UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN ÉLECTRONIQUE INDUSTIELLE

PAR
PATRICE HAMELIN

INTERFACE PARALLÈLE POUR L'ANALYSE NUMÉRIQUE DES PHÉNOMÈNES ÉLECTROTHERMIQUES

AVRIL 1998

Résumé

L'affichage graphique est une tâche qui consomme énormément de mémoire et de temps processeur. Les stations de travail monoprocesseur les plus performantes mettent plusieurs secondes et même plusieurs minutes pour effectuer l'affichage d'un graphique de très grande dimension provenant des calculs de simulation électromagnétique. Dans cette recherche, nous allons tenter de diminuer ce temps d'affichage en employant différents ordinateurs parallèles, et en programmant une interface graphique parallèle. Nous allons présenter les résultats obtenus avec une application typique et ces résultats seront comparés et analysés.

Plusieurs méthodes numériques sont utilisées en science aujourd'hui pour simuler des phénomènes physiques prévoyant ainsi leurs conditions d'opération. La méthode des éléments finis (FEM) et la méthode des différences finies dans le domaine du temps (FDTD) sont des exemples de méthodes numériques. Elles sont implantées dans les logiciels de simulation électromagnétique qui sont utilisés par les ingénieurs afin de concevoir et d'optimiser leurs réalisations. De façon générale, ces logiciels produisent des données de très grande taille, spécialement dans le domaine des hautes fréquences où la résolution doit être très élevée, générant ainsi un grand nombre de cellules différentes à calculer et à afficher.

Nous allons utiliser le mot "biochamps" pour décrire les champs électriques, magnétiques et thermiques en interaction avec les tissus vivants.

Remerciements

Il y a trois ans, lorsqu'on m'a approché pour la réalisation de cette maîtrise, je n'aurais jamais pensé pousser jusqu'à cette limite ma connaissance de la programmation parallèle. Ce projet est le fruit d'un travail intense pendant plus de deux ans car la première année du projet d'analyse parallèle des biochamps, d'une durée totale de trois ans, a été vouée entièrement à l'installation des machines parallèles dans notre laboratoire, et à l'étude de leur fonctionnement et de la programmation parallèle.

Je voudrais remercier sincèrement le professeur Adam Skorek pour la confiance qu'il m'a toujours montrée en tant que responsable du laboratoire des biochamps ainsi qu'en tant qu'étudiant sous sa supervision, et aussi pour m'avoir permis de réaliser cette maîtrise dans les conditions sous lesquelles je l'ai réalisée. J'entend par "conditions" la disponibilité des locaux, le confort et l'ambiance de travail sans pareille et la mise à ma disposition d'équipements de pointe pour la réalisation du projet.

Remerciements au professeur Marek Zaremba qui a agit à titre de codirecteur de recherche et qui à mis a ma disposition son ordinateur AVX3 à l'Université du Québec à Hull.

Remerciements aussi à M. Guy Boisclair qui m'a aidé dans l'élaboration de la structure de base du graphisme par ordinateur. Son expérience ainsi que sa disponibilité m'ont aidé à établir les bases de ce projet.

Remerciements finalement à ma famille pour avoir supporté mon stress à la veille d'une présentation ou de la remise d'un travail.

Table des matières

Liste des illustrations

Liste des tableaux

1 Introduction

Ce projet vise le développement d'un logiciel de visualisation des résultats des calculs, dans le cadre du projet d'analyse parallèle des biochamps. Ce logiciel, appelé aussi interface graphique ou "post-processeur", devra exploiter toutes les capacités des ordinateurs parallèles, afin d'assurer un temps de réponse optimal dans l'affichage des résultats, et il devra faire appel aux capacités graphiques des postes de travail afin de le rendre simple d'utilisation. Il devra effectuer l'affichage graphique des résultats de simulation obtenus par les modules de calcul à hautes fréquences et à basses fréquences du projet d'analyse parallèle des biochamps. Le mot "biochamps" fait ici référence aux champs électriques, électromagnétiques et thermiques en interaction avec les tissus du corps humain.

L'implantation parallèle d'un tel outil graphique implique un certain nombre de problèmes tels que la distribution des tâches entre les processeurs, et la synchronisation, problèmes qu'on ne rencontre pas dans la programmation séquentielle. Aussi, la possibilité de portabilité étant limitée par la topologie des différentes machines parallèles et le standard de passage de messages utilisé, une approche très structurée du développement du logiciel a été prise en compte, afin d'assurer une portabilité optimale. Finalement, une attention particulière doit être portée à la communication entre le module de calcul et le post-processeur, ce qui pourrait bien être le facteur de limitation de la vitesse d'affichage.

Le traitement parallèle est une technologie moderne qui va changer d'une manière substantielle la conception des systèmes informatiques. L'expansion d'ordinateurs

parallèles va conduire à réévaluer la plupart des algorithmes usuels en fonction de nouveaux critères de viabilité et de performance. Pour mettre en parallèle un algorithme, on commence par segmenter le problème en sous-tâches. Reste alors à affecter les tâches aux processeurs, en respectant les contraintes matérielles liées à l'architecture de la machine. En général, ces contraintes sont de deux types: accès limité aux données et problèmes de synchronisation des processeurs.

Dans un premier temps, les outils nécessaires au bon déroulement du projet doivent être mis en place; ceci consiste en l'installation, l'essai et l'optimisation des outils de programmation et de compilation parallèle. Des tests de performance seront effectués afin de déterminer la configuration optimale des processeurs.

Lors du développement du modèle numérique de l'analyse parallèle des biochamps, un support sera apporté au développement du modèle afin de bien synchroniser la réalisation des différents modules sur une topologie parallèle commune, ainsi que sur une base de données commune. Ceci permettra l'acquisition d'une connaissance de base de la théorie des éléments finis. Suite à ces étapes préparatoires, on pourra passer à la conception proprement dite d'un prototype du post-processeur. Des essais seront effectués afin de déterminer la fiabilité du produit et sa rapidité, suite à quoi il y aura certaines modifications apportées. Ces étapes conduiront à la mise en place d'une version exécutable et utilisable du post-processeur. La dernière étape consistera à documenter ce produit afin de rendre les utilisateurs autonomes dans leur apprentissage du produit.

Dans le présent document, nous allons voir d'abord comment les post-processeurs commerciaux sont construits et comment ils opèrent. Il n'y a pas lieu de comparer le résultat de la présente recherche avec les logiciels commerciaux car les ressources

impliquées dans la conception ne sont pas du tout les mêmes. Dans un deuxième temps, nous allons faire une brève étude des ordinateurs parallèles, et principalement ceux qui ont été impliqués dans la recherche décrite dans ce document. La théorie du graphisme par ordinateur devra être maîtrisée afin de réaliser ce projet et c'est pourquoi le chapitre 4 se consacrera à cette théorie. Le chapitre 5 quant à lui fera une description détaillée de la méthode de programmation utilisée dans la réalisation de l'interface graphique, et cela en trois volets. Le standard graphique Motif a été choisi à cause de sa grande popularité auprès des fabriquants de stations de travail et il sera décrit dans une premier temps. Ensuite, on verra comment l'interface usager a été conçue, et finalement comment l'implantation graphique s'est faite. Le chapitre 6 fera la description de la méthode de mise en parallèle utilisée pour réaliser les buts de ce projet. Le chapitre 7 fera état des résultats obtenus lors des essais effectués sur différents types d'ordinateurs séquentiels et parallèles. Les résultats seront compilés et interprétés afin de mener vers une conclusion concernant l'efficacité du logiciel et des machines parallèles impliquées dans ce projet.

Certains chercheurs se sont penchés au cours des dernières années sur le sujet du graphisme utilisant les ordinateurs parallèles. Depuis 1993, plusieurs librairies graphiques ont été développées afin de rendre simple la réalisation de logiciel de graphisme par ordinateur [1][2][3][4]. Toutes ces recherches font état de logiciels ou librairies écrites sous différents types d'ordinateurs parallèles allant des plus simples aux plus complexes. Quoique des plus intéressantes, ces recherches ne peuvent pas être utilisées ou appliquées au projet d'interface graphique discuté dans ce document. L'absence de portabilité est un problème rencontré fréquemment dans le monde de la programmation parallèle, et même en séquentiel, la portabilité peut devenir un problème si le programmeur ne respecte pas

certaines règles de programmation précises. Par exemple, la taille des différents types de variables peut être différente d'un processeur à l'autre. C'est pourquoi le standard Motif est respecté dans ce travail. D'ailleurs, la partie séquentielle de l'interface graphique peut tourner sur SUN, HP et IBM. C'est aussi pourquoi en cours de projet il a été décidé de tenter de passer vers le standard MPI afin d'atteindre un certain niveau de compatibilité qui permettrait à l'interface graphique de tourner sur un réseau de stations de travail aussi bien que sur une machine parallèle telle que l'AVX3 de Alex.

2 Les post-processeurs commerciaux

2.1 Introduction

Il existe sur le marché une grande quantité de logiciels servant à effectuer l'analyse numérique des champs électriques, magnétiques et thermiques. Tous ces logiciels comportent généralement un module de pré-traitement, ou encore le pré-processeur, qui sert d'abord à concevoir le géométrie à étudier. À l'aide d'une interface DAO (Dessin Assisté par Ordinateur), l'utilisateur peut effectuer une modélisation du phénomène physique à étudier, par exemple, un transformateur. Il est préférable d'exploiter la symétrie des structures puisque l'analyse en trois dimensions requiert énormément d'efforts de conception et de temps de calcul. C'est pourquoi on utilise très souvent une partie de la structure à étudier en deux dimensions, rendant plus simple la conception et la réalisation de l'étude. Le pré-processeur a comme deuxième tâche de discrétiser la structure à étudier en petits éléments qui serviront dans les calculs par les méthodes numériques standards telles la méthode des éléments finis (FEM) ou la méthode des différences finies dans le domaine du temps (FDTD). Chacun de ces éléments doit posséder des caractéristiques physiques au niveau des matériaux. Par exemple, pour effectuer une étude thermique, on doit connaître la chaleur spécifique, la masse volumique et la conductivité thermique du matériau. Finalement, le pré-processeur doit imposer les conditions aux limites qui sont à l'origine du champ thermique, électrique ou magnétique qui est l'objet de l'étude.

Suite à l'établissement des conditions mentionnées ci-haut, le pré-processeur lance la balle au module de calcul qui, selon les données en provenance du pré-processeur,

effectue le calcul de la distribution du champ étudié. De façon générale, dans le domaine des hautes fréquences, l'utilisation de la méthode des différences finies est mieux adaptée, et dans le domaine des basses fréquences, l'utilisation de la méthode des éléments finis est de mise. Dans les deux cas, les modules de calcul produiront une série de nombres représentant les résultats de l'étude voulue. Mais l'interprétation de ces résultats, dans cette forme, est pratiquement impossible pour les structures complexes comprenant plusieurs milliers d'éléments différents. C'est à ce niveau qu'intervient le module de post-traitement (post processing). Il devra lire ces données et les afficher sur un écran graphique de façon à ce que les résultats puissent être interprétés facilement par l'utilisateur. L'utilisation de la couleur et des transformations graphiques tels que le zoom, la rotation et la translation permet aussi d'aider à interpréter les résultats.

Nous allons voir de façon brève comment différents post-processeur effectuent leur travail afin de donner un aperçu de l'éventail des logiciels disponibles sur le marché, et comment le travail effectué dans ce projet répond aux exigences de la réalisation d'un post-processeur. Il n'y a pas lieu de faire une comparaison stricte du post-processeur réalisé ici avec ceux réalisés par les grandes compagnies puisque les ressources ne sont pas les mêmes. Le travail présenté dans ce document a été réalisé par une seule personne tandis que les grandes compagnies comptent sur toute une équipe de programmeurs chevronnés. Mais il est intéressant de voir ce qui existe et jusqu'où peut aller la réalisation d'un projet dans lequel on est impliqué.

2.2 *Flux2D*

Le logiciel Flux2D est subdivisé en divers modules effectuant une tâche spécifique; c'est d'ailleurs son principal défaut. Parmi ceux-ci, on retrouve le module de post traitement EXPGEN à l'aide duquel on exploite les résultats. Il permet le calcul des grandeurs locales et globales en plus de la production de graphiques et d'images des potentiels et des champs [5]. Ses possibilités comprennent:

- Courbes équipotentielles
- Dégradé de couleurs des variables pertinentes
- Courbe de la valeur d'une variable
- Visualisation de valeurs à des points spécifiques
- Calcul de différentes valeurs spécifiques à un point ou à une région
- Enregistrement de fichiers des valeurs, etc...

Voici quelques exemples des capacités d'affichage de Flux2D.

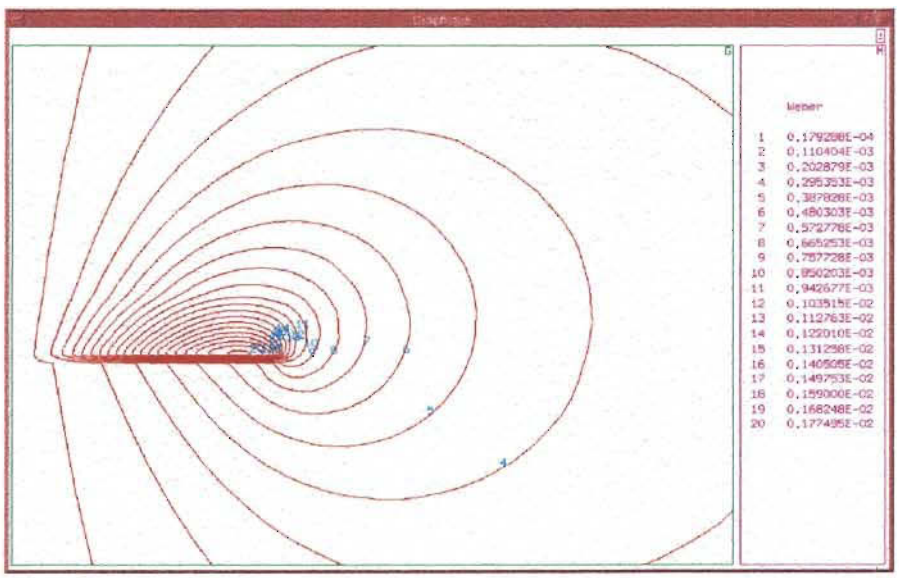

Figure 2.1: Affichage des lignes de flux magnétique en mode axisymétrique sur Flux2d.

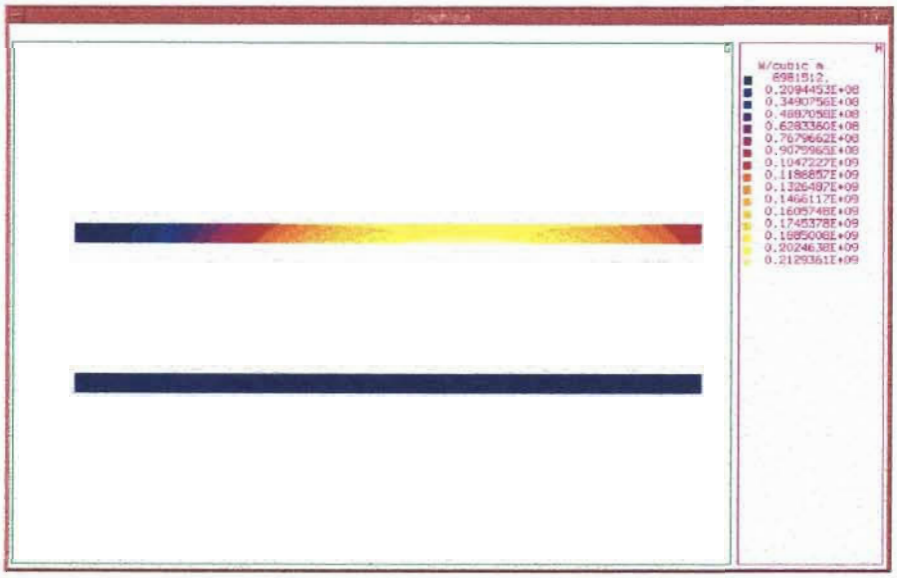

Figure 2.2 : Densité de puissance sur Flux2d.

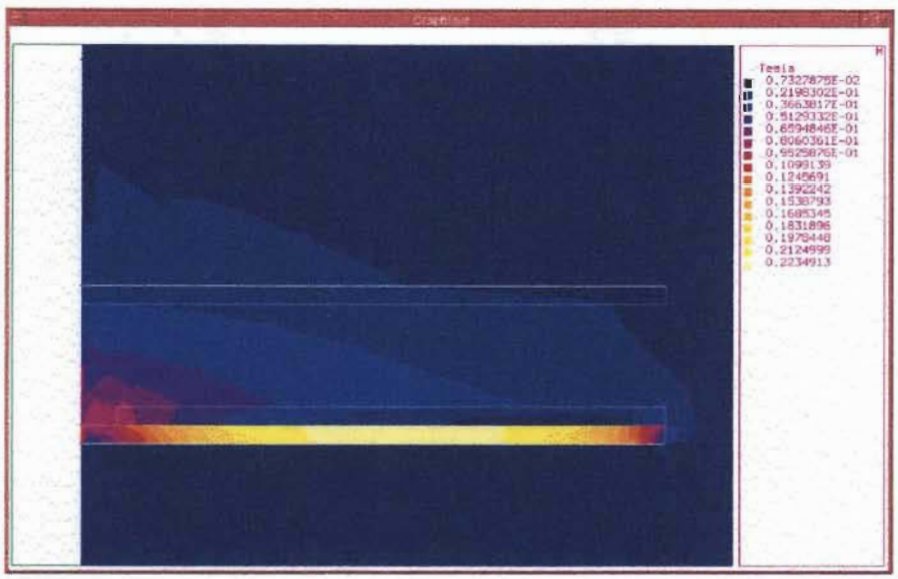

Figure 2.3 : Induction magnétique sur Flux2d.

Le module de post-traitement de Flux2D possède certains défauts. Par exemple, il possède une fenêtre graphique et une fenêtre de commande. Le passage entre ces deux fenêtres n'est pas coulé, c'est-à-dire que l'on doit souvent sélectionner une de ces fenêtres avec l'aide de la souris, ce qui constitue une perte de temps. Par contre au niveau de l'étude proprement dite il effectue de bonnes analyses en deux dimensions et présente des résultats

précis et interprétables aisément. Il ne respecte pas le standard Motif ce qui fait que son interface ne se fond pas dans le reste de l'affichage XWindow. Les menus sont assez simples à utiliser pour quelqu'un qui ne le connaît pas. La documentation quant à elle est plutôt pauvre.

2.3 Flux3D

Le logiciel Flux3D est le pendant.en trois dimensions de Flux2D, tel que leurs noms l'indiquent. Flux3D est beaucoup plus intégré, c'est-à-dire que tous les modules sont accessibles par une seule et même fenêtre graphique. Les modules ne sont pas tous séparés dans plusieurs programmes exécutables. Flux3D calcule les champs électromagnétiques dans des géométries trois dimensions par l'utilisation de la méthode des éléments finis.

Flux3D offre plusieurs options pour l'interprétation des résultats de calcul des champs :

- Représentation des valeurs locales telles que le potentiel scalaire ou le potentiel vecteur, le champ magnétique ou la densité de flux. Ces valeurs peuvent être représentées sur des faces ou dans une coupe de la géométrie par des vecteurs ou par des dégradés de couleurs.
- Tracés de courbes des variations de ces valeurs dans le temps.
- Calcul des variables globales telles que la force ou le couple, et la puissance active ou réactive.

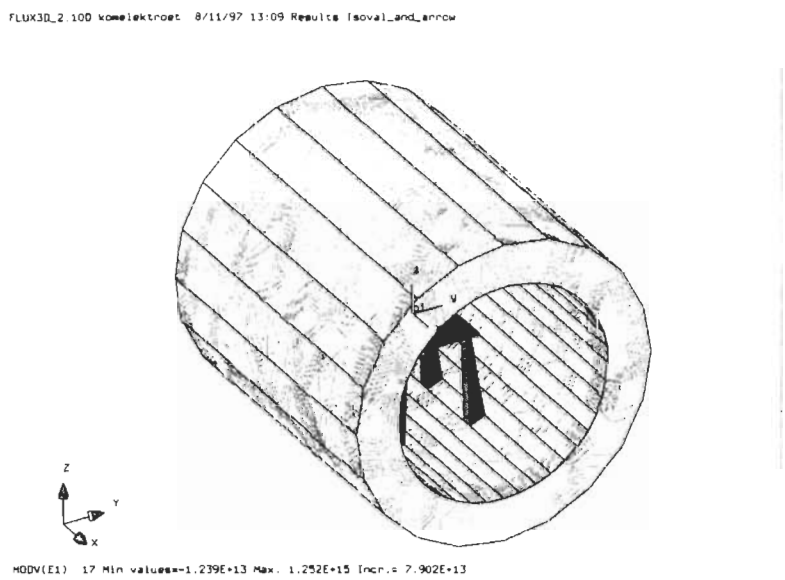

Figure 2.4: Exemple d'affichage du post-processeur de Flux3D.

2.4 NISA

La famille des produits de NISA consiste en une variété de modules d'analyse numérique capables de solutionner virtuellement tous les problèmes d'analyse en ingénierie. Elle comporte entre autres le module DISPLAY III qui est le programme de pré- et post-traitement en trois dimensions. Dans NISA, ces deux modules sont regroupés en un seul et les différents modules de calcul sont constitués de programmes distincts. NISA dans son ensemble est très versatile et très performant, quoiqu'il ne s'identifie pas au standard Motif. Il représente une meilleure intégration que le cas de Flux2D et Flux3D. Le programme est très convivial et les menus sont complexes, permettant une grande puissance. Il peut afficher en couleurs les résultats des modules de déformation, de contrainte, de température, de pression et autre. Les analyses électriques, magnétiques, statiques et thermiques sont possibles avec NISA.

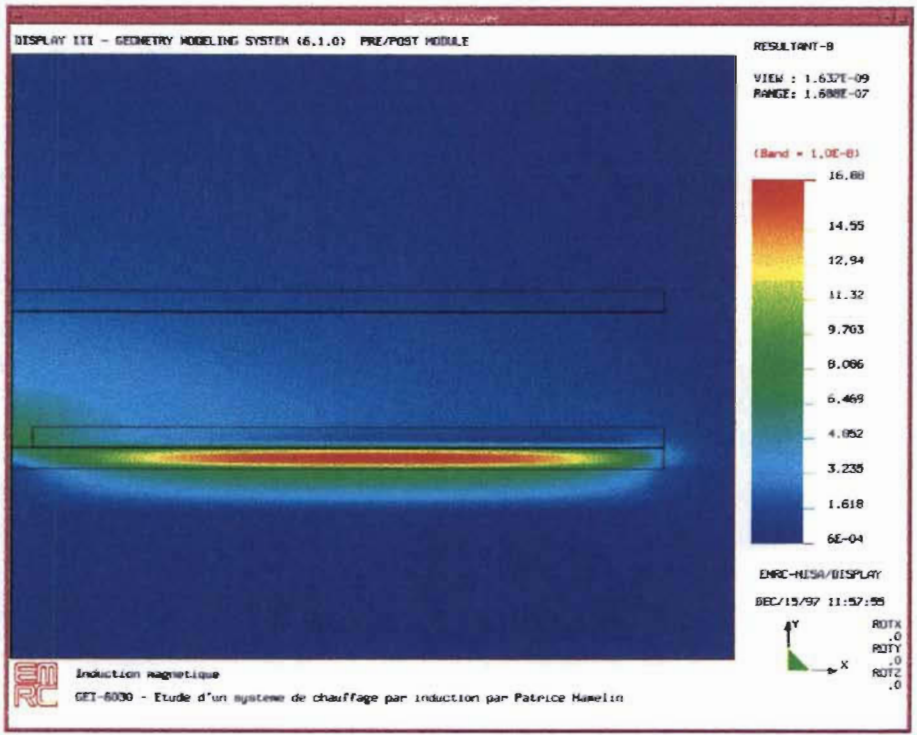

Figure 2.5 : Distribution de l'induction magnétique calculée par NISA.

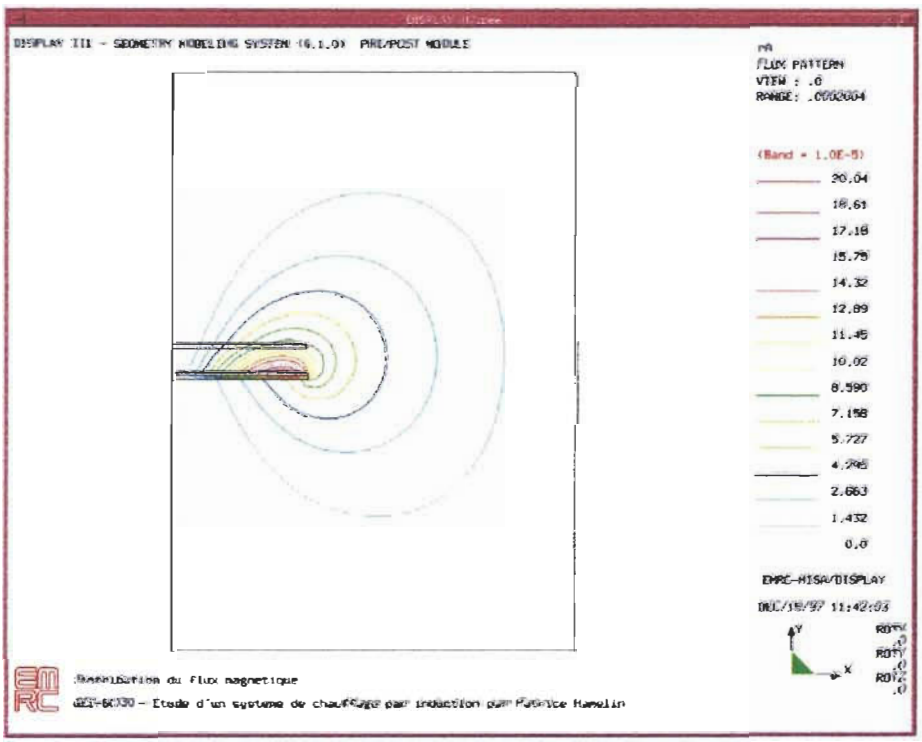

Figure 2.6 : Distribution du flux magnétique sur NISA

2.5 JMAG

Le logiciel JMAG est la dernière trouvaille du laboratoire d'électrothermie industrielle de notre université. Écrit par le Japan Research Institute, il est le fruit de plus de dix ans de travail et de recherche acharnés. Connaissant le travail inlassable de nos confrères japonais, il n'est pas étonnant qu'il soit de loin le meilleur logiciel d'analyse numérique à avoir été en notre possession. Il comporte un excellent module de post traitement qui effectue l'affichage des résultats sous forme de contours de couleurs ou de lignes, de vecteurs, tout cela avec possibilité d'animation. Il est possible d'afficher le flux magnétique, la densité de courant, les champs électriques, magnétiques et thermiques, et autres grandeurs. JMAG fournit aussi des outils graphiques permettant une analyse plus avancée des données comme par exemple la transformée de Fourier, l'intégration, la différentiation, la moyenne et le filtrage. Les données peuvent être importées ou exportées pour fin de comparaison ou d'analyse avec les données expérimentales par exemple. Son interface graphique est très complète et conviviale; elle respecte aussi le standard Motif ce qui lui permet de rouler aisément sur à peu près n'importe quelle plate-forme comme les stations de travail SUN, HP, SGI, DEC ou IBM et les PC sous Windows NT. Il est aussi possible d'extraire les résultats en format Postscript ou ASCII pour fin d'impression.

Au moment d'écrire ces lignes, JMAG est à l'essai par plusieurs étudiants à la maîtrise afin de bien comprendre son fonctionnement et ses possibilités. Le JRI fournit un support sans pareil, ce qui n'est pas le cas des autres compagnies dont les produits ont été mentionnés dans ce chapitre. Le seul problème rencontré pour l'instant est le manque de documentation qui n'est disponible qu'en japonais!

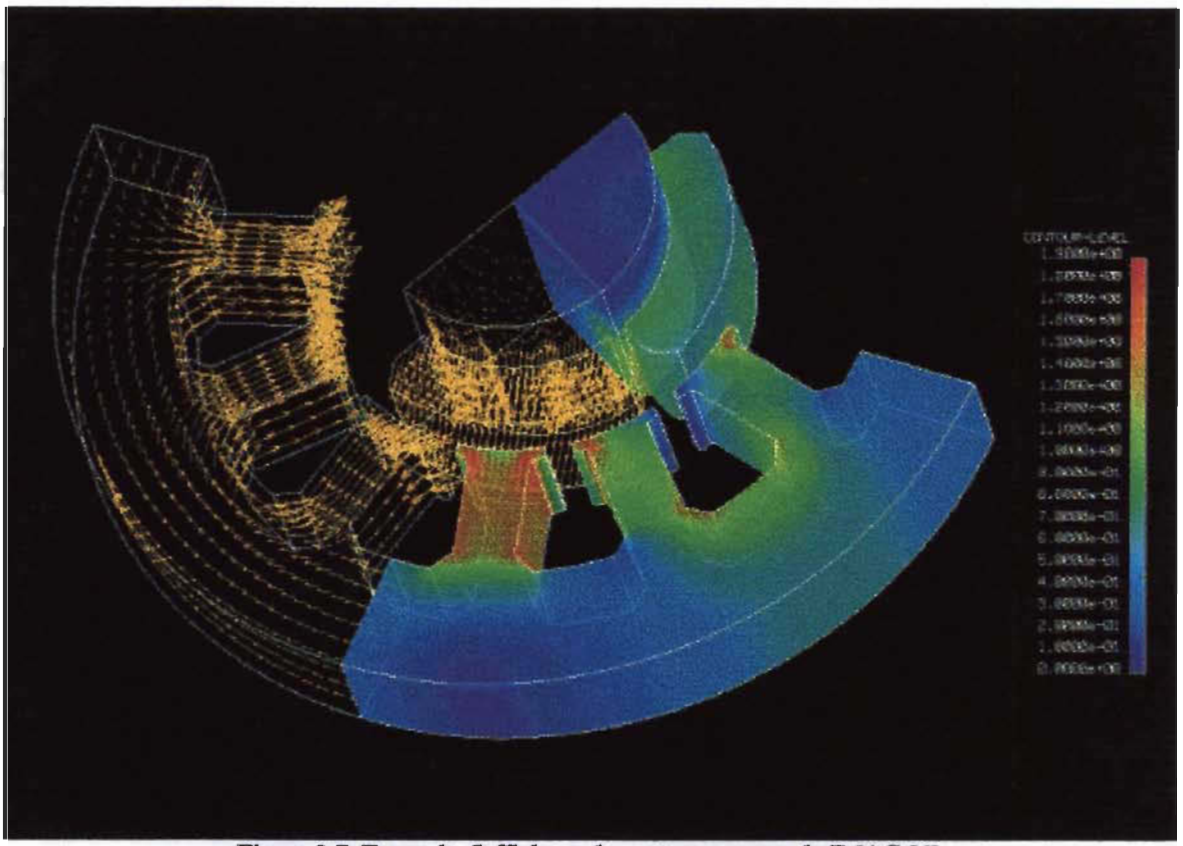

Figure 2.7: Exemple d'affichage du post-processeur de JMAG [6].

Il ne faut surtout pas perdre de vue dans lecture de ce document que la complexité des logiciels commerciaux mentionnés dans ce chapitre est proportionnelle à la quantité des ressources impliquées dans leur conception. Par exemple, le logiciel conçu pendant ce projet ne comporte pas d'affichage en dégradé de couleurs tel que montré à la figure 2.7. Il ne possède pas non plus d'affichage de champs vectoriels. L'emphase a plutôt été mis sur la mise en parallèle d'un tel logiciel. Par contre, avec le travail effectué, sachez qu'il est très possible d'analyser de façon efficace les données produites par les modules de calcul en basses et en hautes fréquences, sans toutefois offrir les mêmes possibilités que les logiciels commerciaux.

2.6 Quelques réalisations parallèles

Plusieurs projets de mise en parallèle de codes existants sont en cours ou ont été réalisés dans divers domaines tels que l'animation de dessins animés, la simulation de collisions automobiles, le traitement de images satellites et bien sûr la simulation électromagnétique pour ne nommer que ceux-là. C'est le cas du projet PEPSE (Parallel Electromagnetics Problem Solving Environment) [7]. Ce projet consiste en la mise en parallèle du code d'analyse électromagnétique EMA3D, et des pré- et post-processeurs FAM. On note que la motivation principale pour la mise en parallèle du post-processeur dans le cas de PEPSE est la diminution du temps d'attente de l'utilisateur lors de la mise à jour de son affichage. Le nombre impressionnant des données est donc séparé en un certain nombre de processeurs distincts. Le degré de parallélisme du post-processeur est bien différent de celui du module de calcul, et cela est imposé par la nature bien différente des deux modules. PEPSE conserve donc un faible nombre de bases de données différentes pour la mise en parallèle du post-processeur, soit environ 10, tandis que le module de calcul peut atteindre un degré de parallélisme de plus de 100.

C'est aussi le cas du projet PARTEL [8], réalisé dans le cadre du programme ESPRIT de l'union européenne. PARTEL vise la mise en parallèle des codes TOSCA, ELEKTRA et SCALA écrits à l'origine par la compagnie Vector Fields, partenaire du projet. On vise dans ce projet une large gamme de plates-formes allant des réseaux de stations de travail aux machines à mémoire partagée. Dans les deux cas mentionnés ci-haut, les résultats de mise en parallèle sont très encourageants. Les deux projets montrent des gains en vitesse considérables.

3 Les ordinateurs parallèles

3.1 Introduction

Les ordinateurs traditionnels, ou séquentiels, sont basés sur le modèle introduit par John von Neumann. Ce modèle est constitué d'une unité de traitement centrale (CPU) et de mémoire reliée directement au CPU. Ce modèle considère une simple séquence d'instructions et opère sur une simple séquence de données. Les ordinateurs de ce type sont souvent appelés "*single instruction stream, single data stream*" ou SISD [9]. La vitesse d'un ordinateur SISD est limitée par deux facteurs:

- La vitesse d'exécution des instructions,
- La vitesse de transfert des informations entre la mémoire et le CPU.

