fichier : une fois crée, le fichier mémoire peut être ouvert depuis n'importe quelle machine du réseau, et permet ainsi, à n'importe quel processus d'accéder aux données en lecture comme en écriture. Un système de verrous est mis à disposition des développeurs pour gérer les accès concurrents aux données.

Pour limiter les accès à distance qui, bien que rapides, sont plus coûteux que les accès locaux, *SciFS* implémente plusieurs techniques [KCR98]. Nous citerons les trois couramment utilisées :

- *Fixed* : la machine qui crée le fichier de mémoire partagée le garde en local jusqu'à sa libération.
- *First touch* : la première machine à accéder aux données du fichier de mémoire partagée le garde en local jusqu'à sa libération.
- *Global* : les données de la mémoire partagée sont dupliquées sur toutes les machines y faisant des accès nombreux en lecture. Un système de jeton est utilisé pour l'écriture : la machine disposant du jeton possède les droits en écriture, cela signifie que les écritures effectuées par les autres machines lui sont transmises, puis les données dupliquées sur les autres machines sont mises à jour. En suivant les statistiques, la machine effectuant le plus d'écritures dispose du jeton.

Le développeur de haut niveau fournit en options une de ces trois techniques lors de la création du fichier de mémoire partagée, puis ne s'en occupe plus.

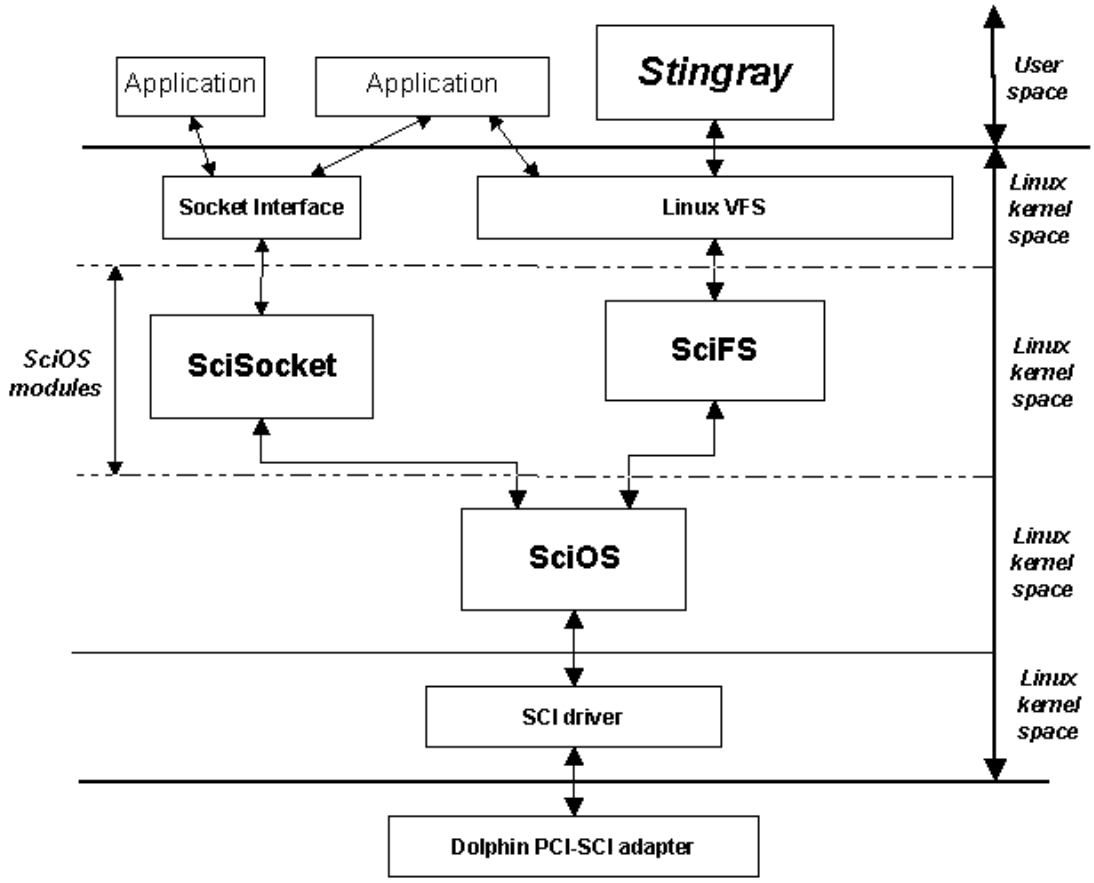

FIG. B.2 – Notre lancer de cônes parallèle est implémenté sur une grappe de PC équipée d'un noyau *Linux* étendu par le module *SciFS*.