Différentes techniques sont utilisées dans les ordinateurs séquentiels afin d'augmenter leur vitesse. On dénote les techniques suivantes:

- Séparation de la mémoire en banques, chacune pouvant être accédée simultanément,
- Utilisation de mémoire cache,
- Utilisation du pipeline d'instructions dans le CPU.

Malgré toutes ces améliorations pour augmenter la vitesse, les limites des processeurs séquentiels sont quand même atteintes rapidement. Alors, une autre façon d'augmenter le taux des instructions exécutées est d'utiliser plusieurs unités de traitement et de mémoire interconnectées par un réseau de communication.

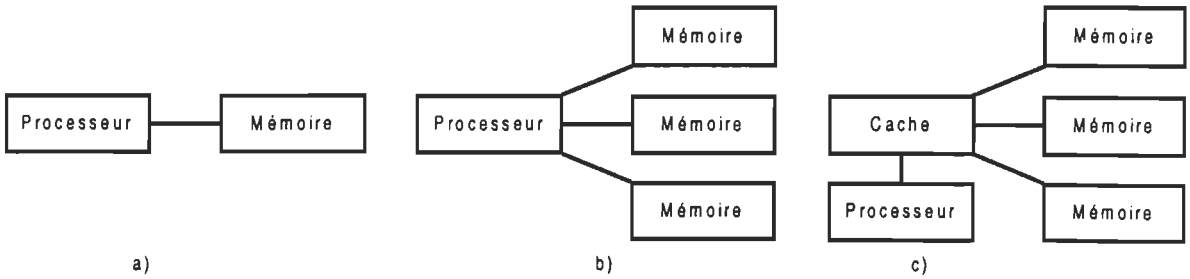

Figure 3.1: Ordinateurs séquentiels: a) simple; b) avec banques de mémoire; et c) avec banques de mémoire et cache [9].

3.2 Classification

Les ordinateurs parallèles peuvent être construits de différentes façons. Ils varient premièrement selon leur mécanisme de contrôle. Les deux classes importantes de mécanismes de contrôle sont la classe SIMD (*Single Instruction, Multiple Data*) et MIMD (*Multiple Instruction, Multiple Data*) [9]. Dans la première classe, une seule unité de contrôle distribue la tâche aux unités de calcul, et les mêmes instructions sont exécutées par toutes les unités de calcul et en même temps.

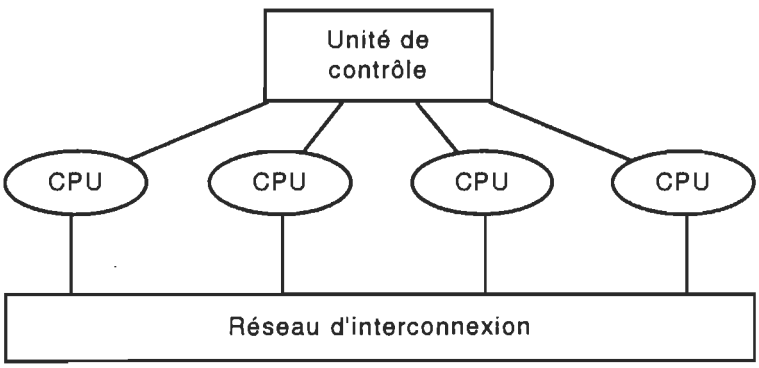

Figure 3.2: Architecture SIMD [9].

Les ordinateurs parallèles dont chacun des processeurs est capable d'exécuter des programmes différents indépendamment des autres processeurs appartiennent à la classe MIMD.

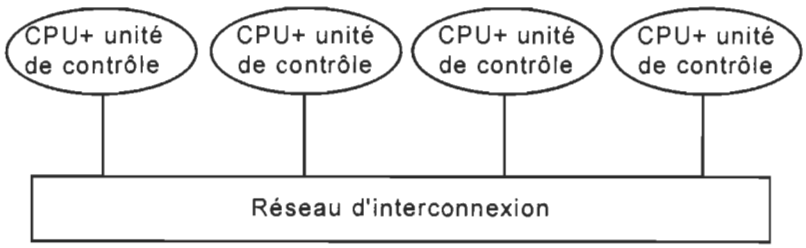

Figure 3.3: Architecture MIMD [9].

Les ordinateurs parallèles de la classe SIMD nécessitent moins de matériel que ceux de la classe MIMD car il n'y a qu'une seule unité de contrôle. De plus, les ordinateurs SIMD requièrent moins de mémoire car il n'y a qu'une seule copie du programme en mémoire. En contraste, les ordinateurs MIMD gardent en mémoire le programme et le système d'exploitation sur chacun des processeurs. Ces derniers sont plus complexes et il pourrait sembler qu'ils sont plus coûteux mais il est possible d'utiliser des processeurs d'usage général dans un ordinateur parallèle MIMD [9]. C'est la principale caractéristique des réseaux de stations de travail (*network of workstations* ou *NOW*) dans lesquels on peut utiliser des stations de travail SUN ou HP ou encore des ordinateurs personnels (PC) connectés à un réseau afin de construire un ordinateur parallèle. Les processeurs d'un système SIMD sont plus spécialisés et plus coûteux.

Dans les ordinateurs de type MIMD, les processeurs sont connectés par une architecture de passage de messages. Chacun des processeurs possède sa mémoire locale qui n'est accessible que par lui-même. Les processeurs s'échangent l'information par passage de messages. Cette architecture est aussi connue sous le nom d'architecture à "*mémoire distribuée*". Les machines MIMD à passage de messages sont connues sous le nom de "multi-ordinateur" (*multicomputer*).

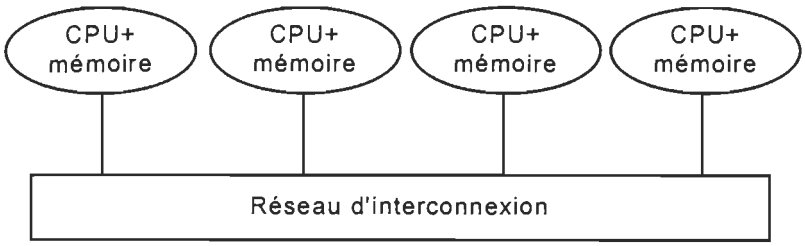

Figure 3.4: Architecture de passage de messages [9].

Les ordinateurs SIMD quant à eux possèdent le matériel requis pour permettre à chacun des processeurs de lire et d'écrire dans une zone de mémoire partagée par tous les processeurs. C'est pourquoi on nomme ces machines "machines à *mémoire partagée*". Les ordinateurs de type MIMD peuvent aussi fonctionner avec l'architecture à mémoire partagée par émulation.

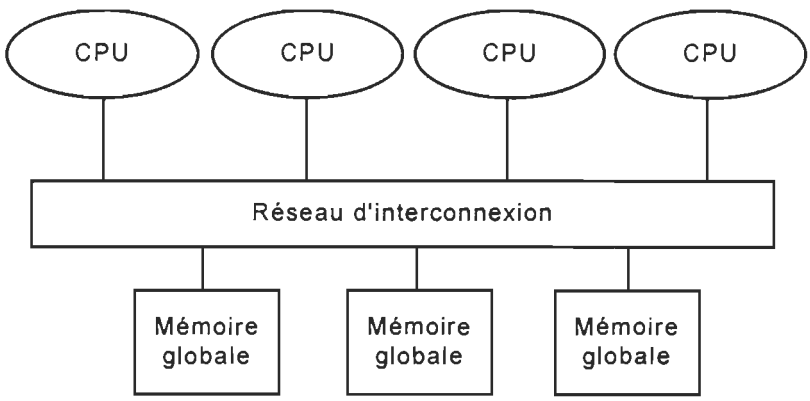

Figure 3.5: Architecture à mémoire partagée [9].

Les réseaux d'interconnexion reliant les processeurs peuvent être statiques ou dynamiques. Un réseau d'interconnexion statique consiste en un lien direct ou point à point entre les différents processeurs du système. Ce type de réseau d'interconnexion est généralement utilisé pour construire les ordinateurs à passage de messages. Les réseaux d'interconnexion dynamiques sont constitués d'un réseau de commutation entre les différents processeurs et les banques de mémoire. Les processeurs et les banques de

mémoire sont connectés dynamiquement par les éléments de commutation. Ce réseau d'interconnexion est utilisé dans les ordinateurs à mémoire partagée.

Les réseaux d'interconnexion statiques peuvent se retrouver sous les formes suivantes:

- Complètement connecté: Ces réseaux permettent à chacun des processeurs d'avoir un lien direct avec tous les autres processeurs. Ceci équivaut à un réseau relié par TCP/IP et il est implanté sur les machines AVX3 de Alex.

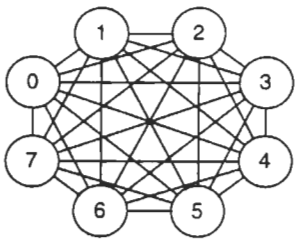

Figure 3.6: Réseau complètement connecté [9].

- Étoile: Les réseaux en étoile possèdent un processeur central qui possède un lien de communication avec chacun des autres processeurs. Les processeurs autres que le processeur central doivent donc passer nécessairement par celui-ci afin de communiquer entre eux.

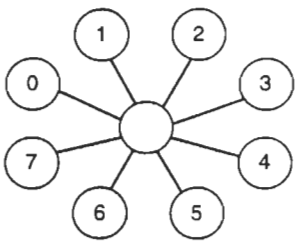

Figure 3.7: Réseau en étoile [9].

- Linéaire et anneaux: Ce réseau permet un échange direct entre les processeurs voisins. Un réseau linéaire avec un lien direct entre le premier et le dernier processeur devient un anneau.

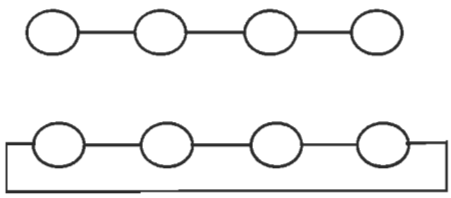

Figure 3.8: Réseau linéaire et en anneau [9].

- Grille: Le réseau en grille à deux dimensions permet à chacun des processeurs de communiquer directement avec ses quatre voisins immédiats. On peut retrouver des grilles carrées, ou rectangulaires lorsque le nombre de processeurs est différent dans une des deux directions. On retrouve des grilles avec bouclage permettant ainsi à tous les processeurs de communiquer directement avec leurs quatre voisins. Finalement, certains ordinateurs parallèles possèdent des grilles à trois dimensions.

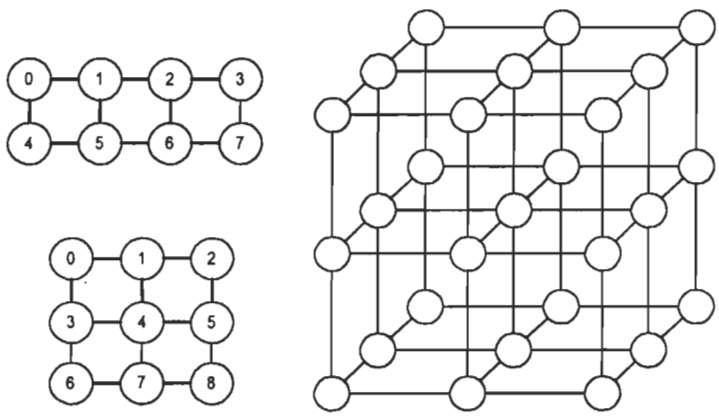

Figure 3.9: Grille rectangulaire, carrée et tridimensionnelle [9].

- Arbre: Les topologies en arbre se retrouvent sous deux formes principales, soient binaires ou ternaires. Le nombre de niveaux d'arbre est limité par le nombre de processeurs disponibles.

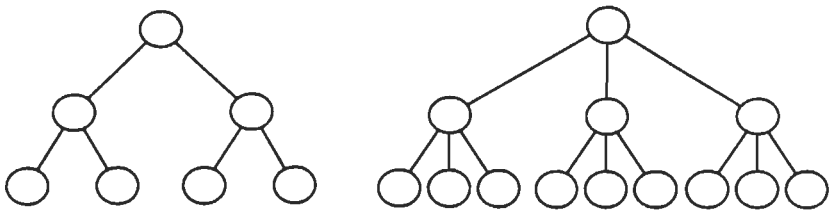

Figure 3.10: Arbres binaire et ternaire à trois niveaux [9].

- Hypercube: L'hypercube est construit avec deux cubes imbriqués l'un dans l'autre, et dont les huit coins de chacun des cubes est relié avec son correspondant.

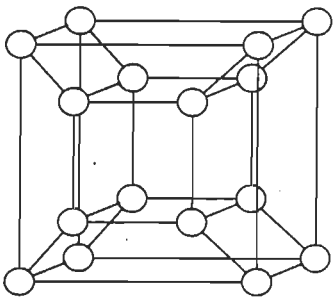

Figure 3.11: Hypercube [9].

Un ordinateur parallèle peut être constitué d'un petit nombre de processeurs très puissants ou d'un grand nombre processeurs de faible puissance. Dans le premier cas on parle de granularité grossière (*coarse-grain granularity*) et dans le deuxième cas, on dira que ces ordinateurs sont de granularité fine (*fine-grain granularity*) [9]. Les ordinateurs parallèles à granularité grossière peuvent contenir par exemple 8 ou 16 processeurs chacun étant capable de quelques GFLOPS tandis que les ordinateurs à granularité fine peuvent contenir par exemple 65536 très petits processeurs. Entre ces deux extrêmes, on retrouve une classe d'ordinateurs à granularité moyenne.

3.3 Outils de programmation et d'évaluation de performance

Il est fréquent dans la programmation parallèle d'écrire d'abord un nouveau code séquentiel et d'en effectuer par la suite la mise en parallèle. Pour effectuer la mise en

parallèle de codes séquentiels existants, certaines compagnies offrent des outils tels HPC (*High Performance C*) ou HPF (*High Performance Fortran*). Ces outils inspectent le code séquentiel et parviennent à faire la mise en parallèle de façon automatique. Par contre, le programmeur doit mettre une touche finale afin de faire certains ajustements au code parallèle. Les outils tels que HPC et HPF ne sont disponibles que sur les machines à mémoire partagée. Les machines utilisées dans ce projet étant de type MIMD ou à mémoire distribuée, le seul outil de programmation disponible est la bonne vieille programmation en langage C de bas niveau. Mais c'est aussi ce qui est le plus performant, lorsque bien utilisé. Par expérience, les générateurs de codes ne font pas d'optimisation et ils génèrent un code qui est difficile à lire ou à modifier.

Les constructeurs d'ordinateurs parallèles fournissent en général tous les outils et toutes les librairies nécessaires afin d'opérer et de programmer de façon adéquate leurs propres machines. Les machines parallèles utilisées dans ce projet fonctionnent sur la base de passage de messages avec "Trollius" comme support. Trollius est un "système d'exploitation" qui est chargé sur chacun des processeurs utilisés dans un multi-ordinateur. Il reconnaît que chaque processeur accessible par les liens de communication est un participant potentiel dans les opérations de passage de messages. Trollius offre un environnement de programmation uniforme pour les différentes stations de travail et aussi pour les ordinateurs parallèles dédiés que l'on appelle "machines parallèles", tels les AVX2 et AVX3 de la compagnie Alex Inc.. Dans l'environnement Trollius, on retrouve deux types de processeurs, soit les processeurs de calcul et les processeurs hôte. Les premiers sont nommés ITB (In The Box) et les second OTB (Out The Box) [10]. La structure d'un multi-ordinateur se présente telle que le schéma de la figure 3.12 le montre.

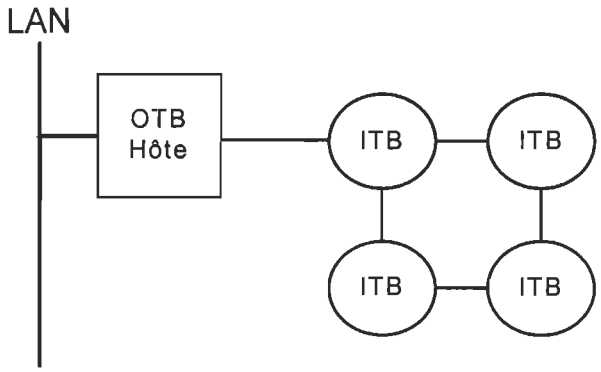

Figure 3.12: Structure d'un multi-ordinateur [10].

On retrouve une succession à Trollius sous le nom de LAM (Local Area Multicomputer). LAM est une évolution de Trollius qui fonctionne seulement sur un réseau de stations de travail UNIX agissant comme OTB et n'inclut pas de ITB [11]. LAM sert à connecter ensemble des stations de travail afin de constituer un ordinateur parallèle. Ce dernier sera de type complètement connecté et utilisera le passage de messages sur le réseau TCP/IP pour effectuer les communications. Les ordinateurs parallèles AVX3 sont finalement quelques stations de travail connectées en réseau utilisant LAM afin d'effectuer la programmation des passages de messages.

Tous ces développements en terme d'ordinateurs parallèles et de passage de messages ont amené la communauté scientifique à établir un standard au niveau du passage de messages que l'on a nommé MPI (Message Passing Interface). MPI est une librairie contenant un ensemble de fonctions servant à effectuer le passage de messages dans les ordinateurs parallèles. Les avantages de MPI sont la portabilité, la flexibilité, l'efficacité et la fiabilité.

Il existe certaines applications qui permettent d'évaluer avec une grande exactitude le temps que chacun des processeurs consacrent à la communication et au calcul ou la

balance de la charge entre les processeurs. Un outil de ce genre, "*ParaGraph*", a été essayé sur les machines Alex AVX2 mais sans succès. Des modifications importantes étaient nécessaires aux sources de Trollius, ce qui ne fut pas fait par la compagnie Alex.

Le principal but de la programmation parallèle étant d'améliorer les performances, il faut être capable d'évaluer avec exactitude l'apport des machines parallèles à notre application. Certaines mesures sont donc nécessaires et elles sont [9]:

- Temps d'exécution: Le temps d'exécution d'un programme séquentiel est le temps écoulé entre l'instant où le programme démarre et l'instant où il se termine. Pour un programme parallèle, le temps d'exécution est le temps écoulé entre le moment où le calcul commence et le moment où le dernier processeur termine son calcul.
- Le gain en vitesse (*Speedup*): Le gain en vitesse est définit comme le rapport du temps d'exécution séquentiel sur le temps d'exécution parallèle.
- L'efficacité: Elle est une mesure de la fraction de temps où un processeur est utilisé de façon efficace. Les algorithmes parallèles impliquent toujours des communications d'informations entre les processeurs. L'efficacité est définie comme étant le rapport du gain en vitesse sur le nombre de processeurs.

Avec ces informations, le programmeur est en mesure d'évaluer l'impact du parallélisme sur l'application étudiée. Dans le chapitre 7, ces valeurs seront utilisées afin de comparer les performances de l'algorithme en fonction du nombre de processeurs et de la taille du problème à traiter.

3.4 Description des ordinateurs Alex AVX-2 et AVX-3

Les ordinateurs Alex de série AVX2 utilisent l'architecture à mémoire distribuée MIMD. Une caractéristique importante de ces machines est qu'elles n'impose pas de réseau de communication fixe. C'est-à-dire que les utilisateurs peuvent, par l'utilisation des outils "Altools", configurer l'ordinateur pour reproduire une des topologies décrites dans la section précédente. Chacun des processeurs de calcul, nommés aussi nœuds, sont composés de deux processeurs: d'abord un transputeur T805 est dédié surtout à la communication. Ensuite un processeur Intel i860 sert de processeur de calcul. Ces deux processeurs possèdent leur mémoire locale et il partagent aussi une zone de mémoire où les messages reçus par le T805 seront transmis au i860.

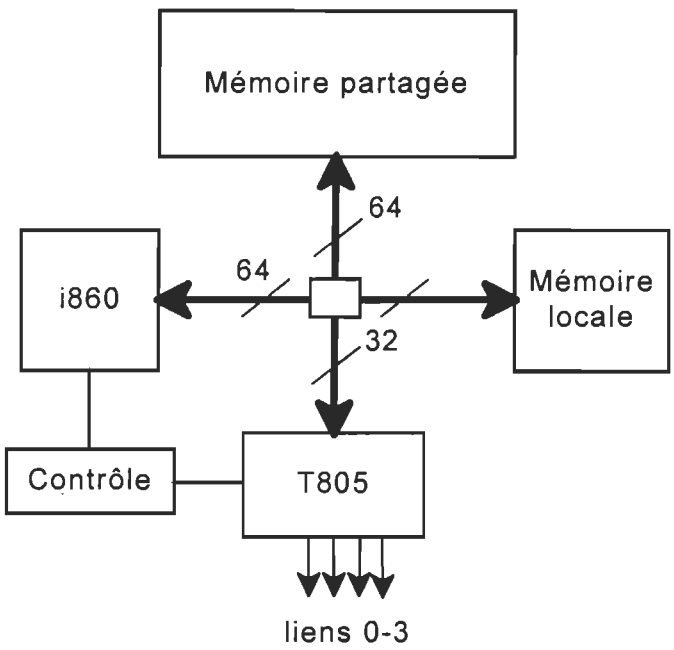

Figure 3.13: Diagramme bloc d'un nœud i860 [10].

Le processeur T805 possède une unité de calcul en virgule flottante de 64 bits et 4 KB de mémoire vive intégrée. Il est capable d'une performance maximale de 30 MIPS et

4.3 MFLOPS avec une fréquence d'horloge de 25 MHz. Dans la configuration des ordinateurs Alex, il peut servir aussi bien de processeur de communication comme de processeur de calcul. De son côté, le processeur i860 de technologie RISC possède une horloge à 40 MHz. Il peut délivrer une performance maximale de 80 MFLOPS.

Les ordinateurs AVX2 du laboratoire de recherche sur les biochamps se retrouvent sous deux versions, soient la version 64 nœuds et la version 24 nœuds. Chacun des nœuds de la première version possède 16 MB de mémoire pour le T805 et 64 MB pour le i860. Dans la deuxième version, on retrouve 4 MB de mémoire pour le T805 et 16 MB pour le i860.

Les nœuds sont accessibles via une station de travail SUN Sparc 20 par l'intermédiaire des outils "Altools" [10]. L'utilisateur configure d'abord la topologie voulue, soit grille, anneau, pipeline, hypercube, arbre binaire ou arbre ternaire. Notons que les maximum de 4 liens sur le T805 impose certaines limites quant à certaines configurations comme la grille tridimensionnelle qui nécessite 6 liens par processeur. Après avoir configuré sa topologie, l'utilisateur peut ensuite passer à la programmation en utilisant les fonction d'envoi et de réception de messages de Trollius.

Quant à l'ordinateur AVX3 utilisé lors de ce projet, il contient 16 stations de travail avec processeur Power PC 604 à 133 MHz avec 256 MB de mémoire vive [12]. Ces 16 stations de travail sont reliées par un réseau non commuté de 100 Mbps. Chacun des nœuds de ce système possède sa propre version du système d'exploitation AIX et son propre disque rigide la contenant. Contrairement à la série AVX2, la série AVX3 permet l'utilisation du protocole MPI pour le transfert des messages. LAM sert à relier ensemble ces stations de travail afin de constituer une machine parallèle.

Au cours de ce projet, le logiciel écrit sera testé en mode séquentiel sur une station SUN Sparc 20, et en parallèle sur AVX2 et sur AVX3 avec Trollius comme support de passage de messages. En plus, une version MPI du post-processeur sera testée sur un réseau de stations de travail et sur l'ordinateur AVX3, et les résultats seront comparés

4 Théorie du graphisme par ordinateur

4.1 Introduction

Dans ce chapitre, nous allons introduire les transformations de base en deux dimensions (2D) et en trois dimensions (3D), qui sont utilisées dans le graphisme par ordinateur. Nous nous attarderons principalement sur la translation et la rotation. Nous allons d'abord comprendre les transformations 2D, ce qui permettra au lecteur de mieux comprendre la théorie relative aux transformations 3D. Les transformations présentées dans cette section ont été implantées dans le code de l'interface graphique qui fait l'objet de cette étude.

4.2 Transformations 2D

On peut effectuer la *translation* de points dans le plan x-y en additionnant simplement la valeur de la translation à appliquer directement aux coordonnées des points. Pour chaque point $P(x, y)$ à être déplacé, on additionne d_x unités parallèlement à l'axe des x, et d_y unités parallèlement à l'axe des y, pour obtenir le nouveau point $P'(x', y')$ [13].

$$x' = x + d_x, \qquad y' = y + d_y \tag{4.1}$$

Sous forme vectorielle, on peut donc définir les vecteurs colonne

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \mathbf{P'} = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} d_x \\ d_y \end{bmatrix} \tag{4.2}$$

Finalement, on peut écrire sous forme plus concise que

$$\mathbf{P'} = \mathbf{P} + \mathbf{T} \tag{4.3}$$

Par exemple, le point *P(5, 7)* auquel on applique une translation *T(-3, 2)* sera déplacé à *P'(2, 9)*.

La deuxième transformation que nous allons étudier est la *rotation* d'un angle θ par rapport à l'origine dans le plan *x-y*. En coordonnées polaires, la droite illustrée à la figure 4.1 passera de sa position originale *(r, ϕ)* vers *(r, $\phi+\theta$)*. Nous aurons donc:

$$x = r\cos(\phi), \qquad y = r\sin(\phi)$$

$$x' = r\cos(\phi+\theta), \qquad y' = r\sin(\phi+\theta)$$

Ce qui se traduit par:

$$x' = r\cos\phi\cos\theta - r\sin\phi\sin\theta,$$
$$y' = r\sin\phi\cos\theta + r\cos\phi\sin\theta$$

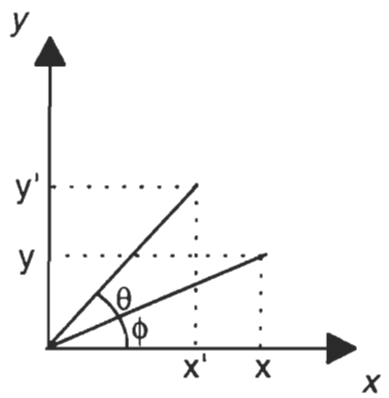

Figure 4.1: Rotation dans le plan x-y [13].

et finalement:

$$x' = x\cdot\cos\theta - y\cdot\sin\theta, \qquad y' = x\cdot\sin\theta + y\cdot\cos\theta \tag{4.4}$$

Sous forme matricielle, on aura [13]:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{ou} \quad \mathbf{P'} = \mathbf{R}\cdot\mathbf{P} \tag{4.5}$$

où **R** est la matrice de rotation.

Comme on le voit par (4.3) et (4.5), la translation est traitée comme une addition, et la rotation est traitée comme une multiplication. Nous allons introduire dans la section suivante une théorie qui permettra de traiter ces transformations d'une façon consistante, et elles pourront alors être combinées en une seule opération.

4.3 Coordonnées homogènes 2D

Si les points sont exprimés en coordonnées homogènes, la rotation et la translation pourront alors être traitées par multiplication. Pour exprimer les points en coordonnées homogènes, on ajoute une troisième coordonnée à la paire *(x, y)*, ce qui résultera en la description d'un point par *(x, y, W)*. On dira que deux coordonnées homogènes représentent le même point si et seulement si l'une des coordonnées est le multiple de l'autre [13]. Ainsi, (2, 3, 6) et (4, 6, 12) sont deux coordonnées homogènes qui représentent le même point. De plus, au moins une des coordonnées doit être non-nulle; (0, 0, 0) n'est pas acceptable. Lorsque W = 0, nous avons la représentation d'un point situé à l'infini, et nous ne discuterons pas de ce cas dans la présente recherche.

Normalement, la valeur de *W* est non-nulle et nous divisons les trois coordonnées par *W*. Nous obtenons donc *(x/W, y/W, 1)*, et dans ce cas *x/W* et *y/W* sont appelées les *coordonnées cartésiennes* du point homogène.

En coordonnées homogènes, on peut écrire la translation comme

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \text{ ou } \mathbf{P'} = \mathbf{T}(d_x, d_y) \cdot \mathbf{P} \tag{4.6}$$

Demandons-nous maintenant ce qui se passe lorsqu'on applique deux translations successives $\mathbf{T}(d_{x1}, d_{y1})$ et $\mathbf{T}(d_{x2}, d_{y2})$ au même point P. Intuitivement, le résultat devrait être une translation nette de $\mathbf{T}(d_{x1}+ d_{x2}, d_{y1}+ d_{y2})$. Pour confirmer cette intuition, posons:

$$\mathbf{P'} = \mathbf{T}(d_{x1}, d_{y1}) \cdot \mathbf{P}, \tag{4.7}$$

$$\mathbf{P''} = \mathbf{T}(d_{x2}, d_{y2}) \cdot \mathbf{P'} \tag{4.8}$$

En substituant (4.7) dans (4.8), on obtient:

$$\mathbf{P''} = \left(\mathbf{T}(d_{x1}, d_{y1}) \cdot \mathbf{T}(d_{x2}, d_{y2})\right) \cdot \mathbf{P} \tag{4.9}$$

et le produit matriciel de $\mathbf{T}(d_{x1}, d_{y1})$ et $\mathbf{T}(d_{x2}, d_{y2})$ est:

$$\begin{bmatrix} 1 & 0 & d_{x1} \\ 0 & 1 & d_{y1} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & d_{x2} \\ 0 & 1 & d_{y2} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{x1}+d_{x2} \\ 0 & 1 & d_{y1}+d_{y2} \\ 0 & 0 & 1 \end{bmatrix} \tag{4.10}$$

ce qui nous permet de montrer que le résultat de deux translations successives correspond bien à $\mathbf{T}(d_{x1}+ d_{x2}, d_{y1}+ d_{y2})$, soit à la multiplication des deux matrices de translation. On dit donc que la translation est *additive*.

Dans le cas de la rotation, l'équation (4.5) devient:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4.11}$$

On peut démontrer que la rotation est, comme la translation, additive elle aussi.

$$\mathbf{P'} = \mathbf{R}(\theta_1) \cdot \mathbf{P}, \tag{4.12}$$

$$\mathbf{P''} = \mathbf{R}(\theta_2) \cdot \mathbf{P'} \tag{4.13}$$

En substituant (4.12) dans (4.13), on obtient:

$$\mathbf{P''} = \left(\mathbf{R}(\theta_1) \cdot \mathbf{R}(\theta_2)\right) \cdot \mathbf{P} \tag{4.14}$$

et le produit matriciel de $\mathbf{R}(\theta_1)$ et $\mathbf{R}(\theta_2)$ est:

$$\mathbf{R}(\theta_1)\cdot\mathbf{R}(\theta_2) = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}(\theta_1)\cdot\mathbf{R}(\theta_2) = \begin{bmatrix} (\cos\theta_1\cos\theta_2 - \sin\theta_1\sin\theta_2) & -(\sin\theta_1\cos\theta_2 + \cos\theta_1\sin\theta_2) & 0 \\ (\sin\theta_1\cos\theta_2 + \cos\theta_1\sin\theta_2) & (\cos\theta_1\cos\theta_2 - \sin\theta_1\sin\theta_2) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}(\theta_1)\cdot\mathbf{R}(\theta_2) = \begin{bmatrix} \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 \\ \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}(\theta_1)\cdot\mathbf{R}(\theta_2) = \mathbf{R}(\theta_1+\theta_2) \tag{4.15}$$

4.4 *Transformations 2D par rapport à un point quelconque*

Dans la section précédente, nous avons démontré que la translation et la rotation sont toutes deux additives, lorsque exprimées en coordonnées homogènes. Ceci nous permet de combiner la translation et la rotation afin d'effectuer des transformations par rapport à un point P_1 quelconque dans le plan x-y. Lorsque l'on effectue cette rotation, on le fait en trois étapes distinctes, soient:

1. effectuer une translation de façon à ce que P_1 devienne l'origine,
2. effectuer la rotation,
3. et refaire la translation inverse de l'étape 1.

Ces trois étapes, mathématiquement, seront représentées d'abord par une translation de $(-x_1, -y_1)$, une rotation de θ et par une translation de (x_1, y_1). La transformation nette sera donc :

$$\mathbf{T}(x_1, y_1)\cdot\mathbf{R}(\theta)\cdot\mathbf{T}(-x_1,-y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix}\cdot\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\cdot\begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}(x_1, y_1)\cdot\mathbf{R}(\theta)\cdot\mathbf{T}(-x_1,-y_1) = \begin{bmatrix} \cos\theta & -\sin\theta & x_1(1-\cos\theta)+y_1\sin\theta \\ \sin\theta & \cos\theta & y_1(1-\cos\theta)+x_1\sin\theta \\ 0 & 0 & 1 \end{bmatrix} \qquad (4.16)$$

4.5 Transformations 3D

Tout comme les transformations 2D peuvent être représentées par des matrices 3×3, les transformations 3D peuvent l'être par des matrices 4×4 en utilisant la représentation par coordonnées homogènes *(x, y, z, W)*. Les mêmes règles s'appliquent au niveau des coordonnées homogènes 3D, et ces règles apparaissent au premier paragraphe de la section 4.3. Il est important de noter ici que la *règle de la main droite* est respectée lors de la détermination du sens de rotation par rapport à un axe. Lorsque le pouce pointe vers la direction positive de l'axe de rotation, le sens de rotation positif est dans la direction des doigts ou dans le sens anti-horaire lorsque le sens positif de l'axe pointe vers nous. On utilise la règle de la main droite parce que c'est la convention mathématique standard mais généralement, en graphisme 3D, la règle de la main gauche est utilisée car il est plus naturel d'interpréter une valeur positive de *z* comme étant derrière l'écran où s'effectue l'affichage.

Tableau 4.1 : Sens de direction de la rotation positive [13].

Axe de rotation	Direction de la rotation positive
x	*y* à *z*
y	*z* à *x*
z	*x* à *y*

La translation en 3D est simplement une extension de celle qui a été exposée pour le cas 2D dans la section 4.2.

$$\mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.17}$$

Le cas de la rotation 2D présenté à (4.5) est en fait une rotation autour de l'axe des z; nous aurons donc:

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.18}$$

De même, les rotations autour de l'axe des x et des y seront:

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.19}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.20}$$

4.6 *Transformations 3D par rapport à une droite quelconque*

On peut effectuer une rotation par rapport à une droite quelconque $\mathbf{U}(u_x, u_y, u_z)$ dans l'espace trois dimensions. Pour ce faire, on doit exécuter les étapes suivantes:

- Amener la droite de rotation à l'origine par translation (figure 4.1 a-b).

- Effectuer une rotation sur l'axe des x et amener la droite de rotation dans le plan x-z (figure 4.1 b-c).
- Effectuer une rotation sur l'axe des y pour amener la droite de rotation sur l'axe des z (figure 4.1 c-d).
- Effectuer la rotation sur l'axe des z.
- Refaire les trois premières étapes dans l'ordre inverse.

Généralement, on compose les cinq étapes de rotation dans une seule opération par multiplication, et la translation est effectuée dans une autre opération, tel que:

$$\mathbf{P}' = \mathbf{T}(-x_1, -y_1, -z_1) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(x_1, y_1, z_1) \cdot \mathbf{P} \tag{4.21}$$

Dans cette équation, la valeur de $\mathbf{R}(\theta)$ sera donc:

$$\mathbf{R}(\theta) = \mathbf{M} \cdot \mathbf{R}_z(\theta) \cdot \mathbf{M}^{-1} \tag{4.22}$$

où $\mathbf{M}$ est la matrice de transformation qui effectue une rotation de $\mathbf{U}$ sur les axes x et y pour que la droite $\mathbf{U}$ prenne la même direction que l'axe des z.

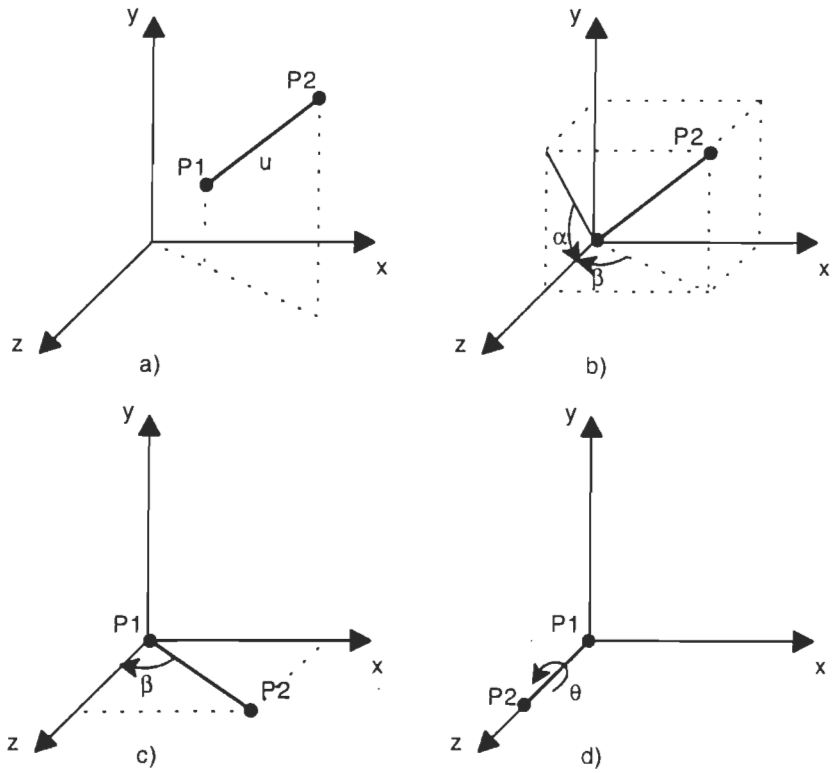

Figure 4.2: Opérations de rotation par rapport à une droite quelconque dans l'espace 3D.

$$\mathbf{M} = \mathbf{R}_x(\alpha) \cdot \mathbf{R}_y(\beta)$$

et

$$\sin(\alpha) = \frac{u_y}{\sqrt{u_y^2 + u_z^2}}, \qquad \cos(\alpha) = \frac{u_z}{\sqrt{u_y^2 + u_z^2}} \tag{4.23}$$

$$\sin(\beta) = \frac{-u_x}{\sqrt{u_x^2 + u_y^2 + u_z^2}}, \qquad \cos(\beta) = \frac{\sqrt{u_y^2 + u_z^2}}{\sqrt{u_x^2 + u_y^2 + u_z^2}} \tag{4.24}$$

En substituant (4.23) dans (4.19) et (4.24) dans (4.20), nous aurons:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \dfrac{u_z}{\sqrt{u_y^2+u_z^2}} & -\dfrac{u_y}{\sqrt{u_y^2+u_z^2}} & 0 \\ 0 & \dfrac{u_y}{\sqrt{u_y^2+u_z^2}} & \dfrac{u_z}{\sqrt{u_y^2+u_z^2}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.25}$$

et

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \dfrac{\sqrt{u_y^2+u_z^2}}{\sqrt{u_x^2+u_y^2+u_z^2}} & 0 & \dfrac{-u_x}{\sqrt{u_x^2+u_y^2+u_z^2}} & 0 \\ 0 & 1 & 0 & 0 \\ \dfrac{u_x}{\sqrt{u_x^2+u_y^2+u_z^2}} & 0 & \dfrac{\sqrt{u_y^2+u_z^2}}{\sqrt{u_x^2+u_y^2+u_z^2}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.26}$$

La matrice de rotation finale sera donc le résultat de (4.22), soit:

$$\mathbf{A} = \begin{bmatrix} u_x^2(1-\cos\theta)+\cos\theta & u_x u_y(1-\cos\theta)-u_z\sin\theta & u_x u_z(1-\cos\theta)+u_y\sin\theta & 0 \\ u_x u_y(1-\cos\theta)+u_z\sin\theta & u_y^2(1-\cos\theta)+\cos\theta & u_y u_z(1-\cos\theta)-u_x\sin\theta & 0 \\ u_x u_z(1-\cos\theta)-u_y\sin\theta & u_y u_z(1-\cos\theta)+u_x\sin\theta & u_z^2(1-\cos\theta)+\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.27}$$

Cette matrice est très bien connue dans le milieu du graphisme. Tous les logiciels effectuant la rotation graphique d'un élément quelconque utilisent cette matrice de calcul de la rotation. L'interface graphique parallèle que nous décrivons dans le présent rapport de recherche ne fait pas exception à la règle. Cette matrice est implantée dans la fonction de rotation qui sera décrite dans la section 5.4.3.

5 Programmation

5.1 Introduction

L'interface parallèle dont il est question dans cette section se compose de deux parties principales. Dans un premier temps, la section 5.3 fera la description de l'interface usager, première partie qui interagit directement avec l'usager par ses fenêtres et ses menus. Cette interface usager a été écrite complètement en langage C avec un respect du standard Motif [14], standard qui sera décrit en 5.2. Elle permet donc d'opérer le post-processeur en effectuant des opérations telles que:

- Chargement ou enregistrement des fichiers,
- Rotation ou translation des images,
- Zoom,
- Changement des options d'affichage.

La section 5.4 fera la description de la deuxième partie de l'interface parallèle, soit l'implantation graphique. On pourra constater la façon d'utiliser le langage C afin de construire une véritable base de données graphique, les différents formats de fichier utilisés et finalement les différentes opérations effectuées sur les données graphiques.

On a opté pour le langage C afin de réaliser la programmation de la totalité de l'interface. Il a été question à un certain moment de considérer l'utilisation d'un générateur de code pour effectuer l'interface usager, ce qui rend un peu plus simple la programmation en standard Motif, mais après quelques essais effectués, il s'est avéré difficile de mettre en liaison le code de l'interface usager généré automatiquement, et le code écrit pour

l'implantation graphique. Le langage C permet une portabilité optimale; la partie séquentielle du code a d'ailleurs été essayée sur HP-UX et sur AIX avec succès. L'utilisation des ordinateurs ALEX AVX-2 impose aussi l'utilisation de deux langages, soient le Fortran ou le C, mais le C a été retenu.

5.2 *Le standard Motif*

Motif est un ensemble de règles à suivre qui spécifient comment une interface usager doit paraître à l'écran, et comment elle doit interagir avec l'utilisateur. La figure 5.1 montre un exemple d'une application Motif.

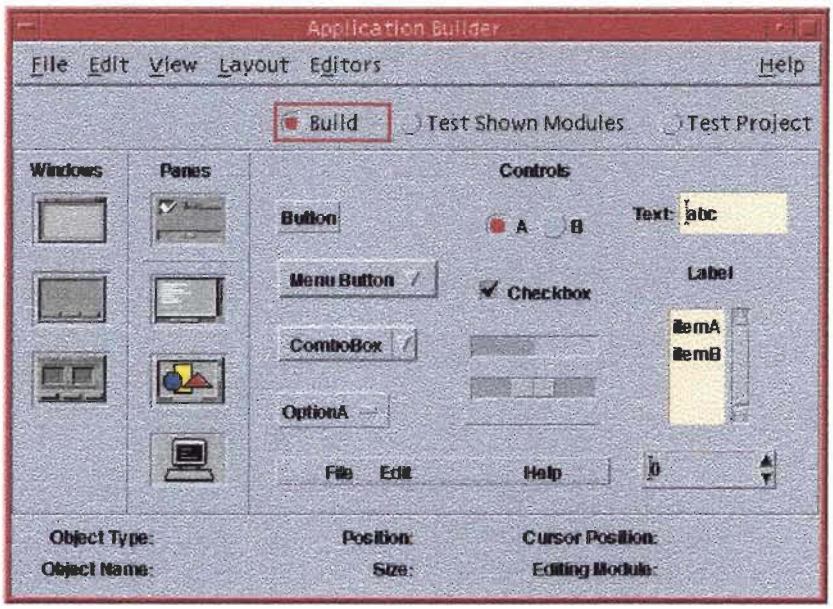

Figure 5.1: Exemple d'application Motif

L'utilisateur interagit avec l'interface en tapant au clavier, et en cliquant et en déplaçant la souris sur différents éléments de l'interface. Par exemple, une application Motif doit permettre de déplacer une fenêtre en gardant le bouton de la souris enfoncé tout en la déplaçant. Lorsque le bouton de la souris est relâché, la fenêtre prend cette nouvelle position. Une fenêtre peut aussi être redimensionnée en effectuant la même opération sur

les bordures de la fenêtre. On retrouve ces caractéristiques aussi bien sur les stations de travail UNIX que sur les ordinateurs personnels. La fenêtre Motif possède une barre de boutons située au haut de l'écran constituant les menus d'opération de l'application. Ces boutons lancent des applications précises ou encore font apparaître des fenêtres supplémentaires à l'écran appelées "*boîtes de dialogue*".

Motif a été conçu par OSF (Open Software Foundation), un consortium à but non lucratif composé de compagnies telles HP, Digital, IBM et une bonne douzaine d'autres compagnies[14]. Il a été décidé alors de concevoir OSF/Motif sur le système "*X Window*", système que l'on retrouve sur toutes les stations de travail UNIX, VMS, et autres.

Motif est une spécification et non une application. On retrouve deux grandes classes de spécifications Motif :

- Le modèle de sortie qui définit comment les objets apparaîtront à l'écran, incluant les formes de boutons, les effets trois dimensions sur ces boutons, le curseur, l'arrangement des fenêtres, etc.
- Le modèle d'entrée qui spécifie comment l'usager interagit avec les éléments sur l'écran.

La clé de Motif est que ces spécifications doivent être uniformes à travers toutes les applications. Des éléments des interfaces usager sont donc définis dans une boîte à outil (*toolkit*) qui sera utilisée pour écrire les applications, les rendant ainsi uniformes entre elles. Lorsque ce standard est atteint, l'utilisateur passe directement à la production plutôt que de perdre un temps énorme à apprendre le fonctionnement de l'application. Motif permet d'atteindre ce but de façon très efficace.

Pour écrire une application Motif, il existe deux solutions. D'abord, le programmeur peut écrire entièrement l'application tout en s'assurant de la conformité des moindres détails avec le standard Motif, ou il peut utiliser une boîte à outil de programmation, ce qui est la solution la plus logique et la plus simple. La boîte à outil Motif contient une collection de fonctions pré-écrites qui assurent que l'application respectera tous les standards.

Une interface usager Motif est créée en utilisant la librairie Motif Xm et la librairie *Intrinsics* Xt. Xt fournit des fonctions pour créer et pour gérer les ressources des *widgets* (structure orientée objet permettant de créer des composantes d'interface usager que nous allons traduire en français par *trucs* dans le reste de ce document). Xm fournit les trucs eux-mêmes, plus des utilitaires et des fonctions pour créer des groupes de trucs formant une composante fondamentale de l'interface usager. Par exemple, la barre de menu Motif n'est pas un truc mais une collection de trucs plus petits mis ensemble par une fonction spéciale.

Une application peut aussi faire appel directement au niveau Xlib pour faire l'affichage graphique ou pour gérer les événements du système de fenêtres. L'application, et non l'interface usager, peut aussi faire appel à des fonctions de bas niveau du système d'exploitation ou du système de fichiers. Aussi l'application peut effectuer des appels à diverses librairies telle que la librairie mathématique par exemple. La figure 5.2 représente le modèle d'interface des différentes librairies.

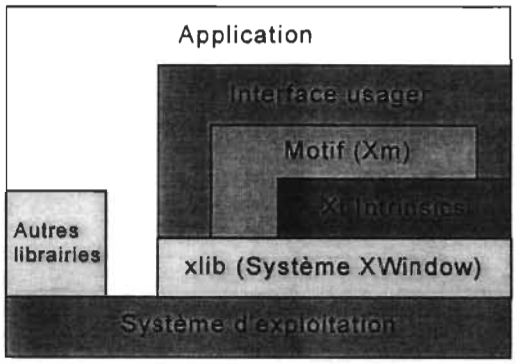

Figure 5.2: Modèle d'interface des librairies.

5.3 *Interface usager*

Comme elle est décrite dans la description du sujet de cette recherche, l'interface usager doit être simple d'utilisation. C'est pourquoi il a été décidé dès les premières étapes du projet d'utiliser une interface graphique Motif sur X Window. La figure 5.3 montre une représentation de l'interface. On y retrouve toutes les composantes qui se retrouvent dans les logiciels commerciaux respectant le standard Motif. La langue anglaise a été choisie par souci d'internationalisation car les travaux effectués ont été présentés dans deux conférences internationales.

La barre supérieure comprend les cinq menus principaux et le menu d'aide qui est situé complètement à droite, ce qui est une des exigences du standard Motif. Certaines de ces options ne sont pas encore implantées au moment d'écrire ces lignes.

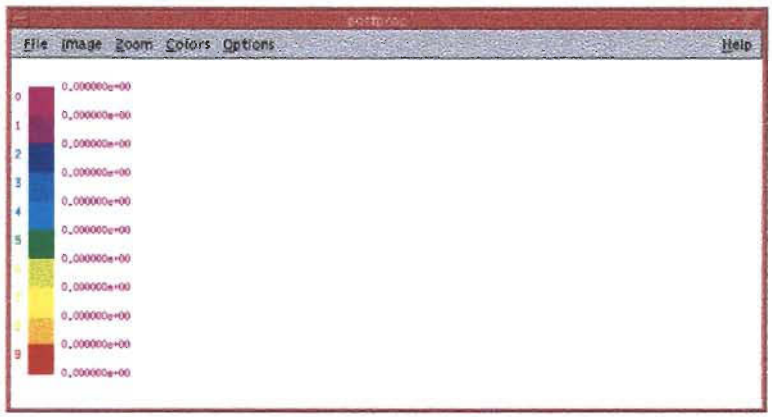

Figure 5.3: Interface usager sur X Window.

La zone d'affichage comprend la légende que l'on retrouve sur la gauche, et la zone d'affichage proprement dit sur la droite. Le nombre de couleurs est fixé par le logiciel à 10 et la couleur de fond peut être changée pour le noir. Par souci de clarté, nous allons utiliser la couleur blanche comme couleur de fond dans ce document. L'interface graphique qui sera présentée dans ce chapitre est simple et peut difficilement se comparer avec les interfaces commerciales telles que nous les avons vues dans le chapitre 2. Elle a été écrite par une seule et même personne sur environ 18 mois, tandis que les interfaces commerciales comptent des années de développement par toute une équipe de programmeurs.

Dans les sections suivantes, nous allons voir comment l'interface usager du présent projet de recherche a été construite, en programmation Motif, étape par étape.

5.3.1 Fenêtres

Les premières étapes de la programmation Motif consistent en la définition des fenêtres et des trucs. On se doit de construire une hiérarchie de fenêtres selon le schéma présenté à la figure 5.4.

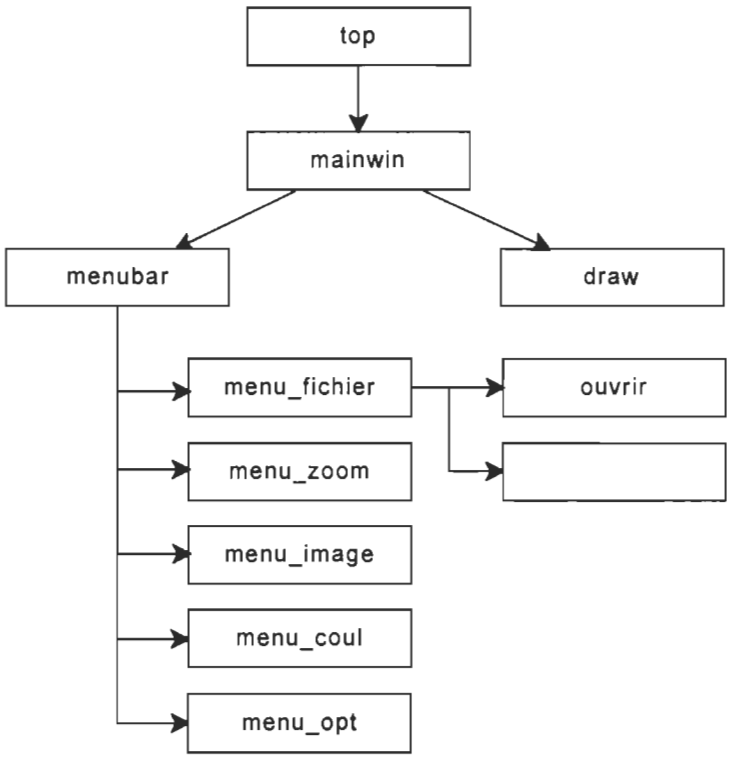

Figure 5.4: Hiérarchie des fenêtres et des trucs de l'interface usager.

Comme on le voit sur la figure 5.4, la première fenêtre "top" est celle qui contiendra toutes les autres. La fenêtre "mainwin" quant à elle contient la fenêtre "draw" qui est la zone graphique où seront dessinés la légende et les objets à afficher. La fenêtre "menubar" est, comme son nom l'indique, la barre de menu qui contient des trucs de menus en cascade que nous verront plus loin en détail. Chacun de ces trucs à leur tour peuvent appeler une fenêtre. Par exemple, l'option d'ouverture de fichier appelle la boîte de dialogue d'ouverture de fichier de la librairie Motif.

Avant de poursuivre, notons que les fonctions appartenant à la boîte à outil Motif ont un nom commençant par "Xm", les fonctions de la boîte à outil *Intrinsics* commencent par "Xt" et les fonctions de bas niveau de Xlib commencent par un "X".

Initialisation

L'initialisation est effectuée par deux commandes. La première fait appel à la fonction de la boîte à outil *Intrinsics* `XtSetLanguageProc()` et sert à la configuration du langage de la boîte à outil et des formats de date et heure. Ici nous avons utilisé le langage par défaut, soit l'anglais, par les trois options "`NULL`".

```
XtSetLanguageProc ( NULL, NULL, NULL );
```

La deuxième commande appelle la fonction `XtVaAppInitialize()` afin d'effectuer l'initialisation proprement dite de la boîte à outil *Intrinsics*. Cette fonction ouvre une connexion avec le serveur X Window du système et elle retourne une valeur de type "Widget".

```
top = XtVaAppInitialize (&app,"Postproc",NULL,0,&argc,argv,NULL,NULL );
```

Fenêtre principale

Suite à l'initialisation, on peut alors passer à la création des trucs. D'abord la création de la fenêtre "mainwin" que l'on retrouve sur la figure 5.4 est créée par:

```
mainwin = XtVaCreateManagedWidget("mainwin",
                  xmMainWindowWidgetClass, top,
                  XmNtitle, "Biomag: Postprocessor Display",NULL );
```

Motif possède un système complexe de classes de trucs qui ne sera pas exposé en détail dans ce document. Disons seulement que le truc "mainwin" appartient à la classe nommée `xmMainWindowWidgetClass`, classe de la librairie Motif Xm qui agit comme fenêtre principale dans notre application. Elle servira à supporter la barre de menu et la zone de dessin graphique.

Zone graphique

La zone graphique est créée par la ligne de programme :

```
draw = XtVaCreateWidget ( "draw", xmDrawingAreaWidgetClass, mainwin,
                          XmNwidth, draw_x,
                          XmNheight, draw_y,
                          XmNtitle, "Biomag: Postprocessor Display",
                          NULL );
```

Cette fois-ci, on effectue la création d'un truc appartenant à la classe de zone de dessin nommée `xmDrawingAreaWidgetClass`. On la définit comme étant "fille" de la fenêtre "mainwin" et de dimensions "`draw_x`" et "`draw_y`". C'est dans cette zone graphique que seront affichées les informations par des appels aux fonctions de dessin de Xlib. Lors de la création d'une telle fenêtre, il est important de s'assurer que son contenu sera redessiné lorsque celle-ci sera mise en avant plan après avoir été cachée par une autre fenêtre, et lorsqu'il y aura redimensionnement de la dite fenêtre. Les deux lignes de programme suivantes servent à gérer ces deux cas particuliers. La fonction `expose_callback()` sera appelée dans le premier cas et la fonction `resize_callback()` prendra en charge le deuxième cas.

```
XtAddCallback (draw, XmNexposeCallback, expose_callback, NULL);
XtAddCallback (draw, XmNresizeCallback, resize_callback, NULL);
```

Contexte graphique

Pour une raison d'efficacité du protocole de X Window, les fonctions de dessin de Xlib ne transportent pas beaucoup d'information sur les options de dessin à effectuer telles que la couleur d'arrière plan, la couleur d'avant plan, l'épaisseur des lignes, le type de remplissage, etc. Ces informations sont contenues dans un contexte graphique (Graphic context), et chacun des appels aux fonctions de dessin se font avec ce contexte graphique.

Lorsque le programmeur veut dessiner avec des options de dessin différentes, il doit d'abord modifier le contexte graphique et ensuite dessiner en appelant ce même contexte graphique. La ligne suivante permet donc de créer ce contexte graphique.

```
gc = XCreateGC (  XtDisplay ( draw ),
                  RootWindowOfScreen ( XtScreen ( draw )),
                  GCForeground | GCBackground,  &gcv );
```

5.3.2 Menus

Nous allons voir dans cette section la création et la configuration de la barre de menu. D'abord, il y a création du truc "menubar" composé de six boutons qui sont nommés en langage de Motif "`button_0`" à "`button_5`". Ces noms serviront entre autres lorsqu'on voudra placer le menu d'aide à la droite de la barre de menu. La commande `XmVaCreateSimpleMenuBar` de la librairie Motif est une commande de haut niveau qui sert exclusivement à la création de la barre de menu.

```
menubar =   XmVaCreateSimpleMenuBar ( mainwin, "menubar",
            XmVaCASCADEBUTTON, XmStringCreateLocalized ("File"),    'F',
            XmVaCASCADEBUTTON, XmStringCreateLocalized ("Image"),   'I',
            XmVaCASCADEBUTTON, XmStringCreateLocalized ("Zoom"),    'Z',
            XmVaCASCADEBUTTON, XmStringCreateLocalized ("Colors"),  'C',
            XmVaCASCADEBUTTON, XmStringCreateLocalized ("Options"), 'O',
            XmVaCASCADEBUTTON, XmStringCreateLocalized ("Help");    'H',
            NULL );
```

On remarque les noms des différents menus et la première lettre qui sera soulignée à l'écran permettant d'accéder les menus sans souris par les combinaisons de touches comme Alt+F. Le standard Motif exige que le menu aide soit placé du coté droit de la barre de menu, ce qui est fait par:

```
if (widget = XtNameToWidget (menubar, "button_5"))
    XtVaSetValues (menubar, XmNmenuHelpWidget, widget, NULL);
```

Le menu "Fichier"

La création des options des différents menus s'effectue avec l'aide le la fonction de la boîte à outil Motif `XmVaCreateSimplePulldownMenu`. Le premier menu sur la gauche est le menu de lecture, de sauvegarde et d'impression des fichiers. C'est une autre spécification de Motif que de placer ce menu complètement à gauche. Il est créé par:

```
menu_fichier = XmVaCreateSimplePulldownMenu ( menubar, "menu_fichier", 0,
                                              menufichier,
XmVaPUSHBUTTON, XmStringCreateLocalized ("Open Articul"),'A',NULL,NULL,
XmVaPUSHBUTTON, XmStringCreateLocalized ("Open HF"),'H',NULL,NULL,
XmVaPUSHBUTTON, XmStringCreateLocalized ("Open LF"),'L',NULL,NULL,
XmVaPUSHBUTTON, XmStringCreateLocalized ("Save"),'S',NULL,NULL,
XmVaSEPARATOR,
XmVaPUSHBUTTON, XmStringCreateLocalized ("Print"),'P',NULL,NULL,
XmVaSEPARATOR,
XmVaPUSHBUTTON, XmStringCreateLocalized ("Quit"),'Q',NULL,NULL,NULL );
```

Il est important de remarquer le quatrième argument passé à cette commande, soit "`menufichier`", qui est le nom de la fonction qui sera appelée lorsqu'un des items de cette barre de menu sera sélectionné. On ne verra pas en détail ces fonctions; il s'agit simplement de fonctions qui en appellent d'autres beaucoup plus importantes sur lesquelles nous porterons notre attention.

Le menu fichier permet d'ouvrir trois formats de fichier différents qui seront expliqués en détail dans la section 5.4.2. Lors de l'ouverture d'un de ces trois types de fichier, une boîte de dialogue permettra la sélection du fichier en question, tel que montré à la figure 5.5.

Figure 5.5: Boîte de dialogue de sélection de fichier.

On peut vraiment apprécier la puissance de la boîte à outil de Motif en regardant le code qui génère et affiche cette boîte de sélection de fichier.

```
dialog=(Widget)XmCreateFileSelectionDialog(top,"Ouvrir Fichier HF",
                                                          args,n);
XtAddCallback ( dialog, XmNokCallback, load_file_HF, NULL);
XtAddCallback(dialog,XmNcancelCallback,CallbackProc)XtUnmanageChild,NULL);
XtManageChild (dialog);
XtPopup (XtParent (dialog), XtGrabExclusive);
```

Il aurait fallu un nombre considérable de lignes de programmation et plusieurs heures de travail pour effectuer la même tâche au niveau de la boîte à outil Xt ou au niveau de Xlib, et il n'est pas certain que l'apparence requise par le standard Motif ait été respectée. L'option "Save" du menu permet de sauvegarder les fichiers; seul le format "Articulation" est supporté dans cette version. L'option "Print" n'est pas implantée et finalement l'option "Quit" permet bien évidemment de quitter le logiciel de façon propre et adéquate.

Le menu "Image"

Le menu "Image" permet d'accéder aux options de translation et de rotation. Premièrement, la translation permet de changer l'emplacement de l'image sur l'écran. Une boîte de dialogue permet d'entrer les coordonnées du déplacement à effectuer sous la forme "*X,Y*".

Figure 5.6: Boîte de dialogue de translation.

Il est à noter que dans cette version de l'interface graphique, l'origine de l'écran est située dans le coin supérieur droit de l'écran. Tout comme dans le cas du menu "Fichier", la fonction `XmVaCreateSimplePulldownMenu` est appelée pour effectuer la création du menu.

```
XmVaCreateSimplePulldownMenu ( menubar, "menu_image", 1, menuimage,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Translation"),'T', NULL, NULL,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Rotation"),'R',NULL,NULL,NULL,
NULL );
```

Et la boîte de dialogue montrée sur la figure 5.6 est créée par les fonctions de la librairie de Motif:

```
prompt = XmStringCreateLocalized ("Entrez X,Y :");
XtSetArg (args[n], XmNtitle, "Translation"); n++;
XtSetArg (args[n], XmNselectionLabelString, prompt); n++;
XtSetArg (args[n], XmNautoUnmanage, False); n++;
XtSetArg (args[n], XmNuserData, 0); n++;
dialog = XmCreatePromptDialog (top, "XY", args, n);
```

La deuxième option du menu "Image" est la rotation. La figure 5.7 montre la boîte de dialogue qui permet d'effectuer cette opération.

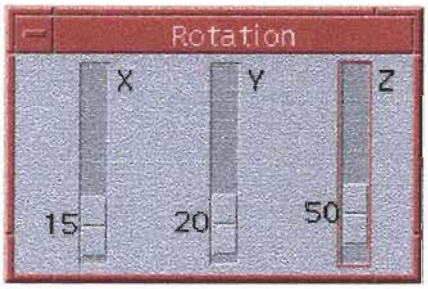

Figure 5.7: Boîte de dialogue de rotation.

Les trucs de la classe "`xmScaleWidgetClass`" ont été utilisés au nombre de trois afin d'effectuer la rotation dans les trois axes simultanément. Une technique quelque peu différente a été utilisée ici afin de créer cette boîte de dialogue. D'abord la commande `XtCreatePopupShell` a été utilisée pour créer un "*shell*" contenant un truc de classe "*rowcol*" qui à son tour supporte les trois trucs de type "*scale*".

```
shell = XtCreatePopupShell("shell",xmDialogShellWidgetClass,top,args, n);
rowcol = XtCreateWidget ("rowcol",xmRowColumnWidgetClass,shell, args, n);

scale = XtVaCreateManagedWidget ("X", xmScaleWidgetClass, rowcol,
        XtVaTypedArg, XmNtitleString, XmRString, "X", 2,
        XmNmaximum,   360, XmNminimum,   0, XmNscaleMultiple,     5,
        XmNshowValue, True, NULL);
XtAddCallback (scale, XmNvalueChangedCallback, Rotation, NULL);
XtManageChild (rowcol);
```

La ligne "`scale = XtVaCreateManagedWidget...`" est répétée pour les deux autres axes de rotation Y et Z.

Le menu "Zoom"

Le menu "Zoom" contenant les items "In" et "Out" vient ensuite dans l'ordre de gauche à droite dans la barre de menu de l'interface graphique. Sa création est similaire à celle des autres items de la barre de menu.

Le facteur d'échelle de la fonction zoom ne peut être modifié dans cette version autrement que par programmation.

Le menu "Colors"

Le menu "Colors" permet de changer la couleur de fond de l'écran entre le noir et le blanc. La facilité d'interprétation de l'image est supérieure en fond noir lorsque l'image est affichée à l'écran. Cette option a été conçue spécialement pour le cas où les images doivent être exportées vers un logiciel de traitement de texte effectuant l'impression des documents en noir et blanc ou en couleurs. La qualité graphique est ainsi améliorée sur papier. La création de ce menu se fait par:

```
XmVaCreateSimplePulldownMenu ( menubar, "menu_coul", 3, menucouleurs,
XmVaPUSHBUTTON,XmStringCreateLocalized("Black Background"),'B',NULL,
NULL,
XmVaPUSHBUTTON,XmStringCreateLocalized("White Background"),'W', NULL,
NULL,NULL,NULL );
```

Le menu "Options"

Le menu "Options" permet de configurer les données affichées sur l'écran afin de mieux interpréter les résultats disponibles. Sa création est encore une fois similaire aux autres items du menu.

```
XmVaCreateSimplePulldownMenu ( menubar, "menu_opt", 4, menuoptions,
XmVaPUSHBUTTON,XmStringCreateLocalized("Hide zero values"),'H',NULL,
NULL,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Show zero values"),'S', NULL,
NULL,
XmVaSEPARATOR,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Display by colors"),'D', NULL,
NULL,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Show all colors"),'S', NULL,
NULL,
XmVaSEPARATOR,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Display by plan"),'p', NULL,
NULL,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Next plan"),'N', NULL, NULL,
```

```
XmVaPUSHBUTTON,XmStringCreateLocalized ("Prev plan"),'P', NULL, NULL,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Show all plans"),'a', NULL,
NULL,
XmVaSEPARATOR,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Hold ON"),'O', NULL, NULL,
XmVaPUSHBUTTON,XmStringCreateLocalized ("Hold OFF"),'F', NULL, NULL,
NULL, NULL );
```

Les deux premières options "Hide zero values" et "Show zero values" permettent d'afficher ou de ne pas afficher les valeurs nulles des données affichées à l'écran. Cette option permet à l'usager de mieux se concentrer sur les valeurs qui offrent un plus grand intérêt, soit les valeurs non nulles. La première option, "Hide zero values" enlève donc les valeurs nulles sur l'écran et la deuxième les replace.

Les deux autres options qui suivent , "Display by colors" et "Show all colors" donnent la possibilité à l'utilisateur d'afficher une seule couleur. Une boîte de dialogue permet d'entrer le numéro de la couleur qui apparaît à gauche de la légende. Conjointement avec les deux dernières options, soient "Hold ON" et "Hold OFF", il est possible d'afficher plus d'une seule couleur à la fois. L'usager doit d'abord utiliser l'option "Display by colors" afin d'afficher une seule couleur et activer ensuite l'option "Hold ON". Le mot "HOLD" apparaîtra alors au bas de la règle de couleur de la légende. Par la suite, l'usager peut afficher plusieurs couleurs qui seront superposées à l'écran. La figure 5.8 montre une telle situation.