3 Développement d'une application sur grappe SciFS

L'objectif de cette section est de discuter de l'utilisation de la *DSM SciFS* avec notre lancer de cônes parallèle et de donner des informations techniques aux futures développeurs amener à l'utiliser.

3.1 Répartition des données

Dans nos deux schémas de parallélisation (*cf.* section 1), nous utilisons l'option *Global* pour la zone mémoire contenant la scène et, l'option *Fixed* pour l'image. Avec la technique *Global*, le système *SciFS* duplique les données de la scène sur les machines accèdent en lecture aux segments partagés[2]. Le calcul se fera alors, sans pratiquement aucun surcoût dû à la répartition des données, puisque chaque processus accède à des données qui sont dupliquées en locale. Nous fixons le segment de l'image sur la machine du processus collecteur de données. Afin de ne pas surcharger le réseau en requête de petite taille, nos processus de calcul stockent leurs résultats dans un tampon et l'envoient au processus collecteur une fois qu'il est plein[3].

3.2 Implémentation

L'implémentation de notre lancer de cônes sur *SciFS* a été rapide (environ 2 semaines), aucun changement n'a été nécessaire au niveau des schémas de parallélisation. Par contre des problèmes techniques non triviaux ont été à résoudre. Cette section a pour but de faire gagner du temps aux futures développeurs ($C++$) amener à utiliser *SciFS* :

- Gestion mémoire : la fonction *new* (C++) doit être surchargée pour allouer un segment partagé (*i.e.* ouvrir un fichier sur le disque). Une nouvelle classe *allocateur* pour *STL* doit être écrite.
- Processus légers et processus lourds : l'utilisation de processus légers (*thread*) ne peut plus se faire puisque chaque processus tourne sur des machines différentes. Nous les avons remplacés par des appels systèmes créant un processus lourd (*fork*) exécutant une commande à distance de type *rsh*.
- Initialisation des données : la table de symboles utilisée par un programme écrit en C++ (*VTBL*) ne se trouve pas dans un segment de données partagées. Il faut que chaque processus chargent la scène dans le segment de mémoire partagée, ce qui est redondant, mais qui permet d'initialiser correctement la *VTBL* de tous les processus. Ce chargement est rapide et n'est effectué qu'une fois au début du programme.

De plus amples informations technique à propos de *SciFS* peuvent être trouvées dans [SIR].

4 Performances et conclusions

Nos deux plates-formes de tests ont été une $Onyx^2$ Infinite Reality équipé de 6 processeurs R12000 à 400 MHz avec 4 Go de mémoire, et une grappe de 12 PC Penium II à 450 MHz avec 512Mo de mémoire chacun (totalisant 6 Go de mémoire partagée). La parallélisation utilisée offre, sur les deux plates-formes, des performances linéaires dans le cas d'une animation et quasi-linéaire

[2] Les tailles de nos données sont toujours inférieures à la mémoire de chaque machine, donc cette duplication ne pose pas de problème au système.

[3] La taille de paquets donnant un résultat optimal est de 4Ko, la taille de nos tampons est donc identique, mais ceci est paramétrable.

dans le cas d'une image (sur la grappe le temps de calcul est 5.9 fois plus rapide avec 6 processus qu'avec un). Ces résultats sont dus au faible partage des ressources, donc à l'utilisation de peu de verrous, qui sont les principales sources d'attente et de ralentissement. En performances absolues, l'*Onyx* avec 6 processus et environ 30% plus rapide que 6 processus répartis sur six PC de la grappe[4]. En revanche, en profitant des capacités maximales de la grappe (soit 12 machines), les temps lui deviennent favorables (30% plus rapide que l'*Onyx* à 6 processeurs). Avec 12 processeurs, il est évident que l'*Onyx* repasserait devant, mais ceci se ferait au prix d'un coût financier élevés. Une *Onyx* avec 6 processeurs et 4 Go de mémoire (sans carte graphique) coûte environ 1MF alors qu'une grappe de 12 PC équipés de cartes *SCI* (sans écran) coûte environ 250KF, soit quatre fois moins.

Dans le cas de notre application particulièrement bien profilée à l'utilisation d'une grappe (peu de partage de données), il est financièrement plus avantageux d'utiliser une grappe de PC qu'une machine parallèle, et ce pour des performances équivalentes. Il est évident que cette affirmation n'est pas généralisable, et dépend du type d'application développé mais les résultats de cette annexe prouve qu'il est utile d'investiguer la cette direction de grappe de machines.

[4]Un processeur R12000 à 400MHz est plus rapide qu'un Pentium II à 450MHz au niveau des opérations flottantes, ce qui explique, en grande partie, que l'*Onyx* est plus rapide que la grappe en performance absolue.