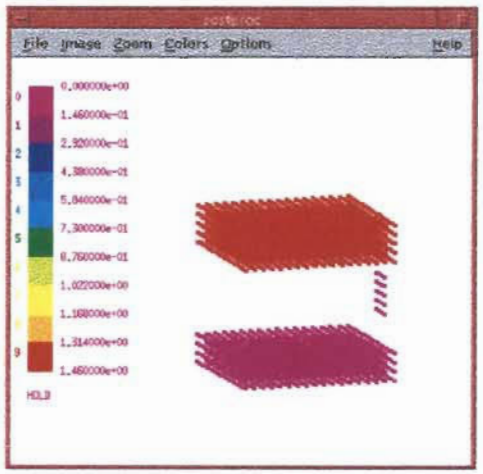

Figure 5.8: Affichage de deux couleurs seulement.

Les autres options du menu portant le même nom concernent les plans. On peut, par l'option "Display by plan" afficher une seul plan en spécifiant dans la boîte de dialogue l'axe et la valeur à afficher (*x=15* par exemple). Deux options permettent de parcourir les plans dans un sens ou dans l'autre, soient "Next plan" et "Prev plan". Comme dans le cas de l'affichage par couleur, l'usager peut utiliser l'option "Hold ON" pour réaliser un affichage de plusieurs plans superposés, tel que montré à la figure 5.9

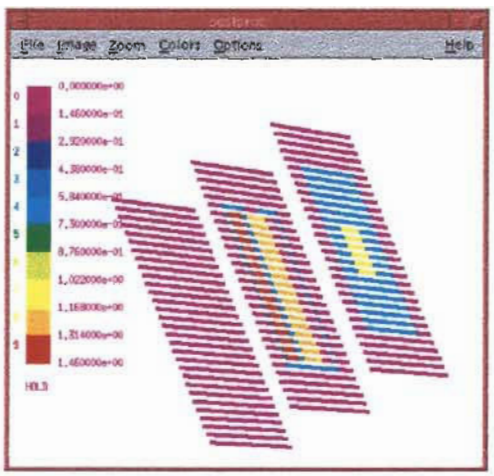

Figure 5.9: Affichage de trois plans superposés.

Les dernières étapes de la configuration des menus est en fait de les afficher sur l'écran du serveur X Window. Les commandes suivantes effectuent cette tâche.

```
XtManageChild ( draw );
XtManageChild ( menubar );
XtRealizeWidget ( top );
Fond_noir();
XtAppMainLoop ( app );
```

Il faut se souvenir qu'il s'agit ici d'un programme de type "événement". La fonction `XtAppMainLoop` est celle qui permet de détecter les mouvements de la souris afin d'exécuter conjointement avec le gestionnaire de fenêtre (window manager) les tâches correspondant aux options des menus et les déplacements et redimensionnements des fenêtres de l'application de l'interface graphique.

5.4 Implantation Graphique

La section précédente s'est surtout concentrée sur la programmation X Window avec la librairie Motif. Cette étape est nécessaire afin de réaliser un environnement graphique avec lequel il est facile de travailler, tout comme dans le cas d'une application commerciale que l'on peut acheter à gros prix. Nous allons maintenant nous tourner vers la base de données constituant les fondements du graphisme réalisé dans cette interface graphique. Au début du projet, la structure commune aux différents modules n'était pas encore connue, et on se devait d'effectuer des essais en graphisme avec les fonctions de rotation, translation, etc. Pour se faire, on a utilisé une base de données que nous avons appelé "Articulation" qui sera décrite dans la section 5.4.2. Par la suite, lorsque les modules de calcul en basses fréquences et en hautes fréquences furent prêt, nous avons utilisé une structure différente s'adaptant aux modules de calcul.

5.4.1 Base de données graphique

La représentation graphique des objets dans l'espace tridimensionnel nécessite l'utilisation des coordonnées x, y et z. Afin de comprendre comment la structure graphique est réalisée ici, prenons l'exemple d'un simple cube.

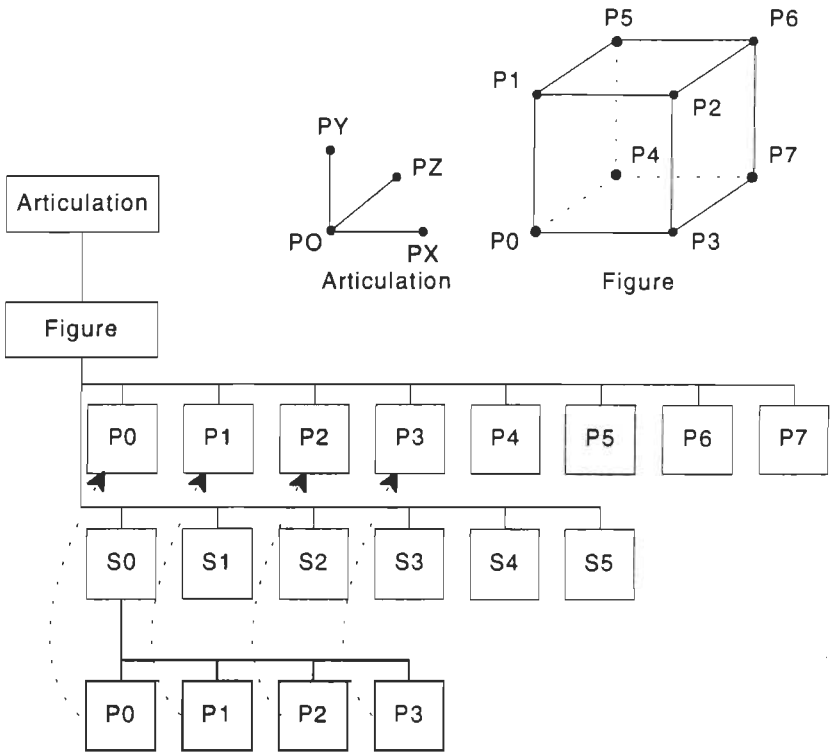

Figure 5.10 : La structure graphique.

L'articulation, comprenant les points *PO*, *PX*, *PY*, et *PZ* sert de référence quant à l'emplacement et l'orientation de la figure dans l'espace tridimensionnel. Lorsqu'une figure est déplacée ou tournée selon l'un des axes *X*, *Y* ou *Z*, les coordonnées *PX*, *PY* et *PZ* le sont aussi par rapport à l'origine *PO*. Le mouvement de la figure peut donc être fait par rapport à la position actuelle de la figure, et non par rapport à la position originale. L'articulation, peut comprendre une ou plusieurs figures, chacune de celles-ci étant composée de points et de surfaces. Dans le cas du cube, il est composé de 8 points et de 6

surfaces, chacune étant elle-même composée de 4 points. Ces points sont référencés en langage C par un pointeur sur les points composant la figure. Ceci permet d'économiser l'espace mémoire à allouer pour une figure. La figure 5.10 représente l'ensemble de cette structure, les lignes pointillées signifiant l'utilisation de pointeurs.

En langage C, cette structure est inscrite comme :

```
typedef struct
{
      double      x, y, z;
      double      ppx, ppy;
      short int   x0,y0,z0;   // Valeurs Originales pour affichage plan
      double      valeur;     // Valeur du point (pot. vecteur, etc..)
      short int   coul;       // Couleur (0-MAXCOLORS)
      int         hold;       // Hold (VRAI|FAUX)
} POINT;

typedef struct
{
        POINT **pts;          // Pointeur sur un point
        int     numpts;       // Nombre de points
} SURFACE;

typedef struct
{
        SURFACE *surf;        // Pointeur sur une surface
        POINT *pts;           // Pointeur sur un point
        int numsurf, numpts;  // Nombre de surface et nombre de points
} FIGURE;

typedef struct
{
        POINT PO,PX,PY,PZ;    // Coordonnées de l'origine
        FIGURE *fig;          // Pointeur sur une figure
        int numfig;           // Nombre de figure
} ARTICUL;
```

Il est plus simple de lire cette partie de code en commençant par le bas car elle représente dans cette direction ce qui se trouve sur la figure 5.10 de haut en bas ; donc on retrouve le type `ARTICUL` qui est composé de 4 points (`POINT PO,PX,PY,PZ;`), d'un pointeur sur une figure (`FIGURE *fig;`) et d'un entier représentant le nombre de figures (`int numfig;`). Par la suite, en remontant vers le haut, on retrouve le type `FIGURE` qui est

composé d'un pointeur sur une surface (`SURFACE *surf;`), d'un pointeur de points (`POINT *pts;`) et de deux entiers définissant le nombre de points et le nombre de surfaces (`int numsurf, numpts;`). En remontant vers le haut toujours, on a un type `SURFACE` comportant un pointeur de pointeur de point (`POINT **pts;`) et un nombre de points par surface (`int numpts;`). Finalement, le type `POINT` est défini par 3 variables de type double pour les coordonnées x, y et z du point (`double x, y, z;`), par deux autres variables représentant les points de projection en x et y (`double ppx, ppy;`) et par la valeur calculée du point qui est fournie par le module de calcul. De plus, le point comprend les valeurs x0, y0 et z0 qui conserveront les valeur originales des coordonnées du point. Ces valeurs seront utiles dans la fonction d'affichage par plan. Finalement, la variable "`coul`" représente la couleur à afficher pour le point et la variable "`hold`" détermine si oui ou non le point a été mis en "hold" par l'option "hold ON" décrite dans la section 5.3.2.

Donc pour définir cette structure en programmation, il faut simplement écrire la ligne suivante :

```
ARTICUL a ;
```

définissant ainsi une variable `a` de type `ARTICUL`.

5.4.2 Formats de fichiers

La communication entre les différents modules constituant l'analyse parallèle des biochamps se fait par l'intermédiaire de fichiers binaires. Les modules de calcul hautes fréquences et basses fréquences écrivent leurs résultats dans un fichier qui est ensuite lu par l'interface graphique. Dans cette section, nous allons décrire premièrement le format

"Articulation" qui a servi pour fin de test au tout début du projet pour ensuite définir les formats des fichiers des modules de calcul hautes et basses fréquences.

Articulation

Le format de fichier suivant est utilisé pour représenter et stocker les objets graphiques sous forme "ASCII", ou en format texte si vous préférez. Le première ligne (`Articulation (1)`) donne l'information sur le nombre total de figures. Par la suite, les quatre points de référence sont définis. On retrouvera aussi la ligne donnant le nombre de points et le nombre de surfaces pour chacune des figures (`Figure (8,6)`). Seront donc mis en liste tous les points et toutes les surfaces composant la figure. Les lignes suivantes décrivent exactement un cube de 100x100x100.

```
Articulation (1)
PO=500,500,150
PX=510,500,150
PY=500,510,150
PZ=500,500,160
Figure (8,6)
P0=(450,450,100)
P1=(450,550,100)
P2=(550,550,100)
P3=(550,450,100)
P4=(450,450,200)
P5=(450,550,200)
P6=(550,550,200)
P7=(550,450,200)
S0(4)=(P0,P1,P2,P3)
S1(4)=(P3,P2,P6,P7)
S2(4)=(P7,P6,P5,P4)
S3(4)=(P4,P5,P1,P0)
S4(4)=(P1,P5,P6,P2)
S5(4)=(P4,P0,P3,P7)
```

Tel que mentionné plus haut, cette structure fut utilisée au tout début du projet afin de tester l'environnement graphique Motif, et les fonctions de rotation et de translation. La

même structure graphique fut utilisée, soit celle qui a été décrite en 5.4.1. Cette description représente exactement le cube de la figure 5.10.

Hautes fréquences

Pour ce qui est des hautes fréquences, le fichier utilise d'abord trois valeurs entières (`int`) au tout début. Ces trois valeurs représentent le nombre de points dans les directions x, y et z. Viennent par la suite les valeurs associées à chacun des points pour chacun des plans x-y. Le total de points à lire sera le résultat des trois premières valeurs multipliées entre elles.

```
for ( z=0; z<nbz; ++z)
    for ( y=0; y<nby; ++y)
        for ( x=0; x<nbx; ++x)
        {
           fread (&buffer[ctr], sizeof(float), 1, sol);
           ++ctr;
        }
```

Dans la section de code ci-haut, `nbx`, `nby` et `nbz` sont les trois premières valeurs lues dans le fichier. Les trois boucles "`for`" imbriquées effectuent la lecture de `nbx*nby*nbz` valeurs de type `float`. La totalité du fichier est donc lue après l'exécution des ces instructions.

Basses fréquences

Le format de fichier pour les basses fréquences est quelque peu différent. On y retrouve trois valeurs de type "`float`" représentant les cordonnées x, y et z du point, et la valeur de point à afficher ensuite étant de type "`float`" également. Les lignes de codes suivantes effectuent la boucle de lecture de ces points.

```
for (i=0; i<a.fig[0].numpts; ++i)
{
```

```
    if ( ( fread (&x, sizeof(float), 1, sol)) != 1)
    {
        printf ("Erreur de lecture de x\n");
        exit(0);
    }
    if ( ( fread (&y, sizeof(float), 1, sol)) != 1)
    {
        printf ("Erreur de lecture de y\n");
        exit(0);
    }
    if ( ( fread (&z, sizeof(float), 1, sol)) != 1)
    {
        printf ("Erreur de lecture de z\n");
        exit(0);
    }
    if ( ( fread (&valeur, sizeof(float), 1, sol)) != 1)
    {
        printf ("Erreur de lecture de valeur\n");
        exit(0);
    }
}
```

5.4.3 Opérations

Nous allons décrire dans cette section les opérations de base du graphisme qui ont été implantées dans l'interface graphique. Comme il est important de bien comprendre ces opérations, nous allons d'abord décrire la technique séquentielle qui a été utilisée. Dans le chapitre 6, les fonctions qui ont été mise en parallèle seront revues dans leur forme parallèle.

Dans un premier temps, il y a lieu de voir comment sont établies les bases de ces opérations. Nous allons prendre pour exemple un cube afin de bien cerner les explications.

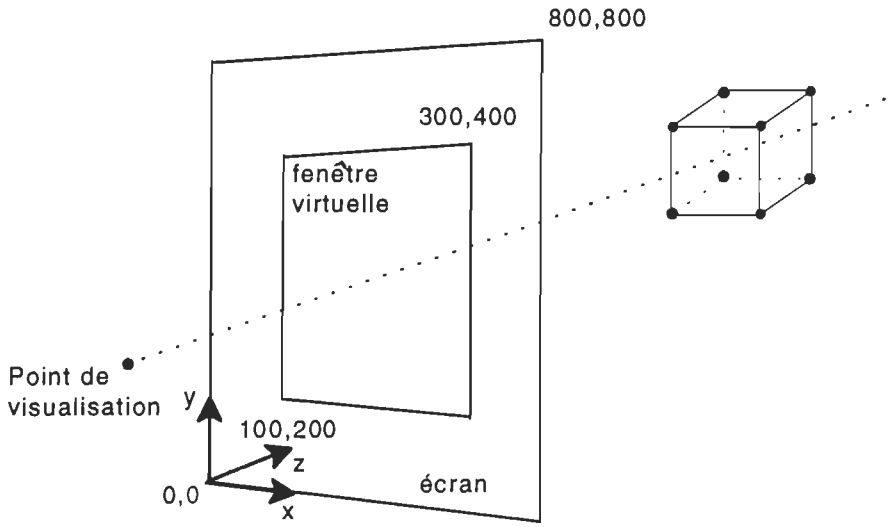

Figure 5.11: Vision en trois dimensions.

La figure 5.11 représente la façon utilisée dans ce travail pour effectuer la vision en trois dimensions. L'objet, le cube, est vu à travers une fenêtre virtuelle de dimension fixe. Cette dimension peut être modifiée pour effectuer l'opération de "zoom" que l'on verra plus tard. On remarque l'inversion de l'axe des z par rapport à ce qui a été vu dans la partie théorique au chapitre 4. Il est plus utile de travailler de cette façon en graphisme par ordinateur, les valeurs plus positives de z étant plus loin de l'œil.

```
struct
{
    double  x;
    double  y;
    double  z;
} pv;

double  vir_deb_x = 100.0;          /* Fenetre virtuelle debut x       */
double  vir_deb_y = 200.0;          /* Fenetre virtuelle debut y       */
double  vir_fin_x = 300.0;          /* Fenetre virtuelle fin x         */
double  vir_fin_y = 400.0;          /* Fenetre virtuelle fin y         */
double  ecr_deb_x =   0.0;          /* Fenetre ecran debut x           */
double  ecr_deb_y =   0.0;          /* Fenetre ecran debut y           */
double  ecr_fin_x = 800.0;          /* Fenetre ecran fin x             */
double  ecr_fin_y = 800.0;          /* Fenetre ecran fin y             */

pv.x = (vir_fin_x + vir_deb_x)/2;
pv.y = (vir_fin_y + vir_deb_y)/2;
pv.z = -1000.0;
```

Le segment de code précédent montre comment sont définies les dimensions des deux fenêtres et comment le point de visualisation est défini. Le point de visualisation est placé au centre de la fenêtre virtuelle à 1000 unités derrière cette même fenêtre.

Rotation

Tel que décrit au chapitre 4 portant sur la théorie du graphisme par ordinateur, la fonction de rotation utilise la matrice rotation (4.27) en coordonnées homogènes. Cette matrice est définie par les lignes de programmation:

```
A[0][0] = vx * vx * (1-cosinus) + cosinus;
A[0][1] = vx * vy * (1-cosinus) - (vz * sinus);
A[0][2] = vx * vz * (1-cosinus) + (vy * sinus);
A[1][0] = vx * vy * (1-cosinus) + (vz * sinus);
A[1][1] = vy * vy * (1-cosinus) + cosinus;
A[1][2] = vy * vz * (1-cosinus) - (vx * sinus);
A[2][0] = vx * vz * (1-cosinus) - (vy * sinus);
A[2][1] = vy * vz * (1-cosinus) + (vx * sinus);
A[2][2] = vz * vz * (1-cosinus) + cosinus;
A[3][3] = 1;

/* mettre ligne 4 a 0,0,0,1 et colonne 4 a 0,0,0,1 */
for (i = 0; i <= 2; ++i)
{
    A[3][i] = 0;
    A[i][3] = 0;
}
```

Les variables vx, vy et vz sont les composantes normalisées de la droite autour de laquelle doit s'effectuer la rotation. Dans la présente version de l'interface graphique, la rotation s'effectue seulement sur les axes principaux x, y et z. Il y a donc toujours deux de ces composantes qui sont nulles. La fonction de rotation reçoit donc l'angle autour duquel la rotation doit se faire, et l'angle de rotation. Le vecteur normal de rotation est calculé par:

```
racine=sqrt(((p1x-p2x)*(p1x-p2x))+((p1y-p2y)*(p1y-p2y))+((p1z-2z)*(p1z-
p2z)));
radian  = 2 * PI * (double) angle / 360;
cosinus = cos((double)radian);
sinus   = sin((double)radian);
```

```
vx = (p2x - p1x) / racine;
vy = (p2y - p1y) / racine;
vz = (p2z - p1z) / racine;
```

Suite à cela, les deux matrices de translation sont bâties. Comme on l'a vu au chapitre 4, l'opération de rotation doit être réalisée conjointement avec une translation positive et une translation négative. Les deux matrices T1 et T2 réaliseront donc ces deux translation et le produit matriciel sera calculé.

```
/* creation des matrice de translation T1 et T2 */
for (i = 0; i <= 3; ++i)
    for (j = 0; j <= 3; ++j)
    {
        if (i == j)
        {
            T1[i][j] = 1 ; T2[i][j] = 1;
        }
        else
        {
            T1[i][j] = 0 ; T2[i][j] = 0;
        }
    }
/* creation de la ligne 4 de T1 et T2 */
    T1[3][0] = -p1x;
    T1[3][1] = -p1y;
    T1[3][2] = -p1z;
    T2[3][0] = p1x;
    T2[3][1] = p1y;
    T2[3][2] = p1z;
/* calcul du produit matriciel entre T1 et A */
    for (i=0; i<4; ++i)
        for (j=0; j<4; ++j)
        {
            tmp[i][j] = 0;
            for (k=0; k<4; ++k)
                tmp[i][j] += (T1[i][k] * A[k][j]);
        }
/* calcul du produit matriciel entre le resultat de T1 et A par T2 */
    for (i=0; i<4; ++i)
        for (j=0; j<4; ++j)
        {
            total[i][j] = 0;
            for (k=0; k<4; ++k)
                total[i][j] += (tmp[i][k] * T2[k][j]);
        }
```

Nous nous retrouvons maintenant dans la situation où la matrice de transformation complète a été calculée et il ne reste plus qu'à évaluer les nouvelles coordonnées de chacun

des points par la fonction "Transformation". Elle effectue d'abord la transformation sur les points des coordonnées de référence PX, PY et PZ pour ensuite entrer dans une boucle effectuant la transformation sur tous les points de la figure. En fait, cette fonction effectue la multiplication de la matrice de transformation complète par les coordonnées du point étant l'objet de la rotation.

```
void Transformation (double total[4][4])
{
    int i, j;
    Transforme_point (&(a.PX), total);
    Transforme_point (&(a.PY), total);
    Transforme_point (&(a.PZ), total);
    for (i=0; i<a.numfig; ++i)
        for (j=0; j<a.fig[i].numpts; ++j)
            Transforme_point (&(a.fig[i].pts[j]), total);
}

void  Transforme_point (POINT *pt, double total[4][4])
{
    double x,y,z;
    x = pt->x;
    y = pt->y;
    z = pt->z;
    pt->x = x*total[0][0] + y*total[1][0] + z*total[2][0] +total[3][0];
    pt->y = x*total[0][1] + y*total[1][1] + z*total[2][1] +total[3][1];
    pt->z = x*total[0][2] + y*total[1][2] + z*total[2][2] +total[3][2];
}
```

Suite à l'opération de rotation, la projection est recalculée sur les points et l'affichage est effectué.

Translation

La translation est une opération plutôt simple, en théorie et en pratique. C'est pourquoi elle ne fut pas l'objet d'une quelconque mise en parallèle au cours de ce projet. La translation est d'abord effectuée sur les coordonnées x et y de l'axe des coordonnées de référence PO, PX, PY et PZ.

```
a.PO.x += x ; a.PO.y += y;
a.PX.x += x ; a.PX.y += y;
```

```
a.PY.x += x ; a.PY.y += y;
a.PZ.x += x ; a.PZ.y += y;
```

Par la suite, la translation est calculée sur les coordonnées x et y de tous les points de la figure, et finalement la projection est recalculée et l'affichage est refait pour refléter les changements à la figure.

```
for (i=0; i< a.numfig; ++i)
    for (j=0; j< a.fig[i].numpts; ++j)
    {
        a.fig[i].pts[j].x += x;
        a.fig[i].pts[j].y += y;
    }
Projection ();
Affiche_Articul (&a);
```

Projection

La méthode de projection permet l'affichage d'objets en trois dimensions sur un plan à deux dimensions tel qu'un écran cathodique. On la calcule en prenant comme référence que le rapport de *ppx* sur *d* sera égal au rapport des coordonnées x et z du point à "projeter", comme on peut le voir sur la figure 5.12.

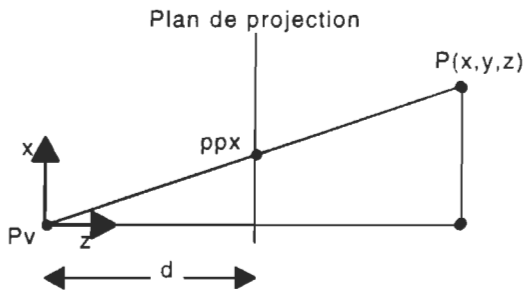

Figure 5.12: Méthode de projection.

On peut donc dire que:

$$\frac{ppx}{d} = \frac{x}{z}; \quad \frac{ppy}{d} = \frac{y}{z} \tag{5.1}$$

En multipliant chaque côté de l'équation (5.1) par d, on obtient:

$$ppx = \frac{d \cdot x}{z} = \frac{x}{z/d}; \qquad ppy = \frac{d \cdot y}{z} = \frac{y}{z/d} \tag{5.2}$$

C'est exactement ce qui est implanté dans le code suivant sous une forme quelque peu différente car le zéro de l'axe des z est situé au niveau du plan de projection, contrairement à ce que l'on retrouve sur la figure 5.12.

```
for (i=0; i<a.numfig; ++i)
    for (j=0; j<a.fig[i].numpts; ++j)
    {
        a.fig[i].pts[j].ppy=pv.y + (((a.fig[i].pts[j].y - pv.y) * pv.z) /
                                    (a.fig[i].pts[j].z - pv.z));
        a.fig[i].pts[j].ppx=pv.x + (((a.fig[i].pts[j].x - pv.x) * pv.z) /
                                    (a.fig[i].pts[j].z - pv.z));
    }
```

Rognage

Le rognage (*clipping*) est la technique utilisée en graphisme pour déterminer si un objet ou une partie d'un objet est visible sur l'écran. L'interface graphique discutée dans ce document effectue le rognage des lignes. Si une ligne est visible, elle sera affichée, si elle n'est pas visible, elle ne sera pas affichée, et si une partie de la ligne est visible, la partie visible sera affichée.

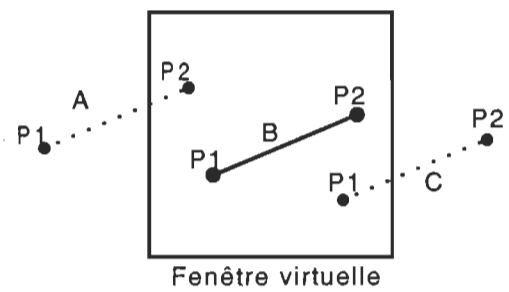

Figure 5.13: Rognage des lignes.

La fonction de rognage de l'interface graphique reçoit deux points comme paramètres et retourne une valeur entière. Les deux points sont les extrémités de la ligne à

afficher et la valeur retournée est 1 si la ligne est invisible et 0 si elle est visible. Dans une première étape, on effectue le rognage sur l'axe des x et ensuite la deuxième étape effectue le rognage en y. D'abord, il faut ordonner les points P1 et P2 selon leur coordonnées en x:

```
if (p1x > p2x)
{
    tampon = p1x;
    p1x = p2x;
    p2x = tampon;
    tampon = p1y;
    p1y = p2y;
    p2y = tampon;
}
```

Suite à cela, on peut effectuer le rognage proprement dit. Comme P1 et P2 sont ordonnées en x, on vérifie d'abord que P1 ne soit pas plus grand que la limite supérieure en x de la fenêtre virtuelle. Si c'est le cas, la ligne sera alors complètement à l'extérieur de la fenêtre du côté droit et sera donc invisible dans sa totalité.

```
    if (p1x > vir_fin_x)
    {
        invis = VRAI;          /* le point est invisible */
        return(invis);
    }
```

La même opération est effectuée en comparant la coordonnées x du point P2 et la limite inférieure de la fenêtre virtuelle en x. Si donc la coordonnées x du point P2 est inférieure, la ligne sera donc située complètement à gauche de la fenêtre virtuelle et sera donc invisible. Dans ces deux cas, la fonction de rognage retourne la valeur 1 (`VRAI`).

```
    if (p2x < vir_deb_x)
    {
        invis = VRAI;         /* le point est invisible */
        return(invis);
    }
```

Maintenant, on doit vérifier si on se trouve dans les cas A ou C représentés sur la figure 5.13. On compare donc la position en x de P1 avec la limite de gauche de la fenêtre

virtuelle. Si cette comparaison est vraie, on a alors le cas A et on doit rogner la ligne avant de l'afficher.

```
if (p1x < vir_deb_x)
{
    p1y += (p2y - p1y) * (vir_deb_x - p1x) / (p2x - p1x);
    p1x = vir_deb_x;
    if ((p1y >= vir_fin_y) || (p1y <= vir_deb_y))
    {
      invis = VRAI;     /* le point est invisible */
      return(invis);
  }
}
```

La même opération est effectuée pour le point P2 afin de déterminer si on se trouve dans la situation C de la figure 5.13.

```
if (p2x > vir_fin_x)
{
    p2y -= (p2y - p1y) * (p2x - vir_fin_x) / (p2x - p1x);
    p2x = vir_fin_x;
    if ((p2y >= vir_fin_y) || (p2y <= vir_deb_y))
    {
        invis = VRAI;     /* le point est invisible */
        return(invis);
    }
}
```

Si aucun de ces cas n'est vrai, on se trouve donc dans la situation B de la figure 5.13 et la ligne à afficher est complètement visible. Aucun rognage ne doit alors être effectué. La fonction d'affichage de ligne appelle la fonction de rognage de la façon suivante:

```
if (!Clipping (&lp1, &lp2))
{
  Normalise (&lp1);
  Normalise (&lp2);
    XDrawLine ( dpy, w, gc, lp1.ppx, lp1.ppy, lp2.ppx, lp2.ppy);
}
```

Si donc la fonction de rognage retourne 0, l'affichage est effectué car la ligne est visible, ou encore la partie visible doit être affichée. La fonction de normalisation sera discutée dans la section suivante. Finalement, on remarque la fonction `XdrawLine`

appartenant à la librairie Xlib qui permet d'effectuer le traçage d'une ligne à l'écran. La figure 5.14 montre un exemple où le rognage a été effectué.

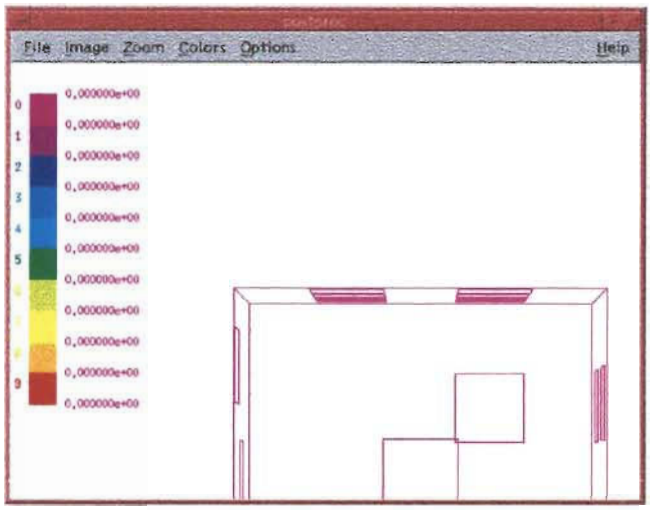

Figure 5.14: Exemple de figure rognée.

Le cas de la figure 5.14 représente l'affichage d'une figure en format "Articulation". Dans le cas de l'affichage des résultats de calcul en hautes et basses fréquences, le rognage s'effectue simplement sur un point et non sur une ligne.

Normalisation

La fonction de normalisation est utilisée dans la fonction d'affichage afin de "normaliser" les points de projection `ppx` et `ppy` les convertissant des coordonnées relatives de la fenêtre virtuelle vers les coordonnées de l'écran, ce dernier étant plus grand.

```
void Normalise(POINT *p)
{
    double  nx, ny;
    nx = (p->ppx - vir_deb_x)/(vir_fin_x - vir_deb_x);
    ny = (p->ppy - vir_deb_y)/(vir_fin_y - vir_deb_y);

    p->ppx = ecr_deb_x +(nx*(ecr_fin_x-ecr_deb_x));
    p->ppy = ecr_deb_y +(ny*(ecr_fin_y-ecr_deb_y));
    p->ppy = ecr_fin_y -(p->ppy - ecr_deb_y);
}
```

Zoom

La fonction zoom permet, comme dans tout logiciel de traitement graphique, de mieux voir les informations à l'écran en augmentant la dimension de l'objet à observer. Dans le contexte de la présente interface graphique, la fonction zoom est implantée de façon très simple. Il n'y a pas de possibilité de sélectionner une zone de l'écran avec la souris. Le menu zoom et ses deux options "In" et "Out" permettent d'effectuer des mouvements de l'objet à l'écran. Afin d'obtenir le résultat voulu, on change simplement les dimensions de la fenêtre virtuelle et on refait la projection et l'affichage.

```
switch (item_no)
{
case 0:     /*    Zoom in     */
    vir_fin_x -= ZOOM_FACTOR;
    vir_fin_y -= ZOOM_FACTOR;
    vir_deb_x += ZOOM_FACTOR;
    vir_deb_y += ZOOM_FACTOR;
break;

case 1:     /*    Zoom out    */
    vir_fin_x += ZOOM_FACTOR;
    vir_fin_y += ZOOM_FACTOR;
    vir_deb_x -= ZOOM_FACTOR;
    vir_deb_y -= ZOOM_FACTOR;
break;
}
```

La constante `ZOOM_FACTOR` est définie dans le fichier "postproc.h" et elle ne peut être modifiée que par programmation dans la présente version de l'interface graphique.

Affichage

La fonction d'affichage en mode séquentiel se divise en deux parties. D'abord, l'affichage des images en format de fichier "Articulation" tel que décrit en 5.4.2 sont affichées par la fonction "`Affiche_Articul()`" et les formats de ficher en basses et hautes

fréquences sont affichés par la fonction "`Affiche_HF`". Elle porte ce nom car elle a été écrite avant l'affichage en basses fréquences, mais elle affiche les deux formats de fichier.

Donc dans un premier temps, voyons le fonctionnement de la fonction d'affichage des images de format "Articulation". Cette fonction travaille de façon très exacte avec la structure des données graphiques implantée au début de ce projet comme on pourra le constater.

```
void Affiche_Articul (ARTICUL *p)
{
    int i;
    Window w = XtWindow(draw);
    XClearWindow (dpy, w);
    for (i=0; i< p->numfig; ++i)
        Affiche_Figure (&(p->fig[i]));
    Legende( );
}
```

La fonction reçoit un pointeur sur l'articulation à afficher et appelle ensuite la fonction `Affiche_Figure()` pour chacune des figures comprises dans l'articulation. Dans notre cas, il n'y a seulement qu'une figure par articulation, mais la structure de données est conçue pour en accepter plus d'une.

```
void Affiche_Figure (FIGURE *p)
{
    int i;
    for (i=0; i< p->numsurf; ++i)
        Affiche_Surface (&(p->surf[i]));
}
```

La fonction `Affiche_Figure()` reçoit donc un pointeur sur une figure et affichera toutes les surfaces de cette figure par l'appel de la fonction `Affiche_Surface()`.

```
void Affiche_Surface (SURFACE *p)
{
    int i;
    for (i=0; i< p->numpts-1; ++i)
        Affiche_Ligne ((p->pts[i]),(p->pts[i+1]));
    Affiche_Ligne ((p->pts[i]),(p->pts[0]));
}
```

À son tour, la fonction Affiche_Surface() appellera la fonction Affiche_Ligne() pour afficher chacune des lignes de la surface. Cette façon de faire peut paraître compliquée mais en fait, lorsque la structure graphique est bien comprise, c'est très simple et particulièrement efficace car toutes ces fonctions travaillent sur des pointeurs.

```
void Affiche_Ligne (POINT *p1, POINT *p2)
{
    Window w = XtWindow(draw);
    POINT   lp1, lp2;
    lp1 = *p1 ; lp2 = *p2;
    if (!Clipping (&lp1, &lp2))
    {
        Normalise (&lp1) ; Normalise (&lp2);
        XDrawLine ( dpy, w, gc, lp1.ppx, lp1.ppy, lp2.ppx, lp2.ppy);
    }
}
```

On remarquera l'utilisation de la fonction `Clipping()` qui effectue le rognage des lignes, et la fonction `Normalise()` qui effectue la normalisation des points avant affichage. Ces deux fonctions ont été décrites précédemment. Le traçage des lignes est fait par la fonction de la librairie Xlib nommée `XDrawLine()`.

Dans un deuxième temps, la fonction d'affichage des points en mode basses et hautes fréquences se veut complètement différente. Il s'agira maintenant d'effectuer l'affichage de données de simulation réelles qui devront être interprétées le plus facilement possible par les scientifiques. Il a été choisi de faire en sorte que la figure à afficher soit transparente pour l'utilisateur. Par contre, les options d'affichage dont nous verrons le codage plus loin permettent de sélectionner un ensemble de données permettant l'interprétation simple et efficace des résultats de simulation étudiés.

```
#define     FAUX  0
#define     VRAI  1

int         cache_zero = FAUX;
int         affiche_par_couleur = FAUX;
int         affiche_par_plan = FAUX;
int         plan_a_afficher;
```

```
int          hold_enable = FAUX;
```

Les valeurs ci-haut permettent conserver l'état des options d'affichage. La valeur de la variable `cache_zero` garde l'état de l'option d'enlèvement des valeurs nulles. Si cette valeur est "`VRAI`", les valeurs égales à zéro ne sont pas affichées. Il en va de même pour la variable `affiche_par_couleur` et `affiche_par_plan`. La variable `plan_a_afficher` conserve la coordonnée du plan à afficher dans l'option d'affichage par plan et finalement, la variable `hold_enable` sera "VRAI" si l'option "Hold ON" a été mise en service. La fonction d'affichage doit gérer toutes ces options et il en résulte un code assez complexe que nous allons tenter de comprendre partie par partie.

Comme il y a trois options différentes, soient l'enlèvement des valeurs nulles, l'affichage par couleur et l'affichage par plan, la fonction d'affichage consistera en 8 (2^3) conditions "si". Mais avant d'entrer dans ces conditions, la fonction effectue le rognage sur un point par la fonction `Clipping()`, la normalisation du point par la fonction `Normalise()` et finalement, elle active la couleur d'avant plan à utiliser pour ce point. Notons que la valeur de la couleur d'un point est calculée lors de la lecture du fichier en fonction des valeurs minimales et maximales de l'ensemble des points, et du nombre de couleurs maximum, soit 10. Notons simplement que la fonction `Affiche_HF()` utilise la fonction `Affiche_Rectangle()` pour effectuer l'affichage d'un petit rectangle dont la dimension est déterminée par la constante `REC_SIZE`.

```
void Affiche_Rectangle ( POINT *lp, int i)
{
    XDrawRectangle ( dpy, XtWindow(draw), gc, lp->ppx, lp->ppy, REC_SIZE,
REC_SIZE);
    XFillRectangle ( dpy, XtWindow(draw), gc, lp->ppx, lp->ppy, REC_SIZE,
REC_SIZE);
    if (hold_enable==VRAI && lp->hold==FAUX)
        a.fig[0].pts[i].hold = ON;
```

```
}
```

Cette fonction fait appel aux fonctions de la librairie Xlib `XdrawRectangle()` et `XfillRectangle()` pour afficher un rectangle à l'écran.

Maintenant que l'essentiel de la programmation dans la partie séquentielle est bien documenté dans les pages précédentes, il est temps de passer à la partie parallèle de la codification.

6 Mise en parallèle

6.1 Introduction

L'interface graphique a d'abord été programmée en séquentiel et en elle a été mise en parallèle, et ce pour simplifier l'apprentissage de la programmation graphique sur X Window. Alors que l'interface fut réalisée et son fonctionnement mis à l'épreuve en séquentiel, le temps était venu de réaliser la mise en parallèle proprement dite. Le format de fichier "Articulation" quant à lui a simplement servi dans les toutes premières étapes de la programmation et il ne sera pas mis en parallèle. Seul l'affichage des modules basses et hautes fréquences a été réalisé en parallèle. Pour ce faire, il aura fallu distribuer les données emmagasinées dans la base de données graphique sur tous les processeurs disponibles.

Un soin particulier à été porté lors de la programmation afin que l'interface puisse accepter n'importe quel nombre de processeurs jusqu'à un maximum de 64. De plus, l'interface graphique peut accepter un problème de dimension quelconque. Dans la recherche scientifique en programmation parallèle, on retrouve souvent la condition stipulant que la dimension du problème doit être divisible exactement par le nombre de processeurs, ce qui, selon moi, pose une limite quant à la souplesse d'un programme. Aussi, dans la "vraie vie", il est très rare d'être assez chanceux pour que cette condition se réalise dans tous les cas rencontrés.

Il existe seulement deux programmes constituant l'interface parallèle, soient "postproc" et "slave". L'idéal en programmation parallèle est d'avoir le même programme

tournant sur tous les nœuds, mais cette exigence est outrepassée ici vu le nombre impressionnant de lignes constituant chacun des programmes, et l'affichage graphique effectué par le programme maître.

La figure 6.1 montre comment l'interface graphique se place par rapport aux autres modules du projet complet, nommé "Analyse parallèle des Biochamps". La partie hachurée constitue l'interface graphique parallèle dont nous allons discuter dans ce chapitre. Idéalement, tous les modules devraient être actifs de façon concurrentielle, c'est-à-dire qu'ils devraient toujours être accessibles dans le même programme exécutable et les données de tous les modules partagées sur tous les processeurs disponibles. Mais on est loin de réaliser cette condition et le transfert des données d'un module à l'autre s'effectue par fichier interposé. Par exemple, le module de calcul basses fréquences compile ses résultats dans un fichier et l'interface graphique va lire ce fichier afin de réaliser l'affichage et le traitement des résultats.

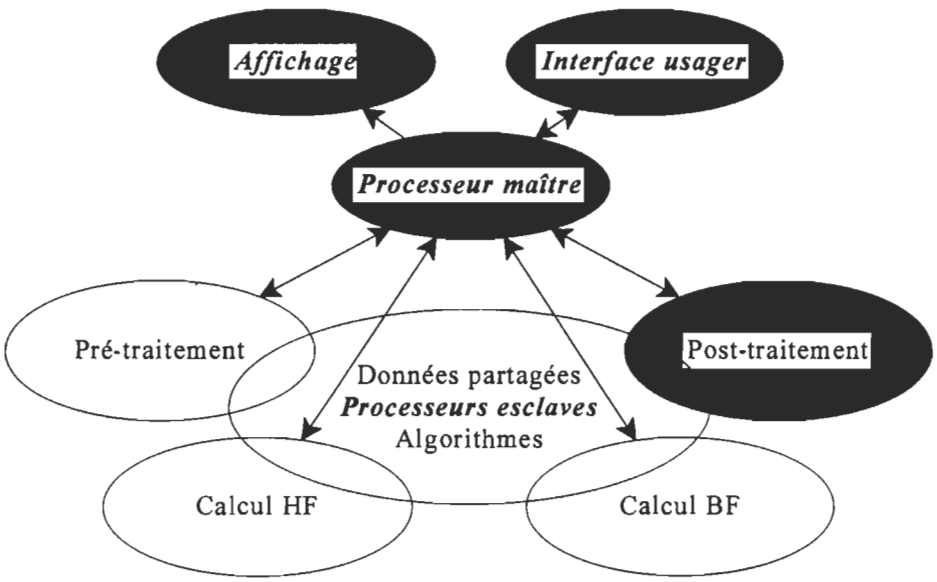

Figure 6.1: Vue globale du projet d'analyse parallèle des biochamps.

Nous allons donc voir au cours de ce chapitre comment est effectuée la séparation des tâches dans un premier temps. Cette partie est très importante puis qu'elle détermine en autres choses la distribution de la charge de travail de chacun des processeurs. Cette charge de travail se doit d'être aussi uniforme que possible pour obtenir des résultats optimums. Ensuite nous allons regarder comment les fonctions d'affichage et de rotation utilisent les processeurs parallèles afin de réaliser le cœur de l'interface graphique parallèle.

6.2 *Algorithme de séparation des tâches*

La séparation des tâches dans l'interface graphique parallèle s'effectue lors de la lecture du fichier sur le disque. Il est nécessaire de connaître le nombre de processeurs disponibles (`numprocs`) et la dimension du problème. Lors de son initialisation, le programme maître est en mesure de connaître le nombre total de processeurs par la commande:

```
numprocs = getnitb();                    // Nombre de processeurs disponibles
```

La variable `numprocs` contient donc le nombre de processeurs total sur le système qui peuvent être utilisés pour emmagasiner les données graphiques. Les données seront séparées entre les processeurs par plan *x*-*y*. L'algorithme de séparation des tâches construit deux tables de dimension égale au nombre de processeurs. La première table est la table "d'offset" qui contient, pour chacun des processeurs, le plan de départ du processeur. Ces données seront très utiles lors de la reconstruction des données quand celles-ci reviendront en provenance des processeurs esclaves pour fin d'affichage. La deuxième table contient le nombre de plans total attribué à chacun des processeurs.

```
unsigned int offset[numprocs];
unsigned int elem_par_proc[numprocs];
```

Chacun des processeurs se verra donc attribuer un certain nombre de plans en fonction du nombre de processeurs au total et de la dimension du problème à traiter. Par exemple, si on a un problème d'ordre 20 (20X20X20), et que nous avons 4 processeurs, chacun des processeurs se verra confier 5 plans de 20X20 pour fin de traitement. Lors de la séparation des tâches, le programme conserve le nombre d'éléments restant à attribuer et le nombre de processeurs restant.

```
int reste;              // Nombre d'elements restant
int reste_procs;        // Nombre de processeurs restant
```

Le nombre de plans restant est fixé au départ à la dimension du problème dans l'axe des z (`nbz`) et le nombre de processeurs restant est fixé au nombre de processeurs total.

```
reste = nbz;
reste_procs = numprocs;
```

Suite à cette initialisation, la séparation des tâches commence. Pour chacun des processeurs, le programme divise le nombre d'éléments restant par le nombre de processeurs restant, ce qui donne le nombre d'éléments pour le premier processeur. Ce même nombre d'éléments est soustrait au nombre d'éléments restant et le nombre de processeurs restant est diminué de un.

```
elem_par_proc[i]=reste/reste_procs;      // Calcul du nombre d'elements
reste -= elem_par_proc[i];               // par processeur
--reste_procs;                           //
```

Par la suite, le programme calcule les "offsets" pour ce processeur.

```
    offset[i] = 0;                       //
    for (k=0; k<i; ++k)                  // Calcul de l'offset
    offset[i] +=elem_par_proc[k];        //
```

Par exemple, prenons un problème d'ordre 30 (30x30x30) et 4 processeurs. Le premier processeur se verra attribuer 7 plans (30/4). Il reste donc 23 plans et 3 processeurs.

Le deuxième processeur aura 7 plans également (23/3) et il reste 16 plans et 2 processeurs. Les deux derniers processeurs auront donc 8 plans chacun.

Suite à ces calculs, deux messages sont envoyés à chacun des processeurs. Le premier message contient toutes les valeurs relatives aux éléments en format virgule flottante (float), les dimensions du problème (nbx, nby, nbz) et le nombre de plans pour le processeur. Le deuxième message, tout petit, contient les valeurs minimales et maximales qui serviront pour le calcul des couleurs de chacun des points, et les coordonnées du point de visualisation utiles pour le calcul de la projection.

Cette méthode de séparation des tâches comporte un désavantage au niveau de la balance de la charge (*load balancing*). En prenant le nombre de points traités par chacun des processeurs, on remarque que les deux premiers processeurs ont chacun 6300 points à calculer (30x30x7) et les deux derniers processeurs ont 7200 points (30x30x8). On est porté à se demander pourquoi séparer le problème d'une telle façon si la charge n'est pas balancée correctement ? En programmation parallèle, il faut toujours faire un compromis entre temps de calcul et temps de communication. Cette façon de séparer la tâche permet de ne pas transférer les coordonnées des points, ce qui représente un temps de communication plus élevé par rapport au temps de calcul très court.

De son côté, le processeur esclave reçoit le premier message qui contient une commande incluse dans le message. Ces commandes sont au nombre de trois:

- Lecture des données
- Affichage
- Rotation.

La commande de lecture des données est donc reçue dans le cas présent et le processeur esclave insère les valeurs reçues dans sa propre base de données graphique. Il calcule ensuite les valeurs des couleurs pour chacun des points et il appelle ensuite la fonction de calcul de la projection.

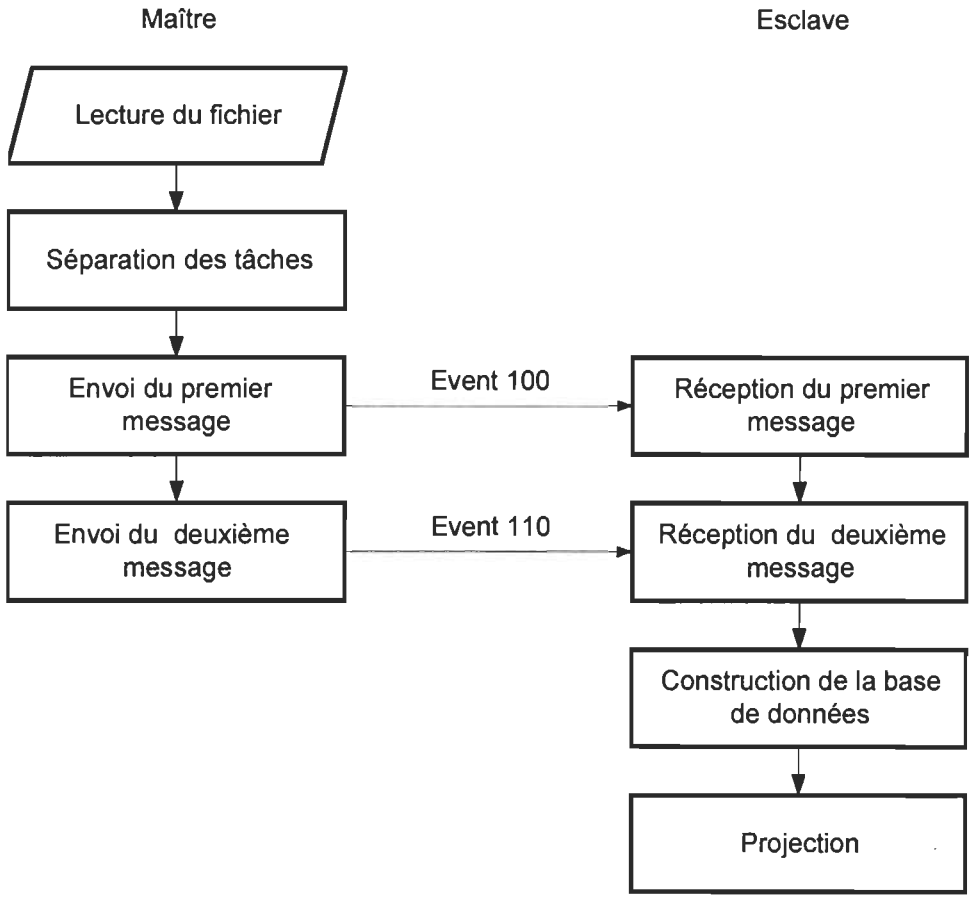

Figure 6.2: Séquence des opérations de la séparation des tâches.

6.3 *Affichage parallèle*

La technique utilisée pour effectuer l'affichage en parallèle implique que le processeur maître envoie une commande aux processeurs esclaves pour les instruire d'envoyer leurs données pour affichage. Les données nécessaires sont les coordonnées de projection dans le plan de l'écran d'affichage `ppx` et `ppy`, ainsi que la couleur à utiliser pour

afficher le point. Il s'avère intéressant d'effectuer une compression des données vu le grand nombre de points à transférer. Par exemple, prenons un problème d'ordre 100 et une topologie à 10 processeurs. Chacun des processeurs possède donc 100,000 points, ce qui fait 2 valeurs de type "`float`" pour les coordonnées `ppx` et `ppy`, et une valeur de type "`short int`" pour la couleur, ce qui fait en tout 10 octets par point à transférer. Donc chacun des processeurs devra transférer un million d'octets, à un taux de transfert de 20 Mbps, on obtient un temps de transfert de 0.4 secondes. En plus, sur les machines utilisées, il est impossible de transférer 10 messages de cette taille de façon simultanée, on doit donc effectuer un artifice qui sera expliqué plus loin. La compression se fait en regroupant les coordonnées des points par couleurs. Chacune des couleurs portant une valeur de 0 à 9 sera envoyée, si et seulement si il y a des points de cette couleur à afficher pour ce processeur.

La fonction `Req_Affichage()` se charge de communiquer avec les processeurs esclaves afin d'effectuer l'affichage. D'abord, un premier message contenant la commande d'affichage sera envoyé à tous les esclaves. Lors de la réception du message, les esclaves appellent leur fonction `Affichage()`. Cette fonction reprend à peu près les mêmes instructions que dans le cas de l'affichage séquentiel, avec une toute petite différence. Les programmes esclaves reçoivent les options d'affichage tel que:

```
mess.nh_data[0] = COMM_REQ_AFF;
mess.nh_data[1] = cache_zero;
mess.nh_data[2] = affiche_par_couleur;
mess.nh_data[3] = affiche_par_plan;
mess.nh_data[4] = hold_enable;
mess.nh_data[5] = plan_a_afficher;
mess.nh_data[6] = plan;
mess.nh_data[7] = coul;
```

Ces options permettront d'effectuer l'envoi des points qui doivent être affichés seulement. Si donc l'utilisateur a sélectionné l'affichage d'un seul plan, seuls les points de

ce plan seront transférés des processeurs esclaves au processeur maître. Un champ "`enable`" a été ajouté dans la structure des données graphiques sur les processeurs esclaves. Ce dernier inspecte les points et détermine les points qui devront être transférés en plaçant leur "enable" à ON. Par la suite, les processeurs esclaves attendent une requête du processeur maître d'envoyer leurs données. Comme il a été mentionné plus haut, les données sont envoyées par groupe de couleurs identiques. Lorsque les données d'un processeur sont reçues par le maître, elles sont affichées par la fonction du programme maître `Affiche_Rectangle_par()` qui est presque identique à la fonction `Affiche_Rectangle()`.

Lorsque les processeurs esclaves envoient leurs données au processeur maître, il est impossible qu'ils le fassent tous ensemble. C'est pourquoi le processeur maître effectue deux requêtes aux processeurs esclaves 0 et 1 la première fois, et la deuxième fois, il demande au processeur 2 d'envoyer ses données pendant qu'il lit les données du processeur 1. Les limitations de la machine ALEX AVX2 nous empêchent d'effectuer des envois de plusieurs processeurs vers le maître, surtout dans le cas où les messages sont très gros, il survient un gel complet du programme. Voici cette logique exprimée en langage C.

```
mess.nh_msg     = (char *) buf;
mess.nh_length  = sizeof (buf);
mess.nh_event   = 20;
for (i=0; i<numprocs; ++i)
{
    if (i==0)     // Si premiere fois
    {
        mess.nh_node    = 0;        // Envoi event 20 au node 0
        while (nsend (&mess) < 0 )
        {}
        ++mess.nh_node;             // Et au node 1
        while (nsend (&mess) < 0 )
        {}
    }
    else          // Si non
```

```
    {
        mess.nh_node    = i+1;      // Envoie event 20 au node i+1
        while (nsend (&mess) < 0 )
        {}
    }
}
```

Suite à ces étapes, le programme maître reçoit les données du processeur esclave par groupe de couleur identique. Lorsque le processeur esclave a terminé d'envoyer ses données, il envoie une couleur égale à 99 ce qui signifie la fin de la transmission.

```
do
{
    while (nrecv (&mess) < 0 ) {}
    p.coul = mess.nh_data[1];
    if (p.coul != 99)
    {   ctr=0;
        for (j=0; j<mess.nh_data[0]; ++j)
        {
            p.ppx = (double) recept[ctr];       ++ctr;
            p.ppy = (double) recept[ctr];       ++ctr;
            if (!Clipping (&p, &p))
            {
                Normalise (&p);
                Affiche_Rectangle_par (p);
                ++ctr2;
            }
        }
    }
            ++mess.nh_event;
}   while (mess.nh_data[1] != 99);
```

Il faut remarquer la numérotation des "events" dans le transfert des informations des processeurs esclaves vers le processeur maître. Chacun des processeurs esclaves débute sa numérotation à 1000 fois son numéro de nœud plus 1. Par exemple, le processeur 1 débutera l'envoi des données avec un numéro de 2000. À chaque envoi, ce numéro d'événement est augmenté de 1

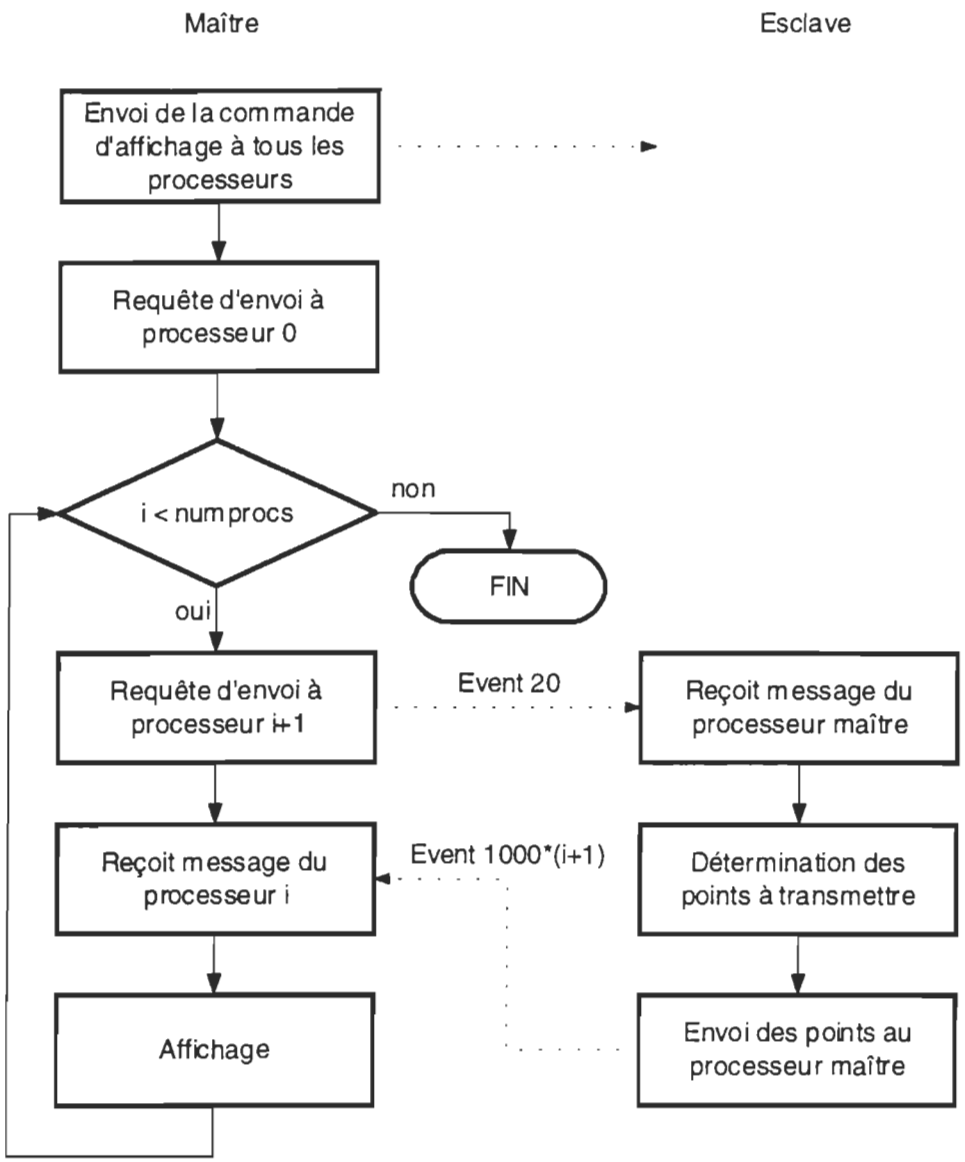

Figure 6.3: Séquence des opérations d'affichage parallèle.

6.4 *Rotation parallèle*

La rotation est effectuée en parallèle par la distribution des opérations de calcul matriciel sur tous les processeurs. En d'autres termes, la fonction de rotation en séquentiel a été implantée sur les processeurs esclaves. Lorsque le processeur maître envoie une requête de rotation aux processeurs esclaves, chacun des esclaves calcule la rotation, puis la

projection pour ses points. Tout comme dans le cas de la rotation en séquentiel, les processeurs esclaves doivent connaître l'axe de rotation ainsi que l'angle de rotation. Ces données sont transmises aux processeurs esclaves.

```
mess.nh_data[0] = COMM_ROTATION;
mess.nh_data[1] = (int)axe;
mess.nh_data[2] = angle;
```

Suite à la rotation, le programme maître effectue une requête d'affichage. Les mêmes étapes que nous avons vues dans la section 6.3 sont alors exécutées. Notons que pour effectuer la rotation, le seul message qui est transféré aux processeurs esclaves est celui décrit ci-haut. La rotation n'implique donc pas de transfert de gros messages. Seule la fonction de rotation transmettra de gros messages suite au calcul de la rotation.

6.5 Programmation MPI

Le standard MPI se doit d'être respecté dans la programmation parallèle. L'utilisation des ordinateurs AVX2 ayant été privilégiée pendant l'élaboration du projet, il a été impossible avant il y a quelques mois de s'adonner à la programmation MPI. L'expérience et la recherche de fiabilité nous ont amenés à découvrir et utiliser MPI. Tel que mentionné dans la section 3.3, les avantages de MPI sont la portabilité, la flexibilité, l'efficacité et la fiabilité. La portabilité permet de rouler notre application sur un réseau de stations de travail aussi bien que sur les ordinateurs AVX3, sans modifications majeures au code. Mais le plus important de ces avantages est la fiabilité. Les erreurs de communication expérimentées lors des essais sur AVX2 avec Trollius ont été complètement éliminées lors des essais avec MPI, ce qui prouve que les erreurs provenaient bel et bien de Trollius et non d'une erreur de programmation.

De façon bien évidente, la programmation MPI est beaucoup plus simple à réaliser, résultant en un nombre de lignes plus faible pour effectuer le passage d'un message, allégeant ainsi le code. L'exemple suivant montre comment le transfert d'un message s'effectue avec Trollius. Il faut d'abord affecter les valeurs de la structure "nmsg" afin d'envoyer le message, et ensuite effectuer l'envoi proprement dit à l'aide de la commande "nsend".

```
mess.nh_event   = 12;
mess.nh_type    = 0;
mess.nh_flags   = DFLTMSG;
mess.nh_length  = sizeof ( buffer );
mess.nh_msg     = (char *) buffer;
mess.nh_node    = 1;
mess.nh_data[0] = size;

nsend (&mess)
```

Cette façon de faire consomme plusieurs lignes de programmation. Trollius possède le désavantage de ne pas supporter les tampons (*buffers*) de transmission alloués dynamiquement, ce qui est utilisé largement dans l'interface graphique parallèle, et ce que supporte MPI. L'allocation dynamique des tampons de transmission permet de consommer moins de mémoire sur chaque nœud et elle permet aussi d'envoyer le nombre exact d'octets entre les processeurs, optimisant ainsi le temps de communication. Aussi, nous avons expérimenté quelques problèmes avec le transfert des valeurs de type "double" dans Trollius, problème qui nous n'avons pas rencontré avec MPI.

Une seule ligne de programmation est nécessaire pour envoyer un message avec MPI.

```
MPI_Send (buffer, size, MPI_DOUBLE, DEST, TAG, MPI_COMM_WORLD);
```

On ne regardera pas en détails ici tout le code de version MPI. La même stratégie de programmation a été utilisée dans les deux versions, soient Trollius et MPI. Seules les lignes effectuant le passage des messages ont été modifiées afin de les adapter au standard MPI.

Pour terminer, il faut aussi savoir que MPI possède des capacités de "déverminage" (*debugging*), avec l'outil "XMPI" disponible sur le site Internet de LAM/MPI [15]. Cet outil permet de suivre à la trace tous les messages envoyés et reçus par chacun des processeurs, leur taille, leur identification, et ainsi de suite. Cette facilité ne fut pas disponible avec Trollius sur les ordinateurs AVX2.

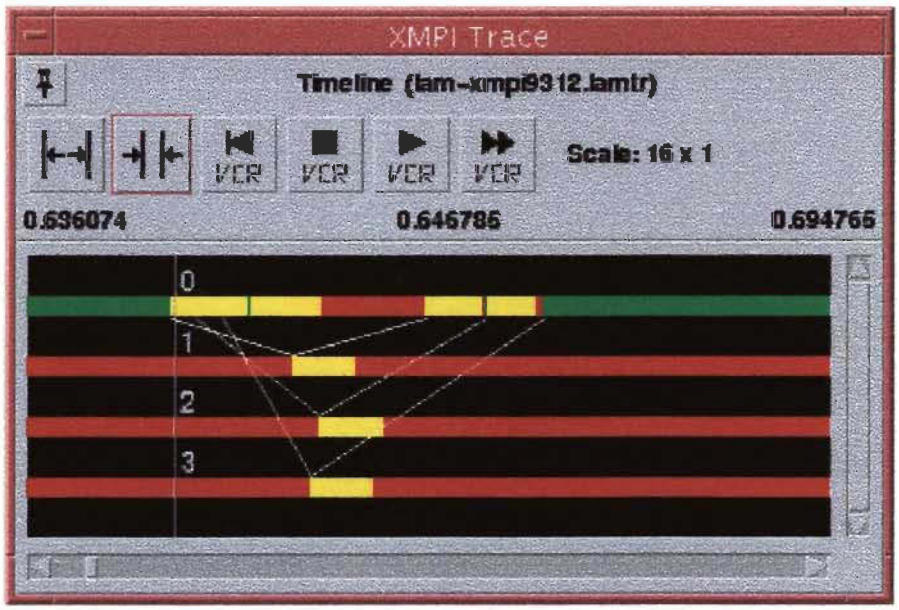

Figure 6.4: Écran de XMPI permettant de tracer les messages MPI.

7 Essais et résultats

7.1 Introduction

Le présent chapitre se consacre entièrement aux résultats obtenus lors des essais. On verra d'abord la version séquentielle du programme et les résultats de temps d'affichage et de rotation. On a choisi de prendre comme base de référence pour le calcul séquentiel l'utilisation d'une machine Sparc 20. Pourquoi ? Parce que cela représente vraiment l'utilisation séquentielle d'un logiciel. En d'autres termes, c'est ça la "vraie vie". Jamais quelqu'un ne programmera un logiciel parallèle pour le faire rouler sur un seul nœud ! De toute façon, le processeur SPARC et le processeur i860 possèdent des performances à peu près équivalentes, tous les deux étant de technologique RISC et fonctionnant à une fréquence de 40 MHz.

Nous verrons par la suite les résultats des essais sur les ordinateurs AVX2 disponibles au laboratoire de recherche sur les biochamps, et finalement, nous verrons les résultats de calcul sur AVX3, machine qui est gracieusement mise à notre disposition par M. Marek Zaremba de l'Université du Québec à Hull. Finalement, on verra quels sont les résultats obtenus avec la programmation MPI conjointement avec l'utilisation de la combinaison LAM/SPARC 20 et des machines AVX3. Bien évidemment, tous ces résultats seront ensuite compilés, interprétés et comparés dans la section 7.6.

Lorsque nous parlerons de dimension de problème de N, notons qu'il s'agira d'essai effectuée avec N^3 points à calculer, soit un cube de NxNxN. Afin d'effectuer les essais tant séquentiels que parallèles, des données de test ont dû être construites. Des fichiers ont été

bâtis par un programme C afin de simuler des résultats en provenance des modules de calcul hautes et basses fréquences.

Les calculs de temps ont été réalisés par deux fonctions nommées `start_time()` et `prn_time()`. Ces fonctions retournent deux valeurs soient le "User time" et le "Real time". Le dernier ne sera pas considéré car la précision n'est que de 1 seconde et ce temps est influencé par la charge totale du système. On considérera seulement de premier temps donné, soit le "User time" qui lui est précis au centième de seconde, ce qui s'avérera très utile pour les temps plus courts. Pour chacun des essais, l'emplacement des fonctions sera précisé de façon à avoir un aperçu des opérations incluses dans les temps mentionnés.

7.2 Essais en séquentiel

Comme cité précédemment, l'utilisation d'une station de travail SPARC 20 a été faite afin de réaliser les essais en séquentiel. Bien que ces stations de travail soient quelque peu âgées, il n'en reste pas moins qu'elles sont assez performantes. Mais le calcul graphique demandant énormément d'opérations, il atteint rapidement leur limite, comme on le verra pour des dimensions de problèmes assez élevées. Par exemple, les dimensions utilisées dans les essais sur SPARC 20 atteignent un maximum de 100, contrairement à 128 pour les essais à quatre processeurs et plus. En effet, pour une dimension de 128, la limite de la mémoire de 64 Méga-octets est atteinte.

Notons que tous les temps mentionnés dans les résultats sont la moyenne de plus d'une dizaine d'essais effectués. Les résultats ont été obtenus suite à une compilation des temps écrits dans des fichiers par les programmes maître et esclaves.

Tableau 7.1: Temps de rotation et d'affichage en séquentiel sur Sparc 20.

Dimension	Temps de rotation (s)	Temps d'affichage(s)
8	0,01	0,01
16	0,03	0,06
32	0,19	0,54
64	1,56	4,41
100	26,30	33,34

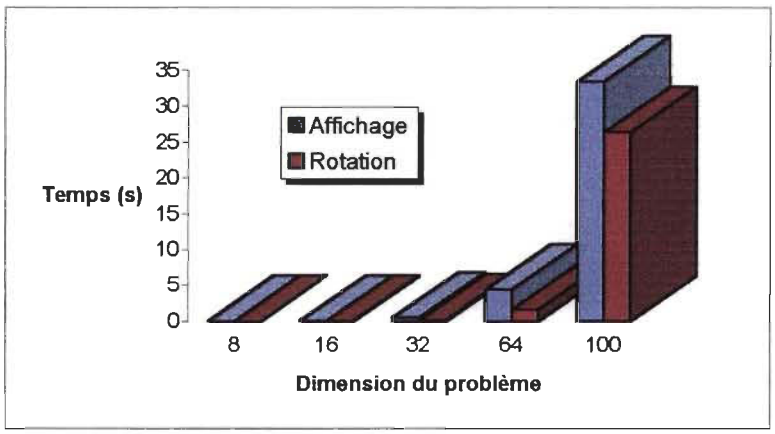

Figure 7.1: Temps d'affichage en séquentiel sur Sparc 20.

On remarque donc des temps relativement courts pour des problèmes de petite taille et une augmentation exponentielle par la suite. En théorie, on devrait avoir une relation $O(N^3)$. En parallèle, on devrait s'attendre à voir cette courbe descendre pour les problèmes de grande dimension. Pour les problèmes de petite dimension, il se pourrait que le programme parallèle soit plus lent que le programme séquentiel car les communications entre les processeurs impliquent un temps supplémentaire aux calculs eux-mêmes.

En séquentiel, l'emplacement des fonctions de calcul du temps est relativement simple. Pour la rotation, le calcul du temps commence au tout début de la fonction

`Rotation()` et se termine à la toute fin de cette fonction, juste avant l'appel à la fonction d'affichage.

```
else        // Sequentiel
{
    printf ("Temps de rotation sequentiel\n");
    if (timer) start_time();
    .
    .
    .
    Rotation();
    Transformation ( total);
    Projection();
    if (timer) prn_time();
    Affiche_Articul (&a);
}
```

Dans le cas de l'affichage, le temps se calcule dans la fonction `Affiche_Articul()`. Il démarre juste avant la boucle qui effectue l'affichage pour chacune des figures de l'articulation et se termine juste avant l'affichage de la légende, opération qui n'est pas incluse dans les temps d'affichage.

```
printf ("Temps d'affichage sequentiel\n");
if (timer) start_time();
for (i=0; i< p->numfig; ++i)
    Affiche_Figure (&(p->fig[i]));

if (timer) prn_time();
Legende();
```

7.3 Essais en parallèle sur AVX-2

Les essais sur la machine AVX2 se sont faits sur 2, 4, 8, 16, 32 et 64 processeurs. La dimension des problèmes utilisée pour les essais a été de 8, 16, 32, 64 et 128. Certaines de ces combinaisons n'ont pas pu être réalisées. Par exemple, dans le cas de deux processeurs, la limite de mémoire est atteinte pour les dimensions de problème de 64 et 128. Aussi, dans le cas de 64 processeurs seule la dimension de problème de 64 a été effectuée. On peut résumer par le tableau 7.2 les essais effectués.

Tableau 7.2: Essais réalisés sur AVX2.

	Nombre de processeurs					
	2	4	8	16	32	64
Dimension du problème 8						
16						
32						
64						
128						

Dans le cas de l'affichage, le calcul du temps débute lors de l'appel de la fonction `Req_Affichage()` par le processeur maître et se termine juste avant l'affichage de la légende. Il inclut donc tous les transferts d'information entre le processeur maître et les processeurs esclaves tel que décrit dans la section 6.3 et résumé sur la figure 6.3.

```
XClearWindow (dpy, w);
printf ("Affichage\n");
if (timer) start_time();
.
.
.
if (timer) prn_time();
Legende();
```

Dans le cas de la rotation maintenant, il faut se souvenir qu'aucun transfert de données n'est nécessaire pour le calcul de la rotation. Seul un message de commande est envoyé aux processeurs esclaves pour les instruire d'effectuer le calcul de la rotation, suite à quoi le processeur maître effectue une requête d'affichage. Ces étapes sont décrites en 6.4.

```
printf ("Rotation\n");
    if (timer) start_time();
    .
    .
```

```
    if (timer) prn_time();
    Req_Affichage ();
}
```

Les résultats de rotation parallèle présentés par le processeur maître se résument donc à l'envoi d'un message de commande à chacun des processeurs esclaves, ce qui n'est pas très significatif. Vu par le processeur maître, le temps de rotation augmentera avec le nombre de processeurs (figure 7.2), mais du point de vue des processeurs esclaves, le temps de rotation de chacun des processeurs esclaves devrait diminuer avec l'augmentation du nombre total de processeurs (figure 7.3). Pour un nombre de processeurs fixe, le temps de rotation de chacun des processeurs esclaves devrait augmenter en fonction de la dimension du problème à traiter (figure 7.4).

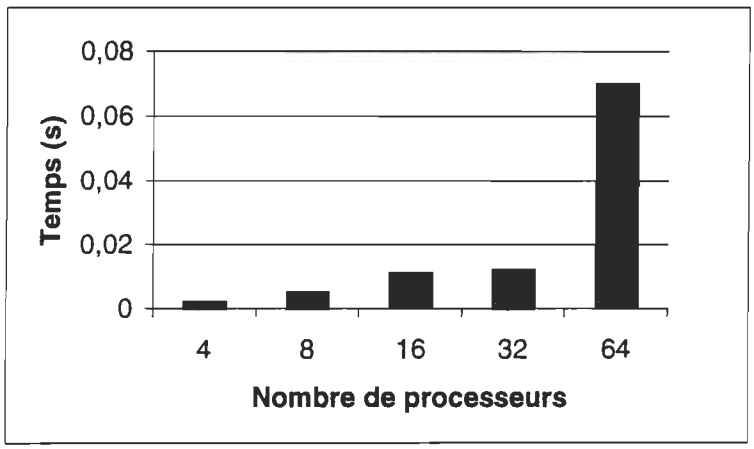

Figure 7.2: Temps de rotation sur le maître en fonction du nombre de processeurs.

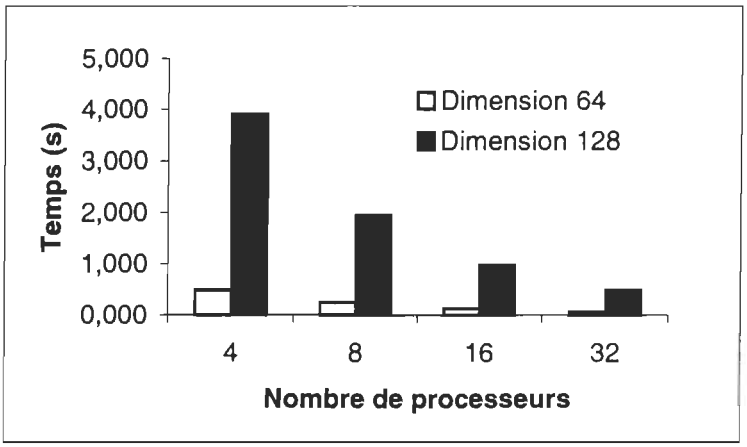

Figure 7.3: Temps de rotation des esclaves en fonction du nombre de processeurs.

Il est donc lieu de prendre pour acquis que le temps de rotation le plus élevé sera le temps de rotation effectif du programme. Le processeur le plus lent d'entre tous déterminera la vitesse du système total, ce qui est finalement très logique.

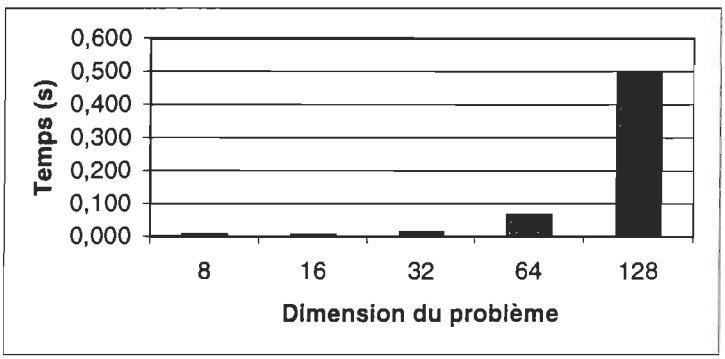

Figure 7.4: Temps de rotation des esclaves sur 32 processeurs.

Le tableau 7.3 résume tous les essais effectués sur les ordinateurs AVX2. Il faut se rappeler que ces temps sont des temps "*User*" et ils ne représentent pas le temps réel nécessaire pour effectuer l'opération. Comme mentionné auparavant, le temps utilisé convient mieux à notre cas vu la précision nécessaire pour des petites dimensions de problème. Les temps dans le tableau 7.3 servent pour fin de comparaison seulement entre les algorithmes séquentiel et parallèle.

Tableau 7.3: Compilation des résultats sur AVX2.

	Affichage Dimension du problème					Temps de rotation moyen par esclave Dimension du problème				
Procs.	8	16	32	64	128	8	16	32	64	128
2	0,015	0,074	0,551	N/D	N/D	0,009	0,021	0,128	N/D	N/D
4	0,016	0,071	0,538	4,244	34,946	0,006	0,013	0,066	0,493	3,906
8	0,025	0,079	0,506	3,947	32,320	0,006	0,009	0,036	0,249	1,956
16	0,042	0,095	0,559	4,022	29,135	0,005	0,007	0,020	0,127	0,980
32	0,069	0,128	0,568	4,017	29,456	0,005	0,006	0,013	0,066	0,493
64	N/D	N/D	N/D	4,515	N/D	N/D	N/D	N/D	0,036	N/D

N/D = Non Disponible

Il est important de remarquer que le nombre de processeurs optimal n'est pas nécessairement égal au nombre de processeurs maximal. En examinant les chiffres du tableau 7.3 du côté "affichage", soit sur la gauche, on remarque d'abord, pour une dimension de "8" que le temps augmente en fonction du nombre de processeurs (0.015, 0.016, 0.025...). Il faut donc conclure que la mise en parallèle est inefficace pour une telle dimension de problème. Pour la dimension de "16", on obtient un minimum à 4 processeurs, et pour les dimensions de "32" et de "64", on observe une diminution du temps d'affichage jusqu'à une valeur de 8 processeurs, après quoi les temps commencent à augmenter. Le phénomène se produit aussi pour la dimension de 128 mais le nombre de processeurs est plus élevé. On note donc que le nombre de processeurs optimal se situe environ à 8 pour les dimensions de "32" et "64", et pour la dimension du problème de 128, le nombre de processeurs optimal serait plutôt autour de 16. Ceci fait donc ressortir le fait que pour de petits problèmes, la mise en parallèle ne présente aucun avantage, et que pour

les problèmes de taille plus élevée, le nombre de processeurs optimal peut varier avec celle-ci. La figure 7.5 met en lumière le nombre de processeur optimal pour un affichage d'un problème d'ordre 32.

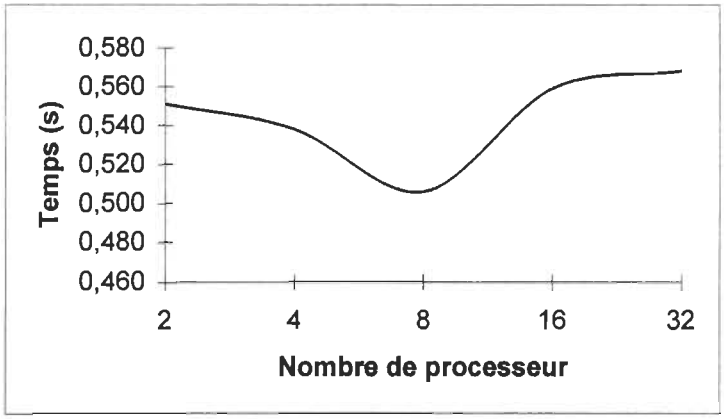

Figure 7.5: Temps d'affichage sur AVX2 pour une dimension de 32.

On peut finalement évaluer le gain en vitesse (*Speedup*) et l'efficacité pour les essais effectués. Ceux-ci sont résumés dans le tableau suivant pour une dimension de problème de 64.

Tableau 7.4: Gain en vitesse et efficacité pour une dimension de 64 sur AVX2.

	Affichage			Rotation		
Nombre de processeurs	Temps	Gain en vitesse[1]	Efficacité	Temps	Gain en vitesse[2]	Efficacité
4	4,244	1.039	0.260	0,493	3,164	0,791
8	3,947	1.117	0.140	0,249	6,265	0,783
16	4,022	1.096	0.069	0,127	12,283	0,768
32	4,017	1.098	0.034	0,066	23,636	0,739
64	4,515	0.977	0.015	0,036	43,333	0,677

Note 1: Le gain en vitesse de l'affichage est calculé avec un temps séquentiel de 4.41.
Note 2: Le gain en vitesse de la rotation est calculé avec un temps séquentiel de 1.56.

Un système parallèle idéal contenant N processeurs devrait avoir un gain en vitesse de N, mais en pratique cela est impossible parce que les processeurs ne consacrent pas tout leur temps au calcul. L'algorithme nécessite beaucoup de communication et l'efficacité (gain en vitesse/N) est faible.

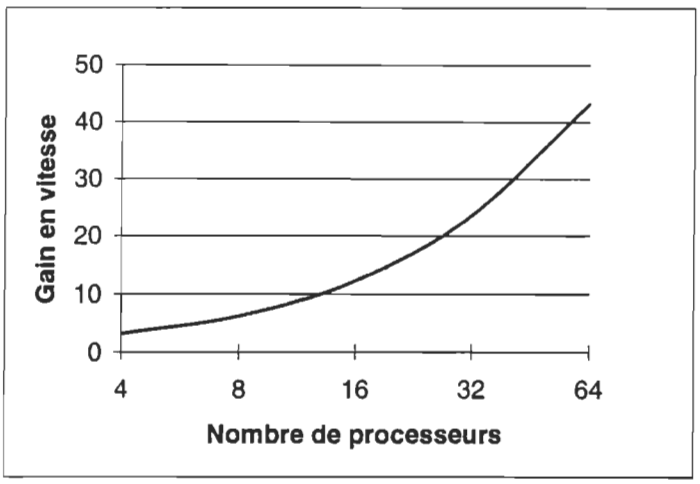

Figure 7.6: Gain en vitesse en fonction du nombre de processeurs pour une dimension de 64

Finalement, pour ce qui est des temps de rotation, on note qu'ils diminuent tous en fonction du nombre de processeurs car le calcul du temps de rotation n'implique aucune communication. Les temps de rotation représentés au tableau 7.3 sont la moyenne des temps de chacun des esclaves ensemble. Il est important de vérifier, lors des essais, comment se distribue la charge de travail sur chacun des processeurs. Dans le meilleur des cas, le temps de calcul de chacun des processeurs sera identique, mais il arrive que les temps de calcul soient très différents. La figure 7.7 montre le pire cas de balance de charge rencontré lors des essais.

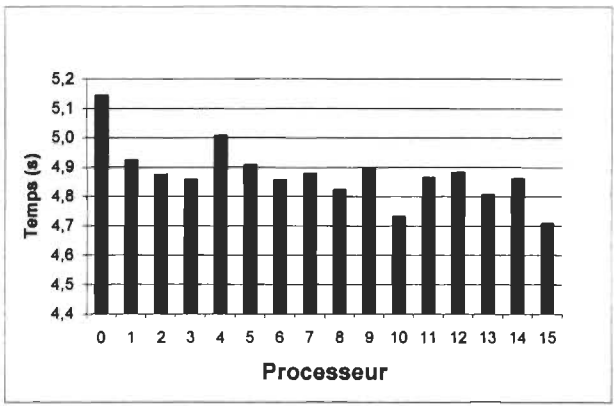

Figure 7.7: Répartition de la charge sur 16 processeurs pour un problème d'ordre 128.

7.4 Essais en parallèle sur AVX-3

Le principal avantage des ordinateurs AVX3 est la vitesse plus élevée de l'horloge des processeurs et la bande passante plus large au niveau du réseau de transmission entre les processeurs (100 Mbps). Les essais présentés dans cette section sont moins nombreux que dans la section précédente portant sur les ordinateurs AVX2 et ont été faits dans le but d'évaluer l'impact d'un système plus performant sur l'algorithme parallèle écrit au cours de ce projet. De plus, le nombre de processeurs disponibles était de 14 lors des essais sur un maximum de 16. Les essais furent donc effectués sur 1, 2, 4, 8 et 12 processeurs afin de comparer les temps avec ceux des essais effectués sur les ordinateurs AVX2. Mais dans le cas séquentiel, on ne peut pas comparer un processeur Sparc 20 à 40 MHz et un PowerPC 604 à 133 MHz. C'est pourquoi on retrouvera dans le tableau des résultats des temps pour un seul processeur auxquels seront comparés les calculs parallèles.

Tableau 7.5: Temps d'affichage et de rotation sur AVX3.

	Affichage Dimension du problème					Temps de rotation Dimension du problème				
Nombre de processeurs	8	16	32	64	128	8	16	32	64	128
1	0,003	0,013	0,127	1,188	N/D	0,001	0,011	0,105	0,856	N/D
2	0,011	0,041	0,305	2,414	N/D	0,000	0,005	0,051	0,350	N/D
4	0,014	0,054	0,298	2,334	N/D	0,001	0,005	0,025	0,175	N/D
8	0,027	0,073	0,316	2,335	N/D	0,000	0,002	0,009	0,092	N/D
12	N/D	0,069	0,310	2,633	18,078	0,000	0,001	0,007	0,061	0,430

N/D = Non Disponible

On remarque donc, pour les cas de l'affichage de dimension de 8, 16, 32 et 64, une augmentation du temps en fonction du nombre de processeurs. D'ailleurs, tous les temps des calculs effectués en parallèles sont plus élevés que ceux effectués en séquentiel. Dans le cas de la rotation, on remarque une efficacité beaucoup plus grande. Il ne faut pas oublier que l'affichage implique beaucoup de communication et que la rotation n'implique aucune communication. C'est là la raison pour laquelle l'efficacité est si différente entre la rotation et l'affichage.

Autre remarque: il s'agit ici du même programme qui est exécuté. On ne roule pas une version MPI sur AVX3 et une version Trollius sur AVX2. La version qui est exécutée sur les deux machines est la version Trollius du programme, avec certaines modifications mineures apportées sur AVX3 en raison de la présence de LAM. Remarquons de plus qu'il n'est pas possible de déterminer un nombre de processeur optimal sur AVX3, ou plutôt devrait-on dire que le nombre de processeur optimal est 1!

Tableau 7.6: Gain en vitesse et efficacité pour une dimension de 64 sur AVX3.

	Affichage			Rotation		
Nombre de processeurs	Temps	Gain en vitesse	Efficacité	Temps	Gain en vitesse	Efficacité
2	2.414	0.492	0.246	0.350	2.446	1.223
4	2,334	0.509	0.127	0,175	4.891	1.223
8	2,335	0.509	0.064	0,092	9.304	1.163
12	2.633	0.451	0.038	0.061	14.033	1.169

On remarque encore une fois une inefficacité au niveau de l'affichage. Trop de communication est en jeu dans l'affichage des résultats et les processeurs passent le plus clair de leur temps à communiquer ou à attendre. On le voit bien par l'efficacité qui est une représentation de la fraction du temps où le processeur est utilisé de façon efficace. Pour ce qui est de la rotation, pourquoi avons-nous un gain en vitesse plus grand que le nombre de processeurs ? Cela est illogique. Voici quelques hypothèse qui pourraient apporter une partie de la réponse à cette question:

- Le temps de rotation parallèle qui apparaît dans le tableau 7.6 est le temps de rotation du processeur 0, soit celui de l'hôte. Les autres processeurs ne peuvent pas effectuer d'affichage sur la console du processeur hôte.
- Le temps de rotation pour le programme séquentiel étant calculé par la même fonction que le temps du programme parallèle, il peut y avoir une différence au niveau du temps par le fait que le programme parallèle est exécuté à travers LAM, tandis que le programme séquentiel est exécuté directement à la ligne de commande UNIX.

7.5 Essais en MPI

La quantité de travail nécessaire pour convertir le programme de Trollius à MPI vaut largement l'effort. Quelques 4 ou 5 jours furent nécessaires pour le convertir et le déboguer. Rappelons-nous les quatre avantages de MPI, soient la portabilité, l'efficacité, la fiabilité et la flexibilité. De celles-ci, l'efficacité et la fiabilité sont grandement démontrés dans les essais effectués. Les essais ont été réalisés sur un réseau de stations de travail SUN SPARC 20 et sur l'ordinateur AVX3 de l'université du Québec à Hull. Les essais en MPI seront difficilement comparables avec les essais effectués avec Trollius puisque la fonction utilisée pour calculer les temps n'est pas la même. Par contre, on peut dire tout de suite que la fiabilité et l'efficacité de la programmation MPI sont sans contredit des avantages marqués si on compare avec la fiabilité et l'efficacité de Trollius.

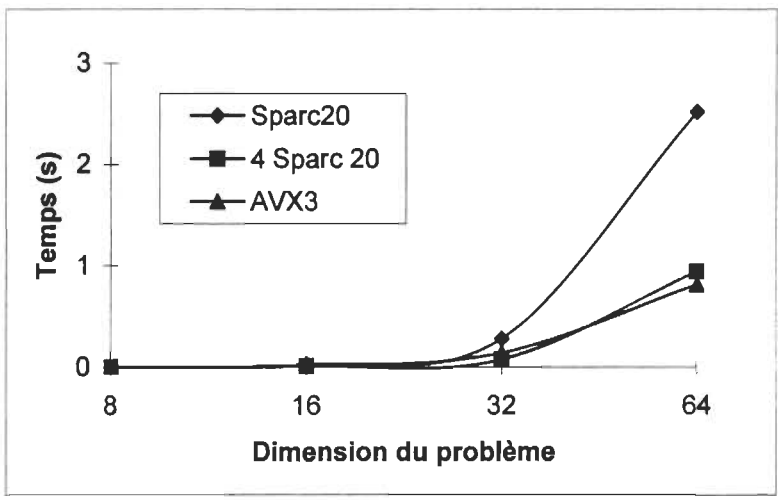

Figure 7.8: Temps de rotation en MPI.

Sur la figure 7.8, on observe que le temps de rotation séquentiel est plus court sur AVX3 que sur SPARC, ce qui est normal vu la grande différence entre la puissance des processeurs. On remarque aussi que le temps de rotation du réseau de 4 SPARC est à peu

près équivalent à celui d'un seul nœud de la machine AVX3. On peut donc dire que l'opération de rotation est efficace dans ce cas, comme le montre le tableau suivant.

Tableau 7.7: Gain en vitesse et efficacité pour une dimension de 64 sur SPARC 20 en MPI.

	Affichage			Rotation		
Nombre de processeurs	Temps	Gain en vitesse	Efficacité	Temps	Gain en vitesse	Efficacité
1	21,811			2,520		
4	9,884	2,207	0,552	0,942	2,675	0,669

Malheureusement, le nombre de processeurs étant limité à quatre, on ne peut pas expérimenter pour un nombre de processeurs plus élevé. Pour ce qui est de l'affichage, on observe aussi un bon gain en vitesse et une efficacité moyenne. Dans la version MPI du programme, une petite option intéressante a été ajoutée, soit de calculer le temps de communication et le temps d'affichage proprement dit. Cela permet de mettre en évidence la quantité importante du temps que le programme passe en communication, ou en attente d'une communication. On sait que lors d'une réception, le receveur doit attendre que son message soit arrivé avant que le programme puisse continuer plus loin. Cette version de l'interface graphique affiche donc directement sous la légende le temps moyen pour effectuer l'affichage (Tt), le temps moyen de communication (Tc), le temps moyen actif ou consacré au calcul (Ta) et finalement le temps moyen pour effectuer la rotation (Tr). La figure 7.9 montre un exemple de l'affichage avec cette version de l'interface graphique. En plus de l'affichage direct, les résultats sont compilés dans un fichier pour effectuer une analyse plus détaillée.

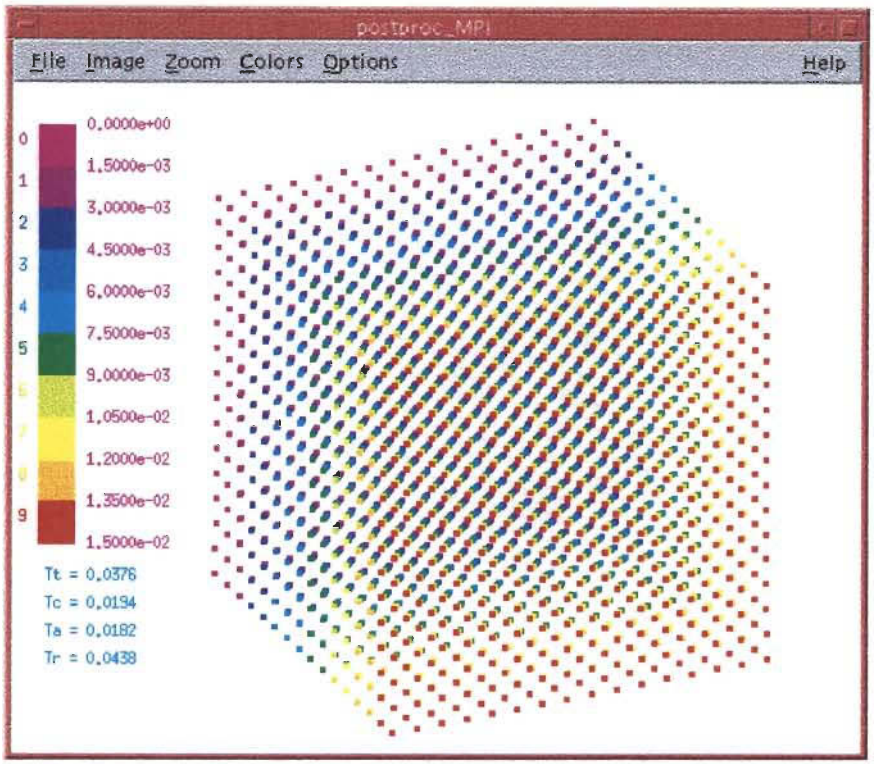

Figure 7.9: Exemple d'affichage des temps sur l'écran.

Cette option nous permet donc de statuer au niveau du fameux temps de communication qui peut littéralement "tuer" une application parallèle. La figure 7.10 montre donc un rapport Tc/Tt minimal pour une dimension de 16 sur un réseau de 4 stations SPARC 20. D'ailleurs, on calcule un gain en vitesse de 8.34 pour cette dimension de 16, ce qui est plus élevé que le nombre de processeurs !

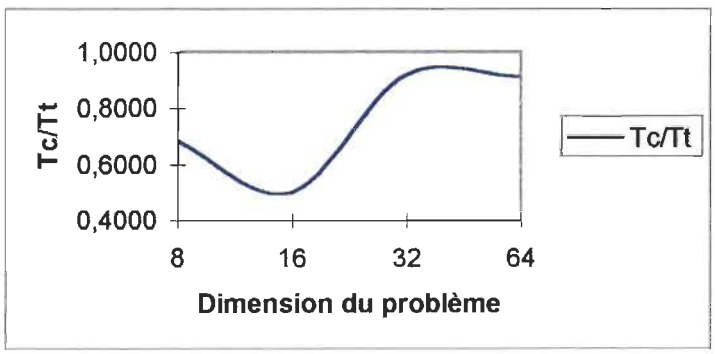

Figure 7.10: Temps de communication sur le temps total sur 4 processeurs SPARC 20.

Pour ce qui est maintenant de cas de l'ordinateur AVX3 en MPI, les tests ont été effectués sur 1, 4, 8 et 12 processeurs et les résultats sont résumés au tableau 7.8.

Tableau 7.8: Gain en vitesse et efficacité pour une dimension de 32 sur AVX3 en MPI.

	Affichage			Rotation		
Nombre de processeurs	Temps	Gain en vitesse	Efficacité	Temps	Gain en vitesse	Efficacité
1	3,153			0,142		
2	0,153	20,608	10,304	0,029	4.897	2.448
4	0,097	32,607	8,152	0,030	4,791	1,198
8	3,594	0,877	0,110	0,029	4,941	0,618
12	5,845	0,539	0,045	0,027	5,194	0,433

On observe donc une efficacité très élevée pour 2 et 4 processeurs tandis que pour 8 et 12 processeurs, l'efficacité décroît rapidement. Cela est dû au fait que la machine AVX3 de Alex possède un réseau de communication inter-processeurs non commuté. Ce fait peut être vérifié par les chercheurs travaillant sur le même type de machine qui ont obtenus les mêmes résultats.

On s'est intéressé aussi, pour montrer que le temps de communication augmente en fonction du nombre de processeurs, au rapport temps de communication sur temps total, pour le cas de l'affichage. Pour les cas de dimension de 16 et 32, on remarque que le temps de communication augmente avec le nombre de processeurs. À la suite de l'observation de la figure 7.10 et du tableau 7.9, il convient de dire que le cas de 4 processeurs est celui qui présente les meilleures performances.

Tableau 7.9: Rapport du temps de communication sur le temps total d'affichage en MPI.

Nombre de processeurs	Dimension du problème			
	8	16	32	64
2	0,6250	0,4009	0,3185	0,9381
4	0,6136	0,5324	0,4829	N/D
8	0,4717	0,5313	0,9245	N/D
12	N/D	0,9588	0,9962	N/D

On remarque donc que le rapport Tc/Tt augmente en fonction du nombre de processeurs ce qui a pour effet de ralentir énormément l'application. Par exemple, le problème de dimension 64 qui tourne très bien sur 4 stations SPARC ne tourne absolument pas sur 4 processeurs AVX3, ce qui permet de douter grandement de la performance de ces machines lorsqu'un grand nombre de processeurs est impliqué. Plusieurs personnes ont d'ailleurs confirmé cette hypothèse au fil des mois de recherche. Le problème de dimension 64 tourne péniblement sur 2 processeurs AVX3 avec un rapport Tc/Tt de 0.94!

7.6 Compilation et interprétation des résultats

Nous allons faire l'interprétation des résultats en deux parties, soient premièrement la rotation et ensuite l'affichage. Pour la rotation, comme il n'y a aucune communication impliquée, les temps vont en diminuant de façon considérable en fonction du nombre de processeurs. Aussi il existe une bonne différence entre AVX2 et AVX3 du fait que les processeurs impliqués dans le dernier cas sont plus puissants. La figure 7.8 représente tous les résultats compilés sur AVX2 et sur sa version séquentielle, soit le processeur Sparc 20.

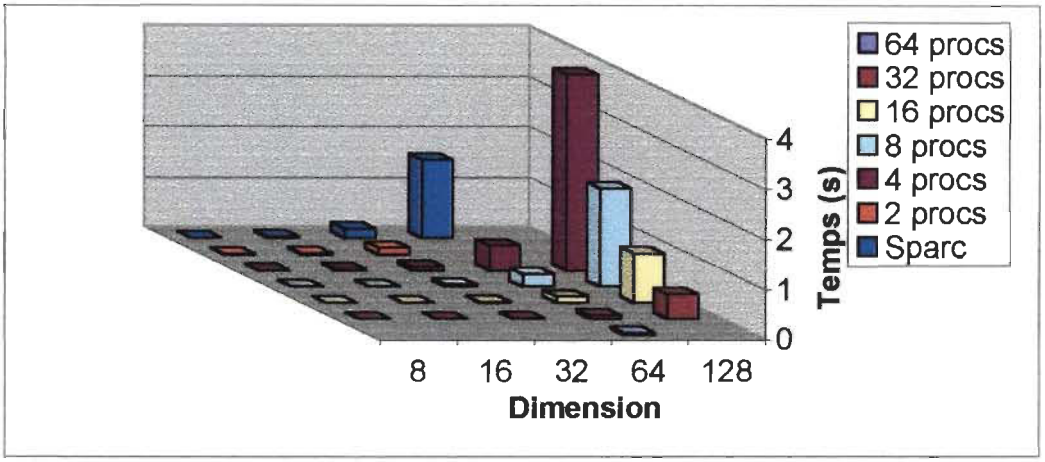

Figure 7.11: Temps de rotation sur AVX2 et Sparc.

On note donc que tout est normal du côté de la rotation et les résultats sont concordants avec ce à quoi on doit s'attendre, soit une diminution du temps de rotation en fonction du nombre de processeurs et en fonction de la diminution de la taille du problème.

Le même phénomène se passe sur AVX3, tel que représenté sur la figure 7.9. Les temps sont plus longs sur 1 processeur et ils diminuent en fonction du nombre de processeurs. On peut donc dire que pour ce qui est du mouvement des images graphiques, en particulier dans le cas de la rotation, l'utilisation de l'ordinateur parallèle est très intéressante côté du temps de calcul et elle apporte une nette amélioration. La façon dont est conçue l'interface graphique et le fait que l'opération de rotation n'implique pas de transfert d'information fait en sorte que l'efficacité du système est quasi optimale.

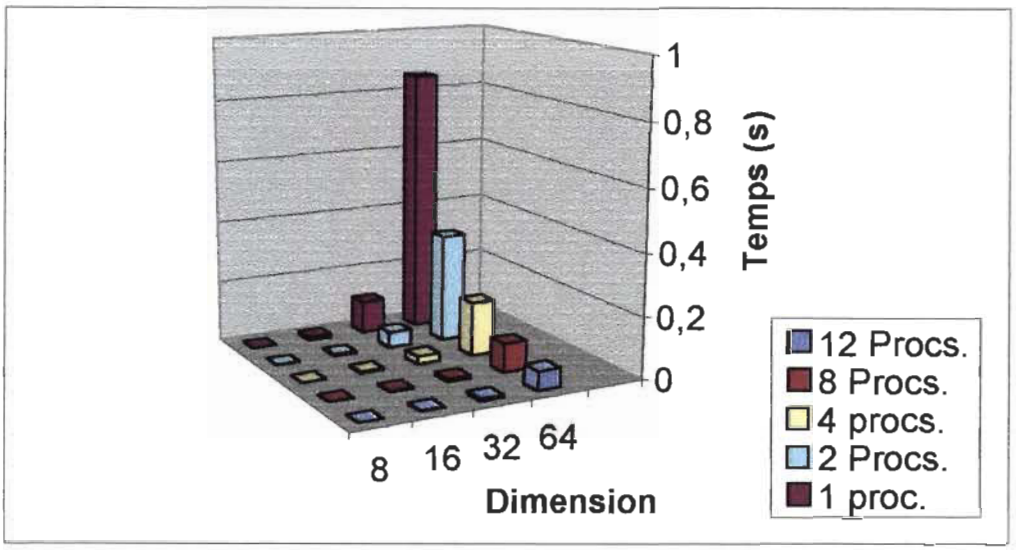

Figure 7.12: Temps de rotation sur AVX3

Finalement, que dire par rapport aux machines elles-mêmes. Il existe une différence marquée entre les deux types de machines utilisées, soient l'AVX2 et l'AVX3, différence d'environ 70% en terme de vitesse de rotation pure. La figure 7.10 montre cette différence pour les dimensions de problème de 32 et pour 2, 4 et 8 processeurs.

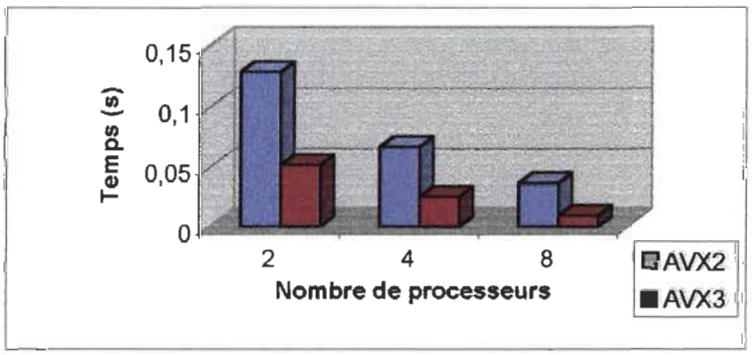

Figure 7.13: Comparaison entre AVX2 et AVX3 pour une dimension de 32.

Passons maintenant à la deuxième partie de notre compilation et interprétation des résultats, soit la partie de l'affichage. C'est celle qui est la plus difficile à expliquer car c'est aussi celle qui amène les moins bons résultats.

Ce que l'on remarque c'est que le temps d'affichage diminue bien évidemment avec la dimension du problème car la quantité d'information à transmettre entre les processeurs diminue elle aussi, mais de plus, le temps d'affichage ne diminue pas nécessairement en fonction du nombre de processeurs. Ceci est simplement dû au fait que l'affichage est constitué presque exclusivement de communication sur le réseau entre les processeurs. La figure 7.14 étale tous le résultats sur l'ordinateur parallèle AVX2. Notez particulièrement la constance des temps pour le cas de la dimension de 64. Le lecteur peut se reporter au tableau 7.3 afin de connaître les temps exacts. Pour la dimension de 128, on note une légère amélioration et un nombre de processeurs optimal entre 8 et 16.

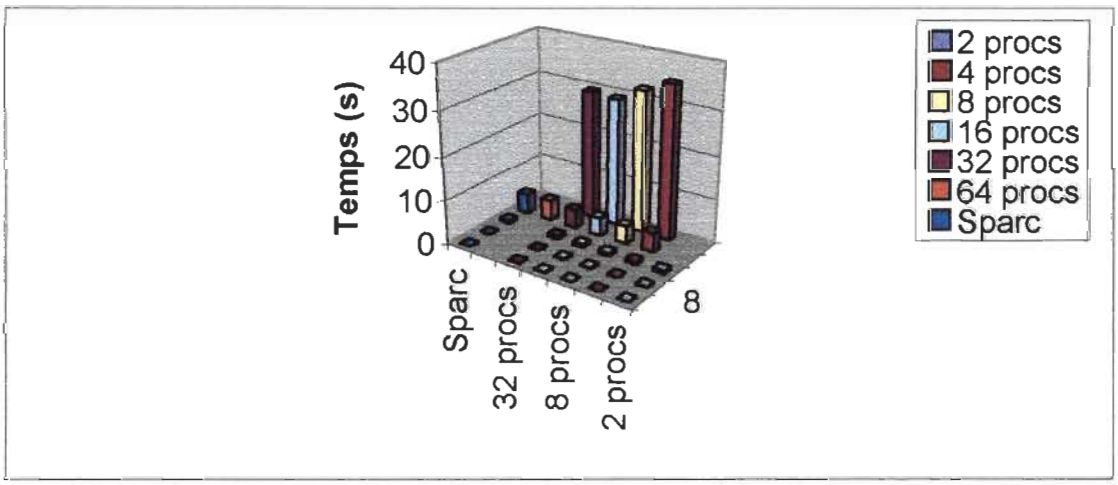

Figure 7.14: Compilation des résultats sur AVX2 et sur Sparc 20.

Pour le cas de l'AVX3, la figure 7.15 montre les résultats. On dénote très facilement la grande inefficacité avec les dimensions de 32 et 64. La dimension de 128 n'apparaît pas ici car les essais n'ont été effectués que pour un seul nombre de processeur.

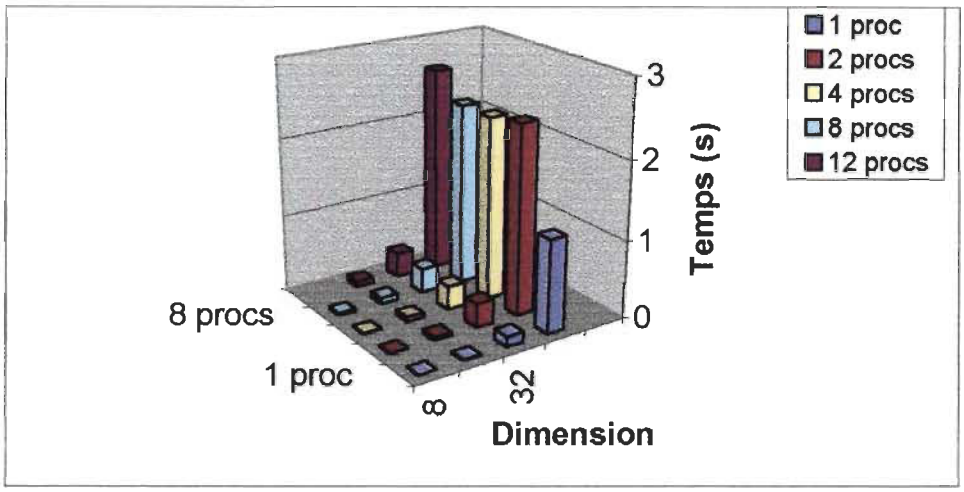

Figure 7.15: Compilation des résultats sur AVX3.

C'est donc en MPI sur AVX3 que les performances sont les meilleures et ce pour un faible nombre de processeurs. Il faut comparer les gains en vitesse et non les temps car ceux-ci ne sont pas calculés par la même fonction et sont par le fait même impossibles à comparer. Toute l'étude comparative des résultats entre la version Trollius et MPI du programme se fait par rapport au gain en vitesse.

8 Conclusions

Comme on peut le remarquer, le mot conclusion est au pluriel ici car il y a plusieurs conclusions à tirer de ce travail. D'abord le programme lui-même est, je pense humblement, une réussite. Bien sûr, il existe certaines améliorations à apporter, tant au niveau de la présentation de l'interface que de l'algorithme parallèle. L'amélioration la plus importante à apporter serait de faire en sorte que tous les processeurs puissent afficher directement dans la fenêtre X Window principale. Ceci ferait en sorte que le transfert d'information entre les processeurs serait réduit au minimum, et par le fait même, augmenterait considérablement les performances de l'interface graphique au niveau de l'affichage qui sont, comme on l'a vu, plutôt médiocres. Cette modification implique beaucoup trop de modifications aux quelques 3500 lignes de codes écrites dans ce projet. Mais d'abord, il faut passer vers le standard MPI car il assure une fiabilité parfaite au niveau des communications. On n'en a pas discuté encore dans le présent document, mais il y a eu des erreurs de communication non résolues lors des essais sur l'ordinateur AVX2. En effet, certaines valeurs se retrouvaient, après communication, tout simplement nulles, engendrant des défectuosités d'affichage. Tous les efforts ont été déployés afin de résoudre ce problème, mais sans succès.

Parlant de l'ordinateur AVX2 le problème important rencontré avec cette machine est le fait que les tampons de communication devenaient souvent pleins, ce qui causait des confusions dans les messages. Dans le cas où plusieurs processeurs esclaves tentaient d'envoyer un message au processeur maître en même temps, les messages se trouvaient entremêlés. Ce phénomène ne s'est pas produit sur AVX3 et tous les processeurs esclaves

envoyaient leur messages en même temps au maître sans confusion. Par contre ce fait n'augmente que très peu les performances de l'interface graphique.

L'ordinateur AVX3 de son côté possède des nœuds plus puissants que sur l'ordinateur AVX2, ce qui fait sa grande force. Par contre, son réseau de communication est très peu efficace. En effet, il s'agit d'un réseau à 100 Mbps, mais il est non commuté. Ceci veut dire que plus on augmente le nombre de processeurs, plus on est pénalisé au niveau des communications, et la bande passante de 100 Mbps est partagée entre tous les nœuds. L'utilisation de commutateurs (*switch*) aurait eu un meilleur effet sur les performances du système. Le fait d'avoir implanté exactement le même code dans les deux machines et d'avoir des performances telles que décrites dans la section d'interprétation des résultats dénote un problème réel avec l'AVX3.

Les résultats obtenus sont intéressants car il font ressortir un point très important en parallélisme. Comme la rotation n'implique pas de communication et que l'affichage en implique beaucoup, nous avons donc les deux cas extrêmes. Les performances sont excellentes en rotation et pauvres en affichage. En programmation parallèle, il faut obtenir le meilleur compromis entre le calcul pur et la communication. Trop de communication engendre des performance médiocres, ce qui a été prouvé de manière efficace dans ce travail. Si le temps le permettait, il y aurait lieu de travailler pour diminuer davantage le temps de communication augmentant ainsi les performances de l'interface graphique. De plus, cela constituerait un cas intermédiaire autour duquel il serait intéressant d'effectuer des comparaisons.

Que conclure au niveau du parallélisme en graphisme par ordinateur? D'abord, comme mentionné précédemment, on doit utiliser le paradigme où tous les processeurs sont

capables d'afficher directement dans la fenêtre graphique, ce qui diminue grandement les communications augmentant ainsi grandement les performances de l'interface graphique. Notons que ce type ce connexion n'est possible que sur l'ordinateur AVX3 ou encore sur un réseau de stations de travail.

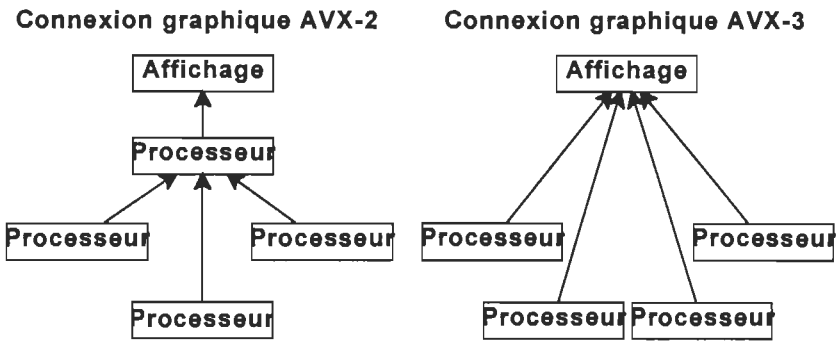

Figure 8.1: Connexions graphiques sur AVX2 et sur AVX3.[4]

Aussi, l'utilisation d'un réseau commuté est de mise. D'ailleurs, le service de l'informatique de l'UQTR se dirige vers cette pratique, remplaçant graduellement tous les "*hubs*" non commutés par des "*switchs*" sur son réseau interne. Je pense que l'utilisation d'un réseau de stations de travail sur un réseau 100 Mbps commuté, avec LAM 6.1 et le standard MPI en programmation serait la meilleure alternative pour implanter les améliorations nécessaires à l'interface graphique. Aussi, le réseau de stations de travail possède le meilleur rapport qualité/prix lorsqu'on veut faire l'acquisition d'un ordinateur parallèle. Le nombre de processeurs doit aussi être pris en compte sérieusement. Une des références utilisée cite que le nombre de processeurs ne doit pas dépasser 10 pour les applications de graphisme [7], ce que nous avons démontré par nos résultats dans lesquels on a retrouvé des nombres de processeurs optimaux de 4 ou 8.

Que dire maintenant au niveau de la programmation MPI. En terme de programmation, il est plus simple de coder les lignes MPI que les lignes Trollius. L'envoi

et la réception des messages est simple et efficace. MPI possède plusieurs caractéristiques qui n'ont pas été exploitées dans ce travail. Ces caractéristiques en font un outil performant mais surtout fiable. En effet, le transfert des messages entre les processeurs se fait sans erreurs. Le fait de voir disparaître les erreurs au passage vers MPI montre sa grande fiabilité. Il est dommage que les machines AVX2 ne puissent pas se soumettre à ce standard car leur avenir est compromis puisque le standard MPI est largement utilisé dans la communauté scientifique, et on parle même de MPI-2.

Finalement, la réalisation de ce travail de recherche m'a permis de parfaire ma connaissance du langage C, surtout dans les domaines de la programmation XWindow et de la programmation parallèle. En plus, les travaux réalisés m'ont permis de connaître à fond les dessous de la simulation numérique dans le domaine de l'électromagnétisme. Les ordinateurs parallèles sont porteurs d'avenir dans la réalisation des interfaces graphiques et des programmes de calcul demandant beaucoup de ressources, et nous avons démontré dans ce travail qu'il est possible de sauver un temps précieux par l'utilisation des ordinateurs parallèles.

Références

[1] S. Xueqin, W. Qing, P. Yuqing, "Parallel Graph Display and Animation Technique for Three Dimensional Magnetic Distributive Field", *Proceedings of the 4th International Conference on Computer - Aided Drafting, Design and Manufacturing Technology*, Vol. 1, pp. 37-40, China, 1994.

[2] P. Zhigeng, S. Jiaoying, "DGPSE: A Distributed Graphics Processing Support Environment", *Proceedings of the Third International Conference on CAD and Computer Graphics*, Vol.1, pp. 54-57, 1993.

[3] P. Zhigeng, S. Jiaoying, "The Design and Implementation of a Distributed Graphics Language", *Proceedings TENCON '93. 1993 IEEE Region 10 Conference on 'Computer, Communication, Control and Power*, Vol.1, p. 181-183, 1993

[4] W. Gropp, E. Karrels, E. Lusk, "MPE Graphics-Scalable X11 Graphics in MPI", *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pp. 49-54, 1995.

[5] P. Tousignant, *Manuel d'introduction au logiciel FLUX2D*, Université du Québec à Trois-Rivières, 1994.

[6] Site World Wide Web du Japan Research Institute, http://www.jri.co.jp/pro-eng/jmagworks/E/postproc.html, The Japan Research Institute Ltd, Tokyo, Japan, 1998.

[7] R. Perala, S. Whittle, M. Hargreaves, P. Kerton, "An Introduction to PEPSE (Parallel Electromagnetics Problem Solving Environment): and Considerations for Parallelization of a Finite Difference Time-Domain Solver", *International Journal of*

Numerical Modelling: Electronic Networks, Devices and Fields, Vol. 8, pp. 187-203, 1995.

[8] R. Janssen, M. Dracopoulos, K. Parrott, E. Slessor, P. Alotto, P. Molfino, M. Nervi, J. Simkin, "Parallelisation of Electromagnetic Simulation Code", *Proceedings of the XIth Conference on the Computation of Electromagnetic Fields (COMPUMAG)*, Rio de Janeiro, Nov. 1997, pp. 199-200.

[9] V. Kumar, A. Grama, A. Gupta, G. Karipis, *Introduction to Parallel Computing, design and analysis of algorithms*, The Benjamin Cummings Publishing Company Inc., Redwood City, California, 1994.

[10] *AVX Series 2 and Configuration Software*, Reference Manuals, Alex Informatics Inc., 1995.

[11] G. Burns, R. Daoud, J. Vaigl, "LAM: An Open Cluster Environment for MPI", Ohio Supercomputer Center, Columbus, Ohio.

[12] AVX-3 World Wide Web page, *http://www.alex.qc.ca/e/avx.htm*, Alex Informatics Inc., Montreal, 1997.

[13] J .D. Foley, A. VanDam, S.K. Feiner, J.F. Hughes, *Computer Graphics, Principles and Practice, Second Edition*, Addison-Wesley Publishing Company, 1990.

[14] D. Heller, P.M. Ferguson, *Motif Programming Manual*, O'Reilly & Associates, 1994.

[15] Site World Wide Web LAM/MPI, http://www.osc.edu/lam.html, Ohio Supercomputer Center, Columbus, Ohio, 1998.

Bibliographie

- M.S. Mirotznik, D. Prather, "How to choose EM software", *IEEE Spectrum*, Vol. 34, No. 12, Dec. 1997, pp. 53-58.
- P. Hamelin , S. Legendre, C. Ortiz, and A. Skorek, "Biofields Parallel Modeling", *Proceedings of the XIth Conference on the Computation of Electromagnetic Fields (COMPUMAG)*, Rio de Janeiro, Nov. 1997, pp. 87-88.
- Message Passing Interface Forum, *MPI: A message passing interface standard*, Computer Science Dept. Technical Report, University of Tennessee, Knoxville, TN, 1994.
- *Alex-Trollius Alex-Xtrollius Alex-Brenda*, Reference Manuals, Alex Informatics Inc., 1995.
- G. D. Burns, "A Local Area Multicomputer", *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, 1989.
- Ohio Supercomputer Center, The Ohio State University, *MPI Primer / Developing With LAM*, Nov. 1996.

Annexe A: Guide d'utilisation du post-processeur

A.1 Lancement du programme

En séquentiel

La version séquentielle du post-processeur est lancée simplement par la commande "postproc" à partir du répertoire "/u/contrib/hamelin" accessible par tous les usagers. Ce répertoire contient:

- La version exécutable du programme "postproc"
- Trois fichiers de simulation en hautes fréquences (`HF16`, `HF32` et `HF64`)
- Un fichier de données hautes fréquences de dimension 30 provenant des travaux de Mme. Sylvie Legendre (`tete_1`)
- Un fichier de données en basses fréquences de dimension 10 provenant des travaux de M. Carlos Ortiz (`vecsol_1000`).

Pour les instructions d'utilisation proprement dite, référez-vous à la section A.2.

En parallèle version Trollius

Afin de faire tourner l'interface graphique en mode Trollius sur les ordinateurs AVX-2, connectez-vous d'abord sur l'ordinateur nommé "lei04" par la commande:

```
telnet lei04
```

Vous aurez ainsi accès à un maximum de 64 processeurs. Assurez-vous que votre fichier ".profile" contient bel et bien les lignes:

```
SKEL=/u/uq/canvas/skel
DTSOURCEPROFILE=true
```

```
export SKEL DTSOURCEPROFILE
. $SKEL/Profile.general
. $SKEL/Profile.alex
. $SKEL/Profile.sdk
cd $HOME
```

Après avoir configuré la variable d'environnement "DISPLAY", déplacez-vous dans le répertoire "`/u/contrib/hamelin`" et lancez la commande "`run`" en respectant le format:

```
run <dimension X> <dimension Y> <no du dernier nœud>
```

Par exemple, la commande "`run 4 4 15`" lancera le post-processeur sur une grille de 4 par 4 et le numéro du dernier processeur est 15 (noeuds 0 à 15). Pour les instructions d'utilisation du logiciel, reportez-vous à la section A.2.

En parallèle version MPI

Afin de faire tourner l'interface graphique en mode MPI, vous devez d'abord avoir accès à un des 4 ordinateurs nommés lei01, lei02, lei03 ou lei04, et vous devez vous assurer d'avoir les lignes suivantes dans votre fichier ".profile":

```
SKEL=/u/uq/canvas/skel
DTSOURCEPROFILE=true
export SKEL DTSOURCEPROFILE
. $SKEL/Profile.general
. $SKEL/Profile.lam
. $SKEL/Profile.sdk
cd $HOME
```

Ensuite, déplacez-vous dans le répertoire "`/u/contrib/hamelin`" et exécutez la commande:

```
runMPI
```

Cette commande exécutera l'interface graphique sur 4 processeurs dans sa version MPI. Reportez-vous à la section suivante pour les détails concernant l'utilisation des menus.

A.2 Instructions d'utilisation

L'interface graphique a été construite afin qu'elle soit simple d'utilisation. Elle possède donc des menus permettant un accès rapide aux différentes fonctions disponibles.

Le menu "File" permet la gestion des fichiers. Tel que vu dans la section 5.4.2, il est possible d'ouvrir des fichiers sous trois formats différents. Le premier format, soit le format "Articulation" a été délaissé quelque peu car il a servi principalement au développement de l'interface usager. Le format HF (Haute Fréquence) quant à lui permet d'ouvrir les fichiers de résultats du module de calcul en hautes fréquences. Le répertoire "/u/contrib/hamelin" contient quelques fichiers possédant le préfixe "HF" qui peuvent être ouverts par cette option du menu "File". Le seul fichier basses fréquences disponible dans ce même répertoire est le fichier "vecsol_1000". L'option "Save" effectue la sauvegarde en format "Articulation" seulement et la fonction "Print" n'est implantée. Finalement l'option "Quit" sert bien évidemment à quitter le logiciel.

Le menu "Image" effectue les opérations de translation et de rotation des images. Pour la rotation, une fenêtre avec trois curseurs apparaîtra à l'écran vous permettant d'effectuer les rotations dans les trois axes. La translation possède son origine dans le haut de l'écran du côté droit. Lors de l'ouverture d'un fichier "HF", une translation positive d'environ 35 permettra de ramener l'image vers le centre de l'écran.

Le menu "Zoom" effectue les "zoom in" et "zoom out" de l'image. Il fonctionne exactement comme tous les logiciels commerciaux permettant d'effectuer cette opération.

Finalement, le menu option permet d'aider à l'interprétation des résultats en effectuant des affichages par couleur ou par plan. Premièrement, on peut effectuer

l'enlèvement des valeurs nulles qui pourraient ne pas porter grand intérêt dans les résultats dans certaines situations. Les deux premières options du menu "Options" permettent donc d'enlever et de réafficher les valeurs nulles. L'option suivante, "Display by colors", permet à l'utilisateur d'afficher seulement une couleur. La couleur est identifiée par le numéro inscrit complètement à gauche de la légende. Après affichage de la première couleur, l'option "Hold ON" peut être utilisée pour conserver cette couleur à l'écran pour en superposer une ou plusieurs autres. Le mot "HOLD" apparaîtra alors sous la légende. Lorsque l'option "Hold OFF" est choisie, seule la dernière couleur affichée sera conservée sur l'écran. Pour annuler l'affichage par couleur, l'option "Show all colors" doit être sélectionnée.

L'autre option du menu portant le même nom est l'option d'affichage par plans. En choisissant l'option "Display by plan", l'utilisateur peut afficher seulement un plan en entrant l'axe et la coordonnée du plan désiré, par exemple x=15. Les options "Next plan" et "Previous plan" peuvent être sélectionnées pour afficher les plans suivants et précédents. Encore une fois, l'option "Hold ON" peut être sélectionnée pour effectuer l'affichage de plusieurs plans simultanément. Pour annuler l'option d'affichage par plan, il faut choisir "Show all plans" dans le menu option. Il est à noter que les options d'affichage par couleur et d'affichage par plan, si utilisées avec l'option "Hold ON", effectuera l'affichage de l'intersection des deux options, c'est-à-dire que seulement les points appartenant aux deux options seront affichés.

Annexe B: Listings

Pour une raison de confidentialité, il est impossible de placer dans ce rapport le listing complet de l'interface graphique. Seuls des extraits ou encore des fonctions bien précises seront étalées dans cette annexe. La version MPI de l'interface graphique sera utilisée car elle est la plus récente et la plus prometteuse d'avenir. Nous allons donc voir premièrement l'entête du programme (fichier `postproc_MPI.h`) pour ensuite voir des extraits du programme principal nommé `postproc_MPI.c`.

La taille des caractères a été réduite à 8 points en raison du volume important du programme, soit environ 2900 lignes de programmation en langage C.

Postproc_MPI.h

```
/*
 *      Inclusions
 */
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
#include <tstdio.h>
#include <fcntl.h>
#include <math.h>
#include <labo.h>
#include <t_types.h>
#include <Xm/MainW.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>
#include <Xm/DrawingA.h>
#include <Xm/DialogS.h>
#include <Xm/RowColumn.h>
#include <Xm/ScrolledW.h>
#include <Xm/Form.h>
#include <Xm/LabelG.h>
#include <Xm/Text.h>
#include <Xm/SelectioB.h>
#include <Xm/Scale.h>
#include <Xm/Frame.h>
#include <X11/Xcms.h>
/*
 *      Definition des constantes
 */
#define         PI                      3.141592654
#define         VRAI                    1
#define         FAUX                    0
#define         ON                      1
#define         OFF                     0
#define         MAXCOLORS               10
#define         COMM_INIT_HF            0       // Jeu de commande aux esclaves
#define         COMM_REQ_AFF            1       // Jeu de commande aux esclaves
#define         COMM_ROTATION           2       // Jeu de commande aux esclaves
#define         COMM_TRANSLAT           3       // Jeu de commande aux esclaves
#define         COMM_LEGEND             4       // Jeu de commande aux esclaves
#define         COMM_QUIT               5       // Jeu de commande aux esclaves
#define         COMM_TAG                100     // TAG MPI pour load_file_HF
#define         COMM_TAG_2              110     // TAG MPI pour load_file_HF
#define         AFF_TAG                 20      // TAG MPI pour Affichage
#define         COMM_BUF_SIZE           10
#define         MAXPOINTS               1000000
#define         WIDTH                   800
#define         HEIGHT                  800
#define         LEGEND_X                20
#define         LEGEND_Y                30
#define         LEGEND_H                30
#define         LEGEND_W                25
#define         ZOOM_FACTOR             10
#define         REC_SIZE                3
#define         RECEPT_SIZE             1000000
```

```
/*
 *      Structure graphique
 */
typedef struct
{
        int     x0,y0,z0;       // Valeurs Originales pour affichage_plan
        float   x, y, z;        // Coordonnees
        float   ppx, ppy;       // Points de projection
        float   valeur;         // Valeur associee au point
        int     coul;           // Couleur (0-MAXCOLORS)
        int     hold;           // Hold    (VRAI|FAUX)
        int     enable;         // Enable (VRAI|FAUX)
} POINT;

typedef struct
{
        POINT **pts;            // Pointeur sur un point
        int     numpts;         // Nombre de points
} SURFACE;

typedef struct
{
        SURFACE *surf;          // Pointeur sur une surface
        POINT *pts;             // Pointeur sur un point
        int numsurf, numpts;    // Nombre de points et de surfaces
} FIGURE;

typedef struct
{
        POINT PO,PX,PY,PZ;      // Coordonnees de reference
        FIGURE *fig;            // Pointeur sur une figure
        int numfig;             // Nombre de figures
} ARTICUL;

typedef struct
{
        float r, g, b;          // Valeur RGB pour couleurs
} TAB_RGB;
/*
 *      Prototypes des fonctions
 */
void    menufichier             ( Widget, XtPointer, XtPointer      );
void    menuzoom                ( Widget, XtPointer, XtPointer      );
void    menuimage               ( Widget, XtPointer, XtPointer      );
void    menurotation            ( Widget, XtPointer, XtPointer      );
void    menucouleurs            ( Widget, XtPointer, XtPointer      );
void    menuoptions             ( Widget, XtPointer, XtPointer      );
void    load_file               ( Widget, XtPointer, XtPointer      );
void    load_file_HF            ( Widget, XtPointer, XtPointer      );
void    load_file_LF            ( Widget, XtPointer, XtPointer      );
void    save_file               ( Widget, XtPointer, XtPointer      );
void    expose_callback         ( Widget, XtPointer, XtPointer      );
void    resize_callback         ( Widget, XtPointer, XtPointer      );
void    Erreur                  ( Widget, char *                    );
void    Translation             ( Widget, XtPointer, XtPointer      );
void    Projection              ( void                              );
void    Affiche_HF              ( void                              );
void    Affiche_Rectangle       ( POINT *, int                      );
void    Affiche_Rectangle_par   ( POINT                             );
void    Affiche_Articul         ( ARTICUL *                         );
void    Affiche_Figure          ( FIGURE *                          );
void    Affiche_Surface         ( SURFACE *                         );
float   Surface_test            ( SURFACE *                         );
void    Affiche_Ligne           ( POINT *, POINT *                  );
void    Affiche_Couleur         ( Widget, XtPointer, XtPointer      );
void    Affiche_Plan            ( Widget, XtPointer, XtPointer      );
void    Rotation                ( Widget, XtPointer, XtPointer      );
void    Transformation          ( float total[4][4]                 );
```

```
void    Transforme_point      ( POINT *, float total[4][4]        );
int     Clipping              ( POINT *, POINT *                  );
void    Normalise             ( POINT *                           );
void    Fond_noir             ( void                              );
void    Fond_blanc            ( void                              );
void    Get_Colors            ( void                              );
void    Legende               ( void                              );
void    Usage                 ( void                              );
void    Req_Affichage         ( void                              );
void    Lecture_Fichier_HF    ( void                              );
void    Affichage             ( void                              );
void    Rotation_slave        ( char, float                       );
void    Translation_slave     ( float, float                      );
```

Fonction main()

```
main(int argc, char **argv)
{
    XtAppContext      app;                          // Application context
    XGCValues         gcv;                          // Graphic context values
    Widget            menubar;                      // Barre de menu superieure
    Widget            widget;                       // Widget temporaire
    Widget            menu_fichier;                 // Widget menu fichier
    char              *window_name = "Biomag: Postprocessor Display";

    MPI_Init(&argc, &argv);                         // Initialisation MPI
    MPI_Comm_rank (MPI_COMM_WORLD, &noeud);         // Nombre de nœuds total
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);      // Numero de processeur (rank)

    pv.x = (vir_fin_x + vir_deb_x)/2;               // Point de visualisation X
    pv.y = (vir_fin_y + vir_deb_y)/2;               // Point de visualisation Y
    pv.z = -1000.0;                                 // Point de visualisation Z

    if (noeud == 0)
     {
        XtSetLanguageProc ( NULL, NULL, NULL );
        top = XtAppInitialize ( &app, "Postproc", NULL, 0, &argc, argv, NULL, NULL,0 );
        mainwin = XtVaCreateManagedWidget ( "mainwin",
                            xmMainWindowWidgetClass, top,
                            XmNtitle, "Biomag: Postprocessor Display",
                            NULL );
        draw = XtVaCreateWidget ( "draw", xmDrawingAreaWidgetClass, mainwin,
                            XmNwidth, draw_x,
                            XmNheight, draw_y,
                            XmNtitle, "Biomag: Postprocessor Display",
                            NULL );
        XtAddCallback (draw, XmNexposeCallback, expose_callback, NULL);
        XtAddCallback (draw, XmNresizeCallback, resize_callback, NULL);

        dpy = XtDisplay ( draw );
        gc = XCreateGC (      dpy,
                    RootWindowOfScreen ( XtScreen ( draw )),
                    GCForeground | GCBackground,
                    &gcv );

        XtVaSetValues ( draw, XmNuserData, gc, NULL );

        menubar =   XmVaCreateSimpleMenuBar ( mainwin, "menubar",
               XmVaCASCADEBUTTON, XmStringCreateLocalized ("File"), 'F',
               XmVaCASCADEBUTTON, XmStringCreateLocalized ("Image"), 'I',
               XmVaCASCADEBUTTON, XmStringCreateLocalized ("Zoom"), 'Z',
               XmVaCASCADEBUTTON, XmStringCreateLocalized ("Colors"), 'C',
               XmVaCASCADEBUTTON, XmStringCreateLocalized ("Options"), 'O',
               XmVaCASCADEBUTTON, XmStringCreateLocalized ("Help"), 'H',
               NULL );
```

```
        if (widget = XtNameToWidget (menubar, "button_5"))
             XtVaSetValues (menubar, XmNmenuHelpWidget, widget, NULL);
        menu_fichier = XmVaCreateSimplePulldownMenu ( menubar,
                "menu_fichier", 0, menufichier,
                   XmVaPUSHBUTTON,XmStringCreateLocalized ("Open Articul"),'A',NULL,NULL,
                   XmVaPUSHBUTTON,XmStringCreateLocalized ("Open HF"),'H',NULL,NULL,
                   XmVaPUSHBUTTON,XmStringCreateLocalized ("Open LF"),'L',NULL,NULL,
                   XmVaPUSHBUTTON,XmStringCreateLocalized ("Save"),'S',NULL,NULL,
                   XmVaSEPARATOR,
                   XmVaPUSHBUTTON,XmStringCreateLocalized ("Print"),'P',NULL,NULL,
                   XmVaSEPARATOR,
                   XmVaPUSHBUTTON,XmStringCreateLocalized ("Quit"),'Q',NULL,NULL,NULL );

        XmVaCreateSimplePulldownMenu ( menubar, "menu_zoom", 2, menuzoom,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("In"),'I',NULL,NULL,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Out"),'O',NULL,NULL,
                NULL );
        XmVaCreateSimplePulldownMenu ( menubar, "menu_image", 1, menuimage,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Translation"),'T', NULL, NULL,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Rotation"),'R', NULL, NULL,
                NULL, NULL );
       XmVaCreateSimplePulldownMenu ( menubar, "menu_coul", 3, menucouleurs,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Black Background"),'B', NULL, NULL,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("White Background"),'W', NULL, NULL,
                NULL, NULL );
       XmVaCreateSimplePulldownMenu ( menubar, "menu_opt", 4, menuoptions,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Hide zero values"),'H', NULL, NULL,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Show zero values"),'S', NULL, NULL,
                XmVaSEPARATOR,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Display by colors"),'D', NULL, NULL,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Show all colors"),'S', NULL, NULL,
                XmVaSEPARATOR,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Display by plan"),'p', NULL, NULL,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Next plan"),'N', NULL, NULL,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Prev plan"),'P', NULL, NULL,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Show all plans"),'a', NULL, NULL,
                XmVaSEPARATOR,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Hold ON"),'O', NULL, NULL,
                XmVaPUSHBUTTON,XmStringCreateLocalized ("Hold OFF"),'F', NULL, NULL,
                NULL, NULL );

        XtManageChild ( draw );
        XtManageChild ( menubar );
        XtRealizeWidget ( top );
        Fond_noir();
        XtAppMainLoop ( app );

    }
    else
        slave();

    MPI_Finalize();

}
```

Programme esclave

```
slave()
{
    int        commande;
    int        i;

    while ( VRAI )
    {
        MPI_Recv (comm_buf,COMM_BUF_SIZE,MPI_FLOAT,0,COMM_TAG,MPI_COMM_WORLD,&status);
        commande = (int) comm_buf[0];
        switch (commande)
        {
        case 0:                              // Commande de chargement de donnees
            Lecture_Fichier_HF ();
            break;
        case 1:                              // Commande d'affichage
            cache_zero          = (int) comm_buf[1];
            affiche_par_couleur = (int) comm_buf[2];
            affiche_par_plan    = (int) comm_buf[3];
            hold_enable         = (int) comm_buf[4];
            plan_a_afficher     = (int) comm_buf[5];
            plan                = (int) comm_buf[6];
            coul                = (int) comm_buf[7];
            Affichage();
            break;
        case 2:                              // Commande de rotation
            temps=MPI_Wtime();
            Rotation_slave ( (char)comm_buf[1], comm_buf[2]);
            temps_rot = MPI_Wtime()-temps;
            if (logger) tprintf ("S%d:Rotation=%f\n",noeud,temps_rot);
            break;
        case 3:                              // Commande de translation
            Translation_slave(comm_buf[1], comm_buf[2]);
            Projection();
            break;
        case 4:                              // Envoie des temps au maitre
            comm_buf[0] = (float) temps_tot;
            comm_buf[1] = (float) temps_comm;
            comm_buf[2] = (float) temps_rot;
            MPI_Send (comm_buf, COMM_BUF_SIZE,MPI_FLOAT, 0, COMM_TAG_2, MPI_COMM_WORLD);
            break;
        case 5:                              // Commande de fin
            return;
            break;
        }
    }
}
```

Fonction de lecture de fichier haute fréquence et séparation des tâches

```
void load_file_HF (widget, client_data, call_data)
Widget widget;
XtPointer client_data;
XtPointer call_data;
{
    int         i, k;
    float       valeur;
    int         nbx, nby, nbz;
    int         x,y,z;
    int         ctr = 0;
    FILE        *sol;
    char        *text;
    int         reste, reste_procs;
    int         max_x = 0;                      // Maximum en x
    int         max_y = 0;                      // Maximum en y
    int         max_z = 0;                      // Maximum en y
    int         min_x = 10000;                  // Minimum en x
    int         min_y = 10000;                  // Minimum en y
    int         min_z = 10000;                  // Minimum en y
    float       dim_vir_x = 0;                  // Dimension de la fenetre virtuelle en X
    float       dim_vir_y = 0;                  // Dimension de la fenetre virtuelle en Y
    float       min_val =  10000.0;             // Minimum Valeur
    float       max_val = -10000.0;             // Maximum Valeur
    float       lim;                            // limite pour le calcul des couleurs
    unsigned int offset[numprocs];              // Table d'offset (plan de depart du proc
    unsigned int elem_par_proc[numprocs];       // Table d'elemenet par processeur

    /*
     * Definition de la structure callback cbs qui va contenir le nom de
     * fichier a ouvrir
     */

    XmSelectionBoxCallbackStruct *cbs = (XmSelectionBoxCallbackStruct *)call_data;
    XmStringGetLtoR (cbs->value, XmFONTLIST_DEFAULT_TAG, &text);
    XtDestroyWidget (widget);

    affiche_par_couleur = FAUX;
    affiche_par_plan = FAUX;
    cache_zero = FAUX;
    temps_tot = 0.0;
    temps_comm= 0.0;
    temps_rot = 0.0;
    ctr_aff = 0;
    ctr_rot = 0;
    if (logger) tprintf ("Ouverture\n");

    /*
     * Ouverture du fichier des coordonnees
     */

    if ((sol = fopen (text, "rb")) == NULL)
    {
        Erreur (widget, "Erreur d'ouverture de fichier!");
        return;
    }

    if ( ( fread (&nbx, sizeof(int), 1, sol)) != 1)
    {
        printf ("Erreur de lecture de x\n");
        exit(0);
    }
```

```
if ( ( fread (&nby, sizeof(int), 1, sol)) != 1)
{
    printf ("Erreur de lecture de y\n");
    exit(0);
}

if ( ( fread (&nbz, sizeof(int), 1, sol)) != 1)
{
    printf ("Erreur de lecture de z\n");
    exit(0);
}
min_x = 0;
min_y = 0;
min_z = 0;
max_x = nbx-1;
max_y = nby-1;
max_z = nbz-1;

a.PO.x = (float)  nbx/2;
a.PO.y = (float)  nby/2;
a.PO.z = (float)  nbz/2;
a.PX.x = (float)  a.PO.x+10.0;
a.PX.y = (float)  a.PO.y;
a.PX.z = (float)  a.PO.z;
a.PY.x = (float)  a.PO.x;
a.PY.y = (float)  a.PO.y+10.0;
a.PY.z = (float)  a.PO.z;
a.PZ.x = (float)  a.PO.x;
a.PZ.y = (float)  a.PO.y;
a.PZ.z = (float)  a.PO.z+10.0;

vir_deb_x = 0.0;
vir_deb_y = 0.0;
vir_fin_x = (float)(((max_x / 100) * 100) +100);
vir_fin_y = (float)(((max_y / 100) * 100) +100);

dim_vir_x = vir_fin_x - vir_deb_x;
dim_vir_y = vir_fin_y - vir_deb_y;

/*
 *  Dimension en x et y doivent etre egales
 */

    if ( dim_vir_x != dim_vir_y )
    {
        if ( dim_vir_x > dim_vir_y )
            vir_fin_y = vir_deb_y + dim_vir_x;
        else
            vir_fin_x = vir_deb_x + dim_vir_y;
    }

/*
 *  Calcul du point de visualisation au centre de la fenetre virtuelle
 */

pv.x = (vir_fin_x + vir_deb_x)/2;
pv.y = (vir_fin_y + vir_deb_y)/2;

reste = nbz;
reste_procs = numprocs-1;

comm_buf[0] = (float) COMM_INIT_HF;
comm_buf[1] = (float) nbx;
comm_buf[2] = (float) nby;
comm_buf[3] = (float) nbz;
comm_buf[5] = (float) pv.x;
comm_buf[6] = (float) pv.y;
comm_buf[7] = (float) pv.z;
```

```
//
// Separation de taches
//

    for (i=1; i<numprocs; ++i)
    {
        elem_par_proc[i] = reste / reste_procs;
        reste -= elem_par_proc[i];
        --reste_procs;
        offset[i] = 0;
        for (k=1; k<i; ++k)
            offset[i] +=elem_par_proc[k];

        MPI_Send (comm_buf,COMM_BUF_SIZE,MPI_FLOAT,i,COMM_TAG,MPI_COMM_WORLD);
    }
    for (i=1; i<numprocs; ++i)
    {
        float buffer[nbx*nbz*elem_par_proc[i]];
        buffer[0] = (float) elem_par_proc[i];
        buffer[1] = (float) offset[i];
        ctr=2;

        for ( z=0; z<elem_par_proc[i]; ++z)
        {
            for ( y=0; y<nby; ++y)
            {
                for ( x=0; x<nbx; ++x)
                {
                    if ( ( fread (&buffer[ctr], sizeof(float), 1, sol)) != 1)
                    {
                        Erreur (widget, "Erreur de lecture de valeur");
                        fclose (sol);
                        exit(0);
                    }
                    if (buffer[ctr] < (float)min_val)  min_val = (float)buffer[ctr];
                    if (buffer[ctr] > (float)max_val)  max_val = (float)buffer[ctr];
                    ++ctr;
                }
            }
        }
        MPI_Send (buffer,nbx*nbz*elem_par_proc[i]+2,MPI_FLOAT,i,COMM_TAG_2,MPI_COMM_WORLD);
    }

    fclose(sol);
    lim = (max_val-min_val)/MAXCOLORS;
    for (i=0; i<MAXCOLORS+1; ++i)
        limites[i] = i*lim + min_val;
    file_loaded = VRAI;

    Req_Affichage ();
}
```

Fonction de calcul de la projection

```
/*
 *      Fonction de calcul des points de projection ppx et ppy pour chacun
 *      des points de la geometrie
 */

void Projection (void)
{
    int         i, j;

    for (i=0; i<a.numfig; ++i)
    {
        for (j=0; j<a.fig[i].numpts; ++j)
        {
            a.fig[i].pts[j].ppy = pv.y + (((a.fig[i].pts[j].y - pv.y) * pv.z) /
                                  (a.fig[i].pts[j].z - pv.z));

            a.fig[i].pts[j].ppx = pv.x + (((a.fig[i].pts[j].x - pv.x) * pv.z) /
                                  (a.fig[i].pts[j].z - pv.z));
        }
    }
}
```

Fonction de calcul de la translation en parallèle (maître)

```
void Translation (widget, client_data, call_data)
Widget widget;
XtPointer client_data;
XtPointer call_data;
{
    char                *text, *ptr;
    static int          x, y;
    static int          i, j;
    XmSelectionBoxCallbackStruct *cbs=(XmSelectionBoxCallbackStruct *)call_data;

    XmStringGetLtoR (cbs->value, XmFONTLIST_DEFAULT_TAG, &text);
    XtDestroyWidget (widget);

    ptr = strchr(text,',');
    if(ptr)
    {
        x = atoi(text);
        y = atoi(ptr+1);
        comm_buf[0] = (float) COMM_TRANSLAT;
        comm_buf[1] = (float) x;
        comm_buf[2] = (float) y;
        for (i=1; i<numprocs; ++i)
            MPI_Send (comm_buf,COMM_BUF_SIZE,MPI_FLOAT,i,COMM_TAG,MPI_COMM_WORLD);

    }
    else
    {
        Erreur (widget, "Entrez X,Y");
        return;
    }

    Req_Affichage ();
}
```

Fonction de calcul de la rotation en parallèle (maître)

```
void Rotation (widget, client_data, call_data)
Widget          widget;
XtPointer       client_data;
XtPointer       call_data;
{
    XmScaleCallbackStruct *cbs = (XmScaleCallbackStruct *) call_data;
    float     A[4][4], T1[4][4], T2[4][4], tmp[4][4], total[4][4],
              p1x, p2x, p1y, p2y, p1z, p2z, vx, vy, vz,
              racine, radian, cosinus, sinus;
    int angle;
    int i,j,k;
    float buf[1];
    sprintf (&axe, "%s", XtName( widget ));
    p1x = a.PO.x;
    p1y = a.PO.y;
    p1z = a.PO.z;

    switch (axe)
    {
        case 'X':
            p2x = a.PX.x;
            p2y = a.PX.y;
            p2z = a.PX.z;
            angle = cbs->value -rx;
            rx = cbs->value;
            break;
        case 'Y':
            p2x = a.PY.x;
            p2y = a.PY.y;
            p2z = a.PY.z;
            angle = cbs->value -ry;
            ry = cbs->value;
            break;
        case 'Z':
            p2x = a.PZ.x;
            p2y = a.PZ.y;
            p2z = a.PZ.z;
            angle = cbs->value -rz;
            rz = cbs->value;
            break;
    }

    if (logger)
    {
        fprintf (logfile, "Commande ROTATION en %s de %d\n",&axe, angle);
        fprintf (logfile, "P1= %8.2f %8.2f %8.2f\n",p1x,p1y,p1z);
        fprintf (logfile, "P2= %8.2f %8.2f %8.2f\n",p2x,p2y,p2z);
        fprintf (logfile, "PV = %8.2f, %8.2f, %8.2f\n",pv.x,pv.y,pv.z);
    }

    if (file_loaded)
    {

        ++ctr_rot;
        comm_buf[0] = (float) COMM_ROTATION;
        comm_buf[1] = (float) axe;
        comm_buf[2] = (float) angle;
        for (i=1; i<numprocs; ++i)
            MPI_Send (comm_buf,COMM_BUF_SIZE,MPI_FLOAT,i,COMM_TAG,MPI_COMM_WORLD);

        Req_Affichage ();
    }
}
```

Fonction de calcul de la transformation

```
//
// Calcul de la transformation appellee par la fonction de rotation
//
void Transformation (float total[4][4])
{
    int i, j;

    Transforme_point (&(a.PX), total);
    Transforme_point (&(a.PY), total);
    Transforme_point (&(a.PZ), total);

    for (i=0; i<a.numfig; ++i)
        for (j=0; j<a.fig[i].numpts; ++j)
            Transforme_point (&(a.fig[i].pts[j]), total);

}

void  Transforme_point (POINT *pt, float total[4][4])
{
float x,y,z;

    x = pt->x;
    y = pt->y;
    z = pt->z;
    pt->x = x*total[0][0] + y*total[1][0] + z*total[2][0] +total[3][0];
    pt->y = x*total[0][1] + y*total[1][1] + z*total[2][1] +total[3][1];
    pt->z = x*total[0][2] + y*total[1][2] + z*total[2][2] +total[3][2];
    if (logger)
        fprintf (logfile,"%8.2f, %8.2f, %8.2f ---> %8.2f, %8.2f, %8.2f\n",x,y,z,pt->x,pt-
>y,pt->z);
}
```

Fonction de rognage

```
int Clipping(POINT *pp1, POINT *pp2)
{
    float tampon;
    float p1x,p2x,p1y,p2y;
    int invis = FAUX;

    p1x = pp1->ppx;
    p1y = pp1->ppy;
    p2x = pp2->ppx;
    p2y = pp2->ppy;

    /* effectue le clippage pour l'axe X */

    if (p1x > p2x)
    {
        tampon = p1x;
        p1x = p2x;
        p2x = tampon;
                                /* permutter P1 avec P2 */
        tampon = p1y;
        p1y = p2y;
        p2y = tampon;
    }

    /* verification de la borne maximum en x du point P1 */

    if (p1x > vir_fin_x)
    {
```

```
    invis = VRAI;          /* le point est invisible */
    return(invis);
}

/* verification de la borne minimum en x du point P2 */

if (p2x < vir_deb_x)
{
    invis = VRAI;         /* le point est invisible */
    return(invis);
}

/* calcul la nouvelle coordonnee x et y du point P1 */

if (p1x < vir_deb_x)
{
    p1y += (p2y - p1y) * (vir_deb_x - p1x) / (p2x - p1x);
    p1x = vir_deb_x;

    if ((p1y >= vir_fin_y) || (p1y <= vir_deb_y))
    {
        invis = VRAI;     /* le point est invisible */
        return(invis);
    }
}

/* calcul la nouvelle coordonnee x et y du point P2 */

if (p2x > vir_fin_x)
{
    p2y -= (p2y - p1y) * (p2x - vir_fin_x) / (p2x - p1x);
    p2x = vir_fin_x;

    if ((p2y >= vir_fin_y) || (p2y <= vir_deb_y))
    {
        invis = VRAI;     /* le point est invisible */
        return(invis);
    }
}

/* effectue le clippage pour l'axe Y */

if (p1y > p2y)
{
    tampon = p1x;
    p1x = p2x;
    p2x = tampon;
                          /* permutter P1 avec P2 */
    tampon = p1y;
    p1y = p2y;
    p2y = tampon;
}

/* verification de la borne maximum en y du point P1 */

if (p1y > vir_fin_y)
{
    invis = VRAI;          /* le point est invisible */
    return(invis);
}

/* verification de la borne minimum en x du point P2 */

if (p2y < vir_deb_y)
{
    invis = VRAI;         /* le point est invisible */
    return(invis);
}
```

```
    /* calcul la nouvelle coordonnee x et y du point P1 */

    if (p1y < vir_deb_y)
    {
        p1x += (p2x - p1x) * (vir_deb_y - p1y) / (p2y - p1y);
        p1y = vir_deb_y;

        if ((p2x >= vir_fin_x) || (p2x <= vir_deb_x))
        {
            invis = VRAI;
            return(invis);
        }
    }

    /* calcul la nouvelle coordonnee x et y du point P2 */

    if (p2y > vir_fin_y)
    {
        p2x -= (p2x - p1x) * (p2y - vir_fin_y) / (p2y - p1y);
        p2y = vir_fin_y;

        if ((p2x >= vir_fin_x) || (p2x <= vir_deb_x))
        {
            invis = VRAI;     /* le point est invisible */
            return(invis);
        }
    }


    /* place les nouveaux points dans la cellule cx et cy */

    pp1->ppx = p1x;
    pp1->ppy = p1y;
    pp2->ppx = p2x;
    pp2->ppy = p2y;

    return(invis);
}
```

Fonction de normalisation des points de projection

```
void Normalise(POINT *p)
{
    float  nx, ny;

    nx = (p->ppx - vir_deb_x)/(vir_fin_x - vir_deb_x);
    ny = (p->ppy - vir_deb_y)/(vir_fin_y - vir_deb_y);

    p->ppx = ecr_deb_x +(nx*(ecr_fin_x-ecr_deb_x));
    p->ppy = ecr_deb_y +(ny*(ecr_fin_y-ecr_deb_y));
    p->ppy = ecr_fin_y - (p->ppy - ecr_deb_y);
}
```

Fonction d'affichage de la légende

```
/*
*       Fonction d'affichage de la legende
*
*       La selection des couleurs se fait dans le "colormap" standard de
*       XWindow
*/
void Legende(void)
{
    int i;
    char s[10];
    int legend_y;
    double moy_tot, moy_comm,moy_rot;
    Window w=XtWindow(draw);

    cmap = DefaultColormap (dpy,0);

    Get_Colors();

    for (i=0; i<MAXCOLORS; ++i)
    {
        XSetForeground(dpy,gc,couleur[i]);
        sprintf (s,"%2d",i);

        legend_y = LEGEND_Y * ( i+1 );
        XDrawRectangle ( dpy, w, gc, LEGEND_X, legend_y, LEGEND_W, LEGEND_H);
        XFillRectangle ( dpy, w, gc, LEGEND_X, legend_y, LEGEND_W, LEGEND_H);
        XDrawString (dpy, w, gc, 0, legend_y+(LEGEND_H/2), s, 2);
    }
    if (hold_enable)
    {
        sprintf (s,"%s", "HOLD");
        XSetForeground(dpy,gc,couleur[0]);
        XDrawString (dpy, w, gc, LEGEND_X, legend_y+(2*LEGEND_H), s, strlen(s));
    }
    if (logger)
    {
        sprintf (s,"%s", "LOG");
        XSetForeground(dpy,gc,couleur[0]);
        XDrawString (dpy, w, gc, LEGEND_X, legend_y+(3*LEGEND_H), s, strlen(s));
    }

    XSetForeground(dpy,gc,couleur[0]);
    for (i=0; i<MAXCOLORS+1; ++i)
    {
        sprintf (s,"%.4e",limites[i]);
        legend_y = LEGEND_Y * ( i+1 );
        XDrawString (dpy, w, gc, LEGEND_X+LEGEND_W+10, legend_y+4, s, strlen(s));
    }
    XSetForeground(dpy,gc,couleur[4]);
    comm_buf[0] = COMM_LEGEND;
    for (i=1; i<numprocs; ++i)
        MPI_Send (comm_buf, COMM_BUF_SIZE,MPI_FLOAT, i, COMM_TAG, MPI_COMM_WORLD);

    for (i=1; i<numprocs; ++i)
    {
        MPI_Recv (comm_buf, COMM_BUF_SIZE,MPI_FLOAT, i, COMM_TAG_2, MPI_COMM_WORLD,
&status);

        temps_tot   += (double) comm_buf[0];
        temps_comm  += (double) comm_buf[1];
        temps_rot   += (double) comm_buf[2];

    }
```

```
    if (ctr_aff != 0)
    {
        moy_tot = temps_tot / (3.0 * ctr_aff);
        moy_comm= temps_comm/ (3.0 * ctr_aff);
    }
    else
    {
        moy_tot = 0.0;
        moy_comm= 0.0;
    }
    if (ctr_rot != 0)
        moy_rot = temps_rot / (3.0 * ctr_rot);
    else
        moy_rot = 0.0;

    legend_y += 27;
    sprintf (s,"Tt = %.4f",moy_tot);
    XDrawString (dpy, w, gc, LEGEND_X+5, legend_y, s, strlen(s));

    legend_y += 20;
    sprintf (s,"Tc = %.4f",moy_comm);
    XDrawString (dpy, w, gc, LEGEND_X+5, legend_y, s, strlen(s));

    legend_y += 20;
    sprintf (s,"Ta = %.4f",moy_tot-moy_comm);
    XDrawString (dpy, w, gc, LEGEND_X+5, legend_y, s, strlen(s));

    legend_y += 20;
    sprintf (s,"Tr = %.4f",moy_rot);
    XDrawString (dpy, w, gc, LEGEND_X+5, legend_y, s, strlen(s));
}
```

Fonction d'affichage (maître)

```
void Req_Affichage ( void )
{
    int i,j,ctr;
    int tag;
    POINT p;
    float buf[1];
    Window w = XtWindow(draw);

    XClearWindow (dpy, w);
    ++ctr_aff;
    comm_buf[0] = (float) COMM_REQ_AFF;
    comm_buf[1] = (float) cache_zero;
    comm_buf[2] = (float) affiche_par_couleur;
    comm_buf[3] = (float) affiche_par_plan;
    comm_buf[4] = (float) hold_enable;
    comm_buf[5] = (float) plan_a_afficher;
    comm_buf[6] = (float) plan;
    comm_buf[7] = (float) coul;

/*
*  Send command "Req_display" to all nodes
*/

    for (i=1; i<numprocs; ++i)
        MPI_Send (comm_buf,COMM_BUF_SIZE,MPI_FLOAT,i,COMM_TAG,MPI_COMM_WORLD);

    for (i=1; i<numprocs; ++i)
    {
        tag = 1000*(i+1);
        do
        {
            MPI_Recv(recept,RECEPT_SIZE,MPI_FLOAT,i,tag,MPI_COMM_WORLD,&status);
```

```
            p.coul = (int)recept[0];
            if (p.coul != 99)
            {
                ctr=2;
                for (j=0; j<(int)recept[1]; ++j)
                {
                    p.ppx = recept[ctr];  ++ctr;
                    p.ppy = recept[ctr];  ++ctr;

                    if (!Clipping (&p, &p))
                    {
                        Normalise (&p);
                        Affiche_Rectangle_par (p);
                    }
                }
            }
            ++tag;
        }
        while (p.coul != 99);
    }
    if (timer) prn_time();
    Legende();
}
```

Fonction de chargement de fichier (esclave)

```
void Lecture_Fichier_HF ( void )
{
    unsigned int ctr=0;                         /* Compteur */
    unsigned int i,x,y,z;                /* Variables de boucles */
    float min_val = 10000.0;
    float max_val = -10000.0;
    float      lim;                      /* limite pour le calcul des couleurs                */
    int        offset;
    free (a.fig);
    nbx    = (int) comm_buf[1];
    nby    = (int) comm_buf[2];
    nbz    = (int) comm_buf[3];

    MPI_Recv (recept,RECEPT_SIZE,MPI_FLOAT,0,COMM_TAG_2,MPI_COMM_WORLD,&status);
    my_nbz = (int) recept[0];
    offset = (int) recept[1];

    a.fig = (FIGURE *) malloc ( sizeof(FIGURE) );
    if (a.fig == NULL)
    {
            tprintf ("S%d: Erreur d'allocation a.fig\n", noeud);
            return;
    }

    fig = a.fig;
    a.numfig = 1;
    fig->numpts = nbx*nby*my_nbz;
    fig->numsurf = 0;

    fig->pts = (POINT *) malloc( sizeof (POINT) * nbx*nby*my_nbz );
    if ( fig->pts == NULL )
    {
        tprintf ("S%d:Erreur d'allocation a.fig->pts\n", noeud);
        return;
    }
    i=2;
    ctr=0;
```

```
    for ( z=0; z<my_nbz; ++z)
    {
        for ( y=0; y<nby; ++y)
        {
            for ( x=0; x<nbx; ++x)
            {
                fig->pts[ctr].x =  x;
                fig->pts[ctr].y =  y;
                fig->pts[ctr].z =  z+offset;

                fig->pts[ctr].x0 = x;
                fig->pts[ctr].y0 = y;
                fig->pts[ctr].z0 = z+offset;

                fig->pts[ctr].hold    = OFF;
                fig->pts[ctr].enable = OFF;

                fig->pts[ctr].valeur = recept[i];

                if (recept[i] < min_val)  min_val = recept[i];
                if (recept[i] > max_val)  max_val = recept[i];

                ++ctr;  ++i;
            }
        }
    }
    a.PO.x = (float)nbx/2 ;
    a.PO.y = (float)nby/2 ;
    a.PO.z = (float)nbz/2 ;
    a.PX.x = a.PO.x +10.0 ;
    a.PX.y = a.PO.y ;
    a.PX.z = a.PO.z ;
    a.PY.x = a.PO.x ;
    a.PY.y = a.PO.y +10.0 ;
    a.PY.z = a.PO.z ;
    a.PZ.x = a.PO.x ;
    a.PZ.y = a.PO.y ;
    a.PZ.z = a.PO.z +10.0 ;

    pv.x = comm_buf[5];
    pv.y = comm_buf[6];
    pv.z = comm_buf[7];

/*
 *      Calcul des couleurs pour chaque points
 */
    comm_buf[0] = min_val;
    comm_buf[1] = max_val;


    for (i=1; i<numprocs; ++i)
        if (i != noeud)
            MPI_Send (comm_buf,2,MPI_FLOAT,i,COMM_TAG_2,MPI_COMM_WORLD);

    for (i=1; i<numprocs; ++i)
        if (i != noeud)
        {
            MPI_Recv (comm_buf,2,MPI_FLOAT,i,COMM_TAG_2,MPI_COMM_WORLD,&status);
            if (comm_buf[0] < min_val)  min_val = comm_buf[0];
            if (comm_buf[1] > max_val)  max_val = comm_buf[1];
        }

    if (logger) tprintf ("S%d:MIN=%f MAX=%f\n",noeud,min_val,max_val);

     for (i=0; i<MAXCOLORS; ++i)            /* initialisation de la table */
        table_couleur[i] = 0;

    lim = (max_val-min_val)/MAXCOLORS;
```

```
    if (logger) tprintf ("S%d:LIM=%f\n",noeud,lim);

    for (i=0; i<fig->numpts; ++i)
    {
        fig->pts[i].coul = (int) ((fig->pts[i].valeur-min_val) / lim);
        if (logger) tprintf ("S%d:VAL=%f COUL=%d\n",noeud,fig->pts[i].valeur,fig-
>pts[i].coul);

        if (fig->pts[i].coul > MAXCOLORS-1 )
            fig->pts[i].coul=MAXCOLORS-1;

        ++table_couleur[fig->pts[i].coul];
    }
    Projection();
}
```

Fonction d'affichage (esclave)

```
void Affichage (void)
{
    int i,j,tag;
    int ctr = 0;
    float *buffer;
    float buffer_fin[1] = {99.0};
    unsigned int size;
    unsigned int couleur;

    temps_comm=0.0;
    temps_tot=0.0;
    temps_tot = MPI_Wtime();
    ctr=0;
    for (i=0; i<fig->numpts; ++i)
    {
        fig->pts[i].enable = OFF;
        if (affiche_par_couleur==VRAI && cache_zero==VRAI && affiche_par_plan==VRAI)
        {
            if (plan==1)
                if (fig->pts[i].coul==coul && fig->pts[i].valeur!=0.0 && fig-
>pts[i].x0==plan_a_afficher)
                {
                    fig->pts[i].enable=ON;  ++ctr;
                }
            if (plan==2)
                if (fig->pts[i].coul==coul && fig->pts[i].valeur!=0.0 && fig-
>pts[i].y0==plan_a_afficher)
                {
                    fig->pts[i].enable=ON;  ++ctr;
                }
            if (plan==3)
                if (fig->pts[i].coul==coul && fig->pts[i].valeur!=0.0 && fig-
>pts[i].z0==plan_a_afficher)
                {
                    fig->pts[i].enable=ON;  ++ctr;
                }
        }

        else if (affiche_par_plan==VRAI && cache_zero == VRAI)
        {
            if (plan==1)
                if (fig->pts[i].valeur!=0.0 && fig->pts[i].x0==plan_a_afficher)
                {
                    fig->pts[i].enable=ON;  ++ctr;
                }
            if (plan==2)
                if (fig->pts[i].valeur!=0.0 && fig->pts[i].y0==plan_a_afficher)
                {
```

```
                fig->pts[i].enable=ON;  ++ctr;
            }
        if (plan==3)
            if (fig->pts[i].valeur!=0.0 && fig->pts[i].z0==plan_a_afficher)
            {
                fig->pts[i].enable=ON;  ++ctr;
            }
}

else if (affiche_par_couleur==VRAI && affiche_par_plan==VRAI)
{
    if (plan==1)
        if (fig->pts[i].coul==coul  && fig->pts[i].x0==plan_a_afficher)
        {
            fig->pts[i].enable=ON;  ++ctr;
        }
    if (plan==2)
        if (fig->pts[i].coul==coul && fig->pts[i].y0==plan_a_afficher)
        {
            fig->pts[i].enable=ON;  ++ctr;
        }
    if (plan==3)
        if (fig->pts[i].coul==coul && fig->pts[i].z0==plan_a_afficher)
        {
            fig->pts[i].enable=ON;  ++ctr;
        }
}

else if ( affiche_par_couleur==VRAI && cache_zero==VRAI)
{
    if (fig->pts[i].coul==coul && fig->pts[i].valeur!=0.0)
    {
        fig->pts[i].enable=ON;  ++ctr;
    }
}

else if ( affiche_par_plan==VRAI)
{
    if (plan==1)
        if (fig->pts[i].x0==plan_a_afficher)
        {
            fig->pts[i].enable=ON;  ++ctr;
        }
    if (plan==2)
        if (fig->pts[i].y0==plan_a_afficher)
        {
            fig->pts[i].enable=ON;  ++ctr;
        }
    if (plan==3)
        if (fig->pts[i].z0==plan_a_afficher)
        {
            fig->pts[i].enable=ON;  ++ctr;
        }
}
else if ( affiche_par_couleur==VRAI)
{
    if (fig->pts[i].coul==coul)
    {
        fig->pts[i].enable=ON;  ++ctr;
    }
}
else if ( cache_zero==VRAI)
{
    if (fig->pts[i].valeur!=0.0)
    {
        fig->pts[i].enable=ON;  ++ctr;
    }
}
```

```
        else
        {
            fig->pts[i].enable=ON;  ++ctr;
        }

        if (hold_enable==VRAI && fig->pts[i].hold==ON)
        {
            fig->pts[i].enable=ON;  ++ctr;
        }
    }

    ctr=0;
    tag  = 1000*(noeud+1);
    for (j=0; j<MAXCOLORS; ++j)
    {
        if (table_couleur[j] != 0)
        {
            for (i=0;i<fig->numpts; ++i)
                if (fig->pts[i].enable == ON)
                    ++ctr;

            size = (ctr * 2 +2) * sizeof(float);
            buffer = (float *) malloc (size);
            if (buffer == NULL)
            {
                printf ("S%d: Erreur d'allocation buffer dans Affichage()\n", noeud);
                return;
            }
            ctr = 0;
            buffer[ctr]=(float)j;                    ++ctr;
            buffer[ctr]=(float)table_couleur[j];     ++ctr;

            for (i=0; i<fig->numpts; ++i)
            {
                if (fig->pts[i].coul==j && fig->pts[i].enable==ON)
                {
                    buffer[ctr] =  fig->pts[i].ppx;  ++ctr;
                    buffer[ctr] =  fig->pts[i].ppy;  ++ctr;
                }
            }
            if (logger) printf ("S%d: table_couleur[%d]=%d et
ctr=%d\n",noeud,j,table_couleur[j],ctr);
            temps=MPI_Wtime();
            MPI_Send(buffer,table_couleur[j]*2+2,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
            temps_comm += MPI_Wtime()-temps;
            free (buffer);
            ++tag;
        }
    }
    MPI_Send(buffer_fin,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
    temps_tot = MPI_Wtime()-temps_tot;
    if (logger) tprintf
("S%d:Affichage:Tt=%f:Tc=%f:Ta=%f\n",noeud,temps_tot,temps_comm,temps_tot-temps_comm);

}
```